SOLIDProof

Bring trust into your projects

Blockchain Security | Smart Contract Audits | KYC

MADE IN GERMANY

Bullperks

Audit

Security Assessment

13. August, 2022

For

BullPerks

SolidProof_io

@solidproof_io

Disclaimer

SolidProof.io reports are not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. These reports are not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team. SolidProof.io do not cover testing or auditing the integration with external contract or services (such as Unicrypt, Uniswap, PancakeSwap etc’...)

SolidProof.io Audits do 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 proprietors. SolidProof Audits should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

SolidProof.io Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology. Blockchain technology and cryptographic assets present a high level of ongoing risk. SolidProof’s position is that each company and individual are responsible for their own due diligence and continuous security. SolidProof in no way claims any guarantee of security or functionality of the technology we agree to analyze.

Version	Date	Description
1.0	06. - 12. August 2022	• Layout project • Automated- /Manual-Security Testing • Summary

Network

Solana (Rust)

Website

https://bullperks.com/

Telegram

https://t.me/BullPerksAnnouncements

Twitter

https://twitter.com/bullperks

Facebook

https://www.facebook.com/bullperks

Instagram

https://www.instagram.com/bullperks_vc/

Reddit

https://www.reddit.com/r/BullPerks_VC/

Medium

https://medium.com/bull-perks

Discord

https://discord.gg/5avWfavp2n

Youtube

https://www.youtube.com/channel/UCIY2Vz-X3vmBLSqbMcPZyMQ

LinkedIn

https://www.linkedin.com/company/77364631/admin/

Description

The fairest and most community-oriented decentralized VC and multichain launchpad

Project Engagement

During the 3rd of August 2022, **Bullperks Team** engaged Solidproof.io to audit smart contracts that they created. The engagement was technical in nature and focused on identifying security flaws in the design and implementation of the contracts. They provided Solidproof.io with access to their code repository and whitepaper.

Logo

Contract Link

v1.0

- Github
    - https://github.com/bullperks/claiming_contracts_solana
    - Commit: 664d08ae95392e6491f9b52cb554f85c11a4c112

Vulnerability & Risk Level

Risk represents the probability that a certain source-threat will exploit vulnerability, and the impact of that event on the organization or system. Risk Level is computed based on CVSS version 3.0.

Level	Value	Vulnerability	Risk (Required Action)
Critical	9 - 10	A vulnerability that can disrupt the contract functioning in a number of scenarios, or creates a risk that the contract may be broken.	Immediate action to reduce risk level.
High	7 – 8.9	A vulnerability that affects the desired outcome when using a contract, or provides the opportunity to use a contract in an unintended way.	Implementation of corrective actions as soon aspossible.
Medium	4 – 6.9	A vulnerability that could affect the desired outcome of executing the contract in a specific scenario.	Implementation of corrective actions in a certain period.
Low	2 – 3.9	A vulnerability that does not have a significant impact on possible scenarios for the use of the contract and is probably subjective.	Implementation of certain corrective actions or accepting the risk.
Informational	0 – 1.9	A vulnerability that have informational character but is not effecting any of the code.	An observation that does not determine a level of risk

Auditing Strategy and Techniques Applied

Throughout the review process, care was taken to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices. To do so, reviewed line-by-line by our team of expert pentesters and smart contract developers, documenting any issues as there were discovered.

Methodology

The auditing process follows a routine series of steps:

1. Code review that includes the following:
   i) Review of the specifications, sources, and instructions provided to SolidProof to make sure we understand the size, scope, and functionality of the smart contract.
   ii) Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   iii) Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to SolidProof describe.

2. Testing and automated analysis that includes the following:
   i) Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.

3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.

4. Specific, itemized, actionable recommendations to help you take steps to secure your smart contracts.

Scope of Work/Verify Claims

The above token Team provided us with the files that needs to be tested (Github, Bscscan, Etherscan, files, etc.). The scope of the audit is the main contract (usual the same name as team appended with .rs).

