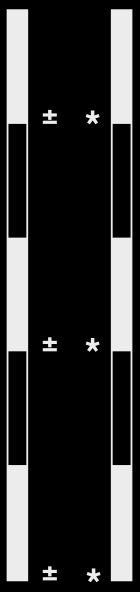
VERILOG

2023

REDACTED HIDDEN HAND MARIONETTE:

SECURITY REVIEW REPORT



Nov 10, 2023

Contents

1 | Introduction

Figure 1.1: Hidden Hand Marionette Report Cover

This report presents our engineering engagement with the Redacted-Cartel team on their veNFT wrapper protocol.

Project Name	**Hidden Hand Marionette**
Repository Link	https://github.com/redacted-cartel/hidden-hand-marionette
First Commit Hash	First: 273104d;
Final Commit Hash	Final: 7fa65d3;
Language	Solidity
Chain	Optimism

2 | About Verilog Solutions

Founded by a group of cryptography researchers and smart contract engineers in North America, Verilog Solutions elevates the security standards for Web3 ecosystems by being a full-stack Web3 security firm covering smart contract security, consensus security, and operational security for Web3 projects.

Verilog Solutions team works closely with major ecosystems and Web3 projects and applies a quality-above-quantity approach with a continuous security model. Verilog Solutions onboards the best and most innovative projects and provides the best-in-class advisory services on security needs, including on-chain and off-chain components.

3 | Service Scope

3.1 | Service Stages

Our auditing service includes the following two stages:

- Smart Contract Auditing Service

3.1.1 | Smart Contract Auditing Service

The Verilog Solutions team analyzed the entire project using a detailed-oriented approach to capture the fundamental logic and suggested improvements to the existing code. Details can be found under Findings And Improvement Suggestions.

3.2 | Methodology

- **Code Assessment**
  - We evaluate the overall quality of the code and comments as well as the architecture of the repository.
  - We help the project dev team improve the overall quality of the repository by providing suggestions on refactorization to follow the best practices of Web3 software engineering.
- **Code Logic Analysis**
  - We dive into the data structures and algorithms in the repository and provide suggestions to improve the data structures and algorithms for the lower time and space complexities.
  - We analyze the hierarchy among multiple modules and the relations among the source code files in the repository and provide suggestions to improve the code architecture with better readability, reusability, and extensibility.
- **Business Logic Analysis**
  - We study the technical whitepaper and other documents of the project and compare its specifications with the functionality implemented in the code for any potential mismatch between them.
  - We analyze the risks and potential vulnerabilities in the business logic and make suggestions to improve the robustness of the project.
- **Access Control Analysis**
  - We perform a comprehensive assessment of the special roles of the project, including their authorities and privileges.
  - We provide suggestions regarding the best practice of privilege role management according to the standard operating procedures (SOP).
- **Off-Chain Components Analysis**
  - We analyze the off-chain modules that are interacting with the on-chain functionalities and provide suggestions according to the SOP.
  - We conduct a comprehensive investigation for potential risks and hacks that may happen on the off-chain components and provide suggestions for patches.

3.3 | Audit Scope

Our auditing for Redacted Cartel covered the Solidity smart contracts under the folder 'solidity/contracts' in the repository (https://github.com/redacted-cartel/hidden-hand-marionette) with commit hash **273104d**.

4 | Project Summary

Marionette veNFT wrapper is an ERC721-based voting and reward management service, engineered to interface with various voting escrowed protocols seamlessly. The service offers bribe optimization, reward consolidation, automatic compounding functionality, and automatization of veNFTs functionalities, thereby improving user experience within these protocols.
Some of the main features include:

- Vote Optimization

  Execute voting strategy for multiple veNFTs aiming to maximize the bribe amount.

- Reward Consolidation opt-in

  Consolidate rewards from voting rounds in one token, reducing management complexity.

- Auto Compounding opt-in

  Swap rewards for the protocol's token to increase veNFTs voting power.

- Duration Management opt-in

  Automatically extend the lock period of the veNFTs to maintain or maximize voting power.

