CD SECURITY

AUDIT REPORT

HexPayDay
September 2023

Introduction

A time-boxed security review of the **HexPayDay** protocol was done by **ddimitrov22** and **chrisdior4**, with a focus on the security aspects of the application's implementation.

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 find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **HexPayDay**

When a user is ending a HEX stake, we can observe a series of SLOADs, each costing 2100 gas to perform, for each and every day that the stake was active and that is gas expensive. Hex Pay Day is striving to improve this in a way where everyone uses the same contract to end their stake. Simple example:

A user deposits a HSI (hex stake instance) from hedron.pro / icosa.pro into the protocol so that the existing stake manager is the owner. If certain amount of people do that, then they can all have their stake ended in one transaction through the `ExistingStakeManager` contract.

A user is incentivized to end stakes by collecting tokens in form of tips. Those tokens can be either `attributed` or `unattributed` and any token can be added through the `CurrencyList` contract.

More docs

Severity classification

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

Impact - the technical, economic and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***initial review commit hash* - f4209ff870decb61e1e1eaadf2b78c0c85131803**

Scope

The following smart contracts were in scope of the audit:

- `Bank.sol`
- `CurrencyList.sol`
- `EarningsOracle.sol`
- `EncodableSettings.sol`
- `ExistingStakeManager.sol`
- `GoodAccounting.sol`
- `HSIStakeManager.sol`
- `Magnitude.sol`
- `MaximusStakeManager.sol`
- `MulticallExtension.sol`
- `SingletonHedronManager.sol`
- `SingletonCommunis.sol`
- `SingletonMintManager.sol`
- `StakeEnder.sol`
- `StakeInfo.sol`
- `StakeManager.sol`
- `StakeStarter.sol`
- `Tipper.sol`
- `TransferableStakeManager.sol`
- `UnderlyingStakeManager.sol`
- `UnderlyingStakeable.sol`
- `Utils.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 0 issues
- Medium: 5 issues
- Low: 3 issues
- Informational: 4 issues

Findings Summary

ID	Title	Severity
[M-01]	Wrong accounting for rebase/fee-on-transfer tokens	Medium
[M-02]	Missing input sanitization	Medium
[M-03]	Unsafe downcasts can lead to errors	Medium
[M-04]	`safeTransfer`method does not check the codesize of the token address	Medium

ID	Title	Severity
[M-05]	A require check is creating a false sense of security	Medium
[L-01]	Possible overflow in `_addToTokenWithdrawable`	Low
[L-02]	Use `abi.encodeCall` instead of `abi.encodeWithSelector`	Low
[L-03]	Functions should revert when a condition is not met	Low
[I-01]	Wrong comment in `HSIStakeManager.sol`	Informational
[I-02]	Missing event emissions in state changing methods	Informational
[I-03]	Use `address.code.size` attribute	Informational
[I-04]	NatSpec docs are incomplete	Informational

Detailed Findings

[M-01] Wrong accounting for rebase/fee-on-transfer tokens

Severity

Impact: High, as the balances for such tokens will be wrong and funds can be lost

Likelihood: Low, as this will happen only for the tips for the specific tokens

Description

Currently, anyone can add any tokens from the `CurrencyList` contract. This means that rebase and fee-on-transfer tokens can be added intentionally or by mistake which will cause errors in the accounting of the balances of such tokens. Rebasing tokens are designed to adjust the supply of tokens automatically based on the market demand while fee-on-transfer tokens charge a fee on each transfer. This would lead to having less tokens than expected in the contract and the accounting of such tokens will be completely wrong.

Recommendations

Consider checking the balance of the contract before and after token transfers and using it instead of the amount specified in the contract. Another possible solution is to add access control and allow only specific roles to call `addCurrencyToList`.

Client - fixed

[M-02] Missing input sanitization

Severity

Impact: High, because it could lead to underflow errors and unexpected behavior

Likelihood: Low, because an error from the user is required

Description

There are a few instances within the project where lack of checks can lead to problems.