We will verify the following claims:

1. Missing signer checks
2. Missing ownership checks
3. Missing rent exemption checks
4. Signed invocation of unverified programs
5. Solana account confusions
6. Re-initiation with cross-instance confusion
7. Arithmetic overflow/underflows
8. Numerical precision errors
9. Loss of precision in calculation
10. Incorrect calculation
11. Casting truncation
12. Exponential complexity in calculation
13. Missing freeze authority checks
14. Insufficient SPL-Token account verification
15. Over/under payment of loans
16. Overall checkup (Smart Contract Security)

Overall checkup (Smart Contract Security)

Tested	Verified
✓	✓

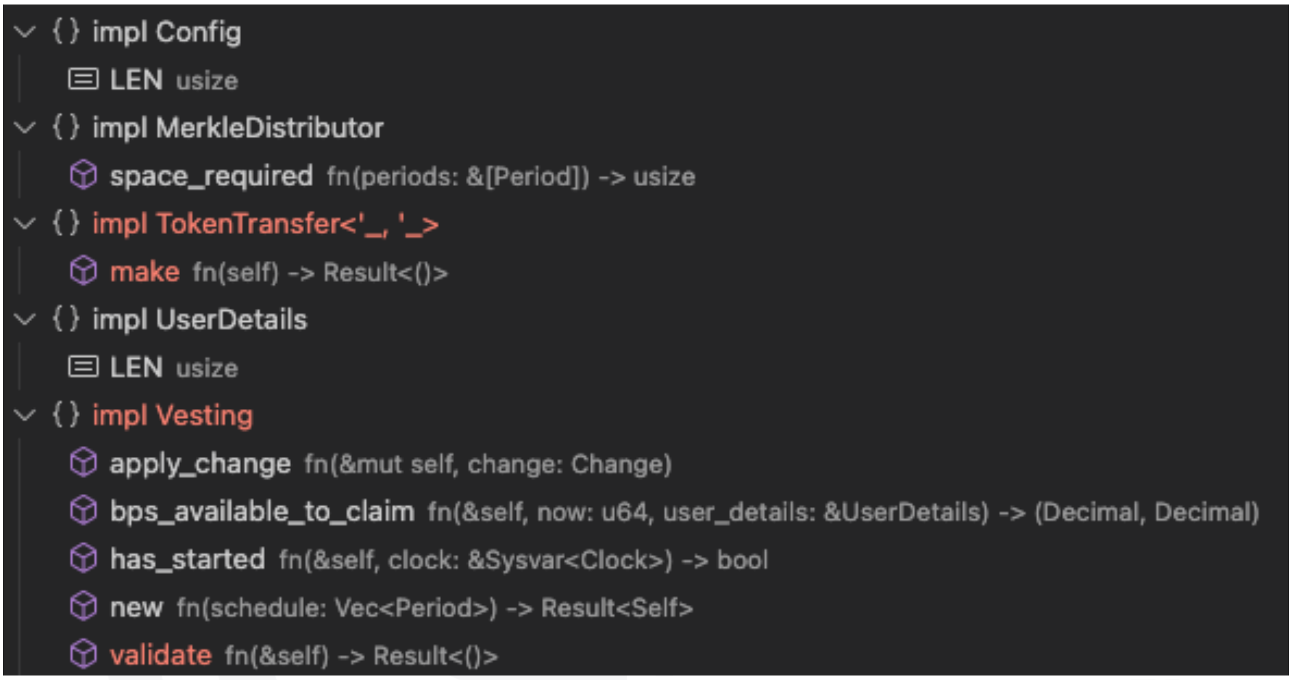
Legend

Attribute	Symbol
Verfified / Checked	✓
Partly Verified	⚑
Unverified / Not checked	✗
Not available	–

