

시큐어 코딩 C

202107006	홍서빈
202107003	양나은
202107011	이승현

제 1절
입력데이터 검증 및 표현

<시큐어 코딩>

- 시큐어 코딩이란 : 소프트웨어 개발 보안에서 개발 과정 중 소스코드 구현단계에서 보안 약점을 배제하기 위한 코딩을 시큐어 코딩이라 한다

[1. SQL 삽입]

- 정의: 데이터 베이스와 연동된 것들은 데이터 유효성 검증을 해야하는데 그렇지 않으면 공격자가 폼 및 URL 입력란에 SQL 문을 삽입해 **DB로부터 정보를 열람 및 조작 가능**해지는 보안 약점을 의미

- 안전한 코딩기법

- I) 외부 입력이나 외부 변수로부터 받은 값이 직접 SQL 함수의 인자로 전달되거나 문자열 복사를 통해 전달되는 것은 위험 -> **인자화된 질의문을 사용할 것**
- II) 외부 입력값을 그대로 사용해야한다면? -> 입력받은 값을 **필터링**을 통해 처리후 사용

```
#include <stdlib.h>
#include <sql.h>
void Sql_process(SQLHSTMT sqlh)
{
    char *query = getenv("query_string"); //select문등의 sql 질의어에 어떠한 처리도 없이 삽입됨()
    SQLExecDirect(sqlh, query, SQL_NTS);
}
```

```
#include <sql.h>
void Sql_process(SQLHSTMT sqlh)
{
    char *query_items = "SELECT * FROM items"; //인자화된 질의를 사용함으로써 질의 구조의 변경 방지
    SQLExecDirect(sqlh, query_items, SQL_NTS);
}
```

[2. 자원삽입]

- 정의: 외부 입력값을 검증하지 않고 시스템 자원에 대한 식별자로 사용하는 경우, 공격자는 입력값 조작을 통해 시스템이 보호하는 자원에 임의로 접근하거나 수정 가능

- 안전한 코딩기법

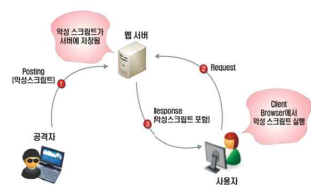
- I) 외부입력을 자원식별자(파일,소켓의 포트 등)으로 사용하는 경우, 적절한 검증과정을 수행
or 사전에 정의된 적합한 **리스트**에서 선택
- II) 외부입력이 파일명인 경우, 경로 순회를 수행할수 있는 문자를 제거

```
int main()
{
    char* rPort = getenv("rPort");
    // 포트번호를 외부입력으로 받음 즉 getenv("rPort") 함수를 사용한게
    // 포트번호를 외부에서 받는 거
    => 공격자가 포트번호를 마음대로 하여 소켓을 생성할수 있다
    struct sockaddr_in serv_addr;
    int sockfd = 0;
    ...
    serv_addr.sin_port = htons(atoi(rPort));
    if (connect(sockfd, &serv_addr, sizeof(serv_addr)) < 0)
    {
        exit(1);
    }
    return 0;
}
```

```
// 외부입력값에 따라 미리 포트번호를 선택 => 공격자가 임의의 포트를 사용 불가
int main()
{
    char* rPort = getenv("rPort");
    struct sockaddr_in serv_addr;
    int sockfd = 0;
    int port = 0;
    ...
    if(strcmp(rPort,"") < 0)
    {
        printf("bad input");
    } // 포트번호가 없다면 bad input을 출력
    port = atoi(rPort);
    if( port < 3000 || port > 3003)
    {
        port = 3000;
    }
    serv_addr.sin_port = htons(atoi(rPort));
    if(connect(sockfd,&serv_addr, sizeof(serv_addr)) < 0)
    {
        exit(1);
    }
    return 0;
}
```

[3. 크로스사이트 스크립트 cross-site scripting, XSS]

-정의: 웹페이지에 악의적인 스크립트를 포함시켜 사용자 측에서 실행되게 유도



© STEP 1 > 검증되지 않은 외부입력이 동적 웹페이지 생성에 사용됨

STEP 2 > 잘못된 동적 웹 페이지를 열람하는 접속자는 부적절한 스크립트를 실행시키게 되어 정보유출

공격을 유발할수 있음

- 안전한 코딩기법

I) 사용자가 문자열에 스크립트를 삽입하여 실행하는 것을 막기위해 사용자가 입력한 문자열에서

<,>,&," 등을 replace등의 문자 변환 함수 아니면 메소드를 사용하여 <,>,&,"로 치환

특수코드 값	실제 표현	&	& (앰퍼샌드)
<	< (부등호 역치)		
>	> (부등호 역치)		
 	· (공백, Space 한칸)	"	" (큰따옴표 하나)

II) html 태그를 허용하는 게시판에선 지원하는 html 태그의 [리스트를 선정후](#), 해당 태그만 허용하는 방식

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
// 게시판에서 허용하고자 하는 HTML 태그리스트(White List) 정의
static unsigned char *allowed_formatters[]
= {
    "b", "big", "blink", "i", "s", "small", "strike", "sub", "sup", "tt", "u",
    "abbr", "acronym", "cite", "code", "del", "dfn", "em", "ins", "kbd", "samp",
    "strong", "var", "dir", "li", "dl", "dd", "dt", "menu", "ol", "ul", "hr",
    "br", "p", "h1", "h2", "h3", "h4", "h5", "h6", "center", "bdo", "blockquote",
    "nobr", "plaintext", "pre", "q", "spacer",
    "a"
};
#define SKIP_WHITESPACE(p) while (isspace(*p)) p++; 15:
// 요청들어온 사이트 링크에 대한 유효성 검증
static int spc_is_valid_link(const char *input) {
    static const char *href="href";
    static const char *http = "http://";
    int
    quoted_string = 0, seen_whitespace = 0;
    if (!isspace(*input)) return 0;
    SKIP_WHITESPACE(input);
    if (strncasecmp(href, input, strlen(href))) return 0;
    input += strlen(href);
    SKIP_WHITESPACE(input);
    if (*input++ != '=') return 0;
    SKIP_WHITESPACE(input);
    if (*input == '"') {
        quoted_string = 1;
        input++;
    }
    if (strncasecmp(http, input, strlen(http))) return 0;
    for (input += strlen(http); *input && *input != '>'; input++) {
        switch (*input) {
            case '.': case '/': case '-': case '_':
                break;
            case '"':
                if (!quoted_string) return 0;
                SKIP_WHITESPACE(input);
                if (*input != '>') return 0;
                return 1;
            default:
                if (isspace(*input)) {
```

```
if (seen_whitespace && !quoted_string) return 0;
        SKIP_WHITESPACE(input);
        seen_whitespace = 1;
        break;
    }
    if (!isalnum(*input)) return 0;
    break;
}
return (*input && !quoted_string); 55:
}
//HTML 연결 문자열에 대한 구성확인
static int spc_allow_tag(const char *input) {
    int i;
    char *tmp;
    if (*input == 'a')
        return spc_is_valid_link(input + 1);
    if (*input == '/') {
        input++;
        SKIP_WHITESPACE(input);
    }
    for (i = 0; i < sizeof(allowed_formatters); i++) {
        if (strncasecmp(allowed_formatters[i], input, strlen(allowed_formatters[i])))
            continue;
        else {
            tmp = input + strlen(allowed_formatters[i]);
            SKIP_WHITESPACE(tmp);
            if (*input == '>') return 1;
        }
    }
    return 0; 77:
}
// HTML 스크립트 생성에 사용되는 문자열 변경
char *spc_escape_html(const char *input)
{
    char
    *output, *ptr;89:
    *ptr++ = '&'; *ptr++ = 'l'; *ptr++ = 't'; *ptr++ = ';';
    break;
}
else
{
    do
    {
        *ptr++ = *c;
    } while (++c != '>');
    *ptr++ = '>';
    break;
}
case '>':
    *ptr++ = '&'; *ptr++ = 'g'; *ptr++ = 't'; *ptr++ = ';';
    break;
case '&':
    *ptr++ = '&'; *ptr++ = 'a'; *ptr++ = 'm'; *ptr++ = 'p';
    *ptr++ = ';';
    break;
case '"':
    *ptr++ = '&'; *ptr++ = 'q'; *ptr++ = 'u'; *ptr++ = 'o';
    *ptr++ = 't'; *ptr++ = 't';
    break;
default:
    *ptr++ = *c;
    break;
}
}
*ptr = 0;
return output;
}
```

[4. 운영체제 명령어 삽입 os]

-정의: 외부 입력이 시스템 명령어 실행 인수로 적절한 처리 없이 사용되면 위험하다
일반적으로 명령줄 인수나 **스트림 입력**등의 외부입력을 사용하여 시스템 명령어를 생성하는 프로그램이 많이 있다. 하지만 이런 경우 외부입력 문자열은 신뢰할 수 없기 때문에 적절한 처리를 해주지 않는 경우, 공격자가 원하는 명령어 실행이 가능하게 된다.

* 스트림 입력: 스트림(시냇물을 의미) . 컴퓨터 공학에서의 스트림은 연속적인 데이터의 흐름, 혹은 데이터를 전송하는 소프트웨어 모듈을 의미

- 안전한 코딩기법

- I) 외부 입력이 직접 또는 문자열 복사를 통해 system() 함수로 직접 전달되는 것은 위험
=> 미리 **적절한 후보 명령어 리스트**를 만들고 이 중에 선택 or **위험한 문자열의 존재 여부를 검사**

```
// 이 프로그램을 catWrapper라고 했을 때
공격자가 story.txt: ls인자로 전달하면 Strory.txt 파일의 내용을 보여준후
현재 작업 디렉터리 목록을 출력한다

fgets(arg,80,stdin);
commandLength = strlen(cat) + strlen(arg) + 1;
command = (char *) malloc(commandLength);
strncpy(command, cat, commandLength);
strncat(command, argv[1], (commandLength - strlen(cat)) );
system(command);
return 0;
```

```
fgets(arg,80,stdin);
if (strpbrk(arg,":\\\"'"))
{
    exit(1); // strpbrk() 함수, 위험 문자가 있으면 명령을 수행하지 않는다
}
commandLength = strlen(cat) + strlen(arg) + 1;
if(commandLength < 20)
{
    command = (char *) malloc(commandLength);
}
```

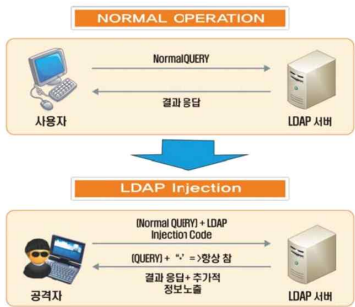
[5.LDAP 삽입]

LDAP란..(<-하이퍼링크 클릭)

-정의: 공격자가 외부 입력을 통해서 의도치 않은 LDAP 명령어를 수행할 수 있다. 즉, **웹 응용프로그램이 사용자가 제공한 입력을 올바르게 처리하지 못하면, 공격자가 LDAP 명령문의 구성을 바꿀수 있다.**

이로 인해 프로세스가 명령을 실행한 **컴포넌트**와 동일한 권한을 가지고 동작하게 된다
LDAP 쿼리문이나 결과에 외부 입력이 부분적으로 적절한 처리없이 사용되면, LDAP 쿼리문이 실행될 때 공격자는 LDAP 쿼리문의 내용을 마음대로 변경할수 있다

* 컴포넌트 :소프트웨어 시스템에서 독립적인 업무 또는 독립적인 기능을 수행하는 '모듈'로서 이후 시스템을 유지보수 하는데 있어 교체 가능한 부품이다



- 안전한 코딩기법

- I) 위험 문자에 대한 검사 없이 외부 입력을 LDAP 질의어 생성에 사용하면 안된다.
- II) DN과 필터에 사용되는 사용자 입력값에는 특수문자가 포함되지 않도록 **특수문자를 제거**
특수문자를 사용해야 하는 경우 실행명령이 아닌 일반문자로 인식되도록 처리

```
int main()
{
    char* filter = getenv("filter_string");
    int rc;
    LDAP *ld = NULL;
    LDAPMessage* result;

    rc=ldap_search_ext_s(ld,FIND_DN,LDAP_SCOPE_BASE,filter,NULL,0,NULL,
    NULL,LDAP_NO_LIMIT, LDAP_NO_LIMIT, &result);
    return 0;
}
```

```
#include <stdio.h>
#include <ldap.h>
#define FIND_DN ""
int main()
{
    char* filter = "(manager=admin)"; // 필터값을 외부 입력값이 아닌 특정한 값을 사용해 안전
    int rc;
    LDAP *ld = NULL;
    LDAPMessage* result;
    rc=ldap_search_ext_s(ld,FIND_DN,LDAP_SCOPE_BASE,filter,
    NULL,0,NULL,NULL,LDAP_NO_LIMIT,LDAP_NO_LIMIT, &result);
    return 0;
}
```

[6. 디렉터리 경로 조작] _상대 디렉터리 경로조작, 절대 디렉터리 경로조작으로 나뉨

6-1 > 상대 디렉터리 경로조작

- 정의: 외부입력을 통해 디렉터리 경로 문자열을 생성하는 경우, 결과 디렉터리가 제한된 디렉터리여야 할 때, 악의적인 외부 압력을 제대로 변환 시키지 않으면 예상 밖 영역의 경로 문자열이 생성될 수 있다.
- 이 취약점을 이용해 제한된 디렉터리 영역 밖을 공격자가 접근할수 있게된다.

.. 상위디렉터리를 의미하는 문자를 사용해 상위를 접근할수 있는 경로를 생성 하게 되는 것

- 안전한 코딩기법

- 1) 파일 접근 함수의 파일 경로 인수의 일부를 외부 입력으로부터 조합하여 사용하지 않는 것이 바람직

```
void f()
{
    char* rName = getenv("reportName");

    char buf[30];
    strncpy(buf, "/home/www/tmp/", 30);
    strncat(buf, rName, 30);
    unlink(buf);
}
```

** /home/www/tmp에서 파일 처리 함수를 실행 하지만 reportName이 ../../etc/passwd와 같이 입력되면 상위 디렉터리에 있는 파일에 접근하게 됨*

```
void f()
{
    char buf[30];
    strncpy(buf, "/home/www/tmp/", 30);
    strncat(buf, "report", 30);
    unlink(buf);
}
```

** 외부입력으로 파일경로를 조합하여 파일 시스템에 접근하는 경로를 만들지 말아야 함*

6-2 > 절대 디렉터리 경로조작

- 정의: 외부 입력이 파일 시스템을 조작하는 경로를 직접 제어할 수 있거나 영향을 끼치면 위험함
- 이 취약점은 공격자가 응용프로그램에 치명적인 시스템파일 또는 일반파일을 접근하거나 변경가능하도록 함.

다음 두 조건이 만족되면 공격은 성공

1. 공격자가 파일 시스템 조작에 사용되는 경로를 결정
2. 그 파일 조작을 통해 공격자가 용납할 수 없는 권한을 획득, 예를 들어 공격자가 설정에 관계된 파일을 변경, 실행 가능

- 안전한 코딩기법

- 1) 파일 접근 함수의 파일 경로 인수로 외부 입력을 직접 사용하지 않는 것이 바람직

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
void absolute()
{
    /* 외부 입력값으로 파일 경로가 설정되고 있다*/
    char* rName = getenv("reportName");
    unlink(rName);
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
void absolute()
{
    /* 외부 입력값으로 파일 경로를 설정하지 않아야 한다*/
    unlink("/home/www/tmp/report");
}
```

[7. 정수 오버플로우]

-정의: 정수 연산시 정수형 변수가 취할수 있는 범위 초과할 때 발생하는 경우([하이퍼링크 클릭](#))
정수형 변수의 오버플로우는 정수값이 증가하면서 가장 큰 값보다 더 커져서 실제 비트 패턴으로 저장되는 값은 **아주 작은 수** 이거나 **음수**로 될 수 있음.

