CANTINA

Ethereum Pectra: Grandine

Competition

July 21, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
High	*Must* fix as soon as possible (if already deployed) and can be triggered by any user without significant constraints, generating outsized returns to the exploiter. For example: loss of user funds (significant amount of funds being stolen or lost) or breaking core functionality (failure in fundamental protocol operations).
Medium	Global losses <10% or losses to only a subset of users, requiring significant constraints (capital, planning, other users...) to be exploited. For example: temporary disruption or denial of service (DoS), minor fund loss or exposure or breaking non-core functionality
Low	Losses will be annoying but easily recoverable, requiring unusual scenarios or admin actions to be exploited.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above matrix. High severity findings represent the most critical issues that must be addressed immediately, as they either have high impact and high likelihood of occurrence, or medium impact with high likelihood.

Medium severity findings represent issues that, while not immediately critical, still pose significant risks and should be addressed promptly. These typically involve scenarios with medium impact and medium likelihood, or high impact with low likelihood.

Low severity findings represent issues that, while not posing immediate threats, could potentially cause problems in specific scenarios. These typically involve medium impact with low likelihood, or low impact with medium likelihood.

Lastly, some findings might represent improvements that don't directly impact security but could enhance the codebase's quality, readability, or efficiency (Gas and Informational findings).

2 Security Review Summary

Ethereum is a worldwide system, an open-source platform to write computer code that stores and automates digital databases using smart contracts, without relying upon a central intermediary, solving trust with cryptographic techniques.

From Feb 21st to Mar 27th Cantina hosted a competition based on the Ethereum Pectra upgrade. The present report focuses in the grandine implementation. The participants identified a total of **2** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 0
- Low Risk: 0
- Gas Optimizations: 0
- Informational: 2

3 Findings

3.1 Informational

3.1.1 Grandine does not validate the ordering and uniqueness of execution requests during deserialization

Submitted by alexfilippov314

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The Engine API specification outlines specific requirements for `executionRequests` in the response of `engine_getPayloadV4`:

> The call MUST return executionRequests list representing execution layer triggered requests. Each list element is a requests byte array as defined by EIP-7685. The first byte of each element is the request_type and the remaining bytes are the request_data. Elements of the list MUST be ordered by request_type in ascending order. Elements with empty request_data MUST be excluded from the list. Elements MUST be longer than 1-byte, and each request_type MUST be unique within the list.

Prysm, Lighthouse, and Teku validate ordering and uniqueness of `executionRequests` during the deserialization of the `engine_getPayloadV4` response from the execution client. This behavior is evident in the following code snippets:

- Prysm:

```
if prevTypeNum != nil && *prevTypeNum >= requestType {
    return nil, errors.New("invalid execution request type order or duplicate requests, requests should
    ↪  be in sorted order and unique")
}
```

- Lighthouse:

```
if prev.to_u8() >= current_prefix.to_u8() {
    return Err(RequestsError::InvalidOrdering);
}
```

- Teku:

```
if (requestType <= previousRequestType) {
    throw new IllegalArgumentException(
        "Execution requests are not in strictly ascending order");
}
```

However, Grandine does not check (types.rs#L748) ordering and uniqueness of execution requests. As a result, duplicate or incorrectly ordered requests are not detected, violating the Engine API specification. To my knowledge, the only impact of this issue is that it can complicate debugging and monitoring if the execution client behaves incorrectly. In some scenarios, it could potentially lead to proposing an invalid block when incorrect behavior could have been caught locally. However, as far as I know, it does not have any direct impact. Still, it is better to add this validation for the sake of code consistency across consensus clients and to improve errors handling.

