My talk is going to be about “Reactive Protection Mechanisms” and Why do we need them to deal with buggy and crash-prone software.
My talk is going to be about “Reactive Protection Mechanisms” and Why do we need them to deal with buggy and crash-prone software.
Overview

• Self-Healing Software (SHS)
  • Definition & Rationale
• Example: Assure
• What’s Next?

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

3 Important Points: What is it, Use–Case, What’s next for our Research (Kernel Assure)
What is SHS?

Adapt: react to a changing environment (block, immunize), by monitoring (introspection: detect the fault, extract information from raw data) we want to adjust.

4 stage Observe–Orient–Decide–Act Feedback Loop

Learn (intelligence): by adapting.
What is SHS?

- Adaptive, introspective system design applied to security
- Learn from past failures by changing self

Adapt: react to a changing environment (block, immunize), by monitoring (introspection: detect the fault, extract information from raw data) we want to adjust.

4 stage Observe–Orient–Decide–Act Feedback Loop

Learn (intelligence): by adapting.
What is SHS?

- Adaptive, introspective system design applied to security
- Learn from past failures by changing self

Different Security Paradigm: (example of learning from attack)

Security Paradigm Shift Rationale: Gödel's Incompleteness Theorem: any logical model of reality is incomplete (and possibly inconsistent) and must be continuously refined/adapted in the face of new observations.

WHY: any (except trivial) software program has pseudo-infinite (extremely large) many variables to account for, Exception Handling accounts for anticipated control flow. BUT, what ends up happening in zero-day exploits is that someone finds an UnAnticipated control flow. The Saying: “Give me infinite money and infinite time”
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

Different Security Paradigm: (example of learning from attack)

Security Paradigm Shift Rationale: Gödel's Incompleteness Theorem: any logical model of reality is incomplete (and possibly inconsistent) and must be continuously refined/adapted in the face of new observations.

WHY: any (except trivial) software program has pseudo-infinite (extremely large) many variables to account for, Exception Handling accounts for anticipated control flow. BUT, what ends up happening in zero-day exploits is that someone finds an UnAnticipated control flow. The Saying: “Give me infinite money and infinite time”
Observe: Detecting anomalous event
Orient: by identifying the fault
Decide: Generate afix
Act: deploying that fix
Observe: Detecting anomalous event
Orient: by identifying the fault
Decide: Generate afix
Act: deploying that fix
Observe: Detecting anomalous event
Orient: by identifying the fault
Decide: Generate afix
Act: deploying that fix
Observe: Detecting anomalous event
Orient: by identifying the fault
Decide: Generate a fix
Act: deploying that fix
SHS workflow

Observe: Detecting anomalous event
Orient: by identifying the fault
Decide: Generate a fix
Act: deploying that fix
SHS workflow

4 stage Observe–Orient–Decide–Act
Feedback Loop
SHS workflow

4 stage Observe–Orient–Decide–Act Feedback Loop
4 stage Observe–Orient–Decide–Act Feedback Loop
How to direct one's energies to defeat an adversary and survive.

- **Gödel's Incompleteness Theorem**: any logical model of reality is incomplete (and possibly inconsistent) and must be continuously refined/adapted in the face of new observations.

- **Heisenberg's Uncertainty Principle**: there is a limit on our ability to observe reality with accuracy.

- **Second Law of Thermodynamics**: The entropy of any closed system always tends to increase, and thus the nature of any given system is continuously changing even as efforts are directed toward maintaining it in its original form.
Tuesday, June 23, 2009

How to direct one's energies to defeat an adversary and survive.

- **Gödel's Incompleteness Theorem**: any logical model of reality is incomplete (and possibly inconsistent) and must be continuously refined/adapted in the face of new observations.
- **Heisenberg's Uncertainty Principle**: there is a limit on our ability to observe reality with accuracy.
- **Second Law of Thermodynamics**: The entropy of any closed system always tends to increase, and thus the nature of any given system is continuously changing even as efforts are directed toward maintaining it in its original form.
Tuesday, June 23, 2009

How to direct one's energies to defeat an adversary and survive.

- **Gödel's Incompleteness Theorem**: any logical model of reality is incomplete (and possibly inconsistent) and must be continuously refined/adapted in the face of new observations.

- **Heisenberg's Uncertainty Principle**: there is a limit on our ability to observe reality with accuracy.

