情 報 保 護 學 會 誌
第 16 卷 第 5 號, 2006. 10

버퍼 오버플로우 검출을 위한 정적 분석 도구의 현황과 전망

김 유 일*, 한 환 수**

요 약

버퍼 오버플로우는 C 언어로 작성한 소프트웨어에 보안 취약점을 남기는 대표적인 원인이다. 프로그램 정적 분석 기법을 통해 버퍼 오버플로우 취약점을 검출하는 방법은 버퍼 오버플로우 취약점을 활용한 다양한 보안 공격에 대한 근본적인 해결책이 될 수 있다. 본 고에서는 버퍼 오버플로우 취약점을 검출할 수 있는 몇 가지 정적 분석 도구들의 특징과 성능을 살펴보고, 보다 효율적이고 정확한 버퍼 오버플로우 정적 분석 도구를 개발하기 위한 우리의 연구 성과를 소개한다.

Ⅰ. 서 론

버퍼 오버플로우(buffer overflow)는 프로그램이 배열이나 동적으로 할당한 메모리 영역의 경계를 넘어 데이터를 읽거나 쓰는 상황을 말한다. 버퍼의 경계를 지나 데이터를 기록하면 다른 변수의 값에 영향을 미치거나 프로그램의 제어 흐름을 바꿀 수 있는데, 이는 종종 소프트웨어의 심각한 보안 취약점으로 남는다. 특히 C 언어로 프로그램을 작성하는 경우에는 버퍼의 경계를 검사하는 코드를 작성하는 일이 프로그래머의 책임이므로 프로그래머의 실수로 버퍼 오버플로우 취약점이 나타나기 쉽다.

버퍼 오버플로우는 C 언어로 작성된 소프트웨어가 보안에 취약하게 만드는 대표적인 원인이고, 알려진 보안 취약점의 상당수가 버퍼 오버플로우와 관련되어 있다. 예를 들어, 1998년의 Morris 웜은 유닉스 fingerd 소프트웨어의 버퍼 오버플로우 취약점을 이용한 것이었고, 2001년의 Code Red 바이러스는 마이크로소프트 윈도우의 IIS 소프트웨어의 버퍼 오버플로우 취약점을 이용한 것이었다. D. Wagner 등의 연구를 인용하면 1988년부터 1998년까지 CERT Advisories를 통해 발표된 보안 취약점의 50% 정도가 버퍼 오버플로우와 관련된 것이었고, 최근으로 갈수록 그 비율이 증가하는 추세를 보이고 있다 [5].

프로그램 정적 분석은 프로그램 수행 중에 발생하는 일을 프로그램을 실행하지 않고 파악하기 위한 체계적인 방법이다. 정적 분석 도구가 동적 분석 도구에 비해 우수한 점을 세 가지로 요약할 수 있다. 첫째, 정적 분석 도구는 사용하기 쉽다. 동적 분석 도구를 이용하기 위해서는 대상 프로그램을 실행할 수 있는 환경을 만들고 적절한 입력 데이터를 제공해 주어야 하지만, 정적 분석 도구를 사용하는 경우에는 대상 프로그램의 소스 코드만 있으면 된다. 소스 코드의 일부만 주어지더라도 정적 분석은 가능하다. 둘째, 정적 분석 도구는 프로그램의 모든 실행 가능한 경로를 살펴볼 수 있다. 동적 분석 도구를 사용한 검사는 프로그램 테스트와 유사하게 존재하는 취약점을 발견하지 못할 가능성이 있다. 셋째, 정적 분석 도구는 대상 프로그램의 성능에 영향을 미치지 않는다. 동적 분석 기법은 실행 중에 오류 검사를 수행하도록 대상 프로그램을 변형하는 방법이므로 대상 프로그램의 성능을 현저하게 저하시킬 가능성이 있다.

이처럼 정적 분석 도구가 다수의 장점을 가지고 있으므로, 지금까지 정적 분석 기법을 통해 C 언어로 작성한 소프트웨어의 버퍼 오버플로우 취약점을 검출하기 위한 연구가 활발하게 수행되었다. 본 고에서는 지금까지 개발된 버퍼 오버플로우 정적 분석 도구들의 성능과 한계에 대해 살펴보고, 보다 효율적이고 정확한 정적 분석 도구를 개발하기 위한 연구 방향을 제안하려고 한다.

