Understanding the Vulnerabilities of a SOA Platform

A Case Study

Andrew Gronosky, Michael Atighetchi, Partha Pal
9th IEEE NCA Symposium
July 16, 2010
Outline

• Introduction to SOA and its security challenges
• Description of the System Evaluated
• Evaluation Methodology
• Findings
• Ideas for Mitigating Vulnerabilities
• Conclusions
What is SOA?

- SOA = Service Oriented Architecture
- The latest stage in an evolving paradigm of distributed computing
- Consists of loosely coupled services communicating via coarse-grained messages
- Includes but is not limited to Web Services technology
Security Challenges in SOA

- Current SOA platforms lack sufficient protection against maliciously initiated loss, corruption and leakage of information
  - Existing practices focus on securing the perimeter
- Correct operational assurance in the presence of attacks is difficult, because:
  - Systems are interconnected and dynamic, making protection difficult
  - Adversaries typically only need to find one weakness
  - Developers focus on functionality and neglect impact on security

State of the Practice

<table>
<thead>
<tr>
<th>Security Layer</th>
<th>Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network and Transport Layer Security (IPsec, TLS)</td>
<td>Sole reliance on these causes issues; only protect network channels and typically fail to resist insider attacks; vulnerable to remote exploits; unable to track data provenance</td>
</tr>
<tr>
<td>Application Layer Security (DMZs, Application Firewalls)</td>
<td>Static configuration and lack of diversity increase vulnerabilities, limitations of signature-based detection</td>
</tr>
<tr>
<td>Service Infrastructure/Platform Security (Web Services)</td>
<td>12+ individual OASIS standards; not a solution, but a protocol construction kit with little guidance on how to construct survivable systems; mismatched assumptions</td>
</tr>
</tbody>
</table>
Outline

• Introduction to SOA and its security challenges
• **Description of the System Evaluated**
• Evaluation Methodology
• Findings
• Ideas for Mitigating Vulnerabilities
• Conclusions
System Under Test

- The Reference Implementation is a prototype SOA being developed by U.S. Department of Defense researchers
- Consists largely of open-source components with some Government off-the-shelf (GOTS) software
Components Tested

- PostgreSQL database and JBoss Application Server were selected for initial testing (the scope of this paper)
- Both of these are widely used, open-source components
Outline

- Introduction to SOA and its security challenges
- Description of the System Evaluated
- Evaluation Methodology
- Findings
- Ideas for Mitigating Vulnerabilities
- Conclusions
Threat Model

- Since this is a U.S. Department of Defense system, we assume a nation-state adversary with extensive resources
  - Unlimited computing resources
  - Unlimited time
  - Unlimited expertise
  - Full knowledge of system design including code artifacts
- We assume attacker has access to the network
- We consider the possibility of insider attacks
Generalized Approach to Survivability Testing

- We have developed a generalized, layered model for survivability testing.
- This depth and breadth of coverage can be tailored to fit budget and schedule constraints.
- For this study we emphasize stress testing and direct attacks.

Pictorial representation of the evaluation process.
Progression of Tests and Attacks

1. **Network Sniffing**
   - Break confidentiality

2. **Port Scanning**
   - Identify vulnerable TCP/IP ports

3. **TCP Connection Flooding**
   - Deny service to legitimate clients

4. **Serialization Attacks**
   - Take control over JVM

5. **Application Level Floods**
   - Overload Registry

6. **Password Cracking**
   - Determine passwords through brute-force trial and error

7. **SQL Injection**
   - Circumvent authentication & cause loss of confidentiality and integrity

8. **Fault Injection**
   - Cause Memory/CPU overload

9. **Simulated Node Failures**
   - Test liveliness monitoring
Outline

• Introduction to SOA and its security challenges
• Description of the System Evaluated
• Evaluation Methodology
• Findings
• Ideas for Mitigating Vulnerabilities
• Conclusions
# Summary of Findings

