심볼릭 링크 공격 취약성 검출을 위한 분석 기법

주 성 용[†] · 안 준 선[††] · 조 장 우[†††]

요 약

본 논문에서는 심볼릭 링크 공격에 취약한 코드를 정의하고 프로그램 분석 기법을 사용하여 이를 검출하는 방법을 제안한다. 심볼릭 링크 공격을 해결하기 위한 기존의 방법들은 심볼릭 링크 공격을 방어하기 위한 기법들로써 임시 파일에 대한 접근 시 공격이 이루어졌는지에 대한 적절한 검사가 이루어져야 하나, 이를 간과할 경우 공격의 위험에 노출되게 된다. 본 논문에서 제안하는 방법은 심볼릭 링크 공격에 취약한 부분을 자동으로 모두 검출함으로써 프로그래머가 심볼릭 링크 공격을 안전하게 방어할 수 있도록 한다. 제안하는 방법은 취약점 분석을 위해서 기존의 타입 시스템에 새로운 타입 한정자를 추가하고, 추가된 타입 한정자를 고려한 타입 검사를 통해서 심볼릭 링크 공격의 취약점을 식별한다. 제안하는 방법은 자동으로 심볼릭 링크 공격의 취약점을 모두 검출할 수 있기 때문에, 프로그래머가 취약한 것으로 식별된 코드에 대해서만 기존의 방어 기법을 적용하도록 함으로써 프로그램을 전반적으로 검토하거나 수정해야 하는 부담을 줄여준다는 장점을 가진다. 제안하는 방법을 널리 알려진 실용적인 프로그램을 대상으로 실험해 본 결과 전체 fopen() 함수 중 일부만 심볼릭 링크 공격에 취약한 것으로 분석되었으며, 이는 제안한 방법이 프로그래머의 부담을 줄이는데 유용함을 보여준다.

키워드 : 심볼릭 링크 공격, 경쟁 조건 탐색, 흐름독립 분석, 소프트웨어 취약점, 소프트웨어 보안

An Analysis Method for Detecting Vulnerability to Symbolic Link Exploit

Seongyong Joo[†] · Joonseon Ahn[††] · Jang-wu Jo[†††]

ABSTRACT

In this paper we define a vulnerable code to symbolic link exploit and propose a technique to detect this using program analysis. The existing methods to solve symbolic link exploit is for protecting it, on accessing a temporary file they should perform an investigation whether the file is attacked by symbolic link exploit. If programmers miss the investigation, the program may be revealed to symbolic link exploit. Because our technique detects all the vulnerable codes to symbolic link exploit, it helps programmers keep the program safety. Our technique add two type qualifiers to the existing type system to analyze vulnerable codes to symbolic link exploit, it detects the vulnerable codes using type checking including the added type qualifiers. Our technique detects all the vulnerable codes to symbolic link exploit automatically, it has the advantage of saving costs of modifying and of overviewing all codes because programmers apply the methods protecting symbolic link exploit to only the detected codes as vulnerable. We experiment our analyzer with widely used programs. In our experiments only a portion of all the function fopen() is analyzed as the vulnerabilities to symbolic link exploit. It shows that our technique is useful to diminish modifying codes.

Keyword : SymboLic Link Exploit, Race Condition Detection, Flow-Insensitive Analysis, Software Vulnerabilities, Software Security

1. 서 론

소프트웨어 보안 문제를 해결하기 위해서 프로그램 분석 방법을 사용하는 연구가 활발하게 이루어지고 있다[1-3,9,11,12,14]. 이러한 접근법은 특정 공격에 대한 프로그램의 안전성을 프로그래머의 큰 도움 없이 자동화된 방법으로 보장할 수 있다는 장점을 가진다. 심볼릭 링크 공격은 심볼릭 링크와 프로세스간의 경쟁 조건(race condition)을 이용한 공격으로 프로그램 실행 중 fopen() 함수로 생성되는 임시 파일을 공격 대상으로 하는 널리 알려진 공격 방법이다[7,10,13]. 본 연구에서는 심볼릭 링크 공격에 취약한 코드를 정의하고 프로그램 분석 기법을 사용하여 이를 검출하는 방법을 제안한다.

심볼릭 링크 공격을 해결하기 위한 기존의 방법들로는 임시 파일을 사용하기 전 심볼릭 링크인지 확인 후 사용하는 방법, 신뢰하는 파일명을 사용하는 방법, 공유 속성을 제거하는 방법 등이 있다[10]. 이러한 방법을 통하여 심볼릭 링크 공격을 방어할 수는 있으나 프로그래머가 취약한 코드 부분에 대하여 이러한 대책을 간과할 경우 공격에 취약한

† 준 회 원 : 동아대학교 대학원 컴퓨터공학과 박사과정
†† 정 회 원 : 한국항공대학교 항공전자 및 정보통신공학부 교수
††† 정 회 원 : 동아대학교 컴퓨터공학과 부교수(교신저자)
논문접수 : 2007년 11월 1일, 심사완료 : 2007년 12월 21일

DOI: 10.3745/KIPSTA.2008.15-A.1.45

프로그램이 생성되게 된다. 본 연구에서는 심볼릭 링크 공격에 취약한 부분을 자동으로 모두 검출함으로써 프로그래머가 심볼릭 링크 공격을 안전하게 방어할 수 있도록 하고자 한다.

본 논문에서 제안하는 분석에서는 기존의 타입 시스템에 새로운 타입 한정자(Type Qualifier)[1-3,11,14]를 추가하고, 추가된 타입 한정자를 고려한 타입 검사를 통해서 심볼릭 링크 공격의 취약점을 식별하는 방법을 사용한다. 이러한 타입 한정자 기반 분석은 기존의 타입 시스템으로 표현할 수 없는 속성들을 검사하기 위한 가볍고 실용적인 메커니즘으로서[3], 타입 한정자를 이용한 기존 연구로 포맷 스트링 취약점 검출 기법이 발표된 바 있다[11].

