Security Assessment Report

Berally Smart Contracts Audit

February 12, 2025

Version 0.1

sub7

Contents

1 Confidentiality statement

2 Disclaimer

3 About Sub7

Founded in 2022, Sub7 Security is a pioneering cybersecurity company dedicated to creating innovative and efficient solutions for a secure digital future. In 2023, we established our presence in Luxembourg after a successful pitch of our cutting-edge platform, SecHub. Our solutions earned us the prestigious recognition of being named one of the top three cybersecurity solutions in Luxembourg, further cementing our position as a leader in the field.

We are the creators of SecHub, a transformative platform designed to deliver fast, transparent, and secure cybersecurity management. SecHub empowers clients by providing direct access to top-tier security researchers, real-time findings, and rapid issue resolution. Our services include smart contract audits, penetration testing, and a range of tailored security solutions. By significantly reducing audit timelines while

maintaining robust security, SecHub is revolutionizing the way organizations manage their digital risks.

Our team brings together a wealth of expertise spanning banking, finance, cybersecurity, law, and business management. We also work closely with a dedicated lawyer, ensuring compliance with regulatory standards and providing a comprehensive approach to security and legal considerations.

Our team members have professional experience at leading organizations such as ServiceNow, Polygon, Tokeny, Bitstamp, Kreditech, Mash, Deloitte, and Bitflyer, bringing invaluable insights from diverse industries.

At Sub7 Security, we are committed to innovation, trust, and resilience. By combining cutting-edge technology, industry expertise, and legal acumen, we deliver solutions that empower businesses and build a safer, more secure digital ecosystem.

Join us as we shape the future of cybersecurity.

4 Project Overview

Berally is a social trading platform for AI Agents on Berachain, designed to help users create, own, and profit from their agents efforts.

5 Executive Summary

Sub7 Security has been engaged to what is formally referred to as a Security Audit of Solidity Smart Contracts, a combination of automated and manual assessments in search for vulnerabilities, bugs, unintended outputs, among others inside deployed Smart Contracts.

The goal of such a Security Audit is to assess project code (with any associated specification, and documentation) and provide our clients with a report of potential security-related issues that should be addressed to improve security posture, decrease attack surface and mitigate risk.

As well general recommendations around the methodology and usability of the related project are also included during this activity

1 (One) Security Auditors/Consultants were engaged in this activity.

5.1 Scope

(1) berally pots smartcontracts

(2) berally passes smartcontracts

(3) berally staking smartcontract

5.2 Timeline

15 January 2025 to 10 February 2025

5.3 Summary of Findings Identified

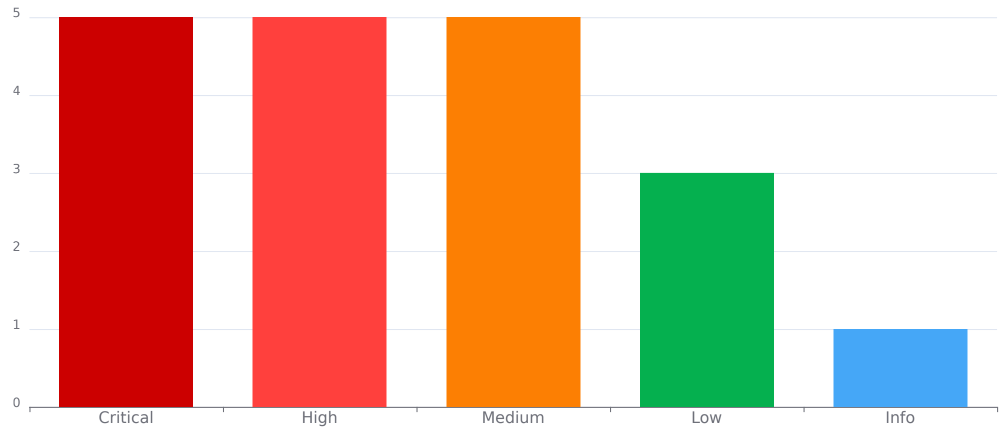

Figure 1: Executive Summary

# 1 Critical Recursive Fee Generation Leads to Unfair Manager Profit on Treasury Withdrawals – ***Fixed***

# 2 Critical Platform Fee Incorrectly Charged on Balance Changes Rather Than True Profit – ***Fixed***

# 3 Critical Partial Protocol Fees Can Be Stolen By Anyone Setting Themselves as Referrer – ***Fixed***

# 4 Critical Frontrunning Vulnerability in Signature-Based Pot Creation – ***Fixed***

# 5 Critical Users Can Block Pot Closure by Sending LP Tokens Directly to Contract - Critical – ***Fixed***

# 6 High Silent Transfer Failures and Zero-Transfer Reverts in Withdrawal function Lead to Permanent Fund Loss or Complete Withdrawal DOS – ***Fixed***

# 7 High Malicious Manager Can Block All Pass Sales By Reverting Fee Transfer – ***Fixed***

# 8 High Cross-Chain Replay and Nonce Vulnerability in Pot Creation – ***Fixed***

# 9 High Unbounded BERP Order Iteration Can Block Withdrawals and Closures Through Gas-Intensive Value Calculations – ***Fixed***

