Week 1 Report

Zachary Porter

2022-08-31

1 Todos

- Understand checkpoint system for rr
- Demo of librr_rs
- Begin thinking about similartity metrics
- Outline usecase

2 Checkpoint system for rr

```
/**
 * Everything we know about the tracee state for a particular Mark.
 * This data alone does not allow us to determine the time ordering
 * of two Marks.
 */
struct InternalMark {
  InternalMark(ReplayTimeline* owner, ReplaySession& session,
               const MarkKey& key)
      : owner(owner),
        proto(key),
        ticks_at_event_start(session.ticks_at_start_of_current_event()),
        checkpoint_refcount(0),
        singlestep_to_next_mark_no_signal(false) {
    ReplayTask* t = session.current_task();
    if (t) {
      proto = ProtoMark(key, t);
      extra_regs = t->extra_regs();
    }
  }
  ~InternalMark();

  bool operator<(const std::shared_ptr<InternalMark> other);

  bool equal_states(ReplaySession& session) const;
  void full_print(FILE* out) const;

  ReplayTimeline* owner;
```

```
  // Reuse ProtoMark to contain the MarkKey + Registers + ReturnAddressList.
  ProtoMark proto;
  ExtraRegisters extra_regs;
  // Optional checkpoint for this Mark.
  ReplaySession::shr_ptr checkpoint;
  Ticks ticks_at_event_start;
  // Number of users of `checkpoint`.
  uint32_t checkpoint_refcount;
  // The next InternalMark in the ReplayTimeline's Mark vector is the result
  // of singlestepping from this mark *and* no signal is reported in the
  // break_status when doing such a singlestep.
  bool singlestep_to_next_mark_no_signal;
};

/**
 * Return a semantic copy of all the state managed by this,
 * that is the entire tracee tree and the state it depends on.
 * Any mutations of the returned Session can't affect the
 * state of this, and vice versa.
 *
 * This operation is also called "checkpointing" the replay
 * session.
 *
 * The returned clone is only partially initialized. This uses less
 * system resources than a fully-initialized session, so if you're going
 * to keep a session around inactive, keep the clone and not the original
 * session. Partially initialized sessions automatically finish
 * initializing when necessary.
 */
shr_ptr ReplaySession::clone();
```

See Session::copy_state_to

3 Project Assumptions

- The frontend is a good proxy for what is running in the background. In other words, visual updates are given to the user quickly after code runs in the background (quickly is subjective but probably less than 1s)
- It is not feasible to store every instruction (or even close to it). As such, it must be sufficient to scan and build paths intelligently
- WX circumvented with mprotect will guanratee that code section changes will not happen without an associated syscall.

  I am not sure if this applies with double mapped pages.

4 Use cases / Scenario

4.1 Date Gated Function

4.1.1 Description

Part of an executable contains a check that ensures that certain code only runs when f(current_time)==true
To start, f may be a simple operation such as x === 0 mod 10

You do not have access to the source and the binary has been compiled without debug symbols. You wish to understand exactly what the mechanism is that determines when the check passes and fails.

Examples of this may include malware or programs that have timing bugs. After you have identified the code in question, it may be possible to manipulate it with other outside tools.

4.1.2 Required Functionality

1. Replay the binary
2. Figure out screen recordings.
3. Trace back where the instruction for manipulating the instruction is stored
4. Identify similarity between multiple traces and highlight the divergence

5 Data Structures

```
struct RanInstruction{
    address: usize,
    frametime: u32, // W^X Assumption,
    inputs: Vec<DataValue>,
    instruction: DataValue<Instruction>, // todo.
}

enum DataPathNode{
    SYSCALL(name: String), // todo
    FILE(name:String, offset:usize),
    RAN_INSTRUCTION(v: RanInstruction),
    UNEXPANDED,
}
struct DataValue<T> {
    value: T,
    source: DataPathNode,
}

enum Trace{
    BLACKBOX(
        inputs:??
        size:usize, // Number of instructions?
        start: usize,
        end: usize,
        subtraces:Vec<Trace>,
    ),
```

```
    EXPANDED(
        instructions:Vec<RanInstruction>
    )
    BRANCH(
        // include the instruction they split off from and where they go
        targets: Vec<(RanInstruction, Rc<Trace>)>,


    )
    UNEXPANDED


}
```