December 2, 2024

Pectra System Contracts

Comprehensive Security Assessment

contact@plainshift.io
plainshift.io

Table of Contents

Disclaimer

This security assessment represents a time-boxed security review using tooling and manual review methodologies. Our findings reflect our comprehensive evaluation of the materials provided in-scope and are specific to the commit hash referenced in this report.

The scope of this security assessment is strictly limited to the code explicitly specified in the report. External dependencies, integrated third-party services, libraries, and any other code components not explicitly listed in the scope have not been reviewed and are excluded from this assessment.

Any modifications to the reviewed codebase, including but not limited to smart contract upgrades, protocol changes, or external dependency updates will require a new security assessment, as they may introduce considerations not covered in the current review.

In no event shall Plainshift's aggregate liability for all claims, whether in contract or any other theory of liability, exceed the Services Fee paid for this assessment. The client agrees to hold Plainshift harmless against any and all claims for loss, liability, damages, judgments and/or civil charges arising out of exploitation of security vulnerabilities in the contracts reviewed.

By accepting this report, you acknowledge that deployment and implementation decisions rest solely with the client. Any reliance upon the information in this report is at your own discretion and risk.

This disclaimer is governed by and construed in accordance with the laws specified in the engagement agreement between Plainshift and the client.

1. Introduction

Plainshift is a full-stack security firm covering all aspects of security be it web app, smart contract, penetration testing, fuzzing and program verification or operational security. We were founded in September 2024 by industry veterans seeking to meaningfully revolutionize how teams approach security and guide them towards a holistic approach rather than the single sided approach so prevalent today.

Learn more about us at plainshift.io

1.1. Executive Summary

Plainshift was tasked with reviewing Ethereum's upcoming Prague+Electra hard fork system contract bytecode from November 18th to December 2nd. The engagement was conducted by 0xsomnus, designated lead auditor, and Surya. By our accounts, the system contracts have no significant issues and are all clear for deployment. However, we did find small deviations from the spec outlined in their respective EIPs in addition to comment mismatches.

Methods used in the engagement were manual review of the contracts alongside formal verification of the fake_expo contract.

1.2. Project Timeline

Date	Phase
2024-11-18	Audit Kickoff
2024-11-29	End of Security Review
2024-12-02	Report Delivery

1.3. Scope

Repositories	https://github.com/lightclient/sys-asm
Versions	`1d679164e03d73dc7f9a5331b67fd51e7032b104`
Programs	EIP-2935: - ctor.eas - main.eas EIP-7002: - ctor.eas - main.eas EIP-7251: - ctor.eas - main.eas Common: - fake_expo.eas
Type	Geas assembly
Platform	EVM-compatible

1.4. Overview of Findings

Our comprehensive review yielded no major functional or security issues. The only issues we were able to incur were 2 minor deviations from the spec and various code comment mismatches which have been divided per target file in the report.

Severity/Impact Level	Count
Critical	0
High	0
Medium	0
Low	1
Informational	4

2. Detailed Findings

2.1. Comment mismatches in `withdrawals`

Target	`withdrawals/main.eas`	**Category**	Documentation
Severity	Informational	**Status**	Unresolved

2.1.1. Description

We found a non-zero number of documentation issues when it came to the stack visualization comments in parallel with the code. These include the following:

On line 168, the comments should be `[tail_idx_slot, tail_idx+1]` over this.

```
add                 ;; [tail_idx+1]
push QUEUE_TAIL     ;; [tail_idx_slot]
```

Line 298 should be `[mask, pk2_am, pk2_am, offset, ..]` over this.`

```
dup1                ;; [pk2_am, pk2_am, offset, ..]
push pk2_mask       ;; [mask, pk2_am, offset, ..]
```

On lines 363 to 370, there's an extra `count` appended to the comments that shouldn't be there.

```
sload               ;; [excess, count]


;; Check if excess needs to be reset to 0 for first iteration after
;; activation.
dup1                ;; [excess, excess, count, count]
push EXCESS_INHIBITOR ;; [inhibitor, excess, excess, count, count]
eq                  ;; [inhibitor == excess, excess, count, count]
iszero              ;; [inhibitor != excess, excess, count, count]
jumpi @skip_reset   ;; [excess, count, count]


;; Drop the excess from stack and use 0.
```

```
  pop                   ;; [count, count]
```

Line 381 should be `[excess, target, count, excess, count]` over this.

```
  push TARGET_PER_BLOCK ;; [target, count, excess, count]
  dup3                  ;; [excess, target, count, excess]
```

2.1.2. Remediation Status

Unresolved.

