SMART CONTRACT AUDIT REPORT

for

Versailles Heroes DAO

Prepared By: Xiaomi Huang

PeckShield
July 1, 2022

Document Properties

Client	Versailles Heroes
Title	Smart Contract Audit Report
Target	VRH DAO
Version	1.0
Author	Xuxian Jiang
Auditors	Shulin Bie, Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 1, 2022	Xuxian Jiang	Final Release
1.0-rc1	June 24, 2022	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Versailles Heroes DAO protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Versailles Heroes DAO

Versailles Heroes DAO (VRH DAO) and the Versailles Heroes Token (VRH) are designed to allow users to vote for proposals that will affect the future development of the Versailles Heroes game, as well as other aspects of game governance. Players can participate in voting and mining by joining a game guild in order to receive corresponding rewards based on the voting stake. On top of that, the rewards can be further increased by burning game tokens in the form of GAS. The VRH DAO is inspired from the Curve DAO. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of VRH DAO

Item	Description
Target	VRH DAO
Website	https://versaillesheroes.com/
Type	Smart Contract
Language	Vyper + Solidity
Audit Method	Whitebox
Latest Audit Report	July 1, 2022

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit.

- https://github.com/Versailles-heroes-com/versailles-heroes-DAO.git (37d89e1)
- https://github.com/Versailles-heroes-com/VRH-Aragon-DAO.git (ce10003)

And here are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/Versailles-heroes-com/versailles-heroes-DAO.git (e9b5b1c)
- https://github.com/Versailles-heroes-com/VRH-Aragon-DAO.git (ce10003)

1.2 About PeckShield

PeckShield Inc. [15] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [14]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [13], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `VRH DAO` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	3	■ ■ ■
Low	4	■ ■ ■ ■
Informational	1	■
Total	9	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 3 medium-severity vulnerabilities, 4 low-severity vulnerabilities, and 1 informational recommendations.

Table 2.1: Key VRH DAO Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Handling of Ownership Transfer in VotingEscrow/VestingEscrow	Security Features	Confirmed
PVE-002	High	Improper Funding Source In VotingEscrow::_deposit_for()	Business Logic	Resolved
PVE-003	Low	Improved Binary Search in find_block_epoch()	Coding Practices	Resolved
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-005	Medium	Lack of Protection Against Oversized Gauge/Type Weights	Numeric Errors	Confirmed
PVE-006	Medium	Implicit Threshold on Supported Distinct Guild Type	Business Logic	Confirmed
PVE-007	Informational	Improved AddType() Event Generation	Error Conditions, Return Values, Status Codes	Confirmed
PVE-008	Low	Improved Sanity Checks of Guild/Type Weight Updates	Coding Practices	Confirmed
PVE-009	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Better Handling of Privilege Transfers

- ID: PVE-001
- Severity: Low
- Likelihood: N/A
- Impact: High
- Targets: `Multiple Contracts`
- Category: Security Features [8]
- CWE subcategory: CWE-282 [3]

Description

The `VRH DAO` protocol implements a rather basic access control mechanism that allows a privileged account, i.e., `admin`, to be granted exclusive access to typically sensitive functions (e.g., setting the new `minter` and adding a new `guild/type`). Because of the privileged access and the implications of these sensitive functions, the `admin` account is essential for the protocol-level safety and operation.

Within the contract `GuildController`, a specific function, i.e., `commit_transfer_ownership(addr: address)`, is provided to allow for possible `admin` updates. However, current implementation achieves its goal by providing another companion function `apply_transfer_ownership()`. This is reasonable under the assumption that the `future_admin` parameter is always correctly provided. However, in the unlikely situation, when an incorrect `future_admin` is provided, the contract owner may be forever lost, which might be devastating for protocol-wide operation and maintenance.

