TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

강사 : Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 : 황수정
sue100012@naver.com
41일차 (2018. 04. 19)

※ screen

한 창에 여러 개의 터미널을 생성할 수 있게 해주는 명령어이다. screen 은 한 터미널에서 여러 쉘과 프로그램을 사용할 수 있다. 세션관리 기능도 지원 한다. 그래서 screen 을 종료하고 심지어 터미널까지 로그아웃하고 종료하여도 세션이 유지된다. 다음에 다시 터미널로 screen으로 세션을 불러와서 다시 이전 작업을 이어서 할 수 있다.

sudo apt-get install screen : screen 앱 다운

screen : screen 을 시작 하는 기본 명령어로 기본 세션명으로 시작한다.
screen -S 세션명 : -S 다음에 주는 세션명으로 시작한다.
screen –list : -list 옵션을 주고 실행하면 이전에 작업했었던 screen 리스트가 있으면 세션명과 함께 리스트를 보여준다.
screen -R 세션명 : 이전에 세션이 있을 경우 -R 다음에 오는 세션명으로 이전 작업을 불러온다. -R 다음에 세션명을 주지 않았을 경우에는 이전 세션이 한 개만 있을 경우 그 작업을 불러온다. 이전 작업이 여러 개 있을 경우에는 이전 작업 리스트를 보여준다.

Screen 실행 후 사용 명령어

Ctrl a, c : 새로운 screen이 생기면서 그 screen으로 이동한다.
Ctrl a, a : 이전 창으로 이동한다.
Ctrl a, n : 다음 창으로 이동한다.
Ctrl a, p : 이전 창으로 이동한다.
Ctrl a, 숫자 : 숫자에 해당하는 창으로 이동한다.
Ctrl a, ' : 창 번호 또는 창 이름으로 이동한다.
Ctrl a, " : 창 번호를 보여준다.
Ctrl a, A : 현재 창의 title을 수정한다.
Ctrl a, w : 창 리스트 보여준다.
Ctrl a, esc : Copy 모드로 전환. Copy 모드에서는 vi의 이동키로 이동을 할 수 있다.
Crtl a, [: 커서 이동을 할 수 있고 특정 블럭을 복사하는 기능으로 사용한다. 먼저 시작 위치에서 space 바를 누르고 끝 위치에서 space 바를 누르면 해당 부분이 buffer로 복사된다.
Ctrl a,] : buffer의 내용을 stdin으로 쏟아 넣는다. vi의 입력모드에서 사용하면 유용하다.
Ctrl a, :(콜론) : 명령 모드로 전환한다.
Ctrl a, d : (detach) 현재 작업을 유지하면서 screen 세션에서 빠져 나온다. 세션은 종료 되지 않는다.
Ctrl a, x : lock screen
ctrl a, K : 보고있는 가상터미널 화면을 지운다.
Ctrl a, S : (split) 창을 나눈다.

Ctrl a, Tab : 다른 screen으로 이동한다
Ctrl a, Q : 현재 screen을 제외한 나머지 숨긴다.
exit : screen 에서 exit 를 입력하면 세션이 완전히 종료된다.

