디지털산업정보학회 논문지 제6권 제1호-2010년 3월

버퍼 오버플로우 공격 방지를 위한 취약 함수 변환기 구현

김 익 수* · 조 용 윤**

Implementation of a function translator converting vulnerable functions for preventing buffer overflow attacks

Kim, Ik Su · Cho, Yong Yun

〈Abstract〉

C language is frequently used to develop application and system programs. However, programs using C language are vulnerable to buffer overflow attacks. To prevent buffer overflow, programmers have to check boundaries of buffer areas when they develop programs. But vulnerable programs frequently result from improper programming habits and mistakes of programmers. Existing researches for preventing buffer overflow attacks only inform programmers of warnings about vulnerabilities and not remove vulnerabilities in advance so that the programs still include vulnerabilities. In this paper, we propose a function translator which prevents creating programs including buffer overflow vulnerabilities. To prevent creating binary from source including vulnerabilities, the proposed translator searches vulnerable functions which cause buffer overflows, and converts them into secure functions. Accordingly, developing vulnerable programs by programmers which lack in knowledge on security can be prevented.

Key Words : Buffer Overflow, Vulnerable Function, Secure Programming, Security

Ⅰ. 서론

오늘날 공격자에 의한 보안 취약점 공격이 빈번하게 발생하고 있다. 보안 취약점 공격을 통해 공격자들은 인증 과정을 우회할 수 있으며, 심각한 경우에는 시스템 관리자 권한을 획득할 수 있다. 대표적인 보안 취약점 공격에는 관리자에 의해 잘못 설정된 환경변수를 악용하는 공격, 버퍼의 크기보다 큰 입력 값을 삽입하는 버퍼 오버플로우 공격, 프로세스 간의 경쟁을 통한 레이스 컨디션, 예기치 않은 입력 값을 통한 인증 우회 등의 다양한 공격들이 존재한다. 특히, C 언어로 작성된 프로그램의 버퍼 오버플로우 공격은 현재 가장 빈번하게 발생하는 보안 취약점 공격이다. 버퍼 오버플로우 공격에 의한 피해를 사전에 예방하기 위해서는 C 프로그램 개발자들이 버퍼의 경계를 검사하는 코드를 작성해야 하지만 잘못된 프로그래밍 습관과 실수로 여전히 버퍼 오버플로우 공격

* 숭실대학교 컴퓨터학부 조교수
** 국립순천대학교 정보통신공학부 조교수(교신저자)

에 취약한 프로그램이 생산되고 있다.

버퍼 오버플로우 공격에 대응하기 위해 코드 내에 보안상 취약한 함수가 존재하는지의 여부를 알려주는 도구들이 개발되었으며[1-5], 프로그램 실행 시에 임의의 값으로 함수의 리턴 주소가 변경되는 것을 탐지하고 프로세스를 종료하는 코드를 생성하는 컴파일러도 개발되었다[6]. 하지만 기존 연구들은 단순히 취약점에 대한 경고만을 제공하거나 사전에 취약점을 제거하는 방법이 아니기 때문에 해당 프로그램은 여전히 취약점을 가질 수밖에 없다.

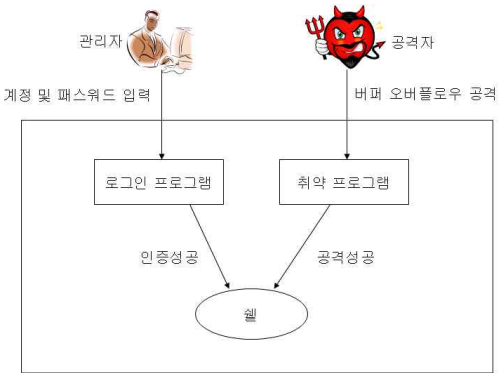
이에 본 논문에서는 버퍼 오버플로우 취약점이 내재된 프로그램 생성을 사전에 방지하기 위해 취약 함수가 포함된 코드를 안전한 코드로 변환하는 프로그램을 제안한다. 제안 프로그램은 취약한 코드로부터 이진파일이 생성되는 것을 막기 위해 코드 내에 버퍼 오버플로우를 유발할 수 있는 취약한 함수를 탐지하고, 버퍼 오버플로우의 발생을 최대한 사전에 방지하기 위해 안전한 함수와 코드로 변환한다. 이는 보안 지식이 부족한 개발자나 실수에 의해 취약 프로그램이 개발되는 것을 예방할 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구를 소개하며, 3장에서는 제안하는 프로그램에 대해 기술한다. 4장에서는 구현 프로그램을 통한 코드의 변환 결과를 살펴보고, 마지막으로 5장에서는 결론에 대해 기술한다.