- 이러한 상황을 계산 후 검사하지 않고 쓰면- **순환문의 조건**이나 **메모리 할당**, **메모리 복사**등에 쓰이거나 이 값에 근거해 보안 관련 결정을 하는 경우 보안취약점이 발생할 수 있다.

- 안전한 코딩기법

I) signed(부호있는) 정수타입: 오버플로우 발생시 양수가 음수로 음수가 양수로 변환되면서 잘못된 값 전달

이를 위해, **메모리 할당 함수**에서는 할당량을 표시하는 파라미터를 **unsigned 타입으로 전달해야** 잘못된 값이 사용되는 것을 방지 가능

```
#include <stdlib.h>
void* intAlloc(int size, int reserve)
{
    void *rptr;
    size += reserve;
    //signed int로 선언된 변수를 malloc() 함수의 파라미터로
    사용하는 경우 이전처리 과정에서 오버플로우가 발생하여 음수가
    입력되어도 큰 양수로 잘못 해석하여 진행될수 있다
    rptr = malloc(size * sizeof(int));
    if (rptr == NULL)
    {
        exit(1);
        return rptr;
    }
}
```

```
#include <stdlib.h>
void* intAlloc(int size, int reserve)
{
    void* rptr;
    // malloc() 함수의 파라미터로 사용되는 변수를
    //unsigned로 사용해서 유효한 값의 범위를 넓혀 integer overflow 가능성을 낮춘다
    unsigned s;
    size += reserve;
    s = size * sizeof(int);
    if (s < 0)
    {
        return NULL;
    }
    rptr = malloc(s);
    if (rptr == NULL)
    {
        exit(1);
    }
    return rptr;
}
```

[8. 보호 메커니즘을 우회할수 있는 입력값 변조]

-정의: 응용프로그램이 **외부 입력값에 대한 신뢰를 전제로 보호 메커니즘을 사용**하는 경우 공격자가 입력값을 조작할 수 있다면 보호 메커니즘을 우회할수 있게 된다. 흔히 쿠키, 환경변수 또는 히든 필드 값은 조작될 수 없다고 가정되지만 공격자는 다양한 방법으로 이런 입력값 변경 가능
+조작된 내용은 탐지되지 않을 수 있음

* 인증, 인가같은 보안결정이 이런 **입력값(쿠키,환경변수,히든필드 등)에 기반으로 수행되는 경우**

: 공격자는 이런 **입력값을 조작**하여 응용프로그램의 보안을 우회할수 있으므로 **충분한 암호화, 무결성체크** 또는 **다른 메커니즘이 없는 경우 외부 사용자에 의한 입력값을 신뢰해서는 안됨**

- 안전한 코딩기법

I) 상태정보나 민감한 데이터 특히 사용자가 **세션 정보**와 같은 중요정보는 **서버에 저장** +보안확인도 서버에서 실행

II) 보안설계 관점에서 신뢰 불가한 입력값이 어플리케이션 내부로 들어올 수 있는 지점과 보안결정에 사용되는

입력 값을 식별하고 **제공되는 입력값에 의존할 필요가 없는 구조로 변경**할수 있는지 검토

스택/힙



[9. 스택에 할당된 버퍼 오버플로우]

-정의: 스택에 할당되는 버퍼들(=지역변수로 선언된 변수 or 함수인자로 넘어온 변수들)이 문자열 계산등에 의해 정의된 버퍼의 한계치를 넘는 경우 버퍼 오버플로우가 발생

- 안전한 코딩기법

- I) 프로그램이 버퍼가 저장할 수 있는것보다 많은 데이터를 입력하지 않는다
- II) 프로그램이 **버퍼 경계 밖의 메모리** 영역을 **참조하지 않는다**
- III) 프로그램이 사용할 메모리를 적절하게 계산하여 **로직에서 에러**를 발생시키지 않도록 한다
- IV) 입력에 대해서 경계검사를 한다
- V) strcpy() 문자열복사 같이 버퍼 오버플로우에 취약한 함수를 사용하지 않는다

```
#include <stdlib.h>
void manipulate_string(char* string)
{
    char buf[24];
    //매개변수로 받은 문자열 크기가 지역버퍼에 복사가 되는지
    확인하지 않고 strcpy() 함수를 이용해 데이터를 복사함
    strcpy(buf, string); // => 공격자가 스트림 매개 변수의 내용에 영향을 줄
    수 있다면 버퍼 오버플로우에 취약한 함수를 사용하지 않는다
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
//매개변수로 받은 변수(string)와 복사하려는 buf의 크기를 비교한다(if 문 사용)
void manipulate_string(char * string)
{
    char buf[24];
    /*복사하려는 buf와 길이비교를 한다*/
    If (strlen(string < sizeof(buf)))
    {
        strncpy(buf,string,sizeof(buf)-1);
    }
    /*문자열은 반드시 null로 종료 되어야 함*/
    buf[sizeof(buf)-1]='\0';
}
```

[10. 힙에 할당된 버퍼 오버플로우]

-정의: 힙에 할당되는 버퍼들(예를들어 malloc() 함수를 통해 할당된 버퍼들)에 문자열 등이 저장되어 질 때, 최초정의된 힙의 메모리 사이즈를 초과하여 문자열 등이 저장되는 경우 버퍼오버플로우가 발생한다

- 안전한 코딩기법

- I) 프로그램이 버퍼가 저장할 수 있는것보다 많은 데이터를 입력하지 않는다
- II) 프로그램이 **버퍼 경계 밖의 메모리** 영역을 **참조하지 않는다**
- III) 프로그램이 사용할 메모리를 적절하게 계산하여 **로직에서 에러**를 발생시키지 않도록 한다
- IV) 입력에 대해서 경계검사를 한다
- V) strcpy() 문자열복사 같이 버퍼 오버플로우에 취약한 함수를 사용하지 않는다

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define BUFSIZE 10 //특정크기의 메모리를 할당함
int main(char** argv)
{
    char* dest = NULL;
    dest = (char*)malloc(BUFSIZE);
    //특정 크기의 메모리를 할당한 후에
    strcpy() 함수를 이용하여 사용자 입력값을 그대로 메모리에
    복사하면 버퍼 오버플로우가 발생할수 있다.
    strcpy(dest, argv[1]);
    free(dest);
    return 0;
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define BUFSIZE 10 //특정크기의 메모리를 할당함
int main(int argc, char** argv)
{
    char* dest = NULL;
    dest = (char*)malloc(BUFSIZE);
    //입력된 문자열에 대하여 길이를 제한하는 strlen()함수를 사용해서 값을 저장
    한다.
    //strlen() 함수는 문자 배열의 마지막 부분에 자동적으로 Null을
    채워준다
    /**strcpy 사용법
    strcpy(a,b,size) b값을 a값에 복사
    strcpy(dest,argv[1], BUFSIZE); argv[1]값을 BUFSIZE만큼 dest에 복사
    free(dest);
    return 0;
}
```

[11. LDAP 처리]

-정의: LDAP 질의문이나 결과로 외부 입력이 적절한 처리 없이 사용되면 LDAP 질의문이 실행될 때 공격자는 LDAP 질의문의 내용을 마음대로 변경할수 있다. (여기서는 외부 입력이 LDAP 질의문 자체에 영향을 주는 경우를 뜻한다)

- 안전한 코딩기법

- I) 외부의 입력이 직접 루트 디렉터리 접근에 사용되지 않도록
- II) 사용할 경우 적절한 접근제어를 수행하여야 한다

[12. 시스템 또는 구성 설정의 외부 제어]

-정의: 외부에서 시스템 설정 또는 구성 요소를 제어할 수 있다. 이로 인해 서비스가 중단될 수 있고 예상치 못한 결과가 발생하거나 악용될 가능성이 있다.

- 안전한 코딩기법

I) sethostid() 함수의 인수로 외부입력을 직접 사용하지 않는 것이 바람직

```
#include <stdlib.h>
#include <unistd.h>
main(int argc, char *argv[])
{
    sethostid(atol(argv[1]));
}
```

* atol : 문자열을 long형 정수로 변환하는 함수

```
#include <stdlib.h>
#include <unistd.h>
main(int argc, char *argv[])
{
    sethostid(0xC0A80101);
}
```

[13. 프로세스 제어]

-정의: 신뢰되지 않은 소스나 신뢰되지 않은 환경으로부터 라이브러리를 적재하거나 명령을 실행하면 악의적인 코드의 실행이 가능하다

- 안전한 코딩기법

I) 라이브러리 적재시 상대경로보다 신뢰할 수 있는 디렉터리의 절대경로를 사용한다

II) 신뢰되지 않은 라이브러리의 경우 소스코드를 검사한 다음 사용한다.

```
#include <stdio.h>
#include <dlfcn.h>
int main()
{
    char *filename;
    int *handle;
    filename = getenv("SHAREDFILE");
    /* RRLD_LAZY: 동적 라이브러리가 실행되면서 코드의 정의되지 않은 심볼 */
    if ((handle = dlopen(filename, RTLD_LAZY)) != NULL)
    {
        exit(1);
    }
    ...
    return 0;
}
```

* dlopen() 함수에 전달되는 인수인 filename이 환경변수 값에 의해 영향을 받는다

```
#include <stdio.h>
#include <dlfcn.h>
int main()
{
    char *filename;
    int *handle;
    /*절대경로 지정을 통해 해당경로에서만 라이브러리 찾을*/
    filename = "/usr/lib/hello.so";
    if ((handle = dlopen(filename, RTLD_LAZY)) != NULL);
    {
        exit(1);
    }
    return 0;
}
```

[14. 버퍼 시작 지점 이전에 쓰기]

-정의: 포인터나 인덱스를 통해서 버퍼 시작 지점 이전에 데이터를 기록하는 오류.

배열인 경우 인덱스의 최저 위치보다 적은 위치에 데이터를 기록하는 경우가 발생

- 안전한 코딩기법

I) 버퍼에 인덱스나 포인터를 사용해 데이터를 기록할 때 인덱스가 음수값이 되거나 포인터 연산의 결과가 버퍼 이전의 값을 가지지 않도록 점검 후 사용

```
int main()
{
    int a[10];
    a[-1] = 0; /* 음수 인덱스를 사용하여 버퍼에 접근함으로써 취약점을 발생시킨다 */
    return 0;
}
```

```
int main()
{
    int a[10];
    a[0] = 0; /* 버퍼의 시작 지점위치를 명확하게 제시한다 */
    return 0;
}
```

[15. 범위를 초과해서 읽기]

-정의: 버퍼의 범위를 벗어나서 읽는 경우에 발생하는 취약점.

프로그램이 최대의 인덱스 값보다 많은 값으로 배열을 참조하는 경우에 발생

* 대부분 인덱스의 최대값에 대한 유효성 점검없이 배열을 참조하는 경우 발생

- 안전한 코딩기법

I) 버퍼의 범위를 벗어난 곳에서 읽는 경우를 방지한다. 버퍼의 공간내의 읽기가 아닌 버퍼의 시작전의 위치나

버퍼의 공간뒤의 메모리를 읽게 되면 의도치 않은 오류가 발생한다.

```
int main()
{
    int a[3] = {1,2,3};    배열은 0부터 시작해 [2]까지의 범위를
    int b = a[3]; /* 배열이 가지는 범위를 벗어난 값을 읽어오고 있다 */
    return 0;
}
```

```
int main()
{
    int a[3] = {1,2,3};
    int b = a[2];
    return 0;
}
```

[16. 검사되지 않은 배열 인덱싱]

-정의: 외부 인자에 대한 적절한 검사 없이 그 인자가 배열 참조의 인덱스로 사용하는 경우이다

외부의 인자값이 버퍼의 인덱스로 사용이 되는 경우 해당 인덱스가 0보다 같거나 크고/ 버퍼의 크기보다 작은지

검사를 하고 사용할 것

- 배열인덱스를 사용시 신뢰할 수 없는 입력값을 사용하거나, 참조하는 배열 인덱스의 유효성 검사하지 않을 경우, 시스템 마비등의 문제를 발생시킨다.

- 안전한 코딩기법

I) 배열 정의시 인덱스의 최대치를 정함

II) 배열 참조시 인덱스가 배열의 유효값 내에 있는지 검사

III) 루프안에서 비교 구문을 쓸 때 '>' 보다 '>=' 사용

```
int getsizes(int sock, int count, int *sizes)
{
    char buf[BUFFER_SIZE];
    int ok;
    int num, size;

    while ((ok = gen_recv(sock, buf, sizeof(buf))) != 0)
    {
        if (hasDotInBuffer(buf))
        {
            break;
        }

        else if (sscanf(buf,"%d %d", &num, &size) == 2)
        {
            sizes[num-1] = size;
        }
    }
    ...
}
```

```
int getsizes(int sock, unsigned int MAXCOUNT, int *sizes)
{
    char buf[BUFFER_SIZE];
    int ok;
    int num, size;

    while ((ok = gen_recv(sock, buf, sizeof(buf))) != 0)
    {
        if (hasDotInBuffer(buf)) break;
        // buf를 num, size로 입력받는다
        if (sscanf(buf,"%d %d", &num, &size) == 2)
        {
            // num의 최대치를 검사한다.
            if (num > 0 && num <= MAXCOUNT)
                sizes[num-1] = size;
            else { printf("에러"); return(FAIL); }
        }
    }
    ...
}
```

[17. 널 종료 문제]

-정의: * C에서 문자열은 문자의 배열, 문자배열의 끝을 나타내는 것이 널문자임

문자배열1,2,3,4.....	널
------------------	---

널 문자가 붙지 않은 문자열은 프로그램 작동시 여러 오류를 발생시킬수 있다

* 개발자의 의도와 달리 널 문자가 붙지 않은 문자열이 생성될 수 있는데,

- 1) 문자열이 필요한 크기보다 **작은 배열**에 문자열을 **복사**하는 경우
- 2) 문자열 strcpy() 함수를 사용해 복사할 때 실제 문자열보다 지정 크기가 작은 경우 등.

- 안전한 코딩기법

I) read(), readlink()로 읽은 문자열에 strcpy(), strcat(), strlen() 함수를 적용하면 **안된다**

II) strncpy() 나 strncat()과 같은 **버퍼 오버플로우 위험이 없는 함수를 사용한다.**

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define MAXLEN 1024
extern char *inputbuf;
extern int cfgfile;
void readstr()
{
    char buf[MAXLEN];
    read() 함수를 이용해 읽어들인 inputbuf 문자열은
    널문자로 끝나는 것이 보장되지 않는다
    read(cfgfile, inputbuf, MAXLEN);
    strcpy() 함수를 사용하면 문자열이 널 문자로 끝나는
    것으로 간주하므로 주어진 범위를 넘어서 메모리를 접근
    strcpy(buf, inputbuf);
}
```

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <string.h>
#define MAXLEN 1024
extern char *inputbuf;
extern int cfgfile;
void readstr()
{
    char buf[MAXLEN];
    read() 함수를 이용해 읽어들인 문자열의 수를 MAXLEN으로
    정의한 길이만큼 strcpy()함수를 사용하여 복사한다
    read(cfgfile, inputbuf, MAXLEN);
    strncpy(buf, inputbuf, MAXLEN);
}
```

[18. 의도하지 않은 부호 확장]

-정의: 큰수를 표현하는 데이터 형으로 값이 변환될 때 **음수 값이 큰 양수 값으로 변환될수 있다.**

숫자형 데이터를 처리할 때 해당 데이터 형보다 큰 **데이터 형으로 복사**되는 경우,

부호 확장이 수행된다. 특히 **음수값을 unsigned로 복사**하는 경우 잘못된 부호의 확장으로 엄청난 큰 숫자가 발생할수 있다

- 안전한 코딩기법

I) 큰 사이즈의 변수와 작은 사이즈의 변수 사이의 값 교환 시 타입 체크를 통해 잘못된 부호 확장 예방