- **Second Law of Thermodynamics**: The entropy of any closed system always tends to increase, and thus the nature of any given system is continuously changing even as efforts are directed toward maintaining it in its original form.
How to direct one's energies to defeat an adversary and survive.

- **Gödel's Incompleteness Theorem**: any logical model of reality is incomplete (and possibly inconsistent) and must be continuously refined/adapted in the face of new observations.

- **Heisenberg's Uncertainty Principle**: there is a limit on our ability to observe reality with accuracy.

- **Second Law of Thermodynamics**: The entropy of any closed system always tends to increase, and thus the nature of any given system is continuously changing even as efforts are directed toward maintaining it in its original form.
I. All decisions are based on observations of the evolving situation.

II. Observed information must be processed to orient it for further decision making.

III. Determination of course of action based on one's current perspective. (adaptation, survival)

IV. Physically playing out of decisions.

How to direct one's energies to defeat an adversary and survive.

- **Gödel's Incompleteness Theorem**: any logical model of reality is incomplete (and possibly inconsistent) and must be continuously refined/adapted in the face of new observations.

- **Heisenberg's Uncertainty Principle**: there is a limit on our ability to observe reality with accuracy.

- **Second Law of Thermodynamics**: The entropy of any closed system always tends to increase, and thus the nature of any given system is continuously changing even as efforts are directed toward maintaining it in its original form.
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)

From this set of considerations, Boyd concluded that to maintain an accurate or effective grasp of reality one must undergo a continuous cycle of interaction with the environment geared to assessing its constant changes.
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

As with any security paradigm there are Pro’s & Con’s
Re-stating our definition, let’s add more granularity to this definition. By defining how we implement an adaptive and introspective system.

First, introspective→
Re-stating our definition

• Adaptive, introspective system design applied to security

• Learn from past failures by changing self

Re-stating our definition, let’s add more granularity to this definition. By defining how we implement an adaptive and introspective system.

First, introspective→
What we mean by introspection: is the ability to see without one’s environment. This can be applied to application based introspection from kernel data or VM-based introspection from kernel drivers.
Ultimately, introspection means deploying fault-detection sensors specific to fault types.

“Holes –not–> “Wholes” make you wonder what I was thinking.

What type of error? As good as your sensor.
“Holes –not–> “Wholes” make you wonder what I was thinking.

What type of error? As good as your sensor.
• 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?

Monitor the monitor of the Monitor.
Adaptation

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

What do we mean by adaptation: respond to stimulus and change to accommodate environmental changes.
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

Gödel's Incompleteness Theorem: any logical model of reality is incomplete (and possibly inconsistent) and must be continuously refined/adapted in the face of new observations.

Why: any (except trivial) software program has pseudo-infinite (extremely large) many variables to account for, Exception Handling accounts for anticipated control flow. BUT, what ends up happening in zero-day exploits is that someone finds an UnAnticipated control flow. The Saying: “Give me infinite money and infinite time”
Suicidal software systems?

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

There are also problems to consider.
ASSURE

Automatic Software Self-healing Using Rescue-Points

Tuesday, June 23, 2009

Working prototype presented at ASPLOS (Arch Supp. for Prog Lang and OS)
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


What is it?
Implementation of Self-Healing System
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

Possible Faults

Faults handled by application

BUT FIRST, define Rescue Points:
Rescue point: It’s a location in the program code that is known or at least suspected to handle errors correctly.
Leverage recoverable exceptions in program code!! A form of exception handling for Unanticipated execution!

We use error return values to steer program execution away from the failure. To discover what values to use, we profile application to discover how they correctly behave under "bad" input. We create "bad" input by using application-specific fuzzing and record the return values used by functions. Since not all functions will return errors. We use a heuristic to identify functions that return pointers. If they don't use NULL, then we are forced to skip them. In fact, we envision that users can share their recovery profiles.
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 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

(1) Malicious Input
Analysis of Apache bug

proxy_run_scheme_handler() → ap_proxy_ftp_handler() → ap_pass_brigade() → ap_proxy_senddir_filter()

(1) Malicious Input

(2) Checkpoint

Take rescue point (ckpt)

Tuesday, June 23, 2009
Analysis of Apache bug

proxy_run_scheme_handler() → ap_proxy_ftp_handler() → ap_pass_brigade() → ap_proxy_senddir_filter()

(1) Malicious Input
(2) Checkpoint
(3) Fault Detected (SIGSEGV)
(4) Error Virtualization

Pick rescue point
Detect failure

Take rescue point (ckpt)

