CANTINA

Ethereum Pectra: Teku

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 teku implementation. The participants identified a total of **3** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 1
- Low Risk: 0
- Gas Optimizations: 0
- Informational: 2

3 Findings

3.1 Medium Risk

3.1.1 Teku incorrectly retrieves pending partial withdrawals, leading to chain split due to incorrect "*expected withdrawals*"

Submitted by zigtur

Severity: Medium Risk

Context: ExpectedWithdrawals.java#L157-L205, ExpectedWithdrawalsTest.java#L141-L176

Summary: Teku will collect less pending partial withdrawals than the `MAX_PENDING_PARTIALS_PER_WITHDRAWALS_SWEEP` limit when one of the validator does not have enough balance.

Context and specifications: The beacon chain specifications shows the following in `get_expected_withdrawals` for retrieving pending partial withdrawals.

```
def get_expected_withdrawals(state: BeaconState) -> Tuple[Sequence[Withdrawal], uint64]:
    epoch = get_current_epoch(state)
    withdrawal_index = state.next_withdrawal_index
    validator_index = state.next_withdrawal_validator_index
    withdrawals: List[Withdrawal] = []
    processed_partial_withdrawals_count = 0

    # [New in Electra:EIP7251] Consume pending partial withdrawals
    for withdrawal in state.pending_partial_withdrawals:
        if withdrawal.withdrawable_epoch > epoch or len(withdrawals) ==
        ↪  MAX_PENDING_PARTIALS_PER_WITHDRAWALS_SWEEP:
            break

        validator = state.validators[withdrawal.validator_index]
        has_sufficient_effective_balance = validator.effective_balance >= MIN_ACTIVATION_BALANCE
        has_excess_balance = state.balances[withdrawal.validator_index] > MIN_ACTIVATION_BALANCE
        if validator.exit_epoch == FAR_FUTURE_EPOCH and has_sufficient_effective_balance and has_excess_balance:
            withdrawable_balance = min(
                state.balances[withdrawal.validator_index] - MIN_ACTIVATION_BALANCE,
                withdrawal.amount)
            withdrawals.append(Withdrawal(
                index=withdrawal_index,
                validator_index=withdrawal.validator_index,
                address=ExecutionAddress(validator.withdrawal_credentials[12:]),
                amount=withdrawable_balance,
            ))
            withdrawal_index += WithdrawalIndex(1)

        processed_partial_withdrawals_count += 1
```

As we can see, the `for` loop breaks when `len(withdrawals) == MAX_PENDING_PARTIALS_PER_WITHDRAWALS_SWEEP`. Also note that a pending partial withdrawal is not added to the `withdrawals` when the validator does not have enough balance.

Description: Teku does not comply with the specifications. It does not stop processing pending partial withdrawals when `len(withdrawals) == MAX_PENDING_PARTIALS_PER_WITHDRAWALS_SWEEP`. Indeed, it will stop processing when the number of loop round is `MAX_PENDING_PARTIALS_PER_WITHDRAWALS_SWEEP`. This is different, because a loop round does not always add a withdrawal to the `withdrawals` array.

Impact Explanation: Teku will return incorrect expected withdrawals from `get_expected_withdrawals` function when a partial withdrawal is not processable. This will lead to chain splits of Teku, between 27 and 28% of the network.