As a common best practice, instead of achieving the owner update only from one party, i.e., the current `admin`, it is suggested to get both parties involved. For example, the first step is initiated by the current `admin` and the second step is initiated by the configured `future_admin` who accepts and materializes the update. Both steps should be executed in two separate transactions. By doing so, it can greatly alleviate the concern of accidentally transferring the contract ownership to an uncontrolled address. By doing so, we can guarantee that an owner public key cannot be nominated unless there is an entity that has the corresponding private key. This is explicitly designed to prevent unintentional errors in the owner transfer process.

```
@external
def commit_transfer_ownership(addr: address):
    """
    @notice Transfer ownership of GuildController to `addr`
    @param addr Address to have ownership transferred to
    """
    assert msg.sender == self.admin  # dev: admin only
    self.future_admin = addr
    log CommitOwnership(addr)


@external
def apply_transfer_ownership():
    """
    @notice Apply pending ownership transfer
    """
    assert msg.sender == self.admin  # dev: admin only
    _admin: address = self.future_admin
    assert _admin != ZERO_ADDRESS  # dev: admin not set
    self.admin = _admin
    log ApplyOwnership(_admin)
```

Listing 3.1: Current Admin Transfer in `GuildController`

Recommendation Implement a two-step approach that involves actions from both relevant parties, i.e., `admin` and `future_admin`. In particular, the current `admin` initiates `commit_transfer_ownership()` and the intended `future_admin` executes `accept_transfer_ownership()`. Note that the current ownership transfer only involves actions from one party. The same is also applicable for other privileged accounts.

```
@external
def commit_transfer_ownership(addr: address):
    """
    @notice Transfer ownership of GaugeController to `addr`
    @param addr Address to have ownership transferred to
    """
    assert msg.sender == self.admin  # dev: admin only
    self.future_admin = addr
    log CommitOwnership(addr)


@external
def accept_transfer_ownership():
    """
    @notice Accept pending ownership transfer
    """
    assert msg.sender == self.future_admin  # dev: future_admin only
    assert _admin != ZERO_ADDRESS  # dev: admin not set
    self.admin = msg.sender
    self.future_admin = ZERO_ADDRESS
```

```
    log ApplyOwnership(msg.sender )
```

Listing 3.2: Revised Admin Transfer in `GuildController`

Status This issue has been confirmed. The team decides to address it in the future iteration of development.

3.2 Improper Funding Source In VotingEscrow::_deposit_for()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `VotingEscrow`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

The `VRH DAO` has a key `VotingEscrow` contract that provides the functionality of computing the time-dependent vote weights. By design, the vote weight decays linearly over time and the lock time cannot be more than `MAXTIME` (4 years). While reviewing the current locking logic, we notice the key helper routine `_deposit_for()` needs to be revised.

To elaborate, we show below the implementation of this `_deposit_for()` helper routine. In fact, it is an internal function to perform deposit and lock tokens for a user. This routine has a number of arguments and the first one `_addr` is the address to receive the balance. The second `_from` address is the account that actually provides the assets, `assert ERC20(self.token).transferFrom(_from, self, _value)` (line 366). However, it comes to our attention that is caller `deposit_for()` uses the same given `addr` as the first argument and the second argument! As a result, the current implementation may be abused to lock tokens from users who have approved the locking contract before without their notice. To fix, there is a need to use the `msg.sender` as the second argument to provide the assets for locking!

```
@external
@nonreentrant('lock')
def deposit_for(_addr: address, _value: uint256):
    """
    @notice Deposit `_value` tokens for `_addr` and add to the lock
    @dev Anyone (even a smart contract) can deposit for someone else, but
         cannot extend their locktime and deposit for a brand new user
    @param _addr User's wallet address
    @param _value Amount to add to user's lock
    """
    _locked: LockedBalance = self.locked[_addr]
```

```
    assert _value > 0  # dev: need non-zero value
    assert _locked.amount > 0, "No existing lock found"
    assert _locked.end > block.timestamp, "Cannot add to expired lock. Withdraw"

    self._deposit_for(_addr, _addr, _value, 0, self.locked[_addr], DEPOSIT_FOR_TYPE)
```

