3.2. Function call with stack switch

Stack 사용량을 어느 정도 이하로 보장하기 위해서 어떤 기능들은 stack을 바꾸어 가면 실행하는 경우가 있습니다. 예를 들어, OS에서 interrupt handler와 그에 따라 실행되는 bottom half handler들의 경우 interrupt 발생시의 kernel stack에서 실행된다면 interrupt nesting 등을 생각할 때 모든 kernel thread의 stack 크기가 꽤 커져야 하는데다가 필요한 크기를 정확히 알기도 힘듭니다.

아래의 프로그램에서 call_with_stack_switch는 funcaddr로 주어진 함수를 arg를 인자로 해서 altstack에서 수행하고 그 결과값을 돌려줍니다.

```
#include <stdio.h>

#define fetch_esp(_esp) ₩
        __asm__ __volatile__("movl %%esp, %0" : "=g" (*_esp))

static __inline__ int
call_with_stack_switch(void *funcaddr, unsigned int arg, void *altstack)
{
        int a, b, c, d, D, S;

        __asm__ __volatile__(
                "pushl  %%ebp           ₩n₩t"
                "movl   %%esp, %%eax    ₩n₩t"
                "movl   %8, %%esp       ₩n₩t"
                "pushl  %%eax           ₩n₩t"
                "pushl  %7              ₩n₩t"
                "call   *%6             ₩n₩t"
                "addl   $4, %%esp       ₩n₩t"
                "popl   %%esp           ₩n₩t"
                "popl   %%ebp           "
                : "=&a" (a), "=b" (b), "=c" (c), "=d" (d), "=D" (D), "=S" (S)
                : "r" (funcaddr), "ri" (arg), "ri" (altstack));

                return a;
}

static int
say_hello(unsigned int arg)
{
        unsigned esp;

        fetch_esp(&esp);

        printf("say_hello : hello world... esp=%08x, arg=%d₩n", esp, arg);
        arg *= arg;
        printf("say_hello : returning %d₩n", arg);
        return arg;
}

static char _altstack[8192];
static void *altstack = _altstack + sizeof(_altstack);

int
main(void)
{
        unsigned esp;
        int rv, arg = 1096;

        fetch_esp(&esp);
```

```
        printf("main      : current esp=%08x, altstack=%08p-%08p₩n",
               esp, _altstack, altstack);
        printf("main      : calling say_hello w/ stack switch (arg=%d)₩n",
               arg);

        rv = call_with_stack_switch(say_hello, arg, altstack);

        fetch_esp(&esp);

        printf("main      : esp=%08x, arg=%d, rv=%d₩n", esp, arg, rv);
        return 0;
}
```

call_with_stack_switch에서 6개의 변수가 선언되어 있는데 이 변수들은 모두 레지스터들의 output으로 쓰입니다. a:eax, b:ebx, c:ecx, d:edx, D:edi, S:edi로 대응이 됩니다. a외에는 output이라고 정의된 후 쓰이지 않는데, 단지 그 레지스터들의 값이 바뀐다는 것을 컴파일러에게 알려주는 역활만을 하게됩니다. Clobber list와 거의 같은 기능이라고 할 수 있지만 clobber로 지정된 레지스터는 input, output 어느것으로도 쓰일 수 없고 위의 inline assembly에 있는 세개의 input 변수들이 그 레지스터로 할당될 수 없게됩니다. 즉, dummy 변수를 써서 output으로 정해주게 되면 'input으로 할당될 수 있지만 결과적으로 값은 변한다'라는 뜻입니다.

각 라인을 살펴보도록 하겠습니다.

```
 1: pushl       %%ebp
```

inline assembly의 앞과 끝에서 ebp를 저장하고 복구하는데 ebp는 ix86에서 frame pointer로 쓰입니다. 만약 -fomit-frame-pointer 옵션을 주지않고 컴파일하면 frame pointer의 관리가 함수 entry/exit에서 되어 신경 쓸 필요가 없지만 frame pointer를 생략하게 되면 컴파일러가 ebp를 다른 용도로 쓰게됩니다. 하지만 gcc에게 ebp가 변함을 알려줄 방법이 없기때문에 컴파일러가 모르는 체로 ebp의 값이 바뀌어 버릴 수가 있습니다. 따라서 다른 레지스터들과 달리 따로 저장/복구해 줄 필요가 있습니다.

ebp와 gcc에 대해 조금 더 설명하겠습니다. ebp는 완전한 범용 레지스터는 아니지만 대부분의 주소계산에 사용될 수 있고 값들을 잠시 저장하는 장소로도 사용될 수 있습니다. gcc는 frame pointer로 쓰지 않을 때 ebp를 이런 용도로 사용하지만 input/output에서 직접 ebp를 지정해줄 수 있는 방법이 없고, clobber list에서 지정을 할 수 있지만 무시되기때문에 inline assembly에서 ebp의 값을 변화시킬 때는 반드시 저장/복구 해주어야 합니다. Gcc의 버그라고도 할 수 있습니다.

```
 2: movl        %%esp, %%eax
```

현재 esp값을 eax에 저장합니다. altstack으로 바꾸어 함수를 실행하고 원래의 stack으로 돌아와야 하기 때문에 원래 stack pointer를 기억하고 있어야 합니다. 이를 altstack으로 바꾸고 제일 처음에 push하기 위해 eax에 저장해 두는 것입니다.

```
 3: movl        %8, %%esp
```

%8은 altstack입니다. altstack으로 stack을 바꿉니다.

```
 4: pushl       %%eax
```

원래의 stack pointer를 stack에 저장합니다.

```
5: pushl        %7
```

%7은 arg입니다. 함수 호출을 위해 arg를 새로 바뀐 stack에 push합니다.

```
6: call         *%6
```

funcaddr을 호출합니다. *는 indirect call임을 나타냅니다. Input에서 더 자세히 설명하겠습니다.

```
7: addl         $4, %%esp
```

funcaddr의 실행이 끝났으므로 arg를 없앱니다.

```
8: popl         %%esp
```

원래의 stack으로 돌아옵니다.

```
9: popl         %%ebp
```

1에서 저장해둔 ebp를 복구합니다.

