KUDELSKI SECURITY

Code and implementation Security Assessment

Findings and Recommendations Report Presented to:

Solana Foundation

July 07, 2021
Version: 1.0

Presented by:

Kudelski Security, Inc.
5090 North 40th Street, Suite 450
Phoenix, Arizona 85018

TABLE OF CONTENTS

TABLE OF CONTENTS 2
LIST OF FIGURES 4
LIST OF TABLES 4
EXECUTIVE SUMMARY 5
Overview 5
Key Findings 5
Scope and Rules Of Engagement 6
TECHNICAL ANALYSIS & FINDINGS 8
Threat Model 9
Background 9
Formal Verification 9
Decentralized Application ASSETS 9
Common attacks on Decentralized Applications 10
Attack trees 12
Specific known risks common to stake pools 13
Formal verification 14
Attack scenarios 14
Contract attacks 14
Timestamp Dependence 14
Integer Overflow and Underflow 14
DoS scenarios 14
Mint/Burn, Multi-party game theory attacks 14
Forcibly Sending Crypto Tokens (SOL, Ether, …) to a Contract 14
Infrastructure based attacks 15
Economic Attacks against the protocol (whale, scarcity, …) 15
Findings 15
1 – Update Stake Pool Balance does not check reserve account 16
2 – Vulnerable outdated version of the package "generic-array" found as dependancy 18
3 – Documentation for UpdateStakePoolBalance does not match implementation 19
4 – Ownership checks done implicitly 21
5 – Copy-paste code from solana-stake-program 22
METHODOLOGY 23
Overall methodology 23
Goals of the code review 23
Process and timeline of a code review assignment 23

Code review assignment process 24
Preparation 24
Whitepaper/documentations review 24
Code preparation 24
Threat modeling 25
Threat model workshop 25
Identify protected assets 26
A definition of threats and threat actors 26
Definition of implemented security controls 27
Source Code audit 27
Static code analysis 28
Call tree analysis 28
Verification of business logic 28
Input and return validation checks 28
Identification of critical external functions 28
Identification of complex parts and function 29
Review of cryptographic functions 29
Peer review 29
Classification of identified problems and vulnerabilities 30
Tools 31
Vulnerability Scoring Systems 32
CVSS 33
CWE 33
SWC 33
RustSec.org 33
KUDELSKI SECURITY CONTACTS 34

LIST OF FIGURES

Figure 1: Solana Stake Pool definitions....6
Figure 2: Solana Stake Pool deposit porcess....7
Figure 3: Solana Stake Pool Withdraw process....7
Figure 4: Findings by Severity....8
Figure 5: Solana attack tree....12
Figure 6: Direct & Indirect attacks....13

LIST OF TABLES

Table 1: Scope....6
Table 2: Findings Overview....15

EXECUTIVE SUMMARY

Overview

Solana Foundation engaged Kudelski Security to perform a Code and implementation Security Assessment.

The assessment was conducted remotely by the Kudelski Security Team. The review took place on April 24 – May 31, 2021, based on commit 3dd67672974f92d3b648bb50ee74f4747a5f8973, and focused on the following objectives:

- Provide the customer with an assessment of their overall security posture and any risks that were discovered within the environment during the engagement.
- To provide a professional opinion on the maturity, adequacy, and efficiency of the security measures that are in place.
- To identify potential issues and include improvement recommendations based on the result of our tests.
- Based on the functional discussions, include any ways of bypassing logical safeguards in the code and environment where the binary will execute.

This report summarizes the engagement, tests performed, and findings. It also contains detailed descriptions of the discovered vulnerabilities, steps the Kudelski Security Teams took to identify and validate each issue, as well as any applicable recommendations for remediation.

Key Findings

The following are the major themes and issues identified during the testing period. These, along with other items, within the findings section, should be prioritized for remediation to reduce to the risk they pose:

- Update Stake Pool Balance does not check reserve account – There are no safeguards to handle validation for the reserve accounts so this could post a threat due to the lack of validation.

During the test, the following positive observations were noted regarding the scope of the engagement:

- From a security standpoint the code quality is keeping high standards both in documentation as well as in implementation.
- As a CI/CD pipeline is implemented and made to good use this keeps the project in good shape as all tests and checks are automated and by that keeps the code quality high.

Scope and Rules Of Engagement

Kudelski performed an Code and implementation Security Assessment for Solana Foundation. The following table documents the targets in scope for the engagement. No additional systems or resources were in scope for this assessment.

In-Scope Applications	
Stake Pool	Implementation of the Solana Stake Pool protocol

Table 1: Scope

All the definitions used in the Stake Pool program and adjacent programs are defined as outlined in the following diagram. The only program that has been reviewed is the Stake Pool program and all other definitions are included to give a complete picture of the environment in which this program lives and executes.

Figure 1: Solana Stake Pool definitions

The process to deposit any funds into the stake pool are based on the following process including adjacent programs for reference.

