Growth Labs GSquared

Security Assessment

November 2, 2022

Prepared for:
Kristian Domanski
Growth Labs

Prepared by: **Gustavo Grieco, Michael Colburn, Anish Naik, and Damilola Edwards**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Growth Labs engaged Trail of Bits to review the security of its GSquared Solidity smart contracts. From September 26 to October 7, 2022, a team of four consultants conducted a security review of the client-provided source code, with six person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code and relevant documentation. We performed static and dynamic automated and manual testing of the target system, using both automated and manual processes.

Summary of Findings

The audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	6
Informational	8

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	9
Denial of Service	2
Timing	2
Undefined Behavior	3

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-GRO-1**
  An unbounded loop in `GVault` can cause a denial of service and prevent user withdrawals.

- **TOB-GRO-12**
  The migration from Gro protocol to GSquared protocol is vulnerable to being front-run with a share price manipulation attack, which may lead to a significant loss of funds.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Anne Marie Barry, Project Manager
annemarie.barry@trailofbits.com

The following engineers were associated with this project:

Gustavo Grieco, Consultant
gustavo.grieco@trailofbits.com

Michael Colburn, Consultant
michael.colburn@trailofbits.com

Anish Naik, Consultant
anish.naik@trailofbits.com

Damilola Edwards, Consultant
damilola.edwards@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
September 20, 2022	Pre-project kickoff call
September 30, 2022	Status update meeting #1
October 11, 2022	Delivery of report draft
October 11, 2022	Report readout meeting
November 2, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the Growth Labs's GSquared protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can an attacker steal funds from the system?
- Are there any economic attack vectors?
- Does the protocol design assert that the senior tranche is always protected?
- Is the GVault ERC4626 compliant?
- Are there any issues with oracles and pricing across the protocol?
- Are there front-running opportunities that can impact the system or its users?
- Can access controls be bypassed?
- Are conversions between tokens, shares, and dollars performed correctly? Is the share price prone to manipulation?
- Are profit and losses reported and distributed to the tranches according to the specification?
- Do the stop loss and harvest logic function as documented?

Project Targets

The engagement involved a review and testing of the following target.

GSquared

Repository	https://github.com/groLabs/GSquared/
Version	`b0cf03fa18b4549bd85c571c00e18ddf3218de59`
Type	Solidity
Platform	EVM

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- **`GRouter`**: The `GRouter` contract handles user deposits into and withdrawals out of the system. The contract directly interacts with Curve's three-pool, the `GTranche` contract, and the `GVault` contract. We performed a manual code review to ensure that the contract correctly interacts with the vault / tranche contracts, that the minimum-amount-out value is satisfied for deposits and withdrawals, and that token airdrops are properly handled.

- **`GTranche`**: The `GTranche` contract provides a novel way for insurance to be implemented on the blockchain, allowing users who seek a safer yield opportunity (Senior tranche) to do so by providing part of their deposit as leverage for an insurer (Junior tranche). We performed a manual code review to ensure that this contract correctly interacts with the tranche tokens, the vault, and the profit-and-loss contract. We also reviewed how the internal bookkeeping is tracked and whether the logic of profit distributions is sound. Finally, we reviewed the migration logic to ensure that a protocol migration cannot lead to a loss of funds or to imbalanced tranches.

- **`GVault`**: The `GVault` contract is a standalone token vault following the EIP-4626 standard. The vault's asset balance should increase over time and provide yield to tranche token holders. We performed a manual and automated review to look for issues related to the following: the handling of asset deposits and withdrawals; token-to-share and share-to-token conversions; share price manipulations using token airdrops; internal accounting to handle outstanding debt and to extend additional credit; deviations from the EIP-4626 standard; and general correctness of arithmetic operations.

- **`GMigration`**: The `GMigration` contract is responsible for migrating funds from the old Gro protocol to the new GSquared protocol. It converts stablecoins to 3CRV, deposits the 3CRV into the new `GVault`, and then deposits the G3CRV shares into the `GTranche`. We performed a manual review to look for logical flaws, access control issues, invalid upgrades, and denial of service performed by external users. Additionally, we also investigated the possibility of performing a migration more than once and whether share price manipulation can lead to a loss of funds for the protocol.

- **Oracle contracts**: There are three oracle-related contracts: `CurveOracle`, `RelationModule`, and `RouterOracle`. The `CurveOracle` contract retrieves the virtual price of the 3CRV token to provide pricing information to the `GTranche` contract. We performed a manual review to ensure that dollars-to-tokens and tokens-to-dollars conversions are performed correctly and exclusively used the *latest* virtual price. The `RelationModule` is an abstract contract with no state-changing logic, and the `RouterOracle` was considered out of scope (see the Coverage Limitations section below).

- **PnL contracts**: There are two PnL-related contracts: `PnLFixedRate` and `PnL`. The `PnLFixedRate` contract is designed to distribute the latest profits or losses to each tranche based on some heuristics. We manually reviewed this contract to ensure that the senior tranche receives its fixed APY, that the senior tranche does not incur losses unless the junior tranche is out of funds, that the junior tranche receives all the profits if the tranche utilization is too high, and that the pending rate is reflected upon the next PnL distribution. The `PnL` contract was considered out of scope (see Coverage Limitations section below).

- **`GStrategyGuard` and `GStrategyResolver`**: The `GStrategyResolver` contract is called by the Gelato keeper to identify whether it needs to perform a strategy harvest or to activate or deactivate the stop loss primer. The state-changing logic for performing those actions is in the `GStrategyGuard` contract. We manually reviewed these contracts to ensure that the necessary access controls are in place, that the Resolver calls the correct functions in the Guard to execute on an action, that the criteria for running a harvest are aligned with the documentation, that the criteria for running a stop loss are aligned with the documentation, and that the Guard correctly handles strategies that may behave unexpectedly or revert.

- **`StopLossLogic`**: The `StopLossLogic` contract holds the criteria for whether or not a meta pool is "healthy" or not. If a meta pool is no longer healthy, the stop loss process will initiate. We manually reviewed this contract to ensure that the arithmetic for calculating price deviations was performed correctly, that the necessary access controls were in place, and that a stop loss is a reversible process that can be halted if the pool becomes healthy again.

- **`ConvexStrategy`**: The `ConvexStrategy` contract is the primary strategy employed by the GSquared protocol. It integrates with Convex Finance to maximize yield earnings and reports back its PnL to the `GVault` contract. We manually reviewed that the necessary access controls were in place; that the internal bookkeeping and arithmetic for vault reports, withdrawals, and harvest operations were sound; that the stop loss process prevents further investment into the strategy; that emergency mode leads to a complete divestiture of funds; that the migration to a new meta pool does not open malicious attack vectors; the strategy's resilience to sandwich attacks; and the arithmetic for slippage tolerance and token conversions.

- **utils/**: The utils folder holds the `FixedTokensCurve`, `StrategyQueue`, and `FixedTokens` contracts. The `FixedTokensCurve` contract manages the yield tokens and tranche tokens used by the `GTranche` contract. We manually reviewed the contract to assess whether conversions from assets to shares were performed correctly, conversions from shares to assets were performed correctly, and conversions from shares to dollar value were performed correctly. The `StrategyQueue` contract provides a queue data structure for storing strategies and the necessary functions to manipulate them. We manually reviewed the contract to ensure that adding and removing operations functioned properly and that the queue could not be put into an inconsistent state. The `FixedTokens` contract was considered out of scope (see the Coverage Limitations section below).

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- The client noted that the following contracts (or folders) were of low priority. Thus, these contracts were considered out of scope:
  - The **tokens** folder holds the `AllowedPermit`, `ERC4626`, `GToken`, `JuniorTranche`, and `SeniorTranche` contracts.
  - The **common** folder holds the `Constants`, `Errors`, `PnLErrors`, and `Whitelist` contracts.
  - The **`FixedTokens`** and the **PnL** contracts
- **solmate/**: solmate is an external smart contract repository that holds a variety of gas-optimized contracts and standards. It is used extensively across the GSquared protocol. This folder was considered out of scope for this audit.
- **Complex or unexpected interactions with third-party protocols (e.g., Curve or Convex Finance).** We recommend that further review is performed on contracts that interact intimately with these systems (e.g., `ConvexStrategy`)
- **High-level economic issues with the protocol that may become a security risk**. Due to the size of the codebase and the complexity of some contracts, all potential economic issues or arithmetic risks were not evaluated during this audit. Additional review of the `ConvexStrategy` and `GVault` contracts is recommended.
- **Use of dynamic fuzz testing across various components**. Dynamic fuzz testing on each component as well as end-to-end would aid in validating system properties and help in identifying any edge cases of the system.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables	Used to detect common issues
Echidna	A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation	Appendix E

Areas of Focus

Our automated testing and verification work focused on the ERC4626 properties as well on making sure `GVault` code will not unexpectedly revert.

