STAUBLI
SOFTWARE
SOLUTIONS

Smart Contract Code Review and Audit Report

December 20th, 2022

—

Created for: Gravel 777

Roger Staubli
Staubli-Software-Solutions
St.Niklausengasse 19
6010 Kriens, Switzerland

STAUBLI SOFTWARE SOLUTIONS

Document

Name	Smart Contract Code Review and Audit Report
Carried out by	Roger Staubli | Founder Staubli-Software-Solutions
Language	Solidity
Methods	Best Practice Review, Manual Code inspection, Automated Code inspection
Repository	https://github.com/Gravel777/smart-contracts
Commit	Initial: 427bae57dd9b2cd79f8aabe0b33b8d8c3f390a29
Technical Documentation	Yes: Provided Whitepaper Draft
Unit Tests	Yes
Timeline	29.09.2022 – 20.12.2022
Changelog	18.10.2022 – Initial Audit 26.10.2022 - Updated Audit 20.12.2022 - Final Audit

STAUBLI
SOFTWARE
SOLUTIONS

Executive Summary

After the changes from the updated audit, the final audit resulted in 0 critical severity issues, 0 medium severity issues, and 0 low severity issues. In addition, 1 informational suggestion was acknowledged.

Gravel 777 offers users a decentralized play-and-win platform where everyone can participate by making regular ERC-20 token transfers. It is achieved with a tax on transfers which will automatically buy a ticket for the lottery. In addition, participants can win NFT, and NFT holders can gain rewards in each lottery round.

Overall, the contracts are well programmed, with a clear separation of concerns and respect for Smart Contract best practices.

Scope

The audit contained all Smart Contracts, tests, and deployment scripts from the specified commit described on the second page. Well-known dependencies like OpenZeppelin implementations were out of scope. The marked files were updated since the initial audit

The following Smart Contract files were reviewed:

d2da3ea393166d7f3b981405ea27a1dccd5fd3bb	contracts/Airdrop.sol
964108eca7fe9b6d04746b2066136087766589fe	contracts/ChainlinkVRF.sol
654214c6d533cbc39f2c775b5565c808aa4aefc4	contracts/GERC20.sol
48e8f18c44a93ddefb9a445256452c00a89c54f8	contracts/GERC721.sol
f6b733340124ad55b431d59f4a464dc4e72f735a	contracts/Governor.sol
867f0f419bb97a9e1417f50e13e6c49f7b84aa07	contracts/LiquidityFarm.sol
12da46923e722dd842b9d1fc2b8cedceace415fe	contracts/PublicSale.sol
6959239dcb2b4cd6587ac8815d17dd283ee1cbe5	contracts/RoundPool.sol
6b631aa8e1d99e53b4aee9b5441bf5a20b70eda2	contracts/StakingPool.sol
dfa991c5fec578ac24ec62cb5be87c5ec81f5eb6	contracts/Treasury.sol
c5ec29ff2e205cc5cc86ad073b2d554783116f70	contracts/Vesting.sol

The following tests and deployment scripts were reviewed:

ba59e44860f6d22743cd32da91d4d2a7e047d618	scripts/dev/constants.ts
75ee6f6dbc6defabed3abcb3d2663169a6de52b5	scripts/dev/deploy.ts
05890be1b0ca510f4bfb20f081bece84fd3b8aad	scripts/prod/constants.ts
9929f4de205183f61782c730fa8db889b2935073	scripts/prod/deploy.ts
e48cd7e5e6dcdb88447a2cc6d52d09108a8bf993	test/unit/Airdrop.ts
ea6da610288a25f2bee4b8586b1cc561e4ed115e	test/unit/GERC20.ts

```
d3725b396850480982705343f0f08dc8dcc076b6  test/unit/GERC721.ts
1f98c97247fc6638f015db2c1c717beab1720754  test/unit/Governor.ts
a8fe124effd7a5a5ba4787d5e6b76e74650cdeb0  test/unit/LiquidityFarm.ts
95a1d1f210c69d687d97a462d43d965290d5de67  test/unit/PublicSale.ts
8fb0852c97b278e20204a137bf0f94c81f080892  test/unit/RoundPool.ts
76bfeb3f62fed2ecbeb79a95060182a17ef3aef1  test/unit/StakingPool.ts
130a6773fffa8883c647656e0d8c926055a55fb1  test/unit/Treasury.ts
40a8b72cb172bf24954fba7472539f3909fed079  test/unit/Vesting.ts
```

