Smart Contract Security 101

Summary

Author: Rajeev

Source: Smart Contract Security 101 Audit Findings 101 Audit Findings 201 Security Pitfalls & Best Practices 101 Security Pitfalls & Best Practices 201 **Info**

Smart Contract Security 101 (Checklist)

1. **Solidity versions**: Using very old versions of Solidity prevent benefits of bug fixes and newer security checks. Using the latest versions might make contracts susceptible to undiscovered compiler bugs. Consider using one of these versions: *0.5.11-0.5.13, 0.5.15-0.5.17, 0.6.8 or 0.6.10-0.6.11. *(see here)

2. **Unlocked pragma**: Contracts should be deployed using the same compiler version/flags with which they have been tested. Locking the pragma (for e.g. by not using ^ in *pragma solidity 0.5.10)* ensures that contracts do not accidentally get deployed using an older compiler version with unfixed bugs. (see here)

3. **Multiple Solidity pragma**: It is better to use one Solidity compiler version across all contracts instead of different versions with different bugs and security checks. (see here)

4. **Incorrect access control**: Contract functions executing critical logic should have appropriate access control enforced via address checks (e.g. owner, controller etc.) typically in modifiers. Missing checks allow attackers to control critical logic. (see here and here)

5. **Unprotected withdraw function**: Unprotected (*external/public*) function calls sending Ether/tokens to user-controlled address may allow users to withdraw unauthorized funds. (see here)

6. **Unprotected call to ****selfdestruct**: A user/attacker can mistakenly/intentionally kill the contract. Protect access to such functions. (see here)

7. **Modifier side-effects: **Modifiers should only implement checks and not make state changes and external calls which violates the checks-effects-interactions pattern. These side-effects may go unnoticed by developers/auditors because the modifier code is typically far from the function implementation. (see here)

8. **Incorrect modifier**: If a modifier does not execute* _ *or *revert*, the function using that modifier will return the default value causing unexpected behavior. (see here)

9. **Constructor names: **Before *solc 0.4.22*, constructor names had to be the same name as the contract class containing it. Misnaming it wouldn't make it a constructor which has security implications. *Solc 0.4.22* introduced the *constructor* keyword. Until *solc 0.5.0*, contracts could have both old-style and new-style constructor names with the first defined one taking precedence over the second if both existed, which also led to security issues. *Solc 0.5.0* forced the use of *constructor* keyword. (see here and here)

10. **Void constructor: **Calls to base contract constructors that are unimplemented leads to misplaced assumptions. Check if constructor is implemented or remove call if not. (see here)

11. **Implicit constructor callValue check**: The creation code of a contract that does not define a constructor but has a base that does, did not revert for calls with non-zero callValue when such a constructor was not explicitly payable. This is due to a compiler bug introduced in *v0.4.5* and fixed in *v0.6.8*. (see here)

12. **Controlled delegatecall: ***delegatecall()* or *callcode()* to an address controlled by the user allows execution of malicious contracts in the context of the caller's state. Ensure trusted destination addresses for such calls. (see here)

13. **Reentrancy vulnerabilities**: Untrusted external contract calls could callback leading to unexpected results such as multiple withdrawals or out-of-order events. Use check-effects-interactions pattern or reentrancy guards. (see here)

14. **ERC777 callbacks and reentrancy: **ERC777 tokens allow arbitrary callbacks via hooks that are called during token transfers. Malicious contract addresses may cause reentrancy on such callbacks if reentrancy guards are not used. (see here)

15. **Avoid ***transfer()***/*****send() ****as reentrancy mitigations**: Although *transfer()* and *send()* have been recommended as a security best-practice to prevent reentrancy attacks because they only forward 2300 gas, the gas repricing of opcodes may break deployed contracts. Use *call()* instead, without hardcoded gas limits along with checks-effects-interactions pattern or reentrancy guards for reentrancy protection. (see here and here)

16. **Private on-chain data**: Marking variables *private* does not mean that they cannot be read on-chain. Private data should not be stored unencrypted in contract code or state but instead stored encrypted or off-chain. (see here)

17. **Weak PRNG**: PRNG relying on *block.timestamp*, *now* or *blockhash *can be influenced by miners to some extent and should be avoided. (see here)

18. **Block values as time proxies: ***block.timestamp* and *block.number *are not good proxies for time because of issues with synchronization, miner manipulation and changing block times. (see here)

19. **Integer overflow/underflow**: Not using OpenZeppelin's SafeMath (or similar libraries) that check for overflows/underflows may lead to vulnerabilities or unexpected behavior if user/attacker can control the integer operands of such arithmetic operations. *Solc v0.8.0* introduced default overflow/underflow checks for all arithmetic operations. (see here and here)

20. **Divide before multiply: **Performing multiplication before division is generally better to avoid loss of precision because Solidity integer division might truncate. (see here)

21. **Transaction order dependence: **Race conditions can be forced on specific Ethereum transactions by monitoring the mempool. For example, the classic ERC20 *approve()* change can be front-run using this method. Do not make assumptions about transaction order dependence. (see here)

22. **ERC20 ***approve()* race condition: **Use *safeIncreaseAllowance()* and *safeDecreaseAllowance()* from OpenZepppelin's *SafeERC20* implementation to prevent race conditions from manipulating the allowance amounts. (see here)

23. **Signature malleability**: The *ecrecover* function is susceptible to signature malleability which could lead to replay attacks. Consider using OpenZeppelin's ECDSA library. (see here, here and here)

24. **ERC20 transfer() does not return boolean: **Contracts compiled with *solc > 0.4.22* interacting with such functions will revert. Use OpenZeppelin's SafeERC20 wrappers. (see here and here)

25. **Incorrect return values for ERC721 ***ownerOf()***: **Contracts compiled with *solc > 0.4.22* interacting with ERC721 *ownerOf()* that returns a *bool* instead of *address* type will revert. Use OpenZeppelin's ERC721 contracts. (see here)

26. **Unexpected Ether and this.balance**: A contract can receive Ether via *payable* functions, *selfdestruct(), coinbase *transaction or pre-sent before creation. Contract logic depending on *this.balance* can therefore be manipulated. (see here and here)

27. ***fallback*** vs ****receive()***: Check that all precautions and subtleties of *fallback/receive* functions related to visibility, state mutability and Ether transfers have been considered. (see here and here)

28. **Dangerous strict equalities: **Use of strict equalities with tokens/Ether can accidentally/maliciously cause unexpected behavior. Consider using *>=* or *<=* instead of *==* for such variables depending on the contract logic. (see here)

29. **Locked Ether**: Contracts that accept Ether via *payable* functions but without withdrawal mechanisms will lock up that Ether. Remove *payable* attribute or add withdraw function. (see here)

30. **Dangerous usage of ****tx.origin***: Use of *tx.origin* for authorization may be abused by a MITM malicious contract forwarding calls from the legitimate user who interacts with it. Use *msg.sender* instead. (see here)

31. **Contract check: **Checking if a call was made from an Externally Owned Account (EOA) or a contract account is typically done using *extcodesize* check which may be circumvented by a contract during construction when it does not have source code available. Checking if *tx.origin == msg.sender *is another option. Both have implications that need to be considered. (see here)

32. **Deleting a ***mapping*** within a ****struct***: Deleting a *struct* that contains a *mapping* will not delete the *mapping* contents which may lead to unintended consequences. (see here)

33. **Tautology or contradiction: **Tautologies (always true) or contradictions (always false) indicate potential flawed logic or redundant checks. e.g. *x >= 0* which is always true if *x* is *uint*. (see here)

34. **Boolean constant**: Use of Boolean constants (*true/false*) in code (e.g. conditionals) is indicative of flawed logic. (see here)

35. **Boolean equality**: Boolean variables can be checked within conditionals directly without the use of equality operators to *true/false*. (see here)

36. **State-modifying functions**: Functions that modify state (in assembly or otherwise) but are labelled *constant/pure/view* revert in *solc >=0.5.0* (work in prior versions) because of the use of *STATICCALL* opcode. (see here)

37. **Return values of low-level calls**: Ensure that return values of low-level calls (*call/callcode/delegatecall/send*/etc.) are checked to avoid unexpected failures. (see here)

38. **Account existence check for low-level calls**: Low-level calls *call/delegatecall/staticcall* return true even if the account called is non-existent (per EVM design). Account existence must be checked prior to calling if needed. (see here)

39. **Dangerous shadowing: **Local variables, state variables, functions, modifiers, or events with names that shadow (i.e. override) builtin Solidity symbols e.g. *now *or other declarations from the current scope are misleading and may lead to unexpected usages and behavior. (see here)

40. **Dangerous state variable shadowing**: Shadowing state variables in derived contracts may be dangerous for critical variables such as contract owner (for e.g. where modifiers in base contracts check on base variables but shadowed variables are set mistakenly) and contracts incorrectly use base/shadowed variables. Do not shadow state variables. (see here)

41. **Pre-declaration usage of local variables**: Usage of a variable before its declaration (either declared later or in another scope) leads to unexpected behavior in *solc < 0.5.0 *but *solc >= 0.5.0* implements C99-style scoping rules where variables can only be used after they have been declared and only in the same or nested scopes. (see here)

42. **Costly operations inside a loop**: Operations such as state variable updates (use SSTOREs) inside a loop cost a lot of gas, are expensive and may lead to out-of-gas errors. Optimizations using local variables are preferred. (see here)

43. **Calls inside a loop: **Calls to external contracts inside a loop are dangerous (especially if the loop index can be user-controlled) because it could lead to DoS if one of the calls reverts or execution runs out of gas. Avoid calls within loops, check that loop index cannot be user-controlled or is bounded. (see here)

44. **DoS with block gas limit**: Programming patterns such as looping over arrays of unknown size may lead to DoS when the gas cost of execution exceeds the block gas limit. (see here)

45. **Missing events**: Events for critical state changes (e.g. owner and other critical parameters) should be emitted for tracking this off-chain. (see here)

