제 19 회 한국정보처리학회 춘계학술발표대회 논문집 제 10 권 제 1 호 (2003. 5)

Linux 운영체제에서 Kernel Hardening 설계

문지훈*,김기환*,장승주*,정성인**
*동의대학교 컴퓨터공학과
**한국 전자 통신 연구원

e-mail : ki-hany@hanmail.net
{mjh,sjjang}@dongeui.ac.kr, sijung@etri.re.kr

Design of the Kernel Hardening in the Linux O.S.

Ji-Hoon Moon*, Ki-Hwan Kim*, Seung-Ju Jang*, Seung-In Jung**
*Dept. of Computer Engineering, Dong-Eui University
**ETRI

요 약

본 논문에서는 Linux 운영체제에서의 kernel hardening 을 설계한다. 커널 내에서 panic 이 발생할 경우 복구가 가능한 경우에는 정상적인 동작이 될 수 있도록 한다. 이렇게 함으로써 Linux Kernel Hardening 기능은 안정적인 커널의 동작을 보장한다. 본 논문에서 Linux Kernel Hardening 을 보장하기 위하여 커널 내 ASSERT(), BUG() 함수를 중심으로 설계를 한다.

1. 서론

kernel hardening[4]은 커널의 구현이나 디자인을 분석하는 과정과 이를 더욱 유용하고 정확하게 좀 더 쉽고 안정적이게 유지하며, 실패한 경우의 유용한 정보를 모을 수 있도록 하게 하는 코드 변환을 위한 전략과 제안을 구현하는 과정이다. 즉, hardening 은 시스템의 유용성과 유지성, 그리고 시스템이 잘못되는 경우의 증거 수집을 향상시키고, 복구를 전제로 한 하드웨어의 failure 감지가 필요하다.

hardening 과정은 시스템 적합성에 대한 판단으로 모든 조건에서 각각의 kernel panic 에 대한 검토를 포함하고 있다.

kernel hardening 은 HA 에 기반을 두고 있다. 임의의 fault 설정에 따른 panic 에 적절히 대처할 수 있는 기법으로 code path 시험을 통한, 즉 에러 코드에 대한 모순되는 않는 과정을 피해보고자 하는데서 시도해보는 과정이다. 이를 fault injection 이라고 말하고 있는데, 이러한 실험적 방법을 통하여 구현하게 되는 커널 리소스를 통해 보다 안정된 code 를 생성하고자 이를 구현하게 된다. high availability 는 다중 CPU 사이에서 그들간에 기능성을 분배함으로써 컴퓨터 시스템의 "up-time"를 강화해는 기술을 가르킨다[11].

본 논문은 시스템 panic 으로 인해 발생하는 메시지를 살펴봄으로써 실험을 시작하게 되었다. 감지된 내부 에러에서 레지스터의 덤프나 스택의 내용을 트레이스해 보는 과정에서 문제에 대한 해결 방안을 찾아보려 시도해 본다.

2. 설계

panic()함수는 리눅스 커널 내에서 치명적인 에러를 호출하게 된다. 이러한 상황에 좀더 효율적으로 대처할 수 있는 방법에 대한 부분부터 시작한다. 단일 시스템 내에서의 실험을 통하여 커널 소스의 구도적 조직을 안정화할 수 있도록 한다.

2.1 ip_input.c 파일의 수정과정

시스템에 fault 가 되는 조건을 코드 삽입하여 인위적으로 만드는 작업이 필요하다. 먼저 telnet 접속으로 인하여 접근하게 되는 커널 소스에 대한 접근부터 시

본 논문은 2003 년도 한국전자통신연구원 연구비 지원에 의하여 수행되었음.

도해 본다. telnet 접속시 접근하게 되는 경로를 찾고자 먼저 loopback 을 이용해 kernel 이 panic 으로 일으키게 할 수 있는 조건을 만들어 준다. 다음은 linux/net/ipv4/ip_input.c 에 삽입한 코드의 내용이다. [그림 1]와 같이 ip_rcv()함수 내 소스 구현으로 panic 이 발생함을 볼 수 있고, 그 과정은 linux/kernel/panic.c 에서 살펴볼 수 있다.

```
static int harden_count; //panic이 되기 전의 상황을
        //확인하기 위해 임의의 global변수 선언
        //외부에 설정.
char *aa;

harden_count++;
printk("ip_rcv()->%d₩n", harden_count);
if(harden_count>150){
aa=10+10;
printk("aa=%d", *aa);
}
else{
   printk("harden_count=%d₩n", harden_count);
}
```

[그림 1] ip_input.c 의 ip_rcv()함수내 삽입 코드

[그림 1] 소스 내의 모든 함수에 printk()함수를 이용해 시스템이 panic 이 발생될 시에 들어갈 수 있는 부분들을 살펴보며 프로그램 추적을 하게 된다. 위 소스의 로컬 변수 aa 가 가지게 되는 주소값에 의해 최초 문제가 발생한다. 이 부분은 커널의 유저모드가 아닌 예외 발생을 일으키게 한다. printk()함수의 효율적인 사용으로 시스템의 흐름을 파악하여 시스템에 panic 이 발생시 나타나는 메시지를 살펴보며 호출하게 되는 프로시저를 찾을 수 있다. linux/arch/i386 /mm/fault.c 의 소스 코드에서 do_page_fault()함수를 호출한다.

