shieldify

Pudgy Strategy

Igloo Protocol

SECURITY REVIEW

Date: 21 October 2025

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Rust, Go, Vyper, Move and Cairo.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Pudgy Strategy - Igloo Protocol

Igloo Protocol is a multi-project ecosystem built on AbstractChain, centred around the concept of strategy tokens for NFT collections. Each strategy token applies a small trading tax to automatically buy and sell NFTs from its respective collection, while a portion of every transaction also feeds back into our ecosystem token, $PDGYSTR, through a buy-and-burn loop.

Our first strategy token is HEROSTR, created for the OnChain Heroes (OCH) collection.

This audit focuses on two contracts that form the foundation of this model:

- `HEROSTR.sol` — ERC20 token with configurable taxes, swapback logic, and a timed tax decay system.
- `NFTVault Final.sol` — Vault that receives tax revenue to buy/sell NFTs and execute burn logic.

We are seeking to ensure these contracts are secure, simple, and optimized before expanding the Igloo Protocol ecosystem to additional NFT collections.

4. Risk Classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** - results in a significant risk for the protocol's overall well-being. Affects all or most users

- **Medium** - results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** - losses will be limited but bearable - and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** - almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** - still relatively likely, although only conditionally possible
- **Low** - requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 6 days with a total of 96 hours dedicated by 2 researchers from the Shieldify team.

Overall, the code is well-written. The audit report identified eight Low severity findings, mainly related to missetting of the progressive buy cap, potential block of user sells, potentially stuck ETH and others.

The Pudgy Strategy team has done a great job with the development and has been highly responsive to the Shieldify research team's inquiries and promptly implemented all recommendations.

5.1 Protocol Summary

Project Name	**Pudgy Strategy - Igloo Protocol**
Repository	HEROSTR
Type of Project	**ERC20, NFT Vaults**
Security Review Timeline	**6 days**
Review Commit Hash	cafe69ed029aef74e874471b157afabfacb0d81c
Fixes Review Commit Hash	5c70d874f372d5b85615c425a12df5a99429812d

5.2 Scope

The following smart contracts were in the scope of the security review:

File	**nSLOC**
NFTVault Final.sol	120
HEROSTR.sol	418
Total	**538**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Low** issues: **8**
- **Info** issues: **6**
- **Gas** issues: **1**

ID	Title	Severity	Status
[L-01]	Progressive Buy Cap Does Not Reach Intended 50M at Minute 50	Low	Fixed
[L-02]	Malicious Fee Recipients Can Block User Sells	Low	Acknowledged
[L-03]	The `unwrapAllWETH()` Function Creates Stuck ETH With No Withdrawal Path	Low	Fixed
[L-04]	Controller Can Buy Arbitrary NFTs and Steal OCH Via Approvals	Low	Partially Fixed
[L-05]	The `setRouter()` Function Does Not Set `isCapExempt[r]`	Low	Fixed
[L-06]	The `userStatsFor()` Function Returns `cap = 0` After `minute > 50`	Low	Fixed
[L-07]	The `manualSwapback()` Function Callable By the Owner/SwapperBot Even If `swapEnabled` Is False	Low	Acknowledged
[L-08]	No Way to Recover Locked Tokens in the Vault	Low	Fixed
[I-01]	Enforce Constraints for `swapThreshold` and `maxSwapAmount` in `setSwapBehavior()` Function	Info	Fixed
[I-02]	Newly Set Recipients Are Not Auto-(Un)Exempted	Info	Acknowledged
[I-03]	Max Wallet Enforced Only on Buys and Can Be Trivially Bypassed via Peer Transfers	Info	Acknowledged
[I-04]	Swap Deadline Computed at Execution Always Passes	Info	Acknowledged
[I-05]	Keeper Docs Collection Address Differs from Contract Immutable	Info	Fixed
[I-06]	Misleading Accounting: "Sale Proceeds" Actually Tracks Burned ETH Amount	Info	Fixed
[G-01]	Router Allowance Checked on Every Swap Consumes Gas	Gas	Fixed

7. Findings

[L-01] Progressive Buy Cap Does Not Reach Intended 50M at Minute 50

Severity

Low Risk

Description

The contract intends to ramp cumulative per-address buy caps from 200k to 50M tokens over 50 minutes. The constructor computes `slope` assuming 49 incremental steps (triangular number 1225) and includes 49 fixed 200k steps, but `cumulativeCapFor` uses `k = minute - 2`, which yields only 48 steps by minute 50 (triangular number 1176) and 48 fixed 200k steps. This step-count mismatch causes the minute-50 cap to undershoot the intended 50M target by approximately 1.8M tokens.

