Contents

Wyvern Security Review

Reviewer:
- Stryder

Date: December 24, 2024

---

1. Executive Summary

Over the course of 17 days, **Stryder** conducted a comprehensive security audit of the **Wyvern Protocol**. This review focused on evaluating the protocol's key components for potential vulnerabilities, efficiency, and adherence to best practices in DeFi development.

This specialized review examined core contracts and functionalities that facilitate the protocol's staking, liquidity, and buy-and-burn mechanisms. The contracts reviewed include:

- `WyvernX.sol`
- `TitanBuy.sol`
- `WyvernVault.sol`
- `WyvernBuyAndBurn.sol`
- `WyvernStake.sol`

Summary of Findings

- **Total Issues Found:** 9
- **High Risk Issues:** 2
- **Medium Risk Issues:** 1
- **Low Risk Issues:** 4
- **Gas Optimizations and Informational:** 2

The Wyvern Protocol's codebase demonstrates a robust structure with well-documented components, though areas for improvement were identified and addressed.

Repository Commit

Commit: `e04d05cda298437052ef0ea6110118ca85586e11`

Project Details:
- **Type of Project:** DeFi, Tokenomics, Liquidity Management
- **Timeline:** Dec 7 2024 - Dec 24 2024 - **Methods:** Manual Review and Automated Tools
- **Documentation:** Good
- **Testing Coverage:** Good

---

2. Stryder

Stryder is an independent security auditor specializing in blockchain, smart contract, and Defi systems. With over three years of hands-on experience, Stryder has successfully conducted comprehensive audits for notable projects like Zus Network, Empower Coin, and Moonvera Solutions.

As a recognized expert, Stryder has identified and resolved over 50 critical security issues, ensuring the safety and efficiency of various Web3 protocols. Holding top rankings in global auditing platforms such as Code4rena and Sherlock, Stryder demonstrates an unwavering commitment to maintaining the highest standards of blockchain security.

Stryder's approach integrates cutting-edge tools like Slither, Foundry, and Solidity Metrics to ensure thorough code review, gas optimization, and vulnerability detection. With a solid background in scalable protocol design and real-time data pipelines, Stryder's audits deliver comprehensive assessments and actionable insights tailored to project needs. ### My Approach: - **Thorough Code Review:** Each line of code is meticulously examined to identify vulnerabilities, logic flaws, and deviations from best practices. - **Security Testing:** A focus on identifying attack vectors such as reentrancy, access control issues, and unsafe external interactions. - **Best Practices Compliance:** Ensuring adherence to established smart contract development standards, including gas optimization and upgradeability considerations.

3. Introduction

The Wyvern Protocol is a comprehensive DeFi solution designed to enable efficient staking, liquidity management, and tokenomics on EVM-compatible chains. This review focused on the protocol's unique mechanisms, including the buy-and-burn functionality, staking architecture, and token distribution strategies. The protocol leverages advanced Uniswap V3 integrations and custom liquidity management solutions to optimize user rewards and ensure ecosystem sustainability.

Security Focus Areas

- **Tokenomics Integrity:** Analysis of token-related functions, including minting, burning, and transfers, to ensure they align with the intended tokenomics of the Wyvern ecosystem.
- **Buy-and-Burn Functionality:** Validation of token-burning mechanisms to ensure that no unintended behavior or vulnerabilities arise during the buy-and-burn process.
- **Staking and Rewards Distribution:** Verification of staking mechanisms, including calculations of rewards and their distribution, to prevent misallocation or manipulation.

- **Uniswap V3 Integrations:** Auditing the integration with Uniswap V3 for liquidity management, including fee collection, pool interactions, and TWAP-based pricing mechanisms, to ensure secure and efficient operation.
- **Access Control:** Ensuring that critical functions are protected by robust access controls to prevent unauthorized access or misuse.
- **Reentrancy and External Calls:** Review of external calls within the protocol to mitigate reentrancy risks and ensure secure interactions with external smart contracts.

