Kernel level exception handling

Commentary by Joerg Pommnitz <joerg@raleigh.ibm.com>

When a process runs in kernel mode, it often has to access user mode memory whose address has been passed by an untrusted program. To protect itself the kernel has to verify this address.

In older versions of Linux this was done with the int verify_area(int type, const void * addr, unsigned long size) function (which has since been replaced by access_ok()).

This function verified that the memory area starting at address 'addr' and of size 'size' was accessible for the operation specified in type (read or write). To do this, verify_read had to look up the virtual memory area (vma) that contained the address addr. In the normal case (correctly working program), this test was successful. It only failed for a few buggy programs. In some kernel profiling tests, this normally unneeded verification used up a considerable amount of time.

To overcome this situation, Linus decided to let the virtual memory hardware present in every Linux-capable CPU handle this test.

How does this work?

Whenever the kernel tries to access an address that is currently not accessible, the CPU generates a page fault exception and calls the page fault handler:

```
void do_page_fault(struct pt_regs *regs, unsigned long error_code)
```

in arch/x86/mm/fault.c. The parameters on the stack are set up by the low level assembly glue in arch/x86/entry/entry_32.S. The parameter regs is a pointer to the saved registers on the stack, error_code contains a reason code for the exception.

do_page_fault first obtains the unaccessible address from the CPU control register CR2. If the address is within the virtual address space of the process, the fault probably occurred, because the page was not swapped in, write protected or something similar. However, we are interested in the other case: the address is not valid, there is no vma that contains this address. In this case, the kernel jumps to the bad_area label.

There it uses the address of the instruction that caused the exception (i.e. regs->eip) to find an address where the execution can continue (fixup). If this search is successful, the fault handler modifies the return address (again regs->eip) and returns. The execution will continue at the address in fixup.

Where does fixup point to?

Since we jump to the contents of fixup, fixup obviously points to executable code. This code is hidden inside the user access macros. I have picked the get_user macro defined in arch/x86/include/asm/uaccess.h as an example. The definition is somewhat hard to follow, so let's peek at the code generated by the preprocessor and the compiler. I selected the get_user call in drivers/char/sysrq.c for a detailed examination.

The original code in sysrq.c line 587:

```
get_user(c, buf);
```

The preprocessor output (edited to become somewhat readable):

```
(
  {
    long __gu_err = - 14 , __gu_val = 0;
    const __typeof__(*( (  buf ) )) *__gu_addr = ((buf));
    if (((((0 + current_set[0])->tss.segment) == 0x18 )   ||
       (((sizeof(*(buf))) <= 0xC0000000UL) &&
       ((unsigned long)(__gu_addr ) <= 0xC0000000UL - (sizeof(*(buf)))))))
      do {
        __gu_err  = 0;
        switch ((sizeof(*(buf)))) {
          case 1:
            __asm__ __volatile__(
              "1:      mov" "b" " %2,%" "b" "1\n"
              "2:\n"
              ".section .fixup,\"ax\"\n"
              "3:      movl %3,%0\n"
              "        xor" "b" " %" "b" "1,%" "b" "1\n"
              "        jmp 2b\n"
              ".section __ex_table,\"a\"\n"
              "        .align 4\n"
              "        .long 1b,3b\n"
              ".text"        : "=r"(__gu_err), "=q" (__gu_val): "m"((*(struct __large_struct *)
                            (   __gu_addr   )) ), "i"(- 14 ), "0"(  __gu_err  )) ;
              break;
          case 2:
            __asm__ __volatile__(
              "1:      mov" "w" " %2,%" "w" "1\n"
              "2:\n"
              ".section .fixup,\"ax\"\n"
```

```
                "3:      movl %3,%0\n"
                "        xor" "w" " %" "w" "1,%" "w" "1\n"
                "        jmp 2b\n"
                ".section __ex_table,\"a\"\n"
                "        .align 4\n"
                "        .long 1b,3b\n"
                ".text"        : "=r"(__gu_err), "=r" (__gu_val) : "m"((*(struct __large_struct *)
                              (   __gu_addr   )) ), "i"(- 14 ), "0"(  __gu_err  ));
                break;
            case 4:
              __asm__ __volatile__(
                "1:      mov" "l" " %2,%" "" "1\n"
                "2:\n"
                ".section .fixup,\"ax\"\n"
                "3:      movl %3,%0\n"
                "        xor" "l" " %" "" "1,%" "" "1\n"
                "        jmp 2b\n"
                ".section __ex_table,\"a\"\n"
                "        .align 4\n"        "        .long 1b,3b\n"
                ".text"        : "=r"(__gu_err), "=r" (__gu_val) : "m"((*(struct __large_struct *)
                              (   __gu_addr   )) ), "i"(- 14 ), "0"(__gu_err));
                break;
            default:
              (__gu_val) = __get_user_bad();
          }
        } while (0) ;
      ((c)) = (__typeof__(*((buf))))__gu_val;
      __gu_err;
    }
  );
```

