Ethereum Pectra: Prysm

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 prysm implementation. The participants identified **1** issue in the following risk category:

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

3 Findings

3.1 High Risk

3.1.1 Prysm `process_pending_deposits` does not comply with beacon chain specs, leading to chain split

Submitted by zigtur

Severity: High Risk

Context: deposits.go#L405-L419

Summary: Due to an optimization in Prysm, this implementation does not follow the specifications when two pending deposits with the same previously unknown validator occur in the same slot. This client will ignore the second pending deposit if it has an invalid signature while it should process it. The other clients comply with the specs.

Specifications & Context: The beacon-chain specs show the new `process_pending_deposits` function for EIP-6110 and EIP-7152. This function follow the following specifications.

```
def process_pending_deposits(state: BeaconState) -> None:
    next_epoch = Epoch(get_current_epoch(state) + 1)
    available_for_processing = state.deposit_balance_to_consume + get_activation_exit_churn_limit(state)
    processed_amount = 0
    next_deposit_index = 0
    deposits_to_postpone = []
    is_churn_limit_reached = False
    finalized_slot = compute_start_slot_at_epoch(state.finalized_checkpoint.epoch)

    for deposit in state.pending_deposits:
        # Do not process deposit requests if Eth1 bridge deposits are not yet applied.
        if (
            # Is deposit request
            deposit.slot > GENESIS_SLOT and
            # There are pending Eth1 bridge deposits
            state.eth1_deposit_index < state.deposit_requests_start_index
        ):
            break

        # Check if deposit has been finalized, otherwise, stop processing.
        if deposit.slot > finalized_slot:
            break

        # Check if number of processed deposits has not reached the limit, otherwise, stop processing.
        if next_deposit_index >= MAX_PENDING_DEPOSITS_PER_EPOCH:
            break

        # Read validator state
        is_validator_exited = False
        is_validator_withdrawn = False
        validator_pubkeys = [v.pubkey for v in state.validators]
        if deposit.pubkey in validator_pubkeys:
            validator = state.validators[ValidatorIndex(validator_pubkeys.index(deposit.pubkey))]
            is_validator_exited = validator.exit_epoch < FAR_FUTURE_EPOCH
            is_validator_withdrawn = validator.withdrawable_epoch < next_epoch

        if is_validator_withdrawn:
            # Deposited balance will never become active. Increase balance but do not consume churn
            apply_pending_deposit(state, deposit)
        elif is_validator_exited:
            # Validator is exiting, postpone the deposit until after withdrawable epoch
            deposits_to_postpone.append(deposit)
        else:
            # Check if deposit fits in the churn, otherwise, do no more deposit processing in this epoch.
            is_churn_limit_reached = processed_amount + deposit.amount > available_for_processing
            if is_churn_limit_reached:
                break

            # Consume churn and apply deposit.
            processed_amount += deposit.amount
            apply_pending_deposit(state, deposit)

        # Regardless of how the deposit was handled, we move on in the queue.
        next_deposit_index += 1

    state.pending_deposits = state.pending_deposits[next_deposit_index:] + deposits_to_postpone

    # Accumulate churn only if the churn limit has been hit.
    if is_churn_limit_reached:
        state.deposit_balance_to_consume = available_for_processing - processed_amount
    else:
        state.deposit_balance_to_consume = Gwei(0)
```

This function calls `apply_pending_deposit` when the validator is not marked as withdrawn or exited. Then, `apply_pending_deposit` is defined as the following.

```
def apply_pending_deposit(state: BeaconState, deposit: PendingDeposit) -> None:
    """
    Applies ``deposit`` to the ``state``.
    """
    validator_pubkeys = [v.pubkey for v in state.validators]
    if deposit.pubkey not in validator_pubkeys:
        # Verify the deposit signature (proof of possession) which is not checked by the deposit contract
        if is_valid_deposit_signature(
            deposit.pubkey,
            deposit.withdrawal_credentials,
            deposit.amount,
            deposit.signature
        ):
            add_validator_to_registry(state, deposit.pubkey, deposit.withdrawal_credentials, deposit.amount)
    else:
        validator_index = ValidatorIndex(validator_pubkeys.index(deposit.pubkey))
        increase_balance(state, validator_index, deposit.amount)
```

We note that in `apply_pending_deposit`, the signature is verified if and only if the validator public key is not known. In this case, the public key is added to the registry through `add_validator_to_registry`.