46. **Unindexed event parameters**: Parameters of certain events are expected to be indexed (e.g. ERC20 Transfer/Approval events) so that they're included in the block's bloom filter for faster access. Failure to do so might confuse off-chain tooling looking for such indexed events. (see here)

47. **Incorrect event signature in libraries**: Contract types used in events in libraries cause an incorrect event signature hash. Instead of using the type `address` in the hashed signature, the actual contract name was used, leading to a wrong hash in the logs. This is due to a compiler bug introduced in *v0.5.0* and fixed in *v0.5.8*. (see here)

48. **Dangerous unary expressions**: Unary expressions such as *x =+ 1* are likely errors where the programmer really meant to use *x += 1*. Unary *+* operator was deprecated in *solc v0.5.0*. (see here)

49. **Missing zero address validation**: Setters of address type parameters should include a zero-address check otherwise contract functionality may become inaccessible or tokens burnt forever. (see here)

50. **Critical address change**: Changing critical addresses in contracts should be a two-step process where the first transaction (from the old/current address) registers the new address (i.e. grants ownership) and the second transaction (from the new address) replaces the old address with the new one (i.e.

claims ownership). This gives an opportunity to recover from incorrect addresses mistakenly used in the first step. If not, contract functionality might become inaccessible. (see here and here)

51. **assert() state change**: Invariants in *assert()* statements should not modify the state per best practices. *require()* should be used for such checks. (see here)

52. ***require()***** vs ***assert()***: ***require()* should be used for checking error conditions on inputs and return values while *assert()* should be used for invariant checking. Between *solc 0.4.10 and 0.8.0, require()* used *REVERT* (*0xfd*) opcode which refunded remaining gas on failure while *assert()* used INVALID (*0xfe*) opcode which consumed all the supplied gas. (see here)

53. **Deprecated keywords**: Use of deprecated functions/operators such as *block.blockhash()* for *blockhash()*, *msg.gas* for *gasleft(), throw* for *revert()*, *sha3()* for *keccak256()*, *callcode()* for *delegatecall(), suicide()* for *selfdestruct(), constant for view or var for actual type name* should be avoided to prevent unintended errors with newer compiler versions. (see here)

54. **Function default visibility***: *Functions without a visibility type specifier are *public* by default in *solc < 0.5.0*. This can lead to a vulnerability where a malicious user may make unauthorized state changes. *solc >= 0.5.0* requires explicit function visibility specifiers. (see here)

55. **Incorrect inheritance order**: Contracts inheriting from multiple contracts with identical functions should specify the correct inheritance order i.e. more general to more specific to avoid inheriting the incorrect function implementation. (see here)

56. **Missing inheritance**: A contract might appear (based on name or functions implemented) to inherit from another interface or abstract contract without actually doing so. (see here)

57. **Insufficient gas griefing**: Transaction relayers need to be trusted to provide enough gas for the transaction to succeed. (see here)

58. **Modifying reference type parameters**: Structs/Arrays/Mappings passed as arguments to a function may be by value (memory) or reference (storage) as specified by the data location (optional before *solc 0.5.0*). Ensure correct usage of memory and storage in function parameters and make all data locations explicit. (see here)

59. **Arbitrary jump with function type variable: **Function type variables should be carefully handled and avoided in assembly manipulations to prevent jumps to arbitrary code locations. (see here)

60. **Hash collisions with multiple variable length arguments**: Using *abi.encodePacked()* with multiple variable length arguments can, in certain situations, lead to a hash collision. Do not allow users access to parameters used in *abi.encodePacked()*, use fixed length arrays or use *abi.encode()*. (see here and here)

61. **Malleability risk from dirty high order bits**: Types that do not occupy the full 32 bytes might contain "dirty higher order bits" which does not affect operation on types but gives different results with *msg.data*. (see here)

62. **Incorrect shift in assembly**: Shift operators (*shl(x, y)*, *shr(x, y)*, *sar(x, y)*) in Solidity assembly apply the shift operation of *x* bits on *y *and not the other way around, which may be confusing. Check if the values in a shift operation are reversed. (see here)

63. **Assembly usage**: Use of EVM assembly is error-prone and should be avoided or double-checked for correctness. (see here)

64. **Right-To-Left-Override control character (U+202E)**: Malicious actors can use the Right-To-Left-Override unicode character to force RTL text rendering and confuse users as to the real intent of a contract. U+202E character should not appear in the source code of a smart contract. (see here)

65. **Constant state variables**: Constant state variables should be declared constant to save gas. (see here)

66. **Similar variable names**: Variables with similar names could be confused for each other and therefore should be avoided. (see here)

67. **Uninitialized state/local variables**: Uninitialized state/local variables are assigned zero values by the compiler and may cause unintended results e.g. transferring tokens to zero address. Explicitly initialize all state/local variables. (see here and here)

68. **Uninitialized storage pointers: **Uninitialized local storage variables can point to unexpected storage locations in the contract, which can lead to vulnerabilities. *Solc 0.5.0* and above disallow such pointers. (see here)

69. **Uninitialized function pointers in constructors: **Calling uninitialized function pointers in constructors of contracts compiled with *solc* versions *0.4.5-0.4.25* and *0.5.0-0.5.7* lead to unexpected behavior because of a compiler bug. (see here)

70. **Long number literals**: Number literals with many digits should be carefully checked as they are prone to error. (see here)

71. **Out-of-range enum: ***Solc < 0.4.5 *produced unexpected behavior with out-of-range enums*. *Check enum conversion or use a newer compiler.(see here)

72. **Uncalled public functions**: *Public* functions that are never called from within the contracts should be declared *external* to save gas. (see here)

73. **Dead/Unreachable code**: Dead code may be indicative of programmer error, missing logic or potential optimization opportunity, which needs to be flagged for removal or addressed appropriately. (see here)

74. **Unused return values**: Unused return values of function calls are indicative of programmer errors which may have unexpected behavior. (see here)

75. **Unused variables**: Unused state/local variables may be indicative of programmer error, missing logic or potential optimization opportunity, which needs to be flagged for removal or addressed appropriately. (see here)

76. **Redundant statements**: Statements with no effects that do not produce code may be indicative of programmer error or missing logic, which needs to be flagged for removal or addressed appropriately. (see here)

77. **Storage array with signed Integers with ABIEncoderV2**: Assigning an array of signed integers to a storage array of different type can lead to data corruption in that array. This is due to a compiler bug introduced in *v0.4.7* and fixed in *v0.5.10*. (see here)

78. **Dynamic constructor arguments clipped with ABIEncoderV2**: A contract's constructor which takes structs or arrays that contain dynamically sized arrays reverts or decodes to invalid data when ABIEncoderV2 is used. This is due to a compiler bug introduced in *v0.4.16* and fixed in *v0.5.9*. (see here)

79. **Storage array with multiSlot element with ABIEncoderV2**: Storage arrays containing structs or other statically sized arrays are not read properly when directly encoded in external function calls or in *abi.encode()*. This is due to a compiler bug introduced in *v0.4.16* and fixed in *v0.5.10*. (see here)

80. **Calldata structs with statically sized and dynamically encoded members with ABIEncoderV2**: Reading from calldata structs that contain dynamically encoded, but statically sized members can result in incorrect values. This is due to a compiler bug introduced in *v0.5.6* and fixed in *v0.5.11*. (see here)

81. **Packed storage with ABIEncoderV2: **Storage structs and arrays with types shorter than 32 bytes can cause data corruption if encoded directly from storage using ABIEncoderV2. This is due to a compiler bug introduced in *v0.5.0* and fixed in *v0.5.7*. (see here)

82. **Incorrect loads with Yul optimizer and ABIEncoderV2**: The Yul optimizer incorrectly replaces *MLOAD* and *SLOAD* calls with values that have been previously written to the load location. This can only happen if ABIEncoderV2 is activated and the experimental Yul optimizer has been activated manually in addition to the regular optimizer in the compiler settings. This is due to a compiler bug introduced in *v0.5.14* and fixed in *v0.5.15*. (see here)

83. **Array slice dynamically encoded base type with ABIEncoderV2**: Accessing array slices of arrays with dynamically encoded base types (e.g. multi-dimensional arrays) can result in invalid data being read. This is due to a compiler bug introduced in *v0.6.0* and fixed in *v0.6.8*. (see here)

84. **Missing escaping in formatting with ABIEncoderV2: **String literals containing double backslash characters passed directly to external or encoding function calls can lead to a different string being used when ABIEncoderV2 is enabled. This is due to a compiler bug introduced in *v0.5.14* and fixed in *v0.6.8*. (see here)

85. **Double shift size overflow**: Double bitwise shifts by large constants whose sum overflows 256 bits can result in unexpected values. Nested logical shift operations whose total shift size is *2**256* or more are incorrectly optimized. This only applies to shifts by numbers of bits that are compile-time constant expressions. This happens when the optimizer is used and *evmVersion >= Constantinople. *This is due to a compiler bug introduced in *v0.5.5* and fixed in *v0.5.6*. (see here)

86. **Incorrect byte instruction optimization: **The optimizer incorrectly handles byte opcodes whose second argument is 31 or a constant expression that evaluates to 31. This can result in unexpected values. This can happen when performing index access on *bytesNN* types with a compile time constant value (not index) of 31 or when using the byte opcode in inline assembly. This is due to a compiler bug introduced in *v0.5.5* and fixed in *v0.5.7*. (see here)

87. **Essential assignments removed with Yul Optimizer** : The Yul optimizer can remove essential assignments to variables declared inside *for* loops when Yul's *continue* or *break* statement is used mostly while using inline assembly with *for* loops and *continue* and *break* statements. This is due to a compiler bug introduced in *v0.5.8/v0.6.0* and fixed in *v0.5.16/v0.6.1*. (see here)

88. **Private methods overridden**: While private methods of base contracts are not visible and cannot be called directly from the derived contract, it is still possible to declare a function of the same name and type and thus change the behaviour of the base contract's function. This is due to a compiler bug introduced in *v0.3.0* and fixed in *v0.5.17*. (see here)

