Automatic Software Self-Healing
Present and Future

Angelos D. Keromytis
Columbia University
angelos@cs.columbia.edu
Overview of talk

- Self-healing software (SHS)
- "definition"
- rationale
- example
- What's next?

Work supported over the years by AFOSR, NSF, New York State, Google, DTO/IARPA
What is SHS?

- Adaptive, introspective system design applied to security
- Learn from past failures by changing self
- Different (complementary) security paradigm
  - “fail once” vs. “fail never” vs. “fail-stop”
- long-term reliability vs. short-term integrity
Why SHS for security

- Different approach to security
  - Adaptive systems have been investigated in other contexts, with promising results
  - Performance, provisioning, etc.
- Complements traditional/existing approaches
- We may not have a choice
Current defenses (practice)

- Firewalls and network defenses (IDS, IPS)
- Polymorphism, encryption, performance
- Defenses increasingly need to go to the host...
- ...or the host needs to go to the network
- Proactive software security
- Performance, incentives/externalities, cost, inertia
Current research efforts

- Primary focus on blocking
- Automatic bad-input signature generation
- Generalized via grey-box testing
- Statistical modeling of good or bad traffic
- Use oracle to confirm guesses
- Model code behavior during execution
Security Tradeoffs

Performance

Filtering

Stack-guard
ASLR
ISR
DTA

Accuracy/coverage

Availability
SHS workflow

monitor
SHS workflow

monitor
diagnose
SHS workflow
SHS workflow

monitor → Detect anomalous event → diagnose
SHS workflow

- monitor
  Detect anomalous event
- diagnose
- adapt
SHS workflow

monitor $\rightarrow$ Detect anomalous event $\rightarrow$ diagnose $\rightarrow$ adapt
SHS workflow

- monitor
  - Detect anomalous event
- diagnose
  - Identify fault
- adapt
SHS workflow

- Monitor
- Detect anomalous event
- Diagnose
- Identify fault
- Test
- Adapt
SHS workflow

- **monitor**
- **diagnose**
- **test**
- **adapt**

- Detect anomalous event
- Identify fault
SHS workflow

- Monitor
  - Detect anomalous event
- Diagnose
  - Identify fault
- Test
  - Generate fix
- Adapt
SHS workflow

monitor ➔ detect anomalous event ➔ diagnose

diagnose ➔ identify fault ➔ adapt

adapt ➔ generate fix ➔ test

test ➔ examine ➔ monitor
SHS workflow

- **monitor**
  - Detect anomalous event
  - Deploy fix

- **diagnose**
  - Identify fault

- **test**
  - Generate fix

- **adapt**
Advantages

- Minimally invasive to production systems
- Lightweight detection, heavyweight analysis
- Localized adaptation -> cost containment
- Integrates different areas of security
- Detection, analysis, remediation, testing
- Let the attacker do the difficult part (find exploitable vulnerability)
Caveats

- System must be attacked at least once
- System may be compromised at least once
- Detection only as good as the monitoring
- Analysis may be complex and expensive
- Recovery/mitigation may be impossible
- Testing will be incomplete
Questions

- What does it mean to be introspective?
- What types of adaptation?
- Who does the adaptation?
- How is adaptation instantiated?
- How much human involvement is needed?
- What about autoimmune dysfunction?
Introspection

- Ability to observe the behavior of the system
- Detect fault manifestation
- Extract sufficient information to aid analysis phase
In practice

Ultimately, introspection means deploying fault-detection sensors

- specific to fault types
- “blanket” vs. targeted/partial deployment
- performance-effectiveness tradeoff

Fully vs. partially introspective

can the supervisor self-heal?
Adaptation

- Take action that prevents/mitigates future instances of the fault
  - block (filter) inputs
  - reconfiguration
  - log and replay
  - selective randomization
  - immunization
  - immunization with recovery
Suicidal software systems?