> • Progressive buy caps: 200k -> 50M tokens over 50 minutes

File: HEROSTR.sol#L19

Location of Affected Code

File: HEROSTR.sol#L221-L227

```
constructor(
    address _router,
    address _nftVault,
    address _pdgystrAccumulator,
    address _teamWallet
) ERC20("Hero Strategy", "HEROSTR") Ownable(msg.sender) {
  // code
  // Calculate cap (5% of supply)
  capTokens = (_totalSupply * 500) / BPS;

  // Calculate slope for linear growth (minutes 2-50)
  uint256 baseSum = (400_000 + (49 * 200_000)) * 1e18; // 10.2M
  uint256 numer = capTokens - baseSum;
  slope = numer / 1225;
  // code
}
```

File: HEROSTR.sol#L278-L284

```
function cumulativeCapFor(address user, uint256 minute) public view
   returns (uint256) {
  // code
  // cumulativeCapFor (minutes 2-50)
  uint256 k = minute - 2;
  uint256 extraTokens = (k * 200_000e18); // produces 48 steps by minute
     50
  uint256 slopeContribution = (slope * k * (k + 1)) / 2; // 1176 when k
     =48
  baseCap = DELTA0 + DELTA1 + extraTokens + slopeContribution;
  // code
}
```

Impact

At minute 50, the computed cap is approximately 48.2M instead of 50M, preventing users from reaching the documented cap during the final minute of the ramp. This leads to unexpected

`ExceedsCap` reverts for buyers nearing the limit, undermines the published tokenomics, and poses short-lived fairness and reputational risks until caps lift after minute 50.

Proof of Concept

Calculation at minute 50 (as of now)

```
uint256 k = minute - 2; // 48
uint256 extraTokens = (k * 200_000e18); // 9,600,000
uint256 slopeContribution = (slope * k * (k + 1)) / 2; // ~32,489 * 48 *
   49 / 2 = 38,207,064
baseCap = DELTA0 + DELTA1 + extraTokens + slopeContribution; // 400,000 +
    9,600,000 + 38,207,064 = 48,207,064
```

Calculation at minute 50 (after proposed fix applied)

```
uint256 k = minute - 1; // 49
uint256 extraTokens = (k * 200_000e18); // 9,800,000
uint256 slopeContribution = (slope * k * (k + 1)) / 2; // ~32,489 * 49 *
   50 / 2 = 39,799,025
baseCap = DELTA0 + DELTA1 + extraTokens + slopeContribution; // 400,000 +
    9,800,000 + 39,799,025 = 49,999,025
```

The current implementation caps the amount to ~48.2M, while with the proposed fix, it correctly reaches 50M (with the only discrepancy being due to integer truncation caused by slope).

Recommendation

Consider applying the following recommendations:
- Change the implementation to use `k = minute - 1` for minutes 2, so minute 50 yields 49 steps.
- If exact equality at minute 50 is required, consider rounding `slope` upward or clamping the minute-50 cap to the intended target to counter integer division truncation.

Team Response

Status: FIXED

We have identified and corrected this mathematical inconsistency in the cap calculation. The fix has been implemented by changing `k = minute - 2` to `k = minute - 1` in the `cumulativeCapFor` function to ensure the intended 50M cap is reached at minute 50 as documented.

Implementation:

```
// Fixed in cumulativeCapFor function
uint256 k = minute - 1; // Changed from minute - 2
```

[L-02] Malicious Fee Recipients Can Block User Sells

Severity

Low Risk

Description

During the sell pre-swap path, the contract distributes received ETH to three recipient addresses using low-level value calls. If any recipient implements a gas-hungry fallback function, it can deliberately exhaust gas and cause the entire user to sell to revert, creating a censorship/DoS vector. This attack can be triggered either by the owner setting malicious recipient addresses via `setRecipients()` , or by recipients themselves implementing gas-griefing fallbacks. The vulnerability only affects sell transactions, buy transactions continue to work normally as they do not trigger the pre-swap path.