89. **Tuple assignment multi stack slot components**: Tuple assignments with components that occupy several stack slots, i.e. nested tuples, pointers to external functions or references to dynamically sized calldata arrays, can result in invalid values. This is due to a compiler bug introduced in *v0.1.6* and fixed in *v0.6.6*. (see here)

90. **Dynamic array cleanup**: When assigning a dynamically sized array with types of size at most 16 bytes in storage causing the assigned array to shrink, some parts of deleted slots were not zeroed out. This is due to a compiler bug fixed in *v0.7.3*. (see here)

91. **Empty byte array copy**: Copying an empty byte array (or string) from memory or calldata to storage can result in data corruption if the target array's length is increased subsequently without storing new data. This is due to a compiler bug fixed in *v0.7.4*. (see here)

92. **Memory array creation overflow**: The creation of very large memory arrays can result in overlapping memory regions and thus memory corruption. This is due to a compiler bug introduced in *v0.2.0* and fixed in *v0.6.5*. (see here)

93. **Calldata ***using for***: Function calls to internal library functions with calldata parameters called via "*using for*" can result in invalid data being read. This is due to a compiler bug introduced in *v0.6.9* and fixed in *v0.6.10*. (see here)

94. **Free function redefinition**: The compiler does not flag an error when two or more free functions (functions outside of a contract) with the same name and parameter types are defined in a source unit or when an imported free function alias shadows another free function with a different name but identical parameter types. This is due to a compiler bug introduced in *v0.7.1* and fixed in *v0.7.2*. (see here)

95. **Unprotected initializers in proxy-based upgradeable contracts**: Proxy-based upgradeable contracts need to use *public* initializer functions instead of constructors that need to be explicitly called only once. Preventing multiple invocations of such initializer functions (e.g. via *initializer* modifier from OpenZeppelin's *Initializable* library) is a must. (see here and here)

96. **Initializing state-variables in proxy-based upgradeable contracts**: This should be done in initializer functions and not as part of the state variable declarations in which case they won't be set. (see here)

97. **Import upgradeable contracts in proxy-based upgradeable contracts**: Contracts imported from proxy-based upgradeable contracts should also be upgradeable where such contracts have been modified to use initializers instead of constructors. (see here)

98. **Avoid *selfdestruct* or *delegatecall* in proxy-based upgradeable contracts**: This will cause the logic contract to be destroyed and all contract instances will end up delegating calls to an address without any code. (see here)

99. **State variables in proxy-based upgradeable contracts**: The declaration order/layout and type/mutability of state variables in such contracts should be preserved exactly while upgrading to

prevent critical storage layout mismatch errors. (see here)

100. **Function ID collision between proxy/contract in proxy-based upgradeable contracts**: Malicious proxy contracts may exploit function ID collision to invoke unintended proxy functions instead of contract functions. Check for function ID collisions. (see here and here)

101. **Function shadowing between proxy/contract in proxy-based upgradeable contracts**: Shadow functions in proxy contract prevent functions in logic contract from being invoked. (see here)

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.

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

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

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)

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

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

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

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

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

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.

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

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

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*.

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

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

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().

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

    2. Critical severity finding from Consensys Diligence Audit of DAOfi

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

    1. Recommendation: Check the intended values

    2. Major severity finding from Consensys Diligence Audit of DAOfi

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.

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

    2. Medium severity finding from Consensys Diligence Audit of DAOfi

9. ***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.

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

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

10. **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*.

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

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

11. ***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.

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

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

12. ***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.

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

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

13. ***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.

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

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

14. **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.

    1. 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.

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

15. **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.

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

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

16. ***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.

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

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

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.

    1. Recommendation: Consider using OpenZeppelin's SafeERC20

    2. Major severity finding from Consensys Diligence Audit of bitbank

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.

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

    2. Major severity finding from Consensys Diligence Audit of MetaSwap

19. **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.

    1. 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.

    2. Medium severity finding from Consensys Diligence Audit of MetaSwap

20. **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.

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

    2. Medium severity finding from Consensys Diligence Audit of MetaSwap

21. **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.

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

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

22. **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.

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

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

23. **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

    1. 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.

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

24. **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.

    1. 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.

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

25. **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.

    1. 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.

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

26. ***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.

    1. 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.

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

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.

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

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

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.

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

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

29. **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.

    1. 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

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

30. **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.

    1. 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.

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

31. **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.

    1. 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.

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

32. **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.

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

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

33. **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.

    1. 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.

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

34. **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.

    1. 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.

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

35. **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.

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

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

36. **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*.

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

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

37. **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.

    1. 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.

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

38. **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.

    1. 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.

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

39. **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.

    1. 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.

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

40. **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.

    1. 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.

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

41. **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).

    1. 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.

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

42. **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.

    1. 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.

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

43. **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.

    1. 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.

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

44. **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.

    1. 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.

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

45. **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.

    1. 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.

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

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.

    1. 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.

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

47. **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.

    1. 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.

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

48. **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.

    1. 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.

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

49. **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.

    1. 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.

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

50. **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 <=.

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

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

51. **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.

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

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

52. **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.

    1. 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.

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

53. **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.

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

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

54. **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.

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

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

55. **Use of undefined 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.

    1. 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.

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

56. **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.

    1. 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.

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

57. **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.

    1. 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.

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

58. **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.

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

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

59. ***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*.

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

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

60. **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.

    1. 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.

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

61. **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.

    1. 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.

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

62. **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.

    1. 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.

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

63. ***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.

    1. 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.

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

64. **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.

    1. 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.

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

65. **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.

    1. 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.

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

66. **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.

    1. 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.

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

67. **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.

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

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

68. **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*.

    1. 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.

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

69. **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.

    1. 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.

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

70. **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.

    1. 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.

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

71. **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.

    1. 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.

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

72. **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.

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

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

73. **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.

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

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

74. **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.

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

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

75. **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.

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

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

76. **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.

    1. 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.

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

77. **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.

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

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

78. **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.

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

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

79. **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.

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

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

80. **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.

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

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

81. **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*.

    1. 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.

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

82. **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.

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

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

83. **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.

    1. 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.

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

84. **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.

    1. 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

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

85. **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.

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

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

86. **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.

    1. 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.

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

87. **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.

    1. 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.

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

88. **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.

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

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

89. **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.

    1. 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.

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

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.

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

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

91. **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.

    1. 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.

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

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.

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

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

93. **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.

    1. 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.

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

94. **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.

    1. 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.

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

95. **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.

    1. 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*.

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

96. **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.

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

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

97. **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.

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

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

98. **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.

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

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

99. **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.

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

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

100. **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.

```
1.  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.

2.  Medium Risk severity finding from [OpenZeppelin's Audit of MCDEX Mai
Protocol](https://blog.openzeppelin.com/mcdex-mai-protocol-audit/)
```

101. **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.

```
1.  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.

2.  High Risk severity finding from [OpenZeppelin's Audit of UMA Phase 1]
(https://blog.openzeppelin.com/uma-audit-phase-1/)
```

Audit Findings 201

The numbering starts from 102 in the original article.

1. **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.

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

    2. ConsenSys's Audit of Umbra

2. **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.

    1. 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.

    2. ConsenSys's Audit of Umbra

3. **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.

    1. Recommendation: Add a full coverage test suite.

    2. ConsenSys's Audit of DeFi Saver

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

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

    2. ConsenSys's Audit of DeFi Saver

5. **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*.

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

    2. ConsenSys's Audit of DeFi Saver

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

    1. Recommendation: Add access control to all functions appropriately

    2. Consensys Audit of DeFi Saver

7. **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.

    1. Recommendation: Remove stale comments

    2. ConsenSys's Audit of DAOfi

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

    1. Recommendation: Update the code or the comment to be consistent

    2. ConsenSys's Audit of mstable-1.1

9. **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.

    1. Recommendation: Remove unnecessary calls

    2. ConsenSys's Audit of DAOfi

10. **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.

    1. Recommendation: More validation of mathematical operations

    2. ConsenSys's Audit of DAOfi

11. ***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.

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

    2. ConsenSys's Audit of Fei Protocol

12. **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.

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

    2. ConsenSys's Audit of eRLC

13. **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.

    1. 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.

    2. ConsenSys's Audit of 1inch Liquidity Protocol

14. **Unspecific compiler version pragma**: For most source-units the compiler version pragma is very unspecific *^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.

    1. 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.

2. ConsenSys's Audit of 1inch Liquidity Protocol

15. **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.

    1. 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.

    2. ConsenSys's Audit of 1inch Liquidity Protocol

16. **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.

    1. 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.

    2. Critical finding in ConsenSys's Audit of 1inch Liquidity Protocol

17. **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.

    1. 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.

    2. ConsenSys's Audit of 1inch Liquidity Protocol

18. **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.

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

    2. ConsenSys's Audit of Growth DeFi

19. **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.

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

    2. ConsenSys's Audit of Growth DeFi

20. **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.

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

    2. ConsenSys's Audit of Growth DeFi

21. **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.

    1. Recommendation: Use descriptive names for contracts and libraries

    2. ConsenSys's Audit of Growth DeFi

22. **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.

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

    2. ConsenSys's Audit of Growth DeFi

23. **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.

    1. Recommendation: Reconsider or bound the loop

    2. ConsenSys's Audit of Growth DeFi

24. **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.

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

    2. Critical finding in ConsenSys's Audit of Paxos

25. **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.

    1. 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.

    2. ConsenSys's Audit of Aave Protocol V2

26. **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.

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

    2. ConsenSys's Audit of Aave Governance DAO

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

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

    2. ConsenSys's Audit of Aave CPM Price Provider

28. **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.

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

    2. ConsenSys's Audit of Aave CPM Price Provider

29. **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.

    1. 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.

    2. ConsenSys's Audit of Lien Protocol

30. **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.

    1. Recommendation: Use of modifiers for repeated checks

    2. ConsenSys's Audit of Balancer Finance

31. **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.*

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

    2. ConsenSys's Audit of Balancer Finance

32. **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.

