Coinbase: Precompiles Src

Security Review

Cantina Managed review by:
J4X, Lead Security Researcher
Slowfi, Lead Security Researcher
Cryptara, Security Researcher
Jay, Security Researcher

August 14, 2026

Contents

1 Introduction **2**
1.1 About Cantina . . . 2
1.2 Disclaimer . . . 2
1.3 Risk assessment . . . 2
1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
3.1 High Risk . . . 4
3.1.1 Oversized crypto precompile inputs abort EVM . . . 4
3.2 Low Risk . . . 6
3.2.1 Successful precompile results drop gas refunds . . . 6
3.2.2 Known selector decode failures are misclassified . . . 7
3.2.3 Activation admin zero address allows zero sender deposits . . . 7
3.2.4 B20 factory address hashing is unmetered . . . 8
3.2.5 B20 factory prefunded address creation is undercharged . . . 8
3.2.6 `nonpayable` ABI is not enforced . . . 9
3.2.7 Announce wraps system errors as internal call failures . . . 9
3.2.8 Native precompile calldata fee uses a misleading constant name . . . 10
3.2.9 `announce` internal calls omit the per-call EVM execution overhead . . . 11
3.2.10 Inactive policy registry masks dispatch errors . . . 11
3.2.11 Policy registry decodes account batches before enforcing the size cap . . . 12
3.3 Informational . . . 13
3.3.1 Repeated deactivation returns wrong ABI error . . . 13
3.3.2 Activation admin is not consensus anchored . . . 13
3.3.3 Bool storage decodes any non zero word as true . . . 15
3.3.4 `isB20Initialized` accepts any prefix account with code . . . 15
3.3.5 `set_code` ignores static call context . . . 16
3.3.6 Activation bytecode marker persists after deactivation . . . 16
3.3.7 B20 variant decimals helper is misleading . . . 17
3.3.8 B20 factory accepts dirty ABI words . . . 17
3.3.9 B20 asset zero multiplier normalizes to WAD . . . 18
3.3.10 Dead `beryl()` constructor and missing `cobalt()`: fork dispatch collapses Azul / Beryl / Cobalt into `azul()` . . . 18
3.3.11 `isAnnouncementActive()` is not implemented by the B-20 Asset precompile . . . 19
3.3.12 Factory bootstrap bypasses transfer policies but still enforces the `MintReceiver` policy . . . 20
3.3.13 EIP-2612 `permit` accepts non-canonical (high-s) signatures . . . 21
3.3.14 B20 lookup exposes uninitialized token addresses . . . 23
3.3.15 Beryl lookup install overwrites existing dynamic lookup . . . 23
3.3.16 Installed precompiles map can go stale after spec change . . . 24
3.3.17 Stablecoin currency validation runs after token existence check . . . 24

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

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it 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 that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Base is a secure, low-cost, builder-friendly Ethereum L2 built to bring the next billion users onchain.

From Jun 1st to Jun 9th the Cantina team conducted a review of base on commit hash 6ee3da63. The team identified a total of **29** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	0	0	0
Low Risk	11	10	1
Gas Optimizations	0	0	0
Informational	17	10	7
Total	**29**	**21**	**8**

3 Findings

3.1 High Risk

3.1.1 Oversized crypto precompile inputs abort EVM

Severity: High Risk.

Context: bn254_pair.rs#L14-L17, bn254_pair.rs#L41-L43.

Description: The Base size guards for BN254 pairing and BLS12 381 precompiles return `PrecompileError::Fatal` when calldata is one byte above the configured limit. Revm defines fatal precompile errors as unrecoverable provider errors, while invalid input length is supposed to be reported as a normal precompile halt.

A user can submit a transaction that calls one of these precompile addresses with oversized calldata. Instead of producing a deterministic failed call result that spends gas, the precompile provider returns an EVM custom error before a call outcome is created. In execution pipelines that propagate fatal precompile errors as provider failures, an ordinary transaction input can fail block execution instead of behaving like a normal failed call.

Proof Of Concept:

```
use base_common_precompiles::{
    GRANITE_MAX_INPUT_SIZE, ISTHMUS_G1_MSM_MAX_INPUT_SIZE, ISTHMUS_G2_MSM_MAX_INPUT_SIZE,
    ISTHMUS_PAIRING_MAX_INPUT_SIZE, JOVIAN_G1_MSM_MAX_INPUT_SIZE,
    ↪  JOVIAN_G2_MSM_MAX_INPUT_SIZE,
    JOVIAN_MAX_INPUT_SIZE, JOVIAN_PAIRING_MAX_INPUT_SIZE, run_isthmus_g1_msm,
    ↪  run_pair_granite,
};

const BN254_PAIR_ELEMENT_LEN: usize = 192; // G1(64) + G2(128)

// PROOF 1: every Base cap matches op-geth params/protocol_params.go exactly, and the BLS
↪  caps
// are whole-element multiples (160/288/384). Verified vs op-geth (optimism branch).
#[test]
fn poc_crypto_constants_match_opgeth() {
    assert_eq!(GRANITE_MAX_INPUT_SIZE, 112687); // Bn256PairingMaxInputSizeGranite
    assert_eq!(JOVIAN_MAX_INPUT_SIZE, 81984); // Bn256PairingMaxInputSizeJovian
    assert_eq!(ISTHMUS_G1_MSM_MAX_INPUT_SIZE, 513760); //
    ↪  Bls12381G1MulMaxInputSizeIsthmus
    assert_eq!(JOVIAN_G1_MSM_MAX_INPUT_SIZE, 288960); // Bls12381G1MulMaxInputSizeJovian
    assert_eq!(ISTHMUS_G2_MSM_MAX_INPUT_SIZE, 488448); //
    ↪  Bls12381G2MulMaxInputSizeIsthmus
    assert_eq!(JOVIAN_G2_MSM_MAX_INPUT_SIZE, 278784); // Bls12381G2MulMaxInputSizeJovian
    assert_eq!(ISTHMUS_PAIRING_MAX_INPUT_SIZE, 235008); //
    ↪  Bls12381PairingMaxInputSizeIsthmus
    assert_eq!(JOVIAN_PAIRING_MAX_INPUT_SIZE, 156672); //
    ↪  Bls12381PairingMaxInputSizeJovian
    // whole-element multiples
    assert_eq!(ISTHMUS_G1_MSM_MAX_INPUT_SIZE % 160, 0);
    assert_eq!(JOVIAN_G1_MSM_MAX_INPUT_SIZE % 160, 0);
    assert_eq!(ISTHMUS_G2_MSM_MAX_INPUT_SIZE % 288, 0);
    assert_eq!(ISTHMUS_PAIRING_MAX_INPUT_SIZE % 384, 0);
    assert_eq!(JOVIAN_MAX_INPUT_SIZE % BN254_PAIR_ELEMENT_LEN, 0); // 427 pairs
    println!("all 8 caps match op-geth; off-by-one boundary is `>` (== cap allowed),
    ↪  matching op-geth `len(input) > max`");
}

// PROOF 2 (the finding): oversize input takes the FATAL path (aborts the whole tx)
↪  whereas
// op-geth returns a standard precompile error (`return nil, errBadPairingInputSize`)
↪  that only
```

```rust
// fails the CALL (gas consumed, tx still valid). Demonstrated by contrast: 17 bytes flip
↪  a
// graceful Ok(halt) into an Err(fatal).
#[test]
fn poc_bn254_oversize_is_fatal_vs_graceful_halt_below_cap() {
    // 587 pairs = 112704 bytes > 112687 cap, and a *valid* pair length (multiple of
    ↪  192).
    let over = vec![0u8; 587 * BN254_PAIR_ELEMENT_LEN];
    let r_over = run_pair_granite(&over, u64::MAX, 0);
    assert!(
        matches!(&r_over, Err(e) if e.is_fatal()),
        "EXPECTED divergence: oversize -> Err(PrecompileError::Fatal) which revm
        ↪  propagates as \
         EVMError::Custom (aborts the whole transaction). Got: {r_over:?}"
    );

    // 112687 bytes == cap (passes the length guard), but not a multiple of 192, so
    ↪  revm's
    // run_pair returns a GRACEFUL halt wrapped in Ok -> the call fails, the tx still
    ↪  commits.
    let at = vec![0u8; GRANITE_MAX_INPUT_SIZE];
    let r_at = run_pair_granite(&at, u64::MAX, 0);
    assert!(
        matches!(&r_at, Ok(o) if o.is_halt()),
        "at-cap malformed input should fail GRACEFULLY (Ok halt, tx continues). Got:
        ↪  {r_at:?}"
    );

    println!(
        "bn254 Granite: {} bytes -> Ok(halt) [graceful call-failure, op-geth-equivalent];
        ↪  \
         {} bytes -> Err(fatal) [aborts whole tx, DIVERGES from op-geth]",
        GRANITE_MAX_INPUT_SIZE,
        587 * BN254_PAIR_ELEMENT_LEN
    );
}

// PROOF 3: the same fatal-on-oversize mechanism applies to the BLS precompiles.
#[test]
fn poc_bls_g1msm_oversize_is_fatal() {
    let over = vec![0u8; ISTHMUS_G1_MSM_MAX_INPUT_SIZE + 1];
    let r = run_isthmus_g1_msm(&over, u64::MAX, 0);
    assert!(
        matches!(&r, Err(e) if e.is_fatal()),
        "BLS G1MSM oversize -> Err(PrecompileError::Fatal) (aborts tx) where op-geth
        ↪  returns a \
         standard precompile error that only fails the call. Got: {r:?}"
    );
    println!("BLS G1MSM: {} bytes -> Err(fatal)", ISTHMUS_G1_MSM_MAX_INPUT_SIZE + 1);
}
```

```
all 8 caps match op-geth; boundary `>` matches op-geth `len(input) > max`
bn254 Granite: 112687 bytes -> Ok(halt)  [graceful call-failure,
↪  op-geth/op-revm-equivalent]
               112704 bytes -> Err(fatal) [aborts whole tx, DIVERGES from
               ↪  op-geth/op-revm]
BLS G1MSM:     513761 bytes -> Err(fatal)
```

