Phage
SECURITY

AutoGains

Security Review

Conducted by:

Pyro, Security Researcher

19.09.2024

Table of Contents

Disclaimer

Audits are a time, resource, and expertise-bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can reveal the presence of vulnerabilities **but cannot guarantee their absence**.

System overview

AutoGains gives users the ability to create vaults with custom-managed strategies, interacting and trading leveraged assets using the new version of the Gains Network. The audit focused on the vault, vault factory, and its subcomponents, helping the system integrate with Gains Network properly.

Executive summary

Overview

Project Name	AutoGains
Repository	https://github.com/Gnome101/AutoGains/tree/main
Commit hash	c72ea2aeb07b50ea9cc52c1008891b7667587341
Remediation	fea39cc04f2e334ea99487d6fc926888baf1be20
Methods	Manual review

Timeline

Audit kick-off	01.09.2024
End of audit	09.09.2024
Remediations start	13.09.2024

Scope

contracts/AutoVault.sol

contracts/Helper.sol

contracts/VaultFactory.sol

Issues Found

Severity	Count
High	3
Medium	6
Low	6
Informational	2

Findings

High Severity

[H-01] Users are unable to make deposits when the vault has active strategies

When the vault is operating (having active strategies), users are not able to deposit in the normal way (using 4626 deposit/mint) as totalAssets would revert if the call is not a callback and there are any active strategies.

This is why the Vaults have startAction, which users can use to deposit/mint assets when the vault is operational. However, that function would revert if any normal user calls it because it calls sendVaultBalanceReq, which in turn triggers sendInfoRequest inside our factory. As we can see from the code below, sendInfoRequest requires the caller (our user) to be an approvedCaller.

```
function sendInfoRequest(
   address caller,
   Chainlink.Request memory req,
   uint256 fee
) public returns (bytes32 requestId) {
   if (!approvedCaller[caller]) revert NonApprovedCaller(caller);
   if (!approvedVaults[msg.sender]) revert NonApprovedVault(msg.sender);
   requestId = _sendChainlinkRequest(req, fee);
   requestToCaller[requestId] = AutoVault(msg.sender);
}
```

This means that regular users would not be able to use that function and thus would not be able to deposit when the vault is active.

Recommendation:

Inline the _sendChainlinkRequest call inside sendVaultBalanceReq.

```
function sendVaultBalanceReq(
   address caller,
   uint256 fee
) external returns (bytes32 requestId) {
   Chainlink.Request memory req = buildChainlinkTradeRequest(msg.sender);
-  return sendInfoRequest(caller, req, fee);
+  if (!approvedVaults[msg.sender]) revert NonApprovedVault(msg.sender);
+  requestId = _sendChainlinkRequest(req, fee);
+  requestToCaller[requestId] = AutoVault(msg.sender
}
```

Resolution: Fixed

[H-02] Users are only able to withdraw during withdrawal periods

Users are only able to withdraw during withdrawal periods. During such a period, the vault is inactive, and all trades are closed. Users can set withdrawal periods using setWithdrawPeriod, where each period is 2 hours and occurs in 7-day intervals (MIN_PERIOD_TIME).

```
function setWithdrawPeriod() external {
    if (this.balanceOf(msg.sender) == 0) {
        revert NotTokenHolder(msg.sender);
    }
    if (block.timestamp ⩽ nextWithdrawPeriod + withdrawPeriodLength) {
        revert WithdrawPeriodAlreadySet();
    }
    nextWithdrawPeriod = block.timestamp + MIN_PERIOD_TIME;
    emit WithdrawPeriodSet(nextWithdrawPeriod);
}
```

However, any user can call setWithdrawPeriod just as the withdrawal period starts and set it to after a week, essentially making it last only 1 block or a few seconds.

Recommendation:

Fix the function so it can be called only when the current withdrawal period has finished.

```
-   if (block.timestamp < nextWithdrawPeriod) {
+   if (block.timestamp ⩽ nextWithdrawPeriod + withdrawPeriodLength) {
        revert WithdrawPeriodAlreadySet();
    }
```

Resolution: Fixed

[H-03] processStrategy uses the entire data array as strategy inputs

When the callback returns to `fulfill`, we first verify `data[1]` to check if the callback was timely, and later use `data[0]` as the price inside executeAction. This means that `data[0]` is the price, and `data[1]` is the timestamp of the oracle update.