# 10 High BerpGuard Integration Incompatible with Protocol's Smart Contract Design – ***Fixed***

# 11 Medium Treasury Unable to Fully Withdraw Their Assets Using the Shares it Gets in Form of Fee. – ***Fixed***

# 12 Medium Manager Multi-Pass Initial Purchase Reverts Due to Underflow in Price Calculation – ***Fixed***

# 13 Medium Missing Maximum Price Check in buyPasses Leads to Undesired Purchase Price – ***Fixed***

# 14 Medium High Gas Consumption in removeLot Can Block User's Own Sales Due to Many Small Lot Purchases – ***Fixed***

# 15 Medium Use of Deprecated transfer for ETH Sends Can Lead to Failed Withdrawals – ***Fixed***

# 16 Low Hardcoded BerachainRewardsVaultFactory Address Creates Deployment Risk – ***Fixed***

# 17 Low getAddress Function Uses Non-Initialized Mapping From Governance – ***Fixed***

# 18 Low Dangerously High Maximum Performance Fee Could Lead to Value Extraction – ***Fixed***

# 19 Info Unnecessary Fee Calculation and Transfer Attempts for Manager's Free Initial Pass – ***Fixed***

5.4 Methodology

SUB7's audit methodology involves a combination of different assessments that are performed to the provided code, including but not limited to the following:

Specification Check

Manual assessment of the assets, where they are held, who are the actors, privileges of actors, who is allowed to access what and when, trust relationships, threat model, potential attack vectors, scenarios, and mitigations. Well-specified code with standards such as NatSpec is expected to save time.

Documentation Review

Manual review of all and any documentation available, allowing our auditors to save time in inferring the architecture of the project, contract interactions, program constraints, asset flow, actors, threat model, and risk mitigation measures

Automated Assessments

The provided code is submitted via a series of carefully selected tools to automatically determine if the code produces the expected outputs, attempt to highlight possible vulnerabilities within non-running code (Static Analysis), and providing invalid, unexpected, and/or random data as inputs to a running code, looking for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.

Examples of such tools are Slither, MythX, 4naly3er, Sstan, Natspec-smells, and custom bots built by partners that are actively competing in Code4rena bot races.

Manual Assessments

Manual review of the code in a line-by-line fashion is the only way today to infer and evaluate business logic and application-level constraints which is where a majority of the serious vulnerabilities are being found. This intensive assessment will check business logics, intended functionality, access control & authorization issues, oracle issues, manipulation attempts and multiple others.

Security Consultants make use of checklists such as SCSVS, Solcurity, and their custom notes to ensure every attack vector possible is covered as part of the assessment

6 Findings and Risk Analysis

6.1 Recursive Fee Generation Leads to Unfair Manager Profit on Treasury Withdrawals

Severity: Critical
Status: Fixed

Description

There's a sneaky fee problem happening when the treasury tries to withdraw their earned fees. Here's how it plays out:

Say a user withdraws 100,000 shares: * 3,000 shares (3%) go to treasury as exit fee * 2,000 shares (2%) go to manager as management fee * 10,000 shares (10%) go to manager as performance fee * User gets the remaining 85,000 shares

But when treasury tries to withdraw their 3,000 shares, they get hit with fees again: * Another 60 shares (2%) to manager * Another 300 shares (10%) to manager * Treasury only gets 2,640 shares

The manager is basically getting paid twice on the same money - first when the user withdraws, then again when treasury tries to get their cut. Pretty unfair setup.

6.2 Code Location

```
function withdraw(uint256 _shareAmount, uint256 _minSharePrice) external {
    // Manager gets fees on everything except their own withdrawals
    if(msg.sender != manager) {  // Notice: no check for treasury
        managerExitFee = _shareAmount*_managerExitFeeNumerator/_managerExitFeeDenominator;
        if(profitPercentage > 0) {
            performanceFee = (_shareAmount*profitPercentage/(10**18))*
                _performanceFeeNumerator/_performanceFeeDenominator;
        }
    }
}
```

Location

https://github.com/0xnirlin/sub7/tree/main/berraly

Recommendation

Simple fix - don't charge manager fees when treasury withdraws:

```
if(msg.sender != manager && msg.sender != protocolInfo.treasury()) {
    // Apply manager fees only on regular user withdrawals
    managerExitFee = _shareAmount*_managerExitFeeNumerator/_managerExitFeeDenominator;
    if(profitPercentage > 0) {
        performanceFee = (_shareAmount*profitPercentage/(10**18))*_performanceFeeNumerator
            /_performanceFeeDenominator;
    }
}
```

Comments

6.3 Platform Fee Incorrectly Charged on Balance Changes Rather Than True Profit

Severity: Critical
Status: Fixed

Description

In the execute function, the contract charges platform fees based purely on token balance increases, without considering the overall portfolio value. This flawed logic means fees are charged on any positive balance change, not just actual profits.