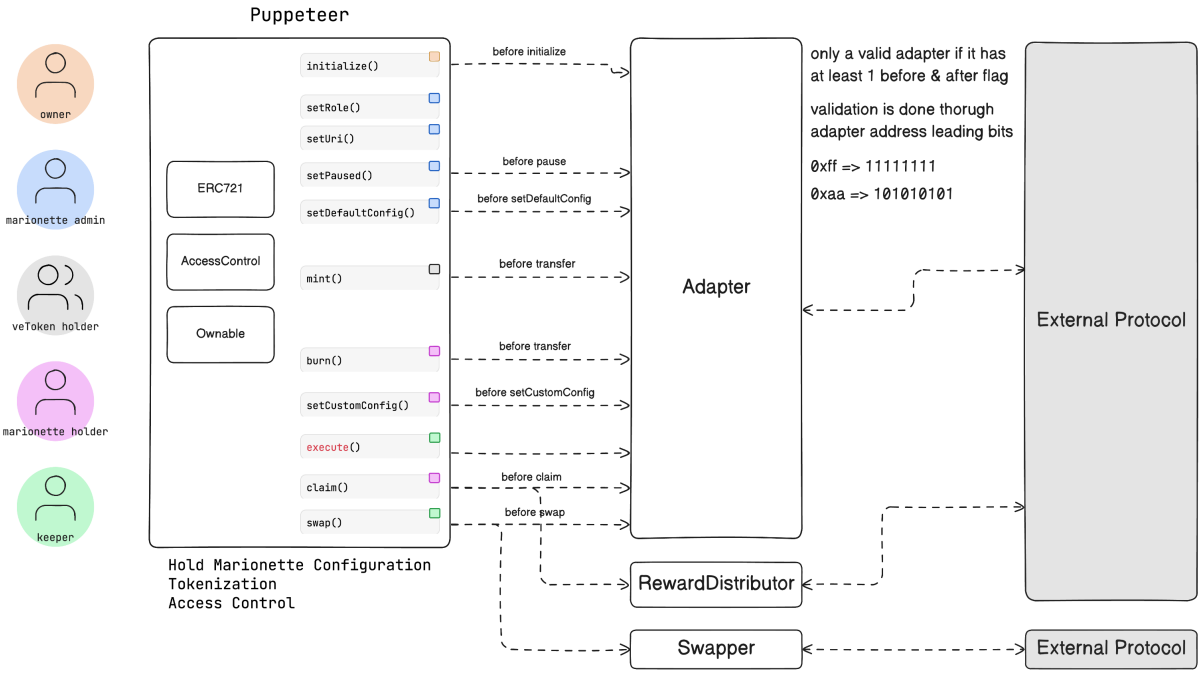


Figure 4.1: Hiidden Hand Marionette Architecture

5 | Findings and Improvement Suggestions

Severity	Total	Acknowledged	Resolved
High	2	2	2
Medium	2	2	2
Low	7	7	7
Informational	10	10	7

5.1 | High

5.1.1 | `VelodromeV2Adapter` should not allow users to set the reward mode as `Custom`

Severity	High
Source	solidity/contracts/adapters/velodrome-v2/VelodromeV2Adapter.sol#L165; solidity/contracts/adapters/velodrome-v2/VelodromeV2Adapter.sol#L227;
Commit	f14677f;
Status	Resolved in commit 2201ed6;

■ **Description**

There are four different reward modes and the `Custom mode` is not used in the `VelodromeV2Adapter`. In the `swap()` function, `totalModeEpochTokenToAmountCollected` gets updated with `Compound` mode or `Default` mode. For every epoch, `swap()` will be called twice, once with `Compound` mode and once with `Default` mode.

When the user claims rewards, the `_epochClaimableAmount()` function will be called. This function only considers the rewards under `Compound` or `Default` when the `fromToken` isn't the reward token. The reward will be zero for users with the `Custom` reward mode as the `swapAmountReceived` is zero.

Therefore, users will not be able to claim rewards with `Custom` as reward mode.

```
function _epochClaimableAmount(
  VeTokenInfo storage _veInfo,
  uint256 _epoch,
  address _token,
  bool _isCompound
) internal view returns (uint256 amount) {
  address[] storage rewards = epochRewardTokens[_epoch];
  uint256 rewardsLen = rewards.length;
  Marionette.RewardMode rm =
      _isCompound ? Marionette.RewardMode.Compound : Marionette.RewardMode.Default;

  for (uint256 i; i < rewardsLen; i++) {
    address fromToken = rewards[i];
    uint256 collected = _veInfo.epochTokenToAmountCollected[_epoch][fromToken];
    uint256 swapAmountReceived = epochTokenSwapAmountReceived[_epoch][fromToken][_token];

    // if there is no swap, just add the collected amount to the total claimable amount
    if (fromToken == _token) {
      amount += collected;
    } else if (swapAmountReceived > 0) {
      amount += collected * swapAmountReceived /
              totalModeEpochTokenToAmountCollected[rm][_epoch][fromToken];
    }
```

```
    }
}
```

- **Exploit Scenario**
  - Alice sets reward mode as `Custom`;
  - Alice tries to claim her rewards;
  - Alice won't get rewards since the `Custom` mode is not considered.
- **Recommendations**
  `VelodromeV2Adapter` should not allow users to set the reward mode as `Custom`.
- **Results**
  Resolved in commit 2201ed6.