The `payoutDelta` function takes two input parameters - `startDay` and `untilDay` to return `payout` and `shares`:

```
  function payoutDelta(uint256 startDay, uint256 untilDay) external view
returns(uint256 payout, uint256 shares) { //@audit - require that untilDay
> startDay
    TotalStore memory start = totals[startDay];
    TotalStore memory until = totals[untilDay];
    unchecked {
      return (
        until.payout - start.payout,
        until.shares - start.shares
      );
    }
  }
```

However, there is no check that the `untilDay > startDay` and an underflow can occur inside the `unchecked` box if `start.payout` and `start.shares` are greater than `until.payout` and `until.shares`.

The same is true for the `payoutDeltaTrucated` function where the function will revert if `untilDay > startDay`.

```
 function payoutDeltaTrucated(
    uint256 startDay,
    uint256 untilDay,
    uint256 multiplier
  ) external view returns(uint256 payout) {
    return ((
      (totals[untilDay].payout - totals[startDay].payout) * multiplier *
(untilDay - startDay)
    ) / (
      totals[untilDay].shares - totals[startDay].shares
    )) - (
      // for a 1 day span, the amount is actually known
      (untilDay - startDay) - 1
    );}
```

Another instance where a check is required is the `collectUnattributedPercent` function:

```
     * @param basisPoints the number of basis points (100% = 10_000)
     */
    function collectUnattributedPercent(
      address token, bool transferOut, address payable recipient,
      uint256 basisPoints
    ) external returns(uint256 amount) {
      uint256 unattributed = _getUnattributed(token);
      amount = (unattributed * basisPoints) / TEN_K; //@audit if
  unattributed * basisPoints) < 10k it will round down to zero
      _collectUnattributed(token, transferOut, recipient, amount,
  unattributed);
    }
```

As we can see from the comment, the input parameter `basisPoints` should not be greater than `(100% = 10_000)`. However, any value can be passed including number greater than `10_000`.

Recommendations

Add appropriate checks to avoid any unexpected results and reverts.

Client - Fixed

The client decided to delete the contract in which the first two issues from this finding were present, so they are not a concern anymore. The third issue from this finding is fixed.

[M-03] Unsafe downcasts can lead to errors

Severity

Impact: High because important data can be lost

Likelihood: Low because it will happen only when a very large amounts are used

Description

There are instances where a function takes a `uint256` as an input parameter and within the function is downcasted to a much smaller uint:

```
    function _mintHedron(uint256 index, uint256 stakeId) internal virtual
  returns(uint256 amount) {
      return IHedron(HEDRON).mintNative(index, uint40(stakeId));
    }
```

This is a problem because if the `stakeId` is bigger than `uint40`, then only the least significant 40 bits will be used. This can lead to lost data. The same applies for the `_stakeGoodAccounting` function:

```
  function _stakeGoodAccounting(address stakerAddr, uint256 stakeIndex,
uint256 stakeIdParam) internal {
    // no data is marked during good accounting, only computed and placed
into logs
    // so we cannot return anything useful to the caller of this method
    UnderlyingStakeable(TARGET).stakeGoodAccounting(stakerAddr,
stakeIndex, uint40(stakeIdParam)); //@audit typecast error
  }
```

Recommendations

Be consistent with `uints`. Change the input parameters to the specific uint (`uint40` for example) to avoid losing any data and undesirable scenarios.

Client - acknowledged:

"This downcast is due to the hex contract only having stake id's up to `uint40`. So long as hex is the target contract, this will not be an issue. Client checks should exist to constrain stake ids, but enforcing on contract side happens in dependency (HEX)."