<table>
<thead>
<tr>
<th>Test</th>
<th>Result</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis of Configuration Files</td>
<td>Passwords stored in clear text, no password policy enforcement at the application level</td>
<td>C, I, A</td>
</tr>
<tr>
<td>Network Sniffing</td>
<td>Default configuration sends out data in the clear</td>
<td>C, I</td>
</tr>
<tr>
<td>Port Scanning</td>
<td>RI susceptible to remote scanning</td>
<td>C</td>
</tr>
<tr>
<td>TCP Connection Flooding</td>
<td>Denial of service without generating log entries</td>
<td>A</td>
</tr>
<tr>
<td>Java Serialization Attacks</td>
<td>Clients can execute arbitrary code in the JVM without authentication</td>
<td>C, I, A</td>
</tr>
<tr>
<td>Application Flooding</td>
<td>~70% increase in round trip completion times</td>
<td>Little A</td>
</tr>
<tr>
<td>Password Cracking</td>
<td>Some passwords are weak, no lock-out to protect against brute-force attacks, attacks have low visibility</td>
<td>C, I</td>
</tr>
<tr>
<td>SQL Injection</td>
<td>Absence of mechanisms to protect against SQL injection attacks, attacks have low visibility</td>
<td>C, I, A</td>
</tr>
<tr>
<td>Fault Injection</td>
<td>No protection or detection mechanisms for faulty applications</td>
<td>A</td>
</tr>
<tr>
<td>Simulated Node Failures</td>
<td>Absence of monitoring protocol and unnecessary dependencies</td>
<td>A</td>
</tr>
</tbody>
</table>

C=Confidentiality, I=Integrity, A=Availability
Patterns of Vulnerabilities

• In some cases, security mechanisms were available but were not properly configured.
  – Passwords stored in clear text
  – Transport-level encryption not everywhere enabled
  – JBoss database connector did not re-try lost connection
• Services are directly exposed to untrustworthy clients
  – TCP connection flooding
  – Application flooding
  – SQL injection
• Reference Implementation lacks capabilities to cope with some attacks
  – Serialization attack
  – Fault injection
Outline

• Introduction to SOA and its security challenges
• Description of the System Evaluated
• Evaluation Methodology
• Findings
• Ideas for Mitigating Vulnerabilities
• Conclusions
Architectural Enhancements

Without architecture enhancement, SOA and SOA-based implementations would have to include point-solutions for individual threats in an ad-hoc manner, leading to more complex and brittle solutions with potentially inadequate coverage.

- **Containment regions** to stop propagation and escalation of attack symptoms
- **Conglomerate management** to make cooperation among replicated services more reliable
- **Adaptive responses** to continue to operate through attacks, possibly degrade and recover
- **Crumple zones** to absorb the impact of attacks before they reach critical services
Outline

• Introduction to SOA and its security challenges
• Description of the System Evaluated
• Evaluation Methodology
• Findings
• Ideas for Mitigating Vulnerabilities
• Conclusions
Conclusions

• Contributions of this paper include
  – Instance of our layered approach to survivability testing
  – Example of vulnerabilities in a real SOA system
    • Reference Implementation was at least somewhat vulnerable to every category of attack
    • Vulnerability profile can be reduced by expending more effort on configuration
  – Identifies some requirements for architectural enhancements to the Reference Implementation
Related Work

Recent papers available at www.dist-systems.bbn.com/papers/


Outline

• Introduction to SOA and its security challenges
• Description of the System Evaluated
• Evaluation Methodology
• Findings
• Ideas for Mitigating Vulnerabilities
• Related Work
Q & A

• Questions?
• Backup slides for explaining particular points in further detail
Distributed Computing Models

- **Distributed objects**
  - E.g., CORBA, DCOM
  - Functionality encapsulated as objects
  - Hides details of distribution, language

- **Component middleware**
  - E.g., CCM, JavaBeans, MBeans
  - Functionality encapsulated as components
  - Abstracts distribution, assembly, runtime containers

- **Service-based and Service-oriented systems**
  - J2EE, JBoss, WebLogic
  - Functionality encapsulated as services
  - Abstracts distribution, discovery, composition, lifecycle