Code snippet: Teku's `get_expected_withdrawals` logic is splitted in two functions (`getPendingPartialWithdrawals` and `getExpectedWithdrawals`). This split seems legit as the specifications `get_expected_withdrawals` execute these two processing independently. The issue comes from `getPendingPartialWithdrawals` which does not comply with the specs. The `@POC` comment should be followed to highlight the issue:

```
private static WithdrawalSummary getPendingPartialWithdrawals(
    final BeaconStateElectra preState,
    final SchemaDefinitionsElectra schemaDefinitions,
    final MiscHelpersElectra miscHelpers,
```

```
    final SpecConfigElectra specConfig) {
  final UInt64 epoch = miscHelpers.computeEpochAtSlot(preState.getSlot());
  final int maxPendingPartialWithdrawals = specConfig.getMaxPendingPartialsPerWithdrawalsSweep();
  final List<Withdrawal> partialWithdrawals = new ArrayList<>();
  final SszList<PendingPartialWithdrawal> pendingPartialWithdrawals =
      preState.getPendingPartialWithdrawals();
  int processedPartialWithdrawalsCount = 0;
  UInt64 withdrawalIndex = preState.getNextWithdrawalIndex();

  for (int i = 0; i < pendingPartialWithdrawals.size() && i < maxPendingPartialWithdrawals; i++) { // @POC: `i <
  ↪  maxPendingPartialWithdrawals` check is not the same as `len(partialWithdrawals) <
  ↪  maxPendingPartialWithdrawals`!!!
    final PendingPartialWithdrawal pendingPartialWithdrawal = pendingPartialWithdrawals.get(i);
    if (pendingPartialWithdrawal.getWithdrawableEpoch().isGreaterThan(epoch)) {
      break;
    }
    final Validator validator =
        preState.getValidators().get(pendingPartialWithdrawal.getValidatorIndex());
    final boolean hasSufficientBalance =
        validator
            .getEffectiveBalance()
            .isGreaterThanOrEqualTo(specConfig.getMinActivationBalance());
    final UInt64 validatorBalance =
        preState.getBalances().get(pendingPartialWithdrawal.getValidatorIndex()).get();
    final boolean hasExcessBalance =
        validatorBalance.isGreaterThan(specConfig.getMinActivationBalance());
    if (validator.getExitEpoch().equals(FAR_FUTURE_EPOCH)
        && hasSufficientBalance // @POC: each round loop will not always add an element in `partialWithdrawals`
        && hasExcessBalance) {
      final UInt64 withdrawableBalance =
          pendingPartialWithdrawal
              .getAmount()
              .min(validatorBalance.minusMinZero(specConfig.getMinActivationBalance()));
      partialWithdrawals.add(
          schemaDefinitions
              .getWithdrawalSchema()
              .create(
                  withdrawalIndex,
                  UInt64.valueOf(pendingPartialWithdrawal.getValidatorIndex()),
                  new Bytes20(validator.getWithdrawalCredentials().slice(12)),
                  withdrawableBalance));
      withdrawalIndex = withdrawalIndex.increment();
    }
    processedPartialWithdrawalsCount++;
  }
  return new WithdrawalSummary(partialWithdrawals, processedPartialWithdrawalsCount);
}
```

Proof of concept: The unit test in the `ExpectedWithdrawalsTest.java` show the inconsistency, especially the following comment:

```
@Test
void electraPendingPartialCountsSkippedWithdrawals() {
  spec = TestSpecFactory.createMinimalElectra();
  dataStructureUtil = new DataStructureUtil(spec);
  final SpecConfigElectra specConfigElectra =
      SpecConfigElectra.required(spec.getGenesisSpec().getConfig());
  final UInt64 electraMaxBalance = specConfigElectra.getMaxEffectiveBalance();
  final long partialWithdrawalBalance = 10241024L;

  final BeaconStateElectra preState =
      BeaconStateElectra.required(
          new BeaconStateTestBuilder(dataStructureUtil)
              .activeConsolidatingValidator(electraMaxBalance.plus(partialWithdrawalBalance))
              // the two validators below are skipped because they are queued for exit therefore
              // they're withdrawals are skipped
              .activeConsolidatingValidatorQueuedForExit(
                  electraMaxBalance.plus(partialWithdrawalBalance + 1))
              .activeConsolidatingValidatorQueuedForExit(
                  electraMaxBalance.plus(partialWithdrawalBalance + 2))
              .pendingPartialWithdrawal(0, electraMaxBalance.plus(partialWithdrawalBalance))
              .pendingPartialWithdrawal(
                  1, electraMaxBalance.plus(partialWithdrawalBalance).plus(1))
              .pendingPartialWithdrawal(
                  2, electraMaxBalance.plus(partialWithdrawalBalance).plus(2))
              .build());

  final ExpectedWithdrawals withdrawals =
      spec.getBlockProcessor(preState.getSlot()).getExpectedWithdrawals(preState);

  // even having 2 of the 3 partial withdrawals skipped,
  // the count should be 2(which is the MAX_PENDING_PARTIALS_PER_WITHDRAWALS_SWEEP for minimal
  // spec)
  // because we consider the skipped in the count
  assertThat(withdrawals.getPartialWithdrawalCount()).isEqualTo(2);
}
```