```
extern int info[256];
int signed_overflow(int id)
{
    short s;
    unsigned sz;
    /* int형 변수값을 short형 변수에 저장함으로써 오버플로우 발생*/
    s = id;
    if (s > 256) return 0;
    /* 위의 값을 unsigned형 변수에 저장함으로써 부호 확장 발생*/
    sz = s;
    return info[sz];
}
```

크기가 다른 변수의 값을 서로 교환하지 못하도록 필요한 값의 조건을 정확하게 검사하고 부득이 하게 교환할 경우 형 변환을 통해 교환되는 값의 타입을 일치시켜야 한다

```
extern int info[256];
int signed_overflow(int id)
{
    int s;
    unsigned sz;
    s = id;
    if (s > 256 || s < 0)
    {
        return 0;
    }

    sz = (unsigned) s;
    return info[sz];
}
```

[19. 무부호 정수를 부호 정수로 타입 변환 오류]

-정의: 무부호 정수를 부호정수 (unsigned를 signed로) 변환하면서 **큰 양수가 음수로 변환** 될 수 있다
이 값이 배열의 인덱스로 사용되는 경우 **배열의 한계 범위를 넘어서 접근이 될 수도 있어** 프로그램 오동작이 발생

- 안전한 코딩기법

1) 타입체크를 해서 signed int를 묵시적으로 unsigned int 로 변환되지 않도록 한다

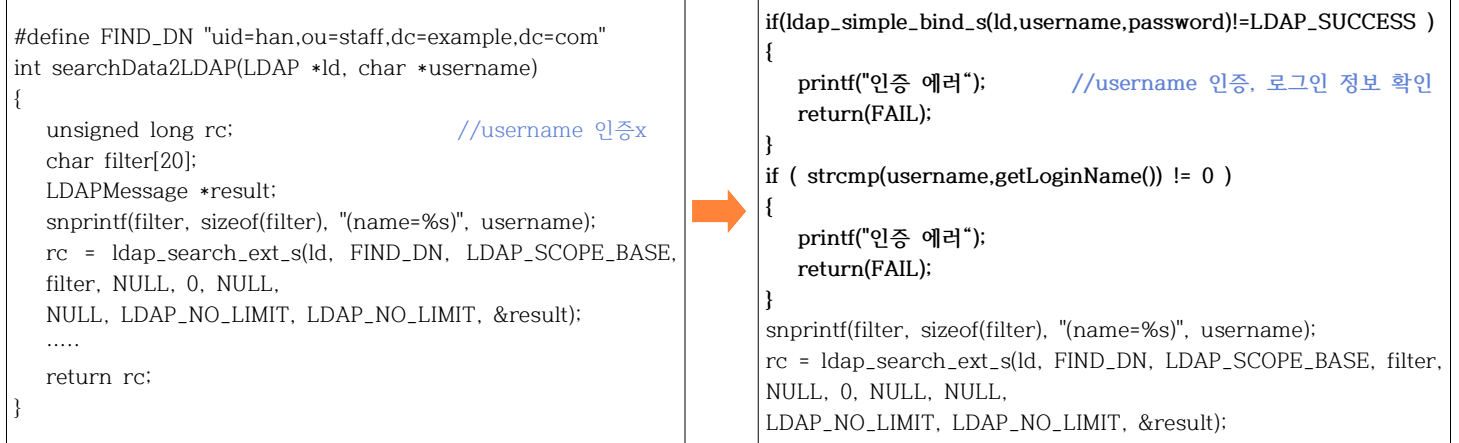
```
#include <stdlib.h>
#include <string.h>
extern int initialized, chunkSize;
int chunkSz()
{
    if (!initialized) return -1;
    return chunkSize;
}
void* chunkCpy(void *dBuf, void *sBuf)
{
    unsigned size;
    size = chunkSz();
    return memcpy(dBuf, sBuf, size);
}
```

```
#include <stdlib.h>
#include <string.h>
extern int initialized, chunkSize;
int chunkSz()
{
    if (!initialized) return -1;
    return chunkSize;
}
void* chunkCpy(void *dBuf, void *sBuf)
{
    Int size;
    size = chunkSz();
    if (size < 0) return NULL;
    return memcpy(dBuf, sBuf, (unsigned)size);
}
```

제 2절
보안기능

[1. 부적절한 인가]

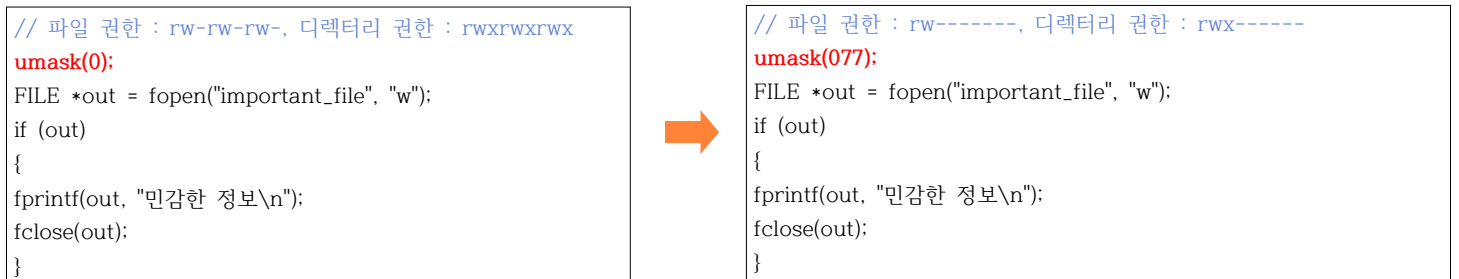
-SW가 실행 경로에 대해서 접근제어를 검사하지 않거나 불완전하게 검사하는 경우, 공격자는 접근 가능한 실행 경로를 통해 접근하여 정보를 유출할 수 있다.



- ▶ 사용자 인증이 성공적으로 끝나면 데이터베이스 접근이 가능하기 때문에 **보안이 취약**
 ->응용프로그램이 제공하는 정보와 기능을 역할에 따라 배분함으로써 공격자에게 노출되는 공격표면을 감소시킴

[2. 중요한 자원에 대한 잘못된 권한허용]

-중요한 보안관련 자원에 대하여 읽기,쓰기 권한을 의도하지 않게 허가할 경우, 권한이 없는 사용자가 해당자원 사용 가능하게 됨



```
// 보호 받아야 하는 file이면 관리자만 열 수 있도록 함
set<string> protectedFiles;
int main()
{
    protectedFiles.insert("/etc/password");
    protectedFiles.insert("/root/keys.cfg");
    ...
    FILE * fp = secureFileOpen("/etc/password/", "r", "normal_user");
    ...
}
FILE * secureFileOpen(const char * filename, const char * mode, const char * userId)
{
    if(protectedFiles.count(filename) > 0){           // file이 보호 받아야 하는 file이면
        if(isAdminUserId(userId) == FALSE){
            return NULL;
        }
    }
    return fopen(filename, mode);
}
```

- ▶사용자를 제외하고는 읽기, 쓰기가 가능하지 않도록 권한을 설정해야 안전하다.

[3. 취약한 암호화 알고리즘 사용]

-정보보호 측면에서 취약하거나 위험한 암호알고리즘을 사용해서는 안된다. -> 공격자가 알고리즘을 분석하여 무력화시킬 수 있음
컴퓨터가 발전함에 따라 해독이 어려운 알고리즘도 최근에는 쉽게 해독이 가능하기도 함. 예)RC2, RC4, RC5, MD4, DES 등

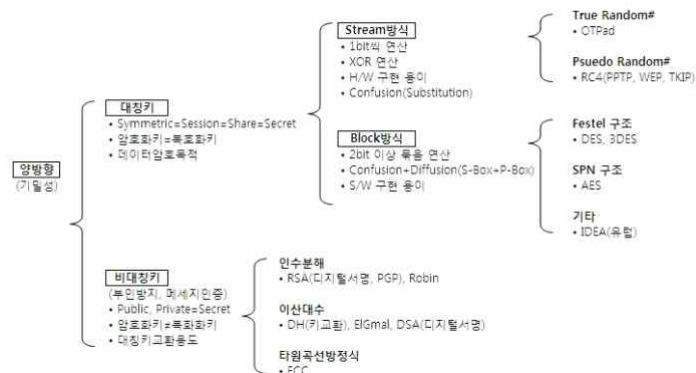
▶안전한 코딩기법

- 3-DES, AES와 같은 알고리즘 사용
- 검증된 표준화된 알고리즘 사용
- 업무관련 내용, 개인정보등에 대한 암호알고리즘 적용 시, IT보안인증 사무국이 안전성을 확인한 검증필 암호모듈을 사용

▶암호화 알고리즘 정의와 분류

- 평문(Plaintext) : 해독 가능한 형태의 메시지(암호화전 메시지)
- 암호문(Ciphertext) : 해독 불가능한 형태의 메시지(암호화된 메시지)
- 암호화(Encryption) : 평문을 암호문으로 변환하는 과정
- 복호화(Decryption) : 암호문을 평문으로 변환하는 과정
- 전자서명
 - 송신자의 Private Key로 메시지를 서명하여 전달
 - 수신자측에서는 송신자의 Public Key를 이용하여 서명값을 검증
- 양방향암호화 : 암호화와 복호화과정을 통해 송.수신간 주고받는 메시지를 안전하게 암호복호화하는 과정
- 단방향암호화 : **해싱(Hashing)**을 이용한 암호화 방식으로 양방향과는 다른 개념으로, 평문을 암호문으로 암호화는 가능하지만 암호문을 평문으로 복호화 하는 것은 불가능.

※**해싱** : 하나의 문자열을 보다 빨리 찾을 수 있도록 주소에 직접 접근할 수 있는 짧은 길이의 값이나 키로 변환하는 것. 문자열을 찾을 때 문자 하나하나를 비교하며 찾는 것보다는 문자열에서 해시 키를 계산하고 그 키에 해당하는 장소에 문자열을 저장해 둔다면, 찾을 때는 한 번의 해시 키 계산만으로도 쉽게 찾을 수 있게 된다.



대칭키: 어떤 키로 암호화를 하면 같은 키로 복호화를 할 수 있다.

비대칭키: A키로 암호화를 하면 오직 B키로만 복호화를 할 수 있고 반대로 B키로 암호화를 하면 A키로만 풀 수 있다.

출처: https://velog.io/@minj9_6/대칭키와-비대칭키는-무슨-차이가-있을까

[4. 사용자 중요정보 평문 저장(또는 전송)]

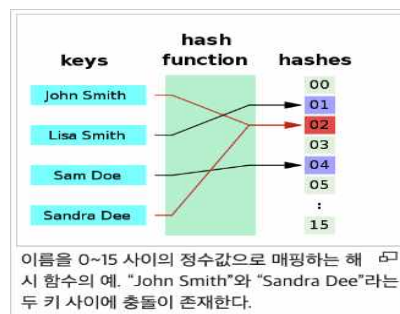
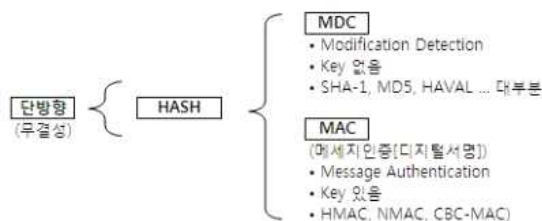
※**스니핑** : 송신자와 수신자 사이 네트워크에서 패킷 정보를 도청하는 행위

-보안과 관련된 민감한 데이터를 평문으로 통신채널을 통해 송수신할 경우, 통신채널 **스니핑**을 통해 인가되지 않은 사용자에게 민감한 데이터가 노출될 수 있는 보안 취약점

▶개인정보, 금융정보, 패스워드 등을 반드시 암호화하여 저장, 중요정보를 통신 채널로 전송할 때에도 암호화하여 정보를 보호

▶패스워드 암호화 → 해시 함수 알고리즘을 사용하여 패스워드를 암호화하면 사용자 외에는 아무도 패스워드를 알 수 없다.

- 해시 함수 : 임의의 길이를 갖는 메시지를 입력받아 고정된 길이의 해시값을 출력하는 함수, 암호알고리즘에는 키가 사용되지만, 해시 함수는 키를 사용하지 않으므로 같은 입력에 대해서는 항상 같은 출력이 나오게 된다.
- 해시 함수 사용목적 : 메시지의 오류나 변조를 탐지할 수 있는 무결성을 제공하기 위해 사용된다.



※출처: https://velog.io/@inyong_pang/Programming-암호화-알고리즘-종류와-분류

<https://steemit.com/kr/@yahweh87/2>

[5. 하드코딩된 패스워드]

-프로그램 코드 내부에 **하드코딩**된 패스워드를 포함하고, 이를 이용하여 내부 인증에 사용하거나 외부 컴포넌트와 통신하는 경우 관리자 정보가 노출될 수 있어 위험하다.

※**하드코딩** : 상수나 변수에 들어가는 값을 소스코드에 직접 쓰는 방식. 모바일 앱 실행 시 사용자에게 입력받아야 할 정보를 소스코드에 입력하거나 변수, 아이디, 비밀번호, 대칭키 등 중요 정보를 주석 처리하는 것도 하드코딩이다.

```
public Connection DBConnect(String url, String id) {
    try {
        conn = DriverManager.getConnection(url, id, "tiger");
    } catch (SQLException e) {
        System.err.println(e);
    }
    return conn;
}

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlext.h>
int dbaccess(char *server, char *user)
{
    SQLHENV henv;
    SQLHDBC hdbc;
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
    SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    /* 패스워드가 "asdf"로 하드코딩 되어있다.*/
    SQLConnect(hdbc, (SQLCHAR*) server, strlen(server), user,
    strlen(user), "asdf", 4); //공격에 쉽게 노출
    return 0;
}
```

패스워드
"tiger"
하드코딩



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlext.h>
int dbaccess(char *server, char *user, char *passwd)
{
    SQLHENV henv;
    SQLHDBC hdbc;
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE,
    &henv);
    SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    SQLConnect(hdbc, (SQLCHAR*) server, strlen(server), user,
    strlen(user), passwd, strlen(passwd));
    SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
    SQLFreeHandle(SQL_HANDLE_ENV, henv);
    return 0;
} //패스워드를 외부에서 얻어오는 경우 암호화를 사용하여 관리해
야 하며 적절한 검증과정을 거쳐야 한다.
```

▶ 외부 컴포넌트는 설정파일에 암호화되어 저장된 패스워드를 사용한다.

▶ 서버에서 관리자를 인증할 때 설정파일이나 데이터베이스에 **일방향 함수**로 암호화되어 저장된 패스워드를 사용한다.