Location of Affected Code

File: HEROSTR.sol#L368-L380

```
function _update( address from, address to, uint256 amount ) internal
    override {
   // code
   // sell pre-swap trigger
   if (isSell && swapEnabled && swapLastTaxOnly && !inSwap) {
       uint256 bal = balanceOf(address(this));
       uint256 toSwap = queuedToSwap;
       if (toSwap > bal) toSwap = bal;
       if (toSwap > maxSwapAmount) toSwap = maxSwapAmount;
       if (toSwap >= swapThreshold && toSwap > 0) {
           uint256 sold = _swapBackExact(toSwap);
           if (sold >= queuedToSwap) queuedToSwap = 0; else queuedToSwap
              -= sold;
       }
   }
   // code
}
```

File: HEROSTR.sol#L491-L498

```
function _swapBackExact(uint256 tokensToSwap) private returns (uint256
    sold) {
   // code
   // ETH distribution to external recipients during the same tx
   if (ethReceived > 0) {
       uint256 vaultAmount = (ethReceived * vaultShare) / 100;
       uint256 accumulatorAmount = (ethReceived * accumulatorShare) / 100;
       uint256 teamAmount = ethReceived - vaultAmount - accumulatorAmount;

       (bool s1,) = nftVault.call{value: vaultAmount}("");
       (bool s2,) = pdgystrAccumulator.call{value: accumulatorAmount}("");
       (bool s3,) = teamWallet.call{value: teamAmount}("");
   }
   // code
}
```

Impact

Owners or fee recipients can induce sell censorship by implementing gas-griefing fallbacks in recipient contracts, causing user sell transactions to revert when pre-swap distribution executes. This blocks users from selling tokens for ETH but does not affect buy transactions, which continue to work normally - creating a honeypot scenario where users can enter positions but cannot exit. The attack can be initiated either by owners setting malicious recipients or by recipients themselves implementing gas-hungry fallbacks, making the severity appropriately Low.

Recommendation

Restrict gas usage in external value calls to prevent gas griefing attacks while maintaining push-based distribution. Update the distribution logic to specify a reasonable gas limit:

```
// ETH distribution with gas limit to prevent griefing
if (ethReceived > 0) {
    uint256 vaultAmount = (ethReceived * vaultShare) / 100;
    uint256 accumulatorAmount = (ethReceived * accumulatorShare) / 100;
    uint256 teamAmount = ethReceived - vaultAmount - accumulatorAmount;

    // Use gas limit to prevent gas exhaustion attacks
    // 100,000 gas is sufficient for most payable fallbacks but prevents
       griefing
    (bool s1,) = nftVault.call{gas: 100_000, value: vaultAmount}("");
    (bool s2,) = pdgystrAccumulator.call{gas: 100_000, value:
       accumulatorAmount}("");
    (bool s3,) = teamWallet.call{gas: 100_000, value: teamAmount}("");
}
```

This approach limits the gas available to each recipient's fallback function, preventing deliberate gas exhaustion while still allowing legitimate payable operations. If a call fails due to gas limitations, it should still be tracked in pending balances for manual recovery. Consider making the gas limit configurable by the owner if different recipients require varying amounts.

Team Response

Status: ACKNOWLEDGED - NO ACTION REQUIRED

The fee recipients in our system are controlled contracts within our ecosystem (NFT Vault and PDGYSTR Accumulator). These contracts are designed to be compatible with the current implementation and do not implement gas-griefing fallback functions. The team has verified that both recipient contracts handle ETH transfers properly without excessive gas consumption. Since we control all recipient addresses, this attack vector is not applicable to our deployment.

[L-03] The `unwrapAllWETH()` Function Creates Stuck ETH With No Withdrawal Path

Severity

Low Risk

Description

The `unwrapAllWETH()` function allows the owner to convert WETH to ETH, but the resulting ETH has no dedicated withdrawal mechanism and will accumulate in the contract balance with no way to withdraw it.

Location of Affected Code

File: HEROSTR.sol#L678-L681

```
function unwrapAllWETH() external onlyOwner {
    uint256 b = IWETH(_WETH()).balanceOf(address(this));
    if (b > 0) IWETH(_WETH()).withdraw(b);
}
```

Impact

The function serves no useful purpose and creates guaranteed stuck ETH.

Recommendation

Remove the `unwrapAllWETH()` function entirely. WETH tokens stuck in the contract can already be rescued using the existing `rescueTokens()` function, allowing the owner to unwrap them externally if needed. This eliminates the risk of creating stuck ETH while maintaining the ability to recover WETH assets.

Team Response

Status: FIXED

Agreed. This function serves no useful purpose and creates unnecessary risk. The team has removed the `unwrapAllWETH()` function entirely from the contract, as WETH can be recovered through the existing `rescueTokens()` mechanism if needed.