2.2 do_page_ fault()함수

do_page_fault()함수는 page fault 인터럽트 서비스 루틴이다. 현재 프로세스의 메모리 영역에 대해서 page fault 가 발생된 선형주소와 비교하는 곳인데, 여기에서 kernel 이 다루게 되는 address 의 값에 따라 처리하는 루틴은 good_area, bad_area, no_context, do_sigbus 가 있는데 각 부분에 대한 기능적인 설명은 다음과 같다.

good_area : process address 영역 안에서 발생한 faulty address 를 다룬다.

bad_area : process address 영역 밖에서 발생한 faulty addresss 를 다룬다.

no_context : 인터럽트가 진행 중이거나 kernel thread 가 실행중일 때 발생하는 exception 을 처리하는 부분이다.

do_sigbus : 획득한 세마포어를 놓고(up), 프로세스의 thread 구조체에 있는 cr2 필드에 오류를 일으킨 주소(address)를 넣는다.

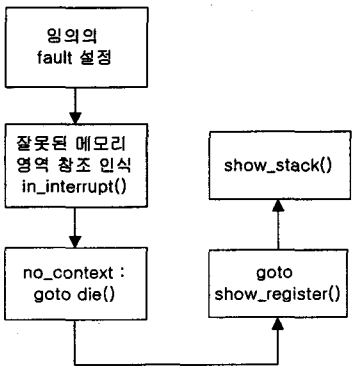
설정해 놓은 작업으로 인해 발생하는 panic 으로 인해 발생하는 address 부분에서 살펴보면 바로 char 형 변수 *aa 가 가지게 되는 0x14 라는 값이 *aa 의 address 주소값을 가르키고 있는 값이 된다. 여기서 설정된 조건으로 인해 커널 영역에서 사용되고 있지 않은 메모리 영역이라는 page interrupt handler 의 흐름에 따라 panic 을 일어나게 된다[2].

```
__asm__(movl %%cr2,%0):""=r(address));"
```

[그림 2] faulty address

[그림 2]의 코드가 do_page_fault() 함수에서 접근할 수 없는 address 를 가지게 되는 부분인데, cr2 레지스터에서 그 값을 얻어 로컬 변수 address 에는 0x14 라는 값을 가지게 된다.

현재 진행 중인 작업 루틴에 호출과정을 [그림 3]과 같다.



[그림 3] fault 시 호출 함수.

[그림 4]는 fault.c 의 no_context 를 호출하는 과정을 살펴보면, 각 해당 프로시져들이 위의 함수들을 호출하게 된다 .

```
no_context:
.....
ret = die("Oops", regs, error_code);
printk("RETURN from die() in do_page_fault()
                !!!!!,ret=%d₩n",ret);
if(ret == 1) {
   //정상적으로 복구가 될 수 있도록 한다.
   return;
}
```

[그림 4] no_context 부 내 삽입 코드

show_stack 함수 특성을 이용하여 sp 에 esp 스택 포인터값을 이동함으로써 스택 프레임에 구성하고 있는 aa 의 주소값을 확인하여 삽입한다.

2.4 die()함수

다음으로 수행되는 부분이 die()함수인데 이는 linux/arch/i386/kernel/traps.c 에서 확인할 수 있다. die() 함수를 살펴보면 다음과 같다.

```
die(){
    .....
    ret = show_register();
    if(ret ==1)
          return ret;
    ....
}
```

[그림 5] die()함수 내 소스 삽입

die()함수에서도 함수 호출을 하는데 show_register()함수를 ret 리턴값을 가지게 하여 정상적인 복구 과정을 거치게 된다.

```
show_register(){
    ...
    if(in_kernel)
          printk("₩nstack: : ");
          ret = show_stack();
          if(ret == 1)
             return ret;
      }
}
```