Test Results

The results of this focused testing are detailed below.

GVault We focused the testing on the ERC4626 expected properties of the vault.

Property	Tool	Result
`totalAssets` will never revert	Echidna	**Passed**
If the preconditions are met, `sharesToAssets` will never revert	Echidna	**Passed**
If the preconditions are met, `assetsToShares` will never	Echidna	**Passed**

revert		
Executing `sharesToAssets` and `assetsToShares` with an initial amount of shares will not result in an expected amount of them.	Echidna	**Passed**
`creditAvailable` will never revert	Echidna	**Passed**
Division rounding errors always favors the protocol	Echidna	TOB-GRO-11
If the preconditions are met, `withdraw` will never revert	Echidna	**Passed**
If the preconditions are met, `redeem` will never revert	Echidna	**Passed**

PnLFixedRate. We focused on testing that assets distribution code will not unexpectedly revert.

Property	Tool	Result
Given the preconditions for `distributeAssets` are met, it will never revert	Echidna	**Passed**

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The codebase uses Solidity v0.8.10 for arithmetic operations. Most arithmetic operations are documented, and a specification for a few of the more critical operations was provided. However, we identified a variety of issues that highlight the requirement for additional testing of arithmetic operations (TOB-GRO-1, TOB-GRO-8, TOB-GRO-13). Additionally, since the internal bookkeeping of the system is highly interconnected and distributed across different components, we recommend that these operations be investigated in detail and that additional testing methodologies, such as dynamic fuzz testing, be employed.	**Further Investigation Required**
Auditing	All functions involved in critical state-changing operations emit events detailing the state changes. Additionally, thorough documentation around the stop loss process was provided. We recommend that the client continue building their emergency response plan and develop documentation on how they plan to use off-chain monitoring to respond to malicious activity or unexpected behavior.	**Satisfactory**
Authentication / Access Controls	There are appropriate access controls in place for privileged operations, both for operations between contracts and for operations regarding contract administration and ownership. Additionally, all privileged actors and operations have been thoroughly documented.	**Satisfactory**

Complexity Management	The functions and contracts are organized and scoped appropriately and contain inline documentation that explains their workings. Additionally, there is a clear separation of duties between each of the core components in the system. However, some contracts have high cyclomatic complexity that make them difficult to review and test.	**Moderate**
Decentralization	The number of privileged actors is limited and all operations performed by privileged actors are documented. Only a privileged actor can update critical system parameters. The system depends on the decentralization of external systems, such as Curve and Convex Finance, but this does not prevent a user from exiting the system at any point, assuming the utilization requirements are met.	**Satisfactory**
Documentation	The project has thorough high-level documentation, uses the NatSpec format, and has extensive inline comments. A mathematical specification on some of the more critical arithmetic operations was also provided. We recommend that the client continue building on their documentation of critical system-level invariants.	**Satisfactory**
Front-Running Resistance	Some potential front-running and arbitrage opportunities have been identified and mitigations are implemented. However, we identified two issues (TOB-GRO-6, TOB-GRO-12) that would allow a bot to front-run token swaps and migrations and cause a loss of funds to the protocol. Given the identification of two potential attack vectors, front-running resistance should be further investigated and all potential vectors should be thoroughly documented.	**Further Investigation Required**
Testing and Verification	The codebase contains a number of unit and integration tests. However, the tests were insufficient to catch some of the issues identified (TOB-GRO-1, TOB-GRO-8, TOB-GRO-13), which indicates that the test suite should be improved. Moreover, the codebase would benefit from advanced testing techniques such as fuzzing.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Unbounded loop can cause denial of service	Denial of Service	High
2	Lack of two-step process for contract ownership changes	Data Validation	Informational
3	Non-zero token balances in the GRouter can be stolen	Data Validation	Informational
4	Uninformative implementation of maxDeposit and maxMint from EIP-4626	Undefined Behavior	Informational
5	moveStrategy runs of out gas for large inputs	Undefined Behavior	High
6	GVault withdrawals from ConvexStrategy are vulnerable to sandwich attacks	Timing	Medium
7	Stop loss primer cannot be deactivated	Data Validation	Medium
8	getYieldTokenAmount uses convertToAssets instead of convertToShares	Data Validation	Medium
9	convertToShares can be manipulated to block deposits	Data Validation	Medium
10	Harvest operation could be blocked if eligibility check on a strategy reverts	Denial of Service	Informational
11	Incorrect rounding direction in GVault	Data Validation	Medium

12	Protocol migration is vulnerable to front-running and a loss of funds	Timing	**High**
13	Incorrect slippage calculation performed during strategy investments and divestitures	Data Validation	**Medium**
14	Potential division by zero in _calcTrancheValue	Data Validation	**Informational**
15	Token withdrawals from GTranche are sent to the incorrect address	Data Validation	**Informational**
16	Solidity compiler optimizations can be problematic	Undefined Behavior	**Informational**

Detailed Findings

1. Unbounded loop can cause denial of service

Severity: **High**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-GRO-1
Target: `contracts/GVault.sol`	

Description

Under certain conditions, the withdrawal code will loop, permanently blocking users from getting their funds.

The `beforeWithdraw` function runs before any withdrawal to ensure that the vault has sufficient assets. If the vault reserves are insufficient to cover the withdrawal, it loops over each strategy, incrementing the `_strategyId` pointer value with each iteration, and withdrawing assets to cover the withdrawal amount.

```
 643    function beforeWithdraw(uint256 _assets, ERC20 _token)
 644       internal
 645       returns (uint256)
 646    {
 647       // If reserves dont cover the withdrawal, start withdrawing from
strategies
 648       if (_assets > _token.balanceOf(address(this))) {
 649           uint48 _strategyId = strategyQueue.head;
 650           while (true) {
 651               address _strategy = nodes[_strategyId].strategy;
 652               uint256 vaultBalance = _token.balanceOf(address(this));
 653               // break if we have withdrawn all we need
 654               if (_assets <= vaultBalance) break;
 655               uint256 amountNeeded = _assets - vaultBalance;
 656
 657               StrategyParams storage _strategyData = strategies[_strategy];
 658               amountNeeded = Math.min(amountNeeded, _strategyData.totalDebt);
 659               // If nothing is needed or strategy has no assets, continue
 660               if (amountNeeded == 0) {
 661                   continue;
 662               }
```

Figure 1.1: The `beforeWithdraw` function in `GVault.sol#L643-662`

However, during an iteration, if the vault raises enough assets that the amount needed by the vault becomes zero or that the current strategy no longer has assets, the loop would keep using the same `strategyId` until the transaction runs out of gas and fails, blocking the withdrawal.

Exploit Scenario

Alice tries to withdraw funds from the protocol. The contract may be in a state that sets the conditions for the internal loop to run indefinitely, resulting in the waste of all sent gas, the failure of the transaction, and blocking all withdrawal requests.

Recommendations

Short term, add logic to increment the `_strategyId` variable to point to the next strategy in the `StrategyQueue` before the continue statement.

Long term, use unit tests and fuzzing tools like Echidna to test that the protocol works as expected, even for edge cases.

2. Lack of two-step process for contract ownership changes

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-GRO-2
Target: `contracts/pnl/PnLFixedRate.sol`	

Description

The `setOwner()` function is used to change the owner of the `PnLFixedRate` contract. Transferring ownership in one function call is error-prone and could result in irrevocable mistakes.

```
56    function setOwner(address _owner) external {
57        if (msg.sender != owner) revert PnLErrors.NotOwner();
58        address previous_owner = msg.sender;
59        owner = _owner;
60
61        emit LogOwnershipTransferred(previous_owner, _owner);
62    }
```

Figure 2.1: `contracts/pnl/PnLFixedRate:56-62`

This issue can also be found in the following locations:

- `contracts/pnl/PnL.sol:36-42`
- `contracts/strategy/ConvexStrategy.sol:447-453`
- `contracts/strategy/keeper/GStrategyGuard.sol:92-97`
- `contracts/strategy/stop-loss/StopLossLogic.sol:73-78`

Exploit Scenario

The owner of the `PnLFixedRate` contract is a governance-controlled multisignature wallet. The community agrees to change the owner of the strategy, but the wrong address is mistakenly provided to its call to `setOwner`, permanently misconfiguring the system.