※ 선생님께서 주신 코드를 이용하여 make하면 여러 파일이 생성되고 sudo insmod 명령어를 이용하여 설치한다. 해킹당했다는 메시지가 뜨면 잘 작동된 것이다.
다음은 그 소스 중 일부를 분석한 것이다.

```
int syscall_hooking_init(void)
{
unsigned long cr0;
if((sys_call_table = locate_sys_call_table()) == NULL)
{
printk("<0>Can't find sys_call_table\n");
return -1;
}
printk("<0>sys_call_table is at[%p]\n", sys_call_table);
// CR0 레지스터를 읽어옴
cr0 = read_cr0();
// Page 쓰기를 허용함
write_cr0(cr0 & ~0x00010000);
/* set_memory_rw 라는 심볼을 찾아와서 fixed_set_memory_rw 에 설정함 */
fixed_set_memory_rw = (void *)kallsyms_lookup_name("set_memory_rw");
if(!fixed_set_memory_rw)
{
printk("<0>Unable to find set_memory_rw symbol\n");
return 0;
}
/* 시스템 콜 테이블이 위치한 물리 메모리에 읽고 쓰기 권한 주기 */
fixed_set_memory_rw(PAGE_ALIGN((unsigned long)sys_call_table) - PAGE_SIZE, 3);
orig_call = (void *)sys_call_table[__NR_open];
sys_call_table[__NR_open] = (void *)sys_our_open;
write_cr0(cr0);
printk("<0>Hooking Success!\n");
return 0;
}
void syscall_hooking_cleanup(void)
{
#if 1
unsigned long cr0 = read_cr0();
write_cr0(cr0 & ~0x00010000);
sys_call_table[__NR_open] = orig_call;
write_cr0(cr0);
printk("<0>Module Cleanup\n");
```

```
#endif
}
module_init(syscall_hooking_init);
module_exit(syscall_hooking_cleanup);
MODULE_LICENSE("GPL");
```

ismod를 하면 'module_init'이 동작하면서 syscall_hooking_init 함수를 불러온다. 그 함수 내부에는

```
unsigned long cr0; // 변수가 선언되어있다.
if((sys_call_table = locate_sys_call_table()) == NULL) //sys_call_table에 값이 없을 경우를 나타낸다.
  { printk("<0>Can't find sys_call_table₩n");
    return -1;                                    }
printk("<0>sys_call_table is at[%p]₩n", sys_call_table);
```

처음 보이는 cr0 레지스터, 인텔 CPU에 있는 레지스터로 데이터시트를 확인한다. 이 데이터시트를 통해 cr0가 C.O.W를 한다는 것을 알 수 있다.

그 다음 if문을 해석하기 위해선 locate_sys_call_table() 함수로 가서 어떤 작동 과정을 거치는지를 살펴보아야 한다. 이 부분은 잘 모르겠어서 sys_call_table이 값이 없을 경우 정상 작동이 안되는 것이기에 넘어가고 다음부분부터 해석을 시도하였다. Printk는 sys_call_table에 관한 커널 메시지를 출력한다. printk() 함수는 커널 메시지를 출력하고 관리할 수 있다. 커널 메시지를 다음과 같이 나누어서 처리한다. "<숫자>"는 로그 레벨을 나타낸다.

상수 선언문	의미
#define KERN_EMERG	"<0>" /*시스템이 동작하지 않는다.*/
#define KERN_ALERT	"<1>" /*항상 출력된다 .*/
#define KERN_CRIT	"<2>" /*치명적인 정보 */
#define KERN_ERR	"<3>" /*오류 정보*/
#define KERN_WARNING	"<4>" /*경고 정보*/
#define KERN_NOTICE	"<5>" /*정상적인 경보*/
#define KERN_INFO	"<6>" /*시스템 정보*/
#define KERN_DEBUG	"<7>" /*시스템이 디버깅 정보*/