The bn254 case is the sharpest demonstration: a 112,687-byte input (≤ cap, not a multiple of 192) fails gracefully as `Ok(halt)`, while a 112,704-byte input — 17 bytes larger — `becomes Err(fatal)` and aborts the entire transaction. Both reference clients treat both inputs as ordinary call-failures.

Recommendation: Return a normal precompile halt for input size violations, such as `PrecompileOutput::halt(PrecompileHalt::other_static(...), reservoir)` or a dedicated input length halt, and reserve

`PrecompileError::Fatal` for internal errors that cannot be caused by calldata.

References:

- Op-revm (canonical upstream, Base's fork source): `ethereum-optimism/optimism` — precompiles.rs#L209-L211.
- Op-geth: `ethereum-optimism/op-geth` — contracts.go#L940-L942 (`bn256PairingGranite.Run` , `bls12381*Isthmus/Jovian.Run`), `core/vm/evm.go` (precompile-error call-site), `params/protocol_params.go` .
- Revm: `revm-precompile 34.0.0/src/interface.rs:26,378-381,529-548` , `bn254.rs:191-192` , `bls12_381/{g1_msm,g2_msm,pairing}.rs` ; `revm-handler 18.1.0/src/frame.rs:203` ; `MIGRATION_GUIDE.md` (PrecompileError restructure, PR 3496/3502).

Coinbase: Fixed in commit 70f6cd23.

Cantina Managed: Fix verified.

3.2 Low Risk

3.2.1 Successful precompile results drop gas refunds

Severity: Low Risk.

Context: dispatch.rs#L16-L28.

Description: The shared `IntoPrecompileResult` success path builds `PrecompileOutput` with `PrecompileOutput::new` and never copies the refund counter into `out.gas_refunded` . Dispatchers for activation, policy, factory, and B20 token calls use this helper after executing storage operations, so successful calls that clear storage can lose their EIP 3529 refund information.

The activation registry shows the issue directly. `ActivationRegistry.dispatch` calls `deactivate(bytes32)` , which deletes the feature mapping slot when the feature is active. That delete reaches the EVM backed `sstore` path, which records the storage refund through `refund_gas(...)` . The refund remains visible through `StorageCtx::gas_refunded()` , and the crate already has `StorageCtx::success_output(...)` to copy it into `PrecompileOutput::gas_refunded` . However, the dispatch path returns through `IntoPrecompileResult` , whose success branch ignores the refund value.

The policy registry has the same dispatcher level issue. `PolicyRegistryStorage::dispatch` returns successful calls through `IntoPrecompileResult` , while policy admin and membership operations can delete storage slots. `finalizeUpdateAdmin` , `renounceAdmin` , `updateAllowlist(false, accounts)` , and `updateBlocklist(false, accounts)` can all reach `delete()` paths that record storage refunds in the provider, but the returned `PrecompileOutput` still leaves `gas_refunded` at zero.

Users are overcharged for successful refund eligible precompile operations, and native precompile gas accounting diverges from normal EVM `SSTORE` accounting. This does not change state correctness, but it drops refunds that the EVM frame handler expects to receive through `PrecompileOutput::gas_refunded` .

Recommendation: Change the conversion helper to accept the refund amount and populate `PrecompileOutput::gas_refunded` , or replace successful dispatch conversions with `StorageCtx::success_output` or `StorageCtx::abi_success` . Add regression tests for successful refund eligible storage clears through the activation registry and the other storage backed dispatchers.

Coinbase: Fixed in PR 3258.

Cantina Managed: Fix verified.

3.2.2 Known selector decode failures are misclassified

Severity: Low Risk.

Context: dispatch.rs#L35.

Description: The shared `decode_precompile_call!` macro extracts the first four bytes, then calls `SolInterface::abi_decode(calldata)`. If the selector belongs to the interface but the calldata body is malformed, the ABI decoder fails, but the macro maps every decode error to `BasePrecompileError::UnknownFunctionSelector(selector)`.

The error type already has a distinct `AbiDecodeFailed { selector, error }` variant for the case where the selector is known but its arguments fail ABI decoding. Activation registry dispatch exposes the shared behavior through `inner`, so malformed calls to known activation selectors are reported as unknown selectors instead of ABI decode failures.

Malformed calldata with a valid activation registry selector is reported with misleading revert diagnostics. Clients, tests, and monitoring that distinguish unknown functions from malformed arguments cannot rely on the advertised `AbiDecodeFailed` path for activation registry calls, and the same shared macro behavior can affect the other precompile dispatchers that use it.

Recommendation: Change `decode_precompile_call!` to first determine whether the selector belongs to the target interface, then return `AbiDecodeFailed { selector, error }` when the selector is known but full calldata decoding fails. Keep `UnknownFunctionSelector` for short calldata and selectors that are not part of the interface. Add ABI dispatch tests with a known selector and malformed body for the activation registry and one shared macro caller.

Coinbase: Fixed in PR 3338.

Cantina Managed: Fix verified.

3.2.3 Activation admin zero address allows zero sender deposits

Severity: Low Risk.

Context: storage.rs#L62-L67.

Description: `ActivationRegistryStorage::admin` returns `Address::ZERO` for both `None` and `Some(Address::ZERO)`, but mutations treat those two states differently. `None` rejects every caller because there is no configured admin, while `Some(Address::ZERO)` configures the zero address as the admin and authorizes any call that reaches the precompile with `msg.sender == Address::ZERO`.

That ambiguity means `admin() == address(0)` does not tell callers whether activation mutations are disabled or whether the zero address is the configured admin. The activation registry accepts `Address::ZERO` as a configured admin, and the activation proof of concept suite shows that zero sender deposit transactions can reach the activation precompile with `msg.sender == Address::ZERO` and successfully toggle activation state.

The practical effect is that a misconfigured chain with `activation_admin_address == Some(Address::ZERO)` does not merely brick activation for ordinary users. It creates a backdoor for any execution path that carries a zero recovered sender, including the deposit transaction path exercised by the repo's own proof of concept. In that state, an attacker can activate or deactivate the registry at will.

Recommendation: Reject `Address::ZERO` for the activation admin everywhere it is configured or loaded. If zero sender deposit transactions must remain supported, add an explicit guard in the activation precompile that rejects zero callers regardless of admin configuration. Add a regression test for the zero sender deposit case.

Coinbase: Fixed in PR 3344.

Cantina Managed: Fix verified.

3.2.4 B20 factory address hashing is unmetered

Severity: Low Risk.

Context: variant.rs#L138.

Description: The B20 factory computes deterministic token addresses by hashing `(creator, salt)` with `alloy_primitives::keccak256` directly. This helper has no access to `StorageCtx`, so it cannot charge the configured keccak gas even when called from the precompile dispatch path.

Both address helpers have the same issue. `createB20` reaches `B20Variant::compute_address` while creating the token, and `getB20Address` reaches the same helper while serving the public address prediction view. `compute_address_for_discriminant` also performs the same direct hash for callers that compute an address from a raw variant byte.

