N-Variant Systems

Slides extracted from talk by David Evans
(provenance in footer)
Inevitability of Failure

• Despite all the best efforts to build secure software, we will still fail (or have to run programs that failed)

• Run programs in ways that make it harder to exploit vulnerabilities
Security Through Diversity

• Today’s Computing Monoculture
  – Exploit can compromise billions of machines since they are all running the same software

• Biological Diversity
  – All successful species use very expensive mechanism (sex) to maintain diversity

• Computer security research: [Cohen 92], [Forrest+ 97], [Cowan+ 2003], [Barrantes+ 2003], [Kc+ 2003], [Bhatkar+2003], [Just+ 2004], [Bhatkar, Sekar, DuVarney 2005]
Instruction Set Randomization

[Barrantes+, CCS 03] [Kc+, CCS 03]

- Code injection attacks depend on knowing the victim machine’s instruction set
- Defuse them all by making instruction sets different and secret
  - It’s expensive to design new ISAs and build new microprocessors
Automating ISR

Original Executable → Randomizer → Randomized Executable

Secret Key

Original Code

Processor

Derandomizer
ISR Defuses Attacks

Original Executable → Randomizer → Randomized Executable

Secret Key

Malicious Injected Code

Broken Malicious Code Processor

Derandomizer
## ISR Designs

<table>
<thead>
<tr>
<th></th>
<th>Columbia [Kc 03]</th>
<th>RISE [Barrantes 03]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Randomization Function</td>
<td>XOR or 32-bit transposition</td>
<td>XOR</td>
</tr>
<tr>
<td>Key Size</td>
<td>32 bits (same key used for all locations)</td>
<td>program length (each location XORed with different byte)</td>
</tr>
<tr>
<td>Transformation Time</td>
<td>Compile Time</td>
<td>Load Time</td>
</tr>
<tr>
<td>Derandomization</td>
<td>Hardware</td>
<td>Software (Valgrind)</td>
</tr>
</tbody>
</table>
How secure is ISR?

Slows down an attack about 6 minutes!

Can use probe injections to incrementally guess the key byte-by-byte (under the right conditions)

Better Solution

- Avoid secrets!
  - Keeping them is hard
  - They can be broken or stolen
- Prove security properties without relying on assumptions about secrets or probabilistic arguments
Polygraphing Processes: N-Variant Systems for Secretless Security

work with Ben Cox, Jack Davidson, Adrian Filipi, Jason Hiser, Wei Hu, John Knight, Anh Nguyen-Tuong, Jonathan Rowanhill
2-Variant System

Input (Possibly Malicious)

Polygrapher

Server Variant 0

Monitor

Output

Server Variant 1
N-Version Programming

- Multiple teams of programmers implement same spec
- Voter compares results and selects most common
- No guarantees: teams may make same mistake

N-Variant Systems

- Transformer automatically produces diverse variants
- Monitor compares results and detects attack
- Guarantees: variants behave differently on particular input classes

[Avizienis & Chen, 1977]
N-Variant System Framework

- **Polygrapher**
  - Replicates input to all variants

- **Variants**
  - \(N\) processes that implement the same service
  - Vary property you hope attack depends on: memory locations, instruction set, system call numbers, scheduler, calling convention, ...

- **Monitor**
  - Observes variants
  - Delays external effects until all variants agree
  - Initiates recovery if variants diverge
Variants Requirements

• *Detection* Property
  Any attack that compromises Variant 0 causes Variant 1 to “crash” (behave in a way that is noticeably different to the monitor)

• *Normal Equivalence* Property
  Under normal inputs, the variants stay in equivalent states:

  \[ \mathcal{A}_0(S_0) \equiv \mathcal{A}_1(S_1) \]

  Actual states are different, but abstract states are equivalent
Memory Partitioning

- **Variation**
  - Variant 0: addresses all start with 0
  - Variant 1: addresses all start with 1

- **Normal Equivalence**
  - Map addresses to same address space

- **Detection Property**
  - Any *absolute* load/store is invalid on one of the variants
Instruction Set Tagging

• Variation: add an extra bit to all opcodes
  – Variation 0: tag bit is a 0
  – Variation 1: tag bit is a 1
  – At run-time check bit and remove it
    • Low-overhead software dynamic translation using Strata
      [Scott, et al., CGO 2003]

• Normal Equivalence: Remove the tag bits

• Detection Property
  – Any (tagged) opcode is invalid on one variant
  – Injected code (identical on both) cannot run on both
Implementing N-Variant Systems

• Competing goals:
  – Isolation: of monitor, polygrapher, variants
  – Synchronization: variants must maintain normal equivalence (nondeterminism)
  – Performance: latency (wait for all variants to finish) and throughput (increased load)

• Two implementations:
  – Divert Sockets (prioritizes isolation over others)
    • Maintaining normal equivalence is too difficult
  – Kernel modification (sacrifices isolation for others)
Kernel Implementation
[Ben Cox]

- Modify process table to record variants
- Create new fork routine to launch variants
- Intercept system calls:
  - 289 calls in Linux
  - Check parameters are the same for all variants
  - Make call once
- Low overhead, lack of isolation
Wrapping System Calls

- I/O system calls (process interacts with external state) (e.g., open, read, write)
  - Make call once, send same result to all variants

- Process system calls (e.g., fork, execve, wait)
  - Make call once per variant, adjusted accordingly

- Special:
  - mmap: each variant maps segment into own address space, only allow MAP_ANONYMOUS (shared segment not mapped to a file) and MAP_PRIVATE (writes do not go back to file)
ssize_t sys_read(int fd, const void *buf, size_t count) {
    if (hasSibling (current)) {
        record that this variant process entered call
        if (!inSystemCall (current->sibling)) { // this variant is first
            save parameters
            sleep // sibling will wake us up
            get result and copy *buf data back into address space
            return result;
        } else if (currentSystemCall (current->sibling) == SYS_READ) {
            // I’m second variant, sibling is waiting
            if (parameters match) { // match depends on variation
                perform system call
                save result and data in kernel buffer
                wake up sibling
                return result;
            } else {
                DIVERGENCE ERROR! }
        } else { // sibling is in a different system call!
            DIVERGENCE ERROR! }
    } else {
        DIVERGENCE ERROR! }
    }
}
# Overhead

Results for Apache running WebBench 5.0 benchmark

<table>
<thead>
<tr>
<th>Description</th>
<th>Unmodified Apache, unmodified kernel</th>
<th>2-variant system, address space partitioning</th>
<th>2-variant system, instruction tagging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput (MB/s)</td>
<td>6.46</td>
<td>5.57</td>
<td>4.01</td>
</tr>
<tr>
<td>Latency (ms)</td>
<td>9.06</td>
<td>10.52</td>
<td>14.84</td>
</tr>
</tbody>
</table>

14% decrease in throughput

68% increase in latency
Diversity depends on your adversary

Slide from my USENIX Security 2004 Talk, *What Biology Can (and Can't) Teach us about Security*
N-Variant Summary

• Producing artificial diversity is easy
  – Defeats undetermined adversaries
• Keeping secrets is hard
• N-variant systems framework offers provable defense without secrets
  – Effectiveness depends on whether variations vary things that matter to attack
Questions?

**N-Variant Systems:**
http://www.cs.virginia.edu/nvariant
Ben Cox, Jack Davidson, Adrian Filipi, Jason Hiser, Wei Hu, John Knight, Anh Nguyen-Tuong, Jonathan Rowanhill

**Perracotta:**
http://www.cs.virginia.edu/perracotta
Jinlin Yang
Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das (Microsoft Research)

**Funding:** National Science Foundation