SHERLOCK

SHERLOCK SECURITY REVIEW FOR

harpie

Prepared for:	Harpie
Prepared by:	Sherlock
Lead Security Expert:	leastwood
Dates Audited:	September 15 - September 18, 2022
Prepared on:	October 12, 2022

Introduction

Harpie is the first on-chain firewall preventing hacks, scams, and theft. Harpie services monitor pending transactions for potential attacks.

Scope

The following contracts in the Harpie Contracts @ 97083d repo are in scope.

- `Transfer.sol`
- `Vault.sol`

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Total Issues

Medium	High
9	0

Security Experts

IIIIIII
ak1
minhquanym
JohnSmith
defsec
sirhashalot
hickuphh3
Tomo
Lambda
pashov
0xNazgul
ladboy233
0xSmartContract
leastwood
HonorLt
xiaoming90
cccz
IEatBabyCarrots
csanuragjain
CodingNameKiki
sach1r0
Waze
yixxas
gogo
hansfriese
dipp
saian
rbserver
millers.planet
Bnke0x0
Chom
TomJ
Sm4rty
chainNue
Dravee

Issue M-1: Use `safeTransferFrom()` instead of `transferFrom()` for outgoing erc721 transfers

Source: https://github.com/sherlock-audit/2022-09-harpie-judging/tree/main/001-M

Found by

CodingNameKiki, millers.planet, 0xNazgul, cccz, Bnke0x0, Chom, Waze, IEatBabyCarrots, TomJ, Tomo, hickuphh3, pashov, sach1r0, Sm4rty, IllIIII, chainNue, Dravee

Summary

It is recommended to use `safeTransferFrom()` instead of `transferFrom()` when transferring ERC721s out of the vault.

Vulnerability Detail

The `transferFrom()` method is used instead of `safeTransferFrom()`, which I assume is a gas-saving measure. I however argue that this isn't recommended because:

- OpenZeppelin's documentation discourages the use of `transferFrom()`; use `safeTransferFrom()` whenever possible
- The recipient could have logic in the `onERC721Received()` function, which is only triggered in the `safeTransferFrom()` function and not in `transferFrom()`. A notable example of such contracts is the Sudoswap pair:

```
function onERC721Received(
  address,
  address,
  uint256 id,
  bytes memory
) public virtual returns (bytes4) {
  IERC721 _nft = nft();
  // If it's from the pair's NFT, add the ID to ID set
  if (msg.sender == address(_nft)) {
    idSet.add(id);
  }
  return this.onERC721Received.selector;
}
```

- It helps ensure that the recipient is indeed capable of handling ERC721s.

Impact

While unlikely because the recipient is the function caller, there is the potential loss of NFTs should the recipient is unable to handle the sent ERC721s.

Code Snippet

https://github.com/Harpieio/contracts/blob/97083d7ce8ae9d85e29a139b1e981464ff92b89e/contracts/Vault.sol#L137

Recommendation

Use `safeTransferFrom()` when sending out the NFT from the vault.

```
- IERC721(_erc721Address).transferFrom(address(this), msg.sender, _id);
+ IERC721(_erc721Address).safeTransferFrom(address(this), msg.sender, _id);
```

Note that the vault would have to inherit the `IERC721Receiver` contract if the change is applied to `Transfer.sol` as well.

Harpie Team

Added safeTransferFrom in withdraw function. Fix here.

Lead Senior Watson

Makes sense to be compatible with contracts as recipients. Confirmed fix.

Issue M-2: Cross-chain replay attacks are possible with `changeRecipientAddress()`

Source: https://github.com/sherlock-audit/2022-09-harpie-judging/tree/main/004-M

Found by

minhquanym, JohnSmith, IllIIII

Summary

Mistakes made on one chain can be re-applied to a new chain

Vulnerability Detail

There is no `chain.id` in the signed data

Impact

If a user does a `changeRecipientAddress()` using the wrong network, an attacker can replay the action on the correct chain, and steal the funds a-la the wintermute gnosis safe attack, where the attacker can create the same address that the user tried to, and steal the funds from there

Code Snippet

https://github.com/Harpieio/contracts/blob/97083d7ce8ae9d85e29a139b1e981464ff92b89e/contracts/Vault.sol#L60-L73

Tool used

Manual Review

Recommendation

Include the `chain.id` in what's hashed

Harpie Team

Added chainId to signature and signature validation. Fix here.

Lead Senior Watson

This is true assuming the contract address is the same across other chains. Confirmed fix.