Recommendations

Short term, implement a two-step process to transfer contract ownership, in which the owner proposes a new address and then the new address executes a call to accept the role, completing the transfer.

Long term, review how critical operations are implemented across the codebase to make sure they are not error-prone.

3. Non-zero token balances in the GRouter can be stolen

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-GRO-3
Target: `GRouter.sol`	

Description

A non-zero balance of 3CRV, DAI, USDC, or USDT in the router contract can be stolen by an attacker.

The `GRouter` contract is the entrypoint for deposits into a tranche and withdrawals out of a tranche. A deposit involves depositing a given number of a supported stablecoin (USDC, DAI, or USDT); converting the deposit, through a series of operations, into G3CRV, the protocol's ERC4626-compatible vault token; and depositing the G3CRV into a tranche. Similarly, for withdrawals, the user burns their G3CRV that was in the tranche and, after a series of operations, receives back some amount of a supported stablecoin (figure 3.1).

```
function withdrawFromTrancheForCaller(
    uint256 _amount,
    uint256 _token_index,
    bool _tranche,
    uint256 _minAmount
) internal returns (uint256 amount) {
    ERC20(address(tranche.getTrancheToken(_tranche))).safeTransferFrom(
        msg.sender,
        address(this),
        _amount
    );
    // withdraw from tranche
    // index is zero for ETH mainnet as their is just one yield token
    // returns usd value of withdrawal
    (uint256 vaultTokenBalance, ) = tranche.withdraw(
        _amount,
        0,
        _tranche,
        address(this)
    );

    // withdraw underlying from GVault
    uint256 underlying = vaultToken.redeem(
        vaultTokenBalance,
        address(this),
        address(this)
    );
448
449        // remove liquidity from 3crv to get desired stable from curve
450        threePool.remove_liquidity_one_coin(
451            underlying,
452            int128(uint128(_token_index)), //value should always be 0,1,2
453            0
454        );
455
456        ERC20 stableToken = ERC20(routerOracle.getToken(_token_index));
457
458        amount = stableToken.balanceOf(address(this));
459
460        if (amount < _minAmount) {
461            revert Errors.LTMinAmountExpected();
462        }
463
464        // send stable to user
465        stableToken.safeTransfer(msg.sender, amount);
466
467        emit LogWithdrawal(msg.sender, _amount, _token_index, _tranche, amount);
468    }
```

Figure 3.1: The `withdrawFromTrancheForCaller` function in `GRouter.sol#L421-468`

However, notice that during withdrawals the amount of `stableTokens` that will be transferred back to the user is a function of the current `stableToken` balance of the contract (see the highlighted line in figure 3.1). In the expected case, the balance should be only the tokens received from the `threePool.remove_liquidity_one_coin` swap (see L450 in figure 3.1). However, a non-zero balance could also occur if a user airdrops some tokens or they transfer tokens by mistake instead of calling the expected `deposit` or `withdraw` functions. As long as the attacker has at least 1 wei of G3CRV to burn, they are capable of withdrawing the whole balance of `stableToken` from the contract, regardless of how much was received as part of the `threePool` swap. A similar situation can happen with deposits. A non-zero balance of G3CRV can be stolen as long as the attacker has at least 1 wei of either DAI, USDC, or USDT.

Exploit Scenario

Alice mistakenly sends a large amount of DAI to the `GRouter` contract instead of calling the `deposit` function. Eve notices that the `GRouter` contract has a non-zero balance of DAI and calls `withdraw` with a negligible balance of G3CRV. Eve is able to steal Alice's DAI at a very small cost.

Recommendations

Short term, consider using the difference between the contract's pre- and post-balance of `stableToken` for withdrawals, and `depositAmount` for deposits, in order to ensure that only the newly received tokens are used for the operations.

Long term, create an external `skim` function that can be used to skim any excess tokens in the contract. Additionally, ensure that the user documentation highlights that users should not transfer tokens directly to the `GRouter` and should instead use the web interface or call the `deposit` and `withdraw` functions. Finally, ensure that token airdrops or unexpected transfers can only benefit the protocol.

4. Uninformative implementation of maxDeposit and maxMint from EIP-4626

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-GRO-4
Target: `GVault.sol`	

Description

The GVault implementation of EIP-4626 is uninformative for `maxDeposit` and `maxMint`, as they return only fixed, extreme values.

EIP-4626 is a standard to implement tokenized vaults. In particular, the following is specified:

- `maxDeposit`: MUST factor in both global and user-specific limits, like if deposits are entirely disabled (even temporarily) it MUST return 0. MUST return 2 `** 256 - 1` if there is no limit on the maximum amount of assets that may be deposited.
- `maxMint`: MUST factor in both global and user-specific limits, like if mints are entirely disabled (even temporarily) it MUST return 0. MUST return 2 `** 256 - 1` if there is no limit on the maximum amount of assets that may be deposited.

The current implementation of `maxDeposit` and `maxMint` in the GVault contract directly return the maximum value of the uint256 type:

```
/// @notice The maximum amount a user can deposit into the vault
function maxDeposit(address)
    public
    pure
    override
    returns (uint256 maxAssets)
{
    return type(uint256).max;
}
.
.
.
/// @notice maximum number of shares that can be minted
function maxMint(address) public pure override returns (uint256 maxShares) {
    return type(uint256).max;
}
```

Figure 4.1: The `maxDeposit` and `maxMint` functions from `GVault.sol`

This implementation, however, does not provide any valuable information to the user and may lead to faulty integrations with third-party systems.

Exploit Scenario

A third-party protocol wants to deposit into a `GVault`. It first calls `maxDeposit` to know the maximum amount of asserts it can deposit and then calls `deposit`. However, the latter function call will revert because the value is too large.

Recommendations

Short term, return suitable values in `maxDeposit` and `maxMint` by considering the amount of assets owned by the caller as well any other global condition (e.g., a contract is paused).

Long term, ensure compliance with the EIP specification that is being implemented (in this case, EIP-4626).

5. moveStrategy runs of out gas for large inputs

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-GRO-5
Target: `GVault.sol`	

Description

Reordering strategies can trigger operations that will run out-of-gas before completion.

A GVault contract allows different strategies to be added into a queue. Since the order of them is important, the contract provides `moveStrategy`, a function to let the owner to move a strategy to a certain position of the queue.

```
/// @notice Move the strategy to a new position
/// @param _strategy Target strategy to move
/// @param _pos desired position of strategy
/// @dev if the _pos value is >= number of strategies in the queue,
///      the strategy will be moved to the tail position
function moveStrategy(address _strategy, uint256 _pos) external onlyOwner {
   uint256 currentPos = getStrategyPositions(_strategy);
   uint256 _strategyId = strategyId[_strategy];
   if (currentPos > _pos)
       move(uint48(_strategyId), uint48(currentPos - _pos), false);
   else move(uint48(_strategyId), uint48(_pos - currentPos), true);
}
```

Figure 5.1: The `moveStrategy` function from `GVault.sol`

The documentation states that if the position to move a certain strategy is larger than the number of strategies in the queue, then it will be moved to the tail of the queue. This implemented using the `move` function:

```
/// @notice move a strategy to a new position in the queue
/// @param _id id of strategy to move
/// @param _steps number of steps to move the strategy
/// @param _back move towards tail (true) or head (false)
/// @dev Moves a strategy a given number of steps. If the number
///        of steps exceeds the position of the head/tail, the
///        strategy will take the place of the current head/tail
function move(
   uint48 _id,
   uint48 _steps,
   bool _back
) internal {
    Strategy storage oldPos = nodes[_id];
    if (_steps == 0) return;
    if (oldPos.strategy == ZERO_ADDRESS) revert NoIdEntry(_id);
    uint48 _newPos = !_back ? oldPos.prev : oldPos.next;

    for (uint256 i = 1; i < _steps; i++) {
        _newPos = !_back ? nodes[_newPos].prev : nodes[_newPos].next;
    }
…
```

Figure 5.2: The header of the move function from `StrategyQueue.sol`

However, if a large number of steps is used, the loop will never finish without running out of gas.

A similar issue affects `StrategyQueue.withdrawalQueue`, if called directly.

Exploit Scenario

Alice creates a smart contract that acts as the owner of a GVault. She includes code to reorder strategies using a call to `moveStrategy`. Since she wants to ensure that a certain strategy is always moved to the end of the queue, she uses a very large value as the position. When the code runs, it will always run out of gas, blocking the operation.