Each factory address computation undercharges one small keccak operation. The effect is minor per call, but it makes native precompile gas accounting diverge from the EVM cost model and gives repeated factory address prediction or creation calls a tiny unmetered CPU cost.

Recommendation: Route factory address hashing through a gas aware helper that receives `StorageCtx`, or deduct the keccak cost in the factory dispatch and creation paths before computing the address. Keep pure helpers only for tests or offchain utilities where no gas accounting is expected. Add a regression test that compares gas used by `getB20Address` or `createB20` before and after the address hash is charged.

Coinbase: Fixed in PR 3369.

Cantina Managed: Fix verified.

3.2.5 B20 factory prefunded address creation is undercharged

Severity: Low Risk.

Context: storage.rs#L61-L66.

Description: `createB20` only rejects a predicted token address when the account already has non empty code. An account that already holds ETH but has no code passes this collision check, so the factory can still deploy the B20 marker bytecode to that predicted address.

The later `set_code` metering path uses a different test. It charges the CREATE equivalent gas only when `AccountInfo::is_empty()` is true. A prefunded no code account is not empty because it has a balance, so `set_code` still writes the marker bytecode but skips the new account branch that charges `create_cost()`, the stored code hash keccak cost, and state creation gas.

A caller can prefund a predicted B20 token address, then call `createB20` and deploy the token marker bytecode with less gas than an empty account creation would charge. With the current one byte marker code, the concrete execution gas undercharge is about `32036` gas, before considering any state gas pricing. This does not create a large economic saving today, but it makes factory creation metering depend on whether the address was prefunded.

Recommendation: Use the same account emptiness rule for the factory collision check and the `set_code` creation charge, or explicitly charge the CREATE equivalent costs when writing code to an account that has balance but no code. Add a regression test that prefunds a predicted B20 address, creates the token, and asserts creation gas is charged the same way as deployment to a fully empty account.

Coinbase: Fixed in PR 3371.

Cantina Managed: Fix verified.

3.2.6 `nonpayable` ABI is not enforced

Severity: Low Risk.

Context: abi.rs#L83-L88.

Description: The B20 factory ABI declares `createB20` as a nonpayable function, and its view selectors are also not payable. The native precompile wrapper does not enforce that ABI property. It only rejects delegate style calls by checking `is_direct_call()`, then constructs the storage provider and dispatches calldata.

`PrecompileInput` carries the call value, but `EvmPrecompileStorageProvider::new` discards it with `..` and exposes no value check to factory dispatch. As a result, a low level `CALL` with non zero value to any successful factory selector can complete successfully even though the ABI says the call is not payable.

Because EVM call value is transferred before successful precompile execution, the sent ETH can end up at `B20FactoryStorage::ADDRESS`. The factory has no withdrawal path, so that value is stuck.

Callers can accidentally or deliberately strand ETH at the B20 factory precompile address by sending value with a successful `createB20`, `getB20Address`, `isB20`, or `isB20Initialized` call. This does not let an attacker steal funds from other users, but it violates ABI expectations and can permanently lock funds sent with these calls.

Recommendation: Reject non zero call value for every nonpayable native precompile selector before dispatch succeeds. The most robust fix is to add a value guard in the shared `base_precompile!` wrapper or provider boundary, while still allowing future explicitly payable precompiles to opt in. Add regression tests for `CALL{value: X}` to `createB20` and one factory view selector, asserting the call reverts and no value remains at the factory address.

Coinbase: Fixed in PR 3362.

Cantina Managed: Fix verified.

3.2.7 Announce wraps system errors as internal call failures

Severity: Low Risk.

Context: dispatch.rs#L423-L426.

Description: `B20AssetToken::announce` executes each `internalCalls` entry by redispatching it through `inner_with_privilege`. If the inner call returns any error, the current code discards the original error and wraps it as `IB20Asset::InternalCallFailed`.

That wrapping is appropriate for ordinary inner reverts, but it also catches system errors such as `OutOfGas`, `Fatal`, `Panic`, and `SlotOverflow`. Those error variants have special semantics when converted to a `PrecompileResult`: `OutOfGas` becomes an out of gas halt, and `Fatal` or `SlotOverflow` becomes a fatal precompile error. By replacing them with a normal ABI revert, `announce` can undercharge gas and hide host or storage failures from the EVM.

The factory init call mapper already handles this distinction by preserving `err.is_system_error()` and only wrapping non system failures. `announce` should follow the same pattern unless it is changed to implement a real inner frame for the documented self dispatch model.

An out of gas or fatal storage error inside an announcement internal call can be reported as a normal `InternalCallFailed(bytes)` revert. This changes error semantics, can avoid the gas behavior expected for out of gas, and can obscure unrecoverable host failures that should abort the precompile instead of appearing as user level call failures.

Recommendation: Only wrap ordinary inner reverts as `InternalCallFailed`. Propagate `err.is_system_error()` unchanged, and preserve meaningful inner revert data where appropriate. Longer term, implement a real inner frame for the documented self dispatch model so gas, revert, and fatal error semantics match EVM call behavior. Add regression tests that inject an inner `OutOfGas` and an inner `Fatal` and assert they are not converted to `InternalCallFailed`.

Coinbase: Fixed in PR 3360.

Cantina Managed: Fix verified.

3.2.8 Native precompile calldata fee uses a misleading constant name

Severity: Low Risk.

Context: macros.rs#L52-L56.

Description: Every native B-20 precompile dispatch charges a per-word fee on its calldata via `deduct_calldata_cost!`, which is invoked at the top of each variant's dispatch (e.g. `b20_asset`, `b20_stablecoin` token dispatch). The fee is:

```
const G_SHA3WORD: u64 = 6;
let calldata_cost = calldata_len.div_ceil(32).saturating_mul(G_SHA3WORD as usize) as
↪  u64; // ceil(len/32) * 6
```

The native precompiles emulate the cost a Solidity predeploy would incur to ingest its calldata. The closest EVM-equivalent for that is the copy/load family, not keccak. Per the evm-opcodes gas reference (gas.md#a3-copy-operations), `CALLDATACOPY` costs `G_verylow (3) + G_copy (3) * ceil(len/32)` plus memory expansion, i.e. a marginal 3 gas per word, and `CALLDATALOAD` is a flat `G_verylow = 3`. The 6-per-word figure used here is therefore roughly 2x the natural EVM per-word cost, and it is borrowed from the keccak256 word constant (`G_SHA3WORD`), which is unrelated to reading calldata since nothing is hashed at this point. The only construction that naturally lands on 6/word is treating it as a flat simplification of copy + memory, `G_copy (3) + G_memory (3) = 6`.

Two concerns:

1. **Pricing.** If faithful EVM-equivalence is the goal, the per-word charge looks like it should be `G_copy = 3` per word (plus memory expansion for dynamic types), so the current charge over-prices input by about 2x. If the 6 is intentional (e.g. a bundled copy + memory simplification, a deliberately conservative margin, or alignment with a base-std spec), that intent is not recoverable from the code.
2. **Clarity / drift.** The constant name `G_SHA3WORD` is misleading for a calldata fee (nothing is being hashed), and there is no test pinning the value and no comment explaining the derivation. This matters for more than readability: the charged amount feeds `gas_used`, which determines the transaction's `gasUsed` and the block's receipts/cumulative-gas commitment, so the per-word value must be identical across every Base execution client. A schedule that lives only as an unexplained `6` with no pinned test can silently drift or diverge from another client's implementation and become a consensus mismatch.

Recommendation:

- Document the intent: add a comment stating exactly what the 6/word is meant to represent and citing its source (EVM-equivalent derivation, base-std spec, or a deliberate margin).
- Decide the value against that intent. For EVM-equivalence, use `G_copy = 3` per word (and account for memory expansion on dynamic ABI data); if the intent is a bundled `copy (3) + memory (3)` simplification, keep 6 but say so explicitly in the comment.
- Rename the constant from `G_SHA3WORD` to something accurate for a calldata fee (e.g. `CALLDATA_WORD_GAS` or `PRECOMPILE_CALLDATA_WORD_GAS`).
- Add a unit test that pins the exact charged value for a known calldata length, so the per-word schedule cannot drift unnoticed (it is part of the gas-used / receipts commitment and must match across all Base clients).

Coinbase: Fixed in PR 3359.

Cantina Managed: Fix verified.

3.2.9 `announce` internal calls omit the per-call EVM execution overhead

Severity: Low Risk.