---

4. Findings

4.1 High Severity

[H-01] Lack of Validation for `Create2` Deployment in `WyvernVault::_deployWyvernStakeInstance`

Severity: High
Location: WyvernVault.sol

Description The `_deployWyvernStakeInstance` function does not validate the output of the `Create2` deployment. If the contract deployment fails, the function will add a `0x0` address to the `wyvernStakeContracts` mapping and set it as the `activeWyvernStakeContract`. This results in the following issues: 1. Any subsequent operations iterating over the `wyvernStakeContracts` mapping will fail due to the presence of invalid addresses. 2. Critical functions like staking and claiming ETH rewards will revert when interacting with the `0x0` address.

Impact

- **Denial of Service (DOS):** Function like `claim` depending on the `wyvernStakeContracts` mapping will fail, halting critical functionalities.
- **Inconsistent State:** Invalid contract addresses (`0x0`) in the mapping and as the `activeWyvernStakeContract` can lead to state inconsistencies and reverts.

Recommendation

1. Validate the result of the `Create2.deploy` function to ensure the contract is successfully deployed.
2. Revert the transaction if the deployment fails.

Example mitigation:

```
function _deployWyvernStakeInstance() private {
    // Deploy an instance of Wyvern staking contract
    bytes memory bytecode = abi.encodePacked(type(WyvernStake).creationCode, abi.encode(addr
    uint256 stakeContractId = numWyvernStakeContracts;

    // Create a unique salt for deployment
    bytes32 salt = keccak256(abi.encodePacked(address(this), stakeContractId));

    // Deploy a new WyvernStake contract instance
    address newWyvernStakeContract = Create2.deploy(0, salt, bytecode);

    // Validate deployment
    if (newWyvernStakeContract == address(0)) {
        revert("WyvernStake deployment failed");
    }

    // Set new contract as active
    activeWyvernStakeContract = newWyvernStakeContract;

    // Update storage
    wyvernStakeContracts[stakeContractId] = newWyvernStakeContract;

    // Allow the WyvernStake instance to send ETH to WyvernVault
    _receiveEthAllowList[newWyvernStakeContract] = true;

    // For functions limited to WyvernStake
    _wyvernStakeAllowList[newWyvernStakeContract] = true;

    // Emit an event to track the creation of a new stake contract
    emit WyvernStakeInstanceCreated(stakeContractId, newWyvernStakeContract);

    // Increment the counter for WyvernStake contracts
    numWyvernStakeContracts += 1;
}
```

[H-02] `Create2` Address Collision Vulnerability in `WyvernVault::_deployWyvernStakeInstance`

Severity: High
Location: WyvernVault.sol

Description The `_deployWyvernStakeInstance` function uses the `Create2` opcode to deterministically generate a new contract address for the WyvernStake contract. However, the salt used for this deployment lacks sufficient randomness and is predictable. This makes it vulnerable to an address collision attack, allowing an attacker to precompute an address that matches the expected deployed contract address. The attacker can then deploy a malicious contract at the same address before the intended contract is deployed.

Impact

- **Theft of Funds:** All funds intended for the deployed WyvernStake contract can be drained by the attacker's malicious contract. The attacker can deploy a contract that approves the attackers address with the tokens and then self destruct the contract. This means that when the actual contract will be deployed it can be directly drained of its funds as the attacker's address will have approval of transferring the funds of the contract
- **Denial of Service (DOS):** Functions depending on the deployed WyvernStake contract will fail if the address is occupied by the attacker's malicious contract.

Steps to Reproduce

1. Precompute the address of the WyvernStake contract using the deterministic `Create2` address calculation.
2. Deploy a malicious contract at the precomputed address using the same salt and bytecode structure.
3. When the legitimate `_deployWyvernStakeInstance` function is called, the deployment will fail as the address is already occupied.
4. Any funds sent to the expected contract address are intercepted and can be drained by the attacker.