Recommendations

Short term, ensure the execution of the move ends in a number of steps that is bounded by the number of strategies in the queue.

Long term, use unit tests and fuzzing tools like Echidna to test that the protocol works as expected, even for edge cases.

6. GVault withdrawals from ConvexStrategy are vulnerable to sandwich attacks	
Severity: **Medium**	Difficulty: **High**
Type: Timing	Finding ID: TOB-GRO-6
Target: `strategy/ConvexStrategy.sol`	

Description

Token swaps that may be executed during vault withdrawals are vulnerable to sandwich attacks. Note that this is applicable only if a user withdraws directly from the `GVault`, not through the `GRouter` contract.

The `ConvexStrategy` contract performs token swaps through Uniswap V2, Uniswap V3, and Curve. All platforms allow the caller to specify the minimum-amount-out value, which indicates the minimum amount of tokens that a user wishes to receive from a swap. This provides protection against illiquid pools and sandwich attacks. Many of the swaps that the `ConvexStrategy` contract performs have the minimum-amount-out value hardcoded to zero. But a majority of these swaps can be triggered only by a Gelato keeper, which uses a private channel to relay all transactions. Thus, these swaps cannot be sandwiched.

However, this is not the case with the `ConvexStrategy.withdraw` function. The withdraw function will be called by the `GVault` contract if the `GVault` does not have enough tokens for a user withdrawal. If the balance is not sufficient, `ConvexStrategy.withdraw` will be called to retrieve additional assets to complete the withdrawal request. Note that the transaction to withdraw assets from the protocol will be visible in the public mempool (figure 6.1).

```
function withdraw(uint256 _amount)
   external
   returns (uint256 withdrawnAssets, uint256 loss)
{
   if (msg.sender != address(VAULT)) revert StrategyErrors.NotVault();
   (uint256 assets, uint256 balance, ) = _estimatedTotalAssets(false);
   // not enough assets to withdraw
   if (_amount >= assets) {
       balance += sellAllRewards();
       balance += divestAll(false);
       if (_amount > balance) {
           loss = _amount - balance;
           withdrawnAssets = balance;
       } else {
           withdrawnAssets = _amount;
```

```
            }
        } else {
            // check if there is a loss, and distribute it proportionally
            //  if it exists
            uint256 debt = VAULT.getStrategyDebt();
            if (debt > assets) {
                loss = ((debt - assets) * _amount) / debt;
                _amount = _amount - loss;
            }
            if (_amount <= balance) {
                withdrawnAssets = _amount;
            } else {
                withdrawnAssets = divest(_amount - balance, false) + balance;
                if (withdrawnAssets < _amount) {
                    loss += _amount - withdrawnAssets;
                } else {
                    if (loss > withdrawnAssets - _amount) {
                        loss -= withdrawnAssets - _amount;
                    } else {
                        loss = 0;
                    }
                }
            }
        }
        ASSET.transfer(msg.sender, withdrawnAssets);
        return (withdrawnAssets, loss);
    }
```

Figure 6.1: The `withdraw` function in `ConvexStrategy.sol#L771-812`

In the situation where the `_amount` that needs to be `withdrawn` is more than or equal to the total number of assets held by the contract, the `withdraw` function will call `sellAllRewards` and `divestAll` with `_slippage` set to `false` (see the highlighted portion of figure 6.1). The `sellAllRewards` function, which will call `_sellRewards`, sells all the additional reward tokens provided by Convex, its balance of CRV, and its balance of CVX for WETH. All these swaps have a hardcoded value of zero for the minimum-amount-out. Similarly, if `_slippage` is set to `false` when calling `divestAll`, the swap specifies a minimum-amount-out of zero.

By specifying zero for all these token swaps, there is no guarantee that the protocol will receive any tokens back from the trade. For example, if one or more of these swaps get sandwiched during a call to `withdraw`, there is an increased risk of reporting a loss that will directly affect the amount the user is able to withdraw.

Exploit Scenario

Alice makes a call to `withdraw` to remove some of her funds from the protocol. Eve notices this call in the public transaction mempool. Knowing that the contract will have to sell some of its rewards, Eve identifies a pure profit opportunity and sandwiches one or more of the swaps performed during the transaction. The strategy now has to report a loss, which results in Alice receiving less than she would have otherwise.

Recommendations

Short term, for `_sellRewards`, use the same `minAmount` calculation as in `divestAll` but replace `debt` with the contract's balance of a given reward token. This can be applied for all swaps performed in `_sellRewards`. For `divestAll`, set `_slippage` to `true` instead of `false` when it is called in `withdraw`.

Long term, document all cases in which front-running may be possible and its implications for the codebase. Additionally, ensure that all users are aware of the risks of front-running and arbitrage when interacting with the GSquared system.

7. Stop loss primer cannot be deactivated

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-GRO-7
Target: `strategy/keeper/GStrategyResolver.sol`	

Description

The stop loss primer cannot be deactivated because the keeper contract uses the incorrect function to check whether or not the meta pool has become healthy again.

The stop loss primer is activated if the meta pool that is being used for yield becomes unhealthy. A meta pool is unhealthy if the price of the 3CRV token deviates from the expected price for a set amount of time. The primer can also be deactivated if, after it has been activated, the price of the token stabilizes back to a healthy value. Deactivating the primer is a critical feature because if the pool becomes healthy again, there is no reason to divest all of the strategy's funds, take potential losses, and start all over again.

The `GStrategyResolver` contract, which is called by a Gelato keeper, will check to identify whether a primer can be deactivated. This is done via the `taskStopStopLossPrimer` function. The function will attempt to call the `GStrategyGuard.endStopLoss` function to see whether the primer can be deactivated (figure 7.1).

```
function taskStopStopLossPrimer()
   external
   view
   returns (bool canExec, bytes memory execPayload)
{
   IGStrategyGuard executor = IGStrategyGuard(stopLossExecutor);
   if (executor.endStopLoss()) {
       canExec = true;
       execPayload = abi.encodeWithSelector(
           executor.stopStopLossPrimer.selector
       );
   }
}
```

Figure 7.1: The `taskStopStopLossPrimer` function in `GStrategyResolver.sol#L46-58`

However, the `GStrategyGuard` contract does not have an `endStopLoss` function. Instead, it has a `canEndStopLoss` function. Note that the `executor` variable in `taskStopStopLossPrimer` is expected to implement the `IGStrategyGuard` function, which does have an `endStopLoss` function. However, the `GStrategyGuard` contract implements the `IGuard` interface, which does not have the `endStopLoss` function. Thus, the call to `endStopLoss` will simply return, which is equivalent to returning `false`, and the primer will not be deactivated.

Exploit Scenario
Due to market conditions, the price of the 3CRV token drops significantly for an extended period of time. This triggers the Gelato keeper to activate the stop loss primer. Soon after, the price of the 3CRV token restabilizes. However, because of the incorrect function call in the `taskStopStopLossPrimer` function, the primer cannot be deactivated, the stop loss process completes, and all the funds in the strategy must be divested.

Recommendations
Short term, change the function call from `endStopLoss` to `canEndStopLoss` in `taskStopStopLossPrimer`.

Long term, ensure that there are no near-duplicate interfaces for a given contract in the protocol that may lead to an edge case similar to this. Additionally, expand the unit test suite to cover additional edge cases and to ensure that the system behaves as expected.

8. getYieldTokenAmount uses convertToAssets instead of convertToShares

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-GRO-8
Target: `GTranche.sol`	

Description

The `getYieldTokenAmount` function does not properly convert a 3CRV token amount into a G3CRV token amount, which may allow a user to withdraw more or less than expected or lead to imbalanced tranches after a migration.

The expected behavior of the `getYieldTokenAmount` function is to return the number of G3CRV tokens represented by a given 3CRV amount. For withdrawals, this will determine how many G3CRV tokens should be returned back to the `GRouter` contract. For migrations, the function is used to figure out how many G3CRV tokens should be allocated to the senior and junior tranches.

To convert a given amount of 3CRV to G3CRV, the `GVault.convertToShares` function should be used. However, the `getYieldTokenAmount` function uses the `GVault.convertToAssets` function (figure 8.1). Thus, `getYieldTokenAmount` takes an amount of 3CRV tokens and treats it as shares in the `GVault`, instead of assets.

```
169    function getYieldTokenAmount(uint256 _index, uint256 _amount)
170       internal
171       view
172       returns (uint256)
173    {
174       return getYieldToken(_index).convertToAssets(_amount);
175    }
```

Figure 8.1: The `getYieldTokenAmount` function in `GTranche.sol#L169-175`

If the system is profitable, each G3CRV share should be worth more over time. Thus, `getYieldTokenAmount` will return a value larger than expected because one share is worth more than one asset. This allows a user to withdraw more from the `GTranche` contract than they should be able to. Additionally, a profitable system will cause the senior tranche to receive more G3CRV tokens than expected during migrations. A similar situation can happen if the system is not profitable.