Context: dispatch.rs#L413-L426.

Description: In the base-std reference, `announce` runs each entry of `internalCalls` as a real EVM message call back into the contract:

```
// MockB20Asset.sol:131
for (uint256 i = 0; i < internalCalls.length; i++) {
    _checkSelector(internalCalls[i]);
    (bool success,) = address(this).delegatecall(internalCalls[i]);
    // ...
}
```

Each `delegatecall` carries real EVM cost: the DELEGATECALL opcode itself (warm address access, ~100 gas under EIP-2929), memory expansion for the input/output regions, and the callee frame's own opcode execution (including its `CALLDATALOAD` / `CALLDATACOPY` reads of the sub-call arguments).

The native precompile instead replays each internal call through a plain in-process function call, `inner_with_privilege`, which performs no EVM call at all:

```
// b20_asset/dispatch.rs:423
self.inner_with_privilege(ctx, call_bytes, privileged) // no DELEGATECALL cost, no
↪  per-call overhead
```

So an `announce` with N internal calls under-charges relative to the equivalent Solidity flow by approximately `N × (DELEGATECALL overhead + callee read opcodes)`. The only gas the native version charges for the batch is the single top-level `deduct_calldata_cost!` (taken once on the outer `announce` calldata) plus the storage work each inner operation performs.

Recommendation: Decide and document the intended gas model for `announce` internal calls, then pin it:

- If the goal is approximate EVM/Solidity-equivalent gas, add a per-internal-call overhead charge that mirrors the EVM `DELEGATECALL` cost (a flat per-call constant, plus the callee execution that is already metered), so `announce(N)` is not materially cheaper than performing the operations directly. Do not add a per-word calldata re-charge, which would double-count.
- If the cheaper batched cost is intentional, state that explicitly at `dispatch.rs:412` and in the B-20 Asset docs, so the asymmetry between batched and direct execution is a documented property rather than an accident.
- Either way, add a test that pins the gas charged by `announce` for a fixed N and fixed inner operations, since the value is part of the receipts/gas-used commitment and must be identical across all Base execution clients. Confirm the chosen model matches any other client's native B20 `announce` accounting.

Coinbase: Fixed in PR 3363.

Cantina Managed: Fix verified.

3.2.10 Inactive policy registry masks dispatch errors

Severity: Low Risk.

Context: dispatch.rs#L18-L28.

Description: `PolicyRegistryStorage::dispatch` decides whether to enforce activation by checking only the first four calldata bytes. If the selector matches one of the view functions, the call bypasses the activation gate and enters ABI dispatch. Every other selector goes through `ActivationRegistryStorage::ensure_activated` before the dispatcher decodes the full calldata.

When the policy registry feature is inactive, this ordering masks selector and ABI errors for non view calls. An unknown selector, short calldata, or a known write selector with malformed arguments returns `FeatureNotActivated` before `inner` can classify it as `UnknownFunctionSelector` or an ABI decode failure. The same malformed write selector can therefore produce a different error depending only on whether the feature is active.

View selectors are treated differently because the selector only check sends them to `inner` even while inactive. A malformed `policyExists` or `isAuthorized` call can reach decode classification while a malformed `updateAllowlist` call is hidden behind the activation error.

Policy registry error behavior becomes activation dependent. Clients, tests, and monitoring cannot reliably distinguish an inactive feature from malformed calldata or an unknown selector for non view calls. This does not change policy state, but it weakens ABI diagnostics and can make cross environment simulations disagree near activation boundaries.

Recommendation: Decode and classify calldata before enforcing the activation gate, or at least determine whether the selector belongs to a known write function before returning `FeatureNotActivated`. Keep view calls callable while inactive, but preserve `UnknownFunctionSelector` and ABI decode errors for invalid calldata. Add regression tests for inactive unknown selectors, malformed view selectors, and malformed write selectors.

Coinbase: Fixed in PR 3421.

Cantina Managed: Fix verified.

3.2.11 Policy registry decodes account batches before enforcing the size cap

Severity: Low Risk.

Context: dispatch.rs#L45-L50.

Description: The policy registry defines a 64 account batch cap for `createPolicyWithAccounts`, `updateAllowlist`, and `updateBlocklist`, but that cap is enforced only after ABI dispatch has decoded the full dynamic `address[]` argument into a `Vec<Address>`.

`PolicyRegistryStorage::inner` first decodes calldata through the shared ABI macro. The generated call structs own their `accounts` field as a vector, so the dynamic array has already been allocated and decoded by the time execution reaches `require_account_batch_size`.

Alloy's dynamic sequence decoder reads the declared array length and reserves a vector with that capacity before decoding the elements. A valid oversized array is therefore fully allocated and decoded before the policy cap returns `BatchSizeTooLarge`. A malformed calldata body can also declare a very large array length with little payload, causing a large allocation attempt before the decoder later discovers that the elements are missing.

The 64 account cap does not protect the ABI decoding path. Callers can force policy registry dispatch to spend memory and CPU on oversized or malformed account arrays before the intended policy level size check rejects the call.

This does not bypass authorization or mutate state by itself, but it weakens resource accounting for the native precompile. The most concerning path is a short malformed call with a very large declared array length, which can reach `Vec::try_reserve` before failing ABI decoding.

Recommendation: Reject oversized policy account batches before full vector allocation. For these selectors, pre-scan the calldata head and dynamic array length, validate that the length is at most `MAX_ACCOUNTS_PER_BATCH`, and only then run the full ABI decoder. Alternatively, implement a capped decoder for the policy account arrays that fails as soon as the declared length exceeds the policy limit.

Add regression tests for `createPolicyWithAccounts`, `updateAllowlist`, and `updateBlocklist` with an array length of 65, and for malformed calldata that declares a very large account array length with insufficient element data. The malformed case should fail without attempting to allocate according to the declared length.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.3 Informational

3.3.1 Repeated deactivation returns wrong ABI error

Severity: Informational.

Context: storage.rs#L136-L146.

Description: The activation registry ABI declares two different errors for inactive features: `AlreadyDeactivated(bytes32)` for a repeated deactivation and `FeatureNotActivated(bytes32)` for checks that require an active feature. The implementation does not make that distinction. `ActivationRegistryStorage::set_activated` uses the requested target state as the only discriminator, so `deactivate(feature)` on an already inactive feature returns `FeatureNotActivated(bytes32)` instead of the ABI advertised `AlreadyDeactivated(bytes32)`.

This makes `AlreadyDeactivated` unreachable on the deactivation path. Clients generated from the ABI and automation that handles activation idempotency cannot reliably distinguish "this feature is already deactivated" from "this feature failed an activation check or was never activated".

Repeated deactivation failures are reported with the wrong selector. Integrators that depend on the ABI cannot observe the advertised `AlreadyDeactivated(bytes32)` error and may misclassify an idempotent admin operation as a missing activation state failure.

Recommendation: Return `IActivationRegistry::AlreadyDeactivated` when `current == activated` and `activated` is `false`. Keep `FeatureNotActivated` for `checkActivated(bytes32)` and other paths that require an active feature. Add ABI regression tests that assert repeated activation returns `AlreadyActivated(bytes32)` and repeated deactivation returns `AlreadyDeactivated(bytes32)` by selector.

Coinbase: Fixed in PR 3253.

Cantina Managed: Fix verified.

3.3.2 Activation admin is not consensus anchored

Severity: Informational.

Context: storage.rs#L127-L133.

Description: The activation registry admin controls a precompile that can change state, but the address is carried as chain spec metadata outside consensus instead of being anchored in consensus state or fork identity. `BaseChainSpec` stores the admin separately from the inner `ChainSpec`, validates only that some admin is present when Beryl is scheduled, accepts arbitrary admin metadata when wrapping an existing `ChainSpec`, and then passes the option into EVM construction. The genesis header is built from `make_genesis_header(&genesis, &hardforks)`; the activation admin is not an input to that header construction, and the `Hardforks` fork ID implementation delegates to `inner` without including the admin.

Two nodes can therefore have the same chain ID, genesis state, genesis hash, and fork schedule while disagreeing about which caller is authorized to execute `ActivationRegistry.activate(bytes32)` or `deactivate(bytes32)`. Once a block contains an activation mutation from either configured admin, one node applies the storage update and emits the event while the other reverts the call. That is a consensus split on the activation registry state and any gated B20 or policy precompile behavior that follows.

