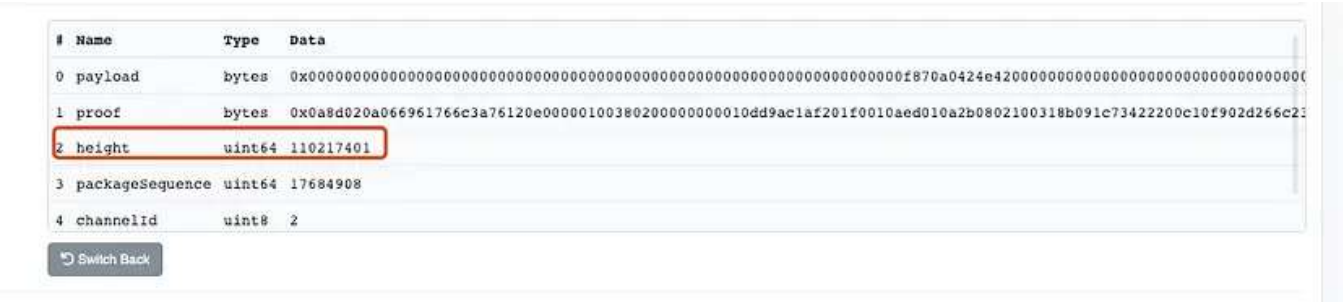
BNB Chain’s $850 Million Hack — Using Beosin Trace to Investigate the Stolen Funds

On Oct 7, Beosin EagleEye monitored that BNB Chain’s cross-chain bridge BSC Token Hub was exploited for about $850 million, including $710 million on BSC and $143 million of bridged assets.

Attack Flow

BSC Token Hub uses a special pre-compiled contract for validating IAVL trees when performing cross-chain transaction verification. There is a bug in its implementation which may allow an attacker to forge arbitrary messages.

1. The attacker first picks the hash of a successfully submitted block (specified block: 110217401).



2. Then construct an attack payload as a leaf node to verify IAVL tree.

3. Add an arbitrary new leaf node to the IAVL tree.

4. At the same time, add a blank internal node to satisfy the implementation proof.

```
func (proof *RangeProof) computeRootHash() (rootHash []byte, treeEnd bool, err error) {
        if len(proof.Leaves) == 0 {
                return nil, false, cmn.ErrorWrap(ErrInvalidProof, "no leaves")
        }
        if len(proof.InnerNodes)+1 != len(proof.Leaves) {
                return nil, false, cmn.ErrorWrap(ErrInvalidProof, "InnerNodes vs Leaves length mismatch, leaves should be 1 more.")
        }

        // Start from the left path and prove each leaf.

        // shared across recursive calls
        var leaves = proof.Leaves
        var innersq = proof.InnerNodes
        var COMPUTEHASH func(path PathToLeaf, rightmost bool) (hash []byte, treeEnd bool, done bool, err error)

        // rightmost: is the root a rightmost child of the tree?
        // treeEnd: true iff the last leaf is the last item of the tree.
        // Returns the (possibly intermediate, possibly root) hash.
        COMPUTEHASH = func(path PathToLeaf, rightmost bool) (hash []byte, treeEnd bool, done bool, err error) {

                // Pop next leaf.
                nleaf, rleaves := leaves[0], leaves[1:]
                leaves = rleaves

                // Compute hash.
                hash = (pathWithLeaf{
                        Path: path,
                        Leaf: nleaf,
                }).computeRootHash()

```

5. Adjust the leaf node added in step 3 so that the computed root hash is equal to the correct root hash selected in step 1 for a successful commit.