- System could be induced to attack itself
  - false faults
  - side effects of adaptation
  - bugs in the supervisor
- Open problem
ASSURE

- Example self-healing system
- Targets faults leading to application crashes or low-level compromises (e.g., buffer overflows)
  - extensible via additional detectors
- Immunizes software against specific vulnerability
  - immune to mutating attacks
- maintains availability
- Operates in pure-binary environment
High-level Architecture

1. Sensors
2. FAULT
3. Rescue points
4. Generate patch
5. Test patch
6. Update application
7. Recover execution

- Bad Input
  - Offline
- Input
  - Online

- Rescue Point Discovery
- Patch Generator
- Testing Environment
- Error Virtualization
- Patch Insertion
- Executing Application

[Diagram showing the flow of the process]
Sensors

- Lightweight detectors on the application simply need to give indication of failure
  - watchdog process
  - ProPolice, StackGuard, etc.
Analysis environment

- Copy of production software
- Instrumented to keep track of interesting information
  - memory regions
  - call graph
  - program state (memory)
- I/O
- Obtain detailed information about conditions leading to fault
Adaptation

- Binary patch that blocks attack and masks fault
- Change code in the specific region
- Use checkpoint/restart at the function level to recover program execution to Rescue Point
- Force process to pretend that the vulnerable code reported an error condition

Not always possible or safe

- Abuse existing error-handling code or ancestor in callstack
Rescue Points

- Locations in the code known to handle faults
- Mapping between set of faults that could occur and those that can be handled by program code
- Recover using program code
Rescue Point Discovery

- Dynamic analysis
- Profile applications behavior under "bad" input (fuzzing)
- Discover common error codes
- Works on stripped binaries
- Happens off-line, once
- Reusable across machines, users
Rescue Point Algorithm

- Replay failure
- Detect failure
- Extract stack trace
- Find rescue point that is closest to failure
- If suitable, select error-return value
- Test rescue point
  - Survivability, correctness, performance
Analysis of Apache bug
Analysis of Apache bug

proxy_run_scheme_handler() → ap_proxy_ftp_handler() → ap_pass_brigade() → ap_proxy_send_dir_filter()

(1) Malicious Input
Analysis of Apache bug

(1) Malicious Input

proxy_run_scheme_handler()

ap_proxy_ftp_handler()

ap_pass_brigade()

(2) Checkpoint

ap_proxy_send_dir_filter()

Take rescue point (ckpt)
Analysis of Apache bug

(1) Malicious Input
proxy_run_scheme_handler()

(2) Checkpoint
ap_proxy_ftp_handler()
ap_pass_brigade()
ap_proxy_send_dir_filter()

(3) Fault Detected
SIGSEGV

(4) Error Virtualization

Pick rescue point
Detect failure
Take rescue point (ckpt)
Analysis of Apache bug

Malicious Input

Take rescue point (ckpt)

Fault Detected

Pick rescue point

Restore & force error

Pick rescue point

Error Virtualization

(1) Malicious Input

proxy_run_scheme_handler()

(2) Checkpoint

ap_proxy_ftp_handler()

(4) Error

(5) Restore

(3) Fault Detected (SIGSEGV)

ap_pass_brigade()

ap_proxy_senddir_filter()
Analysis of Apache bug

1. Malicious Input

2. Checkpoint

3. Fault Detected (SIGSEGV)

4. Error Virtualization

5. Restore

6. Return error 502 (HTTP proxy error)

Pick rescue point

Take rescue point (ckpt)

Detect failure

Restore & force error

Recover execution
Restore & Force Error

- Restore to rescue-point
- Roll-back all processes
- Restore file system
  - consistent memory/file-system view
- Force error return
Why Does This Work?

- Focus on server applications
- Short error propagation distance
- Errors in one request do not affect subsequent requests
- Servers support error handling
- Need to deal with bad/malformed requests
- Programmers are “pretty good”
- They just can’t cover every corner case
Evaluation