본 연구에서 제시하는 방법은 자동으로 심볼릭 링크 공격의 취약점을 모두 검출할 수 있기 때문에, 프로그래머가 취약한 것으로 식별된 코드에 대해서만 기존의 방어 기법을 적용하도록 함으로써 프로그램을 전반적으로 검토하거나 수정해야 하는 부담을 줄여준다는 장점을 가진다. 본 논문의 방법을 널리 알려진 실용적인 프로그램을 대상으로 실험해 본 결과 전체 fopen() 함수 중 일부만 심볼릭 링크 공격에 취약한 것으로 분석되었으며, 이는 제안한 방법이 프로그래머의 부담을 줄이는데 유용함을 보여준다.

이 논문의 구성은 다음과 같다. 2장에서는 심볼릭 링크 공격 기법을 소개하고, 3장에서 심볼릭 링크 공격에 취약한 코드를 검출하는 방법을 설명한다. 4장에서는 제안된 방법에 기반 한 분석기의 구현에 대하여 설명한다. 5장에서는 실험결과를 보이며, 6장에서 결론을 맺는다.

2. 심볼릭 링크 공격

2.1. 심볼릭 링크 공격 방법

심볼릭 링크 공격은 경쟁 조건 공격의 한 종류로서 Linux와 같은 Unix계열의 운영체제에서 제공하는 심볼릭 링크를 이용한 공격 기법이다[10,13]. C 프로그램은 프로그램 실행 중 임시 파일을 생성하고, 그 파일을 읽거나 쓰는 연산을 수행 하는 경우가 있다. Unix 계열의 운영체제는 다중 프로그래밍을 지원하기 때문에 프로세스들 사이에 자원 점유를 위한 경합이 발생할 수 있다. 만일 파일 연산을 수행하는 프로세스가 경쟁 조건에 의해서 실행 상태에서 대기 상태로 전이될 수 있는데, 이 때 공격자는 사용 중인 파일이나 생성할 파일을 심볼릭 링크를 이용해서 다른 파일로 변경할 수 있는 기회를 가지게 된다. 이 같은 특징을 이용한 공격이 심볼릭 링크 공격이다. 심볼릭 링크 공격은 다음과 같은 절차로 이루어진다.

1. 대상 프로그램에서 생성하는 임시 파일명 식별
2. 대상 프로그램이 임시 파일을 생성
3. 대상 프로그램이 프로세스 경합이 발생할 때 까지 대기
4. 프로세스 경합이 발생하고 대상 프로그램이 대기 상태로 진입
   4.1. 대상 프로그램이 사용하고 있는 파일을 삭제
   4.2. 4.1에서 삭제한 파일과 동일한 이름으로 심볼릭 링크 생성
5. 대상 프로그램이 다시 실행 상태로 진입하고 생성한 파일에 내용을 기록하면 2에서 생성된 파일이 아닌 4.2에서 심볼릭 링크로 연결된 파일에 내용을 기록하게 됨

(그림 1)은 프로그램 실행 중 생성하는 임시 파일에 대한 심볼릭 링크 공격의 예를 보이기 위한 코드이다. (그림 1)의 프로그램은 사용자로부터 파일에 기록할 내용을 입력 받고, 그 내용을 "/tmp/temp.txt" 파일에 기록하는 프로그램이다.

```
 1: int main (int argc, char * argv []) {
 2:  struct stat st;
 3:  FILE * fp;
 4:  char *fname = "/tmp/temp.txt";
 5:
 6: if ((fp = fopen (fname, "w")) == NULL)
 7: {
 8:    fprintf (stderr, "Can't open \n");
 9:    exit(EXIT_FAILURE);
10: }
11:
12:  fprintf (fp, "%s \n", argv [1]);
13:  fclose (fp);
14:  fprintf (stderr, "Write Ok \n");
15:
16:  exit(EXIT_SUCCESS);
17:}
```

(그림 1) 임시 파일에 대한 심볼릭 링크 공격의 예

(그림 1)의 프로그램이 7번째 줄에서 11번째 줄의 코드를 수행 중에 프로세스 경합으로 대기 상태로 전이되면 공격자는 공격할 틈을 갖게 된다. (그림 2)와 같은 형태로 공격은 이루어진다.

```
> rm /tmp/temp.txt
> ln -s /etc/passwd /tmp/temp.txt
```

(그림 2) 심볼릭 링크 공격의 예

(그림 2)의 공격이 이루어지고 대기 중인 프로세스가 실행상태로 복귀하면, (그림 1)의 12번째 줄에서는 6번째 줄에서 생성한 "/tmp/temp.txt"가 아닌 심볼릭 링크에 연결된 파일인 "/etc/passwd"에 사용자로부터 입력받은 값을 기록하게 된다.