However, when we process this data inside processStrategy, we input the entire data array. This should not be the case because the first two variables in the data array are not part of the strategy execution but are merely used for callback verification.

Recommendation:

Cut the array so it inputs elements starting at the 3rd index:

```
function processStrategy(uint256[] memory data) internal {
   // Process data starting from index 2
   uint256[] memory trimmedData = new uint256[](data.length - 2);
   for (uint i = 2; i < data.length; i++) {
      trimmedData[i - 2] = data[i];
   }
   // Continue with strategy execution using trimmedData
}
```

Resolution: Fixed

Medium Severity

[M-01] `createVault` will not work for some tokens

createVault used in VaultFactory uses `collateral.name()` to create the name of the Vault asset and its symbol. However, some tokens return name and symbol as `bytes32` instead of `string`, causing the transaction to revert when it tries to concatenate a `string` and `bytes32`. This issue affects tokens like MKR.

```
AutoVault(clonedVault).initialize(
   collateral,
   initialAmount,
   startInfo,
   string.concat("Auto", collName),
   string.concat("a", collSymbol),
   getAddressKeys(apiInfo, listOfStrategies)
);
```

Recommendation:

For compatibility with most tokens, I recommend giving the vault creator the ability to set custom names, or use one hard-coded name.

```
AutoVault(clonedVault).initialize(
   collateral,
   initialAmount,
   startInfo,
+  string.concat("Auto", collName),
+  string.concat("a", collSymbol),
-  string.concat("Auto", collateral.name()),
-  string.concat("a", collateral.symbol()),
```

Resolution: Fixed

[M-02] Vaults will refer to the wrong owner if it's changed

When creating a vault, VaultFactory sets the factoryOwner as the owner of the factory. This variable is later set in initialize as specialRefer.

```
specialRefer = startingInfo.factoryOwner;
```

specialRefer is hardcoded and will be used as a referrer every time executeAction opens up a trade.

```
GainsNetwork.openTrade(trade, uint16(action >> 236), specialRefer);
indexToStrategy[index] = strategy + 1;
strategyToIndex[strategy] = index + 1;
```

However, if the factory owner changes, every deployed vault up to this point will be configured with the old owner, and there will be no way to update it.

Recommendation:

Call VaultFactory(vaultFactory).owner() when executing a trade to ensure that the current factory owner is used as the referrer.

```
-  GainsNetwork.openTrade(trade, uint16(action >> 236), specialRefer);
+  GainsNetwork.openTrade(trade, uint16(action >> 236), VaultFactory(
   vaultFactory).owner());
   indexToStrategy[index] = strategy + 1;
   strategyToIndex[strategy] = index + 1;
```

Resolution: Fixed

[M-03] Vaults are susceptible to first depositor attack

ERC-4626's deposit function relies on the _convertToShares calculation:

```
assets * (totalSupply() + 10 ** _decimalsOffset) / (totalAssets() + 1) =>
assets * (totalSupply + 1) / (totalAssets + 1)
```

When totalSupply == 0 and totalAssets > assets, the user will receive 0 minted shares. To counter this, the vault mints an initial amount of shares, which must be at least minimumDeposit - 100 wei.

```
if (initialAmount < minimumDeposit) revert DepositTooLow();
AutoVault(clonedVault).initialize(
   collateral,
   initialAmount,
   startInfo,
   string.concat("Auto", collName),
   string.concat("a", collSymbol),
   getAddressKeys(apiInfo, listOfStrategies)
);
```

However, users can predict the creation of vaults because createVault uses OZ's clones, which rely on create. The create function is predictable and follows a pattern, deriving its address from the originating address and nonce. clonedVault = payable(Clones.clone(autoVaultImplementation)); This allows users to send tokens to the vault, causing the initial minted shares to be rounded to 0. This means that just after deploying, the vault has no defense against a first depositor attack. Example:

1. User1 predicts the next VaultFactory address and sends 101 USDC.
2. User2 creates a vault and sends a minimumDeposit of 100 USDC. The vault is initialized but only has 1 share and 201 assets.
3. User3 deposits 1000e6 USDC.
4. User1 front-runs this deposit by depositing 202 assets, getting minted 1 share, and donating 2000e6 USDC to the vault.
5. User3's deposit transaction is executed, but he receives 0 shares because the current vault ratio is 1 share to 2000e6 + 403 assets.

```
assets * (totalSupply + 1) / (totalAssets + 1) => 1000e6 * (1 + 1) / (
   2000e6 + 403)=> 0.99 ... => rounded to 0
```

Recommendation:

Increase the minted shares amount to 1e4 and mint the assets directly to the vault contract. This will prevent attacks and also stop the vault creator from interfering with the other depositors.

