Architecture-Based Self-Protecting Software Systems
Adnan Alawneh
CS 788
Outline

Introduction
- What is the problem?
- What is the solution?
- Motivating Example
- How ABSP addresses these challenges?

Architecture Patterns For Self-Protection
- Protective Wrapper Pattern
- Rejuvenation Pattern
- Agreement-based Redundancy

Rainbow Framework Overview
- Tactics in Rainbow
- Strategies in Rainbow
- Choosing Strategies in Rainbow

Conclusion

Future Work

References
Introduction
What is the problem?

– Environment is **dynamic** and constantly changing and the system has to handle **unforeseen** situations in terms of work loads, resources, etc.
  
  • This is absolutely true in the area of security as well where new **vulnerabilities** are often discovered and **attackers** are constantly developing **new** exploits and **threats**.

Traditional approaches such as code reviews, using white list input validation, and tools to detect intrusion detection and SQL injection, etc. are **great** but they are statically built or **hardwired**, into the system. Those solutions cannot cope with these dynamically evolved challenges.

– Most current approaches are **perimeter-based** and they are limited, in the sense:
  
  • They focus on a **particular line of defense** such as network, host, middleware, or application level.
  
  • They focus on a **specific category of threat** or a specific mitigation technique such as intrusion detection, DoS, etc.

These approaches are essential but they provide point solutions that lack the overall picture and they do not consider other **business properties** such as performance, cost, availability, etc.
What is the solution? How it addresses those challenges?

Architecture-Based Self Protecting Software System (ABSP) where a system may dynamically **re-architect** itself using **repeatable patterns** as requirements and environments change.

**Example:**

- Znn is a web-based client-server system
- It has a load balancer with a pool of replicated web servers and a database server
- Znn separates the **meta-level** subsystem from the **base-level** subsystem
- Meta component called **ARchitecture Manager (ARM)**
- ARM implements MAPE-K loop to **monitor** the components and connectors in the system and **adapt** them according to the **self-protection goals**
What is the solution? How it addresses those challenges?

How ABSP addresses these challenges?

- Separation of concerns – separates the application logic of the system from the self-protecting mechanisms which allows those mechanisms to evolve independently.

- Defense in depth – instead of relying on perimeter security and focusing on a particular line of defense, it provides basis to deal with threats at different levels.

- Impact Analysis – provides a basis for reasoning about security properties and their impact on other business properties such as performance, cost, availability, etc.
Architecture Patterns For Self-Protection

Protective Wrapper Pattern

- This technique is used before in SITAR[1] system where it detects and reacts to intrusions.
- If the ARM senses that the system is under attack, for example a sensor detected a buffer overflow event from the system log of the web server or too many requests from the same IP address or requests are taking longer than expected, then the ARM can place an Application Guard in front of the load balancer where any incoming request or outgoing response is intercepted. The ARM can instruct the Application Guard to cut off requests from a suspicious users or adjust their trustworthiness. Or the ARM can instruct the Application Guard to enforce policies on certain events.

Rejuvenation Pattern

- This technique is used before in TALNET[2] system where it proactively “revive” the system to a good state. It uses “cyber moving target” approach by migrating the system across different platform.
- The ARM can utilize a pool of idle and active servers where it can issue a rejuvenation commands periodically. It can as well utilize properties like threat level to adjust the rejuvenation interval if it senses that the attack level has increased.
- The ARM can be more sophisticated where it alternates between two web server implementations such as Microsoft IIS and Apache.
- The ARM can exploit past knowledge, for example restart a more powerful server based on recent work loads.