Listing 3.3: `VotingEscrow::deposit_for()`

```
@internal
def _deposit_for(_addr: address, _from: address, _value: uint256, unlock_time: uint256,
    locked_balance: LockedBalance, type: int128):
    """
    @notice Deposit and lock tokens for a user
    @param _addr User's wallet address
    @param _value Amount to deposit
    @param unlock_time New time when to unlock the tokens, or 0 if unchanged
    @param locked_balance Previous locked amount / timestamp
    """
    _locked: LockedBalance = locked_balance
    supply_before: uint256 = self.supply

    self.supply = supply_before + _value
    old_locked: LockedBalance = _locked
    # Adding to existing lock, or if a lock is expired - creating a new one
    _locked.amount += convert(_value, int128)
    if unlock_time != 0:
        _locked.end = unlock_time
    self.locked[_addr] = _locked

    # Possibilities:
    # Both old_locked.end could be current or expired (>/< block.timestamp)
    # value == 0 (extend lock) or value > 0 (add to lock or extend lock)
    # _locked.end > block.timestamp (always)
    self._checkpoint(_addr, old_locked, _locked)

    if _value != 0:
        assert ERC20(self.token).transferFrom(_from, self, _value)

    log Deposit(_from, _addr, _value, _locked.end, type, block.timestamp)
    log Supply(supply_before, supply_before + _value)
```

Listing 3.4: `VotingEscrow::_deposit_for()`

Recommendation Revise the above calling routine to use the right funding source to transfer the assets for locking.

Status The issue has been fixed in the following commits: 8b0940b and de786bfa.

3.3 Improved Binary Search in find_block_epoch()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: VotingEscrow
- Category: Coding Practices [9]
- CWE subcategory: CWE-1099 [1]

Description

Following CurveDAO, the VRH DAO takes the approach of measuring the vote in a amount-time-weighted manner where the time counted is the time left to unlock, i.e., how long the tokens cannot be moved in the future. (Note the maximum selectable locktime is 4 years.)

Because of the above vote measurement, it becomes necessary to measure current voting power at a particular block and convert from a block number to the corresponding timestamp. To this end, the current codebase takes a binary search algorithm to estimate the related timestamp for a given block number. To elaborate, we show below the related code snippet of the binary search routine, i.e., find_block_epoch().

The routine implements a rather standard binary search algorithm and we find the current implementation can be (slightly) improved by iterating one round less. Specifically, if the comparison with current _block (line 529) shows it is identical with the _mid block number, we can simply return _mid, hence allowing for early termination of the iteration.

```
515 def find_block_epoch(_block: uint256, max_epoch: uint256) -> uint256:
516     """
517     @notice Binary search to estimate timestamp for block number
518     @param _block Block to find
519     @param max_epoch Don't go beyond this epoch
520     @return Approximate timestamp for block
521     """
522     # Binary search
523     _min: uint256 = 0
524     _max: uint256 = max_epoch
525     for i in range(128):  # Will be always enough for 128-bit numbers
526         if _min >= _max:
527             break
528         _mid: uint256 = (_min + _max + 1) / 2
529         if self.point_history[_mid].blk <= _block:
530             _min = _mid
531         else:
532             _max = _mid - 1
533     return _min
```

Listing 3.5: VotingEscrow::find_block_epoch()

In addition, the `balanceOfAt()` routine has an internal `for` loop that can be similarly optimized. Moreover, we notice that the internal `for` loop has an upper bound of at most 128 times. This number can be reduced to 30 (`VotingEscrow.vy` at line 574). The reason is that `user_point_history` holds at most $1,000,000,000$ points and $1,000,000,000 < 2^{30}$. In the same vein, the `for` loop in `find_block_epoch()` can set the upper limit of 100, instead of current 128.

Recommendation Optimize the `find_block_epoch()` implementation as shown below. Note that a similar optimization is also applicable to the `balanceOfAt()` implementation.

```
def find_block_epoch(_block: uint256, max_epoch: uint256) -> uint256:
    """
    @notice Binary search to estimate timestamp for block number
    @param _block Block to find
    @param max_epoch Don't go beyond this epoch
    @return Approximate timestamp for block
    """
    # Binary search
    _min: uint256 = 0
    _max: uint256 = max_epoch
    _tmp_block: uint256 = 0
    for i in range(100):  # Will be always enough for 128-bit numbers
        if _min >= _max:
            break

        _mid: uint256 = (_min + _max + 1) / 2
        _tmp_block = self.point_history[_mid].blk

        if _tmp_block == _block:
            return _mid
        elif _tmp_block < _block:
            _min = _mid
        else:
            _max = _mid - 1
    return _min
```

