// HALBORN

Solana Labs - Runtime and BPF Loader Diff 124aaa95 -> 6fbe8ba

L1 Security Assessment

Prepared by: **Halborn**

Date of Engagement: **August 21st, 2023 - August 24th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	08/21/2023
0.2	Document Updates	08/22/2023
0.3	Final Draft	08/23/2023
0.4	Draft Review	08/24/2023
0.5	Draft Review	08/24/2023
1.0	Remediation Plan	09/25/2023
1.1	Remediation Plan Review	09/25/2023

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Piotr Cielas	Halborn	Piotr.Cielas@halborn.com
Isabel Burruezo	Halborn	Isabel.Burruezo@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Solana is an open-source project implementing a new, high-performance, permissionless blockchain. Changes in scope affected several modules, the most important ones are briefly described. Sealevel, Solana's parallel smart contracts runtime, is a concurrent transaction processor. Transactions specify their data dependencies upfront, and dynamic memory allocation is explicit. By separating program code from the state it operates on, the runtime can choreograph concurrent access. Gulf Stream the transaction forwarding protocol, which is Solana's mempool-less solution for forwarding and storing transactions before processing them. The Gossip Service acts as a gateway to nodes in the control plane. Validators use the service to ensure information is available to all other nodes in a cluster. TPU (Transaction Processing Unit) is the logic of the validator responsible for block production.

Halborn conducted a security audit on a set of changes to the Solana repository made between two different commits, beginning on August 21st, 2023 and ending on August 24th, 2023 . The security assessment was scoped to the updates to the master branch of the solana GitHub repository. Commit hashes and further details can be found in the **Scope** section of this report.

1.2 AUDIT SUMMARY

The team at Halborn was provided four weeks for the engagement and assigned a full-time security engineer to audit the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn did not identify any significant issue; however, some recommendations were given to reduce the likelihood and impact of risks, which were acknowledged by Solana Labs .

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of a manual review of the source code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the program audit. While manual testing is recommended to uncover flaws in business logic, processes, and implementation; automated testing techniques help enhance coverage of programs and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the audit:

- Research into the architecture, purpose, and use of the platform.
- Manual program source code review to identify business logic issues.
- Mapping out possible attack vectors
- Thorough assessment of safety and usage of critical Rust variables and functions in scope that could lead to arithmetic vulnerabilities.
- Finding unsafe Rust code usage (cargo-geiger)
- Scanning dependencies for known vulnerabilities (cargo audit).
- Local runtime testing (solana-test-framework)

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.4 SCOPE

Code repositories:

1. Solana L1

- Repository: solana
    - start: 124aaa95f65aac9a037db85a1ce27724bd2012ff
    - final: 6fbe8ba3b034b3fb79dd701928ea49e0e4271314
- Modules in scope:
    1. program-runtime (solana/program-runtime/src)
    2. runtime (solana/runtime/src)
    3. bpf_loader (solana/programs/bpf_loader/src)

Out-of-scope:

- third-party libraries and dependencies
- financial-related attacks

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	0	2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) UNPATCHED VERSION OF ED25519-DALEK DEPENDENCY	Informational (0.0)	ACKNOWLEDGED
(HAL-02) MISSING CARGO OVERFLOW CHECKS	Informational (0.0)	ACKNOWLEDGED

FINDINGS & TECH DETAILS

4.1 (HAL-01) UNPATCHED VERSION OF ED25519-DALEK DEPENDENCY - INFORMATIONAL (0.0)

Description:

An issue was identified in the used version of the ed25519-dalek which introduces a potential “Double Public Key Signing Function Oracle Attack” vulnerability concern.

In versions of ed25519-dalek prior to v2.0, private and public keys are treated as separate types, allowing them to be combined into a Keypair. Additionally, these versions provided APIs for serializing and deserializing 64-byte private/public key pairs.