  The `Custom` reward mode is no longer considered for the `VelodromeV2Adapter` contract.

5.1.2 | Swap fee issue

Severity	High
Source	solidity/contracts/adapters/velodrome-v2/Swapper.sol#L83;
Commit	f14677f;
Status	Resolved in commit 17ef73f;

■ **Description**

The `swap()` function compares the `fee` with the `receivedAmount`. Since the max swap fee rate is 10%, the `fee` can never be larger than the `receivedAmount`. The if condition will never be triggered. Thus the fee is never sent to the fee recipient.

```
function swap(address receiver, bytes calldata adapterData) external
      onlyOperator returns (bytes4) {
  (SwapData memory swapData) = abi.decode(adapterData, (SwapData));

  uint256 receivedAmount = _swap(swapData);

  uint256 fee = (receivedAmount * tokenSwapFee[swapData.toToken])
              / FEE_PERCENT_DENOMINATOR;

  if (fee > receivedAmount) {
    IERC20(swapData.toToken).safeTransfer(feeRecipient, fee);
  }
  IERC20(swapData.toToken).safeTransfer(receiver, receivedAmount - fee);

  emit Swap(swapData.fromToken, swapData.toToken, swapData.fromAmount,
            swapData.toAmount, receivedAmount);

  return ISwapper.swap.selector;
}
```

■ **Exploit Scenario**

- □ The fee and fee recipient is set in the `Swapper` contract;
- □ A swap is executed directly from the `Puppeteer` contract;
- □ The fee is not transferred to the fee recipient because that piece of code is never reached.

■ **Recommendations**

The fee should be compared with 0.

```
if (fee > 0) {
  IERC20(swapData.toToken).safeTransfer(feeRecipient, fee);
}
```

■ **Results**

Resolved in commit 17ef73f.

The suggestion was implemented.

5.2 | Medium

5.2.1 | The `claim()` function doesn't check if the marionette is `unpaused`

Severity	Medium
Source	solidity/contracts/Puppeteer.sol#L188-L203;
Commit	273104d;
Status	Resolved in commit 17ef73f;

- **Description**
  All the functions that perform actions on the marionettes check that they're unpaused by calling the `_onlyUnpaused()` function, except for the `claim()` function.
- **Exploit Scenario**
  Any user would be able to call the `claim()` function, even when the protocol is being paused.
- **Recommendations**
  Add check to the `claim()` function.
- **Results**
  Resolved in commit 17ef73f.

  The check was added.

5.2.2 | `_beforeTokenTransfer` in `Puppeteer` Design Discussion

Severity	Medium
Source	solidity/contracts/Puppeteer.sol#L294;;
Commit	273104d;
Status	Resolved in commit 66b84a0;

■ **Description**

The internal function `_beforeTokenTransfer` in `Puppeteer` has been overridden to implement the following logic:

```
function _beforeTokenTransfer(address from, address to, uint256 tokenId,
            bytes memory data) internal override {
    MarionetteKey memory key =
                    marionetteIdToMarionetteKey[tokenIdToMarionetteId[tokenId]];
    MarionetteId id = key.toId();
    uint256 veId = marionettes[id].marionette[tokenId].veId;

    // 1st if condition
    if (from == address(0) && veTokenSubscribed[key.protocol.veAddress][veId]) {
      revert AlreadySubscribed();
    }

    // 2nd if condition -> inner if `key.adapter.beforeTransfer`
    // is key for mint and burn function

    if (key.adapter.shouldCallBeforeTransfer()) {
      if (key.adapter.beforeTransfer(from, to, veId, data)
                != IAdapter.beforeTransfer.selector) {
        revert Adapter.InvalidAdapterResponse();
      }
    }
  }
```

The first if condition checks whether veId has already been subscribed or not. The second if condition should ideally always be triggered and turned on because the inner if condition, key.adapter.beforeTransfer, is essential when users trigger Mint and Burn operations. However, there are some concerns regarding `key.adapter.shouldCallBeforeTransfer`:

- For normal transfer transactions, if allowed, the second logic block should be bypassed since it results in an unnecessary external call.
- For normal transfer transactions, if not allowed, the second logic block will also limit user `Mint` and `Burn` operations in the current design. Ideally, the permission for normal transfers and `Mint/Burn` operations should be managed separately.
- For `Mint` and `Burn` transactions, `key.adapter.shouldCallBeforeTransfer` should always be active to enable their execution.

