Ethereum Foundation

c-kzg & go-kzg

Security Assessment Report

Version: 2.0

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Ethereum Foundation KZG implementations. The review focused solely on the security aspects of the source code, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the KZG implementations. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Ethereum Foundation KZG implementations contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Ethereum Foundation smart contracts.

Overview

An upcoming improvement to the Ethereum protocol is the introduction of EIP-4844. EIP-4844 introduces the KZG Polynomial Commitment Scheme.

The implementation in EIP-4844 is aimed at polynomials of degree 4096 where each coefficient or evaluation is in the finite field denoted by the prime `r`, where `r` is the size of the subgroup group use in BLS12-381 curves. The implementation uses BLS12-381 elliptic curve points for the elliptic curve discrete logarithm (ECDLP).

A trusted setup is required for the scheme. The trusted setup has not yet completed, contributions can be added here.

Scope of the review includes the following repositories including the bindings of `c-kzg-4844` into the languages Rust, Golang, Nim, Java, Node JS, C Sharp and Python.

- go-kzg-4844
- c-kzg-4844

Security Assessment Summary

This review was conducted on the files hosted on the crate-crypto/go-kzg-4844 and ethereum/c-kzg-4844 repositories. They were assessed at commits a201da1 and fd24cf8 respectively.

The manual code review section of the report is focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the source code. This includes their internal interactions, intended functionality and correct implementation with respect to the consensus specifications.

Additionally, differential fuzzing was performed between the two implementations. Differential fuzzing targets were designed to discover variances between the execution of clients given the same input.

Findings Summary

The testing team identified a total of 8 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- High: 2 issues.
- Low: 1 issue.
- Informational: 4 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Ethereum Foundation smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the source code are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
EKZG-01	Incorrect Deserialisation of BLS12-381 Points	Critical	Resolved
EKZG-02	Panics in `from_hex()` for Rust Bindings	High	Resolved
EKZG-03	Panics in `UnmarshalText()` for Go Bindings	High	Resolved
EKZG-04	Potential Panics Loading Trusted Setup	Low	Closed
EKZG-05	Lack of Validation of Parameter Length	Informational	Closed
EKZG-06	`NewDomain()` Will Panic for Certain Input	Informational	Closed
EKZG-07	Random Oracle for Batch Proofs may be Zero	Informational	Closed
EKZG-08	Miscellaneous General Comments	Informational	Closed

EKZG-01	Incorrect Deserialisation of BLS12-381 Points		
Asset	`ConsensysSys/gnark-crypto/ecc/bls12-381/marshal.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The upstream library ConsensysSys/gnark-crypto contains a bug in the decoding of BLS12-381 points. The error allows invalid points to be successfully decoded.

The three leading bits of BLS12-381 encoded points are used to determine if the bytes are compressed, the point at infinity and the sign of the y-coordinate. There are three cases which are invalid and should return an error.

- `0b111`
- `0b011`
- `0b001`

Points with these bit combinations are treated as valid compressed points and decoded successfully. Therefore, the `go-kzg` implementation will accept proofs which have an invalid bit combination. As a result, verification of the proofs may be successful for invalid encodings.

`go-kzg` and `c-kzg` will therefore have a consensus fault when verifying proofs which contain these bit combinations.

The issue occurs for points for each of the groups G1 and G2.

Recommendations

To resolve the issue update the library ConsensysSys/gnark-crypto such that it returns an error when decoding points with these bit combinations.

Resolution

A resolution can be seen in commit da59459. The resolution is, to return an error for each of the invalid bit combinations decoding points in both G1 and G2.

EKZG-02	Panics in `from_hex()` for Rust Bindings		
Asset	`c-kzg-4844/bindings/rust/src/bindings/mode.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

There are three `from_hex()` functions in the `c-kzg` rust bindings. Each of these functions contains an index out of bounds panic and a reachable `unwrap()`.

The functions decode hex strings for the following types:

- Blob
- Bytes32
- Bytes48

The following code snippet is taken from the decoding of `Blob`, however each of `Bytes32` and `Bytes48` contains the same issues.

```
pub fn from_hex(hex_str: &str) -> Result<Self, Error> {
    Self::from_bytes(&hex::decode(&hex_str[2..]).unwrap())
}
```

An index out of bounds panic will occur `hex_str` is less than two bytes due to `hex_str[2:]`. An `unwrap()` panic will occur if `hex_str` is not valid hex encoding. That is the length is not even length or consists of characters outside `0..f`.

Recommendations

To improve the robustness of these functions three actions should be implemented:

- ensure the length of `hex_str` is more than 2,
- ensure the first two bytes of `hex_str` are `0x` and
- remove the `unwrap()` and propagate an error if `hex::decode()` errors.

Resolution

Correct error handling has been added to `from_hex()` to remove potential panics. The resolution can be seen in PR #307.

EKZG-03	Panics in `UnmarshalText()` for Go Bindings		
Asset	`c-kzg-4844/bindings/go/main.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The functions `UnmarshalText()` for types `Bytes32`, `Bytes48` and `Blob` contain a potential index out of bounds panic.

