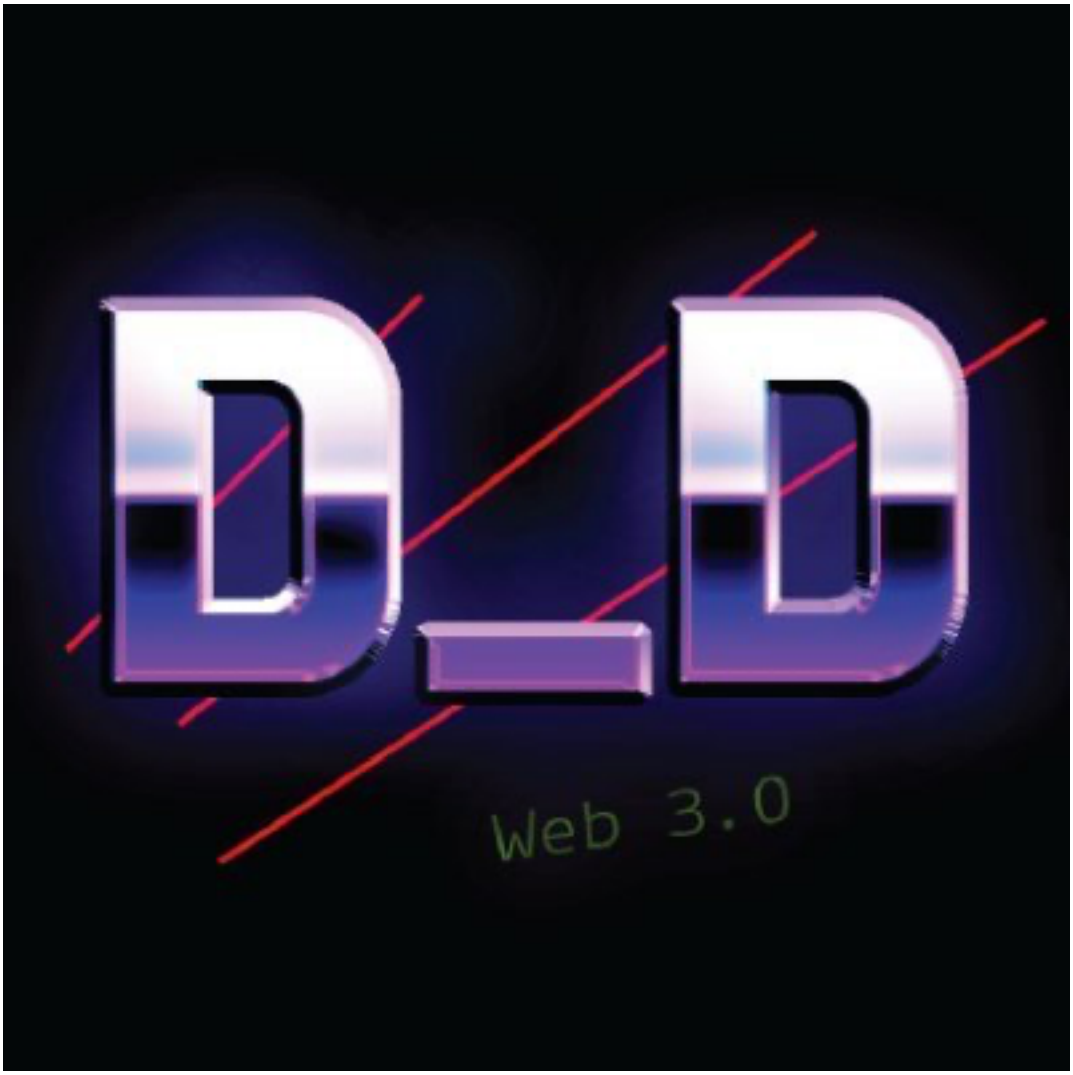

Internal $CODE token security review

Authors:
BowTiedHeron, Dravee

Contributors:
BowTiedHeron, Dravee, Parseb, Sparx, BowTiedPickle, PeterPan, Cachemonet, Crypdough, Indreams

Developers:
Heyo, Martin, Mario

Table of Contents

Disclaimer

This internal review is not, nor should be considered, an “endorsement” or “disapproval” of any project or team. This internal review is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by Developer DAO.