Furthermore, `key.adapter` uses the contract address as a condition to determine whether the call `key.adapter.shouldCallBeforeTransfer` is active or not. The project team intends to keep this mechanism closed, which may block `Mint` and `Burn` activities.

■ **Exploit Scenario**

N/A.

■ **Recommendations**

There are multiple ways to improve this issue, ideally:

- there is a logic that can be inserted in `key.adapter.shouldCallBeforeTransfer` to allow normal transfer TX bypass the rest of logic
- a simple switch can be placed in the adapter to trigger whether to allow normal transfer TX or not.

- **Results**
  Resolved in commit 66b84a0.

  The logic was simplified as the suggestion and the configuration is set to false now.

5.3 | Low

5.3.1 | `VeTokenInfo.marionetteId` is never used

Severity	Low
Source	solidity/contracts/adapters/velodrome-v2/VelodromeV2Adapter.sol#L31;
Commit	273104d;
Status	Resolved in commit 30a5213;

- **Description**
  The `marionetteId` field of the `veTokenInfo` is never used.
- **Exploit Scenario**
  N/A.
- **Recommendations**
  Set the `marionetteId` when user mints the marionette NFT.
- **Results**
  Resolved in commit 30a5213.

  The `marionetteId` element of the struct was removed.

5.3.2 | Inconsistency between function name and implementation

Severity	Low
Source	solidity/contracts/Puppeteer.sol#L314-L316;
Commit	273104d;
Status	Resolved in commit 031d1a7f;

- **Description**
  The `_onlyUnpaused()` internal function checks whether the marionette is uninitialized or unpaused. The name is misleading since this modifier also checks initialization.

```
function _onlyUnpaused(MarionetteId id) internal view {
  if (marionettes[id].initialized && marionettes[id].paused)
          revert PausedMarionette(); // initialized
}
```

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Change the name of the `_onlyUnpaused()` modifier to `_onlyUnitializedOrUnpaused()`.

- **Results**
  Resolved in commit 031d1a7f.

  The name of the modifier was changed and the logic was corrected.

5.3.3 | Lack of event emission for critical operations

Severity	Low
Source	solidity/contracts/Puppeteer.sol#L177; solidity/contracts/Puppeteer.sol#L188; solidity/contracts/Puppeteer.sol#L206;
Commit	273104d;
Status	Resolved in commit cfc83cf;

- **Description**
  Events are vital aids in monitoring contracts and detecting suspicious behavior. The `execute()`, `claim()`, and `swap()` are functions that perform state changes, therefore they should consider the emission of an event.
- **Exploit Scenario**
  N/A.
- **Recommendations**
  Add events to functions that produce state changes.
- **Results**
  Resolved in commit cfc83cf.

  The events were added.

5.3.4 | Add non-reentrant modifiers to the `mint()`, `claim()` and `burn()` functions

Severity	Low
Source	solidity/contracts/Puppeteer.sol#L137; solidity/contracts/Puppeteer.sol#L163; solidity/contracts/Puppeteer.sol#L188;
Commit	273104d;
Status	Resolved in commit 7fa65d3;

- **Description**
  The `mint()` and `burn()` functions perform external calls to the adapter before updating the `veTokenSubscribed` mapping. Since there could be other adapters in the future that also perform other external calls, we recommend adding non-reentrant modifiers to these functions.

  For the mint() function, we have the following:

```
_mint(msg.sender, tokenId, data);
veTokenSubscribed[key.protocol.veAddress][veId] = true;
```

  While the following code is in the burn() function:

```
_burn(tokenId, adapterData);
veTokenSubscribed[key.protocol.veAddress];
marionettes[id].marionette[tokenId].veId = false;
```

  Additionally, we recommend adding a non-reentrant modifier to the claim() function because it is a user entry-point.

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Add non-reentrant modifiers to the specified functions.

- **Results**
  Resolved in commit 7fa65d3.

  The suggestion was implemented.

5.3.5 | Mapping `tokenIdToMarionetteId` is not cleared when the user burns the marionette NFT

Severity	Low
Source	solidity/contracts/Puppeteer.sol#L39;
Commit	273104d;
Status	Resolved in commit 08b17450;

- **Description**
  A single marionette NFT contract is used to mint marionette NFT for different marionettes. The mapping `tokenIdToMarionetteId` maps the marionette NFT token id to marionette id. However, this mapping is not cleared when the marionette NFT gets burned.
- **Exploit Scenario**
  N/A.
- **Recommendations**
  Clear the `tokenIdToMarionetteId` mapping when the user burns the marionette NFT.
- **Results**
  Resolved in commit 08b17450.

  The suggestion was implemented.