[그림 6] show_register()내 소스 삽입부

show_register()함수를 위와 같이 리턴값을 설정하여 show_stack()함수의 리턴값을 받는다. show_stack()에서는 fault 난 address 값인 0x14 를 가지게 되는 값을 새로이 메모리를 할당해 주는 작업을 한다. show_stack을 이용하면 esp 레지터스값이 가르키는 스택값을 차례로 보여주게 된다.

```
show_stack(unsigned long * esp){
    ....
stack=esp;
   ....
__asm__("movl %%cr2,%0":"=r"(address));
    ....
if(*stack == address){
*stack = kmalloc(8,GFP_KERNEL);
return 1;
}

....

//show_trace(esp);
return 0;
}
```

[그림 7] show_stack()내 소스 삽입부

[그림 7]의 작업 루틴을 통해 do_page_fault() 함수까지 다시 반환하게 되는 루틴을 통하여 ip_rcv() 함수로 정상적인 반환값을 리턴함으로써 복구가 가능하다.

2.5 ASSERT()구조 파악 및 BUG()

panic 을 일으키게 하는 또 다른 함수중에 ASSERT 함수가 존재한다. 이 함수를 구현하는 부분은 매크로로 정의된다.

매크로 함수는 [그림 8]과 같다.

```
#define ASSERT(expr) do { ₩
   if(!expr){₩
   printk("Assertino[%s]failed %s:%s(line=%d)₩n"₩
      #expr,__FILE__,__FUNCTION__,__LINE__);₩
   panic("Assertion panic₩n");₩
   }}while(0)
```

[그림 8] ASSERT() 매크로 코드

[그림 8]은 ASSERT()함수가 가지는 매크로 정의를 보면, 이 함수의 인자값(expr1)을 비교 판단하게 되어 panic 함수를 거치게 된다. expl 의 조건이 false 를 가지게 되는 조건을 만족하게 되면 panic 이 일어나게 되지만 false 조건내에 true 조건을 삽입함과 동시에 ASSERT()함수의 인자값(expr2) 조건도 추가적으로 삽입을 하여 ASSERT()함수 호출시 정상적인 값으로 복귀하여 처리한다.

ASSERT()함수가 가지게 되는 인자값들에 대한 비교는 value 타입과 address 타입을 생각해 볼 수 있는데, 이런 타입들의 비교를 통해 리눅스 커널에서는 여러 가지의 경우가 발생하고 있다.

위의 설명처럼 강제적으로 expr2 에 참인 조건을 만들어 줌으로서 value 타입과 address 타입에서는 하나의 인자값을 강제적으로 삽입함으로써 hardening 작업을 거치는 소프트웨어적 기법을 살펴보았다. 여기서 address 타입에 메모리를 할당하면 kfree() 함수를 사용하여 메모리를 해제해 주어야 한다. [그림 9]는 수정된 ASSERT()함수의 매크로 코드이다.

```
#define ASSERT(expr1,expr2) do { ₩
   if(!expr1){₩
   printk("Assertino[%s]failed %s:%s(line=%d)₩n"₩
      #expr,__FILE__,__FUNCTION__,__LINE__);₩
   expr2;₩
   }}while(0)
   //panic("Assertion panic₩n");₩
```

[그림 9] 수정된 ASSERT 매크로 코드

또 다른 panic 을 일으키게 하는 함수를 살펴보면, BUG()라는 함수가 있는데 이 또한 매크로로 정의되어져 있다.

```
#define BUG() __asm__ __volatile__(".byte 0x0f, 0x0b")
```

[그림 10] BUG() 매크로 코드

[그림 10]은 커널이 사용할 수 없는 이외의 영역을 지정함으로써 panic 을 일으킨다. BUG() 함수와 관련한 Linux 커널 부분을 ASSERT()함수 형식으로 변환함으로써 kernel hardening 을 구현한다.

3. 실험

단일 시스템 내에서 루프백을 이용하여 시스템이 다운이 될 수 있게 만든다. 시스템이 panic 이 일어나게 되면, 터미널에 메시지가 나타난다. 이 메시지의 call trace 값을 살펴보고 nm -n 이라는 심볼 보기 옵션을 통해 최초에 출력되는 스택값들을 확인할 수 없으므로 call trace 가 일어나는 값을 이용하는 방법에 대한 결과값을 얻지 못했다. 프로그램 작성시 stack 에 가지게 되는 stack frame[13]을 통해 문제를 해결하였다. 실험에 사용한 intel cpu architecture 의 구조에서 하나의 프로시저에 대한 정보를 가지게 되는 것은 스택 프레임의 베이스 포인터와 반환 명령어 포인터이다. 스택 프레임의 베이스 포인터를 사용하는 것은 호출되는 프로시저가 우선적으로 스택의 로컬 변수들을 push 하여 ESP 레지스터의 내용을 EBP 레지스터에 복사한다. 즉, 스택 프레임의 EBP 레지스터 처음부터가 로컬변수가 가지게 되는 부분임을 확인해 보면, 간단한 서브루틴 콜을 부르는 프로그램에서 각각의 서브루틴들마다 몇 개의 로컬 변수를 선언한 후 .c 파일의 컴파일을 gcc -S 옵션을 통해 생기게 되는 어셈블리의 작업 내역을 살펴보면 쉽게 이 부분에 대한 이해를 할 수 있다[1,5].