Ⅱ. 관련연구

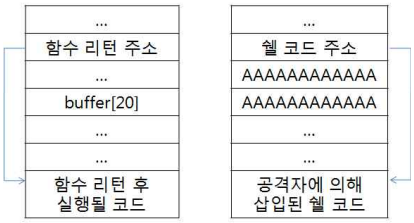
2.1 버퍼 오버플로우 공격

버퍼 오버플로우 공격은 1996년 Aleph가 'Smashing the stack for fun and profit' 문서를 통해 상세히 기술한 공격 방법으로, <그림 1>에서는 버퍼 오버플로우 공격을 통해 공격자가 불법으로 쉘을 획득하는 과정과 리눅스에서 일반 사용자가 정상적인 로그인을 통해 쉘을 획득하는 과정을 보여주고 있다. 정상적인 로그인 과정에서는 사용자가 자신의 계정과 패스워드를 입력하며, 운영체제는 입력된 정보와 패스워드 파일을 이용하여 사용자를 인증한다. 인증된 사용자는 운영체제로부터 시스템 서비스를 이용할 수 있는 쉘을 얻게 된다. 하지만 공격자가 버퍼 오버플로우 취약점을 지니고 있는 프로그램에 공격을 시도하여 성공할 경우에도 시스템 서비스를 이용할 수 있는 쉘을 얻을 수 있기 때문에 버퍼 오버플로우 취약점은 시스템 보안에 커다란 문제가 된다.



<그림 1> 정상 로그인과 버퍼 오버플로우 공격에 의한 쉘 획득

버퍼 오버플로우 취약점이 내재된 프로그램에 공격을 시도하여 성공하기 위해서는 버퍼의 길이를 초과하는 값을 입력하여 함수의 리턴 주소 값을 변경해야 한다.



<그림 2> 정상적인 메모리(좌)와 버퍼 오버플로우 공격에 의한 메모리(우)

<그림 2>는 함수 호출 시의 정상적인 메모리 상태와 버퍼 오버플로우 공격에 의한 메모리 상태를 나타낸다. 정상적인 프로그램에서는 함수가 종료되면 프로그램 흐

름이 함수 리턴 주소에 의해 함수 호출문 이후로 이동한다. 하지만 버퍼 오버플로우 공격에서는 공격자가 미리 메모리 특정 영역에 쉘 코드를 삽입하고, 입력 버퍼 공간에 불필요한 문자들을 삽입함과 동시에 함수 리턴 주소가 저장되어야 할 공간에 쉘 코드가 저장된 주소를 덮어쓴다. 결국 실행 중인 함수가 종료되면, 실제로 실행되어야 할 코드가 아닌 공격자에 의해 삽입된 쉘 코드가 실행되기 때문에 공격자는 쉘을 획득할 수 있다. 쉘 코드는 쉘을 실행시키는 기계어 수준의 코드로서 취약 프로그램이 루트권한의 SetUID가 설정된 프로그램일 경우에 공격자는 루트 권한의 쉘을 획득할 수 있다.

<그림 3> 버퍼 오버플로우 공격

<그림 3>은 원거리 시스템 상에서 버퍼 오버플로우 취약점이 내재된 서비스 프로그램에 버퍼 오버플로우 공격을 수행한 결과로서 공격자가 루트 권한의 쉘을 획득한 후, 새로운 계정 생성 및 루트 계정의 패스워드를 변경한 것을 알 수 있다.

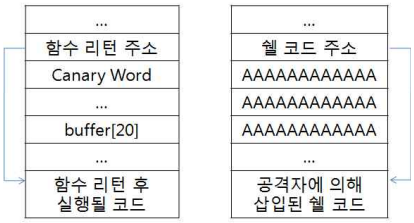
오늘날 빈번하게 발생하는 인터넷 웜에 의한 공격들은 대부분 버퍼 오버플로우 취약점을 악용하기 때문에 버퍼 오버플로우 공격에 대한 대응이 매우 절실하다.

2.2 버퍼 오버플로우 취약점 대응

버퍼 오버플로우를 유발하는 대표적인 보안 취약 함수들에는 strcpy( ), strcat( ), gets( ), sprintf( )가 있으며, 공격자는 버퍼의 경계를 초과하는 값을 입력함으로써 공격에 성공할 수 있다. 이러한 버퍼 오버플로우 공격을 방지하기 위한 다양한 코딩 기법이 소개되었으며[7], 취약 함수의 존재 여부를 소스코드의 정적 분석을 통해 미리 탐지하여 알려주는 도구들이 이미 개발되었다[1-5]. 그리고 소스의 정적 분석과 결합하여 인터넷 패킷 분석을 통한 버퍼 오버플로우 공격의 가능성을 알려주는 연구도 진행되어 왔다[8].