Modifiers and public functions

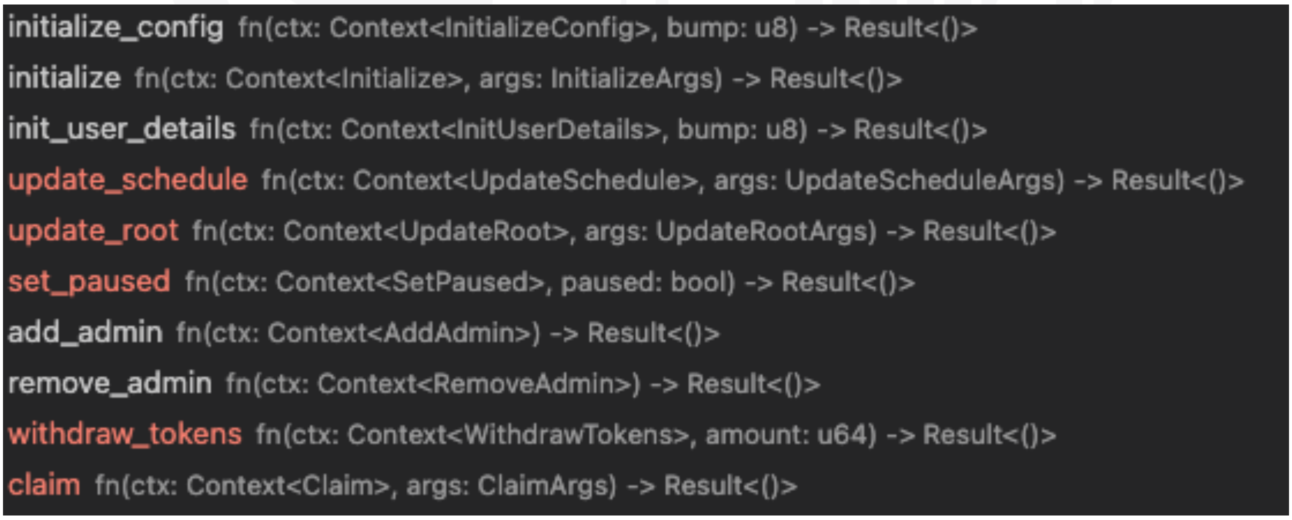
v1.0

Implements



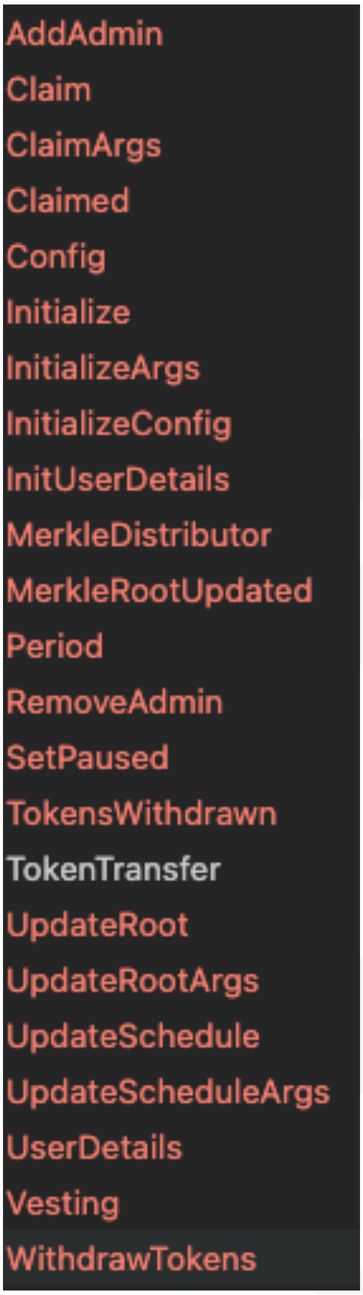
```
{} impl Config
    LEN usize
{} impl MerkleDistributor
    space_required fn(periods: &[Period]) -> usize
{} impl TokenTransfer<'_, '_>
    make fn(self) -> Result<()>
{} impl UserDetails
    LEN usize
{} impl Vesting
    apply_change fn(&mut self, change: Change)
    bps_available_to_claim fn(&self, now: u64, user_details: &UserDetails) -> (Decimal, Decimal)
    has_started fn(&self, clock: &Sysvar<Clock>) -> bool
    new fn(schedule: Vec<Period>) -> Result<Self>
    validate fn(&self) -> Result<()>
```

Public functions



```
initialize_config fn(ctx: Context<InitializeConfig>, bump: u8) -> Result<()>
initialize fn(ctx: Context<Initialize>, args: InitializeArgs) -> Result<()>
init_user_details fn(ctx: Context<InitUserDetails>, bump: u8) -> Result<()>
update_schedule fn(ctx: Context<UpdateSchedule>, args: UpdateScheduleArgs) -> Result<()>
update_root fn(ctx: Context<UpdateRoot>, args: UpdateRootArgs) -> Result<()>
set_paused fn(ctx: Context<SetPaused>, paused: bool) -> Result<()>
add_admin fn(ctx: Context<AddAdmin>) -> Result<()>
remove_admin fn(ctx: Context<RemoveAdmin>) -> Result<()>
withdraw_tokens fn(ctx: Context<WithdrawTokens>, amount: u64) -> Result<()>
claim fn(ctx: Context<Claim>, args: ClaimArgs) -> Result<()>
```

Structs



```
AddAdmin
Claim
ClaimArgs
Claimed
Config
Initialize
InitializeArgs
InitializeConfig
InitUserDetails
MerkleDistributor
MerkleRootUpdated
Period
RemoveAdmin
SetPaused
TokensWithdrawn
TokenTransfer
UpdateRoot
UpdateRootArgs
UpdateSchedule
UpdateScheduleArgs
UserDetails
Vesting
WithdrawTokens
```

Comments

- N/A

Please check if an OnlyOwner or similar restrictive modifier has been forgotten.

