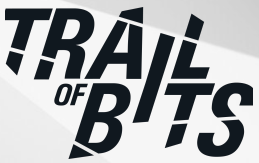

Cowri Labs Shell Protocol v2

Security Assessment

September 27, 2022

Prepared for:

Kenny White

Cowri Labs

Thomas Sciaroni

Cowri Labs

Prepared by: **Felipe Manzano and Anish Naik**

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

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

Cowri Labs engaged Trail of Bits to review the security of version 2 of its Shell Protocol. From April 25 to May 6, 2022, 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

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 documentation. We performed static analysis, fuzz testing, and a manual review of the project's Solidity components.

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	4
Medium	2
Low	0
Informational	2
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Auditing and Logging	1
Data Validation	2
Patching	1
Testing	1
Timing	1
Undefined Behavior	1

Notable Findings

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

- **Denial-of-service conditions caused by the use of more than 256 slices (TOB-SHELL-1)**
  When performing a swap, deposit, or withdrawal, the Proteus system first tries to identify the slice in which it should begin the operation (i.e., the slice with the correct token balance ratio). However, the iterator used to index into the `slices` array is defined as a `uint8`; if there are more than 256 slices, the iterator's operation will *silently* overflow. Thus, if a malicious owner created 257 slices and included the token balance ratio in the 257th slice, the system would be unable to find the correct slice, and all transactions would revert with an out-of-gas (OOG) exception. In other words, the system would experience a denial-of-service condition.

- **LiquidityPoolProxy owners can steal user funds (TOB-SHELL-2)**
  The `LiquidityPoolProxy` contract handles incoming calls from the 0cean contract and passes them to a contract that supports the `ILiquidityPoolImplementation` interface. However, the owner of the `LiquidityPoolProxy` can change the underlying implementation contract address at any point and could cause users to incur a loss of funds by setting it to a malicious implementation contract.

- **Risk of sandwich attacks (TOB-SHELL-3)**
  Because the `LiquidityPool` contract lacks a slippage parameter, trades executed through that contract are vulnerable to sandwich attacks. This means that a user could lose all of the funds that he or she had provided for a swap.

- **Certain identity curve configurations can lead to a loss of pool tokens (TOB-SHELL-6)**
  Our dynamic testing of the Proteus engine (via Echidna) identified certain curve configurations that could lead to a loss of pool tokens or the dilution of liquidity provider tokens. These configuration issues presumably stem from rounding errors in integer division operations. The Shell Protocol team should perform further analysis of the system to ensure that all rounding operations benefit the pool.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

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

The following engineers were associated with this project:

Felipe Manzano, Consultant
felipe@trailofbits.com

Anish Naik, Consultant
anishnaik@trailofbits.com

Project Timeline

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

Date	Event
April 21, 2022	Pre-project kickoff call
April 29, 2022	Status update meeting #1
May 9, 2022	Delivery of report draft; report readout meeting
June 7, 2022	Delivery of final report
September 26, 2022	Delivery of public report

Project Goals

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

- Could a malicious user wrap or unwrap another user's tokens?
- Could an attacker craft an array of interactions that would lead to the loss or theft of funds?
- Are the token balances of users and primitives maintained and updated correctly?
- Are there any denial-of-service attack vectors?
- Does dynamic testing invalidate any of the system properties?
- Does the system validate the data it receives from external contracts?
- Could a malicious actor sandwich or front-run a user's trade?
- Are the privileges of the `Proteus` and `LiquidityPoolProxy` contract owners sufficiently limited?

Project Targets

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

Shell Protocol v2

Repository	https://github.com/cowri/shell-protocol-v2-contracts
Version	`fc95907`
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:

0cean. This non-upgradeable smart contract is the accounting layer of the Shell Protocol. The 0cean contract allows users to interact with a variety of primitives (e.g., NFT exchanges, automated market makers (AMMs), and lending pools) by wrapping ERC20, ERC721, or ERC1155 tokens in exchange for the minting of representative shell tokens. Users can exit the system by burning their shell tokens and redeeming their original ERC20, ERC721, or ERC1155 tokens. In analyzing the 0cean contract's security posture, we reviewed the following:

- Whether it is possible to wrap or unwrap tokens without owning the requisite tokens
- Whether it is possible to use an incorrect `InteractionType` in the execution of an interaction
- Whether the rounding and truncation arithmetic is performed correctly
- Whether primitives' native and non-native token balances are tracked correctly
- Whether there are any edge cases in the execution of `doMultipleInteractions` that could lead to the loss or theft of funds
- Whether transactions forwarded by trusted forwarders maintain their integrity
- Whether any external token / contract interactions could lead to unintended outcomes