This internal review does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technology, business model or legal compliance of Developer DAO.

This internal review should not be used in any way to make decisions around investment or involvement with any particular project. This internal review does not provide investment advice, nor should be leveraged as investment advice of any sort.

This internal review represents an attempt to increase the quality of the $CODE token and claim contract code. Blockchain technology and cryptographic assets present a high level of ongoing risk and everyone should do their own due diligence before interacting with any token/contracts on the blockchain.

What is an internal security review?

- A document containing (potential) security vulnerabilities, mitigation measures and suggestions for gas optimization.
- A representation that Developer DAO completed an internal review with the intention to increase the quality of the contracts.

What is an internal security review not?

- A statement about the overall bug free or vulnerability free nature of a piece of source code or any modules, technologies or code it interacts with.
- A Guarantee or warranty of any sort regarding the intended functionality or security of any or all technology referenced in the report.
- An endorsement or disapproval of any company, team or technology.

Summaries

Project Summary

Name	$CODE Governance Token
Description	$CODE Developer DAO governance token and claim contract.
Platform	Ethereum
Codebase	https://github.com/Developer-DAO/code-claim-site/tree/67150ee4a05aa9c96c4765e70143185476ae37bc/packages/hardhat

Review Summary

Delivery date	27-6-2022
Methods used	Manual Review, Automated Scans (proprietary and Myth-x)
Internal members engaged	10
Timeline	29-5-2022 / 17-6-2022

Vulnerability Summary

Total Findings	14
High Risk	0
Medium Risk	2
Low Risk	2
Informational	5
Gas Optimizations	5

Risk classification

High	Assets can be stolen/lost/compromised.
Medium	Assets are not at risk: state handling, function incorrect as to spec, issues with comments.
Low	Assets are not at risk: state handling, function incorrect as to spec, issues with comments.
Informational	Code style, clarity, syntax, versioning, off-chain monitoring (events, etc)
Gas	Optimizations to save on transaction and deployment costs

Findings

ID	Title	Type	Severity
D_D - 01	Use safeTransfer / safeTransferFrom consistently instead of transfer / transferFrom		Medium
D_D - 02	Using transferFrom on ERC721 tokens		Medium
D_D - 03	abi.encodePacked() should not be used with dynamic types when passing the result to a hash function such as keccak256()		Low
D_D - 04	ClaimCODE.sol should implement a 2-step ownership transfer pattern		Low
D_D - 05	It's better to emit after all processing is done		Informational
D_D - 06	Avoid floating pragmas: the version should be locked		Informational
D_D - 07	Use a more **recent/stable** version of solidity		Informational
D_D - 08	Remove import: hardhat/console.sol		Informational
D_D - 09	Use msg.sender instead of OpenZeppelin's msgSender()		Informational / Gas
D_D - 10	Amounts should be checked for 0 before calling a transfer		Gas
D_D - 11	An array's length should be cached to save gas in for-loops		Gas
D_D - 12	++i costs less gas compared to i++		Gas
D_D - 13	Increments/decrements can be unchecked in for-loops		Gas
D_D - 14	No need to explicitly initialize variables with default values		Gas

D_D – 01: Use safeTransfer/safeTransferFrom consistently instead of transfer/transferFrom

Type	Severity	Reference
	Medium/Low	ClaimCode.sol#L71 Code.sol#L30

Description

Transfer functions don't use a require() statement that checks the return value of token transfers. It is good practice to use something like OpenZeppelin's safeTransfer / safeTransferFrom unless one is sure the given token reverts in case of a failure. Failure to do so will cause silent failures of transfers and affect token accounting in contract.

