Virtualization as a Defensive Technique

Ken Birman

Virtualization as a Defense

- We know that our systems are under attack by all sorts of threats

- Can we use virtual machines as a defensive tool?
- Ideas:
  - Encapsulate applications: infection can't escape
  - Checkpoint virtual state. If something goes badly wrong, roll back and "do something else"
  - Use virtualized platforms to lure viruses in, then study them, design a tailored antivirus solution

2

Encapsulation for Protection

- This is still a speculative prospect
  - Basic idea: instead of running applications on a standard platform, each application runs in its own virtual machine, created when you launch the application
  - The VMM tracks the things the application is doing, such as file I/O
  - If we see evidence of a virus infection, we can potentially undo the damage
    - Obviously, this will have its limits
    - Can't undo messages sent on the network before we noticed the infection! (And this precisely what many viruses do…)

3

Issues raised

- While it is easy to talk about running applications within VMM environments, actually doing it is more tricky than it sounds
  - Many applications involve multiple programs that need to talk to one-another
  - Any many applications talk to O/S services
- Issue is that when we allow these behaviors, we create tunnels that the virus might be able to use too
  - Yet if we disallow them, those applications would break

4

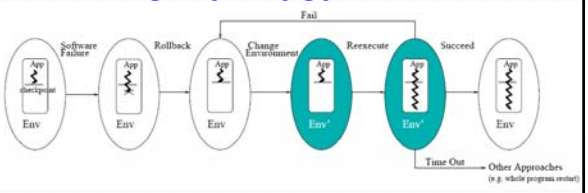
Rx: Treating Bugs as Allergies

Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou
University of Illinois

Motivation

- Bugs are not going away anytime soon
  - We would still like it be highly available
  - Server downtime is very costly
  - Recovery-oriented computing (Stanford/Berkeley)
- Insight
  - Many bugs are environmentally dependent
  - Try to make the environment more forgiving
  - Avoid the bug
- Example bugs
  - Memory issues
  - Data races

Rx: "The main idea"



- Low nominal overhead
- Legacy code friendly

Solution: Rx

- Apply checkpoint and rollback
  - On failure, rollback to most recent checkpoint
  - Replay with tweaked environment
  - Iterate through environments until it works
- Timely recovery
- Low overhead nominal overhead
- Legacy-code friendly

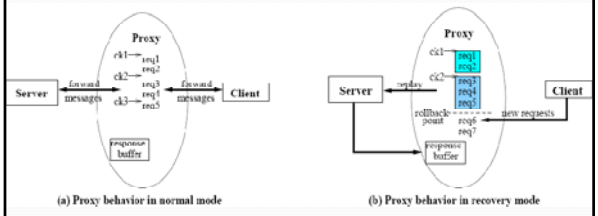
Environmental tweaks

- Memory-related environment
  - Reschedule memory recycling (double free)
  - Pad-out memory blocks (buffer overflow)
  - Readdress allocated memory (corruption)
  - Zero-fill allocated memory (uninitialized read)
- Timing-related environment
  - Thread scheduling (races)
  - Signal delivery (races)
  - Message ordering (races)

System architecture

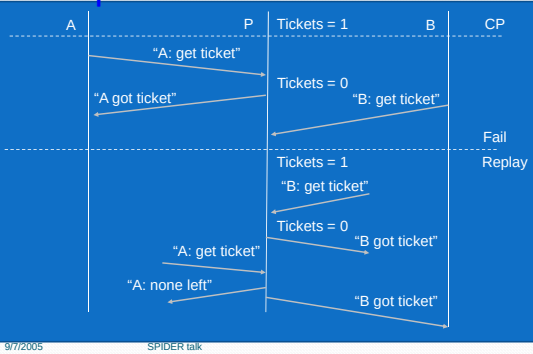
- Sensors
  - Notify the system of failures
- Checkpoint and rollback (Flashback)
  - Only checkpoint process memory, file state
- Environment wrappers
  - Interpose on malloc/free calls, apply tweak
  - Increase scheduling quantum (deadlock?)
  - Replay/reorder protocol requests (proxy)

Proxy



- Introspect on http, mysql, cvs protocols
- What about request dependencies?

Example: ticket sales



Evaluation results

- Recovered from 6 bugs
  - Rx recovers much faster than restart
    - Except for CVS, which is strange
  - Problem: times are for second bug occurrence
    - Table is already in place (factor of two better)
- Overhead with no bugs essentially zero
  - Throughput, response time unchanged
- Have they made the right comparisons?
  - Performance of simpler solutions?