[L-04] Controller Can Buy Arbitrary NFTs and Steal OCH Via Approvals

Severity

Low Risk

Description

1. The purchase path accepts opaque Seaport calldata and only checks the function selector, without enforcing that acquired assets belong to `COLLECTION`, and it uses explicit ETH forwarding with no contract-side spend cap (`valueWei` is fully chosen by the controller).
2. The approval path allows the controller to designate an arbitrary operator for any `nft`, enabling transfer of all vault-held tokens of that collection, including OCH, by an attacker-controlled operator.

These behaviors contradict stated assurances in the documentation that the executor cannot drain the vault and the vault cannot buy arbitrary NFTs outside the OCH collection.

Relevant documentation statements:

> - Executor cannot drain vault funds.
> - ...
> - Vault cannot buy arbitrary NFTs outside OCH collection.

Location of Affected Code

File: NFTVault%20Final.sol#L155

```
function buyViaSeaport(bytes calldata data, uint256 valueWei) external
    onlyController nonReentrant {
     require(address(this).balance >= valueWei, "insufficient vault ETH");
     require(data.length >= 4, "calldata too short");
     require(bytes4(data[0:4]) == FULFILL_ADVANCED_SELECTOR, "bad seaport
        selector");

     uint256 ethBefore = address(this).balance;

     (bool success, bytes memory returnData) = SEAPORT.call{value:
        valueWei}(data);
     require(success, _getRevertMsg(returnData, "seaport call failed"));

     uint256 ethSpent = ethBefore - address(this).balance;
     totalETHSpent += ethSpent;
     totalNFTsBought++;
     emit NFTBought(0, ethSpent, bytes32(0));
}
```

File: NFTVault%20Final.sol#L184-L191

```
function setApprovalForCollection(address nft, address operator, bool
    approved) external onlyController nonReentrant {
     IERC721(nft).setApprovalForAll(operator, approved);
     emit ApprovalSet(nft, operator, approved);
}
```

Impact

1. The controller can drain vault ETH by fulfilling arbitrary Seaport orders, including buying non-OCH or worthless assets at inflated prices, because the function neither validates the collection against `COLLECTION` nor enforces any contract-side max spend; it relies on explicit ETH forwarding where `valueWei` is fully chosen by the controller.
2. The controller can steal OCH NFTs held by the vault by setting approval for the OCH collection to an operator they control, then transferring all tokens through that operator.

Recommendation

Consider applying the following changes: - Restrict purchases to the OCH collection and bounded spend:

- Replace opaque `bytes data` with typed parameters (or decode `AdvancedOrder`) and assert that every received ERC721 item has `token` equal to `COLLECTION` and expected `identifier` constraints. - Enforce a configurable per-tx and per-interval max spend and validate `valueWei <= maxSpend` and that consideration recipients are expected marketplace addresses. - Enforce operator and collection allowlists on approvals:
- Remove or restrict `setApprovalForCollection()` to only permit `nft == COLLECTION` and `operator == OPENSEA_CONDUIT`.
- If flexibility is required, gate changes behind an owner-controlled whitelist and cooldown, not the controller.

Team Response

Status: PARTIALLY FIXED

The team acknowledges the security concern regarding arbitrary NFT purchases. We have implemented collection validation to restrict purchases to only the OCH Heroes collection. However, the controller trust model is by design for operational flexibility. The following changes have been implemented:

- Added collection validation in `buyViaSeaport()` to ensure only OCH Heroes NFTs can be purchased
- Restricted `setApprovalForCollection()` to only allow the OCH collection and OpenSea conduit
- Maintained controller trust for operational requirements

Implementation:

```
// Added validation function
function _validateOCHPurchase(bytes calldata data) private view {
    // Validates that OCH collection address appears in Seaport calldata
}

// Updated setApprovalForCollection with restrictions
function setApprovalForCollection(address nft, address operator, bool
   approved) external onlyController {
    require(nft == COLLECTION, "only OCH collection allowed");
    require(operator == OPENSEA_CONDUIT, "only OpenSea conduit allowed");
    // ...
}
```

[L-05] The `setRouter()` Function Does Not Set `isCapExempt[r]`

Severity

Low Risk

Description

Owner can add routers via `setRouter()` , but the function does not mark the router as `isCapExempt` . In constructor the initial router was added to `isCapExempt` but `setRouter()` does not follow that behavior.