5.3.6 | `VelodromeV2Adapter.RebaseMany()` should check whether the `veToken` is active

Severity	Low
Source	solidity/contracts/adapters/velodrome-v2/VelodromeV2Adapter.sol#L404;
Commit	273104d;
Status	Resolved in commit 421f948;

- **Description**
  `VelodromeV2Adapter.RebaseMany()` calls the `claimMany()` function in the Velodrome reward distributor. This function does not check veNFT ownership and will transfer the rewards to veNFT owner directly. The keeper might claim for others. Therefore, we should check if the veNFT is active or not to save the gas.
- **Exploit Scenario**
  N/A.
- **Recommendations**
  Check if the veToken is active in `VelodromeV2Adapter.RebaseMany()`.
- **Results**
  Resolved in commit 421f948.

  The suggestion was implemented.

5.3.7 | Infinite Approval of VELO from Adapter to VE contract

Severity	Low
Source	solidity/contracts/adapters/velodrome-v2/VelodromeV2Adapter.sol#L159;
Commit	273104d;
Status	Resolved in commit fa7629d;

- **Description**
  During the initialization of Puppeteer.sol, function beforeInitialize will be triggered from VelodromeV2Adapter.sol which contains a logic of Infinite Approval of VELO from Adapter to VE contract. In general, infinite approval to a smart contract is not recommended as this will extend the security vulnerability attack surfaces.

```
/// @inheritdoc IAdapter
/// @dev set default gauges and weights, default reward token, swapper,
//and minimum balance of the veNFT
function beforeInitialize(
  address,
  Marionette.VoteConfig memory _defaultVoteConfig,
  Marionette.RewardConfig memory _defaultRewardConfig,
  Marionette.LockConfig memory,
  bytes calldata _adapterData
) external override onlyPuppeteer returns (bytes4) {
  // set default vote config
  _setGaugesAndWeights(_defaultVoteConfig.gauges, _defaultVoteConfig.weights);
  // set default reward config
  _setDefaultRewardToken(_defaultRewardConfig.defaultRewardToken);
  // set swapper address
  if (_defaultRewardConfig.data.length > 0) {
    (address _swapper) = abi.decode(_defaultRewardConfig.data, (address));
    _setSwapper(_swapper);
  }
  // set minBalanceOfToken
  (uint256 _minBalanceOfToken) = abi.decode(_adapterData, (uint256));
  minBalanceOfToken = _minBalanceOfToken;
  // add VELO to rewards
  rewardTokens.push(VELO);
  // approve VE to VELO
  IERC20(VELO).approve(address(VE), type(uint256).max); // ** -> this step,
  // also the commeent needs to change to approve VELO to VE

  return IAdapter.beforeInitialize.selector;
}
```

- **Exploit Scenario**
  Potential vulnerability in the VE contract will lead to fund loss in `VelodromeV2Adapter.sol`.

- **Recommendations**
  Trigger approval with the needed amount each time to reduce the potential risks from VE contract exploit.

- **Results**
  Resolved in commit fa7629d.

  The `approveVelo()` function was added to modify the approved amount.

5.4 | Informational

5.4.1 | Floating solidity pragma version

Severity	Informational
Source	Global;
Commit	273104d;
Status	Resolved in commit 377281b;

- **Description**
  Current smart contracts use ^0.8.20. And compilers within versions ≥ 0.8.20 and <0.9.0 can be used to compile those contracts. Therefore, the contract may be deployed with a newer or latest compiler version which generally has higher risks of undiscovered bugs.

  It is a good practice to fix the solidity pragma version if the contract is not designed as a package or library that will be used by other projects or developers.

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Use the fixed solidity pragma version.

- **Results**
  Resolved in commit 377281b.

  The compiler version was fixed to 0.8.22.

5.4.2 | Unused custom error

Severity	Informational
Source	solidity/contracts/AccessControl.sol#L11;
Commit	273104d;
Status	Resolved in commit 1ffe411;

- **Description**
  The `AlreadyAdmin()` error is defined in the `AccessControl` contract but is never used within or in any inherited contracts.
- **Exploit Scenario**
  N/A.
- **Recommendations**
  Remove the `AlreadyAdmin()` custom error.
- **Results**
  Resolved in commit 1ffe411.

  The custom error was removed.

