기술문서	‘09. 11. 06. 작성

보안 프로그래밍

작성자 : 경일대학교 KICOM 박율기 pak8198@dreamwiz.com

1. 들어가며

가. 보안 프로그래밍

- 보안 프로그래밍이란 일반적인 논리적 흐름 오류로 인한 버그가 아닌 보안상 문제점을 일으키는 무결성, 가용성 및 기밀성에 영향을 주어 예기치 않은 프로그램적 결함이 프로그램 제작 시 보안에 취약한 코드들을 배제하여 프로그램을 개발하는 것.

나. 필요성

- 대부분의 프로그램의 보안 취약점은 설계상의 문제로 인하여 발생한다. 많은 프로그램이 설계상 보안성 취약으로 정보 유출로 이어져 왔으며, 보안 프로그래밍이 되어 있지 않다면 리버싱이 쉬워지며 많은 Cracker들에게 '난 취약점이 많습니다' 라고 말하는 것과 같다. 이처럼 프로그래밍 자체에 자료에 대한 무결성이 검증된다면 리버싱 등이 어려우며 여러 가지 후킹등을 방해할 수 있다.

다. 주의!

- 보안 프로그래밍 이란 단순히 버퍼 오버플로우나 레이스 컨디션에 신경써서 될 것이 아니다. 버퍼 오버플로우 등은 기본적인 프로그래밍 지식이나 습관으로 방지 할 수 있다. 우린 이런것들 보다 좀더 Cracking을 하는 입장에서 생각할 필요가 있다. Cracking에는 기본적으로 리버싱과 후킹 등이 사용되며 이러한 기술들을 이용하여 자료의 무결성을 무너뜨리며 수정을 하는 것이 대부분이다.

라. 궁극적 목적

-이 문서의 목적은 기술적 부분 보다 우리들의 인식을 고쳐 줌에 중점을 두었다. 예제들은 누구나 보면 쉽게 아는 프로그램을 예제로 쓸 것이다. 이점은 프로그래밍의 기초를 아는 분들도 이해할 수 있도록 하여 프로그램 설계상 취약점에 대한 인지도를 알리기 위함이며 기초 지식을 습득하는데 도움이 되었으면 한다.

##소스 코드는 오류로 인한 리턴 등은 -1로 주었다. 그 이유는 시스템 프로그래밍을
##하다보면 알겟지만 OS에서 넘겨주는 값들중 음수로 넘어오는 값은 잘 없기 때문이다.

1. 유효값 제한

- 가장 기초적인 것이다. 누구든 생각 할 것이다. "이건 당연한거잔아!" 라고 하지만 작은 프로젝트라면 당연히 눈에도 잘 띄고 유효값 제한을 할 것이다.
하지만 이것은 자신이 맡고 있는 프로젝트의 일부분일 뿐이다. 시대가 지남에 따라 프로젝트의 크기는 증가하였고 여러사람이 작성한 작은 프로젝트를 결합하여 하나의 완성된 프로젝트가 되는 체제가 되었다. 이러하듯 한 프로젝트가 여러사람의 작은 프로젝트에 의해 만들어 졌으므로 그 작은 프로젝트끼리의 자료교환이 발생하며 이런 자료의 교환에 있어서도 유효값에 제한을 하여 행여나 있을 수 있는 취약성을 막아야 한다.

```
void CharCheck(char *char_str)
{
        if(IsInclude(input, " ;&|'`₩"?$^~*<>[](){}₩n₩r"))
        {
                LOG("ERROR!!₩n");
        }
        else
        {
                int revalue = execl(path,char_str)
        }
}
```

이런 함수를 이용하여(리눅스) 매번 특수 문자 등에 제한을 걸어 두어야 한다.
물론 네트워크 데이터에 대해서도 제한을 해두는 것이 좋다.

2. 리턴값 검사

- 흔히 초보자들이 범하기 쉬운 것이며 대부분 함수에 대한 독립성을 고려하지 않아 생기게 된다. 함수는 독립성이 강하며 이것을 이용하여 리버싱시 인젝션 하는 경우도 많기 때문에 리턴값에 검사는 필수 요소중 하나라고 볼 수 있다.
이 부분은 간단히 구현이 가능하여 소스등은 생략.

3. 권한 조정

- IIS 웹서버가 시스템 계정이 부여되어 있다면 IIS 취약성을 이용하여 시스템 권한을 가질 수 있다. 물론 업로드된 파일 중 서버용 실행 파일인 ASP 등의 실행을 막아 취약점을 해결할 수 있으나 미리 권한적 문제를 설계에 고려 했다면 취약점에 의한 피해가 없었을 것이다. 아파치(apache)의 경우 nobody로 실행 되는데 setuid() 및 seteuid 등의 함수를 이용하여 권한을 강제조정 하여 이를 막을 수 있다.

```
int GradeCtrl()
{
        struct passwd *structp_user = getpwnam("nobody");
        if(!pep)
        {
                return -1;
        }
        if(setgid(structp_user->pw_gid) < 0)
        {
                return -1;
        }
        if(setuid(structp_user->pw_uid) < 0)
        {
                return -1;
        }
        return 0;
}
```

이런식으로 권한을 조정 하여 시스템 권한에 대한 문제를 방지를 할 수 있다.

4. 환경 변수

- 환경변수는 명시적 및 암시적인 환경변수가 있다.
환경변수에 의한 취약점이 될 수 있는 부분을 예를 들자면 프로그램은 시작할 때 기본적으로 정의된 폴더로 쓰기 보다는 특정 디렉토리로 정의하는 것이 좋으며 umask도 적당하게 설정해놓아야 한다.
exec함수를 통해 환경변수를 전달할 때에는 새로운 환경변수를 만들어서 전달하는 것이 안전하다.

```
if( (chroot("₩usr₩local₩ccn") < 0) || (chdir("₩") < 0) )
{
        perror("Fail New Root");
}
```

위 예는 chroot명령을 통해 기본 디렉토리를 변경하는 방법 이다.