앞선 연구들과는 달리 컴파일러 확장과 커널 수준의 제어를 통해 취약점이 내재된 프로그램이 실행 시에 버퍼 오버플로우 공격에 악용되는 것을 차단하기 위해 연구도 진행되어 왔다. 하지만 이러한 방법들은 버퍼 오버플로우 공격을 근본적으로 차단할 수 없으며, 최근 들어서는 이들을 우회하여 버퍼 오버플로우 공격을 수행하는 진보된 방법들이 소개되고 있다[9].

버퍼 오버플로우 공격 방지 컴파일러인 Stackguard[6]는 <그림 4>와 같이 리턴 주소 값과 스택 사이에 canary라고 불리는 임의의 값을 끼워 두고 버퍼 오버플로우에 의해 canary가 새로운 값으로 변경되는지를 검사함으로써 공격을 탐지한다. 이와 유사한 방법으로 Stack shield는 Global Ret Stack이라 불리는 자료구조에 함수의 리턴 주소를 저장하고 함수가 리턴할 때 주소 값을 읽어와 리턴 주소로 사용한다.



<그림 4> 정상적인 상태에서의 Canary(좌)와 버퍼 오버플로우 공격에 의해 변조된 Canary(우)

그 외에 gcc 2.96이후 버전의 컴파일러는 버퍼를 할당할 때 각 버퍼 다음에 가변 공간을 채움으로써 공격자가 리턴 주소가 저장된 위치를 찾아 변경하는 것을 어렵게 한다.

앞서 소개한 정적분석 방법은 단순히 취약점에 대한

경고만을 제공할 뿐 취약점에 대한 방지 기능이 없으며, Stackguard와 Stack shield는 함수들의 리턴 주소 검사를 위한 처리 및 리턴 주소를 저장하기 위한 자료구조가 요구된다. 그리고 컴파일러에 의해서 버퍼를 할당할 때 가변 공간을 생성하는 방법은 공격자가 시행착오를 통해 리턴 주소가 저장된 위치를 찾을 수 있는 문제가 있다.

또한, 버퍼 오버플로우가 발생한 이후 루트권한 명령의 실행을 차단하기 위해 커널 수준에서 서버 관리자의 추가적인 패스워드를 요구하는 커널 인증 시스템[10]도 제안되었는데, 이 모듈은 로컬에서 발생하는 버퍼 오버플로우 공격에 적절히 대응하지만 원거리 시스템 상에서 발생하는 버퍼 오버플로우 공격에 대응하기 위한 방법을 제시하지 않았다.

Ⅲ. 취약 함수 변환기

3.1 취약 함수 분석과 대체 코드

C 언어에는 버퍼의 크기를 초과하는 입력 값에 의해 버퍼 오버플로우를 유발하는 여러 취약 함수들이 존재한다. <표 1>에 명시된 함수들은 프로그램 개발자들이 자주 사용하지만 실수로 인해 쉽게 버퍼 오버플로우를 유발하는 함수들과 이를 보완하기 위해 사용될 수 있는 함수들을 나타낸다.

<표 1> 취약 함수와 입력 길이 제한 함수

취약 함수	입력 길이 제한 함수
strcpy( )	strncpy( )
strcat( )	strncat( )
sprintf( )	snprintf( )
gets( )	fget( )
scanf( ), fscanf( ), sscanf( )...	없음

strcpy(char *dst, const char *src) 함수는 dst 버퍼에 src 데이터를 복사하는 함수로서 복사 과정에서 버퍼의 크기를 검사하지 않아 리턴 주소의 변경이 가능하다. 이러한 문제를 예방하기 위해 지정된 크기만큼만 복사를 수행하는 strncpy(char *dst, const char *src, size_t len) 함수를 사용할 수 있다. 이 함수의 세 번째 인자는 복사할 데이터의 크기를 의미하며 버퍼의 크기를 초과하지 않도록 하기 위해서는 다음과 같이 사용할 수 있다.

```
strncpy(dst, str, sizeof(dst)-1);
dst[sizeof(dst)-1]=0;
```

문자열의 종료를 명시하기 위해 버퍼의 마지막 요소는 널 문자로 구성되어야 한다.

