Towards Run-time Verification in Access Control

Fatih Turkmen\textsuperscript{1}, EJ Jung\textsuperscript{2}, Bruno Crispo\textsuperscript{1}
\textsuperscript{1}DISI, University of Trento, Italy
\textsuperscript{2}CS dept, University of San Francisco, USA
Motivation

- How can we exploit existing software model checking tools that
  - Allow mimicking RBAC run-time
  - Support finite/infinite state analysis

- What concepts/techniques can be borrowed from run-time verification\(^1\)?

1: http://en.wikipedia.org/wiki/Runtime_verification
Desired properties

➢ Mutually Exclusive Roles (MER)
  • e.g. accountant vs. auditor

➢ Separation of Duty (SoD)
  • e.g. two signatures required for payment over $5K
  • even static version is co-NP
  • simulate with MER

➢ Conflict of Interest (CoI)
  • e.g. applicant vs. reviewer

➢ All require run-time verification
Example policy

Alice is a senior accountant and Bob is a junior accountant at Firm Fermata.

- An accountant can view and edit his or her clients’ information.
- An accountant can audit his or her junior accountants’ edits.
- A junior accountant covers for the senior accountant when the senior is out of office.
Static vs. Dynamic verification

- **Dynamic verification**
  - benefit from all the run-time information
    - e.g. user name, activated roles, session information, ...
  - imposes overhead on the system
    - coordination and synchronization over distributed systems is costly

- **Static verification**
  - can use more resource and time
  - lacks the run-time information
State Explosion

- Exponential combination of session interleavings
- Also for all possible values of dynamic info

\[
M = \frac{\left(\sum_{i=1}^{N} r_i \right)!}{\prod_{i=1}^{N} (r_i)!}
\]

- Each session runs in its own thread
- Atomic operations are authorization events \(A_n, D_n\) and \(A_p\)
Static approximates Dynamic

➢ Our idea
  • static approximation of run-time verification
  • via simulating dynamic simultaneous sessions
  • similar to approaches in software verification

➢ Current status
  • given the policy, simulate roles and users
  • adopted simple temporal properties to simulate events
  • can verify dynamic Mutually Exclusive Roles
    – thus approximates dynamic Separation of Duty
Scala Actor Framework

- Event-based
  - role (de-)activation, permission request: all events
  - scales well with many actors

- Thread-based
  - send/receive requests for role and permission activation

- Our approach: access control policy is modeled in Scala Actor Framework and treated as “software” that needs verification
Example codes

Coordinator
```scala
def act(){
  authorizer.start
  var authActorClose: Int = 0
  while(true){
    receive{
      case s: Session =>
        requestCount(s.getUser.getUserID)
        if user is allowed more sessions
          s.getUser.addSession
          s.getUser ! SessionPositive
        else {
          s.getUser ! SessionNegative
          authActorClose = authActorClose + 1
          if (authActorClose == userNum){
            authorizer ! Stop
            exit()
          }
        }
    case event : PA =>
      authorizer ! event
      receive{
        case Permit =>
          if (checkConstraints){
            history += event
            event.getOwner ! Permit
          }
          case Deny => event.getOwner ! Deny
        }
    case event : RA =>
      authorizer ! event
      receive{
        case Permit =>
          if (!checkConstraints){
            history += event
            event.getOwner ! Permit
          }
          case Deny => event.getOwner ! Deny
        }
    case Stop =>
      exit()
    }
  }
}
```

Authorizer
```scala
def act(){
  initialize
  while(true){
    receive{
      case e: RA =>
        if (checkRA(e)) sender ! Permit
        else sender ! Deny
      case e: PA =>
        if (checkPA(e)) sender ! Permit
        else sender ! Deny
        case Stop =>
          exit()
    }
  }
}
```

User
```scala
def act(){
  var session : Session = createSession(generateRoleEntropy)
  while(true){
    receive{
      case SessionPositive =>
        session.start
        sessions += session
        Thread.sleep(random.nextInt(500))
        session = createSession(generateRoleEntropy)
      case SessionNegative =>
        exit()
    }
  }
}
```
Test cases (policies) initiate verification

1. User requests roles and permissions
2. Session information is given to coordinator
3. Coordinator asks Authorizer for decision
User behavior modeling

- Session creation
- $A_R$: Activating a Role
- $D_R$: Deactivating a Role
- $A_P$: Activation a Permission

$$M = \frac{\left( \sum_{i=1}^{N} r_i \right)!}{\prod_{i=1}^{N} (r_i!)}$$

- Each session runs in its own thread
- Atomic operations are authorization events ($A_R$, $D_R$ and $A_P$)
Three levels of verification

- **Core simulation**
  - partial simulation without run-time properties
  - e.g. race conditions, policy conflicts

- **Symbolic evaluation**
  - using some approximations of run-time properties
  - temporal and location parameters approximated
  - Monte Carlo simulation of parameters

- **Monitor development**
  - partial step-wise verification at run-time
  - just-in-time verification using Aspect Oriented Programming
Three levels of verification

- **Core simulation**
  - partial simulation without run-time properties
  - e.g. race conditions, policy conflicts

- **Symbolic evaluation**
  - using some approximations of run-time properties
  - temporal and location parameters approximated
  - Monte Carlo simulation of parameters

- **Monitor development**
  - partial step-wise verification at run-time
  - just-in-time verification using Aspect Oriented Programming
DMER support example

- Bob is in charge of Bravo account.
- Alice is out of office today.
- Bob covers of Alice today.
- Bob can audit accounts of Alice’s junior accountants
- Bob can audit his own account?
How DMER is supported

- Using Basset and Java Path Finder for verification

- Constraint: DMER
  - Given a new role activation request
  - Compute the intersection of all active roles and
  - the roles under DMER constraint
  - If the intersection of two sets is greater than a preset threshold
  - Deny the new role activation request
How DMER is supported (2)

- Encode RBAC, properties and the constraint checking function

```scala
def checkConstraints(ev:Event): Boolean{
  if (event.isInstanceOf[RA]){
    ract = ev.asInstanceOf[RA]
    if (ract != null) {
      // get the authorizer actor
      r = authorizer.getRoleFromID(ract.getRole)
      if subRoles is a subset of allActiveRoles &
        subRoles is a subset of DMER1 &
        subRoles.length == DMER1.getK()
      return false;
    }
  }
}
```

- Execute JPF and Basset for state analysis.

```bash
..\jpf-core\bin\jpf +basset.language=scala
gov.nasa.jpf.actor.Basset Simulator
```
How DMER is supported (3)

- Bob is an Accountant of Bravo account
  - Bob cannot disable Accountant role on this object
- Bob requests to activate an Auditor role
- The activated roles = \{Accountant\}
- The roles under constraint = \{Accountant, Auditor\}
- The intersection = \{Accountant\}
- Threshold = 1
- Bob’s request is denied
Extra support

➤ Session concurrency
  • What if Bob logs in on two different computers?
  • Verification per session is not enough
  • Simultaneous sessions are evaluated together
    – if they touch the same object

➤ History support
  • support for Conflict of Interest, Chinese Wall policies
  • if Bob has been an accountant of this account in the past, then he is not eligible to audit this account
Conclusion and Future work

- **Conclusion**
  - approximate run-time verification in static
  - can verify dynamic mutual exclusive roles

- **Future work**
  - support for more generic COI
  - large scale experiments for performance testing