0ceanERC1155. The 0cean contract inherits from the non-upgradeable 0ceanERC1155 contract, which is an implementation of OpenZeppelin's ERC1155 token standard. This contract is responsible for locking / unlocking external tokens and minting / burning representative shell tokens. The contract also enables users to transfer their shell tokens to each other and includes functions added specifically to support the use case of the Shell Protocol. In analyzing the 0ceanERC1155 contract's security posture, we checked the following:

- That the contract complies with the ERC1155 standard and that any deviations from that standard do not lead to undefined behavior
- That the custom additions to the ERC1155 standard work as expected

- That calls to the `Ocean` contract cannot lead to a loss of funds or the theft of funds from the `OceanERC1155` contract
- That `oceanId` hash collisions are not possible

`Interactions`. This non-upgradeable smart contract defines the data structure that enables user interactions with the `Ocean` contract and an interface that the `Ocean` contract must comply with. We ensured that the `Ocean` contract complies with the interface specification and uses the data structure correctly.

`BalanceDelta`. The `Ocean` contract uses this library to store token balances in memory and to perform arithmetic operations on those balances. Tracking token balances in memory helps optimize gas costs and enables users to perform multiple operations atomically through the `Ocean` contract. In reviewing the security posture of the `BalanceDelta` library, we checked the following:

- Whether all conversions between unsigned and signed integers are performed correctly
- Whether the token balance arithmetic could lead to any overflows or a loss of user funds
- Whether final token balances are committed to memory correctly

`Proteus` and `ProteusLogic`. These non-upgradeable smart contracts are the Shell Protocol's implementation of a generalizable AMM engine that allows for flexibility in the design of the underlying bonding curve. Any protocol or team can use the Proteus engine to create an AMM and to register that AMM with the `Ocean` contract as a primitive. Our analysis of the Proteus contracts' security posture included the following:

- A review of the `Proteus` contract owner's privileges to identify any privileged operations that could enable the owner to steal user funds
- Dynamic testing of the `Proteus` contract's arithmetic operations to check the arithmetic invariants and to identify any edge cases that could lead to undefined behavior
- A check of `Proteus` and `ProteusLogic` for any overlapping code that could lead to discrepancies as the protocol changes

Liquidity pool contracts. These non-upgradeable smart contracts implement a liquidity pool that uses the `Proteus` contract as its underlying AMM engine. The liquidity pool contracts implement the interface necessary for a primitive's integration with the `Ocean` contract. Our analysis of these contracts' security posture included the following:

- A review of the liquidity pool contracts' compliance with the `IOceanPrimitive` interface, which facilitates interactions with the `Ocean` contract
- A review of the privileges of the `LiquidityPoolProxy` contract owner to identify any privileged operations that could lead to the theft or loss of user funds
- Checks for ways in which a malicious actor could front-run or sandwich a user's trade
- Checks of the contracts' interactions with the `Proteus` contract to identify any interactions that could lead to undefined behavior

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:

- **Forwarder logic.** Users can interact with the `Ocean` contract both directly and through trusted forwarders. At Cowri Labs's request, we excluded the forwarder logic from the scope of the audit.
- **`MockOcean`.** This non-upgradeable contract is used primarily to test the Proteus engine and was thus not reviewed in this audit.
- **`ABDKMath`.** This out-of-scope library makes it possible to perform arithmetic operations using signed 64.64 integers. The Proteus engine uses the library to perform a large number of its calculations.

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.

In this assessment, we used Echidna, a smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation, to check various system states.

Test Results

The results of this testing are detailed below.

`Proteus.sol` and `ProteusLogic.sol`. We reviewed a number of Foundry-based fuzz tests written by the Shell Protocol team and repurposed those tests so that they could be run with Echidna. We used Echidna to assess the test coverage and updated tests that explored a limited amount of input space (TOB-SHELL-6).