Questions

- Why not apply changes all the time?
  - Much simpler and prevents 5/6 bugs
  - Can recompile with CCured (4 bugs)
    - 30-150% slower for SPECINT95 benchmarks
    - SPECINT95 is CPU bound (no IO)
    - Network I/O would probably mask CPU slowdown
  - Can schedule w/ larger quanta? (1 bug)
  - Where is the performance comparison?
- Sixth bug is fixed by dropping the request
  - Could multiplex requests across n servers

Related work

- Rx lies at the intersection of two hot topics
- Availability + dependability
  - ROC (Berkeley, Stanford)
  - Nooks (Washington)
- Process replay + dependencies
  - Tasmania (Duke)
  - ReVirt, CoVirt, Speculator (Michigan)

Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm

Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage

Slides borrowed from Martin Krogel

Overview

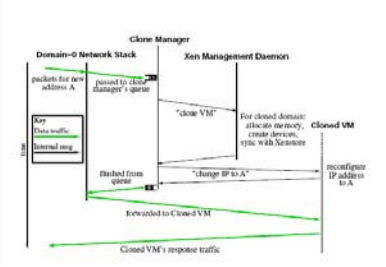
- Definition - What is a honeyfarm?
- Tradeoffs in standard honeyfarms
  - Low-interaction honeypots
  - High-interaction honeypots
- Desirable Qualities
  - Scalability
  - Fidelity
  - Containment
- Potemkin
- Facets
  - Gateway Router
  - Virtual Machine Monitor

What is a honeyfarm?

- Honeypot
  - "A honeypot is an information system resource whose value lies in unauthorized or illicit use of that resource."[1]
- Honeyfarm
  - A collection of honeypots.

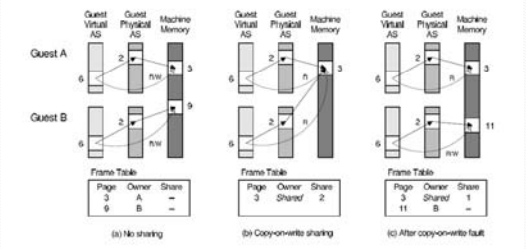
Flash Cloning

- Creates a lightweight virtual machine from a reference copy.



Delta Virtualization

- Allocates memory for a new virtual machine only when it diverges from the reference image (using copy-on-write).



Low-Interaction Honeypots

- Benefits:
  - Simulating a large number of network interfaces, increasing likelihood of being targeted by malicious code. (Scalability)
- Drawbacks:
  - Can't simulate operating systems and applications for all monitored IP addresses, so the effects of malware can't be studied, just sources identified. Also, attacks that requires multiple communications for infection won't be caught. (Fidelity)
  - Since all simulated network interfaces are on one system, if that operating system gets infected, all of the simulated network interfaces are compromised. (Containment)

High-interaction honeypots

- Benefits:
  - By having the actual operating system and applications running on the honeypot, the behavior of malicious code can be analyzed. (Fidelity)
- Drawbacks:
  - Without virtualization, it would require one computer per honeypot. (Scalability)
  - In order to analyze the full extent of malicious code, it's communications with other computers must be studied, not just it's affects on the operating system and applications. (Containment)

Desirable Qualities (Scalability)

- Why is it desirable?
  - With the large number of IP addresses on the internet, the likelihood of a single honeypot being contacted by malicious code is very low
- Why conventional honeyfarms fall short.
  - Conventional networks of honeypot servers do not take advantage of the long idle times between incoming communications, and even when servicing incoming requests, memory requirements are typically low.
- How this system improves scalability.

Desirable Qualities (Fidelity)

- Why is it desirable?
  - The more information we have about a possible threat, the better we are able to defend against it.
- Why conventional honeyfarms fall short.
  - In order to prevent any infection from spreading out of the honeyfarm, the conventional approach with the highest fidelity blocks outbound traffic to anywhere but back to the calling address. Misses interaction with third party machines.
- How this system improves fidelity.
  - Simulating OS and application allows observation of malware's interaction with the system.
  - By creating new virtual machines to play the part of the destination of any outbound traffic, more information can be gained on the behavior of the malware.

Desirable Qualities (Containment)

- Why is it desirable?
  - Without containment, the honeypot could be used by the malicious code to infest other machines (outside of the honeyfarm.)
  - Containment also gives the director of the honeyfarm more information on the behavior of the attack.
- Why conventional honeyfarms fall short.
  - Conventional honeyfarms either don't permit any outbound traffic, which wouldn't trigger any attack that requires handshaking, or only respond to incoming traffic, which interferes with possible DNS translations or a botnet's "phone home" attempt.
- How this system improves containment.
  - Using virtual machines allows multiple honeypots on one server while maintaining isolation. (Scalability without loss of containment :))
  - By using *internal reflection*, outbound communication is redirected to another newly created virtual machine with the IP address of the destination prevents spreading infection.
  - By keeping track of the "universe" of incoming traffic, infections can be kept isolated from other infections communicating with the same IP address.