Issue M-3: Incompatability with deflationary / fee-on-transfer tokens

Source: https://github.com/sherlock-audit/2022-09-harpie-judging/tree/main/005-M

Found by

Lambda, cccz, hansfriese, IEatBabyCarrots, rbserver, JohnSmith, minhquanym, Tomo, leastwood, dipp, defsec, HonorLt, IllIIII, saian, csanuragjain

Summary

https://github.com/Harpieio/contracts/blob/97083d7ce8ae9d85e29a139b1e981464ff92b89e/contracts/Transfer.sol#L93-L100

In case ERC20 token is fee-on-transfer, Vault can loss funds when users withdraw

Vulnerability Detail

In `Transfer.transferERC20()` function, this function called `logIncomingERC20()` with the exact amount used when it called `safeTransferFrom()`. In case ERC20 token is fee-on-transfer, the actual amount that Vault received may be less than the amount is recorded in `logIncomingERC20()`.

The result is when a user withdraws his funds from `Vault`, Vault can be lost and it may make unable for later users to withdraw their funds.

Proof of Concept

Consider the scenario

1. Token X is fee-on-transfer and it took 10% for each transfer. Alice has 1000 token X and Bob has 2000 token X
2. Assume that both Alice and Bob are attacked. Harpie transfers all token of Alice and Bob to Vault. It recorded that the amount stored for token X of Alice is 1000 and Bob is 2000. But since token X has 10% fee, Vault only receives 2700 token X.
3. Now Bob withdraw his funds back. With `amountStored=2000`, he will transfer 2000 token X out of the Vault and received 1800.
4. Now the Vault only has 700 token X left and obviously it's unable for Alice to withdraw

Tool used

Manual Review

Recommendation

Consider calculating the actual amount Vault received to call `logIncomingERC20()` Transfer the tokens first and compare pre-/after token balances to compute the actual transferred amount.

Harpie Team

Using difference in balance in vault rather than token transfer amount. Fix here.

Lead Senior Watson

While it's true the fix does allow for compatabiliy with fee-on-transfer tokens, it does not correctly handle rebasing tokens. Might be useful to explicily note that rebasing tokens are not supported or instead you could adopt mint shares to represent the ownership over the vault's tokens.

Harpie Team

On rebasing tokens, we just won't be able to support them for now.

Issue M-4: Usage of deprecated transfer() can result in revert.

Source: https://github.com/sherlock-audit/2022-09-harpie-judging/tree/main/007-M

Found by

Lambda, cccz, yixxas, Waze, IEatBabyCarrots, pashov, 0xSmartContract, JohnSmith, Tomo, CodingNameKiki, sach1r0, IllIIII, csanuragjain, gogo

Summary

The function withdrawPayments() is used by the Owners to withdraw the fees.

Vulnerability Detail

transfer() uses a fixed amount of gas, which was used to prevent reentrancy. However this limit your protocol to interact with others contracts that need more than that to process the transaction.

Specifically, the withdrawal will inevitably fail when: 1.The withdrawer smart contract does not implement a payable fallback function. 2.The withdrawer smart contract implements a payable fallback function which uses more than 2300 gas units. 3.The withdrawer smart contract implements a payable fallback function which needs less than 2300 gas units but is called through a proxy that raises the call's gas usage above 2300.

Impact

transfer() uses a fixed amount of gas, which can result in revert. https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/

Code Snippet

https://github.com/Harpieio/contracts/blob/97083d7ce8ae9d85e29a139b1e981464ff92b89e/contracts/Vault.sol#L159 https://github.com/Harpieio/contracts/blob/97083d7ce8ae9d85e29a139b1e981464ff92b89e/contracts/Vault.sol#L156-L160

Tool used

Manual Review

Recommendation

Use call instead of transfer(). Example: (bool succeeded,) = _to.call{value: _amount}(""); require(succeeded, "Transfer failed.");

Lead Senior Watson

Fair considering recipient may be a contract with custom logic for `receive()`. But this is definitely recoverable if the fee recipient wasn't able to receive funds.

Harpie Team

Moved to .call. Fix here.

Lead Senior Watson

Confirmed fix.

Issue M-5: There is no limit on the amount of fee users have to pay

Source: https://github.com/sherlock-audit/2022-09-harpie-judging/tree/main/008-M

Found by

hickuphh3, 0xSmartContract, xiaoming90, ak1, minhquanym, leastwood, defsec, HonorLt

Summary

https://github.com/Harpieio/contracts/blob/97083d7ce8ae9d85e29a139b1e981464ff92b89e/contracts/Transfer.sol#L57 https://github.com/Harpieio/contracts/blob/97083d7ce8ae9d85e29a139b1e981464ff92b89e/contracts/Transfer.sol#L88

Vulnerability Detail

There is no upper limit on the amount of fee users have to pay to withdraw their funds back. So any EOA can call transfer function on `Transfer` contract can set an unreasonable amount of fee and users have to pay it if they want their funds back. We need to make sure that users' funds cannot be loss even when the protocol acts maliciously.