Exploit Scenario

Alice deposits $100 worth of USDC into the system. After a certain amount of time, the GSquared protocol becomes profitable and Alice should be able to withdraw $110, making $10 in profit. However, due to the incorrect arithmetic performed in the `getYieldTokenAmount` function, Alice is able to withdraw $120 of USDC.

Recommendations

Short term, use `convertToShares` instead of `convertToAssets` in `getYieldTokenAmount`.

Long term, expand the unit test suite to cover additional edge cases and to ensure that the system behaves as expected.

9. convertToShares can be manipulated to block deposits	
Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-GRO-9
Target: `GVault.sol`	

Description

An attacker can block operations by using direct token transfers to manipulate `convertToShares`, which computes the amount of shares to deposit.

`convertToShares` is used in the GVault code to know how many shares correspond to certain amount of assets:

```
/// @notice Value of asset in shares
/// @param _assets amount of asset to convert to shares
function convertToShares(uint256 _assets)
   public
   view
   override
   returns (uint256 shares)
{
   uint256 freeFunds_ = _freeFunds(); // Saves an extra SLOAD if _freeFunds
is non-zero.
   return freeFunds_ == 0 ? _assets : (_assets * totalSupply) / freeFunds_;
}
```

Figure 9.1: The `convertToShares` function in `GVault.sol`

This function relies on the `_freeFunds` function to calculate the amount of shares:

```
/// @notice the number of total assets the GVault has excluding and profits
/// and losses
function _freeFunds() internal view returns (uint256) {
   return _totalAssets() - _calculateLockedProfit();
}
```

Figure 9.2: The `_freeFunds` function in `GVault.sol`

In the simplest case, `_calculateLockedProfit()` can be assumed as zero if there is no locked profit. The `_totalAssets` function is implemented as follows:

```
/// @notice Vault adapters total assets including loose assets and debts
/// @dev note that this does not consider estimated gains/losses from the
strategies
function _totalAssets() private view returns (uint256) {
    return asset.balanceOf(address(this)) + vaultTotalDebt;
}
```

Figure 9.3: The `_totalAssets` function in `GVault.sol`

However, the fact that `_totalAssets` has a lower bound determined by `asset.balanceOf(address(this))` can be exploited to manipulate the result by "donating" assets to the GVault address.

Exploit Scenario

Alice deploys a new GVault. Eve observes the deployment and quickly transfers an amount of tokens to the GVault address. One of two scenarios can happen:

1. Eve transfers a minimal amount of tokens, forcing a positive amount of `freeFunds`. This will block any immediate calls to deposit, since it will result in zero shares to be minted.
2. Eve transfers a large amount of tokens, forcing future deposits to be more expensive or resulting in zero shares. Every new deposit can increase the amount of free funds, making the effect more severe.

It is important to note that although Alice cannot use the `deposit` function, she can still call `mint` to bypass the exploit.

Recommendations

Short term, use a state variable, `assetBalance`, to track the total balance of assets in the contract. Avoid using `balanceOf`, which is prone to manipulation.

Long term, expand the unit test suite to cover additional edge cases and to ensure that the system behaves as expected.

10. Harvest operation could be blocked if eligibility check on a strategy reverts

Severity: **Informational**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-GRO-10
Target: `contracts/strategy/keeper/GStrategyGuard.sol`	

Description

During harvest, if any of the strategies in the queue were to revert, it would prevent the loop from reaching the end of the queue and also block the entire harvest operation.

When the `harvest` function is executed, a loop iterates through each of the strategies in the strategies queue, and the `canHarvest()` check runs on each strategy to determine if it is eligible for harvesting; if it is, the harvest logic is executed on that strategy.

```
/// @notice Execute strategy harvest
function harvest() external {
    if (msg.sender != keeper) revert GuardErrors.NotKeeper();
    uint256 strategiesLength = strategies.length;
    for (uint256 i; i < strategiesLength; i++) {
        address strategy = strategies[i];
        if (strategy == address(0)) continue;
        if (IStrategy(strategy).canHarvest()) {
            if (strategyCheck[strategy].active) {
                IStrategy(strategy).runHarvest();
                try IStrategy(strategy).runHarvest() {} catch Error(
        ...
```

Figure 10.1: The `harvest` function in `GStrategyGuard.sol`

However, if the `canHarvest()` check on a particular strategy within the loop reverts, external calls from the `canHarvest()` function to check the status of rewards could also revert. Since the call to `canHarvest()` is not inside of a try block, this would prevent the loop from proceeding to the next strategy in the queue (if there is one) and would block the entire harvest operation.

Additionally, within the `harvest` function, the `runHarvest` function is called twice on a strategy on each iteration of the loop. This could lead to unnecessary waste of gas and possibly undefined behavior.

Recommendations

Short term, wrap external calls within the loop in try and catch blocks, so that reverts can be handled gracefully without blocking the entire operation. Additionally, ensure that the `canHarvest` function of a strategy can never revert.

Long term, carefully audit operations that consume a large amount of gas, especially those in loops. Additionally, when designing logic loops that make external calls, be mindful as to whether the calls can revert, and wrap them in try and catch blocks when necessary.

11. Incorrect rounding direction in GVault

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-GRO-11
Target: `GVault.sol`	

Description

The minting and withdrawal operations in the GVault use rounding in favor of the user instead of the protocol, giving away a small amount of shares or assets that can accumulate over time.

`convertToShares` is used in the GVault code to know how many shares correspond to a certain amount of assets:

```
/// @notice Value of asset in shares
/// @param _assets amount of asset to convert to shares
function convertToShares(uint256 _assets)
    public
    view
    override
    returns (uint256 shares)
{
    uint256 freeFunds_ = _freeFunds(); // Saves an extra SLOAD if _freeFunds
is non-zero.
    return freeFunds_ == 0 ? _assets : (_assets * totalSupply) / freeFunds_;
}
```

Figure 11.1: The `convertToShares` function in `GVault.sol`

This function rounds down, providing slightly fewer shares than expected for some amount of assets.

Additionally, `convertToAssets` is used in the GVault code to know how many assets correspond to certain amount of shares:

```
/// @notice Value of shares in underlying asset
/// @param _shares amount of shares to convert to tokens
function convertToAssets(uint256 _shares)
    public
    view
    override
    returns (uint256 assets)
{
        uint256 _totalSupply = totalSupply; // Saves an extra SLOAD if
_totalSupply is non-zero.
        return
            _totalSupply == 0
                ? _shares
                : ((_shares * _freeFunds()) / _totalSupply);
    }
```

Figure 11.2: The `convertToAssets` function in `GVault.sol`

This function also rounds down, providing slightly fewer assets than expected for some amount of shares.

However, the `mint` function uses `previewMint`, which uses `convertToAssets`:

```
    function mint(uint256 _shares, address _receiver)
        external
        override
        nonReentrant
        returns (uint256 assets)
    {
        // Check for rounding error in previewMint.
        if ((assets = previewMint(_shares)) == 0) revert Errors.ZeroAssets();

        _mint(_receiver, _shares);

        asset.safeTransferFrom(msg.sender, address(this), assets);

        emit Deposit(msg.sender, _receiver, assets, _shares);

        return assets;
    }
```

Figure 12.3: The `mint` function in `GVault.sol`

This means that the function favors the user, since they get some fixed amount of shares for a rounded-down amount of assets.

In a similar way, the `withdraw` function uses `convertToShares`:

```
    function withdraw(
        uint256 _assets,
        address _receiver,
        address _owner
    ) external override nonReentrant returns (uint256 shares) {
        if (_assets == 0) revert Errors.ZeroAssets();

        shares = convertToShares(_assets);

        if (shares > balanceOf[_owner]) revert Errors.InsufficientShares();

        if (msg.sender != _owner) {
            uint256 allowed = allowance[_owner][msg.sender]; // Saves gas for
limited approvals.

            if (allowed != type(uint256).max)
                allowance[_owner][msg.sender] = allowed - shares;
        }

        _assets = beforeWithdraw(_assets, asset);

        _burn(_owner, shares);

        asset.safeTransfer(_receiver, _assets);

        emit Withdraw(msg.sender, _receiver, _owner, _assets, shares);

        return shares;
    }
```

Figure 11.4: The `withdraw` function in `GVault.sol`

This means that the function favors the user, since they get some fixed amount of assets for a rounded-down amount of shares.

