PUBLIC

Code Assessment

of the Verified ERC20 Smart Contracts

June 30, 2025

Produced for

velodrome

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Velodrome team,

Thank you for trusting us to help Velodrome with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Verified ERC20 according to Scope to support you in forming an opinion on their security risks.

Velodrome implements the VerifiedERC20 system, which provides enhanced ERC20 tokens with pluggable hook functionality. The system is designed to be modular, such that new hooks can be later added, and comes with a set of hooks already implemented.

The most critical subjects covered in our audit are functional correctness and access control. Security regarding all the aforementioned subjects is high.

The general subjects covered are error handling, gas efficiency, and correct integration with the CreateX factory. Security regarding all the aforementioned subjects is high.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Verified ERC20 repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	12 June 2025	0784bf938facf33fb04a470956b4ac7cdfc057db	Initial Version
2	24 June 2025	79e88c113be4c7642fa82782b3d6ef705cefae09	Fixes

For the solidity smart contracts, the compiler version `0.8.27` was chosen.

The following files were included in the scope of the assessment:

```
src/external/ERC20Lockbox.sol
src/hooks/BaseHook.sol
src/hooks/BaseTransferHook.sol
src/hooks/extensions/SelfTransferHook.sol
src/hooks/extensions/SinglePermissionHook.sol
src/hooks/extensions/AutoUnwrapHook.sol
src/hooks/HookRegistry.sol
src/libraries/CreateXLibrary.sol
src/VerifiedERC20.sol
src/VerifiedERC20Factory.sol
```

2.1.1 Excluded from scope

Anything outside the scope of the above files, such as the test files, documentation, and other files in the repository, was not included in the assessment.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

The `VerifiedERC20` system provides enhanced ERC20 tokens with pluggable hook functionality. The system allows for customizable token behavior through a registry-based hook system, enabling features like user verification, automatic unwrapping, and permission controls.

2.2.1 Core Contracts

2.2.1.1 VerifiedERC20

The VerifiedERC20 contract extends the standard ERC20 token with a hook architecture that enables customizable behavior during token operations. It implements a hook system that allows up to 8 hooks per entry-point (16 per operation type: 8 before the operation, 8 after), each with controlled gas limits. The contract uses OpenZeppelin components for reentrancy protection and proxy compatibility, supporting the factory deployment pattern which uses minimal proxy contracts for efficient token creation.

Main Entry Points:

- `initialize(name, symbol, owner, hookRegistry, hooks)`: Initializes the token with basic parameters and hooks
- Usual ERC20 functions with hook validation:
    - `mint(account, value)`: Mints tokens
    - `burn(account, value)`: Burns tokens
    - `approve(spender, value)`: Approves spending
    - `transfer(to, value)`: Transfers tokens
    - `transferFrom(from, to, value)`: Transfers tokens on behalf of another account

Available Hook Entrypoints:

The VerifiedERC20 contract supports the following hook entry points for customizable behavior:

- BEFORE_MINT
- AFTER_MINT
- BEFORE_BURN
- AFTER_BURN
- BEFORE_APPROVE
- AFTER_APPROVE
- BEFORE_TRANSFER
- AFTER_TRANSFER

Restricted Functions:

The VerifiedERC20 contract's owner can manage hooks and their activation. A hook can only be activated if it is registered in the HookRegistry. The entry point for the hook must be set within the HookRegistry by the owner of the registry. There can be a maximum of 8 hooks per entry point, and each hook execution is limited to 200,000 gas to prevent excessive resource consumption.

- `activateHook(hook)`
- `deactivateHook(hook)`

2.2.1.2 VerifiedERC20Factory

The VerifiedERC20Factory acts as the deployment hub for creating new VerifiedERC20 tokens using CREATE2 minimal proxies through the CreateX library. The contract is fully permissionless, and maintains a registry of all deployed tokens, providing discovery mechanisms.

Main Entry Points:

- `deployVerifiedERC20(name, symbol, owner, hooks)`: Deploys new VerifiedERC20 instance with specified parameters

2.2.1.3 HookRegistry

The HookRegistry serves as the central authority for the hook ecosystem, ensuring that only validated hooks can be integrated into VerifiedERC20 tokens. Operating under an ownership model, it maintains a registry of hooks with their supported entry points, providing centralized validation and storage. This registry acts as the source of truth for hook legitimacy, preventing unauthorized hooks from being used in token operations.