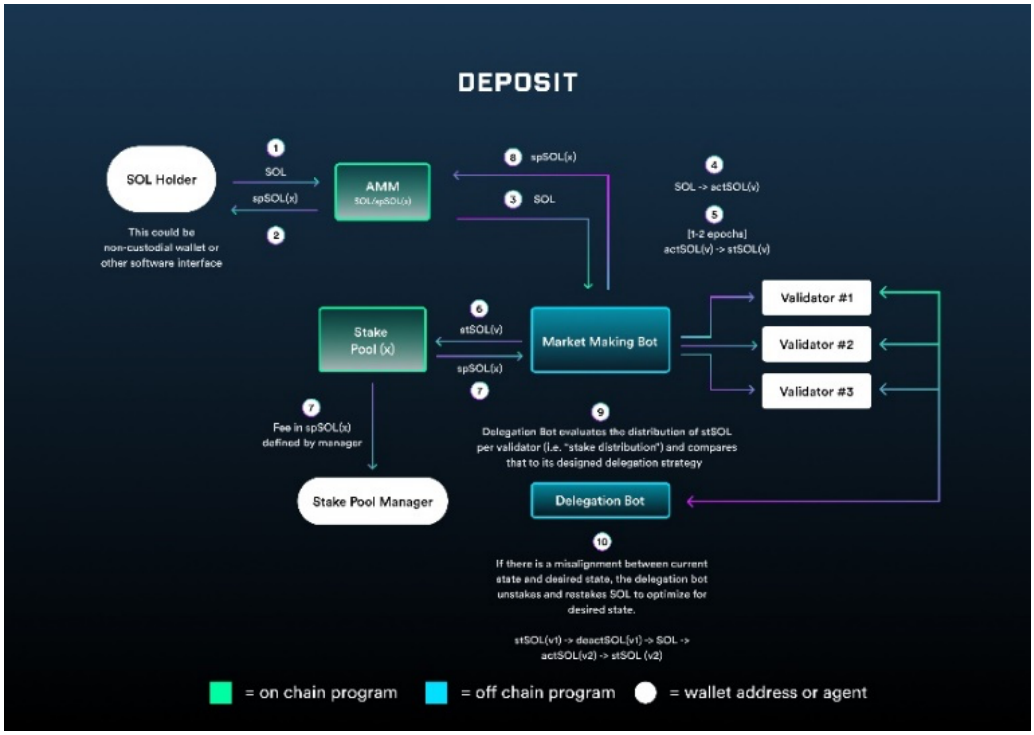


Figure 2: Solana Stake Pool deposit porcess

The process to withdraw funds from the Staking Pool is based on the following process including the adjacent programs for reference:

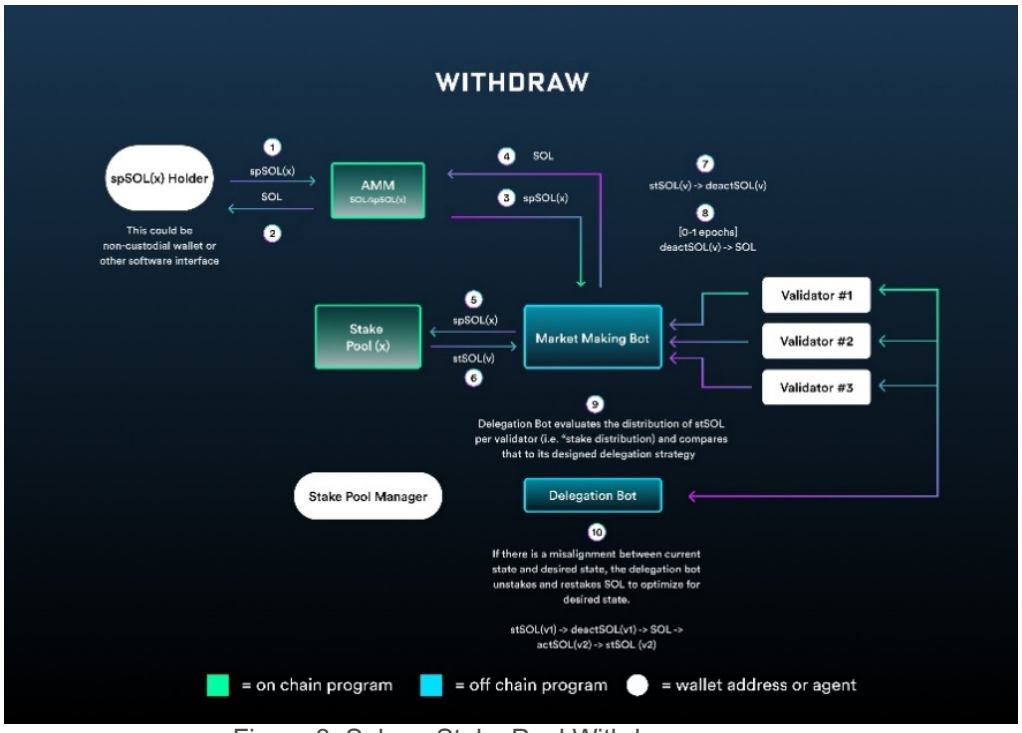


Figure 3: Solana Stake Pool Withdraw process

TECHNICAL ANALYSIS & FINDINGS

During the Code and implementation Security Assessment, we discovered 1 finding that had a MEDIUM severity rating, as well as 2 LOW.

The following chart displays the findings by severity:

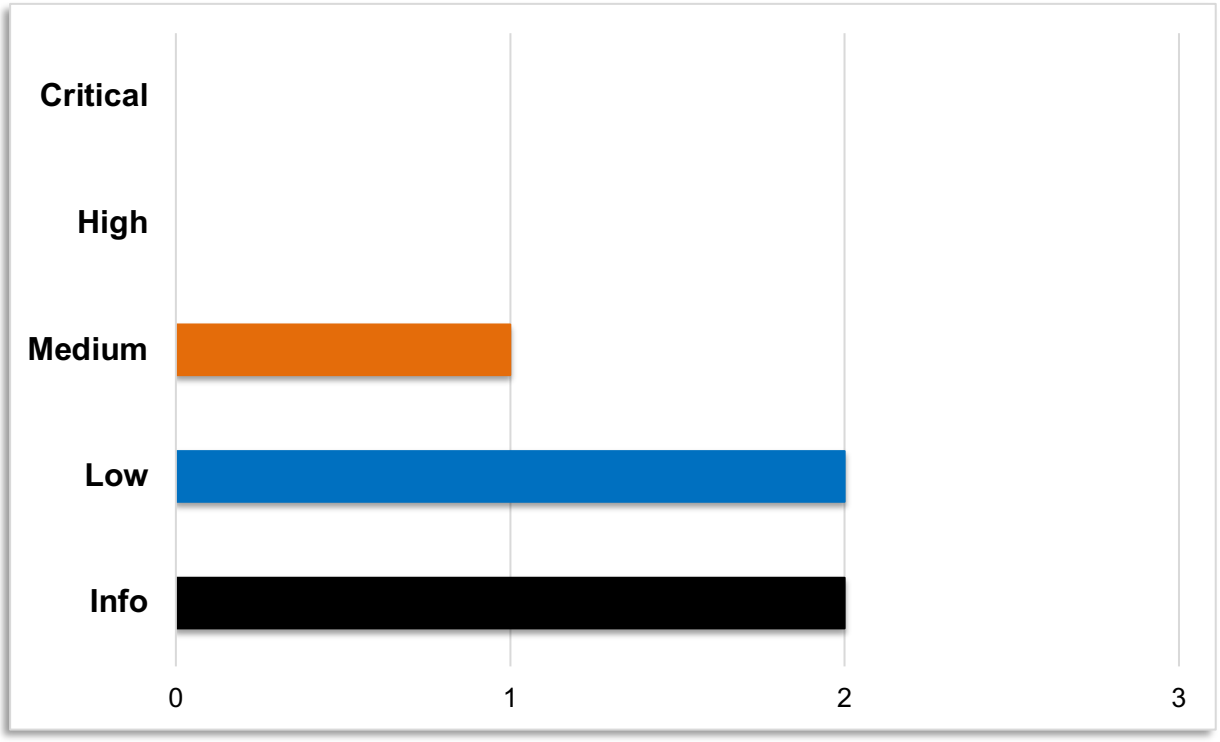


Figure 4: Findings by Severity

Threat Model

Background

Threat modeling is a process by which threats can be identified, enumerated, and mitigation and then can be prioritized.