이 부분에 대한 구조적 이해를 살펴보려면, [그림 11]의 소스 코드를 통해서 확인 하였다.

```
__asm__("pop %%eax₩n₩t" : : );
__asm__("movl %%eax, %0" : "=r"(reg));
printf(" 0x%x₩n", reg);
```

[그림 11] 스택값 확인

[그림 11] 소스 부분에 대한 .c 파일의 메인 루틴에 삽입하여 반복적 작업으로 서브 루틴 콜에 의해 쌓이게 되는 스택값의 정보들을 확인할 수 있을 것이다. 이를 통해 로컬변수가 EBP 레지스터의 각 서브루틴들의 처음 포인터부터 가지게 됨을 확인할 수 있다.

do_page_fault 함수부에서 show_stack 을 이용하여 ESP 레지스터의 정보값을 저장하게 되는 sp 변수를 스택 프레임을 구성하게 되는 구조에 의하여 fault 난 주소값을 찾을 수 있다. 이에 대해 새로운 주소값을 할당해 줌으로써 해결할 수 있다.

4. 결론

리눅스 운영체제에서 panic 이 발생 하였을 경우 커널 내에서 치명적인 에러를 호출하게 된다. 이러한 상황에 좀더 효율적으로 대처할 수 있는 방안이 kernel hardening 이다.

본 논문에서는 안정적인 커널 모델로 동작하게 하기 위해 " panic" 이 발생하게 되는 특정 조건에서 복구할 수 있는 방안을 제시한다. 실험한 과정들은 kernel hardening 에서 리눅스 시스템에 다양한 fault 의 경우를 삽입하여 panic handler 를 더욱 더 강화할 수 있는 방법에서 안정된 커널 소스로의 수정으로 fault 를 예방하기 위한 설계이다. 부팅시에 리눅스 커널은 시스템의 모든 리소스를 감지하는데 fault 응답시 리눅스 커널은 안정적으로 동작할 수 있도록 설계 되어야 한다. 실험 대상은 커널 메모리 영역에서 fault 가 발생할 가능성을 가지고 있다 가정하여 실험을 하였으며, 기본적으로 커널 소스를 통하여 fault 가 발생하는 경로를 분석하여 문제를 해결할 수 있을 것이다.

참고문헌

[1] IA-32 Intel® Software Developer's Manual, Volume 1: Basic Architecture, charter 6. Procedure Calls, Interrupts,and exceptions , 1995

[2] Understanding the Linux Kernel, BOVET & CESATI, OREILLY, p216~p222, "Page Fault Exception Handler", 2001

[3] 한지붕 두커널-kernel compile http://kldp.org/KoreanDoc/html/kernel_Compile-KLDP/Kernel_Compile-KLDP.html

[4] SEQUOIA "Kernel Hardening Guidelines", Tim Udall, 1994

[5] Using Assembly Language in Linux, Phillip, 2001

[6] Introductino to Linux Intel Assembly Language, Norman Matloff, 2002

[7] Inline assembly for x86 in Linux http://www-106.ibm.com/developerworks/library/lia.html

[8] Intel 80386 Protected Mode , 이호, 2000

[9] Intel Dveloper Forum HAN-S170 : Hardened Linux that Enables Fault Management, Glenn Seiler, 2001

[10] Carrier Grade Linux on IA, Intel Developer Forum Spring, 2002

[11] Montavista™ Linux® Carrier Grade Edition[WHITE PAPER] ,Montavista Software Inc. John Mehaffey, April 8, 2002

[12] Linux Kernel Internals, Michael Beck,Mirko Dziadzka, Ulrich Kunitz and Harald Bohme,Addison-Wesley,1997

[13] Intel® Architecture Optimazation (Reference Manual) Intel, Intel Corporation, Appendix E, 1998

[14] Operating System Concepts(6th), SILBERSCHATZ&GALVIN&GAGNE, JOHNWILEY&SONGS INC. , 2002

[15] Linux programming bible, 권수호, 글로벌, 2002

[16] Linux Assembly Language Progaramming, BOB NEVELN, PHPTR, 2000

[17] Kernel Projects for Linux,Gary Nutt, Addison Wesley Longman,2001

[18] Linux Device Driver(2nd), A.Rubini&J.Corbet, O'Relly , 2001