2.2 심볼릭 링크 공격을 방어하기 위한 기존의 방법

심볼릭 링크 공격을 방어하기 위한 기존의 방법은 프로그램 작성 시 몇 가지 조건 검사를 추가하는 것이다. (그림 3)은 심볼릭 링크 공격을 방어하기 위해서 조건 검사를 추가한 코드이다[10].

```
1:        ...
2: struct stat lstat_info, fstat_info;
3: int fd;
4:
5: if (lstat("some_file", &lstat_info) == -1) {
6:        err(1, "lstat");
7: }
8:
9: if ((fd = open("some_file", O_EXCL |
                    O_RDWR, 0600)) = -1) {
10:       err(2, "some_file");
11: }
12:
13: if (fstat(fd, &fstat_info) == -1) {
14:       err(3, "fstat");
15: }
16:
17: if (lstat_info.st_mode == fstat_intfo.st_mode
        && lstat_info.st_ino == fstat_intfo.st_ino)
18: // 파일 처리
19:       ...
```

(그림 3) 심볼릭 링크 공격을 회피하기 위한 조건 검사

(그림 3)은 시스템에 이미 존재하는 파일을 열고자 할 때, 파일을 열기 전에 lstat() 함수를 이용해서 그 파일이 실제 파일인지 심볼릭 링크인지를 확인한 후 파일을 연다. 그러나 7번째 줄과 9번째 줄 사이에서도 공격자가 심볼릭 링크 공격을 수행할 수 있기 때문에 17번째 줄에서는 개방된 파일을 사용하기 전 5번째 줄에서 취득한 파일 디스크립터와 13번째 줄에서 취득한 파일 디스크립터가 일치하는지 확인한 후 파일을 사용한다.

(그림 3)과 같은 코드를 추가하는 것은 응용프로그램의 본질적인 문제가 아니기 때문에 실수로 간과하기 쉽고 모든 파일 개방 연산에 대해서 이 같은 코드를 추가하는 것 역시 프로그래머와 프로그램 실행에 과중한 부담이 된다. 또한 이미 작성된 프로그램의 경우 모든 소스를 검토하는 것 역시 쉬운 일은 아니다. 이 같은 문제를 해결하기 위해서 먼저 심볼릭 링크 공격에 취약한 코드를 찾아내고, 취약점으로 분석된 코드에 한해서 (그림 3)의 코드를 적용함으로써 프로그래머의 부담을 줄일 수 있다.

3. 심볼릭 링크 공격에 취약한 코드 검출 기법

본 장에서는 심볼릭 링크 공격에 취약한 코드를 정의하고, 심볼릭 링크 공격에 취약한 코드를 찾아내기 위해서 타입 한정자를 이용하는 방법을 제안한다. 그리고 타입 한정자를 고려한 타입 추론 기법을 기술한다.

3.1 심볼릭 링크 공격에 취약한 코드 정의

심볼릭 링크 공격을 하기 위해서는 임시파일을 제거하고 임시 파일과 같은 이름의 심볼릭 링크를 생성해야 된다. 이를 위해서 공격자는 대상 프로그램에서 사용하는 임시 파일명을 정확하게 알아야 한다. 프로그램 내에서 임시 파일명을 문자열 상수 형태로 사용하고 있다면 공격자는 공개된

```
00000df0  01 00 00 0E 00 00 00 0A  00 54 68 69 73 20 66 69  .........This fi
00000e00  6C 65 20 69 73 20 66 6F  72 20 74 65 73 74 69 6E  le is for testin
00000e10  67 2E 00 77 00 2F 74 6D  70 2F 74 65 6D 70 2E 74  g..w./tmp/temp.t
00000e20  78 74 00 00 00 00 00 00  00 00 00 23 00 00 00 06  xt.........#....
00000e30  00 00 00 00 00 00 00 00  00 28 00 00 00 06 00 04  .........(......
```

(그림 4) 이진 코드 편집기를 이용한 파일명 식별

소스 코드를 통해서 임시 파일명을 알아낼 수도 있고, 이진 편집기를 이용해서 대상 프로그램의 실행 이미지로부터 임시 파일명을 알아낼 수 있다. (그림 4)는 실행 파일을 일반적인 이진 코드 편집기를 통해서 살펴 본 것이다. (그림 4)에서 이 프로그램이 "/tmp/temp.txt"라는 파일을 사용하고 있음을 알 수 있다. 이 같은 경우 특별한 지식이나 도구 없이도 쉽게 대상 프로그램에서 사용하는 임시 파일명을 알아낼 수 있다. 그러므로 본 논문에서는 파일 개방 연산을 위해서 사용되는 임시 파일명이 문자열 상수 형태로 사용된다면 심볼릭 링크 공격에 취약한 코드로 정의한다.

본 논문에서는 제안하는 시스템의 설명을 용이하게 하기 위해서 심볼릭 링크 공격의 취약점을 문자열 상수 형태의 파일 이름으로 한정하였다. 그러나 문자열 상수 형태의 파일 이름이 아니라도 동적인 모니터링 등으로 임시 파일 이름을 알 수 있는 경우가 있다. 그러므로 심볼릭 링크 공격의 취약한 코드의 정의를 랜덤하게 생성되거나 암호화된 형태를 제외한 문자열로 변경할 수 있다. 이와 같이 심볼릭 링크 공격에 취약한 코드 정의를 변경하더라도 본 논문에서 제안하는 시스템을 변경하지 않고 취약점을 검출할 수 있음을 4.5절에서 보인다.

3.2 타입 한정자 : trusted와 untrusted

본 논문에서는 심볼릭 링크 공격에 취약한 코드를 판별하기 위하여 기존의 일반적인 타입정보에 추가로 타입 한정자 trusted와 untrusted를 정의하여 사용한다. 타입 한정자 trusted는 소스 코드로부터 알아낼 수 없는 문자열을 지정하기 위해 사용된다. trusted 값의 예로 실행 중에 외부로부터 읽어오는 문자열이나 난수 생성 등을 통하여 생성된 이름 등이 있다. 타입 한정자 untrusted는 소스 코드로부터 알아낼 수 있는 문자열을 지정하기 위해서 사용된다. untrusted 값의 예로 문자열 상수가 있다.