2.2. Comment mismatches in `consolidations`

Target	`consolidations/main.eas`	**Category**	Documentation
Severity	Informational	**Status**	Unresolved

2.2.1. Description

Consolidations were mostly free from comment mismatches however we did find one issue.
On line 301, the comments should be `[2, slotbase, offset, slotbate, ..]`over this.

```
    push 2                ;; [1, slotbase, offset, slotbase, ..]
```

On line 299, there's a typo. Should be `storage`

```
    ;; Read slot 'spk2_tpk1' from stoarge.
```

2.2.2. Remediation Status

Unresolved.

2.3. Comment mismatches in `execution_hash`

Target	`execution_hash/main.eas`	**Category**	Documentation
Severity	Informational	**Status**	Unresolved

2.3.1. Description

There’s a typo on line 55, it should be `verified`. Additionally, there’s a greater issue with the entire comment block in the entire second half documenting the code is missing. We verified that this second half was in a previous commit hash but removed in the current version. See the attached code-block for a diff of the comments in the previous version vs how they currently appear. If you prefer, this links to the github diff.

```
  ;; Check if the input is requesting a block hash before the earliest available
  ;; hash currently. Since we've verfied that input <= number - 1, it's safe to

- ;; check the following:
- ;; number - 1 - input <= BUFLEN, which also equals: number - input < BUFLEN
- dup1 ;; [input, input]
- number ;; [number, input, input]
- sub ;; [number - input, input]
- push BUFLEN+1 ;; [buflen, number - input, input]
- lt ;; [buflen < number - input, input]
+ push BUFLEN ;; [buflen, input]
+ dup2 ;; [input, buflen, input]
+ number ;; [number, input, buflen, input]
+ sub ;; [number - input, buflen, input]
+ gt ;; [number - input > buflen, input]
```

2.3.2. Remediation Status

Unresolved.

2.4. `execution_hash` doesn't check if input > 8 bytes

Target	`execution_hash/main.eas`	**Category**	Spec Deviation
Severity	Informational	**Status**	Unresolved

2.4.1. Description

In lines 35 to 38, the system contract checks whether the calldata (input) is exactly 32 bytes long.

```
;; Verify input is 32 bytes long.
push 32          ;; [32]
calldatasize     ;; [calldatasize, 32]
sub              ;; [calldatasize - 32]
```

However, it is not sufficient according to the spec outlined in EIP-2935. It states that the calldata to read a historical block hash must be a 32 bytes long right-padded number that is not greater than the value `2^64-1` or `0xFFFFFFFFFFFFFFFF` (the greatest number in a uint64), if it is then revert. This is true even in the spec sample bytecode which explicitly checks that the input is not greater than 8 bytes and reverts if it is.

```
// check if input > 8 byte value and revert if this isn't the case
// the check is performed by comparing the biggest 8 byte number with
// the call data, which is a right-padded 32 byte number.
push8 0xffffffffffffffff
push0
calldataload
gt
push1 0x53
jumpi
```

The geas implementation deviates from this behavior and only checks whether the calldata is 32 bytes long, not if it's greater than 8 bytes.

2.4.2 Impact

We suspect that this check was included in spec because `BlockNumber` is of type int64 in the `geth` code, in which case the max number to compare against would be `0x7FFFFFFFFFFFFFFF` which is allowed within the bounds of `2^64-1` anyways.

2.4.3 Resolution status

Unresolved.

2.5. `execution_hash` reverts where it should return zero

Target	`execution_hash/main.eas`	**Category**	Spec Deviation
Severity	Low	**Status**	Unresolved

2.5.1. Description

On lines 43 to 52, the contract checks if the requested block hash is greater than the current block number minus 1 and reverts if so.

```
  ;; Check if input is requesting a block hash greater than current block number
  ;; minus 1.
  push 0          ;; [0]
  calldataload    ;; [input]
  push 1          ;; [1, input]
  number          ;; [number, 1, input]
  sub             ;; [number-1, input]
  dup2            ;; [input, number-1, input]
  gt              ;; [input > number-1, input]
  jumpi @throw    ;; [input]
```

The same happens for lines 54 to 61, where the contract checks if a block hash earlier than the first hash recorded in the contract storage is requested.

```
;; Check if the input is requesting a block hash before the earliest
available
;; hash currently. Since we've verfied that input ≤ number - 1, it's safe to
push BUFLEN       ;; [buflen, input]
dup2              ;; [input, buflen, input]
number            ;; [number, input, buflen, input]
sub               ;; [number - input, buflen, input]
gt                ;; [number - input > buflen, input]
jumpi @throw      ;; [input]
```

This is in contrast to the spec outlined in EIP-2935 which states that these checks should return 0 for output outside, not revert. Verbatim, the spec states `For any output outside the range of` `[block.number-HISTORY_SERVE_WINDOW, block.number-1] return 0.` This is true even in the spec's example implementation.