    1. 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

    2. ConsenSys's Audit of MCDEX Mai Protocol V2

33. **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.

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

    2. ToB's Audit of Liquidity

34. **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.

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

    2. ToB's Audit of Origin Dollar

35. **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.

    1. 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.

    2. ToB's Audit of Origin Dollar

36. **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.

    1. Recommendation: Ensure contracts inherit from their interfaces

    2. ToB's Audit of Origin Dollar

37. **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 .

    1. 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.

    2. ToB's Audit of Yield Protocol

38. **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.

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

    2. ToB's Audit of Yield Protocol

39. **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.

    1. 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.

    2. ToB's Audit of Yield Protocol

40. **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.

    1. 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.

    2. ToB's Audit of Yield Protocol

41. **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.

    1. 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.

    2. ToB's Audit of 0x Protocol

42. ***_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.

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

    2. ToB's Audit of 0x Protocol

43. *_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.

    1. 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).

    2. ToB's Audit of DFX Finance

44. ***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.

    1. 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.

    2. ToB's Audit of DFX Finance

45. **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.

1. 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

2. ToB's Audit of DFX Finance

46. ***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 undefined behavior.

    1. 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.

    2. ToB's Audit of DFX Finance

47. **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*.

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

    2. ToB's Audit of DFX Finance

48. **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*.

    1. 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.

    2. ToB's Audit of DFX Finance

49. ***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.

    1. 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.

    2. ToB's Audit of DFX Finance

50. **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.

    1. 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.

    2. ToB's Audit of Hermez Network

51. **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.

    1. 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.

    2. ToB's Audit of Hermez Network

52. ***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.

    1. 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.

    2. ToB's Audit of Hermez Network

53. **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.

1. 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.

2. ToB's Audit of Hermez Network

54. **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.

    1. 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.

    2. ToB's Audit of Hermez Network

55. **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.

    1. 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.

    2. ToB's Audit of Hermez Network

56. **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.

    1. 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.

    2. ToB's Audit of Advanced Blockchains

57. **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.

    1. 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.

    2. ToB's Audit of Advanced Blockchains

58. **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.

    1. 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.

    2. ToB's Audit of Advanced Blockchains

59. **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.

    1. 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.

    2. ToB's Audit of Advanced Blockchains

60. **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.

    1. 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.

    2. ToB's Audit of Advanced Blockchains

61. **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.

    1. 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.

2. ToB's Audit of Advanced Blockchains

62. **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.

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

    2. ToB's Audit of Advanced Blockchains

63. **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.

    1. 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.

    2. ToB's Audit of Advanced Blockchains

64. **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.

    1. 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.

    2. ToB's Audit of dForce Lending

65. **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.

    1. 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.

    2. ToB's Audit of dForce Lending

66. **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.

    1. 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]

    2. Sigma Prime's Audit of Synthetix EtherCollateral

67. **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.

    1. 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.

    2. Sigma Prime's Audit of Synthetix EtherCollateral

68. **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.

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

    2. Sigma Prime's Audit of Synthetix EtherCollateral

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

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

    2. Sigma Prime's Audit of Synthetix EtherCollateral

70. **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.

    1. 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* .

    2. Sigma Prime's Audit of InfiniGold

71. **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).

    1. 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).

    2. Sigma Prime's Audit of InfiniGold

72. **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.

    1. 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.

    2. Sigma Prime's Audit of InfiniGold

73. **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.

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

2. Sigma Prime's Audit of InfiniGold

74. **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.

    1. 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.

    2. Sigma Prime's Audit of Synthetix Unipool

75. **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.

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

    2. Sigma Prime's Audit of Synthetix Unipool

76. **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

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

    2. OpenZeppelin's Audit of HoldeFi

77. **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.

    1. 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.

2. OpenZeppelin's Audit of HoldeFi

78. **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.

    1. 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.

    2. OpenZeppelin's Audit of HoldeFi

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

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

    2. OpenZeppelin's Audit of HoldeFi

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

    1. 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.

    2. OpenZeppelin's Audit of HoldeFi

81. **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.

    1. 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.

    2. OpenZeppelin's Audit of HoldeFi

82. **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.

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

    2. OpenZeppelin's Audit of BarnBrige Smart Yield Bonds

83. **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.

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

    2. OpenZeppelin's Audit of BarnBrige Smart Yield Bonds

84. **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.

    1. 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.

    2. OpenZeppelin's Audit of Compound Governor Bravo

85. **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.

    1. 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.

    2. OpenZeppelin's Audit of Fei Protocol

86. **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.

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

    2. OpenZeppelin's Audit of Fei Protocol

87. **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.

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

    2. OpenZeppelin's Audit of Fei Protocol

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

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

    2. OpenZeppelin's Audit of Fei Protocol

89. **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.

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

    2. OpenZeppelin's Audit of Uma Phase 4

90. **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.

    1. 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.

    2. OpenZeppelin's Audit of GEB Protocol

91. **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.

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

    2. OpenZeppelin's Audit of GEB Protocol

92. **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.

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

    2. OpenZeppelin's Audit of GEB Protocol

93. **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.

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

    2. OpenZeppelin's Audit of GEB Protocol

94. **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.

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

    2. OpenZeppelin's Audit of GEB Protocol

95. ***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.

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

    2. OpenZeppelin's Audit of Opyn Gamma Protocol

96. **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.

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

    2. OpenZeppelin's Audit of Opyn Gamma Protocol

97. **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.

    1. 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.

    2. OpenZeppelin's Audit of Opyn Gamma Protocol

98. **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.

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

    2. OpenZeppelin's Audit of Opyn Gamma Protocol

99. **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.

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

    2. OpenZeppelin's Audit of PoolTogether V3

100. **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.