Location of Affected Code

File: HEROSTR.sol#L599-L603

```
function setRouter(address r, bool allowed) external onlyOwner {
    require(r != address(0), "zero");
    isRouter[r] = allowed;
    emit RouterSet(r, allowed);
}
```

Impact

If the owner adds an aggregator router that wraps buys, but does not mark it cap-exempt, buys routed through that aggregator might be counted against per-address caps or trigger unexpected cap enforcement.

Recommendation

When setting a router, consider also setting `isCapExempt[r] = true` or require the admin to intentionally mark cap exemption.

Team Response

Status: FIXED

We have implemented the suggested fix to automatically exempt new routers from caps when they are added, while maintaining flexibility for operational requirements.

Implementation:

```
function setRouter(address r, bool allowed) external onlyOwner {
    require(r != address(0), "zero");
    isRouter[r] = allowed;
    if (allowed) {
        isCapExempt[r] = true; // Auto-exempt new routers from caps
    }
    emit RouterSet(r, allowed);
}
```

[L-06] The `userStatsFor()` Function Returns `cap = 0` After `minute > 50`

Severity

Low Risk

Description

If `minutesSince() > 50`, `cumulativeCapFor` returns `type(uint256).max`, but `userStatsFor` maps that to `cap = 0` and `available = 0`. That is misleading: on-chain caps are lifted but this view reports zero available capacity.

Location of Affected Code

File: HEROSTR.sol#L293-L314

```
function userStatsFor(address user) external view returns (
    uint256 minute,
    uint256 cap,
    uint256 bought,
    uint256 available,
    uint256 taxBps
) {
    uint256 m = minutesSince();
    uint256 userCap = cumulativeCapFor(user, m);
    uint256 userBought = totalBought[user];

    minute = m;
    bought = userBought;
    if (userCap == type(uint256).max) {
        cap = 0;
        available = 0;
    } else {
        cap = userCap > bought ? (userCap - bought) : 0;
        available = cap;
    }
    taxBps = currentTaxBps();
}
```

Impact

Front-ends and bots using `userStatsFor` will incorrectly show users they cannot buy more after minute 50, even though on-chain buys are allowed.

Recommendation

Return `cap = type(uint256).max` or return a boolean `capsLifted` so front-ends can correctly interpret.

Team Response

Status: FIXED

We have modified the function to return `cap = type(uint256).max` when caps are lifted after minute 50, providing clearer information to front-ends and users about the unlimited buying capacity.

Implementation:

```
if (userCap == type(uint256).max) {
    cap = type(uint256).max; // Changed from 0
    available = type(uint256).max; // Changed from 0
}
```

[L-07] The `manualSwapback()` Function Callable By the Owner/ SwapperBot Even If `swapEnabled` Is False

Severity

Low Risk

Description

The `manualSwapback()` function can be called by the `owner` or `swapperBot` regardless of `swapEnabled` or `swapActivated` flags. This allows bypassing swap gating.

Location of Affected Code

File: HEROSTR.sol#L644

```
function manualSwapback(uint256 amount) external onlyOwnerOrBot
    nonReentrant {
     require(amount > 0, "amount must be > 0");
     uint256 bal = balanceOf(address(this));
     if (amount > bal) amount = bal;
     if (amount == 0) return;
     _swapBackExact(amount);
}
```

Impact

Owner or bot could manually trigger large swaps even when swapback was supposed to be disabled (a malicious bot/owner can cause unexpected swaps.

Recommendation

Add `require(swapEnabled, "swap disabled");` or `require(swapActivated, "swap not activated");` to `manualSwapback()` function.

Team Response

Status: ACKNOWLEDGED - BY DESIGN

This functionality is intentional. It allows the owner and authorized bots to execute manual swapbacks even when automatic swapping is disabled, providing operational control during edge cases or maintenance periods. This design choice ensures system flexibility while maintaining proper access controls.

[L-08] No Way to Recover Locked Tokens in the Vault

Severity

Low Risk

Description

The vault lacks a mechanism to recover arbitrary ERC-20 tokens accidentally sent to it. Over time, stray tokens may become irrecoverable. For example the vault expects ETH and later swaps `ETH` -> `HEROSTR`. If a sale is settled in WETH (very common) and the marketplace transfers WETH (ERC-20) to the vault, it just sits there. The contract has no WETH unwrap and no generic ERC-20 withdrawal.