Finally, `add_validator_to_registry` adds the validator to the `state.validators`.

```
def add_validator_to_registry(state: BeaconState,
                              pubkey: BLSPubkey,
                              withdrawal_credentials: Bytes32,
                              amount: uint64) -> None:
    index = get_index_for_new_validator(state)
    validator = get_validator_from_deposit(pubkey, withdrawal_credentials, amount)  # [Modified in
    ↪  Electra:EIP7251]
    set_or_append_list(state.validators, index, validator)
    # ...
```

Important: This means that when a validator makes two subsequent deposits that are processed in the same slot, only the signature of the first deposit must be verified.

Description: According to specifications, the pending deposits are supposed to be processed subsequently. In case a pending deposit with a new validator is found, `apply_pending_deposit` will verify the signature and add it to the registry. Any subsequent pending deposit with the same validator will be processed as validator is already known and will not verify the signature.

Due to an optimization in the `process_pending_deposits` implementation, Prysm derives from the specifications. When two deposits for the same validator previously unknown are processed in the same slot, Prysm will verify the signature of both deposits. Other clients will verify the signature of the first deposits only.

Impact: Any registering validator can slash Prysm validators through malicious transaction by sending two deposit transactions in the same block. As Prysm does not comply with the specifications, all validators using it will misbehave. This raises the splited percentage of validators to ~32% according to client-diversity.org. Combined with Besu vulnerability with v-parity on EIP-7702 transactions, the total splited percentage of validators will reach the 33% limit, making this a High vulnerability issue.

Code snippet:

Summary:

Client	Compliant with specs
Lighthouse	✓
Prysm	X
Teku	✓
Nimbus	✓
Grandine	✓
Lodestar	✓

- Lighthouse - compliant with specs: The Lighthouse implementation is interesting. Through the following comment, it details the edge-case that Prysm is vulnerable to:

> `new_validator_deposits` may contain multiple deposits with the same pubkey where the first deposit creates the new validator and the others are topups.

~~~
```rust
// Finish processing pending balance deposits if relevant.
//
// This *could* be reordered after `process_pending_consolidations` which pushes only to the end
// of the `pending_deposits` list. But we may as well preserve the write ordering used
// by the spec and do this first.
if let Some(ctxt) = pending_deposits_ctxt {
    let mut new_balance_deposits = List::try_from_iter(
        state
            .pending_deposits()?
            .iter_from(ctxt.next_deposit_index)?
            .cloned(),
    )?;
    for deposit in ctxt.deposits_to_postpone {
        new_balance_deposits.push(deposit)?;
    }
    *state.pending_deposits_mut()? = new_balance_deposits;
    *state.deposit_balance_to_consume_mut()? = ctxt.deposit_balance_to_consume;

    // `new_validator_deposits` may contain multiple deposits with the same pubkey where
    // the first deposit creates the new validator and the others are topups.
    // Each item in the vec is a (pubkey, validator_index)
    let mut added_validators = Vec::new();
    for deposit in ctxt.new_validator_deposits {
        let deposit_data = DepositData {
            pubkey: deposit.pubkey,
            withdrawal_credentials: deposit.withdrawal_credentials,
            amount: deposit.amount,
            signature: deposit.signature,
        };
        // Only check the signature if this is the first deposit for the validator,
        // following the logic from `apply_pending_deposit` in the spec.
        if let Some(validator_index) = state.get_validator_index(&deposit_data.pubkey)? {
            state
                .get_balance_mut(validator_index)?
                .safe_add_assign(deposit_data.amount)?;
        } else if is_valid_deposit_signature(&deposit_data, spec).is_ok() {
            // Apply the new deposit to the state
            let validator_index = state.add_validator_to_registry(
                deposit_data.pubkey,
                deposit_data.withdrawal_credentials,
                deposit_data.amount,
                spec,
            )?;
            added_validators.push((deposit_data.pubkey, validator_index));
        }
    }
```
~~~