1 Wang et al. 2003
2 Okhravi et al. 2010, 2012
Agreement-based Redundancy

- This technique is a middleware level defense that configures a cluster of redundant application servers under a Byzantine agreement protocol [3].
- The ARM maintains a number of identical active database servers. A DB Guard connector is placed between the web servers and the databases.
- The ARM will inform the DB Guard of the number of replicas and the quorum thresholds that can be used to check the requests against.
- DB Guard will execute all database requests on the primary database node and it will execute an algorithm to detect malicious requests such as checking predefined properties like the number of news articles.
- Once a malicious request is detect, the DB Guard will inform the ARM that will in turn take countermeasures such as nullifying the attacker session or disable his account.

```
string kw = request.getParameter("keyword");
string query = "SELECT * FROM my_news WHERE keyword = \\
xyz' + kw + \\
xyz'; DELETE FROM news; --
```

All this is great! But, how would we know if we are under attack? If the system is under attack, which tactic/strategy to choose from? Should we throttle the traffic from the attacker or should we black list the attacker? What is the impact on the overall system?
Rainbow Framework Overview

- Rainbow framework can be utilized as the ARchitectural Manger (ARM).
- Rainbow uses architectural models of a software system as the basis for reasoning about whether the system is operating within an acceptable criteria. Otherwise, Rainbow chooses appropriate adaptation strategies to return the system to normality.
- Rainbow has different customization points.
  - Types and Properties
    such as the request rates of clients and the probability that a client is engaging a DoS for example
  - Rules/Constraints
    Check the correctness of the model. If it is not within the acceptable criteria then the mitigation strategy is applied
  - Tactics and Strategies
  - Operators
    such as AddService or RemoveService operators
  - Mappings
    When attack is detected, rainbow aggregates this data into actionable information within gauges. It then uses this information to update the architectural model via effectors
Tactics in Rainbow

- Different tactics exist to thwart different attacks. For example, the table shows the tactics that can be used to change the system configuration if a system is under DoS attack, either absorb the attack or suppress it.

- Tactics in Rainbow are expressed through the Stitch adaptation language.

```plaintext
  tactic addCaptcha () {
    condition {exists lb:D.ZNewsLBT in M.components | !lbcaptchaEnabled;}
    action {
      set lbs = {select l : D.ZNewsLBT in M.components | !l.captchaEnabled;};
      for (D.ZNewsLBT l : lbs) {
        M.setCaptchaEnabled (l, true);
      }
    }
    effect {forall lb:D.ZNewsLBT in M.components | lb.captchaEnabled;}
  }
```

- Line 2 – a condition that says the tactic is chosen if any load balancer does not have Captcha enabled
- Line 4-6 – an action that selects the set of load balancers with Captcha disabled and calls an operation to enable it
- Line 9 – says the tactic will proceed only if all load balancers will have Captcha enabled
Strategies in Rainbow

• Strategies – Common patterns that combine different tactics. For example, in the case of DoS the conditions under which throttling is applicable overlap the condition under which blackholding applies.
  – which tactic to choose first?
  – What about uncertainty and timing?
  – What is the effect on the system of choosing one over the other?

• Strategies in Rainbow are expressed through the Stitch adaptation language.

```java
1 strategy Challenge [unhandledMalicious || unhandledSuspicious] {
2   t0: (cNotChallenging) -> addCaptcha () @5000 {
3     t0a: (success) -> done;
4     t0b: (default) -> fail;
5   }
6   t1: (!cNotChallenging) -> forceReauthentication () @5000 {
7     t1a: (success) -> done;
8     t1b: (default) -> fail;
9   }
10 }
```
Choosing Strategies in Rainbow

1. Define the quality objective – (Utility functions)

Four quality objectives are defined in deciding how to mitigate DoS:

- **Cost**: organization is interested in minimizing the cost incurring from operating the infrastructure such as number of active servers.
- **Maliciousness**: organization is interested in minimizing the cost that corresponds to resources exploited by malicious clients.
- **Response time**: users are interested in experiencing service with minimum response time.
- **Annoyance**: user’s level of annoyance as a side effect of defensive mechanism such as completing a Captcha.