Impact

In case the protocol acts maliciously and set `fee=1e18` to transfer users' fund to `Vault`, users cannot withdraw their funds since fee is too high.

Proof of Concept

In both `transferERC20()` and `transferERC721()`, EOA is caller and can set `fee` param to any value it wants.

```
function transferERC721(address _ownerAddress, address _erc721Address, uint256
↪  _erc721Id, uint128 _fee) public returns (bool) {
      require(_transferEOAs[msg.sender] == true || msg.sender == address(this),
↪  "Caller must be an approved caller.");
      require(_erc721Address != address(this));
      (bool transferSuccess, bytes memory transferResult) =
↪  address(_erc721Address).call(
          abi.encodeCall(IERC721(_erc721Address).transferFrom, (_ownerAddress,
↪  vaultAddress, _erc721Id))
      );
      require(transferSuccess, string (transferResult));
      (bool loggingSuccess, bytes memory loggingResult) =
 ↪  address(vaultAddress).call(
          abi.encodeCall(Vault.logIncomingERC721, (_ownerAddress, _erc721Address,
 ↪  _erc721Id, _fee))
      );
      require(loggingSuccess, string (loggingResult));
      emit successfulERC721Transfer(_ownerAddress, _erc721Address, _erc721Id);
      return transferSuccess;
  }
```

And users need to send enough fee (native token) to withdraw their fund back on `Vault`

```
function withdrawERC721(address _originalAddress, address _erc721Address, uint256
 ↪  _id) payable external {
      require(_recipientAddress[_originalAddress] == msg.sender, "Function caller
 ↪  is not an authorized recipientAddress.");
      require(_erc721Address != address(this), "The vault is not a token
 ↪  address");
      require(canWithdrawERC721(_originalAddress, _erc721Address, _id),
 ↪  "Insufficient withdrawal allowance.");
      require(msg.value >=
 ↪  _erc721WithdrawalAllowances[_originalAddress][_erc721Address][_id].fee,
 ↪  "Insufficient payment.");

      _erc721WithdrawalAllowances[_originalAddress][_erc721Address][_id].isStored
 ↪  = false;
      _erc721WithdrawalAllowances[_originalAddress][_erc721Address][_id].fee = 0;
      IERC721(_erc721Address).transferFrom(address(this), msg.sender, _id);
}
```

Tool used

Manual Review

Recommendation

Consider adding an upper limit on the amount of fee users need to pay

Lead Senior Watson

Currently there is no way to revoke a change fee controller request. I'd shy away from using a mapping, adds unnecessary overhead when it can be handled by a `pendingFeeController` variable. Also important to note that mapping in `changeFeeController()` is not cleared.

Harpie Team

Using leastwood's suggestion of a timelock for feeController. Fix here. Supplementary fixes for this issue: 1, 2, 3.

Lead Senior Watson

Confirmed fixes.

Issue M-6: Signature malleability not protected against

Source: https://github.com/sherlock-audit/2022-09-harpie-judging/tree/main/010-M

Found by

0xNazgul, pashov, IllIIII, ladboy233, defsec, sirhashalot

Summary

OpenZeppelin has a vulnerability in versions lower than 4.7.3, which can be exploited by an attacker. The project uses a vulnerable version

Vulnerability Detail

All of the conditions from the advisory are satisfied: the signature comes in a single `bytes` argument, `ECDSA.recover()` is used, and the signatures themselves are used for replay protection checks https://github.com/OpenZeppelin/openzeppelin-contracts/security/advisories/GHSA-4h98-2769-gh6h

If a user calls `changeRecipientAddress()`, notices a mistake, then calls `changeRecipientAddress()` again, an attacker can use signature malleability to re-submit the first change request, as long as the old request has not expired yet.

Impact

The wrong, potentially now-malicious, address will be the valid change recipient, which could lead to the loss of funds (e.g. the attacker attacked, the user changed to another compromised address, noticed the issue, then changed to a whole new account address, but the attacker was able to change it back and withdraw the funds to the unprotected address).

Code Snippet

https://github.com/Harpieio/contracts/blob/97083d7ce8ae9d85e29a139b1e981464ff92b89e/package.json#L23

Tool used

Manual Review

Recommendation

Change to version 4.7.3

Lead Senior Watson

Good find and the fix seems straightforward. Upgrade OZ.

Harpie Team

Updated openzeppelin NPM package to do ECDSA 4.7.3. Fix here.

