The N-Variant Systems Framework
Polygraphing Processes for Secretless Security

David Evans
http://www.cs.virginia.edu/evans
University of Virginia
Computer Science

University of Texas at San Antonio
4 October 2005

Security Through Diversity

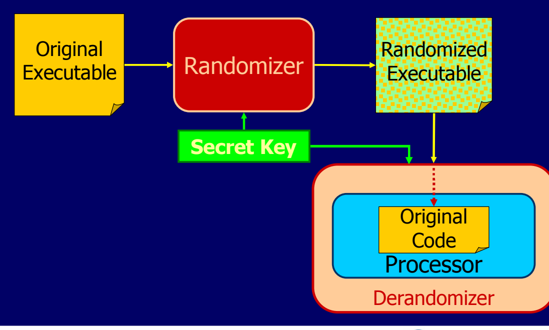
- Today's Computing Monoculture
  - Exploit can compromise billions of machines since they are all running the same software
- Biological Diversity
  - All successful species use very expensive mechanism (sex) to maintain diversity
- Computer security research: [Cohen 92], [Forrest+ 97], [Cowan+ 2003], [Barrantes+ 2003], [Kc+ 2003], [Bhatkar+2003], [Just+ 2004], [Bhatkar, Sekar, DuVarney 2005]

Instruction Set Randomization
[Barrantes+, CCS 03] [Kc+, CCS 03]

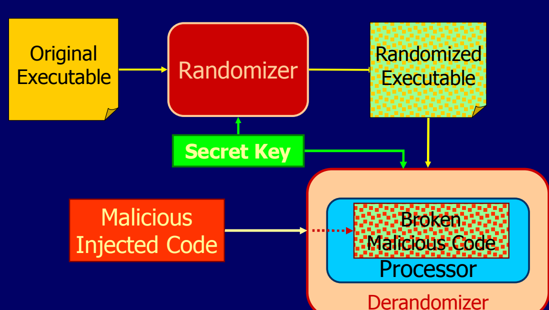
- Code injection attacks depend on knowing the victim machine's instruction set
- Defuse them all by making instruction sets different and secret
  - Its expensive to design new ISAs and build new microprocessors

---

Automating ISR



ISR Defuses Attacks



ISR Designs

	Columbia [Kc 03]	RISE [Barrantes 03]
Randomization Function	XOR or 32-bit transposition	XOR
Key Size	32 bits (same key used for all locations)	program length (each location XORed with different byte)
Transformation Time	Compile Time	Load Time
Derandomization	Hardware	Software (Valgrind)

How secure is ISR?

Slows down an attack about 6 minutes!

Under the right circumstances…

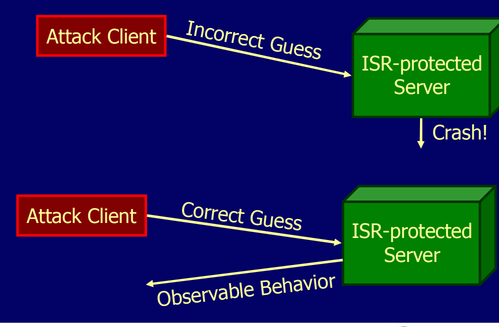
Where's the FEEB? Effectiveness of Instruction Set Randomization

In *USENIX Security Symposium*, August 2005.
Ana Nora Sovarel, David Evans and Nathanael Paul.

Memory Randomization Attack

- Brute force attack on memory address space randomization (Shacham et. al. [CCS 2004]): 24-bit effective key space
- Can a similar attack work against ISR?
  - Larger key space: must attack in fragments
  - Need to tell if partial guess is correct

ISR Attack



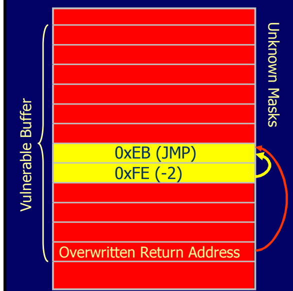
Server Requirements

- Vulnerable: buffer overflow is fine
- Able to make repeated guesses
  - No rerandomization after crash
  - Likely if server forks requests (Apache)
- Observable: notice server crashes
- Cryptanalyzable
  - Learn key from one ciphertext-plaintext pair
  - Easy with XOR

Two Attack Ideas

- RET (0xC3): return from procedure
  - 1-byte instruction: up to 256 guesses
  - Returns, leaves stack inconsistent
    - Only works if server does something observable before crashing
- JMP -2 (0xEBFE): jump offset -2
  - 2-byte instruction: up to 2^{16} guesses
  - Produces infinite loop
- Incorrect guess usually crashes server