Main Entry Points:

Both functions are restricted to the owner of the registry, and allow the owner to manage the hook ecosystem:

- registerHook(hook, entrypoint): Registers a new hook (owner only)
- deregisterHook(hook): Removes a hook from the registry (owner only)

2.2.2 Hook System

2.2.2.1 Base Hook Contracts

2.2.2.1.1 BaseHook

BaseHook provides the foundational framework for implementing hooks that interact with mint, burn, and approve operations in the VerifiedERC20 ecosystem. This abstract contract handles parameter extraction and validation for two-parameter operations, allowing developers to implement their specific logic without handling data parsing and interface compliance.

2.2.2.1.2 BaseTransferHook

BaseTransferHook serves as the foundation for hooks that intercept and process transfer operations, which involve three parameters representing the sender, recipient, and amount. By abstracting parameter decoding and providing an interface for transfer-specific logic, this contract enables developers to build transfer hooks such as compliance checks, automatic conversions, or conditional routing.

2.2.2.2 Hook Extensions

2.2.2.2.1 SinglePermissionHook

SinglePermissionHook implements an authorization mechanism that restricts token operations to designated addresses, providing access control for VerifiedERC20 tokens. This hook is used for scenarios where tokens need controlled issuance and destruction by ensuring that only pre-authorized addresses can perform mint and burn operations while allowing the token owner to delegate these permissions.

Using the function setAuthorized, the token owner can set a single authorized address for minting or burning operations (depending on how the hook was configured in the registry). If a token wants to control both minting and burning, it should use two instances of this hook, one for each operation.

Supported Entrypoints:

- BEFORE_MINT: Validates authorization before minting
- BEFORE_BURN: Validates authorization before burning

Main Entry Points:

2.2.2.2.2 SelfTransferHook

SelfTransferHook provides user verification capabilities by integrating with the Self.xyz protocol's identity verification system, ensuring that only verified users can claim incentive rewards. By detecting reward claim transactions and cross-referencing them with Self Passport SBT verification status, this hook enables reward distribution based on the Self identity system.

Supported Entrypoints:

- BEFORE_TRANSFER: Validates user verification before transfers

2.2.2.2.3 AutoUnwrapHook

AutoUnwrapHook automatically converts VerifiedERC20 tokens back to their underlying base tokens when users claim incentive rewards. This hook eliminates manual unwrapping steps in reward-claiming workflows, ensuring users receive standard token format.

Using the function setLockbox, the token owner can set the correct lockbox address for their VerifiedERC20 for automatic unwrapping.

Supported Entrypoints:

- AFTER_TRANSFER: Automatically unwraps tokens after transfers

2.2.3 External Contracts

2.2.3.1 ERC20Lockbox

ERC20Lockbox enables transitions between standard ERC20 tokens and their VerifiedERC20 counterparts, maintaining a 1:1 backing relationship.

Main Entry Points:

- deposit(amount): Locks ERC20 tokens and mints VerifiedERC20
- withdraw(amount): Burns VerifiedERC20 and releases ERC20 tokens
- withdrawTo(to, amount): Burns VerifiedERC20 and sends ERC20 to specified address

2.2.4 Libraries

2.2.4.1 CreateXLibrary

CreateXLibrary provides utility functions for CREATE2 deployments within the VerifiedERC20 system. It calculates salts to provide to the CreateX library, enabling the guarded salt feature. And computes addresses for CREATE3 deployments.

Key Features:

- Provides guarded salt calculation.
- Supports CREATE3 address computation

2.2.5 System Flow

1. **Deployment**: VerifiedERC20Factory deploys new token instances with specified hooks
2. **Hook Registration**: HookRegistry owner registers available hooks with their entrypoints
3. **Hook Activation**: Each VerifiedERC20 token adds hooks from the registry
4. **Token Operations**: VerifiedERC20 tokens call registered hooks during operations
5. **Hook Execution**: Hooks perform validation/modification based on their specific logic

2.2.6 Changes in Version 2

Version 2 introduces the following update, in addition to addressing findings from Version 1:

- The `SelfTransferHook` and `AutoUnwrapHook` hooks now use `IReward.authorized()` instead of `IReward.DURATION()` to identify incentive claim transactions. This adjustment improves compatibility with Velodrome Gauges, which implement a similar interface to the `Reward` contract (including `voter()` and `DURATION()`).