strcat(char *dst, const char *src) 함수는 dst 버퍼에 저장된 데이터에 src 데이터를 덧붙이는 함수로서 strcpy( ) 함수와 마찬가지로 리턴 주소의 변경이 가능하다. 버퍼 오버플로우를 예방하기 위한 코드 작성 방법은 다음과 같다.

```
strncat(dst, str, sizeof(dst)-strlen(dst)-1);
```

dst 버퍼를 초과하지 않기 위해서 strncat(char *dst, const char *src, size_t len) 함수의 세 번째 인자, 즉 복사하고자 하는 데이터의 길이는 위 코드에 명시한 바와 같이 여분의 버퍼 크기를 초과해서는 안 된다. strncat( ) 함수의 경우 마지막에 널 문자를 자동으로 삽입하기 때문에 별도의 널 문자를 넣을 필요가 없다.

sprintf(char *str, const char *format,...) 함수는 세 번째 이하의 인자들과 두 번째 인자인 포맷 스트링을 이용하여 str 버퍼에 데이터를 저장하는 함수로서 str 버퍼의 크기를 초과할 경우 버퍼 오버플로우가 발생한다. 이를 방지하기 위해서는 포맷화 된 문자열의 길이가 str 버퍼의 용량보다 작거나 같아야 한다. sprintf( )함수에 의해 발생할 수 있는 버퍼 오버플로우를 예방하는 코드는 다음과 같이 대체될 수 있다.

```
snprintf(dst, sizeof(dst)-1, "%s", str);
dst[sizeof(dst)-1]=0;
```

gets(char *buf) 함수는 표준 입력으로부터 문자열을 입력받아 buf에 저장하는 함수로 입력 길이를 검사하지 않기 때문에 버퍼 오버플로우가 발생한다. 이를 방지하기 위해서는 fgets(char *buf, int size, FILE *s) 함수를 사용할 수 있다.

```
fgets(buf, sizeof(buf)-1, stdin);
```

위 코드에서 세 번째 인자로 stdin을 지정하였는데 stdin은 표준 입력을 의미한다. fgets( ) 함수 역시 마지막에 널 문자를 자동으로 삽입한다.

앞서 살펴본 취약 함수들은 버퍼의 크기를 고려한 입력 길이를 계산하고 해당 길이만을 입력 값으로 제한하는 대체 함수를 조합하여 안전한 코드로 변환되었다.

scanf(const char *format,...) 함수는 사용자가 몇 바이트를 입력할지 예측이 불가능하며, 함수의 구조상 버퍼의 길이에 따라 입력 값의 길이를 제한할 수 없다.

```
scanf("%s", buf);
```

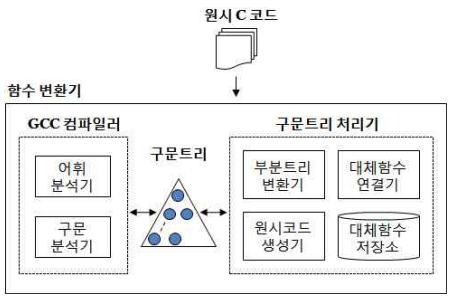
하지만 다음과 같이 할당된 변수의 크기를 참조하여 buf의 오버플로우를 방지할 수 있다.

```
char buf[10];
scanf("%9s", buf);
```

즉, 변수의 크기를 조사한 후 scanf( ) 함수의 포맷 스트링의 형식을 변경할 수 있다. 그 외의 scanf( )와 같이 포맷 스트링 인자 값을 가지는 유사한 함수들도 동일한 방법을 통해서 코드 변환이 가능하다.

3.2 함수 변환기 모듈

<그림 5>는 본 논문에서 제안하는 취약 함수 변환기 모듈의 구성을 나타낸다.



<그림 5> 취약함수 변환기 구조

<그림 5>에서 제안하는 함수 변환기는 원시 C 코드의 문서구조 정보인 구문트리 생성을 위한 GCC 컴파일러 모듈[11] 과 구문트리 순환을 통해 취약 함수를 안전한 대체함수로 변환하기 위한 구문트리 처리기로 구성된다.

<그림 5>의 GCC 어휘분석기와 구문분석기는 입력 원시 C 코드의 문서 파싱을 통해 문서구조 정보인 구문트리를 생성한다. 그리고 구문트리 처리기는 생성된 구문트리를 이용해 원시 C 코드에 포함된 오버플로우 발생 가능 취약 함수의 위치를 인지하고, 발견된 취약 함수 부분을 대체 함수의 부분트리로 대체한다.

<그림 5>의 대체함수 연결기는 발견된 취약함수를 위한 대체함수를 저장소로부터 찾아 연결하며, 부분트리 변환기는 대체함수 연결기가 발견한 특정 대체함수를 부분트리로 변환/생성한 후, 이것을 구문트리에 존재하는 취약함수의 부분트리와 교체한다.

