Smart Contract Security Audit

Altcoinist Project

Author: *Awalcon Hacker Team from CCTF DAO*
Date: *2023.12.21.*

Table of Contents

Disclaimer

The list of findings and recommendations are summarized in the Audit Results.

The matters raised in this report are only those identified during the review and are not necessarily a comprehensive statement of all weaknesses that exist or all actions that might be taken. This work was performed under limitations of time and scope that are not potentially relevant to the actions of a malicious attack.

The review is based at a specific point in time, in an environment where both the systems and the threat profiles are dynamically evolving. It is therefore possible that vulnerabilities exist or will arise that were not identified during the review and there may or will have been events, developments and changes in circumstances subsequent to its issue.

The security analysis is purely based on the provided smart contracts alone. No other products or systems have been reviewed. The purpose of the audit is to identify issues related to the logic and quality of the code.

Executive Summary

The security code audit of the Altcoinist project revealed a robust and fairly secure codebase, demonstrating strong understanding of best practices in smart contract design.

There were no critical vulnerabilities identified, affirming the contract's resilience, however, the audit did uncover one high-level vulnerability. This issue, while significant, is relatively straightforward to fix and does not undermine the overall integrity of the contract.

The recommended fixes are documented here and should be easy for the developer to implement, further enhancing the contracts' security and performance.

Overall, the audit's findings are positive, reflecting the developer's expertise and the contracts' sound architecture.

Audit overview

The contracted project requested a security code audit on their new smart contracts and on some of the changes they made after the first audit.

Start date of the audit: 2023.12.11.

Report date: 2023.12.21.

Project website:
Platform: Solidity / Ethereum L2
Code author:
Audited commit:

> *sha256sum contracts.tar.gz*
cb7861913b6bfd85d02c5aef5f933ffa60125bb7315d45e88db8669ae96f1382

Overall result: **Pass**

Auditors:

six ~ *PGP 450F 4AC8 0BD8 ~ six@cryptoctf.org*

def1dec0ded ~ 0xdef1dec0ded@cryptoctf.org

Objective and methodology

The objective of the security assessment is to gain insight into the security of the smart contracts listed in the scope.

Code review main check items:

- Line-by-line audit
- Business logic
- Data consistency
- Coding style violations
- Gas usage
- Reentrancy
- Test with automated tools:
  - Static analysis with Slither
  - Fuzzing with Harvey
  - Symbolic execution with Myth

Further documents incorporated in the methodology:

- Smart Contract Weakness Classification Registry - https://swcregistry.io/
- https://github.com/miguelmota/solidity-audit-checklist

Smart contracts and scenarios summary

The audit includes the following contracts:

- ALTT.sol – Main contract
- SubscribeRegistry.sol – Subscription handling contract
- SubstakingFactory.sol – Staking contract
- SubstakingVault.sol – Vault contract
- XPRedeem.sol – Redeeming tokens, including ECDSA logic

Manual, fuzzing and symbolic execution test cases

Tool used for symbolic execution: Myth.

Tool used for fuzzing: Harvey

List of test scenarios

- Attacks against implementation of ECDSA cryptography (forging, replay, logic)
- Caller can redirect execution to arbitrary bytecode locations
- Caller can write to arbitrary storage locations
- Delegatecall to a user-specified address
- Control flow depends on a predictable environment variable
- Control flow depends on tx.origin
- Any sender can withdraw ETH from the contract account
- Assertion violation
- External call to another contract
- Integer overflow or underflow
- Multiple external calls in the same transaction
- State change after an external call
- Contract can be accidentally killed by anyone
- Return value of an external call is not checked
- A user-defined assertion has been triggered

Audit results

Critical severity

No critical severity issue have been found during the manual code review or by using automated tools.

High severity

ECDSA signature replay attack on multi-chain scenario [Fixed]

Description and exploitation

The chain ID is not checked on the XPRedeem smart contract at the redeemXP() function.

If there is no test network or any way to get a valid signature from the backend, this is not exploitable. However, in case of a public test run or any other scenario where a valid signature appears publicly, it opens up to possibility to use it on any chain. This can leak to invalid amount processed on the attacked smart contracts.

Exploitation: copy the signed data and transact it on another chain.

Recommendation

Check the Chain ID and the smart contract address. Example for the first:

```
function getChainID() external view returns (uint256) {
    uint256 id;
    assembly {
        id := chainid()
    }
    return id;}
```

Reference

https://eips.ethereum.org/EIPS/eip-155

Medium severity

Unmatching values for SubscribeRegistry (docs!=code) [Fixed]

Description and exploitation

Numbers does not match the Gitbook (80% goes to author is mentioned no that 32% goes to ref if there is a ref - the 32% is too much, one is too tempted to set up a referral pool contract - poor protocol design). The number 32 is not in the Gitbook at all.