* KAIST 전자전산학과 전산학전공 (youil.kim@arcs.kaist.ac.kr)
** KAIST 전자전산학과 전산학전공 (hshan@cs.kaist.ac.kr)

본문의 전체적인 구성은 다음과 같다. 2 장에서는 버퍼 오버플로우 취약점과 이를 이용한 공격 방법에 대해 자세히 살펴본다. 3 장에서는 버퍼 오버플로우 취약점을 검출하는 정적 분석 도구를 개발하기 위한 기반 지식을 설명한다. 4 장에서는 지금까지 개발된 정적 분석 도구들을 살펴본다. 5 장에서는 더 나은 정적 분석 도구를 개발하기 위한 우리의 연구 성과와 향후 계획을 소개하고, 6 장에서 결론을 맺는다.

II. 버퍼 오버플로우 공격의 실제

이 장에서는 버퍼 오버플로우 취약점이 있는 예제 프로그램들을 사용하여 버퍼 오버플로우 취약점을 이용한 공격이 어떤 원리로 이루어지는지 살펴본다. 여기서 소개하는 공격 방법은 지금까지 개발된 공격 방법 중 일부에 지나지 않는다. 점차 발전하는 공격 방법을 차례로 소개하면서, 버퍼 오버플로우를 근본적으로 방지하지 않는 해결책은 공격을 어렵게 만들지라도 잠재적인 보안 취약점을 완전히 해결하는 것이 아니라는 사실을 강조하고 싶다.

[그림 1]의 예제 프로그램은 1996년 Aleph One이 스택 스매싱(Stack Smashing)이라는 공격 기법을 소개하면서 사용한 것이다[2]. argv[1]에 저장된 문자열의 길이가 배열 buf의 크기를 넘을 수 있으므로, 프로그램의 다섯째 줄에서 버퍼 오버플로우가 발생할 수 있다. 이 프로그램에서 버퍼 오버플로우가 특히 문제가 되는 이유는 기존의 시스템에서는 이러한 유형의 프로그램에 버퍼 오버플로우를 발생시켜 임의의 코드를 수행시킬 수 있었기 때문이다. 대상 프로그램이 관리자 권한으로 수행되는 경우라면, 악의적인 사용자가 관리자 권한으로 임의의 작업을 수행할 수 있다.

스택에 할당한 버퍼에 데이터를 기록하면서 버퍼 오버플로우를 발생시켜 스택에 저장되어 있는 함수의 리턴 주소(return address)를 변경하는 것이 스택 스매싱 공격의 핵심이다. 이러한 변경이 가능한 이유는 스택에 할당한 버퍼에는 주소가 증가하는 방향으로 값을 기록하는 반면, 스택 프레임은 주소가 감소하는 방향으로 쌓아나가기 때문이다. 버퍼의 경계를 넘어서 악의적인 코드로 스택을 덮어쓰고 현재 수행 중인 함수의 리턴 주소를 변경하면, 함수가 작업을 마치고 호출 지점으로 돌아가는 대신 삽입한 코드를 수행하도록 할 수 있다.

지금까지 스택 스매싱 공격을 방지하기 위한 다양한 해결 방안이 제안되었다[1]. 프로그램을 실행할 때마다 스택의 구조가 무작위로 결정되도록 하는 방법, 함수의 지역 변수와 리턴 주소 사이에 비밀 값을 삽입해 두었다가 이 값이 바뀌었는지의 여부로 버퍼 오버플로우 발생 여부를 판단하는 방법, 함수의 리턴 주소를 별도의 자료구조에 저장하는 방법, 스택에 저장한 코드는 실행할 수 없도록 하는 방법 등이 있다. 제안된 방법들은 스택 스매싱 공격을 어렵게 만들지만, 버퍼 오버플로우를 근본적으로 방지하는 방법은 아니다. 따라서 이러한 방법들을 무력화시키는 개선된 공격 방법들도 계속 발표되고 있다[4].