The purpose of the threat model is to provide application defenders with a systematic analysis of what controls and defenses are included, given the nature of the system, the probable attacker's profile, the most likely attack vectors, and the assets most desired by an attacker.

The model also can act as a guide to the code audit part of the assignment.

Decentralized Application development requires a different engineering mindset than we may be used to as the cost of failure can be high, and change can be difficult, making it in some ways more similar to hardware programming or financial services programming than web or mobile development.

Formal Verification

Besides threat modeling, formal verification has been performed as part of this assessment. Formal verification is the act of proving or disproving the correctness of intended functionality and algorithms underlying a system concerning a specific formal specification or property, using formal methods of mathematics. This task's goal is to verify that the code does what it was supposed to do.

Decentralized Application ASSETS

Analyzing the information from the Decentralized Application ecosystem will define what assets are in the context/scope of these Decentralized Applications.

Defined Assets

- The Decentralized Application function we are required to protect
- The Decentralized Application data we are required to protect
- The Coins and tokens handled and transferred by the smart program

Threat actors

A threat actor or malicious actor is a person or entity responsible for an event or incident that impacts Decentralized Application applications' safety or security. In this context, the term describes individuals and groups that are the instigator of malicious acts or threats.

Traditionally, we can broadly divide the general threat actors into the following categories.

- Nation-State Actor with unlimited resources
- Organized Crime Actor with significant resources
- Insiders Actor with extensive knowledge of the system
- Hacktivists Actor with ideology as a motive and limited resources
- Others Actors, such as natural and accidental attacks

Decentralized Application threat actors
This section describes the specifics of threat actors related to the Decentralized Application.

The actors that are related to Decentralized Applications can broadly be classified into two types: Malicious Decentralized Applications (other programs on the chain) & Humans who are interacting with Decentralized applications directly or via API.

Decentralized Applications can call functions of other Applications and are even able to create and deploy other applications and programs (e.g. issuing coins and tokens). There are several use-cases for this behavior.

The identified actors for this application are

- Contract owner
- Stake owner
- Staker
- External Program on chain

Solanas' ability to create and deploy contracts to the chain gives many opportunities for the developer but it also makes it very hard to verify. We had to conduct a very thorough analysis to include as many scenarios as possible.

CROSS PROGRAM INTERACTION
A few use cases of interacting with other contracts are described below.

1. Use contracts as data stores
2. Use other contracts as libraries.

In the context of this report, an Actor interacting with a Decentralized Application outside the blockchain is an external Actor, and an Actor interacting with a Decentralized Application inside the blockchain is an internal Actor.

Common attacks on Decentralized Applications

The following is a list of known attacks which we should be aware of and defend against when writing Decentralized Applications.

- Reentrancy
- Reentrancy on a Single Function
- Cross-function Reentrancy
- Timestamp Dependence
- Integer Overflow and Underflow
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Limit DoS on the Network via Block Stuffing
- Insufficient gas griefing
- Mint/Burn, Multi-party game theory attacks
- Forcibly Sending Crypto Tokens (SOL, Ether, …) to a Contract
- Attacks against Full Nodes
- Attacks against Staking / Validator Nodes

- Attacks against the Internet, Hardware, Geographies, Participating Governments
- Economic Attacks against the protocol (whale, scarcity, …)

Reentrancy
One of the major dangers of calling external contracts is that they can take over the control flow and make changes to your data that the calling function wasn't expecting.

This is the most common problem that we see in Decentralized applications.

Reentrancy on a Single Function
The first version of this bug to be noticed involved functions that could be called repeatedly before the first invocation of the function was finished. This may cause the different invocations of the function to interact in destructive ways.

Cross-function Reentrancy
An attacker may also be able to do a similar attack using two different functions that share the same state.

Cross program Reentrancy
Reentrancy can also occur if calling the between programs is not explicitly checked.

Mint/Burn, Multi-party game theory attacks
When multiple programs interact inside and outside of a system, one calculation can be used to manipulate another calculation, especially as relates to token creation, external oracles, cross-swaps

Forcibly Sending Crypto Tokens (SOL, Ether, …) to a Contract
An attacker may be able to force a contract to execute instructions, not by first gaining access/entitlements, but by simply sending the instructions and overriding the permissions

Attacks against Full Nodes
An attacker may be able to manipulate a full node to perform inappropriate operations that are not stopped by the rest of the nodes on the network

Attacks against Staking / Validator Nodes
An attacker may be able to manipulate a single or set of validator/staking nodes to use that majority/minority stake to change the behavior of the network, especially in DAO, vote driven networks

Attacks against the Internet, Hardware, Geographies, Participating Governments
A nation-state level attacker may be able to conduct large scale attacks on a significant majority of a certain internet segment through routing protocols, DNS attacks, outages, or other large scale interruption

Economic Attacks against the protocol (whale, scarcity, …)
An attacker may be able to influence the pricing, token scarcity, availability, or speed in which the network operates simply by having a large portion of the tokens in circulation or through very quick implementation of behaviours which drastically change the operating parameters

Attack trees

Attack Trees provide a formal, systematic way of describing the security of systems based on varying attacks. A tree structure represents attacks against a system, with the goal as the root node and different ways of achieving that goal as leaf nodes.

The attack attributes to assist in associating risk with an attack. An Attack Tree can include special knowledge or equipment needed, the time required to complete a step, and the physical and legal risks assumed by the attacker. The values in the Attack Tree could also be operational or development expenses. An Attack Tree supports design and requirement decisions. If an attack costs the perpetrator more than the benefit, that attack will most likely not occur. However, if there are easy attacks that may result in benefits, then those need a defense.

Specific attack tree to Solana staking applications
Based on what we have seen in other assignments as well as known other attacks we can construct the following basic attack tree.