구문트리 처리기는 구문분석 단계에서 생성된 구문트리의 트리순회를 통해 취약함수의 부분트리 위치를 인식하고, 해당 부분트리의 뿌리노드(root node)와 부분트리 영역 위치를 표시(marking)한다. 구문트리 처리기는 반복적인 취약함수 부분트리 인식 및 표시 작업을 통해 원

시 C 코드에 포함된 모든 취약함수의 위치를 파악할 수 있다. 파서를 통해 완성된 구문트리 정보는 구문트리 내에 표시된 취약함수를 포함하는 입력 C 프로그램의 구조 정보를 나타낸다. 따라서 <그림 5>의 부분트리 변환기는 구문트리를 순회하며, 표시된 취약함수의 부분트리를 대체함수의 부분트리로 대체/변환하기 위해 함수 연결기와 유기적으로 연결된다. 제안하는 함수 변환기에 포함된 부분트리 변환기와 대체함수 연결기는 다음과 같은 취약함수 부분트리 인식 및 변환 작업을 수행한다.

① 구문트리에 표시된 취약함수 노드를 부트리(subtree)로 하여, 구문적으로 취약함수 노드의 변환에 영향을 받는 연결 노드 범위를 인식한다.

② 대체함수 부분트리의 속성정보를 수정한다.

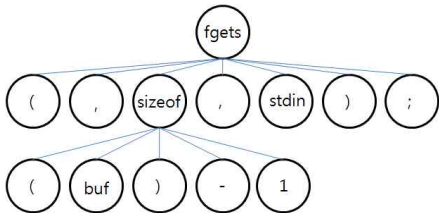
③ 해당 부트리를 미리 정의된 대체함수 부분트리로 대체한다.

변환 작업의 ②에서 속성정보의 수정은 취약함수에 대해 미리 정의된 대체함수 부분트리를 구성하는 함수의 모형(prototype)에 실제 속성 유형과 값을 채워넣는 것을 의미한다.

<그림 5>의 원시코드 생성기는 대체함수의 부분트리로 새롭게 생성된 원시 C 코드의 구문트리에 대해 변경된 원시 C 코드를 생성할 수 있다. 이때, 원시코드 생성기는 구문트리를 대상으로 언파싱 스킴(unparsing scheme)을 통해 새로운 원시 코드를 생성한다. 또한, 원시코드 생성기는 구문트리의 변환된 부분트리 영역에 대해서만 점진적 언파싱(incremental unparsing) 을 실시하여 생성 속도를 향상할 수 있다.

<그림 6>은 <표 1>에 정의된 취약함수 gets( ) 함수를 위해, 함수변환기의 구문트리 처리기와 함수 연결기가 사용하는 대체함수 fgets( )의 부분트리 이다.

<그림 6>에서 fgets( ) 함수 부분트리의 루트 노드인 fgets 노드는 원시 C 코드에 포함되는 gets( ) 함수 부분트리의 루트 노드인 gets 노드를 대체한다. 먼저, 부분트



<그림 6> fgets( ) 함수에 대한 부분트리

리 변환기는 구문분석기가 파싱을 통해 생성한 구문트리를 순회하여 취약함수 gets( )의 정확한 위치를 인식한다. 이후 대체함수 연결기는 발견된 취약함수 gets( )에 대해 적절한 대체함수 fgets( )를 저장소로부터 검색하여 부분트리 변환기에 전달한다. 그리고 마지막으로 부분트리 변환기는 fgets 노드를 루트 노드로 하는 대체함수 fgets( )의 부분트리를 생성하고, gets( ) 부분트리를 대체하여 구문트리 변환을 완료 한다.

```
<Substitute>
/* 컴파일러로부터 원시 C 코드에 대한 AST를 입력받아 대체함수 노드를
삽입한다 */
void traverseTree(AST ast) {
  node = ast. rootNode;
  while (node. child != NULL) {
    if (/* <표 1>에 정의된 취약함수를 발견하면
           처리함수를 호출한다. */) {
      dealFunction(/* 발견노드 */);
    }
    else
      /* 취약함수 노드가 아니면 해당 노드의 자식노드 순회를 계속 한
다 */
  }
}
void dealFunction(Node node) {
  if (ast. node == definitionType. sscanf) {
    /* 취약함수가 sscanf( )이면, 새로운 노드를
       현재 노드 위치의 자식노드로 삽입한다. */
   ast. insertChild(new insertFunction(/* 함수정보 */));
  }
  else
     /* 취약함수노드를 대체함수노드로 대체한다 */
    ast. replaceNode(0, new substituteFunction(/* 대체함수정보 */));
}
```
<그림 2> 관리시스템의 구조도