Property	Tool	Result
A call to `swapOutput` on token X or token Y should not decrease the pool's utility.	Echidna	**Failed**
A call to `swapInput` on token X or token Y should not decrease the pool's utility.	Echidna	**Failed**
A call to `depositOutput` on token X or token Y should not decrease utility per shell (UPS).	Echidna	**Failed**
A call to `depositInput` on token X or token Y should not decrease UPS.	Echidna	**Failed**
A call to `withdrawOutput` on token X or token Y should not decrease UPS.	Echidna	**Failed**
A call to `withdrawInput` on token X or token Y should not decrease UPS.	Echidna	**Failed**
A call to `withdrawInput` to withdraw some amount of token X followed by a call to `depositInput` to deposit that same amount of token X should not decrease the	Echidna	**Failed**

pool's balance of token X.		
A call to `swapOutput` on token X or token Y should not decrease the pool's utility by more than 10^{-6} %.	Echidna	**Passed**
A call to `swapInput` on token X or token Y should not decrease the pool's utility by more than 10^{-6} %.	Echidna	**Passed**
A call to `depositOutput` on token X or token Y should not decrease UPS by more than 10^{-6} %.	Echidna	**Passed**
A call to `depositInput` on token X or token Y should not decrease UPS by more than 10^{-6} %.	Echidna	**Passed**
A call to `withdrawOutput` on token X or token Y should not decrease UPS by more than 10^{-6} %.	Echidna	**Passed**
A call to `withdrawInput` on token X or token Y should not decrease UPS by more than 10^{-6} %.	Echidna	**Passed**

`Ocean.sol`. The `Ocean` contract is an ERC1155-like ledger that can wrap / unwrap tokens and compose operations over external tokens and contracts. We used Echidna to test an emulated version of the contract. We also used small proxy contracts that emulated users to execute a random set of interactions through the `doMultipleInteractions` function. This allowed the fuzzer to exercise various code paths, such as an attempt to wrap an ERC721 token with an incorrect `InteractionType`. At the end of each run, we checked whether any account held more tokens than it had held before the run. We did not dynamically test the primitive integrations or native ERC1155 token transfers.

Property	Tool	Result
No account holds more ERC{20, 721, 1155} tokens after a call to the `doMultipleInteractions` function than it held before the call.	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 system uses Solidity v0.8.0 arithmetic operations, and the expected behavior of the system arithmetic is well documented. There are also thorough unit tests of the token balance arithmetic in the `Ocean` contract. The Proteus engine's bonding curve arithmetic is tested through fuzz testing. However, our Echidna testing identified certain invariants that do not hold across all curve configurations (TOB-SHELL-6). The Shell Protocol team should further investigate these failing cases to ensure that they do not break the underlying AMM architecture.	**Further Investigation Required**
Auditing	All critical state-changing operations in the `Ocean` contract emit events. However, two events are missing from the `Proteus` and `LiquidityPoolProxy` contracts (TOB-SHELL-7). Additionally, the Shell Protocol team did not provide an incident response plan. Recommendations on developing and maintaining an incident response plan are provided in appendix D.	**Moderate**
Authentication / Access Controls	The `Ocean` contract is owned by a multisignature wallet that can change only the fee for unwrapping operations, which is capped at 0.05%. The system follows the principle of least privilege, and users can enter and exit the system at will. The owners of the `LiquidityPoolProxy` and `Proteus` contracts, however, have excessive privileges that could enable them to steal user funds (TOB-SHELL-1,	**Moderate**

	TOB-SHELL-2).	
Complexity Management	All logic is separated into functions that are well documented and have clear purposes. However, some functions in the `Ocean` contract and many functions in the `Proteus` contract are very large and use excessive branching. Over time, the maintenance and testing of those functions could become difficult.	**Moderate**
Decentralization	The `Ocean` contract is a permissionless system whose owner can change only the fee charged for token unwrapping operations. Any user, team, or protocol can integrate with the `Ocean` contract as a primitive. However, because the contract is permissionless, the user documentation should cover the risks associated with a malicious primitive in the system.	**Satisfactory**
Documentation	We were provided with documentation sufficient for analysis of the protocol's process flows, data structures, and arithmetic operations. However, additional documentation on the differences between constant product slices and constant sum slices and the ways in which slices are represented and traversed programmatically would be beneficial.	**Satisfactory**
Front-Running Resistance	The `LiquidityPool` contract is vulnerable to sandwich attacks. Although the pool receives the data necessary to prevent excessive slippage, it does not use that data to protect users from excessive slippage (TOB-SHELL-3). Additionally, the documentation does not emphasize the fact that AMM primitives must protect users from front-running attacks.	**Weak**
Testing and Verification	The `Ocean` contract has 100% unit test coverage. The Proteus engine is extensively tested through fuzz testing; however, implementing the recommendations provided in TOB-SHELL-6 and appendix C would improve this fuzz testing.	**Satisfactory**