Proof of Concept: To reproduce the issue, add the following test to `execution_engine/src/types.rs` and run it:

```
#[test]:
fn test_poc_execution_requests_incorrect_ordering() -> Result<()> {
    let execution_requests = ExecutionRequests::<Mainnet> {
        deposits: ContiguousList::try_from(vec![
            DepositRequest {
                pubkey: hex!("92f9fe7570a6650d030bb2227d699c744303d08a887cd2e1592e30906cd8cedf9646c1a1afd90223
                ↪  5bb36620180eb688").into(),
                withdrawal_credentials:
                ↪  hex!("020000000000000000000000065d08a056c17ae13370565b04cf77d2afa1cb9fa").into(),
                amount: 1_000_000_000_000,
```

```
            signature: hex!("a13741d65b47825c147201cfce3360438d4011fe81b455e86226c95a2669bfde14712ba36d1c2
            ↪   f44371a98bf28ff38370ce7d28c65872bf65ff88d6014468676029e298903c89c51c27ab5f07e178b8b14d3ca1
            ↪   91e2ce3b24703629e3994e05b").into(),
            index: 0,
        },
        DepositRequest {
            pubkey: hex!("90a58546229c585cef35f3afab904411530303d95c371e246a2e9a1ef6beb5db7a98c2fd79a38870
            ↪   9a30ec782576a5d6").into(),
            withdrawal_credentials:
            ↪   hex!("020000000000000000000000065d08a056c17ae13370565b04cf77d2afa1cb9fa").into(),
            amount: 1_000_000_000_000,
            signature: hex!("b23e205d2fcfc3e9d3ae58c0f78b55b19f97f59eaf43d85113a1960ee2c38f6b4ef705302e46e
            ↪   0593fc41ba5632b047a14d76dc82bb2619d7c73e0d89da2eda2ea11fff9036c2d08f9d457c07f23b1411ecd13f
            ↪   f0e9c00eeb85d851bae2494e0").into(),
            index: 1,
        }
    ])?,
    withdrawals: ContiguousList::try_from(vec![
        WithdrawalRequest {
            source_address: ExecutionAddress::repeat_byte(2),
            validator_pubkey: hex!("aaf9fe7570a6650d030bb2227d699c744303d08a887cd2e1592e30906cd8cedf9646c1
            ↪   a1afd902235bb36620180eb688").into(),
            amount: 4_000_000_000_000,
        },
    ])?,
    consolidations: ContiguousList::try_from(vec![
        ConsolidationRequest {
            source_pubkey: hex!("aaf9fe7570a6650d030bb2227d699c744303d08a887cd2e1592e30906cd8cedf9646c1a1a
            ↪   fd902235bb36620180eb688").into(),
            source_address: ExecutionAddress::repeat_byte(3),
            target_pubkey: hex!("abc9fe7570a6650d030bb2227d699c744303d08a887cd2e1592e30906cd8cedf9646c1a1a
            ↪   fd902235bb36620180eb688").into(),
        },
    ])?,
};

let raw_execution_requests = RawExecutionRequests::from(execution_requests);
let serialized = serde_json::to_value(raw_execution_requests.clone())?;

let expected_serialzed = json!([
    "0x0092f9fe7570a6650d030bb2227d699c744303d08a887cd2e1592e30906cd8cedf9646c1a1afd902235bb36620180eb6880
    ↪   20000000000000000000000065d08a056c17ae13370565b04cf77d2afa1cb9fa0010a5d4e8000000a13741d65b47825c147
    ↪   201cfce3360438d4011fe81b455e86226c95a2669bfde14712ba36d1c2f44371a98bf28ff38370ce7d28c65872bf65ff88
    ↪   d6014468676029e298903c89c51c27ab5f07e178b8b14d3ca191e2ce3b24703629e3994e05b000000000000000090a5854
    ↪   6229c585cef35f3afab904411530303d95c371e246a2e9a1ef6beb5db7a98c2fd79a388709a30ec782576a5d6020000000
    ↪   00000000000000065d08a056c17ae13370565b04cf77d2afa1cb9fa0010a5d4e8000000b23e205d2fcfc3e9d3ae58c0f78
    ↪   b55b19f97f59eaf43d85113a1960ee2c38f6b4ef705302e46e0593fc41ba5632b047a14d76dc82bb2619d7c73e0d89da2e
    ↪   da2ea11fff9036c2d08f9d457c07f23b1411ecd13ff0e9c00eeb85d851bae2494e00100000000000000",
    "0x010202020202020202020202020202020202020202aaf9fe7570a6650d030bb2227d699c744303d08a887cd2e1592e30906
    ↪   cd8cedf9646c1a1afd902235bb36620180eb68800409452a3030000",
    "0x020303030303030303030303030303030303030303aaf9fe7570a6650d030bb2227d699c744303d08a887cd2e1592e30906
    ↪   cd8cedf9646c1a1afd902235bb36620180eb688abc9fe7570a6650d030bb2227d699c744303d08a887cd2e1592e30906cd
    ↪   8cedf9646c1a1afd902235bb36620180eb688",
]);

// The ordering here is incorrect and there are also duplicate requests
let incorrect = json!([
    "0x020303030303030303030303030303030303030303aaf9fe7570a6650d030bb2227d699c744303d08a887cd2e1592e30906
    ↪   cd8cedf9646c1a1afd902235bb36620180eb688abc9fe7570a6650d030bb2227d699c744303d08a887cd2e1592e30906cd
    ↪   8cedf9646c1a1afd902235bb36620180eb688",
    "0x0092f9fe7570a6650d030bb2227d699c744303d08a887cd2e1592e30906cd8cedf9646c1a1afd902235bb36620180eb6880
    ↪   20000000000000000000000065d08a056c17ae13370565b04cf77d2afa1cb9fa0010a5d4e8000000a13741d65b47825c147
    ↪   201cfce3360438d4011fe81b455e86226c95a2669bfde14712ba36d1c2f44371a98bf28ff38370ce7d28c65872bf65ff88
    ↪   d6014468676029e298903c89c51c27ab5f07e178b8b14d3ca191e2ce3b24703629e3994e05b000000000000000090a5854
    ↪   6229c585cef35f3afab904411530303d95c371e246a2e9a1ef6beb5db7a98c2fd79a388709a30ec782576a5d6020000000
    ↪   00000000000000065d08a056c17ae13370565b04cf77d2afa1cb9fa0010a5d4e8000000b23e205d2fcfc3e9d3ae58c0f78
    ↪   b55b19f97f59eaf43d85113a1960ee2c38f6b4ef705302e46e0593fc41ba5632b047a14d76dc82bb2619d7c73e0d89da2e
    ↪   da2ea11fff9036c2d08f9d457c07f23b1411ecd13ff0e9c00eeb85d851bae2494e00100000000000000",
    "0x010202020202020202020202020202020202020202aaf9fe7570a6650d030bb2227d699c744303d08a887cd2e1592e30906
    ↪   cd8cedf9646c1a1afd902235bb36620180eb68800409452a3030000",
    "0x010202020202020202020202020202020202020202aaf9fe7570a6650d030bb2227d699c744303d08a887cd2e1592e30906
    ↪   cd8cedf9646c1a1afd902235bb36620180eb68800409452a3030000",
]);

assert_eq!(serialized, expected_serialzed);
```

```
    assert_eq!(
        serde_json::from_value::<RawExecutionRequests::<Mainnet>>(expected_serialzed)?,
        raw_execution_requests,
    );
    // Despite all of this, deserialization still doesn't produce an error
    assert_eq!(
        serde_json::from_value::<RawExecutionRequests::<Mainnet>>(incorrect)?,
        raw_execution_requests,
    );

    Ok(())
}
```