The critical concern lies in the inherent insecurity of these APIs and serializations. Specifically, while computing the ‘S’ part of a cryptographic signature, these versions use the public key as one of the inputs, but the ‘R’ value is calculated independently of it.

Code Location:

Listing 1: Cargo.toml (Line 180)

```
178 dlopen_derive = "0.1.4"
179 eager = "0.1.0"
180 ed25519-dalek = "=1.0.1"
```

Listing 2: runtime/Cargo.toml (Line 80)

```
78 [dev-dependencies]
79 assert_matches = { workspace = true }
80 ed25519-dalek = { workspace = true }
```

Recommendation:

It is strongly recommended to upgrade the dependency to a secure and patched version (v2.0 or later) of ed25519-dalek. This will ensure improved security and mitigate the potential vulnerabilities and risks associated with the earlier versions.

Remediation Plan:

ACKNOWLDGED: The Solana Labs team acknowledged this finding.

4.2 (HAL-02) MISSING CARGO OVERFLOW CHECKS - INFORMATIONAL (0.0)

Description:

It has been observed that the changes applied to the assessment's scope do not include the implementation of the overflow-checks=true flag in the Cargo.toml files.

By default, overflow checks are disabled in optimized release builds. Consequently, any overflows occurring in release builds will remain silent, potentially causing unexpected application behavior. To ensure proper overflow handling, it is advisable to include the overflow-checks=true setting in the Cargo.toml file, even when using checked arithmetic through functions like checked_* or saturating_*.

Code Location:

- program-runtime/Cargo.toml
- programs/bpf_loader/Cargo.toml

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:U (0.0)

Recommendation:

Add overflow-checks=true under the release profile in the Cargo.toml files specified for recommended behavior.

Remediation Plan:

ACKNOWLDGED: The Solana Labs team acknowledged this finding.

MANUAL TESTING

In the manual testing phase, the following scenarios were simulated. The scenarios listed below were selected based on the severity of the vulnerabilities Halborn was testing the program for.

5.1 COMPUTE BUDGET

Description:

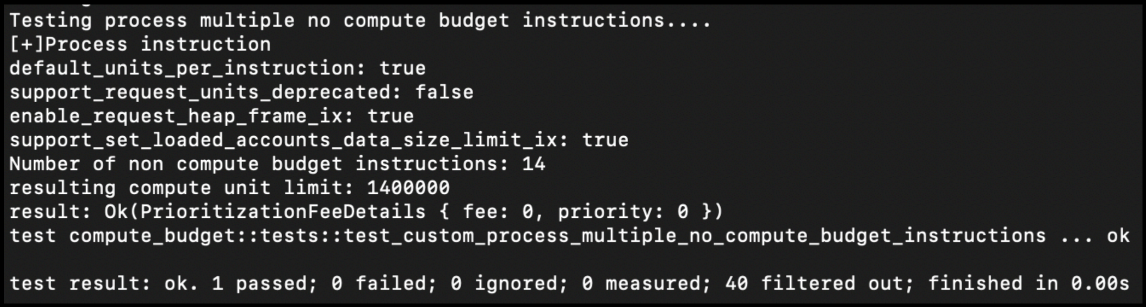
In commit 5de4cd2 The inappropriate use of usize for the variable representing the count of non-compute budget instructions, num_non_compute_budget_instructions was addressed in the compute budget module. It was replaced with the more suitable data type: u32.

This change was reviewed, and several tests have been performed to ensure that no security risks have been accidentally introduced beside the proper and expected calculation's result.