Proof of Concept (PoC)

1. Calculate the expected address using `Create2`:

```
function _deployWyvernStakeInstance() private {
      // Deploy an instance of Wyvern staking contract
      // bytes memory bytecode = type(WyvernStake).creationCode;
      // Prepare the bytecode with constructor argument
      bytes memory bytecode = abi.encodePacked(type(WyvernStake).creationCode, abi.encode(
      uint256 stakeContractId = numWyvernStakeContracts;

      // Create a unique salt for deployment
      bytes32 salt = keccak256(
          abi.encodePacked(address(this), stakeContractId)
      );
```

```
        // Deploy a new WyvernStake contract instance
        address newWyvernStakeContract = Create2.deploy(0, salt, bytecode);

        // Set new contract as active
        activeWyvernStakeContract = newWyvernStakeContract;

        // Update storage
        wyvernStakeContracts[stakeContractId] = newWyvernStakeContract;

        // Allow the WyvernStake instance to send ETH to WyvernVault
        _receiveEthAllowList[newWyvernStakeContract] = true;

        // For functions limited to WyvernStake
        _wyvernStakeAllowList[newWyvernStakeContract] = true;

        // Emit an event to track the creation of a new stake contract
        emit WyvernStakeInstanceCreated(
            stakeContractId,
            newWyvernStakeContract
        );

        // Increment the counter for WyvernStake contracts
        numWyvernStakeContracts += 1;
    }
```

4.2 Medium Severity

[M-01] Vulnerable `TitanBuy::getTitanQuoteForEth` Function Allows Price Manipulation via Low Cardinality in Uniswap V3 Pools

Severity: High
Location: `getTitanQuoteForEth` function

Description The `getTitanQuoteForEth` function in the contract is vulnerable to price manipulation in Uniswap V3 pools with low observation cardinality or insufficient observation history. This vulnerability arises due to the following behaviors:

1. **Low Cardinality Issue:**
    - If the pool's cardinality is low (e.g., initialized to 1), the function falls back to the most recent observation as the oldest observation. A malicious actor can manipulate the pool price by front-running a swap in the same block, writing a manipulated price into the pool's observation slots.
2. **Shortened TWAP Window:**

    - The function adjusts the requested TWAP window (`secondsAgo`) to the available observation history (`oldestObservation`). If the pool lacks sufficient history, the TWAP window is shortened, making the price calculation vulnerable to manipulation.
3. **Current Price Fallback:**
    - If `secondsAgo` equals `0`, the function uses the current spot price (`slot0`). Spot prices can be easily manipulated by a single transaction.

Impact

- **Price Manipulation:**
    - Attackers can manipulate the Uniswap V3 pool price by performing a front-running transaction, leading to inaccurate token quotes.
    - This can result in the protocol or users overpaying or underpaying for swaps.
- **Increased Slippage Risk:**
    - Shortened TWAP windows are less robust against manipulation, exposing the protocol to volatile and manipulated prices.

Steps to Reproduce

1. **Setup:**
    - Deploy a new Uniswap V3 pool with low cardinality (e.g., 1).
    - Set up the contract using this pool.
2. **Attack Execution:**
    - An attacker front-runs the `getTitanQuoteForEth` function with a swap that manipulates the pool's price.
    - The manipulated price is written into the observation slots.
3. **Result:**
    - The function retrieves the manipulated price as either:
        - The TWAP (if `secondsAgo` falls back to `oldestObservation`).
        - The spot price (if `secondsAgo` is `0`).
4. **Consequence:**
    - The contract calculates an inaccurate quote based on the manipulated price.