Current implementation treats any of these as "profit": - Token swaps (1000 USDC → 1000 USDT) If I swap 1000 USDC → 1000USDT: Before balances: 1000 USDC, 0 USDT After balances: 0 USDC, 1000 USDT Fee is deducted even though overall value remains same. - Moving liquidity between protocols - Receiving tokens in exchange for other assets - Asset rebalancing operations - Any operation where one token's balance increases, regardless of overall portfolio value

The flawed check:

```
if(afterAssets[i] == beforeAssets[i] && afterBalances[i] > beforeBalances[i]) {
    // Charges fee on any balance increase
    uint256 fee = (afterBalances[i] - beforeBalances[i])*_feeNumerator/_feeDenominator;
    if(fee > 0) {
        IERC20(afterAssets[i]).transfer(treasury, fee);
    }
}
```

This means users get unfairly charged fees even when there's no actual profit, just asset recomposition. In some cases, fees might even be charged when the overall portfolio value has decreased.

6.4 Code Location

```
function execute(address _to, bytes calldata _data) external {
    // ...
    if(isPlatform) {
        if(_feeNumerator > 0) {
            for(uint256 i = 0; i < afterAssets.length; i++) {
                if(afterAssets[i] == beforeAssets[i] &&
                   afterBalances[i] > beforeBalances[i]) {
                    uint256 fee = (afterBalances[i] - beforeBalances[i])
```

```
                            *_feeNumerator/_feeDenominator;
                        // Charges fee on simple balance increases
                    }
                }
            }
        }
    }
```

Location

https://github.com/0xnirlin/sub7/tree/main/berraly

Recommendation

Charge fee based on overall value change not a certain asset balance change.

Comments

6.5 Partial Protocol Fees Can Be Stolen By Anyone Setting Themselves as Referrer

Severity: Critical
Status: Fixed

Description

The `buyPasses` and `sellPasses` functions allow anyone to take a cut of protocol's fees by simply setting themselves as a referrer. The referral fee is directly deducted from protocol's revenue without any verification:

```
if(referral != address(0)) {
    referralFee = price * referralFeePercent / 1 ether;
    protocolFee -= referralFee;  // Takes away from protocol's revenue
}
```

Example exploit: 1. User buying/selling passes for 10 ETH 2. Protocol fee is 0.5 ETH (5%) 3. User sets themselves as referrer 4. Gets 0.1 ETH (1%) cut from protocol's share 5. Protocol only gets 0.4 ETH instead of 0.5 ETH

6.6 Code Location

```
// In both buyPasses and sellPasses:
if(referral != address(0)) {
    referralFee = price * referralFeePercent / 1 ether;
    protocolFee -= referralFee;  // Direct theft from protocol
}

if(referralFee > 0) {
    (success4, ) = referral.call{value: referralFee}("");
}
```

Location

main/contracts/Passes.sol

Recommendation

Options to fix: 1. Implement on-chain whitelisted referral system

```
mapping(address => bool) public isReferrer;

function registerReferrer(address referrer) external onlyOwner {
    isReferrer[referrer] = true;
}

function buyPasses(...) {
    if(referral != address(0)) {
        require(isReferrer[referral], "Invalid referrer");
        // ... rest of code
    }
}
```

2. Using zk is also an option but will have to wait for Brevis implementation on Berachain before enabling referral system

Comments

6.7 Frontrunning Vulnerability in Signature-Based Pot Creation

Severity: Critical
Status: Fixed

Description

Vulnerable Code

```
function createPot(
    string calldata _name,
    string calldata _symbol,
    uint256 _performanceFee,
    string[] calldata _platforms,
    uint256 _id,
    bytes calldata _signature,
    uint8 _potType
) external payable whenNotPaused returns (address pot) {
    if(potIds[_id]) revert("Invalid Id");
    potIds[_id] = true;

    if(
        ECDSA.recover(
            MessageHashUtils.toEthSignedMessageHash(
                getCreatePotDigest(_id, _potType)
            ),
            _signature
        ) != creator
    ) revert("Invalid Signature");

    // Rest of the function...
}
```

How the Vulnerability Works

1. **Signature Reuse**: The `_signature` is derived from a digest of `_id` and `_potType`. Since these parameters are predictable, an attacker can:
   - Observe a pending transaction containing the signed message.
   - Extract the signature and use it in their own transaction with the same `_id` and `_potType`.
   -

6.8 Frontrun the original transaction to create the pot first.

6.8.1 3. Exploit Scenario

1. **User A** prepares a transaction to create a pot with `_id = 123` and `_potType = 1`, signing the message with their private key.
2. **User A** broadcasts the transaction to the network.
3. **Attacker B** monitors the mempool and extracts the signature from User A's transaction.
4. **Attacker B** submits their own transaction with the same `_id`, `_potType`, and signature but with a higher gas price.
5. **Attacker B**'s transaction is mined first, creating the pot and consuming the `_id`.
6. **User A**'s transaction fails because `potIds[123]` is already set to `true`.

Location

main/contracts/PotFactory.sol

Recommendation

6.8.2 4. Recommendations

Fix 1: Include `msg.sender` in the Signed Message

- Modify the `getCreatePotDigest` function to include the `msg.sender` address. This ensures the signature is only valid for the intended sender.

```
function getCreatePotDigest(
    uint256 _id,
    uint8 _potType,
    address _sender
) public view returns (bytes32) {
    return
        _hashTypedDataV4(
            keccak256(
                abi.encode(
                    keccak256("CreatePot"),
                    _id,
                    _potType,
                    _sender
                )
            )
        );
}
```

- Update the `createPot` function to pass `msg.sender` to `getCreatePotDigest`.