Results:

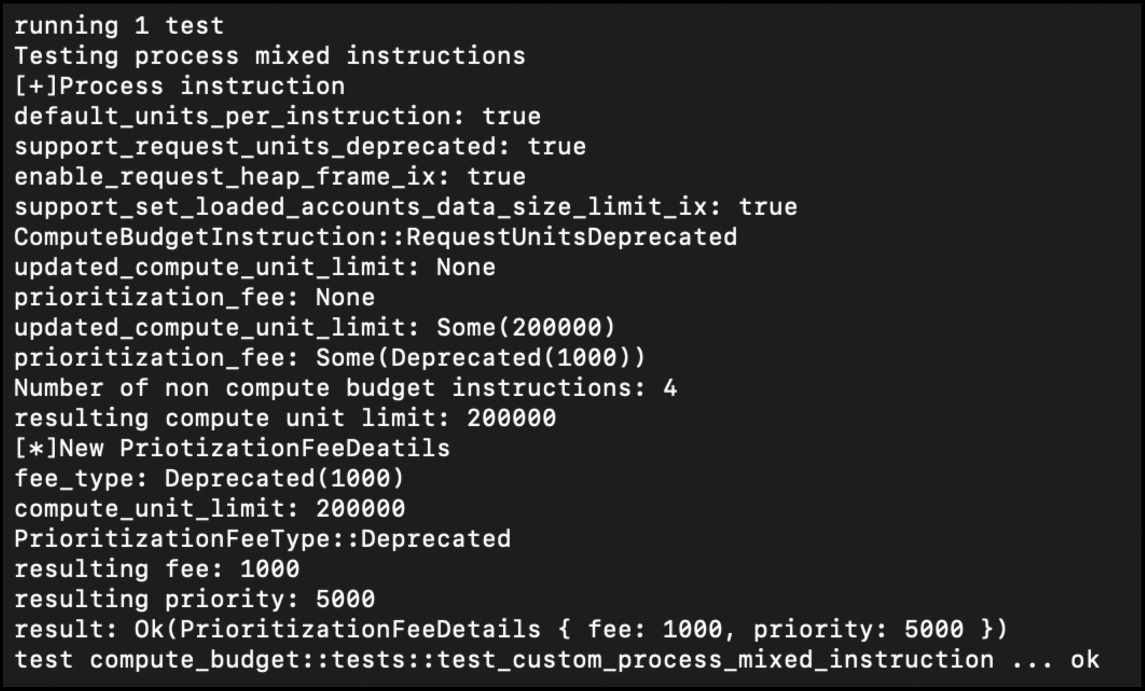
No vulnerabilities were identified.

Process multiple no compute budget instructions which would exceed the maximum value of the limit of compute units



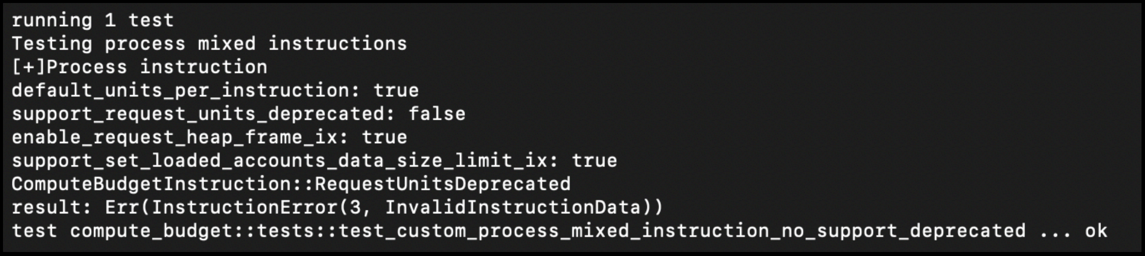
```
Testing process multiple no compute budget instructions....
[+]Process instruction
default_units_per_instruction: true
support_request_units_deprecated: false
enable_request_heap_frame_ix: true
support_set_loaded_accounts_data_size_limit_ix: true
Number of non compute budget instructions: 14
resulting compute unit limit: 1400000
result: Ok(PrioritizationFeeDetails { fee: 0, priority: 0 })
test compute_budget::tests::test_custom_process_multiple_no_compute_budget_instructions ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 40 filtered out; finished in 0.00s
```