This also affects fault proof execution: proof boot information derives the activation admin from a built in `ChainConfig` based only on the committed chain ID, then threads it into the EVM factory. If an execution node was configured with a different admin for the same chain ID and fork schedule, native execution and proof execution can disagree on precompile results. The succinct public `BootInfoStruct` commits the rollup config hash but not the activation admin itself.

This issue is scoped to the new Beryl activation registry functionality. The activation registry is not part of the already live Base mainnet or Sepolia execution rules while Beryl remains unscheduled there. It becomes consensus critical once Beryl is active, or on custom chains where operators can supply Beryl active genesis or chain spec data.

Recommendation: Make the activation admin a value committed by consensus. Prefer storing it in activation registry state at genesis and reading it from storage during precompile execution, or otherwise include it in a genesis or fork identity that consensus checks so peers and provers cannot silently disagree on it. Reject `Address::ZERO` as a configured admin, and add tests proving that changing the admin either changes the committed chain identity or is rejected before block execution.

```
//Store admin in genesis state and read it from storage

// Add admin storage to the activation registry:

  #[contract(addr = Self::ADDRESS)]
  #[namespace("base.activation_registry")]
  pub struct ActivationRegistryStorage {
      pub features: Mapping<B256, bool>,
      pub admin: Address,
  }

  impl ActivationRegistryStorage<'_> {
      pub fn admin(&self) -> Result<Address> {
          self.admin.read()
      }

      fn require_admin(&self) -> Result<()> {
          let caller = self.storage.caller();
          let admin = self.admin()?;

          if admin.is_zero() || caller != admin {
              return Err(BasePrecompileError::revert(IActivationRegistry::Unauthorized {
              ↪  caller }));
          }

          Ok(())
      }

      pub fn set_activated(&mut self, feature: B256, activated: bool) -> Result<()> {
          if self.storage.is_static() {
              return Err(BasePrecompileError::revert(IActivationRegistry::StaticCallNotA
              ↪  llowed
              ↪  {}));
          }

          self.require_admin()?;

          // existing feature mutation logic...
          Ok(())
      }
  }
```

Then genesis construction must write `activation_admin_address` into `ActivationRegistryStorage::admin storage`. After that, changing admin changes genesis state root.

Coinbase: The network has a single centralized sequencer node. Thus this would not need consensus for the config at the moment.

Cantina Managed: Acknowledged by **Coinbase** team.

3.3.3 Bool storage decodes any non zero word as true

Severity: Informational.

Context: storage.rs#L71.

Description: The activation registry stores feature flags as `Mapping<B256, bool>`, and `is_activated` reads the mapped value directly through the storage library. The storage library writes booleans canonically as `0` or `1`, but its `FromWord for bool` decoder accepts any non zero storage word as `true`.

That means a feature slot containing `2`, `0xff`, or any other non zero value is treated as activated even though it is not a canonical boolean encoding. The same primitive decoder is shared by the other boolean mappings in the native precompiles, including policy membership flags, B20 role membership flags, and consumed announcement ids.

If raw state, genesis state, state import, migration code, or a future writer leaves a non canonical value in one of these boolean slots, the precompile logic treats it as `true`. For the activation registry specifically, any non zero value for a feature key enables the feature gate. For policy and role mappings, any non zero value grants membership semantics for the relevant mapping entry.

Recommendation: Make boolean decoding canonical by accepting only `0` and `1`. Return a storage decoding error for any other value, or normalize all boolean storage values before they can be observed by precompile logic. Add regression tests that write raw values such as `2` and `U256::MAX` into activation, policy, and role boolean slots and assert the intended behavior.

Coinbase: Fixed in PR 3364.

Cantina Managed: Fix verified.

3.3.4 `isB20Initialized` accepts any prefix account with code

Severity: Informational.

Context: storage.rs#L97-L101.

Description: `isB20Initialized(address)` is documented as returning whether `createB20` initialized the token, but the implementation only checks two structural conditions: the address must have the B20 prefix and the account must have non empty code. It does not verify that the factory predicted or created the address, that the marker bytecode is the factory marker, or that the token storage was initialized by `createB20`.

As a result, any B20 prefix account with arbitrary code returns `true` from `isB20Initialized`, even if the factory never created it. Normal transactions cannot realistically deploy arbitrary code to a chosen B20 prefix address because the prefix fixes the first byte and the next nine bytes to zero, but genesis state, state import, migration code, or other privileged state setup can create such an account.

Factory clients can receive a semantic false positive from `isB20Initialized`. An address can be reported as an initialized B20 token even though it was not created by the factory and may not have valid B20 storage. This is mostly an integration and migration safety issue rather than a practical user transaction exploit.

Recommendation: Track factory initialized tokens explicitly, or verify a stronger factory marker than generic code presence. For example, store a factory created flag keyed by token address during `createB20`, or require the exact marker code hash and any required initialization storage sentinel before returning `true`. Add a regression test that injects arbitrary code at a B20 prefix address without calling `createB20` and asserts `isB20Initialized` returns the intended value.

Coinbase: Fixed in PR 3382.

Cantina Managed: Fix verified.

3.3.5 `set_code` ignores static call context

Severity: Informational.

Context: storage.rs#L69-L71.

Description: `createB20` installs the B20 marker bytecode before token initialization by calling `StorageCtx::set_code`. The EVM backed `set_code` provider path does not check whether the current precompile execution is static. It deducts code deposit gas, may deduct create gas and state gas, and then writes code through the revm internals before any later token storage write reaches `sstore`.

Other state mutation operations reject static execution at the provider boundary. `sstore`, `tstore`, and log emission all check `self.is_static` before charging gas or mutating state. `set_code` is the exception, so a static `createB20` call observes the first attempted mutation incorrectly: marker code installation runs and charges gas first, then a later initialization storage write rejects with `StaticCallViolation`.

The factory checkpoint should roll back the marker bytecode in production, so this does not appear to leave a persistent token from a static call. The observable issue is wrong static call semantics and extra gas charged before the call eventually reverts.

Static `createB20` calls are overcharged and do not fail at the first state mutation. The final state should be rolled back by the factory checkpoint, but gas accounting and mutation ordering differ from normal EVM static call behavior.

Recommendation: Add a static call guard before any gas deduction or mutation in `set_code`, preferably at the provider implementation or `StorageCtx::set_code` boundary so all callers get the same behavior. Add a regression test that executes `createB20` through a static call and asserts it reverts before marker code gas is charged or code is written.

Coinbase: Fixed in PR 3370.

Cantina Managed: Fix verified.

3.3.6 Activation bytecode marker persists after deactivation

Severity: Informational.

Context: storage.rs#L148-L155.

Description: The activation registry writes marker bytecode to its own account when a feature is first activated. The deactivation path only deletes the feature flag mapping entry and emits `FeatureDeactivated`; it never clears the marker bytecode from `ActivationRegistryStorage::ADDRESS`.

As a result, after an activate then deactivate sequence, `isActivated(feature)` correctly returns `false`, but `EXTCODESIZE(ActivationRegistryStorage::ADDRESS)` and `EXTCODEHASH(ActivationRegistryStorage::ADDRESS)` remain non zero. Any integration or onchain guard that treats code presence at the activation registry address as a proxy for feature enablement receives a permanent false positive after the first successful activation.

This is not a bypass of the registry's own `isActivated` or `checkActivated` calls. The documented live state signal is the feature flag API, and the README does not document the bytecode marker as an activation signal. The issue is an externally visible state inconsistency that can mislead integrations that use code presence as a shortcut.

Integrations can incorrectly conclude that activation is still enabled after deactivation if they inspect code presence instead of calling the registry. This can cause stale onchain guards, monitoring, or migration checks to behave as though a feature remains live even after the admin disabled it.

Recommendation: Either clear the marker bytecode when the activation registry has no active features, or explicitly document that activation registry code presence is only an initialization marker and not a live feature state signal. Add a regression test that verifies the intended post deactivation model for both `isActivated` and account code presence.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.3.7 B20 variant decimals helper is misleading

Severity: Informational.

Context: variant.rs#L97-L100.

Description: `B20Variant::decimals()` returns `6` for both asset and stablecoin variants. That is accurate for stablecoin creation, but asset decimals are not a variant constant. They are supplied during asset token creation, validated against the allowed range, and stored per token.

The current factory implementation does not use this helper for asset creation events. `init_asset_token` copies `init.decimals` before initialization and emits that value in `B20Created`, while `init_stablecoin` uses `B20Variant::Stablecoin.decimals()` for the fixed stablecoin event value. There is no current asset decimals event bug, but the helper name and asset arm make future callers likely to assume it returns the decimals for any variant.