Summary of Findings

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

ID	Title	Type	Severity
1	Denial-of-service conditions caused by the use of more than 256 slices	Data Validation	High
2	LiquidityPoolProxy owners can steal user funds	Access Controls	High
3	Risk of sandwich attacks	Timing	High
4	Project dependencies contain vulnerabilities	Patching	Medium
5	Use of duplicate functions	Undefined Behavior	Informational
6	Certain identity curve configurations can lead to a loss of pool tokens	Testing	High
7	Lack of events for critical operations	Auditing and Logging	Medium
8	Ocean may accept unexpected airdrops	Data Validation	Informational

Detailed Findings

1. Denial-of-service conditions caused by the use of more than 256 slices	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SHELL-1
Target: `proteus-v3-solidity-main/src/Proteus.sol`	

Description
The owner of a Proteus-based automated market maker (AMM) can update the system parameters to cause a denial of service (DoS) upon the execution of swaps, withdrawals, and deposits.

The Proteus AMM engine design supports the creation of an arbitrary number of *slices*. Slices are used to segment an underlying bonding curve and provide variable liquidity across that curve. The owner of a `Proteus` contract can update the number of slices by calling the `_updateSlices` function at any point.

When a user requests a swap, deposit, or withdrawal operation, the `Proteus` contract first calls the `_findSlice` function (figure 1.1) to identify the slice in which it should perform the operation. The function iterates across the `slices` array and returns the index, `i`, of the slice that has the current ratio of token balances, `m`.

```
function _findSlice(int128 m) internal view returns (uint8 i) {
        i = 0;
        while (i < slices.length) {
            if (m <= slices[i].mLeft && m > slices[i].mRight) return i;
            unchecked {
                ++i;
            }
        }
        // while loop terminates at i == slices.length
        // if we just return i here we'll get an index out of bounds.
        return i - 1;
    }
```

Figure 1.1: The `_findSlice()` function in `Proteus.sol#L1168-1179`

However, the index, `i`, is defined as a `uint8`. If the owner sets the number of slices to at least 257 (by calling `_updateSlices`) and the current `m` is in the 257th slice, `i` will silently overflow, and the `while` loop will continue until an out-of-gas (OOG) exception occurs. If a

deposit, withdrawal, or swap requires the 257th slice to be accessed, the operation will fail because the `_findSlice` function will be unable to reach that slice.

Exploit Scenario

Eve creates a seemingly correct Proteus-based primitive (one with only two slices near the asymptotes of the bonding curve). Alice deposits assets worth USD 100,000 into a pool. Eve then makes a deposit of X and Y tokens that results in a token balance ratio, `m`, of 1. Immediately thereafter, Eve calls `_updateSlices` and sets the number of slices to 257, causing the 256th slice to have an `m` of 1.01. Because the current `m` resides in the 257th slice, the `_findSlice` function will be unable to find that slice in any subsequent swap, deposit, or withdrawal operation. The system will enter a DoS condition in which all future transactions will fail.

If Eve identifies an arbitrage opportunity on another exchange, Eve will be able to call `_updateSlices` again, use the unlocked curve to buy the token of interest, and sell that token on the other exchange for a pure profit. Effectively, Eve will be able to steal user funds.

Recommendations

Short term, change the index, `i`, from the `uint8` type to `uint256`; alternatively, create an upper limit for the number of slices that can be created and ensure that `i` will not overflow when the `_findSlice` function searches through the `slices` array.

Long term, consider adding a delay between a call to `_updateSlices` and the time at which the call takes effect on the bonding curve. This will allow users to withdraw from the system if they are unhappy with the new parameters. Additionally, consider making slices immutable after their construction; this will significantly reduce the risk of undefined behavior.

2. LiquidityPoolProxy owners can steal user funds

Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-SHELL-2
Target: `proteus-v3-solidity-main/src/LiquidityPoolProxy.sol`	

Description

The `LiquidityPoolProxy` contract implements the `IOceanPrimitive` interface and can integrate with the `Ocean` contract as a primitive. The proxy contract calls into an implementation contract to perform deposit, swap, and withdrawal operations (figure 2.1).

```
function swapOutput(uint256 inputToken, uint256 inputAmount)
    public
    view
    override
    returns (uint256 outputAmount)
{
    (uint256 xBalance, uint256 yBalance) = _getBalances();
    outputAmount = implementation.swapOutput(xBalance, yBalance, inputToken ==
xToken ? 0 : 1, inputAmount);
}
```