Recommendation

Decide on the correct values and change the docs or the code.

Low severity

Locking balances through XPRedeem has a SPoF [Accepted]

Description and exploitation

The XPRedeem smart contract relies on a single signer in the background that potentially locks balances and can only by recover itself (the offchainSigner).

The “offchainSigner” can’t tell upfront what is the amount of ALTT required for the payouts. A lot may get stuck in the contract in an irrecoverable way if users redeem early.

Users either must trust the signer fully to provide "refill" the contract form time to time or the signer needs to provide the full promised amount to all users along with a signed longer deadline proof.

Also, the signature contains msg.sender implicitly so it would be more "unexpected-user-error-prone" to add the address in the signature to the parameters of the function call.

Recommendation

- Consider using only backend’s system as the smart contract seems to be an extra layer that is not making it more decentralized and opens up the potential locked balances.
- Change it to be less centralized.

Lack of LICENSES [Fixed]

Description

Licenses are not defined in XPRedeem smart contract and the git repository also has LICENSE.md missing.

Proposed solution

Provide a business compatible license for the whole project.

Miscellaneous findings

Function declarations could be stricter

Description

We found functions can be declared as external, for more limited access.

This is considered low because the project will known to have slight changes before deployment, and these declarations might have a more serious impact on the future version.

Reference

https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external

Further list of misc findings

- We highly recommend to run swap tests after public deployment because there are differences between the L1 and L2 implementations. Some of these might actually break logic or functionality. Only announce the project after carefully testing all functionalities!
- SubscribeRegistry: Why cant the user preemptively subscribe/extend subscription expiration? *require(expiry < block.timestamp, "AS")* should take > of the timestamp, expiry and add the amount to that one. This is not user-friendly and they might miss important content for them! Make sure to add replay attack protection when implementing the +time for the subscription.
- Centralization: you really need to make sure that the owner/admin accounts are actually not handled by a single entity. This is not on the smart contract side now, but could be, eg. https://github.com/Qrucial/Voronoi-Docs
- ReentrancyGuard is not needed as ALTT transfers has no callback hook, though better have it then not, this will slightly increase gas usage.
- ERC20Permit is replayable, usage is not recommended. We list it at misc as it is know to be an optional future for the project. If it won't get deployed, there will be no problem.
- ALTT addLiquidity()'s return: why return anything when ret? Value is not used and increases gas costs. Compiler optimization may cut this. Not a security risk.
- ALTT amount0Min: Why not equal amount0toMint? It makes the owner able to start the pool 2 % below target price.
- SubstakingVault: probably in init() could also require _name != ""
- Swapwethforaltt: "amountOutMinimum: 0" - this could be potentially frontrunnable. However it doesn't seem to be a realistically worthy attack.

- SubstakingVault: Why return amount is depositweth? The caller knows the ret val anyways.
- SubstakingVault: What will make this not to revert, how will it be:
  - *>0? : require(wethBalance > wethDepositSum && wethDepositSum > 0);*
  - After TGE, no ETH is passed to the pool, before TGE, wethbalance=wethdepositsum, who will deposit the additional WETH?
- SubscribeRegistry: This line is redundant as the transfer would fail w/o this, too: require(weth.balanceOf(sender) >= basePrice, "IB");
- SubscribeRegistry: It would be likely more gas efficient to first transfer all amount to the Registry, then safetransfer to red and author and team.
- SubscribeRegistry: This will always pass, unless the tx reverts so unnecessary gas. require(sent == toPool);
- Why approve and transfer? Why not just altt.transfer?
  - TransferHelper.safeApprove(address(altt), factory.vaults(author), alttReceived);
  - TransferHelper.safeTransfer(address(altt), factory.vaults(author), alttReceived)
- SubscribeRegistry: The team does not trust their own token to receive their share in it, same applies to author:
  - *weth.safeTransferFrom(sender, teamAddress, toTeam);*

Contact

Awalcon Team – six

Website: *https://awalcon.org/*

E-mail: *six@cryptoctf.org*

Git: *https://git.hsbp.org/six*

PGP*: B1F7 B1D6 8838 98B4 2212 1D90 CA71 D1E4 078E 99C5*

Awalcon Team – mPeter

Website: *https://awalcon.org/*

E-mail: *mpeteriii@cryptoctf.org*