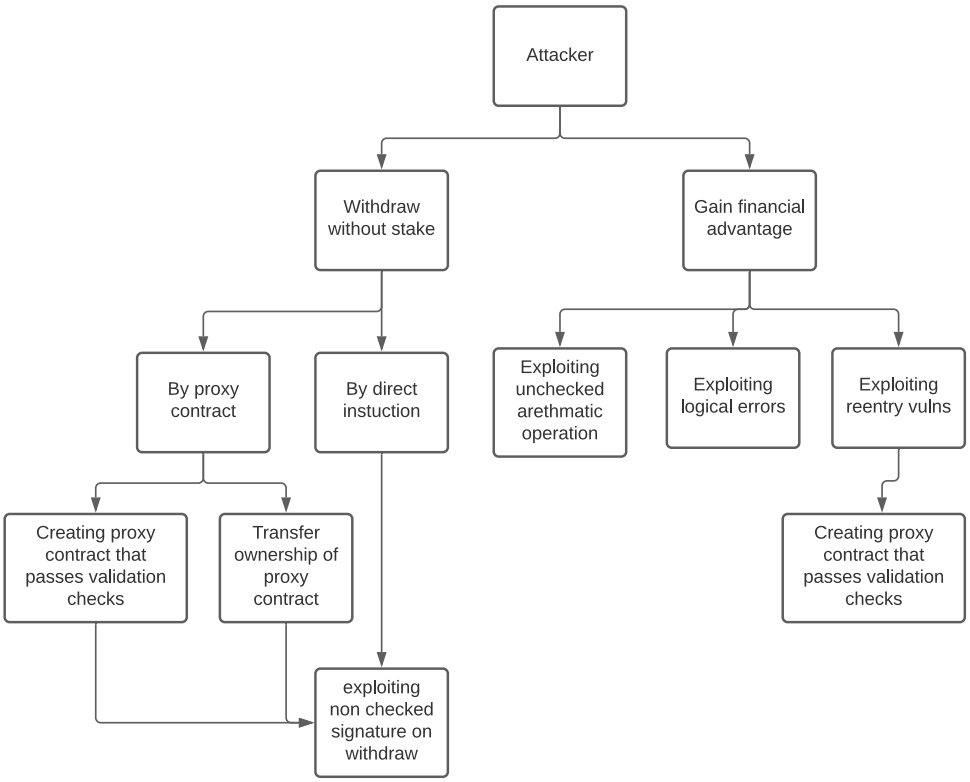


Figure 5: Solana attack tree

Indirect and direct types of attacks
In many cases, we are seeing that the attackers will have to create Proxy programs that will interact with the targeted program and can be created with data in such a manner that it will pass validation checks for certain functions. e.g. withdrawal without actually having any connection with the stakes assets. We have also seen certain attacks where an attacker can give away ownership of the proxy contract to the staking contract and thereby fool the program's owner checks.

Below we can see a simple abuse case diagram with an external actor:

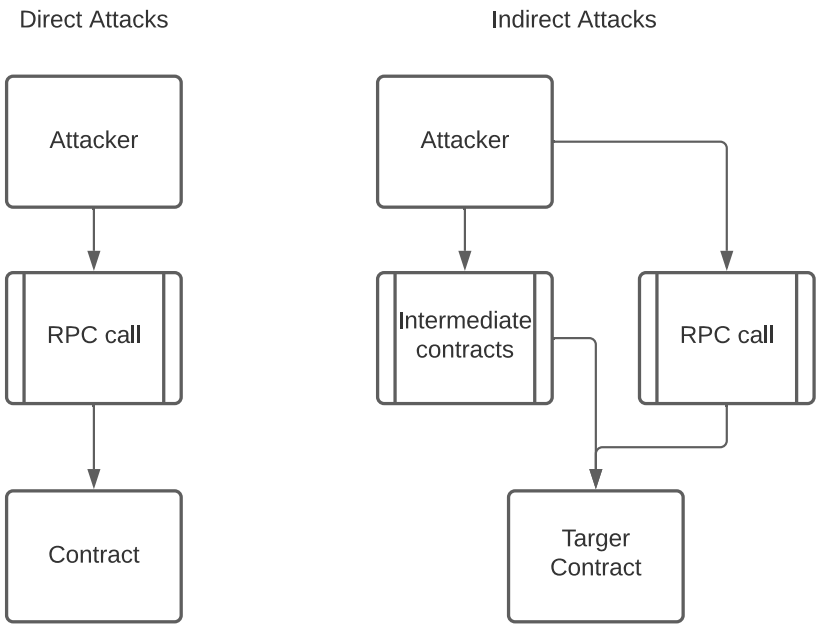


Figure 6: Direct & Indirect attacks

Specific known risks common to stake pools

Slashing while Staking
One risk that has been seen to affect stake pools in the Ethereum eco-system has been that the validator nodes has been slashed while staking. This is not as an immediate threat to the Solana eco-system as the validator node will be slashed as a manual process and not automatically.

This risk may still be present as the Stake Pool will be automated from a user perspective as seen in the architectual diagrams presented in this report.

Formal verification

At the time of the assessment the code provided did pass the formal verification. The code is providing the functionality that is described in the documentation. The code review concurs with that conclusion.

Attack scenarios

As the applications that the Stake Pool application relies on to function haven't been finalized at the time of the assessment the only way to test these scenarios was on the Stake Pool functionality itself. These are the attack scenarios we have looked at and tested when applicable.

Contract attacks

These kinds of attacks like Reentrancy, Reentrancy on a Single Function or Cross-function Reentrancy are not something that we can se would happen in the Stake Pool application as the functionality is not implemented through smart contracts in the ordinary sense.

Timestamp Dependence

As the Solana Blockchain is very time dependent to keep the epochs and PoH intact timestamp dependencies are a viable attack vector. In this case the only way we can see this happening is that the node running the validator would be subjected to an internal attack where the time on the machine would be altered and therefore be able to manipulate the execution. As the Stake Pool program only concerns itself with the Stake Pool operation, the AMM and the Market Making Bot, which lies in front of the Stake Pool program, needs to handle such cases as they will be present in the Solana eco-system. Based on the above this attack is an infrastructure attack and thus out of scope for this assessment.

Integer Overflow and Underflow

The code has been checked during our review for overflow/underflow attacks and we have not been able to find any at the time of the assessment. As the Stake Pool program only concerns itself with the Stake Pool operation, the AMM and the Market Making Bot, which lies in front of the Stake Pool program, needs to handle such cases as they will be present in the Solana eco-system.