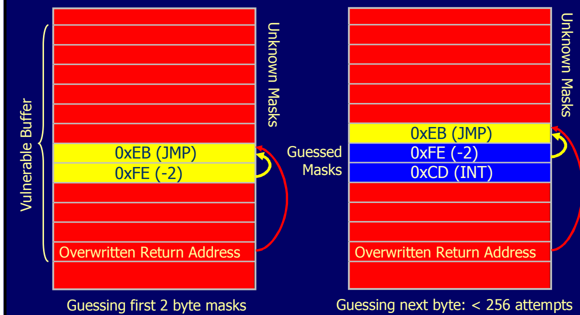
Jump Attack



2^{16} possible guesses for 2-byte instruction

Correct guess produces infinite loop

Incremental Jump Attack



Guess Outcomes

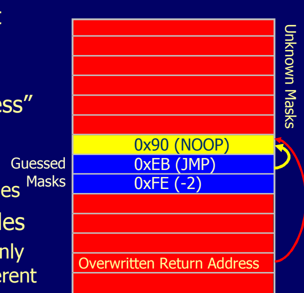
	Observe "Correct" Behavior	Observe "Incorrect" Behavior
Correct Guess	Success	False Negative
Incorrect Guess	False Positive	Progress

False Positives

- Injected bytes produce an infinite loop:
  - JMP -4
  - JNZ -2
- Injected bytes are "harmless", later executed instruction causes infinite loop
- Injected guess causes crash, but timeout expires before remote attacker observes

False Positives – Good News

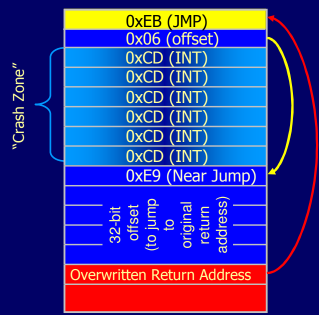
- Can distinguish correct mask using other instructions
- Try injecting a "harmless" one-byte instruction
  - Correct: get loop
  - Incorrect: *usually* crashes
- Difficulty: dense opcodes
  - No pair that differs in only last bit are reliably different in harmfullness



False Positives – Better News

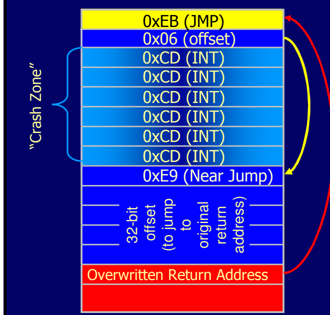
- False positives are not random
  - Conditional jump instructions
  - Opcodes **0111**0000-**0111**1111
- **All** are complementary pairs: 0111$xyza$ not taken ⇔ 0111$xyz\bar{a}$ is!
- 32 guesses always find an infinite loop
- About 8 additional guesses to determine correct mask

Extended Attack



- Near jump to return location
  - Execution continues normally
  - No infinite loops
- 0xCD 0xCD is interrupt instruction guaranteed to crash

Expected Attempts



~ 15½ to find first jumping instruction

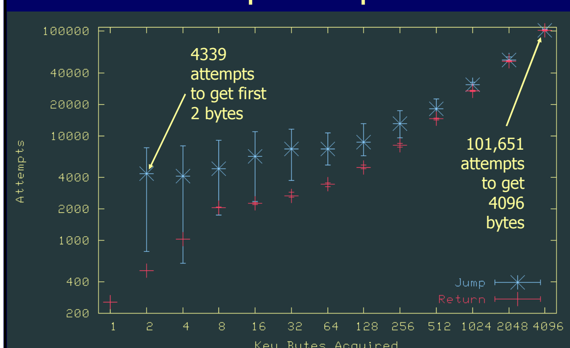
\+ ~ 8 to determine correct mask

23½ expected attempts per byte

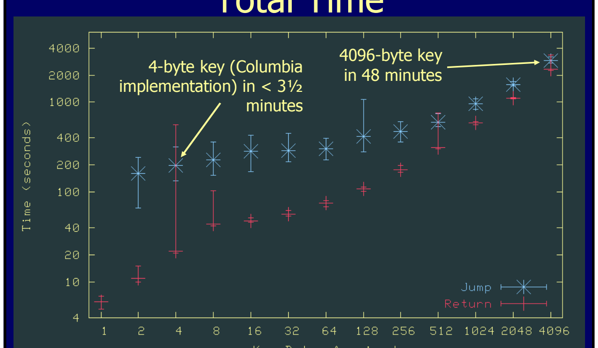
Experiments

- Implemented attack against constructed vulnerable server protected with RISE [Barrantes et. al, 2003]
  - Memory space randomization works!
    - Turned of Fedora's address space randomization
  - Needed to modify RISE
    - Ensure forked processes use same randomization key (other proposed ISR implementations wouldn't need this)
- Obtain correct key over 95% of the time
  - Sometimes can't because unable to inject NULLs