Recommendation: Consider adding ordering and uniqueness validation of execution requests.

3.1.2 Grandine does not validate the non-emptiness of execution requests during deserialization

Submitted by alexfilippov314

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The Engine API specification outlines specific requirements for `executionRequests` in the response of `engine_getPayloadV4`:

> The call MUST return executionRequests list representing execution layer triggered requests. Each list element is a requests byte array as defined by EIP-7685. The first byte of each element is the request_type and the remaining bytes are the request_data. Elements of the list MUST be ordered by request_type in ascending order. Elements with empty request_data MUST be excluded from the list. Elements MUST be longer than 1-byte, and each request_type MUST be unique within the list.

Prysm, Lighthouse, and Teku validate the non-emptiness of `executionRequests` during the deserialization of the `engine_getPayloadV4` response from the execution client. This behavior is evident in the following code snippets:

- Prysm:

```
if len(requestListInSSZBytes) < drSize {
    return nil, fmt.Errorf("invalid deposit requests SSZ size, got %d expected at least %d",
    ↪  len(requestListInSSZBytes), drSize)
}
```

- Lighthouse:

```
if request_bytes.is_empty() {
    return Err(RequestsError::EmptyRequest(i));
}
```

- Teku:

```
if (requestData.isEmpty()) {
    throw new IllegalArgumentException("Empty data for request type " + requestType);
}
```

However, Grandine does not check the non-emptiness of execution requests (types.rs#L748). As a result, empty requests are not detected, violating the Engine API specification. To my knowledge, the only impact of this issue is that it can complicate debugging and monitoring if the execution client behaves incorrectly. In some scenarios, it could potentially lead to proposing an invalid block when incorrect behavior could have been caught locally. However it doesn't seem to have any direct impact. Still, it is better to add this validation for the sake of code consistency across consensus clients and to improve errors handling.

Proof of Concept: To reproduce the issue, add the following test to `execution_engine/src/types.rs` and run it:

```rust
#[test]:
fn test_poc_empty_execution_requests() {
    let execution_requests = ExecutionRequests::<Mainnet> {
        deposits: ContiguousList::default(),
        withdrawals: ContiguousList::default(),
        consolidations: ContiguousList::default(),
    };

    // All requests are empty
    let incorrect = json!([
        "0x00",
        "0x01",
        "0x02",
    ]);

    let raw_execution_requests = RawExecutionRequests::from(execution_requests);
    // Despite this deserialization doesn't produce an error
    assert_eq!(
        serde_json::from_value::<RawExecutionRequests::<Mainnet>>(incorrect).unwrap(),
        raw_execution_requests,
    );
}
```

Recommendation: Consider validating that execution requests are not empty.