Comments

6.9 Users Can Block Pot Closure by Sending LP Tokens Directly to Contract - Critical

Severity: Critical
Status: Fixed

Description

The `close()` function in NonFlexiblePot contains a critical vulnerability where any user can prevent the pot from being closed by transferring LP tokens directly to the contract. The function reverts if there is any LP token balance:

```
function close() external onlyManager whenNotPaused {
    // Get all current asset balances
    (address[] memory assets, uint256[] memory balances) = getAssetComposition();

    // Check for LP tokens
    for (uint256 i = 0; i < assets.length; i++) {
        uint16 assetType = getAssetType(assets[i]);
        if(
            balances[i] > 0 && (
                assetType == uint16(IWhitelistedTokens.TokenType.Liquidity)
                || assetType == uint16(IWhitelistedTokens.TokenType.BexLiquidity)
            )
        ) revert("Can't withdraw");
    }
}
```

The balance check flows through:

```
getAssetComposition() ->
```

```
assetBalance(asset) ->
guard.getBalance() ->
ERC20Guard.getBalance() ->
IERC20(asset).balanceOf(pot)
```

This means: 1. Anyone can send LP tokens directly to pot 2. `balanceOf` will show this balance 3. `close()` will revert on LP balance check 4. Manager can never close the pot 5. Can also happen unintentionally due to: - Dust amounts from rounding - Failed transactions leaving small balances - Direct token transfers

6.10 Code Location

```
// NonFlexiblePot.sol
function close() external onlyManager {
    (address[] memory assets, uint256[] memory balances) = getAssetComposition();
    for (uint256 i = 0; i < assets.length; i++) {
        if(balances[i] > 0 && isLPToken(assets[i])) revert("Can't withdraw");
    }
}

// Via inheritance chain to ERC20Guard
function getBalance(address pot, address asset) returns (uint256) {
    return IERC20(asset).balanceOf(pot);  // Checks direct balance
}
```

Location

https://github.com/berally/berally-pots-smartcontract

Recommendation

1. Only check LP balances that were acquired through official pot functions aka do internal tracking istead of relying on balance of

Comments

6.11 Silent Transfer Failures and Zero-Transfer Reverts in Withdrawal function Lead to Permanent Fund Loss or Complete Withdrawal DOS

Severity: High
Status: Fixed

Description

The withdrawal implementation has several problems in how it handles ERC20 token transfers:

1. Silent Transfer Failures = Permanent Loss:

```
IERC20(asset).transfer(msg.sender, amountOut);  // Doesn't check return value
```

Not all ERC20s follow the standard strictly: - Some tokens return `false` on failure instead of reverting - Contract doesn't check the return value, assumes success - Example: Early versions of ZRX token return `false` on failure - Result: Transfer fails silently, but code continues as if successful

2. Zero-Transfer Reverts = Complete DOS:

```
uint256 amountOut = balances[i]*portion/(10**18);
if(amountOut > 0) {
    IERC20(asset).transfer(msg.sender, amountOut);
}
```

- Tokens like USDT revert on zero-amount transfers
- If such a token is in pot with tiny balance = whole function reverts
- Everyone's withdrawals blocked until this token removed
- No way to execute partial withdrawals

What makes it worse: - Shares get burned after the loop regardless of transfer success - Event shows tokens as withdrawn even if they're still in contract - Both failed states are permanent with no recovery path

6.12 Code Location

```
function withdraw(uint256 _shareAmount, uint256 _minSharePrice) external {
    // ...
    for (uint256 i = 0; i < assets.length; i++) {
        uint256 amountOut = balances[i]*portion/(10**18);
        if(amountOut > 0) {
            IERC20(asset).transfer(msg.sender, amountOut);  // No return check
            withdrawnAssets[index] = WithdrawnAsset({...}); // Recorded even if failed
        }
    }
    _burn(msg.sender, _shareAmount);  // Burns regardless of transfer success
}
```

Location

https://github.com/berally/berally-pots-smartcontract/tree/main/contracts/pots

Recommendation

1. Use OpenZeppelin's SafeERC20 to handle the silent failure case.
2. Add zero address checks.

Comments

6.13 Malicious Manager Can Block All Pass Sales By Reverting Fee Transfer

Severity: High
Status: Fixed

Description

In the `sellPasses` function, if a manager is a contract is set to revert ETH transfers in it's receive or fallback function, all pass sales will fail since manager fee transfer is required for withdrawal completion:

```
function sellPasses(address manager, uint256 amount, uint256 minPrice, address referral)
    public {
    // ... code ...

    // If this transfer fails, entire transaction reverts
    (bool success3, ) = manager.call{value: managerFee}("");

    // All transfers must succeed
    require(success1 && success2 && success3 && success4, "Unable to send funds");
}
```

A malicious manager can: 1. Deploy contract as manager that reverts ETH transfers 2. Let users buy passes normally 3. When users try to sell, manager's contract reverts ETH transfer 4. Users are unable to sell/exit their positions

