CERTORA

Formal Verification of FullyBackedBonding

Summary

This document describes the specification and verification of Keep using Certora Prover. The work was undertaken from Nov 2-16, 2020. The latest commit that was reviewed and run through the Certora Prover was 658b02f237e25c370feaba80a7a3b13824190c4.

We verified the contract FullyBackedBonding. An owner of ETH can delegate it to a *keep operator* by depositing it in FullyBackedBonding. Bonds are locked by application (*keep owner*), and when released, they stay in the *bonding contract*. The scope of verification was for ETH-only keep operators.

The Certora Prover proved the implementation of the FullyBackedBonding contract is correct with respect to the formal rules written by Keep and the Certora teams. During the verification process, the Certora Prover discovered minor bugs in the code listed in the table below. All issues were promptly corrected, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The Certora development team is currently handling these limitations. The next section formally defines high level specifications of FullyBackedBonding. All the rules are publically available in a public github: https://github.com/keep-network/keep-ecdsa/tree/certora/fullybackedbonding-spec/solidity/specs.

Main Issues Discovered

Issue	**Rule broken**	**Description**	**Severity**	**Mitigation**
System accepts invalid parameters	validOperatorState	Operator can be set to zero.	Low	Fixed code, added require
System accepts invalid parameters	validOperatorState	Authorizer can be set to zero, resulting in a bond that cannot be authorized and used in a keep.	Low	Fixed code, added require

Denial of service on cyclic authorization	No cyclic authorization	A cyclic authorization setting can be made, causing stack overflow and denial of service on subsequent operations.	Low	In current version, cyclic authorization is impossible as the user authorized the high level contract only

Disclaimer

The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Importantly, the guarantees of the Certora Prover are scoped to the provided specification, and any cases not covered by the specification are not checked by the Certora Prover.

We hope that this information is useful, but provide no warranty of any kind, express or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Notations

1. ✔ indicates the rule is formally verified on the latest commit. We write ✔* when the rule was verified on a simplified version of the code (or under some assumptions).
2. ✍ indicates the rule is not yet formally specified.
3. 🔁 indicates the rule is postponed.
4. We use Hoare triples of the form {p} C {q}, which means that if the execution of program C starts in any state satisfying p, it will end in a state satisfying q. In Solidity, p is similar to require, and q is similar to assert.
   The syntax {p} ($C_1 \sim C_2$) {q} is a generalization of Hoare rules, called relational properties. {p} is a requirement on the states before C_1 and C_2, and {q} describes the states after their executions. Notice that C_1 and C_2 result in different states. As a special case, $C_1 \sim_{op} C_2$, where op is a getter, indicating that C_1 and C_2 result in states with the same value for op.

Verification of FullyBackedBonding

Overview of scope

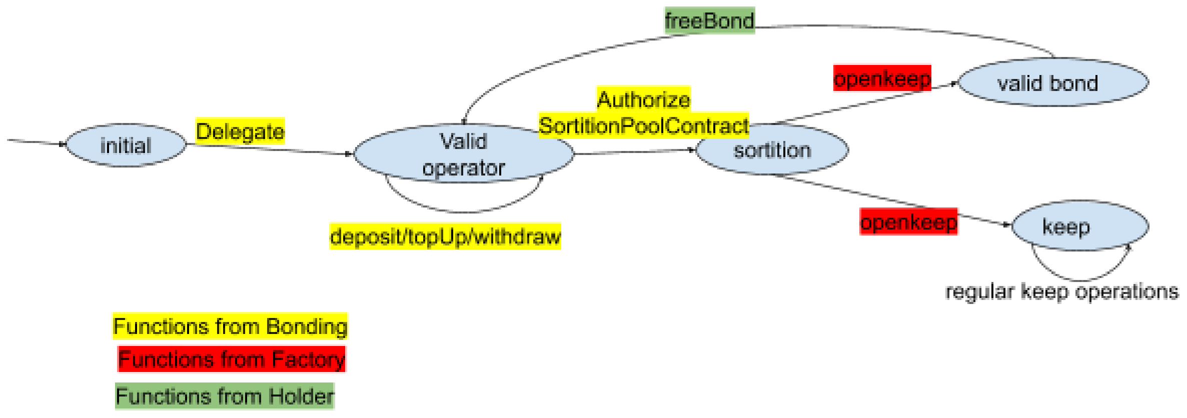

Figure 1: A finite state machine illustrating the functionality of FullyBackBonding.

Figure 1 displays a finite state machine which explains the contract's functionality.
The keep owner, when delegating, appoints three addresses: an operator, a beneficiary, and an authorizer. Bonds are locked by application (keep owner), and when released, they stay in the bonding contract. Unbonded value can be withdrawn from the bonding contract to the operator's beneficiary. A keep owner can seize a bond as a result of the operator's misbehavior.

The Functions section lists the view functions used in the verification process. Some functions that do not exist in the source code were added by ghost and harness variables.

The Operations section lists the functions we verified. As a convention, we assume a function succeeds and returns true unless stated otherwise. We use the notation *op(u, x)* to denote any operation instantiated by *u* (as *msg.sender)* with sending of *x* (in *msg.value)*.

