[MOOC] Secure Coding in C

Chapter 1. About Secure Coding

● 시큐어코딩

1. 개념

안전한 소프트웨어 개발을 위해 소스 코드 등에 존재할 수 있는 잠재적인 보안 취약점을 제거하고 보안을 고려하여 기능을 설계 및 구현하는 등 소프트웨어 개발 과정에서 지켜야 할 일련의 보안 활동을 의미함

2. 필요성

- 릴리즈 이전 SW 취약점을 50% 줄이면, 침해사고 대응 비용 75% 감소 - Gartner
- 보안 취약점의 92%는 네트워크가 아닌 애플리케이션에서 발견 - NIST
- 릴리즈 이후 오류 수정은 약 $30,000 의 비용 소요, 하지만 개발 중 오류 수정은 약 $5,000 면 충분 - NIST
- 릴리즈 이후 오류를 수정하고자 할 경우 설계 단계보다 비용 100 배 증가 - IBM

이러한 이유로 미국의 경우, 국토안보부(DHS)를 중심으로 시큐어코딩을 포함한 SW 개발 전과정(설계・구현・시험 등)에 대한 보안활동 연구를 활발히 진행하고 있다. 국내의 경우 2009 년부터 전자정부서비스 개발단계에서 SW 보안약점을 진단하여 제거하는 시큐어코딩 관련 연구를 진행하면서 2012 년까지 전자정부지원사업 등을 대상으로 SW 보안약점 시범진단을 수행하였다. 또한 2012 년 년 6 월부터는 행정안전부 '정보시스템 구축・운영 지침(행안부고시 제 2012-25 호)'이 개정・고시 됨에 따라 전자정부서비스 개발시 적용토록 의무화되었다.

● 가이드라인

1. OWASP

- OWASP 는 The Open Web Application Security Project 의 약자로 국제 웹 보안 표준 기구를 의미
- 주로 웹에 관한 정보노출, 악성 파일 및 스크립트, 보안 취약점 등을 연구하는 기관으로 2004 년부터 2013 년까지 총 4 회에 걸쳐(2004, 2007, 2010, 2013) 10 대 웹 애플리케이션의 취약점을 발표함
- OWASP TOP 10 으로 불려지는 이 가이드는 특히 웹 애플리케이션 취약점 중에서 빈도가 많이 발생하고, 보안상 영향을 크게 줄 수 있는 점들을 주요로 다루고 있어 시큐어코딩의 중요한 기준으로 활용되고 있음

2. CERT

- CERT 는 Computer Emergency Response Team 의 약자로 1988 년 미국에서 발생한 해킹 사고 이후 결성된 기관임
- 컴퓨터 프로그래밍 언어별로 각 시큐어코딩 표준 가이드 제공하여 보다 안전한 프로그램을 개발할 수 있도록 하고 있음

3. CWE

CWE 는 Common Weakness Enumeration 의 약자로 프로그래밍에 있어 공통적으로 나타날 수 있는 취약점들을 표준화한 가이드라인

4. MISRA C

MISRA C 는 차량용 소프트웨어의 안전성을 높이기 위해 출판된 코딩 가이드라인으로 자동차 업체뿐만 아니라 철도, 의료 등 다양한 산업에서 적용되고 있음

이밖에도 다양한 가이드라인이 존재한다.

● 국내 사례

1. 소프트웨어 개발보안(시큐어 코딩) 관련 가이드

SW 개발단계부터 보안약점 제거(시큐어코딩) 의무화

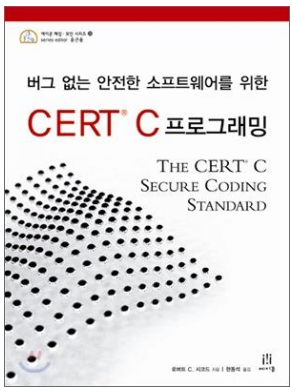
- 정보시스템 구축·운영 지침 개정안 행정예고 –

행정안전부는 사이버공격의 주요 원인인 소프트웨어 보안약점을 개발단계부터 제거하기 위해, '소프트웨어 개발보안(시큐어코딩)'을 의무화하는 「정보시스템 구축·운영 지침」개정안을 마련하여 행정예고 한다고 밝혔다.

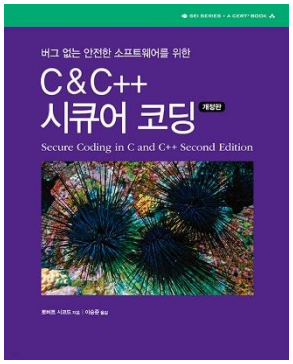
금번 개정안은 행정기관 및 공공기관이 정보화사업을 추진함에 있어 소프트웨어 개발보안을 적용·점검하기 위한 기준과 절차 등을 규정한 것으로 주요 내용은 다음과 같다. 우선, 금년 12 월부터(본 지침 고시후 6 개월 경과한 날부터) 행정기관 등에서 추진하는 개발비 40 억원 이상 정보화사업에 소프트웨어 개발보안 적용을 의무화하고, 단계적으로 의무 대상을 확대하여 '15 년에는 감리대상 전 정보화사업에 소프트웨어 개발보안을 적용한다. 다만, 상용소프트웨어는 소프트웨어 개발보안 적용대상에서 제외된다.

소프트웨어 개발사업자가 반드시 제거해야 할 보안약점은 SQL 삽입, 크로스사이트스크립트 등 43 개이다. 감리법인은 정보시스템 감리시 검사항목에 보안약점 제거여부를 반드시 포함하여야 한다. 감리법인은 효과적인 보안약점 진단을 위해 진단도구를 사용할 경우 국정원장이 인증한 보안약점 진단도구를 사용하여야 한다. (2014 년 1 월부터 적용)

참고 서적 :



로버트 C. 시코드 저 / 현동석 역 | 에이콘출판사 | 2010 년 02 월 16 일 | 원제 : The CERT C Secure Coding Standard



로버트 시코드 저 / 이승준 역 | 에이콘출판사 | 2015 년 01 월 09 일 | 원서 : Secure Coding in C and C++

Chapter 2. Preprocessor

PRE-01. 매크로에서는 매개 변수에 괄호를 사용하라

부수 효과를 줄이려면 반드시 매개 변수에 괄호를 사용해야 한다.

위험한 코드

- 아래의 함수형 매크로는 괄호를 사용하지 않아 문제가 된다.

```c
#include <stdio.h>

#define SQR(x)  x * x

int main() {
    int result = SQR(1 + 2);    // 1 + 2 * 1 + 2
    printf("result = %d\n", result);

    return 0;
}
```

- 해결 방법 - 매크로의 모든 매개변수에 괄호를 사용하면 된다.

```c
#include <stdio.h>

#define SQR(x)  (x) * (x)

int main() {
    int result = SQR(1 + 2);    // 1 + 2 * 1 + 2
    printf("result = %d\n", result);

    return 0;
}
```

PRE-02. 매크로로 치환될 영역은 반드시 괄호로 둘러싸야 한다

매크로로 치환될 영역을 괄호로 둘러싸면 근처의 표현식으로 인해 우선순위가 바뀌는 일을 방지할 수 있다.

위험한 코드 1.

- 의도하지 않은 결과가 나오게 된다.

```
#include <stdio.h>

#define DBL(x)  (x) + (x)

int main() {
    int result = DBL(2) * DBL(2);
    printf("result = %d\n", result);

    return 0;
}
```

- 해결 방법 - 매크로로 치환될 영역을 괄호로 둘러싼다.

```
#include <stdio.h>

#define DBL(x)  ((x) + (x))

int main() {
    int result = DBL(2) * DBL(2);
    printf("result = %d\n", result);

    return 0;
}
```

위험한 코드 2

- 치환된 코드가 수식으로 평가되어 정상적으로 동작하지 않는다.

```c
#include <stdio.h>

#define INFO    -1
#define WARN    -2
#define ERR     -3

void log_print(int level, const char *msg) {
    if (level INFO)
        printf("   [INFO] %s\n", msg);
    else if (level == WARN)
        printf("[WARNING] %s\n", msg);
    else if (level == ERR)
        printf("  [ERROR] %s\n", msg);
}

int main() {
    log_print(ERR, "out of service");
    return 0;
}
```

- 해결 방법 - 치환될 영역은 반드시 괄호로 둘러싸거나 열거 타입의 상수로 치환한다.

```c
#include <stdio.h>

#define INFO    (-1)
#define WARN    (-2)
#define ERR     (-3)

void log_print(int level, const char *msg) {
    if (level INFO)
        printf("   [INFO] %s\n", msg);
    else if (level == WARN)
        printf("[WARNING] %s\n", msg);
    else if (level == ERR)
        printf("  [ERROR] %s\n", msg);
}

int main() {
    log_print(ERR, "out of service");
    return 0;
}
```

PRE-03. 함수형의 매크로보다는 인라인이나 정적 함수를 사용하라

일반 함수는 함수 호출의 오버헤드가 존재한다. 때문에 간단한 코드에 대한 함수 호출은 성능 상의 이슈가 발생할 수 있다. 그래서 전통적인 프로그래머들은 간단한 코드에 대해 일반 함수가 아닌 함수형 매크로를 사용하기도 한다. 그러나 매크로는 컴파일러에 의한 평가가 아닌 단순 치환 구조이기 때문에 부수 효과가 발생할 수 있다.

인라인 함수란 함수 호출 코드가 함수의 기계어 코드로 치환되는 함수를 의미한다. 인라인 함수는 컴파일 타입에 처리되므로 함수 호출의 오버헤드가 없어진다. 그러므로 함수형 매크로보다는 인라인 함수를 사용하는 것이 좋다.

위험한 코드 1.

- 매크로 함수 호출 시 사용된 인자에 대한 증가, 감소, 메모리 변수 접근 등은 부수 효과를 발생시킬 수 있다.

```
#include <stdio.h>

#define SQR(x)  ((x) * (x))

int main() {
    int i = 2;
    printf("%d\n", SQR(++i));

    return 0;
}
```

- 해결 방법 - 인라인 함수를 사용하여 해결할 수 있다.

```
#include <stdio.h>

inline int sqr(int x) {
    return x * x;
}

int main() {
    int i = 2;
    printf("%d\n", sqr(++i));
```

```
    return 0;
}
```

위험한 코드 2

- 다음의 코드는 매크로를 사용해 전역 카운터를 증가시키는 코드이다. 하지만 매크로가 치환되면서 지역 카운터를 증가시키고 있다.

main.c

```
#include <stdio.h>

int cnt = 0;

#define CALL_FUNC(fp)  (++cnt, fp())

void print_cnt() {
        printf("cnt = %d\n", cnt);
}

int main() {
        int cnt = 0;
        CALL_FUNC(print_cnt);

        return 0;
}
```

- 해결 방법 - 인라인 함수를 사용한다. 인라인 함수는 함수 구현부가 컴파일될 때 식별자를 전역 변수로 처리한다.

```
#include <stdio.h>

int cnt = 0;

inline call_func(void(*fp)()) {
        ++cnt;
        fp();
}

void print_cnt() {
        printf("cnt = %d\n", cnt);
}
```

```
int main() {
        int cnt = 0;
        call_func(print_cnt);

        return 0;
}
```

PRE-04. 타입 인코딩 시 매크로 정의 대신 타입 정의를 사용하라

타입 정의는 스코프(scope) 규칙이 적용되는 반면 매크로 정의는 적용되지 않는다.

위험한 코드

- 아래의 코드에서 name은 포인터로 선언되었지만 tel은 포인터로 선언되어 있지 않다.

```
#include <stdio.h>
#define cstring char *

int main() {
    cstring name, tel;

    name = "honggildong";
    tel = "010-000-0000";
    printf("name: %s, tel: %s\n", name, tel);

    return 0;
}
```

- 해결 방법 - 타입 정의를 사용한다.

```
#include <stdio.h>

typedef char * cstring;

int main() {
    cstring name, tel;

    name = "honggildong";
    tel = "010-000-0000";
    printf("name: %s, tel: %s\n", name, tel);

    return 0;
}
```

PRE-05. 토큰들을 연결하거나 문자열 변환을 할 때 매크로 치환을 고려하라

매크로 연산자

- ## : 매크로가 치환되는 과정에서 두 개의 토큰을 하나로 병합(concatencation)
- # : # 연산자 다음에 있는 토큰을 문자열화

위험한 코드

- 아래의 코드는 줄 번호가 의도한대로 출력되지 않는다.

```
#include <stdio.h>
#define TOSTR(x)    #x
#define LOG(msg)    printf("["__FILE__"("TOSTR(__LINE__)")] "msg)

int main() {
        LOG("hello, world");
        return 0;
}
```

- 해결 방법 - 매크로 인자를 치환한 다음에 문자열로 만들려면 두 단계의 매크로를 사용해야 한다.

```
#include <stdio.h>
#define _TOSTR(x)  #x
#define TOSTR(x)    _TOSTR(x)
#define LOG(msg)    printf("["__FILE__"("TOSTR(__LINE__)")] "msg)

int main() {
        LOG("hello, world");
        return 0;
}
```

PRE-06. 헤더 파일에 항상 인클루드 가드를 둬라

소프트웨어 개발 프로젝트에서 헤더 파일의 중복으로 인한 문제가 일어나기 쉽다. 때문에 헤더 파일을 설계할 때는 반드시 인클루드 가드(include guard)를 사용해야 한다.

위험한 코드

- 다음 코드는 헤더 파일의 중복으로 인한 문제가 발생할 수 있다.

Math.h

```
typedef struct _Complex {
    double real, image;
} Complex;

#define PI  (3.14)
```

- 해결 방법 - 인클루드 가드를 적용한다.

Math.h

```
#ifndef _MATH_H_
#define _MATH_H_

typedef struct _Complex {
    double real, image;
} Complex;

#define PI  (3.14)

#endif
```

PRE-07. 복수 구문 매크로를 do-while 루프로 감싸라

여러 실행문을 그룹으로 만들어 연속적으로 실행하는 경우, 구문상 루프로 묶어줘야만 나중에 매크로가 if처럼 한 개의 실행문이나 중괄호로 묶인 실행 단위를 처리하는 부분에서 사용될 때 안전하게 동작한다.