Recommendation

1. **Validate Cardinality:**
    - Ensure the pool's cardinality is sufficient to cover the requested TWAP window.
    - Example:

```
uint16 cardinality = IUniswapV3Pool(poolAddress).slot0().observationCardinalityNext
require(cardinality >= MIN_CARDINALITY, "Insufficient cardinality");
```

2. **Minimum TWAP Window:**
   - Enforce a minimum TWAP window, and revert if the pool cannot satisfy it.
3. **Increase Cardinality:**
   - Call `increaseObservationCardinalityNext` on the pool during initialization to increase the pool's observation slots.
4. **Spot Price Safeguards:**
   - Avoid relying on the spot price (`slot0`) as a fallback unless robust safeguards are implemented.
5. **Alternative Price Feeds:**
   - Use decentralized oracles (e.g., Chainlink) as a backup mechanism when Uniswap TWAP is unavailable or unreliable.
6. **Reverting incase of Low Cardinality:**
   - Otherwise reverting if the cardinality is low or if `oldestObservation < secondsAgo` to ensure the full intended TWAP period is used.

Example Mitigation

```
function getTitanQuoteForEth(uint256 baseAmount) public view returns (uint256 quote) {
    address poolAddress = PoolAddress.computeAddress(
        UNI_FACTORY,
        PoolAddress.getPoolKey(WETH9_ADDRESS, TITANX_ADDRESS, FEE_TIER)
    );

    uint32 secondsAgo = _titanPriceTwa * 60;
    uint32 oldestObservation = OracleLibrary.getOldestObservationSecondsAgo(poolAddress);

    // Validate sufficient cardinality
    uint16 cardinality = IUniswapV3Pool(poolAddress).slot0().observationCardinalityNext;
    require(cardinality > MIN_CARDINALITY, "Insufficient cardinality");

    if (oldestObservation < secondsAgo) {
        secondsAgo = oldestObservation;
    }

    uint160 sqrtPriceX96;
    if (secondsAgo == 0) {
        revert("Fallback to spot price is unsafe");
    } else {
        (int24 arithmeticMeanTick, ) = OracleLibrary.consult(poolAddress, secondsAgo);
        sqrtPriceX96 = TickMath.getSqrtRatioAtTick(arithmeticMeanTick);
    }
    return OracleLibrary.getQuoteForSqrtRatioX96(
        sqrtPriceX96,
        baseAmount,
        WETH9_ADDRESS,
        TITANX_ADDRESS
    );
}
```

4.3 Low Severity

[L-01] Inconsistent Documentation for `TitanBuy::setSlippage` Function

Severity: Low
Location: `TitanBuy.sol`

Description The `setSlippage` function's documentation specifies that the allowable range for slippage is from `0-50`. However, the function enforces a stricter range of `5-15` using the `require` statement. This discrepancy between the documentation and implementation may lead to confusion for developers or users reviewing the contract.

Impact

- Misleading documentation could cause developers to assume broader flexibility for the `slippage` parameter than is actually permitted by the contract.

Recommendation

- Update the documentation to correctly reflect the enforced range (`5-15% only`).

Proof of Concept The `setSlippage` function enforces the range of `5-15` through a `require` statement:

```
* @notice set slippage % for buynburn minimum received amount. Only callable by owner addres
* @param amount amount from 0 - 50
function setSlippage(uint256 amount) external onlyOwner {
    require(amount >= 5 && amount <= 15, "5-15% only");
    slippage = amount;
}
```

[L-02] Minting Can Proceed After the Minting Phase Has Ended in `WyvernX::startMint` function

Severity: Low
Location: WyvernX.sol

Description In the `startMint` function, the condition checking if the minting phase has ended (`s_mintingPhaseFinished == WyvernMintingPhaseFinished.YES`) is placed before updating the `s_mintingPhaseFinished` variable. This allows a scenario where the last user in the minting phase can proceed with minting, as the `s_mintingPhaseFinished` flag is updated only after the check. This logic creates an unintended behavior where minting is allowed even if the minting phase is technically over.