Figure 2.1: The `swapOutput()` function in `LiquidityPoolProxy.sol#L39–47`

However, the owner of a `LiquidityPoolProxy` contract can perform the privileged operation of changing the underlying implementation contract via a call to `setImplementation` (figure 2.2). The owner could thus replace the underlying implementation with a malicious contract to steal user funds.

```
function setImplementation(address _implementation)
        external
        onlyOwner
    {
        implementation = ILiquidityPoolImplementation(_implementation);
    }
```

Figure 2.2: The `setImplementation()` function in `LiquidityPoolProxy.sol#L28–33`

This level of privilege creates a single point of failure in the system. It increases the likelihood that a contract's owner will be targeted by an attacker and incentivizes the owner to act maliciously.

Exploit Scenario

Alice deploys a `LiquidityPoolProxy` contract as an `Ocean` primitive. Eve gains access to Alice's machine and upgrades the implementation to a malicious contract that she controls. Bob attempts to swap USD 1 million worth of shDAI for shUSDC by calling `computeOutputAmount`. Eve's contract returns 0 for `outputAmount`. As a result, the malicious primitive's balance of shDAI increases by USD 1 million, but Bob does not receive any tokens in exchange for his shDAI.

Recommendations

Short term, document the functions and implementations that `LiquidityPoolProxy` contract owners can change. Additionally, split the privileges provided to the owner role across multiple roles to ensure that no one address has excessive control over the system.

Long term, develop user documentation on all risks associated with the system, including those associated with privileged users and the existence of a single point of failure.

3. Risk of sandwich attacks

Severity: **High**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-SHELL-3
Target: `proteus-v3-solidity-main/src/LiquidityPool.sol`	

Description

The Proteus liquidity pool implementation does not use a parameter to prevent slippage. Without such a parameter, there is no guarantee that users will receive any tokens in a swap.

The `LiquidityPool` contract's `computeOutputAmount` function returns an `outputAmount` value indicating the number of tokens a user should receive in exchange for the `inputAmount`. Many AMM protocols enable users to specify the *minimum* number of tokens that they would like to receive in a swap. This minimum number of tokens (indicated by a slippage parameter) protects users from receiving fewer tokens than expected.

As shown in figure 3.1, the `computeOutputAmount` function signature includes a 32-byte `metadata` field that would allow a user to encode a slippage parameter.

```
function computeOutputAmount(
    uint256 inputToken,
    uint256 outputToken,
    uint256 inputAmount,
    address userAddress,
    bytes32 metadata
) external override onlyOcean returns (uint256 outputAmount) {
```

Figure 3.1: The signature of the `computeOutputAmount()` function in `LiquidityPool.sol#L192–198`

However, this field is not used in swaps (figure 3.2) and thus does not provide any protection against excessive slippage. By using a bot to sandwich a user's trade, an attacker could increase the slippage incurred by the user and profit off of the spread at the user's expense.

```
function computeOutputAmount(
    uint256 inputToken,
    uint256 outputToken,
    uint256 inputAmount,
```

```
    address userAddress,
    bytes32 metadata
) external override onlyOcean returns (uint256 outputAmount) {
    ComputeType action = _determineComputeType(inputToken, outputToken);
    [...]
    } else if (action == ComputeType.Swap) {
        // Swap action + computeOutput context => swapOutput()
        outputAmount = swapOutput(inputToken, inputAmount);

        emit Swap(
            inputAmount,
            outputAmount,
            metadata,
            userAddress,
            (inputToken == xToken),
            true
        );
    }
    [...]
```

Figure 3.2: Part of the `computeOutputAmount()` function in LiquidityPool.sol#L192–260

Exploit Scenario

Alice wishes to swap her shUSDC for shwETH. Because the `computeOutputAmount` function's `metadata` field is not used in swaps to prevent excessive slippage, the trade can be executed at any price. As a result, when Eve sandwiches the trade with a buy and sell order, Alice sells the tokens without purchasing any, effectively giving away tokens for free.

Recommendations

Short term, document the fact that protocols that choose to use the Proteus AMM engine should encode a slippage parameter in the `metadata` field. The use of this parameter will reduce the likelihood of sandwich attacks against protocol users.

Long term, ensure that all calls to `computeOutputAmount` and `computeInputAmount` use slippage parameters when necessary, and consider relying on an oracle to ensure that the amount of slippage that users can incur in trades is appropriately limited.