The following is an excerpt from `Bytes32.UnmarshalText()`.

```
func (b *Bytes32) UnmarshalText(input []byte) error {
    if string(input[:2]) == "0x" { //@audit index array without validating length
        input = input[2:]
        }
        bytes, err := hex.DecodeString(string(input))
if err != nil {
    return err
}
if len(bytes) != len(b) {
    return ErrBadArgs
}
copy(b[:], bytes)
return nil
}
```

If the `input` length is less than 2 then `input[:2]` will cause an index out of bounds panic.

The issue can be seen in the following locations.

- Bytes32.UnmarshalText()
- Bytes48.UnmarshalText()
- Blob.UnmarshalText()

Recommendations

To improve the robustness of these functions validate the length of `input` is more than 2.

Resolution

The function `UnmarshalText()` for each of the listed types has been updated to use `strings.HasPrefix()` and `strings.RemovePrefix()`. The first two characters will be safely removed if the length of the string is greater than or equal to 2 and the characters are `"0x"`. Updates have been made in PR #306.

EKZG-04	Potential Panics Loading Trusted Setup		
Asset	`c-kzg-4844/bindings/go/main.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

A range of reachable panics exist in the Golang bindings related to the trusted setup.

There are direct panics in the function `LoadTrustedSetup()` which occur on malformed input:

- line [131]
- line [134]
- line [137]

Similarly, for `LoadTrustedSetupFile()`:

- line [162]
- line [170]

Additionally, the remaining API functions will panic if the trusted setup is not `loaded`.

- line [188]
- line [204]
- line [226]
- line [274]
- line [299]
- line [323]

Finally, when an unknown error type is returned from `c-kzg` a panic occurs on line [61].

Recommendations

It is recommended to propagate errors as opposed to panicking. Errors may be easily handed by the calling function and prevent program failure by safely exiting.

Resolution

The testing team have opted not to fix these issues as they only arise when there is a malformed trusted setup. The trusted setup should be shared and pre-validated before runtime and thus this represents a minimal security risk.

EKZG-05	Lack of Validation of Parameter Length
Asset	`c-kzg-4844/bindings/node.js/src/kzg.cxx`
Status	**Closed:** See Resolution
Rating	Informational

Description

There is insufficient validation on the number of input parameters before they are indexed in the node bindings.

The variable `info` is not validated before it is indexed, to fetch the input parameters. Each of the following lines contains unchecked indexing of `info`:

- line [176]
- line [213]
- line [249]
- line [298]
- line [343]
- line [393]
- line [448]

The impact is considered informational as the Typescript interface `kzg.d.ts` specifies the number of arguments that should be passed to each function.

Recommendations

To mitigate this issue add validation of `info.Length` for each of the binding functions to prevent an index out of bounds.

Resolution

The testing team have opted not to fix this issue as it does not pose a security risk.

EKZG-06	`NewDomain()` Will Panic for Certain Input
Asset	`go-kzg-4844/internal/kzg/domain.go`
Status	**Closed:** See Resolution
Rating	Informational

Description

There are multiple panic statements in the function `NewDomain()` which are triggered if invalid input is supplied.

Panics will occur if the input, x, is not a power of 2 i.e. $x \neq 2^a$ for some a. Additionally, a panic will occur if the input is a power of two greater than 32, which is the largest power of 2 that is a factor of $r - 1$.

The function is only called for constant input 4096 from `api.go` and `trusted_setup.go` and thus is raised as informational severity.

Recommendations

Consider propagating an error as opposed to panicking in invalid input is supplied.

Resolution

The testing team have opted not to fix this issue as it does not pose a security risk since each call to this function is supplied with constant input.

EKZG-07	Random Oracle for Batch Proofs may be Zero
Asset	`go-kzg-4844/internal/kzg/kzg_verify.go` & `consensus-specs/specs/deneb/polynomial-commitments.md`
Status	**Closed:** See Resolution
Rating	Informational

Description

The consensus specs generate a random oracle used to verify batch commitments. The random oracle is generated in the range `[0, q)` as it uses the output of the SHA256 hash function and takes the modulus of q.

Batch verification is seen in the following equation where r is the random oracle.

$$e(\sum r^i proof_i, [s]) == e(\sum r^i (commitment_i - [y_i]) + \sum r^i z_i proof_i, [1])$$

It is unadvisable to allow $r = 0$ as the random oracle. That is because each individual pairing in the batch has components multiplied by a power of the random oracle. Hence, if the random oracle is zero each pairing will contain the point at infinity and therefore verify as true, irrelevant of the correctness of the proof.

The issue is raised as informational as the probability of this occurring is approximately $\frac{1}{q}$ or $\frac{3}{2^{256}}$.

Recommendations

Consider preventing the random oracle from being zero. Alternative solutions are to use rejection sampling or setting the value to a fixed non-zero value if it is zero e.g. 2.

Resolution

The testing team have opted not to fix this issue as the probability for this edge case to arise is negligible and therefore not worth the additional complexity to the specifications and source code lines.

EKZG-08	Miscellaneous General Comments
Asset	*
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **BLST interface defines parameter as `bool` but takes `int`.**

   The function `blst_p1_cneg(POINTonE1 *a, int cbit)` expects `cbit` to be of type `int` in . However, the interface in blst.h has definition `void blst_p1_cneg(blst_p1 *p, bool cbit)`, where `cbit` is of type `bool`.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team have acknowledged these findings, addressing them where appropriate.

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact				
	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

σ'