Lead Senior Watson

Confirmed fix.

Issue M-7: Unsafe casting of user amount from `uint256` to `uint128`

Source: https://github.com/sherlock-audit/2022-09-harpie-judging/tree/main/018-M

Found by

Lambda, Tomo, hickuphh3, IllIIl, defsec, sirhashalot

Summary

The unsafe casting of the recovered amount from `uint256` to `uint128` means the users' funds will be lost.

Vulnerability Detail

`logIncomingERC20()` has the recovered amount as type `uint256`, but `amountStored` is of type `uint128`. There is an unsafe casting when incrementing `amountStored`:

```
_erc20WithdrawalAllowances[_originalAddress][_erc20Address].amountStored +=
↪   uint128(_amount);
```

It is thus possible for the amount recorded to be less than the actual amount recovered.

Impact

Loss of funds.

Proof of Concept

The user's balance is `type(uint128).max=2**128`, but the incremented amount will be zero.

Recommendation

`amountStored` should be of type `uint256`. Alternatively, use OpenZeppelin's SafeCast library when casting from `uint256` to `uint128`.

Lead Senior Watson

Not sure, any tokens which would have a token supply over `type(uint128).max` but I guess it's best to be proactive. The proposed fix does create some issues. Instead of having less tokens transferred to the vault, the contract will revert and prevent the transfer entirely. Arguably more funds would be at risk, so you may as well use `uint256` then or accept the risk and keep the slot packing.

Harpie Team

Decided to accept the risk of reverts on leastwood's comment on this issue since it's a lot of gas savings and there probably arent useful tokens w/ supply over (uint128).max. Used @openzeppelin/SafeCast. Fix here.

Lead Senior Watson

Confirmed fix.

Issue M-8: reduceERC721Fee function can not set fee when the NFT token ID is more than type(uint128).max

Source: https://github.com/sherlock-audit/2022-09-harpie-judging/tree/main/081-M

Found by

ak1

Summary

`reduceERC721Fee` function can not set fee when the NFT token ID is more than `type(uint128).max`

Vulnerability Detail

The NFT token ID can be any value within uint 256. As the reduceERC721Fee takes the `_id` argument as `uint128`, when the reduceERC721Fee function is called with an NFT id that has above `type(uint128).max` , the fee can not set to the expected NFT id.

Impact

`High:` RC721Fee can not set fee when the NFT token ID value is more than `type(uint128).max`

Code Snippet

https://github.com/Harpieio/contracts/blob/97083d7ce8ae9d85e29a139b1e981464ff92b89e/contracts/Vault.sol#L148

Tool used

Manual Review

Recommendation

Change the function argument for `reduceERC721Fee` as shown below. `beforefix:` function reduceERC721Fee(address _originalAddress, address _erc721Address, `uint128_id`, uint128 _reduceBy) external returns (uint128)

`afterfix:` function reduceERC721Fee(address _originalAddress, address _erc721Address, `uint256_id`, uint128 _reduceBy) external returns (uint128)

Lead Senior Watson

Good find! ERC721 standard doesn't enforce how `tokenId` is implemented for a given NFT. Could definitely be greater than `uint128`, although I've never seen a case where this is true.

Harpie Team

Changed to uint256. Fix here.

Lead Senior Watson

Confirmed fix.

Issue M-9: Nonces not used in signed data

Source: https://github.com/sherlock-audit/2022-09-harpie-judging/tree/main/160-M

Found by

IIIIIII

Summary

Nonces are not used in the signature checks

Vulnerability Detail

A nonce can prevent an old value from being used when a new value exists. Without one, two transactions submitted in one order, can appear in a block in a different order

Impact

If a user is attacked, then tries to change the recipient address to a more secure address, initially chooses an insecure compromised one, but immediately notices the problem, then re-submits as a different, uncompromised address, a malicious miner can change the order of the transactions, so the insecure one is the one that ends up taking effect, letting the attacker transfer the funds

Code Snippet

https://github.com/Harpieio/contracts/blob/97083d7ce8ae9d85e29a139b1e981464ff92b89e/contracts/Vault.sol#L67-L71

Tool used

Manual Review

Recommendation

Include a nonce in what is signed

Harpie Team

Fixed by changing nonce system to an incremental system. Fix here.

Lead Senior Watson

Not an issue AFAIK, miners can't reorder txs unless they are signed with the same nonce. There would have to be some serious mis-use of this function by the recipient address, i.e. they would have to ask the server to sign for two different addresses and then broadcast the txs with the same nonce for this call. The proposed fix could probably be safely removed but doesn't hurt to keep it there.