Verifying programs with Complex data structures using Coq

Kenneth Roe http://www.cs.jhu.edu/~roe The Johns Hopkins University

Software development with formal methods

Traditional

Our Approach

Finding and fixing bugs is the most difficult part of software development

- Many bugs can be traced to data structure invariant violations
- Documenting and verifying these invariants provides an important new approach to program verification

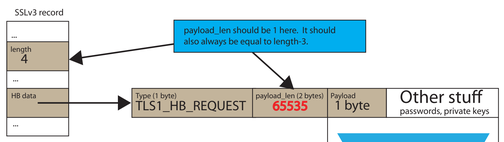
Preventing Heartbleeds

Logo from heartbleed.com

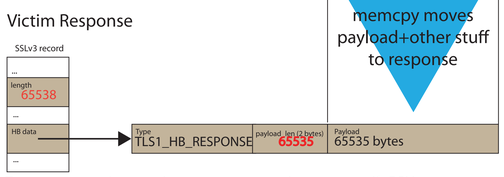
Heartbleed is a bug in OpenSSL code
Thousands of websites use OpenSSL for HTTPS
Websites vulnerable to sensitive data stealing attacks
Bug is in newly added heartbeat code
- A heart beat is sent once every few seconds by one side of an SSL connection to check if the other side is alive
- The other side responds with a message echoing the payload data

Exploitation of this bug is shown below
- Message has broken payload_len field
- Response constructed based on payload_len field
- No bounds check on memcpy
- Data beyond end of allocated block copied over

Attacker request



Victim Response



The theorem proving techniques presented on this poster to verify DPLL could also be applied to OpenSSL and would have almost certainly found the Heartbleed bug. We suggest two approaches as to how our techniques could be used to block Heartbleed.

1) Since packets being received cannot be trusted, add sanity checks to verify the invariants. Formal methods could be used to verify that the sanity checks work and to verify that once the invariants are satisfied, that unauthorized information cannot leak out
2) Separation logic can be used to add a pre-condition to memcpy that insures that it does not copy beyond the end of the record which the source pointer references. An unchecked parameter to memcpy was key to the Heartbleed bug. Formal methods could verify that the parameters to memcpy are sound.

Our approach, step 1: Write program and invariants

Below is shown a tree traversal program that has been converted from C to the PEDANTIC imperative langauge and its pre- and post condition. The user needs to create the pre- and post- conditions.

```
(*{{AbsExistsT v0 .
   (AbsExistsT v1.
     (AbsExistsT v2.
       (TREE(R, v0, 2, [0,1]) *
        TREE(I, v1, 2, [0]) *
        TREE(P, v2, 2, [0]) *
        AbsAll v3 in TreeRecords(v1). ([nth(find(v1, v3), 2) inTree v0]) *
        AbsAll v3 in TreeRecords(v2). ([nth(find(v2, v3), 2) inTree v0]) *
        [T == 0 \/ T inTree v0])))}}
WHILE not (T == 0)
DO N := P;
  NEW P, 2;
  *(P + 1) := T;
  *P := N;
  Tmp_I := *T;
  Tmp_r := *(T + 1);
  IF (Tmp_I == 0) and (Tmp_r == 0)
  THEN IF I == 0 THEN T ::= 0
       ELSE T := *(I + 1);
            Tmp_I := *(I + A0); DELETE I, 2; I ::= Tmp_I
  ELSE IF Tmp_I == 0 THEN T ::= *(T + 1)
       ELSE IF Tmp_r == A0 THEN T ::= *T
            ELSE N := I;
                 I := NEW 2;
                 *(I + 0) := N;
                 Tmp_I := *(T + 1);
                 *(I + 1) := (!Tmp_I); T ::= *T LOOP
{{AbsExistsT v0 .
   (AbsExistsT v1.
     (AbsExistsT v2.
       (TREE(R, v0, 2, [0,1]) *
        TREE(I, v1, 2, [0]) *
        TREE(P, v2, 2, [0]) *
        AbsAll v3 in TreeRecords(v2), ([nth(find(v2, v3), 2) inTree v0]) *
        [T == 0])))}}*)
```

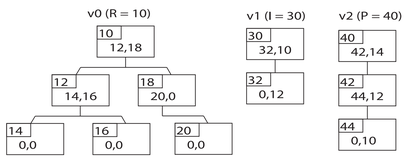
Pre-condition

Program

Post-condition

Coq Verification goal

Heap snapshot



Invariant

(1) There is a tree and two lists
all properly formed--no cycles, no dangling points
AbsExistsv0.AbsExistsv1.AbsExistsv2.
TREE(R, v0,2,[0,1])*TREE(I, v1,2,[0])*TREE(P, v2,2,[0])*

(2) Each of the nodes in the two lists contains a pointer a node in the tree.
AbsAll v3 in TreeRecords(v1).
[nth(find(v1, v3),2)inTreev0]*
AbsAll v3 in TreeRecords(v2).
[nth(find(v2, v3),2)inTreev0]*
[T=0\/T inTreev0]