To go further, the following patch can be applied to modify the unit test:

```
diff --git
↪  a/ethereum/spec/src/test/java/tech/pegasys/teku/spec/datastructures/execution/ExpectedWithdrawalsTest.java
↪  b/ethereum/spec/src/test/java/tech/pegasys/teku/spec/datastructures/execution/ExpectedWithdrawalsTest.java
index 1abd324d72..a4d1594ac9 100644
--- a/ethereum/spec/src/test/java/tech/pegasys/teku/spec/datastructures/execution/ExpectedWithdrawalsTest.java
+++ b/ethereum/spec/src/test/java/tech/pegasys/teku/spec/datastructures/execution/ExpectedWithdrawalsTest.java
@@ -14,8 +14,8 @@
 package tech.pegasys.teku.spec.datastructures.execution;

 import static org.assertj.core.api.Assertions.assertThat;
-
 import org.junit.jupiter.api.Test;
+
 import tech.pegasys.teku.infrastructure.unsigned.UInt64;
 import tech.pegasys.teku.spec.Spec;
 import tech.pegasys.teku.spec.TestSpecFactory;
@@ -172,5 +172,44 @@ class ExpectedWithdrawalsTest {
     // spec)
     // because we consider the skipped in the count
     assertThat(withdrawals.getPartialWithdrawalCount()).isEqualTo(2);
+    // @POC: The explanations above is incorrect as it does not comply with the specs
+    // @POC: It shouldn't be the partial withdrawal count that is equal to
↪  MAX_PENDING_PARTIALS_PER_WITHDRAWALS_SWEEP
+    // @POC: It should be the number of partial withdrawals in the withdrawals list that is equal to
↪  MAX_PENDING_PARTIALS_PER_WITHDRAWALS_SWEEP
+    assertThat(withdrawals.getWithdrawalList().size()).isEqualTo(1);
+  }
+
+
+  @Test
+  void electraPendingPartialCountsSkippedWithdrawalsPocIncorrectExpectedPendingWithdrawals() {
+    spec = TestSpecFactory.createMinimalElectra();
+    dataStructureUtil = new DataStructureUtil(spec);
+    final SpecConfigElectra specConfigElectra =
+        SpecConfigElectra.required(spec.getGenesisSpec().getConfig());
+    final UInt64 electraMaxBalance = specConfigElectra.getMaxEffectiveBalance();
+    final long partialWithdrawalBalance = 10241024L;
```

```
+
+    final BeaconStateElectra preState =
+        BeaconStateElectra.required(
+            new BeaconStateTestBuilder(dataStructureUtil)
+                .activeConsolidatingValidator(electraMaxBalance.plus(partialWithdrawalBalance))
+                // Only one validator is queued for exit. The pending withdrawal for the other validator
↪  should be processed
+                .activeConsolidatingValidatorQueuedForExit(
+                    electraMaxBalance.plus(partialWithdrawalBalance + 1))
+                .activeConsolidatingValidator(
+                    electraMaxBalance.plus(partialWithdrawalBalance + 2))
+                .pendingPartialWithdrawal(0, electraMaxBalance.plus(partialWithdrawalBalance))
+                .pendingPartialWithdrawal(
+                    1, electraMaxBalance.plus(partialWithdrawalBalance).plus(1))
+                .pendingPartialWithdrawal(
+                    2, electraMaxBalance.plus(partialWithdrawalBalance).plus(2))
+                .build());
+
+    final ExpectedWithdrawals withdrawals =
+        spec.getBlockProcessor(preState.getSlot()).getExpectedWithdrawals(preState);
+
+    // @POC: with only 1 of the 3 partial withdrawals skipped,
+    // the count should be 3 and the withdrawals size should be 2 (MAX_PENDING_PARTIALS_PER_WITHDRAWALS_SWEEP
↪  for minimal spec is 2)
+    assertThat(withdrawals.getPartialWithdrawalCount()).isEqualTo(2); // @POC: should be 3
+    assertThat(withdrawals.getWithdrawalList().size()).isEqualTo(1);  // @POC: should be 2
   }
 }
```

Recommendation: Teku's `getPendingPartialWithdrawals` must implement the length check on `partialWithdrawals`.

3.2 Informational

3.2.1 EIP7251 - `PendingDeposit` container `withdrawal_credentials` should use `Bytes32` instead of `Bytes20`

Submitted by franfran

Severity: Informational

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

Summary: In the consensus-specs, the `withdrawal_credentials` field use the `Bytes32` type, and Teku uses the `Bytes20`. This doesn't pose a direct security impact but a risk given the drift from the specs.