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

2.  [OpenZeppelin's Audit of Set Protocol]
(https://blog.openzeppelin.com/set-protocol-audit/)
```

Security Pitfalls & Best Practices 101

1. **Solidity versions**: Using very old versions of Solidity prevents benefits of bug fixes and newer security checks. Using the latest versions might make contracts susceptible to undiscovered compiler bugs. Consider using one of these versions: *0.7.5, 0.7.6 or 0.8.4 . *(see here)

2. **Unlocked pragma**: Contracts should be deployed using the same compiler version/flags with which they have been tested. Locking the pragma (for e.g. by not using ^ in *pragma solidity 0.5.10)* ensures that contracts do not accidentally get deployed using an older compiler version with unfixed bugs. (see here)

3. **Multiple Solidity pragma**: It is better to use one Solidity compiler version across all contracts instead of different versions with different bugs and security checks. (see here)

4. **Incorrect access control**: Contract functions executing critical logic should have appropriate access control enforced via address checks (e.g. owner, controller etc.) typically in modifiers. Missing checks allow attackers to control critical logic. (see here and here)

5. **Unprotected withdraw function**: Unprotected (*external/public*) function calls sending Ether/tokens to user-controlled addresses may allow users to withdraw unauthorized funds. (see here)

6. **Unprotected call to **_selfdestruct_**: A user/attacker can mistakenly/intentionally kill the contract. Protect access to such functions. (see here)

7. **Modifier side-effects: **Modifiers should only implement checks and not make state changes and external calls which violates the checks-effects-interactions pattern. These side-effects may go

unnoticed by developers/auditors because the modifier code is typically far from the function implementation. (see here)

8. **Incorrect modifier**: If a modifier does not execute* _ *or *revert*, the function using that modifier will return the default value causing unexpected behavior. (see here)

9. **Constructor names: **Before *solc 0.4.22*, constructor names had to be the same name as the contract class containing it. Misnaming it wouldn't make it a constructor which has security implications. *Solc 0.4.22* introduced the *constructor* keyword. Until *solc 0.5.0*, contracts could have both old-style and new-style constructor names with the first defined one taking precedence over the second if both existed, which also led to security issues. *Solc 0.5.0* forced the use of the *constructor* keyword. (see here and here)

10. **Void constructor: **Calls to base contract constructors that are unimplemented leads to misplaced assumptions. Check if the constructor is implemented or remove call if not. (see here)

11. **Implicit constructor callValue check**: The creation code of a contract that does not define a constructor but has a base that does, did not revert for calls with non-zero callValue when such a constructor was not explicitly payable. This is due to a compiler bug introduced in *v0.4.5* and fixed in *v0.6.8*. Starting from Solidity 0.4.5 the creation code of contracts without explicit payable constructor is supposed to contain a callvalue check that results in contract creation reverting, if non-zero value is passed. However, this check was missing in case no explicit constructor was defined in a contract at all, but the contract has a base that does define a constructor. In these cases it is possible to send value in a contract creation transaction or using inline assembly without revert, even though the creation code is supposed to be non-payable. (see here)

12. **Controlled delegatecall: ***delegatecall()* or *callcode()* to an address controlled by the user allows execution of malicious contracts in the context of the caller's state. Ensure trusted destination addresses for such calls. (see here)

13. **Reentrancy vulnerabilities**: Untrusted external contract calls could callback leading to unexpected results such as multiple withdrawals or out-of-order events. Use check-effects-interactions pattern or reentrancy guards. (see here)

14. **ERC777 callbacks and reentrancy: **ERC777 tokens allow arbitrary callbacks via hooks that are called during token transfers. Malicious contract addresses may cause reentrancy on such callbacks if reentrancy guards are not used. (see here)

15. **Avoid ***transfer()***/*****send() ****as reentrancy mitigations**: Although *transfer()* and *send()* have been recommended as a security best-practice to prevent reentrancy attacks because they only forward 2300 gas, the gas repricing of opcodes may break deployed contracts. Use *call()* instead, without hardcoded gas limits along with checks-effects-interactions pattern or reentrancy guards for reentrancy protection. (see here and here)

16. **Private on-chain data**: Marking variables *private* does not mean that they cannot be read on-chain. Private data should not be stored unencrypted in contract code or state but instead stored encrypted or off-chain. (see here)

17. **Weak PRNG**: PRNG relying on *block.timestamp*, *now* or *blockhash *can be influenced by miners to some extent and should be avoided. (see here)

18. **Block values as time proxies: **_block.timestamp_ and *block.number *are not good proxies (i.e. representations, not to be confused with smart contract proxy/implementation pattern) for time because of issues with synchronization, miner manipulation and changing block times. (see here)

19. **Integer overflow/underflow**: Not using OpenZeppelin's SafeMath (or similar libraries) that check for overflows/underflows may lead to vulnerabilities or unexpected behavior if user/attacker can control the integer operands of such arithmetic operations. *Solc v0.8.0* introduced default overflow/underflow checks for all arithmetic operations. (see here and here)

20. **Divide before multiply: **Performing multiplication before division is generally better to avoid loss of precision because Solidity integer division might truncate. (see here)

21. **Transaction order dependence: **Race conditions can be forced on specific Ethereum transactions by monitoring the mempool. For example, the classic ERC20 *approve()* change can be front-run using this method. Do not make assumptions about transaction order dependence. (see here)

22. **ERC20 ***approve()*** race condition: **Use *safeIncreaseAllowance()* and *safeDecreaseAllowance()* from OpenZeppelin's *SafeERC20* implementation to prevent race conditions from manipulating the allowance amounts. (see here)

23. **Signature malleability**: The *ecrecover* function is susceptible to signature malleability which could lead to replay attacks. Consider using OpenZeppelin's ECDSA library. (see here, here and here)

24. **ERC20 transfer() does not return boolean: **Contracts compiled with *solc >= 0.4.22* interacting with such functions will revert. Use OpenZeppelin's SafeERC20 wrappers. (see here and here)

25. **Incorrect return values for ERC721 ***ownerOf()***: **Contracts compiled with *solc >= 0.4.22* interacting with ERC721 *ownerOf()* that returns a *bool* instead of *address* type will revert. Use OpenZeppelin's ERC721 contracts. (see here)

26. **Unexpected Ether and this.balance**: A contract can receive Ether via *payable* functions, *selfdestruct(), coinbase *transaction or pre-sent before creation. Contract logic depending on *this.balance* can therefore be manipulated. (see here and here)

27. ***fallback**** vs ****receive()***: Check that all precautions and subtleties of *fallback*/*receive* functions related to visibility, state mutability and Ether transfers have been considered. (see here and here)

28. **Dangerous strict equalities: **Use of strict equalities with tokens/Ether can accidentally/maliciously cause unexpected behavior. Consider using >= or <= instead of == for such variables depending on the contract logic. (see here)

29. **Locked Ether**: Contracts that accept Ether via *payable* functions but without withdrawal mechanisms will lock up that Ether. Remove *payable* attribute or add withdraw function. (see here)

30. **Dangerous usage of ****tx.origin***: Use of *tx.origin* for authorization may be abused by a MITM malicious contract forwarding calls from the legitimate user who interacts with it. Use *msg.sender* instead. (see here)

31. **Contract check: **Checking if a call was made from an Externally Owned Account (EOA) or a contract account is typically done using *extcodesize* check which may be circumvented by a contract

during construction when it does not have source code available. Checking if *tx.origin == msg.sender *is another option. Both have implications that need to be considered. (see here)

32. **Deleting a ***mapping*** within a ****struct***: Deleting a *struct* that contains a *mapping* will not delete the *mapping* contents which may lead to unintended consequences. (see here)

33. **Tautology or contradiction: **Tautologies (always true) or contradictions (always false) indicate potential flawed logic or redundant checks. e.g. *x >= 0* which is always true if *x* is *uint*. (see here)

34. **Boolean constant**: Use of Boolean constants (*true/false*) in code (e.g. conditionals) is indicative of flawed logic. (see here)

35. **Boolean equality**: Boolean variables can be checked within conditionals directly without the use of equality operators to *true/false*. (see here)

36. **State-modifying functions**: Functions that modify state (in assembly or otherwise) but are labelled *constant/pure/view* revert in *solc >=0.5.0* (work in prior versions) because of the use of *STATICCALL* opcode. (see here)

37. **Return values of low-level calls**: Ensure that return values of low-level calls (*call/callcode/delegatecall/send*/etc.) are checked to avoid unexpected failures. (see here)

38. **Account existence check for low-level calls**: Low-level calls *call/delegatecall/staticcall* return true even if the account called is non-existent (per EVM design). Account existence must be checked prior to calling if needed. (see here)

39. **Dangerous shadowing: **Local variables, state variables, functions, modifiers, or events with names that shadow (i.e. override) builtin Solidity symbols e.g. *now *or other declarations from the current scope are misleading and may lead to unexpected usages and behavior. (see here)

40. **Dangerous state variable shadowing**: Shadowing state variables in derived contracts may be dangerous for critical variables such as contract owner(for e.g. where modifiers in base contracts check on base variables but shadowed variables are set mistakenly) and contracts incorrectly use base/shadowed variables. Do not shadow state variables. (see here)

41. **Pre-declaration usage of local variables**: Usage of a variable before its declaration (either declared later or in another scope) leads to unexpected behavior in *solc < 0.5.0 *but *solc >= 0.5.0* implements C99-style scoping rules where variables can only be used after they have been declared and only in the same or nested scopes. (see here)

42. **Costly operations inside a loop**: Operations such as state variable updates (use SSTOREs) inside a loop cost a lot of gas, are expensive and may lead to out-of-gas errors. Optimizations using local variables are preferred. (see here)

43. **Calls inside a loop: **Calls to external contracts inside a loop are dangerous (especially if the loop index can be user-controlled) because it could lead to DoS if one of the calls reverts or execution runs out of gas. Avoid calls within loops, check that loop index cannot be user-controlled or is bounded. (see here)

44. **DoS with block gas limit**: Programming patterns such as looping over arrays of unknown size may lead to DoS when the gas cost of execution exceeds the block gas limit. (see here)

45. **Missing events**: Events for critical state changes (e.g. owner and other critical parameters) should be emitted for tracking this off-chain. (see here)

46. **Unindexed event parameters**: Parameters of certain events are expected to be indexed (e.g. ERC20 Transfer/Approval events) so that they're included in the block's bloom filter for faster access. Failure to do so might confuse off-chain tooling looking for such indexed events. (see here)

47. **Incorrect event signature in libraries**: Contract types used in events in libraries cause an incorrect event signature hash. Instead of using the type `address` in the hashed signature, the actual contract name was used, leading to a wrong hash in the logs. This is due to a compiler bug introduced in *v0.5.0* and fixed in *v0.5.8*. (see here)

48. **Dangerous unary expressions**: Unary expressions such as *x =+ 1* are likely errors where the programmer really meant to use *x += 1*. Unary *+* operator was deprecated in *solc v0.5.0*. (see here)

49. **Missing zero address validation**: Setters of address type parameters should include a zero-address check otherwise contract functionality may become inaccessible or tokens burnt forever. (see here)

50. **Critical address change**: Changing critical addresses in contracts should be a two-step process where the first transaction (from the old/current address) registers the new address (i.e. grants ownership) and the second transaction (from the new address) replaces the old address with the new one (i.e. claims ownership). This gives an opportunity to recover from incorrect addresses mistakenly used in the first step. If not, contract functionality might become inaccessible. (see here and here)

51. **assert()/require() state change**: Invariants in *assert()* and *require()* statements should not modify the state per best practices. (see here)