Recommendation: Remove the asset arm from `B20Variant::decimals()`, rename the helper to clarify that it is only a stablecoin default, or return an optional fixed decimals value where assets return `None`. Add a regression test or compile time check around asset creation events so they continue to emit the token specific `init.decimals` value.

Coinbase: Fixed in PR 3345.

Cantina Managed: Fix verified.

3.3.8 B20 factory accepts dirty ABI words

Severity: Informational.

Context: abi.rs#L83-L91.

Description: Factory dispatch decodes calldata with `SolInterface::abi_decode` through the shared `decode_precompile_call!` macro. Nested `createB20.params` bytes are also decoded with `abi_decode`. These decode paths do not enforce canonical Solidity ABI words for narrow integer, enum, and address fields, so dirty high bits can be masked or truncated instead of rejected.

This affects the outer factory arguments and the inner creation parameter structs. A dirty enum word with a low byte matching `STABLECOIN` can decode as `B20Variant::STABLECOIN`. A dirty address word with non zero high 12 bytes can decode as its low 20 byte address. Dirty `uint8` words can similarly decode to their low byte. Affected fields include `variant`, `sender`, `token`, `version`, `initialAdmin`, and asset `decimals`.

Solidity's canonical ABI decoder rejects these dirty words. The native factory therefore accepts some calldata that a Solidity implementation of the same ABI would reject.

Callers and integrations can observe permissive factory behavior for non canonical calldata. This can make offchain simulation, Solidity wrappers, and native precompile execution disagree about whether a call is valid. The most direct effects are semantic ambiguity around variant selection, address arguments, version checks, and asset decimals validation.

Recommendation: Use the validating ABI decoder for factory dispatch and nested creation params. Replace `abi_decode` with `abi_decode_validate` where available, or add explicit canonical word checks for enum, address, and narrow integer fields before accepting decoded values. Add regression tests with dirty high bits for `variant`, `sender`, `token`, `version`, `initialAdmin`, and `decimals`.

Coinbase: Fixed in PR 3368.

Cantina Managed: Fix verified.

3.3.9 B20 asset zero multiplier normalizes to WAD

Severity: Informational.

Context: token.rs#L219.

Description: `updateMultiplier(0)` stores and emits a zero multiplier, but the public read path normalizes a stored zero back to `WAD`. The factory uses zero as the initial storage sentinel so newly created asset tokens can read as `1e18` without writing `WAD`. That sentinel behavior is reasonable for uninitialized storage, but `updateMultiplier` exposes the same raw storage slot as an operator controlled value without rejecting zero or normalizing it before storage and event emission.

An operator can therefore call `updateMultiplier(0)` and emit `MultiplierUpdated(0)`. After the transaction, `multiplier()` returns `1e18`, and `toScaledBalance`, `toRawBalance`, and `scaledBalanceOf` all behave as if the multiplier is WAD. Emitting `MultiplierUpdated(0)` and emitting `MultiplierUpdated(WAD)` produce different logs but the same onchain accounting result. Offchain systems that trust the emitted multiplier event will observe a different value than onchain reads and conversions.

The issue does not change raw balances or total supply, because the multiplier is only a derived view. It does create an event and read inconsistency on the asset accounting surface.

Indexers and integrations can record a multiplier of zero from `MultiplierUpdated(0)` while onchain reads return WAD and conversions use WAD. This can desynchronize displayed balances and audit records for a token until the next nonzero multiplier update.

Recommendation: Reject `newMultiplier == 0` in `updateMultiplier`, or normalize zero to `WAD` before both storage and event emission. If zero is meant to remain a storage sentinel only, the public mutation path should not be able to write it as an intentional multiplier update.

Coinbase: Fixed in PR 153.

Cantina Managed: Fix verified.

3.3.10 Dead `beryl()` constructor and missing `cobalt()`: fork dispatch collapses Azul / Beryl / Cobalt into `azul()`

Severity: Informational.

Context: provider.rs#L164-L166.

Description: `BasePrecompiles::new_with_spec` selects the static precompile table per fork via a match. The Azul, Beryl, and Cobalt upgrades are collapsed into a single arm that calls `Self::azul()`:

```
BaseUpgrade::Azul | BaseUpgrade::Beryl | BaseUpgrade::Cobalt => Self::azul(),
```

A dedicated `beryl()` constructor exists but is **never reached** — it only forwards to `azul()`:

```
/// Static precompiles are the same as Azul; Beryl adds dynamic precompiles at install
 ↪  time.
pub fn beryl() -> &'static Precompiles {
    Self::azul()
}
```

and there is **no `cobalt()` constructor** at all. This is functionally correct today — Beryl/Cobalt deliberately share Azul's static table, and the Beryl+ dynamic precompiles (B-20 factory, dynamic token lookup, policy registry, activation registry) are installed separately in `install_with_observer`, gated on `self.spec.upgrade() >= BaseUpgrade::Beryl` (`provider.rs:185`). So no behavior is wrong.

The issue is consistency and a future-divergence footgun. The naming is asymmetric: every other supported fork, `fjord()`, `granite()`, `isthmus()`, `jovian()`, `azul()`, has its own constructor that the match uses, but `beryl()` is dead and `cobalt()` is absent. If a later change ever needs Beryl or Cobalt to carry a *different* static table, a developer would naturally edit `beryl()` (or add to `cobalt()`) and expect

it to take effect — but the match ignores `beryl()` entirely, so the change would silently have no effect. That is exactly the kind of subtle, hard-to-spot mistake that can produce a fork/consensus discrepancy.

Recommendation: Pick one of two consistent shapes:

1. **Per-spec constructors (preferred for future divergence):** give each fork its own arm and constructor, with `beryl()` / `cobalt()` forwarding to `azul()` for now:

```
BaseUpgrade::Azul   => Self::azul(),
BaseUpgrade::Beryl  => Self::beryl(),
BaseUpgrade::Cobalt => Self::cobalt(),
```

```
pub fn beryl()  -> &'static Precompiles { Self::azul() }
pub fn cobalt() -> &'static Precompiles { Self::azul() } // add this
```

   This keeps the per-fork extension point live so a future divergence is a one-line edit in the obvious place.

2. **Drop the dead code:** remove `beryl()` and keep the shared `Azul | Beryl | Cobalt => Self::azul()` arm, with a comment stating these forks intentionally share Azul's static table and that any divergence must be added here.

Coinbase: Fixed in PR 3342.

Cantina Managed: Fix verified.

3.3.11 `isAnnouncementActive()` is not implemented by the B-20 Asset precompile

Severity: Informational.

Context: abi.rs#L68-L69.

Description: The B-20 Asset variant declares `isAnnouncementIdUsed(string)` in its Rust `IB20Asset` interface but does not declare or handle `isAnnouncementActive()`. As a result a standards-conformant call to `isAnnouncementActive()` (selector `0xf152925d`) is not decoded by any arm and falls through to `UnknownFunctionSelector`, i.e. the call reverts instead of returning a `bool`. For the same calldata the base-std reference returns a value, so this is an observable behavioral divergence from the spec for valid input.

The underlying state does exist: `B20AssetToken` carries an `in_announcement: bool` and an internal `is_announcement_active()` accessor (`token.rs:44`), but it is used only for the in-bracket recursion guard and is never exposed as a precompile function. Two gaps follow:

1. The selector is simply absent from the ABI surface and dispatch, so the documented read is unavailable to integrators.
2. The backing flag is process-local to a single `B20AssetToken` instance, and a fresh instance is constructed per precompile invocation (`b20_asset/precompile.rs`). The natspec promises the value is observable to inner-call contracts dispatched inside the bracket, which re-enter the token through a new precompile call; those calls would see a freshly-constructed `in_announcement = false`. So even adding the selector but backing it with the in-memory bool would not satisfy the cross-call, reset-per-transaction semantics the spec requires; that requires EIP-1153 transient storage.

Provenance / traceability: the function was added on base-std and then removed in the port, per:

- Commit 2979199b.
- PR 140.

This is raised for traceability within the requested audit scope; it appears to be reconciled in parallel. The team should confirm the intended end state (implement in the precompile vs. remove from the base-std interface) and keep the Rust ABI and the base-std interface in sync.

Recommendation: No code change is required. The discrepancy is an artifact of the spec snapshot used for this review, not a defect in the precompile: the audited base-std commit (`ea1b5b0`) still declared `isAnnouncementActive()` , but the function was subsequently removed from base-std (PR 140, commit `2979199`), so the precompile's omission of it is correct and the implementation and the current spec now agree.

