wadeAlexC / A_DESCRIPTION.md Secret

Last active 3 weeks ago

A_DESCRIPTION.md

BLS "Malleability" PoC Description

Problem 1: BLS signature validation in lotus uses `blst` library method `VerifyCompressed`. This method accepts signatures in 2 forms: "serialized", and "compressed", meaning that BLS signatures can be provided as either of 2 unique byte arrays.

- (Link - `VerifyCompressed`): See `AggregateVerifyCompressed`, which accepts both signatures and public keys in serialized and compressed forms.
- (Link - `blsSigner.Verify`): Invokes `VerifyCompressed`

Problem 2: Lotus block validation functions perform a uniqueness check on provided blocks. Two blocks are considered distinct if the CIDs of their blockheader do not match. The CID method for blockheader includes the `BlockSig` of the block.

- (Link - `BlockHeader.ToStorageBlock`): Serializes the blockheader with `BlockSig` included

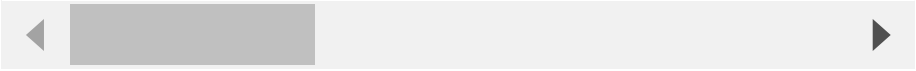
As a result: Two blocks that are identical in every way (except that one uses a "serialized" `BlockSig` and the other "compressed"), will be considered distinct blocks. These problems occur in at least 3 locations in lotus code:

1. `/chain/sync.go::ValidateBlock`:
   - Checks if the provided block has already been validated by comparing its CID against already-validated blocks (ref)
   - Checks block signature using `VerifyCompressed` (ref)
2. `/chain/vm/syscalls.go::VerifyConsensusFault`:
   - Compares two submitted blocks using their CID (ref)
   - Verifies both blocks' signatures using `VerifyCompressed` (ref)
3. `/chain/sub/incoming.go::Validate`:
   - Checks against blocks in cache using CID (ref)
   - Verifies block signature using `VerifyCompressed` (ref)

Remediation: Blocks should be checked for uniqueness without the inclusion of the `BlockSig`.

Notes: The code below is a POC that `VerifyCompressed` will accept signatures when provided in both forms: "serialized" and "compressed". The console output of the POC follows:

```
Verifying signature...
P2Affine.Verify: Valid!
================
Serialized:
04f423a63f915e347f19ef629741251e26518ced0a14d5130ed8760e0bf91c72f085ba6b984f15ef7f7ea9d3421fd4a910d40ec76a3a31452574234e
(len: 192)
================
Compressed:
a4f423a63f915e347f19ef629741251e26518ced0a14d5130ed8760e0bf91c72f085ba6b984f15ef7f7ea9d3421fd4a910d40ec76a3a31452574234e
(len: 96)
================
VerifyCompressed(sigSerialized, pkSerialized): Valid!
VerifyCompressed(sigSerialized, pkCompressed): Valid!
VerifyCompressed(sigCompressed, pkSerialized): Valid!
VerifyCompressed(sigCompressed, pkCompressed): Valid!
```



go.mod

```
module github.com/wadeAlexC/bls-poc

go 1.15

require github.com/supranational/blst v0.2.0
```

main.go

```
package main

import (
        "crypto/rand"
        "fmt"

        blst "github.com/supranational/blst/bindings/go"
)

type PublicKey = blst.P1Affine
```

```go
type Signature = blst.P2Affine
type AggregateSignature = blst.P2Aggregate
type AggregatePublicKey = blst.P1Aggregate

func main() {
	var ikm [32]byte
	_, _ = rand.Read(ikm[:])
	sk := blst.KeyGen(ikm[:])
	pk := new(PublicKey).From(sk)

	var dst = []byte("BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_NUL_")
	msg := []byte("BLS-MALLEABILITY-POC")
	sig := new(Signature).Sign(sk, msg, dst)

	fmt.Println("Verifying signature...")

	fmt.Printf("P2Affine.Verify: ")
	if !sig.Verify(pk, msg, dst) {
		fmt.Println("ERROR: Invalid!")
	} else {
		fmt.Println("Valid!")
	}

	fmt.Println("================")

	// Serialize and compress signatures and pks
	sigSerialized := sig.Serialize()
	sigCompressed := sig.Compress()
	pkSerialized := pk.Serialize()
	pkCompressed := pk.Compress()

	// Print signature bytes. Length must be under 200 to meet CBOR unmarshal restrictions
	fmt.Printf("Serialized: %x\n(len: %d)\n", sigSerialized, len(sigSerialized))
	fmt.Println("================")
	fmt.Printf("Compressed: %x\n(len: %d)\n", sigCompressed, len(sigCompressed))
	fmt.Println("================")

	// 1. VerifyCompressed with serialized signature / serialized PK
	fmt.Printf("VerifyCompressed(sigSerialized, pkSerialized): ")
	if !new(Signature).VerifyCompressed(sigSerialized, pkSerialized, msg, dst) {
		fmt.Println("ERROR: Invalid!")
	} else {
		fmt.Println("Valid!")
	}

	// 2. VerifyCompressed with serialized signature / compressed PK
	fmt.Printf("VerifyCompressed(sigSerialized, pkCompressed): ")
	if !new(Signature).VerifyCompressed(sigSerialized, pkCompressed, msg, dst) {
		fmt.Println("ERROR: Invalid!")
	} else {
		fmt.Println("Valid!")
	}

	// 3. VerifyCompressed with compressed signature / serialized PK
	fmt.Printf("VerifyCompressed(sigCompressed, pkSerialized): ")
	if !new(Signature).VerifyCompressed(sigCompressed, pkSerialized, msg, dst) {
		fmt.Println("ERROR: Invalid!")
	} else {
		fmt.Println("Valid!")
	}

	// 4. VerifyCompressed with compressed signature / compressed PK
	fmt.Printf("VerifyCompressed(sigCompressed, pkCompressed): ")
	if !new(Signature).VerifyCompressed(sigCompressed, pkCompressed, msg, dst) {
		fmt.Println("ERROR: Invalid!")
	} else {
		fmt.Println("Valid!")
	}
}
```