WyvernX 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 **WyvernX 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`
- `HuntersGuild`
- `TitanBuyer.sol`
- `WyvernVault.sol`
- `WyvernBuyAndBurn.sol`
- `WyvernStaker.sol`

Summary of Findings

- **Total Issues Found:** 5
- **High Risk Issues:** 2
- **Low Risk Issues:** 1
- **Informational:** 2

The WyvernX 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. Stryder's approach integrates cutting-edge tools like Slither, Foundry, and Solidity Metrics to ensure thorough code review, gas optimization, and vulnerability detection.

3. Introduction

WyvernX is a comprehensive DeFi protocol designed to enable efficient staking within WyvernX and TitanX protocols, liquidity management, and tokenomics on Ethereum. 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 WyvernX 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

Severity: High
Location: `WyvernVault.sol` **Function:** `_createTitanStaker`

Description

The `_createTitanStaker` function does not validate the output of the `Create2` deployment. If the contract deployment fails, the function will add a `0x0` address to the `titanStakerContracts` mapping and set it as the `activeTitanStakerContract`. This results in the following issues:

1. Any subsequent operations iterating over the `titanStakerContracts` 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 `titanStakerContracts` mapping will fail, halting critical functionalities.
- **Inconsistent State:** Invalid contract addresses (`0x0`) in the mapping and as the `activeTitanStakerContract` 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 _createTitanStaker() private {

    ...code

    // Validate deployment
    if (newTitanStakerContract == address(0)) {
        revert("TitanStaker deployment failed");
    }

    // Set new contract as active
    activeTitanStakerContract = newTitanStakerContract;

    ...rest of the code
}
```

Action Taken

1. **Fixed**. Commit `e3b11c01c7bfad343880cd9f2fc2d500fb0a9f17`

[H-02] `Create2` Address Collision Vulnerability

Severity: High
Location: `WyvernVault.sol` **Function:** `_createTitanStaker`

Description

The `_createTitanStaker` function uses the `Create2` opcode to deterministically generate a new contract address for the TitanStaker 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 TitanStaker 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 TitanStaker contract will fail if the address is occupied by the attacker's malicious contract.

Steps to Reproduce

1. Precompute the address of the TitanStaker 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 `_createTitanStaker` 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 _createTitanStaker() private {

      ...previous code

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

      address precomputedAddress = Create2.deploy(0, salt, bytecode);

      ...rest of the code
  }
```

Action Taken

1. **Fixed**. Commit `785ade7b39b4897251f77943e8627357a355212a`

4.3 Low Severity

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

Severity: Low
Location: `TitanBuyer.sol` **Function** `setSlippage`

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:

```
function setSlippage(uint256 amount) external onlyOwner {
    require(amount >= 5 && amount <= 15, "5-15% only");
    slippage = amount;
}
```

Action Taken

1. **Fixed**. Commit `85dfe8c8186b1f0bc0f67d8addd4a0574c86dfb0`

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

Severity: Low
Location: `HuntersGuild.sol` **Function**: `increaseDifficultyLevel`

Description

The `increaseDifficultyLevel` modifier in the `HuntersGuild` 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 `_increaseDifficultyLevel` 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 `_increaseDifficultyLevel` within the individual external or public functions where needed.

Proof of Concept

The `increaseDifficultyLevel` modifier directly calls `_increaseDifficultyLevel` :

```
modifier increaseDifficultyLevel() {
    _increaseDifficultyLevel();
    _;
}
```

Action Taken

1. **Acknowledged.**

[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 `capPerBuy` 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_;
}
```

Action Taken

1. **Acknowledged.**