※**일방향 함수** : 결과값이 주어졌을 때 입력값을 구하는 것이 어려운 함수

[6. 충분하지 않은 키 길이 사용]

-길이가 짧은 키를 사용하는 것은 암호화 알고리즘을 취약하게 만들 수 있다. → 짧은 시간 안에 키를 찾아낼 수 있고 이를 이용 해 공격자가 암호화된 데이터 복호화가 가능해짐

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>
EVP_PKEY *RSAKey()
{
    EVP_PKEY *pkey;
    RSA *rsa;
    /* 키값이 512bit로 너무 짧다 */
    rsa = RSA_generate_key(512, 35, NULL, NULL);
    if (rsa == NULL)
    {
        printf("Error\n");
        return NULL;
    }
    pkey = EVP_PKEY_new();
    EVP_PKEY_assign_RSA(pkey, rsa);
    return pkey;
}
```



```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>
EVP_PKEY *RSAKey()
{
    EVP_PKEY *pkey;
    RSA *rsa;
    /* 키의 길이는 최소 2048Bit 이상이어야 한다. */
    rsa = RSA_generate_key(2048, 35, NULL, NULL);
    if (rsa == NULL)
    {
        printf("Error\n");
        return NULL;
    }
    EVP_PKEY *pkey = EVP_PKEY_new();
    EVP_PKEY_assign_RSA(pkey, rsa);
    return pkey;
}
```

▶ RSA 알고리즘 → 2048비트 이상의 길이를 가진 키와 함께 사용해야 안전하다.

대칭암호화 알고리즘 → 최소 128비트 이상의 키를 사용해야 안전하다.

[7. 적절하지 않은 난수 값의 사용]

- 예측 가능한 난수의 사용은 시스템 취약점을 유발한다. → SW에서 생성되는 다음 숫자를 예상하여 시스템 공격이 가능해짐
▶랜덤 seed를 변경하여 사용하도록 설계한다.

/* 랜덤함수 생성에서는 seed를 사용하지 않거나, 상수 seed를 사용하거나, 예측 가능한 seed를 사용하는 경우 충분한 난수값을 얻을 수 없다. */

```
#include <stdafx.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
    int count = 0;
    int temp;
    printf("\n%s\n%s\n",
        "Some randomly distributed integers will be printed.",
        "How many do you want to see? ");
    /* srand()에서 상수 형태의 seed를 사용하고 있다. */
    srand( 100 );
    while ( 1 )
    {
        if ( count % 6 == 0) printf("%s", "\n");
        temp = rand()%101;
        if( temp != 100 )
            count++;
        else
            break;
        printf("%5d", temp );
    }
    printf("\n100이 나오기 전까지 나온 숫자의 개수 : %d \n" , count );
    return 0;
}
```



/* 랜덤 seed를 변경하여 사용할 수 있도록 프로그램 설계 */

```
#include <stdafx.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
    int count = 0;
    int temp;
    int randNum = 0;
    printf("\n%s\n%s\n",
        "Some randomly distributed integers will be printed.",
        "How many do you want to see? ");
    printf("%s\n", "Init random number : ");
    /* 임의의 랜덤값을 받아옴 */
    randNum = getch();
    while ( 1 )
    {
        if ( count % 6 == 0)
            printf("%s", "\n");
        /* 랜덤 seed를 임의의 값으로 변경 */
        srand(randNum);
        temp = rand()%101;
        if( temp != 100 )
            count++;
        else
            break;
        printf("%5d", temp );
    }
    printf("\n100이 나오기 전까지 나온 숫자의 개수 : %d \n" , count );
    return 0;
}
```

```
int main()
{
    const int RANDOM_RANGE = 200;
    srand(time(NULL));
    ...
    int randomNum = rand() % RANDOM_RANGE;
    ...
    return 0;
}
```



/* 메모리 할당 시 무작위한 값이 초기화되어 있지 않은 상태로 들어가 있는 것을 이용 */

```
int main()
{
    const int RANDOM_RANGE = 200;
    srand(time(NULL));
    ...
    int randomNum = SufficientRand() % RANDOM_RANGE;
    ...
    return 0;
}

unsigned int SufficientRand()
{
    int size = rand() % 100;
    // malloc은 할당된 메모리를 초기화 하지 않기 때문에 어떤 값이 들어 있을지 예측할 수 없다. 그 상태에서 배열의 무작위 위치로부터 시드값을 뽑아온다.
    unsigned int * unknownMem = (unsigned int *)malloc(size * sizeof(unsigned int));
    int index = rand() % size;
    srand(unknownMem[index]);

    return rand();
}
```

[8. 패스워드 평문 저장]

-패스워드를 암호화되지 않은 텍스트로 저장 → 환경설정 파일에 접근할 수 있는 사람 누구나 패스워드를 알아낼 수 있다.

```
/* 패스워드를 파일에서 읽어서 직접 DB연결에 사용->암호화X */
int dbaccess()
{
    FILE *fp; char *server = "DBserver";
    char passwd[20];
    char user[20];
    SQLHENV henv;
    SQLHDBC hdbc;
    fp = fopen("config", "r");
    fgets(user, sizeof(user), fp);
    fgets(passwd, sizeof(passwd), fp);
    fclose(fp);
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE,
    &henv);
    SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    SQLConnect(hdbc,
    (SQLCHAR*) server,
    (SQLSMALLINT) strlen(server),
    (SQLCHAR*) user,
    (SQLSMALLINT) strlen(user),
    /* 파일로부터 패스워드를 평문으로 읽어들이고 있다. */
    (SQLCHAR*) passwd,
    (SQLSMALLINT) strlen(passwd) );
    return 0;
}
```



```
/* 외부에서 입력된 패스워드 검증 */
int dbaccess()
{
    FILE *fp;
    char *server = "DBserver";
    char passwd[20];
    char user[20];
    char *verifiedPwd;
    SQLHENV henv;
    SQLHDBC hdbc;
    fp = fopen("config", "r");
    fgets(user, sizeof(user), fp);
    fgets(passwd, sizeof(passwd), fp);
    fclose(fp);
    verifiedPwd = verify(passwd);
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE,
    &henv);
    SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    SQLConnect(hdbc,
    (SQLCHAR*) server,
    (SQLSMALLINT) strlen(server),
    (SQLCHAR*) user,
    (SQLSMALLINT) strlen(user),
    /* 파일로부터 암호화된 패스워드를 읽어들이고 있다. */
    (SQLCHAR*) verifiedPwd,
    (SQLSMALLINT) strlen(verifiedPwd ) );
    return 0;
}
```

▶ 패스워드는 암호화해서 파일로 저장하도록 설계한다.

[9. 하드코딩된 암호화 키]

-코드 내부에 하드코딩된 암호화키를 사용하여 암호화를 수행하면 정보가 유출될 가능성이 높아진다.

→ 많은 개발자들이 코드 내부의 고정된 패스워드의 해쉬를 계산하여 저장하는 것이 안전하다고 생각한다.

→ 많은 해쉬 함수들이 역계산이 가능하며, 적어도 **brute-force 공격**에는 취약하다는 것을 고려해야만 한다.

※**브루트 포스 공격**: 브루트 포스는 조합 가능한 모든 경우의 수를 다 대입해보는 것이다. 단순한 방법으로 보이기도 하지만 사실 정확도 100%를 자랑한다.

▶ 암호화가 되었더라도 패스워드를 상수의 형태로 프로그램 내부에 저장하여 사용하면 안된다.

- 대칭형 알고리즘 : AES, ARIA, SEED 3DES 사용 권고
- 비대칭형 알고리즘 : 키 길이 2048bit 이상 권고
- 해쉬 함수 : MD4, MD5, SHA1은 사용하지 않아야 한다.

```
extern char *salt;
typedef int SQLSMALLINT;
int dbaccess(char *user, char *passwd)
{
    char *server = "DBserver";
    char *cpasswd;
    SQLHENV henv;
    SQLHDBC hdbc;
    SQLAllocHandle(SQL_HANDLE_ENV,      SQL_NULL_HANDLE,
    &henv);
    SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    cpasswd = crypt(passwd, salt);

    /* 암호화 키가 소스코드내에 상수로 하드코드 되어 있다. */
    if (strcmp(cpasswd, "68af404b513073582b6c63e6b") != 0)
    {
        printf("Incorrect password\n");
        return -1;
    }
    .....
}
```



```
/* 암호를 적절한 검증 과정을 거쳐서 얻어오는 별도의 모듈을 작성
하고,
이를 암호화된 스트링과 비교하도록 코드가 작성되어야 함 */
extern char *salt;
typedef int SQLSMALLINT;
char* getPassword()
{
    static char* pass="password";
    return pass;
}
int dbaccess(char *user, char *passwd)
{
    char *server = "DBserver";
    char *cpasswd;
    char* storedpasswd;
    SQLHENV henv;
    SQLHDBC hdbc;
    SQLAllocHandle(SQL_HANDLE_ENV,      SQL_NULL_HANDLE,
    &henv);
    SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    cpasswd = crypt(passwd, salt);
    storedpasswd = getPassword();
    if (strcmp(cpasswd, storedpasswd) != 0)
    {
        printf("Incorrect password\n");
        SQLFreeHandle(SQL_HANDLE_DBC, &hdbc);
        SQLFreeHandle(SQL_HANDLE_ENV, &henv);
        return -1;
    }
    .....
}
```

[10. 주석문 안에 포함된 패스워드 등 시스템 주요정보]

-패스워드를 주석문에 넣어두면 시스템 보안이 훼손될 수 있다. → SW가 완성된 후에 주석을 제거하는 것이 매우 어려움, 또한 공격자가 소스코드에 접근하거나 **역공학(Reverse Engineering)**을 통해 시스템의 패스워드를 쉽게 확인할 수 있음.

▶개발 시에 주석 부분에 남겨놓은 패스워드 및 보안에 관련된 내용은 개발이 끝난 후에는 반드시 제거해야 한다.

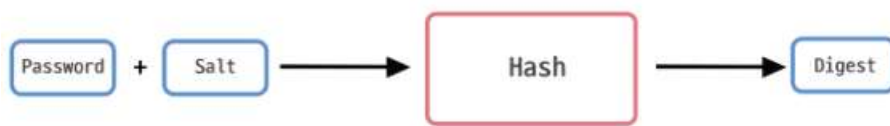
※**역공학** : 완성된 제품을 상세하게 분석하여 그 기본적인 설계내용을 추적하는 것, 소프트웨어에서는 기계어 코드로부터 원시 소스코드로 변환하는 기술을 특화하는 것을 의미한다.

```
1: #include <stdio.h>
2: #include <stdlib.h> //주석에 패스워드를 기입하지 않는다.
3: /* default password is "abracadabra" 주석문안에 패스워드가 포함되어있다 */
4: int verifyAuth(char *ipasswd, char *orgpasswd)
5: {
6:     char* admin="admin";
7:     if (strcmp(ipasswd, orgpasswd, sizeof(ipasswd)) != 0)
8:     {
9:         printf("Authetication Fail!\n");
10:    }
11:    return admin;
12: }
```

[11. 솔트 없이 일방향 해쉬 함수 사용]

- 패스워드를 저장 시 일방향 해쉬 함수의 성질을 이용하여 패스워드의 해쉬값을 저장한다.
 - 만약 패스워드를 솔트 없이 해쉬하여 저장한다면, 공격자는 레인보우 테이블과 같이 가능한 모든 패스워드에 대해 해쉬값을 미리 계산하고, 이를 전수조사에 이용하여 패스워드를 찾을 수 있게 된다.
- ▶ 안전한 코딩 기법 : 패스워드를 저장 시 패스워드와 솔트를 해쉬 함수의 입력으로 하여 얻은 해쉬값을 저장한다.
- ▶ 솔트란?
 - 해시 함수를 사용하여 단방향 암호화를 했을 때, 언제나 같은 값을 갖게 된다. → 해커는 잘 알려진 해시값의 원문을 저장해 두고 쉽게 원문을 유추할 수 있다.(해시 함수를 사용하여 만들어낼 수 있는 값들을 저장해둔 테이블 → 레인보우 테이블)
 - 솔트는 이러한 콀수를 피하기 위해 해시된 값에 추가적으로 들어가는 랜덤 데이터이다.
 - 예시)

사용자 이름	비밀번호	사용자 이름	솔트값	해시될 문자열	해시된 값 = SHA256 (비밀번호 + 솔트값)
user1	password123	user1	E1F53135E559C253	password123E1F53135E559C253	72AE25495A7981C4D622D49F9A52E4F1565C90F048F59027BD9C8C8900D5C3D8
user2	password123	user2	84B03D034B409D4E	password12384B03D034B409D4E	B4B6603ABC670967E99C7E7F1389E40CD16E78AD38EB1468EC2AA1E62B8BED3A



→ 각기 다른 솔트값들은 완전히 다른 해시 처리된 값을 만들어내며 이는 평문 비밀번호가 완전히 동일하더라도 마찬가지이다

※출처: [https://ko.wikipedia.org/wiki/솔트_\(암호학\)](https://ko.wikipedia.org/wiki/솔트_(암호학))
<https://jizard.tistory.com/340>

※패스워드의 암호화와 저장 - 해시와 솔트 정리 : <https://st-lab.tistory.com/100>

[12. 하드코딩된 사용자 계정]

- 코드 내부에 고정된 사용자 계정 이름을 포함하고, 이를 이용하여 내부 인증에 사용하거나 외부 컴포넌트와 통신 하는 것은 위험
 - 코드 내부에 하드코딩된 사용자 계정이 인증 실패를 야기하게 되면 원인을 찾아내기가 매우 힘들. 원인 파악이 되더라도 하드코딩된 패스워드를 수정해야 하기 때문에 SW 시스템 전체를 중지시켜 해결해야 하는 경우가 발생할 수 있다.
- ▶ 사용자 계정이나 패스워드를 코드 내부에 하드코딩하여 사용하지 않고 검증과정을 거쳐서 얻은 값을 사용해야 한다.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlext.h>
int dbaccess(char *server, char *passwd)
{
    SQLHENV henv;
    SQLHDBC hdbc;
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE,
    &henv);
    SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    SQLConnect(hdbc,
                (SQLCHAR*) server,
                (SQLSMALLINT) strlen(server),
                /*사용자 이름을 코드에 상수로 사용하는 경우*/
                "root",
                4,
                passwd,
                strlen(passwd));
    return 0;
}
  
```



```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlext.h>
int dbaccess(char *server, char *user, char *passwd)
{
    SQLHENV henv;
    SQLHDBC hdbc;
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE,
    &henv);
    SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    SQLConnect(hdbc,
                (SQLCHAR*) server,
                strlen(server),
                user,
                strlen(user),
                passwd,
                strlen(passwd));
    return 0;
}
/* 이름과 패스워드는 검증과정을 거쳐서 얻어지도록 설계 */
  
```

[13. 잘못된 권한 부여]

-SW의 일부분에서 의도하지 않게 잘못된 권한을 부여해 SW의 특정부분을 수행할 때 상위 권한을 가지게 되는 취약점이다.

▶ 응용프로그램을 실행할 때는 사용자의 권한만을 통해 수행되도록 설계한다.

```

1: #include <stdio.h>
2: #include <sys/types.h>
3: #include <unistd.h>
4: char buf[100];
5: char *privilegeUp()
6: {
7:     FILE *fp;
8:     /* 사용자 UID 생성 */
9:     setuid(0); //잘못된 권한 부여는 하지 않는다.
10:    fp = fopen("/etc/passwd", "r");
11:    fgets(buf, sizeof(buf), fp);
12:    /* 현재 프로세스의 유효 사용자 UID를 획득 */
13:    setuid(getuid());
14:    fclose(fp);
15:    return buf;
16: }
17: //사용자 권한을 높이지 않고 접근가능한
18: //자원만을 사용하도록 설계해야 한다.
19: int main()
20: {
21:     printf("비밀번호의 내용을 찍어보아요.\n");
22:     char *buffer=privilegeUp();
23:     printf("비밀번호의 내용은 %s입니다.\n",buffer);
24:     return 0;
25: }

```

```

21: ...
22: if(isCorrectUser(userId, password))
23: {
24:     setuid(0); //권한에 대한 체크없이 정상적인 유저인지에
25:     ... 대한 사항만 체크
26: }

```



```

7: map<string, uid_t> userGroupMinimumEffectUid;
8: userGroupMinimumEffectUid["Normal"] = 1;
9: userGroupMinimumEffectUid["Admin"] = 0;
10: userGroupMinimumEffectUid["Free"] = 2;
11: ...
27: if(isCorrectUser(userId, password))
28: {
29:     String userGroup = GetUserGroup(userId);
30:     setuid(userGroupMinimumEffectUid[userGroup]);
31:     ... //각 유저의 권한을 미리 정의해두고 그에 맞춘
32:     } 권한에 따른 작업만을 허용

```

[14. 최소 권한 적용 위배]

-필요에 따라서 상위 권한을 가지고 수행되어야 할 부분이 있을 수 있으며, 작업이 끝난 후에는 본래의 권한으로 되돌아 가야 함.

→ 그렇지 않을 경우 의도하지 않게 상위 권한을 가지고 수행하게 되는 부분이 나타날 수 있다.