---


Static Analysis

Description
- We manually inspected security-related configuration files
  - PostgreSQL host-based access control
  - SSH daemon configuration
  - JBoss Application Server configuration files
  - etc.
- We performed an analysis of password strength

Results
- JBoss Application server stored several passwords in clear text
  - This includes passwords to SSL key stores
  - Available security features were not enabled
- Strong password policies were enforced at the operating system level but not at the application level

Recommendations
- Ensure administrators have adequate training and documentation
- Implement password auditing at the application level
**Direct Attack: Network Sniffing**

**Description**
- Intercept packets on the network and inspect their contents
- Difficult to detect and can be done with off-the-shelf tools

**Results**
- Reference Implementation was configured with HTTP (not HTTPS) for JBoss administration console
- No encryption of RMI traffic between JBoss AS and remote clients
- No encryption of traffic between JBoss AS and PostgreSQL
- We were unable to capture any unencrypted passwords

**Recommendations**
- Enable transport-level encryption everywhere possible including but not limited to the weak points identified above
- Effective use of encryption requires a key management capability, which is absent from the RI
Direct Attack: Port Scanning

```
matighet@mango > nmap -p- rics41
Starting Nmap 5.00 ( http://nmap.org ) at 2009-12-15 14:09 EST
Interesting ports on rics41 (172.21.19.41):
Not shown: 65518 closed ports
PORT       STATE SERVICE
22/tcp     open  ssh
1098/tcp   open  unknown
1099/tcp   open  unknown
3306/tcp   open  mysql
3873/tcp   open  unknown
4444/tcp   open  krb524
4445/tcp   open  unknown
4446/tcp   open  unknown
4457/tcp   open  unknown
8009/tcp   open  ajp13
8080/tcp   open  http-proxy
8083/tcp   open  unknown
16578/tcp  open  unknown
44651/tcp  open  unknown
47020/tcp  open  unknown
```

Description
- Scan for listening ports as a precursor to further attack

Results
- Reference Implementation has no defense against port scanning

Recommendations
- Network protections such as IPsec and firewalls can help
- Port scanning is detectable so monitoring is possible
- We are investigating single-packet authorization (SPA) firewalls

_Nmap tells us all about interesting ports on rics41_
Details: TCP Connection Flooding

Description
- Open many TCP connections to exhaust resources or starve other clients

Results
- Temporary loss of database availability
- Potential loss of service due to huge log files exhausting storage
- Port 1098 is especially vulnerable
- Port 1099 was not seriously affected

<table>
<thead>
<tr>
<th>Server</th>
<th>Port</th>
<th>Service</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Postgres</td>
<td>5432</td>
<td>PostgreSQL</td>
<td>Loss of database availability for 65 seconds</td>
</tr>
<tr>
<td>JBoss</td>
<td>4446</td>
<td>JBoss Application Server</td>
<td>&gt; 100x increase in round-trip time, frequent failure of client to connect</td>
</tr>
<tr>
<td>JBoss</td>
<td>1099</td>
<td>RMI Registry</td>
<td>Server terminated idle connections</td>
</tr>
<tr>
<td>JBoss</td>
<td>1098</td>
<td>RMI Activation</td>
<td>All client connections blocked</td>
</tr>
</tbody>
</table>

Recommendations
- Integrate an intrusion detection system into the Reference Implementation
- Use firewall rules to limit incoming connection rates per IP address
Findings: Java Serialization Attack

Description
• Send a specially constructed Java object to an RMI port on the application server to exploit known type safety issues during de-serialization
• Short cut: write a Bomber class and put it on the server’s class path

Results
• Reference Implementation is vulnerable to serialization attack
• Class path contains 503 serializable classes

Recommendations
• Review and limit contents of server class path
• Design a proxy that can de-serialize objects before they reach the server
Findings: Fault Injection

Description
- Write malicious services and deploy them to the platform
- We introduced two main types of faults: CPU overload and memory exhaustion
- Study what would happen if an attacker compromised a service (rather than how an attack could work)

Results
- As shipped, Reference Implementation has no Java security manager configured, making it easy to deploy unsigned code.
- Memory exhaustion is an effective denial-of-service attack against JBoss Application Server. JBoss logging process failed, hindering forensics.
- JBoss Application Server does not enforce EJB 2.0 requirement that services not spawn their own threads.

