Countering Code-Injection Attacks
With Instruction-Set Randomization

Gaurav S. Kc, Angelos D. Keromytis
Columbia University

Vassilis Prevelakis
Drexel University
Overview of Technique

• Protect from code-injection attacks
  – create unique execution environment
    (instruction set)
  – invalidate attack vector
  – equally applicable for interpreted environments
    and native machine code
    (*prototype designed for porting to hardware*)
Outline

• Attack Techniques & Defense Mechanisms
• Instruction-Set Randomization (ISR)
• Using ISR to protect Linux processes in the Bochs $x86$ emulator
• Conclusions and Future Work
Attack Techniques

• Application-level attacks exploit flaws
  – Causes:
    • Software bugs
    • Poor programming practices
    • Language features
  – Exploits:
    • Buffer overflows, Format-string vulnerabilities
    • Code-injection, Process subversion
    • SQL / shell injection attacks
Defense Mechanisms

• *Safer* languages and libraries: *Java, Cyclone, Libsafe, strl*

• Prevent and detect buffer overflows
  – Static code analyses: *MOPS, MetaCompilation*
  – Runtime stack protection: *StackGuard, ProPolice, .NET /GS*

• Sandboxing (profiling, monitoring)
  – Application-level sandboxes: *Janus, Consh, ptrace, /proc*
  – Kernel-based system-call interception: *Tron, SubDomain*
  – Virtual environments: *VMWare, UML, Program shepherding, chroot*

• Non-executable data areas
  – user stack/heap areas: *PaX Team, SolarDesigner*
Defense Mechanisms: problems

• Shortcomings of individual approaches
  – Languages/libraries:
    • Stuck with C for systems, binary legacy applications
  – Prevent/detect overflows
    • Bypass overflow-detection logic in stack
  – Application-level Sandboxing
    • Overhead on system calls due to policy-based decision making
  – Non-executable data areas
    • Protect only specific areas

• Best-effort ideology: grand unified scheme for protection, combining multiple techniques

• New proposed technique:
  – *Instruction-set randomization*: all injected code is disabled
  – Applicable across the board:
    • Handle buffer overflow and SQL injection
Instruction sets of Attack code

- Match language / instruction set
  - SQL injection attacks
  - Embedded Perl code
  - x86 machine code

Typical x86 shellcode

```
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
"\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
"\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xDc\xff\xff\xff/bin/sh"
```

x86 shellcode demystified

1. Prepare parameters
   - jmp IP + 1f
   - pop %esi
   - mov %esi, 08(%esi)
   - xor %eax, %eax
   - mov %al, 07(%esi)
   - mov %eax, 0c(%esi)
   - mov 0b, %al
   - mov %esi, %ebx
   - lea 08(%esi), %ecx
   - lea 0c(%esi), %edx
   - int $0x80
   - xor %ebx, %ebx
   - mov %ebx, %eax
   - inc %eax
   - int $0x80
   - call 0xdceffffff

2. execve
ISR: per-process instruction-set

- #1 reason for ISR: invalidate injected code
- Perl prototype: instruction-set randomization
  - randomization of keywords, operators and function calls
  - interpreter appends 9-digit “tag” to lexer tokens when loading
  - parser rejects untagged code, e.g. injected Perl code

```perl
foreach $k (sort keys %$tre) {
    $v = $tre->{$k};
    die "duplicate key $k\n"
        if defined $list{$k};
    push @list, @{$list{$k} };
}
```
```perl
foreach123456789 $k (sort123456789 keys %$tre) {
    $v =1234567889 $tre->{$k};
    die123456789 "duplicate key $k\n"
        if123456789 defined123456789 $list{$k};
    push123456789 @list, @{$list{$k} };  
}
ISR: per-process instruction-set

- **ISR x86**: proof-of-concept
  Randomized code segments in programs
- **Use objcopy for randomizing** program image
  - Bit re-ordering within n-bit blocks (n! possibilities)
    
    \[
    \begin{align*}
    0x89d8 : & \quad 1000 \ 1001 \ 1101 \ 1000 : \text{mov} \ %\text{ebx}, \ %\text{eax} \\
    0x40fc : & \quad 0100 \ 0000 \ 1111 \ 1100 : \text{inc} \ %\text{eax}
    \end{align*}
    \]
  - n-bit XOR mask (\(2^n\) possibilities)
    
    \[
    \begin{align*}
    0x89d8 \ ^{\oplus} \ 0xc924 & \quad 0x40fc
    \end{align*}
    \]
- **Processor reverses** randomization when executing instructions
  - Fetch - decode - execute
  - Fetch - **de-randomize** - decode - execute
Prototype: instruction-set randomization on *modified* x86 hardware

- ISR-aware objcopy(1)
  - Randomize executable content

- ISR-aware x86 emulator
  - De-randomize, execute instructions

- ISR-aware Linux kernel
  - Intermediary between *randomized* processes and *de-randomizing* hardware
ISR-aware objcopy(1)

- **objcopy** – copy and translate object files
  *Executable and Linking Format* (ELF)

- New ELF section to store key
- Using the key, *randomize* instruction blocks in the code sections in *statically-compiled executables*