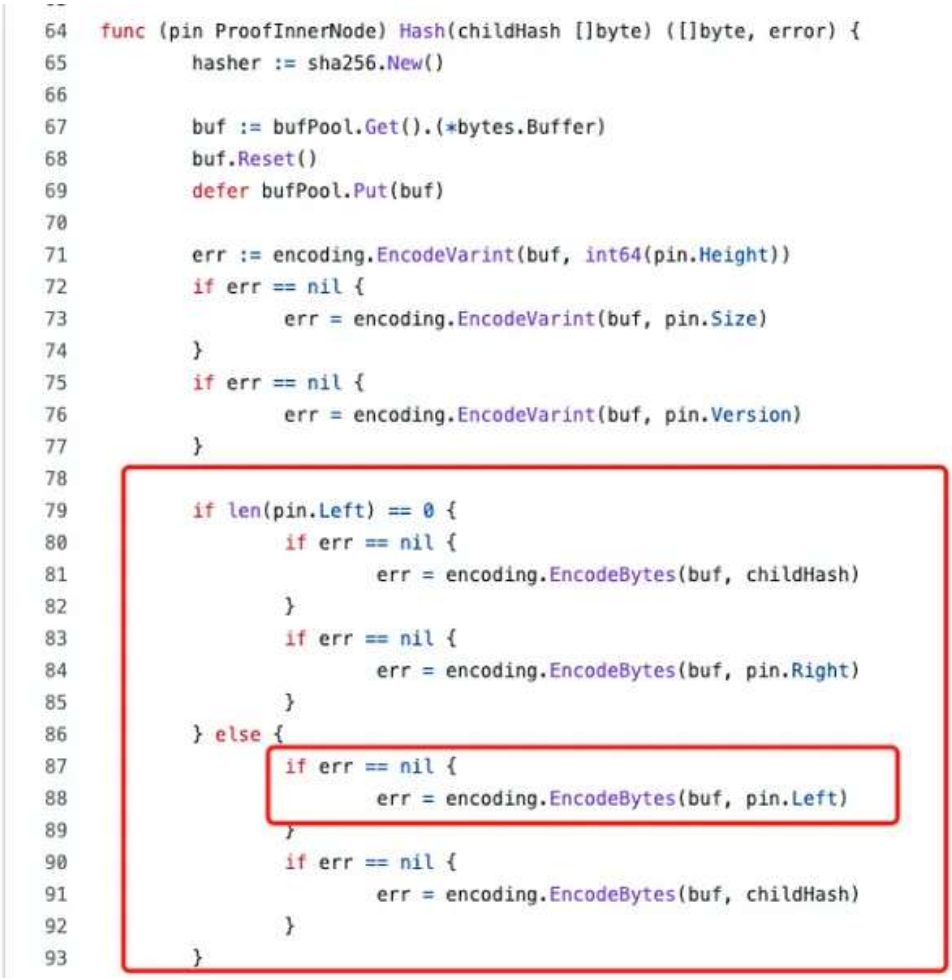
```
                // Recursively verify inners against remaining leaves.
                derivedRoot, treeEnd, done, err := COMPUTEHASH(inners, rightmost && rpath.isRightmost())
                if err != nil {
                        return nil, treeEnd, false, errors.Wrap(err, "recursive COMPUTEHASH call")
                }

                if !bytes.Equal(derivedRoot, lpath.Right) {
                        return nil, treeEnd, false, errors.Wrapf(ErrInvalidRoot, "intermediate root hash %X doesn't match, got %X", lpath.Right, derived
                }

                if done {
                        return hash, treeEnd, true, nil
                }
        }

```

```
// Run executes a multi-store proof operation for a given value. It returns
// the root hash if the value matches all the store's commitID's hash or an
// error otherwise.
func (op MultiStoreProofOp) Run(args [][]byte) ([][]byte, error) {
        if len(args) != 1 {
                return nil, cmn.NewError("Value size is not 1")
        }

        value := args[0]
        root := op.Proof.ComputeRootHash()

        for _, si := range op.Proof.StoreInfos {
                if si.Name == string(op.key) {
                        if bytes.Equal(value, si.Core.CommitID.Hash) {
                                return [][]byte{root}, nil
                        }

                        return nil, cmn.NewError("hash mismatch for substore %v: %X vs %X", si.Name, si.Core.CommitID.Hash, value)
                }
        }

        return nil, cmn.NewError("key %v not found in multistore proof", op.key)
}