4. Project dependencies contain vulnerabilities	
Severity: **Medium**	Difficulty: **Low**
Type: Patching	Finding ID: TOB-SHELL-4
Target: `shell-protocol-v2-contracts`	

Description

Although dependency scans did not identify a direct threat to the project under review, `npm` and `yarn audit` identified dependencies with known vulnerabilities. Due to the sensitivity of the deployment code and its environment, it is important to ensure that dependencies are not malicious. Problems with dependencies in the JavaScript community could have a significant effect on the repository under review. The output below details these issues:

CVE ID	Description	Dependency
CVE-2021-23358	Arbitrary code injection vulnerability	`underscore`
CVE-2021-43138	Prototype pollution	`async`
CVE-2021-23337	Command injection vulnerability	`lodash`
CVE-2022-0235	`node-fetch` is vulnerable to exposure of sensitive information to an unauthorized actor	`node-fetch`

Figure 4.1: Advisories affecting `shell-protocol-v2-contracts` dependencies

Exploit Scenario

Alice installs the dependencies of the in-scope repository on a clean machine. Unbeknownst to Alice, a dependency of the project has become malicious or exploitable. Alice subsequently uses the dependency, disclosing sensitive information to an unknown actor.

Recommendations

Short term, ensure that the Shell Protocol dependencies are up to date. Several node modules have been documented as malicious because they execute malicious code when installing dependencies to projects. Keep modules current and verify their integrity after installation.

Long term, integrate automated dependency auditing into the development workflow. If a dependency cannot be updated when a vulnerability is disclosed, ensure that the code does not use and is not affected by the vulnerable functionality of the dependency.

5. Use of duplicate functions

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-SHELL-5
Target: `proteus-v3-solidity-main/src/ProteusLogic.sol`, `proteus-v3-solidity-main/src/Proteus.sol`	

Description

The `ProteusLogic` and `Proteus` contracts both contain a function used to update the internal `slices` array. Although calls to these functions currently lead to identical outcomes, there is a risk that a future update could be applied to one function but not the other, which would be problematic.

```
<     function _updateSlices(int128[] memory slopes, int128[] memory rootPrices)
internal {
<         require(slopes.length == rootPrices.length);
<         require(slopes.length > 1);
---
>     function _updateSlices(int128[] memory slopes, int128[] memory rootPrices)
>         internal
>     {
>         if (slopes.length != rootPrices.length) {
>             revert UnequalArrayLengths();
>         }
>         if (slopes.length < 2) {
>             revert TooFewParameters();
>         }
```

Figure 5.1: The diff between the `ProteusLogic` and `Proteus` contracts' `_updateSlices()` functions

Using duplicate functions in different contracts is not best practice. It increases the risk of a divergence between the contracts and could significantly affect the system properties. Defining a function in one contract and having other contracts call that function is less risky.

Exploit Scenario

Alice, a developer of the Shell Protocol, is tasked with updating the `ProteusLogic` contract. The update requires a change to the `Proteus._updateSlices` function. However, Alice forgets to update the `ProteusLogic._updateSlices` function. Because of this omission, the functions' updates to the internal `slices` array may produce different results.

Recommendations

Short term, select one of the two `_updateSlices` functions to retain in the codebase and to maintain going forward.

Long term, consider consolidating the `Proteus` and `ProteusLogic` contracts into a single implementation, and avoid duplicating logic.

6. Certain identity curve configurations can lead to a loss of pool tokens

Severity: **High**	Difficulty: **Undetermined**
Type: Testing	Finding ID: TOB-SHELL-6
Target: `proteus-v3-solidity-main/src/test/ComplexSwapStability.t.sol`, `proteus-v3-solidity-main/src/test/Util.t.sol`	

Description
A rounding error in an integer division operation could lead to a loss of pool tokens and the dilution of liquidity provider (LP) tokens.

We reimplemented certain of Cowri Labs's fuzz tests and used Echidna to test the system properties specified in the Automated Testing section. The original fuzz testing used a fixed amount of 100 tokens for the initial `xBalance` and `yBalance` values; after we removed that limitation, Echidna was able to break some of the invariants. The Shell Protocol team should identify the *largest* possible percentage decrease in pool utility or utility per shell (UPS) to better quantify the impact of a broken invariant on system behavior.

In some of the breaking cases, the ratio of token balances, `m`, was close to the X or Y asymptote of the identity curve. This means that an attacker might be able to disturb the balance of the pool (through flash minting or a large swap, for example) and then exploit the broken invariants.