System Description

The Gravel 777 contracts is a set of contracts which provide a play and win environment where users can participate with simple ERC20 transfers. The main idea is that the transfers are taxed and sent into the round pool which elects a jackpot, an NFT, and an NFT holder winner each predefined round. Chainlink VRF oracle is used to retrieve a provably fair and verifiable random number. Using this random seed, each winner is found with a binary search in the participants' list. Next to that, additional contracts like vesting, staking pool, liquidity farm, public sale, airdrop, governance, and treasury are used in the ecosystem. The following section briefly describes their functionalities. It is important to note that all contracts are upgradeable using OpenZeppelin upgrades.

Airdrop.sol

The airdrop contract provides the functionality of granting linearly vested airdrops to participants. The airdrop token can simply be transferred to the airdrop contract. Then, an account with a granter role can enter participants, their granted amount, and their linearly vested period in a batch request. The amount is reserved for these specific participants and can then be vested linearly. The contract also offers address migrations (after approving the migration by the migrating participant) and canceling of specific airdrop participants.

ChainlinkVRF.sol

This contract is abstract and provides an interface to retrieve the random seed from chainlink. It inherits from the VRFConsumerBaseV2Upgradeable and provides a callback function “loadRoundSeed” to the consumer. This function is called upon fulfillRandomWords when a random seed was generated by chainlink, and the round and seed are provided.

GERC20.sol

This is the contract for the ERC20 token of the ecosystem. It is based on an OpenZeppelin implementation and extends it with a transfer tax. The tax goes into the Round Pool to participate in the lottery. If the sender is a contract, it is not taxed as long as not specified in

“isTaxedContract”. Transfers to the Round Pool are not taxed either. If the sender is an EOA it is taxed as long as the address is not in the “isTaxFreeUser” mapping. In addition, recipients can also be exempted from the tax by adding them to the “isExemptRecipient” mapping. The ERC20 token also inherits from the ERC20Votes and Permit extensions such that it can be used as a governance token.

GERC721.sol

This is the contract for the NFTs in the ecosystem. It is based on the OpenZeppelin implementation and adds ERC721Enumberable in order to get the total supply of the NFTs. An address needs to be granted a minter role such that this address can mint new NFTs. The round pool will be granted as a minter to mint an NFT for the round winner. The contract stores an additional mapping “tokenDescriptor” which is a hash of the random seed of the winning NFT.

Governor.sol

The governor contract enables the community to participate in the decision-making process. It is inherited from the OpenZeppelin governor and uses the Counting Simple (3 voting options: Against, For, and Abstain) and the Votes (extracting voting weights from an ERC20Votes token: here GERC20.sol) extension. The governor holds a list of lockedReserves, which are deducted from the total supply resulting in the circulating supply. The circulating supply is then used to calculate the quorum. An Updater Role address can update voting parameters like the voting period blocks, locked supplies, quorum percentage, etc. and a Proposer Role can create new proposals.

LiquidityFarm.sol

The liquidity farm allows users to stake the ecosystem's LP token in order to receive rewards in the form of GERC20 tokens. In addition, it allows the user to stake the native currency, which is then swapped to the ecosystem's LP token. This amount of LP is then deposited into the farm. The liquidity farm uses EIP-1973 scalable rewards to calculate each depositor's rewards. The depositors can harvest their rewards in a pull manner. The Liquidity farm supports a static block reward and a dynamic reward. The dynamic reward kicks in when the static reward is set to 0. The liquidity farm can be rewarded by anyone (simple ERC20 transfer) but is intended to be rewarded from the round pool. Moreover, it supports the migration of addresses if they are approved by the migrating participant.