Listing 3.6: Revised `VotingEscrow::find_block_epoch()`

Status This issue has been confirmed. The team decides to address it in the future iteration of development.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.7: `ZRX::transfer()/transferFrom()`

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `add_tokens()` routine in the `VestingEscrow` contract. If the `USDT` token is supported as `self.token`, the unsafe version of `ERC20(self.token).transferFrom(msg.sender, self, _amount)` (line 92) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
84 @external
85 def add_tokens(_amount: uint256):
86     """
87     @notice Transfer vestable tokens into the contract
88     @dev Handled separate from `fund` to reduce transaction count when using funding
           admins
89     @param _amount Number of tokens to transfer
90     """
91     assert msg.sender == self.admin  # dev: admin only
92     assert ERC20(self.token).transferFrom(msg.sender, self, _amount)  # dev: transfer
           failed
93     self.unallocated_supply += _amount
```

Listing 3.8: `VestingEscrow::add_tokens()`

The same issue is also applicable to a number of other routines, including `deposit()` and `withdraw()` in `VotingEscrow` and `GasEscrow.vy` contracts.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status This issue has been resolved as the team confirms the use of ERC20-compliant tokens only.

3.5 Lack of Protection Against Oversized Gauge/Type Weights

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: `GuildController`
- Category: Numeric Errors [12]
- CWE subcategory: CWE-190 [2]

Description

The `VRH DAO` is based on the `CurveDAO` where the `GuildController` contract is the central of the entire governance subsystem. In particular, this `GuildController` contract is responsible for adding new `guilds` and their `types`, changing their `weights`, as well as casting votes on different `guilds`.

Our analysis leads to the discovery of a potential pitfall when a new oversized `guild` (or `type`) weight is updated on current pools. In particular, as the `guild_relative_weight()` routine involves the multiplication of three `uint256` integer, it is possible for their multiplication to have an undesirable overflow (`MULTIPLIER * _type_weight * _guild_weight` in `GuildController` at line 456), especially when `_type_weight` or `_guild_weight` is largely controlled by an external entity. Fortunately, an authentication check is in place that effectively restricts the caller to be `admin` and thus greatly alleviates such concern.

```
509 def _change_type_weight(type_id: int128, weight: uint256):
510     """
511     @notice Change type weight
512     @param type_id Type id
513     @param weight New type weight
514     """
515     old_weight: uint256 = self._get_type_weight(type_id)
516     old_sum: uint256 = self._get_sum(type_id)
517     _total_weight: uint256 = self._get_total()
518     next_time: uint256 = (block.timestamp + WEEK) / WEEK * WEEK
519 
520     _total_weight = _total_weight + old_sum * weight - old_sum * old_weight
521     self.points_total[next_time] = _total_weight
522     self.points_type_weight[type_id][next_time] = weight
523     self.time_total = next_time
524     self.time_type_weight[type_id] = next_time
525 
526     log NewTypeWeight(type_id, next_time, weight, _total_weight)
```

Listing 3.9: `GuildController::_change_type_weight()`

However, any mis-configuration on the given weight may block the reward-claiming attempts of users who have staked on the affected `guilds` or `types`. If we use the `change_type_weight()` as an example, this issue is made possible if the `weight` amount is given as the argument to `_change_type_weight()` such that the calculation of `MULTIPLIER * _type_weight * _guild_weight` always overflows, hence reverting every `guild_relative_weight()` calculation of affected `guilds` in reward-claiming attempts. Note either only the specific `guild` with the misconfigured oversized weight or all `guilds` sharing the same oversized `guild type` will be affected.

```
440 def _guild_relative_weight(addr: address, time: uint256) -> uint256:
441     """
442     @notice Get Guild relative weight (not more than 1.0) normalized to 1e18
443             (e.g. 1.0 == 1e18). Inflation which will be received by it is
444             inflation_rate * relative_weight / 1e18
```