Location of Affected Code

File: NFTVault%20Final.sol

Impact

Locked tokens

Recommendation

Add a simple, restricted rescue function:

```
function rescueTokens(address token, uint256 amount, address to) external
    onlyController nonReentrant {
    require(token != HEROSTR, "cannot rescue HEROSTR");
    IERC20(token).transfer(to, amount);
}
```

Team Response

Status: FIXED

While the vault's normal operation only involves ETH and OCH NFTs, we have implemented a basic token recovery function for edge cases where tokens might get accidentally sent to the vault (such as WETH from marketplace settlements).

Implementation:

```
function rescueTokens(address token, uint256 amount, address to) external
    onlyController nonReentrant {
    require(token != HEROSTR, "cannot rescue HEROSTR");
    IERC20(token).transfer(to, amount);
}
```

[I-01] Enforse Constraints for `swapThreshold` and `maxSwapAmount` in `setSwapBehavior()` Function

Severity

Informational Risk

Description

Owner can set `swapThreshold` and `maxSwapAmount` via `setSwapBehavior()` to arbitrary values (no upper/lower bounds), while elsewhere `setSwapThreshold()` has a safe cap.

Location of Affected Code

File: HEROSTR.sol#L623-L633

```
function setSwapBehavior(bool _enabled, bool _lastOnly, uint256
    _threshold, uint256 _max) external onlyOwner {
     swapEnabled = _enabled;
     swapLastTaxOnly = _lastOnly;
     swapThreshold = _threshold;
     maxSwapAmount = _max;
}
```

Recommendation

Add sanity checks similar to `setSwapThreshold()` (e.g., `_threshold <= totalSupply()/100` and `_max <= totalSupply()/100`).

Team Response

Status: FIXED

We have implemented sanity checks for swap parameters to prevent misconfiguration:

Implementation:

```
function setSwapBehavior(bool _enabled, bool _lastOnly, uint256
    _threshold, uint256 _max) external onlyOwner {
     require(_threshold <= totalSupply() / 100, "threshold too high");
     require(_max <= totalSupply() / 100, "max amount too high");
     require(_threshold > 0, "threshold must be > 0");
     require(_max >= _threshold, "max must be >= threshold");

     swapEnabled = _enabled;
     swapLastTaxOnly = _lastOnly;
     swapThreshold = _threshold;
     maxSwapAmount = _max;
}
```

[I-02] Newly Set Recipients Are Not Auto-(Un)Exempted

Severity

Informational Risk

Description

The `setRecipients()` replaces the vault/accumulator/team addresses but does not update `isTaxExempt()` / `isCapExempt()`. Newly set recipients may unexpectedly be taxed or restricted, while the previous recipients remain exempt unless the owner manually updates exemptions.

Location of Affected Code

File: HEROSTR.sol#L659-L671

```
function setRecipients( address _vault, address _accumulator, address
    _team ) external onlyOwner {
     if (_vault == address(0)) revert ZeroAddress();
     if (_accumulator == address(0)) revert ZeroAddress();
     if (_team == address(0)) revert ZeroAddress();

     nftVault = _vault;
     pdgystrAccumulator = _accumulator;
     teamWallet = _team;
}
```

Impact

- Newly set recipients may be taxed or subject to cap restrictions despite the intended exemption policy.
- Previous recipients remain exempt and could receive preferential treatment unintentionally until manually updated.

Recommendation

Update `setRecipients()` to atomically (un)exempt old and new recipients, e.g.: - Clear exemptions for old `nftVault/pdgystrAccumulator/teamWallet` . - Set exemptions for new `_vault/_accumulator/_team`.

Team Response

Status: ACKNOWLEDGED - ACCEPTABLE RISK

The team acknowledges this behavior. Given that recipient changes are rare administrative actions, manual exemption management provides more explicit control over tax and cap exemptions. This approach allows for intentional review of exemption status when recipients are updated.

[I-03] Max Wallet Enforced Only on Buys and Can Be Trivially Bypassed via Peer Transfers

Severity

Informational Risk

Description

The max wallet check is applied only on buy transfers. Direct peer-to-peer transfers to a holder are not subject to this check, allowing users to exceed the configured max wallet via incoming transfers.

Location of Affected Code

File: HEROSTR.sol#L434-L439

```
function _update( address from, address to, uint256 amount ) internal
    override {
    // code
    // Check max wallet after tax
    if (maxWalletEnabled && isBuy && !isCapExempt[to]) {
        uint256 maxWallet = (totalSupply() * maxWalletBps) / BPS;
        if (balanceOf(to) > maxWallet) {
            revert ExceedsMaxWallet();
        }
    }
    // code
}
```

Impact

- Users can bypass max wallet by receiving tokens via direct transfers.
- Without a whitelist mechanism, the number of wallets is effectively unlimited, reducing the practical value of the max wallet feature for distribution control.