Exploit Scenario
Alice withdraws USD 100 worth of token X from a Proteus-based liquidity pool by burning her LP tokens. She eventually decides to reenter the pool and to provide the same amount of liquidity. Even though the curve's configuration is similar to the configuration at the time of her withdrawal, her deposit leads to only a USD 90 increase in the pool's balance of token X; thus, Alice receives fewer LP tokens than she should in return, effectively losing money because of an arithmetic error.

Recommendations
Short term, investigate the root cause of the failing properties. Document and test the expected rounding direction (up or down) of each arithmetic operation, and ensure that the rounding direction used in each operation benefits the pool.

Long term, implement the fuzz testing recommendations outlined in appendix C.

7. Lack of events for critical operations

Severity: **Medium**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-SHELL-7
Target: `proteus-v3-solidity-main/src/Proteus.sol`, `proteus-v3-solidity-main/src/LiquidityPoolProxy.sol`	

Description

Two critical operations do not trigger events. As a result, it will be difficult to review the correct behavior of the contracts once they have been deployed.

The `LiquidityPoolProxy` contract's `setImplementation` function is called to set the implementation address of the liquidity pool and does not emit an event providing confirmation of that operation to the contract's caller (figure 7.1).

```
function setImplementation(address _implementation)
        external
        onlyOwner
    {
        implementation = ILiquidityPoolImplementation(_implementation);
    }
```

Figure 7.1: The `setImplementation()` function in `LiquidityPoolProxy.sol#L28–33`

Calls to the `updateSlices` function in the `Proteus` contract do not trigger events either (figure 7.2). This is problematic because updates to the `slices` array have a significant effect on the configuration of the identity curve (TOB-SHELL-1).

```
function updateSlices(int128[] memory slopes, int128[] memory rootPrices)
    external
    onlyOwner
{
    _updateSlices(slopes, rootPrices);
}
```

Figure 7.2: The `updateSlices()` function in `Proteus.sol#L623–628`

Without events, users and blockchain-monitoring systems cannot easily detect suspicious behavior.

Exploit Scenario

Eve, an attacker, is able to take ownership of the `LiquidityPoolProxy` contract. She then sets a new implementation address. Alice, a Shell Protocol team member, is unaware of the change and does not raise a security incident.

Recommendations

Short term, add events for all critical operations that result in state changes. Events aid in contract monitoring and the detection of suspicious behavior.

Long term, consider using a blockchain-monitoring system to track any suspicious behavior in the contracts. The system relies on several contracts to behave as expected. A monitoring mechanism for critical events would quickly detect any compromised system components.

8. Ocean may accept unexpected airdrops

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SHELL-8
Target: `contracts/Ocean.sol`	

Description

Unexpected transfers of tokens to the `Ocean` contract may break its internal accounting, essentially leading to the loss of the transferred asset. To mitigate this risk, `Ocean` attempts to reject airdrops.

Per the ERC721 and ERC1155 standards, contracts must implement specific methods to accept or deny token transfers. To do this, the `Ocean` contract uses the `onERC721Received` and `onERC1155Received` callbacks and `_ERC1155InteractionStatus` and `_ERC721InteractionStatus` storage flags. These storage flags are enabled in ERC721 and ERC1155 wrapping operations to facilitate successful standard-compliant transfers.

However, the `_erc721Unwrap` and `_erc1155Unwrap` functions also enable the `_ERC721InteractionStatus` and `_ERC1155InteractionStatus` flags, respectively. Enabling these flags allows for airdrops, since the `Ocean` contract, not the user, is the recipient of the tokens in unwrapping operations.

```
    function _erc721Unwrap(
        address tokenAddress,
        uint256 tokenId,
        address userAddress,
        uint256 oceanId
    ) private {
        _ERC721InteractionStatus = INTERACTION;
        IERC721(tokenAddress).safeTransferFrom(
            address(this),
            userAddress,
            tokenId
        );
        _ERC721InteractionStatus = NOT_INTERACTION;
        emit Erc721Unwrap(tokenAddress, tokenId, userAddress, oceanId);
    }
```

Figure 8.1: The `_erc721Unwrap()` function in `Ocean.sol#L1020-1034`

Exploit Scenario

Alice calls the `_erc721Unwrap` function. When the `onERC721Received` callback function in Alice's contract is called, Alice mistakenly sends the ERC721 tokens back to the `0cean` contract. As a result, her ERC721 is permanently locked in the contract and effectively burned.