Resolution: Fixed

[M-04] Fee is taken before there are any actual deposits

Our internal _deposit function is overridden to handle factory and manager fees. It performs the actual token transfer inside ERC-4626's original _deposit, called after transferring the fees. In essence, this function first transfers the fees and then pulls tokens from users. This can sometimes cause a revert if there are not enough tokens inside the vault.

Example: 1. The vault currently has 10 tokenA with a fee of 0.8%. 2. User1 tries to deposit 10k tokenA, but his transaction reverts, as the system tries to send 10,000 * 0.8% = 80 tokens to the fee recipients before taking his deposit.

Recommendation:

Take the fee before making the deposit. This ensures that the fee is collected first, preventing reverts due to insufficient tokens in the vault.

```
+  super._deposit(caller, receiver, assets, shares);
   if (fee > 0 && recipient2 == receiver) {
      SafeERC20Upgradeable.safeTransfer(IERC20Upgradeable(asset()),
         recipient1, fee);
   } else if (fee > 0 && recipient2 != address(this)) {
      (uint256 fee1, uint256 fee2) = splitFees(fee);
      SafeERC20Upgradeable.safeTransfer(IERC20Upgradeable(asset()),
         recipient1, fee1);
      SafeERC20Upgradeable.safeTransfer(IERC20Upgradeable(asset()),
         recipient2, fee2);
   }
-  super._deposit(caller, receiver, assets, shares);
```

Resolution: Fixed

[M-05] Fees are split wrongly

Inside `_deposit`, the `_feeOnTotal` function splits the fee if `_msgSender` is `recipient1` or `recipient2`.

```
if ((receiver == recipient1 || receiver == recipient2)) {
   if (_doesRecipientPayFee()) {
      return _getMinFee() - _getMinFee() / 2;
   } else {
      return 0;
   }
}
```

However, that fee is split again inside the second else if within `_deposit`.

```
if (fee > 0 && recipient2 == receiver) {
   SafeERC20Upgradeable.safeTransfer(IERC20Upgradeable(asset()), recipient1,
      fee);
} else if (fee > 0 && recipient2 != address(this)) {
   (uint256 fee1, uint256 fee2) = splitFees(fee);
   SafeERC20Upgradeable.safeTransfer(IERC20Upgradeable(asset()), recipient1,
      fee1);
   SafeERC20Upgradeable.safeTransfer(IERC20Upgradeable(asset()), recipient2,
      fee2);
}
```

The final fee would be 25% sent to each party (factory and vault manager) if receiver == recipient1 . This is because if receiver == recipient1, _feeOnTotal splits the fee in half, but then the second else if would split that fee again in half using splitFees.

Recommendation:

Fix the fee handling inside `_deposit` and _withdraw to avoid double-splitting of fees. Ensure that the fee is properly calculated and distributed without unnecessary splits.

```
-  } else if (fee > 0 && recipient2 ≠ address(this)) {
+  } else if (fee > 0 && recipient1 == receiver) {
+     SafeERC20Upgradeable.safeTransfer(
+        IERC20Upgradeable(asset()),
+        recipient2,
+        fee
+     );
+  } else if (fee > 0) {
      (uint256 fee1, uint256 fee2) = splitFees(fee);
      SafeERC20Upgradeable.safeTransfer(
         IERC20Upgradeable(asset()),
         recipient1,
         fee1
      );
      SafeERC20Upgradeable.safeTransfer(
         IERC20Upgradeable(asset()),
         recipient2,
         fee2
      );
```

Resolution: Fixed

[M-06] Factory fees for opening a position can be avoided

Vault managers are charged a percentage of their collateral each time they open a trade:

```
if (actionType == 0) {
   (Trade memory trade, uint32 posPercent) = extractTrade(action, index,
      openPrice);
   trade.collateralAmount -= applySwapFee(trade.collateralAmount,
      feeMultiplier);
   GainsNetwork.openTrade(trade, uint16(action >> 236), specialRefer);
}
```

To avoid this fee, the vault manager can open a small position and later increase its collateral without being charged a fee.