위험한 코드

- 다음은 복수 구문으로 정의된 함수형 매크로를 사용할 경우, 의도하지 않은 결과를 초래한다

```
#include <stdio.h>

#define SWAP(x, y)  \
    int tmp = x;    \
    x = y;          \
    y = tmp;

int main() {
    int x, y;
    int tmp;

    scanf("%d%d", &x, &y);
    if (x != y)
        SWAP(x, y);

    printf("x = %d, y = %d\n", x, y);

    return 0;
}
```

- 괄호로 묶어도 되지만 제어문과 함께 사용되면 if 와 else의 짝을 잃게 된다.

```
#include <stdio.h>

#define SWAP(x, y)  \
    { int tmp = x;    \
      x = y;          \
      y = tmp; }
```

```
int main() {
        int x, y;
        int tmp;

        scanf("%d%d", &x, &y);
        if (x != y)
                SWAP(x, y);
        else
                printf("same\n");

        printf("x = %d, y = %d\n", x, y);

        return 0;
}
```

- 해결 방법 - do-while 루프로 감싼다.

```
#include <stdio.h>

#define SWAP(x, y)      \
    do {                \
        int tmp = x;    \
        x = y;          \
        y = tmp;        \
    } while (0)

int main() {
        int x, y;

        scanf("%d%d", &x, &y);
        if (x != y)
                SWAP(x, y);
        else
                printf("same\n");


        printf("x = %d, y = %d\n", x, y);
        return 0;
}
```

Chapter 3. Declaration

DCL-01. 변하지 않는 객체는 const로 보장해둬라

변하지 않는 객체는 const로 보장해야 한다. 객체의 불변성을 const를 사용해 보장하면 애플리케이션의 정확성과 안전성을 보장하는데 도움이 된다.

위험한 코드

- 다음의 코드에서 객체의 값이 의도하지 않게 변경되고 있다.

```
#include <stdio.h>

typedef struct {
    int x, y;
} Point;

void print_point(Point* p) {
    printf("x = %d, y = %d\n", p->x, p->y);
    p->x = -1;
}

int main() {
    Point p = { 0, };
    print_point(&p);

    return 0;
}
```

- 해결 방법 – const 키워드를 사용하면 이를 해결할 수 있다.

```
#include <stdio.h>

typedef struct {
    int x, y;
} Point;

void print_point(const Point* p) {
```

```
    printf("x = %d, y = %d\n", p->x, p->y);
    p->x = -1;
}

int main() {
    Point p = { 0, };
    print_point(&p);

    return 0;
}
```

DCL-02. 내부 스코프에서 변수 이름을 재사용하지 마라

변수의 이름을 재사용할 경우, 코드 상에서 혼란을 가중시키게 된다. 때문에 다음의 경우는 피해야 한다. 전역 변수가 사용될 수 있는 범위 내에서 어떤 변수든 중복해서 전역 변수 이름을 사용하면 안 된다.

어떤 블록 안에서 이미 사용되고 있는 변수와 동일한 이름으로 다른 블록에서 선언하면 안 된다.

위험한 코드

- 지역 변수가 전역 변수의 이름을 가리게 되므로 코드는 정상적으로 수행되지 않는다.

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

char msg[32];

void set_error(const char* error) {
    char msg[32];
    // ...
    strncpy(msg, error, sizeof(msg));
}

int get_fd() {
    // ...
    return -1;
}

int main() {
    int fd = get_fd();
    if (fd < 0)
        set_error("get_fd error");
    // ...

    printf("error message: %s\n", msg);
    return 0;
}
```

- 해결 방법 – 변수의 이름을 재사용하고 있다는 것은 변수의 이름 자체가 너무 일반적이라는 것을 의미하기 때문에 변수의 이름을 좀 더 설명적으로 정의한다.

```

```

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

char system_msg[32];

void set_error(const char* error) {
    char msg[32];
    // ...
    strncpy(system_msg, error, sizeof(system_msg));
}

int get_fd() {
    // ...
    return -1;
}

int main() {
    int fd = get_fd();
    if (fd < 0)
        set_error("get_fd error");
    // ...

    printf("error message: %s\n", system_msg);
    return 0;
}
```

DCL-03. 상수 수식의 값을 테스트할 때는 정적 어썰션(static assertion)을 사용해라

어썰션은 취약성이 될 수 있는 소프트웨어의 결점을 찾아 제거하는데 사용되는 효과적인 진단 도구이다. 다만 일반적인 어썰션(assert())를 사용하는 것은 몇 가지 제약이 존재하며 다음과 같다.

일반적인 어썰션 도구는 프로그램이 구동 중에 동작하므로 런타임 오버헤드(overhead)가 존재한다.

일반적인 어썰션 도구의 마지막 동작은 abort 함수를 호출하는 것이므로 서버 프로그램이나 임베디드 시스템에서는 사용하기 어렵다.

위험한 코드

- 다음 코드는 구조체의 패딩 비트를 검사를 하기 위해 assert 함수를 사용하고 있다. 진단은 런타임에만 일어나고 그것도 assert 함수가 포함된 코드가 실행될 때만 발견된다.

```
#include <stdio.h>
#include <assert.h>

typedef struct _Packet {
    char cmd;
    int  len;
} Packet;

int main() {
    assert(sizeof(Packet) == 3);

    Packet data;
    // ...

    return 0;
}
```

- 해결 방법 - 정적 어썰션을 구현하여 해결한다.

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

```c
#define JOIN_AGAIN(x, y)    x##y
#define JOIN(x, y)  JOIN_AGAIN(x, y)
#define static_assert(e)    \
    typedef char JOIN(assertion_failed_at_line, __LINE__) [(e) ? 1 : -1]

typedef struct _Packet {
    char cmd;
    int  len;
} Packet;

int main() {
    static_assert(sizeof(Packet) == 3);

    Packet data;
    // ...

    return 0;
}
```

DCL-04. 프로그램 로직 상의 고정적인 값을 나타낼 때는 의미 있는 심볼릭 상수를 사용하라

코드 상에서 리터럴(literal)을 사용할 경우, 가독성이 떨어질 수 있다. 때문에 가급적 리터럴을 직접 사용하기 보다는 심볼릭(symbolic) 상수를 통해 적절한 이름을 붙여 코드의 의도를 명확히 하는 것이 좋다. C 언어에서 심볼릭 상수를 만드는 방법은 다음과 같다.

- const로 지정된 객체
- 열거형 상수
- 객체형 매크로

const로 지정된 객체

const로 지정된 객체는 특정 스코프 내에서 사용 가능하며 컴파일러가 타입 체크를 해주며 디버깅 도구 사용 시 객체의 이름을 나타낼 수 있다. 대신 각 객체는 메모리를 사용하며 런타임에 약간의 오버헤드가 발생한다. 또한 아래의 경우처럼 컴파일 타임에서 정수형 상수가 필요한 곳에서는 사용할 수 없다.

구조체 내부의 비트 단위로 정의된 멤버의 크기

배열의 크기(가변 배열은 예외)

열거형 상수의 값

case 상수의 값

열거형 상수

열거형 상수는 정수로 나타낼 수 있는 정수형 상수 표현식을 나타낼 때 사용한다. const 키워드를 사용한 객체와 달리 메모리를 소모하지 않는다. 다만 열거형 상수는 값의 타입을 정의할 수 없으며 항상 정수(int)이다.

객체형 매크로

전처리 단계에서 사용되는 객체형 매크로(object-like macro)의 정의는 다음과 같다.

#define identifier replacement-list

객체형 매크로는 전처리기에 의해 치환되는 구조이므로 컴파일 과정에서는 매크로의 심볼을 볼 수 없다. 그래서 대부분의 컴파일러들은 매크로 이름들을 따로 저장하고 디버거에 전달하기도 한다.

매크로는 다음 이름으로 적용될 수 있는 스코프 규칙들을 고려하지 않았기 때문에 의도하지 않는 방식으로 치환되어 기대하지 않은 결과를 만들기도 한다. 그리고 객체형 매크로는 메모리를 소비하지 않으며 따라서 포인터로 가리킬 수도 없다. C 프로그래머들은 심볼릭 상수로 객체형 매크로를 사용하는 편이다.

정리하면 다음과 같다.

방식	평가 시점	메모리 소비	디버깅 시의 심볼	타입 체크	컴파일 타임 상수 표현식
열거형	컴파일 타임	없음	있음	있음	있음
const 지정	런타임	있음	있음	있음	없음
매크로	전처리	없음	없음	없음	있음

위험한 코드 1.

- 정수 리터럴에 대한 의미가 분명하지 않다.

```
void draw_color(int color) {
    switch (color) {
    case 0: /* ... */ break;
    case 1: /* ... */ break;
    case 2: /* ... */ break;
    }
}

int main() {
    // ...
    draw_color(0);

    return 0;
}
```

```

```

- 해결 방법 - 심볼릭 상수로 변경하여 의미를 분명하게 한다.

```
enum { RED, GREEN, BLUE };
void draw_color(int color) {
    switch (color) {
    case RED: /* ... */ break;
    case GREEN: /* ... */ break;
    case BLUE: /* ... */ break;
    }
}

int main() {
    // ...
    draw_color(RED);

    return 0;
}
```

위험한 코드 2.

- 버퍼의 크기가 일치하지 않다 버퍼 오버플로가 발생할 수 있다.

```
#include <stdio.h>

int main() {
    char buff[16];

    fgets(buff, 32, stdin);
    printf("%s\n", buff);

    return 0;
}
```

- 해결 방법 1. 열거형을 사용하여 해결한다.

```
#include <stdio.h>

enum { BUFF_SIZE = 16 };
```

```c
int main() {
    char buff[BUFF_SIZE];

    fgets(buff, BUFF_SIZE, stdin);
    printf("%s\n", buff);

    return 0;
}
```

- 해결 방법 2. sizeof 연산자를 사용한다.

```c
#include <stdio.h>

int main() {
    char buff[16];

    fgets(buff, sizeof(buff), stdin);
    printf("%s\n", buff);

    return 0;
}
```

DCL-05. 함수 선언 시 적절한 타입 정보를 포함시켜라

함수 호출 전 반드시 함수의 선언 정보가 반드시 존재해야 한다. 이는 함수 선언 정보가 없을 경우, 컴파일러는 타입 정보를 정확하게 체크할 수 없기 때문이다. 표준 라이브러리의 함수를 사용할 때 함수 선언 정보를 삽입하는 방법은 적절한 헤더 파일을 삽입하는 것이다. 함수의 선언 정보가 없는 상태로 함수를 호출할 경우, 컴파일러는 경고를 내지만 에러는 발생하지 않는다.

위험한 코드 1.

- 함수의 선언 정보 없이 함수를 호출하면 잘못된 값이 함수의 인자로 전달될 수 있다.

```c
#include <stdio.h>

int main() {
    func(1, 2);
    return 0;
}

int func(int a, int b, int c) {
    printf("func(%d, %d, %d)\n", a, b, c);
    return 0;
}
```

- 해결 방법 - 함수의 선언 정보를 정확하게 기술한다.

```c
#include <stdio.h>

int func(int a, int b, int c);

int main() {
    func(1, 2, 3);
    return 0;
}

int func(int a, int b, int c) {
    printf("func(%d, %d, %d)\n", a, b, c);
    return 0;
}
```

위험한 코드 2.

- 잘못된 함수 포인터를 선언해서 사용하면 의도하지 않는 결과가 나타날 수 있다.

```c
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int main() {
    int(*fp)(int);

    fp = add;
    printf("%d\n", fp(1));

    return 0;
}
```

- 해결 방법 - 함수 포인터를 정확하게 정의한다.

```c
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int main() {
    int(*fp)(int, int);

    fp = add;
    printf("%d\n", fp(1, 1));

    return 0;
}
```

DCL-06. 가변 인자를 가진 함수에서는 함수 작성자와 함수 사용자 간의 약속이 지켜져야 한다.

가변 인자 함수는 여러 개의 인자를 취하는데 문제의 소지가 있다. 때문에 가변 인자 함수의 정확한 사용 방법을 인지하여 의도하지 않는 결과가 발생되지 않도록 해야 한다.

위험한 코드

- 아래의 함수에서 마지막 인자로 VA_END 값을 넘겨주지 않을 경우, 코드는 정상적으로 동작하지 않을 수 있다.

```
#include <stdio.h>
#include <stdarg.h>

enum { VA_END = -1 };
int average(int first, ...) {
    int sum = 0;
    int cnt = 0;

    va_list args;
    va_start(args, first);

    int i = first;
    while (i != VA_END) {
        sum += i;
        ++cnt;
        i = va_arg(args, int);
    }
    va_end(args);

    return cnt ? sum / cnt : 0;
}

int main() {
    int avg = average(100, 100, 100, 100);
    printf("%d\n", avg);

    return 0;
}
```

- 해결 방법 1. 마지막 인자로 VA_END를 넣어 해결한다.

```c
#include <stdio.h>
#include <stdarg.h>

enum { VA_END = -1 };
int average(int first, ...) {
    int sum = 0;
    int cnt = 0;

    va_list args;
    va_start(args, first);

    int i = first;
    while (i != VA_END) {
        sum += i;
        ++cnt;
        i = va_arg(args, int);
    }
    va_end(args);

    return cnt ? sum / cnt : 0;
}

int main() {
    int avg = average(100, 100, 100, 100, VA_END);
    printf("%d\n", avg);

    return 0;
}
```

- 해결 방법 2. average 함수를 다시 설계한다.

```c
#include <stdio.h>
#include <stdarg.h>

int average(int cnt, ...) {
    if (cnt == 0)
        return 0;

    va_list args;
    va_start(args, cnt);

    int sum = 0;
    for (int i = 0; i < cnt; i++)
        sum += va_arg(args, int);
    va_end(args);

    return sum / cnt;
}
```

```c
int main() {
    int avg = average(4, 100, 100, 100, 100);
    printf("%d\n", avg);

    return 0;
}
```

DCL-07. 함수의 의해 바뀌지 않을 값에 대한 포인터를 함수의 매개 변수로 사용할 때는 const로 정의하라.

포인터를 매개 변수로 하는 함수는 내부적으로 대상체의 값을 변경하는 위험이 존재한다. 때문에 의도하지 않는 변경을 막으려면 const 키워드를 사용해야 한다.

위험한 코드

- 다음의 코드는 인자로 전달된 배열의 값을 임의로 변경하고 있다.

```c
#include <stdio.h>

int sum_arr(int *arr, int cnt) {
    int sum = 0;
    for (int i = 0; i < cnt; i++)
        sum += arr[i];
    *arr = 0;
    return sum;
}

int main() {
    int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };
    printf("%d\n", sum_arr(arr, 10));

    return 0;
}
```

- 해결 방법 - 상수 객체를 가리키는 포인터로 변경한다.

```c
#include <stdio.h>

int sum_arr(const int *arr, int cnt) {
    int sum = 0;
    for (int i = 0; i < cnt; i++)
        sum += arr[i];
    *arr = 0;
    return sum;
}

int main() {
    int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };
```

```
    printf("%d\n", sum_arr(arr, 10));

    return 0;
}
```

DCL-08. 객체를 선언할 때 적절한 지속공간을 지정하라.

객체가 자신의 수명을 다한 후에도 참조된다면 정의되지 않은 행동을 유발할 수 있다. 예를 들어 수명을 다한 객체를 참조하는 포인터는 정의되지 않은 값을 갖게 된다. 수명을 다한 객체에 접근하는 것은 매우 위험하고 취약성을 만드는 결과를 초래하기도 한다.