Process multiple instructions including a ComputeBudget::Deprectaed with supporting request units deprecated



```
running 1 test
Testing process mixed instructions
[+]Process instruction
default_units_per_instruction: true
support_request_units_deprecated: true
enable_request_heap_frame_ix: true
support_set_loaded_accounts_data_size_limit_ix: true
ComputeBudgetInstruction::RequestUnitsDeprecated
updated_compute_unit_limit: None
prioritization_fee: None
updated_compute_unit_limit: Some(200000)
prioritization_fee: Some(Deprecated(1000))
Number of non compute budget instructions: 4
resulting compute unit limit: 200000
[*]New PriotizationFeeDeatils
fee_type: Deprecated(1000)
compute_unit_limit: 200000
PrioritizationFeeType::Deprecated
resulting fee: 1000
resulting priority: 5000
result: Ok(PrioritizationFeeDetails { fee: 1000, priority: 5000 })
test compute_budget::tests::test_custom_process_mixed_instruction ... ok
```

Process multiple instructions including a ComputeBudget::Deprectaed with-out supporting request units deprecated



```
running 1 test
Testing process mixed instructions
[+]Process instruction
default_units_per_instruction: true
support_request_units_deprecated: false
enable_request_heap_frame_ix: true
support_set_loaded_accounts_data_size_limit_ix: true
ComputeBudgetInstruction::RequestUnitsDeprecated
result: Err(InstructionError(3, InvalidInstructionData))
test compute_budget::tests::test_custom_process_mixed_instruction_no_support_deprecated ... ok
```

5.2 TRANSACTION PRIORITY DETAILS

Description:

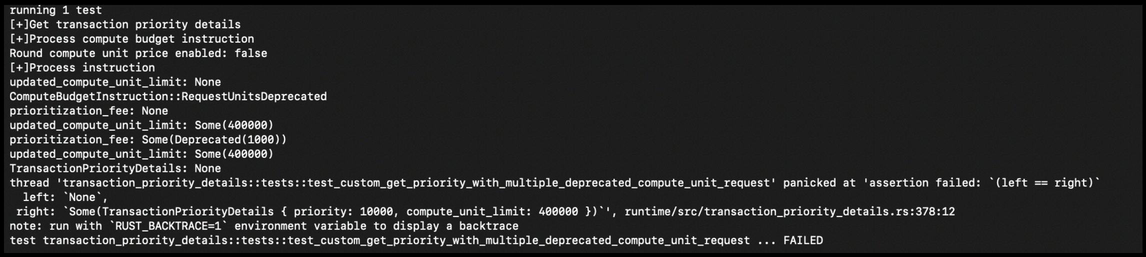
In commit 195469a support for deprecated compute budget instructions was added when obtaining transaction priority details.

Tests were conducted to ensure that these changes do not introduce any vulnerabilities, security risks, or compromise the correct functionality.

Results:

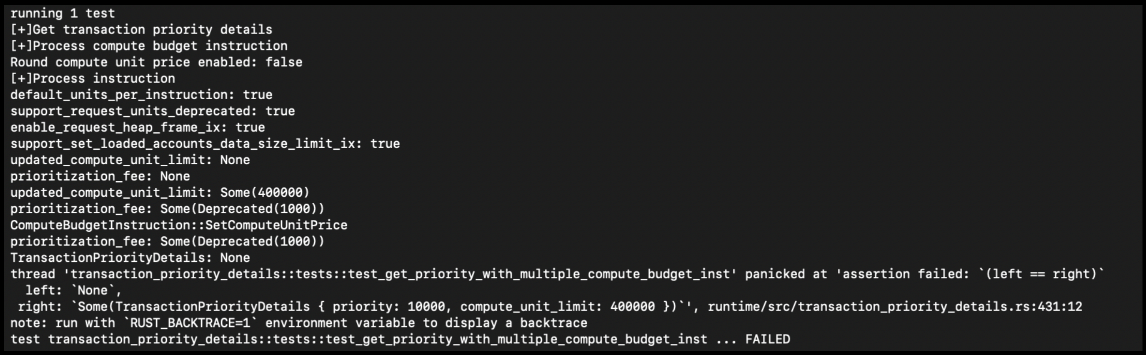
No vulnerabilities were identified.