Recommendations
- Implement a least privileges authorization scheme to limit clients’ ability to call arbitrary code.
- Use static checking at or before deployment time to detect faulty code.
Findings: SQL Injection

Description
• SQL injection exploits user input that is either incorrectly filtered for string literal escape characters embedded in SQL statements or not strongly typed and thereby unexpectedly executed.

Results
• JBoss Application Server contains no protection against string-literal database input. Preventing SQL injection is left to developers of business services.

Recommendations
• Use static analysis to detect unsafe database access in business services.
• Incorporate SQL injection into quality assurance testing of business services.
• Add a proxy layer in front of databases that can sanitize input.
Stress Test: Application Flooding

Description

- Generate service requests at a high rate, then study the effects on the service and on other clients attempting to use the service
- We studied two forms of application flooding:
  - JNDI (Java Naming and Directory Interface) lookup: 2000 requests in a tight loop
  - RMI invocation flooding on a persistent EJB service: 100 requests in parallel

Results

- JNDI flooding had minimal effect at this scale
- RMI flooding increased round-trip time ~70% but did not cause any failures

Recommendations

- Use a proxy layer to limit rate of inbound requests
- Deploy a statistical anomaly based intrusion detection system
- Incorporate application flooding into quality assurance tests to identify vulnerable services
Direct Attack: Password Cracking

Description
- A password cracking attack attempts to guess passwords through repeated trials of passwords, usually from an attack dictionary
- Insiders may attempt password cracking to escalate privileges

Results
- The host OS of the Reference Implementation (Red Hat Enterprise Linux 5) is vulnerable to password cracking attacks since attack scripts can make unlimited login attempts without impacting account status
- Reference Implementation ships with default password for JBoss Administration console
- JBoss console is also susceptible to unlimited password attempts

Recommendations
- Configure host OS and applications to lock out or delay login attempts after a certain number of failures
- Enable logging to a management station to alert administrators
Direct Attack: Fault Injection

Description

• Write a malicious service and deploy it to the platform.
• The intent is to study what would happen if an attacker co-opted a service, rather than how that might occur
• We introduced two main types of faults: CPU overload and memory exhaustion

Results

• Memory exhaustion is an effective denial-of-service attack against JBoss Application Server. Its logging process failed, hindering forensics.
• As shipped, Reference Implementation has no Java security manager configured, making it easy to deploy unsigned code.
• JBoss Application Server does not enforce EJB 2.0 requirement that services not spawn their own threads. No warning is generated at run time or deployment time.

Recommendations

• Implement a least privileges authorization scheme to limit clients’ ability to call arbitrary code.
• Use static checking at or before deployment time to detect faulty code.
Direct Attack: Simulated Node Failure

Description
• Shut down one host in the platform and see what happens to other hosts.
• The purpose is to see whether the failure is recoverable, not to find a way to prevent the failure.

Results
• JBoss Application Server could not automatically re-connect to PostgreSQL after the database re-started.

Recommendations
• Implement a secure heartbeat monitoring protocol with low failure-detection latency and resistance to spoofing attacks.
• Properly configure JDBC connectors to automatically re-try failed connections.
• Use proxy servers that automatically re-establish connections and fail over to redundant service instances.
Summary of Recommendations

• Utilize available security features in PostgreSQL server and JBoss Application server.
  – Security features were used inconsistently in the Reference Implementation.
  – This may be attributable to lack of documentation and training, or insufficient time to thoroughly secure the configuration.

• The SOA platform does not ensure business services will be secure. Use quality assurance practices.

• Security should be a primary consideration in SOA design. Effective defenses will have an impact on the software topology and system architecture.

• Design *containment regions* using firewalls and proxy servers to isolate critical infrastructure from attackers.