위험한 코드 1.

- 수명을 다한 지역 객체를 전역 변수가 참조함으로 임의의 코드를 수행할 수 있는 위험성이 존재한다.

```
#include <stdio.h>
const char* path;

void open_path() {
    const char str[] = "c:\\a.txt";
    path = str;
}

void hack() {
    const char str[] = "d:\\a.exe";
    printf("path = %s\n", path);
}

int main() {
    open_path();
    hack();

    printf("path = %s\n", path);
    return 0;
}
```

- 해결 방법 - 전역 포인터가 유효한 객체를 가리키도록 한다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
const char* path;

void open_path() {
    const char str[] = "c:\\a.txt";
```

```
    path = calloc(1, strlen(str) + 1);
    strncpy(path, str, strlen(str) + 1);
}

void hack() {
    const char str[] = "d:\\a.exe";
    printf("path = %s\n", path);
}

int main() {
    open_path();
    hack();
    printf("path = %s\n", path);
    return 0;
}
```

위험한 코드 2.

- 다음 코드는 지역 객체의 주소를 반환하고 있다.

```
#include <stdio.h>

char* init_array() {
    char arr[10] = { 0, };
    return arr;
}

int main() {
    char* pArr = init_array();

    for (int i = 0; i < 10; i++)
        printf("%d ", pArr[i]);
    printf("\n");
    return 0;
}
```

- 해결 방법 - 초기화할 배열을 함수의 인자로 받아 처리한다.

```
#include <stdio.h>

void init_array(char arr[]) {
    for (int i = 0; i < 10; i++)
        arr[i] = 0;
```

```
}

int main() {
    char arr[10];
    init_array(arr);

    for (int i = 0; i < 10; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

DCL-09. 함수 인자에서 restrict로 지정된 소스 포인터와 목적 포인터가 동일한 객체를 참조하지 않게 하라

restrict 키워드는 오직 포인터에만 적용되는 키워드로 그 포인터가 데이터 객체에 접근할 수 있는 유일하고도 최초가 되는 수단임을 나타낸다. 즉 포인터가 restrict로 한정되면 그 포인터가 가리키는 데이터 블록은 그 포인터만이 접근이 가능하므로 컴파일러가 더 효율적으로 코드를 최적화할 수 있다.

그러나 restrict 지정자를 사용할 때는 포인터들이 서로 동일한 객체를 참조하지 말아야 한다. 함수의 두 포인터가 동일한 객체를 참조하는 경우 그 결과는 미정의 동작이다.

위험한 코드

- memcpy는 restrict 지정자를 사용한 함수로 동일한 객체의 주소를 인자로 전달하면 그 결과는 알 수 없다.

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

void print_arr(int* arr, int len) {
    for (int i = 0; i < len; i++)
        printf("%d ", arr[i]);
    getchar();
}

int main() {
    int cnt = 10;
    int arr[10] = { 0,1,2,3,4,5,6,7,8,9 };
    print_arr(arr, cnt);

    --cnt;
    memcpy(arr, arr + 1, sizeof(int)*cnt);
    print_arr(arr, cnt);

    return 0;
}
```

- 해결 방법 1. 메모리의 순서를 고려하지 않는다면 마지막 요소를 앞쪽에 복사한다.

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

void print_arr(int* arr, int len) {
    for (int i = 0; i < len; i++)
        printf("%d ", arr[i]);
    getchar();
}

int main() {
    int cnt = 10;
    int arr[10] = { 0,1,2,3,4,5,6,7,8,9 };
    print_arr(arr, cnt);

    arr[0] = arr[--cnt];
    print_arr(arr, cnt);

    return 0;
}
```

- 해결 방법 2. memmove 함수를 사용하여 처리한다. memmove 함수는 내부적으로 버퍼를 가지고 있어 안전하게 복사가 가능하다.

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

void print_arr(int* arr, int len) {
    for (int i = 0; i < len; i++)
        printf("%d ", arr[i]);
    getchar();
}

int main() {
    int cnt = 10;
    int arr[10] = { 0,1,2,3,4,5,6,7,8,9 };
    print_arr(arr, cnt);

    --cnt;
    memmove(arr, arr + 1, sizeof(int)*cnt);
    print_arr(arr, cnt);

    return 0;
}
```

DCL-10. 캐시되어서는 안 되는 데이터에는 volatile 을 사용하라

일반적으로 CPU는 성능 상의 이유로 데이터를 메모리에서 직접 읽어오지 않고 캐시를 사용하여 읽어온다. 프로그램이 아닌 하드웨어 의해 변경되는 데이터는 캐시에 반영되지 않으므로 메모리에서 직접 읽어와야 한다. volatile는 데이터가 프로그램이 아닌 외부적인 요인에 의해 그 값이 변경될 수 있다고 컴파일러에 알려 캐싱을 제한할 때 사용한다.

위험한 코드

- 최적화에 의해 flag가 캐시되었다면 SIGINT를 받아도 종료되지 않는다.

```
#include <stdio.h>
#include <windows.h>
#include <signal.h>

int flag;

void handler(int signum) {
    printf("\thandler\n");
    flag = 0;
}

int main() {
    flag = 1;
    signal(SIGINT, handler);

    while (flag) {
        printf("do something\n");
        Sleep(1000);
    }

    return 0;
}
```

- 해결 방법 - 변수를 volatile로 선언한다.

```
#include <stdio.h>
#include <windows.h>
#include <signal.h>

volatile int flag;
```

```c
void handler(int signum) {
    printf("\thandler\n");
    flag = 0;
}

int main() {
    flag = 1;
    signal(SIGINT, handler);

    while (flag) {
        printf("do something\n");
        Sleep(1000);
    }

    return 0;
}
```

DCL-11. 함수 정의와 맞지 않는 타입으로 함수를 변환하지 마라

원래의 타입과 다른 타입의 함수를 참조하도록 함수 포인터를 사용하면 정의되지 않은 결과를 초래한다. 때문에 함수 포인터를 사용할 경우, 정확한 타입을 선언해서 사용해야 한다. C99에서는 다음과 같이 언급하고 있다.

특정 타입에 대한 함수의 포인터는 다른 타입의 함수 포인터로 변환될 수 있고 결과는 원래의 포인터와 같아진다. 호환되지 않는 타입의 함수를 호출하는데 함수 포인터가 사용되면 알 수 없는 행동을 초래한다.

위험한 코드

- 함수 포인터와 함수의 타입이 일치하지 않아 정의되지 않은 결과를 초래한다.

```c
#include <stdio.h>
#include <signal.h>

int square(int a) { return a * a; }
int cube(int a) { return a * a * a; }
int add(int a, int b) { return a + b; }

int main() {
    int(*fp)(int);
    // ...

    fp = add;
    printf("%d\n", fp(1));

    return 0;
}
```

- 해결 방법 - 정확한 타입의 포인터 변수를 선언한다.

```c
#include <stdio.h>
#include <signal.h>

int square(int a) { return a * a; }
int cube(int a) { return a * a * a; }
int add(int a, int b) { return a + b; }

int main() {
```

```
    int(*fp)(int, int);
    // ...

    fp = add;
    printf("%d\n", fp(1, 1));

    return 0;
}
```

Chapter 4. Expression

EXP-01. 연산자 우선순위를 나타내는데 괄호를 사용하라

괄호를 적절하게 사용하면 우선순위 때문에 발생하는 실수를 피할 수 있으며 방어적으로 에러를 줄일 수 있고 코드 가독성도 높아진다.

위험한 코드 1.

- 연산자 우선순위를 잘못 이해하여 코드가 의도대로 실행되지 않는다.

```c
#include <stdio.h>

int is_even(int x) {
    return x & 1 == 0 ? 1 : 0;
}

int main() {
    int n = 2;

    if (is_even(n))
        printf("even\n");
    else
        printf("odd\n");

    return 0;
}
```

- 해결 방법 - 표현식이 의도대로 평가되도록 괄호를 사용한다.

```c
#include <stdio.h>

int is_even(int x) {
    return (x & 1) == 0 ? 1 : 0;
}

int main() {
    int n = 2;
```

```
    if (is_even(n))
        printf("even\n");
    else
        printf("odd\n");

    return 0;
}
```

위험한 코드 2.

- 연산자 우선순위를 잘못 이해하여 코드가 의도대로 실행되지 않는다.

```
#include <stdio.h>

void incr(int* p) {
    *p++;
}

int main() {
    int cnt = 0;

    incr(&cnt);
    printf("cnt = %d\n", cnt);

    return 0;
}
```

- 해결 방법 - 표현식이 의도대로 평가되도록 괄호를 사용한다.

```
#include <stdio.h>

void incr(int* p) {
    (*p)++;
}

int main() {
    int cnt = 0;

    incr(&cnt);
    printf("cnt = %d\n", cnt);

    return 0;
}
```

EXP-02. 논리 연산자 AND와 OR의 단축 평가 방식을 알고 있어라.

논리 연산자 AND와 OR은 단축 평가를 수행한다. 즉, 첫 번째 피연산자로 평가가 완료되면 두 번째 피연산자는 평가하지 않는다.

연산자	첫 번째 피 연산자	두 번째 피 연산자
AND(&&)	거짓	평가 안함
OR(||)	참	평가 안함

위험한 코드

- 다음 코드는 배열에서 0이 몇 개인지를 세는 코드이다. 그러나 AND 연산자의 첫 번째 피 연산자가 거짓이므로 두 번째 피 연산자는 평가되지 않는다.

```c
#include <stdio.h>
#define SIZE_MAX    (5)

int main() {
    int arr[SIZE_MAX] = { 0,1,2,3,0 };

    int i = 0;
    int cnt = 0;
    while ((arr[i] == 0) && (++i < SIZE_MAX))
        ++cnt;
    printf("%d\n", cnt);

    return 0;
}
```

- 해결 방법 - 단축 평가가 이루어지지 않도록 한다.

```c
#include <stdio.h>
#define SIZE_MAX    (5)

int main() {
    int arr[SIZE_MAX] = { 0,2,3,4,0 };

    int cnt = 0;
    for (int i = 0; i < SIZE_MAX; i++) {
        if (arr[i] == 0)
            ++cnt;
```

```
    }
    printf("%d\n", cnt);

    return 0;
}
```

EXP-03. 구조체의 크기가 구조체 멤버들 크기의 합이라고 가정하지 마라

구조체의 크기는 항상 멤버들 크기의 총합과 일치하지 않는다. C99 표준에 의하면 "구조체 객체에는 명명되지 않는 패딩(padding)이 들어갈 수 있으며, 앞쪽에는 위치하지 않는다"고 명시되어 있다. 이는 CPU가 메모리에 빠르게 접근할 수 있도록 하기 위함이다. 구조체의 패딩이 어떻게 포함될지는 컴파일러의 정의에 따른다.

위험한 코드

- 다음은 구조체의 크기가 멤버들의 총합과 같다고 가정하고 있다.

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

typedef struct _string {
    char arr[10];
    int len;
} string;

string* make_str(const char* str) {
    string* p = malloc(14);
    strcpy(p->arr, str);
    p->len = strlen(str);

    return p;
}

int main() {
    string* hello = make_str("hello");

    string buf;
    memcpy(&buf, hello, sizeof(string));

    free(hello);
    return 0;
}
```

- 해결 방법 - sizeof 연산자를 사용한다.

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

typedef struct _string {
    char arr[10];
    int len;
} string;

string* make_str(const char* str) {
    string* p = malloc(sizeof(string));
    strcpy(p->arr, str);
    p->len = strlen(str);

    return p;
}

int main() {
    string* hello = make_str("hello");

    string buf;
    memcpy(&buf, hello, sizeof(string));

    free(hello);
    return 0;
}
```

EXP-04. const를 캐스트로 없애지 마라

const로 선언된 객체를 비 상수 객체로 변환하여 사용할 경우, 의도하지 않은 결과가 나타날 수 있다.

위험한 코드

- strlen 함수 내부에서 인자로 전달된 문자열의 값을 변경할 수 있다.

```
#include <stdio.h>

int str_len(const char* s) {
    char* p = s;
    while (*p)
        ++p;
    return p - s;
}

int main() {
    char str[] = "hello";
    printf("%d\n", str_len(str));

    return 0;
}
```

- 해결 방법 - 비 상수 객체의 포인터로 캐스팅하지 않는다.

```
#include <stdio.h>

int str_len(const char* s) {
    const char* p = s;
    while (*p)
        ++p;
    return p - s;
}

int main() {
    char str[] = "hello";
    printf("%d\n", str_len(str));

    return 0;
}
```

EXP-05. 포인터 연산이 정확하게 수행되고 있는지 보장하라

포인터 연산을 수행할 때 포인터에 더해지는 값은 자동적으로 포인터가 가리키는 객체의 타입으로 조정된다. 때문에 포인터 연산이 어떻게 동작하는지 이해하고 있지 않는다면 심각한 에러나 버퍼 오버플로를 초래한다.

어떤 포인터 ptr과 임의의 정수 n이 있을 때, 포인터 연산은 내부적으로 다음과 같이 동작한다.

- `ptr + n == ptr + (sizeof(*ptr) * n)`

위험한 코드

- 다음 코드는 버퍼 오버플로가 발생한다.

```
#include <stdio.h>
#define BUF_SIZE    (10)

int main() {
    int buf[BUF_SIZE];

    int* cur = buf;
    while (cur < (buf + sizeof(buf)))
        *cur++ = 0;

    for (int i = 0; i < BUF_SIZE; i++)
        printf("buf[%d] = %d\n", i, buf[i]);

    return 0;
}
```

- 해결 방법 1. 포인터 연산 시, 정수의 크기만큼 계산될 수 있도록 char* 타입으로 캐스팅한다.

```
#include <stdio.h>
#define BUF_SIZE    (10)

int main() {
    int buf[BUF_SIZE];

    int* cur = buf;
    while (cur < ((char*)buf + sizeof(buf)))
        *cur++ = 0;
```

```
    for (int i = 0; i < BUF_SIZE; i++)
        printf("buf[%d] = %d\n", i, buf[i]);

    return 0;
}
```

- 해결 방법 2. 첨자 연산자를 사용한다.

```
#include <stdio.h>
#define BUF_SIZE    (10)

int main() {
    int buf[BUF_SIZE];

    for (int i = 0; i < BUF_SIZE; i++)
        buf[i] = 0;

    for (int i = 0; i < BUF_SIZE; i++)
        printf("buf[%d] = %d\n", i, buf[i]);

    return 0;
}
```

EXP-06. 타입이나 변수의 크기를 결정할 때는 sizeof를 사용하라

애플리케이션에서 타입 크기를 하드 코딩하지 않는 것이 좋다. 대부분 타입의 크기가 컴파일러마다 다르고 동일한 컴파일러 내에서도 버전에 따라 다를 수 있다.

위험한 코드

- 다음의 코드는 포인터와 정수를 4바이트로 가정하고 있다. 시스템에 따라 정수의 크기는 달라질 수 있으므로 의도하지 않는 결과를 초래할 수 있다.