타입 한정자 trusted와 untrusted 사이에는 다음의 관계가 성립하며 이는 타입 분석 시에 일반적인 타입 유추와 함께 적용된다.

$$\text{untrusted} \not\leq \text{trusted}$$
$$\text{trusted} \leq \text{untrusted}$$

untrusted $\not\leq$ trusted 관계는 trusted로 선언된 위치에 untrusted 값이 오는 것을 허용하지 않음을 나타내며, trusted $\leq$ untrusted 관계는 untrusted로 선언된 위치에 trusted 값이 오는 것을 허용하는 것을 의미하며, 이는 untrusted로 선언한 것은 심볼릭 링크 공격과 무관한 값을 의미하기 때문이다. 결과적으로 타입 한정자 untrusted는

trusted의 슈퍼타입이 된다 (trusted < untrusted).

타입 한정자 사이에 슈퍼타입 관계를 이용해서 분석의 정확성을 높일 수 있다. 예를 들어 슈퍼타입 관계가 존재하지 않는다면 untrusted로 선언된 위치에 trusted 값이 오는 것은 타입 불일치가 되고 타입 오류를 보고한다. 그러나 untrusted로 자리에 trusted 값이 오는 것은 심볼릭 링크 공격과 무관하다. 그러므로 이 보고는 거짓 경고(false alarm)가 된다. 이 경우 슈퍼타입 관계를 이용하게 되면 untrusted로 선언된 위치에 trusted 값이 오는 것을 허용하기 때문에 거짓 경고를 줄일 수 있다.

3.3 타입 한정자를 사용한 함수 선언

타입 한정자를 사용하여 심볼릭 링크 공격에 취약한 코드를 찾는 방법은 대상 프로그램의 문자열과 관련된 모든 식에 대해서 타입 한정자를 선언하고, 타입 검사를 수행하여 타입이 일치하는지를 살펴보는 것이다. 그러나 대상 프로그램의 문자열과 관련된 모든 식에 대해서 타입 한정자 선언을 추가하는 것은 프로그래머에게 부담이다. 본 논문에서는 이러한 부담을 해결하기 위해서 문자열과 관련된 라이브러리 함수에 대해서만 타입 한정자를 선언하고, 나머지 코드의 타입 한정자는 타입 추론을 통해서 결정하도록 한다. 그러므로 실제적으로 프로그래머의 부담 없이 타입 한정자 선언은 이루어질 수 있다.

예를 들어 (그림 5)에서 main() 함수의 경우 두 번째 인수는 소스 코드로부터 알 수 없는 문자열이므로 trusted로 선언한다. 그리고 fopen() 함수의 경우 파일 이름을 의미하는 첫 번째 인수는 심볼릭 링크 공격의 대상이기 때문에 trusted로 선언하고, 파일 모드를 의미하는 두 번째 인수는 파일 이름과 무관하기 때문에 untrusted로 선언한다. 이러한 타입 정보를 이용해서 타입 검사를 수행한다. 만일 trusted로 선언된 인수의 값으로 untrusted 값이 온다면 타입 오류를 유발하고, 이것은 심볼릭 링크 공격에 취약한 코드가 된다.

(그림 5)는 타입 한정자를 이용하여 심볼릭 링크 공격에 취약한 코드를 찾아내는 예이다. 1번째 줄에서 fopen() 함수의 첫 번째 형식인수는 trusted로 선언되었다. 7번째 줄에서 fopen() 함수의 첫 번째 인수로 fname1이 전달되었는데, fname1은 trusted로 선언된 main() 함수의 인수 argv로부터 전파된 값이기 때문에 trusted이다. 그러므로 7번째 줄의 fopen()은 첫 번째 형식인수와 실인수의 타입 한정자가 일치하기 때문에 심볼릭 링크 공격으로부터 안전하다. 프로그램에서 사용된 문자열 상수의 기본 타입 한정자는 untrusted이므로 6번째 줄의 fname2는 untrusted 이다. 8번째 줄에서 fopen() 함수의 첫 번째 인수는 trusted여야 하는데 untrusted인 fname2가 왔기 때문에 오류를 유발한다. 그러므로 8번째 줄은 심볼릭 링크 공격에 취약한 코드가 된다.

```
1 : FILE *fopen(trusted const char
      *filename, untrusted const char *mode );
2 :
3 : void main(int argc, trusted
                        char* argv[])
4 : {
5 :       char *fname1 = argv[1];
6 :       char *fname2 = "d.txt";
7 :       fopen(fname1, "w");
8 :       fopen(fname2, "w");
9 :     ...
10: }
```

(그림 5) 심볼릭 링크 공격에 취약한 코드의 예

4. 타입 추론에 기반 한 취약성 분석기의 구현

본 장에서는 취약성 분석기의 동작 구조를 설명한다. 개발된 분석기는 먼저 프로그램을 입력으로 받아 타입 한정자를 고려한 타입 제약식을 생성하며, 이러한 제약식을 풀어 그 해를 생성할 수 있을 경우 입력 프로그램이 안전함을 보장할 수 있게 된다. 타입 오류가 있을 경우 제약식의 해가 생성되지 못하며, 이 경우 보안 취약성이 존재할 수 있음을 나타낸다.

4.1 타입 추론을 위한 타입 제약식의 생성

타입 한정자를 이용한 타입 검사는 프로그램의 모든 식에 타입 한정자가 명시된 프로그램을 대상으로 타입이 일치하는지 검사하는 것이다. 예를 들어, 배정문 x=y에서 y의 타입이 x의 서브타입이면 타입이 일치하고, 함수 호출 f(x)에서 x의 타입이 함수 f()의 형식인수 타입의 서브 타입이면 타입이 일치한다. 그러나 프로그램의 모든 식에 타입 한정자를 명시하는 것은 프로그래머에게 부담이 될 수 있으며, 기 작성된 소스의 경우 타입 한정자를 추가하는 것은 어려운 일이다. 본 연구에서는 이런 부담을 줄이기 위해서 타입 추론을 이용한 방법을 제안한다.

