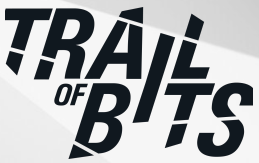

Sherlock Protocol V2

Security Assessment

December 17, 2021

Prepared for:
Jack Sanford
Sherlock

Prepared by:
Alexander Remie
Simone Monica

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 80+ 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

Classification and Copyright

This report is confidential and intended for the sole internal use of Sherlock.

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 mutually 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 partners. As such, 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.

Table of Contents

Executive Summary

Overview

Sherlock engaged Trail of Bits to review the security of its smart contracts. From December 6 to December 17, 2021, a team of two consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

We focused our testing efforts on the identification of flaws that could result in a compromise or lapse of confidentiality, integrity, or availability of the target system. We performed automated testing and a manual review of the code, in addition to running system elements.

Summary of Findings

Our review resulted in two high-severity, two medium-severity, two low-severity, two informational-severity, and one undetermined-severity issues.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	2
Low	2
Informational	2
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Undefined Behavior	6
Data Validation	2
Access Control	1

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan.guido@trailofbits.com

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineers were associated with this project:

Alexander Remie, Consultant
alexander.remie@trailofbits.com

Simone Monica, Consultant
simone.monica@trailofbits.com

Project Timeline

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

Date	Event
December 2, 2021	Project pre-kickoff call
December 10, 2021	Status update meeting #1
December 17, 2021	Delivery of report draft
December 17, 2021	Report readout meeting
January 5, 2022	Fix Log added (Appendix H)

Project Goals

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

- Are the SHER reward calculations sound?
- Is the system vulnerable to reentrancy attacks?
- Could UMA subvert the claims process in any way?
- Does the system's pausing/unpausing mechanism work correctly?
- Does the protocol agent have excessive powers?
- Does updating the protocol agent cause problems?
- Does the use of previous rather than current coverage cause any problems?
- Is the system correctly integrated with the UMA contract?
- Could the callback mechanism be used to cause a denial of service in the claim payout process?
- Are there incentives to restake instead of redeeming an existing stake and receiving a fresh stake?
- Are the transitions between all the states correct?
- Is it possible for stakers to lose their share of SHER rewards?
- Could funds be swept from an active contract?
- Is the system correctly integrated with Aave V2?
- Do all functions have appropriate input validation?
- Are protocols added, updated, and removed correctly?
- Does the contract owner have excessive powers?
- Could an attacker use the arbitration functions to steal funds?
- Are there bugs in the internal accounting process of stake shares?
- Does the system correctly handle ERC721 tokens?

Project Targets

The engagement involved a review and testing of the targets listed below.

Sherlock Protocol

Repository	https://github.com/sherlock-protocol/sherlock-v2-core
Version	`877056fd1d30aa2d74db7f673ee289ddc75e449f`
Type	Solidity
Platform	Ethereum

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:

- **`Sherlock`**. Stakers interact with the Sherlock protocol through this contract to create new stakes and to restake or redeem existing stakes. Furthermore, arbitrageurs can restake on behalf of a staker through this contract if a stake has expired. Lastly, the owner can use update functions to replace any of the connected manager contracts. We used static analysis, a manual review, and Echidna to test the contract's staking behavior, input validation, access controls, pausing/unpausing mechanism, reentrancy prevention, and SHER reward payment process.
- **`SherlockClaimManager`**. This contract is used by protocol agents and the Sherlock Protocol Claims Committee (SPCC) to handle the claims process. Furthermore, this contract integrates with UMA when escalating a claim. We used static analysis and a manual review to test the contract's access controls, input validation, and pausing/unpausing mechanism; we also tested that the claims process cannot be used to steal funds, that the claim payout process cannot be blocked, that the UMA integration is implemented correctly, that claim state transitions are correct, and that updating a protocol agent does not prevent existing claims from proceeding.
- **`SherlockProtocolManager`**. This contract is used to manage the protocols that are currently insured through the Sherlock protocol. The contract owner can create, update, and remove protocols and update various global and per-protocol configuration values. Furthermore, any account can force the removal of a protocol whose active balance is insufficient. We used static analysis and a manual review to test the contract's access controls, input validation, protocol management, internal accounting of balances and debts, and process for removing protocols due to insufficient balance.
- **`SherDistributionManager`**. This contract is used to calculate and pay out SHER rewards according to a given stake's USDC amount and period. We used static analysis and a manual review to test the contract's access controls, input validation, and calculation of the SHER reward.
- **`AaveV2Strategy`**. This contract contains the logic to deposit USDC into and to withdraw USDC from Aave and receive aTokens in exchange. We used static analysis

and a manual review to test the contract's access controls, input validation, and integration with Aave.

- **`Manager`**. This is a base contract for the three manager contracts and the `AaveV2Strategy` contract. It contains functions to set the `Sherlock` contract's address, to pause and unpause the contract, and to sweep funds (ETH and ERC20) to a chosen receiver address. We used static analysis and a manual review to test the contract's access controls, input validation, and process for correctly transferring ETH and ERC20 funds.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- The process of settling total debt when arbitration is not executed in time
- The system's front-running resistance to the various arbitrage opportunities in the Sherlock protocol

Automated Testing Results

Trail of Bits has developed three unique tools for testing smart contracts. Descriptions of these tools and details on the use of tools in this project are provided below.

- Slither is a static analysis framework that can statically verify algebraic relationships between Solidity variables. We used Slither to check for common vulnerabilities. We also created two custom scripts that check for missing pause modifiers and the use of require statements, described in Appendix E.
- Echidna is a smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation. We used Echidna to test that initializing a new stake is not more profitable than restaking, described in Appendix F.
- Manticore is a symbolic execution framework that can exhaustively test security properties. We did not use Manticore during this assessment.

Automated testing techniques augment our manual security review but do not replace it. Each technique has limitations: Slither may identify security properties that fail to hold when Solidity is compiled to EVM bytecode, Echidna may not randomly generate an edge case that violates a property, and Manticore may fail to complete its analysis.

We follow a consistent process to maximize the efficacy of testing security properties. When using Echidna, we generate 10,000 test cases per property; when testing with Manticore, we run the tool for a minimum of one hour. In both cases, we then manually review all results.