```
#include <stdio.h>
#include <stdlib.h>
#define ARR_SIZE    (10)

int main() {
    int** pArr = (int**)malloc(4 * ARR_SIZE);

    for (int i = 0; i < ARR_SIZE; i++)
        pArr[i] = (int*)malloc(4 * ARR_SIZE);

    // ...

    for (int i = 0; i < ARR_SIZE; i++)
        free(pArr[i]);
    free(pArr);

    return 0;
}
```

- 해결 방법 - sizeof 연산자를 사용한다.

```
#include <stdio.h>
#include <stdlib.h>
#define ARR_SIZE    (10)

int main() {
    int** pArr = (int**)malloc(sizeof(int*) * ARR_SIZE);

    for (int i = 0; i < ARR_SIZE; i++)
        pArr[i] = (int*)malloc(sizeof(int) * ARR_SIZE);

    // ...
```

```
    for (int i = 0; i < ARR_SIZE; i++)
        free(pArr[i]);
    free(pArr);

    return 0;
}
```

EXP-07. 함수에 의해 반환되는 값을 무시하지 마라

일반적으로 함수는 반환 시 유용한 값을 전달하는데 대개 이 값은 함수가 작업을 성공적으로 수행했는지 혹은 에러가 발생했는지를 확인하는데 사용한다.

위험한 코드

- 다음 코드는 정수를 입력 받기 위해 scanf 함수를 사용하지만 에러가 발생하는지를 체크하지 않는다.

```c
#include <stdio.h>

int main() {
    int num;

    printf("input number: ");
    scanf("%d", &num);
    printf("-> %d\n", num);

    return 0;
}
```

- 해결 방법 - 입력 에러가 발생했는지 체크한다.

```c
#include <stdio.h>

int main() {
    int num;

    printf("input number: ");
    int ret = scanf("%d", &num);
    if (ret == 0 || ret == EOF)
        printf("input error\n");
    else
        printf("-> %d\n", num);

    return 0;
}
```

EXP-08. 어썰션의 부수 효과를 피하라

assert와 함께 사용되는 표현식은 부수 효과를 가지면 안 된다. 이는 assert 함수가 매크로이기 때문이고 매크로 함수 내에서 값의 할당, 증가, 감소, 메모리 변수의 접근, 함수 호출 등은 미정의 동작이다.

위험한 코드

- assert 함수에서 값을 증가시키고 있다.

```
#include <stdio.h>
#include <assert.h>

int process(int i) {
    assert(i++ > 0);
    printf("%d\n", i);
}

int main() {
    int num;

    printf("input size: ");
    int ret = scanf("%d", &num);
    if (ret != 1)
        printf("scanf error\n");
    else
        process(num);

    return 0;
}
```

- 해결 방법 - assert 함수에서 증가 연산자를 제거한다.

```
#include <stdio.h>
#include <assert.h>

int process(int i) {
    assert(i > 0);
    ++i;
    printf("%d\n", i);
}
```

```c
int main() {
    int num;

    printf("input size: ");
    int ret = scanf("%d", &num);
    if (ret != 1)
        printf("scanf error\n");
    else
        process(num);

    return 0;
}
```

EXP-09. 초기화되지 않는 메모리를 참조하지 마라

함수 내에 선언된 자동 변수(auto variable)는 초기화하지 않으면 쓰레기(garbage) 값으로 설정된다. C99 표준에서는 자동 변수에 대해 다음과 같이 정의하고 있다.

객체가 자동 변수 저장 공간에 있는 경우, 초기화되지 않았다면 변수의 값은 정의되어 있지 않다.

따라서 초기화되지 않는 변수를 사용하면 프로그램은 제대로 동작하지 않을 수 있다.

위험한 코드

- sum_to 함수 내에서 지역 변수 sum의 값을 제대로 초기화하지 않아 프로그램은 의도하지 않게 실행된다.

```c
#include <stdio.h>

int sum_to(int num) {
    int sum;
    for (int i = 1; i <= num; i++)
        sum += i;
    return sum;
}

int main() {
    int input;
    printf("input number: ");
    scanf("%d", &input);
    printf("sum 1 to %d: %d\n", input, sum_to(input));

    return 0;
}
```

- 해결 방법 - 지역 변수 sum을 적절한 값으로 초기화해준다.

```c
#include <stdio.h>

int sum_to(int num) {
    int sum = 0;
    for (int i = 1; i <= num; i++)
        sum += i;
    return sum;
}
```

```c
int main() {
    int input;
    printf("input number: ");
    scanf("%d", &input);
    printf("sum 1 to %d: %d\n", input, sum_to(input));

    return 0;
}
```

EXP-10. 널 포인터가 역참조 되지 않음을 보장하라

널 포인터를 역참조 하면 프로그램은 알 수 없는 상태가 되며 보통은 종료된다. 때문에 널 포인터는 역참조 하지 않는 것이 좋다.

위험한 코드

- malloc 함수 호출 후, 리턴 값을 조사하지 않고 있다. 여기서 만약 널 포인터가 반환될 경우, 프로그램은 비정상 종료된다.

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

int main() {
    int* pArr = malloc(-1);

    for (int i = 0; i < 10; i++)
        pArr[i] = 0;

    // ...

    free(pArr);
    return 0;
}
```

- 해결 방법 - malloc 함수의 반환 값을 조사한다.

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

int main() {
    int* pArr = malloc(-1);
    if (pArr == NULL) {
        printf("malloc error\n");
        exit(-1);
    }

    for (int i = 0; i < 10; i++)
        pArr[i] = 0;

    // ...
```

```
    free(pArr);
    return 0;
}
```

EXP-11. 함수의 반환 값을 인접한 다음 시퀀스 포인터에서 접근하거나 수정하지 마라

C99에서는 다음과 같이 언급하고 있다.

함수 호출 결과 값을 다음 시퀀스 포인트에서 수정하려고 한다면 정의되지 않은 결과를 얻게 한다.

C 함수는 배열을 반환할 수 없으나 배열을 가진 구조체나 공용체는 반환할 수 있다. 만약 함수 호출 후 얻은 반환 값에 배열이 있다면 그 배열은 표현식 내에서 접근되거나 수정되면 안 된다.

위험한 코드

- 다음의 코드는 함수 호출 후 얻은 구조체부터 배열을 얻어오려 하고 있다. 함수 반환 값의 수명은 다음 시퀀스 포인트 전에 끝나게 되므로 함수에 의해 반환된 구조체는 더 이상 유효하지 않거나 다른 값으로 변경될 가능성이 크다.

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

typedef struct {
    char buf[32];
} String;

String make_str(const char* s) {
    String str;
    strcpy(str.buf, s);
    return str;
}

int main() {
    printf("%s\n", make_str("hello").buf);
    return 0;
}
```

- 해결 방법 - 반환되는 구조체를 저장 후 사용한다.

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

```
typedef struct {
    char buf[32];
} String;

String make_str(const char* s) {
    String str;
    strcpy(str.buf, s);
    return str;
}

int main() {
    String str = make_str("hello");
    printf("%s\n", str.buf);

    return 0;
}
```

Chapter 5. Integer

보수 (Complement)

보수(Complement)는 어떤 수(기수)가 되기 위하여 보충하는 수를 의미

A + B = R에서 A는 R에 대한 B의 보수 또는 B는 R에 대한 A의 보수라고 함

2의 보수 (2's Complement)

2의 보수화 방식은 양의 정수를 1의 보수로 변경한 다음 1을 더함

-2를 2의 보수로 변경하는 방법

```
   0  0  0  0  0  0  1  0   -> 2
   ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓   반전
   1  1  1  1  1  1  0  1   -> -2(1의 보수)
+  0  0  0  0  0  0  0  1   -> 1을 더함
-------------------------
   1  1  1  1  1  1  1  0   -> -2(2의 보수)
```

현대의 모든 프로세서는 2의 보수화 방식을 사용

```c
#include <stdio.h>

typedef struct {
    char a : 3;
    char b : 2;
    char c : 1;
    unsigned char d : 2;
} BIT;

int main() {
    BIT bit = { .a = 4,.b = 3,.c = 1,.d = 3 };

    printf("bit.a = %d\n", bit.a);
    printf("bit.b = %d\n", bit.b);
    printf("bit.c = %d\n", bit.c);
    printf("bit.d = %d\n", bit.d);

    return 0;
}
```

2의 보수화 방식의 장점

- 최상위 비트의 성질이 그대로 유지됨(0이면 양수, 1이면 음수)
- 감산기가 필요 없으므로 비용이 감소함
- 0에 대한 중복 표현이 사라짐
- 연산 시 자리 올림을 계산하지 않아도 됨
- 가산기만을 이용하여 사칙연산이 가능함(곱셈은 덧셈의 연속이고 나눗셈은 뺄셈의 연속임)

INT-01. 구현 시 사용되는 데이터 모델을 이해하라

데이터 모델(data model)은 표준 데이터 타입에 할당되는 크기를 정의한다. 사용하는 아키텍처의 데이터 모델을 이해하는 것은 중요하다.

Data Type	iAPX86	IA-32	IA-64	SPARC-64	ARM-32	Alpha	64-bit Linux, FreeBSD, NetBSD, and OpenBSD
char	8	8	8	8	8	8	8
short	16	16	16	16	16	16	16
int	16	32	32	32	32	32	32
long	32	32	32	64	32	64	64
long long	N/A	64	64	64	64	64	64
Pointer	16/32	32	64	64	32	64	64

또한 주어진 타입에 대한 범위를 가정하기 보다는 limits.h 파일을 사용하는 것이 더 좋다. limits.h 파일 안에는 모든 일치하는 플랫폼에 대한 표준 정수 타입의 범위를 결정하는 매크로를 가지고 있으며 예는 다음과 같다.

- `UINT_MAX`: unsigned int가 가질 수 있는 최대값
- `LONG_MIN`: long int가 가질 수 있는 최소값

추가적으로 stdint.h 파일에는 특정한 데이터 모델에 종속되지 않고 사용할 수 있는 특정 크기로 제한된 타입들이 있으며 예는 다음과 같다.

- `int_least_32_t`: 플랫폼에서 지원하는 최소 32비트 이상을 가진 부호 있는 정수 타입
- `uint_fast16_t`: 최소 16비트 이상을 갖는 부호 없는 정수 타입 중 플랫폼에서 지원하는 가장 빠른 타입

위험한 코드 1.

- 다음 코드는 **`sizeof(int) == sizeof(long)`** 인 플랫폼에서는 정확하게 동작하지만 그렇지 않은 경우 버퍼 오버플로가 발생함

```c
#include <stdio.h>

int main() {
    int input;

    if (scanf("%ld", &input) < 1) {
        perror("scanf");
        return -1;
    }

    printf("%d\n", input);
    return 0;
}
```

- 해결 방법 - 사용하는 타입과 맞는 정확한 포맷을 사용

```c
#include <stdio.h>

int main() {
    int input;

    if (scanf("%d", &input) < 1) {
        perror("scanf");
        return -1;
    }

    printf("%d\n", input);
    return 0;
}
```

위험한 코드 2.

- 다음 코드에서는 unsigned int 값 두개를 곱할 때 모든 비트가 유지될 것이라고 가정하고 있다. 그러나 이 코드는 64비트 리눅스 플랫폼에서는 정상 동작하지만 윈도우즈 플랫폼에서는 실패할 수 있다.

```c
#include <stdio.h>

int main() {
    unsigned int a, b;
    unsigned long c;
```

```
    a = 1000000000;
    b = 10;
    c = (unsigned long)a * b;
    printf("c = %lu\n", c);

    return 0;
}
```

- 해결 방법 - 결과를 보존할 수 있는 큰 타입을 사용한다.

```
#include <stdio.h>
#include <stdint.h> // uintmax_t

int main() {
    unsigned int a, b;
    uintmax_t c;

    a = 1000000000;
    b = 10;
    c = (uintmax_t)a * b;
    printf("c = %llu\n", c);

    return 0;
}
```

INT-02. 정수 변환 규칙을 이해해라

정수 변환

- 명시적 변환 (explicit conversion) - 변환 연산자를 사용한 변환
- 묵시적 변환 (implicit conversion) - 특정 연산에 의한 변환

정수의 승계 (Integer Promotion)

- int보다 작은 정수 타입은 연산이 수행될 때, int나 unsigned int 타입으로 변환된다.
- 이는 오버플로로 인한 산술 연산 에러를 피하기 위함이다.

```c
#include <stdio.h>

int main() {
    char c1 = 100;
    char c2 = 3;
    char c3 = 4;

    char result = c1 * c2 / c3;
    printf("result = %d\n", result);

    return 0;
}
```

정수 변환 순위

- 비트 수가 많은 타입이 높은 순위를 갖는다.
- 각기 다른 두 부호 있는 정수는 순위가 다르다.
- 부호 있는 정수 타입의 순위는 자신보다 정밀도가 낮은 다른 어떤 부호 있는 정수보다 순위가 높다.
- long long int > long int > int > short int > signed char
- unsigned long long int > unsigned long int > unsigned int > unsigned short int > unsigned char

일반적인 산술 변환 규칙

- 이항 연산의 경우 두 피연산자는 같은 타입으로 변환된다.
- 두 개의 피연산자가 같은 타입이면 변환하지 않는다.
- 두 개의 피연산자가 같은 정수 타입이면 범위가 큰 타입으로 변환된다.
- 부호 없는 정수 타입의 피연산자가 다른 피연산자의 순위보다 크거나 같은 경우, 부호 있는 정수 타입의 피연산자는 부호 없는 정수 타입으로 변환된다.
- 부호 있는 정수 타입의 피연산자가 부호 없는 타입의 모든 값을 표현할 수 있는 경우, 부호 없는 정수 타입의 피연산자는 부호 있는 정수 타입으로 변환된다.
- 부호 있는 정수 타입의 피연산자가 부호 없는 타입의 모든 값을 표현할 수 없는 경우, 부호 있는 정수 타입의 피연산자는 동일 타입의 부호 없는 타입으로 두 피연산자 모두 변환된다.