```
    @param addr Guild address
    @param time Relative weight at the specified timestamp in the past or present
    @return Value of relative weight normalized to 1e18
    """
    t: uint256 = time / WEEK * WEEK
    _total_weight: uint256 = self.points_total[t]

    if _total_weight > 0:
        guild_type: int128 = self.guild_types_[addr] - 1
        _type_weight: uint256 = self.points_type_weight[guild_type][t]
        _guild_weight: uint256 = self.points_weight[addr][t].bias
        return MULTIPLIER * _type_weight * _guild_weight / _total_weight

    else:
        return 0
```

Listing 3.10: `GuildController::_guild_relative_weight()`

To mitigate, it is best to apply a threshold check on the allowed weight update on a current `guild` or a supported `guild type`. Specifically, we can define `TOTAL_WEIGHT_THRESHOLD` that aims to restrict the total weight calculated from all current `guilds`. Therefore, for any change on a `guild` weight or a `type` weight, we can guarantee that the total weight is within an appropriate range. A candidate choice should be no larger than `TOTAL_WEIGHT_THRESHOLD: constant(uint256)= convert(-1, uint256)/ MULTIPLIER`.

Recommendation Add sanity checks to prevent the changed weight of a `guild` or an existing `guild type` from leading to an overflow calculation.

Status This issue has been confirmed. The team indicates that this specific issue is best mitigated through an off-chain vetting process to avoid configuring with an over-weighted number.

3.6 Implicit Threshold On Supported Distinct Guild Types

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `GuildController`
- Category: Business Logic [10]
- CWE subcategory: CWE-837 [6]

Description

In `VRH DAO`, there is an implicit restriction on the number of `guild types` that can be supported. However, this restriction is not enforced when a new `guild type` is being added. As a result, if a new `guild type` is assigned with an type id that exceeds the limit, the new `guild type` as well as all `guilds` of this `guild type` will not be able to participate in the governance token distribution.

To elaborate, we show below the `_get_total()` routine that is responsible for calculating and maintaining the total weighted-sum of all current `guilds`. For each `guild`, its weight is counted by multiplying the `guild` weight with the corresponding `guild type` weight.

```
305 def _get_total() -> uint256:
306     """
307     @notice Fill historic total weights week-over-week for missed checkins
308             and return the total for the future week
309     @return Total weight
310     """
311     t: uint256 = self.time_total
312     _n_guild_types: int128 = self.n_guild_types
313     if t > block.timestamp:
314         # If we have already checkpointed - still need to change the value
315         t -= WEEK
316     pt: uint256 = self.points_total[t]

318     for guild_type in range(100):
319         if guild_type == _n_guild_types:
320             break
321         self._get_sum(guild_type)
322         self._get_type_weight(guild_type)

324     for i in range(500):
325         if t > block.timestamp:
326             break
327         t += WEEK
328         pt = 0
329         # Scales as n_types * n_unchecked_weeks (hopefully 1 at most)
330         for guild_type in range(100):
331             if guild_type == _n_guild_types:
332                 break
333             type_sum: uint256 = self.points_sum[guild_type][t].bias
334             type_weight: uint256 = self.points_type_weight[guild_type][t]
335             pt += type_sum * type_weight
336         self.points_total[t] = pt

338         if t > block.timestamp:
339             self.time_total = t
340     return pt
```

Listing 3.11: `GuildController::_get_total()`

Apparently, as shown in the line 318, only the first 100 `guild types` are taken into consideration, excluding all other `guild types` and their `guilds` from participating in the distribution of governance tokens.

```
def add_type(_name: String[64], _symbol: String[32], gas_addr: address, weight: uint256
    = 0):
    """
    @notice Add guild type with name `_name` and weight `weight`
    @param _name Name of guild type
```

```
    @param gas_addr Address of the gas token
    @param weight Weight of guild type
    """
    assert msg.sender == self.admin
    assert self.gas_addr_escrow[gas_addr] == ZERO_ADDRESS, "Already has gas escrow" # one gas token can only have one gas escrow

    escrow_addr: address = create_forwarder_to(self.gas_escrow)
    _isSuccess: bool = GasEscrow(escrow_addr).initialize(self.admin, gas_addr, _name, _symbol)