2.3 Trust Model

2.3.1 Roles

HookRegistry Owner

- **Trust Level**: Full system trust
- **Responsibilities**:
  - Register and deregister hooks in the `HookRegistry`
  - Maintain security of the hook ecosystem
  - Validate hook implementations before registration
  - Validate hook configuration made by the constructor before registration
- **Powers**: Can control which hooks are available system-wide
- **Risk**: Complete control over hook availability; malicious hooks could compromise all tokens if activated by token owners

Token Owner (VerifiedERC20 Owner)

- **Trust Level**: Per-token trust
- **Responsibilities**:
  - Activate and deactivate hooks for their specific token
  - Manage token-specific configurations
  - Ensure hooks are compatible with each other and the token's use case
- **Powers**: Control hook activation for individual tokens
- **Risk**: Can modify token behavior through hook selection; cannot introduce new hooks

Users

- **Trust Level**: No special trust required
- **Responsibilities**: Use only tokens that have a coherent and secure set of hooks activated
- **Powers**: Standard ERC20 operations subject to hook restrictions

2.3.2 Trust Assumptions

1. The system is expected to operate on chains with the Cancun hard fork or later, which supports `TSTORE` and `TLOAD`.
2. ERC20 tokens to be wrapped in `VerifiedERC20` are assumed to be ERC-compliant tokens that are not malicious and present no specific risks or unusual behaviors (e.g., rebasing, transferring a different amount than requested, fee-on-transfer, double entry points, non-compliant interface, or hooks)

3. **HookRegistry Owner** is assumed to be a trusted entity that will not register malicious hooks
4. **Token Owners** are trusted to select appropriate hooks for their use case
5. **Hook implementations** are assumed to be audited and secure before registration, they are trusted to operate correctly and not DOS-attack the system by consuming excessive gas or reverting. They also are trusted to reenter the `VerifiedERC20` contract only for legitimate purposes, such as performing additional token transfers or checks.
6. **External dependencies** Self Passport is trusted to operate correctly. That is, it is assumed that `getTokenIdByAddress` and `isTokenValid` do not revert and that they return the expected values.
7. **ERC20Lockbox** is assumed to be given mint and burn permissions for the `VerifiedERC20` token they are defined for.
8. The **Reward** contracts are assumed to define both the `authorized()` and `voter()` functions, which should both always consume less than 5,000 gas each. The values returned by these functions are assumed to be constant over time and always match the immutables defined in the hook contracts: `authorized` and `voter` respectively.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0
Informational Findings	7

- Event Fields Not Indexed Code Corrected
- Gas Savings Code Corrected
- Magic Value for address[][] Initialization Code Corrected
- Missing AuthorizedSet Event Code Corrected
- Missing Sanity Check Code Corrected
- State Variables Return Misleading Values Specification Changed
- Unused Constant MAX_GAS_PER_HOOK Code Corrected

6.1 Event Fields Not Indexed

Informational Version 1 Code Corrected

CS-VELO-VERC20-002

The `Deposit` and `Withdraw` events in the `IERC20Lockbox` interface have no indexed fields. Indexing fields would improve event filtering and searching capabilities.

Code Corrected:

The `_sender` and `_receiver` arguments are now indexed.

6.2 Gas Savings

Informational Version 1 Code Corrected

CS-VELO-VERC20-004

The following gas optimizations were identified in the codebase:

1. In `VerifiedERC20._activateHook()`, the `hookRegistry` could be cached in memory to avoid repeated storage reads.
2. In `VerifiedERC20`, the `transfer` and `transferFrom` functions simply call the base class respective functions. These could be removed, as they do not add any additional logic.

3. In `HookRegistry`, the contract uses a reentrancy lock but does not make any external non-static calls.

Code Corrected:

All the above gas savings have been implemented in the codebase.

6.3 Magic Value for `address[][]` Initialization

Informational | Version 1 | Code Corrected

CS-VELO-VERC20-006

In the `initialize` function of the `VerifiedERC20` contract, the `address[][]` array `_hooksByEntrypoint` is initialized using a hardcoded value of `8`. This assumes there are exactly 8 types of entry points. If the number of entry point types changes in the future, this value will be incorrect.

Code Corrected:

The hardcoded value 8 has been replaced with a `MAX_ENTRYPOINTS` constant.

6.4 Missing AuthorizedSet Event

Informational | Version 1 | Code Corrected

CS-VELO-VERC20-007

The constructor in the `SinglePermissionHook` contract does not emit the `AuthorizedSet` event after setting the authorized address for each respective `VerifiedERC20` address.