Attempts Required



Total Time



Attacker: "Is this good enough?" Defender: "Is this bad enough?"

How many key bytes needed?

- Inject malcode in one ISR-protected host
  - Sapphire worm = 376 bytes
- Create a worm that spreads on a network of ISR-protected servers
  - Space for FEEB attack code: 34,723 bytes
  - Need to crash server ~800K times

Maybe less…?

- VMWare: 3,530,821 bytes
- Java VM: 135,328 bytes
- Minsky's UTM: 7 states, 4 colors
- **MicroVM: 100 bytes**

Entire MicroVM Code

```
  push dword ebp   mov ebp, WORM_ADDRESS + WORM_REG_OFFSET
  pop dword [ebp + WORM_DATA_OFFSET]
  xor eax, eax        ; WormIP = 0 (load from ebp + eax)
read_more_worm: ; read NUM_BYTES at a time until worm is done
  cld      xor ecx, ecx         mov byte cl, NUM_BYTES
  mov dword esi, WORM_ADDRESS     ; get saved WormIP
  add dword esi, eax            mov edi, begin_worm_exec
  rep movsb                     ; copies next Worm block into execution buffer
  add eax, NUM_BYTES            ; change WormIP
  pushad                        ; save register vals
  mov edi, dword [ebp]          ; restore worm registers
  mov esi, dword [ebp + ESI_OFFSET]     mov ebx, dword [ebp + EBX_OFFSET]
  mov edx, dword [ebp + EDX_OFFSET]     mov ecx, dword [ebp + ECX_OFFSET]
  mov eax, dword [ebp + EAX_OFFSET]
begin_worm_exec:                ; this is the worm execution buffer
   nop nop nop nop nop nop nop nop nop nop nop nop
   nop nop nop nop nop nop nop nop nop nop nop nop
   mov [ebp], edi     ; save worm registers
   mov [ebp + ESI_OFFSET], esi        mov [ebp + EBX_OFFSET], ebx
   mov [ebp + EDX_OFFSET], edx        mov [ebp + ECX_OFFSET], ecx
   mov [ebp + EAX_OFFSET], eax
   popad              ; restore microVM register vals
   jmp read_more_worm
```


MicroVM

Learned Key Bytes

- save worm address in **ebp**
- move stack frame pointer
- WormIP ← 0
- copy worm code into buffer
- update WormIP
- save MicroVM registers
- load worm registers
- 22-byte worm execution buffer
- save worm registers
- load MicroVM registers
- jmp to read next block

- saved registers
- worm code
- host key masks
- guessed (target) masks
- other worm data

  76 bytes of code
\+ 22 bytes for execution
\+ 2 bytes to avoid NULL
= 100 bytes is enough
  > 99% of the time

Worm code must be coded in blocks that fit into execution buffer (pad with noops so instructions do not cross block boundaries)

Making Jumps

- Within a block - short relative jump is fine
- Between worm blocks
  - From end of block, to beginning of block
  - Update the WormIP stored on the stack
  - Code conditional jump, **JZ target** in worm as:

    JNZ +5 *; if opposite condition, skip*
    MOV [ebp + WORMIP_OFFSET] **target**

Deploying a Worm

- Learn 100 key bytes to inject MicroVM
  - Median time: 311 seconds, 8422 attempts
  - Fast enough for a worm to spread effectively
- Inject pre-encrypted worm code
  - XORed with the known key at location
  - Insert NOOPs when necessary to avoid NULLs
- Inject key bytes
  - Needed to propagate worm

Preventing Attack: Break Attack Requirements

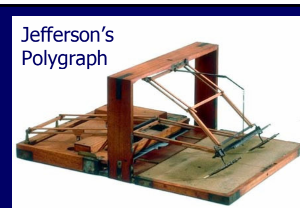
- Vulnerable: eliminate vulnerabilities
  - Rewrite all your code in a type safe language
- Able to make repeated guesses
  - Rerandomize after crash
- Observable: notice server crashes
  - Maintain client socket after crash?
- Cryptanalyzable
  - Use a strong cipher like AES instead of XOR

