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

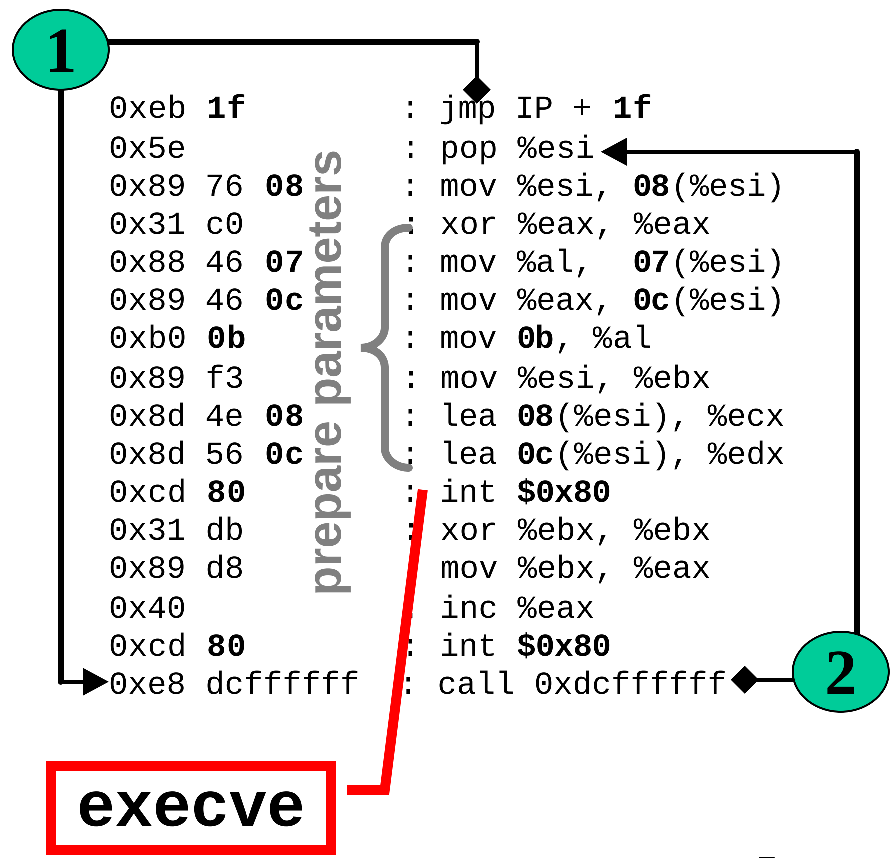
Instruction sets of Attack code

- 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*



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

```
foreach $k (sort keys %$tre) {
  $v = $tre->{$k};
  die “duplicate key $k\n”
      if defined $list{$k};
  push @list, @{ $list{$k} };
}
```

```
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)

```
0x89d8 : 1000 1001 1101 1000 : mov %ebx, %eax
0x40fc : 0100 0000 1111 1100 : inc %eax
```

  - n-bit XOR mask (2^n possibilities)

```
0x89d8 ^ 0xc924 → 0x40fc
```

- 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*