- Prysm - not compliant with specs: Prysm implements the optimization through the `batchProcessNewPendingDeposits` function. However, this `batchProcessNewPendingDeposits` function expects the signature to be valid even if the validator is now known. To comply with the specs, it should not be the case if the validator was already added in a previous pending deposit.

```go
func ProcessPendingDeposits(ctx context.Context, st state.BeaconState, activeBalance primitives.Gwei)
↪   error {
    // ...

    for _, pendingDeposit := range pendingDeposits {
        // ...

        if isValidatorWithdrawn {
            // note: the validator will never be active, just increase the balance
            if err := helpers.IncreaseBalance(st, index, pendingDeposit.Amount); err != nil {
                return errors.Wrap(err, "could not increase balance")
            }
        } else if isValidatorExited {
            pendingDepositsToPostpone = append(pendingDepositsToPostpone, pendingDeposit)
        } else {
            isChurnLimitReached = primitives.Gwei(processedAmount+pendingDeposit.Amount) >
            ↪   availableForProcessing
            if isChurnLimitReached {
                break
            }
            processedAmount += pendingDeposit.Amount

            // note: the following code deviates from the spec in order to perform batch signature
            ↪   verification
            if found {
                if err := helpers.IncreaseBalance(st, index, pendingDeposit.Amount); err != nil {
                    return errors.Wrap(err, "could not increase balance")
                }
            } else { // @POC: OPTIMIZATION!!! collect all pending deposits and process them **after**
                // Collect deposit for batch signature verification
                pendingDepositsToBatchVerify = append(pendingDepositsToBatchVerify, pendingDeposit)
            }
        }

        // Regardless of how the pendingDeposit was handled, we move on in the queue.
        nextDepositIndex++
    }

    // Perform batch signature verification on pending deposits that require validator registration
    if err = batchProcessNewPendingDeposits(ctx, st, pendingDepositsToBatchVerify); err != nil { //
    ↪   @POC: verify pending deposits for which
        return errors.Wrap(err, "could not process pending deposits with new public keys")
    }
```

Then, `batchProcessNewPendingDeposits` will ignore any pending deposit with an invalid signature. This is incorrect as the validator may already be known and so the signature should be ignored in this case.

```go
// batchProcessNewPendingDeposits should only be used to process new deposits that require validator
↪   registration
func batchProcessNewPendingDeposits(ctx context.Context, state state.BeaconState, pendingDeposits
↪   []*ethpb.PendingDeposit) error {
    // ...
    // Process each deposit individually
    for _, pendingDeposit := range pendingDeposits {
        validSignature := allSignaturesVerified

        // If batch verification failed, check the individual deposit signature
        if !allSignaturesVerified {
            validSignature, err = blocks.IsValidDepositSignature(&ethpb.Deposit_Data{
                PublicKey:             bytesutil.SafeCopyBytes(pendingDeposit.PublicKey),
                WithdrawalCredentials: bytesutil.SafeCopyBytes(pendingDeposit.WithdrawalCredentials),
                Amount:                pendingDeposit.Amount,
                Signature:             bytesutil.SafeCopyBytes(pendingDeposit.Signature),
            })
            if err != nil {
                return errors.Wrap(err, "individual deposit signature verification failed")
            }
        }

        // Add validator to the registry if the signature is valid
        if validSignature {                              // @POC: in case signature is invalid, just ignore.
        ↪   THIS IS INCORRECT!!!
            _, has := state.ValidatorIndexByPubkey(bytesutil.ToBytes48(pendingDeposit.PublicKey))
            if has {
                index, _ := state.ValidatorIndexByPubkey(bytesutil.ToBytes48(pendingDeposit.PublicKey))
                if err := helpers.IncreaseBalance(state, index, pendingDeposit.Amount); err != nil {
                    return errors.Wrap(err, "could not increase balance")
                }
            } else {
                err = AddValidatorToRegistry(state, pendingDeposit.PublicKey,
                ↪  pendingDeposit.WithdrawalCredentials, pendingDeposit.Amount)
                if err != nil {
                    return errors.Wrap(err, "failed to add validator to registry")
                }
            }
        }
    }

    return nil
}
```

As we can see, the pending deposit will be ignored when the signature is invalid, even if the validator is now known.