Output에서 a가 early clobber로 지정된 것 이외에는 특별한 점이 없습니다. eax를 제외한 레지스터들은 funcaddr의 함수가 실행하면서 즉, 모든 input이 다 쓰인 후에 바뀔 수 있기 때문에 early clobber로 지정할 필요가 없고 따라서 input에 할당될 수 있습니다.

Input에서 funcaddr은 범용 레지스터, arg와 altstack은 범용 레지스터 또는 immediate constraint를 가지고 있습니다. Memory operand는 esp에 대한 offset addressing으로 표현될 수 있고, esp를 바꾼 후에 input들을 사용하기 때문에 memory operand는 혀용할 수 없으므로 레지스터나 immediate을 사용해야 하는데 ix86의 call instruction은 immediate operand로는 relative call밖에 지원하지 않기 때문에 indirect call을 해야하고 따라서 'r' constraint를 써야합니다. 나머지 둘은 immediate이어도 관계가 없기 때문에 'ri' constraint를 가지고 있습니다.

arg와 altstack이 call_with_stack_switch의 인자이기 때문에 immediate이 의미없다고 생각할 수도 있지만, __inline__으로 선언되어 있기 때문에 인자가 compile time에 결정될 수 있으면 immediate이 됩니다. 아래의 컴파일한 assembly를 보면 알 수 있습니다.

```
        .file   "call_with_stack_switch.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
        .align 32
.LC0:
        .string "say_hello : hello world... esp=%08x, arg=%d₩n"
.LC1:
        .string "say_hello : returning %d₩n"
.text
        .align 4
        .type    say_hello,@function
say_hello:
        subl $4,%esp
        pushl %ebx
        movl 12(%esp),%ebx
#APP
        movl %esp, 4(%esp)
#NO_APP
        pushl %ebx
```

```
        pushl 8(%esp)
        pushl $.LC0
        call printf
        imull %ebx,%ebx
        pushl %ebx
        pushl $.LC1
        call printf
        movl %ebx,%eax
        addl $20,%esp
        popl %ebx
        popl %ecx
        ret
.Lfe1:
        .size    say_hello,.Lfe1-say_hello
.data
        .align 4
        .type    altstack,@object
        .size    altstack,4
altstack:
        .long _altstack+8192
.section        .rodata
        .align 32
.LC2:
        .string "main      : current esp=%08x, altstack=%08p-%08p₩n"
        .align 32
.LC3:
        .string "main      : calling say_hello w/ stack switch (arg=%d)₩n"
        .align 32
.LC4:
        .string "main      : esp=%08x, arg=%d, rv=%d₩n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        subl $4,%esp
        pushl %ebp
        pushl %edi
        pushl %esi
        pushl %ebx
#APP
        movl %esp, 16(%esp)
#NO_APP
        pushl altstack
        pushl $_altstack
        pushl 24(%esp)
        pushl $.LC2
        call printf
        pushl $1096
        pushl $.LC3
        call printf
        movl $say_hello,%edx
        movl altstack,%eax
        addl $24,%esp
        movl %eax,%ebp
#APP
        pushl   %ebp
        movl    %esp, %eax
        movl    %ebp, %esp
        pushl   %eax
        pushl   $1096
        call    *%edx
        addl    $4, %esp
        popl    %esp
        popl    %ebp
        movl %esp, 16(%esp)
#NO_APP
        pushl %eax
        pushl $1096
```

```
        pushl 24(%esp)
        pushl $.LC4
        call printf
        xorl %eax,%eax
        addl $16,%esp
        popl %ebx
        popl %esi
        popl %edi
        popl %ebp
        popl %ecx
        ret
.Lfe2:
        .size    main,.Lfe2-main
        .local  _altstack
        .comm   _altstack,8192,32
        .ident  "GCC: (GNU) 2.95.4 20010902 (Debian prerelease)"
```

call_with_stack_switch가 main안에 inlining 되었고, altstack이 %ebp로, arg는 immediate operand로, funcaddr이 %edx로 할당된 것을 볼 수 있습니다. 또, Dummy 변수들은 모두 사라졌고, return 값인 a도 %eax에 있는 그대로 사용되고 있습니다.

위의 프로그램을 실행하면 다음과 같은 결과가 나옵니다.

```
% ./call_with_stack_switch
main      : current esp=bffffc3c, altstack=0x80497c0-0x804b7c0
main      : calling say_hello w/ stack switch (arg=1096)
say_hello : hello world... esp=0804b7ac, arg=1096
say_hello : returning 1201216
main      : esp=bffffc3c, arg=1096, rv=1201216
```

Inline assembly를 사용할 때는 레지스터 할당이 정확히 어떻게 되는지 프로그램을 쓰면서는 알 수 없고, 특히 early clobber 옵션은 잊기가 쉽고 잘못되었을 때 찾기가 상당히 힘들기 때문에 제대로 작동하는 것 같더라도 -S 옵션을 주어 원하는 코드가 생성되었는지를 확인해보는 것이 좋습니다.

이전
Atomic bit operations & spin lock

처음으로
위로