Our automated testing and verification focused on the following system properties:

Property	Approach	Result
Redeeming an existing stake and initializing a new stake is not more profitable than restaking.	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 use of `solc` 0.8.10 prevents underflows and overflows. No `unchecked` blocks are used. Additional automated analysis could be used to provide more assurance of correct arithmetic.	**Satisfactory**
Auditing	The system correctly emits events for important system actions, allowing the contracts' behavior to be monitored. Nonetheless, there is no plan to use off-chain monitoring to detect incorrect behavior that requires manual intervention.	**Satisfactory**
Authentication / Access Controls	The system correctly applies access controls to all functions. However, documentation that clearly defines the powers of each privileged role is not present.	**Moderate**
Complexity Management	The system is divided into several contracts that each deal with a particular facet of the system. Functions are small and perform a single specific task.	**Satisfactory**
Cryptography and Key Management	This was not part of the scope of this assessment.	**Not Considered**

Decentralization	The system uses a per-contract owner role that can perform privileged actions. The Sherlock team plans to transfer the owner role to a DAO that will progressively decentralize the protocol's governance. We recommend adding documentation to explain this plan to the users of Sherlock.	**Moderate**
Documentation	The codebase contains extensive inline and function-level comments that clearly describe the system's behavior. However, we found multiple incorrect comments, and the user-facing documentation is still a work in progress.	**Moderate**
Front-Running Resistance	We did not find any front-running issues; however, there is a possible front-running between arbitrageurs to remove a protocol. We recommend investigating this possibility.	**Further Investigation Required**
Low-Level Calls	The system does not use assembly and contains only a single low-level call to transfer ETH.	**Strong**
Testing and Verification	The test suite is extensive and uses mainnet hardforking. However, we uncovered multiple issues that reveal that the test suite does not cover enough cases (e.g. TOB-SHER-004, TOB-SHER-006). Furthermore, we recommend using property-based testing to test that important system invariants hold.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Solidity compiler optimizations can be problematic	Undefined Behavior	Undetermined
2	Certain functions lack zero address checks	Data Validation	Medium
3	updateYieldStrategy could leave funds in the old strategy	Undefined Behavior	High
4	Pausing and unpausing the system may not be possible when removing or replacing connected contracts	Undefined Behavior	Low
5	SHER reward calculation uses confusing six-decimal SHER reward rate	Undefined Behavior	Informational
6	A claim cannot be paid out or escalated if the protocol agent changes after the claim has been initialized	Access Controls	Medium
7	Missing input validation in setMinActiveBalance could cause a confusing event to be emitted	Data Validation	Informational
8	payoutClaim's calling of external contracts in a loop could cause a denial of service	Undefined Behavior	Low
9	pullReward could silently fail and cause stakers to lose all earned SHER rewards	Undefined Behavior	High

Detailed Findings

1. Solidity compiler optimizations can be problematic	
Severity: **Undetermined**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-SHER-001
Target: `hardhat.config.js`	

Description
Sherlock has 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.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

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—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the Sherlock 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.

2. Certain functions lack zero address checks

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SHER-002
Target: `contracts/managers/(AaveV2Strategy|SherlockProtocolManager|SherDistributionManager|Manager).sol`, `contracts/Sherlock.sol`	

Description

Certain functions fail to validate incoming arguments, so callers can accidentally set important state variables to the zero address.

For example, the `AaveV2Strategy` contract's `constructor` function does not validate the `aaveLmReceiver`, which is the address that receives Aave rewards on calls to `AaveV2Strategy.claimRewards`.

```
39    constructor(IAToken _aWant, address _aaveLmReceiver) {
40      aWant = _aWant;
41      // This gets the underlying token associated with aUSDC (USDC)
42      want = IERC20(_aWant.UNDERLYING_ASSET_ADDRESS());
43      // Gets the specific rewards controller for this token type
44      aaveIncentivesController = _aWant.getIncentivesController();
45
46      aaveLmReceiver = _aaveLmReceiver;
47    }
```

Figure 2.1: `managers/AaveV2Strategy.sol:39-47`

If the `aaveLmReceiver` variable is set to the address zero, the Aave contract will revert with `INVALID_TO_ADDRESS`. This prevents any Aave rewards from being claimed for the designated token.

The following functions are missing zero address checks:

- `Manager.setSherlockCoreAddress`
- `AaveV2Strategy.sweep`
- `SherDistributionManager.sweep`
- `SherlockProtocolManager.sweep`
- `Sherlock.constructor`

Exploit Scenario
Bob deploys `AaveV2Strategy` with `aaveLmReceiver` set to the zero address. All calls to `claimRewards` revert.

Recommendations
Short term, add zero address checks on all function arguments to ensure that users cannot accidentally set incorrect values.

Long term, use Slither, which will catch functions that do not have zero address checks.

3. updateYieldStrategy could leave funds in the old strategy

Severity: **High**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-SHER-003
Target: `contracts/Sherlock.sol`	

Description

The `updateYieldStrategy` function sets a new yield strategy manager contract without calling `yieldStrategy.withdrawAll()` on the old strategy, potentially leaving funds in it.

```
257  // Sets a new yield strategy manager contract
258  /// @notice Update yield strategy
259  /// @param _yieldStrategy News address of the strategy
260  /// @dev try a yieldStrategyWithdrawAll() on old, ignore failure
261  function updateYieldStrategy(IStrategyManager _yieldStrategy) external override
onlyOwner {
262    if (address(_yieldStrategy) == address(0)) revert ZeroArgument();
263    if (yieldStrategy == _yieldStrategy) revert InvalidArgument();
264
265    emit YieldStrategyUpdated(yieldStrategy, _yieldStrategy);
266    yieldStrategy = _yieldStrategy;
267  }
```

Figure 3.1: `contracts/Sherlock.sol:257-267`

Even though one could re-add the old strategy to recover the funds, this issue could cause stakers and the protocols insured by Sherlock to lose trust in the system.

This issue has a significant impact on the result of `totalTokenBalanceStakers`, which is used when calculating the shares in `initialStake`. `totalTokenBalanceStakers` uses the balance of the yield strategy. If the balance is missing the funds that should have been withdrawn from a previous strategy, the result will be incorrect.

```
151  function totalTokenBalanceStakers() public view override returns (uint256) {
152    return
153      token.balanceOf(address(this)) +
154      yieldStrategy.balanceOf() +
155      sherlockProtocolManager.claimablePremiums();
156  }
```

Figure 3.2: contracts/Sherlock.sol:151-156

```
483  function initialStake(
484    uint256 _amount,
485    uint256 _period,
486    address _receiver
487  ) external override whenNotPaused returns (uint256 _id, uint256 _sher) {
...
501   if (totalStakeShares_ != 0)
502      stakeShares_ = (_amount * totalStakeShares_) / (totalTokenBalanceStakers() -
_amount);
503      // If this is the first stake ever, we just mint stake shares equal to the amount
of USDC staked
504    else stakeShares_ = _amount;
```

Figure 3.3: contracts/Sherlock.sol:483-504

Exploit Scenario

Bob, the owner of the `Sherlock` contract, calls `updateYieldStrategy` with a new strategy. Eve calls `initialStake` and receives more shares than she is due because `totalTokenBalanceStakers` returns a significantly lower balance than it should. Bob notices the missing funds, calls `updateYieldStrategy` with the old strategy and then `yieldStrategy.WithdrawAll` to recover the funds, and switches back to the new strategy. Eve's shares now have notably more value.