[그림 2]의 예제 프로그램은 함수의 리턴 주소를 변경하지 못하는 상황에서도 임의의 코드를 수행하는 버퍼 오버플로우 공격이 가능하다는 것을 보여주는 예이다. 배열 buf에 대해 버퍼 오버플로우를 발생시켜 함수 포인터 pf의 값을 바꿀 수 있다. 스택 스매싱 공격과 비슷하게 배열 buf에 수행하려는 코드를 저장하고 버퍼 오버플로우를 통해 함수 포인터 pf가 버퍼에 저장한 코드의 시작 주소를 가리키도록 하면, 이후에 함수 포인터 pf를 통한 함수 호출이 발생할 때 프로그래머가 의도한 함수가 호출되는 대신 버퍼에 삽입한 코드가 수행된다. 이러한 유형의 공격은 함수의 리턴 주소만을 보호하기 위해 제안된 기존의 많은 해결책들을 우회할 수 있다.

```
1: int main(int argc, char **argv)
2: {
3:     char buf[512];
4:     if (argc > 1)
5:         strcpy(buf, argv[1]);
6:     return 0;
7: }
```

[그림 1] 스택 스매싱 취약점이 있는 프로그램

```
1: int main(int argc, char **argv)
2: {
3:     char buf[512];
4:     void (*pf)() = f;
5:     strcpy(buf, argv[1]);
6:     pf();
7:     return 0;
8: }
```

[그림 2] 포인터 변조 취약점이 있는 프로그램

이러한 유형의 공격 방법은 버퍼 오버플로우 취약점을 활용하여 얼마나 다양한 공격이 가능한지를 잘 보여준다. 버퍼 오버플로우를 통해 변경되는 변수가 함수 포인터가 아닌 경우에도 보안 취약점이 발생할 수 있다. 만일 포인터 변수의 값과 포인터 변수에 저장될 값을 한꺼번에 변경할 수 있다면 임의의 메모리 주소에 4 바이트 데이터를 기록할 수 있는데, 이는 다양한 공격을 허용하는 보안 취약점이 된다.

〔그림 3〕의 예제 프로그램은 M. Kaempf가 힙 오버플로우를 활용한 언링크(unlink) 기술이라는 공격 기법을 소개하면서 사용한 예제이다[3]. 이 프로그램은 버퍼를 스택이 아닌 힙에 할당하고 있다. 힙에 할당한 버퍼는 버퍼 오버플로우가 발생하더라도 함수의 리턴 주소가 변경되거나 지역 변수의 값이 변경되지 않으므로, 지금까지 설명한 공격 방법으로부터는 안전하다고 말할 수 있다. 그러나 버퍼 오버플로우가 발생할 수 있다는 것은 여전히 잠재 위험으로 남는다.

실제로 〔그림 3〕의 프로그램을 GNU C 라이브러리 2.3.4 이전 버전이 설치된 리눅스 시스템에서 컴파일 하여 실행하는 경우에 버퍼 오버플로우를 활용한 공격이 가능하다. GNU C 라이브러리의 동적 메모리 관리 함수들은 Doug Lea의 동적 메모리 관리 함수 구현에 기반을 두고 있는데, Doug Lea의 초기 구현은 언링크 기술이라고 불리는 공격 기법에 취약하다는 것이 알려져 있다[3]. 라이브러리는 free 함수를 통해 해제된 메모리 영역들을 환형 이중 연결 리스트로 관리하는데, unlink는 환형 이중 연결 리스트에서 하나의 항목을 제거하는 매크로 함수의 이름이다.