▶ 권한을 높여 수행이 완료된 직후에는 원래 권한으로 바로 복귀시켜야 한다.

```

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#define AAP_HOME "/var/tmp"
char buf[100];
char *privilegeDown()
{
    FILE *fp;
    /* root 권한을 주고 작업이 끝났으나 복귀시키지 않음 */
    chroot(APP_HOME); //chroot() 함수는 root권한이 필요
    chdir("/");
    fopen("important_file", "r");
    fgets(buf, sizeof(buf), fp);
    fclose(fp);
    return buf;
}

```



```

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#define AAP_HOME "/var/tmp"
char buf[100];
char *privilegeDown()
{
    FILE *fp;
    chroot(APP_HOME);
    chdir("/");
    fp=fopen("important_file", "r");
    fgets(buf, sizeof(buf), fp);
    /* 작업 후 원래 사용자 권한으로 복귀 */
    setuid(1);
    fclose(fp);
    return buf;
}

```

[15. 취약한 암호화 : 적절하지 못한 RSA 패딩]

-OAEP패딩을 사용하지 않고 RSA알고리즘을 이용하는 것은 위험하다.

→ 패딩 기법을 사용함으로써 패딩이 없는 RSA알고리즘의 취약점을 이용하는 공격을 막을 수 있게 된다.

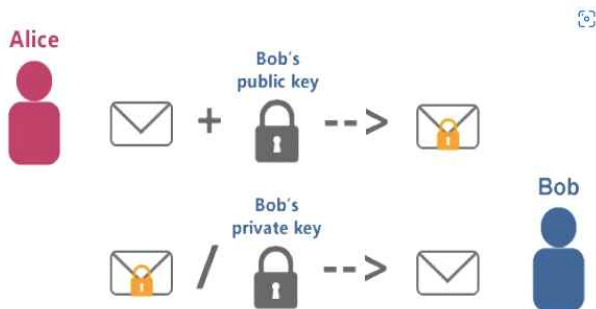
▶RSA_public_encrypt()함수에서 RSA_NO_PADDING파라미터를 사용하지 않는다.

▶RSA알고리즘이란?

· 공개키 암호시스템의 하나로, 암호화뿐만 아니라 전자서명(전자상거래)이 가능한 최초의 알고리즘으로 알려져 있다.

· 암호화/복호화를 목적으로 두 개의 다른 key를 사용하는 비대칭 암호화 알고리즘이다.(공개키→암호화, 개인키→복호화)

▶RSA 암호화 알고리즘의 예시



1. Alice 가 Bob 에게 정보를 안전하게 보내려 한다.
2. Bob 이 공개키와 개인키를 만들어 Alice 에게 공개키를 보낸다.
(개인키는 Bob 만 가지고 있다.)
3. Alice 가 Bob 로부터 받은 공개키로 정보를 암호화한다.
4. Alice 가 암호화된 정보를 Bob 에게 보낸다.
5. Bob 이 암호화된 정보를 받고 개인키로 암호를 해독한다.

출처: https://ko.wikipedia.org/wiki/RSA_암호 <https://blog.naver.com/mincoding/221717869646>

RSA에 대한 더 많은 정보 : <https://yishin.tistory.com/entry/암호학-비대칭키-암호-RSA-암호시스템>

▶최적 비대칭키 암호 패딩(OAEP)

- RSA-OAEP의 암호화에서는 평문 해시값과 정해진 개수의 '0'등으로 만들어진 “인증 정보”를 평문의 앞에 추가하고 RSA로 암호화한다.(패딩)
- RSA-OAEP의 복호화에서는 RSA로 복호화한 후 올바른 “인증 정보”가 나타나지 않으면 평문을 알고 있는 사람이 만든 암호문이 아니라고 판단해서 “decryption error”이라는 일종의 오류 메시지를 회신한다.
- 이와 같이 하면, 공격자가 RSA-OAEP의 복호 오라클로부터 정보를 얻을 수가 없기 때문에 선택 암호문 공격으로부터 안전하다.

※선택 암호문 공격 : 임의의 데이터를 송신하면 그것을 암호문으로 간주하고 회신해 주는 서비스(복호 오라클)를 공격자가 이용할 수 있다는 것을 가정한 공격이다. 서버가 회신해주는 오류 메시지나 타이밍을 해석하여 키나 평문 정보를 조금이라도 얻으려고 하는 것이다.

```

1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <openssl/rsa.h>
4: #define MAX_TEXT 512
5: void RSAEncrypt(char *text, int size)
6: {
7:     char out[MAX_TEXT];
8:     RSA *rsa_p = RSA_new();
9:     /* RSA함수에서 NO_PADDING과 같이 적절하지 않은 파라미터를 사용하는 경우*/
10:    RSA_public_encrypt(size, text, out, rsa_p, RSA_NO_PADDING);
11: }

```


↪ RSA_PKCS1_OAEP_PADDING

[16. 취약한 암호화 해쉬함수: 하드코딩된 솔트]

-코드에 고정된 솔트값을 사용하는 것은 모든 개발자가 그 값을 볼 수 있으며, 추후 수정이 어렵다는 점에서 시스템 취약점으로 작용할 수 있다. → 솔트값 유출 시 레인보우 테이블을 작성하여 해쉬값을 역으로 계산할 수 있음.

▶crypt()등의 암호화 함수에서 사용하는 솔트값을 상수로 사용하지 않도록 설계한다.

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <unistd.h>
4: #define MAX_TEXT 100
5: void hardSalt(const char *text)
6: {
7:     char *out;
8:     /* salt값으로 상수를 사용하고 있다 */
9:     out = (char*) crypt(text, "xp");
10: }
```

 //예측이 어려운 난수값으로 사용할 때마다 새로 만들어서 사용해야 한다.

void hardSalt(const char *text, const char *os)


out = (char *) crypt(text, os);

[17. 같은 포트번호로의 다중 연결]

-하나의 포트에 다수의 소켓이 연결되는 것을 허용하는 경우 → 포트에서 전달되는 패킷이 도난당하거나 도용당할 수 있음

▶소켓 옵션 SO_REUSEADDR와 서버 주소값 INADDR_ANY를 동시에 사용하지 않는다.

```
1: #include <stdio.h>
2: #include <sys/types.h>
3: #include <sys/socket.h>
4: #include <netinet/in.h>
5: void bind_socket(void)
6: {
7:     int server_sockfd;
8:     int server_len;
9:     struct sockaddr_in server_address;
10:    int optval;
11:    unlink("server_socket");
12:    server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
13:    optval = 1;
14:    setsockopt(server_sockfd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof optval);
15:    server_address.sin_family = AF_INET;
16:    server_address.sin_port = 21;
17:    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
18:    /* 소켓 옵션 SO_REUSEADDR과 서버 주소값 INADDR_ANY을 동시에 사용 */
19:    server_len = sizeof(struct sockaddr_in);
20:    bind(server_sockfd, (struct sockaddr *) &server_address, server_len);
21: }
```



```
5: void bind_socket(void)
6: {
7:     int server_sockfd;
8:     int server_len;
9:     struct sockaddr_in server_address;
10:    unlink("server_socket");
11:    server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
12:    server_address.sin_family = AF_INET;
13:    server_address.sin_port = 21;
14:    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
15:    server_len = sizeof(struct sockaddr_in);
16:    bind(server_sockfd, (struct sockaddr *) &server_address, server_len);
17:    closesocket(server_sockfd);
18: }
```

→ 소켓의 옵션을 SO_REUSEADDR으로 세팅하고, 서버 주소값으로 INADDR_ANY을 사용하는 경우 하나의 포트에 여러 개의 소켓이 바인딩 될 수 있다.

제 3절
시간 및 상태

<3절 시간 및 상태>

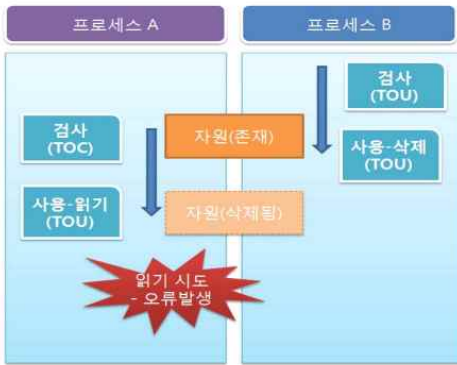
1. 경쟁 조건: 검사 시점과 사용시점(Time-of-check Time-of-use(TOCTOU) Race Condition)

- 경쟁 상태 : 프로세스가 어떤 순서로 데이터에 접근하나에 따라 결과값이 달라질 수 있는 상황, 둘 이상의 입력이나 조작이 동시에 일어나 의도하지 않은 결과를 가져오는 경우 (교착 상태의 종류 중에 하나) ¹⁾

1) 정의

하나의 자원에 대하여 검사시점(time of check)의 상태와 사용시점(time of use)의 상태가 달라 발생하는 취약점

=>동기화 오류, 교착 상태 등과 같은 문제점 발생



<그림 2-7> 경쟁조건 : 검사시점과 사용시점(TOCTOU)

2) 안전한 코딩 기법

- 파일 핸들 사용 권장 (파일 이름으로 권한 검사 후 오픈 시 보안에 취약)

핸들 : 자신의 리소스를 안전하게 관리하기 위해 운영체제 내부에 있는 어떤 리소스의 주소를 정수로 치환한 값²⁾

리소스 : 운영 체제 내에 들어 있는 장치의 구성요소나 설정값 데이터와 같은 시스템 내 장치를 제어하기 위한 여러 가지 정보들³⁾

- 상호배제(mutex)를 사용

상호배제 : 임계 구역을 어느 시점에서 단지 한 개의 프로세스만이 사용할 수 있도록 하며, 다른 프로세스가 현재 사용 중인 임계 구역에 대하여 접근하려고 할 때 이를 금지하는 행위⁴⁾

3) 예제

예제1)

■ 안전하지 않은 코드의 예 - C

```
1: #include <stdio.h>
2: #include <unistd.h>
3: void file_operation(char* file)
4: {
5:     /* 파일검사 후 파일열기를 수행 시 자원의 상태가 다를 수 있어 취약하다 */
6:     if(!access(file,W_OK)) → 파일의 존재 확인
7:     {
8:         f = fopen(file,"w+"); → 실제로 파일 오픈 후 사용
9:         operate(f);
10:    }
11:    else
12:    {
13:        fprintf(stderr,"Unable to open file %s.\n",file);
14:    }
15: }
```

존재 확인과 파일 오픈의 시간차가 발생하는 경우 예측불가한 수행 발생
ex)) 다른 파일 오픈

1) Jinho0705, “교착상태와 경쟁상태.” 2021-09-16,

<https://velog.io/@jinho0705/%EA%B5%90%EC%B0%A9-%EC%83%81%ED%83%9C%EC%99%80-%EA%B2%BD%EC%9F%81-%EC%83%81%ED%83%9C>

2) 김성엽, “핸들(Handle)에 대하여.” 2017-08-02,

<https://m.blog.naver.com/PostView.nhn?isHttpsRedirect=true&blogId=tipsware&logNo=221065382244&proxyReferer=>

3) Seung Hyun Tark, “[Window] 핸들Handle이란?.” 2019-05-14, <https://crobbit-lucy.tistory.com/7>

4) const_p, “[운영체제] 상호배제(Mutual Exclusion).” 2021-09-27, <https://overcome-the-limits.tistory.com/m/339>

■ 안전한 코드의 예 - C

```

1: #include <stdio.h>
2: #include <unistd.h>
3: void file_operation(char* file)
4: {
5:     char *file_name;
6:     int fd;
7:     /* 파일을 열 때, open()을 사용하여, mode를 지정한다. */
8:     fd = open( file_name, O_WRONLY | O_CREAT | O_EXCL, S_IRWXU);
9:     if (fd == -1)
10:    {
11:        ...
12:    }
13:    /* chmod 대신 fchmod를 사용하여, open의 fd를 그대로 사용한다. */
14:    if (fchmod(fd, S_IRUSR) == -1)
15:    {
16:        ...
17:    }
18:    close(fd);
19: }

```

파일 이름이 아닌 파일 핸들을 사용하여 파일 오픈

예제2)

■ 안전하지 않은 코드의 예 - C

```

1: void handle_mutex(pthread_mutex_t *mutex)
2: {
3:     ... ..
4:     pthread_mutex_lock(mutex);
5:     /* 임계코드 안에서 공유메모리 참조 */
6:     pthread_mutex_unlock(mutex);
7:     ... ..
8: }

```

→ 반환값에 대한 에러검사 X

-> pthread_mutex_lock()이 mutex를 얻어오는데 실패할 경우, 경쟁상태발생 (예측불가한 동작으로 이어짐)

■ 안전한 코드의 예 - C

```

1: int handle_mutex(pthread_mutex_t *mutex)
2: {
3:     int result;
4:     // mutex 사용 시에 함수 결과를 항상 체크
5:     result = pthread_mutex_lock(mutex);
6:     if (0 != result)
7:         return result;
8:     /* 공유메모리 참조 */
9:     return pthread_mutex_unlock(mutex);
10: }

```

-> 임계 코드에 대한 lock을 실행 후 결과 체크

예제 2-1) 로그인 기록

mutex 개념을 넣지 않은 로그 사용 중, 같은 작업 요청이 들어 올 경우, 같은 로그 파일에 대해 동시에 쓰기를 수행하여 오작동이 발생한다

-> log파일을 쓰기 전에 먼저 lock을 시켜놓고, 로그를 쓰도록 하여, 로그를 다 쓴 후에는 lock을 해제한다.

2. 제대로 제어되지 않은 재귀(Uncontrolled Recursion)

1) 정의

재귀의 순환 횟수를 제어하지 못하여 할당된 메모리나 프로그램 스택 등의 자원 고갈을 유발하여 시스템의 정상적인 서비스 제공불가

2) 안전한 코딩기법

- 모든 재귀 호출은 조건문 이나 반복문 내에서만 수행
- 일정 시간 안에 결과가 도출되지 않을 경우 강제로 재귀 탈출

3) 예제

예제1)

■ 안전하지 않은 코드의 예 - C

```
1: int fac(n)
2: {
3:     return n*fac(n-1);
4: }
```

무한 재귀가 될 수 있다

■ 안전한 코드의 예 - C

```
1: #include <stdio.h>
2: #include <unistd.h>
3: #include <string.h>
4: int i=0;
5: int fac(n)
6: {
7:     if (n <= 0)
8:         return 1;
9:     else
10:    {
11:        if(i>10000) exit(1);
12:        else i+=1;
13:        return n*fac(n-1);
14:    }
15: }
```

→ 10000번 이상 실행 시 강제 종료

→ 재귀 루프를 빠져나올 수 있도록 한다

3. 심볼릭명이 정확한 대상에 매핑되어 있지 않음(Symbolic Name not Mapping to Correct Object)

- 심볼릭 링크(symbolic link) : 링크를 연결하여 원본 파일을 직접 사용하는 것과 같은 효과를 내는 링크이다. 윈도우의 바로가기와 비슷한 개념⁵⁾

1) 정의

- TOUTOC문제와 같은 취약점으로 어떤 대상에 대한 심볼릭 참조가 시간에 따라 바뀌는 경우.
- 프로그램이 심볼릭명을 사용하여 특정 대상을 지정하는 경우 공격자는 심볼릭명이 가리키는 대상을 조작하여 프로그램이 원래 의도했던 동작을 못하게 할 수 있다.

2) 안전한 코딩기법

- 심볼릭 링크를 사용한 파일 바뀔치기를 염두해 두고 프로그래밍

3) 예제

예제1)

■ 안전하지 않은 코드의 예 - C

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: int main(int argc, char* argv[])
4: {
5:     char* file;
6:     FILE *f;
7:     /* 파일검사 후 파일을 열 때 생기는 시간에 심볼릭 링크를 변경할 수 있는 취약점이 존재 한다 */
8:     if(!access(file,W_OK))
9:     {
10:         f = fopen(file,"w+"); 시간차 발생
11:         operate(f);
12:         ..... 파일 이름을 조작하여 공격자가 파일을 변경 할 수 있다
13:     }
14:     else
15:     {
16:         fprintf(stderr,"Unable to open file %s.\n",file);
17:     }
18: }
```

■ 안전한 코드의 예 - C

```
1: #include <stdio.h>
2: #include <unistd.h>
3: #include <string.h>
4: int main (int argc, char* argv[])
5: {
6:     char* filename;
7:     /* 랜덤한 임시 파일 이름 생성*/
8:     if(mkstemp(filename)) → 파일이름을 유기적으로 생성
9:     {
10:         FILE* tmp = fopen(filename,"wb+");
11:         while((recv(sock,recvbuf,DATA_SIZE, 0) > 0) && (amt!=0))
12:             amt = fwrite(recvbuf,1,DATA_SIZE,tmp);
13:     }
14: }
```

예제1-1) 디렉토리에서 특정 파일 찾기

디렉터리나 파일 중 심볼릭 링크나 바로가기 같은 것이 존재하면 엉뚱한 결과 값을 얻거나 무한루프에 빠질 수 있다. 재귀 호출에서는 심볼릭 링크를 따라가지 않도록 막는 루틴이 필요하다.