Recommendations

Short term, in `updateYieldStrategy`, add a call to `yieldStrategy.withdrawAll()` on the old strategy.

Long term, when designing systems that store funds, use extensive unit testing and property-based testing to ensure that funds cannot become stuck.

4. Pausing and unpausing the system may not be possible when removing or replacing connected contracts

Severity: **Low**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-SHER-004
Target: `contracts/Sherlock.sol`	

Description

The `Sherlock` contract allows all of the connected contracts to be paused or unpaused at the same time. However, if the `sherDistributionManager` contract is removed, or if any of the connected contracts are replaced when the system is paused, it might not be possible to pause or unpause the system.

```
206   function removeSherDistributionManager() external override onlyOwner {
207     if (address(sherDistributionManager) == address(0)) revert InvalidConditions();
208
209     emit SherDistributionManagerUpdated(
210       sherDistributionManager,
211       ISherDistributionManager(address(0))
212     );
213     delete sherDistributionManager;
214   }
```

Figure 4.1: `contracts/Sherlock.sol:206-214`

Of all the connected contracts, the only one that can be removed is the `sherDistributionManager` contract. On the other hand, all of the connected contracts can be replaced through an update function.

```
302   function pause() external onlyOwner {
303     _pause();
304     yieldStrategy.pause();
305     sherDistributionManager.pause();
306     sherlockProtocolManager.pause();
307     sherlockClaimManager.pause();
308   }
309
310   /// @notice Unpause external functions in all contracts
311   function unpause() external onlyOwner {
```

```
312       _unpause();
313       yieldStrategy.unpause();
314       sherDistributionManager.unpause();
315       sherlockProtocolManager.unpause();
316       sherlockClaimManager.unpause();
317     }
```

Figure 4.2: contracts/Sherlock.sol:302-317

If the `sherDistributionManager` contract is removed, a call to `Sherlock.pause` will revert, as it is attempting to call the zero address. If `sherDistributionManager` is removed while the system is paused, then a call to `Sherlock.unpause` will revert for the same reason.

If any of the contracts is replaced while the system is paused, the replaced contract will be in an unpaused state while the other contracts are still paused. As a result, a call to `Sherlock.unpause` will revert, as it is attempting to `unpause` an already unpaused contract.

Exploit Scenario

Bob, the owner of the `Sherlock` contract, pauses the system to replace the `sherlockProtocolManager` contract, which contains a bug. Bob deploys a new `sherlockProtocolManager` contract and calls `updateSherlockProtocolManager` to set the new address in the `Sherlock` contract. To unpause the system, Bob calls `Sherlock.unpause`, which reverts because `sherlockProtocolManager` is already unpaused.

Recommendations

Short term, add conditional checks to the `Sherlock.pause` and `Sherlock.unpause` functions to check that a contract is either paused or unpaused, as expected, before attempting to update its state. For `sherDistributionManager`, the check should verify that the contract to be paused or unpaused is not the zero address.

Long term, for pieces of code that depend on the states of multiple contracts, implement unit tests that cover each possible combination of contract states.

5. SHER reward calculation uses confusing six-decimal SHER reward rate

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-SHER-005
Target: `contracts/managers/SherDistributionManager.sol`	

Description

The reward calculation in `calcReward` uses a six-decimal SHER reward rate value. This might confuse readers and developers of the contracts because the SHER token has 18 decimals, and the calculated reward will also have 18 decimals. Also, this value does not allow the SHER reward rate to be set below 0.000001000000000000 SHER.

```
 89    function calcReward(
 90      uint256 _tvl,
 91      uint256 _amount,
 92      uint256 _period
 93    ) public view override returns (uint256 _sher) {
         [..]
109      // If there are some max rewards available...
110      if (maxRewardsAvailable != 0) {
111        // And if the entire stake is still within the maxRewardsAvailable amount
112        if (_amount <= maxRewardsAvailable) {
113          // Then the entire stake amount should accrue max SHER rewards
114          return (_amount * maxRewardsRate * _period) * DECIMALS;
115        } else {
116          // Otherwise, the stake takes all the maxRewardsAvailable left ...
117          // We add the maxRewardsAvailable amount to the TVL (now _tvl...
118          _tvl += maxRewardsAvailable;
119          // We subtract the amount of the stake that received max rewards
120          _amount -= maxRewardsAvailable;
121
122          // We accrue the max rewards available at the max rewards ...
123          // This could be: $20M of maxRewardsAvailable which gets ...
124          // Calculation continues after this
125          _sher += (maxRewardsAvailable * maxRewardsRate * _period) * DECIMALS;
126        }
127      }
128
129      // If there are SHER rewards still available ...
130      if (slopeRewardsAvailable != 0) {
```

```
144       _sher +=
145         (((zeroRewardsStartTVL - position) * _amount * maxRewardsRate * _period) /
146           (zeroRewardsStartTVL - maxRewardsEndTVL)) *
147         DECIMALS;
148     }
149    }
```

Figure 5.1: contracts/managers/SherDistributionManager.sol:89-149

In the reward calculation, the 6-decimal `maxRewardsRate` is first multiplied by `_amount` and `_period`, resulting in a 12-decimal intermediate product. To output a final 18-decimal product, this 12-decimal product is multiplied by `DECIMALS` to add 6 decimals. Although this leads to a correct result, it would be clearer to use an 18-decimal value for `maxRewardsRate` and to divide by `DECIMALS` at the end of the calculation.

```
// using 6 decimal maxRewardsRate
(10e6 *  1e6 * 10) * 1e6 = 100e18 = 100 SHER

// using 18 decimal maxRewardsRate
(10e6 *  1e18 * 10) / 1e6 = 100e18 = 100 SHER
```

Figure 5.2: Comparison of a 6-decimal and an 18-decimal `maxRewardsRate`

Exploit Scenario

Bob, a developer of the Sherlock protocol, writes a new version of the `SherDistributionManager` contract that changes the reward calculation. He mistakenly assumes that the SHER `maxRewardsRate` has 18 decimals and updates the calculation incorrectly. As a result, the newly calculated reward is incorrect.

Recommendations

Short term, use an 18-decimal value for `maxRewardsRate` and divide by `DECIMALS` instead of multiplying.

Long term, when implementing calculations that use the rate of a given token, strive to use a rate variable with the same number of decimals as the token. This will prevent any confusion with regard to decimals, which might lead to introducing precision bugs when updating the contracts.