Transaction priority details for multiple deprecated compute budget instructions



```
running 1 test
[+]Get transaction priority details
[+]Process compute budget instruction
Round compute unit price enabled: false
[+]Process instruction
updated_compute_unit_limit: None
ComputeBudgetInstruction::RequestUnitsDeprecated
prioritization_fee: None
updated_compute_unit_limit: Some(400000)
prioritization_fee: Some(Deprecated(1000))
updated_compute_unit_limit: Some(400000)
TransactionPriorityDetails: None
thread 'transaction_priority_details::tests::test_custom_get_priority_with_multiple_deprecated_compute_unit_request' panicked at 'assertion failed: `(left == right)`
  left: `None`,
 right: `Some(TransactionPriorityDetails { priority: 10000, compute_unit_limit: 400000 })`', runtime/src/transaction_priority_details.rs:378:12
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
test transaction_priority_details::tests::test_custom_get_priority_with_multiple_deprecated_compute_unit_request ... FAILED
```

Transaction priority details for deprecated and set unit price compute budget instructions



```
running 1 test
[+]Get transaction priority details
[+]Process compute budget instruction
Round compute unit price enabled: false
[+]Process instruction
default_units_per_instruction: true
support_request_units_deprecated: true
enable_request_heap_frame_ix: true
support_set_loaded_accounts_data_size_limit_ix: true
updated_compute_unit_limit: None
prioritization_fee: None
updated_compute_unit_limit: Some(400000)
prioritization_fee: Some(Deprecated(1000))
ComputeBudgetInstruction::SetComputeUnitPrice
prioritization_fee: Some(Deprecated(1000))
TransactionPriorityDetails: None
thread 'transaction_priority_details::tests::test_get_priority_with_multiple_compute_budget_inst' panicked at 'assertion failed: `(left == right)`
  left: `None`,
 right: `Some(TransactionPriorityDetails { priority: 10000, compute_unit_limit: 400000 })`', runtime/src/transaction_priority_details.rs:431:12
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
test transaction_priority_details::tests::test_get_priority_with_multiple_compute_budget_inst ... FAILED
```

5.3 COST MODEL

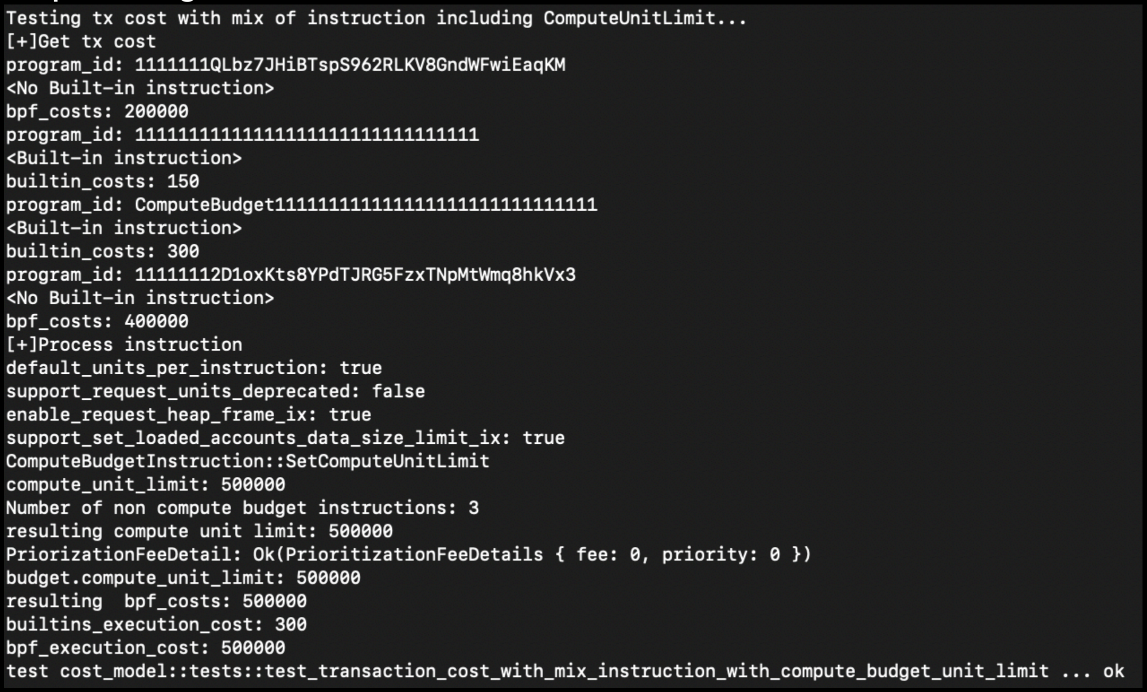
Description:

In commit 8ed7ebe modifications were made to address the issue about cost model being able to double count built-in instruction cost, described in #32422.
These changes were implemented to ensure that the computation of bpf_execution_cost adheres to the expected behavior.

Results:

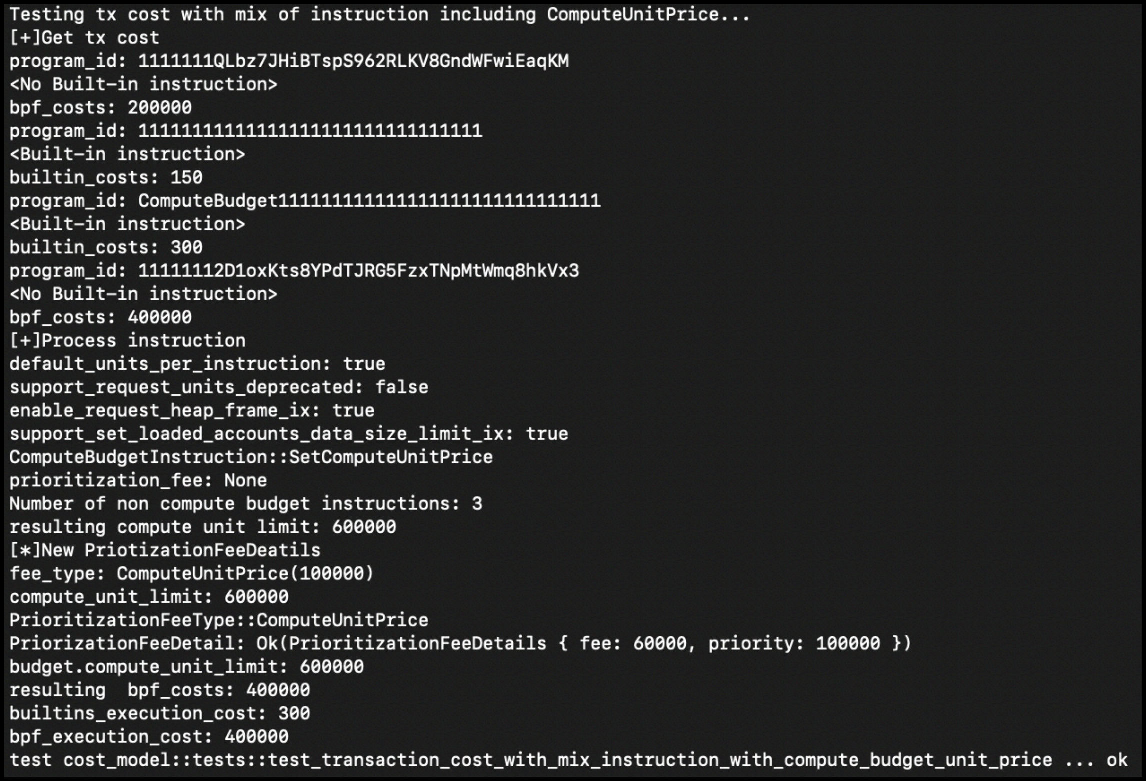
No vulnerabilities were identified.