This issue should also be also considered when minting fees, since they should favor the protocol instead of the user or the strategy.

Exploit Scenario

Alice deploys a new GVault and provides some liquidity. Eve uses mints and withdrawals to slowly drain the liquidity, possibly affecting the internal bookkeeping of the GVault.

Recommendations

Short term, consider refactoring the GVault code to specify the rounding direction across the codebase in order keep the error in favor of the user or the protocol.

Long term, expand the unit test suite to cover additional edge cases and to ensure that the system behaves as expected.

12. Protocol migration is vulnerable to front-running and a loss of funds

Severity: **High**	Difficulty: **High**
Type: Timing	Finding ID: TOB-GRO-12
Target: `GMigration.sol`	

Description

The migration from Gro protocol to GSquared protocol can be front-run by manipulating the share price enough that the protocol loses a large amount of funds.

The `GMigration` contract is responsible for initiating the migration from Gro to GSquared. The `GMigration.prepareMigration` function will deposit liquidity into the three-pool and then attempt to deposit the 3CRV LP token into the `GVault` contract in exchange for G3CRV shares (figure 12.1). Note that this migration occurs on a newly deployed `GVault` contract that holds no assets and has no supply of shares.

```
    function prepareMigration(uint256 minAmountThreeCRV) external onlyOwner {
        if (!IsGTrancheSet) {
            revert Errors.TrancheNotSet();
        }

        // read senior tranche value before migration
        seniorTrancheDollarAmount = SeniorTranche(PWRD).totalAssets();

        uint256 DAI_BALANCE = ERC20(DAI).balanceOf(address(this));
        uint256 USDC_BALANCE = ERC20(USDC).balanceOf(address(this));
        uint256 USDT_BALANCE = ERC20(USDT).balanceOf(address(this));

        // approve three pool
        ERC20(DAI).safeApprove(THREE_POOL, DAI_BALANCE);
        ERC20(USDC).safeApprove(THREE_POOL, USDC_BALANCE);
        ERC20(USDT).safeApprove(THREE_POOL, USDT_BALANCE);

        // swap for 3crv
        IThreePool(THREE_POOL).add_liquidity(
            [DAI_BALANCE, USDC_BALANCE, USDT_BALANCE],
            minAmountThreeCRV
        );

        //check 3crv amount received
        uint256 depositAmount = ERC20(THREE_POOL_TOKEN).balanceOf(
            address(this)
        );

```

```
            // approve 3crv for GVault
            ERC20(THREE_POOL_TOKEN).safeApprove(address(gVault), depositAmount);

            // deposit into GVault
            uint256 shareAmount = gVault.deposit(depositAmount, address(this));

            // approve gVaultTokens for gTranche
            ERC20(address(gVault)).safeApprove(address(gTranche), shareAmount);
        }
    }
```
Figure 12.1: The `prepareMigration` function in `GMigration.sol#L61-98`

However, this `prepareMigration` function call is vulnerable to a share price inflation attack. As noted in this issue, the end result of the attack is that the shares (G3CRV) that the `GMigration` contract will receive can redeem only a portion of the assets that were originally deposited by `GMigration` into the `GVault` contract. This occurs because the first depositor in the `GVault` is capable of manipulating the share price significantly, which is compounded by the fact that the `deposit` function in `GVault` rounds in favor of the protocol due to a division in `convertToShares` (see TOB-GRO-11).

Exploit Scenario
Alice, a GSquared developer, calls `prepareMigration` to begin the process of migrating funds from Gro to GSquared. Eve notices this transaction in the public mempool, and front-runs it with a small deposit and a large token (3CRV) airdrop. This leads to a significant change in the share price. The `prepareMigration` call completes, but `GMigration` is left with a small, insufficient amount of shares because it has suffered from truncation in the `convertToShares` function. These shares can be redeemed for only a portion of the original deposit.

Recommendations
Short term, perform the GSquared system deployment and protocol migration using a private relay. This will mitigate the risk of front-running the migration or price share manipulation.

Long term, implement the short- and long-term recommendations outlined in TOB-GRO-11. Additionally, implement an `ERC4626Router` similar to Fei protocol's implementation so that a minimum-amount-out can be specified for deposit, mint, redeem, and withdraw operations.

References
- ERC4626RouterBase.sol
- ERC4626 share price inflation

13. Incorrect slippage calculation performed during strategy investments and divestitures

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-GRO-13
Target: `strategy/ConvexStrategy.sol`	

Description

The incorrect arithmetic calculation for slippage tolerance during strategy investments and divestitures can lead to an increased rate of failed profit-and-loss (PnL) reports and withdrawals.

The `ConvexStrategy` contract is tasked with investing excess funds into a meta pool to obtain yield and divesting those funds from the pool whenever necessary. Investments are done via the `invest` function, and divestitures for a given amount are done via the `divest` function. Both functions have the ability to manage the amount of slippage that is allowed during the deposit and withdrawal from the meta pool. For example, in the `divest` function, the withdrawal will go through only if the amount of 3CRV tokens that *will be* transferred out from the pool (by burning meta pool tokens) is greater than or equal to the `_debt`, the amount of 3CRV that *needs to be* transferred out from the pool, discounted by `baseSlippage` (figure 13.1). Thus, both sides of the comparison must have units of 3CRV.

```
function divest(uint256 _debt, bool _slippage) internal returns (uint256) {
   uint256 meta_amount = ICurveMeta(metaPool).calc_token_amount(
       [0, _debt],
       false
   );
   if (_slippage) {
       uint256 ratio = curveValue();
       if (
           (meta_amount * PERCENTAGE_DECIMAL_FACTOR) / ratio <
           ((_debt * (PERCENTAGE_DECIMAL_FACTOR - baseSlippage)) /
               PERCENTAGE_DECIMAL_FACTOR)
       ) {
           revert StrategyErrors.LTMinAmountExpected();
       }
   }
   Rewards(rewardContract).withdrawAndUnwrap(meta_amount, false);
   return
       ICurveMeta(metaPool).remove_liquidity_one_coin(
           meta_amount,
           CRV3_INDEX,
           0
904            );
905     }
```

Figure 13.1: The `divest` function in `ConvexStrategy.sol#L883-905`

To calculate the value of a meta pool token (mpLP) in terms of 3CRV, the `curveValue` function is called (figure 13.2). The units of the return value, `ratio`, are 3CRV/mpLP.

```
1170    function curveValue() internal view returns (uint256) {
1171        uint256 three_pool_vp = ICurve3Pool(CRV_3POOL).get_virtual_price();
1172        uint256 meta_pool_vp = ICurve3Pool(metaPool).get_virtual_price();
1173        return (meta_pool_vp * PERCENTAGE_DECIMAL_FACTOR) / three_pool_vp;
1174    }
```

Figure 13.2: The `curveValue` function in `ConvexStrategy.sol#L1170-1174`

However, note that in figure 13.1, `meta_amount` value, which is the amount of mpLP tokens that need to be burned, is divided by `ratio`. From a unit perspective, this is multiplying an mpLP amount by a mpLP/3CRV ratio. The resultant units are not 3CRV. Instead, the arithmetic should be `meta_amount` *multiplied* by ratio. This would be mpLP times 3CRV/mpLP, which would result in the final units of 3CRV.

Assuming 3CRV/mpLP is greater than one, the division instead of multiplication will result in a smaller value, which increases the likelihood that the slippage tolerance is not met. The `invest` and `divest` functions are called during PnL reporting and withdrawals. If there is a higher risk for the functions to revert because the slippage tolerance is not met, the likelihood of failed PnL reports and withdrawals also increases.

Exploit Scenario

Alice wishes to withdraw some funds from the GSquared protocol. She calls `GRouter.withdraw` and with a reasonable `minAmount`. The `GVault` contract calls the `ConvexStrategy` contract to withdraw some funds to meet the necessary withdrawal amount. The strategy attempts to divest the necessary amount of funds. However, due to the incorrect slippage arithmetic, the `divest` function reverts and Alice's withdrawal is unsuccessful.

Recommendations

Short term, in `divest`, multiply `meta_amount` by `ratio`. In `invest`, multiply `amount` by `ratio`.

Long term, expand the unit test suite to cover additional edge cases and to ensure that the system behaves as expected.