WOW! Black GCC/assembly magic. This is impossible to follow, so let's see what code gcc generates:

```
> 	xorl %edx,%edx
> 	movl current_set,%eax
> 	cmpl $24,788(%eax)
> 	je .L1424
> 	cmpl $-1073741825,64(%esp)
> 	ja .L1423
> .L1424:
> 	movl %edx,%eax
> 	movl 64(%esp),%ebx
> #APP
> 1:	movb (%ebx),%dl                /* this is the actual user access */
> 2:
> .section .fixup,"ax"
> 3:	movl $-14,%eax
> 	xorb %dl,%dl
> 	jmp 2b
> .section __ex_table,"a"
> 	.align 4
> 	.long 1b,3b
> .text
> #NO_APP
> .L1423:
> 	movzbl %dl,%esi
```

The optimizer does a good job and gives us something we can actually understand. Can we? The actual user access is quite obvious. Thanks to the unified address space we can just access the address in user memory. But what does the .section stuff do?????

To understand this we have to look at the final kernel:

```
> objdump --section-headers vmlinux
>
> vmlinux:     file format elf32-i386
>
> Sections:
> Idx Name          Size      VMA       LMA       File off  Algn
>   0 .text         00098f40  c0100000  c0100000  00001000  2**4
>                   CONTENTS, ALLOC, LOAD, READONLY, CODE
>   1 .fixup        000016bc  c0198f40  c0198f40  00099f40  2**0
>                   CONTENTS, ALLOC, LOAD, READONLY, CODE
>   2 .rodata       0000f127  c019a5fc  c019a5fc  0009b5fc  2**2
>                   CONTENTS, ALLOC, LOAD, READONLY, DATA
>   3 __ex_table    000015c0  c01a9724  c01a9724  000aa724  2**2
>                   CONTENTS, ALLOC, LOAD, READONLY, DATA
>   4 .data         0000ea58  c01abcf0  c01abcf0  000abcf0  2**4
>                   CONTENTS, ALLOC, LOAD, DATA
>   5 .bss          00018e21  c01ba748  c01ba748  000ba748  2**2
>                   ALLOC
>   6 .comment      00000ec4  00000000  00000000  000ba748  2**0
>                   CONTENTS, READONLY
```

```
>   7 .note         00001068  00000ec4  00000ec4  000bb60c  2**0
>                   CONTENTS, READONLY
```

There are obviously 2 non standard ELF sections in the generated object file. But first we want to find out what happened to our code in the final kernel executable:

```
> objdump --disassemble --section=.text vmlinux
>
> c017e785 <do_con_write+c1> xorl   %edx,%edx
> c017e787 <do_con_write+c3> movl   0xc01c7bec,%eax
> c017e78c <do_con_write+c8> cmpl   $0x18,0x314(%eax)
> c017e793 <do_con_write+cf> je     c017e79f <do_con_write+db>
> c017e795 <do_con_write+d1> cmpl   $0xbfffffff,0x40(%esp,1)
> c017e79d <do_con_write+d9> ja     c017e7a7 <do_con_write+e3>
> c017e79f <do_con_write+db> movl   %edx,%eax
> c017e7a1 <do_con_write+dd> movl   0x40(%esp,1),%ebx
> c017e7a5 <do_con_write+e1> movb   (%ebx),%dl
> c017e7a7 <do_con_write+e3> movzbl %dl,%esi
```

The whole user memory access is reduced to 10 x86 machine instructions. The instructions bracketed in the .section directives are no longer in the normal execution path. They are located in a different section of the executable file:

```
> objdump --disassemble --section=.fixup vmlinux
>
> c0199ff5 <.fixup+10b5> movl   $0xfffffff2,%eax
> c0199ffa <.fixup+10ba> xorb   %dl,%dl
> c0199ffc <.fixup+10bc> jmp    c017e7a7 <do_con_write+e3>
```

And finally:

```
> objdump --full-contents --section=__ex_table vmlinux
>
>  c01aa7c4 93c017c0 e09f19c0 97c017c0 99c017c0  ................
>  c01aa7d4 f6c217c0 e99f19c0 a5e717c0 f59f19c0  ................
>  c01aa7e4 080a18c0 01a019c0 0a0a18c0 04a019c0  ................
```

or in human readable byte order:

```
>  c01aa7c4 c017c093 c0199fe0 c017c097 c017c099  ................
>  c01aa7d4 c017c2f6 c0199fe9 c017e7a5 c0199ff5  ................
                               ^^^^^^^^^^^^^^^^^
                               this is the interesting part!
>  c01aa7e4 c0180a08 c019a001 c0180a0a c019a004  ................
```