타입 추론을 위해서는 프로그램의 모든 부분식들의 타입 관계를 식으로 표현한 타입 제약식을 생성하게 된다. 문자열을 인수 또는 반환 값으로 갖는 라이브러리 함수에 대해서만 타입 한정자를 명시하고, 프로그램 내의 모든 부분식들은 타입을 제약식을 사용해서 계산하게 된다. 생성한 제약식에 대해 만족하는 해가 존재하지 않으면 타입 불일치가 발생되고 이는 심볼릭 링크 공격에 취약한 코드를 의미한다.

심볼릭 링크 공격 취약성을 분석하기 위하여 추가된 trusted와 untrusted 타입 한정자와 관련한 타입 제약식 생성 규칙은 <표 1>과 같다.

〈표 1〉 타입 제약식 생성 방법

식	예	타입 제약식
배정문	x = exp	exp_q ≤ x_q
라이브러리 함수 선언	Q_1 char * f(Q_2 char *x)	f_ret ≤ {Q_1} f_arg0 ≤ {Q_2}

반환문	char* f(x) {
	...
	return e;
	}
		Q_e ≤ f_ret

함수 호출문	f(x)	x_q ≤ f_arg0 f_ret ≤ f_q

<표 1>의 타입 제약식에서 사용된 타입 한정자 변수의 명명법은 표현식과 변수 그리고 함수의 경우 뒤에 _q를 추가해서 표현하고, 함수 반환 값의 타입 한정자 변수는 함수명에 _ret을 추가해서 표현한다. 함수 인수에 대한 타입 한정자 변수는 "함수명 ++ _arg ++ 선언된 순서"로 표현한다. 선언된 순서는 0부터 1씩 증가하는 값이다. 예를 들어서 함수 f(x)의 타입 한정자 변수는 f_q이고, 반환 값에 대한 타입 한정자 변수는 f_ret이다. 그리고 인수 x에 대한 타입 한정자 변수는 f_arg0이다.

본 논문에서 문자열 상수의 타입 한정자는 untrusted로 정의한다. 문자열을 인수나 반환 값으로 가지는 라이브러리 함수는 함수 원형에 타입 한정자를 선언하여야 한다. 예를 들면 fopen() 함수의 경우 (그림 6)과 같이 선언할 수 있다.

<표 1>의 배정문 x = exp에서 생성되는 타입 제약식은 exp_q ≤ x_q이다. exp_q는 식 exp에 대한 타입 한정자 변수이고, x_q는 변수 x에 대한 타입 한정자 변수이다. 배정문 우측의 표현식의 타입 한정자는 배정문 좌측의 변수 타입의 서브 타입이어야 하므로, exp_q ≤ x_q를 생성한다.

본 논문에서 함수는 두 가지로 구분한다. 첫째는 사용자 정의 함수같이 함수 몸체의 소스코드가 있는 경우이고, 나머지는 라이브러리 함수와 같이 함수 몸체의 소스코드가 없는 경우이다. 함수 몸체의 소스코드가 있는 경우는 타입 추론을 이용해서 함수의 타입 한정자와 인수의 타입 한정자를 결정할 수 있지만, 소스코드가 없는 경우는 타입 추론이 불가능하다. 그러므로 이 경우는 사용자가 직접 타입 한정자를 명시해야 한다. 본 논문에서 관심의 대상은 문자열을 반환하거나 조작하는 함수이기 때문에 이런 종류의 라이브러리 함수들에 대해서만 타입 한정자를 명시하면 된다. 타입 한정자를 명시하는 기준은 외부로부터 전달 받는 문자열과 프로그램 내부에서 조작되는 문자열은 trusted로 선언하고, 파일명과 무관하게 사용되는 문자열은 untrusted로 선언한다. 예를 들어, gets()와 strcat()의 경우 다음과 같이 타입 한정자를 선언한다.

```
trusted char *gets(trusted char *s);
trusted char *strcat(trusted char *s1, untrusted const
  char *s2);
size_t strlen(untrusted const char *s);
```

gets()는 콘솔로부터 문자열을 입력 받아서 s에 그 값을 저장하고, 입력 받은 문자열을 반환하는 함수이기 때문에 인수와 반환 값의 타입 한정자 둘 다 trusted로 선언한다. strcat()은 두 문자열 s1과 s2를 결합해서 s1에 저장하고, 그 결과를 반환하는 함수이다. s1은 입력과 출력 모두에 사용되는 값이기 때문에 trusted로 선언하고, 두 번째 인수 s2는 입력으로만 사용되며 함수 실행 후에 그 결과가 변경되지 않기 때문에 untrusted로 선언한다. 이 함수의 결과 값은 입력 값과 다른 결과를 제공하기 때문에 trusted로 선언한다. strlen()은 인수로 문자열을 갖지만 문자열 조작 연산을 수행하지 않기 때문에 인수를 untrusted로 선언한다.

반환문의 경우 함수가 지니는 타입 한정자의 서브 타입이어야 한다. <표 1>의 반환문의 경우 return문에 사용된 표현식 e는 사실상 f_ret과 동일한 값이 지만, 타입 추론의 명확성을 위해서 별도의 타입 제약식을 생성한다.

함수 호출문의 경우 실인수의 타입 한정자는 가인수의 타입 한정자의 서브 타입이어야 한다는 타입 제약식과 함수를 수행한 결과 값의 타입 한정자는 함수 자체 타입 한정자의 서브 타입이어야 한다는 타입 제약식을 생성한다.