[M-04] Solmate `safeTransfer` method does not check the codesize of the token address

Severity

Impact: Medium, as this could lead to wrong accounting

Likelihood: Medium, as the protocol is designed that users can add any tokens

Description

The Solmate `safeTransfer` methods are used for transferring ERC20 tokens. However, as we can see from the docs of the library, the methods doesn't check the existence of code at the token address:

```
/// @dev Note that none of the functions in this library check that a
token has code at all! That responsibility is delegated to the caller.
```

Hence, when the `safeTransfer` and `safeTransferFrom` methods are called on a token address that doesn't have contract in it, it will always return success, bypassing the return value check. Due to this protocol will think that funds has been transferred and successful , and records will be accordingly calculated, but in reality funds were never transferred.

Recommendations

Use OpenZeppelin's safeERC20 or implement a code existence check.

Client - acknowledged:

"While this is true, the use of `code.size` in currency list does this check once when the address is first seen. After that point, it is up to the ender and staker to coordinate regarding which tokens are valuable enough to use in the contract. If the accounting does not bleed between tokens, then there is no effect outside of tokens that act in this non normative way. The code size may fail at later stages if the token in question is self destructed, however the wrong accounting would be isolated to that token only and front ends can choose to ignore those tokens as they see fit. So, to recap, 1) tokens added only affect off chain services which can blacklist and increased gas cost is used here to reduce that issue 2) anyone adding bad tokens to this list will only end up affecting stakes that they control so there is no incenive to add bad tokens."

[M-05] A require check is creating a false sense of security

Severity

Impact: Medium, as this could lead to false assumption from the users

Likelihood: Medium, as it is easy to bypass the check

Description

In `addCurrencyToList()` we have the following check:

```
    if (ERC20(token).balanceOf(msg.sender) == ZERO) {
        revert MustBeHolder();
      }
```

It creates false sense of security because everybody can create a malicious token ot whatever token and buy just 1 token. This will be enough to bypass the check.

Recommendations

If you truly want to make this work, it got to be with a whitelist for allowed tokens.

Client - acknowledged:

"This check is only intended to increase gas costs for anyone who wishes to manipulate publicly creatable data. If this check were not in place, then anyone would be able to add any token to the currency list, even ones that they may not hold. With this check, one at least has to be a token holder at some point in time to add the currency to the list."

[L-01] Possible overflow in `_addToTokenWithdrawable`

The functions `_addToTokenWithdrawable` is just an `unchecked` box that updates balances:

```
  function _addToTokenWithdrawable(address token, address to, uint256
amount) internal {
    unchecked {
      withdrawableBalanceOf[token][to] = withdrawableBalanceOf[token][to]
+ amount; //@audit - there is no guarantee that this will not overflow
      attributed[token] = attributed[token] + amount;
    }
  }
```

However, there is no guarantee that the `withdrawableBalanceOf[token][to]` will not overflow. If a user deposits a large amount of a particular tokens several times, potentially it can overflow and lose all of his tokens in the `withdrawableBalanceOf` mapping. Even though, the main purpose of this contract is to save gas, consider to remove the `unchecked` box to avoid overflow problems.

Client - acknowledged:

"While this may be possible for some tokens, overflow would be isolated to those tokens used by those accounts. Any token that is approaching uint256 in total supply or has bad accounting should not be used with this contract. All other points where this method is used, the amount is checked / limited."

[L-02] Use `abi.encodeCall` instead of `abi.encodeWithSelector`

The problem with `abi.encodeWithSelector` is that the compiler does not check whether the supplied values actually match the types expected by the called function. `abi.encodeCall` which is similar to `abi.encodeWithSelector`, just that it does perform this type checks. Note that `abi.encodeCall` is available from version `pragma solidity 0.8.11`.

```
   function stakeTransfer(uint256 stakeId, address to) external payable {
//@audit - why is it payable
    ...
    }
    (bool success, bytes memory data) = to.call(
      abi.encodeWithSelector(IStakeReceiver.onStakeReceived.selector,
msg.sender, stakeId) //@audit use abi.encodeCall
```

