Assessing the Security of a Mission-Critical Software System

David Gilliam, John Powell, & John Kelly
Jet Propulsion Laboratory

Matt Bishop
University of California at Davis

California Institute of Technology, Jet Propulsion Lab
Computer Security Laboratory, UC Davis
NOTE:

This work is sponsored by NASA's Office of Safety and Mission Assurance under the NASA Software Program lead by the NASA Software IV&V Facility

This activity is managed locally at JPL through the Assurance and Technology Program Office (502)
Research Goal

- Reduce security risk to the computing environment by mitigating vulnerabilities in the software development and maintenance life cycles
  - Vulnerability matrix
    - Vulnerabilities exploits and signatures
  - Security Assessment Tools List
  - Property-based testing tool—Tester’s Assistant
  - Model-based security specification and verification tool and report
Vulnerability Matrix

- Vulnerability matrix to assist security experts and programmers where best to expend their efforts
  - DOVES database (maintained by UC Davis): http://seclab.cs.ucdavis.edu/projects/DOVES
  - Uses the Common Vulnerabilities and Exposures (CVE) Listing (MITRE)
  - Contains signatures used to exploit the vulnerability – signatures to be used with the Tester’s Assistant and the Modeling SPIN Tool
Security Assessment Tools

- Software Security Assessment Instrument
  - Security assessment tools
    - Description of each tool and its purpose
    - Pros and Cons of each tool
    - Alternate and related tools
Property-Based Testing

- Property-based testing tool – Tester’s Assistant (Matt Bishop, UC Davis)
  - Perform code slicing on applications for a known set of vulnerabilities
  - Test for vulnerabilities in code on the system or whenever the computing environment changes
  - Initially, checks software developed in JAVA
    - The goal is to have the tool check other programming and scripting languages as well (C, C++, Perl, ActiveX, etc.)
Property-Based Testing (Cont.)

- Compare program actions with specifications
  - Create low-level specifications
  - Instrument program to check that these hold
  - Run program under run-time monitor
  - Report violations of specifications
Property-Based Testing (Cont.): How It Works

*Backup Slides provide an example on how this works with the TASPEC*
Property-Based Tester

- **TASPEC language definitions**
  - Handle ambiguous specifications and facts
  - Resetting, non-resetting temporal operators
  - Existential, universal logical operators

- **Design Decisions**
  - Instrumenter does most work
TASpec and the TEM

- Invariants (properties) given to TEM
- Test Execution Monitor accepts TASpec statements from executing program
  - Statements record facts about current state relevant to properties
  - Can assert, retract facts
- TEM verifies current state satisfies desired properties
Relation to Other Spec Languages

- **Can go from Z to TASpec**
  - Not everything translates
  - TASpec has no notion of type, Z does
  - Translation straightforward, in the sense of known algorithms

- **Differences limit translation**
  - Z high-level, not concerned with implementation details
  - TASpec low-level, lots of implementation details
  - Z *a priori* specification
  - TASpec *a posteriori* specification
TASpec Languages

- Predicates
- Arithmetic operators: + - * / %
- Relational operators: == != > < >= <=
- Logical operators: and or not implies
  - and, or existential; not, implies universal
- Temporal operators: before until eventually
- Location specifiers: func variable decl
- Miscellaneous
  - assert, assertonce, retract, check
  - exec, forall
Complications

- **Logical operators**: existential or universal?
  - and, or existential
  - not, implies universal
- **Temporal operators**
  - $a$ before $b$: when $b$ becomes true, $a$ is true
  - $a$ until $b$: from the time this property is entered, $a$ is true until $b$ becomes true, at which point $a$ must be false
  - eventually $a$: $a$ is true when the program terminates
Tester's Assistant Specifications

- Example: "a user must authenticate himself or herself before acquiring privileges"

  is password correct? {
    Compare user's password hash to hash stored for that user name
    If match, set UID to user's uid
    If no match, set UID to ERROR
  }

  if privileges granted {
    compare UID to the uid for which privileges are granted
    if match, all is well
    if no match, specification violated
  }
Example C Code

```c
if (fgets(stdin, uname, sizeof(uname)-1) == NULL)
    return(FAILED);

typedpwd = getpass("Password: ");
if ((pw = getpwnam(uname)) != NULL) {
    hashtp = crypt(pw->pw_passwd, typedpwd);
    if (strcmp(pw->pw_passwd, hashtp) == 0) {
        setuid(pw->pw_uid);
        return(SUCCESS);
    }
}
return(FAILED);
```
In TASPEC

location func `setuid(uid)` result 1
   { assert `privileges_acquired(uid)`; }

location func `crypt(password,salt)` result `encryptpwd`
   { assert `password_entered(encryptpwd)`; }

location func `getpwnam(name)` result `pwent`
   { assert `user_password(name, pwent->pw_passwd, pwent->pw_uid)`; }

location func `strcmp(s1, s2)` result 0
   { assert `equals(s1, s2)`; }

`password_entered(pwd1)` and
   `user_password(name, pwd2, uid)` and `equal(pwd1, pwd2)`
   { assert `authenticated(uid)` ; }

`authenticated(uid)` before `privileges_acquired(uid)`
if (fgets(stdin, uname, sizeof(uname)-1) == NULL)
    return(FAILED);

  typedpwd = getpass("Password: ");
  if ((pw = getpwnam(uname)) != NULL) {
    hashtp = crypt(pw->pw_passwd, typedpwd);
    if (strcmp(pw->pw_passwd, hashtp) == 0) {
      setuid(pw->pw_uid);
      return(SUCCESS);
    }
  }

return(FAILED);
Model-Based Security Specification

- Model-based security specification and verification involves applying formal modeling to the IT security arena
- Verification systems that perform logical verification of temporal properties over models are referred to as model checkers
  - Exhaustive search of a model’s corresponding state space
  - Can be used on suitably restricted “partial specifications”
State Charts