What happened? The assembly directives:

```
.section .fixup,"ax"
.section __ex_table,"a"
```

told the assembler to move the following code to the specified sections in the ELF object file. So the instructions:

```
3:      movl $-14,%eax
        xorb %dl,%dl
        jmp 2b
```

ended up in the .fixup section of the object file and the addresses:

```
.long 1b,3b
```

ended up in the __ex_table section of the object file. 1b and 3b are local labels. The local label 1b (1b stands for next label 1 backward) is the address of the instruction that might fault, i.e. in our case the address of the label 1 is c017e7a5: the original assembly code: > 1: movb (%ebx),%dl and linked in vmlinux : > c017e7a5 <do_con_write+e1> movb (%ebx),%dl

The local label 3 (backwards again) is the address of the code to handle the fault, in our case the actual value is c0199ff5: the original assembly code: > 3: movl $-14,%eax and linked in vmlinux : > c0199ff5 <.fixup+10b5> movl $0xfffffff2,%eax

If the fixup was able to handle the exception, control flow may be returned to the instruction after the one that triggered the fault, ie. local label 2b.

The assembly code:

```
> .section __ex_table,"a"
>         .align 4
>         .long 1b,3b
```

becomes the value pair:

```
>  c01aa7d4 c017c2f6 c0199fe9 c017e7a5 c0199ff5  ................
                               ^this is ^this is
                               1b       3b
```

c017e7a5,c0199ff5 in the exception table of the kernel.

So, what actually happens if a fault from kernel mode with no suitable vma occurs?

1. access to invalid address:

   ```
   > c017e7a5 <do_con_write+e1> movb   (%ebx),%dl
   ```

2. MMU generates exception
3. CPU calls do_page_fault
4. do page fault calls search_exception_table (regs->eip == c017e7a5);
5. search_exception_table looks up the address c017e7a5 in the exception table (i.e. the contents of the ELF section __ex_table) and returns the address of the associated fault handle code c0199ff5.
6. do_page_fault modifies its own return address to point to the fault handle code and returns.
7. execution continues in the fault handling code.
8. a. EAX becomes -EFAULT (== -14)
   b. DL becomes zero (the value we "read" from user space)
   c. execution continues at local label 2 (address of the instruction immediately after the faulting user access).

The steps 8a to 8c in a certain way emulate the faulting instruction.

That's it, mostly. If you look at our example, you might ask why we set EAX to -EFAULT in the exception handler code. Well, the get_user macro actually returns a value: 0, if the user access was successful, -EFAULT on failure. Our original code did not test this return value, however the inline assembly code in get_user tries to return -EFAULT. GCC selected EAX to return this value.

NOTE: Due to the way that the exception table is built and needs to be ordered, only use exceptions for code in the .text section. Any other section will cause the exception table to not be sorted correctly, and the exceptions will fail.

Things changed when 64-bit support was added to x86 Linux. Rather than double the size of the exception table by expanding the two entries from 32-bits to 64 bits, a clever trick was used to store addresses as relative offsets from the table itself. The assembly code changed from:

```
    .long 1b,3b
to:
        .long (from) - .
        .long (to) - .
```

and the C-code that uses these values converts back to absolute addresses like this:

```
ex_insn_addr(const struct exception_table_entry *x)
{
        return (unsigned long)&x->insn + x->insn;
}
```

In v4.6 the exception table entry was expanded with a new field "handler". This is also 32-bits wide and contains a third relative function pointer which points to one of:

1. `int ex_handler_default(const struct exception_table_entry *fixup)`
   This is legacy case that just jumps to the fixup code
2. `int ex_handler_fault(const struct exception_table_entry *fixup)`
   This case provides the fault number of the trap that occurred at entry->insn. It is used to distinguish page faults from machine check.

More functions can easily be added.

CONFIG_BUILDTIME_TABLE_SORT allows the __ex_table section to be sorted post link of the kernel image, via a host utility scripts/sorttable. It will set the symbol main_extable_sort_needed to 0, avoiding sorting the __ex_table section at boot time. With the exception table sorted, at runtime when an exception occurs we can quickly lookup the __ex_table entry via binary search.

This is not just a boot time optimization, some architectures require this table to be sorted in order to handle exceptions relatively early in the boot process. For example, i386 makes use of this form of exception handling before paging support is even enabled!