이 다음 코드는

```
cr0 = read_cr0();
```

cr0변수에 레지스터의 값을 넣어 줘야 하기 때문에 cr0를 읽어주는 동작을 실행시킨다. cr0의 값

을 알기 위해선 read_cr0가 무엇인지 알아야 한다. 커널 소스 코드로 들어가면

```
  #   line  filename / context / line
  1     59  arch/x86/include/asm/paravirt.h <<read_cr0>>
            static inline unsigned long read_cr0(void )
  2    114  arch/x86/include/asm/paravirt_types.h <<read_cr0>>
            unsigned long (*read_cr0)(void );
  3    112  arch/x86/include/asm/special_insns.h <<read_cr0>>
            static inline unsigned long read_cr0(void )
  4    350  arch/x86/kernel/paravirt.c <<read_cr0>>
            .read_cr0 = native_read_cr0,
  5   1211  arch/x86/xen/enlighten.c <<read_cr0>>
            .read_cr0 = xen_read_cr0,
  6     59  linux-4.4/arch/x86/include/asm/paravirt.h <<read_cr0>>
            static inline unsigned long read_cr0(void )
  7    114  linux-4.4/arch/x86/include/asm/paravirt_types.h <<read_cr0>>
            unsigned long (*read_cr0)(void );
  8    112  linux-4.4/arch/x86/include/asm/special_insns.h <<read_cr0>>
            static inline unsigned long read_cr0(void )
  9    350  linux-4.4/arch/x86/kernel/paravirt.c <<read_cr0>>
            .read_cr0 = native_read_cr0,
 10   1211  linux-4.4/arch/x86/xen/enlighten.c <<read_cr0>>
            .read_cr0 = xen_read_cr0,
Type number and <Enter> (empty cancels):
```

다음과 같이 뜬다. 그럼 3번으로 들어가면 되는데 이는 우리가 분석하는 것이 인텔로 x86이기 때문에 x86이 아닌 것은 걸러낸다. 그 후, xen과 virt는 가상을 뜻하므로 제외한다.

```
111
112 static inline unsigned long read_cr0(void)
113 {
114     return native_read_cr0();
115 }
116
```

위의 read_cr0는 native_read_cr0()를 반환한다. 반환 값을 알기 위해서는 native_read_cr0()를 해석해 주어야 한다.

```
122 static inline unsigned long read_cr2(void)
123 {
cscope tag: native_read_cr0al_insns.h" 269L, 6144C          114,10-13     40%
  #   line  filename / context / line
  1     23  arch/x86/include/asm/special_insns.h <<native_read_cr0>>
            static inline unsigned long native_read_cr0(void )
  2     23  linux-4.4/arch/x86/include/asm/special_insns.h <<native_read_cr0>>
            static inline unsigned long native_read_cr0(void )
Type number and <Enter> (empty cancels):
 23 static inline unsigned long native_read_cr0(void)
 24 {
 25     unsigned long val;
 26     asm volatile("mov %%cr0,%0\n\t" : "=r" (val), "=m" (__force_order));
 27     return val;
 28 }
 29
```

val이라는 변수가 선언되었고 asm volatile() 함수를 이용한다. 여기서 asm은 어셈블리어를 사용한다는 뜻이고 volatile은 컴파일러는 프로그래머가 입력한 그래도 남겨두는 작업을 한다. 즉, 최적화나 위치를 옮기는 등의 일을 하지 않는다. 컴파일러가 자동으로 해준 일 때문에 버그가 발생할 수도 있기 때문이다. Volatile (asms : output : input : clobber);와 같이 사용된다.이는 (구동시킬 명

령어 : 입력 : 출력 : 어셈블리지시어)를 나타낸다. 여기서 output은 변수들을 적어 주고 각각은 쉼표고 구분된다. 결과 값을 출력하는 변수를 적는다. Input은 output과 같은 방식으로 사용하고 인라인 어셈블리 코드에 넘겨주는 파라미터를 적는다.

각각의 인자들 중에서 asms는 반드시 있어야 한다. 그러나 output, input, clobber는 각각 없을 수도 있다. 만약 clobber가 없는 경우 라면 clobber와 바로 앞의 콜론(:)을 같이 쓰지 않아도 된다. 마찬가지로 input, clobber가 없다면 output까지만 쓰면 된다.

함수 인자를 살펴보면, 어셈블리어가 나온다. mov %%cr0,%0 여기서 %0은 처음 선언된 변수를 가리킨다. 즉 %0 = val이라는 뜻이다. =r는 레지스터를 뜻하고, =m 메모리를 사용하겠다는 뜻이다.

위 코드에서는 ':' 가 1개밖에 없으므로 구동시킬 명령어를 읽어온다. cr0레지스터 값이 val 로 들어가게 되므로 read가 가능(mov 연산은 왼쪽에서 오른쪽으로 이동)하게 된다. 메모리의 값은 __force_order로 갱신하게 된다.