```c
static void copy_section(...) {  
  if (issection->flags & (SEC_LOAD|SEC_CODE))  
    // randomize-this-section-before-copy
}
```
ISR-aware Bochs \texttt{x86} emulator

- Emulator for Intel \texttt{x86} CPU, common I/O devices, and a custom BIOS
  
  \url{http://bochs.sourceforge.net}

- \texttt{x86} extensions for hardware support:
  - new 16-bit register (\texttt{gav}) to store de-randomizing key
  - new instruction (\texttt{gavl}) for loading key into register
  - In user-mode execution
    \texttt{fetchDecode} loop \rightarrow
    
    \{ fetch, \textit{de-randomize}, decode \} loop
ISR-aware Linux kernel

- **Process control-block (PCB)** for storing process-specific key
  - loader reads key from file and stores in PCB
  - scheduler loads key into special-purpose register \( gav \) from PCB using new x86 instruction \( gavl \) when switching process
  - key protected from illegal access and malicious modifications
Summary: ISR for x86

- **SOURCE CODE**
- **MACHINE EXECUTABLE FILE**

**gcc -static**

**RANDOMIZED EXECUTABLE FILE**

- **key**
- **randomize via objcopy**

**Modified x86**

**fetch**

| 0101 1010 |
| 1101 0010 |

**decode**

| 1001 1010 |
| 1101 0010 |

**de-randomize**

| 1001 1010 0001 1011 |
x86 shellcode – *de-randomized*

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xeb 1f</td>
<td>jmp IP + 1f</td>
<td>: jmp IP + 1f</td>
</tr>
<tr>
<td>0x5e</td>
<td>pop %esi</td>
<td>: pop %esi</td>
</tr>
<tr>
<td>0x89 76 08</td>
<td>mov %esi, 08(%esi)</td>
<td>: mov %esi, 08(%esi)</td>
</tr>
<tr>
<td>0x31 c0</td>
<td>xor %eax, %eax</td>
<td>: xor %eax, %eax</td>
</tr>
<tr>
<td>0x88 46 07</td>
<td>mov %al, 07(%esi)</td>
<td>: mov %al, 07(%esi)</td>
</tr>
<tr>
<td>0x89 46 0c</td>
<td>mov %eax, 0c(%esi)</td>
<td>: mov %eax, 0c(%esi)</td>
</tr>
<tr>
<td>0xb0 0b</td>
<td>mov 0b, %al</td>
<td>: mov 0b, %al</td>
</tr>
<tr>
<td>0x89 f3</td>
<td>mov %esi, %ebx</td>
<td>: mov %esi, %ebx</td>
</tr>
<tr>
<td>0x8d 4e 08</td>
<td>lea 08(%esi), %ecx</td>
<td>: lea 08(%esi), %ecx</td>
</tr>
<tr>
<td>0x8d 56 0c</td>
<td>lea 0c(%esi), %edx</td>
<td>: lea 0c(%esi), %edx</td>
</tr>
<tr>
<td>0xcd 80</td>
<td>int $0x80</td>
<td>: int $0x80</td>
</tr>
<tr>
<td>0x31 db</td>
<td>xor %ebx, %ebx</td>
<td>: xor %ebx, %ebx</td>
</tr>
<tr>
<td>0x89 d8</td>
<td>mov %ebx, %eax</td>
<td>: mov %ebx, %eax</td>
</tr>
<tr>
<td>0x40</td>
<td>inc %eax</td>
<td>: inc %eax</td>
</tr>
<tr>
<td>0x89 80</td>
<td>int $0x80</td>
<td>: int $0x80</td>
</tr>
<tr>
<td>0xe8 dcffffff</td>
<td>call 0xdcffffff</td>
<td>: call 0xdcffffff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xcd d6</td>
<td>int $0xd7</td>
<td>: int $0xd7</td>
</tr>
<tr>
<td>0x24 c9</td>
<td>and $0xc9,%al</td>
<td>: and $0xc9,%al</td>
</tr>
<tr>
<td>0x24 97</td>
<td>and $0x97,%al</td>
<td>: and $0x97,%al</td>
</tr>
<tr>
<td>0xad</td>
<td>lods %ds:(%esi),%eax</td>
<td>: lods %ds:(%esi),%eax</td>
</tr>
<tr>
<td>0xbf 2c f8 e4 41</td>
<td>mov $0x41e4f82c,%edi</td>
<td>: mov $0x41e4f82c,%edi</td>
</tr>
<tr>
<td>0x62 ce</td>
<td>bound %ecx,%esi</td>
<td>: bound %ecx,%esi</td>