```

6. Finally construct the withdrawal proof for that particular block (110217401).

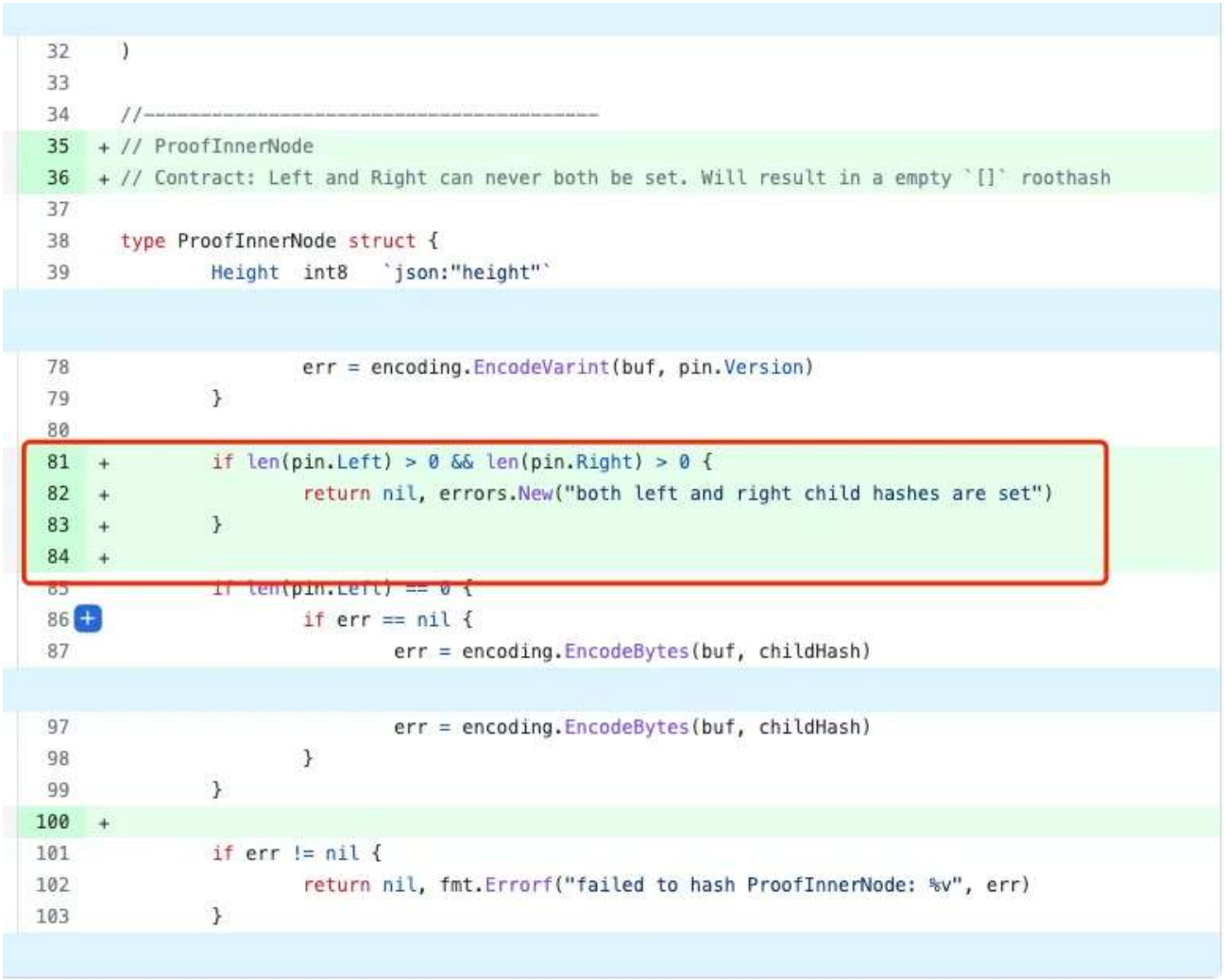
Vulnerability

In the branch where len(pin.Left) is not 0, the hash calculation does not use the pin.Right data. The hacker uses this vulnerability to construct data and add proof.LeftPath[1].Right data, but this data is not involved in the hash calculation.



```
64  func (pin ProofInnerNode) Hash(childHash []byte) ([]byte, error) {
65          hasher := sha256.New()
66
67          buf := bufPool.Get().(*bytes.Buffer)
68          buf.Reset()
69          defer bufPool.Put(buf)
70
71          err := encoding.EncodeVarint(buf, int64(pin.Height))
72          if err == nil {
73                  err = encoding.EncodeVarint(buf, pin.Size)
74          }
75          if err == nil {
76                  err = encoding.EncodeVarint(buf, pin.Version)
77          }
78
79          if len(pin.Left) == 0 {
80                  if err == nil {
81                          err = encoding.EncodeBytes(buf, childHash)
82                  }
83                  if err == nil {
84                          err = encoding.EncodeBytes(buf, pin.Right)
85                  }
86          } else {
87                  if err == nil {
88                          err = encoding.EncodeBytes(buf, pin.Left)
89                  }
90                  if err == nil {
91                          err = encoding.EncodeBytes(buf, childHash)
92                  }
93          }
```

