Analysis of a Secure Service Proxy Toolkit

Carolyn Talcott
SRI International

In collaboration with
John Mitchell, Ninghui Li, Derrick Tong
Stanford
FTN Case Studies

- Proxy-based Distributed Systems
  - secure distribution of component based systems
- DDOS models
  - formal attack models, formal v&v?
- Secure Spread (Maude + uCAPSL)
  - is it secure?
  - is the group semantics preserved?
- TIARA project
  - intrusion tolerance for ad hoc networks
- Distributed/replicated databases
  - formal verification of core algorithms
  - reuse to verify DB specific optimizations
**Maude Methodology**

- Impact rapid prototyping
- State space search
- Model checking

---

- Yearly Monthly Weekly Daily Momentally
- Database
- Aggregate
- Trap poil filtering
- Resource monitoring
- Fault Configuration Accounting Performance Security

---

- $S \models \square$
- rapid prototyping
- state space search
- model checking
Remote Services

Requirements

• Publication and discovery
• Remote messaging
• Qos
  - Transparency
  - Security
    • Getting the right / expected service
    • Confidentiality

Approach: Service Proxy Toolkit (SPTK)
Service Proxy Toolkit Architecture
The Secure Server Proxy Toolkit in Pictures
Security Goals

- **Goal 0**: Client VM protected from evil proxy
- **Goal 1**: Secure client-server communication
- **Goal 2**: Client can authenticate service proxy
- **Goal 3**: Server can also authenticate client
Registering a signed secure proxy

1. RegReq(svcName, Svc)
2. create(sAP)
3. sign(cAPd)
4. register(svcName, signed cAPd)
Getting an authentication proxy

1. findSvc(svcName)
2. lookup(svcName)
3. reply(signedcAPd)
4. verify(signedcAPd,tsK)  
   must check description!
5. ok(apxD)
5a. install(apxD)
6. authenticate(ccred)
6a. setup secure cnx
7. authenticate(ccred)
8. checkClient(ccred, svcName)
9. clientOk(perms)
10. install(sspd)

11. encryptReq(csspd, ccred)
12. encrypted(csspd, ccred)
13. encrypted(csspd, ccred)
14. decrypt(encrypted(csspd, ccred))
15. ok(csspd)
16. install(csspd)
17. findSvcReply(cSP)
16. serviceCall(args)
17. serviceCall(args)
18a. check(args, per)
18. serviceCall(args, cId)

19. serviceReply(result)
20. serviceReply(result)
21. serviceReply(result)
The Secure Server Proxy Toolkit in Maude
Overview!

• **Model**
  - SPTK Architecture
  - Attacker
  - Toolkit

• **Compose and analyze**

• **Use**
  - Maude object notation (with Russian dolls)
  - ACI (multi-set) rewriting
  - Rule conditions
Security Goals

• Goal 0: Client VM protected from evil proxy

• Goal 1: Secure client-server communication

• Goal 2: Client can authenticate service proxy

• Goal 3: Server can also authenticate client
Proxy Toolkit Models

• Level 0: naive proxy (just does rmi)
  - Relies on JVM to achieve goal 0

• Level 1: Level 0 + secure communication
  - Achieves goal 1

• Level 2[†,f]: Level 1 + signed proxy
  - [with,without] checking proxy service name
  - Achieves goal 2

• Level 3: Level 2† + mutual authentication
  - Achieves goal 3
SPTK Architecture in Maude

• Infrastructure
  - JVM  -- java execution env
  - ETHER-CLASS  -- communication media (synch)

• Main components
  - SVC  -- generic service
  - APP  -- generic app/client
  - LOOKUP-IFACE  -- registry interface

• Toolkit
  - SPTK-CLASS  -- role independent structure
  - SSPTK-IFACE  -- server-side interface
  - CSPTK-IFACE  -- client-side interface
  - SP-CLASS  -- proxy interface and structure
Server Node

< JS : JVM | jsatts:AttributeSet,
    { socf:Configuration
        < JS . sptk : SSPTK | tkatts:AttributeSet >
        < JS . svc : SVC | svcatts:AttributeSet >
        msg( JS . sptk, JS . mgr!, registerReq("Quote", JS . svc))
    } >
Attacker Models

• **Attacker in the ether**
  - *Can read and modify communications!*

• **Attacker lookup service**
  - *Chooses what to serve*

• **Both can**
  - *Register (dis) services*
  - *Impersonate valid client*
ETHER! Attack Configuration

- TEST
- ATTACKER
- LOOKUP
- initial configuration template

\[ \text{econf}\_\_ = (\text{eacf ljcf ajcf}\_\_ \text{ sjcf}\_\_ \text{ cjcf}\_\_ )} \]

- \text{eacf} = ether attack
- \text{ljcf} = lookup node
- \text{ajcf}\_\_ = impersonator
- \text{sjcf}\_\_ = server node
- \text{cjcf}\_\_ = client node
LOOKUP! Attack Configuration