언링크 기술의 원리를 간단히 설명하면 다음과 같다. 〔그림 3〕의 malloc 함수를 통해 버퍼를 할당하면 정확히 지정한 만큼의 공간이 할당되는 것이 아니고 버퍼의 경계에 malloc 관련 함수들이 사용하는 약간의 정보가 추가되는데, 그 중에는 버퍼가 해제되었는지의 여부를 나타내는 비트도 포함된다. 〔그림 3〕의 프로그램과 같이 연속적으로 두 개의 버퍼를 할당한 경우는 이 비트를 조작하여 여섯째 줄의 첫 번째 free 함수 호출에서 unlink 매크로가 수행되도록 할 수 있다. 버퍼가 해제되면 사용자 데이터를 저장하던 공간은 환형 이중 연결 리스트를 구성하는 포인터를 저장하는 공간으로 재활용되는데, 이곳에 저장되는 데이터를 조작하면 unlink 매크로가 임의의 메모리 주소에 4 바이트 데이터를 기록하도록 할 수 있다.

```
1: int main(int argc, char **argv)
2: {
3:     char *first = malloc(666);
4:     char *second = malloc(12);
5:     strcpy(first, argv[1]);
6:     free(first);
7:     free(second);
8:     return 0;
9: }
```

〔그림 3〕 힙 오버플로우 취약점이 있는 프로그램

임의의 주소에 4 바이트 데이터를 기록하는 것을 활용한 대표적인 공격 방법은 GOT(Global Offset Table)을 변경하는 것이다. 프로그램이 사용하는 공유 라이브러리 함수가 적재된 위치는 GOT에 기록되어 있고, 프로그램은 공유 라이브러리를 호출할 때 GOT 항목을 참조하여 공유 라이브러리 함수의 시작 주소로 분기한다. 〔그림 3〕의 프로그램에서 여섯째 줄의 free 함수를 수행할 때 free 함수의 시작 주소를 저장하고 있는 GOT 항목을 버퍼에 삽입한 코드의 시작 주소로 변경한다면, 일곱째 줄에서는 free 함수가 호출되는 대신 버퍼에 삽입한 코드가 수행된다.

이처럼 다양한 버퍼 오버플로우 공격에 효과적으로 대응하기 위해서는 버퍼 오버플로우 자체를 근본적으로 방지하는 해결책이 필요하다. 최근 버전의 GNU C 라이브러리는 소스 코드가 수정되어 더 이상 언링크 기술을 통한 공격은 가능하지 않지만, 향후 계속하여 지금까지 알려지지 않은 새로운 공격 방법이 발견되고 개발될 것이다. 따라서 프로그램에 버퍼 오버플로우 가능성이 남아있는 한 잠재적인 보안 위험이 완전히 사라진 것은 아니다.

Ⅲ. 버퍼 오버플로우 검출을 위한 정적 분석

정적 분석 기법은 모든 버퍼 오버플로우 취약점을 검출하는 효과적인 해결책이 될 수 있다. 이 장에서는 정적 분석에 대한 기반 지식을 설명하고, 버퍼 오버플로우 취약점을 검출하기 위한 정적 분석 도구를 설계할 때에 고려할 사항들을 살펴본다. 이 장에서 설명하는 내용들은 지금까지 개발된 정적 분석 도구들의 특성을 이해하고 더 나은 도구를 개발하기 위한 기반이 될 것이다.

우리의 목표는 모든 버퍼 오버플로우 가능성을 검출하는 안전한 정적 분석 도구를 개발하는 것이다. 이 장의 내용은 이러한 목표를 기준으로 하여 설명하는 것

으로, 이 장의 예제 프로그램들은 3 장의 예제 프로그램들에 비해 다소 복잡하고 의도적인 면이 있다.

1. 정수 변수 분석

모든 버퍼 오버플로우 검출을 위한 정적 분석은 정수 변수의 값이나 버퍼의 크기, 문자열의 길이 등을 분석할 수 있어야 한다. 프로그램 실행 중에 각 정수 변수에 저장되는 모든 값을 계산하는 것은 거의 불가능하고 또한 불필요한 작업이다. 정적 분석을 설계할 때에는 정수 변수가 실행 중에 가지는 값에 대한 정보를 효과적으로 표현하기 위한 방법을 선택해야 한다.

정수 변수의 값을 분석하기 위한 가장 간단한 방법으로 구간(Interval) 도메인 [16] 을 사용한 분석을 들 수 있다. 구간 도메인을 사용하여 분석하면 정수 변수가 실행 중에 가지는 값의 최소값과 최대값을 알 수 있다. 버퍼 오버플로우를 검출하는 데에는 구간 도메인이 효과적으로 사용될 수 있는데, 대체적으로 최소값과 최대값이 버퍼 오버플로우 여부를 판단하는 데 중요한 역할을 하기 때문이다.