DoS scenarios

These kinds of attacks like DoS with (Unexpected) revert, DoS with Block Gas Limit, Gas Limit DoS on the Network via Block Stuffing, and Insufficient gas griefing are common in situations where there is a way to create a DoS scenario through using scarce resources in the execution of smart contracts. In the case of the Solana Blockchain these scenarios are not valid in the case of the Stake Pool application as there are two layers above the Stake Pool program when interacting with SOL holders.

Mint/Burn, Multi-party game theory attacks

These kinds of attacks is not something that the Stake Pool application needs to handle as it only concerns itself with the Stake Pool operation. The AMM and the Market Making Bot, which lies in front of the Stake Pool program, needs to handle such cases as they will be present in the Solana eco-system by implementing safeguards for this kind of attacks.

Forcibly Sending Crypto Tokens (SOL, Ether, …) to a Contract

These kinds of attacks, which has been seen on the Ethereum chain targets the contracts balance and could therefore trigger unforeseen effect on a Smart Contract if depends on logic that controls the balance. As the Stake Pool program only concerns itself with the Stake Pool operation, the AMM and the

Market Making Bot, which lies in front of the Stake Pool program, needs to handle such cases as they will be present in the Solana eco-system.

Infrastructure based attacks

These kinds of attacks like “Attacks against Full Nodes”, “Attacks against Staking / Validator Nodes”, and “Attacks against the Internet, Hardware, Geographies, Participating Governments” are all based on the infrastructure where the nodes are executing the program. This is out of scope for this assessment. From a holistic view on the Solana Blockchain, a very detailed guide to help the operators of the nodes to not subject themselves to any of these attack vectors would be valuable.

Economic Attacks against the protocol (whale, scarcity, …)

As the Stake Pool program only concerns itself with the Stake Pool operation the AMM and the Market Making Bot, which lies in front of the Stake Pool program, needs to handle such cases as they will be present in the Solana eco-system, before they reach the Stake Pool program.

Findings

The *Findings* section provides detailed information on each of the findings, including methods of discovery, explanation of severity determination, recommendations, and applicable references.

The following table provides an overview of the findings:

#	Severity	Description	Affected Resources
1	Medium	Update Stake Pool Balance does not check reserve account	**1**
2	Low	Vulnerable outdated version of the package "generic-array" found as dependancy	**1**
3	Low	Documentation for UpdateStakePoolBalance does not match implementation	**1**

Table 2: Findings Overview

1 – Update Stake Pool Balance does not check reserve account

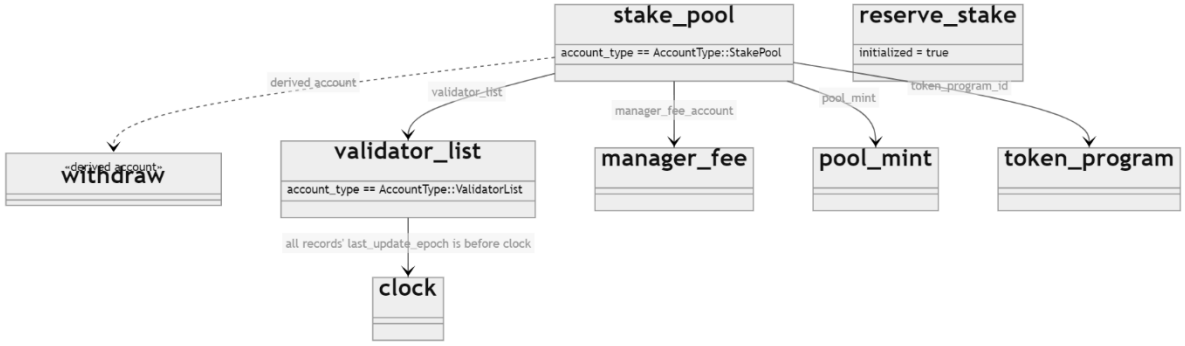
Severity	MEDIUM
Score	4.8 (Vector String - CVSS:3.1/AV:N/AC:H/PR:H/UI:R/S:U/C:N/I:L/A:H)
SWC	SWC-122: Lack of Proper Signature Verification
CWE	CWE-345: Insufficient Verification of Data Authenticity

Description

The `reserve_stake` account is only verified for:

- being a stake account
- holding the minimum amount of lamports

However, it is not checked if the `reserve_stake` account is actually the `stake_pool` account's associated `reserve_stake`.



Impact

If this is exploited this may allow injection of other `reserve_stake` accounts that will pass validation even though the `stake_pool` account's actual reserve is insufficient.

By exploiting this fee can be minted to the `manager_fee` account on invalid terms.

Steps to Reproduce

The following code show the missing check in context.

```
let reserve_stake = try_from_slice_unchecked::<stake_program::StakeState>(
    &reserve_stake_info.data.borrow(),
)?;
let mut total_stake_lamports =
    if let stake_program::StakeState::Initialized(meta) = reserve_stake {
        reserve_stake_info
            .lamports()
            .checked_sub(minimum_reserve_lamports(&meta))
            .ok_or(StakePoolError::CalculationFailure)?
    } else {
        msg!("Reserve stake account in unknown state, aborting");
        return Err(StakePoolError::WrongStakeState.into());
    };
```

Affected Resource

- processor.rs beginning line 1387

Recommendation

Creating a safeguard against this vulnerability there will need to be a check for

`stake_pool.reserve_stake == reserve_stake_info.key`

This would create a check that would match only if the key matches and therefore stop the exploit.

Reference

- N/A

2 – Vulnerable outdated version of the package "generic-array" found as dependancy

Severity	LOW
Score	3.5 (Vector String - CVSS:3.1/AV:N/AC:L/PR:H/UI:R/S:U/C:L/I:L/A:N)
SWC	N/A
CWE	CWE-928: Using Components with Known Vulnerabilities

Description
When the static code analysis was conducted, a potential vulnerability was identified by the RustSec tool. The use of a crate that is known to be vulnerable was found. An outdated version of the package "generic-array" was found in the dependencies and was flagged as a potential attack vector.

Impact
This may result in a variety of memory corruption scenarios, most likely use-after-free. Affected versions of this crate allowed unsoundly extending lifetimes using arr! macro. The direct risk of this vulnerability is reduced to low since the auditor did not see any directly exploitable links to the vulnerability leading to damage for the protected assets.