    if _isSuccess:
        type_id: int128 = self.n_guild_types
        self.guild_type_names[type_id] = _name
        self.n_guild_types = type_id + 1
        if weight != 0:
            self._change_type_weight(type_id, weight)
            self.gas_type_escrow[type_id] = escrow_addr
            self.gas_addr_escrow[gas_addr] = escrow_addr
            log AddType(_name, type_id, gas_addr, weight, escrow_addr)
```

Listing 3.12: `GuildController::add_type()`

Meanwhile, the `add_type()` routine that handles the addition of new types is not enforcing the above (implicit) limit. With that, it is strongly suggested to define the `MAX_GUILD_TYPES` and make the limit explicit. This explicit limit is necessary as we observe blurred or confused declaration of the number of `guild types` reflected in other data structures. For example, both `time_sum` and `time_type_weight` denote the mapping from a specific `guild type` to the last scheduled time of all `guild` of the same type and the type weight respectively. The current declaration (line 150 and 156) misleadingly indicate the protocol support $1,000,000,000$ types!

By having the explicit limit, we can re-define both `time_sum` and `time_type_weight` in an unambiguous manner that greatly reduces the storage reservation from $1,000,000,000$ to 100, i.e., `time_sum: public(uint256[100])` and `time_type_weight: public(uint256[100])`.

```
150 time_sum: public(uint256[1000000000])  # type_id -> last scheduled time (next week)

152 points_total: public(HashMap[uint256, uint256])  # time -> total weight
153 time_total: public(uint256)  # last scheduled time

155 points_type_weight: public(HashMap[int128, HashMap[uint256, uint256]])  # type_id -> time -> type weight
156 time_type_weight: public(uint256[1000000000])  # type_id -> last scheduled time (next week)
```

Listing 3.13: `GuildController.vy`

Recommendation Explicitly limit the number of `guild types` that can be supported in the protocol and enforce the limit when a new type is being added.

```
MAX_GAUGE_TYPES: constant(uint256) = 100
def add_type(_name: String[64], _symbol: String[32], gas_addr: address, weight: uint256
    = 0):
    """
    @notice Add guild type with name `_name` and weight `weight`
    @param _name Name of guild type
    @param gas_addr Address of the gas token
    @param weight Weight of guild type
    """
    assert msg.sender == self.admin
    assert self.gas_addr_escrow[gas_addr] == ZERO_ADDRESS, "Already has gas escrow" #
        one gas token can only have one gas escrow

    escrow_addr: address = create_forwarder_to(self.gas_escrow)
    _isSuccess: bool = GasEscrow(escrow_addr).initialize(self.admin, gas_addr, _name,
        _symbol)

    if _isSuccess:
        type_id: int128 = self.n_guild_types
        assert type_id < MAX_GAUGE_TYPES
        self.guild_type_names[type_id] = _name
        self.n_guild_types = type_id + 1
        if weight != 0:
            self._change_type_weight(type_id, weight)
            self.gas_type_escrow[type_id] = escrow_addr
            self.gas_addr_escrow[gas_addr] = escrow_addr
            log AddType(_name, type_id, gas_addr, weight, escrow_addr)
```

Listing 3.14: Revised GuildController::add_type()

Status This issue has been confirmed. The team decides to address it in the future iteration of development.

3.7 Improved AddType() Event Generation

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `GuildController`
- Category: Status Codes [11]
- CWE subcategory: CWE-682 [5]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools.

`Events` can be emitted in a number of scenarios, e.g., when updating system-wide parameters or adding new components. For example, VRH DAO defines new `guilds` and `types`. However, the current implementation can be improved by correctly emitting related events when they are being changed.

In the following, we use the `AddType` event as an example. This event is defined in the `GuildController` contract and represents the state of adding a new `guild type`.

```
530 def add_type(_name: String[64], _symbol: String[32], gas_addr: address, weight: uint256 = 0):
531     """
532     @notice Add guild type with name `_name` and weight `weight`
533     @param _name Name of guild type
534     @param gas_addr Address of the gas token
535     @param weight Weight of guild type
536     """
537     assert msg.sender == self.admin
538     assert self.gas_addr_escrow[gas_addr] == ZERO_ADDRESS, "Already has gas escrow" # one gas token can only have one gas escrow

540     escrow_addr: address = create_forwarder_to(self.gas_escrow)
541     _isSuccess: bool = GasEscrow(escrow_addr).initialize(self.admin, gas_addr, _name, _symbol)