[그림 4]의 예제 프로그램은 구간 도메인을 사용한 분석으로 충분한 정보를 얻지 못하는 예이다. 구간 도메인을 사용한 분석은 분석 속도를 향상시키기 위한 축지법(widening) 및 좁히기(narrowing) 기법과 함께 사용된다. [그림 4]의 프로그램을 이러한 방법으로 분석하면 반복문 안에서 변수 i의 최대값이 9라는 정보를 얻을 수 있지만 변수 j의 최대값은 분석하지 못한다.

지금까지 제안된 방법 중에서 가장 정확도가 높은 값 분석 방법은 다각형(Polyhedra) 도메인 [17] 을 이용한 분석 방법이다. 다각형 도메인을 이용한 분석은 정수 변수들 사이의 모든 선형 관계식을 이끌어 낸다. 예를 들어 [그림 4]의 프로그램을 다각형 도메인을 이용하여 분석하면 반복문 안에서 변수 i와 변수 j가 항상 동일한 값을 가진다는 정보를 함께 얻게 된다. 이 정보와 변수 i의 최대값이 9라는 정보를 종합하면 변수 j의 최대값도 9라는 것을 알 수 있다. 그러나 다각형 도메인은 복잡도가 지나치게 높아 큰 프로그램에 적용하기 어렵다는 단점이 있다. A. Mine는 구간 도메인과 다각형 도메인 사이의 정확도 및 복잡도를 가지는 분석 방법들을 제안했는데 [18], [그림 4]의 프로그램은 A. Mine가 제안한 분석 방법으로도 충분한 분석 결과를 얻을 수 있다.

```
1: void figure4(void)
2: {
3:     char buf[10];
4:     int i, j = 0;
5:     for (i = 0; i < 10; i++) {
6:         buf[j] = 0;
7:         j++;
8:     }
9: }
```

[그림 4] 변수 사이의 관계가 중요한 예

2. 포인터 분석

C 언어로 작성한 프로그램을 정확하게 분석하기 위해서는 포인터 연산을 정확하게 다루는 것이 필수적이다. C 언어로 작성한 프로그램은 포인터 변수를 정수 변수나 함수의 별명(alias)으로 사용하기도 하고, 포인터를 통해 버퍼에 접근하기도 한다.

포인터 분석은 각 포인터 변수가 가리키는 대상의 집합을 구하는 문제인데, 정확하게 분석하려면 계산의 시간 복잡도가 $O(n^3)$에 달한다. B. Steensgaard는 기존의 포인터 분석 방법으로 소스 코드가 10만 줄 이상인 대규모 소프트웨어를 한꺼번에 분석하기는 어렵다는 것을 지적하면서, 정확도가 다소 떨어지지만 거의 선형 시간 복잡도를 보이는 포인터 분석 방법을 제안하였다 [15].

버퍼 오버플로우 검출을 위해서는 포인터가 버퍼를 가리키는 경우 버퍼의 크기와 오프셋까지 분석해야 한다. [그림 5]의 프로그램은 정확한 포인터 분석이 필요한 대상 프로그램의 예이다. 포인터 변수 p가 크기가 10인 배열 buf의 다섯째 항목을 가리킨다는 것과 포인터 변수 q가 정수 변수 i를 가리킨다는 것을 파악해야 이 프로그램이 버퍼 오버플로우를 발생시키는 원인을 올바르게 분석할 수 있다.

```
1: void figure5(void)
2: {
3:     char buf[10], *p;
4:     int i, *q = &i;
5:     p = &buf[5];
6:     i = 5;
7:     p[*q] = 0;
8: }
```

[그림 5] 포인터 분석이 중요한 예

3. 제어 흐름을 고려하는 분석

제어 흐름을 고려하는(flow-sensitive) 분석 기법은 프로그램의 제어 흐름이 도달하는 각 지점에 대해 분석 정보를 구분하여 유지한다. 예를 들어, [그림 6]의 프로그램을 제어 흐름을 고려하는 분석 기법으로 분석하면, 변수 i가 여섯째 줄에서는 0과 9 사이의 값을 가지고 일곱째 줄에서는 10이 된다는 비교적 자세한 분석 결과를 얻을 수 있다.