위험한 코드 1

- 서로 다른 타입에 대하여 연산을 수행할 경우에는 주의해야 함

```
#include <stdio.h>

int main() {
    int si = -1;
    unsigned int ui = 1;

    if (ui > si)
        printf("true\n");
    else
        printf("false\n");

    return 0;
}
```

- 해결 방법 - 타입을 일치시킨다.

```
#include <stdio.h>

int main() {
```

```
    int si = -1;
    unsigned int ui = 1;

    if ((int)ui > si)
        printf("true\n");
    else
        printf("false\n");

    return 0;
}
```

위험한 코드 2

- 동적 메모리 할당 함수들은 size_t 타입을 사용하므로 음수를 사용할 경우, 할당할 수 없는 아주 큰 양수로 변환되어 할당이 실패할 수 있음

```
#include <stdlib.h>

int main() {
    int *pArr = (int*)malloc(-1);
    pArr[0] = 0;

    free(pArr);
    return 0;
}
```

- 해결 방법 - 음수를 사용하지 않거나 반환된 값을 조사해서 사용

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

int main() {
    int *pArr = (int*)malloc(-1);
    if (pArr == NULL) {
        perror("malloc");
        exit(-1);
    }
    pArr[0] = 0;

    free(pArr);
    return 0;
}
```

INT-03. 불분명한 소스에서 얻어지는 정수 값은 제한을 강제하라

신뢰할 수 없는 소스로부터 얻어지는 정수 값은 식별할 수 있는 상한 값과 하한 값이 있는지를 확인하기 위해 반드시 평가되어야 한다.

위험한 코드

- 메모리를 동적으로 할당하는 코드에서 크기 값을 검사하지 않을 경우, 메모리 할당이 실패할 수 있으며 에러 처리의 구현에 따라 서비스 거부(denial of service) 상태가 될 수 있음

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

char *create_buff(size_t len) {
    return malloc(sizeof(char) * len);
}

int main() {
    char buff[1024];

    printf("input string: ");
    scanf("%s", buff);

    int len = strlen(buff);
    char *str = create_buff(len + 1);
    strcpy(str, buff);

    printf("-> %s\n", str);
    free(str);

    return 0;
}
```

- 해결 방법 - 적용 가능한 값의 범위를 검사함으로써 해결 가능

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

```c
enum { MAX_STRING_LEN = 1024 };

char *create_buff(size_t len) {
    if (len == 0 || len > MAX_STRING_LEN)
        return NULL;
    return malloc(sizeof(char) * len);
}

int main() {
    char buff[MAX_STRING_LEN];
    printf("input string: ");
    scanf("%s", buff);

    int len = strlen(buff);
    char *str = create_buff(len + 1);
    if (str == NULL) {
        fprintf(stderr, "create_buff error\n");
        exit(-1);
    }
    strcpy(str, buff);

    printf("-> %s\n", str);
    free(str);

    return 0;
}
```

INT-04. 모든 가능한 입력을 처리할 수 없다면 문자 데이터 변환을 위해 입력 함수를 사용하지 마라

scanf, fscanf, vscanf, vfscanf 함수는 stdin 또는 다른 입력 스트림으로부터 문자열 데이터를 읽는데 사용한다. 이 함수들은 유효한 정수 값에 대해 잘 동작하지만 유효하지 않는 값에 대해서는 신뢰성 있는 처리를 하지 못한다.

위험한 코드

- 정수나 부동소수점 수를 입력 받기 위해 scanf 또는 fscanf 함수를 사용하는 경우, 미정의 동작이 발생할 가능성이 있음

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int data;
    printf("input integer: ");

    int ret = scanf("%d", &data);
    if (ret != 1 || ret == EOF) {
        perror("scanf");
        exit(-1);
    }
    printf("-> %d\n", data);

    return 0;
}
```

- 해결 방법 - 입력 문자열을 처리하는데 fgets 함수를 사용하고, 문자열을 정수로 변환하기 위해서 strtol 함수를 사용하는 것이 좋다. strtol 함수는 입력 값이 long 영역에서 유효한지를 점검하는 에러 체크를 제공한다.

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main() {
    char buff[32];
```

```
    printf("input integer: ");
    fgets(buff, sizeof(buff), stdin);

    errno = 0;
    char *end_ptr;
    long data = strtol(buff, &end_ptr, 10);
    if (errno == ERANGE) {
        perror("strtol");
        exit(-1);
    }
    else if (end_ptr == buff) {
        fprintf(stderr, "not valid numeric input\n");
        exit(-1);
    }
    else if (*end_ptr != '\n' && *end_ptr != '\0') {
        fprintf(stderr, "extra characters on input line\n");
        exit(-1);
    }

    printf("-> %ld\n", data);
    return 0;
}
```

INT-05. 문자열 토큰을 정수로 변환할 때는 strtol 함수나 다른 관련된 함수를 사용하라.

다음의 함수들은 다른 방법들보다 신뢰성 있는 에러 처리를 제공한다.

- strtol - long int 반환
- strtoll - long long int 반환
- strtoul - unsigned long int 반환
- strtoull - unsigned long long int 반환

위험한 코드 1

- atoi, atol, atoll 함수들은 다음에 대한 문제점들이 존재
- 에러 발생 시에 errno를 설정하지 않음
- 표시할 수 없는 결과 값을 얻은 경우 정의되지 않은 행동을 유발함
- 문자열에 정수 값이 없는 경우 0을 반환하기 때문에 0이 입력되어 정상적으로 해석된 값인지 그렇지 않은지 확인할 수 없음

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    char buff[32];
    printf("input integer: ");
    fgets(buff, sizeof(buff), stdin);

    int data = atoi(buff);
    printf("-> %d\n", data);

    return 0;
}
```

위험한 코드 2

- sscanf 함수는 atoi 함수와 동일한 문제를 내포하고 있음

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    char buff[32];
    printf("input integer: ");
    fgets(buff, sizeof(buff), stdin);

    int data;
    int ret = sscanf(buff, "%d", &data);
    if (ret == 0 || ret == EOF) {
        perror("sscanf");
        exit(-1);
    }

    printf("-> %d\n", data);
    return 0;
}
```

해결 방법

- strtol, strtoll, strtoul, strtoull 함수군들을 사용하는 것이 좋음

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main() {
    char buff[32];
    printf("input integer: ");
    fgets(buff, sizeof(buff), stdin);

    errno = 0;
    char *end_ptr;
    long data = strtol(buff, &end_ptr, 10);
    if (errno == ERANGE) {
        perror("strtol");
        exit(-1);
    }
    else if (end_ptr == buff) {
```

```
        fprintf(stderr, "not valid numeric input\n");
        exit(-1);
    }
    else if (*end_ptr != '\n' && *end_ptr != '\0') {
        fprintf(stderr, "extra characters on input line\n");
        exit(-1);
    }

    printf("-> %ld\n", data);
    return 0;
}
```

INT-06. 숫자 값에는 명시적으로 signed 또는 unsigned 값을 사용하라

char, signed char, unsigned char를 통틀어 문자 타입(character type)이라고 한다. 숫자 값을 저장할 경우, signed char나 unsigned char를 사용해야 문자 타입의 부호와 상관없이 호환 가능한 코드를 만드는 유일한 방법이다.

위험한 코드

- 문자 타입의 변수 c는 signed char또는 unsigned char 타입일 수 있다. 부호의 유무에 따라 값은 달라질 수 있다.

```
#include <stdio.h>

int main() {
    char c = 200;
    int i = 1000;

    printf("%d / %d = %d\n", i, c, i / c);
    return 0;
}
```

- 해결방법 - unsigned char로 선언하면 char의 부호와 상관없이 동작해 그 값을 예측할 수 있다.

```
#include <stdio.h>

int main() {
    unsigned char c = 200;
    int i = 1000;

    printf("%d / %d = %d\n", i, c, i / c);
    return 0;
}
```

INT-07. 열거형 상수가 유일한 값으로 매핑되도록 보장하라

C 언어의 열거형의 열거자는 정수형으로 매핑된다. 일반적으로 열거형의 각 열거자가 개별 값으로 대응된다고 생각하지만 서로 같은 갖는 명확하지 않은 에러가 종종 만들어지기도 한다.

위험한 코드

- 열거자의 값이 중복되면 모호성으로 인해 에러가 발생할 가능성이 높다.

```c
#include <stdio.h>

enum { DEBUG, WARN, ERROR = 0 };

int main() {
    printf("DEBUG = %d\n", DEBUG);
    printf(" WARN = %d\n", WARN);
    printf("  ERR = %d\n", ERROR);

    if (DEBUG == ERROR)
        printf("true\n");
    else
        printf("false\n");

    return 0;
}
```

- 해결 방법 열거형 선언 시, 아래의 방법을 따르도록 하는 것이 좋다.

  명시적 선언을 하지 않는다.

  첫 번째 열거자에 대해서만 값을 지정한다.

  모든 열거자에 대하여 명시적으로 값을 지정한다.

```c
#include <stdio.h>

enum { DEBUG, WARN, ERROR };
```

```c
int main() {
    printf("DEBUG = %d\n", DEBUG);
    printf(" WARN = %d\n", WARN);
    printf("  ERR = %d\n", ERROR);

    if (DEBUG == ERROR)
        printf("true\n");
    else
        printf("false\n");

    return 0;
}
```

INT-08. 표현식에서 signed, unsigned 표시가 없는 int 비트 필드의 타입을 가정하지 마라

비트 필드에서 정수 타입이 부호가 있는 정수 타입인지 아닌지는 구현마다 다르다. C99 표준에 따르면 "만일 int 타입으로 승계될 때, 원래 타입의 모든 값을 표현할 수 있다면 값은 int로 변환되며 그렇지 않은 경우 unsigned int로 변환된다"라고 되어있다.

위험한 코드

- 비트 필드에서 int 비트 필드를 지정하면 구현에 따라 그 값(-1 또는 255)이 달라지수 있다.

```c
#include <stdio.h>

struct {
    int a : 8;
} bits = { 255 };

int main() {
    printf("bits.a = %d\n", bits.a);
    return 0;
}
```

- 해결 방법 - 부호의 유무를 명시적으로 선언하면 의도가 명확해진다.

```c
#include <stdio.h>

struct {
    unsigned int a : 8;
} bits = { 255 };

int main() {
    printf("bits.a = %d\n", bits.a);
    return 0;
}
```

INT-09. 비트 연산자는 unsigned 피연산자에만 사용해라

비트 연산자(~, >>, <<, &, |, ^)는 signed 정수에 대한 비트 연산이 구현마다 다르게 정의되어 있기 때문에 unsigned 정수 피연산자에 대해서만 사용해야 한다.

위험한 코드

- 부호 있는 정수의 오른쪽 시프트 연산은 컴파일러에 따라 비어 있는 비트가 0 또는 부호 비트로 채워질 수 있다. 따라서 다음의 코드는 경우에 따라서 버퍼 오버플로가 발생할 수 있다.

```
#include <stdio.h>

int main() {
    int packet = 0x80000000;

    printf("%x", packet >> 24); // 0xFFFFFF80
    return 0;
}
```

- 해결 방법 - 부호 없는 정수를 사용하면 비어 있는 비트가 0으로 채워지게 되므로 오버플로가 발생하지 않는다.

```
#include <stdio.h>

int main() {
    unsigned int packet = 0x80000000;

    printf("%x", packet >> 24); // 0x80
    return 0;
}
```

INT-10. unsinged 정수 연산이 래핑되지 않도록 주의하라

C99 표준에 따르면 unsigned 피연산자를 사용한 계산은 결코 오버플로가 발생하지 않는다. 결과 값이 저장될 정수 타입으로 표현될 수 없는 경우 나머지 연산으로 값을 줄여(wrap around) 표현하기 때문이다. 이를 정수 래핑(wrapping)이라고 한다. 때문에 신뢰할 수 없는 소스로부터 얻어진 정수 값이 아래와 같은 곳에 사용된다면 절대 래핑을 허용해서는 안 된다.

- 배열의 인덱스
- 포인터 연산의 일부
- 루프 카운터
- 메모리 할당 함수의 인자
- 그 밖의 보안에 민감한 코드

위험한 코드 1

- 부호 없는 정수의 덧셈 연산은 정수 래핑이 일어날 수 있기 때문에 다음의 코드는 의도하지 않은 결과를 초래할 수 있다.

```c
#include <stdio.h>
#include <limits.h>

int main()
{
    unsigned int uint1, uint2, sum = 0;
    uint1 = UINT_MAX;
    uint2 = 1;

    sum = uint1 + uint2;
    printf("sum : %u\n", sum);

    return 0;
}
```

- 해결 방법 - 정수 래핑이 발생하기 전에 미리 테스트한다.

```c
#include <stdio.h>
#include <limits.h>

int main()
{
    unsigned int uint1, uint2, result = 0;
    uint1 = UINT_MAX;
    uint2 = 1;

    if (uint1 > UINT_MAX - uint2) {
        fprintf(stderr, "int wrapping!\n");
        exit(-1);
    }
    result = uint1 + uint2;
    printf("result = %u\n", result);

    return 0;
}
```

위험한 코드 2

- 부호 없는 정수의 뺄셈 연산은 정수 래핑이 일어날 수 있기 때문에 다음의 코드는 의도하지 않는 결과를 초래할 수 있다.

```c
#include <stdio.h>
#include <limits.h>

int main()
{
    unsigned int uint1, uint2, result = 0;
    uint1 = 0;
    uint2 = 1;

    result = uint1 - uint2;
    printf("result = %u\n", result);

    return 0;
}
```

- 해결 방법 - 정수 래핑이 발생하기 전에 미리 테스트한다.

```c
#include <stdio.h>
#include <limits.h>

int main()
{
    unsigned int uint1, uint2, result = 0;
    uint1 = 0;
    uint2 = 1;

    if (uint1 < uint2) {
        fprintf(stderr, "int wrapping!\n");
        exit(-1);
    }

    result = uint1 - uint2;
    printf("result = %u\n", result);

    return 0;
}
```

위험한 코드 3

- 부호 없는 정수의 곱셈 연산은 정수 래핑이 일어날 수 있기 때문에 다음의 코드는 의도하지 않는 결과를 초래할 수 있다.

```c
#include <stdio.h>
#include <limits.h>

int main() {
    unsigned int uint1, uint2, result = 0;
    uint1 = UINT_MAX;
    uint2 = 2;

    result = uint1 * uint2;
    printf("result = %u\n", result);

    return 0;
}
```

- 해결 방법 - 정수 래핑이 발생하기 전에 미리 테스트한다.

```c
#include <stdio.h>
```

```c
#include <limits.h>

int main() {
    unsigned int uint1, uint2, result = 0;
    uint1 = UINT_MAX;
    uint2 = 2;

    if (uint1 > UINT_MAX / uint2) {
        fprintf(stderr, "int wrapping!\n");
        exit(-1);
    }

    result = uint1 * uint2;
    printf("result = %u\n", result);

    return 0;
}
```

INT-11. signed 정수의 연산이 오버플로되지 않도록 보장하라

C 언어에서 정수 오버플로는 정의되지 않은 동작이다. 구현에 따라서는 오버플로를 감지할 수 있지만 그렇지 않은 경우도 있으므로 오버플로가 발생되지 않도록 하는 것이 중요하다. 특히 신뢰할 수 없는 소스로부터 얻어진 부호 있는 정수 값에 대한 연산이 다음과 같이 사용될 때 더욱 그렇다.

- 배열의 인덱스
- 포인터 연산
- 객체의 길이나 크기
- 배열의 경계
- 메모리 할당 함수의 인자
- 보안에 민감한 코드

위험한 코드 1

- 부호 있는 정수의 덧셈 또는 뺄셈 연산은 정수 오버플로가 일어날 수 있기 때문에 다음의 코드는 의도하지 않는 결과를 초래할 수 있다.