(그림 6)은 타입 제약식을 생성하는 예를 보여준다.

1 : FILE *fopen(trusted const char *filename, untrusted const char *mode);	fopen_arg0 ≤ {trusted} fopen_arg1 ≤ {untrusted}
2 :	
3 : void main(int argc, trusted char* argv[])	main_arg1 ≤ {trusted}
4 : {	
5 : char *ptF1 = argv[1];	main_arg1 ≤ ptF1_q
6 : char *ptF2 = "d.txt";	untrusted ≤ ptF2_q
7 : fopen(ptF1, "w");	ptF1_q ≤ fopen_arg0 untrusted ≤ fopen_arg1
8 : fopen(ptF2, "w");	
9 : ...	ptF2_q ≤ fopen_arg0 ...
10: }	

(그림 6) 타입 제약식 생성의 예

4.2 제약식의 해 구하기

제약식의 해를 구하는 알고리즘은 [5,7,8]에서 제안되었고, 기존에 제안된 알고리즘을 본 논문에 적용할 수 있도록 간단하게 만든 알고리즘은 (그림 7)과 같다. (그림 7)에서 S는 제약식의 가능한 해의 집합이고, C는 제약식의 집합, Q는 한정자의 집합을 의미한다. 그리고 κ는 타입 한정자 변수이며, q는 타입 한정자이다. (그림 7) 알고리즘은 모든 타입 한정자 변수들과 타입 한정자들로 초기화를 한다. 루틴은 제약식의 집합이 공집합이 될 때까지 반복하며 while문을 종료했을 때 남아 있는 해의 집합이 제약식의 해가된다.

```
SOLVER(C) =
  for all κ ∈ C do S(κ) ← Q
  for all q ∈ Q do S(q) ← {q}
  let C' = C
  while C' ≠ ∅ do
    remove an L ≤ R from C'
    let S'_L = S(L) ∩ S(R)
    let S'_R = S(R) ∩ S(L)
    if S'_L = ∅ or S'_R = ∅
      then return unsatisfiable
    if S(L) ≠ S'_L
      then S(L) ← S'_L
        Add each L' ≤ L and L ≤ R' in C to C'
    if S(R) ≠ S'_R
      then S(R) ← S'_R
        Add each L' ≤ R and R ≤ R' in C to C'
  return S
```

(그림 7) 제약식의 해를 구하기 위한 알고리즘

(그림 7)의 알고리즘을 이용하여 (그림 6)에서 생성된 제약식을 만족하는 해를 구하는 과정은 다음 <표 2>와 같다. <표 2>에서 T와 U는 각각 타입한정자 trusted와 untrusted를 나타내고, while 반복에서 제약식을 제거하는 순서는 (그림 6)에서 제약식이 나열된 순서이다. 그리고 각 반복에서 S'_L 과 S'_R 의 값만을 표현하였고 표현되지 않은 제약변수의 값들은 이전 반복의 값과 동일하다. <표 2>의 while 반복 #8에서 S'_L 과 S'_R 의 값이 $\varnothing$ 이 되어 해가 존재하지 않음을 나타낸다. 해가 존재하지 않는다는 것은 심볼릭 링크 공격에 취약한 코드를 의미한다.

〈표 2〉 제약식의 해를 구하는 과정

제약변수	초기 값	while 반복							
		#1	#2	#3	#4	#5	#6	#7	#8
fopen_arg0	{T, U}	{T}					{T}		∅
fopen_arg1	{T, U}		{U}					{U}	
main_arg1	{T, U}			{T}	{T}				
ptF1_q	{T, U}				{T}		{T}		
ptF2_q	{T, U}					{U}			∅
T	{T}	{T}		{T}					
U	{U}		{U}			{U}		{U}	

4.3 흐름독립 타입 추론과 흐름 기반 타입추론

흐름독립 타입추론은 프로그램 전체에서 변수 등의 타입을 일정하게 유지하면서 분석을 수행하는 방법이다[2,11,14]. 타입 추론의 정확도를 향상시키기 위해 흐름의존 타입추론(flow-sensitive type inference)이 제안되었다. 흐름의존 타입 추론은 값의 타입이 프로그램의 지점에 따라 다른 타입을 가질 수 있다[3,11]. 그러나 흐름의존 분석은 각 프로그램 지점마다 값의 타입을 유지해야 되므로 많은 메모리와 수행시간을 필요로 한다.

(그림 8)은 흐름독립 분석과 흐름의존 분석의 차이를 보이기 위한 예이다. 흐름독립적으로 타입추론을 수행하면 변수 fname의 제약변수의 해는 untrusted이므로 6번째 줄에서 타입 불일치를 유발한다. 그러나 흐름의존적으로 타입추론을 수행하면 변수 fname의 제약변수의 해는 trusted 이므로 6번째 줄에서 타입 불일치가 발생하지 않는다. 흐름독립 타입추론은 실제 취약하지 않은 코드를 취약하다고 판단하는 거짓 경고를 하는 반면, 흐름의존 타입추론에서는 이와 같은 거짓 경고를 줄일 수 있어 정확도를 향상시킬 수 있다.