- State Charts are specification notations to define systems
  - Defines the collection of (abstract) variable value pairs at a given point in the system (execution) – referred to as a state
  - Defines the relationships with which the system transitions from one state to the another
Model Based Verification (MBV) within an Integrated Approach

- Flexible Modeling Framework (FMF)
  - Compositional Approach
  - Makes use of SPIN
  - Infers Results from a partial model
- Property Interaction with
  - Vulnerability (VMatrix)
  - Property Based Testing (PBT)
- Potentially discovers new vulnerabilities
The Flexible Modeling Framework (FMF) Approach to MBV

- A Component (c) is some logical unit of process or application behavior
  - A single application often will need to broken into multiple model components

- Combining two components C1 and C2
  - Model Checking (MC)
    1. Non-trivial combination of C1 and C2
    2. Searches the Cartesian Product of the sizes of C1 and C2
  - FMF
    1. MC of C1 and C2 individually
    2. Combines the State Charts (SC) of C1 and C2
    3. Integrates assumptions that follow from 1 above
    4. SC traversal or localized MC of appropriate sub-model
Domain Specifics and FMF

**C1**

\[ O \oplus t(x) \quad \text{Property p must hold} \quad \neg O \oplus t(x+n) \]

**C2**

\[ O \oplus t(y) \quad \text{Property p must hold} \quad \neg O \oplus t(y+m) \]

- **MC reports p holds for C1 and C2**
  - Assumptions can be made about transitions (T) in C1/C2 SC
    - P holds on T from C1 \^ C2
    - P holds on T from C1 \^ (Unknown in C2)
    - P holds on T from (Unknown in C1) \^ C
- **Unify consistent states in the SCs of C1 and C2**
  - Condition: All variables that are known in C1 and C2 agree
- **Any path from “O” that does not reach “\neg O” produces an unknown security result when the combined C1/C2**
Combinatorial Network Aware Cases being Addressed

C1

\[ O @ t(x) \quad \text{Property p must hold} \quad \sim O @ t(x+n) \]

C2

\[ O @ t(y) \quad \text{Property p must hold} \quad \sim O @ t(y+m) \]

Network Aware (NA) Cases:

1. \( t(x) = t(y) \) – C1 and C2 are NA simultaneously

2. \( t(x+n) = t(y) \) – C1 ends NA sequence and C2 starts NA sequence simultaneously

3. \( t(x) = t(y+m) \) – C2 ends NA sequence and C1 starts NA sequence simultaneously

* Sub cases where \( n = m \) and \( n \neq m \) – not currently known if this distinction is significant with an abstract model in this domain
Combinatorial Network Aware Cases being Addressed (Cont.)

- The same timing cases seen on the previous slide must be considered in the context of one NA component (C1) and one non-NA component (C2)
  - C1 occurring in a time relation case previously discussed while sharing resources in common may potentially create vulnerabilities.
    - E.g. A NA control application and a printer
  - Non NA components (application pieces) may have been justifiably engineered with little or no consideration of network security issues
  - A non-NA component may represent a piece of a NA application that does not interact with a network.
    - I.E. \( t(X+n) < t(y), t(x) > t(y+m) \)
Model Checking: A Case Study
Simplified State Machine for Prime

"Validating Requirements for Fault Tolerant Systems Using Model Checking", Schneider, Callahan & Easterbrook, 1998
This Case Study was funded by the NASA Software Program at the NASA IV&V Facility and JPL under a separate task
David Gilliam - Network & Computer Security, JPL
Matt Bishop - Computer Security Laboratory, UC Davis
August 8, 2001
Real Project Application

- JPL Class A Flight Project
  - Will test toolset on Flight Mission internet-aware communication software
- IsoWAN & Information Power Grid testbeds
  - Isolated wide-area networks using a modified VPN solution to create a secure, isolated, computing environment
  - Use with high-performance supercomputing collaborative environment
Potential Follow-On Work

- Training in use of security assessment tools in the software development and maintenance lifecycle
- Development of re-composable model sub-components
- Develop capability for easy storage and access of a library of common network security model components and past verification results
- Develop a programmer interface to assist users with generating properties for input into the tools
Potential Follow-On Work (cont.)

- Enhancing and augmenting the toolset
  - Port the code to run on different operating systems
  - Include additional programming and scripting languages that the Tester’s Assistant tool can slice for vulnerabilities
  - Augment the toolset by incorporating or developing additional tools
  - Develop a graphical user interface front-end checklist and decision tree to assist in building the Model to be verified
Collaborators

- David Gilliam – Principle Investigator
  Network and Computer Security, JPL
- John Powell – Research Engineer
  Quality Assurance, JPL
- John Kelly – RTOP Manager
  Quality Assurance, JPL
- Matt Bishop – Associate Professor of Computer Science
  University of California at Davis
FOR MORE INFO...

David Gilliam  
JPL  
400 Oak Grove Dr., MS 144-210  
Pasadena, CA 91109  
Phone: (818) 354-0900  
FAX: (818) 393-1377  
Email: david.p.gilliam@jpl.nasa.gov

John Powell  
MS 125-233  
Phone: (818) 393-1377  
Email: john.d.powell@jpl.nasa.gov