6.14 Code Location

```
// In sellPasses:
(bool success1, ) = msg.sender.call{value: price - protocolFee - managerFee - referralFee
    }("");
(bool success2, ) = treasury.call{value: protocolFee}("");
(bool success3, ) = manager.call{value: managerFee}("");  // Can block all sales
require(success1 && success2 && success3 && success4, "Unable to send funds");
```

Location

main/contracts/Passes.sol

Recommendation

Implement a pull-based fee system for manager fees:

```
mapping(address => uint256) public managerFees;

function sellPasses(...) {
    // Instead of direct transfer
    managerFees[manager] += managerFee;
}

function withdrawManagerFees() external {
    uint256 fees = managerFees[msg.sender];
    managerFees[msg.sender] = 0;
    (bool success,) = msg.sender.call{value: fees}("");
    require(success, "Fee transfer failed");
}
```

Comments

6.15 Cross-Chain Replay and Nonce Vulnerability in Pot Creation

Severity: High
Status: Fixed

Description

6.15.1 1. Vulnerability Overview

- **Issue**: The `createPot` function is vulnerable to **cross-chain replay attacks** and **signature reuse** due to the lack of `chainId` in the signed message.
- **Impact**: An attacker can reuse the same signature to create pots on multiple chains

---

6.15.2 2. Technical Analysis

Vulnerable Code

```
function getCreatePotDigest(
    uint256 _id,
    uint8 _potType
) public view returns (bytes32) {
    return
        _hashTypedDataV4(
            keccak256(
                abi.encode(
                    keccak256("CreatePot"),
                    _id,
                    _potType
                )
            )
        );
}
```

How the Vulnerability Works

1. **Cross-Chain Replay**:

- The signed message does not include the `chainId`, making it valid across multiple chains.
- An attacker can use the same signature to create pots on different chains (e.g., Ethereum mainnet and berachainn).

Location

main/contracts/PotFactory.sol

Recommendation

6.15.3 3. Recommendations

Fix 1: Include `chainId` in the Signed Message

- Add the `chainId` to the signed message to prevent cross-chain replay attacks.

```
function getCreatePotDigest(
    uint256 _id,
    uint8 _potType,
    uint256 _chainId
) public view returns (bytes32) {
    return
        _hashTypedDataV4(
            keccak256(
                abi.encode(
                    keccak256("CreatePot"),
                    _id,
                    _potType,
                    _chainId
                )
            )
        );
}
```

Comments

6.16 Unbounded BERP Order Iteration Can Block Withdrawals and Closures Through Gas-Intensive Value Calculations

Severity: High
Status: Fixed

Description

Let me trace the exact flow that makes this vulnerability serious:

1. User/Manager calls important functions:

```
// NonFlexiblePot.sol
function withdraw() or function close()
```

2. These functions need to get fund value, so they call:

```
getFundComposition() in Pot.sol
```

3. `getFundComposition` needs each asset's value, so for each asset it calls:

```
assetValue(asset, balances[i])
```

4. For BERP assets, this leads to:

```
calcValue() in BerpAssetGuard
-> _calcValue()
```

5. Finally hits the problematic loop:

```
// BerpAssetGuard.sol
function _calcValue(address pot) private view {
    Order[] memory orders = orderTracker.getAllOrders(pot);
    for(uint i = 0; i < orders.length; i++) {
        // External calls for EACH order:
        berpOrders.getOpenTrade(order.id);
        // or
        berpOrders.getOpenLimitOrder(order.id);
    }
}
```

Attack Scenario: 1. Manager opens many small perpetual trades on BERP 2. Each trade is tracked by `orderTracker` 3. When users try to withdraw or manager tries to close: - Must loop through all trades - Make external calls for each trade - Gas cost becomes astronomical - Transaction reverts due to gas limit 4. Users' funds are effectively locked 5. Manager can't close the pot

Why this is dangerous: - Each withdraw/close attempt must perform external calls for every BERP order - No limit on number of orders - Each order requires 1-2 external calls to load data - Gas cost increases linearly with order count - Eventually hits block gas limit - Makes withdraw/close impossible

Location

https://github.com/berally/berally-pots-smartcontract

Recommendation

6.17 Recommendation

1. Implement strict order limit:

```
uint256 public constant MAX_ORDERS = 100;

function _calcValue(address pot) private view {
    Order[] memory orders = orderTracker.getAllOrders(pot);
    require(orders.length <= MAX_ORDERS, "Too many orders");
    // Continue with safe number of orders
}
```

Comments

6.18 BerpGuard Integration Incompatible with Protocol’s Smart Contract Design

Severity: High

Status: Fixed

Description

The BerpGuard integration in the protocol is fundamentally flawed because Berp’s platform only allows EOA (Externally Owned Accounts) to interact with it, while the protocol’s design relies on smart contract interactions. This incompatibility makes the entire BerpGuard integration non-functional and potentially misleading to users.

In the BerpGuard contract:

```
function txGuard(address _pot, address _to, bytes memory _data)
    public override returns (uint16 txType)
{
    txType = _txGuard(_pot, _to, _data, 0);
    return txType;
}
```

The guard validates and allows transactions from smart contract addresses (_pot), but these transactions will ultimately fail on Berp’s platform since it only accepts EOA interactions.