Steps to Reproduce

By using the “cargo audit” command to check vulnerable dependencies the following results is recorded

```
Crate:         generic-array
Version:       0.12.3
Title:         arr! macro erases lifetimes
Date:          2020-04-09
ID:            RUSTSEC-2020-0146
URL:           https://rustsec.org/advisories/RUSTSEC-2020-0146
Solution:      Upgrade to >=0.8.4, <0.9.0 OR >=0.9.1, <0.10.0 OR >=0.10.1, <0.11.0 OR >=0.11.2, <0.12.0 OR >=0.12.4, <0.13.0 OR >=0.
Dependency tree:
generic-array 0.12.3
```

Affected Resource

At the time of the audit this affected the whole of the project as the dependency is project wide.

Recommendation

Upgrade the dependencies to use the latest version of the package will fix the problem. As the project has a CI/CD pipeline, we would also recommend that an additional step in the pipeline is added so that a problem like this is caught early in the process. By adding a step that breaks the build based on “cargo audit” would be a simple safeguard against using outdated and vulnerable components.

Reference

- N/A

3 – Documentation for UpdateStakePoolBalance does not match implementation

Severity	LOW
Score	N/A
SWC	N/A
CWE	CWE-684: Incorrect Provision of Specified Functionality

Description

At the time for the audit the documentation in the project for the `UpdateStakePoolBalance` enum does not match the implementation.

Impact

At the time of the audit the implementation wrote data to the `validator_list_info` account which is the "validator stake list storage account". As the `validator_list` is serialized in the mutable `validator_list_info.data` the "Validator stake list storage account" should be marked as "writable". This leads to confusion about how to use the implementation correctly and could lead to future introduction of code that does break the security in the system. An implementation based on the documentation found at the time of the audit, will not be correct when it comes to the actual functionality. This will make the maintenance and use of the implemented code harder than it needs to be.

Steps to Reproduce

N/A

Affected Resource

The affected lines of code and documentation is seen here.

- instruction.rs line 183

```
    ///   Updates total pool balance based on balances in the reserve and validator list
    ///
    ///   0. `[w]` Stake pool
    ///   1. `[]` Stake pool withdraw authority
    ///   2. `[]` Validator stake list storage account
    ///   3. `[]` Reserve stake account
    ///   4. `[w]` Account to receive pool fee tokens
    ///   5. `[w]` Pool mint account
    ///   6. `[]` Sysvar clock account
    ///   7. `[]` Pool token program
    UpdateStakePoolBalance,
```

- processor.rs line 1434

```
        validator_list.serialize(&mut *validator_list_info.data.borrow_mut())?;
```

Recommendation

The recommendation is to rectify the documentation to reflect the correct use of the code to reduce the risk of any intentional or unintentional introduction of erroneous code.

Reference

- N/A

4 – Ownership checks done implicitly

Severity	INFORMATIONAL
Score	N/A
SWC	N/A
CWE	N/A

Description

At the time of the audit the code did follow a model of checking for ownership and/or writeability implicitly somewhere else in the ecosystem code base.

Impact

To implicitly validate ownership makes any attempt to assess the security implications of a specific check or omission to check harder and introduces real risk of missing that any security checks follows the strict rules in the Solana code eosystem.

Steps to Reproduce

At the time of the audit you could reproduce the issue. In the process_update_stake_pool_balance function in process.rs:1348 the `stake_pool` account is updated. This requires ownership by the program but it is not checked explicitly. The check is done implicitly when the `stake_pool` struct is serialized back into the account data.

Affected Resource

The affected lines of code and documentation is seen here.

- processor.rs line 1437

```
stake_pool.serialize(&mut *stake_pool_info.data.borrow_mut())?;
```

Recommendation

We recommend that to make the validation more clear and to make the code easier to review, all checks for ownership/writability should be done explicitly where the validation needs to happen. This would also create a good culture of explicitly check for permissions to show that you are aware of the risks in omitting the checks. By doing this you would avoid mistakes, intentional or unintentional, that could lead to the introduction of a vulnerability.

Reference

- N/A

5 – Copy-paste code from solana-stake-program

Severity	INFORMATIONAL
Score	N/A
SWC	N/A
CWE	N/A

Description

At the time of the review, the `stake_program.rs` contains 32 comments about code copied directly from the "solana-stake-program" crate.

Impact

We can't see any current problems in the code based on this but as there is many copies of code introduced into a different code base. Copy-paste is not seen as a good practice as it may propagate erroneous code into other part of a project. When the original code is updated the code that has been copied could be missed to be updated too. In this case the code is enum and struct definitions used to interact with the solana-stake-program.

Steps to Reproduce

N/A

Affected Resource

An example of the affected lines of code and documentation is seen here.

- `stake_program.rs` on 32 places

```
/// FIXME copied from Solana stake program
```

Recommendation

We recommend that the copied code should be integrated much better and reviewed on the merit of where they are placed in the new codebase and not from where they come. In this case, as the copied code is definitions from a dependent code base, the solution would be to have common definitions that can be referenced from the new and the original code making any changes covering all usage in the project.

Reference

- N/A

METHODOLOGY

OVERALL METHODOLOGY

This document describes the overall methodology for conducting reviews of code is based on a threat model approach to code review.

This document is created to streamline the process of the code reviews performed and lift the quality of work delivered during the review.

It is important that the code review team has an in-depth understanding of the solution's problems and what problems the code is trying to address.

Goals of the code review

The goals of the code review performed is to:

1. Validate technical design claims and cryptographic coding underlying the behavior and intent of the technical systems or solution
2. Verify through threat modeling and code verification that the solution adequately protects the assets it is intended to protect against any identifiable threats
3. Verify via **manual** code-review that the claimed implemented security controls are sound, especially focusing on code written by the client's team, assuming third-party libraries can be liable and should be investigated as and when expected to be critical
4. Validate implementation choices, completeness, and assumptions according to the design provisions and deployment
5. Provide recommendations for security-related improvements and corrections to the code, infrastructure, and architecture, if found
6. Provide general recommendations for architectural and implementation related improvements to the infrastructure and architecture, if found

Process and timeline of a code review assignment

The following section describes all the steps that should be followed during the code review process

- Preparation and research
  - White-paper / documentation review
  - Make the code available to internal team / peer reviewers
  - Start of shared documentation document as a readme.md file under /review

- Threat Modelling
  - identification of all inputs and outputs
  - identification of security boundaries
  - Identification of relevant threat scenarios

  - Identification of implemented security controls

- Code audit
  - Set up of relevant tooling
  - Identification of process flow
  - Identification of critical paths
  - identification of critical function
  - Logical code analysis of critical functions
  - Static code analysis
  - Crypto implementation analysis

- Peer review
- Preliminary report
- Final report

CODE REVIEW ASSIGNMENT PROCESS

This section covers in-depth the parts of a code review process that is expected by the review teams to perform in each assignment.