Code Corrected:

The constructor now calls the internal `_setAuthorized` function, which properly emits the `AuthorizedSet` event when setting authorized addresses during contract initialization.

6.5 Missing Sanity Check

Informational | Version 1 | Code Corrected

CS-VELO-VERC20-008

The constructors of `AutoUnwrapHook` and `SinglePermissionHook` do not check if the provided two arrays have the same length.

Code Corrected:

The constructors now validate that the input arrays have matching lengths. If the arrays have different lengths, the contracts revert with custom errors: `AutoUnwrapHook_LengthMismatch()` for the `AutoUnwrapHook` contract and `SinglePermissionHook_LengthMismatch()` for the `SinglePermissionHook` contract.

6.6 State Variables Return Misleading Values

Informational | Version 1 | Specification Changed

CS-VELO-VERC20-010

In the `HookRegistry` contract, the variable `hookEntrypoints` is used to store the entry points of the hooks. However, by default, any hook will have `Entrypoint(0)` as its value, which corresponds to `BEFORE_APPROVE`. One should first check if the hook is registered using `isHookRegistered()` before using this variable.

Similarly, in `VerifiedERC20`, `hookToIndex` is used to store the index of the hook in the `_hooksByEntrypoint[hookEntrypoint]` array. However, by default, any hook will have `0` as its value, which corresponds to the first index in the array. One should first check if the hook is activated using `isHookActivated()` before using this variable.

Specification changed:

The NatSpec of the external getters for the state variables has been updated to clarify that the values returned by these variables may not be valid and that `isHookRegistered()` or `isHookActivated()` should be used to check the validity of the hook before using these variables.

6.7 Unused Constant MAX_GAS_PER_HOOK

Informational | Version 1 | Code Corrected

CS-VELO-VERC20-012

In the `VerifiedERC20` contract, the constant `MAX_GAS_PER_HOOK` is defined but not used in the `_checkHooks` function. Instead, a hardcoded value of `200_000` is used for the `_gas` parameter in the `excessivelySafeCall`.

Code Corrected:

The hardcoded value `200_000` has been replaced with the `MAX_GAS_PER_HOOK` constant in the `_checkHooks()` function.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Broken Checks-Effects-Interactions Pattern

Informational **Version 1** **Acknowledged**

CS-VELO-VERC20-001

The following functions do not adhere to the Checks-Effects-Interactions pattern:

- `setLockbox()` in `AutoUnwrapHook`: Storage is updated after calling the `owner` function of the `_verifiedERC20`.
- `setAuthorized()` in `SinglePermissionHook`: Storage is updated after calling the `owner` function of the `_verifiedERC20`.
- `registerHook()` in `HookRegistry`: Storage is updated after calling the `supportsEntrypoint` function of the `_hook`.

Although all the interactions here are static calls, which cannot change the state, it is still recommended to follow the Checks-Effects-Interactions pattern to avoid potential issues in the future if the code is modified.

Acknowledged:

The Velodrome team acknowledges the risk and states:

```
External calls are to trusted contracts.
```

7.2 Event Ordering With Reentrancy

Informational **Version 1** **Acknowledged**

CS-VELO-VERC20-003

The `VerifiedERC20` contract is partially reentrant by design to allow hooks like `AutoUnwrapHook` to perform token transfers as part of the transfer hook execution. It should be noted that this reentrancy can lead to unexpected event ordering.

Given a `transferFrom` call with a `BEFORE_TRANSFER` hook that performs a transfer of tokens, the following sequence of events occurs:

1. `Approval` event is emitted for spending the allowance
2. The `BEFORE_TRANSFER` hook is executed, which may call the `transfer` function
3. The `Transfer` event is emitted for the transfer performed by the hook
4. The `Transfer` event is emitted for the original `transferFrom` call

The ordering of these events reflects the ordering of the state changes but can lead to confusion, as one usually expects the `Approval` and `Transfer` events of a `transferFrom` call to be emitted in an

atomic manner. Note that this behavior is enabled by the Inconsistent Allowance Spending Behavior issue, as the hook is called between the allowance being spent and the transfer being executed.

Acknowledged:

The Velodrome team acknowledges the issue.

7.3 Inconsistent Allowance Spending Behavior

Informational | Version 1 | Code Partially Corrected

CS-VELO-VERC20-005

In the `_update` function in the `VerifiedERC20` contract, the allowance spending behavior is inconsistent. When burning tokens for oneself (`to == address(0)` and `msg.sender == from`), the allowance is not spent. However, in the `transferFrom` function, the allowance is always spent.