Code Location The issue spans across two primary locations:

1. NonFlexiblePot.sol:

```
function execute(address _to, bytes calldata _data, uint256 _nativeTokenAmount)
    external
    onlyManager
    nonReentrant
    whenNotPaused
    returns (bool success)
{
    return _execute(_to, _data, _nativeTokenAmount);
}
```

2. BerpGuard.sol:

```
function _txGuard(address _pot, address _to, bytes memory _data, uint256
    _nativeTokenAmount)
    internal returns (uint16 txType)
{
    // ... validation logic ...
    if (method == IEntrypoint.openTrade.selector) {
        // Allows contract interactions that will fail on Berp
    }
}
```

Impact 1. All transactions attempting to interact with Berp through the protocol will fail 2. Gas costs wasted on failed transactions 3. Potential loss of trading opportunities 4. Misleading protocol functionality that appears valid but can never succeed

Location

https://github.com/berally/berally-pots-smartcontract

Recommendation

There are several possible approaches to address this:

1. Remove Berp integration entirely if it cannot support smart contract interactions:
2. Work with Berp to support smart contract interactions before implementing the integration

Comments

6.19 Treasury Unable to Fully Withdraw Their Assets Using the Shares it Gets in Form of Fee.

Severity: Medium
Status: Fixed

Description

When a regular user withdraws from the pot: 1. They are charged an exit fee that goes to the treasury 2. Their remaining shares are burned and they get their assets

When the treasury tries to withdraw these fee shares: 1. They try to withdraw 100 shares 2. The contract charges them an exit fee (let's say 10 shares) 3. These 10 shares get sent back to the treasury 4. Treasury only gets assets for 90 shares 5. Now they need to withdraw those 10 shares, but same thing happens again - they'll get charged fees on their own fees

This creates an endless loop where treasury can never fully exit because each withdrawal creates new fee shares they need to withdraw.

Code Location:

```
// Exit fees always apply, even to treasury
uint256 protocolExitFee = _shareAmount*_protocolExitFeeNumerator/
    _protocolExitFeeDenominator;

// No checks, fees go right back to treasury
if(protocolExitFee > 0) transfer(protocolInfo.treasury(), protocolExitFee);
```

Adding this as medium and not high since treasury can still claim big chunk by looping into withdraw but not full amount still.

Location

https://github.com/berally/berally-pots-smartcontract/tree/main/contracts/pots

Recommendation

Skip exit fees when treasury is withdrawing:

```
if(msg.sender != protocolInfo.treasury()) {
    if(protocolExitFee > 0) transfer(protocolInfo.treasury(), protocolExitFee);
}
```

Comments

6.20 Manager Multi-Pass Initial Purchase Reverts Due to Underflow in Price Calculation

Severity: Medium
Status: Fixed

Description

When a manager makes their first purchase (supply = 0) with amount > 1, the price calculation reverts due to underflow. Here's exactly where and how:

```
function getPrice(uint256 supply, uint256 amount, uint256 factor) public pure returns (
    uint256) {
    // Calculate sum1
    // When supply = 0, this is fine because of the ternary
    uint256 sum1 = supply == 0 ? 0 : (supply - 1)* (supply) * (2 * (supply - 1) + 1) / 6;

    // Here's where it underflows
    // When supply = 0 and amount > 1, the ternary's condition is false
    // So it tries to calculate: (supply - 1 + amount)...
    // Which means: (0 - 1 + amount)
    // This underflows at (0 - 1)
    uint256 sum2 = supply == 0 && amount == 1 ? 0 :
        (supply - 1 + amount) * (supply + amount) * (2 * (supply - 1 + amount) + 1) / 6;
```

For example, if manager tries to buy 2 passes initially: 1. `supply = 0` 2. `amount = 2` 3. `supply == 0 && amount == 1` is false 4. So it executes: `(0 - 1 + 2)* (0 + 2)* (2 * (0 - 1 + 2)+ 1)/ 6` 5. The `(0 - 1)` happens first, causing an underflow revert

Location

main/contracts/Passes.sol

Recommendation

Add explicit check in `buyPasses`:

```
if(supply == 0 && manager == msg.sender) {
    require(amount == 1, "First purchase must be exactly 1 pass");
    require(defaultFactors[factor], "Invalid factor value");
    factors[manager] = factor;
}
```

Comments

6.21 Missing Maximum Price Check in buyPasses Leads to Undesired Purchase Price

Severity: Medium
Status: Fixed

Description

The `buyPasses` function accepts excess ETH and refunds the difference, but lacks a maximum price check. This means users can unintentionally pay a much higher price than intended:

```
function buyPasses(address manager, uint256 amount, uint256 factor, address referral)
    public payable {
    // Price can be different than what user calculated off-chain
    uint256 price = getPrice(supply, amount, factor);

    uint256 protocolFee = price * protocolFeePercentage / 1 ether;
    uint256 managerFee = price * managerFeePercentage / 1 ether;

    // User might end up paying much more than expected
    uint256 refundedAmount = msg.value - (price + protocolFee + managerFee);
    if(refundedAmount > 0) {
        (success4, ) = msg.sender.call{value: refundedAmount}("");
    }
```