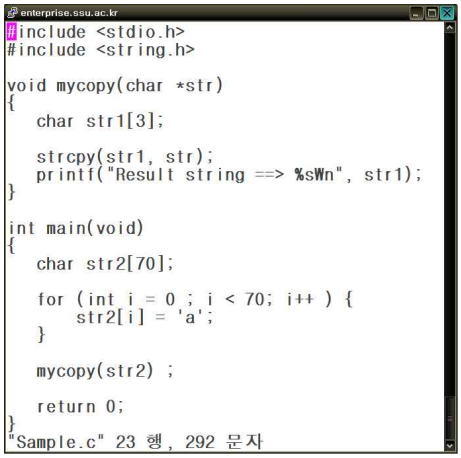
<그림 7>은 구문트리 처리기가 구문트리 순회와 부분트리 대체를 통해 취약함수 변환을 수행하기 위한 알고리즘이다. <그림 7>의 traverseTree( )는 원시 C 코드에 대한 구문트리의 루트를 시작으로 트리순회를 실시하여, 취약함수를 찾아 취약함수 변환처리 루틴인 dealFunction( )을 호출한다. dealFunction은 <표 1>에 정의된 sscanf( ) 취약함수 여부에 따라 다른 변환을 수행한다. 즉, 발견된 취약함수가 sscanf( )인 경우에는 현재 취약함수가 발견된 위치의 sscanf 노드를 루트로 하는 부분트리의 형제노드로 버퍼크기 및 입력 변환 규칙을 기술하기 위한 노드를 새롭게 삽입한다. 그러나 발견된 취약함수가 sscanf( ) 가 아닌 경우에는 해당 취약함수의 이름을 루트 노드로 하는 부분트리를 미리 정의된 변환 부분트리로 대체한다.

전달된 대체함수는 구문트리 처리기에 의해 정확한 위치로 인식된 취약함수와 대치 변환되어 새로운 변환 C 코드를 생성한다. 그리고 변환된 새로운 C 코드는 gcc 컴파일러의 파싱을 통해 실행 가능한 C 실행코드로 생성된다.

Ⅳ. 실험 및 평가

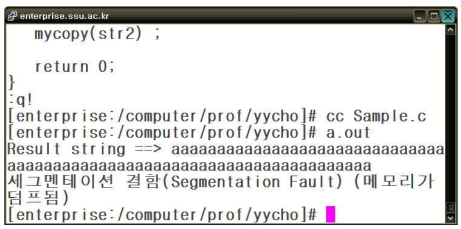
본 장에서는 제안하는 취약함수 변환기가 취약함수가 포함된 예제 원시 C 코드를 입력 받아 취약함수의 정확한 변환과 생성된 코드의 올바른 실행 여부에 대해 실험한다. <그림 8>은 실험을 위해 사용되는 취약함수 strcpy( )가 포함된 예제 원시 C 프로그램인 Sample.c 이다. 실험은 GCC version 3.3을 이용했으며, strcpy( ) 취약함수가 포함된 간단한 예제 C 코드를 대상으로 Solaris 9 OS와 2기가 메모리가 설치된 Enterprise 서버 시스템에서 실행하였다.

<그림 8>에서 함수 mycopy( )에서 선언된 문자 배열 str1[3]의 크기는 main( )에서 선언된 문자 배열 str2[70]에 비해 월등히 작은 크기를 갖는다. 따라서 str2[] 배열



<그림 8> 취약함수 strcpy를 포함한 Sample. c 파일

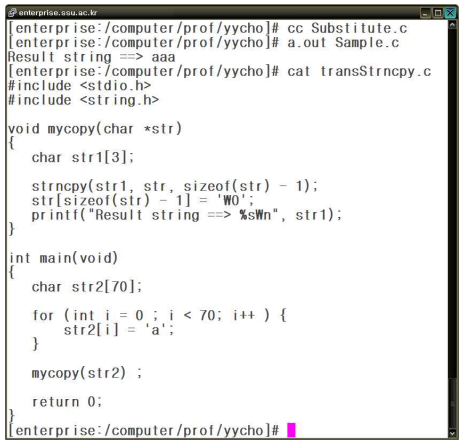
에 저장되는 정보는 str1[] 배열에 올바르게 복사 되지 못하고 오버플로우를 발생시키게 된다. <그림 9>는 오버플로우가 발생된 실행 결과이다.