Action items are administrative only:

- Confirm the canonical base-std commit the precompiles are expected to track, and re-pin the audit/scope baseline to a post-removal commit so the interface diff is clean.
- Treat this as closed (no remediation), recorded here for traceability of the scope/spec mismatch that existed at the start of the review.

Coinbase: Fixed in PR 140.

Cantina Managed: Fix verified.

3.3.12 Factory bootstrap bypasses transfer policies but still enforces the `MintReceiver` policy

Severity: Informational.

Context: mintable.rs#L21.

Description: `privileged=true` is set only during the factory bootstrap window, when the factory dispatches the creation `initCalls` on the new token (`b20_factory/storage.rs:139,181` , via `inner_with_privilege(.., true)`). In that window the policy checks are applied inconsistently across operations:

- `transfer` / `transfer_from` skip the transfer-side policies when privileged:

```
// transferable.rs:45
if !privileged {
    B20Guards::ensure_policy_type::<Self>(self, B20PolicyType::TransferSender,
    ↪  from)?;
    B20Guards::ensure_policy_type::<Self>(self, B20PolicyType::TransferReceiver,
    ↪  to)?;
}
// transferable.rs:99 (delegated transfer)
if !privileged && spender != from {
    B20Guards::ensure_policy_type::<Self>(self, B20PolicyType::TransferExecutor,
    ↪  spender)?;
}
```

- `mint` enforces the `MintReceiver` policy unconditionally; only the role gate is behind the privilege check:

```
// mintable.rs:15
if !privileged {
    B20Guards::ensure_token_role::<Self>(self, caller, B20TokenRole::Mint)?;
}
// ...
// mintable.rs:21 (runs even when privileged == true)
B20Guards::ensure_policy_type::<Self>(self, B20PolicyType::MintReceiver, to)?;
```

So during bootstrap, `TransferSender` / `TransferReceiver` / `TransferExecutor` are bypassed, but `MintReceiver` is not. The behavior is defensible (a reasonable invariant is ”never mint to a policy-denied address, even at bootstrap”), and it is not reachable by untrusted callers (privileged is factory-only, and fresh tokens default every scope to `ALWAYS_ALLOW`). The problem is that the asymmetry is implicit: nothing in the code comments, the `IB20` / `IB20Factory` natspec, or the B-20 docs states that policy bypass during the bootstrap window applies to transfers but deliberately excludes mint. An integrator whose `initCalls` set a restrictive MintReceiver policy and then mint in the same bootstrap bundle would hit a `PolicyForbids` revert that the ”init bypasses policy gates” mental model does not predict.

Recommendation: Make the intent explicit and pin it:

- Document the asymmetry where the bootstrap window is described (`IB20Factory` natspec / B-20 Factory docs) and at the two call sites (`transferable.rs:45` , `mintable.rs:21`): during factory `initCalls` , transfer-side policies are bypassed but the MintReceiver policy is always enforced, with the rationale (never issue new supply to a policy-denied recipient, even at creation).
- Add a unit test that pins both behaviors under `privileged=true` (privileged transfer to a denied receiver succeeds; privileged mint to a denied `MintReceiver` reverts), so the asymmetry is intentional and protected against accidental change.
- Confirm this matches the base-std reference's stated bootstrap-policy semantics; if base-std documents a blanket "policy gates bypassed during init", reconcile the two so the spec and implementation agree.

Coinbase: Fixed in PR 150.

Cantina Managed: Fix verified.

3.3.13 EIP-2612 `permit` accepts non-canonical (high-s) signatures

Severity: Informational.

Context: permittable.rs#L149-L164.

Description: `recover_signer` applies no low-s guard. It builds the signature with `Signature::from_scalars_and_parity(r, s, parity)` and calls `recover_address_from_prehash` , which normalizes a high-s value back to low-s (and flips parity) before recovery. As a result the malleated twin `(r, N - s, v xor 1)` of a valid signature recovers the same `owner` and is accepted, i.e. `permit` treats both the canonical and the non-canonical encoding of a signature as valid.

This is standard ECDSA signature malleability. As implemented it is not exploitable:

- Replay is prevented by the per-owner nonce, which is part of the signed digest and is incremented on each successful `permit` (`permittable.rs:157-162`). Once either encoding is used, the nonce advances and the other no longer matches the recomputed digest, so only one can ever execute.
- No state is keyed by the signature bytes. Malleability is dangerous when a system stores or dedups by `keccak(r, s, v)` ; the twin hashes differently and slips past such a check. This code keys replay off the nonce only, so the alternate encoding has nothing to bypass.
- `owner` , `spender` , `value` , and `deadline` are all inside the signed digest, so the twin cannot redirect or alter the approval. The only possible effect is that the owner's own one-time approval executes via an equivalent encoding of their signature, with an identical `Approval` result.

Why harden anyway: accepting low-s only (canonical signatures) is the convention enforced by OpenZeppelin's `ECDSA` library and is what many wallets and signing tools assume an EIP-2612 endpoint enforces. Rejecting high-s removes this class of malleability outright and provides a cheap layer of defense-in-depth against any future code path that might key off a signature instead of the nonce.

Proof of Concept: Integration test of `base-common-precompiles` (`--features test-utils`): sign a valid permit (k256 emits low-s), construct the high-s twin `s' = N - s` with flipped parity, submit only the twin, and observe `permit` accepts it and sets the allowance.

```
use alloy_primitives::{Address, B256, U256, keccak256};
use base_common_precompiles::{
    InMemoryPolicy, InMemoryTokenAccounting, PermitArgs, Permittable, TestToken, Token,
    TokenAccounting,
};
use k256::ecdsa::SigningKey;

// secp256k1 group order N
const SECP256K1_N: U256 = U256::from_be_bytes([
    0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
    ↪  0xff, 0xfe,
```

```
    0xba, 0xae, 0xdc, 0xe6, 0xaf, 0x48, 0xa0, 0x3b, 0xbf, 0xd2, 0x5e, 0x8c, 0xd0, 0x36,
    ↪  0x41, 0x41,
]);
const PRIVATE_KEY: [u8; 32] =
    alloy_primitives::hex!("ac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2
    ↪  ff80");
const SPENDER: Address = Address::repeat_byte(0xbb);
const TOKEN_ADDR: Address = Address::repeat_byte(1);
const CHAIN_ID: u64 = 1;

fn owner_address() -> Address {
    let key = SigningKey::from_slice(&PRIVATE_KEY).unwrap();
    let point = key.verifying_key().to_encoded_point(false);
    Address::from_slice(&keccak256(&point.as_bytes()[1..])[12..])
}

#[test]
fn poc_permit_accepts_high_s() {
    let owner = owner_address();
    let mut accounting = InMemoryTokenAccounting::new(TOKEN_ADDR);
    accounting.name = "TestToken".to_string();
    let mut token: TestToken = TestToken::with_storage_and_policy(accounting,
    ↪  InMemoryPolicy::new());

    let value = U256::from(500u64);
    // Sign a valid canonical (low-s) permit.
    let domain_sep = token.domain_separator(CHAIN_ID).unwrap();
    let nonce = token.accounting().nonce(owner).unwrap();
    let mut args = PermitArgs {
        owner, spender: SPENDER, value, deadline: U256::MAX, v: 0, r: B256::ZERO, s:
        ↪  B256::ZERO,
    };
    let (sig, recid) = SigningKey::from_slice(&PRIVATE_KEY)
        .unwrap()
        .sign_prehash_recoverable(args.signing_hash(domain_sep, nonce).as_slice())
        .unwrap();
    let sig_bytes = sig.to_bytes();
    args.r = B256::from_slice(&sig_bytes[..32]);
    args.s = B256::from_slice(&sig_bytes[32..]);
    args.v = if recid.is_y_odd() { 28 } else { 27 };

    let low_s = U256::from_be_bytes(args.s.0);
    assert!(low_s <= SECP256K1_N / U256::from(2u64), "k256 emits low-s");

    // Malleate to the non-canonical high-s twin: s' = N - s, flip parity.
    let high_args = PermitArgs {
        s: B256::from((SECP256K1_N - low_s).to_be_bytes()),
        v: if args.v == 27 { 28 } else { 27 },
        ..args
    };

    // The high-s twin is accepted: no low-s guard in recover_signer.
    assert!(token.permit(CHAIN_ID, U256::ZERO, high_args).is_ok());
    assert_eq!(token.accounting().allowance(owner, SPENDER).unwrap(), value);
}
```

Output:

```
low-s  = 0x4fdc8b597feb140608a1a4d2bb6df7543fcde85c041849a125e6354300ee9783
high-s = 0xb02374a68014ebf9f75e5b2d449208aa7ae0f48aab30569a99ec2949cf47a9be  (= N - s,
↪  non-canonical)
test poc_crypto001_high_s_permit_accepted ... ok
```

Recommendation: Add a low-s check to `recover_signer`: reject signatures with `s > N/2` (the secp256k1 half-order) and return the existing `InvalidSigner` revert, before treating the signature as

valid. This makes `permit` accept canonical signatures only, matching the OpenZeppelin `ECDSA` convention and eliminating the malleable second encoding. The change has no effect on the current (already nonce-safe) flow; it only rejects the non-canonical twin. Add a unit test asserting that a high-s signature reverts with `InvalidSigner` while its canonical low-s counterpart succeeds.