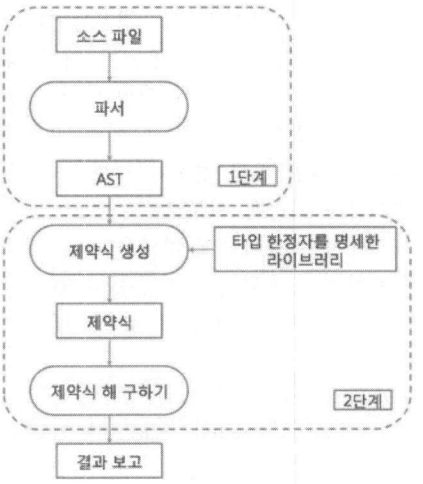
```
1: trusted char* sgets (void);
2:   ...
3: char *fname = "test1.txt";
4:   ...
5: fname = sgets();
6: fopen(fname, "r");
7:   ...
```

(그림 8) 흐름독립 분석과 흐름의존 분석의 차이를 보이는 예

본 연구에서는 실험에 앞서 대상 프로그램들을 분석해 본 결과 문자열 변수에 저장된 파일명을 변경하거나 다른 목적으로 사용하는 사례가 없었다. 그러므로 흐름 의존 분석의 사용으로 인한 정확도 향상의 효과가 미미 할 것으로 예상되기 때문에 분석의 속도를 높이기 위하여 흐름독립 추론을 사용하였다.

4.4 분석기의 구현

구현된 분석기의 전체적인 구조는 (그림 9)와 같이 2단계로 구성된다. 1단계는 대상 프로그램에 대한 추상구문트리(AST)를 생성하는 단계로 컴파일러 생성기인 SableCC[13]를 이용하여 작성되었다. SableCC는 소스 프로그램을 파싱해서 추상구문트리와 트리 노드를 방문하는 기능인 tree-walker[6]를 생성한다. tree-walker는 디자인 패턴의 Visitor 모델을 확장한 것으로서, 추상구문트리 노드들을 깊이 우선 탐색과 역 깊이 우선 탐색으로 방문할 수 있도록 해준다. 2단계는 1단계에서 생성된 추상구문트리와 타입 한정자를 명세한 라이브러리를 대상으로 제약식을 생성하고 해를 찾는 단계이다. 제약식 생성은 1 단계에서 생성된 추상구문트리 노드를 tree-walker를 이용해서 방문하면서 <표 1>의 식에 해당하는 구문을 만나는 경우 제약식을 생성한다.



(그림 9) 분석기의 구조

4.5 심볼릭 링크 공격에 취약한 코드 정의의 변경

본 논문에서는 심볼릭 공격에 취약한 코드를 문자열 상수 형태의 파일 이름으로 정의하였다. 그러나 문자열 상수 형태의 파일 이름이 아니라도 동적인 모니터링 등으로 임시 파일 이름을 알 수 있는 경우가 있다. 그러므로 심볼릭 링크 공격의 취약한 코드의 정의를 랜덤하게 생성되거나 암호화된 형태를 제외한 문자열로 변경할 수 있다. 이와 같이 심볼릭 링크 공격의 취약한 코드의 정의를 변경하더라도 본 논문에서 제안한 시스템을 수정하지 않고 라이브러리 함수 선언의 일부만 변경함으로써 새로 정의된 취약점을 찾을 수 있다. 변경되는 라이브러리 함수의 선언은 다음과 같다: 외

부로부터 입력을 받는 라이브러리 함수의 입력 문자열은 untrusted로 선언, 랜덤 또는 암호화된 문자열을 생성하는 라이브러리 함수의 생성 문자열은 trusted로 선언, 문자열을 변경하는 라이브러리 함수인 경우 입력문자열이 trusted 이면 변경된 문자열은 trusted 로 선언.

5. 실험 및 고찰

제안된 방법의 유용성을 검증하기 위하여 실용적으로 널리 사용되는 응용 프로그램들을 대상으로 하여 제안한 방법을 실험하였다. <표 3>은 실험에서 사용된 응용 프로그램들의 간단한 설명이다.

〈표 3〉 대상 프로그램 설명

프로그램	버전	설 명	라인
bftpd	1.6	FTP 서버	2K
cfengine	2.0.6	시스템 관리 도구	24K
httpd	2.2.4	HTTP 서버	33K
mingetty	1.0.7	원격 터미널 제어 도구	0.2K
sshd	4.6p1	OpenSSH 데몬	26K
pexec	1.0	명령을 병렬로 실행하는 도구	4.3K
xmlstarlet	1.0.1	간단한 XML 편집, 변환 도구	5.4K

<표 4>는 <표 3>의 대상 프로그램들에 대해서 제안한 방법을 적용해서 분석한 결과이다.

〈표 4〉 대상프로그램의 취약점 분석 결과

프로그램	#fopen()	취약점 수	비율(%)
bftpd	15	3	20.0
cfengine	52	2	3.8
httpd	16	0	0.0
mingetty	1	1	100.0
sshd	21	0	0.0
pexec	5	1	20.0
xmlstarlet	1	0	0
전체	111	7	6.3

<표 4>의 #fopen()은 프로그램에서 사용된 fopen() 함수의 개수이며, 취약점 수는 사용된 fopen() 함수 중 심볼릭 링크 공격에 취약한 것으로 판명된 것이다. 비율은 전체 fopen() 함수 중 심볼릭 링크 공격에 취약한 fopen()함수의 비율을 의미한다. <표 4>에서 심볼릭 링크 공격에 취약한 코드는 전체 111개의 fopen() 함수 중에서 7개로, 약 6.3% 정도이다.

<표 4>의 실험 결과는 본 논문의 유용성을 보여준다. 본 논문에서 제안한 방법을 이용하지 않는 경우 105개의 fopen() 함수에 대해서 심볼릭 링크 공격이 가능한지를 프로그래머가 일일이 분석하여 방어하기 위한 코드를 삽입해야 하나, 제안하는 방법을 적용하는 경우 6개의 fopen() 함수에 대해서만 방어하기 위한 코드를 삽입하면 된다.