Audit Results

AUDIT PASSED

Critical issues

No critical issues

High issues

Issue	File	Type	Line/ Category	Description
#1	Main	Missing ownership checks	InitUserDetails	Always check the owner field of accounts that aren't supposed to be fully user-controlled. Ideally, you'd create a helper function that takes an untrusted account, checks the owner and returns an object of a different, trusted type. Your contract should only trust accounts owned by itself. Since the smart contract does not check that config is owned by the correct entity, an attacker can supply a maliciously crafted config account with an arbitrary admin field. Now if the smart contract tries to verify that the given admin account is indeed the admin account stored in its config account, it will be fooled by the malicious config. The contract will then happily withdraw funds to the attacker-controlled admin account.

Medium issues

Issue	File	Type	Line/Category	Description
#1	Main	Solana account confusions	• Claimed • MerkleRootUpdated • TokensWithdrawn • Config • UserDetails • Vesting • Period • UpdateRootArgs • MerkleDistributor • UpdateScheduleArgs • Change • TokenTransfer • ClaimArgs	Always keep in mind that a user can supply arbitrary accounts as inputs. Even if an account is owned by the contract, you have to ensure that the account data has the type you expect it to have. When you create a new account, you could set the “TYPE” field to a value that is unique to accounts of that type. Your deserialisation function will also have to validate the “TYPE” and error out if the account does not have the type you’re expecting

Low issues

Issue	File	Type	Line/ Category	Description
#1	Main	Arithmetic overflow/ underflow	398, 410, 413, 781, 271, 272, 356	It’s a common error to think that Rust catches overflows, when in fact this is only true in debug mode. Rust integers have fixed sizes and can only represent values within their supported ranges. If an arithmetic operation results in a higher or lower value, the value will wrap around with two’s complement. Citing from the Rust documentation (https://doc.rust-lang.org/book/ch03-02-data-types.html#integer-overflow). When you’re compiling in release mode with the —release flag, Rust does not include checks for integer overflow that cause panics. Instead, if overflow occurs, Rust performs two’s complement wrapping. In short, values greater than the maximum value the type can hold “wrap around” to the minimum of the values the type can hold. In the case of a u8, 256 becomes 0, 257 becomes 1, and so on. The program won’t panic but the variable will have a value that probably isn’t what you were expecting to have. We suggest that you use some safe math function supported by Rust, ex: checked_add, checked_sub instead of using normal operation #, -.

#2	Main	Warning code	166, 256, 338	There were some warnings in using Rust code. We suggest • user_details instead of &user_details • admin_slot.is_none() instead of None=admin_slot • !self.schedule.is_empty() instead of self.schedule.len()>0

Informational issues

Issue	File	Type	Line/ Category	Description
#1	Main	Loss of precisions in calculation	414, 425	Keep in mind that there should be loss of precision in integer division calculation. Be careful in the calculation of integer division for user reward

Audit Comments

August 2022:

Contract was compiled on Ubuntu 18.04 x64 with actual Rust, Solana, NPM and Yarn packages.

Automated testing results:

✓ Compiler Optimization Passes
✓ Pointer Analysis
✓ Building Static Happens-Before Graph
✓ Detecting Vulnerabilities

No Vulnerabilities were found in the crate dependencies.
Not lint mistakes were found against 450 lint rules.

Comment

Always check the owner field of accounts that aren't supposed to be fully user-controlled. Ideally, you'd create a helper function that takes an untrusted account, checks the owner and returns an object of a different, trusted type. Your contract should only trust accounts owned by itself.

Always keep in mind that a user can supply arbitrary accounts as inputs. Even if an account is owned by the contract, you have to ensure that the account data has the type you expect it to have.

Keep in mind that swap_rate has limitations and loss of precision.

Recommendation

Since the smart contract does not check that config is owned by the correct entity, an attacker can supply a maliciously crafted config account with an arbitrary admin field. Now if the smart contract tries to verify that the given admin account is indeed the admin account stored in its config account, it will be fooled by the malicious config. The contract will then happily withdraw funds to the attacker-controlled admin account.

When you create a new account, you could set the TYPE field to a value that is unique to accounts of that type. Your deserialization function will also have to validate the TYPE and error out if the account does not have the type you're expecting.

When accounts call set_swap_rate, you would check its validation.

No unit tests were performed because no corresponding tests were supplied.

Solid Proofed

Blockchain Security | Smart Contract Audits | KYC

MADE IN GERMANY