Proof of Concept: Import the following diff in prysm:

```
diff --git a/beacon-chain/core/electra/deposits_test.go b/beacon-chain/core/electra/deposits_test.go
index 484c3297a..5008a7523 100644
--- a/beacon-chain/core/electra/deposits_test.go
+++ b/beacon-chain/core/electra/deposits_test.go
@@ -329,6 +329,51 @@ func TestBatchProcessNewPendingDeposits(t *testing.T) {
    })
 }

+func TestBatchProcessNewPendingDepositsZigtur_BothValidSigDeposits(t *testing.T) {
+   t.Run("invalid batch initiates correct individual validation", func(t *testing.T) {
+       st := stateWithActiveBalanceETH(t, 0)
+       require.Equal(t, 0, len(st.Validators()))
+       require.Equal(t, 0, len(st.Balances()))
+       sk, err := bls.RandKey()
+       require.NoError(t, err)
+       wc := make([]byte, 32)
+       wc[0] = params.BeaconConfig().ETH1AddressWithdrawalPrefixByte
+       wc[31] = byte(0)
+       validDep := stateTesting.GeneratePendingDeposit(t, sk, params.BeaconConfig().MinActivationBalance,
↪   bytesutil.ToBytes32(wc), 0)
+       // have a combination of valid and invalid deposits
+       deps := []*eth.PendingDeposit{validDep, validDep}
+       require.NoError(t, electra.BatchProcessNewPendingDeposits(context.Background(), st, deps))
+       // successfully added to register
+       require.Equal(t, 1, len(st.Validators()))
+       require.Equal(t, 1, len(st.Balances()))
+       require.Equal(t, params.BeaconConfig().MinActivationBalance*2, st.Balances()[0])
+   })
+}
+
+func TestBatchProcessNewPendingDepositsZigtur_OneValidOneInvalidSigDeposits(t *testing.T) {
+   t.Run("invalid batch initiates correct individual validation", func(t *testing.T) {
+       st := stateWithActiveBalanceETH(t, 0)
+       require.Equal(t, 0, len(st.Validators()))
+       require.Equal(t, 0, len(st.Balances()))
+       sk, err := bls.RandKey()
+       require.NoError(t, err)
+       wc := make([]byte, 32)
+       wc[0] = params.BeaconConfig().ETH1AddressWithdrawalPrefixByte
+       wc[31] = byte(0)
+       validDep := stateTesting.GeneratePendingDeposit(t, sk, params.BeaconConfig().MinActivationBalance,
↪   bytesutil.ToBytes32(wc), 0)
+       validDep2 := stateTesting.GeneratePendingDeposit(t, sk, params.BeaconConfig().MinActivationBalance,
↪   bytesutil.ToBytes32(wc), 0)
+       validDep2.Signature = make([]byte, 96)
+       // have a combination of valid and invalid deposits
+       deps := []*eth.PendingDeposit{validDep, validDep2}
+       require.NoError(t, electra.BatchProcessNewPendingDeposits(context.Background(), st, deps))
+       // successfully added to register
+       require.Equal(t, 1, len(st.Validators()))
+       require.Equal(t, 1, len(st.Balances()))
+       // @POC: here, validator balance is only 32 ETH due to the invalid signature
+       require.Equal(t, params.BeaconConfig().MinActivationBalance, st.Balances()[0])
+   })
```

```
+}
+
 func TestProcessDepositRequests(t *testing.T) {
    st, _ := util.DeterministicGenesisStateElectra(t, 1)
    sk, err := bls.RandKey()
```

Then, run the following command:

```
go test -timeout 30s -run ^TestBatchProcessNewPendingDepositsZigtur
↪  github.com/prysmaticlabs/prysm/v5/beacon-chain/core/electra
```

Results: We can see that when the signature is invalid, the balance of the validator is not updated.