Impact A user could exploit this sequence to perform one additional mint after the minting phase has ended. While this issue is low severity, it undermines the intended behavior of the minting phase mechanism and could lead to inconsistencies in the minting process.

Recommendation Move the update of `s_mintingPhaseFinished` to occur before the check on `s_mintingPhaseFinished`. This ensures the minting phase state is finalized before any minting actions can occur.

Proof of Concept Current implementation:

```
if (s_mintingPhaseFinished == WyvernMintingPhaseFinished.YES) {
    revert Wyvern_MintingPhaseFinished();
}

if (activeSharesPercentage >= MINIMUM_SHAREPOOL_FOR_MINTING_PHASE) {
    // finish minting phase
    s_mintingPhaseFinished = WyvernMintingPhaseFinished.YES;
}
```

Similar Findings https://github.com/devdomsos/wyvernX-smart-contract-audit/blob/e04d05cda298437052ef0ea6110118ca85586e11/packages/hardhat/contracts/WyvernX.sol#L261-L309

[L-03] `WyvernVault::stake` Allows First Stake Without Sufficient Vault Funds or Enforcing Cooldown

Severity: Low
Location: `stake` function

Description In the `stake` function, the first stake can be initiated without ensuring that the `vault` has the necessary funds to meet the required threshold (`TITANX_BPB_MAX_TITAN`). Additionally, the cooldown mechanism (`nextStakeTs`) is bypassed during the first call to the function. This introduces the risk of staking being initiated prematurely, without adequate funds or cooldown enforcement.

Impact

- The first stake can occur even if the vault does not have sufficient TitanX tokens to maximize the "bigger pays better" bonus.
- The cooldown period meant to regulate staking frequency is not enforced for the first staking transaction.

Recommendation

- Validate that the `vault` has sufficient TitanX tokens during the first staking operation. Add a condition to enforce the cooldown even for the initial stake. For example:

```
function stake() external {
    WyvernStake wyvernStake = WyvernStake(
        payable(activeWyvernStakeContract)
    );

    if (wyvernStake.openedStakes() >= TITANX_MAX_STAKE_PER_WALLET) {
        revert NoAdditionalStakesAllowed();
    }

    updateVault();

    uint256 vault_ = vault;

    // Ensure sufficient TitanX in the vault for the first stake
    if (nextStakeTs == 0 && vault_ < TITANX_BPB_MAX_TITAN) {
        revert NoTokensToStake();
    }
    if (vault_ >= TITANX_BPB_MAX_TITAN) {
        _startTitanXStake();
        nextStakeTs = block.timestamp + 7 days;
    } else {
        // Enforce cooldown even for the first stake
        if (block.timestamp < nextStakeTs) {
            revert CooldownPeriodActive();
        }

        _startTitanXStake();
        nextStakeTs = block.timestamp + 7 days;
    }
}
```

[L-04] Ineffective Protection Against Bot Abuse in `claim`

Severity: Low

Description The `claim` function employs the `msg.sender != tx.origin` check to prevent bots from interacting with the function. However, this mechanism is insufficient to block bots that use EOAs (Externally Owned Accounts) to sign transactions, as these bots can effectively bypass the restriction.

Impact

- **Bot Exploitation:** Malicious bots leveraging EOAs to sign transactions can repeatedly call the `claim` function, gaining an unfair advantage over regular users.
- **Economic Impact:** Legitimate users may miss out on fair rewards due to the exploitative actions of bots.

Recommendation To mitigate this issue, consider implementing a **commit-reveal scheme** or similar mechanism to make bot abuse significantly more challenging:
1. **Commit Phase:** Users submit a hash of their intent (e.g., `keccak256(userAddress, salt)`).
2. **Reveal Phase:** After a specified delay, users reveal their commitment (e.g., address and salt).
3. The function verifies the revealed values against the commitment before allowing execution.

This approach introduces unpredictability, making it harder for bots to gain an edge while maintaining fairness for legitimate users.