543     if _isSuccess:
544         type_id: int128 = self.n_guild_types
545         self.guild_type_names[type_id] = _name
546         self.n_guild_types = type_id + 1
547         if weight != 0:
548             self._change_type_weight(type_id, weight)
549             self.gas_type_escrow[type_id] = escrow_addr
550             self.gas_addr_escrow[gas_addr] = escrow_addr
551             log AddType(_name, type_id, gas_addr, weight, escrow_addr)
```

Listing 3.15: `GuildController::add_type()`

However, we notice that this event is not emitted if `weight==0` (line 547). It may cause issues for off-chain components to monitor the set of `guild types` being supported in the system. Moreover, the event can be improved by encoding the weight information as well, which is currently missing.

In the same vein, we suggest to refine the `NewGuild`, `NewGuildWeight`, and `VoteForGauge` events by indexing the related `guild_address`. In addition, we can enhance the `Minted` event by encoding `to_mint` as well. By doing so, we can better facilitate off-chain analytics and reporting tools.

Recommendation Emit necessary events to timely reflect system dynamics.

```
def add_type(_name: String[64], _symbol: String[32], gas_addr: address, weight: uint256 = 0):
    """
    @notice Add guild type with name `_name` and weight `weight`
    @param _name Name of guild type
```

```
    @param gas_addr Address of the gas token
    @param weight Weight of guild type
    """
    assert msg.sender == self.admin
    assert self.gas_addr_escrow[gas_addr] == ZERO_ADDRESS, "Already has gas escrow" #
        one gas token can only have one gas escrow

    escrow_addr: address = create_forwarder_to(self.gas_escrow)
    _isSuccess: bool = GasEscrow(escrow_addr).initialize(self.admin, gas_addr, _name,
        _symbol)

    if _isSuccess:
        type_id: int128 = self.n_guild_types
        self.guild_type_names[type_id] = _name
        self.n_guild_types = type_id + 1
        if weight != 0:
            self._change_type_weight(type_id, weight)
            self.gas_type_escrow[type_id] = escrow_addr
            self.gas_addr_escrow[gas_addr] = escrow_addr
        log AddType(_name, type_id, gas_addr, weight, escrow_addr)
```

Listing 3.16: `GuildController::add_type()`

Status This issue has been confirmed. The teams plans to emit the above events in the next iteration of development.

3.8 Improved Sanity Checks of Guild/Type Weight Updates

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GuildController`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1099 [1]

Description

The distribution of `VRH` governance tokens requires proper setup of participating `guilds`, `guild types` as well as their respective weights. The share of each `guild` is proportional to each `guild` weight multiplied with the `guild type` weight and then divided by the total weighted sum.

In this section, we examine the logic related to the updates to these weights. Our result shows the update logic can be improved by applying more rigorous sanity checks. Based on current implementation, certain corner cases may be exploited to lead to undesirable consequences, including reporting a lower `guild_relative_weight()` and a higher `get_total_weight()`. These two routines are essential for the calculation of `guild` proportions for reward distribution.

To elaborate, we show its code snippet below of two essential functions, i.e., `_change_type_weight()` and `_change_guild_weight()`. These two functions handles the weight updates to `guilds` and `guild types`, respectively. Both routines do not validate the given `guild` or `guild type` as part of input arguments.

```
def _change_type_weight(type_id: int128, weight: uint256):
    """
    @notice Change type weight
    @param type_id Type id
    @param weight New type weight
    """
    old_weight: uint256 = self._get_type_weight(type_id)
    old_sum: uint256 = self._get_sum(type_id)
    _total_weight: uint256 = self._get_total()
    next_time: uint256 = (block.timestamp + WEEK) / WEEK * WEEK

    _total_weight = _total_weight + old_sum * weight - old_sum * old_weight
    self.points_total[next_time] = _total_weight
    self.points_type_weight[type_id][next_time] = weight
    self.time_total = next_time
    self.time_type_weight[type_id] = next_time

    log NewTypeWeight(type_id, next_time, weight, _total_weight)
```

Listing 3.17: `GuildController::_change_type_weight()`