```
static void copy_section(...) {
    if (isection->flags & (SEC_LOAD|SEC_CODE))
          // randomize-this-section-before-copy
}
```

ISR-aware Bochs *x86* emulator

- Emulator for Intel *x86* CPU, common I/O devices, and a custom BIOS

  http*://bochs.sourceforge.net*

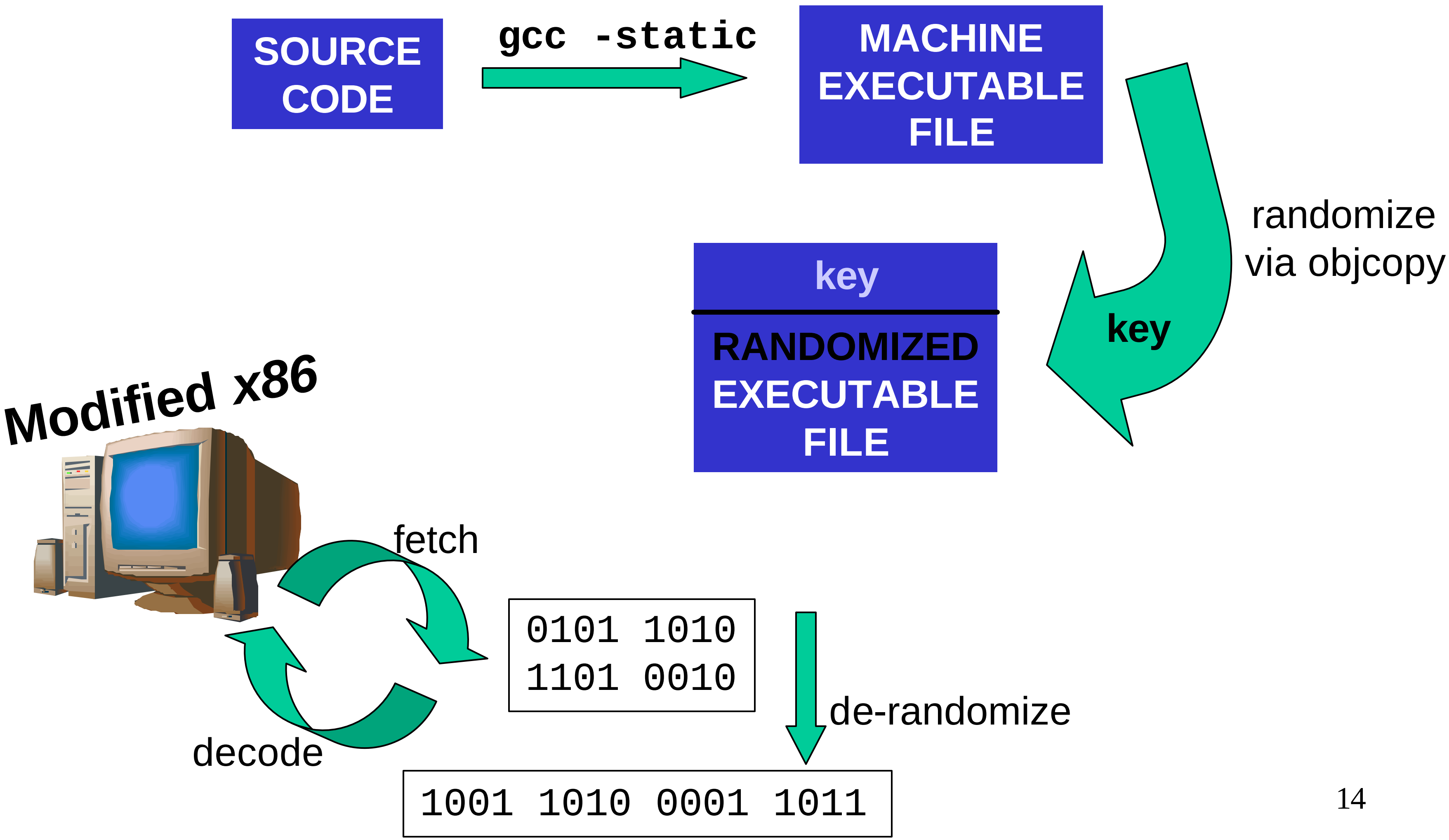
- *x86* extensions for hardware support:
  - new 16-bit register (`gav`) to store de-randomizing key
  - new instruction (`gavl`) for loading key into register
  - In user-mode execution `fetchDecode` loop →

    { fetch, *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*



x86 shellcode – *de-randomized*

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

```
0xcd d6               : int     $0xd7
0x24 c9               : and     $0xc9,%al
0x24 97               : and     $0x97,%al
0xad                  : lods    %ds:(%esi),%eax
0xbf 2c f8 e4 41      : mov     $0x41e4f82c,%edi
0x62 ce               : bound   %ecx,%esi
0xad                  : lods    %ds:(%esi),%[illegible]
0x8f 28               : popl    (%eax)
0x79 2f               : jns     4a <fo[illegible]
0x40                  : inc     %ea[illegible]
0xd7                  : xlat    %d[illegible]bx)
0x44                  : in[illegible] %esp
0x6a c1               : [illegible] $0xffffffc1
0xa9 9f 28 04 a4      : [illegible]est $0xa404289f,%eax
0xf8                  : [illegible]lc
0xff 40 [illegible]   : [illegible]ncl 0xfffffffc(%eax)
0x89 e9               : mov     %ebp,%ecx
0x4[illegible]        : dec     %ecx
[illegible]           : int3
[illegible]           : push    %ds
[illegible]db 36      : (bad)   (%esi)
[illegible]xdb 59 b4  : fistpl  0xffffffb4(%ecx)
```

Highly likely to crash

emulation overhead

	ftp	sendmail	fibonacci
bochs	39.0s	$\approx 28s$	5.73s (*93s*)
linux	29.2s	$\approx 1.35s$	0.322s

- Maximum computation overhead: x10^2
- Services: ipchains, sshd (*lower latency than <u>hi-speed</u> 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