6. 결 론

본 논문에서는 심볼릭 링크 공격에 취약한 코드를 찾기 위하여 타입 한정자를 이용하는 방법을 제안하였다. 제안된 방법은 프로그래머의 추가적인 큰 노력 없이 심볼릭 링크 공격에 취약한 부분을 자동으로 찾아주므로, 프로그래머가 해당 취약성을 효과적으로 방어할 수 있도록 해준다.

실용적인 프로그램에 대상으로 본 논문의 방법을 실험해 본 결과 심볼릭 링크 공격 취약성의 일차적인 후보가 되는 fopen() 함수 호출 중 소수가 취약한 것으로 분석되었으며, 이는 개발된 도구를 사용하여 취약성 검사의 노력을 줄이고 효과적으로 취약성을 제거할 수 있음을 보여준다.

본 논문에서는 설명의 편의를 위해서 심볼릭 링크 공격에 취약한 코드를 상수 문자열로 정의하였으나, 동적 모니터링 등을 이용해서 대상 프로그램에서 사용하는 파일 이름을 알 수 있는 경우도 있다. 이 경우 새로운 취약점을 찾기 위해서 심볼릭 링크 공격에 취약한 코드를 새로 정의할 수 있으며, 심볼릭 링크 공격에 취약한 코드를 새로 정의하더라도 본 논문에서 제안한 타입 시스템을 수정하지 않고 타입 한정자 명세만 변경함으로써 새로운 취약점을 검출할 수 있음을 보였다.

참 고 문 헌

[1] Kyung-Goo Doh, Seung Cheol Shin, "Detection of Information Leak by Data Flow Analysis", ACM SIGPLAN Notices, Volume 37, Issue 8, pages 66-71, 2002.

[2] Jeffrey S. Foster, Manuel Fähndrich, Alexander Aiken, "A Theory of Type Qualifiers", ACM SIGPLAN Notices, Conference on Programming language design and implementation PLDI '99, Volume 34, Issue 5, pages 192-203, 1999.

[3] Jeffrey S. Foster, Tachio Terauchi, Alex Aiken, "Flow-sensitive Type Qualifiers", ACM SIGPLAN Notices, Conference on Programming language design and implementation PLDI '02, Volume 37, Issue 5, pages 1-12, 2002.

[4] Jeffrey S. Foster, Robert Johnson, John Kodumal, Alex Aiken, "Flow-insenstive Type Qualifiers", ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 28, Issue 6, pages 1035-1087, 2006.

[5] Jeffrey S. Foster, "Type Qualifiers: Lightweight Specifications to Improve Software Quality", Ph.D. thesis.

University of California, Berkeley, 2002.

[6] Etienne Gagnon, "SABLECC, AN OBJECT-ORIENTED COMPILER FRAMEWORK", School of Computer Science, McGill University, Montreal, pages 58-60, 1998

[7] Flemming Nielson, Hanne Riis Nielson, Chris Hankin, "Principles of Program Analysis", Springer, pages 174-175, 1998

[8] Jakob Rehof and Torben Æ. Mogensen. Tractable Constraints in Finite Semilattices. In Hadhia Cousot and David A. Schmidt, editors, *Static Analysis, Third International Symposium*, volume 1145 of *Lecture Notes in Computer Science*, Pages 285-300, Aachen, Germany, September 1996. Springer-Verlag.

[9] Andrei Sabelfeld, Andrew C. Myers, "Language-Based Information-Flow Security". IEEE Journal on selected areas in communications, Vol. 21, No.1, January 2003.

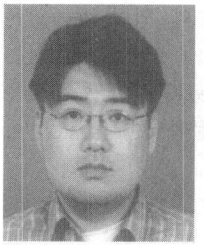
[10] Robert C. Seacord, "Secure Coding in C and C++(한국어판)", Addison-Wesley, pages 277-305, 2006.

[11] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner, "Detecting Format-String Vulnerabilities with Type Qualifiers", 10th USENIX Security Symposium, pages 201-218, 2001.

[12] John Viega, Gary McGraw, "Building Secure Software", pages 209-265, 2001.

[13] SableCC homepage, http://sablecc.org/

[14] 양대일, "정보 보안 개론과 실습", 한빛 미디어, pages 227-234, 2004.



주 성 용

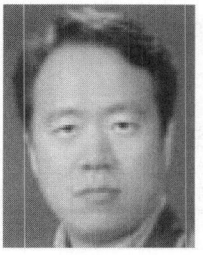
e-mail : jheaven1@donga.ac.kr

2000년 동아대학교 컴퓨터공학과(공학사)

2003년 동아대학교 대학원 컴퓨터공학과 (공학석사)

2003년~현 재 동아대학교 대학원 컴퓨터 공학과(박사과정)

관심분야 : 프로그래밍 언어, 컴파일러, 임베디드 시스템 등



안 준 선

e-mail : jsahn@kau.ac.kr

1992년 서울대학교 계산통계학과(이학사)

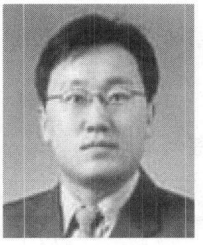
1994년 KAIST 전산학과(공학석사)

2000년 KAIST 전자전산학과(공학박사 (전산학)

2000년~2001년 KAIST 프로그램분석 시스템연구단 연구원

2001년~현 재 한국항공대학교 항공전자 및 정보통신공학부 교수

관심분야 : 프로그래밍언어, 프로그램 분석, 웹보안, 프로그램보안, 유비쿼터스컴퓨팅, 정보검색 등



조 장 우

e-mail : jwjo@dau.ac.kr

1992년 서울대학교 계산통계학과(학사)

1994년 서울대학교 전산과학과(석사)

2003년 한국과학기술원 전산학과(박사)

현 재 동아대학교 컴퓨터공학과 부교수

관심분야 : 프로그램 분석, 임베디드 시스템