6. A claim cannot be paid out or escalated if the protocol agent changes after the claim has been initialized

Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-SHER-006
Target: `contracts/managers/SherlockClaimManager.sol`	

Description

The `escalate` and `payoutClaim` functions can be called only by the protocol agent that started the claim. Therefore, if the protocol agent role is reassigned after a claim is started, the new protocol agent will be unable to call these functions and complete the claim.

```
388    function escalate(uint256 _claimID, uint256 _amount)
389     external
390     override
391     nonReentrant
392     whenNotPaused
393    {
394     if (_amount < BOND) revert InvalidArgument();
395
396     // Gets the internal ID of the claim
397     bytes32 claimIdentifier = publicToInternalID[_claimID];
398     if (claimIdentifier == bytes32(0)) revert InvalidArgument();
399
400     // Retrieves the claim struct
401     Claim storage claim = claims_[claimIdentifier];
402     // Requires the caller to be the protocol agent
403     if (msg.sender != claim.initiator) revert InvalidSender();
```

Figure 6.1: `contracts/managers/SherlockClaimManager.sol:388-403`

Due to this scheme, care should be taken when updating the protocol agent. That is, the protocol agent should not be reassigned if there is an existing claim. However, if the protocol agent is changed when there is an existing claim, the protocol agent role could be transferred back to the original protocol agent to complete the claim.

Exploit Scenario

Alice is the protocol agent and starts a claim. Alice transfers the protocol agent role to Bob. The claim is approved by SPCC and can be paid out. Bob calls `payoutClaim`, but the transaction reverts.

Recommendations

Short term, update the comment in the `escalate` and `payoutClaim` functions to state that the caller needs to be the protocol agent that started the claim, and clearly describe this requirement in the protocol agent documentation. Alternatively, update the check to verify that `msg.sender` is the current protocol agent rather than specifically the protocol agent who initiated the claim.

Long term, review and document the effects of the reassignment of privileged roles on the system's state transitions. Such a review will help uncover cases in which the reassignment of privileged roles causes issues and possibly a denial of service to (part of) the system.

7. Missing input validation in setMinActiveBalance could cause a confusing event to be emitted

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SHER-007
Target: `contracts/managers/SherlockProtocolManager.sol`	

Description

The `setMinActiveBalance` function's input validation is incomplete: it should check that the `minActiveBalance` has not been set to its existing value, but this check is missing. Additionally, if the `minActiveBalance` is set to its existing value, the emitted `MinBalance` event will indicate that the old and new values are identical. This could confuse systems monitoring the contract that expect this event to be emitted only when the `minActiveBalance` changes.

```
422   function setMinActiveBalance(uint256 _minActiveBalance) external override onlyOwner {
423     // Can't set a value that is too high to be reasonable
424     require(_minActiveBalance < MIN_BALANCE_SANITY_CEILING, 'INSANE');
425
426     emit MinBalance(minActiveBalance, _minActiveBalance);
427     minActiveBalance = _minActiveBalance;
428   }
```

Figure 7.1: `contracts/managers/SherlockProtocolManager.sol:422-428`

Exploit Scenario

An off-chain monitoring system controlled by the Sherlock protocol is listening for events that indicate that a contract configuration value has changed. When such events are detected, the monitoring system sends an email to the admins of the Sherlock protocol. Alice, a contract owner, calls `setMinActiveBalance` with the existing `minActiveBalance` as input. The off-chain monitoring system detects the emitted event and notifies the Sherlock protocol admins. The Sherlock protocol admins are confused since the value did not change.

Recommendations

Short term, add input validation that causes `setMinActiveBalance` to revert if the proposed `minActiveBalance` value equals the current value.

Long term, document and test the expected behavior of all the system's events. Consider using a blockchain-monitoring system to track any suspicious behavior in the contracts.

8. payoutClaim's calling of external contracts in a loop could cause a denial of service

Severity: **Low**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-SHER-008
Target: `contracts/managers/SherlockClaimManager.sol`	

Description

The `payoutClaim` function uses a loop to call the `PreCorePayoutCallback` function on a list of external contracts. If any of these calls reverts, the entire `payoutClaim` function reverts, and, hence, the transaction reverts. This may not be the desired behavior; if that is the case, a denial of service would prevent claims from being paid out.

```
499    for (uint256 i; i < claimCallbacks.length; i++) {
500     claimCallbacks[i].PreCorePayoutCallback(protocol, _claimID, amount);
501    }
```

Figure 8.1: `contracts/managers/SherlockClaimManager.sol:499-501`

The owner of the `SherlockClaimManager` contract controls the list of contracts on which the `PreCorePayoutCallback` function is called. The owner can add or remove contracts from this list at any time. Therefore, if a contract is causing unexpected reverts, the owner can fix the problem by (temporarily) removing that contract from the list.

It might be expected that some of these calls revert and cause the entire transaction to revert. However, the external contracts that will be called and the expected behavior in the event of a revert are currently unknown. If a revert should not cause the entire transaction to revert, the current implementation does not fulfill that requirement.

To accommodate both cases—a revert of an external call reverts the entire transaction or allows the transaction to continue —a middle road can be taken. For each contract in the list, a boolean could indicate whether the transaction should revert or continue if the external call fails. If the boolean indicates that the transaction should continue, an emitted event would indicate the contract address and the input arguments of the callback that reverted. This would allow the system to continue functioning while admins investigate the cause of the revert and fix the issue(s) if needed.

Exploit Scenario

Alice, the owner of the `SherlockClaimManager` contract, registers contract A in the list of contracts on which `PreCorePayoutCallback` is called. Contract A contains a bug that causes the callback to revert every time. Bob, a protocol agent, successfully files a claim and calls `payoutClaim`. The transaction reverts because the call to contract A reverts.

Recommendations

Short term, review the requirements of contracts that will be called by callback functions, and adjust the implementation to fulfill those requirements.

Long term, when designing a system reliant on external components that have not yet been determined, carefully consider whether to include those integrations during the development process or to wait until those components have been identified. This will prevent unforeseen problems due to incomplete or incorrect integrations with unknown contracts.