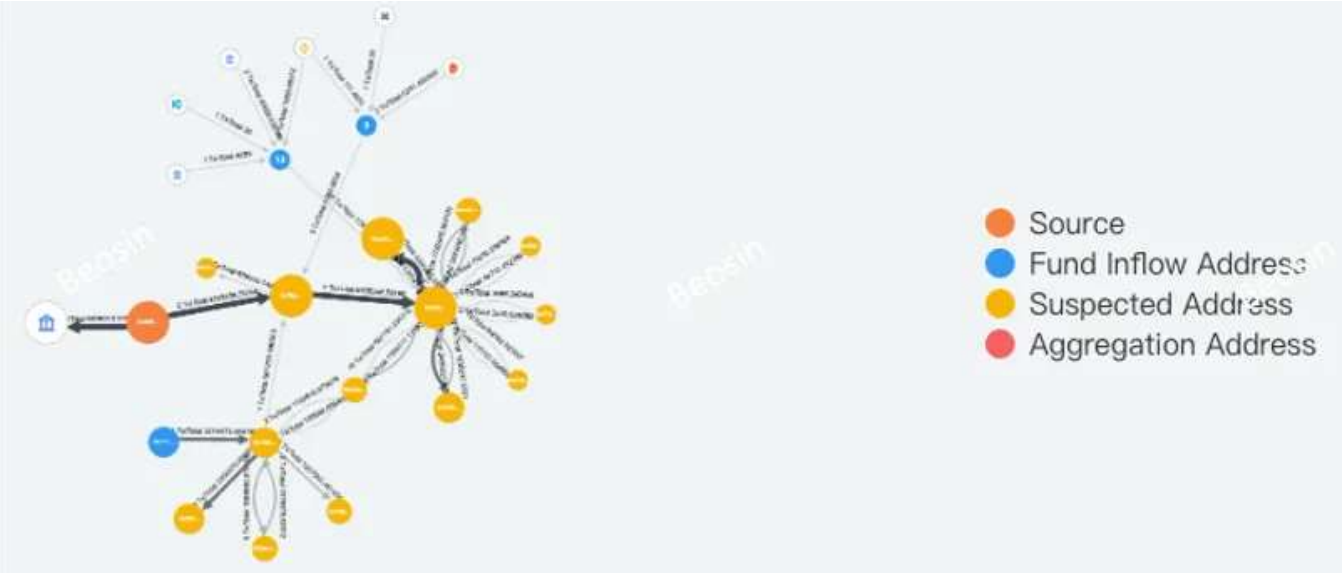
The current version has fixed.



```
32    )
33
34    //----------------------------------------
35  + // ProofInnerNode
36  + // Contract: Left and Right can never both be set. Will result in a empty `[]` roothash
37
38    type ProofInnerNode struct {
39            Height  int8   `json:"height"`

78                    err = encoding.EncodeVarint(buf, pin.Version)
79            }
80
81  +         if len(pin.Left) > 0 && len(pin.Right) > 0 {
82  +                 return nil, errors.New("both left and right child hashes are set")
83  +         }
84  +
85            if len(pin.Left) == 0 {
86                    if err == nil {
87                            err = encoding.EncodeBytes(buf, childHash)

97                            err = encoding.EncodeBytes(buf, childHash)
98                    }
99            }
100 +
101           if err != nil {
102                   return nil, fmt.Errorf("failed to hash ProofInnerNode: %v", err)
103           }
```

Fund Trace

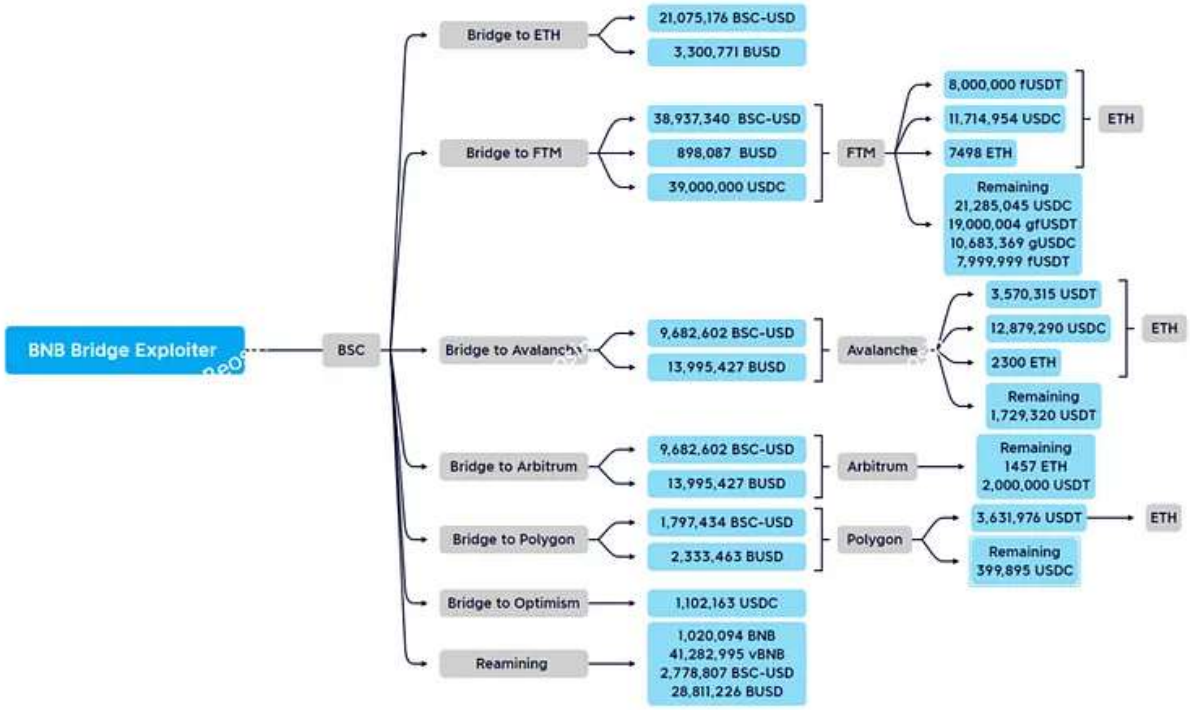
Using Beosin Trace — a crypto investigation and compliance tool, Beosin security team found that BSC Token Hub exploiter had bridged about $143.57M of the stolen funds to other chains (including lending). About $77.39M were bridged to Ethereum, ~$58.96M to Fantom (including gUSDT), ~$17.2M to Avalanche, ~$4M to Arbitrum, ~$400K to Polygon and ~$1.1M to Optimism.