PublicSale.sol

The public sale allows users to invest a purchase token (e.g. USDC or any other predefined ERC20 token) or the native currency in the open state and receive a destination token

(GERC20) in the finished state. The contract supports four different states: pending, open, finished, and canceled. In the pending state, users can not invest or receive the destination token. Only the Updater Role can set some parameters like the swapPath from the purchase token to the destination token and can open the sale. In the open state, users can deposit their purchase tokens. The Updater Role can finish the sale or cancel the sale from there. In case of finishing the sale, users will be able to linearly claim their tokens for a predefined period (releasePeriodBlocks). It is important to note that the claimable tokens will be swapped to the destination token at the time when the user claims them. In addition, the contract accepts additional rewards, which will be distributed according to the bought amount of each address using EIP-1973 (it is intended that the round pool additionally rewards the public sale). When the Updater cancels the sale, the depositors are able to refund their purchase tokens. In addition, the liquidity farm supports the migration of addresses.

RoundPool.sol

The round pool is one of the core contracts of the ecosystem. Here, the lottery takes place for each round, the winners are elected, and the winning rewards are distributed. Each round can have one of six different states: pending, open, no_seed, pending_seed, pending_resolve, and resolved.

In the open state, participants can add an entry to participate in the lottery. This is publicly available, but it is also called for each transfer (if the transfer is taxed) from the GERC20 token. The entry is stored chronologically in an array with the player's address, the previous total bets, and the player's bet. If a current round has ended (roundTimeSeconds is passed), the round state is changed to no_seed, and a new round is opened. The player's bet is then added to the new round.

When a round is in no_seed state, an external function needs to be called by an arbitrary address, to request the random seed from the Chainlink VRF. If done so, the state is changed to pending_seed.

When the random seed arrives, it is stored in the contract, and the stage is changed to pending_resolve.

Again, a new function needs to be called by an arbitrary address to find the winners and distribute the winning amounts. In this function, a round winner, a new NFT winner, and an NFT holder winner are elected. The round winner and the new NFT winner are elected separately with a binary search on the entry array of the player’s bets. For that, the random seed is hashed and then calculated modulo the sum of each player's entry. It results in a random number between 0 and the sum of each player’s entry. With binary search, the player is found, which holds the slot in the chronologically ordered entry array.

When the winners are found, the NFT is minted, the transfers are executed (NFT holder, round winner, staking pool share, liquidity farm share, vesting share, team share, public sale share, and the rest to the treasury), and the round is set to the resolved stage. The contract uses an Updater Role to set parameters like the recipient's shares, the round time, or the VRF configuration parameters.

StakingPool.sol

The staking pool allows users to stake the ecosystem's GERC20 token in order to receive rewards in the form of GERC20 tokens. It uses EIP-1973 scalable rewards to calculate each depositor's rewards. The depositors can harvest their rewards in a pull manner. The staking pool supports a static block reward and a dynamic reward. The dynamic reward kicks in when the static reward is set to 0. The staking pool can be rewarded by anyone (simple ERC20 transfer) but is intended to be rewarded from the round pool. Moreover, the staking pool supports the migration of addresses if they are approved by the migrating participant.

Treasury.sol

The treasury supports the holding of native, ERC20, ERC721, and ERC1155 tokens. It has a Governance Role, which is allowed to spend the funds.

Vesting.sol

The vesting contract allows setting user shares, which can then be vested linearly by each user. An address with Updater Role can set new user shares in a batch request. The users can then claim their shares and additionally get rewards which were distributed according to the bought amount of each address using EIP-1973 (it is intended that the round pool additionally rewards the vesting contract). The vesting contract is initially set to not open and needs to be opened by an address with Updater Role. It supports the migration of addresses.

Best Practices

The Best Practices analysis does not cover direct vulnerabilities. It shows the overall project and code structure and if best practices were applied. This is a good measurement to validate the audit result, as a clean and understandable code base indicates that a majority of bugs and vulnerabilities were found.

☑	The code was provided in a source control
☑	A technical documentation was provided
☑	Minimal code duplication
☑	Smart Contracts are unflattened
☑	A recent solidity version was used
☑	A framework for testing and deployment was used (e.g. Hardhat)
☑	There are tests
☑	Tests are easy to run
☑	There is no unused code
☑	The code follows standard Solidity naming conventions

Findings

The findings were categorized into the following four different levels:

- **Critical:** Potential loss of funds is expected. They need to be fixed immediately.
- **Medium:** Errors that can cause the contracts to fail. Manual changing needs to be done to restore the contract functionality
- **Low:** Errors that can cause the contracts to fail in specific conditions like edge cases
- **Informational:** Suggested improvements of the contracts that do not have security-related issues (e.g. gas optimization)

Critical

No Critical issues were found

Medium

RP1: Unhandled revert of NFT minting

In `contracts/RoundPool.sol` on line 268, the `mintToken` function reverts in case of the `maxSupply` being reached in `contracts/GERC721.sol`. Since this exception is not handled, the function revealRound will revert, and no winner (jackpot winner) can be evaluated for the specific round.