9. pullReward could silently fail and cause stakers to lose all earned SHER rewards

Severity: **High**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-SHER-009
Target: `contracts/Sherlock.sol`	

Description

If the `SherDistributionManager.pullReward` function reverts, the calling function (`_stake`) will not set the SHER rewards in the staker's position NFT. As a result, the staker will not receive the payout of SHER rewards after the stake `period` has passed.

```
354    function _stake(
355     uint256 _amount,
356     uint256 _period,
357     uint256 _id,
358     address _receiver
359    ) internal returns (uint256 _sher) {
360     // Sets the timestamp at which this position can first be unstaked/restaked
361     lockupEnd_[_id] = block.timestamp + _period;
362
363     if (address(sherDistributionManager) == address(0)) return 0;
364     // Does not allow restaking of 0 tokens
365     if (_amount == 0) return 0;
366
367     // Checks this amount of SHER tokens in this contract before we transfer new ones
368     uint256 before = sher.balanceOf(address(this));
369
370     // pullReward() calcs then actually transfers the SHER tokens to this contract
371     try sherDistributionManager.pullReward(_amount, _period, _id, _receiver) returns (
372       uint256 amount
373     ) {
374       _sher = amount;
375     } catch (bytes memory reason) {
376       // If for whatever reason the sherDistributionManager call fails
377       emit SherRewardsError(reason);
378       return 0;
379     }
380
381     // actualAmount should represent the amount of SHER tokens transferred to this
```

```
contract for the current stake position
 382     uint256 actualAmount = sher.balanceOf(address(this)) - before;
 383     if (actualAmount != _sher) revert InvalidSherAmount(_sher, actualAmount);
 384     // Assigns the newly created SHER tokens to the current stake position
 385     sherRewards_[_id] = _sher;
 386    }
```

Figure 9.1: contracts/Sherlock.sol:354-386

When the `pullReward` call reverts, the `SherRewardsError` event is emitted. The staker could check this event and see that no SHER rewards were set. The staker could also call the `sherRewards` function and provide the position's NFT ID to check whether the SHER rewards were set. However, stakers should not be expected to make these checks after every (re)stake.

There are two ways in which the `pullReward` function can fail. First, a bug in the arithmetic could cause an overflow and revert the function. Second, if the `SherDistributionManager` contract does not hold enough SHER to be able to transfer the calculated amount, the `pullReward` function will fail. The SHER balance of the contract needs to be manually topped up.

If a staker detects that no SHER was set for her (re)stake, she may want to cancel the stake. However, stakers are not able to cancel a stake until the stake's period has passed (currently, at least three months).

Exploit Scenario

Alice creates a new stake, but the `SherDistributionManager` contract does not hold enough SHER to transfer the rewards, and the transaction reverts. The execution continues and sets Alice's stake allocation to zero.

Recommendations

Short term, have the system revert transactions if `pullReward` reverts.

Long term, have the system revert transactions if part of the expected rewards are not allocated due to an internal revert. This will prevent situations in which certain users get rewards while others do not.

Summary of Recommendations

The Sherlock protocol V2 smart contracts are a second iteration of the protocol, but they still require some work to be ready to be deployed and used in production. Trail of Bits recommends that the Sherlock protocol team address the findings detailed in this report and take the following additional steps prior to deployment:

- Improve the user-facing documentation, which is currently a work in progress with several missing sections. This will help the participants of the protocol fully understand the system and the implementation.

- Expand the unit test suite to cover cases such as the pausing or unpausing of the system while replacing or removing manager contracts (TOB-SHER-004) and the changing of the protocol agent during an ongoing claims process (TOB-SHER-006).

- Create important system invariants and test them using Echidna. This will both ensure that the test suite is robust and uncover edge cases that are not covered by the unit test suite.

- Consider the external contracts that will be called by callbacks and appropriately adjust the implementation to suit their requirements (TOB-SHER-008). Perform an additional review to ensure that changes in this area do not introduce new issues.

- Consider, document, and test the process of replacing the manager contracts without breaking the system or losing (access to) user funds.

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 commonly 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 presence of event auditing and logging to support monitoring
Authentication / Access Controls	The presence of robust controls to handle identification, authorization, and safe interactions with the system (e.g., rate-limiting)
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 to mitigate insider threats and manage risks posed to the system during contract upgrade
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Calls	The justified use of inline assembly and low-level calls within the system
Testing and Verification	The presence of robust testing procedures (e.g., specifications, fuzzing, and verification)

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed, and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.
Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

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.

General Suggestions

- **Consider adding the `indexed` attribute to certain events' arguments, such as the `protocol` argument.** This will make filtering easier for off-chain monitoring systems.

```
28 event AccountingError(bytes32 protocol, uint256 amount, uint256 insufficientTokens);
29
30 event ProtocolAdded(bytes32 protocol);
31
32 event ProtocolRemovedByArb(bytes32 protocol, address arb, uint256 profit);
...
```

Figure C.1: `contracts/interfaces/managers/ISherlockProtocolManager.sol:28-32`

- **Consider explicitly defining the storage variables' visibility.** This will improve code readability.

- **Replace `require(...)` with `if (condition) revert customError()`.** This will make the code style consistent throughout the codebase (refer to this Slither script).

```
424    require(_minActiveBalance < MIN_BALANCE_SANITY_CEILING, 'INSANE');
```

Figure C.2: `contracts/managers/SherlockProtocolManager.sol:424`

```
163    require(isActive() == false, 'is_active');
```

Figure C.3: `contracts/managers/SherlockDistributionManager.sol:163`

- **Replace `variable == false` with `!variable`.** This will improve code readability.

```
51    if (success == false) revert InvalidConditions();
```

Figure C.4: `contracts/managers/Manager.sol:51`

```
678    if (able == false) revert InvalidConditions();
```

Figure C.5: contracts/managers/SherlockProtocolManager.sol:678

```
283    if (_isCleanupState(_oldState) == false) revert InvalidState();
...
411    if (_isEscalateState(_oldState, updated) == false) revert InvalidState();
...
496    if (_isPayoutState(_oldState, updated) == false) revert InvalidState();
```

Figure C.6: contracts/managers/SherlockClaimManager.sol:283-496

ISherlockClaimManager.sol

- **Replace `previousCoverageAmount` with `previousCoverageUsed`.** This will improve code readability; because this is a boolean argument, the word "amount" is confusing.

```
22    event ClaimCreated(
23       uint256 claimID,
24       bytes32 protocol,
25       uint256 amount,
26       address receiver,
27       bool previousCoverageAmount
28     );
```

Figure C.7: contracts/interfaces/managers/ISherlockClaimManager.sol:22-28

- **Move `ancillaryData` to after `state`.** This will improve the gas usage, as the struct will use one fewer storage slot; the `state` field uses only one byte and will be packed into a single storage slot with `receiver` and `timestamp`.

```
57    struct Claim {
58      uint256 created;
59      uint256 updated;
60      address initiator;
61      bytes32 protocol;
62      uint256 amount;
63      address receiver;
64      uint32 timestamp;
65      bytes ancillaryData;
66      State state;
67  }
```