Client - fixed

[L-03] Functions should revert when a condition is not met

The `_deductWithdrawable` function has a check that the `value > ZERO` before the withdrawable balances are updated. However, if this is not true, the `if` block will just be skipped and the call will succeed.

```
 function _deductWithdrawable(address token, address account, uint256
amount) internal returns(uint256 value) {
  uint256 withdrawable = withdrawableBalanceOf[token][account];
  value = _clamp({
   amount: amount,
   max: withdrawable
  });
  if (value > ZERO) {
   unchecked {
    withdrawableBalanceOf[token][account] = withdrawable - value;
    attributed[token] = attributed[token] - value;
   } //@audit consider reverting if the value is zero
  }
 }
```

Consider adding a custom error so there is no confusion if the function call was actually successful or not.

Client - acknowledged:

"This check is not an imperative that should waste the gas of any end stakes that occur before this, say, if it occurs in a multicall loop (multicallextension.sol), in the sequence of, 1) end stake 2) end stake 3) end stake 4) withdraw funds, if they were not tipped in a currency and thought that they would be, then the 3 end stakes would fail to be ended. An error does occur downstream if a caller tries to start a stake with 0 amount (stakestarter.sol:52) but other cases should not have to fail to cover that case."

[I-01] Wrong comment in `HSIStakeManager.sol`

In `depositHsi()`, `_deposit721()` is called with token param: `HSIM` which is the hardcoded address of the `HSI` token.

```
function depositHsi(uint256 tokenId, uint256 encodedSettings) external
returns(address hsiAddress) {
  address owner = _deposit721({
   token: HSIM,
   ...
...
```

But in `_deposit721()` NatSpec, it says that the address of the token is `HEDRON`.

```
* deposit a tokenized hsi into this contract
  * @param token the address of the token (HEDRON)//@audit here should be
HSI not Hedron
  * @param tokenId the token id to deposit into this contract
```

```
     */
    function _deposit721(address token, uint256 tokenId) internal
  returns(address owner) {
      ...
```

Client - fixed

[I-02] Missing event emissions in state changing methods

It's a best practice to emit events on every state changing method for off-chain monitoring. The `setExternalPerpetualFilter()` in `MaximusStakeManager.sol` is missing event emission, which should be added:

```
  function setExternalPerpetualFilter(address _externalPerpetualFilter)
  external {
      if (msg.sender != externalPerpetualSetter) revert NotAllowed();
      externalPerpetualSetter = address(0);
      externalPerpetualFilter = _externalPerpetualFilter;
    }
```

In `MaximusStakeManager.sol` the `setExternalPerpetualFilter()` is changing 2 state variables meaning that it should emit an event.

Client - acknowledged:

"Generally, state changes are a good thing to emit events for, this update is one time and never changed after it is updated once. Neither front ends, nor contracts should be going to the future validation contract in order to maintain a single entry point for determining validity before and after the new pools launch. Polling is a far better pattern in this case becase it can be added in the flow of an mds1 multicall and not require log checks / listener services."

[I-03] Use `address.code.size` attribute

Solidity version 0.8.13 introduced an address.code.size attribute, which is equivalent to calling the extcodesize opcode in inline assembly. It can be used to simplify the `isContract` function in `Hedron.sol`:

```
   function isContract(address account) internal view returns (bool) {
          // This method relies on extcodesize, which returns 0 for
  contracts in
          // construction, since the code is only stored at the end of the
          // constructor execution.

          uint256 size;
```

```
        assembly {
            size := extcodesize(account)
        }
        return size > 0;
    }
```

Change it to:

```
 function isContract(address account) internal view returns (bool) {
        return address.code.size > 0;
    }
```

Client - fixed

[I-04] NatSpec docs are incomplete

NatSpec docs are incomplete

Some external methods are missing certain components from the NatSpec documentation such as `@param` & `@returns` (`collectUnattributed()`) and some methods are missing it completely such as `createTo()`. NatSpec documentation is essential for better understanding of the code by developers and auditors and is strongly recommended. Please refer to the NatSpec format and follow the guidelines outlined there.

Client - fixed