Finding Description: Here is where the `withdrawal_credentials` field is defined in the `PendingDeposit` container: PendingDeposit.java#L38-L39.

The name of the field with the field is a bit misleading because it uses `Eth1Address` which extends `Bytes20` while a "withdrawal credential" here should be a `Bytes32` since the withdrawal credentials is defined as `PREFIX (1 byte) || 0 (11 bytes) || address (20 bytes)`. Only allowing the last 20 bytes would remove the prefix which might be critical for Ethereum, as it might indicate that the validator uses. The `WITHDRAWAL_REQUEST_TYPE` defined in EIP-7002 is `0x01`, the one defined in EIP-7251 is `0x02`, and so on... Fortunately, this function is never used in the codebase, excepted in a test.

The buggy `PendingDeposit` `withdrawal_credentials` is only read when calling `asInternalDeposit` which is called by `applyElectraFields` and called in `applyAdditionalFields` by `asInternalBeaconState` in the `shouldConvertToInternalObject` test.

This may have been caught by the `shouldConvertToInternalObject` test, but additional fields from the Electra blocks seems to not be generated random data. When launching the test and adding a print statement for the `getPendingDeposits()` function, the list is always empty.

```
diff --git a/data/serializer/src/main/java/tech/pegasys/teku/api/schema/electra/BeaconStateElectra.java
↪  b/data/serializer/src/main/java/tech/pegasys/teku/api/schema/electra/BeaconStateElectra.java
index 0317d461e6..608a9d6b09 100644
--- a/data/serializer/src/main/java/tech/pegasys/teku/api/schema/electra/BeaconStateElectra.java
+++ b/data/serializer/src/main/java/tech/pegasys/teku/api/schema/electra/BeaconStateElectra.java
@@ -181,6 +181,7 @@ public class BeaconStateElectra extends BeaconStateAltair {
     this.earliestExitEpoch = electra.getEarliestExitEpoch();
     this.consolidationBalanceToConsume = electra.getConsolidationBalanceToConsume();
     this.earliestConsolidationEpoch = electra.getEarliestConsolidationEpoch();
+    System.out.println(electra.getPendingDeposits());
     this.pendingDeposits = electra.getPendingDeposits().stream().map(PendingDeposit::new).toList();
     this.pendingPartialWithdrawals =
         electra.getPendingPartialWithdrawals().stream().map(PendingPartialWithdrawal::new).toList();
```

Impact Explanation: Low - This is a theoretical impact and nothing is directly under a potential attack.

Likelihood Explanation: Low - Can be misleading if the code is updated, if the prefix is ever changed from the EIP (unlikely!), or if a new request is forked from this EIP, will use some code that requires extra care because of the context.

Recommendation: Replace the `withdrawal_credentials` type from `Eth1Address` to `Bytes32`:

```
diff --git a/data/serializer/src/main/java/tech/pegasys/teku/api/schema/electra/PendingDeposit.java
↪  b/data/serializer/src/main/java/tech/pegasys/teku/api/schema/electra/PendingDeposit.java
index 3c14bfaf06..e4d39c9063 100644
--- a/data/serializer/src/main/java/tech/pegasys/teku/api/schema/electra/PendingDeposit.java
+++ b/data/serializer/src/main/java/tech/pegasys/teku/api/schema/electra/PendingDeposit.java
@@ -36,7 +36,7 @@ public class PendingDeposit {
   private final BLSPublicKey publicKey;

   @JsonProperty("withdrawal_credentials")
-  private final Eth1Address withdrawalCredentials;
+  private final Bytes32 withdrawalCredentials;

   @JsonProperty("amount")
   public final UInt64 amount;
@@ -49,7 +49,7 @@ public class PendingDeposit {

   public PendingDeposit(
       final @JsonProperty("pubkey") BLSPublicKey publicKey,
-      final @JsonProperty("withdrawal_credentials") Eth1Address withdrawalCredentials,
+      final @JsonProperty("withdrawal_credentials") Bytes32 withdrawalCredentials,
       final @JsonProperty("amount") UInt64 amount,
       final @JsonProperty("signature") BLSSignature signature,
       final @JsonProperty("slot") UInt64 slot) {
@@ -64,8 +64,8 @@ public class PendingDeposit {
       final tech.pegasys.teku.spec.datastructures.state.versions.electra.PendingDeposit
           internalPendingDeposit) {
     this.publicKey = internalPendingDeposit.getPublicKey();
-    this.withdrawalCredentials =
-        Eth1Address.fromBytes(internalPendingDeposit.getWithdrawalCredentials());
+    System.out.println(internalPendingDeposit.getWithdrawalCredentials());
+    this.withdrawalCredentials = internalPendingDeposit.getWithdrawalCredentials();
     this.amount = internalPendingDeposit.getAmount();
     this.signature = new BLSSignature(internalPendingDeposit.getSignature());
     this.slot = internalPendingDeposit.getSlot();
@@ -83,7 +83,7 @@ public class PendingDeposit {
         .create(
             new SszPublicKey(publicKey),
-            SszBytes32.of(Bytes32.wrap(withdrawalCredentials.getWrappedBytes())),
+            SszBytes32.of(withdrawalCredentials),
             SszUInt64.of(amount),
             new SszSignature(signature.asInternalBLSSignature()),
             SszUInt64.of(slot));
```