We denote an operator as o.

Invariants

Invariants are properties that should hold after every contract operation.

1. **Valid Operator ✔**
   The operator cannot be zero.

   ```
   beneficiaryOf(o) ≠ 0  ⇔
      ( o ≠ 0 ∧ ownerOf(o) ≠ 0  ∧ authorizerOf(o) ≠ 0 )
   ```

2. **Valid state of an Operator ✔**
   An operator holding assets must have an owner, a beneficiary, and an authorizer.

   ```
   (unbondedValue(o) + totalLockedBonds(o)) > 0 ⇒
          ( ownerOf(o) ≠ 0 ∧ beneficiaryOf(o) ≠ 0 ∧ authorizerOf(o) ≠ 0 )
   ```

3. **No bankruptcy of the system (no money lost) ✔***
   The balance of the system is more than its obligations (the total assets deposited in the system and is in either an unbounded or locked state).

   $$\text{ethSystemBalance} \geq \sum_{o \in operator} (\ \text{unbondedValue}(o) + \text{totalLockedBonds}(o)\ )$$

4. **User cannot gain assets ✔***
   The total assets of an operator o cannot be more than ever deposited to o. Note that a strict inequality may hold since a holder may seize a bond of o.

   ```
   everDeposited(o) ≥ ( unbondedValue(o) + totalLockedBonds(o) )
   ```

5. **No cyclic authorization 🔁**
   (This invariant does not need to hold on the current implementation.)
   There is no cycle in delegating authorities, as it can cause a denial of service.

   ```
   ¬ ( delegatedAuthority(a) = b ∧ delegatedAuthority(b) = c ∧
         delegatedAuthority(c) = a )
   ```

   This property can be generalized for cycles of any length.

Properties

6. **Total assets are preserved ✔***

The total assets of a user is the sum of assets within the system (either locked or unbounded) and outside the system.

The total assets of an operator within the system are preserved, except on deposits and on withdrawal and seizing.

```
{  b = unbondedValue(o) + totalLockedBonds(o) }
       op(u, x)
{ b =  unbondedValue(o) + totalLockedBonds(o) + x }
```

Note that this property holds for every successful operation performed by any user or holder, *u*, possibly sending in *x* amount of wei (msg.value), except for sizeBond and withdraw which are defined differently:

On withdraw, the total assets within the system of operator and the balance of the beneficiary is preserved

```
{ b = unbondedValue(o) + totalLockedBonds(o) + beneficiaryOf(o).balance}
       withdraw(x, o)
{ b = unbondedValue(o) + totalLockedBonds(o) + beneficiaryOf(o).balance}
```

Upon seizing a bond, the total assets between the operator's system and the destination address receiving the bond value are preserved.

```
{ b = unbondedValue(o) + totalLockedBonds(o) + destination.balance }
       seizeBond(o, ref, x, destination)
{ b = unbondedValue(o) + totalLockedBonds(o) + destination.balance }
```

7. **Integrity of withdrawal ✔**
   A successful withdrawal of a value x of an operator decreases the operator's value by x and transfers x to the operator's beneficiary. The maximal withdrawal is limited to the total amount the operator has deposited.

```
{
        beneficiary = beneficiaryOf(o) ∧
        u = unbondedValue(o)           ∧
        b = beneficiary.balanceOf()    ∧
        s = ethSystemBalance()
}
        withdraw(x, o)
{
      u - x = unbondedValue(o)     ∧
      b + x = beneficiary.balanceOf() ∧
      s - x = ethSystemBalance()   ∧
      x ≤ everDeposited(o)
}
```

8. **Maximal withdraw ✔**
   When withdrawing all deposits of an operator, the assets of that operator are zeroed out.

```
withdraw(everDeposited(o), o) ⟹
        ( unbondedValue(o) = 0 ∧ totalLockedBonds(o) = 0 )
```

9. **Additivity of withdraw ✔**
   Withdrawing is additive: the sum of two withdrawals is identical to a withdrawal of the sum.

```
( withdraw(x, o); withdraw(y, o) ) ~ withdraw(x+y, o)
```

   Here we expect the effect of withdrawing x and then withdrawing y to be the same as withdrawing them simultaneously. The correctness of this rule on all inputs increases the confidence that the protocol is less fragile, e.g., to rounding errors.

10. **No front running on withdraw ✔**

Withdrawals from different operators is independent. If one can withdraw x from the unbounded amount of operator o_1, then she should also be able to withdraw x after another user has performed an operation as operator o_2.

$$\{ o_1 \neq o_2 \} \quad r_1 = \texttt{withdraw}(x,\ o_1) \quad \sim_{r1\ =\ r2} \ (\texttt{op}(o_2,\ y);\ r_2 = \texttt{withdraw}(x,\ o_1))$$

Here we compare two arbitrary executions of the program: one with a single withdrawal and another in which a different operator performs a Keep operation. We require that if one withdrawal succeeds, so will the other and vice versa.