Proof of Concept

1. A single user controls two wallets: Wallet A and Wallet B.
2. The user buys up to the max wallet limit into Wallet A.
3. The user buys up to the max wallet limit into Wallet B.
4. The user transfers tokens from Wallet B to Wallet A via a direct peer transfer.
5. Because the max wallet check runs only on buys, the transfer to Wallet A is not checked, Wallet A now holds more than the configured max wallet (effectively up to 2× the limit in this example).

Recommendation

Enforce max wallet on all incoming transfers (buys and peer transfers).

Team Response

Status: ACKNOWLEDGED - BY DESIGN

The current implementation focusing on buy transactions is intentional. Max wallet limits are primarily designed to prevent large single purchases during launch, not to restrict peer-to-peer transfers. Enforcing limits on all transfers would create user experience friction without significant security benefits, as determined users can always use multiple wallets.

[I-04] Swap Deadline Computed at Execution Always Passes

Severity

Informational Risk

Description

The deadline is set as `block.timestamp + 600` at execution time inside the vault. Since the router validates `block.timestamp <= deadline` using the same `block.timestamp`, this check always passes and provides no meaningful timeout. The value is also not caller-controlled, so you cannot enforce stricter or different expiry policies per transaction.

Location of Affected Code

File: NFTVault%20Final.sol#L236-L241

```
function burnProceedsETH(uint256 amountEth, uint256 minOut) external
    onlyController nonReentrant {
  // code
  IUniswapV2Router02(ROUTER).
     swapExactETHForTokensSupportingFeeOnTransferTokens{value: amountEth
     }(
      minOut,
      path,
      address(this),
      block.timestamp + 600  // 10 minute deadline
  );
  // code
}
```

Impact

1. There is effectively no expiry/timebox on the swap path; the deadline always passes.
2. Off-chain automation cannot enforce a specific TTL policy per trade.
3. The hardcoded "+600" may give a false impression of protection while providing none.

Recommendation

Add a `deadline` parameter to `burnProceedsETH()` and require it to be in the future, optionally cap the maximum horizon to avoid excessively lax expiries.

```
- function burnProceedsETH(uint256 amountEth, uint256 minOut) external
    onlyController nonReentrant {
+ function burnProceedsETH(uint256 amountEth, uint256 minOut, uint256
    deadline) external onlyController nonReentrant {
     require(address(this).balance >= amountEth, "insufficient ETH");
     require(amountEth > 0, "zero amount");
     require(minOut > 0, "zero minOut");
+    require(deadline > block.timestamp, "deadline not in future");

     address[] memory path = new address[](2);
     path[0] = WETH;
     path[1] = HEROSTR;

     uint256 herostrBefore = IERC20(HEROSTR).balanceOf(address(this));

     IUniswapV2Router02(ROUTER).
        swapExactETHForTokensSupportingFeeOnTransferTokens{value:
        amountEth}(
         minOut,
         path,
         address(this),
+        deadline
     );
     //  code
}
```

Team Response

Status: ACKNOWLEDGED - ACCEPTABLE

The current deadline implementation provides basic protection against extremely delayed transactions. Our design uses `minOut = 0` for reliability combined with small swap tranches for price protection, making additional deadline complexity unnecessary for our use case.

[I-05] Keeper Docs Collection Address Differs from Contract Immutable

Severity

Informational Risk

Description

Keeper documentation lists a different OCH collection address than the vault's immutable `COLLECTION` in code. This configuration drift can cause bots to monitor or trade the wrong collection and create inconsistencies across tooling.

The documentation refers to GenesisHero (GHERO) collection.

While the Vault1271 contract refers to OCH_GACHA_WEAPON (OGW), which might not be correct.

Location of Affected Code

File: keeper_documentation

```
# OCH Heroes System
NFT_VAULT = "0x7E9Ed861B4b998B4fd942216DB11fec1caA93e7B"
NFT_COLLECTION = "0x7c47ea32fd27d1a74fc6e9f31ce8162e6ce070eb" // <@
    different address!
SEAPORT = "0x0000000000000068F116a894984e2DB1123eB395"
ROUTER = "0xad1eCa41E6F772bE3cb5A48A6141f9bcc1AF9F7c"
```

File: NFTVault%20Final.sol#L80

```
address public immutable COLLECTION = 0
    x686bFe70F061507065f3E939C12aC9EE5a564dCf;  // OCH Heroes
```

Impact

Operational mistakes and inconsistent configuration between the contract and the off-chain system.