Recommendation:

Apply a fee when increasing the position size:

```
+  trade.collateralAmount -= applySwapFee(trade.collateralAmount,
   feeMultiplier );
   GainsNetwork.increasePositionSize(
      index,
      uint120(Math.mulDiv(trade.collateralAmount, uint32(action >> 204),
         1_000_000)),
      uint24(action >> 180),
      uint64(Math.mulDiv(openPrice, uint32(action >> 148), 1_000_000)),
      uint16(action >> 236)
   );
```

Resolution: Fixed

Low Severity

[L-01] Approve will revert on some tokens

The approve function will revert on tokens such as USDT, which require the allowance to be set to 0 before making any new approvals. This issue can be seen in the extendApproval function.

```
function extendApproval() public {
   getAsset().forceApprove(address(GainsNetwork), type(uint256).max);
   emit ApprovalExtended(msg.sender);
}
```

Recommendation:

Since uint256.max approval was originally granted in the constructor, a change may not be necessary. However, it would still be better to use forceApprove in this case.

```
function extendApproval() public {
-  getAsset().approve(address(GainsNetwork), type(uint256).max);
+  getAsset().forceApprove(address(GainsNetwork), type(uint256).max);
   emit ApprovalExtended(msg.sender);
}
```

Resolution: Fixed

Low Severity

[L-02] extractTrade cannot utilize the vault full balance

The extractTrade function is unable to utilize 100% of the vault balance due to a >= check, which reverts if the vault owner tries to execute a trade with collateralAmount == asset().balanceOf(address(this)).

```
if (collateralAmount ⩾ IERC20(asset()).balanceOf(address(this)))
   revert InsufficientBalance();
```

Recommendation:

Change the verification method to allow for collateralPercentage to be 1_000_000 (100%) by changing the operator from >= to >.

```
-  if (collateralAmount ⩾ IERC20(asset()).balanceOf(address(this)))
+  if (collateralAmount > IERC20(asset()).balanceOf(address(this)))
     revert InsufficientBalance();
```

Resolution: Fixed

Low Severity

[L-03] fulfill will not transfer the fees correctly

In the fulfill function, a fee is taken from the provided collateral and split between the caller and the factory. However, the intended split of 50/50 between the caller and the factory is incorrect. Currently, the fee is split as 1/3 to the caller and 2/3 to the factory.

```
// Send one-third of the oracle fee to the rewardBot
getAsset().safeTransfer(
   rewardInfo.caller,
   VaultFactory(vaultFactory).getOracleFee(asset()) / 3
);
// Send the remaining balance to the vaultFactory
getAsset().safeTransfer(
   vaultFactory,
   rewardInfo.masterFee -
   VaultFactory(vaultFactory).getOracleFee(asset()) / 3
);
```

Recommendation:

Change the division to / 2 instead of / 3 to properly split the fee 50/50.

```
   //Send half of the oracle fee to the rewardBot
   getAsset().safeTransfer(
      rewardInfo.caller,
-     VaultFactory(vaultFactory).getOracleFee(asset()) / 3
+     VaultFactory(vaultFactory).getOracleFee(asset()) / 2
   );
   //Send the remaining balance to the vaultFactory
   getAsset().safeTransfer(
      vaultFactory,
      rewardInfo.masterFee -
-     VaultFactory(vaultFactory).getOracleFee(asset()) / 3
+     VaultFactory(vaultFactory).getOracleFee(asset()) / 2
   );
```

Resolution: We acknowledge this one but the comment was outdated, the caller should recieve 2/3 of an oracle fee which they will since the oracle fee is doubled and they recieve 1/3 of it.

[L-04] revertDuringWithdrawPeriod will work if we are in forced withdraw period

The withdraw period is a specific time when users can withdraw their assets from the vault. These periods occur weekly, with 2 hours allocated for withdrawals, during which the vault does not execute any strategies. The vault manager can also force a withdraw period at any time, for any duration.

However, the revertDuringWithdrawPeriod modifier is missing a check to determine if the vault is in a forced withdraw period.

```
modifier revertDuringWithdrawPeriod() {
   if (
      block.timestamp ≥ nextWithdrawPeriod &&
      block.timestamp ⩽ nextWithdrawPeriod + withdrawPeriodLength
   ) {
      revert NoTradesDuringWithdrawPeriod();
   }
   _;
}
```

This allows the vault manager to execute functions, such as executeStrategy and fulfill, that should not be executable during the withdraw period.