```c
#include <stdio.h>
#include <limits.h>

int main() {
    signed int sint1, sint2, result = 0;
    sint1 = INT_MAX;
    sint2 = 1;

    result = sint1 + sint2;
    printf("result = %d\n", result);

    return 0;
}
```

- 해결 방법 - 오버플로가 발생하기 전에 미리 테스트한다.

```c
#include <stdio.h>
#include <limits.h>

int main() {
    signed int sint1, sint2, result = 0;
    sint1 = INT_MAX;
    sint2 = 1;

    if (((sint1 > 0) && (sint2 > 0) && (sint1 > (INT_MAX - sint2))) ||
        ((sint1 < 0) && (sint2 < 0) && (sint1 < (INT_MIN - sint2)))) {
        fprintf(stderr, "int overflow!\n");
        exit(-1);
    }

    result = sint1 + sint2;
    printf("result = %d\n", result);

    return 0;
}
```

위험한 코드 2

- 부호 있는 정수의 곱셈 연산은 정수 오버플로가 일어날 수 있기 때문에 다음의 코드는 의도하지 않는 결과를 초래할 수 있다.

```c
#include <stdio.h>
#include <limits.h>

int main() {
    signed int sint1, sint2, result = 0;
    sint1 = INT_MAX;
    sint2 = 2;

    result = sint1 * sint2;
    printf("result = %d\n", result);

    return 0;
}
```

- 해결 방법 - 오버플로가 발생하기 전에 미리 테스트한다.

```c
#include <stdio.h>
```

```c
#include <limits.h>

int main() {
    signed int sint1, sint2, result = 0;
    sint1 = INT_MAX;
    sint2 = 2;

    if (sint1 > 0) {
        if (sint2 > 0) {    // 양수 * 양수
            if (sint1 > (INT_MAX / sint2)) {
                fprintf(stderr, "int overflow!\n");
                exit(-1);
            }
        }
        else {  // 양수 * 음수
            if (sint2 < (INT_MIN / sint1)) {
                fprintf(stderr, "int overflow!\n");
                exit(-1);

            }
        }
    }
    else {
        if (sint2 > 0) {    // 음수 * 양수
            if (sint1 < (INT_MIN / sint2)) {
                fprintf(stderr, "int overflow!\n");
                exit(-1);

            }
        }
        else {  // 음수 * 음수
            if ((sint1 != 0) && (sint2 < (INT_MAX / sint1))) {
                fprintf(stderr, "int overflow!\n");
                exit(-1);

            }
        }
    }

    result = sint1 * sint2;
    printf("result = %d\n", result);

    return 0;
}
```

위험한 코드 3

- 부호 있는 정수의 나눗셈 연산은 피제수가 최소값(INT_MIN)이고 제수가 -2인 경우,2의 보수 표기에서 정수 오버플로가 일어날 수 있다. 때문에 다음의 코드는 의도하지 않는 결과를 초래할 수 있다.

```c
#include <stdio.h>
#include <limits.h>

int main() {
    signed int sint1, sint2, result = 0;
    sint1 = INT_MIN;
    sint2 = -1;

    result = sint1 / sint2;
    printf("result = %d\n", result);

    return 0;
}
```

- 해결 방법 - 오버플로가 발생하기 전에 미리 테스트한다.

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

int main() {
    signed int sint1, sint2, result = 0;
    sint1 = INT_MIN;
    sint2 = -1;

    if ((sint2 == 0) || ((sint1 == INT_MIN) && (sint2 == -1))) {
        fprintf(stderr, "int overflow!\n");
        exit(-1);
    }

    result = sint1 / sint2;
    printf("result = %d\n", result);

    return 0;
}
```

INT-12. 음수나 피연산자의 비트보다 더 많은 비트를 시프트하지 마라

오른쪽 피연산자의 값이 음수 또는 승계된 왼쪽 피연산자의 값과 같거나 큰 경우 정의되지 않은 동작이 일어난다.

위험한 코드 1

- 오른쪽 피연산자의 값이 음수이므로 다음의 코드는 미정의 동작이다.

```
#include <stdio.h>
#include <limits.h>

int main() {
    signed char sint1, sint2, result = 0;
    sint1 = 1;
    sint2 = -1;

    result = sint1 << sint2;
    printf("result = %d\n", result);

    return 0;
}
```

- 해결 방법 - 오른쪽 피연산자가 음수인지 검사한다.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

int main() {
    signed char sint1, sint2, result = 0;
    sint1 = 1;
    sint2 = -1;

    if (sint2 < 0) {
        fprintf(stderr, "undefined behavior!\n");
        exit(-1);
    }

    result = sint1 << sint2;
```

```
    printf("result = %d\n", result);

    return 0;
}
```

위험한 코드 2

- 피연산자의 비트보다 더 많은 비트를 시프트하는 것은 미정의 동작이다.

```
#include <stdio.h>
#include <limits.h>

int main() {
    unsigned int uint1, uint2, result;
    uint1 = 1;
    uint2 = 32;

    result = uint1 << uint2;
    printf("result = %u\n", result);

    return 0;
}
```

- 해결 방법 - 시프트 연산을 수행하기 전에 검사를 수행한다.

```
#include <stdio.h>
#include <limits.h>

#define INT_BIT (sizeof(unsigned int) * CHAR_BIT)

int main() {
    unsigned int uint1, uint2, result;
    uint1 = 1;
    uint2 = 32;

    if (uint2 >= INT_BIT) {
        fprintf(stderr, "undefined behavior!\n");
        exit(-1);
    }

    result = uint1 << uint2;
    printf("result = %u\n", result);
```

```
    return 0;
}
```

Chapter 6. Float Point

FLP-01. 부동 소수점의 제한을 이해해라

C 프로그래밍 언어는 계산을 위해 부동소수점 수를 사용할 수 있도록 지원한다. C99에서는 부동소수점 수에 대한 요구사항을 플랫폼에서 어떻게 지원해야 하는지 명시하고 있지만, 부동소수점을 구현한 시스템들 간의 경쟁 때문에 특정 부동소수점 표현 방식을 보장하지 않는다.

아래의 코드는 시스템에 따라 다른 결과를 출력할 수 있다.

```c
#include <stdio.h>

int main() {
    float f = 1.0f / 3.0f;
    printf("Float is %.40f\n", f);

    return 0;
}
```

또한 아래의 코드는 컴파일러 최적화 레벨에 따라 다른 다른 결과를 출력할 수 있다.

```c
#include <stdio.h>

int main() {
    double a = 3.0;
    double b = 7.0;
    double c = a / b;

    if (c == (a / b))
        printf("same\n");
    else
        printf("not same\n");

    return 0;
}
```

따라서 각 시스템에 따라 부동소수점 수에 대한 제한을 잘 알고 있어야 데이터에 대한 무결성이 보장될 수 있다.

FLP-02. 정확한 계산이 필요할 때는 부동소수점 수를 배제할 수 있는지 고려하라

컴퓨터는 유한한 개수의 숫자를 표현할 수 있다. 따라서 1/3이나 1/5처럼 반복되는 이진 표기값을 정확하게 표현하는 것은 대부분의 부동소수점 표현으로는 불가능하다. 따라서 정확한 계산이 필요한 경우에는 값을 완벽하게 표현할 수 있는 다른 표현 방법을 생각하는 편이 좋다.

위험한 코드

```
#include <stdio.h>

float mean(float* arr, size_t len) {
    float total = 0.0f;
    for (int i = 0; i < len; i++) {
        total += arr[i];
        printf("arr[%d]: %f\ttotal = %f\n", i, arr[i], total);
    }

    if (len != 0)
        return total / len;
    else
        return 0.0F;
}

#define ARR_SIZE    (10)
int main() {
    float array[ARR_SIZE];
    for (int i = 0; i < ARR_SIZE; i++)
        array[i] = 10.1F;

    float total = mean(array, ARR_SIZE);
    printf("total = %.10f\n", total);
    return 0;

}
```

- 해결 방법 - 부동 소수점을 정수로 변경하여 해결한다.

```
#include <stdio.h>

float mean(float* arr, size_t len) {
```

```
    int total = 0;
    for (int i = 0; i < len; i++) {
        total += arr[i];
        printf("arr[%d]: %f\ttotal = %f\n", i, arr[i], (float)total);
    }

    if (len != 0)
        return (float)total / len;
    else
        return 0.0F;
}

#define ARR_SIZE    (10)
int main() {
    float array[ARR_SIZE];
    for (int i = 0; i < ARR_SIZE; i++)
        array[i] = 1010;

    float total = mean(array, ARR_SIZE);
    printf("total = %.10f\n", total / 100.0);
    return 0;

}
```

FLP-03. 부동소수점 변수를 루프 카운터로 사용하지 마라

큰 부동소수점 값에 증가 연산을 적용하면 경우에 따라서 가능한 정밀도의 한계 때문에 값이 전혀 변하지 않을 수 있다. 또한 플랫폼마다 정밀도의 한계 값이 다르므로 이식 가능한 코드를 구현하려면 부동소수점을 루프 카운터로 사용해선 안 된다.

위험한 코드

- 다음 코드는 경우에 따라서 9번만 실행될 수 있다.

```
#include <stdio.h>

int main() {
    int cnt = 0;
    for (float i = 0.1f; i <= 1.0f; i += 0.1f)
        ++cnt;

    printf("cnt = %d\n", cnt);
    return 0;
}
```

- 해결 방법 - 루프 카운터는 정수를 사용한다.

```
#include <stdio.h>

int main() {
    int cnt = 0;
    for (size_t i = 0; i < 10; i++)
        ++cnt;

    printf("cnt = %d\n", cnt);
    return 0;
}
```

FLP-04. 부동소수점 연산용 정수는 먼저 부동소수점으로 바꿔라

계산 시 정수를 사용해 부동소수점 변수에 값을 할당한느 경우 정보가 손실될 수 있다. 이 경우 표현식의 정수 중 하나를 부동소수점 타입으로 변환해 피할 수 있다.

위험한 코드

```
#include <stdio.h>

int main() {
    short a = 533;
    int b = 6789;
    long c = 466438237;

    float d = a / 7;    // d는 76.0
    printf("d : %f\n", d);

    double e = b / 30;  // e는 226.0
    printf("d : %f\n", e);

    double f = c * 789; // f는 오버플로우되어 음수일 수 있다.
    printf("d : %f\n", f);

    return 0;
}
```

- 해결 방법 1. 피연산자 하나를 부동소수점으로 표기한다.

```
#include <stdio.h>

int main() {
    short a = 533;
    int b = 6789;
    long c = 466438237;

    float d = a / 7.0f;
    printf("d : %f\n", d);

    double e = b / 30.0;
    printf("d : %f\n", e);

    double f = (double)c * 789;
```

```
    printf("d : %f\n", f);

    return 0;
}
```

- 해결 방법 2. 정수 값을 부동소수점 변수에 저장하여 처리한다.

```
#include <stdio.h>

int main() {
    short a = 533;
    int b = 6789;
    long c = 466438237;

    float d = a;
    d /= 7;
    printf("d : %f\n", d);

    double e = b;
    e /= b;
    printf("d : %f\n", e);

    double f = c;
    f *= 789;
    printf("d : %f\n", f);

    return 0;
}
```

FLP-05. 부동소수점 변환이 새로운 타입의 범위 안에 들어가는지 확인하라

부동소수점 값이 더 작은 범위나 정밀도를 가진 부동소수점 값으로 변환되거나 정수로 변환되는 경우, 혹은 정수가 부동소수점으로 변환되는 경우 값은 변환될 타입으로 표현 가능해야 한다.

위험한 코드

- 부동소수점의 정수 부분이 정수 타입의 크기를 넘어설 경우는 정의되어 있지 않다.

```c
#include <stdio.h>

int main() {
    float f = 10000000000.0f;
    int i = f;

    printf("f: %f\n", f);
    printf("i: %d\n", i);

    return 0;
}
```

- 해결 방법 - 부동소수점의 정수 값이 정수 타입의 범위 안에 있는지 조사한다.

```c
#include <stdio.h>
#include <limits.h>

int main() {
    float f = 10000000000.0f;
    int i;

    if (f > (float)INT_MAX || f < (float)(INT_MIN)) {
        printf("error\n");
        return -1;
    }

    i = f;

    printf("f: %f\n", f);
    printf("i: %d\n", i);
```

```
    return 0;
}
```

Chapter 7. Array

ARR-01. 배열의 크기를 얻을 때 포인터를 sizeof의 피연산자로 사용하지 마라.

sizeof는 피 연산자의 크기를 바이트 단위로 계산하는 연산자로 sizeof 연산자로 배열의 크기를 계산할 때는 주의해야 한다.

위험한 코드

- 배열이 전달된 함수 내에서 배열의 크기를 계산하고 있다.

```
#include <stdio.h>

void clear(int arr[]) {
    for (size_t i = 0; i < sizeof(arr) / sizeof(*arr); i++)
        arr[i] = 0;
}

int main() {
    int arr[5];

    clear(arr);
    for (size_t i = 0; i < sizeof(arr) / sizeof(*arr); i++)
        printf("arr[%u] = %d\n", i, arr[i]);

    return 0;
}
```

- 해결 방법 - 배열이 선언된 블록 안에서 크기를 계산한 후, 함수의 인자로 전달한다.

```
#include <stdio.h>

void clear(int arr[], int len) {
    for (size_t i = 0; i < len; i++)
        arr[i] = 0;
}
```

```c
int main() {
    int arr[5];
    size_t len = sizeof(arr) / sizeof(*arr);

    clear(arr, len);
    for (size_t i = 0; i < len; i++)
        printf("arr[%u] = %d\n", i, arr[i]);

    return 0;
}
```

ARR-02. 배열의 인덱스가 유효한 범위 안에 있음을 보장하라

배열의 참조가 배열의 경계 안에서 일어나게 하는 일은 전적으로 프로그래머의 책임이다.

위험한 코드

- insert_in_table 함수는 배열의 위쪽 경계를 넘지 않도록 보장하고 있지만 아래쪽 경계를 체크하고 있지 않다.

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

enum { TABLESIZE = 10 };
int* table = NULL;

int insert_in_table(int pos, int value) {
    if (!table) {
        table = (int *)malloc(sizeof(int) * TABLESIZE);
        if (table == NULL) {
            perror("malloc");
            exit(-1);
        }
    }

    if (pos >= TABLESIZE)
        return -1;

    table[pos] = value;
    return 0;
}

int main() {
    if (insert_in_table(-5, 100) < 0)
        printf("insert_in_table error\n");

    return 0;
}
```

- 해결 방법 - pos를 size_t로 선언해 음수의 전달을 막는다.