14. Potential division by zero in _calcTrancheValue

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-GRO-14
Target: `GTranche.sol`	

Description

Junior tranche withdrawals may fail due to an unexpected division by zero error.

One of the key steps performed during junior tranche withdrawals is to identify the dollar value of the tranche tokens that will be burned by calling `_calcTrancheValue` (figure 14.1).

```
function _calcTrancheValue(
   bool _tranche,
   uint256 _amount,
   uint256 _total
) public view returns (uint256) {
   uint256 factor = getTrancheToken(_tranche).factor(_total);
   uint256 amount = (_amount * DEFAULT_FACTOR) / factor;
   if (amount > _total) return _total;
   return amount;
}
```

Figure 14.1: The `_calcTrancheValue` function in `GTranche.sol#L559-568`

To calculate the dollar value, the `factor` function is called to identify how many tokens represent one dollar. The dollar value, `amount`, is then the token amount provided, `_amount`, divided by `factor`.

However, an edge case in the `factor` function will occur if the total supply of tranche tokens (junior or senior) is non-zero while the amount of assets backing those tokens is zero. Practically, this can happen only if the system is exposed to a loss large enough that the assets backing the junior tranche tokens are completely wiped. In this edge case, the factor function returns zero (figure 14.2). The subsequent division by zero in `_calcTrancheValue` will cause the transaction to revert.

```
function factor(uint256 _totalAssets)
   public
   view
   override
   returns (uint256)
```

```
    {
        if (totalSupplyBase() == 0) {
            return getInitialBase();
        }

        if (_totalAssets > 0) {
            return totalSupplyBase().mul(BASE).div(_totalAssets);
        }

        // This case is totalSupply > 0 && totalAssets == 0, and only occurs on
system loss
        return 0;
    }
```

Figure 14.2: The `factor` function in `GToken.sol#L525-541`

It is important to note that if the system enters a state where there are no assets backing the junior tranche, junior tranche token holders would be unable to withdraw anyway. However, this division by zero should be caught in `_calcTrancheValue`, and the requisite error code should be thrown.

Recommendations

Short term, add a check before the division to ensure that `factor` is greater than zero. If `factor` is zero, throw a custom error code specifically created for this situation.

Long term, expand the unit test suite to cover additional edge cases and to ensure that the system behaves as expected.

15. Token withdrawals from GTranche are sent to the incorrect address

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-GRO-15
Target: `GTranche.sol`	

Description

The `GTranche` withdrawal function takes in a `_recipient` address to send the G3CRV shares to, but instead sends those shares to `msg.sender` (figure 15.1).

```
212    function withdraw(
213       uint256 _amount,
214       uint256 _index,
215       bool _tranche,
216       address _recipient
217    )
218       external
219       override
220       returns (uint256 yieldTokenAmounts, uint256 calcAmount)
221    {
 .        [...]
 .
245
246       trancheToken.burn(msg.sender, factor, calcAmount);
247       token.transfer(msg.sender, yieldTokenAmounts);
248
249       emit LogNewWithdrawal(
250           msg.sender,
251           _recipient,
252           _amount,
253           _index,
254           _tranche,
255           yieldTokenAmounts,
256           calcAmount
257       );
258       return (yieldTokenAmounts, calcAmount);
259    }
```

Figure 15.1: The `withdraw` function in `GTranche.sol#L219-259`

Since `GTranche` withdrawals are performed by the `GRouter` contract on behalf of the user, the `msg.sender` and `_recipient` address are the same. However, a direct call to `GTranche.withdraw` by a user could lead to unexpected consequences.

Recommendations

Short term, change the destination address to `_recipient` instead of `msg.sender`.

Long term, increase unit test coverage to include tests directly on `GTranche` and associated contracts in addition to performing the unit tests through the `GRouter` contract.

16. Solidity compiler optimizations can be problematic

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-GRO-16
Target: GSquared Protocol	

Description

The GSquared Protocol contracts have enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. 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.

Security issues due to optimization bugs have occurred in the past. A medium- to high-severity bug in the Yul optimizer was introduced in Solidity version 0.8.13 and was fixed only recently, in Solidity version 0.8.17. Another medium-severity optimization bug—one that caused memory writes in inline assembly blocks to be removed under certain conditions— was patched in Solidity 0.8.15.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario

A latent or future bug in Solidity compiler optimizations causes a security vulnerability in the GSquared Protocol contracts.

Recommendations

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.

Summary of Recommendations

The GSquared contracts under audit are a work in progress with multiple planned iterations. Trail of Bits recommends that Growth Labs address the findings detailed in this report and take the following additional steps prior to deployment:

- **Document the steps involved in the migration process**, and **write tests to ensure that all relevant funds are successfully transferred** while taking adequate measures to mitigate the risks of front-running the migration and price share manipulation.
- **Ensure that the bug fixes from previous audits are maintained and write related tests**. Some issues found during this audit had already been highlighted in previous audit reports, but the fixes were overwritten during code migration.
- **Shorten the system's inheritance tree** by removing unnecessary and/or duplicate interfaces and using abstract contracts only where necessary.
- **Document all potential front-running attack vectors** and ensure that the necessary mitigations are in place.
- **Integrate dynamic fuzz testing** to validate critical arithmetic operations and identify potential edge cases.
- **Develop a detailed incident response plan** to ensure that any issues that arise can be addressed promptly and without confusion.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

GVault

- **By default, feeCollector address is 0x0, which is error-prone.** Consider using a parameter to specify this address during deployment, checking that it is non-zero. This will avoid burning fees minting them to 0x0 if `setVaultFee` is increased without calling `setFeeCollector`.

- **Integer division before multiplication might truncate, causing loss of precision.** Consider ordering multiplication before division. In general, it is usually a good idea to rearrange arithmetic operations to perform multiplication before division, unless the limit of a smaller type makes this dangerous.

ConvexStrategy

- **Create a separate utilization threshold variable for the `distributeProfit` function**. The `distributeProfit` function uses a utilization threshold parameter that is not `utilisationThreshold`; currently, `DEFAULT_DECIMALS` (which is `10_000`) is used, but the rationale behind this choice is not well documented. Creating a separate state variable for `distributeProfit` will help with ease of understanding.

PnLFixedRate

- **Create a separate slippage variable for the `divestAll` function**. The `divestAll` function uses a separate slippage parameter that is not `baseSlippage`; this is currently hardcoded but should be isolated into a `constant` state variable.

General

- **Update inline documentation to highlight current code functionality.** For example, the `minAmount` value in `GRouter.withdraw` and `GRouter.deposit` should be a dollar value; however, the documentation specifies it as a token amount.

- **Maintain a consistent style across the codebase**. The Solidity style guide provides a variety of tips on coding conventions while writing Solidity code.

- **Use `safeTransfer` for all tokens that are not maintained by the GSquared protocol.** For example, the `ConvexStrategy.sweep` function interacts with external tokens. Using `safeTransfer` will prevent any unexpected behavior from non-compliant ERC20 tokens.

- **Remove `SafeMath` calls from `GToken` and `SeniorTranche`.** The codebase uses a version of Solidity with built-in overflow checks, so this library is no longer necessary outside of `unchecked` blocks or when passing custom error messages.

D. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Each unchecked item should be justified, and its associated risks understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the below output from Slither for the token:

```
slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
slither [target] --print human-summary
slither [target] --print contract-summary
slither-prop . --contract ContractName # requires configuration, and use of Echidna
and Manticore
```

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.

- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.

- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.

- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.

- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❑ **The token has only 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]` may not reflect the actual balance).

Owner Privileges

- ❑ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.

- ❑ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.

- ❑ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.

- ❑ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.

- ❑ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC20 Tokens

ERC20 Conformity Checks

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❑ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.

- ❑ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.

- ❑ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.

- ❑ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Risks of ERC20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.

- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.

- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.

- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.

- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.

- ❏ **The 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.

E. Recommendations on Fuzz Testing with Echidna

This appendix outlines a list of recommendations for creating a robust and mature suite of Echidna fuzz tests. We recommend that the Growth Labs team first review the documentation, complete the exercises provided for Echidna in the `building-secure-contracts` repository, and review Echidna's configuration options and their functionalities.

- **Use an external testing approach.** An external testing approach involves testing system properties by making external calls to a different contract. The only accessible functions / state variables in the external contract will be those marked as `public` / `external`. This testing approach is useful for systems that have complex initialization flows.