Proof of Concept in Go format:

```go
func TestBatchProcessNewPendingDepositsZigtur_BothValidSigDeposits(t *testing.T) {
    t.Run("invalid batch initiates correct individual validation", func(t *testing.T) {
        st := stateWithActiveBalanceETH(t, 0)
        require.Equal(t, 0, len(st.Validators()))
        require.Equal(t, 0, len(st.Balances()))
        sk, err := bls.RandKey()
        require.NoError(t, err)
        wc := make([]byte, 32)
        wc[0] = params.BeaconConfig().ETH1AddressWithdrawalPrefixByte
        wc[31] = byte(0)
        validDep := stateTesting.GeneratePendingDeposit(t, sk, params.BeaconConfig().MinActivationBalance,
        ↪  bytesutil.ToBytes32(wc), 0)
        // have a combination of valid and invalid deposits
        deps := []*eth.PendingDeposit{validDep, validDep}
        require.NoError(t, electra.BatchProcessNewPendingDeposits(context.Background(), st, deps))
        // successfully added to register
        require.Equal(t, 1, len(st.Validators()))
        require.Equal(t, 1, len(st.Balances()))
        require.Equal(t, params.BeaconConfig().MinActivationBalance*2, st.Balances()[0])
    })
}

func TestBatchProcessNewPendingDepositsZigtur_OneValidOneInvalidSigDeposits(t *testing.T) {
    t.Run("invalid batch initiates correct individual validation", func(t *testing.T) {
        st := stateWithActiveBalanceETH(t, 0)
        require.Equal(t, 0, len(st.Validators()))
        require.Equal(t, 0, len(st.Balances()))
        sk, err := bls.RandKey()
        require.NoError(t, err)
        wc := make([]byte, 32)
        wc[0] = params.BeaconConfig().ETH1AddressWithdrawalPrefixByte
        wc[31] = byte(0)
        validDep := stateTesting.GeneratePendingDeposit(t, sk, params.BeaconConfig().MinActivationBalance,
        ↪  bytesutil.ToBytes32(wc), 0)
        validDep2 := stateTesting.GeneratePendingDeposit(t, sk, params.BeaconConfig().MinActivationBalance,
        ↪  bytesutil.ToBytes32(wc), 0)
        validDep2.Signature = make([]byte, 96)
        // have a combination of valid and invalid deposits
        deps := []*eth.PendingDeposit{validDep, validDep2}
        require.NoError(t, electra.BatchProcessNewPendingDeposits(context.Background(), st, deps))
        // successfully added to register
        require.Equal(t, 1, len(st.Validators()))
        require.Equal(t, 1, len(st.Balances()))
        // @POC: here, validator balance is only 32 ETH due to the invalid signature
        require.Equal(t, params.BeaconConfig().MinActivationBalance, st.Balances()[0])
    })
}
```

Recommendation: Prysm `batchProcessNewPendingDeposits` function must implement the `validSignature` check only if the validator has not been found. An implementation similar to the following should be implemented:

```
diff --git a/beacon-chain/core/electra/deposits.go b/beacon-chain/core/electra/deposits.go
index c0baaf3cf..7c6c8861c 100644
--- a/beacon-chain/core/electra/deposits.go
+++ b/beacon-chain/core/electra/deposits.go
@@ -403,14 +403,14 @@ func batchProcessNewPendingDeposits(ctx context.Context, state state.BeaconState
                }

                // Add validator to the registry if the signature is valid
-               if validSignature {
-                       _, has := state.ValidatorIndexByPubkey(bytesutil.ToBytes48(pendingDeposit.PublicKey))
-                       if has {
-                               index, _ :=
↪   state.ValidatorIndexByPubkey(bytesutil.ToBytes48(pendingDeposit.PublicKey))
-                               if err := helpers.IncreaseBalance(state, index, pendingDeposit.Amount); err !=
↪   nil {
-                                       return errors.Wrap(err, "could not increase balance")
-                               }
-                       } else {
+               _, has := state.ValidatorIndexByPubkey(bytesutil.ToBytes48(pendingDeposit.PublicKey))
+               if has {
+                       index, _ := state.ValidatorIndexByPubkey(bytesutil.ToBytes48(pendingDeposit.PublicKey))
+                       if err := helpers.IncreaseBalance(state, index, pendingDeposit.Amount); err != nil {
+                               return errors.Wrap(err, "could not increase balance")
+                       }
+               } else {
+                       if validSignature {
                                err = AddValidatorToRegistry(state, pendingDeposit.PublicKey,
                                ↪   pendingDeposit.WithdrawalCredentials, pendingDeposit.Amount)
                                if err != nil {
                                        return errors.Wrap(err, "failed to add validator to registry")
```