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

```c
enum { TABLESIZE = 10 };
int* table = NULL;

int insert_in_table(size_t pos, int value) {
    if (!table) {
        table = (int *)malloc(sizeof(int) * TABLESIZE);
        if (table == NULL) {
            perror("malloc");
            exit(-1);
        }
    }

    if (pos >= TABLESIZE)
        return -1;

    table[pos] = value;
    return 0;
}

int main() {
    if (insert_in_table(-5, 100) < 0)
        printf("insert_in_table error\n");

    return 0;
}
```

ARR-03. 충분한 크기의 공간에서 복사가 진행됨을 보장하라

모든 데이터를 담을 수 있을 만큼 크지 않은 배열에 데이터를 복사하면 버퍼 오퍼플로를 발생시킬 수 있다.

위험한 코드

- memcpy 함수처럼 크기를 지정해 제한된 복사를 수행하는 함수를 부적절하게 사용하면 버퍼 오버플로가 발생할 수 있다.

```
#include <stdio.h>
enum { BUF_SIZE = 1024 };

void func(const char src[], size_t len) {
    char dest[BUF_SIZE];
    memcpy(dest, src, len * sizeof(char));
    printf("%s\n", dest);
}

int main() {
    char str[] = "hello, world";
    func(str, strlen(str) + 1);

    return 0;
}
```

- 해결 방법 1. 배열의 경계를 체크한다.

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

enum { BUF_SIZE = 1024 };

void func(const char src[], size_t len) {
    if (len >= BUF_SIZE)
        return;

    char dest[BUF_SIZE];
    memcpy(dest, src, len * sizeof(char));
    printf("%s\n", dest);
}
```

```
int main() {
    char str[] = "hello, world";

    func(str, strlen(str) + 1);
    return 0;
}
```

- 해결 방법 2. 동적 메모리 할당을 수행한다.

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

void func(const char src[], size_t len) {
    char* dest = (char*)malloc(sizeof(char) * len);
    memcpy(dest, src, len * sizeof(char));
    printf("%s\n", dest);
    free(dest);
}

int main() {
    char str[] = "hello, world";

    func(str, strlen(str) + 1);
    return 0;
}
```

Chapter 8. String

STR-01. 문자열 상수를 가리키는 포인터는 const로 선언하라

문자열 리터럴은 상수이므로 const 지정자에 의해 보호되어야 한다.

위험한 코드

```c
#include <stdio.h>

int main() {
    char *str = "hello";
    str[0] = 'c';

    printf("%s\n", str);
    return 0;
}
```

- 해결 방법 - const로 지정한다.

```c
#include <stdio.h>

int main() {
    const char *str = "hello";
    str[0] = 'c';

    printf("%s\n", str);
    return 0;
}
```

STR-02. strtok()에서 파싱되는 문자열이 보존된다고 가정하지 마라

strtok 함수는 처음 호출되면 문자열내의 구분자가 처음 나타나는 부분까지 파싱하고, 구분자를 널 문자로 바꾼 후, 토큰의 처음 주소를 반환한다. 이후, 다시 strtok 함수를 호출하면 가장 최근에 널 문자로 바뀐 부분부터 파싱이 시작된다.

때문에 strtok 함수는 인자를 수정하므로 원본 문자열은 안전하지가 않으며 원본 문자열을 보존하고 싶다면 문자열의 복사본을 만들어 사용하도록 하여야 한다.

위험한 코드

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

int main() {
    char str[] = "hello, world";
    char* token = strtok(str, ",");
    printf("%s\n", str);

    return 0;
}
```

- 해결 방법 - 문자열을 복사해서 사용한다.

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

int main() {
    char str[] = "hello, world";

    char* copy = malloc(strlen(str) + 1);
    strncpy(copy, str, strlen(str) + 1);

    char* token = strtok(copy, ",");
    printf("%s\n", str);

    free(copy);
    return 0;
}
```

STR-03. 문자열을 위한 공간이 문자 데이터와 널 종료 문자를 담기에 충분함을 보장하라

데이터를 모두 담을 만큼 충분히 크지 않은 버퍼에 복사하면 버퍼 오버플로가 발생한다. 특히 널 문자로 종료되는 문자열의 경우, 데이터를 조작할 때 문자열의 크기를 제한하지 않아 종종 오버플로가 발생한다. 공격자는 이 상태를 악용해 취약한 프로세스의 권한으로 임의 코드를 실행할 수 있게 된다.

위험한 코드 1.

- 다음 코드에서 널 종료 문자를 저장하다가 버퍼 오버플로가 발생할 수 있다.

```
#include <stdio.h>

enum { ARR_SIZE = 6 };

int main()
{
        char src[ARR_SIZE];
        for (int i = 0; i < ARR_SIZE; i++)
                src[i] = 'A' + i;
        char dst[ARR_SIZE];
        int i;
        for (i = 0; src[i] && i < sizeof(dst); i++)
                dst[i] = src[i];
        dst[i] = '\0';
        printf("%s\n", dst);
        return 0;
}
```

- 해결 방법 - dst에 추가되는 널 종료문자를 고려하여 루프의 종료 조건을 수정한다.

```
#include <stdio.h>

enum { ARR_SIZE = 6 };

int main()
{
        char src[ARR_SIZE];
```

```
        for (int i = 0; i < ARR_SIZE; i++)
                src[i] = 'A' + i;
        char dst[ARR_SIZE];
        int i;
        for (i = 0; src[i] && i < sizeof(dst) - 1; i++)
                dst[i] = src[i];
        dst[i] = '\0';
        printf("%s\n", dst);
        return 0;
}
```

위험한 코드 2.

- 다음의 코드는 명령행의 인자를 수정하기 위해 복사를 수행하고 있다. 이 때 복사하기 위한 공간을 잘못 할당한 경우, 버퍼 오버플로를 일으킬 수 있다.

```
#include <stdio.h>
int main(int argc, char* argv[])
{
        char prog_name[128];
        strcpy(prog_name, argv[0]);
        // ...
        return 0;
}
```

- 해결 방법 - 복사되는 공간의 크기가 적절하게 생성될 수 있도록 동적 메모리 할당을 사용한다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char* argv[])
{
        char* prog_name = malloc(strlen(argv[0]) + 1);
        if (prog_name == NULL) {
                perror("malloc");
                return -1;
        }
        strcpy(prog_name, argv[0]);
        // ...
        free(prog_name);
        return 0;
}
```

STR-04. 경계가 불분명한 소스로부터 고정된 길이의 배열에 데이터를 복사하지 마라

별도의 경계 없이 복사를 수행하는 함수들은 종종 외부 입력에서 적절한 크기가 들어올 것이 라고 생각한다. 그러나 이러한 가정은 틀릴 수 있고 버퍼 오버플로우를 발생시킬 수 있다.

위험한 코드 1.

```c
#include <stdio.h>

int main() {
    char buf[32];
    gets(buf);
    printf("%s\n", buf);

    return 0;
}
```

- 해결 방법 - fgets 사용한다.

```c
#include <stdio.h>

int main() {
    char buf[32];
    fgets(buf, sizeof(buf), stdin);
    printf("%s\n", buf);

    return 0;
}
```

위험한 코드 2.

- 다음의 코드는 버퍼 오버플로가 발생할 가능성이 있다.

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
        char name[16];
        printf("input name: ");
        scanf("%s", name);
        printf("-> %s\n", name);
        return 0;
}
```

- 해결 방법 - 포맷 지정자로 경계를 지정한다.

```c
#include <stdio.h>
#define TO_STR(x) #x
#define STR_LIMIT(x) TO_STR(x)
#define INPUT_SIZE 15
int main(int argc, char* argv[]) {
        char name[INPUT_SIZE + 1];
        printf("input name: ");
        scanf("%"STR_LIMIT(INPUT_SIZE)"s", name);
        printf("-> %s\n", name);
        return 0;
}
```

Chapter 9. Memory

MEM-01. 동일한 추상화 레벨의 같은 모듈 안에서 메모리를 할당하고 해제하라

다른 모듈과 다른 추상화 레벨에서 메모리를 할당하고 해제하면 언제 메모리 블록을 해제해야 할지 판한다하거 혹은 언제 해제됐는지 파악하기가 어려워서 중복 해제, 해제된 메모리에 접근, 해제되거나 할당되지 않는 메모리에 쓰기 등 취약성을 일으키는 프로그래밍 결점으로 발전할 수 있다.

위험한 코드 1.

- 다음 코드는 서로 다른 추상화 레벨에서 메모리가 해제되어 중복 해제 문제가 발생하게 된다.

```
#include <stdio.h>
#include <stdlib.h>
enum { MIN_SIZE_ALLOWED = 32 };
int verify_list(char* list, size_t size)
{
        if (size < MIN_SIZE_ALLOWED) {
                free(list);
                return -1;
        }
        return 0;
}
void process_list(size_t size)
{
        char* list = malloc(size);
        if (list == NULL) {
                perror("malloc");
                exit(-1);
        }
        if (verify_list(list, size) < 0) {
                free(list);
                return;
        }
        // ...
        free(list);
}
```

```
int main()
{
        process_list(10);
        return 0;
}
```

- 해결 방법 - 동일한 추상화 레벨에서만 해제한다.

```
#include <stdio.h>
#include <stdlib.h>
enum { MIN_SIZE_ALLOWED = 32 };
int verify_list(char* list, size_t size)
{
        if (size < MIN_SIZE_ALLOWED)
                return -1;
        return 0;
}
void process_list(size_t size)
{
        char* list = malloc(size);
        if (list == NULL) {
                perror("malloc");
                exit(-1);
        }
        if (verify_list(list, size) < 0) {
                free(list);
                return;
        }
        // ...
        free(list);
}
int main()
{
        process_list(10);
        return 0;
}
```

MEM-02. 메모리 해제 후, 즉시 포인터에 새로운 값을 저장하라

댕글링 포인터는 중복 해제나 해제된 메모리를 액세스하는 취약성이 있다. 댕글링 포인터를 비롯한 메모리 관련 취약점을 제거하는 간단하면서도 효과적인 방법은 포인터를 해제한 후 다른 유효한 객체를 참조하게 하거나 NULL 값을 할당하는 것이다.

위험한 코드

- 다음 코드는 이중 해제 문제가 발생한다.

```
#include <stdio.h>
#include <stdlib.h>
enum { ARR_SIZE = 10 };
int main()
{
        int* pArr = malloc(sizeof(int) * ARR_SIZE);
        if (pArr == NULL) {
                perror("malloc");
                return -1;
        }
        free(pArr);
        // ...
        free(pArr);
        return 0;
}
```

- 해결 방법 - 해제 후 바로 널로 초기화한다.

```
#include <stdio.h>
#include <stdlib.h>
enum { ARR_SIZE = 10 };
int main()
{
        int* pArr = malloc(sizeof(int) * ARR_SIZE);
        if (pArr == NULL) {
                perror("malloc");
                return -1;
        }
        free(pArr);
        pArr = NULL;
```

```
        // ...
        free(pArr);
        return 0;
}
```

MEM-03. 메모리 할당 함수의 반환 값을 즉시 할당된 타입의 포인터로 변환시켜라

malloc 함수의 결과를 적절한 타입의 포인터로 캐스팅하면 컴파일러가 이후의 코드에서 발생할 수 있는 부적절한 포인터 변환을 잡아낼 수 있다.

위험한 코드

- 다음의 코드는 잘못된 메모리 참조로 인해 프로그램은 비정상적으로 종료될 수 있다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct {
        char name[32];
        int age;
} Human;
typedef struct {
        char name[32];
        int age;
        int id;
} Student;
int main()
{
        Student* p = malloc(sizeof(Human));
        if (p == NULL) {
                perror("malloc");
                return -1;
        }
        // ...
        p->id = 123456;
        // ...
        free(p);
        return 0;
}
```

- 해결 방법 - sizeof 표현식에서의 타입과 같은 타입을 포인터 캐스팅에 사용한다.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct {
        char name[32];
        int age;
} Human;
typedef struct {
        char name[32];
        int age;
        int id;
} Student;
int main()
{
        Student* p = (Human*)malloc(sizeof(Human));
        if (p == NULL) {
                perror("malloc");
                return -1;
        }
        // ...
        p->id = 123456;
        //...
        free(p);
        return 0;
}
```

MEM-04. 재사용을 위해 반환된 재사용 가능한 리소스에 있는 중요한 정보를 클리어하라

재사용하게 된 리소스에는 민감한 데이터가 들어 있어 이를 클리어하지 않는 경우 접근할 권한이 없거나 없는 사용자나 봐서는 안 될 사람에게 노출될 수 있다. 재사용 가능한 리소스의 예는 다음과 같다.
동적으로 할당된 메모리

- 정적으로 할당된 메모리
- 자동으로 할당된 스택 메모리
- 메모리 캐시
- 디스크 또는 디스크 캐시

위험한 코드

- 다음의 코드에서 힙에 할당된 메모리는 해제된 이후에도 메모리 블록에 계속 유지될 수 있다. 따라서 의도하지 않은 정보 노출로 이어질 수 있다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
const char* password = "qwer1234";
int main()
{
        char* p = malloc(strlen(password) + 1);
        if (p == NULL) {
                perror("malloc");
                return -1;
        }
        strcpy(p, password);
        free(p);
        printf("%s\n", p);
        return 0;
}
```

- 해결 방법 - 동적 메모리를 해제하기 전 0으로 클리어한다.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
const char* password = "qwer1234";
int main()
{
        char* p = malloc(strlen(password) + 1);
        if (p == NULL) {
                perror("malloc");
                return -1;
        }
        strcpy(p, password);
        memset(p, '\0', strlen(p));
        free(p);
        printf("%s\n", p);
        return 0;
}
```

MEM-05. 크기가 0인 할당을 수행하지 마라

malloc, calloc, realloc 함수 사용 시, 크기가 0인 할당을 수행했을 때의 결과는 시스템마다 다르게 나타난다. C99 표준에 따르면

요청된 크기가 0인 경우, 동작은 구현에 따라 다르게 정의된다. 널 포인터가 반환되거나 크기가 0이 아닌 경우와 같게 동작할 수 있다. 다만 이 경우 반환된 포인터를 객체에 접근하기 위해 사용해선 안된다.