52. ***require()**** vs ***assert()***: ***require()* should be used for checking error conditions on inputs and return values while *assert()* should be used for invariant checking. Between *solc 0.4.10 *and *0.8.0*, *require()* used *REVERT* (*0xfd*) opcode which refunded remaining gas on failure while *assert()* used INVALID (*0xfe*) opcode which consumed all the supplied gas. (see here)

53. **Deprecated keywords**: Use of deprecated functions/operators such as *block.blockhash()* for *blockhash()*, *msg.gas* for *gasleft()*, *throw* for *revert()*, *sha3()* for *keccak256()*, *callcode()* for *delegatecall()*, *suicide()* for *selfdestruct()*, *constant for view or var for actual type name* should be avoided to prevent unintended errors with newer compiler versions. (see here)

54. **Function default visibility***: *Functions without a visibility type specifier are *public* by default in *solc < 0.5.0*. This can lead to a vulnerability where a malicious user may make unauthorized state changes. *solc >= 0.5.0* requires explicit function visibility specifiers. (see here)

55. **Incorrect inheritance order**: Contracts inheriting from multiple contracts with identical functions should specify the correct inheritance order i.e. more general to more specific to avoid inheriting the incorrect function implementation. (see here)

56. **Missing inheritance**: A contract might appear (based on name or functions implemented) to inherit from another interface or abstract contract without actually doing so. (see here)

57. **Insufficient gas griefing**: Transaction relayers need to be trusted to provide enough gas for the transaction to succeed. (see here)

58. **Modifying reference type parameters**: Structs/Arrays/Mappings passed as arguments to a function may be by value (memory) or reference (storage) as specified by the data location (optional before *solc 0.5.0*). Ensure correct usage of memory and storage in function parameters and make all data locations explicit. (see here)

59. **Arbitrary jump with function type variable: **Function type variables should be carefully handled and avoided in assembly manipulations to prevent jumps to arbitrary code locations. (see here)

60. **Hash collisions with multiple variable length arguments**: Using *abi.encodePacked()* with multiple variable length arguments can, in certain situations, lead to a hash collision. Do not allow users access to parameters used in *abi.encodePacked()*, use fixed length arrays or use *abi.encode()*.
(see here and here)

61. **Malleability risk from dirty high order bits**: Types that do not occupy the full 32 bytes might contain "dirty higher order bits" which does not affect operation on types but gives different results with *msg.data*. (see here)

62. **Incorrect shift in assembly**: Shift operators (*shl(x, y)*, *shr(x, y)*, *sar(x, y)*) in Solidity assembly apply the shift operation of *x* bits on *y *and not the other way around, which may be confusing. Check if the values in a shift operation are reversed. (see here)

63. **Assembly usage**: Use of EVM assembly is error-prone and should be avoided or double-checked for correctness. (see here)

64. **Right-To-Left-Override control character (U+202E)**: Malicious actors can use the Right-To-Left-Override unicode character to force RTL text rendering and confuse users as to the real intent of a contract. U+202E character should not appear in the source code of a smart contract. (see here)

65. **Constant state variables**: Constant state variables should be declared constant to save gas. (see here)

66. **Similar variable names**: Variables with similar names could be confused for each other and therefore should be avoided. (see here)

67. **Uninitialized state/local variables**: Uninitialized state/local variables are assigned zero values by the compiler and may cause unintended results e.g. transferring tokens to zero address. Explicitly initialize all state/local variables. (see here and here)

68. **Uninitialized storage pointers: **Uninitialized local storage variables can point to unexpected storage locations in the contract, which can lead to vulnerabilities. *Solc 0.5.0* and above disallow such pointers. (see here)

69. **Uninitialized function pointers in constructors: **Calling uninitialized function pointers in constructors of contracts compiled with *solc* versions *0.4.5-0.4.25* and *0.5.0-0.5.7* lead to unexpected behavior because of a compiler bug. (see here)

70. **Long number literals**: Number literals with many digits should be carefully checked as they are prone to error. (see here)

71. **Out-of-range enum: ***Solc < 0.4.5 *produced unexpected behavior with out-of-range enums*. *Check enum conversion or use a newer compiler.(see here)

72. **Uncalled public functions**: *Public* functions that are never called from within the contracts should be declared *external* to save gas. (see here)

73. **Dead/Unreachable code**: Dead code may be indicative of programmer error, missing logic or potential optimization opportunity, which needs to be flagged for removal or addressed appropriately. (see here)

74. **Unused return values**: Unused return values of function calls are indicative of programmer errors which may have unexpected behavior. (see here)

75. **Unused variables**: Unused state/local variables may be indicative of programmer error, missing logic or potential optimization opportunity, which needs to be flagged for removal or addressed appropriately. (see here)

76. **Redundant statements**: Statements with no effects that do not produce code may be indicative of programmer error or missing logic, which needs to be flagged for removal or addressed appropriately. (see here)

77. **Storage array with signed Integers with ABIEncoderV2**: Assigning an array of signed integers to a storage array of different type can lead to data corruption in that array. This is due to a compiler bug introduced in *v0.4.7* and fixed in *v0.5.10*. (see here)

78. **Dynamic constructor arguments clipped with ABIEncoderV2**: A contract's constructor which takes structs or arrays that contain dynamically sized arrays reverts or decodes to invalid data when ABIEncoderV2 is used. This is due to a compiler bug introduced in *v0.4.16* and fixed in *v0.5.9*. (see here)

79. **Storage array with multiSlot element with ABIEncoderV2**: Storage arrays containing structs or other statically sized arrays are not read properly when directly encoded in external function calls or in *abi.encode()*. This is due to a compiler bug introduced in *v0.4.16* and fixed in *v0.5.10*. (see here)

80. **Calldata structs with statically sized and dynamically encoded members with ABIEncoderV2**: Reading from calldata structs that contain dynamically encoded, but statically sized members can result in incorrect values. This is due to a compiler bug introduced in *v0.5.6* and fixed in *v0.5.11*. (see here)

81. **Packed storage with ABIEncoderV2: **Storage structs and arrays with types shorter than 32 bytes can cause data corruption if encoded directly from storage using ABIEncoderV2. This is due to a compiler bug introduced in *v0.5.0* and fixed in *v0.5.7*. (see here)

82. **Incorrect loads with Yul optimizer and ABIEncoderV2**: The Yul optimizer incorrectly replaces *MLOAD* and *SLOAD* calls with values that have been previously written to the load location. This can only happen if ABIEncoderV2 is activated and the experimental Yul optimizer has been activated manually in addition to the regular optimizer in the compiler settings. This is due to a compiler bug introduced in *v0.5.14* and fixed in *v0.5.15*. (see here)

83. **Array slice dynamically encoded base type with ABIEncoderV2**: Accessing array slices of arrays with dynamically encoded base types (e.g. multi-dimensional arrays) can result in invalid data being read. This is due to a compiler bug introduced in *v0.6.0* and fixed in *v0.6.8*. (see here)

84. **Missing escaping in formatting with ABIEncoderV2: **String literals containing double backslash characters passed directly to external or encoding function calls can lead to a different string being used when ABIEncoderV2 is enabled. This is due to a compiler bug introduced in *v0.5.14* and fixed in *v0.6.8*. (see here)

85. **Double shift size overflow**: Double bitwise shifts by large constants whose sum overflows 256 bits can result in unexpected values. Nested logical shift operations whose total shift size is *2**256* or more are incorrectly optimized. This only applies to shifts by numbers of bits that are compile-time constant expressions. This happens when the optimizer is used and *evmVersion >= Constantinople. *This is due to a compiler bug introduced in *v0.5.5* and fixed in *v0.5.6*. (see here)

86. **Incorrect byte instruction optimization: **The optimizer incorrectly handles byte opcodes whose second argument is 31 or a constant expression that evaluates to 31. This can result in unexpected values. This can happen when performing index access on *bytesNN* types with a compile time constant value (not index) of 31 or when using the byte opcode in inline assembly. This is due to a compiler bug introduced in *v0.5.5* and fixed in *v0.5.7*. (see here)

87. **Essential assignments removed with Yul Optimizer** : The Yul optimizer can remove essential assignments to variables declared inside *for* loops when Yul's *continue* or *break* statement is used mostly while using inline assembly with *for* loops and *continue* and *break* statements. This is due to a compiler bug introduced in *v0.5.8/v0.6.0* and fixed in *v0.5.16/v0.6.1*. (see here)

88. **Private methods overridden**: While private methods of base contracts are not visible and cannot be called directly from the derived contract, it is still possible to declare a function of the same name and type and thus change the behaviour of the base contract's function. This is due to a compiler bug introduced in *v0.3.0* and fixed in *v0.5.17*. (see here)

89. **Tuple assignment multi stack slot components**: Tuple assignments with components that occupy several stack slots, i.e. nested tuples, pointers to external functions or references to dynamically sized calldata arrays, can result in invalid values. This is due to a compiler bug introduced in *v0.1.6* and fixed in *v0.6.6*. (see here)

90. **Dynamic array cleanup**: When assigning a dynamically sized array with types of size at most 16 bytes in storage causing the assigned array to shrink, some parts of deleted slots were not zeroed out. This is due to a compiler bug fixed in *v0.7.3*. (see here)

91. **Empty byte array copy**: Copying an empty byte array (or string) from memory or calldata to storage can result in data corruption if the target array's length is increased subsequently without storing new data. This is due to a compiler bug fixed in *v0.7.4*. (see here)

92. **Memory array creation overflow**: The creation of very large memory arrays can result in overlapping memory regions and thus memory corruption. This is due to a compiler bug introduced in *v0.2.0* and fixed in *v0.6.5*. (see here)

93. **Calldata *using for***: Function calls to internal library functions with calldata parameters called via "*using for*" can result in invalid data being read. This is due to a compiler bug introduced in *v0.6.9* and fixed in *v0.6.10*. (see here)

94. **Free function redefinition**: The compiler does not flag an error when two or more free functions (functions outside of a contract) with the same name and parameter types are defined in a source unit

or when an imported free function alias shadows another free function with a different name but identical parameter types. This is due to a compiler bug introduced in *v0.7.1* and fixed in *v0.7.2*. (see here)

95. **Unprotected initializers in proxy-based upgradeable contracts**: Proxy-based upgradeable contracts need to use *public* initializer functions instead of constructors that need to be explicitly called only once. Preventing multiple invocations of such initializer functions (e.g. via *initializer* modifier from OpenZeppelin's *Initializable* library) is a must. (see here and here)

96. **Initializing state-variables in proxy-based upgradeable contracts**: This should be done in initializer functions and not as part of the state variable declarations in which case they won't be set. (see here)

97. **Import upgradeable contracts in proxy-based upgradeable contracts**: Contracts imported from proxy-based upgradeable contracts should also be upgradeable where such contracts have been modified to use initializers instead of constructors. (see here)

98. **Avoid *selfdestruct* or *delegatecall* in proxy-based upgradeable contracts**: This will cause the logic contract to be destroyed and all contract instances will end up delegating calls to an address without any code. (see here)

99. **State variables in proxy-based upgradeable contracts**: The declaration order/layout and type/mutability of state variables in such contracts should be preserved exactly while upgrading to prevent critical storage layout mismatch errors. (see here)

100. **Function ID collision between proxy/implementation in proxy-based upgradeable contracts**: Malicious proxy contracts may exploit function ID collision to invoke unintended proxy functions instead of delegating to implementation functions. Check for function ID collisions. (see here and here)

101. **Function shadowing between proxy/contract in proxy-based upgradeable contracts**: Shadow functions in proxy contract prevent functions in logic contract from being invoked. (see here)

Security Pitfalls & Best Practices 201