Potemkin

- Prototype honeyfarm
- Derived from pre-release version of Xen 3.0, running Debian GNU/Linux 3.1.
- Dynamically creates a new virtual machine, using *flash cloning* and *delta virtualization*, upon receipt of communication on a simulated network interface.
- Dynamically binds physical resources only when needed.
- Network gateway can redirect outbound communication to another virtual machine dynamically created to act as the destination of the malicious code communication for increased fidelity.

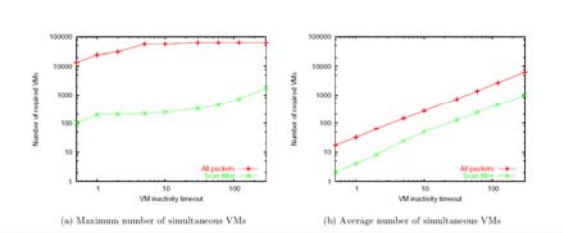
Potemkin (Details)

- Servers: Dell 1750 and SC1425 servers.
- VMM: Modified pre-release version of Xen 3.0.
- Operating System: Debian GNU/Linux 3.1.
- Application: Apache Web Server
- Gateway Server: Based on Click 1.4.1 distribution running in kernel mode.
- Number of IP addresses: 64k, using GRE-tunneled /16 network.

Facet (Gateway Router)

- Gateway Router
  - Inbound traffic
    - Attracted by routing (visible to traceroute tool) and tunneling (increased risk of packet loss).
  - Outbound traffic
    - DNS translation lookups are allowed.
    - Uses *internal reflection* to contain potentially infected communication.
  - Resource allocation and detection
    - Dynamically programmable filter prevents redundant VMs from being created.
    - Allows infected VM to continue execution, or freezes VM state and moves to storage for later analysis, or decommissions uninfected VM.

Effects of Gateway filtering



(a) Maximum number of simultaneous VMs (b) Average number of simultaneous VMs

Facet (Virtual Machine Monitor)

- Virtual Machine Monitor
  - Reference Image Instantiation
    - A snapshot of a normally booted operating system with a loaded application is used as a reference image for future *flash cloning*. (Currently uses a memory-based filesystem, but already planning on incorporating Parallax for low overhead disk snapshots.
  - Flash Cloning
    - It takes about 521ms to clone a virtual machine from the reference image. Future incarnations hope to reduce overhead by reusing decommissioned VMs rather than tearing them down for a savings of almost 200ms.
  - Delta Virtualization
    - The number of concurrent virtual machines are currently limited to 116 by Xen's heap. Extrapolation reveals a potential limit of 1500 VMs when using 2GB of RAM like in the test Potemkin servers.

Conclusion

- Through the use of late binding of resources, aggressive memory sharing, and exploiting the properties of virtual machines, the architecture in this paper overcome some of the limitations of conventional honeyfarms.

Vigilante: End-to-End Containment of Internet Worms

Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, Paul Barham

Slides by Mahesh Balakrishnan

Worms

- The Morris Worm… 1988

- Zotob – Aug 2005, Windows Plug-and-Play
- Sasser – May 2004, Windows LSASS
- Mydoom – Jan 2004, email
- SoBig, Aug 2003, email
- Blaster – Aug 2003, Windows RPC
- Slammer – Jan 2003, SQL
- Nimda – Sep 2001, email + IE
- CodeRed – July 2001, IIS

The Anatomy of a Worm

- Replicate, replicate, replicate… exponential growth
- Exploit Vulnerability in Network-facing Software
- Worm Defense involves
  - Detection – *of what?*
  - Response

Existing Work

- Network Level Approaches …
  - Polymorphic? Slow worms?
- Host-based Approaches: Instrument code extensively. Slow (?)…
- Do it elsewhere: Honeypots… honeyfarms
- Worm Defense now has three components: Detection, Propagation, Response

Vigilante

- Automates worm defense
- 'Collaborative Infrastructure' to detect worms
- Required: Negligible rate of false positives
- Network-level approaches do not have access to vulnerability specifics

Solution Overview

- Run heavily instrumented versions of software on honeypot or detector machines
- Broadcast exploit descriptions to regular machines
- Generate message filters at regular machines to block worm traffic
- Requires separate detection infrastructure for each particular service: is this a problem?

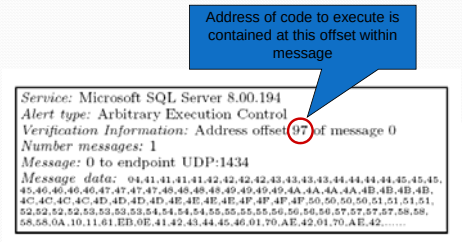
SCA: Self-Certifying Alert

- Allows exploits to be described, shipped, and *reproduced*
- Self-Certifying: to verify authenticity, just execute within sandbox
- Expressiveness: Concise or Inadequate?
  - Worms defined as 'exploiters of vulnerability' rather than 'generators of traffic'

Types of Vulnerabilities

- *Arbitrary Execution Control*: message contains address of code to execute
- *Arbitrary Code Execution*: message contains code to execute
- *Arbitrary Function Argument*: changing arguments to 'critical' functions. e.g exec
- Is this list comprehensive?
  - C/C++ based vulnerabilities...
  - what about email-based worms – SoBig, Mydoom, Nimda?