5. 버퍼오버플로우

- 많이 들어보고 해봤을 것이다. 이것은 C의 특성상 경계 검사를 하지 않는 점과 스택 구조라는 점을 이용하여 공격자가 공격 코드를 실행하게 하거나 특정 프로그램의 오류를 유도하여 종료 시키는 등의 일을 할 수 있다. 이런 버퍼오버플로우를 일으키는 이유는 앞서 말했듯이 경계 검사를 하지 않아서 이다. strcpy(), strcat(), sprintf(), gets() 등이 경계검사를 하지 않는 함수들이다. 이런 함수들을 사용하지 않는 대신 strncpy(), strncat(), snprintf(), fget() 등을 사용 해야 한다. strncpy나 strncat를 사용 시 null문자를 추가하지 않으므로 크기를 -1만큼 주어야 한다.

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

int main()
{
        int int_i;
        char char_num[6]="01234";
        char char_pat1[3],char_pat2[3];

        memset(char_pat1, 0, sizeof(char_pat1));
        memset(char_pat2, 0, sizeof(char_pat2));

        strncpy(char_pat1, char_num, sizeof(char_pat1));

        for(int_i = 0; int_i < sizeof(char_pat1); int_i++)
        {
                printf("%c", char_pat1[int_i]);
        }
        printf("₩n");

        snprintf(char_pat2, sizeof(char_pat2), "%s", char_num);
        for(int_i = 0; int_i < sizeof(char_pat2); int_i++)
        {
                printf("%c", char_pat2[int_i]);
        }
        printf("₩n");
        return 0;
}
```

결과값은 1234 와 123이 나온다.
위 소스에서 %s를 사용하지 않은 것은 null문자가 없기 때문에 적당하지 않은 사용이다. 두 번째 결과값은 null문자가 삽입되어 맨 뒷자리가 지워진 것을 볼수 있다.

6. 레이스 컨디션

- 프로그램이 입출력을 위해 사용하는 리소스의 제어권과 관련된 취약점 이며 프로그램이 해당 리소스에 접근하기 전에 비 인가자가 리소스를 선점하여 문제가 발생하는 것이며 일반 권한으로 접근가능한 곳에서 발생한다. 이러한 레이스 컨디션을 막기 위해선 리소스를 열고 사용하기 전에 믿을만한지 검사하거나 리소스를 특수한 권한으로만 접근하도록 제한하여야 한다.

```
int RaceGuard(char *charp_filename)
{
        struct stat st1, st2;
        int int_revalue;

        if(lstat(charp_filename, &st1) != 0)
                return -1;
        if(!S_ISREG(st1.st_mode))r
                return -1;
        if(st1.st_uid != 0)
                return -1;
        int_revalue = open(charp_filename, O_RDWR,0);
        if(int_revalue < 0)
                return -1;
        if(fstat(int_revalue, &st2) != 0)
        {
                close(int_revalue);
                return -1;
        }

        if(st1.st_ino != st2.st_ino || st1.st_dev != st2.st_dev)
        {
                close(int_revalue);
                return -1;
        }
        return int_revalue;
}
```

예에서는 리소스에 대한 신뢰검사를 심볼릭 링크인지 여부와 루트의 소유인지 여부를 가지고 판단하는 것을 보여주고 있다.

7. 시스템 자원 관리

- 시스템 자원과 관련된 함수의 사용은 결과에 주의해야 한다.
일반적으로 함수를 실패할 경우는 드물며 그만큼 발생할 확률이 낮지만
DOS(Denial of Service)공격으로 시스템 자원이 모두 소모되어 실패하는
경우도 있다. 자원할당이 실패하는 경우에 프로그램이 해당
경고메시지와 수행을 방지하여 오작동을 막아야 한다. 자원 호출이
실패하는 경우는 메모리부족으로 인한 malloc실패,
File Descriptor 부족으로 인한 open실패,
리소스 부족으로 인한 Fork실패 등이 있다.
getrlimit(), setrlimit(), getrusage()등의 함수를 활용하여 파일의 크기,
자식 프로세스의 수, 파일열기 숫자 등에 제한을 주어 자원 사용의
최적를 하는 것이 바람직 하다.

```
#include <sys/resource.h>
#include <sys/time.h>
#include <unistd.h>

int main()
{
        struct rli mit rl;

        getrli mit(RLIMIT_CPU, &rl);

        rl.rlim_cur = 1;
        setrli mit(RLIMIT_CPU, &rl);

        while(1);

        return0;
}
```

위 예제는 시스템 자원할당에 제한을 두는 예를 보이고 있다.
이 예제는 커널 영역의 프로그래밍 자세한 사항은 난해한 부분이 많아
http://blog.naver.com/mybrainz?Redirect=Log&logNo=140004734579
이곳의 시스템 자원 관리 부분을 참조 하였다.
보안 프로그래밍은 범위가 넓으며 배우기 위해선 많은 경험이 필요하다.
그리고 많은 지식이 필요하다. 이부분을 심도 있게 공부하기 위해선 시스템
프로그래밍 및 메모리 구조등을 자세히 알아야하며 이렇게 시스템을 파고드는 것은
많은 시간과 노력을 요구한다. 하지만 꼭 필요한 부분이며 가장 지켜지기 힘든
부분이라고 생각한다.

참고 자료

-완벽한 보안을 위한 C와 C++코딩

-http://blog.naver.com/mybrainz?Redirect=Log&logNo=140004734579