Coinbase: Acknowledged.

Cantina Managed: Since replay protection is already handled via the per-owner nonce, rejecting high-s signatures provides no security benefit in this context. Additionally, enforcing a low-s constraint may introduce compatibility issues with wallets and signing libraries that produce high-s signatures.

3.3.14 B20 lookup exposes uninitialized token addresses

Severity: Informational.

Context: lookup.rs#L35-L47.

Description: `BerylLookup` returns a dynamic B20 token precompile for any address that structurally matches a supported B20 prefix and variant byte. The lookup does not check whether the factory actually created the token or whether the token account has the marker bytecode.

The token dispatchers do perform an initialization check before executing ABI handlers. If the address has not been initialized by the factory, `dispatch_with_observer` returns an empty revert. This prevents fake token operations from succeeding, but it means `PrecompilesMap::get` and `PrecompilesMap::contains` can still report a B20 precompile for an address that was never created.

This does not let callers operate an uninitialized token. Calls to never created B20 prefix addresses revert before reaching token logic.

The observable issue is address classification. Integrations that treat precompile map lookup as evidence that a token exists can get false positives for reserved B20 prefix addresses. Calls to those addresses also behave like failed precompile calls instead of ordinary calls to empty accounts.

Recommendation: Document that B20 dynamic lookup is a structural route, not a token existence check, and require integrations to use `isB20Initialized` or the token dispatch initialization check for existence. If lookup is intended to expose only created tokens, move the bytecode or initialization check into a state aware lookup path and add a regression test for a structurally valid but never created B20 address.

Coinbase: Acknowledged. Calls to structurally valid but never-created B20 addresses still revert before any token logic executes, so no fake token operation can succeed. The lookup is intentionally a structural route rather than an existence check. We will add documentation clarifying that integrations should use `isB20Initialized` rather than the precompile map for token existence checks.

Cantina Managed: Acknowledged by **Coinbase** team.

3.3.15 Beryl lookup install overwrites existing dynamic lookup

Severity: Informational.

Context: lookup.rs#L22-L27.

Description: `BerylLookup::install_with_observer` installs the Beryl dynamic precompile lookup by calling `PrecompilesMap::set_precompile_lookup`. The map stores a single dynamic lookup, so installing Beryl lookup on a map that already has a dynamic lookup silently replaces the previous lookup.

The normal `BasePrecompiles::install_with_observer` path starts from `PrecompilesMap::from_static(self.precompiles())`, then installs the Beryl lookup. That default path is not affected because there is no prior dynamic lookup to preserve. The issue is at the public helper boundary for embedders or tests that compose Base precompiles with another dynamic lookup.

Custom embedders can accidentally lose an existing dynamic precompile resolver when they call `BerylLookup::install` or `install_with_observer`. Addresses that were previously resolved by the older lookup then stop behaving as precompiles.

This is not a normal Base execution issue, because the default install path builds a fresh static map before adding Beryl dynamic precompiles.

Recommendation: Either document that Beryl lookup installation replaces any existing dynamic lookup, or provide a composable install helper that chains an existing lookup with Beryl lookup. If replacement is intended, consider asserting or exposing a return value so callers can detect that a previous lookup was overwritten.

Coinbase: Acknowledged.

Cantina Managed: The default `BasePrecompiles::install_with_observer` path always builds a fresh static map before installing the Beryl lookup, so the silent replacement behavior only affects custom embedders composing multiple dynamic lookups. Coinbase plans to add documentation to `install` and `install_with_observer` making the replacement semantics explicit.

3.3.16 Installed precompiles map can go stale after spec change

Severity: Informational.

Context: provider.rs#L172-L174.

Description: `BasePrecompiles::install()` builds a `PrecompilesMap` once from the current Base spec. For Beryl and later, that construction also installs the B20 factory, dynamic B20 lookup, policy registry, and activation registry.

After the map is installed, it is no longer tied to the EVM config spec. The `PrecompileProvider` implementation for Alloy's `PrecompilesMap` returns `false` from `set_spec()` and does not rebuild or replace the installed table. If an embedder reuses an EVM and mutates `cfg.spec` across a Base fork boundary, revm's spec update path can leave the old precompile map in place.

This differs from the normal Base factory path, which creates a fresh `PrecompilesMap` from the `EvmEnv` spec when constructing a new EVM. The issue is reachable for direct embedders or tests that reuse an existing EVM instance across spec changes instead of rebuilding it.

An EVM reused across a spec change can execute with stale precompile behavior. For example, a Jovian map reused after switching to Azul can keep old MODEXP or P256 rules. An Azul map reused after switching to Beryl can miss the Beryl native precompiles. A Beryl map reused after switching back to Azul can leave Beryl native precompiles available before they should exist.

This is not a normal Base block or proof execution issue if those paths construct a fresh EVM and precompile map from each block's `EvmEnv`. It is an API correctness hazard for custom embedders that mutate `ctx.cfg.spec` or otherwise reuse an installed `PrecompilesMap` across fork boundaries.

Recommendation: Document that installed `PrecompilesMap` values are spec-specific and must be rebuilt whenever the Base spec changes, or wrap the map in a Base-aware provider whose `set_spec()` rebuilds the installed map from `BasePrecompiles::new_with_spec(spec).install()`. Add a regression test that constructs an EVM at Azul, changes the config spec to Beryl, and verifies that Beryl native precompiles are not silently missing or stale.

Coinbase: Acknowledged. Normal block and proof execution paths construct a fresh EVM from each block's environment and are not affected. This is an API hazard for custom embedders that mutate the spec on an existing EVM instead of rebuilding it. We will add documentation to install making clear that the returned map is spec-specific and must be rebuilt on fork boundary changes.

Cantina Managed: Acknowledged by **Coinbase** team.

3.3.17 Stablecoin currency validation runs after token existence check

Severity: Informational.

Context: storage.rs#L47-L67, storage.rs#L296-L300.

Description: `B20FactoryStorage::create_b20` appears to validate creation parameters before computing the deterministic token address and checking whether a token already exists there. That ordering is true for asset decimals, but not for stablecoin currency: `TokenCreateParams::validate_stablecoin` is a no-op, and the actual empty/non-uppercase currency validation is deferred until `B20StablecoinStorage::initialize`.

As a result, when the target stablecoin address already exists, a second `createB20` call with the same `(caller, variant, salt)` and an invalid `currency` returns `TokenAlreadyExists` before reaching the stablecoin currency checks. A fresh-address call with the same invalid `currency` returns `MissingRequiredField("currency")` or `InvalidCurrency(code)`.

The `base-std` factory interface documents stablecoin currency validation errors as part of `createB20`, and the local factory already hoists asset-decimals validation into the pre-address-check validation phase. Stablecoin currency should follow the same pattern so field-level validation is not masked by deterministic-address collisions.

The same invalid stablecoin creation parameters can produce different custom errors depending only on whether the deterministic token address has already been deployed. Contracts, tests, or tooling that expect malformed creation parameters to be rejected with `MissingRequiredField` or `InvalidCurrency` can instead observe `TokenAlreadyExists` when the caller reuses a salt.

This does not corrupt state or bypass authorization, but it creates an observable revert-data mismatch and makes factory validation order inconsistent across stablecoin and asset variants.

Recommendation: Move the stablecoin currency checks into `TokenCreateParams::validate_stablecoin`, before `compute_address` and the `TokenAlreadyExists` check. Keep `B20StablecoinStorage::initialize` defensive if desired, but the factory-level validation should reject empty and non-uppercase currency values during the same pre-address-check phase as asset decimals.

Add a regression test that first creates a stablecoin, then repeats `createB20` with the same caller/salt and an empty or lowercase `currency`, asserting that the field-level currency error takes precedence over `TokenAlreadyExists`.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.