5) qjadud22, "[Linux] 심볼릭 링크(Symbolic link)." 2018-09-22, <https://qjadud22.tistory.com/22>

제 4절
에러 처리

<4절 에러 처리>

- 정상적인 에러 : 사전에 정의된 예외사항이 특정 조건에서 발생하는 에러
- 비정상적인 에러 : 사전에 정의되지 않은 상황에서 발생하는 에러

에러 상황에 대비한 안전한 에러 처리 루틴을 정의해야 한다

에러를 부적절하게 처리하거나 전혀 처리하지 않을 경우 에러 정보가 과도하게 많은 정보를 포함하여 보안 취약점이 된다.

1. 오류 메시지를 통한 정보 노출(Information exposure through an error message)

1) 정의

응용 프로그램이 실행 환경, 사용자, 관련 데이터에 대한 민감한 정보(예외이름, 스택트레이스등)를 외부에 제공하여 프로그램 내부구조를 쉽게 파악 가능

2) 안전한 코딩기법

- 오류메세지 : 정해진 사용자에게 유용한 최소한의 정보만 포함
- 소스코드에서의 예외사항 : 에러 정보를 노출하지 않고, 내부적으로 처리한다. 미리 정의된 페이지를 제공한다.

3) 예제

예제1)

■ 안전하지 않은 코드의 예 - C

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <string.h>
4: int main (int argc, char* argv[])
5: {
6:     char* path=getenv("MYPATH");
7:     /* 공격자가 환경변수에 정의된 파일을 유추할 수 있다 */
8:     fprintf(stderr,path); →오류정보가 공개되는 부분 삭제
9:     return 0;
10: }
```

예제2)

■ 안전하지 않은 코드의 예 - C

```
1: public void ReadConfiguration()
2: {
3:     char buffer[BUFFER_SIZE];
4:     File * fp    = fopen("config.cfg", "r");
5:     if(fp == NULL)
6:     {
7:         printf("config.cfg를 찾을 수 없습니다.");
8:         return;
9:     }
10:    if(fgets(buffer, BUFFER_SIZE, fp) == NULL)
11:    {
```

```

12:     printf("config.cfg에서 option1을 읽을 수 없습니다.");
13: }
14: strcpy(configuration.option1, buffer);
15:
16: if(fgets(buffer, BUFFER_SIZE, fp) == NULL)
17: {
18:     printf("config.cfg에서 option2을 읽을 수 없습니다.");
19: }
20: strcpy(configuration.option2, buffer);
21:
22: if(fgets(buffer, BUFFER_SIZE, fp) == NULL)
23: {
24:     printf("config.cfg에서 option3을 읽을 수 없습니다.");
25: }
26: strcpy(configuration.option3, buffer);    에러 발생 위치와 세부내용 공개
27: }

```

■ 안전한 코드의 예 - C

```

1: public int _ReadConfiguration(Configuration * configuration)
2: {
3:     char buffer[BUFFER_SIZE];
4:     File * fp    = fopen("config.cfg", "r");
5:     if(fp == NULL){ return FALSE;    }
6:
7:     if(fgets(buffer, BUFFER_SIZE, fp) == NULL){    return FALSE;    }
8:     strcpy(configuration->option1, buffer);
9:
10:    if(fgets(buffer, BUFFER_SIZE, fp) == NULL){    return FALSE;    }
11:    strcpy(configuration->option2, buffer);
12:
13:    if(fgets(buffer, BUFFER_SIZE, fp) == NULL){    return FALSE;    }
14:    strcpy(configuration->option3, buffer);
15:    ...
16:    return TRUE;
17: }
18: public void ReadConfiguration(Configuration * configuration)
19: {
20:     if(!_ReadConfiguration(configuration) == FALSE)
21:     {
22:         printf("환경 설정에 실패하였습니다.");
23:     }
24: }

```

오류 정보의 공개는 최대한 간단하게 한다.

2. 오류 상황 대응 부재(Detection of Error Condition Without Action)

1) 정의

오류 가능성이 있는 부분에 대하여 예외처리를 하지 않은 경우

-> 프로그램이 충돌하거나 종료되는 등의 개발자가 의도하지 않은 결과 발생

2) 안전한 코딩기법

- 오류가 발생할 수 있는 부분에 대하여 제어문을 적절하게 예외처리한다.

(C/C++ => if문 또는 switch문, Java => try-catch)

3) 예제

예제1)

■ 안전한 코드의 예 - C

```
1: #include "std_testcase.h"
2:
3:
4: #ifndef OMITGOOD
5:
6: static void good1()
7: {
8:     {
9:         /* FIX: check the return value and handle errors properly */
10:        if (fputs("string", stdout) == EOF)
11:        {
12:            printLine("fputs failed!");
13:            exit(1);
14:        }
15:    }
16: }
17:
18: void KRD_402_Error_Without_Action__char_fputs_char_fputs_0101_good()
19: {
20:     good1();
21: }
22:
23: #endif /* OMITGOOD */
24:
25: /* Below is the main(). It is only used when building this testcase on
26: its own for testing or for building a binary to use in testing binary
27: analysis tools. It is not used when compiling all the testcases as one
28: application, which is how source code analysis tools are tested. */
```

```
29:
30: #ifdef INCLUDEMAIN
31:
32: int main(int argc, char * argv[])
33: {
34:     /* seed randomness */
35:     srand( (unsigned)time(NULL) );
36: #ifndef OMITGOOD
37:     printLine("Calling good()...");
38:     KRD_402_Error_Without_Action__char_fputs_char_fputs_0101_good();
39:     printLine("Finished good()");
40: #endif /* OMITGOOD */
41:     return 0;
42: }
43:
44: #endif
```


3. 적절하지 않은 예외처리(Improper Check for Unusual or Exceptional Conditions)

1) 정의

프로그램 수행 중 함수의 결과 값에 대한 적절한 처리 또는 예외상황에 대한 조건을 적절하게 검사하지 않을 경우, 예기치 않은 문제를 야기한다.

2) 안전한 코딩기법

- 값을 반환하는 모든 함수의 결과 값을 검사
- 예외 처리 사용 시, 광범위한 예외처리 대신 구체적인 예외처리

3) 예제

예제1)

■ 안전하지 않은 코드의 예 - C

```
1: char fromBuf[10], toBuf[10];
2: fgets(fromBuf, 10, stdin);
3: strcpy(toBuf, fromBuf);
4: .....
```

에러 발생에 의해 '\0'로 끝나지 않을 경우
strcpy()에서 오버플로우 발생 가능

■ 안전한 코드의 예 - C

```
1: char fromBuf[10], toBuf[10];
2: char *retBuf = fgets(fromBuf, 10, stdin);
3: // 함수호출 이후 결과 값을 비교한다.
4: if ( retBuf != fromBuf )
5: {
6:     printf("에러");
7:     return;
8: }
9: strcpy(toBuf, fromBuf);
10: ...
```

fromBuf와 retBuf 값 비교
-> strcpy()에 의해서 발생하는 버퍼 오버플로우 방지

예제2)

■ 안전하지 않은 코드의 예 - C

```
1: public void ReadConfiguration()
2: {
3:     char buffer[BUFFER_SIZE];
4:     File * fp = fopen("config.cfg", "r");
5:
6:     fgets(buffer, BUFFER_SIZE, fp);
7:     strcpy(configuration.option1, buffer);
8:
9:     fgets(buffer, BUFFER_SIZE, fp);
10:    strcpy(configuration.option2, buffer);
11:
12:    fgets(buffer, BUFFER_SIZE, fp);
13:    strcpy(configuration.option3, buffer);
14:    ...
15: }
16: }
```

데이터를 읽는 작업 실패에 대한 에러 처리가 없다

■ 안전한 코드의 예 - C

```
1: public int _ReadConfiguration(Configuration * configuration)
2: {
3:     char buffer[BUFFER_SIZE];
4:     File * fp = fopen("config.cfg", "r");
5:     if(fp == NULL){ return FALSE; }           파일이 없을 경우 처리
6:
7:     if(fgets(buffer, BUFFER_SIZE, fp) == NULL){ return FALSE; }
8:     strcpy(configuration->option1, buffer);
9:
10:    if(fgets(buffer, BUFFER_SIZE, fp) == NULL){ return FALSE; }
11:    strcpy(configuration->option2, buffer);
12:
13:    if(fgets(buffer, BUFFER_SIZE, fp) == NULL){ return FALSE; }
14:    strcpy(configuration->option3, buffer);
15:    ...
16:    return TRUE;
17: }
18:
19: public void ReadConfiguration(Configuration * configuration)
20: {
21:     if(_ReadConfiguration(configuration) == FALSE)
22:     {
23:         // Configuration을 읽어들이는 것을 실패할 경우, default 값으로 채운다.
24:         if(configuration.option1 == NULL)
25:         {
26:             configuration.option1 = DEFAULT_OPTION1;
```

```
27:     }
28:     if(configuration.option2 == NULL)
29:     {
30:         configuration.option2 = DEFAULT_OPTION2;
31:     }
32:     if(configuration.option3 == NULL)
33:     {
34:         configuration.option3 = DEFAULT_OPTION3;
35:     } _ReadConfiguration 함수의 리턴값이 false라면 3개의 작업 중
36: } 한 곳에서 에러가 발생함을 의미
37: } -> 모든 옵션을 디폴트값으로 초기화한다.
```

제 5절
코드 오류

<5절 코드 오류>

작성 완료된 프로그램은 기능성, 신뢰성, 사용성, 유지보수성, 효율성, 이식성을 충족하여야 한다.

너무 복잡한 코드의 경우 관리성, 유지보수성, 가독성이 떨어지며, 다른 시스템에 이식이 힘들다. 또한 안전성을 위협할 취약점들이 포함될 수 있다.

1. 널포인터 역참조(Null Pointer Dereference)

1) 정의

- Null 포인터 역참조는 ‘일반적으로 그 객체가 Null이 될 수 없다’라고 하는 가정을 위반할 경우 발생
- Null 포인터 역참조를 실행하는 경우, 그 결과가 발생하는 예외 사항을 추후에 공격 계획에 사용 가능
- 플로우 발생, 경쟁 상태, 프로그래밍 누락을 발생시킬 수 있다.⁶⁾

2) 안전한 코딩기법

- 변경될 수 있는 모든 포인터는 사용하기 전 점검한다.
(포인터 참조 응용프로그램에서 포인터가 Null이 아님을 예상하고 접근하는 경우에 프로그램의 충돌이 발생하여 종료가 일어남)
- Null이 될 수 있는 레퍼런스(reference)는 참조하기 전에 Null 값인지 검사 후 안전한 경우만 사용

3) 예제

예제1)

■ 안전하지 않은 코드의 예 - C

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <string.h>
4: #define BUFSIZE 100
5: int main()
6: {
7:     char *p = NULL;
8:     char cgi_home[BUFSIZE];
9:     p = getenv("CGI_HOME");
10:    strncpy(cgi_home, p, BUFSIZE-1);
11:    cgi_home[BUFSIZE-1] = '\0';
12:    return 0;
13: }
```

문자열로 설정된 값이 없을 경우
getenv() 함수에서 Null이 반환

■ 안전한 코드의 예 - C

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <string.h>
4: #define BUFSIZE 100
5: int main()
6: {
7:     char *p = NULL;
8:     char cgi_home[BUFSIZE];
9:     p = getenv("CGI_HOME");
10:    if (p == NULL)
11:    {
12:        exit(1);
13:    }
14:    strncpy(cgi_home, p, BUFSIZE-1);
15:    cgi_home[BUFSIZE-1] = '\0';
16:    return 0;
17: }
```

반환값이 Null인지 여부 검사 후 사용

6) "NULL pointer dereference issues can occur through a number of flaws, including race conditions, and simple programming omissions."

7 Pernicious Kingdoms, "CWE-476: NULL Pointer Dereference." 「CWE」. 2006-07-19.

<https://cwe.mitre.org/data/definitions/476.html>

예제2)

■ 안전한 코드의 예 - C

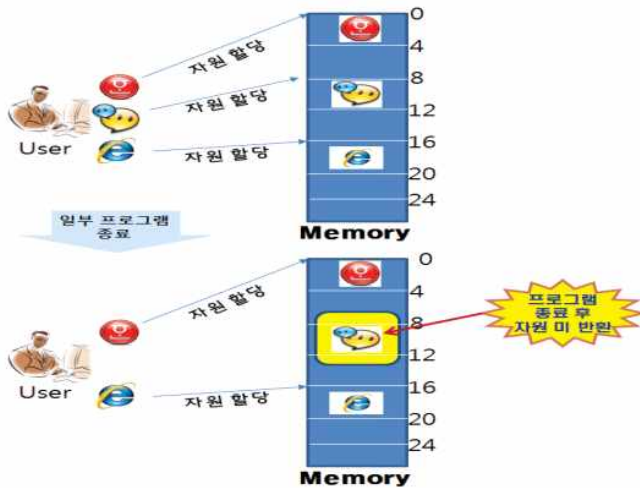
```
1: void host_lookup(char *user_supplied_addr)
2: {
3:     struct hostent *hp;
4:     in_addr_t *addr;
5:     char hostname[64];
6:     in_addr_t inet_addr(const char *cp);
7:
8:     if(user_supplied_addr == NULL) return;
9:
10:    /*routine that ensures user_supplied_addr is in the right format for conversion */
11:    validate_addr_form(user_supplied_addr);
12:    addr = inet_addr(user_supplied_addr);
13:    hp = gethostbyaddr( addr, sizeof(struct in_addr), AF_INET);
14:    if(hp == NULL) return;
15:    if(hp->h_name == NULL) return;
16:    strcpy(hostname, hp->h_name);
17: }
```

->함수의 입력값으로 사용하는 변수들은 Null여부를 체크 후 사용한다.

2. 부적절한 자원 해제(Improper Resource Shutdown or Release)

1) 정의

유한한 자원 (파일기술회자(open file descriptor), 힙 메모리(heap memory), 소켓(socket)등)을 할당받아 사용 후, 프로그램 오류 또는 예외로 사용이 끝난 자원을 반환하지 못 하는 경우



<그림 2-10> 부적절한 자원 해제

2) 안전한 코딩기법

- 자원을 할당하고 반환하는 함수들의 경우 모드 패스에서 한 쌍으로 존재하도록 강제한다.

3) 예제

예제1)

■ 안전한 코드의 예 - C

```
1: .....
2: void sqlDB()
3: {
4:     SQLHANDLE env_hd, con_hd;
5:     SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env_hd);
6:     SQLAllocHandle(SQL_HANDLE_DBC, env_hd, &con_hd);
7:     SQLFreeHandle(SQL_HANDLE_DBC, con_hd); //DBC = dbconnection
8:     SQLFreeHandle(SQL_HANDLE_ENV, env_hd); //ENV = environment
9: }
10: void serverSock()
11: {
12:     struct sockaddr_in serverAddr;
13:     struct sockaddr *server = (struct sockaddr *)&serverAddr;
14:     int listenFd = socket(AF_INET, SOCK_STREAM, 0);
15:     bind(listenFd, server, sizeof(serverAddr));
16:     listen(listenFd, 5);
17:     while (1)
18:     {
19:         int connectFd = accept(listenFd, (struct sockaddr *) NULL, NULL);
20:         shutdown(connectFd, 2);
21:         close(connectFd);
22:     }
23:     shutdown(listenFd, 2);
24:     close(listenFd);
25: }
```

ENV와 DBC 핸들 할당 후 사용이 끝난 뒤 반환

예제2)

■ 안전하지 않은 코드의 예 - C