5.4.3 | Unused imports

Severity	Informational
Source	contracts/interfaces/IRewardDistributor.sol#L4; contracts/interfaces/IAdapter.sol#L4; solidity/contracts/adapters/velodrome-v2/VelodromeV2Adapter.sol#L17-L18;
Commit	273104d;
Status	Resolved in commit 762fde8;

- **Description**
  The `IRewardDistributor` and `IAdapter` interfaces import the `MarionetteKey` structs but they are not used within the interfaces. Similarly, the `Ownable` imports in the VelodromeV2Adapter contract are not used.
- **Exploit Scenario**
  N/A.
- **Recommendations**
  Remove the `MarionetteKey` structs import from the interfaces that don't need it and the `Ownable` import from the `VelodromeV2Adapter` contract.
- **Results**
  Resolved in commit 762fde8.

  The unused imports were removed.

5.4.4 | Duplicate `ISwapper` interface

Severity	Informational
Source	solidity/contracts/interfaces/ISwapper.sol; solidity/contracts/adapters/velodrome-v2/interfaces/ISwapper.sol;
Commit	273104d;
Status	Resolved in commit f0e4fa1;

- **Description**
  There are two `ISwapper` interfaces and in fact, they are both imported in the `VelodromeV2Adapter` contract:

```
port {ISwapper as IVeloSwapper} from "./interfaces/ISwapper.sol";
port {ISwapper} from "contracts/interfaces/ISwapper.sol";
```

  In solidity, interfaces represent a set of functions that can be implemented. Therefore, there should only be one ISwapper that is being used for every adapter. Right now, the only difference between these two interfaces is a struct that could be defined directly in the `VelodromeV2Adapter` contract:

```
struct SwapData {
  address fromToken;
  address toToken;
  uint256 fromAmount;
  uint256 toAmount;
  uint256 deadline;
  address[] callees;
  uint256[] callLengths;
  uint256[] values;
  bytes exchangeData;
}
```

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Only define one interface and customize the functions and definition of variables in every adapter.

- **Results**
  Resolved in commit f0e4fa1.

  The duplicated interface was removed.

5.4.5 | `delete` has no effects on mappings

Severity	Informational
Source	solidity/contracts/adapters/velodrome-v2/VelodromeV2Adapter.sol#L80;
Commit	273104d;
Status	Acknowledged;

- **Description**
  Struct `VeTokenInfo` contains mappings. When we delete the struct, the information stored in those mappings will not be cleaned.

```
struct VeTokenInfo {
  bool active;
  uint256 marionetteId;
  uint256 initialEpoch;
  uint256 lastTimestampExtended;
  Marionette.RewardMode rewardMode;
  Marionette.LockMode lockMode;
  mapping(address => uint256) tokenLastEpochClaimed;
  mapping(uint256 => mapping(address => uint256)) epochTokenToAmountCollected;
}
```

- **Exploit Scenario**
  N/A.

- **Recommendations**
  We would like to discuss this with the team.

- **Results**
  Acknowledged.

  Response from Redacted team:
  "No action is required. The protocol ensures all outstanding rewards are claimed upon withdrawal. Subsequent deposits of the same `tokenId` do not impact the claimable amount, hence the mappings in `VeTokenInfo` struct do not pose a risk"

5.4.6 | `beforeSetCustomConfig()` does not update `veInfo.lastTimestampExtended` when switching lock mode from non-Max to Max

Severity	Informational
Source	solidity/contracts/adapters/velodrome-v2/VelodromeV2Adapter.sol#L248-L259;
Commit	273104d;
Status	Acknowledged;

- **Description**
  Based on the if-else condition `else if (veInfo.lockMode == Marionette.LockMode.Maintain)`, when the old lock mode is Maintain, the `veInfo.lastTimestampExtended` should be updated.

  However, the `veInfo.lastTimestampExtended` isn't updated when the condition is `veInfo.lockMode != Marionette.LockMode.Max & & _lockConfig.lockMode == Marionette.LockMode.Max`. It switches the lock mode from non-Max to Max and the non-max mode includes Maintain.

```
if (veInfo.lockMode == Marionette.LockMode.Max &&
      _lockConfig.lockMode != Marionette.LockMode.Max) {
  _lockPermanent(_tokenId, false);
  veInfo.lockMode = _lockConfig.lockMode;
} else if (veInfo.lockMode != Marionette.LockMode.Max &&
      _lockConfig.lockMode == Marionette.LockMode.Max) {
  _lockPermanent(_tokenId, true);
  veInfo.lockMode = _lockConfig.lockMode;
} else if (veInfo.lockMode == Marionette.LockMode.Maintain) {
  veInfo.lastTimestampExtended = block.timestamp;
  veInfo.lockMode = _lockConfig.lockMode;
} else {
  veInfo.lockMode = _lockConfig.lockMode;
}
```

- **Exploit Scenario**
  N/A.

- **Recommendations**
  We would like to discuss this with the team.

- **Results**
  Acknowledged.