Additionally, correct the test so that it can catch missing withdrawal credentials prefixes mismatches.

3.2.2 Leftover from phase0 with the churn limit

Submitted by franfran

Severity: Informational

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

Summary: When a validator wants to exit, the exit queue epoch may be adjusted if the churn limit is exceeded. This is not in the consensus specs of electra, and is in fact a leftover of phase0. Fortunately, this is dead code and the state variables that are modified here are never used.

Finding Description: It happens in the `initiateValidatorExit` function. And here is the leftover piece of code that is a relic from phase0: BeaconStateMutatorsElectra.java#L122-L128:

```
if (validatorExitContext.getExitQueueChurn().compareTo(validatorExitContext.getChurnLimit())
    >= 0) {
  validatorExitContext.setExitQueueEpoch(validatorExitContext.getExitQueueEpoch().increment());
  validatorExitContext.setExitQueueChurn(UInt64.ONE);
} else {
  validatorExitContext.setExitQueueChurn(validatorExitContext.getExitQueueChurn().increment());
}
```

In fact, we can find the same branches in the overridden function with the same name in the BeaconStateMutators. It even seems to drift from the specs, because the exit queue churn is set to 1 if the limit is exceeded while it is incremented otherwise. But again, this doesn't seem to be even used. Here are the original specs of the phase0 where it's defined: phase0/beacon-chain.md#initiate_validator_exit. And the electra specs where it's missing: `electra/beacon-chain.md#modified-initiate_validator_exit`.

Impact Explanation: Low - The getters are never called in any place that is not in this code block.

Likelihood Explanation: Low.

Recommendation: Remove this leftover:

```
diff --git a/ethereum/spec/src/main/java/tech/pegasys/teku/spec/logic/versions/electra/helpers/BeaconStateMuta
↪  torsElectra.java
↪  b/ethereum/spec/src/main/java/tech/pegasys/teku/spec/logic/versions/electra/helpers/BeaconStateMutatorsEle
↪  ctra.java
index 7b9e7b2a1b..ff5b01ec25 100644
--- a/ethereum/spec/src/main/java/tech/pegasys/teku/spec/logic/versions/electra/helpers/BeaconStateMutatorsEle
↪  ctra.java
+++ b/ethereum/spec/src/main/java/tech/pegasys/teku/spec/logic/versions/electra/helpers/BeaconStateMutatorsEle
↪  ctra.java
@@ -117,16 +117,6 @@ public class BeaconStateMutatorsElectra extends BeaconStateMutatorsBellatrix {

     final MutableBeaconStateElectra stateElectra = MutableBeaconStateElectra.required(state);

-    final ValidatorExitContext validatorExitContext = validatorExitContextSupplier.get();
-
-    if (validatorExitContext.getExitQueueChurn().compareTo(validatorExitContext.getChurnLimit())
-        >= 0) {
-      validatorExitContext.setExitQueueEpoch(validatorExitContext.getExitQueueEpoch().increment());
-      validatorExitContext.setExitQueueChurn(UInt64.ONE);
-    } else {
-      validatorExitContext.setExitQueueChurn(validatorExitContext.getExitQueueChurn().increment());
-    }
-
     final UInt64 exitQueueEpoch =
         computeExitEpochAndUpdateChurn(stateElectra, validator.getEffectiveBalance());
```