Reference:
This similar medium-severity finding from Consensys Diligence Audit of Fei Protocol:
https://consensys.net/diligence/audits/2021/01/fei-protocol/#unchecked-return-value-for-iweth-transfer-call

Recommendation

Consider adding a require-statement or using safeTransfer.

```
packages/hardhat/src/ClaimCODE.sol:
  71:         token.transfer(owner(), token.balanceOf(address(this))); //@audit result of transfer not checked

packages/hardhat/src/CODE.sol:
  30:         token.transfer(_to, token.balanceOf(address(this))); //@audit result of transfer not checked
```

Note that safeTransfer introduces a callback that could be exploited in a reentrancy exploit.

The current version of the contract respects Check Effect Interact patterns, which update the balances before any tokens are sent; this also directly blocks this attack vector. CEI and Reentrancy guards can be used to protect the protocol against such vulnerabilities.

Besides that, the mentioned functions are used to sweep tokens that are sent to the contract by accident and can only be called by someone that has the SWEEP_ROLE or the OWNER of the contract.

D_D – 02: Using transferFrom on ERC721 tokens

Type	Severity	Reference
	Medium	Code.sol#L40

Description

The transferFrom keyword is used instead of safeTransferFrom. If the arbitrary address is a contract and is not aware of the incoming ERC721 token, the sent token could be locked.

```
packages/hardhat/src/CODE.sol:
  40:         token.transferFrom(address(this), _to, _tokenID);
```

Recommendation

We suggest transitioning from transferFrom to safeTransferFrom.

Note that safeTransfer introduces a callback that could be exploited in a reentrancy exploit.

The current version of the contract respects Check Effect Interact patterns, which update the balances before any tokens are sent; this also directly blocks this attack vector. CEI and Reentrancy guards can be used to protect the protocol against such vulnerabilities.

D_D – 03: abi.encodePacked() should not be used with dynamic types when passing the result to a hash function such as keccak256()

Type	Severity	Reference
	Low	ClaimCODE.sol#L46 MerkleProof.sol#L36 MerkleProof.sol#L39

Description

Use of abi.encodePacked in PositionKey is safe, but unnecessary and not recommended. abi.encodePacked can result in hash collisions when used with two dynamic arguments (string/bytes). PositionKey does not use any dynamic types, but for maximum safety against future mistakes, using abi.encode is recommended.

There is also discussion of removing abi.encodePacked from future versions of Solidity (ethereum/solidity#11593), so using abi.encode now will ensure compatibility in the future.

```
packages/hardhat/src/ClaimCODE.sol:
   2: pragma solidity ^0.8.9;
  46:         bytes32 leaf = keccak256(abi.encodePacked(msg.sender, _amount));

packages/hardhat/src/MerkleProof.sol:
   4: pragma solidity ^0.8.9;
  36:                 computedHash = keccak256(abi.encodePacked(computedHash, proofElement));
  39:                 computedHash = keccak256(abi.encodePacked(proofElement, computedHash));
```

Recommendation

Mitigation 1: Use a solidity version of at least 0.8.12 to get string.concat() to be used instead of abi.encodePacked(<str>,<str>).

Mitigation 2: Use abi.encode() instead which will pad items to 32 bytes, which will prevent hash collisions (e.g. abi.encodePacked(0x123,0x456) => 0x123456 => abi.encodePacked(0x1,0x23456), but abi.encode(0x123,0x456) => 0x0...1230...456). If there is only one argument to abi.encodePacked() it can often be cast to bytes() or bytes32() instead.

D_D – 04: ClaimCODE.sol should implement a 2-step ownership transfer pattern

Type	Severity	Reference
	Low	ClaimCODE.sol#L13

Description

This contract inherits from OpenZeppelin's libraries and the transferOwnership() function is the default one (a one-step process). It's possible that the onlyOwner role mistakenly transfers ownership to a wrong address, resulting in a loss of the onlyOwner role.

```
packages/hardhat/src/ClaimCODE.sol:
   5: import "@openzeppelin/contracts/access/Ownable.sol";
  13: contract ClaimCODE is Ownable, Pausable {
```

Recommendation

Consider overriding the default transferOwnership() function to first nominate an address as the pending owner and implementing an acceptOwnership() function which is called by the pending owner to confirm the transfer.