Additionally, the allowance is spent after the `BEFORE_BURN` hook, whereas in `transfer` and `transferFrom` the allowance is spent before the `BEFORE_TRANSFER` hook.

Code Partially Corrected:

The second inconsistency has been addressed by spending allowance in the `burn` function, before the `BEFORE_BURN` hook is called. The first one has not been addressed.

7.4 Set Ordering Is Not Guaranteed

Informational | Version 1 | Acknowledged

CS-VELO-VERC20-009

Several functions allow querying an `EnumerableSet` or an array given an index. As there are no guarantees on the ordering of values inside the set/array, callers should not rely on the index to return a specific value. The affected functions are:

- `VerifiedERC20.getHookAtIndex()`
- `HookRegistry.getHookAt()`

The following function should however maintain ordering given that the set is append-only:

- `VerifiedERC20Factory.getVerifiedERC20At()`

Acknowledged:

The Velodrome team acknowledges the issue.

7.5 Unchecked Calls

Informational | Version 1 | Acknowledged

CS-VELO-VERC20-011

In both the `AutoUnwrapHook` and `SelfTransferHook` hooks, the `_isClaimIncentive` function is used to determine if the transfer is part of an incentive claim:

```
function _isClaimIncentive(address _from) internal view returns (bool) {
    (bool success, bytes memory data) = _from.excessivelySafeStaticCall({
        _gas: 5_000,
        _maxCopy: 32,
        _calldata: abi.encodeWithSelector(IReward.DURATION.selector)
    });

    if (!success || data.length < 32 || (abi.decode(data, (uint256)) != 7 days)) return false;

    (success, data) = _from.excessivelySafeStaticCall({
        _gas: 5_000,
        _maxCopy: 32,
        _calldata: abi.encodeWithSelector(IReward.voter.selector)
    });
    if (!success || data.length < 32 || voter != abi.decode(data, (address))) return false;

    return true;
}
```

The calls made to `_from` are designed not to make the hook revert if the call fails, but rather to check if the call returns the expected data. If the call fails or returns unexpected data, the function returns `false`, indicating that the transfer is not part of an incentive claim.

A potential attack vector involves a malicious actor providing an amount of gas that would cause the execution to revert with an "out-of-gas" error precisely during one of the two static calls. This could lead the hook to mistakenly interpret the transfer as not being an incentive claim. In the case of the `SelfTransferHook`, this could allow a bypass of the self-transfer verification.

However, we demonstrate that this attack is not feasible in practice. The gas consumption of the called view functions defined in Reward.sol is sufficiently low, making it impossible for such an attack to succeed.

In practice, the following amount of gas is provided to each static call:

```
gasForwarded = min(5000, gasBeforeCall * 63 / 64)
```

Assuming that `_from` defines the `DURATION` and `voter` functions as follows:

```
uint256 public constant DURATION = 7 days;

address public immutable voter;
```

We can safely assume that the gas cost of executing these functions is less than 1,000 gas. This means that one would need `gasBeforeCall` (the gas after charging for the `STATICCALL` opcode and before making the call) to be less than 1,015 gas. Consequently, the gas remaining after the failing call would be less than 16 gas, which is insufficient to complete the hook call successfully.

```
   gasForwarded = min(5000, gasBeforeCall * 63 / 64) < 1000
=> gasForwarded = gasBeforeCall * 63 / 64 < 1000
=> gasBeforeCall < 1000 * 64 / 63
=> gasRemaining = gasBeforeCall / 64 < 1000 / 63 < 16
```

Note that the above claims are made under the assumption that both functions consume less than 1,000 gas. If either of the two called functions exceeds this threshold (e.g., by reading from a storage variable), it may be possible to cause the call to fail while still having sufficient gas to complete the execution.

Acknowledged:

The Velodrome team acknowledges informational findings. As of Version 2, `authorized()` is called instead of `DURATION()`. The former is a state variable, so reading it consumes more gas. Given that the call to `authorized()` will consume in the worst case approximately 2,700 gas (2,100 gas for the cold storage read and 600 gas for Solidity overhead), we show below that the gas remaining in the hook after the call will be less than 72 gas (we use here a safe upper-bound of 4,500 gas for the call to `authorized()`):

```
   gasForwarded = min(5000, gasBeforeCall * 63 / 64) < 4500
=> gasForwarded = gasBeforeCall * 63 / 64 < 4500
=> gasBeforeCall < 4500 * 64 / 63
=> gasRemaining = gasBeforeCall / 64 < 4500 / 63 < 72
```

To reach a return after the failed call to `authorized()` in the Hooks using `_isClaimIncentive`, it was found that at least 100 gas (safe lower bound) is required. As the gas remaining after the failed call will be at most 72 gas, this is insufficient to complete the hook call successfully.