Get transaction cost with no-builtin and built-in instructions including compute budget unit limit



```
Testing tx cost with mix of instruction including ComputeUnitLimit...
[+]Get tx cost
program_id: 1111111QLbz7JHiBTspS962RLKV8GndWFwiEaqKM
<No Built-in instruction>
bpf_costs: 200000
program_id: 11111111111111111111111111111111
<Built-in instruction>
builtin_costs: 150
program_id: ComputeBudget111111111111111111111111111111
<Built-in instruction>
builtin_costs: 300
program_id: 11111112D1oxKts8YPdTJRG5FzxTNpMtWmq8hkVx3
<No Built-in instruction>
bpf_costs: 400000
[+]Process instruction
default_units_per_instruction: true
support_request_units_deprecated: false
enable_request_heap_frame_ix: true
support_set_loaded_accounts_data_size_limit_ix: true
ComputeBudgetInstruction::SetComputeUnitLimit
compute_unit_limit: 500000
Number of non compute budget instructions: 3
resulting compute unit limit: 500000
PriorizationFeeDetail: Ok(PrioritizationFeeDetails { fee: 0, priority: 0 })
budget.compute_unit_limit: 500000
resulting  bpf_costs: 500000
builtins_execution_cost: 300
bpf_execution_cost: 500000
test cost_model::tests::test_transaction_cost_with_mix_instruction_with_compute_budget_unit_limit ... ok
```

Get transaction cost with no-builtin and built-in instruction, including compute budget unit price



```
Testing tx cost with mix of instruction including ComputeUnitPrice...
[+]Get tx cost
program_id: 1111111QLbz7JHiBTspS962RLKV8GndWFwiEaqKM
<No Built-in instruction>
bpf_costs: 200000
program_id: 11111111111111111111111111111111
<Built-in instruction>
builtin_costs: 150
program_id: ComputeBudget111111111111111111111111111111
<Built-in instruction>
builtin_costs: 300
program_id: 11111112D1oxKts8YPdTJRG5FzxTNpMtWmq8hkVx3
<No Built-in instruction>
bpf_costs: 400000
[+]Process instruction
default_units_per_instruction: true
support_request_units_deprecated: false
enable_request_heap_frame_ix: true
support_set_loaded_accounts_data_size_limit_ix: true
ComputeBudgetInstruction::SetComputeUnitPrice
prioritization_fee: None
Number of non compute budget instructions: 3
resulting compute unit limit: 600000
[*]New PriotizationFeeDeatils
fee_type: ComputeUnitPrice(100000)
compute_unit_limit: 600000
PrioritizationFeeType::ComputeUnitPrice
PriorizationFeeDetail: Ok(PrioritizationFeeDetails { fee: 60000, priority: 100000 })
budget.compute_unit_limit: 600000
resulting  bpf_costs: 400000
builtins_execution_cost: 300
bpf_execution_cost: 400000
test cost_model::tests::test_transaction_cost_with_mix_instruction_with_compute_budget_unit_price ... ok
```

5.4 STAKES

Description:

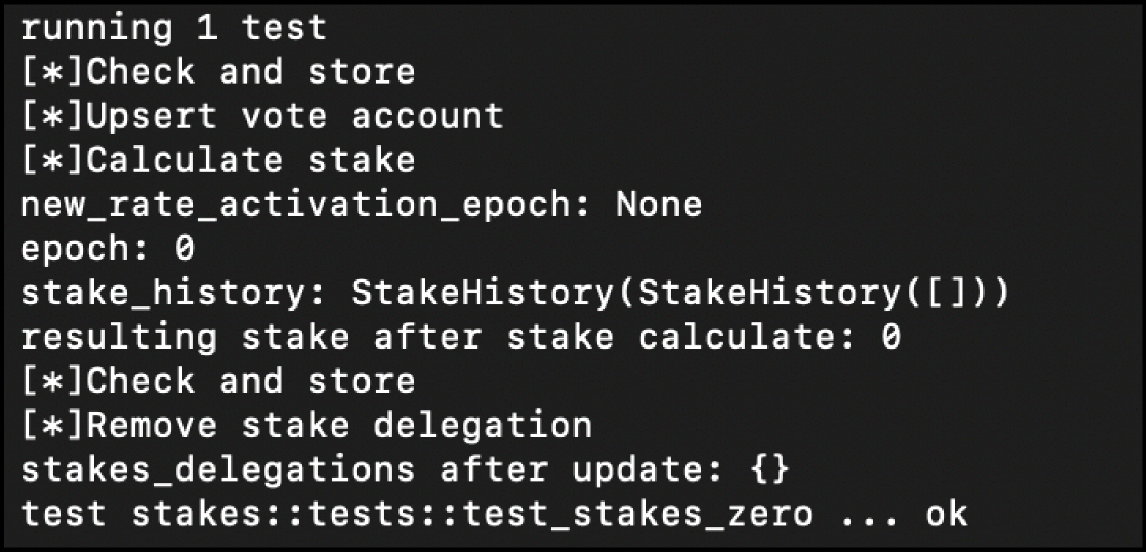
In commit 8fea568 changes were added to solve the issue around the storage of warm-up and cooldown parameters per delegation within the system. In response to this issue, the decision has been made to deprecate the usage of stake config and the warm-up/cooldown rate in the context of delegation. Instead, a new approach is introduced where hardcoded values will be employed directly within the stake program.

This aims to ensure uniformity in delegation behavior across all instances, reducing complexity and potential configuration errors.

Results:

No vulnerabilities were identified.

Check and store with zero stake



```
running 1 test
[*]Check and store
[*]Upsert vote account
[*]Calculate stake
new_rate_activation_epoch: None
epoch: 0
stake_history: StakeHistory(StakeHistory([]))
resulting stake after stake calculate: 0
[*]Check and store
[*]Remove stake delegation
stakes_delegations after update: {}
test stakes::tests::test_stakes_zero ... ok
```

AUTOMATED TESTING

6.1 AUTOMATED ANALYSIS

Description:

Halborn used automated security scanners to assist with the detection of well-known security issues and vulnerabilities. Among the tools used was cargo-audit, a security scanner for vulnerabilities reported to the Rust-Sec Advisory Database. All vulnerabilities published in https://crates.io are stored in a repository named The RustSec Advisory Database. cargo audit is a human-readable version of the advisory database which performs a scanning on Cargo.lock. Security Detections are only in scope. All vulnerabilities shown here were already disclosed in the above report. However, to better assist the developers maintaining this code, the auditors are including the output with the dependencies tree, and this is included in the cargo audit output to better know the dependencies affected by unmaintained and vulnerable crates.

Results:

ID	**package**	**Short Description**
RUSTSEC-2022-0093	ed25519-dalek	Double Public Key Signing Function Oracle Attack on ‘ed25519-dalek‘
RUSTSEC-2020-0071	time	Potential segfault in the time crate
RUSTSEC-2023-0001	tokio	Configuration corruption

6.2 UNSAFE RUST CODE DETECTION

Description:

Halborn used automated security scanners to assist with the detection of well-known security issues and vulnerabilities. Among the tools used was cargo-geiger, a security tool that lists statistics related to the usage of unsafe Rust code in a core Rust codebase and all its dependencies.

Results:

No unsafe code blocks were identified in the packages in scope and their dependencies.

THANK YOU FOR CHOOSING

HALBORN