Figure C.8: contracts/interfaces/managers/ISherlockClaimManager.sol:57-67

Sherlock.sol

- **Normalize the names of the manager argument.** This will improve code readability. Consider using a standard naming convention such as `_distributionManager`, `_protocolManager`, and `_claimManager`.

```
192    function updateSherDistributionManager(ISherDistributionManager _manager)
...
230    function updateSherlockProtocolManager(ISherlockProtocolManager _protocolManager)
...
245    function updateSherlockClaimManager(ISherlockClaimManager _sherlockClaimManager)
```

Figure C.9: contracts/Sherlock.sol:192-245

- **Normalize the names of the storage variables.** This will improve code readability. Consider adding an underscore to or removing the underscore from all of them.

```
43    mapping(uint256 => bool) public override stakingPeriods;
44
45    // Key is a specific position ID (NFT ID), value represents the timestamp at which
the position can be unstaked/restaked
46    mapping(uint256 => uint256) internal lockupEnd_;
47
48    // Key is NFT ID, value is the amount of SHER rewards owed to that NFT position
49    mapping(uint256 => uint256) internal sherRewards_;
50
51    // Key is NFT ID, value is the amount of shares representing the USDC owed to this
position (includes principal, interest, etc.)
52    mapping(uint256 => uint256) internal stakeShares;
```

Figure C.10: contracts/Sherlock.sol:43-52

SherlockProtocolManager.sol

- **Use `premium` instead of `premiums_[_protocol]` as the denominator.** This will improve the gas usage.

```
193    function _secondsOfCoverageLeft(bytes32 _protocol) internal view returns(uint256) {
194      uint256 premium = premiums_[_protocol];
195      if (premium == 0) return 0;
196      return _activeBalance(_protocol) / premiums_[_protocol];
197    }
```

Figure C.11: contracts/managers/SherlockProtocolManager.sol:193-197

SherlockClaimManager.sol

- **Consider replacing the following lines of code with the equivalent single line proposed.** This will improve code readability by avoiding the declaration of variables that are used only once.

```
268   ISherlockProtocolManager protocolManager = sherlockCore.sherlockProtocolManager();
269   // Gets the protocol agent associated with the protocol ID passed in
270   address agent = protocolManager.protocolAgent(_protocol);
271   // Caller of this function must be the protocol agent address associated with the
protocol ID passed in
272   if (msg.sender != agent) revert InvalidSender();

// Equivalent to
if (msg.sender != sherlockCore.sherlockProtocolManager().protocolAgent(_protocol))
revert InvalidSender();
```

Figure C.12: contracts/managers/SherlockClaimManager.sol:268-272

```
278   Claim storage claim = claims_[claimIdentifier];
279   // verify if claim belongs to protocol agent
280   if (claim.protocol != _protocol) revert InvalidArgument();

// Equivalent to
if (claims_[claimIdentifier].protocol != _protocol) revert InvalidArgument();
```

Figure C.13: contracts/managers/SherlockClaimManager.sol:278-280

```
318   address agent = protocolManager.protocolAgent(_protocol);
319   // Caller of this function must be the protocol agent address associated with the
protocol ID passed in
320   if (msg.sender != agent) revert InvalidSender();

// Equivalent to
if (msg.sender != protocolManager.protocolAgent(_protocol)) revert InvalidSender();
```

Figure C.14: contracts/managers/SherlockClaimManager.sol:318-320

- **Declare `MAX_CALLBACKS` as constant.** This will improve code quality and gas usage.

```
52    uint256 MAX_CALLBACKS = 4;
```

Figure C.15: contracts/managers/SherlockClaimManager.sol:52

- **Move the check on the `claimCallbacks`'s length to above the `for` loop.** This will improve gas usage and code quality, as it is best practice to fail early.

```
229   // Checks to see if this callback contract already exists
230   for (uint256 i; i < claimCallbacks.length; i++) {
231     if (claimCallbacks[i] == _callback) revert InvalidArgument();
```

```
232     }
233     // Checks to see if the max amount of callback contracts has been reached
234     if (claimCallbacks.length == MAX_CALLBACKS) revert InvalidState();
```

Figure C.16: contracts/managers/SherlockClaimManager.sol:229-234

- **Replace `memory`, `ancillaryData`'s data location, with `calldata`.** This will improve gas usage.

```
293     function startClaim(
294       bytes32 _protocol,
295       uint256 _amount,
296       address _receiver,
297       uint32 _timestamp,
298       bytes memory ancillaryData
299     ) external override nonReentrant whenNotPaused {
```

Figure C.17: contracts/managers/SherlockClaimManager.sol:293-299

- **Rename the `claims` function to `claim`.** This will improve code readability: `claims` indicates that the function returns multiple claims, but it returns only one claim.

```
210   function claims(uint256 _claimID) external view override returns (Claim memory
claim_) {
```

Figure C.18: contracts/managers/SherlockClaimManager.sol:210

- **Correct the comment in figure C.19 to "Once `settle()` is executed."** This will improve code readability and prevent possible misunderstandings of the UMA integration.

```
575   // Once priceDisputed() is executed in UMA's contracts, this function gets called
```

Figure C.19: contracts/managers/SherlockClaimManager.sol:575

`SherDistributionManager.sol`

- **In the line of code in figure C.21, use = instead of +=, as this is the first assignment to `_sher`.** This will improve code readability.

```
125     _sher += (maxRewardsAvailable * maxRewardsRate * _period) * DECIMALS;
```

Figure C.21: contracts/managers/SherlockDistributionManager.sol:125

- **Revise the pullReward function so that it does not make transfers if _sher is zero.** This will improve code quality and gas usage.

```
69    function pullReward(
70      uint256 _amount,
71      uint256 _period,
72      uint256 _id,
73      address _receiver
74    ) external override onlySherlockCore returns (uint256 _sher) {
75      // Uses calcReward() to get the SHER tokens owed to this stake
76      // Subtracts the amount from the total token balance to get the pre-stake USDC
TVL
77      _sher = calcReward(sherlockCore.totalTokenBalanceStakers() - _amount, _amount,
_period);
78      // Sends the SHER tokens to the core Sherlock contract where they are held until
the unlock period for the stake expires
79      sher.safeTransfer(msg.sender, _sher);
80    }
```

Figure C.22: contracts/managers/SherlockDistributionManager.sol:69-80

D. Slither Risk Assessment

This section describes how Slither can be used to help assess the risk of a given protocol. Slither is an open-source static analysis tool developed by Trail of Bits that has been used to find over 40 vulnerabilities in live contracts. Along with its 76 detectors for common vulnerabilities in smart contracts, Slither offers the following features:

- 18 printers that output contract information
- `slither-check-upgradeability`, which detects 17 common vulnerabilities in upgradeable contracts
- `slither-check-erc`, which tests a contract's conformance to ERC standards

For a protocol to be insured by Sherlock, the Sherlock team assesses the protocol's risk. The outcome of this assessment determines the price of the requested coverage of the protocol.

```
On Sherlock's end, the first step is to do a "first look." This is where a security
expert from Sherlock takes an abbreviated look at the protocol. This look includes
everything from the size and complexity of the codebase, to any oracles and
composable protocols interacted with.

Once a Watson has achieved a better understanding of the protocol, Sherlock can then
give the protocol a range of where pricing will fall. Until a deeper dive is
completed (which can take weeks), the exact price cannot be known. However, if the
protocol agrees to the range given, Sherlock can then spend the resources to do a
deep security assessment.
```

Figure D.1: A snippet of the Sherlock documentation detailing the initial steps that are taken when a protocol requests coverage

Integrating Slither into the "first look" and "deep security assessment" steps would offer valuable information to the security expert, reduce the time to assess the risk, and increase the accuracy of the risk verdict.