Recommendation

Align addresses across code and documentation.

Team Response

Status: FIXED

This discrepancy was due to testing configurations. The team has updated the contract to reflect the correct OCH Heroes collection address used in the production contracts.

Implementation:

```
address public immutable COLLECTION = 0
    x7c47ea32fd27d1a74fc6e9f31ce8162e6ce070eb;  // Corrected OCH Heroes
    address
```

[I-06] Misleading Accounting: "Sale Proceeds" Actually Tracks Burned ETH Amount

Severity

Informational Risk

Description

The metric `totalETHFromSales` is described as tracking ETH earned from selling NFTs, but it is incremented by the ETH amount passed into `burnProceedsETH()` rather than by actual marketplace sale proceeds. This conflates sale revenue with the amount routed into burns, which can diverge from real proceeds.

Location of Affected Code

File: NFTVault%20Final.sol#L95

```
uint256 public totalETHFromSales;   // Total ETH earned from selling OCH
   Heroes
```

File: NFTVault%20Final.sol#L251-L253

```
function burnProceedsETH(uint256 amountEth, uint256 minOut) external
   onlyController nonReentrant {
  // code
  // Update our stats
  totalETHFromSales += amountEth;
  totalHEROSTRBurned += herostrReceived;
  totalBurns++;

  emit HEROSTRBurned(amountEth, herostrReceived, totalBurns);
}
```

Impact

Operational reporting and strategy evaluation can be skewed as "sale proceeds" may over/understate actual marketplace revenue

Recommendation

- Rename `totalETHFromSales` to reflect its meaning (e.g., `totalETHBurned` or `totalETHRoutedToBurn`).
- If actual sale proceeds are required, increment a separate counter when sales are confirmed (e.g., upon receipt events or decoded Seaport receipts).

Team Response

Status: FIXED

We have updated variable naming and documentation to accurately reflect what is being tracked. The variable has been renamed from `totalETHFromSales` to `totalETHBurned` to avoid confusion about its purpose.

Implementation:

```
uint256 public totalETHBurned;  // Renamed from totalETHFromSales for
    clarity

function getStats() external view returns (
    uint256 ethBalance,
    uint256 ethSpent,
    uint256 nftsBought,
    uint256 ethBurned, // Updated return variable name
    uint256 herostrBurned,
    uint256 burnCount
) {
    return (
        address(this).balance,
        totalETHSpent,
        totalNFTsBought,
        totalETHBurned, // Updated variable reference
        totalHEROSTRBurned,
        totalBurns
    );
}
```

[G-01] Router Allowance Checked on Every Swap Consumes Gas

Severity

Gas Optimization

Description

The contract checks the router allowance on every swap and approves only when the allowance is insufficient (e.g., the first swap or after an external reset). This imposes a per-swap SLOAD and occasional SSTORE. Since the router is immutable and approval is to `type(uint256).max`, setting the approval once in the constructor allows removing (or gating) the per-swap check to reduce gas in swap execution.

Location of Affected Code

File: HEROSTR.sol#L462-L465

```
function _swapBackExact(uint256 tokensToSwap) private returns (uint256
    sold) {
   // code
   // Check if we need approval (more gas efficient)
   uint256 currentAllowance = allowance(address(this), address(router));
   if (currentAllowance < tokensToSwap) {
       _approve(address(this), address(router), type(uint256).max);
   }
   // code
}
```

Impact

- Saves an SLOAD on every swap and avoids occasional SSTORE when allowance is topped up.
- Reduces per-swap gas, which compounds across frequent swapbacks.

Recommendation

Consider applying the following changes:

- In the `constructor()`, set a one-time infinite approval:

  ```
      _approve(address(this), address(router), type(uint256).max);
  ```

- Remove the per-swap allowance check/approve in `_swapBackExact()`.
- Remove the extra `approveRouterMax()` function, with constructor-time approval and no per-swap checks, it becomes redundant.

Team Response

Status: FIXED

We have implemented the suggested optimization by setting infinite approval in the constructor and removing per-swap allowance checks.

Implementation:

```
// In constructor
_approve(address(this), address(router), type(uint256).max);

// Removed allowance check from _swapBackExact
```

our shielding · Your smart contracts, our shielding · Your smart

shieldify

Thank you!