D_D – 05: It's better to emit after all processing is done

Type	Severity	Reference
	Informational	Code.sol#L53

Description

Event is emitted before token transfer is completed.

```
File: ClaimCODE.sol
45:     function claimTokens(uint256 _amount, bytes32[] calldata _merkleProof) external whenNotPaused {
....
53:         emit Claim(msg.sender, _amount); //@audit it's better to emit after all is done. Move this further down
54:
55:         codeToken.transfer(msg.sender, _amount);
56:     }
```

Recommendation

It is better to emit events at the end of the function call.

Only example when this is not the case is for bridges.

D_D – 06: Avoid floating pragmas: the version should be locked

Type	Severity	Reference
	Informational	Code.sol ClaimCODE.sol MerkleProof.sol

Description

Contracts should be deployed using the same compiler version/flags with which they have been tested. Locking the floating pragma, i.e. by not using ^ in pragma solidity ^0.8.9, ensures that contracts do not accidentally get deployed using an older compiler version with unfixed bugs.

For reference, see https://swcregistry.io/docs/SWC-103

```
File: low.md
183: ClaimCODE.sol:2:pragma solidity ^0.8.9;
184: CODE.sol:2:pragma solidity ^0.8.9;
185: MerkleProof.sol:4:pragma solidity ^0.8.9;
```

Recommendation

Avoid using floating pragmas and lock the version of the solidity compiler.

D_D – 07: Use a more recent/stable version of solidity

Type	Severity	Reference
	Informational	Code.sol ClaimCODE.sol MerkleProof.sol

Description

There was some internal discussion regarding the Solidity version being used.

Some argued using an older version of Solidity; specifically 0.8.4, since that version of Solidity has the custom error functions and should be stable enough (meaning people are aware of potential bugs).

Others argued that a more recent version of Solidity should be used to make use of recent improvements. Like 0.8.12 or higher, as mentioned in D_D – 03.

```
File: low.md
183: ClaimCODE.sol:2:pragma solidity ^0.8.9;
184: CODE.sol:2:pragma solidity ^0.8.9;
185: MerkleProof.sol:4:pragma solidity ^0.8.9;
```

Recommendation

We suggest having an open discussion between the devs and people that are willing to give input on this to decide on the final version being used.

D_D – 08: Remove import: hardhat/console.sol

Type	Severity	Reference
	Informational	ClaimCODE.sol#L11

Description

Hardhat console.sol import still present in the code.

```
File: ClaimCODE.sol
11: import "hardhat/console.sol"; //@audit : this is debug-code. Remove it
```

Recommendation

Hardhat console.sol import should be removed before deployment/audit.

D_D – 09: Use msg.sender instead of OpenZeppelin's msgSender()

Type	Severity	Reference
	Informational/Gas	Code.sol#L21

Description

msg.sender costs 2 gas (CALLER opcode). _msgSender() represents the following:

```
File: @openzeppelin/contracts/utils/Context.sol
17:     function _msgSender() internal view virtual returns (address) {
18:         return msg.sender;
19:     }
```

When no GSN capabilities are used: msg.sender is enough.

See https://docs.openzeppelin.com/contracts/2.x/gsn for more information about GSN capabilities.

Affected code (see @audit tag):

```
File: CODE.sol
19:     constructor(address _treasury) ERC20("Developer DAO", "CODE") ERC20Permit("Developer DAO") {
20:         _setupRole(DEFAULT_ADMIN_ROLE, _treasury);
21:         _mint(_msgSender(), 10_000_000 * 1e18); //@audit gas: Replace `_msgSender()` with `msg.sender`
22:     }
```

Recommendation

Use msg.sender instead of OpenZeppelin's _msgSender().