Functional Equivalent Representation

- Characterize data fields saved, structure of the tree or list and the locations of records on the heap.
- Represented by lists whose elements are either naturals or (recursively) lists of lists or naturals.
- For the snapshot from the picture above, v0 would for example be the list
[10,([12,[14,[0],[0]],[16,[0],[0]]]),([18,[20,[0],[0]],[0]])]

Step 2: top level verification

A top level proof that propagates the invariant is created
- Much of this work can be automated
- The user needs to supply
  - while loop invariants
  - results from merging if-then-else branches

```
Theorem loopInvariant :
  ({afterInitAssigns}|loop({afterWhile return nil with AbsNone}).
Proof.
  (* Break up the while portion of the loop *)
  unfold loop. unfold afterWhile. unfold afterInitAssigns.

  (* WHILE ALnot (IT === A0) DO *)
  eapply strengthenPost.
  eapply whileThm with (invariant := loopInv). unfold loopInv.
  eapply strengthenPost. simpl.

  (* N := !P; *)
  eapply compose. pcrunch.

  (* NEW P,ANum(Size_I);*)
  eapply compose. pcrunch.

  simp. simp. simp.

  (* CStore ((!P)+++(ANum F_p)) (IT)) *)
  eapply compose. eapply store. compute. reflexivity. compute. reflexivity.
  intros. eapply treeRef1. apply H. apply H0.


  (* CStore ((!P)+++(ANum F_n)) (!N)) *)
  eapply compose.  eapply store. compute. reflexivity. compute. reflexivity.
  eapply treeRef2. (*apply H. apply H0.*)

  simp.

  (* CLoad Tmp_I ((IT)+++ANum(F_I)) *)
  eapply compose. pcrunch.

  (* CLoad Tmp_r ((IT) +++ A1) *)
  eapply compose. pcrunch.

  (* IF (ALand (!Tmp_I === A0) (!Tmp_r === A0)) *)
  eapply if_statement. simpl.

    (* IF (II === A0) *)
    eapply if_statement. simpl.

      (* T := A0 *)
      pcrunch.

    (* ELSE *)

      (* CLoad T (II)++A1 *)
      eapply compose. pcrunch.

      (* CLoad Tmp_I (II)++A0 *)
      eapply compose. pcrunch.

      (* DELETE II, A2 *)
      eapply compose.
      Set Printing Depth 200. pcrunch.
      eapply deleteExists1. apply H0.

      (* I := !Tmp_I *)
      pcrunch.

      pcrunch. pcrunch. pcrunch. pcrunch.

  (* FI *)
  apply mergeTheorem1.

(* ELSE *)

  (* (Clf (!Tmp_I === A0) *)
  simpl.
  eapply if_statement.

    (* CLoad T (IT +++ A1) *)
    simpl. pcrunch.

  (* ELSE *)

    (* Clf (!Tmp_r === A0) *)
    simpl. eapply if_statement.

      (* CLoad T (IT +++ A0) *)
      simpl. pcrunch.

    (* ELSE *)

      (* N := II *)
      simpl. eapply compose. pcrunch.

      (* NEW I, A2 *)
      eapply compose. pcrunch.

      (* CStore (I++++ A0) (IN) *)
      eapply compose. eapply store. compute. reflexivity. compute. reflexivity.
      eapply storeCheck1. (*apply H. apply H0.*)

      (* CLoad Tmp_I (I T +++ A1) *)
      eapply compose. pcrunch.

      (* CStore (I1 +++ A1) (I Tmp_I) *)
      eapply compose. eapply store. compute. reflexivity. compute. reflexivity.
      apply storeCheck2. (*apply H. apply H0.*)

      (* (CLoad T (I T +++ A0) *)
      pcrunch.

  (* FI *)
  Set Printing Depth 2000.
  pcrunch. pcrunch. pcrunch. pcrunch. pcrunch. pcrunch.
  apply mergeTheorem2.

(* FI *)
pcrunch.
apply mergeTheorem3.

(* FI *)
pcrunch.
apply mergeTheorem4.

pcrunch. pcrunch. pcrunch. pcrunch. pcrunch. pcrunch.

apply implication1.

intros. inversion H.
intros. inversion H.

apply implication2.
apply implication3.

intros. apply H.
intros. inversion H.

Grab Existential Variables.

apply nil.

Qed.
```

Step 3: fill in the details.

- Many bugs will be found by step 2
- Developers may opt to skip this step to save time
- Top level proof requires 12 lemmas (only one merge is non-trivial)
  - 5 for checking the validity of pointer references and a delete operation
  - 4 for merge steps
  - 3 for entailment

Merging states: Four of our 12 lemmas involve merging the states from the left and right branches of an if-then-else. Most of the work can be done by pairing off identical pieces as shown by the colored boxes. However, there is one predicate that is formed by or-ing pieces in the left and right branch shown in yellow.