For the "first look" step, Slither detectors could quickly show the number of possible issues with a protocol and offer valuable information that the Sherlock team could take into account for the initial price range.

For the “deep security assessment” step, the security expert could use all of Slither’s capabilities, including its detectors, printers, `slither-check-erc`, and `slither-check-upgradeability`, to determine the risk of a protocol. The following printers are especially useful in this regard:

- `human-summary`, which shows various contract statistics and the number of issues per contract
- `modifiers`, which shows all modifiers per function
- `inheritance` and `inheritance-graph`, which together output a text-based file or dotfile with the inheritance graph of the protocol’s contracts
- `vars-and-auth`, which shows the state variables written and the requirements for `msg.sender` per contract

E. Slither Scripts

The following contains two Slither scripts that we developed during the assessment. We recommend that the Sherlock team review and integrate these scripts into the project's CI pipeline.

Detecting the use of `require` versus `revert`

The Sherlock protocol codebase uses custom errors that are thrown using a `revert` statement. To normalize the method by which errors are thrown, all errors should be thrown using a `revert` statement. The script in figure E.1 can be used to identify places in which a `require` statement is used to check for errors.

```
from slither import Slither
from slither.core.declarations import Contract
from slither.slithir.operations import SolidityCall
from typing import List

slither = Slither(".", ignore_compile=True)

contracts_to_check = ["Sherlock",
                      "SherlockProtocolManager",
                      "SherlockClaimManager",
                      "SherDistributionManager",
                      "AaveV2Strategy",
                      "Manager"]

def find_contract_in_compilation_units(contracts_name: List[str]) -> List[Contract]:
    res = []
    for c in contracts_name:
        contracts = slither.get_contract_from_name(c)
        res.append(contracts[0]) if len(contracts) > 0 else print(f'Contract {c} not
found.')
    return res

def _check_require_in_code(contracts: List[Contract]):
    for contract in contracts:
        for function in contract.functions_declared:
            for node in function.nodes:
                for ir in node.irs:
                    if (isinstance(ir, SolidityCall) and
ir.function.full_name.startswith("require")):
                        print(f'Found usage of require in {function.canonical_name}')

_check_require_in_code(find_contract_in_compilation_units(contracts_to_check))
```

Figure E.1: A Python script to test for the use of `require` statements

Detecting functions that can be called when a contract is paused

The Sherlock protocol codebase has many functions that can be called when the relevant contract is paused. Functions that cannot be called when a contract is paused are

protected by the whenNotPaused modifier. We used Slither to identify and review the functions that are not protected by the whenNotPaused modifier.

The script in figure E.2 follows a whitelist approach, in which every reachable function must either be protected by the whenNotPaused modifier or be whitelisted.

The list of whitelisted functions (callable when the contract is paused) can be divided into two groups: basic ERC721 functions (which do not affect the Sherlock system) and configuration functions that are callable only by the contract owner (controlled by the Sherlock protocol).

The output of the script did not indicate functions that are missing the whenNotPaused modifier.

We recommend that the Sherlock team integrate this script in the project's development lifecycle so that developers are notified by any newly added functions that are missing the whenNotPaused modifier.

```
from slither import Slither
from slither.core.declarations import Contract
from typing import List


# Init slither
contracts = Slither('.')

def _check_access_controls(
    contract: Contract, modifier: str, whitelist: List[str]
):
    print(f"### Check {contract.name} pausable access controls")
    no_bug_found = True
    for function in contract.functions_entry_points:
        if function.is_constructor or "initialize" == function.name:
            continue
        if function.view:
            continue

        # if the function has no modifiers or not one of the modifiers we are searching
for
        if not function.modifiers or (
            not any((str(x) == modifier) for x in function.modifiers)
        ):
            # then if the function is not on the whitelist report that the whenNotPaused
modifier is missing
            if not function.name in whitelist:
                print(f"\t- {function.canonical_name} should have a pausable modifier")
                no_bug_found = False
    if no_bug_found:
```

```
        print("\t- No bug found")


_check_access_controls(
    contracts.get_contract_from_name("SherlockProtocolManager")[0],
    "whenNotPaused",
    [
        # SherlockProtocolManager
        "setMinActiveBalance", "setProtocolPremium", "setProtocolPremiums",
        "sweep",
        "protocolAdd", "protocolUpdate", "protocolRemove",

        # Manager
        "setSherlockCoreAddress", "unpause",
        "pause",       # has onlySherlockCore modifier
                       # internally calls _pause which has whenNotPaused

        # Ownable
        "renounceOwnership", "transferOwnership"
    ]
)

_check_access_controls(
    contracts.get_contract_from_name("SherlockClaimManager")[0],
    "whenNotPaused",
    [
        # SherlockClaimManager
        "renounceUmaHaltOperator",
        "addCallback", "removeCallback",
        "priceSettled",

        # Manager
        "setSherlockCoreAddress", "unpause",
        "pause",       # has onlySherlockCore modifier
                       # internally calls _pause which has whenNotPaused

        # Ownable
        "renounceOwnership", "transferOwnership"
    ]
)

_check_access_controls(
    contracts.get_contract_from_name("SherDistributionManager")[0],
    "whenNotPaused",
    [
        # SherDistributionManager
        "pullReward",  # has onlySherlockCore modifier,
                       # all calling functions have whenNotPaused
        "sweep",

        # Manager
        "setSherlockCoreAddress", "unpause",
        "pause",       # has onlySherlockCore modifier,
                       # internally calls _pause which has whenNotPaused

        # Ownable
        "renounceOwnership", "transferOwnership"
    ]
)
```

```
_check_access_controls(
    contracts.get_contract_from_name("AaveV2Strategy")[0],
    "whenNotPaused",
    [
        # AaveV2Strategy
        "withdrawAll", # has onlySherlockCore modifier,
                       # calling function does not have whenNotPaused
        "withdraw",    # has onlySherlockCore modifier
                       # calling function does not have whenNotPaused
        "sweep",

        # Manager
        "setSherlockCoreAddress", "unpause",
        "pause",       # has onlySherlockCore modifier
                       # internally calls _pause which has whenNotPaused

        # Ownable
        "renounceOwnership", "transferOwnership"
    ]
)

_check_access_controls(
    contracts.get_contract_from_name("Sherlock")[0],
    "whenNotPaused",
    [
        # Sherlock
        "enableStakingPeriod", "disableStakingPeriod",
        "updateSherDistributionManager", "removeSherDistributionManager",
        "updateNonStakersAddress",
        "updateSherlockProtocolManager", "updateSherlockClaimManager",
        "updateSherlockProtocolManager", "updateYieldStrategy",
        "yieldStrategyDeposit", "yieldStrategyWithdraw", "yieldStrategyWithdrawAll",
        "unpause",
        "pause", # whenNotPaused is applied to the internally called _pause() function

        # Ownable
        "renounceOwnership", "transferOwnership",

        # ERC721
        "approve", "setApprovalForAll", "transferFrom", "safeTransferFrom"
    ]
)
```