The numbering starts from 102 in the original article.

1. **ERC20 transfer and transferFrom**: Should return a boolean. Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail. (See here)

2. **ERC20 name, decimals, and symbol functions**: Are present if used. These functions are optional in the ERC20 standard and might not be present. (See here)

3. **ERC20 decimals returns a uint8**: Several tokens incorrectly return a uint256. If this is the case, ensure the value returned is below 255. (See here)

4. **ERC20 *approve* race-condition**: The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens. (See here)

5. **ERC777 hooks**: ERC777 tokens have the concept of a hook function that is called before any calls to send, transfer, operatorSend, minting and burning. While these hooks enable a lot of interesting use

cases, care should be taken to make sure they do not make external calls because that can lead to reentrancies. (See here)

6. **Token Deflation via fees**: Transfer and transferFrom should not take a fee. Deflationary tokens can lead to unexpected behavior. (See here)
7. **Token Inflation via interest**: Potential interest earned from the token should be taken into account. Some tokens distribute interest to token holders. This interest might be trapped in the contract if not taken into account. (See here)
8. **Token contract avoids unneeded complexity**: The token should be a simple contract; a token with complex code requires a higher standard of review. (See here)
9. **Token contract has only a few non--token-related functions**: Non--token-related functions increase the likelihood of an issue in the contract. (See here)
10. **Token only has one address**: Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g. *balances[token_address][msg.sender]* might not reflect the actual balance). (See here)
11. **Token is not upgradeable**: Upgradeable contracts might change their rules over time. (See here)
12. **Token owner has limited minting capabilities**: Malicious or compromised owners can abuse minting capabilities. (See here)
13. **Token is not pausable**: Malicious or compromised owners can trap contracts relying on pausable tokens. (See here)
14. **Token owner cannot blacklist the contract**: Malicious or compromised owners can trap contracts relying on tokens with a blacklist. (See here)
15. **Token development team is known and can be held responsible for abuse**: Contracts with anonymous development teams, or that reside in legal shelters should require a higher standard of review. (See here)
16. **No token user owns most of the supply**: If a few users own most of the tokens, they can influence operations based on the token's repartition. (See here)
17. **Token total supply is sufficient**: Tokens with a low total supply can be easily manipulated. (See here)
18. **Tokens are located in more than a few exchanges**: If all the tokens are in one exchange, a compromise of the exchange can compromise the contract relying on the token. (See here)
19. **Token balance and Flash loans**: Users understand the associated risks of large funds or flash loans. Contracts relying on the token balance must carefully take in consideration attackers with large funds or attacks through flash loans. (See here)
20. **Token does not allow flash minting**: Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token. (See here)
21. **ERC1400 permissioned addresses**: Can block transfers from/to specific addresses. (See here)

22. **ERC1400 forced transfers**: Trusted actors have the ability to transfer funds however they choose. (See here)

23. **ERC1644 forced transfers**: Controller has the ability to steal funds. (See here)

24. **ERC621 control of totalSupply**: totalSupply can be changed by trusted actors (See here)

25. **ERC884 cancel and reissue**: Token implementers have the ability to cancel an address and move its tokens to a new address (See here)

26. **ERC884 whitelisting**: Tokens can only be sent to whitelisted addresses (See here)

27. **Guarded launch via asset limits**: Limiting the total asset value managed by a system initially upon launch and gradually increasing it over time may reduce impact due to initial vulnerabilities or exploits. (See here)

28. **Guarded launch via asset types: **Limiting types of assets that can be used in the protocol initially upon launch and gradually expanding to other assets over time may reduce impact due to initial vulnerabilities or exploits. (See here)

29. **Guarded launch via user limits: **Limiting the total number of users that can interact with a system initially upon launch and gradually increasing it over time may reduce impact due to initial vulnerabilities or exploits. Initial users may also be whitelisted to limit to trusted actors before opening the system to everyone. (See here)

30. **Guarded launch via usage limits**: Enforcing transaction size limits, daily volume limits, per-account limits, or rate-limiting transactions may reduce impact due to initial vulnerabilities or exploits. (See here)

31. **Guarded launch via composability limits: **Restricting the composability of the system to interface only with whitelisted trusted contracts before expanding to arbitrary external contracts may reduce impact due to initial vulnerabilities or exploits. (See here)

32. **Guarded launch via escrows**: Escrowing high value transactions/operations with time locks and a governance capability to nullify or revert transactions may reduce impact due to initial vulnerabilities or exploits. (See here)

33. **Guarded launch via circuit breakers**: Implementing capabilities to pause/unpause a system in extreme scenarios may reduce impact due to initial vulnerabilities or exploits. (See here)

34. **Guarded launch via emergency shutdown: **Implement capabilities that allow governance to shutdown new activity in the system and allow users to reclaim assets may reduce impact due to initial vulnerabilities or exploits. (See here)

35. **System specification**: Ensure that the specification of the entire system is considered, written and evaluated to the greatest detail possible. Specification describes how (and why) the different components of the system behave to achieve the design requirements. Without specification, a system implementation cannot be evaluated against the requirements for correctness.

36. **System documentation**: Ensure that roles, functionalities and interactions of the entire system are well documented to the greatest detail possible. Documentation describes what (and how) the

implementation of different components of the system does to achieve the specification goals. Without documentation, a system implementation cannot be evaluated against the specification for correctness and one will have to rely on analyzing the implementation itself.

37. **Function parameters**: Ensure input validation for all function parameters especially if the visibility is external/public where (untrusted) users can control values. This is especially required for address parameters where maliciously/accidentally used incorrect/zero addresses can cause vulnerabilities or unexpected behavior.

38. **Function arguments**: Ensure that the arguments to function calls at the caller sites are the correct ones and in the right order as expected by the function definition.

39. **Function visibility**: Ensure that the strictest visibility is used for the required functionality. An accidental external/public visibility will allow (untrusted) users to invoke functionality that is supposed to be restricted internally.

40. **Function modifiers**: Ensure that the right set of function modifiers are used (in the correct order) for the specific functions so that the expected access control or validation is correctly enforced.

41. **Function return values**: Ensure that the appropriate return value(s) are returned from functions and checked without ignoring at function call sites, so that error conditions are caught and handled appropriately.

42. **Function invocation timeliness**: Externally accessible functions (*external*/*public* visibility) may be called at any time (or never). It is not safe to assume they will only be called at specific system phases (e.g. after initialization, when unpaused, during liquidation) that is meaningful to the system design. The reason for this can be accidental or malicious. Function implementation should be robust enough to track system state transitions, determine meaningful states for invocations and withstand arbitrary calls. For e.g., initialization functions (used with upgradeable contracts that cannot use constructors) are meant to be called atomically along with contract deployment to prevent anyone else from initializing with arbitrary values.

43. **Function invocation repetitiveness**: Externally accessible functions (*external*/*public* visibility) may be called any number of times. It is not safe to assume they will only be called only once or a specific number of times that is meaningful to the system design. Function implementation should be robust enough to track, prevent, ignore or account for arbitrarily repetitive invocations. For e.g., initialization functions (used with upgradeable contracts that cannot use constructors) are meant to be called only once.

44. **Function invocation order: **Externally accessible functions (*external*/*public* visibility) may be called in any order (with respect to other defined functions). It is not safe to assume they will only be called in the specific order that makes sense to the system design or is implicitly assumed in the code. For e.g., initialization functions (used with upgradeable contracts that cannot use constructors) are meant to be called before other system functions can be called.

45. **Function invocation arguments**: Externally accessible functions (*external*/*public* visibility) may be called with any possible arguments. Without complete and proper validation (e.g. zero address checks, bound checks, threshold checks etc.), they cannot be assumed to comply with any assumptions made about them in the code.

46. **Conditionals**: Ensure that in conditional expressions (e.g. if statements), the correct variables are being checked and the correct operators, if any, are being used to combine them. For e.g. using || instead of && is a common error.

47. **Access control specification**: Ensure that the various system actors, their access control privileges and trust assumptions are accurately specified in great detail so that they are correctly implemented and enforced across different contracts, functions and system transitions/flows.

48. **Access control implementation**: Ensure that the specified access control is implemented uniformly across all the subjects (actors) seeking access and objects (variables, functions) being accessed so that there are no paths/flows where the access control is missing or may be side-stepped.

49. **Missing modifiers**: Access control is typically enforced on functions using modifiers that check if specific addresses/roles are calling these functions. Ensure that such modifiers are present on all relevant functions which require that specific access control.

50. **Incorrectly implemented modifiers**: Access control is typically enforced on functions using modifiers that check if specific addresses/roles are calling these functions. A system can have multiple roles with different privileges. Ensure that modifiers are implementing the expected checks on the correct roles/addresses with the right composition e.g. incorrect use of || instead of && is a common vulnerability while composing access checks.

51. **Incorrectly used modifiers**: Access control is typically enforced on functions using modifiers that check if specific addresses/roles are calling these functions. A system can have multiple roles with different privileges. Ensure that correct modifiers are used on functions requiring specific access control enforced by that modifier.

52. **Access control changes**: Ensure that changes to access control (e.g. change of ownership to new addresses) are handled with extra security so that such transitions happen smoothly without contracts getting locked out or compromised due to use of incorrect credentials.

53. **Comments**: Ensure that the code is well commented both with NatSpec and inline comments for better readability and maintainability. The comments should accurately reflect what the corresponding code does. Stale comments should be removed. Discrepancies between code and comments should be addressed. Any TODO's indicated by comments should be addressed. Commented code should be removed.

54. **Tests**: Tests indicate that the system implementation has been validated against the specification. Unit tests, functional tests and integration tests should have been performed to achieve good test coverage across the entire codebase. Any code or parameterisation used specifically for testing should be removed from production code.

55. **Unused constructs**: Any unused imports, inherited contracts, functions, parameters, variables, modifiers, events or return values should be removed (or used appropriately) after careful evaluation. This will not only reduce gas costs but improve readability and maintainability of the code.

56. **Redundant constructs**: Redundant code and comments can be confusing and should be removed (or changed appropriately) after careful evaluation. This will not only reduce gas costs but improve readability and maintainability of the code.