LEFT

```
(AbsExists v5 (AbsExists v4
  (AbsExists v3 (AbsExists v2 (AbsExists v1
    [I = nth(v5,0)] *
    (P+1) |-> v1 * P |-> N * [0 < v1]*
    TREE(R, v2 , 2, [0, 1])* TREE(I, v5 , 2, [0])*
    TREE(N, v3 , 2, [0]) *
    [T in Treev2 ] *
    AbsAll v0 in TreeRecords(v5).
      [nth(find(v5,v0),2) inTree v2]*
    AbsAll v0 * TreeRecords(v3).
      [nth(find(v3,v0),2) inTree v2]*
    [v1 inTree v2])))))
```

RIGHT

```
(AbsExists v6 (AbsExists v5 (AbsExists v4
  (AbsExists v3 (AbsExists v2 (AbsExists v1
    [T = 0] * [I = 0] *
    (AbsEmpty * AbsEmpty) *
    [N = v5] *
    (P+1) |-> v6 * P |-> N *
    [P = v4] * [0 < v5] *
    TREE(R, v1 , 2, [0, 1]) * TREE(I, v2 , 2, [0]) *
    TREE(N, v3 , 2, [0]) *
    AbsAll v0 in TreeRecords(v2).
      [nth(find(v2,v0),2) inTree v1] *
    AbsAll v0 in TreeRecords(v3).
      [nth(find(v3,v0),2) inTree v1] *
    [v6 inTree v1])))))
```

COMBINED

```
(AbsExists v6 (AbsExists v5 (AbsExists v4
  (AbsExists v3 (AbsExists v2 (AbsExists v1
    [I = I] * (P+1) |-> v6 * P |-> N *
    [0 < v5] *
    TREE(R, v1 , 2, [0, 1]) * TREE(I, v2 , 2, [0]) *
    TREE(N, v3 , 2, [0]) *
    AbsAll v0 in TreeRecords(v5).
      [nth(find(v5,v0),2) inTree v2] *
    AbsAll v0 in TreeRecords(v3).
      [nth(find(v3,v0),2) inTree v2] *
    [v6 inTree v2] *
    [T = 0 \/ T inTree v1])))))
```

Our goal: Minimize proof development time and still find the bugs

- Key measure: "formal verification time/program development time"
- A fully automated static analysis tool is not practical as it cannot fully capture the data structure invariants needed to find many important bugs

A more complex case, 200 line C program implementing DPLL, invariant shown below

```
The first part of the invariant are spacial constructs asserting the two arrays and
three dynamic data structures in the heap.  ARRAY(root, count, functional_representation)
is a spatial predicate for arrays.  The functional representation is a list of the
elements.

AbsExistsT v0 . AbsExistsT v1 . AbsExists v2 . AbsExistsT v3. AbsExistsT v4.
    TREE(clauses,v0,sizeof_clause,[next_offset]) *
    TREE(assignments_to_do_head,v1,sizeof_assignment_stack,[next_offset]) *
    TREE(stack,v2,sizeof_assignment_stack,[next_offset]) *
    ARRAY(assignments,var_count,v3) * ARRAY(watches,var_count,v4) *

Next, we add on two assertions that guarantee that both the assignment_stack v2
and assignment array v3 are consistent.  We use (a,b)--->c as an abbreviation for
nth(find(a,b),c).

    (AbsAll v5 in TreeRecords(v2) . ([nth(v3,(v2,v5))-->stack_var_offset)==(v2,v5)-->stack_val_offset])) *
    (AbsAll v5 in range(0,var_count-1) . ([nth(v3,v5)==0] \/\": AbsExists v6 in (TreeRecords(v2)) .
          ([((v2,v6)-->stack_var_offset==v5 /\ (v2,v6)-->stack_val_offset==nth(v3,v5))])) ) *

The TREE declarations above define linked lists.  They do not define the back
pointers of a double linked list to do this for the assignments_to_do_head list, we
add the following assertion:

    (AbsAll v5 in TreeRecords(v1) .
      ([((v5,v1)-->prev_offset==0 /\ (assignments_to_do_head)==v5) \/
        ((v5,v1)-->prev_offset inTree v1 /\ (v5,(v5,v1)-->prev_offset)-->next_offset==v5)])) *

Now we define the linked lists for each of the watch variables.  Path is like TREE
but it defines lists inside of other structures.  It is used for the embedded watch
variable linked lists.  It takes the form:
Path(root, parent_functional_data,child_functional_form,node_size,pointer_offsets)
We also put in the assertion for the prev links:

    (AbsAll v6 in range(0,var_count-1)
      (Path((nthv(v4,v5)) , v0, v6, sizeof_clause, [watch_next_offset+v5]) *
      (AbsAll v7 in TreeRecords(v6)
        ([((v6,v7)--->(watch_prev_offset+v5)==0 /\ nth(v4,v5)====v6) \/
          ((v6,(v6,v7)--->(watch_prev_offset+v5))--->(watch_next_offset+v5))==v7)])) *
...
```

Coq DPLL invariant fragment

The design of both CoqPIE and the PEDANTIC framework come from informally working many verifications by hand. Our goal is to make formal verification more practical.

CoqPIE: An IDE for Coq

To improve proof development productivity, we developed CoqPIE. CoqPIE features
- full product management
- caching of proof states to allow quick browsing
- replay assist to help replay a proof when its statement changes