D_D – 10: Amounts should be checked for 0 before calling a transfer

Type	Severity	Reference
	Gas	ClaimCODE.sol#L55 Code.sol#L50

Description

Checking non-zero transfer values can avoid an expensive external call and save gas.

```
packages/hardhat/src/ClaimCODE.sol:
  55:         codeToken.transfer(msg.sender, _amount);

packages/hardhat/src/CODE.sol:
  50:         super._afterTokenTransfer(_from, _to, _amount);
```

Recommendation

We suggest adding a non-zero-value check.

D_D – 11: An array's length should be cached to save gas in for-loops

Type	Severity	Reference
	Gas	MerkleProof.sol#L30

Description

Reading array length at each iteration of the loop consumes more gas than necessary.

```
MerkleProof.sol:30:        for (uint256 i = 0; i < proof.length; i++) {
```

In the best case scenario (length read on a memory variable), caching the array length in the stack saves around 3 gas per iteration. In the worst case scenario (external calls at each iteration), the amount of gas wasted can be massive.

Recommendation

Storing the array's length in a variable before the for-loop and use this new variable instead will save gas.

D_D – 12: ++i costs less gas compared to i++

Type	Severity	Reference
	Gas	MerkleProof.sol#L30 MerkleProof.sol#L40

Description
Pre-increments and pre-decrements are cheaper.

For a uint256 i variable, the following is true with the Optimizer enabled at 10k:

Increment:
i += 1 is the most expensive form
i++ costs 6 gas less than i += 1
++i costs 5 gas less than i++ (11 gas less than i += 1)

Decrement:
i -= 1 is the most expensive form
i-- costs 11 gas less than i -= 1
--i costs 5 gas less than i-- (16 gas less than i -= 1)

Note that post-increments (or post-decrements) return the old value before incrementing or decrementing, hence the name post-increment:

```
uint i = 1;
uint j = 2;
require(j == i++, "This will be false as i is incremented after the comparison");
```

However, pre-increments (or pre-decrements) return the new value:

```
uint i = 1;
uint j = 2;
require(j == ++i, "This will be true as i is incremented before the comparison");
```

In the pre-increment case, the compiler has to create a temporary variable (when used) for returning 1 instead of 2.

Affected code:

```
MerkleProof.sol:30:        for (uint256 i = 0; i < proof.length; i++) {
MerkleProof.sol:40:                index += 1;
```

Recommendation

Consider using pre-increments and pre-decrements where they are relevant (meaning: not where post-increments/decrements logic are relevant).

D_D – 13: Increments/decrements can be unchecked in for-loops

Type	Severity	Reference
	Gas	MerkleProof.sol#L30

Description

In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

ethereum/solidity#10695

Affected code:

```
MerkleProof.sol:30:        for (uint256 i = 0; i < proof.length; i++) {
```

Recommendation

The suggested change would be:

```
- for (uint256 i; i < numIterations; i++) {
+ for (uint256 i; i < numIterations;) {
  // ...
+    unchecked { ++i; }
}
```

The same can be applied with decrements (which should use break when i == 0).

The risk of overflow is non-existent for uint256 here.

D_D – 14: No need to explicitly initialize variables with default values

Type	Severity	Reference
	Gas	MerkleProof.sol#L28 MerkleProof.sol#L30

Description

If a variable is not set/initialized, it is assumed to have the default value (0 for uint, false for bool, address(0) for address...). Explicitly initializing it with its default value is an anti-pattern and wastes gas.

As an example: for (uint256 i = 0; i < numIterations; ++i) { should be replaced with for (uint256 i; i < numIterations; ++i) }

Affected code:

```
MerkleProof.sol:28:         uint256 index = 0;
MerkleProof.sol:30:         for (uint256 i = 0; i < proof.length; i++) {
```

Recommendation

We suggest removing explicit initializations for default values.