  Response from Redacted team:
  "`lastTimestampExtended` is only relevant when switching to `Maintain` as `Max` uses `lockPermanent()` method."

5.4.7 | Suggest to only update `epochTokenSwapAmountReceived` when `receivedAmount` is not zero

Severity	Informational
Source	solidity/contracts/adapters/velodrome-v2/VelodromeV2Adapter.sol#L320;
Commit	273104d;
Status	Resolved in commit ae9cd68;

- **Description**
  The `swap()` function updates the `epochTokenSwapAmountReceived` based on the received amount after the swap. However, the received amount could be zero. We should only update `epochTokenSwapAmountReceived` when `receivedAmount` is not zero to avoid unnecessary operations.
- **Exploit Scenario**
  N/A.
- **Recommendations**
  Only update `epochTokenSwapAmountReceived` when `receivedAmount` is not zero to avoid unnecessary operations.
- **Results**
  Resolved in commit ae9cd68.

  The suggestion was implemented.

5.4.8 | Unchecked return values

Severity	Informational
Source	solidity/contracts/adapters/velodrome-v2/VelodromeV2Adapter.sol#L138; solidity/contracts/adapters/velodrome-v2/VelodromeV2Adapter.sol#L312; solidity/contracts/adapters/velodrome-v2/VelodromeV2Adapter.sol#L411;
Commit	273104d;
Status	Acknowledged;

- **Description**
  The functions specified in the source section make external calls to functions that return a boolean. This value should be checked because if the return value is false, that means that the intended action didn't execute successfully.

- **Exploit Scenario**
  - Take as an example the call made to the `approve()` function in the `beforeInitialize()` function;
  - Let's say the approve() function returned a false value;
  - The approval wasn't successful but the `beforeInitialize()` function call was.

- **Recommendations**
  Add `require()` statements to the specified external calls.

- **Results**
  Acknowledged.

  Response from Redacted team:
  "No action required. VELO is a standard ERC20 and we check balances before and after swapping, this way we can set a new swapper without needing a return amount. Additionally, the `claimMany()` function will only return true."

5.4.9 | Inconsistent comment with implementation

Severity	Informational
Source	solidity/contracts/adapters/velodrome-v2/VelodromeV2Adapter.sol#L525;
Commit	273104d;
Status	Resolved in commit 3a78eec;

- **Description**
  The comment in the line specified states that the extension applies to any `tokenId` with lock mode other than `None`. However, it checks for only the lock mode `Maintain`:

```
function extendMany(uint256[] memory _tokenIds) public onlyPuppeteer {
  uint256 tokenIdsLength = _tokenIds.length;

  // Loop through each token ID and extend the lock duration if the lock mode is not None
  for (uint256 i; i < tokenIdsLength; i++) {
    if (veTokenInfo[_tokenIds[i]].lockMode == Marionette.LockMode.Maintain) {
      _increaseUnlockTime(_tokenIds[i]);
    }
  }
}
```

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Change the comment to state that the extension only applies to the `Maintain` lock mode.

- **Results**
  Resolved in commit 3a78eec.

  The comment was fixed.

5.4.10 | Unnecessary function `shouldCallBeforeInitialize()` in `adapter.sol`

Severity	Informational
Source	solidity/contracts/Puppeteer.sol#L71;
Commit	273104d;
Status	Resolved in commit 8bf3474;

■ **Description**

The `key.adapter.beforeInitialize()` function within the inner loop will trigger the setup and config of `VelodromeV2Adapter.sol` contract and as this function can only be invoked through `Puppeteer.sol`. Thus, The function `shouldCallBeforeInitialize()` should always return `TRUE` during the initialization stage. Therefore, we think this function and its relevant flag variables can be removed as it is a must-do step.

```
/// function initialize in Pupeeter.sol
function initialize(
  MarionetteKey memory key,
  string calldata uri,
  Marionette.VoteConfig calldata defaultVoteConfig,
  Marionette.RewardConfig calldata defaultRewardConfig,
  Marionette.LockConfig calldata defaultLockConfig,
  bytes calldata adapterData
) external onlyOwner {
  MarionetteId id = key.toId();
  if (marionettes[id].initialized) revert AlreadyInitialized();

  if (key.adapter.shouldCallBeforeInitialize()) { // ->
        // not necessary as below functions must needed to be called
    if (
      key.adapter.beforeInitialize(msg.sender, defaultVoteConfig,
            defaultRewardConfig, defaultLockConfig, adapterData)
        != IAdapter.beforeInitialize.selector
    ) {
      revert Adapter.InvalidAdapterResponse();
    }
  }

  marionetteIdToMarionetteKey[id] = key;
  marionettes[id].initialize(uri, defaultVoteConfig,
        defaultRewardConfig, defaultLockConfig);
  _grantRole(id, MARIONETTE_ADMIN_ROLE, msg.sender);

  emit Initialize(key, id, uri, msg.sender, defaultVoteConfig, defaultRewardConfig,
        defaultLockConfig, adapterData);
}

//// function beforeInitialize in VelodromeV2Adapter.sol
function beforeInitialize(
  address,
  Marionette.VoteConfig memory _defaultVoteConfig,
  Marionette.RewardConfig memory _defaultRewardConfig,
  Marionette.LockConfig memory,
  bytes calldata _adapterData
) external override onlyPuppeteer returns (bytes4) {  // function is onlyPuppeteer
........
```