Example scenario: 1. User calculates expected price = 1 ETH 2. Sends 2 ETH to cover any small changes 3. Actual price ends up being 1.9 ETH due to market changes 4. Transaction succeeds, only 0.1 ETH refunded 5. User paid 1.9 ETH when they wanted max 1 ETH

Location

main/contracts/Passes.sol

Recommendation

Add maximum price parameter to allow users to specify their maximum acceptable price:

```
function buyPasses(
    address manager,
    uint256 amount,
    uint256 factor,
    address referral,
    uint256 maxPrice
) public payable {
    // ...
    uint256 price = getPrice(supply, amount, factor);
    uint256 totalPrice = price + (price * protocolFeePercentage / 1 ether) +
        (price * managerFeePercentage / 1 ether);
    require(totalPrice <= maxPrice, "Price higher than user's maximum");
    // ...
}
```

Comments

6.22 High Gas Consumption in removeLot Can Block User's Own Sales Due to Many Small Lot Purchases

Severity: Medium
Status: Fixed

Description

The `removeLot` function inefficiently removes elements by shifting all subsequent elements left, causing high gas costs that could prevent users from selling their passes if they've accumulated many small lots:

```
function removeLot(address user, address subject) internal {
    // Each iteration moves a struct - very gas intensive
    for (uint i = 0; i < shareLots[user][subject].length - 1; i++) {
        shareLots[user][subject][i] = shareLots[user][subject][i + 1];
    }
    shareLots[user][subject].pop();
}
```

This affects users who: 1. Make many small purchases over time 2. Each purchase creates a new lot entry 3. When selling, must process each lot 4. Gas costs multiply with array size 5. Could unintentionally make their position unsellable

6.23 Code Location

```
// In sellPasses - loops through lots
while (sharesToSell > 0) {
    if (lot.shares <= sharesToSell) {
        removeLot(msg.sender, manager);  // Gas-heavy for users with many lots
    }
}
```

Adding this as medium because the probability of this happening is low but the impact is high.

Location

main/contracts/Passes.sol

Recommendation

Still thinking about what would be the ideal fix/

Comments

6.24 Use of Deprecated transfer for ETH Sends Can Lead to Failed Withdrawals

Severity: Medium
Status: Fixed

Description

The PotFactory's `withdraw` function uses the deprecated `transfer()` method to send ETH instead of the recommended `call()`. This can cause withdrawals to permanently fail if the receiver has a gas-heavy fallback/receive function:

```
function withdraw(
    address receiver,
    bool isBera,
    address token,
    uint256 amount
) external onlyOwner returns (bool) {
    if (isBera) {
        payable(receiver).transfer(amount);  // Hard 2300 gas limit
    } else {
        IERC20(token).transfer(receiver, amount);
    }
    return true;
}
```

The `transfer()` method: 1. Has a hard 2300 gas limit 2. Will revert if receiver needs more than 2300 gas 3. Makes assumptions about fallback function gas costs 4. Can make withdrawals impossible if receiver is a contract that: - Has gas-heavy fallback/receive function - Needs to do accounting or emit events - Implements complex token acceptance logic

6.25 Code Location

```
function withdraw(address receiver, bool isBera, address token, uint256 amount) external
    onlyOwner {
    if (isBera) {
        payable(receiver).transfer(amount);  // Can brick withdrawals
    }
}
```

Location

https://github.com/berally/berally-pots-smartcontract

Recommendation

Use `call()` with value and check return success:

```
function withdraw(address receiver, bool isBera, address token, uint256 amount) external
    onlyOwner {
    if (isBera) {
        (bool success,) = payable(receiver).call{value: amount}("");
        require(success, "ETH transfer failed");
    }
}
```

Comments

6.26 Hardcoded BerachainRewardsVaultFactory Address Creates Deployment Risk

Severity: Low
Status: Fixed

Description

The contract hardcodes the BerachainRewardsVaultFactory address during initialization:

```
address vaultFactory = 0x2B6e40f65D82A0cB98795bC7587a71bfa49fBB2B;
stakingToken = new PassesStakingToken(address(this));
polVault = IBerachainRewardsVault(
    IBerachainRewardsVaultFactory(vaultFactory).createRewardsVault(
        address(stakingToken)
    )
);
```

This is problematic because: 1. Berachain is not live yet, so the actual factory address could be different on mainnet 2. If deployed on different networks/environments (testnet vs mainnet), the hardcoded address won't work 3. Future upgrades to Berachain infrastructure might change this address 4. Contract deployment will fail if the hardcoded address doesn't match the actual factory address

6.27 Code Location

```
// we are hard coding the vault factory address, since berachain is not live yet
address vaultFactory = 0x2B6e40f65D82A0cB98795bC7587a71bfa49fBB2B;
```

Location

main/contracts/Passes.sol

Recommendation

Instead of hardcoding the address: 1. Add a constructor parameter or initializer function parameter:

```
function initialize(address _vaultFactory) public initializer {
    __Ownable_init();
    treasury = owner();

    stakingToken = new PassesStakingToken(address(this));
    polVault = IBerachainRewardsVault(
        IBerachainRewardsVaultFactory(_vaultFactory).createRewardsVault(
            address(stakingToken)
        )
    );
}
```