```

1: int main(int argc, char * argv[])
2: {
3:     int listenSocket, connectionSocket;
4:     struct sockaddr_in serv_addr, cli_addr;
5:     socklen_t clien;
6:
7:     listenSocket = socket(AF_INET SOCK_STREAM, 0);
8:     ...
9:     while(TRUE)
10:    {
11:        connectionSocket = accept(listenSocket, (struct sockaddr *) &cli_addr, &clilen);
12:        if(fork() == 0)
13:        {
14:            comminucation();
15:        }
16:        ...
17:    }
18:    close(listenSocket);
19: }
20: void comminucation()
21: {
22:     exit(1); // socket을 반환하지 않고 process 종료
23: }

```

fork() 실패 시 소켓 자원 반환을 하지 않고 프로그램을 그대로 종료한다

■ 안전한 코드의 예 - C

```

1: int main(int argc, char * argv[])
2: {
3:     int listenSocket = -1, connectionSocket = -1;
4:     struct sockaddr_in serv_addr, cli_addr;
5:     socklen_t clien;
6:
7:     listenSocket = socket(AF_INET SOCK_STREAM, 0);
8:     if(listenSocket == -1) return 0;
9:     ...
10:    while(TRUE)
11:    {
12:        connectionSocket = accept(listenSocket, (struct sockaddr *) &cli_addr, &clilen);
13:        if(connectionSocket == -1)
14:        {
15:            break;
16:        }
17:        switch(fork())
18:        {
19:            case 0:
20:                listenSocket = -1;
21:                comminucation();
22:                break;
23:            case -1: // error case
24:                break;
25:            default:
26:                connectionSocket = -1;
27:        }
28:    }
29:
30:    // 어떠한 경우에도 열린 socket을 close하고 종료
31:    if(listenSocket != -1)
32:    {
33:        close(listenSocket);
34:    }
35:    if(connectionSocket != -1)
36:    {
37:        close(connectionSocket);
38:    }
39: }
40: void comminucation()
41: {
42:     ...
43: }

```

모든 경우의 쓰여진 자원들을 반환하고 프로그램 종료한다

3. 부호 정수를 무부호 타입 변환 오류(Signed to Unsigned Conversion Error)

1) 정의

부호 정수(signed integer)를 무부호 정수(unsigned integer)로 변환하는 과정에서 음수가 큰 양수로 변환될 수 있다. 이 값을 배열의 인덱스로 사용하는 경우, 배열의 범위를 넘어서 접근하거나, 정수 오버플로우로 인한 SW에 오작동이 발생한다.

2) 안전한 코딩기법

- 강한 타입체크 수행

3) 예제

■ 안전하지 않은 코드의 예 - C

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <string.h>
4: #define BUFSIZE 100
5: unsigned int len(char *s)
6: {
7:     unsigned int l = 0;
8:     if (s == NULL)
9:     {
10:         return -1;
11:     }
12:     l = strlen(s, BUFSIZE-1);
13:     return l;
14: }
15: int main(int argc, char **argv)
16: {
17:     char buf[BUFSIZE];
18:     unsigned int l = 0;
19:     l = len(argv[1]);
20:     strncpy(buf, argv[1], l);
21:     buf[l] = '\0';
22:     printf("last character : %c\n", buf[l-1]);
23:
24:     return 0;
25: }
```

len()의 입력 문자열이 Null 일 경우 -1이 반환
unsigned int로 변환 시 잘못된 값을 가진다

버퍼 오버플로우

또는 잘못된 값으로 인해 코드 다른 부분의 정확성에 영향을 미친다

■ 안전한 코드의 예 - C

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <string.h>
4: #define BUFSIZE 100
5: unsigned int len(char *s)
6: {
7:     unsigned int l = 0;
8:     if (s == NULL)
9:     {
10:         return 0;
11:     }
12:     l = strlen(s, BUFSIZE-1);
13:     return l;
14: }
15: int main(int argc, char **argv)
16: {
17:     char buf[BUFSIZE];
18:     unsigned int l = 0;
19:     l = len(argv[1]);
20:     if (l > 0)
21:     {
22:         strncpy(buf, argv[1], l);
23:         buf[l] = '\0';
24:         printf("last character : %c\n", buf[l-1]);
25:     }
26:     return 0;
27: }
```

len()의 입력 값이 Null인 경우 0을 반환

문자열의 길이가 0보다 큰 경우만 동작

4. 정수를 문자로 변환(Type Mismatch: Integer to Character)

1) 정의

4byte인 정수를 1byte인 문자에 값을 저장하는 경우 표현한 수 없는 범위의 값에 대해서는 값이 삭제되어 문자에 저장된 값이 제대로 된 값이 아닌 경우

-> SW에 영향을 미치며 코드의 오작동이 발생

2) 안전한 코딩기법

- 강한 타입체크 수행

3) 예제

예제1)

■ 안전하지 않은 코드의 예 - C

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: char mismatch(char bc)
4: {
5:     return bc;
6: }
7: char char_type()
8: {
9:     char bA;
10:    int iB;
11:    iB = 24;
12:    bA = iB;
13:    f(iB);
14:    printf("int = %d char = %d\n", iB, bA);
15:    return iB;
16: }
```

작은타입의 변수에 큰타입의 변수/값을 지정 시
오버플로우 발생 가능

-> 충분한 크기의 정확한 타입으로 선언

예제2)

■ 안전하지 않은 코드의 예 - C

```
1: struct DocumentInfo
2: {
3:     char * contents;
4:     int lineCount;
5:     char lineStartOffset[1024];
6:     ...
7: };
8:
9: DocumentInfo * AnalysisDocument
10: {
11:     DocumentInfo * info = (DocumentInfo *)malloc(sizeof(struct DocumentInfo));
12:
13:     char * nextCharPos = contents;
14:     int lineCount = 0;
15:     int lineOffset = 1;
16:     while(*nextCharPos)
17:     {
18:         if(*nextCharPos == '\n')
19:         {
20:             lineStartOffset[lineCount] = lineOffset;
21:             lineCount++;
22:             lineOffset = 0;
23:         }
24:         lineOffset++;
25:     }
26: }
```

int lineOffset != char lineStartOffset
오버플로우 발생

5. 스택 변수 주소 리턴(Return of Stack Variable Address)

1) 정의

함수가 스택의 주소를 반환하는 경우 그 주소가 더 이상 유효하지 않는 경우 발생

-> 예상치 않은 포인터 값 변경으로 인한 유효하지 않은 포인터 주소 참조, 프로그램 종료, 증상이 나타나지 않는 경우로 디버깅이 어려움

2) 원인

프로그램의 특정 함수가 지역 변수의 주소를 반환할 때 스택 주소가 반환. 이후의 동일 함수 호출은 같은 주소를 다시 사용할 가능성이 높아 지역 변수의 포인터 값을 덮어씀. 이 경우, 예상치 않게 포인터 값이 변경되는 문제 발생.

3) 안전한 코딩기법

- 스택 변수 주소의 리턴 스택상의 지역변수의 주소는 반환 X. 스택의 지역변수들은 함수의 소멸과 동시에 같이 제거되어 스택상의 지역변수의 주소를 리턴값을 주게 되면 할당되지 않는 공간을 참조하게 된다.

4) 예제

예제1)

■ 안전하지 않은 코드의 예 - C

```
1: char *rpl()
2: {
3:     char p[10];
4:     /* 로컬 버퍼의 주소를 반환함 */
5:     return p;
6: }
7: int main()
8: {
9:     char *p;
10:    p = rpl();
11:    *p = '1';
12:    return 0;
13: }
```

→ 스택의 지역변수는 함수가 끝난 후 소멸

■ 안전한 코드의 예 - C

```
1: #include <stdlib.h>
2: #include <string.h>
3: char *rpl()
4: {
5:     char p[10];
6:     int size = 10;
7:     if( size < 20 )
8:     {
9:         /* 메모리를 새롭게 할당하여, 로컬 버퍼의 내용을 복사해 넣은 후 새롭게 할당된 메모리 주소를 반환함 */
10:        char *buf = (char *)malloc(size);
11:        if (!buf)
12:            exit(1);
13:        memcpy(buf,p,10);
14:    }
15:    return buf;
16: }
17: int main()
18: {
19:     char *p;
20:    p = rpl();
21:    *p = '1';
22:    free(p);
23:    return 0;
24: }
```

힙 영역의 주소를 반환 또는
메모리 영역을 호출하는 함수로부터 반환값 입력

예제2)

Matrix : 동일한 데이터 타입을 갖는 2차원 배열⁷⁾

■ 안전하지 않은 코드의 예 - C

```
1: typedef struct matrix
2: {
3:     int num11, num12;
4:     int num21, num22;
5: };
6:
7: typedef struct matrix matrix;
8:
9: matrix * addMatrix(matrix * operand1, matrix * operand2)
10: {
11:     matrix result;
12:     result.num11 = operand1->num11 + operand2->num11;
13:     result.num12 = operand1->num12 + operand2->num12;
14:     result.num21 = operand1->num21 + operand2->num21;
15:     result.num22 = operand1->num22 + operand2->num22;
16:     return &result;
17: }
```

matrix(2차원 배열)타입으로 선언했기 때문에 &result 를 사용해야 하지만 생 함수가 종료되면서 소멸되어 프로그램이 오작동 발생

■ 안전한 코드의 예 - C

```
1: typedef struct matrix
2: {
3:     int num11, num12;
4:     int num21, num22;
5: };
6:
7: typedef struct matrix matrix;
8:
9: matrix * addMatrix(matrix * operand1, matrix * operand2)
10: {
11:     matrix * result = (matrix *)malloc(sizeof(matrix));
12:     result->num11 = operand1->num11 + operand2->num11;
13:     result->num12 = operand1->num12 + operand2->num12;
14:     result->num21 = operand1->num21 + operand2->num21;
15:     result->num22 = operand1->num22 + operand2->num22;
16:     return result;
17: }
```

함수 종료 후에도 보존이 가능하도록

힙 메모리에 데이터 저장 & 전체 구조체의 포인터 return

6. 매크로의 잘못된 사용(Code Correctness: Macro Misuse)

1) 정의

각 매크로에 따른 특정한 사용 규칙을 위반한 경우 발생

2) 안전한 코딩기법

- 매크로 사용 시 공유 자원에서 동작하는 특정한 함수 패밀리와 함께 사용

3) 예제

예제1)

■ 안전하지 않은 코드의 예 - C

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <pthread.h>
4: void routine(void *i)
5: {
6:     int j = (*(int*)(i))++;
7: }
8: void helper()
9: {
10:    int a = 0;
11:    pthread_cleanup_push (routine, ((void*)&a));
12: }
```

두 개의 매크로가 같은 레벨의 스코프에서 같이 쌍으로 사용되어야 하는 매크로가 있다. 이러한 매크로를 다른 스코프에서 사용하거나 하나만 사용할 경우 잘못된 코드가 생성된다

스코프 : 어떤 범위에 있는 변수들에 접근 가능한 범위, 즉 유효범위⁸⁾

■ 안전한 코드의 예 - C

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <pthread.h>
4: void routine(void *i)
5: {
6:     int j = (*(int*)(i))++;
7: }
8: void helper()
9: {
10:    int a = 0;
11:    pthread_cleanup_push (routine, ((void*)&a));
12:    pthread_cleanup_pop (1);
13: }
```

pthread_cleanup_push()와 짝을 이루는 pthread_cleanup_pop()을 함께 사용한다

8) 타오르는지천사, “7.자바스크립트 스코프(scope) 및 클로저(closer).” 2020-06-17.

<https://m.blog.naver.com/z1004man/222003749691>

7. 코드정확성: 스택 주소 해제(Code Correctness : Memory Free on Stack Variable)

1) 정의

스택 버퍼를 해제하면 예상치 않은 프로그램 동작이 발생

2) 안전한 코딩기법

- 지역 변수로 배열을 선언하면 스택에 버퍼가 할당되고— 함수가 반환할 때 버퍼를 자동으로 제거하므로 명시적으로 해제하지 않는다.

3) 예제

예제1)

■ 안전하지 않은 코드의 예 - C

```
1: #include <stdlib.h>
2: int main()
3: {
4:     char p[10];
5:     /* 지역변수내에 배열에 대한 메모리 해제 */
6:     free(p);
7:     return 0;
8: }
```

→ 함수가 반환할 때 버퍼를 자동으로 제거하므로 메모리를 명시적으로 해제 x

로컬 영역에 선언된 변수는 Call Stack이 종료되어 자동으로 메모리가 반환
-> 메모리를 해제할 필요가 없다

예제2)

■ 안전하지 않은 코드의 예 - C

```
1: #include <stdio.h>
2: #include <string.h>
3:
4: int main ()
5: {
6:     char str[] = "- This, a sample string.";
7:     char * pch;
8:     printf ("Splitting string \"%s\" into tokens:\n",str);
9:     pch = strtok (str, " ,.-");
10:    free(pch);
11:    while (pch != NULL)
12:    {
13:        printf ("%s\n",pch);
14:        pch = strtok (NULL, " ,.-");
15:        free(pch);
16:    }
17:    return 0;
18: }
```

→ pch를 free할 경우 local 변수에 대한 메모리 해제 시도

strtok() : 별도로 memory를 할당하지 않고 기존 메모리를 사용

로컬영역에 선언된 변수는 자동으로 메모리가 반환되므로 메모리를 해제하지 않는다

8. 코드 정확성: 스레드 조기 종료(Code Correcctness: Premature thread Termination)

1) 정의

부모 스레드가 자식 스레드 보다 먼저 종료되는 경우, 자식 스레드에 분배된 자원 회수를 못하는 경우

2) 원인

자식 스레드를 생성할 경우 내부적으로 스레드 제어를 위한 자료 구조를 생성하게 되는데 이러한 자료 구조는 부모 스레드가 pthread_join()과 같은 종류의 함수를 통해 자식 스레드가 종료되기를 기다렸다가 스레드 제어용 자료구조를 반환하게 된다. 만약, 자식 스레드의 종료를 기다리지 않고 부모 스레드가 종료하는 경우, 자식 스레드에 할당된 자료구조가 회수되지 않음

-> 메모리 누수 현상 발생

3) 안전한 코딩기법

- 부모 스레드가 join을 통해 자식을 기다린다
- 부모 스레드가 먼저 종료되어야 하는 경우, pthread_detach()를 통해서 자식 스레드를 분리(detach)
- 생성 시에 스레드의 attribute를 “분리” 상태로 생성하여 종료 시 스스로 자료구조를 반환하도록 한다.