Example snippet for commit-reveal structure:

```
mapping(address => bytes32) private commitments;

function commit(bytes32 commitment) external {
    commitments[msg.sender] = commitment;
}

function reveal(uint256 salt) external {
    bytes32 expectedCommitment = keccak256(abi.encodePacked(msg.sender, salt));
    require(commitments[msg.sender] == expectedCommitment, "Invalid commitment");
    delete commitments[msg.sender];
    // Proceed with the `claim` functionality here
}
```

4.4 Gas Optimizations and Informational

[G-01] Use Assembly for Null Address Checks

Severity: Gas Optimization
Location: Constructor

Description The constructor currently performs null address checks using `if` conditions. While these checks are functional, they can be optimized for gas efficiency by leveraging inline assembly. Inline assembly provides a direct and cheaper method to validate addresses, reducing the overall gas cost of the constructor execution.

Recommendation Replace the `if` conditions for null address checks with inline assembly as shown below:

```
constructor(
    address titanBuyAddress_,
    address wyvernBuyAndBurnAdddress_,
    address legacyAddress_
) Ownable(msg.sender) {
assembly {
    if iszero(titanBuyAddress_) {
        mstore(0x00, 0x12345678) // Replace 0x12345678 with keccak256 hash of "InvalidAddres
        revert(0x00, 0x04)
    }
    if iszero(wyvernBuyAndBurnAdddress_) {
        mstore(0x00, 0x12345678) // Replace 0x12345678 with keccak256 hash of "InvalidAddres
        revert(0x00, 0x04)
    }
    if iszero(legacyAddress_) {
        mstore(0x00, 0x12345678) // Replace 0x12345678 with keccak256 hash of "InvalidAddres
        revert(0x00, 0x04)
    }
}

}
```

[I-01] Potential State Manipulation by `dailyDifficultyClock` Modifier

Severity: Low
Location: `GlobalInfo.sol`

Description The `dailyDifficultyClock` modifier in the `GlobalInfo` contract is responsible for automatically updating the contract's internal state variables whenever a function using this modifier is called. However, it directly invokes the `_dailyDifficultyClock` function, which alters multiple state variables. This behavior can lead to unintended state changes when the modifier is applied to multiple external or public functions.

Impact

- Modifiers generally should avoid invoking state-altering functions directly, as this can complicate the understanding of the contract's behavior and increase the risk of bugs.

Recommendation

- Avoid calling state-altering functions directly within modifiers. Instead, explicitly call `_dailyDifficultyClock` within the individual external or public functions where needed.

Proof of Concept The `dailyDifficultyClock` modifier directly calls `_dailyDifficultyClock`:

```
modifier dailyDifficultyClock() {
    _dailyDifficultyClock();
    _;
}
```

[I-02] Missing Events for Critical Updates

Severity: Informational

Description The contract lacks event emissions for critical updates to state variables, such as changes to the `wyvernAddress`, `wyvernVaultAddress`, or other configurations like `capPerSwap` and `slippage`. This omission reduces the ability to track and audit these changes off-chain, potentially decreasing transparency for users and auditors.

Impact

- **Reduced Transparency:**
  It becomes challenging to track changes to critical parameters, especially when these updates are performed by the owner or during a contract upgrade.

- **Audit Difficulty:**
  Lack of event logs hinders off-chain monitoring and auditing of contract activity.

Recommendation Emit events in all setter functions to log updates to critical state variables. For example:

```
event WyvernAddressUpdated(address indexed oldAddress, address indexed newAddress);

function setWyvernContractAddress(address wyvernAddress_) external onlyOwner {
    if (wyvernAddress_ == address(0)) {
        revert InvalidWyvernAddress();
    }
    emit WyvernAddressUpdated(wyvernAddress, wyvernAddress_);
    wyvernAddress = wyvernAddress_;
}
```