2. Or make it updatable by admin:

```
address public vaultFactory;

function setVaultFactory(address _vaultFactory) external onlyOwner {
    vaultFactory = _vaultFactory;
}
```

Comments

6.28 getAddress Function Uses Non-Initialized Mapping From Governance

Severity: Low
Status: Fixed

Description

The PotFactory contract calls `getAddress()` which relies on the `nameToDestination` mapping in the Governance contract, but this mapping is never populated as no functions exist to set values in it:

```
// In PotFactory
function getAddress(bytes32 name) external view override returns (address destination) {
    destination = IGovernance(governanceAddress).nameToDestination(name);
    require(destination != address(0), "governance: invalid name");
}

```

```
// In Governance
mapping(bytes32 => address) public override nameToDestination;
// No functions exist to set values in this mapping
```

This means: 1. All calls to `getAddress()` will return address(0) 2. These calls will always revert due to the zero address check 3. Any contract functionality depending on `getAddress()` will be unusable

6.29 Code Location

```
// In Governance.sol - mapping exists but no setter
mapping(bytes32 => address) public override nameToDestination;

// In PotFactory.sol - function will always revert
function getAddress(bytes32 name) external view override returns (address destination) {
    destination = IGovernance(governanceAddress).nameToDestination(name);
    require(destination != address(0), "governance: invalid name");
}
```

Location

main/contracts/Governance.sol

Recommendation

Add function to set values in the mapping:

```
function setDestination(bytes32 name, address destination) external onlyOwner {
    require(destination != address(0), "Zero address");
    nameToDestination[name] = destination;
    emit DestinationSet(name, destination);
}
```

Comments

6.30 Dangerously High Maximum Performance Fee Could Lead to Value Extraction

Severity: Low
Status: Fixed

Description

The contract sets an extremely high maximum performance fee of 70% in the initialization function:

```
function initialize(
    address _usdToken,
    address _passes,
    address _governanceAddress,
    address _oracleAddress
) public initializer {
    // ...
    _setMaxPerformanceFee(70e18); // 70% performance fee
}
```

This means: 1. Managers could set performance fees up to 70% 2. For every 100 USD of profit, manager could take 70 USD 3. Leaves investors with only 30% of their profits 4. Industry standard performance fees are typically 10-20%

6.31 Code Location

```
function _setMaxPerformanceFee(uint256 _maxPerformanceFee) private {
    if (_maxPerformanceFee > FEE_DENOMINATOR)
        revert("AboveMax");
    maxPerformanceFee = _maxPerformanceFee;
    emit MaxPerformanceFeeChanged(maxPerformanceFee);
}
```

Location

main/contracts/PotFactory.sol

Recommendation

Lower the maximum performance fee to a more reasonable level:

```
_setMaxPerformanceFee(30e18); // 30% maximum performance fee
```

Comments

6.32 Unnecessary Fee Calculation and Transfer Attempts for Manager's Free Initial Pass

Severity: Info
Status: Fixed

Description

When a manager buys their first pass, the price is set to 0, yet the contract still calculates and attempts to transfer protocol and manager fees. This leads to unnecessary gas consumption and potential confusion:

```
uint256 price = getPrice(supply, amount, factor);
// When manager buys first pass, price = 0

// Unnecessary calculations when price = 0
uint256 protocolFee = price * protocolFeePercentage / 1 ether;  // = 0
uint256 managerFee = price * managerFeePercentage / 1 ether;    // = 0

// Unnecessary transfer attempts
(bool success1, ) = treasury.call{value: protocolFee}("");      // Transfers 0
```

```
(bool success2, ) = manager.call{value: managerFee}("");        // Transfers 0
```

These operations are wasteful because: 1. Calculating fees on 0 price is unnecessary 2. Attempting to transfer 0 ETH wastes gas 3. Success checks are performed for zero-value transfers

6.33 Code Location

```
function buyPasses(address manager, uint256 amount, uint256 factor, address referral)
    public payable {
    uint256 supply = passesSupply[manager];
    if(supply == 0 && manager == msg.sender) {
        // First pass by manager
        require(defaultFactors[factor], "Invalid factor value");
        factors[manager] = factor;
    }

    uint256 price = getPrice(supply, amount, factor);  // Returns 0
    // Still calculates and transfers fees even when price = 0
}
```

Location

main/contracts/Passes.sol

Recommendation

Skip fee calculations and transfers for manager's initial pass:

```
function buyPasses(address manager, uint256 amount, uint256 factor, address referral)
    public payable {
    uint256 supply = passesSupply[manager];

    bool isFirstManagerPass = (supply == 0 && manager == msg.sender);
    if(isFirstManagerPass) {
        require(defaultFactors[factor], "Invalid factor value");
        factors[manager] = factor;
    }

    uint256 price = getPrice(supply, amount, factor);

    // Skip fee operations for first manager pass
    if(!isFirstManagerPass) {
        uint256 protocolFee = price * protocolFeePercentage / 1 ether;
        uint256 managerFee = price * managerFeePercentage / 1 ether;
        // ... fee transfers
    }
}
```

Comments

sub7

FOLLOW US