- Implemented ASSURE for Linux
- Tested several popular server applications

Metrics
- Survivability
- Correctness
- Performance
Fault-Injection Results

Recovery Rate

thttpd  Apache  Bind  sshd  mysql  wu-ftp

ASSURE
## Bugs

<table>
<thead>
<tr>
<th>Application</th>
<th>Version</th>
<th>Bug</th>
<th>Reference</th>
<th>Value</th>
<th>Depth</th>
<th>Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>2.0.59</td>
<td>NULL dereference</td>
<td>ASF Bug 40733</td>
<td>502</td>
<td>3</td>
<td>Httpperf-0.8</td>
</tr>
<tr>
<td>Apache</td>
<td>2.0.54</td>
<td>off-by-one</td>
<td>CVE-2006-3747</td>
<td>-1</td>
<td>2</td>
<td>Httpperf-0.8</td>
</tr>
<tr>
<td>Apache</td>
<td>1.3.31</td>
<td>Buffer Overflow</td>
<td>CVE-2004-0940</td>
<td>NULL</td>
<td>1</td>
<td>Httpperf-0.8</td>
</tr>
<tr>
<td>MySQL</td>
<td>5.0.20</td>
<td>Buffer Overflow</td>
<td>CAN-2002-1373</td>
<td>1</td>
<td>2</td>
<td>Sql-bench 2.15</td>
</tr>
<tr>
<td>Squid</td>
<td>2.4</td>
<td>Input Validation</td>
<td>CVE-2005-3258</td>
<td>VOID</td>
<td>1</td>
<td>Webstone 2.5b3</td>
</tr>
<tr>
<td>OpenLDAP</td>
<td>2.3.39</td>
<td>Design Error</td>
<td>CVE-2008-0658</td>
<td>80</td>
<td>1</td>
<td>DirectoryMark 1.3</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>8.0</td>
<td>Input Validation</td>
<td>CVE-2005-0246</td>
<td>0</td>
<td>1</td>
<td>BenchmarkSQL 2.3.2</td>
</tr>
<tr>
<td>ISC Bind</td>
<td>8.2.2</td>
<td>Input Validation</td>
<td>CAN-2002-1220</td>
<td>-1</td>
<td>2</td>
<td>Dnsperf 1.0.0.1</td>
</tr>
</tbody>
</table>
Normalized Performance

Normalized performance for various applications:

- apache 1.3.31
- apache 2.0.59
- apache 2.0.54
- bind
- mysql
- squid
- openldap
- postgresql

Comparison with ASSURE and ASSURE with Faults.
Recovery Times

![Bar chart showing recovery times for various applications. The x-axis represents different applications: apache 1.3.31, apache 2.0.59, apache 2.0.54, bind, mysql, squid, openldap, postgresql. The y-axis represents time in seconds, ranging from 0 to 5. Each application has two bars, one for ASSURE and one for Restart. The times vary for each application.]
Healing Time

<table>
<thead>
<tr>
<th>Application</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache 1.3.31</td>
<td>20</td>
</tr>
<tr>
<td>apache 2.0.59</td>
<td>80</td>
</tr>
<tr>
<td>apache 2.0.54</td>
<td>40</td>
</tr>
<tr>
<td>bind</td>
<td>60</td>
</tr>
<tr>
<td>mysql</td>
<td>60</td>
</tr>
<tr>
<td>squid</td>
<td>32</td>
</tr>
<tr>
<td>openldap</td>
<td>60</td>
</tr>
<tr>
<td>postgresql</td>
<td>32</td>
</tr>
</tbody>
</table>
Management console
Console
Console
Limitations

- Cannot guarantee program path on recovery
- Could bypass security checks (e.g., sshd)
- Could exhibit long-term side-effects
- Possible approach: CW integrity constraints
  [Locasto 2007]
- Recovery for multi-process needs improvement
- Forward-error propagation
Other work in SHS

- Failure-oblivious computing [OSDI 04]
- Data-structure repair [ICSE 05]
- Rx: Treating bugs as allergies [SOSP 05]
- STEM [USENIX 05]
- DYBOC [ISC 05]
- Genetic programming to find patches [ICSE 09]
Self-healing Systems

What we need is self-healing systems

1. cleanup
2. forward immunization
3. work conservation

First two properties happen manually today

restore and patch

Note analogy of property (3) with Availability
Why?

- Compromises occur in many different ways
  - trojans, social phishing, web, software, ...
- Often, not obvious until days or weeks later
- SHS doesn’t help with that
- Restore conundrum
  - partial restore is takes time, risks missing data
  - full restore risks re-compromise
Open systems problem; possible elements:

- system restore and immunization
- forensic analysis
- should work on any desktop or server
- since its first boot
- impact analysis
- dis-entanglement
Thoughts

- SHS must be automated
- How do we extend SHS across single-system boundaries?
  - global "undo"
- Disk is cheap, but data use keeps increasing too
Conclusions

- Self healing represents a promising and interesting way to deal with many security and reliability problems in today’s software.
- Complements other defenses.
- Promising results.
- Several major challenges ahead.
- Bigger problem: from healing software to healing systems.