<그림 9> Sample. c 파일의 실행결과

따라서 본 논문에서 제안하는 취약함수 변환기는 <그림 8>에서 보인 예제 원시 C 프로그램인 Sample. c를 입력으로 받아 취약 함수가 안전한 대체 함수로 변환되어 실행될 수 있는 새로운 C 프로그램을 생성한다. 이때, 새롭게 생성되는 C 프로그램은 기본적으로 transStrncpy.c의 파일 이름을 갖는다.

<그림 10>은 취약함수 변환기가 Sample. c 소스파일에 포함된 취약함수 strcpy( )를 새로운 대체함수 strncpy( )로 대체 생성한 실행 결과이다. <그림 10>에서



<그림 10> 취약함수 변환기를 통한 Sample. c 파일의 실행결과

제안하는 취약함수 변환기의 실행을 위한 매개변수로 취약함수가 포함된 입력함수인 Sample. c 소스파일을 사용하였으며, 그 결과물로 새로운 transStrncpy. c 소스파일이 생성되었다. 새롭게 생성된 transStrncpy. c 는 초기 입력 소스파일인 Sample. c 에 포함되어 있던, strcpy( ) 함수 부분이 strncpy( ) 부분으로 변환된 부분을 포함한다.

<표 2>는 기존의 연구들과 제안하는 변환기의 비교 분석 결과이다. 정적 분석을 통해 버퍼 오버플로우 공격 가능성을 탐지하는 ITS4, BOON, ARCHER와 같은 도구들은 개발자에게 소스코드의 취약점 여부를 알려주며 운영체제와 상관없이 사용할 수 있다는 장점이 있다. 하지만 이들은 취약점이 내재된 소스코드나 실행파일을 안전한 소스코드 및 실행파일로 변환하는 기능이 포함되어 있지 않기 때문에 버퍼 오버플로우의 위험성을 인식하지 못하거나 안전한 프로그램 코딩 작성 방법을 모르는 개발자들에게는 큰 도움이 되지 못한다. 아울러 기존에 생성된 모든 취약 프로그램들을 안전한 프로그램으로 변환하기 위해서는 정적 분석 도구로부터 생성된 취약점 분석 결과를 기반으로 개발자가 직접 소스코드를 재작성해야 하기 때문에 많은 시간과 노력이 요구된다.

컴파일러를 강화하여 구현되는 Stackguard나 stackshield의 경우에는 함수의 리턴 주소를 별도의 공간에 저장한 후, 함수가 리턴할 때 미리 저장된 주소와 비교해야 하기 때문에 추가적인 메모리 공간과 함께 실행 시 비교에 의한 오버헤드가 발생하여 프로그램 실행 시간이 증가한다. 또한 이들 방법은 공격자가 시행착오를 통해 버퍼 오버플로우 공격에 성공할 수 있다는 문제가 있다.

커널 인증 시스템은 커널 수준에서 버퍼 오버플로우 공격을 차단하지만 커널 코드의 추가로 커널 성능이 저하된다. 특히 루트 권한의 프로세스가 생성될 때마다 관리자 패스워드 인증 절차를 거쳐야 하기 때문에 안전한 프로그래밍 기법을 통해 개발된 프로그램이 실행될 경우에도 프로세스 인증 절차를 거치게 되어 불필요한 오버헤드가 증가하게 된다. 그리고 커널 코드의 수정을 요구하기 때문에 운영체제 종류나 버전에 독립적이지 못하다는 단점이 있다.

제안하는 취약함수 변환기는 개발자의 무의식적인 취약함수 사용에 대해 버퍼 오버플로우를 회피할 수 있는 대체 함수로 변환하기 때문에 취약점 존재를 사전에 예방할 수 있으며, 운영체제 종류와 버전에 상관없이 사용할 수 있다는 장점이 있다. 아울러 기존에 생성된 모든 취약 프로그램들은 취약함수 변환기를 통해 안전한 프로

<표 2> 기존 연구와 제안 변환기의 비교

버퍼 오버플로우 대응 방법	실행파일 내 취약점 존재 여부	실행 시 오버헤드 유발	플랫폼 독립성	소스코드 변화 여부
ITS4, BOON, ARCHER	O	X	O	X
Stackguard, stackshield	O	O	O	X
커널 인증 시스템	O	O	X	X
제안하는 변환기	X	X	O	O

그램으로 변환할 수 있기 때문에 프로그램의 보안성을 보장하고 개발자의 생산성을 높일 수 있다. 마지막으로 취약함수 변환기는 기존의 다른 연구들과 달리 소스코드 내에 포함된 취약 함수 구문을 안전한 함수가 포함된 문장으로 확장하기 때문에 전체 소스코드와 실행파일의 크기가 다소 증가하지만 안전한 프로그램을 생성한다는 점에 더 큰 의미가 있다고 판단된다.