Better Solution

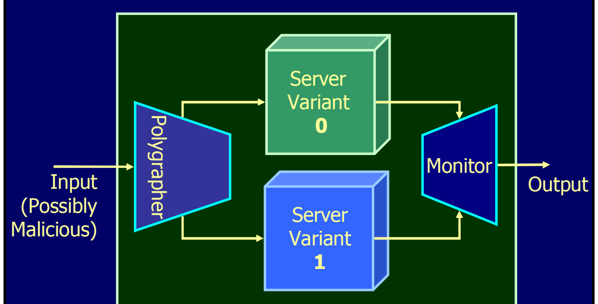
- Avoid secrets!
  - Keeping them is hard
  - They can be broken or stolen
- Prove security properties without relying on assumptions about secrets or probabilistic arguments

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



N-Version Programming

[Avizienis & Chen, 1977]

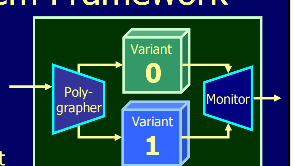
- Multiple teams of programmers implement same spec
- Voter compares results and selects most common
- No guarantees: teams may make same mistake

N-Variant Systems

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

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, file names, 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 Partitioning

JMP
CALL
JO
JNO
JB
JNB
JZ
JNZ
…

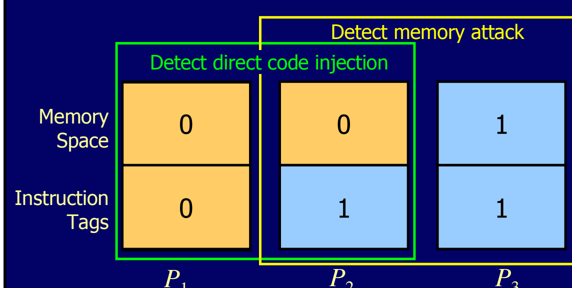
Variant A Variant B

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

Composing Variations

Must preserve normal equivalence property



Indirect Code Injection Attack

- Inject bytes into data buffer
- Original code transforms contents of that buffer (XORing every byte with a different value on P_1 and P_2)
- Relative jump to execute injected, transformed code
- What went wrong?

Normal Equivalence property violated: need to know that data manipulated differently is never used as code

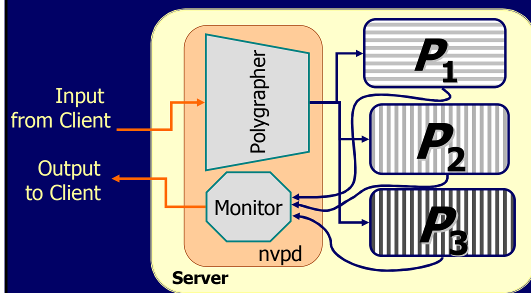
Implementing N-Variant Systems

- 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)
  - Kernel modification (sacrifices isolation for others)

Implementation: Divert Sockets [Adrian Filipi]

- Process intercepts traffic (nvpd)
- Uses divert sockets to send copies to isolated variants (can be on different machines)
- Waits until all variants respond to request before returning to client
- Adjusts TCP sequence numbers to each variant appears to have normal connection

3-Variant System



Implementation: Kernel Modification [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)

System Call Wrapper Example

```
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) {
          // this variant is second, 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! } }
   ...
}
```


Current Status

- Can run apache with address and instruction tag variations
  - Thwarts any attack that depends on referencing an absolute address or executing injected code
- Open problems
  - Non-determinism, persistent state
  - Establishing normal equivalence
- Cost
  - nvpd implementation, https, 4x machines: Latency x 2.3
  - Kernel modification (hopefully better, no numbers yet)

Diversity depends on your perspective

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

www.cs.virginia.edu/nvariant 49 Computer Science at the University of Virginia

Summary

- Producing artificial diversity is easy
  - Defeats undetermined adversaries
- Keeping secrets is hard
  - Remote attacker can break ISR-protected server in < 6 minutes
- N-variant systems framework offers provable (but expensive) defense
  - Effectiveness depends on whether variations vary things that matter to attack

www.cs.virginia.edu/nvariant 50 Computer Science at the University of Virginia