```
def _change_guild_weight(addr: address, weight: uint256):
    # Change guild weight
    # Only needed when testing in reality
    guild_type: int128 = self.guild_types_[addr] - 1
    old_guild_weight: uint256 = self._get_weight(addr)
    type_weight: uint256 = self._get_type_weight(guild_type)
    old_sum: uint256 = self._get_sum(guild_type)
    _total_weight: uint256 = self._get_total()
    next_time: uint256 = (block.timestamp + WEEK) / WEEK * WEEK

    self.points_weight[addr][next_time].bias = weight
    self.time_weight[addr] = next_time

    new_sum: uint256 = old_sum + weight - old_guild_weight
    self.points_sum[guild_type][next_time].bias = new_sum
    self.time_sum[guild_type] = next_time

    _total_weight = _total_weight + new_sum * type_weight - old_sum * type_weight
    self.points_total[next_time] = _total_weight
    self.time_total = next_time

    log NewGuildWeight(addr, block.timestamp, weight, _total_weight)
```

Listing 3.18: `GuildController::_change_guild_weight()`

Without validating the type_id, it is possible to assign the weight of the minusOne (or -1) guild type. Later on, if an unregistered guild is updated, the _change_guild_weight() may eventually contaminate the calculation of points_total[next_time] and time_total (lines $584 - 585$). We have not exhaustively searched through all possible exploitations. However, the lack of thorough validation itself is worrisome and we strongly apply necessary sanity checks to block updating invalid guilds and guild types.

Last but not least, it is also suggested to enhance the add_guild() logic by ensuring the total number of guilds is no more than $1,000,000,000$ – the hard-coded limit in the system. As mentioned earlier, the add_type() logic needs to be revised by ensuring the total number of guild types is no more than 100 – an implicit limit in the system.

Recommendation Validate the given guild address or the guild type before updating their weights in the system.

Status This issue has been confirmed. The teams plans to add necessary validation logics in the next iteration of development.

3.9 Trust Issue of Admin Keys

- ID: PVE-009
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [8]
- CWE subcategory: CWE-287 [4]

Description

In the VRH DAO protocol, there is a privileged account admin that play a critical role in governing and regulating the system-wide operations (e.g., parameter setting and guild adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
@external
def add_type(_name: String[64], _symbol: String[32], gas_addr: address, weight: uint256
    = 0):
    """
    @notice Add guild type with name '_name' and weight 'weight'
    @param _name Name of guild type
    @param gas_addr Address of the gas token
    @param weight Weight of guild type
    """
```

```
    assert msg.sender == self.admin
    assert self.gas_addr_escrow[gas_addr] == ZERO_ADDRESS, "Already has gas escrow" #
        one gas token can only have one gas escrow

    escrow_addr: address = create_forwarder_to(self.gas_escrow)
    _isSuccess: bool = GasEscrow(escrow_addr).initialize(self.admin, gas_addr, _name,
        _symbol)

    if _isSuccess:
        type_id: int128 = self.n_guild_types
        self.guild_type_names[type_id] = _name
        self.n_guild_types = type_id + 1
        if weight != 0:
            self._change_type_weight(type_id, weight)
            self.gas_type_escrow[type_id] = escrow_addr
            self.gas_addr_escrow[gas_addr] = escrow_addr
            log AddType(_name, type_id, gas_addr, weight, escrow_addr)

@external
def change_guild_weight(addr: address, weight: uint256):
    """
    @notice Change weight of guild `addr` to `weight`
    @param addr `GuildController` contract address
    @param weight New Guild weight
    """
    assert msg.sender == self.admin
    self._change_guild_weight(addr, weight)
```

Listing 3.19: Example `setters` in the `GuildController`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the privileges explicit to the protocol users.

Status This issue has been mitigated. The team decides to use multi-sig contract for the privileged `admin` account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `VRH DAO` protocol, which is designed to allow users to vote for proposals that will affect the future development of the `Versailles Heroes` game, as well as other aspects of game governance. Players can participate in voting and mining by joining a game guild in order to receive corresponding rewards based on the voting stake. On top of that, the rewards can be further increased by burning game tokens in the form of `GAS`. The `VRH DAO` is inspired from the `Curve DAO`. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-282: Improper Ownership Management. https://cwe.mitre.org/data/definitions/282.html.

[4] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[5] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[6] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[7] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[8] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[9] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[10] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[11] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[12] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[13] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[14] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[15] PeckShield. PeckShield Inc. https://www.peckshield.com.