■ **Exploit Scenario**

N/A.

- **Recommendations**
  Remove function `shouldCallBeforeInitialize()` in `adapter.sol`.
- **Results**
  Resolved in commit 8bf3474.

  The suggestion was implemented.

6 | Use Case Scenarios

Users of the Marionette veNFT Wrapper can subscribe to the service by depositing or approving the Marionette adapter with their veNFT. As for the Velodrome adapter, depending on the configuration that they choose, there are the following cases:

- If they choose the `VoteOnly` reward mode, they will allow the protocol to execute the current voting strategy for their veNFT. In this case, users don't need to deposit the veNFT into the adapter contract.
- If they choose the `Default` reward mode, they will allow the protocol to consolidate rewards, but users ultimately have to claim them. In this case, users have to deposit their veNFT into the adapter contract.
- If they choose the `Compound` reward mode, they will allow the protocol to claim, consolidate, and auto-compound rewards. In this case, users have to deposit their veNFT into the adapter contract.
- If they choose the `Maintain` lock mode. they will allow the protocol to extend the lock period of their veNFT.

Regardless of the choice of the users, the keeper will trigger functions to claim, extend lock periods, and consolidate rewards. However, only the users who choose the corresponding configuration will see the respective changes considered for their veNFT.

7 | Access Control Analysis

There are three main privileged roles within the Marionette veNFT wrapper protocol. The main actions that can be performed by each role are described below:

7.1 | Owner

- Initialize a new Marionette Key and grant the MARIONETTE_ADMIN_ROLE.
- Transfer ownership functionality.

7.2 | MARIONETTE_ADMIN_ROLE

- Set the Marionette Key default configuration (reward, vote, and lock).
- Set and unset the paused state.
- Set the URI of the Marionette Key.

7.3 | KEEPER_ROLE

- Call the `execute()` function in the adapter.
- Call the `swap()` function implemented in the adapter.

8 | Appendix

8.1 | Appendix I: Severity Categories

Severity	Description
High	Issues that are highly exploitable security vulnerabilities. It may cause direct loss of funds / permanent freezing of funds. All high severity issues should be resolved.
Medium	Issues that are only exploitable under some conditions or with some privileged access to the system. Users' yields/rewards/information is at risk. All medium severity issues should be resolved unless there is a clear reason not to.
Low	Issues that are low risk. Not fixing those issues will not result in the failure of the system. A fix on low severity issues is recommended but subject to the clients' decisions.
Information	Issues that pose no risk to the system and are related to the security best practices. Not fixing those issues will not result in the failure of the system. A fix on informational issues or adoption of those security best practices-related suggestions is recommended but subject to clients' decision.

8.2 | Appendix II: Status Categories

Severity	Description
Unresolved	The issue is not acknowledged and not resolved.
Partially Resolved	The issue has been partially resolved
Acknowledged	The Finding / Suggestion is acknowledged but not fixed / not implemented.
Resolved	The issue has been sufficiently resolved

9 | Disclaimer

Verilog Solutions receives compensation from one or more clients for performing the smart contract and auditing analysis contained in these reports. The report created is solely for Clients and published with their consent. As such, the scope of our audit is limited to a review of code, and only the code we note as being within the scope of our audit is detailed in this report. It is important to note that the Solidity code itself presents unique and unquantifiable risks since the Solidity language itself remains under current development and is subject to unknown risks and flaws. Our sole goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies. Thus, Verilog Solutions in no way claims any guarantee of security or functionality of the technology we agree to analyze.
In addition, Verilog Solutions reports do not provide any indication of the technology's proprietors, business, business model, or legal compliance. As such, reports do not provide investment advice and should not be used to make decisions about investment or involvement with any particular project. Verilog Solutions has the right to distribute the Report through other means, including via Verilog Solutions publications and other distributions. Verilog Solutions makes the reports available to parties other than the Clients (i.e., "third parties") – on its website in hopes that it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.