Note that it is critical that the function `authorized()` in every `Reward` contract does not consume more than 5,000 gas. This can occur even with a simple state variable getter if the contract has many entry points, as it would incur high gas overhead from function selector matching.

7.6 `abi.decode` May Fail

Informational **Version 1** **Acknowledged**

CS-VELO-VERC20-013

In both the `AutoUnwrapHook` and `SelfTransferHook` hooks, the `_isClaimIncentive` function is used to determine if the transfer is part of an incentive claim:

```
function _isClaimIncentive(address _from) internal view returns (bool) {
    (bool success, bytes memory data) = _from.excessivelySafeStaticCall({
        _gas: 5_000,
        _maxCopy: 32,
        _calldata: abi.encodeWithSelector(IReward.DURATION.selector)
    });

    if (!success || data.length < 32 || (abi.decode(data, (uint256)) != 7 days)) return false;

    (success, data) = _from.excessivelySafeStaticCall({
        _gas: 5_000,
        _maxCopy: 32,
        _calldata: abi.encodeWithSelector(IReward.voter.selector)
    });
    if (!success || data.length < 32 || voter != abi.decode(data, (address))) return false;

    return true;
}
```

If the `_from` address is not a `Reward` contract, one of the static calls should fail, and the function should return `false`.

It is possible that a non-`Reward` contract is provided as the `_from` address, but implements the `DURATION` and `voter` functions, returning `7 days` and the address of the voter respectively. In this case, the function would return `true`, and this address would be a false positive.

However, if the `_from` address implements the `DURATION` and `voter` functions, but the `voter` function returns a value that cannot be decoded as an address when ABI-encoded, the hook will revert, preventing the `_from` address from sending tokens.

This could happen if the returned value is a `uint256` larger than `2**160 - 1`, which is the maximum value that can be decoded as an address. In this case, the `abi.decode` internal routing fails.

As of Version 2, the call to `DURATION()` was replaced by `authorized()`. As the latter returns an address, the above issue described for the call to `voter()` now also applies to `authorized()` as well.

Acknowledged:

The Velodrome team acknowledges the issue.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Functions With Unbounded Gas Costs

Note Version 1

Several functions return the result of `EnumerableSet.values()`. As this operation copies the entire storage of the set to memory, these functions can be expensive and will become increasingly costly as the set grows. Integrators should not use these functions on-chain, but rather should only use them off-chain, as the gas cost could become too large for the transaction to fit within the block gas limit.

The affected functions are:

- `VerifiedERC20Factory.getAllVerifiedERC20s()`
- `HookRegistry.getAllHooks()`

8.2 Incentive Claiming False Positive

Note Version 1

In both the `AutoUnwrapHook` and `SelfTransferHook` hooks, the `_isClaimIncentive` function is used to determine if the transfer is part of an incentive claim:

```
function _isClaimIncentive(address _from) internal view returns (bool) {
    (bool success, bytes memory data) = _from.excessivelySafeStaticCall({
        _gas: 5_000,
        _maxCopy: 32,
        _calldata: abi.encodeWithSelector(IReward.DURATION.selector)
    });

    if (!success || data.length < 32 || (abi.decode(data, (uint256)) != 7 days)) return false;

    (success, data) = _from.excessivelySafeStaticCall({
        _gas: 5_000,
        _maxCopy: 32,
        _calldata: abi.encodeWithSelector(IReward.voter.selector)
    });
    if (!success || data.length < 32 || voter != abi.decode(data, (address))) return false;

    return true;
}
```

In the unlikely event that a contract, which is not a `Reward` contract, is provided as the `_from` address, and it implements the `DURATION` and `voter` functions (returning `7 days` and the address of the voter, respectively), the function would incorrectly return `true`, leading to a false positive for this address.

8.3 `VerifiedERC20` Will Have Different Addresses on Different Chains

Note Version 1

The `VerifiedERC20Factory` is not designed to deploy two `VerifiedERC20` contracts to the same address on different chains. The `salt` generation includes `block.chainid`, which ensures the deployment address differs across chains.