Preparation

Background and understanding of the underlying protocols and technologies are key to be able to deliver a relevant code audit. Without fully understanding the larger solution and the part of the code in the larger ecosystem, it is impossible to perform a complete analysis and code review effectively.

During the preparation phase of the code review, the team performing the code review shall read up on the technology stack used and the core concepts of the technology and make sure they have a good understanding of the technology to review.

The results from the preparation of the phase of the review are used as inputs into the threat modeling phase.

Whitepaper/documentations review

1. During the documents review phase, the auditor should make a list of any relevant concepts in the audited solution that is specific to the solution's security (E.g staking, zero-knowledge proofs, specific hashing algorithm)
2. Make an overall **process description and model** of how the software is intended to function
3. Identify potential security boundaries in the audited solution
4. Start a shared document with these findings

Code preparation

During this part of the assignment, the team verifies that the code to be audited during the assignment is the right version and the team leader makes sure that all relevant people have access to the code via a shared repository.

Make the code available to internal team / peer reviewers

- Start a new git repository under the internal repository
- Invite relevant team members to the repository
- Clone the repository as assigned by the client into the new repository
- Make sure that the code builds and that it is possible to run any unit tests
- Update the shared document with links to the repository and which parts of the code are relevant to which features identified in the documentation preparations phase

Threat modeling

On all projects, regardless of how small, a threat model should be defined. Without a well-defined threat model, the rest of the code audit will not have the right focus

The threat model should contain at least the following

- A definition of protected assets
- identification of security boundaries
- Implemented security designs to protect the assets
- A list of potential threat scenarios to each asset

During the initial phase of the analysis, a threat-modeling workshop shall be conducted to assess if there were any assumptions regarding the application that needed to be addressed during the review. As input for the workshop, the project Documentation, as made available, shall be used as background material.

Threat model workshop

The threat model workshop is essential to successfully being able to validate and audit the solution from a security perspective.

A threat model workshop ideally involves the client since it is the client that sits on the knowledge about what to protect. If that is not possible then the threat model should be executed as a team exercise

The workshop shall be concluded using the STRIDE[2] methodology looking at the following attributes of the security impacts.

This is called the stride model and can be used to look a the product audited from different viewpoints and helps in designing the threat model.

Spoofing	Authenticity
Tampering / txs	Integrity

Repudiation/ signing	Non-repudiability
Information disclosure / Key protection	Confidentiality
Denial of Service	Availability
Elevation of Privilege / Spending of funds	Authorization

Identify protected assets

This definition is a summary of the assets that are protected by the security solution. This set differs from the project but should always be defined as WHAT.

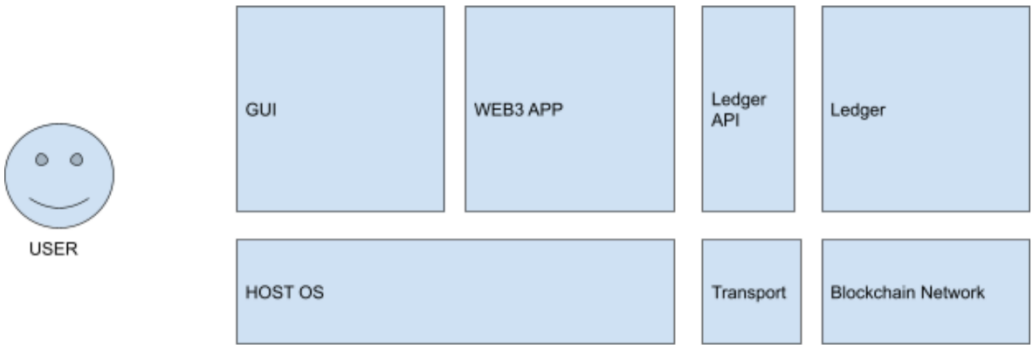
What are we trying to protect? In most blockchain or crypto projects this is gonna be fairly standard. e.g.

Key material

- Private key
- Transactions
- Coins
- Tokens

Identified inputs and outputs and security boundaries.

Looking at the system and solution's code and documentation and defining all inputs and outputs to the system and which interfaces are used by the inputs and outputs to the system. This deliverable should result in a block diagram that defines the part of the application and what data travels between the components in the application e.g



A definition of threats and threat actors

What are we trying to protect is a good start but we also need to look at against whom are we protecting these assets. Here the auditor should create a list of potential threat actors and what threats they pose to the protected assets.

When formulating threat scenarios it is important to think in the following terms. Ask yourself these three questions.

Who is doing **What** to get **What**

Based on this we can drive the threat description. Think of a threat description as a user story for development but rather for threats.

Example:

> **As a user with local system access, I dump the memory of the application when running and extracts the unencrypted private key, and uses this to create a new wallet and transfers all the funds from the target.**

Definition of implemented security controls

This deliverable should include all implemented security controls that map against the protected asset and how they map to the threat scenarios. This information should be able to get from the documentation and whitepapers but sometimes you will need to go through the actual code to see what controls are implemented.

Examples could include:

- encryption of private keys in memory or storage.
- Keys are always referenced and never copied
- All transaction signatures are verified before they are executed
- Double spending is checked by Proof of history or other proofs

Source Code audit

The source code audit part of the project is used to verify and validate the security controls implemented to protect the assets identified during the threat model.

It is not possible to exhaustively and thoroughly go through all the code of an application in the allotted assigned time it is therefore very important that the audit team spends the time on the critical aspects of the code and quickly can discard functions and support code that is not relevant to the protected assets identified in the threat model.

The parts of the source code audit to be performed are

- Static code analysis
- Dynamic code analysis and a call tree analysis
- Follow user input flow and information flow of protected assets
- Manual code review of identified critical functions
- Peer review of findings and critical functions

Static code analysis

During this phase of the audit, the auditor sets up tools as toolchains for performing static code analysis.

- Chose tool for static analysis
- Install and run tools against the codebase
- Validate the findings and cross-reference the validity against the threat model
- Enter finding with validity to the threat model in the repo as issues.