Example SCA: Slammer



Alert Generation

- Many existing approaches
- Non-executable pages: faster, does not catch function argument exploit
- Dynamic Data-flow Analysis: track *dirty* data
- Basic Idea: Do not allow incoming messages to execute or cause arbitrary execution

Alert Verification

- Hosts run same software with identical configuration within sandbox
- Insert call to *Verified* instead of:
  - Address in execution control alerts
  - Code in code execution alerts
- Insert a reference argument value instead of argument in arbitrary function argument alert

Alert Verification

- Verification is *fast, simple and generic,* and *has no false positives*
- Assumes that address/code/argument is supplied verbatim in messages
  - Works for C/C++ buffer overflows, but what about more complex interactions within the service?
- Assumes that message replay is sufficient for exploit reproduction
  - Scheduling policies, etc?
  - Randomization?

Alert Distribution

- Flooding over secure Pastry overlay
- What about DOS?
  - Don't forward already seen or blocked SCAs
  - Forward only after Verification
  - Rate-limit SCAs from each neighbor
- *Secure* Pastry?
  - Infrastructural solution… super-peers

Distribution – out-crawling the worm

- Worm has scanning inefficiencies; Slammer doubled every 8.5 seconds.
- The myth of the anti-worm…
- We have perfect topology: around the world in seconds! What if the worm discovers our topology?
- *Infrastructural support required – worm-resistant super-peers!*
- Is this sterile medium of transport deployable?

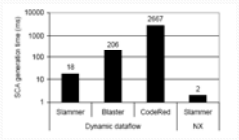
Local Response

- Verify SCA
- Data and Control Flow Analysis
- Generate *filters* – conjunctions of conditions on single messages
- Two levels : general filter with false positives + specific filter with no false positives

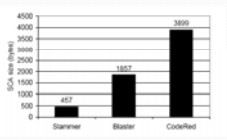
Evaluation

- Target Worms
  - *Slammer* – 75,000 SQL servers, 8.5 seconds to double
    - Execution Control Alert
  - *Code Red* – 360,000 IIS servers, 37 minutes to double
    - Code Execution Alert
  - *Blaster* – 500,000 Windows boxes, doubling time similar to CodeRed
    - Execution Control Alert

Evaluation: Alert Generation



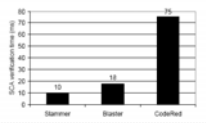
SCA Generation Time



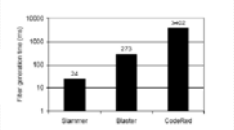
SCA Sizes

Evaluation: Alert Verification

- Verification is fast - Sandbox VM constantly running …



Verification Time



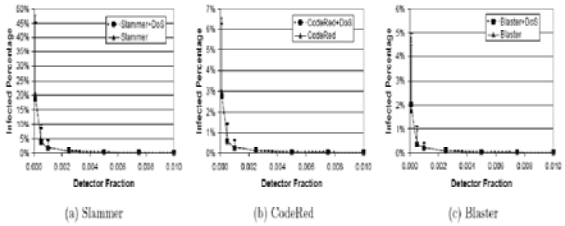
Filter Generation

Simulation Setup

- Transit Stub Topology: 500,000 hosts
- Worm propagation using epidemic model
- 1000 super-peers that are *neither detectors nor susceptible*!
- Includes modeling of worm-induced congestion

Number of Detectors…

- ~0.001 detectors sufficient… 500 nodes



(a) Slammer (b) CodeRed (c) Blaster

Real Numbers

- 5-host network: 1 detector + 3 super-peers + 1 susceptible *on a LAN*
- Time between detector probe to SCA verification on susceptible host:
  - Slammer: 79 ms
  - Blaster: 305 ms
  - CodeRed: 3044 ms

The Worm Turns

- Outwitting Vigilante
  - One interesting idea, courtesy Saikat: Detect instrumented honeypots via timing differences. Does this work?
  - Layered exploits: one for the super-peer topology, another for the hosts
  - Ideas?

Conclusion

- *End-to-end* solution for stopping worms
- Pros:
  - No false positives
  - Per-vulnerability reaction, not per-variant: handles polymorphism, slow worms
- Concerns:
  - Limited to C/C++ buffer overflow worms
  - Infrastructural solution – scope of deployment?