Recommendation

Handle the case if the max supply is reached (e.g. ignore the minting in this case), such that the other winners can still be evaluated.

Status

Resolved in commit 427bae57dd9b2cd79f8aabe0b33b8d8c3f390a29

PS1: Static index to get swap results

In `contracts/PublicSale.sol` on line 185, the index of the destination token is fixed to 1. As the length of `swapPath` can be larger than 2, the amount out of the destination token is always at the last index of the returning array and, therefore, not always at index 1.

Recommendation

Change the index from 1 to `swapPath.length - 1`

Status

Resolved in commit 427bae57dd9b2cd79f8aabe0b33b8d8c3f390a29

Low

RE1: Reentrancy vulnerability

In `contracts/RoundPool.sol` on lines 283-292, `_execureTransfers` calls external contracts, but important storage variables are written after that. If an attacker gained control over the external contracts, he could reenter the `revealRound` function, mint additional NFT, and possibly drain the contract. This is considered low as the external contracts are in the control of the project owners.

Recommendation

Consider respecting the check-interaction-effects pattern by executing the external calls after writing the storage variables. In particular, execute `_execureTransfers` after setting the storage variables.

Status

Resolved in commit 427bae57dd9b2cd79f8aabe0b33b8d8c3f390a29

UT1: Unchecked Token transfers

In the following places, the return value of the token transfers (transfer and transferFrom) is not checked. This issue is considered low as the underlying token is based on the OpenZeppelin ERC-20 implementation and therefore considered safe.

```
contracts/Vesting.sol#L85,L208, contracts/PublicSale.sol#L103,L130,L170,
contracts/RoundPool.sol#L152,L431,L440,L449,L453,L457,L461,L466,L469,
contracts/StakingPool.sol#L75,L97,L114,
contracts/LiquidityFarm.sol#L78,L100,L117, contracts/Treasury.sol#L44,L52,
contracts/Airdrop.sol#L91
```

Recommendation

Validate transfer results or use the SafeERC20 library from OpenZeppelin.

Status

Resolved in commit 427bae57dd9b2cd79f8aabe0b33b8d8c3f390a29

UZ1: Unchecked zero value for modulo operation

In `contracts/RoundPool.sol` on lines 321 and 375 the value for `roundData[round].poolValue` is not checked to be non 0. In the scenario that all entries of the round have 0 amount, the pool value can be 0, and the modulo operation will fail.

Recommendation

Handle the situation when finding the winners if the pool value of a round is zero.

Status

Resolved in commit 427bae57dd9b2cd79f8aabe0b33b8d8c3f390a29

UZ2: Unchecked zero value for airdrop participants

In `contracts/Airdrop.sol` on line 122, the value of `userReleasePeriodBlocks` is not checked to be non-0. If a release period is set to 0, the claimable tokens are reserved for the participant but are never claimable.

Recommendation

Consider checking for non-0 values in `userReleasePeriodBlocks`.

Status

Resolved in commit 427bae57dd9b2cd79f8aabe0b33b8d8c3f390a29

MG1: Missing gap variables

In `contracts/GERC721.sol` and `contracts/RoundPool.sol`, no additional storage is reserved for future contract upgrades. They are missing the suggested __gap variables at the end of the contract.

Recommendation

Add `uint256[50] private __gap` to the end of these contracts.

Status

Resolved in commit 427bae57dd9b2cd79f8aabe0b33b8d8c3f390a29

UC1: Unchecked underflow

In `contracts/PublicSale.sol` on line 212, the `releasedAmount` can become smaller than `userData[msg.sender].spentAmount`. This is the case if the updater sets a new `releasePeriodBlocks` to a value larger than the current one. The release amount will decrease and can become smaller than the already claimed amount.

Recommendation

Handle the situation when the `releasedAmount` gets smaller than the `userData[msg.sender].spentAmount` such that no underflow can occur.