위험한 코드

- 다음 코드의 실행 결과는 미정의 동작이다.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        int* pArr = malloc(0);
        if (pArr == NULL) {
                perror("malloc");
                return -1;
        }
        for (int i = 0; i < 10; i++)
                pArr[i] = 0;
        free(pArr);
        return 0;
}
```

- 해결 방법 - 동적 메모리 할당 전 크기 검사를 수행한다.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
```

```
        size_t size = 0;
        if (size < 1) {
                printf("size is 0\n");
                return -1;
        }
        int* pArr = malloc(size);
        if (pArr == NULL) {
                perror("malloc");
                return -1;
        }
        for (int i = 0; i < 10; i++)
                pArr[i] = 0;
        free(pArr);
        return 0;
}
```

MEM-06. 메모리 할당 루틴이 메모리를 초기화해 줄 것이라 가정하지 마라

malloc 함수로 할당된 또는 realloc 함수에 의해 추가된 메모리는 초기화되지 않는다.

위험한 코드

- 다음의 코드는 의도하지 않은 실행 결과가 나타날 수 있다.

```
#include <stdio.h>
#include <stdlib.h>
#define STACK_SIZE (10)
typedef struct {
        int top;
        int arr[STACK_SIZE];
} Stack;
int is_empty(const Stack* s)
{
        return s->top == 0;
}
Stack* create_stack()
{
        Stack* p = malloc(sizeof(Stack));
        if (p == NULL) {
                perror("malloc");
                return NULL;
        }
        return p;
}
void remove_stack(Stack* s)
{
        free(s);
        s->top = 0;
}
int main()
{
        Stack* stack = create_stack();
        if (is_empty(stack))
                printf("stack is empty₩n");
        remove_stack(stack);
        return 0;
}
```

- 해결 방법 - 명시적으로 초기화하거나 calloc 함수를 사용한다.

```
#include <stdio.h>
#include <stdlib.h>
#define STACK_SIZE (10)
typedef struct {
        int top;
        int arr[STACK_SIZE];
} Stack;
int is_empty(const Stack* s)
{
        return s->top == 0;
}
Stack* create_stack()
{
        Stack* p = calloc(1, sizeof(Stack));
        if (p == NULL) {
                perror("calloc");
                return NULL;
        }
        return p;
}
void remove_stack(Stack* s)
{
        free(s);
        s->top = 0;
}
int main()
{
        Stack* stack = create_stack();
        if (is_empty(stack))
                printf("stack is empty\n");
        remove_stack(stack);
        return 0;
}
```

MEM-07. 메모리 할당 에러를 찾아 해결하라

메모리 할당 함수들은 리턴 값으로 할당이 성공적으로 이루어졌는지 여부를 알려준다. C99 표준에서는 calloc, realloc, malloc 함수는 요청된 메모리 할당이 실패하면 널 포인터를 반환한다. 메모리 관리 에러를 발견하고 적절하게 처리하지 않으면 프로그램이 예측할 수 없는 동작을 일으킬 수 있다.

위험한 코드 1.

- 다음의 코드는 널 검사를 하지 않고 있다.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        int* pArr = (int*)malloc(sizeof(int) * -1);
        for (int i = 0; i < 10; i++)
                pArr[i] = 0;
        free(pArr);
        return 0;
}
```

- 해결 방법 - 널 검사를 수행한다.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        int* pArr = (int*)malloc(sizeof(int) * -1);
        if (pArr == NULL) {
                perror("malloc");
                return -1;
        }
        for (int i = 0; i < 10; i++)
                pArr[i] = 0;
        free(pArr);
```

```
        return 0;
}
```

위험한 코드 2.

- 다음 코드는 메모리 누수가 발생할 수 있다.

```
#include <stdio.h>
#include <stdlib.h>
#define ARR_SIZE (10)
int main()
{
        int* pArr = (int*)malloc(sizeof(int) * ARR_SIZE);
        if (pArr == NULL) {
                perror("malloc");
                return -1;
        }
        // ...
        int new_size = ARR_SIZE * -1;
        pArr = realloc(pArr, new_size);
        for (int i = 0; i < ARR_SIZE; i++)
                pArr[i] = i + 1;
        free(pArr);
        return 0;
}
```

- 해결 방법 - 임시 포인터를 사용하여 해결한다.

```
#include <stdio.h>
#include <stdlib.h>
#define ARR_SIZE (10)
int main()
{
        int* pArr = (int*)malloc(sizeof(int) * ARR_SIZE);
        if (pArr == NULL) {
                perror("malloc");
                return -1;
        }
        // ...
        int new_size = ARR_SIZE * -1;
        int* temp = realloc(pArr, new_size);
        if (temp == NULL) {
```

```
                perror("realloc");
                return -1;
        }
        for (int i = 0; i < ARR_SIZE; i++)
                pArr[i] = i + 1;
        free(pArr);
        return 0;
}
```

Chapter 10. File I/O

FIO-01. 포맷 문자열을 사용할 때 주의하라

포맷 문자열을 정확하지 않게 지정하면 비정상적인 프로그램 종료를 일으킨다. 일반적으로 포맷 문자열을 만들 때 공통적으로 범하는 실수는 다음과 같다.

- 유효하지 않는 변환 지정자의 사용
- 정확하지 않은 지정자에 대한 길이 수정자의 사용
- 인자와 변환 지정자의 타입 불일치
- 유효하지 않는 문자 클래스의 사용

위험한 코드

- 다음 코드에서 입력 값과 변환 지정자가 일치하지 않아 정의되지 않은 동작을 초래한다.

```
#include <stdio.h>
int main()
{
        const char* err_msg = "not enough memory";
        const int err_code = 3;
        printf("Error Message: %d(%s)\n", err_msg, err_code);
        return 0;
}
```

- 해결 방법 - 정확한 변환 지정자를 사용한다.

```c
#include <stdio.h>
int main()
{
        const char* err_msg = "not enough memory";
        const int err_code = 3;
        printf("Error Message: %s(%d)\n", err_msg, err_code);
        return 0;
}
```

FIO-02. 문자 입출력 함수의 반환 값을 캡처할 때는 int를 사용하라

fgetc, getc, getchar 같은 문자 입출력 함수는 모두 스트림으로부터 문자를 읽어 int로 반환한다. 여기서 반환 값을 char 타입으로 저장할 경우, EOF를 구분할 수 없게 된다.

위험한 코드

- 다음 코드는 정상적으로 동작하지 않는다.

```
#include <stdio.h>
int main(int argc, char* argv[])
{
        if (argc != 2) {
                printf("usage: %s <FILE_NAME>\n", argv[0]);
                return -1;
        }
        FILE* fp = fopen(argv[1], "rb");
        if (fp == NULL) {
                perror("fopen");
                return -1;
        }
        size_t size = 0;
        char ch = fgetc(fp);
        while (ch != EOF) {
                ++size;
                ch = fgetc(fp);
        }
        printf("file size: %u\n", size);
        fclose(fp);
        return 0;
}
```

- 해결 방법 - 캡처 타입을 char에서 int로 변경한다.

```
#include <stdio.h>
int main(int argc, char* argv[])
{
        if (argc != 2) {
                printf("usage: %s <FILE_NAME>\n", argv[0]);
                return -1;
        }
```

```
        FILE* fp = fopen(argv[1], "rb");
        if (fp == NULL) {
                perror("fopen");
                return -1;
        }
        size_t size = 0;
        int ch = fgetc(fp);
        while (ch != EOF) {
                ++size;
                ch = fgetc(fp);
        }
        printf("file size: %u\n", size);
        fclose(fp);
        return 0;
}
```

FIO-03. sizeof(char) == sizeof(int) 일 때는 파일의 끝이나 파일 에러를 찾기 위해 feof와 ferror 함수를 사용하라

char 타입이 int 타입과 크기가 같을 경우, EOF를 구분할 수 없다. 만약 파일의 끝이나 파일에 대한 에러를 검출하고 싶다면 다음의 함수를 사용한다.

- feof
- ferror

위험한 코드

- 다음의 코드는 char와 int 타입의 크기가 동일한 시스템에서는 제대로 동작하지 않는다.

```
#include <stdio.h>
int main(int argc, char* argv[])
{
        if (argc != 2) {
                printf("usage: %s <FILE_NAME>\n", argv[0]);
                return -1;
        }
        FILE* fp = fopen(argv[1], "rb");
        if (fp == NULL) {
                perror("fopen");
                return -1;
        }
        size_t size = 0;
        int ch = fgetc(fp);
        while (ch != EOF) {
                ++size;
                ch = fgetc(fp);
        }
        printf("file size: %u\n", size);
        fclose(fp);
        return 0;
}
```

- 해결 방법 1. feof 함수를 사용한다.

```c
#include <stdio.h>
int main(int argc, char* argv[])
{
	if (argc != 2) {
		printf("usage: %s <FILE_NAME>\n", argv[0]);
		return -1;
	}
	FILE* fp = fopen(argv[1], "rb");
	if (fp == NULL) {
		perror("fopen");
		return -1;
	}
	size_t size = 0;
	while (!feof(fp)) {
		++size;
		fgetc(fp);
	}
	printf("file size: %u\n", size - 1);
	fclose(fp);
	return 0;
}
```

- 해결 방법 2. 정적 어썰션을 사용

```c
#include <stdio.h>
int main(int argc, char* argv[])
{
	if (argc != 2) {
		printf("usage: %s <FILE_NAME>\n", argv[0]);
		return -1;
	}
	FILE* fp = fopen(argv[1], "rb");
	if (fp == NULL) {
		perror("fopen");
		return -1;
	}
	static_assert(sizeof(char) < sizeof(int), "diff type");
	size_t size = 0;
	while (!feof(fp)) {
		++size;
		fgetc(fp);
	}
	printf("file size: %u\n", size - 1);
	fclose(fp);
	return 0;
}
```

FIO-04. 문자 데이터를 읽었다고 가정하지 마라

문자 데이터를 읽었다고 가정할 경우, 의도하지 않게 코드가 실행될 수도 있다.

위험한 코드

- 만약 표준 입력으로부터 널이 입력될 경우, 배열의 경계 밖에 쓰기를 수행할 수 있다.

```
#include <stdio.h>
#include <string.h>
int main()
{
        char buf[32];
        if (fgets(buf, sizeof(buf), stdin)) {
                buf[strlen(buf) - 1] = '\0';
                printf("-> %s\n", buf);
        }
        else {
                printf("fgets error\n");
        }
        return 0;
}
```

- 해결 방법 - 데이터가 존재하는지 확인한다.

```
#include <stdio.h>
#include <string.h>
int main()
{
        char buf[32];
        if (fgets(buf, sizeof(buf), stdin)) {
                if (*buf)
                        buf[strlen(buf) - 1] = '\0';
                printf("-> %s\n", buf);
        }
        else {
                printf("fgets error\n");
        }
        return 0;
}
```

FIO-05. fgets 함수를 사용할 때 개행 문자가 읽힌다고 가정하지 마라

fgets 함수는 보통 입력 스트림으로부터 개행 문자로 종료된 문자열을 읽기 위해 사용한다. 이 때 문자열의 끝에는 개행과 널이 포함되게 된다. 그러나 주어진 버퍼의 크기를 넘어서서 입력될 경우, 버퍼의 끝은 널로만 채워지게 된다.

입력된 버퍼의 내용을 10진수로 출력하면 끝에 개행과 널이 붙여지는 것을 확인

위험한 코드

- 다음의 코드에서 입력된 문자열이 버퍼의 크기를 넘어설 경우, 데이터는 유실된다.

```
#include <stdio.h>
#include <string.h>
int main()
{
        char buf[5];
        if (fgets(buf, sizeof(buf), stdin)) {
                buf[strlen(buf) - 1] = '\0';
                printf("-> %s\n", buf);
        }
        else {
                printf("fgets error\n");
        }
        return 0;
}
```

- 해결 방법 - strchr 함수를 사용한다.

```
#include <stdio.h>
#include <string.h>
int main()
{
        char buf[5];
        if (fgets(buf, sizeof(buf), stdin)) {
                char* p = strchr(buf, '\n');
                if (p) {
```

```
                        *p = '\0';
                }
                printf("-> %s\n", buf);
        }
        else {
                printf("fgets error\n");
        }
        return 0;
}
```

FIO-06. 입출력 FILE 객체를 복사해서 사용하지 마라

FILE 객체의 복사본을 실제 객체 대신 사용하면 프로그램이 다운되거나 서비스 거부 공격으로 이어질 수 있다.

위험한 코드 1.

- 표준 입출력 객체를 복사해서 사용하고 있다.

```
#include <stdio.h>
int main()
{
        FILE out = *stdout;
        fputs("hello, world\n", &out);
        return 0;
}
```

- 해결 방법 - 포인터에 대한 복사본을 사용한다.

```
#include <stdio.h>
int main()
{
        FILE* out = stdout;
        fputs("hello, world\n", out);
        return 0;
}
```

FIO-07. 플러시나 위치 조정 함수 없이 스트림으로부터 입출력을 교대로 수행하지 마라

스트림에 대해 출력 다음에 fflush, fseek, fsetpos, rewind 함수의 호출 없이 바로 입력을 받거나, 입력 다음에 fseek, fsetpos, rewind 호출 없이 바로 출력을 수행하면 정의되지 않은 동작을 할 수 있으므로 주의해야 한다.

위험한 코드

- 다음 코드는 제대로 동작하지 않을 수 있다.

```c
#include <stdio.h>
#include <string.h>
enum { BUF_SIZE = 64 };
int main()
{
        const char* msg = "hello, world";
        FILE* fp = fopen("log.txt", "w+");
        if (fp == NULL) {
                perror("fopen");
                goto except;
        }
        char buf[BUF_SIZE];
        if (fwrite(msg, BUF_SIZE, 1, fp) < 0) {
                perror("frwite");
                goto except;
        }
        memset(buf, '\0', sizeof(buf));
        if (fread(buf, BUF_SIZE, 1, fp) < 0) {
                perror("fread");
                goto except;
        }
        printf("-> %s\n", buf);
except:
        fclose(fp);
        return 0;
}
```

- 해결 방법 - 파일 오프셋을 처음으로 이동시킨다.

```c
#include <stdio.h>
#include <string.h>
enum { BUF_SIZE = 64 };
int main()
{
        const char* msg = "hello, world";
        FILE* fp = fopen("log.txt", "w+");
        if (fp == NULL) {
                perror("fopen");
                goto except;
        }
        char buf[BUF_SIZE];
        if (fwrite(msg, BUF_SIZE, 1, fp) < 0) {
                perror("frwite");
                goto except;
        }
        fseek(fp, 0L, SEEK_SET);
        memset(buf, '\0', sizeof(buf));
        if (fread(buf, BUF_SIZE, 1, fp) < 0) {
                perror("fread");
                goto except;
        }
        printf("-> %s\n", buf);
except:
        fclose(fp);
        return 0;
}
```