57. **ETH Handling**: Contracts that accept/manage/transfer ETH should ensure that functions handling ETH are using *msg.value* appropriately, logic that depends on ETH value accounts for less/more ETH sent, logic that depends on contract ETH balance accounts for the different direct/indirect (e.g. *coinbase* transaction, *selfdestruct* recipient) ways of receiving ETH and transfers are reentrancy safe. Functions handling ETH should be checked extra carefully for access control, input validation and error handling.

58. **Token Handling**: Contracts that accept/manage/transfer ERC tokens should ensure that functions handling tokens account for different types of ERC tokens (e.g. ERC20 vs ERC777), deflationary/inflationary tokens, rebasing tokens and trusted/external tokens. Functions handling tokens should be checked extra carefully for access control, input validation and error handling.

59. **Trusted actors**: Ideally there should be no trusted actors while interacting with smart contracts. However, in guarded launch scenarios, the goal is to start with trusted actors and then progressively decentralise towards automated governance by community/DAO. For the trusted phase, all the trusted actors, their roles and capabilities should be clearly specified, implemented accordingly and documented for user information and examination.

60. **Privileged roles and EOAs**: Trusted actors who have privileged roles with capabilities to deploy contracts, change critical parameters, pause/unpause system, trigger emergency shutdown, withdraw/transfer/drain funds and allow/deny other actors should be addresses controlled by multiple, independent, mutually distrusting entities. They should not be controlled by private keys of EOAs but with Multisigs with a high threshold (e.g. 5-of-7, 9-of-11) and eventually by a DAO of token holders. EOA has a single point of failure.

61. **Two-step change of privileged roles**: When privileged roles are being changed, it is recommended to follow a two-step approach: 1) The current privileged role proposes a new address for the change 2) The newly proposed address then claims the privileged role in a separate transaction. This two-step change allows accidental proposals to be corrected instead of leaving the system operationally with no/malicious privileged role. For e.g., in a single-step change, if the current admin accidentally changes the new admin to a zero-address or an incorrect address (where the private keys are not available), the system is left without an operational admin and will have to be redeployed.

62. **Time-delayed change of critical parameters**: When critical parameters of systems need to be changed, it is required to broadcast the change via event emission and recommended to enforce the changes after a time-delay. This is to allow system users to be aware of such critical changes and give them an opportunity to exit or adjust their engagement with the system accordingly. For e.g. reducing the rewards or increasing the fees in a system might not be acceptable to some users who may wish to withdraw their funds and exit.

63. **Explicit over Implicit**: While Solidity has progressively adopted explicit declarations of intent for e.g. with function visibility and variable storage, it is recommended to do the same at the application level where all requirements should be explicitly validated (e.g. input parameters) and assumptions should be documented and checked. Implicit requirements and assumptions should be flagged as dangerous.

64. **Configuration issues**: Misconfiguration of system components such contracts, parameters, addresses and permissions may lead to security issues.

65. **Initialization issues**: Lack of initialization, initializing with incorrect values or allowing untrusted actors to initialize system parameters may lead to security issues.

66. **Cleanup issues**: Missing to clean up old state or cleaning up incorrectly/insufficiently will lead to reuse of stale state which may lead to security issues.

67. **Data processing issues**: Processing data incorrectly will cause unexpected behavior which may lead to security issues.

68. **Data validation issues**: Missing validation of data or incorrectly/insufficiently validating data, especially tainted data from untrusted users, will cause untrustworthy system behavior which may lead to security issues.

69. **Numerical issues**: Incorrect numerical computation will cause unexpected behavior which may lead to security issues.

70. **Accounting issues**: Incorrect or insufficient tracking or accounting of business logic related aspects such as states, phases, permissions, roles, funds (deposits/withdrawals) and tokens (mints/burns/transfers) may lead to security issues.

71. **Access control issues**: Incorrect or insufficient access control or authorization related to system actors, roles, assets and permissions may lead to security issues.

72. **Auditing/logging issues**: Incorrect or insufficient emission of events will impact off-chain monitoring and incident response capabilities which may lead to security issues.

73. **Cryptography issues**: Incorrect or insufficient cryptography especially related to on-chain signature verification or off-chain key management will impact access control and may lead to security issues.

74. **Error-reporting issues**: Incorrect or insufficient detecting, reporting and handling of error conditions will cause exceptional behavior to go unnoticed which may lead to security issues.

75. **Denial-of-Service (DoS) issues**: Preventing other users from successfully accessing system services by modifying system parameters or state causes denial-of-service issues which affects the availability of the system. This may also manifest as security issues if users are not able to access their funds locked in the system.

76. **Timing issues**: Incorrect assumptions on timing of user actions, system state transitions or blockchain state/blocks/transactions may lead to security issues.

77. **Ordering issues**: Incorrect assumptions on ordering of user actions or system state transitions may lead to security issues. For e.g., a user may accidentally/maliciously call a finalization function even before the initialization function if the system allows.

78. **Undefined behavior issues**: Any behavior that is undefined in the specification but is allowed in the implementation will result in unexpected outcomes which may lead to security issues.

79. **External interaction issues**: Interacting with external components (e.g. tokens, contracts, oracles) forces system to trust or make assumptions on their correctness/availability requiring validation of their existence and outputs without which may lead to security issues.

80. **Trust issues**: Incorrect or Insufficient trust assumption about/among system actors and external entities will lead to privilege escalation/abuse which may lead to security issues.

81. **Gas issues**: Incorrect assumptions about gas requirements especially for loops or external calls will lead to out-of-gas exceptions which may lead to security issues such as failed transfers or locked funds.

82. **Dependency issues**: Dependencies on external actors or software (imports, contracts, libraries, tokens etc.) will lead to trust/availability/correctness assumptions which if/when broken may lead to security issues.

83. **Constant issues**: Incorrect assumptions about system actors, entities or parameters being constant may lead to security issues if/when such factors change unexpectedly.

84. **Freshness issues**: Incorrect assumptions about the status of or data from system actors or entities being fresh (i.e. not stale), because of lack of updation or availability, may lead to security issues if/when such factors have been updated. For e.g., getting a stale price from an Oracle.

85. **Scarcity issues**: Incorrect assumptions about the scarcity of tokens/funds available to any system actor will lead to unexpected outcomes which may lead to security issues. For e.g., flash loans.

86. **Incentive issues**: Incorrect assumptions about the incentives of system/external actors to perform or not perform certain actions will lead to unexpected behavior being triggered or expected behavior not being triggered, both of which may lead to security issues. For e.g., incentive to liquidate positions, lack of incentive to DoS or grief system.

87. **Clarity issues**: Lack of clarity in system specification, documentation, implementation or UI/UX will lead to incorrect expectations/outcome which may lead to security issues.

88. **Privacy issues**: Data and transactions on the Ethereum blockchain are not private. Anyone can observe contract state and track transactions (both included in a block and pending in the mempool). Incorrect assumptions about privacy aspects of data or transactions can be abused which may lead to security issues.

89. **Cloning issues**: Copy-pasting code from other libraries, contracts or even different parts of the same contract may result in incorrect code semantics for the context being copied to, copy over any vulnerabilities or miss any security fixes applied to the original code. All these may lead to security issues.

90. **Business logic issues**: Incorrect or insufficient assumptions about the higher-order business logic being implemented in the application will lead to differences in expected and actual behavior, which may result in security issues.

91. **Principle of Least Privilege**: "Every program and every user of the system should operate using the least set of privileges necessary to complete the job" --- Ensure that various system actors have the least amount of privilege granted as required by their roles to execute their specified tasks. Granting excess privilege is prone to misuse/abuse when trusted actors misbehave or their access is hijacked by malicious entities. (See Saltzer and Schroeder's Secure Design Principles)

92. **Principle of Separation of Privilege**: "Where feasible, a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single

key" --- Ensure that critical privileges are separated across multiple actors so that there are no single points of failure/abuse. A good example of this is to require a multisig address (not EOA) for privileged actors (e.g. owner, admin, governor, deployer) who control key contract functionality such as pause/unpause/shutdown, emergency fund drain, upgradeability, allow/deny list and critical parameters. The multisig address should be composed of entities that are different and mutually distrusting/verifying. (See Saltzer and Schroeder's Secure Design Principles)

93. **Principle of Least Common Mechanism**: "Minimize the amount of mechanism common to more than one user and depended on by all users" --- Ensure that only the least number of security-critical modules/paths as required are shared amongst the different actors/code so that impact from any vulnerability/compromise in shared components is limited and contained to the smallest possible subset. (See Saltzer and Schroeder's Secure Design Principles)

94. **Principle of Fail-safe Defaults**: "Base access decisions on permission rather than exclusion" --- Ensure that variables or permissions are initialized to fail-safe default values which can be made more inclusive later instead of opening up the system to everyone including untrusted actors. (See Saltzer and Schroeder's Secure Design Principles)

95. **Principle of Complete Mediation**: "Every access to every object must be checked for authority." --- Ensure that any required access control is enforced along all access paths to the object or function being protected. (See Saltzer and Schroeder's Secure Design Principles)

96. **Principle of Economy of Mechanism**: "Keep the design as simple and small as possible" --- Ensure that contracts and functions are not overly complex or large so as to reduce readability or maintainability. Complexity typically leads to insecurity. (See Saltzer and Schroeder's Secure Design Principles)

97. **Principle of Open Design**: "The design should not be secret" --- Smart contracts are expected to be open-sourced and accessible to everyone. Security by obscurity of code or underlying algorithms is not an option. Security should be derived from the strength of the design and implementation under the assumption that (byzantine) attackers will study their details and try to exploit them in arbitrary ways. (See Saltzer and Schroeder's Secure Design Principles)

98. **Principle of Psychological Acceptability**: "It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly" --- Ensure that security aspects of smart contract interfaces and system designs/flows are user-friendly and intuitive so that users can interact with minimal risk. (See Saltzer and Schroeder's Secure Design Principles)

99. **Principle of Work Factor**: "Compare the cost of circumventing the mechanism with the resources of a potential attacker" --- Given the magnitude of value managed by smart contracts, it is safe to assume that byzantine attackers will risk the greatest amounts of intellectual/financial/social capital possible to subvert such systems. Therefore, the mitigation mechanisms must factor in the highest levels of risk. (See Saltzer and Schroeder's Secure Design Principles)

100. **Principle of Compromise Recording**: "Mechanisms that reliably record that a compromise of information has occurred can be used in place of more elaborate mechanisms that completely prevent loss" --- Ensure that smart contracts and their accompanying operational infrastructure can be monitored/analyzed at all times (development/deployment/runtime) for minimizing loss from any

compromise due to vulnerabilities/exploits. For e.g., critical operations in contracts should necessarily emit events to facilitate monitoring at runtime. (See Saltzer and Schroeder's Secure Design Principles)