Status

Resolved in commit 427bae57dd9b2cd79f8aabe0b33b8d8c3f390a29

MI1: Missing initializers

In `contracts/Governor.sol`, `contracts/GERC20.sol` and `contracts/GERC721.sol`, some child initializers are missing. This is considered low as these initializers are checked to be empty. Nevertheless, calling all initializers is considered best practice.

Recommendation

Add the following initializers:

`contracts/Governor.sol`, add: `GovernorCountingSimpleUpgradeable.__GovernorCountingSimple_init();` to the initialize function

`contracts/GERC20.sol` add:

`__ERC20Votes_init();` to the initialize function

`contracts/GERC721.sol` add:

`__ERC721Enumerable_init();` to the initialize function

Status

Resolved in commit 427bae57dd9b2cd79f8aabe0b33b8d8c3f390a29

Informational

CL1: Costly loop

In `contracts/Vesting.sol` on line 231, `totalShares` is set in every loop iteration. As this is a storage variable, there are high gas costs by increasing it in every iteration.

Recommendation

Consider using a memory variable to track the value and set `totalShares` after the loop.

Status

Resolved in commit 427bae57dd9b2cd79f8aabe0b33b8d8c3f390a29

EF1: Set public functions to external

Functions that are only called from external sources should be declared external to optimize gas costs. The following functions are visible as public but are not accessed within the contract:

```
contracts/Airdrop.sol: claimTokens, grantAirdrops, cancelAirdrop,
migrateAddress, approveAddressMigration

contracts/GERC20.sol: setTransferTaxPercentage, setRoundPool,
setExemptRecipient, setTaxedContract, setTaxFreeUser

contracts/GERC721.sol: setBaseUri, mintToken

contracts/Governor.sol: getLockedReserveList, updateQuorumPercentage,
updateVotingDelayBlocks
```

`contracts/LiquidityFarm.sol: setLpTokenAddress, migrateAddress, approveAddressMigration, setDynamicRewardPeriod, pendingRewards, setStaticBlockRewards, stake, unstake`

`contracts/PublicSale.sol: setSwapPath, setReleasePeriodBlocks, buy, refund, migrateAddress, finishSale, approveAddressMigration, cancelSale, openSale, claimTokens`

`contracts/RoundPool.sol: updatePublicSaleRewardsEndBlock, updateRoundTime, addEntry, updateRecipientShares, updateVrfConfiguration, getRoundTransfer, getRoundTransferCount, updateRecipientAddresses, seedRound, revealRound`

`contracts/StakingPool.sol: migrateAddress, stake, pendingRewards, setStaticBlockReward, approveAddressMigration, setDynamicRewardPeriod, unstake`

`contracts/Treasury.sol: withdrawERC721, withdrawERC20, withdraw, withdrawERC1155, deposit`

`contracts/Vesting.sol: migrateAddress, openVesting, setUserShares, claim, addVestedTokens, getClaimableRewardTokens`

Recommendation

Change the functions from `public` to `external`

Status

Resolved in commit 427bae57dd9b2cd79f8aabe0b33b8d8c3f390a29

MF1: Missing functionality to claim available tokens

In `contracts/Airdrop.sol`, there is no functionality to withdraw available tokens. Although there is a functionality for canceling the airdrop for a specific user, there is no functionality to withdraw the newly available tokens.

Recommendation

Consider adding a function with access control to withdraw the available tokens.

Status

Acknowledged

TP1: Typo in function name

In `contracts/RoundPool.sol`, there is a typo in the function name `_execureTransfers`.

Recommendation

Consider changing the function name to `_executeTransfers`.

Status

Resolved in commit 427bae57dd9b2cd79f8aabe0b33b8d8c3f390a29

SP1: msg.sender in view function

In `contracts/PublicSale.sol`, in the function `getAvailableAmount()`, `msg.sender` is used. As it is a view function, it will be inconvenient to debug the available amounts for a user.

Recommendation

Consider adding a parameter `address _userAddress` to the function declaration and replace `msg.sender` with this address within the function.

Status

Resolved in commit 5b2ef9b4552fb56423ec7dc2003ad51689a88372

Limitation

This code review was conducted carefully and on a best-effort basis. However, this does not guarantee that there are any undiscovered issues and vulnerabilities. This audit gives no warranties on the security of the code.