**Business preferences** for DoS scenarios - to allow reasoning about multiple concerns we assign a specific weight to each one of them.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Priority</th>
<th>$w_{UR}$</th>
<th>$w_{UM}$</th>
<th>$w_{UC}$</th>
<th>$w_{UA}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Minimizing number of malicious clients.</td>
<td>0.15</td>
<td>0.6</td>
<td>0.1</td>
<td>0.15</td>
</tr>
<tr>
<td>2</td>
<td>Challenge Optimizing good client experience.</td>
<td>0.3</td>
<td>0.3</td>
<td>0.1</td>
<td>0.3</td>
</tr>
<tr>
<td>3</td>
<td>Keeping cost within budget.</td>
<td>0.2</td>
<td>0.2</td>
<td>0.4</td>
<td>0.2</td>
</tr>
</tbody>
</table>
Choosing Strategies in Rainbow

2. What is the impact of a tactic on the quality of concern?

<table>
<thead>
<tr>
<th>Tactic</th>
<th>Response Time (R)</th>
<th>Malicious Clients (M)</th>
<th>Cost (C)</th>
<th>User Annoyance (A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>enlistServers</td>
<td>-1000</td>
<td>0</td>
<td>+1.0</td>
<td>0</td>
</tr>
<tr>
<td>lowerFidelity</td>
<td>-500</td>
<td>0</td>
<td>-0.1</td>
<td>0</td>
</tr>
<tr>
<td>addCaptcha</td>
<td>-250</td>
<td>-90</td>
<td>+0.5</td>
<td>+50</td>
</tr>
<tr>
<td>forceReauthenticate</td>
<td>-250</td>
<td>-70</td>
<td>0</td>
<td>+50</td>
</tr>
<tr>
<td>blackholeAttacker</td>
<td>-1000</td>
<td>-100</td>
<td>0</td>
<td>+50</td>
</tr>
<tr>
<td>throttle Suspicious</td>
<td>-500</td>
<td>0</td>
<td>0</td>
<td>+25</td>
</tr>
</tbody>
</table>

- All tactics reduce response time. Some cause more reduction than others like enlistServers and blackholeAttacker.
- The tactics addCaptcha and blackholeAttacker reduces the percentage of malicious clients, whereas enlistServers and lowerFidelity has no effect.
- Cost is increased in case of enlistServers and addCaptcha since additional resources are needed to absorb extra traffic or to process a captcha.
- User’s level of annoyance is increased in case of addCaptcha and forceReauthenticate. It also increases as well in case of blackholeAttacker and throttle Suspicious in the sense that there is a risk of misdetection of a malicious client.
Choosing Strategies in Rainbow

2. What is the impact of a strategy? How can Rainbow choose the best among them.

Example:
Suppose the adaptation cycle is triggered in system state
\([1500 \text{ (R)}, 90 \text{ (M)}, 2 \text{ (C)}, 0 \text{ (A)}]\)

Rainbow will evaluate the three different strategies: Challenge, Eliminate and Outgun to choose the one that gives the best utility.

For Challenge
1- The adaptation manager in Rainbow uses Stochastic Model of strategy. The strategy tree results from the aggregate impacts of its children
\[0.5[-250,-90,+0.5,+50]+0.5[-250,-70,0,+50]=[-250,-80,+0.25,+50]\]

2- Merge it with the current system condition to get the expected system condition after strategy execution
\[[1500,90,2,0]+[-250,-80,+0.25,+50]=[1250,10,2.25,50]\]

3- Map it to the utility space
\[U_R(1250), U_M(10), U_C(2.25), U_A(50)=[0.625, 0.933, 0.25, 0.5]\]

4- apply the utility preferences
\[0.625*0.3+0.933*0.3+0.25*0.1+0.5*0.3=0.6425\]

Challenge score 0.6425, Eliminate score 0.6325 and Outgun score 0.553. Therefore Challenge will be selected.
Conclusion

• The papers utilizes architectural patterns and threat mitigation strategies for self-protecting purposes.
• The papers relate security concerns to other business properties which is a challenge where security concerns can conflict with other quality attributes such as response time.
• Software architecture provides a holistic view of the system and therefore it is the correct instrument to choose to evaluate tradeoffs automatically.
• The papers do not describe how to restore the system once the attack is over.

Future work

• There is a lack of good system security estimators in the literature.
• Most research do not separate self-protecting system from the base-system.
• Most research in the literature assume that the self-protecting system is safe.

I start to explore ways to extend architecture-based self-protection by utilizing “cyber moving target” approach and residual resources in the cloud infrastructure to anticipate attacks on the self-protecting system and move it accordingly by using predictive and moving target techniques.
References