This part of the source code review is important to quickly find tricky and complex parts of the application. The focus here should again be on the code supporting the protected assets. If we find code that is bad and hard to understand e.g. “Warning overly complex function” this should also be recorded

Call tree analysis

During this step of the code review, the reviewer is expected to run applicable tools for dynamic code analysis and this step should generate a call tree diagram of the application. Remember that we are only interested in the core parts of the application and to be able to follow user input and get a better understanding of which functions are critical to the protected assets.

- Chose and install a tool for dynamic code analysis or
- Run the tool and visualize the call graph
- Identify critical function to the protected assets through the call graph or manual process
- Mark these as critical functions in the git repository.

Verification of business logic

In this part, the auditor goes through the stipulations in the documentation of the project including whitepapers, and verifies that the security claims that are made in the documentation are implemented in the code. The auditor also verifies that the code does what it says it does in the documentation

We recommend that this is documented in a readme.md file that is added to the repository with links to which functions handle which security requirements. This will make the peer review much easier

Here we look to verify that the code does what the documentation or project description says any dieviences from the security claims should be reported

Input and return validation checks

Following the call tree and execution path of the protected assets in the code and verify that all inputs and outputs and or returns are

- Ensure proper input validation
- What would happen if an external function returns garbage or null

Identification of critical external functions

In this part, the auditor looks at the execution flow and tries to identify function calls to external libraries and external code where information or protected assets are passed.

This could include

- 3rd party libraries
- Blockchain ledger API
- Virtual Machine functions
- OS cryptography functions

Although these functions are in general excluded from the scope in the statement of work it is up to the auditor to make sure that they are secure and do what they say they do.

Example:

Identification of complex parts and function

> Here the auditor looks for and marks functions too complex
> "Too complex" usually means "can't be understood quickly by code readers." It can also mean "developers are likely to introduce bugs when they try to call or modify this code."

Review of cryptographic functions

All of the functions used for signing, verification, key creation, encryption, and decryption of data must be exhaustively reviewed.

Here the following things should be reported

- Insecure copy of keys
- Insecure random initialization
- Insecure storage of key material. (Clear text)
- Transfers or transactions without verification of signatures

Peer review

Before delivering the preliminary report to the client all of the following should have been peer-reviewed in the repository

- The threat model
- The security claims map document
- Any high or medium finding
- Any complex, critical internal and external functions

We recommend that all of this is documented inside the repo before being transferred to the report template.

- The peer reviewer needs to concur with the finder's assumption and threat level rating.
- Might request clarification and further documentation on exploitation

Classification of identified problems and vulnerabilities

We have four severity levels of a security vulnerability. And in the report, we ONLY want to report problems that affect the protection of that assets

- Critical – vulnerability that will lead to loss of protected assets
  - This is a vulnerability that would lead to immediate loss of protected assets
  - The complexity to exploit is low
  - The probablillty of exploit is high

- High - A vulnerability that can lead to loss of protected assets
  - All discrepancies found where there is a security claim made in the documentation that can not be found in the code
  - All mismatches from the stated and actual functionality
  - Unprotected key material
  - Weak encryption of keys
  - Badly generated key materials
  - Tx signatures not verified
  - Spending of funds through logic errors
  - Calculation errors overflows and underflows

- Medium - a vulnerability that hampers the uptime of the system or can lead to other problems
  - Insecure calls to third party libraries
  - Use of untested or nonstandard or non-peer-revied crypto functions
  - Program crashes leaves core dumps or write sensitive data to log files

- Low - Problems that have a security impact but does not directly impact the protected assets
  - Overly complex functions
  - Unchecked return values from 3rd party libraries that could alter the execution flow

- Informational
  - General recommendations

Tools

The following tools were used during this portion of the test. A link for more information about the tool is provided as well.

Tools used during the code review and assessment

- Rust – cargo tools
- IDE modules for Rust and analysis of source code
- Cargo audit which uses https://rustsec.org/advisories/ to find vulnerabilities cargo.

Vulnerability Scoring Systems

Kudelski Security utilizes three common vulnerability scoring systems to assign a risk severity to findings.

- Common Vulnerability Scoring System (CVSS)
- Common Weakness Enumeration (CWE)
- Smart Contract Weakness Classification and Test Cases Registry (SWC)
- RustSec.org

CVSS

The Common Vulnerability Scoring System (CVSS) is an open framework for communicating the characteristics and severity of software vulnerabilities. CVSS consists of three metric groups: Base, Temporal, and Environmental. The Base group represents the intrinsic qualities of a vulnerability that are constant over time and across user environments, the Temporal group reflects the characteristics of a vulnerability that change over time, and the Environmental group represents the characteristics of a vulnerability that are unique to a user's environment. The Base metrics produce a score ranging from 0 to 10, which can then be modified by scoring the Temporal and Environmental metrics. A CVSS score is also represented as a vector string, a compressed textual representation of the values used to derive the score. This document provides the official specification for CVSS version 3.1.

The most current CVSS resources can be found at https://www.first.org/cvss/

CWE

The CWE system is a community-developed list of common software security weaknesses. It serves as a common language, a measuring stick for software security tools, and as a baseline for weakness identification, mitigation, and prevention efforts. Some common types of software weaknesses classified by the CWE are:

- Buffer Overflows, Format Strings, etc.
- Structure and Validity Problems
- Common Special Element Manipulations
- Channel and Path Errors
- Handler Errors
- User Interface Errors
- Pathname Traversal and Equivalence Errors
- Authentication Errors
- Resource Management Errors
- Insufficient Verification of Data
- Code Evaluation and Injection
- Randomness and Predictability

SWC

This is the Smart Contract Weakness Classification and Test Cases registry

The registry contains tables with an overview of identified SWC Weaknesses. Each wekness consists of an SWC identifier (ID), weakness title, CWE parent and list of related code samples. There are links to the ID and Test Cases to the respective SWC definition. There are also links to the CWE Base or Class type.

RustSec.org

About RustSec
The RustSec Advisory Database is a repository of security advisories filed against Rust crates published and maintained by the Rust Secure Code Working Group.

The RustSec Tool-set used in projects and CI/CD pipelines

‘cargo-audit’ - audit Cargo.lock files for crates with security vulnerabilities.
‘cargo-deny’ - audit `Cargo.lock` files for crates with security vulnerabilities, limit the usage of particular dependencies, their licenses, sources to download from, detect multiple versions of same packages in the dependency tree and more.

KUDELSKI SECURITY CONTACTS

NAME	POSITION	CONTACT INFORMATION
Scott Carlson	**Head of Blockchain Security**	**Scottj.carlson@kudelskisecurity.com**