Recommendation:

If this behavior is unintended, consider adding the appropriate check. Otherwise, the behavior can be acknowledged as intentional.

Resolution: Fixed

Low Severity

[L-05] Users can withdraw even if it's not withdraw period

The documentation and code specify that deposits can be made at any time, but withdrawals are only allowed during the designated withdraw period. The totalAssets function is responsible for preventing users from withdrawing when the vault is active.

```
function totalAssets() public view override returns (uint256) {
   uint256 collateralAmount = 0;
   if (GainsNetwork.getTrades(address(this)).length != 0) {
      if (totalValueCollateral.get() == 0) revert NeedCallback();
      collateralAmount = totalValueCollateral.get() - 1;
   }
   return super.totalAssets() + collateralAmount;
}
```

However, this function still allows users to withdraw even when not in a withdraw period, as long as there are no active strategies at the time of the withdrawal. This may lead to unexpected decreases in the vault balance and potentially reduce profits from strategies.

Example: 1. The vault manager is switching strategies due to a sudden market change. 2. They close all strategies and plan to open new ones. 3. During the brief moment when all strategies are closed, Alice and other users withdraw their balances, reducing the vault's collateral.

Recommendation:

Consider adding a _checkIfWithdrawPeriod to totalAssets, which will prevent users from withdrawing when the vault manager is switching between strategies.

```
   function totalAssets() public view override returns (uint256) {
+     _checkIfWithdrawPeriod();
```

Resolution: We acknowledge this one but it is an intended behavior that users can withdraw when there are no active strategies.

Low Severity

[L-06] Vaults can be created with public API endpoints, but with lower feeMultiplier

The feeMultiplier is used throughout the system to apply fees to certain actions, which are then sent to the factory or, in the case of fulfill, split between the factory and the caller.

However, the feeMultiplier is hardcoded into the strategy during the creation of the vault when getAddressKeys is called. At this point, the fee multiplier can be 100% for normal URLs and 150% for public ones.

```
if (publicAPIEndPoints[apiInfo[i].url]) {
   feeMultiplier = 1_500_000;
}
```

If a URL is later added to the publicAPIEndPoints map, vaults created before this addition will still have the original fee multiplier, while vaults created afterward will have the new one. This creates a discrepancy where different vaults can be charged different fees depending on their creation time.

Recommendation:

Consider extracting feeMultiplier directly from the publicAPIEndPoints map, rather than hardcod- ing it into the strategy, to ensure consistent fees across all vaults.

Resolution: We acknowledge this but any public API will be added to the smart contract before its public.

Informational

[I-01] Use constants

In the buildChainlinkTradeRequest function, the last req._addInt call uses a hardcoded value:

```
req._addInt("multiplier", int256(10 ** 18));
```

However, in other parts of the code, requestDecimals is used, which also represents 1e18. Similarly, there are other instances where 1_000_000 is used instead of requestDecimals, leading to inconsis- tencies.

Recommendation:

Consider using requestDecimals instead of hardcoded values like 1e18 or 1_000_000 for consistency across the codebase.

```
req._add("path","totalnewCollateral;blockTimestamp");
-   req._addInt("multiplier", int256(10 ** 18));
+   req._addInt("multiplier", requestDecimals);
    return req;
```

Also replace all hardcoded 1_000_000 with an appropriate variable: uint256 public constant BIP = 1_000_000;

Resolution: Fixed

[I-02] Remove all unused variables

Some variables in the code are unused, which can be removed to simplify the code and reduce gas costs. These include: - `_lastMintTimestamp` - `indexToPercentPosition`

Additionally, specialRefer is hardcoded to an address but is later set in the constructor:

```
address public specialRefer = 0xB46838207D4CDc3b0F6d8862b8F0d29fee938051;
```

It is then assigned as:

```
specialRefer = startingInfo.factoryOwner;
```

Similarly, chainLinkToken and oracleAddress are used for vault initialization but are not set or used elsewhere:

```
address public chainLinkToken;
address public oracleAddress;
```

Recommendation:

Remove all unused variables and make specialRefer immutable.

```
   // Vault
-  mapping(address => uint256) private _lastMintTimestamp;
-  mapping(uint256 => uint256) public indexToPercentPosition;
-  address public specialRefer = 0xB46838207D4CDc3b0F6d8862b8F0d29fee938051;
+  address public immutable specialRefer;
   // Factory
-  address public oracleAddress;
-  address public chainLinkToken;
```

Resolution: Fixed