2.5.2 Impact

While both approaches are interpreted as errors by dependents, most dependents will follow the EIP and write code that expects to handle returns of 0. With both the implementation and EIP as is, this will result in redundant error handling code added to dependent contracts.

We speculate that reverts were chosen as a change to avoid this error handling in the first place.

2.5.3 Recommendation

We suggest either a change in the EIP to reflect the implementation or that the implementation follows spec.

2.5.4 Resolution status

Unresolved.

3. Formal Verification

For the `fake_expo` system contract, we verified integral properties regarding exponentiation given this is an approximate implementation. We used a16z's SMT solving frontend, Halmos for this purpose. Halmos describes itself as a symbolic testing tool for EVM smart contracts. We leveraged the existing tests and extended them with properties specific to exponentiation. Only the `fake_expo` contract was verified this way as we assessed the existing tests to be sufficient otherwise.

Given that `fake_expo` is meant to be used as a system function with an otherwise fixed factor and denominator, we focus invariants around this case whereas a more general exponentiation function would merit more rigorous testing.

Property ID	**Property**	**Status**
EXPO-01	If base = 1, result != 0	PASS
EXPO-02	If exponent 0, result = 1	PASS
EXPO-03	Exponent monotonicity	PASS

We include these regardless of whether they fail or not for the sake of completeness.

At the time of review, our verification confirms that the approximation is well within intended operation and maintains core mathematical properties that we'd expect from exponentiation.

3.1. If base = 1, result != 0

Target	`fake_expo.eas`	**Category**	Property Test
Severity	N/A	**Status**	Passes

3.1.1. Description

When the factor (base) is 1, mathematical principles dictate the result must never be zero, as 1 raised to any power equals 1. We ensure that the function never returns zero when factor=1. This validates that:

1. The approximation converges to non-zero results
2. No intermediate calculations incorrectly zero the result

```
// Test 1: Result must never be zero when factor is 1
   function check_FakeExpo_baseOneNonZero(uint256 numerator) public {
       vm.assume(numerator < type(uint256).max);
       uint256 result = callFakeExpo(1, int256(numerator), 17);
       assert(result ≠ 0);  // Approximation of 1^n should never be 0
   }
```

3.2. If exponent 0, result = 1

Target	`fake_expo.eas`	**Category**	Property Test
Severity	N/A	**Status**	Passes

3.2.1. Description

Any number raised to the power of zero must equal one. We verify that this property holds by verifying if calling the function with numerator=0 returns exactly 1. This validates that:

1. The series expansion correctly terminates for an exponent of 0
2. The fundamental mathematical property x^0=1 is preserved

Given all inputs here are static, this is more a "regular" property test than verification.

```
// Tests x^0 = 1 property when x=1
// Rationale:
// - Mathematical requirement: anything^0 = 1
// - Essential edge case in the approximation algorithm
   function check_FakeExpo_exponentZero() public {
       uint256 result = callFakeExpo(1, 0, 17);
       assert(result == 1);
   }
```

3.3. Exponent Monotonicity

Target	`fake_expo.eas`	**Category**	Property Test
Severity	N/A	**Status**	Passes

3.3.1. Description

Results must never decrease when the numerator (exponent) increases, this is called exponent monotonicity. To verify that this property holds, we compare results from two different numerator values, ensuring the output maintains or increases when using a larger numerator. This validates that:

1. The series expansion maintains proper growth characteristics
2. The approximation preserves basic exponential behavior
3. The algorithm's convergence behaves consistently across different input sizes

```
// Results should grow (or stay same) as numerator increases
function check_FakeExpo_baseOneMonotonic(uint256 num1, uint256 num2) public {
    vm.assume(num1 < num2);  // ensure first numerator is smaller
    vm.assume(num1 > 0 && num2 < type(uint256).max);

    uint256 result1 = callFakeExpo(1, int256(num1), 17);
    uint256 result2 = callFakeExpo(1, int256(num2), 17);

    assert(result1 <= result2);  // bigger input → bigger or equal output
}
```