11. **Deposit and withdraw are inverse functions ✔**

Withdraw is the inverse function of deposit concerning the system's balance and an operator's unbounded value.

```
{
   u = unbondedValue(o)    ∧
   b = ethSystemBalance()
}

( deposit(x, o); r = withdraw(x, o) )

{
   r                       ∧
   u = unbondedValue(o)    ∧
   b = ethSystemBalance()
}
```

12. Valid change to BalanceOf ✔

On Keep operation op by user *s* sending *x* amount, the ETH balance of a user (operator or not) changes as follows:

```
{ b = o.balance }

op(s,x)

{  b = o.balance ⋁
   ( b + x = o.balance ⋀ isDepositOp(op) ) ⋁
   ( b ≤ o.balance   ⋀  (isSeizeBond(op) || isWithdraw(op) ) )
}
```

where `isDepositOp` is true for any of the deposit operations (deposit, delegate or topup).

13. Valid change to totalLockedBonds ✔*

The total locked bonds of an operator o change as follows:

```
{
   u = unbondedValue(o)   ∧
   l = totalLockedBonds(o)
}
       op
{
  totalLockedBonds(o) = l                                       ⋁
  ( l ≥ totalLockedBonds(o) ∧ isSeizeBond(op) ⋁ isFreeBond(op)) ⋁
  ( l ≤ totalLockedBonds(o) ≤ l + u ∧ isCreateBond(op) )
}
```

14. Valid change to unboundedValue ✔

On Keep operation op by user *u* sending *x* amount, the unbounded value of operator o changes as follows:

```
{  u = unbondedValue(o) }

       op(s, x)

{
    u = unbondedValue(o)                                  ⋁
  ( u + x = unbondedValue(o) ∧ isDepositOp(op) )           ⋁
  ( u ≥ unbondedValue(o) ∧ isWithdraw(op) ∧ (s=o ⋁ u=owner(s) )  ⋁
  ( u ≥ unbondedValue(o) ∧ isCreateBond(op) )
}
```

15. Valid change to everDeposited ✔

On Keep operation op by user *u* sending *x* amount, the unbounded value of operator o changes as follows:

```
{   e = everDeposited(o) }
       op(s,x)
{
    e = everDeposited(o)                        ⋁
  ( e + x = everDeposited(o) ∧ isDepositOp(op) )
}
```

16. CreateBond and freeBond are inverse operations 🔁

17. Only the holder can manipulate the Bond 🔁

18. Deposit is possible (no denial of service) 🔁

19. Additivity of deposit 🔁

20. Integrity of deposit 🔁

21. No change to others 🔁

An operation regarding operator o_1 does not affect operator o_2.

Functions

ethSystemBalance() : uint
Returns the current ETH balance deposited in the system (namely, the FullyBackedBonding contract).

totalLockedBonds(address operator) : uint
Returns the total amount of locked bonds of **operator** across all bonds **operator** is participating in.

everDeposited(address operator) : uint
Returns the total amount deposited to **operator.**

unbondedValue(address operator) : uint
Returns the amount of ETH that **operator** has unlocked and is available for withdrawal.

ownerOf(address operator) : address
Returns the owner of **operator**.

authorizerOf(address operator) : address
Returns the authorizer of **operator**.

beneficiaryOf(address operator) : address
Returns the beneficiary of **operator.**

delegatedAuthority(address u) : address
Returns the immediate delegated authority address of **u**.

Operations

delegate(address owner, address operator, address beneficiary, uint amount) : bool
Returns true when successfully registers a new **owner**, **operator**, **beneficiary** and authorizer, and deposits **amount.**

deposit(address sender, uint x, address operator) : bool
Returns true when successfully deposits **x** amount of wei to **operator**’s bond.

topUp(address sender, uint x, address operator) : bool
Returns true when successfully deposits **x** amount of wei to **operator**’s bond

withdraw(uint x, address operator) : bool
Returns true when successfully withdraws **x** amount of wei from **operator**’s bond.

createBond(address operator, address holder, uint referenceID,
uint amount, address authorizedSortitionPool) : bool
Creates a bond for the given **operator**, **holder**, **referenceID** and **amount.** Returns true if the bond was created successfully, and false otherwise.

reassignBond(address holder, address operator, uint referenceID ,
address newHolder ,uint newReferenceID) : bool
Reassigns the bond **referenceID** from **holder** to **newHolder** under a **newReferenceID**. Returns true if the bond was reassigned successfully, and false otherwise.

freeBond(address holder, address operator, uint referenceID) : bool
Frees the bond **referenceID** and allows **operator** to withdraw. Returns true if the bond was freed successfully, and false otherwise.

seizeBond(address holder, address operator, uint referenceID,
uint x, address destination) : bool
Seizes **x** amount of a bond **referenceID** as a result of **operator**'s misbehaviour and sends it to **destination**. Returns true if the value was seized successfully, and false otherwise.

authorizeSortitionPoolContract(address, address) : bool
Authorizes sortition pool for the provided operator. Returns true if the pool was created successfully, and false otherwise.

authorizeOperatorContract(address operator, address contract) : void
Authorizes **operator**'s **contract** to access staked token balance of the provided **operator**.

claimDelegatedAuthority(address source) : void
Grant the sender the same authority as **source**.