4) 예제

예제1)

■ 안전한 코드의 예 - C

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <pthread.h>
4: int th_worker(void *data)
5: {
6:     int a = *(int *)data;
7:     return a + 100;
8: }
9: int run_thread1(void)
10: {
11:     pthread_t th;
12:     int status, a = 1;
13:     if (pthread_create(&th, NULL, (void*)(void*)th_worker, (void *)&a) < 0)
14:     {
15:         perror("thread create error: ");
16:         exit(0);
17:     }
18:     printf("Return after waiting for child thread\n");
19:     pthread_join(th, (void**)&status);
20:     return 0;
21: }
```

pthread_create() 수행 후,
pthread_join()으로 자식 스레드를 기다려서 자원을 반납하고 반환

■ 안전한 코드의 예 - C

```

1: void * _MakeEstimate(void *parm)
2: {
3:     char * stockName = param;
4:     // 부모 thread와 무관한 동작을 수행
5:     Estimate * estimate = MakeEstimate(stockName);
6:     RecordEstimate(estimate);
7:     return NULL;
8: }
9:
10: int main(int argc, char **argv)
11: {
12:     pthread_attr_t      attr;
13:     pthread_t           thread;
14:     int                 rc=0;
15:     int STOCK_COUNT = 3452;
16:     char * stockNames[STOCK_COUNT] = { ... };
17:     int stockIdx;
18:
19:     pthread_attr_init(&attr);
20:     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED); // detach 상태로
thread를 생성한다.
21:     for(stockIdx = 0; stockIdx < STOCK_COUNT; stockIdx++){
22:         pthread_create(&thread, &attr, _MakeEstimate, stockNames[stockIdx]);
23:     }
24:     pthread_attr_destroy(&attr);
25:     ...
26:     return 0;
27: }

```

스레드를 detach 상태로 생성하고

pthread_attr_destroy(&attr)을 호출하여 자원 반환

9. 무한 자원 할당(Allocation of Resources Without Limits or Throttling)

1) 정의

프로그램이 자원을 사용 후 해제하지 않거나, 한 사용자당 서비스할 수 있는 자원의 양을 제한하지 않고, 서비스 요청마다 요구하는 자원을 할당

2) 안전한 코딩기법

- 프로그램에서 자원을 오픈하여 사용한 뒤, 반드시 자원을 해제
- 사용자가 사용할 수 있는 자원의 사이즈 제한
 - ※ 특히 제한된 자원을 효율적으로 사용하기 위해서 Pool(Thread Pool, Connection Pool 등)을 사용한다.

3) 예제

예제1)

■ 안전하지 않은 코드의 예 - C

```
1: int processMessage(char **message)
2: {
3:     char *body;
4:     int length = getMessageLength(message[0]);
5:     if (length > 0)
6:     {
7:         body = &message[1][0];
8:         processMessageBody(body);
9:         .....
10:    }
11:    else {.....}
12: }
```

인자값이 메시지 길이와 메시지 **body**의 이차원 배열로 된 경우,
메시지 길이에 대한 제한이 없기 때문에 **body**에 아주 큰 데이터
가 오게 되면 시스템 메모리를 고갈시킬 수 있다

■ 안전한 코드의 예 - C

```
1: int processMessage(char **message)
2: {
3:     char *body;
4:     unsigned int length = getMessageLength(message[0]);
5:     // 자원 고갈을 방지위해 사용자의 자원을 제한한다.
6:     if (length > 0 && length < MAX_LENGTH)
7:     {
8:         body = &message[1][0];
9:         processMessageBody(body);
10:        .....
11:    }
12:    else {.....}
13: }
```

음수 발생을 방지 하기 위한 **unsigned int** 형 선언

메시지 처리를 위한 메모리 사용량 제한

예제2)

스레드를 생성할 때마다 새로운 소켓자원을 할당하므로,
스레드 수가 무한정 증가할 경우 시스템 자원을 고갈시킬 수 있다.
사용이 끝난 메모리를 **Pool**에 넣고, 새로 스레드가 생성되는 경우 **Pool**에서 먼저
사용하면 메모리 낭비를 줄일 수 있다.

■ 안전한 코드의 예 - C

```
1: ThreadPool threadPool;
2:
3: int main(int argc, char * argv[])
4: {
5:     int listenSocket, connectionSocket;
6:     struct sockaddr_in serv_addr, cli_addr;
```



```

7:  socklen_t cliLen;
8:
9:  listenSocket = socket(AF_INET SOCK_STREAM, 0);
10:  ...
11:  while(TRUE)
12:  {
13:      connectionSocket = accept(listenSocket, (struct sockaddr *) &cli_addr, &cliLen);
14:      // Thread pool에서 thread를 할당한다.
15:      if(threadPool.alloc(communication, &connectionSocket) == TRUE)
16:      {
17:          connectionSocket = -1;
18:      }
19:      else
20:      {
21:          close(connectionSocket);
22:      }
23:      ...
24:  }
25:  close(listenSocket);
26: }
27: void * communicate(pthread_t thread, void * argument)
28: {
29:     pthread_t thread;
30:     int socket = *((int *)argument)
31:     ...
32:     close(socket);
33:     // Thread pool에서 thread를 해지한다.
34:     threadPool.free(thread);
35: }

```

제 6절
캡슐화

<6절 캡슐화>

SW가 중요한 데이터나 기능을 불충분하게 캡슐화 하는 경우, 인가된 데이터와 인가되지 않은 데이터를 구분하지 못하게 되어 허용되지 않은 사용자들 간의 데이터 누출 발생

1. 제거되지 않고 남은 디버그 코드(Leftover Debug Code)

1) 정의

디버그 코드가 설정 등의 민감한 정보를 내포하거나 시스템 제어와 관련된 부분을 내포하고 남겨진 채로 배포될 경우, 공격자가 식별 과정을 우회하거나 의도하지 않은 정보와 제어 정보가 노출될 수 있다.

2) 안전한 코딩기법

- 디버깅 목적으로 삽입된 코드는 개발이 완료되면 제거
- 개발 단계에서 디버깅 목적으로 사용한 프로그램에서 main과 콘솔에 출력되는 문장 삭제

3) 예제

main()내에 출력하는 코드를 작성하는 대신,
#ifdef분기문으로 디버깅을 할 때에만 활성화시켜 콘솔에 표시한다

■ 안전한 코드의 예 - C

```
1: #define DEBUG_ENABLE
2: #ifdef DEBUG_ENABLE
3:     #define DEBUG(message)    { puts( message ); }
4: #else
5:     #define DEBUG(message)
6: #endif
7:
8: int main()
9: {
10:     ...
11:     DEBUG("Debug message");
12:     ...
13: }
```

2. 시스템 데이터 정보노출(Exposure of System to an Unauthorized Control Sphere)

1) 정의

시스템·관리자DB 정보 등 시스템의 내부 데이터가 공개되면, 이를 통해 공격자에게 아이디어를 제공하는 등 공격의 발みが 된다.



2) 안전한 코딩기법

- 예외상황 발생 시, 시스템의 내부 정보가 화면에 출력되지 않도록한다.

3) 예제

■ 안전하지 않은 코드의 예 - C

```
1: #include "std_testcase.h"
2:
3: #include <stdio.h>
4:
5: #ifndef OMITBAD
6:
7: void
    KRD_603_Exposure_of_System_Data_to_an_Unauthorized_Control_Sphere__w32_sprintf_0101_
    bad()
8: {
9:     {
10:
11:     {
12:         char* path = getenv("PATH");
13:         /* FLAW */
14:         printf(stderr, "cannot find exe on path %s\n", path);
15:     }
16:
17:     }
18: }
19:
20: #endif /* OMITBAD */
~
```

■ 안전한 코드의 예 - C

```
43: #include "std_testcase.h"
44: #include <stdio.h>
45: #ifndef OMITGOOD
46:
47: static void good1()
48: {
49:     {
50:
51:     {
```

```
52:     char* path = getenv("PATH");
53:     /* FIX */
54:     sprintf(stderr, "cannot find exe on path \n");
55: }
56:
57: }
58: }
59:
60: void
61: KRD_603_Exposure_of_System_Data_to_an_Unauthorized_Control_Sphere__w32_sprintf_0101_
62: good()
63: {
64:     good1();
65: }
66: #endif /* OMITGOOD */
```

오류와 관련된 상세정보 출력 X

제 7절
API 오용

<7절 API 오용>

- API(Application Programming Interface) : 운영체제와 응용프로그램간의 통신에 사용되는 언어나 메시지 형식 또는 규약, 프로그램 개발시 개발 편리성 및 효율성을 제공

1. DNS lookup에 의존한 보안결정(Reliance on DNS Lookups in a Security Decision)

1) 정의

- 공격자가 DNS 엔트리를 속일 수 있으므로 도메인명에 의존하여 보안 결정(인증 및 접근 통제 등)을 하지 않아야 한다. 만약 로컬 DNS 서버의 캐시가 공격자에 의해 조작된 상황일 경우,
- > 사용자와 특정 서버간의 네트워크 트래픽이 공격자를 경유하도록 설정 가능
 - > 공격자가 마치 동일 도메인에 속한 서버인 것처럼 위장 가능

2) 안전한 코딩기법

- 보안결정에서 도메인명을 이용한 DNS lookup에 의존하지 않는다.
- DNS 이름 검색 함수를 사용한 후 조건문에서 인증여부를 수행하는 것보다 IP주소를 이용한다.

3) 예제

■ 안전하지 않은 코드의 예 - C

```
1: const char * GetParameter(const char * queryString, const char * key);
2:
3: int main(void)
4: {
5:     const char * COMMAND_PARAM = "command";
6:     const char * SEND_CONTRACT = "send_contract";
7:
8:     set<char *> trustedSites;
9:     trustedSites.insert("www.trust.com");
10:    trustedSites.insert("www.faith.com");
11:
```

DNS 명이 신뢰할 수 있는 사이트에 등록되어 있을지라도 능
공격자가 DNS 캐쉬 내 IP 정보를 공격자의 IP정보로 조작함으로써
해당 요청에 대한 신뢰성 결정 우회 가능

■ 안전한 코드의 예 - C

```
1: const char * GetParameter(const char * queryString, const char * key);
2:
3: int main(void)
4: {
5:     const char * COMMAND_PARAM = "command";
6:     const char * SEND_CONTRACT = "send_contract";
7:     set<char *> trustedSiteIPs;
8:     trustedSiteIPs.insert("232.234.89.52");
9:     trustedSiteIPs.insert("87.123.56.92");
10:    char * queryStr;
11:    queryStr = getenv("QUERY_STRING");
12:    if (queryStr == NULL)
```

DNS 이름 대신 IP 주소를 사용하는 것이 안전하다.

IP주소 역시 DNS서버를 가장하여 위조 가능하지만,

호스트 명을 사용하는 경우보다 안전성을 확보할 수 있다

2. 위험하다고 알려진 함수 사용(Use of Inherently Dangerous Function)

1) 정의

이미 위험하다고 알려진 라이브러리 함수를 사용하는 것은 바람직하지 않다. 특정 라이브러리 함수들은 보안취약점을 전형 고려하지 않고 개발되어 사용 시 취약점이 될 수 있다.

2) 안전한 코딩기법

- 잘 알려진 취약점으로 인해 이미 위험하다고 알려진 함수의 사용 금지

3) 예제

예제1)

■ 안전하지 않은 코드의 예 - C

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #define BUFSIZE 100
4: void requestString()
5: {
6:     char buf[BUFSIZE];
7:     gets(buf);      gets() 대신 fgets() 사용
8: }
```

gets() 함수

입력 크기 제한사항을 점검하지 않기 때문에 입력 버퍼가 초과될 수 있다.

=> 사용 자체로 보안취약점

예제2)

vfork() 함수는 자식 프로세스가 부모 프로세스의 공간을 잠시 빌려 사용하기 때문에 부모 프로세스의 의도를 왜곡시키거나 의도적인 오작동 야기

■ 안전하지 않은 코드의 예 - C

```
1: char *filename = /* something */;
2:
3: pid_t pid = vfork();      부모 프로세스와 별도의 공간을 사용하는
                           fork()함수 사용
4: if (pid == 0) /* child */
5: {
6:     if (execve(filename, NULL, NULL) == -1)
7:     {
8:         /* Handle error */
9:     }
10:     _exit(1); /* in case execve() fails */
11: }
```


3. 작업 디렉터리 변경 없는 chroot jail 생성(Creation of chroot Jail Without Change Working Directory)

1) 정의

프로그램에서 chroot함수를 사용하여 접근 가능한 디렉터리를 제한할 때 작업 디렉터리를 변경하지 않는 경우 공격자가 제한된 디렉터리 외부에 접근 가능

- chroot : 프로그램이 특정 디렉터를 파일 시스템의 루트 디렉터리로 인지하도록 하여 제한된 디렉터리 내에서 파일 접근이 가능하도록 하는 함수. 이때 프로그램의 작업 디렉터를 변경해야 하며 만약 변경하지 않을 경우 상대 경로를 이용하여 다른 디렉터리에 접근 가능

2) 안전한 코딩기법

- chroot() 이후에 chdir("/")를 사용하여 현재 디렉터를 새로운 루트 디렉터리의 하위 경로로 변경되도록 한다.

3) 예제

■ 안전하지 않은 코드의 예 - C

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <unistd.h>
4: void changeRoot(FILE *network)
5: {
6:     FILE *localfile;
7:     char filename[80], buf[80];
8:     int len;
9:
10:    chroot("/var/ftpboot");
11:
12:    fgets(filename, sizeof(filename), network);
13:    localfile = fopen(filename, "r");
14:    while ((len = fread(buf, 1, sizeof(buf), localfile)) != EOF)
15:    {
16:        fwrite(buf, 1, sizeof(buf), network);
17:    }
18:    fclose(localfile);
19: }
```

/var/ftpboot 를 루트 디렉터리로 설정하여 클라이언트가 해당 하위 경로만 접근 가능하도록 하였다.

그러나 작업 디렉터를 변경하지 않았기 때문에 클라이언트가 ../../etc/passwd 경로로 이동하여 패스워드 파일에 접근 가능하다.

■ 안전한 코드의 예 - C

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <unistd.h>
4: void changeRoot(FILE *network)
5: {
6:     FILE *localfile;
7:     char filename[80], buf[80];
8:     int len;
9:
10:    chroot("/var/ftpboot");
11:
12:    /* 현재 디렉터리를 새로운 루트 디렉터리 밑으로 변경 */
13:    chdir("/");
14:
15:    fgets(filename, sizeof(filename), network);
16:    localfile = fopen(filename, "r");
17:    while ((len = fread(buf, 1, sizeof(buf), localfile)) != EOF)
18:    {
19:        fwrite(buf, 1, sizeof(buf), network);
20:    }
21:    fclose(localfile);
22: }
```

4. 오용: 문자열 관리(Often Misused: String Management)

1) 정의

버퍼 오버플로우가 발생할 수 있는 문자열 연산 함수를 호출하면 위험하다.

2) 안전한 코딩기법

- 마이크로소프트 윈도우에서 제공하는 멀티 바이트 문자열 함수들(`_mbsXXX()` 류)은 모두 버퍼 오버플로우를 일으킬 수 있으므로 사용하지 않는다.

3) 예제

멀티 바이트 형식에 맞지 않는 문자열을 전달하게 되면 버퍼 오버플로우 발생

■ 안전하지 않은 코드의 예 - C

```
1: #include <mbstring.h>
2: void changeString(char *str1, char *str2)
3: {
4:     _mbscopy(str1, str2); 안전한 _mbscopy_s()를 사용한다.
5: }
```

5. 다중 스레드 프로그램에서 getlogin() 사용(Use of getlogin() in Multithreaded Application)

1) 정의

사용자 계정은 다중 스레드 환경에서 잠금장치를 하지 않고 획득 가능하다, 이러한 경우 다른 스레드가 사용 중인 사용자 계정을 읽어올 수 있다.

- getlogin() : 호출된 스레드와 연관된 사용자 이름을 리턴하는 함수

이를 다중 스레드 환경에서 사용하면 리턴 받은 사용자 이름값이 다른 스레드에 의해 변경될 수 있어 부정확한 값을 얻을 수 있다.

2) 안전한 코딩기법

- getlogin() 함수 대신 안전한 getlogin_r() 함수를 사용

3) 예제

반환 값의 변경이 발생할 수 있어 다

getlogin()의 반환 값을 근거로 권한 부여는 보안상 위험하다

■ 안전하지 않은 코드의 예 - C

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <unistd.h>
4: #include <sys/types.h>
5: #include <pwd.h>
6: int isTrustedGroup(int);
7: int loginproc()
8: {
9:     struct passwd *pwd = getpwnam(getlogin());
10:    if (isTrustedGroup(pwd->pw_gid))
11:        return 1; // allow
12:    else
13:        return 0; // deny
14: }
```

다중 스레드 환경에서 안전한 **getlogin_r()** 로
계정 정보를 획득하여 권한을 부여