Trace Map from Beosin Trace

Trace Map from Beosin Trace

Trace Map from Beosin Trace



Funding statistics from Beosin Security Team

BNB Chain Back Online

BNB Smart Chain (BSC) resumed operations at around 06:40 Coordinated Universal Time (UTC). Beosin security team find that the current BSC node blocks the flow of stolen funds and potential attacks by blacklisting and suspending the iavlMerkleProofValidate feature.

```
+ package types
+
+ import "github.com/ethereum/go-ethereum/common"
+
+ // This is introduced because of the Tendermint IAVL Merkel Proof verification
  exploitation.
+ var NanoBlackList = []common.Address{
+         common.HexToAddress("0x489A8756C18C0b8B24EC2a2b9FF3D4d447F79BEc"),
+         common.HexToAddress("0xFd6042Df3D74ce9959922FeC559d7995F3933c55"),
+         // Test Account
+         common.HexToAddress("0xdb789Eb5BDb4E559beD199B8b82dED94e1d056C9"),
+ }
```

```
137 +
138 + // tmHeaderValidate implemented as a native contract.
139 + type tmHeaderValidateNano struct{}
140 +
141 + func (c *tmHeaderValidateNano) RequiredGas(input []byte) uint64 {
142 +         return params.TendermintHeaderValidateGas
143 + }
144 +
145 + func (c *tmHeaderValidateNano) Run(input []byte) (result []byte, err error) {
146 +         return nil, fmt.Errorf("suspend")
147 + }
148 +
149 + //------------------------------------------------------------------------------
    ------------------------------------------------------------
150 + type iavlMerkleProofValidateNano struct{}
151 +
152 + func (c *iavlMerkleProofValidateNano) RequiredGas(input []byte) uint64 {
153 +         return params.IAVLMerkleProofValidateGas
154 + }
155 +
156 + // input:
157 + // | payload length | payload    |
158 + // | 32 bytes       |            |
159 + func (c *iavlMerkleProofValidateNano) Run(input []byte) (result []byte, err error) {
160 +         return nil, fmt.Errorf("suspend")
161 + }
```

Cross-chain Bridge Security

Cross-chain bridges are usually projects with large amount of assets. The more code volume is, the more likely that some combinations of multiple vulnerabilities may occur. A high-risk point of cross-chain bridge is the off-chain security, because the on-chain and off-chain codes are generally audited separately, and off-chain security is usually guaranteed by the project itself, resulting in many vulnerabilities to be ignored.

Previous cross-chain bridge attacks are mainly from off-chain vulnerabilities or private key compromise. This time the exploit is through the construction of a specific root hash to construct a specific block of withdrawal proof. The attack is more sophisticated, and the amount is also higher than in the past. This incident also reminds us that vulnerabilities are often found in places we don't expect, so the only thing is to keep improving the security of the project and discover these bugs earlier than those malicious actors.

For more cross-chain bridge related articles, please check our website at https://beosin.com/resources/

Contact

If you have need any blockchain security services, please contact us:

Website Email Official Twitter Alert Telegram LinkedIn