  Figure E.1 shows an example of an external testing approach. The `EchidnaContract.testGRouterDeposit` function calls into the `GRouter.deposit` function. The call enables Echidna to compare the state of the system after the call to `deposit` to the state before the call to identify whether any system properties were violated. Note that in the context of these tests, `msg.sender` will be the `EchidnaContract` and not a `sender`. A subsequent recommendation in this appendix explains how to emulate multiple senders while still using a single `EchidnaContract`.

```
contract EchidnaContract {
    constructor() public {
       gRouterContract = ..;
    }
    function testGRouterDeposit(uint256 amount, ...) public returns (bool) {
       // Precondition checks
       // Action: Call GRouter.deposit()
       try gRouterContract.deposit(amount, ...){
           // Postcondition checks (happy path)
       } catch (bytes memory) {
           // Postcondition checks (not-so-happy path)
       };
    }
}
```

Figure E.1: An example contract for external testing of the `GRouter.deposit` function

- **Use assertion testing to validate more complex and intricate system invariants.** Using Echidna in assertion testing mode makes it possible to test more complex system invariants. Although property-based testing is faster and easier to use than assertion testing, the functions being tested cannot take explicit input parameters, and no coverage results are provided at the end of the testing cycle. Assertion testing solves both of those problems in addition to identifying system invariants that could be broken in the middle of a transaction. Assertion testing can be enabled by setting the `testMode` configuration variable to `assertion`. In general, we recommend using the following workflow when testing a system property through assertion testing:

  - **Precondition checks:** These checks act as barriers to entry for the fuzzer and effectively require that the system meet certain conditions before performing the "action." If the system does not pass these checks, the property should not be tested. In the context of `GRouter.deposit`, the preconditions could include a requirement that the sender own some DAI, USDC, or USDT tokens to mint the amount of tokens specified by `_amount`.
  - **Execution of the action:** The action is the code or function call that is run in order to test the property. In the above example, calling `GRouter.deposit()` is the action.
  - **Postcondition checks:** These checks ensure that the state of the system after the execution of the action maintains the system property. An example postcondition check would be ensuring that the sender's `GToken` balance has increased by `amount` (the return value of `deposit`). The correctness of the testing is directly related to the efficacy of the postcondition checks in validating that property.

- **Use mock oracles and strategies.** Ideally, a fuzz testing environment should mimic the code that will actually be deployed. However, in many cases, this may not be feasible or may add unnecessary complexity to the fuzzing campaign. For example, it would be sufficient to create a `MockCurveOracle` that inherits the `Relation` abstract contract, which simply returns prices within certain ranges that allow both happy and "not-so-happy" paths to be tested. Note that function overriding to create custom behavior is a powerful technique for changing the functionality of code without affecting the rest of the system. The team could take a similar approach to simulating strategies. Creating a simpler `MockStrategy` contract that is used to test the core reporting functionalities of the `GVault` would decrease the overall complexity of the fuzzing campaign.

- **Use mintable ERC20 contracts.** Creating mock ERC20 contracts that have public `mint` functions facilitates complex testing. For example, a `MockUSDC` contract with a public `mint` function would allow a sender to mint enough stablecoin tokens to pass the example precondition check for `GRouter.deposit` mentioned above. Using a `MockUSDC` contract with only an internal `_mint` function would significantly limit the variety of system properties that could be tested through an external testing mode.

- **Use the `Account` contract proxy pattern to emulate multiple system actors.** Figure E.2 shows an example of an `Account` contract. An `Account` contract can hold tokens / ether and perform function calls. Thus, an `Account` contract is *functionally* equivalent to an externally owned account.

The contract shown in figure E.2 contains a `proxy` function through which arbitrary function calls can be made. In the `GRouter.deposit` example, the `target` argument of `Account.proxy` would be the address of the `GRouter` contract, and `_calldata` would be the encoded calldata passed to `GRouter.deposit`. Note, though, that the `msg.sender` of a call to `proxy` would be the `Account` contract, not the main Echidna contract. Creating multiple `Account` contracts would enable the team to emulate multiple senders.

```
contract Account {
    /* An account used to interact with underlying contracts
    account = new Account()
    */
    function proxy(address target, bytes memory _calldata)
        public
        returns (bytes memory)
    {
        (bool success, bytes memory returnData) = address(target).call(
            _calldata
        );
        require(success);
        return returnData;
    }
}
```

Figure E.2: The `Account` contract can be used to emulate multiple senders.

- **Use `try-catch` closures to test system properties**. A common mistake in the execution of fuzz testing is testing only the happy execution paths. However, testing unhappy paths is just as important for debugging and completeness of a testing effort as testing happy paths. Implementing `try-catch` closures allows the tester to specify different postconditions based on the success or failure of a call. Additionally, `try-catch` closures can be useful in identifying errors in fuzz test code, such as unexpected overflows.

- **Use stateful testing.** Echidna can perform both stateful and stateless testing. In stateful testing, the fuzzer is capable of threading together multiple transactions before wiping the state of the EVM. (The `seqLen` configuration variable specifies the number of transactions executed in the sequence.) This makes stateful testing more powerful, as it facilitates testing of more complex system properties.

  For example, say that the team has created a fuzz test called `testGRouterDeposit` that tests the `GRouter.deposit` function. Then the team writes a new fuzz test for the `GRouter.withdraw` function, `testGRouterWithdraw`, which requires an existing balance of tranche tokens as a precondition. Through stateful testing, the fuzzer can call `testGRouterDeposit` and `testGRouterWithdraw` sequentially. Since a call to `testGRouterDeposit` will create a balance of tranche tokens for the sender, the precondition check in `testGRouterWithdraw` will pass. (Note that an explicit call to `GRouter.withdraw` does *not* have to be made in `testGRouterWithdraw`.) Writing strong precondition checks and leveraging stateful testing makes it possible to test a large variety of code paths and sequences without having to explicitly call a large number of functions.

- **Create a separate `Setup` contract that is inherited by the main Echidna contract.** The `Setup` contract should be responsible for deploying the system and maintaining state variables. The main Echidna contract should inherit from the `Setup` contract and hold all of the fuzz tests. With this modular design, fuzz testing can be performed in one place, and the general state can be maintained and setup performed in another. For very large deployments, it may be necessary to increase the value of the `codeSize` configuration variable. Additionally, if the initialization is too complex for a constructor, Etheno can be used to help set up the environment.

- **Use the coverage file for debugging.** The coverage file provided by Echidna is essential to the debugging process. When a team is building a fuzz test, the coverage file can help it determine whether the fuzz test is performing as expected and whether there are requirements or conditions that Echidna will be unable to pass. A common mistake is assuming that a fuzz test is running properly even if Echidna is unable to pass a specific precondition. However, if Echidna cannot pass a precondition required to test a system property, it cannot actually check the system property. The coverage file also aids in determining whether any preconditions required to test a system property are missing. To enable coverage, set the `corpusDir` configuration variable. For more information about Echidna's coverage capabilities, consult its README.

- **Use the `between` function to bound values between a lower and upper limit.** Figure E.3 shows an example of the `between` function, which we recommend using whenever a variable should be bounded. The function is more powerful than a `require` statement because a `require` statement that fails will cause the transaction to revert. The use of `between` prevents such a revert and optimizes the computation. For example, a `require` statement like `require(x < 10)`, where x is of type `uint256`, would almost always fail. If x were bound in the range [0, 9], each transaction would be able to pass the `require(x < 10)` statement, allowing the fuzzer to continue its execution.

```
function between(
    uint256 val,
    uint256 lower,
    uint256 upper
) internal pure returns (uint256) {
    return lower + (val % (upper - lower + 1));
}
```

Figure E.3: The `between` function can be used to bound variables.

- **Add comprehensive event logging mechanisms to all fuzz tests to aid in debugging.** Logging events during smart contract fuzzing is crucial for understanding the state of the system when a system property is broken. Without logging, it is difficult to identify the arithmetic operation or computation that caused the failure. Insufficient logging may also increase the rate of false positives. In general, any action that changes a value should trigger an event.

- **Enrich each fuzz test with comments explaining the pre- and postconditions of the test.** Strong fuzz testing requires well-defined preconditions (for guiding the fuzzer) and postconditions (for properly testing the invariant(s) in question). Comments explaining the bounds on certain values and the importance of the system properties being tested aid in test suite maintenance and debugging efforts.

- **Integrate fuzz testing into the CI/CD workflow.** Continuous fuzz testing can help quickly identify any code changes that will result in a violation of a system property and forces developers to update the fuzz test suite in parallel with the code. Running fuzz campaigns stochastically may cause a divergence between the operations in the code and the fuzz tests. The duration of a fuzzer's execution is dictated by the `testLimit` configuration variable.