Tuesday, June 23, 2009
An analysis of Apache bug

- Restore & force error
- Pick rescue point

(1) Malicious Input

proxy_run_scheme_handler()

ap_proxy_ftp_handler()

(2) Checkpoint

(3) Fault Detected (SIGSEGV)

ap_pass_brigade()

ap_proxy_send_dir_filter()

(4) Error Virtualization

(5) Restore

Tuesday, June 23, 2009
Analysis of Apache bug

**Malicious Input**

1. **Malicious Input**
2. **Checkpoint**
3. **Fault Detected** (SIGSEGV)
4. **Error Virtualization**
5. **Restore Checkpoint**
6. **Return error 502(HTTP proxy error)**

**Restore & force error**

- **Pick rescue point**
- **Detect failure**
- **Take rescue point (ckpt)**
- **Recover execution**

**Tuesday, June 23, 2009**
• 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
Why Does This Work?

Example: redirecting reads that would otherwise cause addressing errors and discarding writes that would otherwise corrupt critical data structures (call stack) localizes the effects of memory errors, prevents addressing exceptions from terminating the computation, and enables the server to continue on to successfully process subsequent requests. The overall result is a substantial extension on the range of requests that the server can successfully process.

Localized errors in the computation for one request tend to have little or no effect on the computations for subsequent Requests.

Thesis: failure oblivious computing converts requests that trigger unanticipated and dangerous exec paths into anticipated invalid Inputs which the error handling in the server rejects. Bc servers tend to have small error propagation distances.

Hypothesis: was that, for at least some programs, this continued execution would produce acceptable results.

Found 1: acceptable continued execution

Found 2: acceptable Performance
Evaluation

- Implemented ASSURE for Linux
- Tested several popular server applications

Metrics
- Survivability
- Correctness
- Performance

Survivability: Make sure that our fix allows the application to continue servicing requests
Correctness: Make sure that our fix does not change program semantics. That is that the application still gives us the results we expect.
Performance: Make sure that our fix does not unduly impoes on application performance
<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>

NOTE: this is a binary system, we used open-source for validation.
Overhead dependent on rescue point. Re-emphasis on “STEM” & “Failure Oblivious Computing”:

Localized errors in the computation for one request tend to have little or no effect on the computations for subsequent Requests.
<table>
<thead>
<tr>
<th>Application</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache 1.3.31</td>
<td>1.2</td>
</tr>
<tr>
<td>apache 2.0.59</td>
<td>1.4</td>
</tr>
<tr>
<td>apache 2.0.54</td>
<td>1.6</td>
</tr>
<tr>
<td>bind</td>
<td>2.5</td>
</tr>
<tr>
<td>mysql</td>
<td>3.1</td>
</tr>
<tr>
<td>squid</td>
<td>4.8</td>
</tr>
<tr>
<td>openldap</td>
<td>0.5</td>
</tr>
<tr>
<td>postgresql</td>
<td>3.3</td>
</tr>
</tbody>
</table>

Checkpoint Restart, based on number of threads (these were all benchmarked), specifically file-system restore overhead! Can be mitigated by running an in-memory filesystem.

**BUT THERE ARE PROBLEMS!**
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

Although forcing an error is better than masking errors, it still does not provide strong guarantees that we are not changing program semantics. For example, an attacker can game our system to bypass some security checks in an authentication mechanism.

- One approach, assuming some application knowledge, would be to identify certain sensitive functions that should not be error virtualized.
- Another limitation of our approach is that our recovery could exhibit long-term side-effects.
- We try to eliminate the chance of that happening by subjecting rescue points to a barrage of requests such that most easy errors such as memory leaks can be easily picked out but it remains a part of future work.
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]

Tuesday, June 23, 2009

Rinard: hides memory error by virtually expanding buffers so that out-of-bound reads/writes can be handled gracefully.

Rx: proposed a technique to deal with non-deterministic errors by changing the underlying environment of a program until it can continue execution without problems.

Rx focused on non-deterministic errors and cannot handle multi-process applications

A characteristic of all these approaches is that they try to mask the occurrence of faults.

This is a problem because you are actively changing program semantics and can lead to unanticipated execution paths.

The approach that we take is different. We force an error and allow the application's code do the cleanup for us.
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

The problem with work conservation is apparent in the context (next slide)
Work Conservation: 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
How?

- 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

Tuesday, June 23, 2009
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

Complement other defenses, provides efficiently fast promising results.
Kernel Assure