Figure E.2: A Python script to test for the whenNotPaused modifier

F. Echidna Property: Restake versus Fresh Stake

This section shows the Echidna property we used to verify that a fresh stake does not give more shares than restaking.

```
pragma solidity 0.8.0;

contract EchidnaTest {
    event Log(uint256, uint256, uint256, uint256, uint256);

    function reedem(
      uint256 shares,
      uint256 totalStakeShares,
      uint256 totalTokenBalanceStakers
    ) internal returns (uint256) {
      return shares * totalTokenBalanceStakers / totalStakeShares;
    }

    function initialStake(
      uint256 amount,
      uint256 totalStakeShares,
      uint256 totalTokenBalanceStakers
    ) internal returns (uint256) {
      // no subtract as in the original code because we didn't receive the tokens
      return amount * totalStakeShares / totalTokenBalanceStakers;
    }

    function check_restake(
      uint256 currShares,
      uint256 totalStakeShares,
      uint256 totalTokenBalanceStakers
    ) public {
      currShares = 10**6 + currShares % (type(uint256).max - 10**6);

      // we don't want to have all the shares
      totalStakeShares = currShares + 1 + totalStakeShares % (
        type(uint256).max - (currShares + 1)
      );

      totalTokenBalanceStakers = 10**6 + totalTokenBalanceStakers % (
        type(uint256).max - 10**6
      );

      uint256 amount_to_restake = reedem(
        currShares, totalStakeShares, totalTokenBalanceStakers
      );

      uint256 shares_if_you_restake = initialStake(
        amount_to_restake,
        totalStakeShares - currShares,
        totalTokenBalanceStakers - amount_to_restake
      );

      emit Log(
        totalStakeShares,
        totalTokenBalanceStakers,
```

```
        amount_to_restake,
        currShares,
        shares_if_you_restake
      );

      assert(shares_if_you_restake <= currShares);
    }
}
```

Figure F.1: Echidna property verifying that a fresh stake does not give more shares than restaking

G. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. Refer to an up-to-date version of the checklist on `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
```

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 Security 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.

ERC Conformity

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.
- ❏ **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.

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.

Finally, there are certain characteristics that are difficult to identify automatically. Conduct a manual review of the following conditions:

- ❏ **`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.

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 if 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.

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.

H. Fix Log

On January 5, 2021, Trail of Bits reviewed the fixes and mitigations implemented by the Sherlock team for the issues identified in this report. The Sherlock team fixed six of the issues reported in the original assessment and did not fix the other three. We reviewed each of the fixes to ensure that the proposed remediation would be effective. For additional information, please refer to the Detailed Fix Log.

ID	Title	Severity	Fix Status
1	Solidity compiler optimizations can be problematic	Undetermined	Not Fixed
2	Certain functions lack zero address checks	Medium	Fixed (`1091f71`)
3	updateYieldStrategy could leave funds in the old strategy	High	Fixed (`884868f`)
4	Pausing and unpausing the system may not be possible when removing or replacing connected contracts	Low	Fixed (`6e73079`)
5	SHER reward calculation uses confusing six-decimal SHER reward rate	Informational	Fixed (`cd0be9d`)
6	A claim cannot be paid out or escalated if the protocol agent changes after the claim has been initialized	Medium	Not Fixed
7	Missing input validation in setMinActiveBalance could cause a confusing event to be emitted	Informational	Fixed (`be43c88`)
8	payoutClaim's calling of external contracts in a loop could cause a denial of service	Low	Not Fixed

9	pullReward could silently fail and cause stakers to lose all earned SHER rewards	**High**	**Fixed** (5248d99)

Detailed Fix Log

TOB-SHER-001: Solidity compiler optimizations can be problematic
Not fixed. The Sherlock team responded to this finding as follows: "Turning off compiler optimizations would make the contracts too big."

TOB-SHER-002: Certain functions lack zero address checks
Fixed. Appropriate zero address checks have been added to the functions highlighted in the detailed finding. (`1091f71`)

TOB-SHER-003: updateYieldStrategy could leave funds in the old strategy
Fixed. The `updateYieldStrategy` function now calls `yieldStrategy.withdrawAll` on the old strategy inside a try/catch operation. Particular attention needs to be made if the `yieldStrategy.withdrawAll` temporarily fails. (`884868f`)

TOB-SHER-004: Pausing and unpausing the system may not be possible when removing or replacing connected contracts
Fixed. The `Sherlock.pause` and `Sherlock.unpause` functions now check that the contracts are in the expected state and that `SherlockDistributionManager` is not set to the zero address. (`6e73079`)

TOB-SHER-005: SHER reward calculation uses confusing six-decimal SHER reward rate
Fixed. `maxRewardsRate` now uses an 18-decimal value. (`cd0be9d`)

TOB-SHER-006: A claim cannot be paid out or escalated if the protocol agent changes after the claim has been initialized
Not fixed. The Sherlock team did not make any code changes; the team added its reason as a comment (`272aca8`) and highlighted this behavior in the documentation (`d9e42b7`).

TOB-SHER-007: Missing input validation in setMinActiveBalance could cause a confusing event to be emitted
Fixed. The `setMinActiveBalance` now reverts if the proposed `_minActiveBalance` value is equal to the current value. (`be43c88`)

TOB-SHER-008: payoutClaim's calling of external contracts in a loop could cause a denial of service
Not fixed.

TOB-SHER-009: pullReward could silently fail and cause stakers to lose all earned SHER rewards
Fixed. A stake/restake transaction now reverts if the call to `pullReward` fails. (`5248d99`)