Recommendations

Short term, disallow airdrops of standard-compliant tokens during unwrapping interactions and document the edge cases in which the `0cean` contract will be unable to stop token airdrops.

Long term, when the `0cean` contract is expecting a specific airdrop, consider storing the originating address of the transfer and the token type alongside the relevant interaction flag.

Summary of Recommendations

Trail of Bits recommends that Cowri Labs address the findings detailed in this report and take the following additional steps prior to deployment:

- Document the risks that users face and the responsibilities they have when interacting with external contracts such as permissionless primitives through the `0cean` contract.
- Create an internal process for reviewing each primitive that will be integrated with the `0cean` contract. Ensure that primitive owners have limited privileges and that there is minimal risk of theft.
- Prevent a primitive's owner from dynamically changing the Proteus engine's identity bonding curve or thoroughly document the changes that the owner can make.
- Implement the recommendations outlined in TOB-SHELL-6 and appendix C to improve the Proteus fuzz testing suite. Ensure that the AMM engine design cannot be broken in a significant way by any of the edge cases identified during this audit.
- Develop a detailed incident response plan.

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
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
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. Fuzz Testing Improvements

This appendix outlines recommendations on improving the fuzz testing of the Proteus engine and the maturity of the Shell Protocol's fuzz testing methodology.

Proteus-Specific Recommendations

The Proteus engine has numerous fuzz tests that are run with Foundry. To assess the fuzz test coverage and identify any flaws in the tests, we moved a number of those tests and repurposed them for execution with Echidna. Recommendations based on our findings and subsequent analysis are provided below.

- **Allow `xBalance`, `yBalance`, and `totalSupply` to be set to any value in the range of (`MINIMUM_BALANCE`, `MAXIMUM_BALANCE`).** Currently, those balances are hard-coded in the fuzz tests, with the exception of those in the `LifecycleTest` contract. This limits the fuzzer's exploration of the input space and prevents it from identifying novel edge cases.

- **Add fuzz tests for constant sum curve slices.** The current fuzz tests are constrained to a bonding curve that contains only constant product slices. Fuzz testing of deposit, withdrawal, and swap operations that traverse constant sum slices would identify system properties that may not hold in those operations.

- **Test the bonding curves with more than five slices.** The `Setup` contract, which is inherited by all testing contracts except for the `LifecycleTest` contract, creates an identity curve with five slices. All fuzz tests, including the ones in the `LifecycleTest` contract, use this bonding curve to test a myriad of operations and system properties. However, since traversal of additional slices will introduce additional error-prone arithmetic (e.g., integer division), testing the bonding curves with various numbers of slices may lead to additional violations of the system properties.

- **Uncomment the `asset=true` statement in the `Util` contract's `testDepositInput` function.** The boolean value of `asset` determines which token will be deposited into a pool. Disallowing a boolean value of `false` limits the fuzzer's analysis to only 50% of the input space.

- **Run the tests in the `LifecycleTest` contract with Echidna to assess their coverage.** The `LifecycleTest` contract fuzz tests are the most intricate ones in the test suite, and proper execution of those tests will provide strong guarantees about the performance of the Proteus engine. However, time constraints prevented us from running those tests with Echidna. Since Foundry does not offer coverage

support, running the tests with Echidna will help identify how much of the Proteus contract's code is traversed by the fuzzer.

General Recommendations

- **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.
- **Add comprehensive logging mechanisms to all fuzz tests to aid in debugging.** Logging 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 or operation that caused the failure.
- **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.

D. Incident Response Plan Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
    - Consider documenting a plan of action for handling failed remediations.
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which Cowri Labs will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific components of the system.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers, etc.) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop "muscle memory." Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

E. Code Quality Recommendations

This appendix lists code quality findings that are not associated with specific vulnerabilities.

Ocean

- **Use `amount > 0` instead of `amount != 0` (since `amount` is of the `uint256` type).**

```
function _grantFeeToOcean(uint256 oceanId, uint256 amount) private {
    if (amount != 0) {
        _mintWithoutSafeTransferAcceptanceCheck(owner(), oceanId, amount);
    }
}
```

Figure E.1: The `_grantFeeToOcean` function in `contracts/Ocean.sol#L1294-1298`

- **Consider using the same scheme to generate every `oceanId`.** For example, use `HASH(TokenType, externalContract, ID|0)`. This will prevent `oceanId` collisions and increase the code's readability.