- TEST
- ATTACKER-LOOKUP
- initial configuration template
  \[ lconf_{\_} = \text{eecf aljcf\_\_} \text{ sjcf\_\_} \text{ cjcf\_\_} \]  
  - eecf = ether node
  - aljcf\_\_ = lookup attack + impersonator
  - sjcf\_\_ = server node
  - cjcf\_\_ = client node
SPTK Modules

SPTK<\text{j}> -- behavior rules for level j

• Toolkit -- CSPTK<\text{j}> + SSPTK<\text{j}>

• Proxies -- CSP<\text{j}> + SSP<\text{j}>

• Level 3 adds pair of authentication proxies

Configurations to analyze:

• econf<\text{j}> in TEST-ETHER + SPTK<\text{j}>

• lconf<\text{j}> in TEST-LOOKUP + SPTK<\text{j}>
Registering a service

Levels 0,1

SvcMgr → sPTKO

Lup

create

sSP

registerReq

registerReply

registerReply

Levels 2,3

SvcMgr → TMS → sPTKO → Lup

spRegister

create

sSP/sAP

signReq

signReply

registerReq

registerReply

registerReply

Levels differ in choice of proxy behavior
Scenario -- finding and using a service via Level 0,1 PTK

Scenario:

1. **App** sends a request to **cSPTk** to find a service.
2. **cSPTk** looks up the service and passes the request to **sSP**.
3. **sSP** installs a secure proxy and returns the result to **cSPTk**.
4. **cSPTk** forwards the request to **Svc**.
5. **Svc** makes a call to **Lup**.
6. **Lup** returns a success or failure status.
7. **Svc** returns the result to **App**.
8. **App** processes the result.

Points of attack:

- Illegal Call at **Svc**

Clear in level 0, secure in level 1 point of attack
Scenario -- finding and using a service via  Level 2 PTK
Scenario -- finding and using a service via Level 3 PTK
Analysis summary

• Define protoypical SPTK configuration!
• Compose with attacker configuration
• Simulate -- execute with some scheduler
• Check security properties
  - Search state space
Properties for Ether Attack

1. Attacker in the ether + impersonator

Properties checked:

• 1.1 attacker see/modify client data
  - sent as service arguments or received as reply
• 1.2 client gets answer from wrong service
• 1.3 unauthorized service call succeeds
• 1.4 client impersonator succeeds
Ether attack search patterns

*** attacker gets private data sent in reply
search [1] (eacf ljcfcjcf cjcḫ) =>+
( cf:Configuration
  < eee:Attacker | atts:AttributeSet,
  clientDs(reply(d:Data, mycall,"sam") ds:DataSet)>
).

*** Attacker impersonates (gets reply to request from "sam")
search [1] (eeecf ljcf sjcf cjcFrançois ajcf ) =>+
( cfx:Configuration < JA : JVM | jatts:AttributeSet,
  { cf:Configuration
    < JA.app : APP | catts:AttributeSet, waitFor(sid:Oid) >
    msg(JA.app, sid:Oid, svcReply(reply(d:Data, dc:Data, sam")))) } >
).
Summary of analyses
compromised ether

<table>
<thead>
<tr>
<th>Property:</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>1.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 0</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Level 1</td>
<td>--</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Level 2f</td>
<td>--</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Level 2+</td>
<td>--</td>
<td>--</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Level 3</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
2. Attacker controls Lookup node

Properties checked:

• 2.1 client app can get proxy to requested service (sanity check)
• 2.2 client accepts proxy to attacker service
• 2.3 client accepts wrong server proxy
• 2.4 service integrity violated
Lookup Attack Search Patterns

*** client accepts proxy to wrong service at trusted server

search [1] icf-aa =>

(cf:Configuration
  < JC : JVM | jcatats:AttributeSet, { cocf:Configuration
    < O:Oid : cl:Cid | svc(S:Oid), oatts:AttributeSet >
    msg(JC . app, JC . csptk, findServiceOk(sn:String, O:Oid)) } >
  < JS : JVM | jsatts:AttributeSet, { socf:Configuration
    < S:Oid : spc:Cid | sname(ssn:String), satts:AttributeSet > } >
)
such that (ssn:String =/= sn:String).
Summary of analyses compromised registry

<table>
<thead>
<tr>
<th>Property:</th>
<th>2.1</th>
<th>2.2</th>
<th>2.3</th>
<th>2.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 0</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Level 1</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Level 2f</td>
<td>+</td>
<td>--</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Level 2t</td>
<td>+</td>
<td>--</td>
<td>--</td>
<td>+</td>
</tr>
<tr>
<td>Level 3</td>
<td>+</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
Conclusions

Value added:
- Documentation of SSPTK architecture
  - Modular, tunable security levels
- Formalization of some security goals
- Security hole closed

What more?
- Formalize simplifications
- Can we reduce arbitrary configurations to finite?