Ⅴ. 결론

버퍼 오버플로우 공격에 대응하기 위한 기존 연구들은 단순히 취약한 함수 사용에 대한 경고만을 제공하거나 특정 메모리 영역의 변조 여부를 조사하여 대응하는 방법들이기 때문에 시행착오를 통한 공격에 여전히 취약하다. 아울러 커널 모듈 기반의 대응 방법은 커널 수준의 제어가 필요하기 때문에 커널 성능에 변화를 가져오는 문제가 있다.

이에 본 논문에서는 버퍼 오버플로우 공격에 취약한 함수를 포함한 프로그램 생성을 사전에 방지하기 위한 취약 함수 변환기를 제안하였다. 취약 함수 변환기는 소스코드로부터 구문트리를 생성하고 이로부터 존재하는 취약 함수의 위치를 탐색한다. 탐색된 취약 함수는 대체 함수 저장소에 저장된 안전한 함수를 기반으로 새롭게 변형되기 생성되기 때문에 버퍼 오버플로우 공격을 예방한다. 기존 연구와 비교할 때 취약 함수 변환기는 사전에 취약한 프로그램 생성을 막을 수 있으며, 실행 시에 버퍼 오버플로우 탐지를 하지 않는다는 측면에서 커널 및 응용 프로그램의 성능 저하를 막을 수 있다는 장점이 있다.

향후 연구 과제로는 포맷 스트링 공격에 취약한 함수를 프로그래머가 올바르게 사용하지 않았을 때 이를 탐지하여 안전한 형식으로 수정하는 기능을 추가하는 것이다.

참고문헌

[1] J. Viega, J. Bloch, T. Kohno and G. McGraw, “ITS4: A static vulnerability scanner for c and c++ code,” In proceeding of the 16th Annual Computer Security Applications Conference, 2000.

[2] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, “A First Step towards Automated Detection of Buffer Overrun Vulnerabilities,” In Proceedings of the Network and Distributed System Security Symposium, 2000.

[3] Y. Xie, A. Chou, and D. Engler, “ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Acess Errors,” In Proceedings of the 9th European Software Engineering Conference, 2003.

[4] Available at http://www.coverity.com/

[5] Available at http://www.polyspace.com/

[6] C. Cowan, C. Pu, D. Maier, H. Ginton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks,” In proceeding of the 7th conference on USENIX Security, 1998.

[7] R. Seacord, “Secure Coding in C and C++,” Addison Wesley, 2005.

[8] E. Gaugh and M. Bishop, “Testing C Programs for Buffer Overflow Vulnerabilities,” In proceedings of the Network and Distributed System Security Symposium, 2003.

[9] J. Pincus and B. Baker, “Beyond stack smashing: Recent advances in exploiting buffer overruns,” IEEE Security and Privacy, Vol. 2, No. 4, 2004, pp. 20-27.

[10] 김익수, 김명호, “관리자 인증 강화를 위한 추가적인 패스워드를 가지는 보안커널모듈 설계 및 구현,”

정보처리학회 논문지, 제10-C권, 제6호, 2003, pp. 675-682.

[11] Kurt Wall, William Von Hagen, "The GCC Book,"APress, October 2003.

■ 저자소개 ■

김 익 수
Kim, Ik Su

2009년 9월~현재 숭실대학교 컴퓨터학부 조교수
2008년 2월 숭실대학교 컴퓨터학과 (공학박사)
2002년 2월 숭실대학교 컴퓨터학과 (공학석사)
2000년 2월 숭실대학교 컴퓨터학과 (공학사)

관심분야 : 시스템 보안, 네트워크 보안, 모바일 보안, 시스템 소프트웨어
E-mail : skycolor@ss.ssu.ac.kr

조 용 윤
Cho, Yong Yun

2009년 ~현재 국립순천대학교 정보통신공학부 조교수
2006년 숭실대학교 컴퓨터학과 (공학박사)
1998년 숭실대학교 컴퓨터학과 (공학석사)
1995년 인천대학교 전산학과 (공학사)

관심분야 : 시스템 소프트웨어, 임베디드 소프트웨어, 유비쿼터스 컴퓨팅
E-mail : sslabyycho@hotmail.com

논문접수일 : 2009년 1월 4일
수 정 일 : 2010년 2월 10일
게재확정일 : 2010년 2월 17일