</tr>
<tr>
<td>0xad</td>
<td>lods %ds:(%esi),%eax</td>
<td>: lods %ds:(%esi),%eax</td>
</tr>
<tr>
<td>0x8f 28</td>
<td>popl (%eax)</td>
<td>: popl (%eax)</td>
</tr>
<tr>
<td>0x79 2f</td>
<td>jns 4a &lt;foo+0x4a&gt;</td>
<td>: jns 4a &lt;foo+0x4a&gt;</td>
</tr>
<tr>
<td>0x40</td>
<td>inc %eax</td>
<td>: inc %eax</td>
</tr>
<tr>
<td>0x79 2f</td>
<td>jns 4a &lt;foo+0x4a&gt;</td>
<td>: jns 4a &lt;foo+0x4a&gt;</td>
</tr>
<tr>
<td>0x40</td>
<td>inc %eax</td>
<td>: inc %eax</td>
</tr>
<tr>
<td>0x44</td>
<td>inc %esp</td>
<td>: inc %esp</td>
</tr>
<tr>
<td>0x6a c1</td>
<td>mov 0xfffffffcl</td>
<td>: mov 0xfffffffcl</td>
</tr>
<tr>
<td>0xa9 9f 28 04 a4</td>
<td>test $0xa404289f,%eax</td>
<td>: test $0xa404289f,%eax</td>
</tr>
<tr>
<td>0xf8</td>
<td>dec %ecx</td>
<td>: dec %ecx</td>
</tr>
<tr>
<td>0xff 40</td>
<td>incl 0xffffffff(%eax)</td>
<td>: incl 0xffffffff(%eax)</td>
</tr>
<tr>
<td>0x89 e9</td>
<td>mov %ebp,%ecx</td>
<td>: mov %ebp,%ecx</td>
</tr>
<tr>
<td>0x40</td>
<td>dec %ecx</td>
<td>: dec %ecx</td>
</tr>
<tr>
<td>0x1e</td>
<td>int3</td>
<td>: int3</td>
</tr>
<tr>
<td>0xdb 36</td>
<td>(bad) (%esi)</td>
<td>: (bad) (%esi)</td>
</tr>
<tr>
<td>0xdb 59 b4</td>
<td>fistpl 0xffffffffb4(%ecx)</td>
<td>: fistpl 0xffffffffb4(%ecx)</td>
</tr>
</tbody>
</table>
emulation overhead

<table>
<thead>
<tr>
<th></th>
<th>ftp</th>
<th>sendmail</th>
<th>fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>bochs</td>
<td>39.0s</td>
<td>$\approx$ 28s</td>
<td>5.73s (93s)</td>
</tr>
<tr>
<td>linux</td>
<td>29.2s</td>
<td>$\approx$ 1.35s</td>
<td>0.322s</td>
</tr>
</tbody>
</table>

- Maximum computation overhead: $x10^2$
- Services: ipchains, sshd (lower latency than hi-speed network from Wyndham)
Related work

• Randomized instruction set emulation to disrupt binary code injection attacks
  Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic
  and Dino Dai Zovi. University of New Mexico

• Valgrind x86-x86 binary translator (emulator) to de-scramble instruction sequences scrambled by loader
  – Attach Valgrind emulator to each randomized process
    • Using our approach, we can run entire OS in Bochs
Limitations & Future Work

• Disadvantages:
  – Precludes self-modifying code
  – Requires statically-built programs
  – Local users can determine key from file system

• Future considerations and extensions:
  – Dynamically re-randomize process (or specific modules)
  – Extend x86 prototype to other operating systems and processor combinations
  – Extend Perl prototype to other scripting languages: shell, TCL, php
  – Re-implement on programmable hardware, e.g. Transmeta

• Find thesis topic
Conclusions

- **Breach** happens!!
  Hard to prevent code injection.
- Defang an attack by disabling execution of injected code
  - Give control to attacker vs. impose self-DoS by killing process
  - Brute-forcing to attack system makes worms infeasible
  - No modifications to program source code
- General approach to prevent any type of code-injection attack
  - Can take advantage of special hardware
  - Applicable to scripting languages