제어 흐름을 무시하는(flow-insensitive) 분석 기법은 프로그램에 대한 전체적인 사실을 빠르게 파악하기 위해서 사용된다. [그림 6]의 프로그램을 제어 흐름을 분석 기법으로 분석하면, 프로그램 수행 중에 변수 i가 0과 10 사이의 값을 가진다는 덜 자세한 분석 결과를 얻는다.

결과적으로 [그림 6]의 프로그램에 대해 제어 흐름을 고려하는 분석 기법을 사용하면 일곱째 줄에서 버퍼 오버플로우가 발생한다는 것을 정확하게 알 수 있지만, 제어 흐름을 무시하는 분석 기법을 사용하면 여섯째 줄에 대해서도 버퍼 오버플로우가 발생한다는 잘못된 오류 메시지를 출력하게 된다.

제어 흐름을 무시하는 분석은 이처럼 부정확한 분석 결과를 제공하지만, 반면 효율적으로 수행할 수 있는 여지가 많다. 실제로 B. Steensgaard가 제안한 포인터 분석 기법은 제어 흐름을 무시하는 분석 기법에 속한다. B. Steensgaard의 포인터 분석이 유용한 이유는 대부분의 포인터 변수가 단순한 형태로 사용되는 프로그램에서는 제어 흐름을 무시하는 분석 기법을 사용해도 정확도를 크게 잃지 않기 때문이다. 예를 들어, 프로그램에서 변수가 단 한 번 나타나거나 값이 변하지 않는다면, 제어 흐름을 무시하는 분석 기법을 사용해도 정확도를 잃지 않는다.

```
1: void figure6(void)
2: {
3:     char buf[10];
4:     int i;
5:     for (i = 0; i < 10; i++)
6:         buf[i] = '0' + i;
7:     buf[i] = '\0';
8: }
```

[그림 6] 제어 흐름을 고려한 분석이 중요한 예

4. 함수 호출 문맥을 고려하는 분석

함수 호출 문맥을 고려하는(context-sensitive) 분석 기법이란 하나의 함수가 서로 다른 인자를 이용해 여러 번 호출되는 경우에 각 호출 문맥을 구분할 수 있는 분석 기법을 말한다. [그림 7]의 프로그램은 여덟째 줄의 함수 호출은 버퍼 오버플로우를 발생시키지 않고, 아홉째 줄의 함수 호출은 버퍼 오버플로우를 발생시키는 프로그램이다. 함수 호출 문맥을 고려하는 분석 기법을 이용하여 이 프로그램을 분석하면 아홉째 줄의 함수 호출이 버퍼 오버플로우를 발생시킨다는 것을 정확히 알 수 있다.

함수 호출 문맥을 무시하는(context-insensitive) 분석 기법을 사용하면 여덟째 줄과 아홉째 줄의 함수 호출을 한꺼번에 분석하므로 여덟째 줄의 함수 호출도 버퍼 오버플로우를 발생시킨다는 잘못된 경고 메시지를 출력하게 된다.

함수 호출 문맥을 고려하는 분석 기법도 여러 가지 방법이 있다. 일반적으로는 함수 호출 경로가 서로 다른 함수 호출을 구분하는 방법과 인자가 서로 다른 함수 호출을 구분하는 방법으로 분류할 수 있다. 실제적으로는 프로그램 소스 코드 상에서 구분되는 함수 호출들 간에만 구분하는 방법이 간단하기 때문에 많이 사용된다. 이 방법은 함수 호출 경로로 함수 호출을 구분하는 방법에 속한다.

5. 정적 분석의 안전성(Soundness)

안전한 정적 분석 기법을 사용해야만 모든 가능한 오류를 발견한다는 정적 분석의 최대 장점을 살릴 수

```
1:  void figure7(char *src)
2:  {
3:      char buf[5];
4:      strcpy(buf, src);
5:  }
6:  int main()
7:  {
8:      figure7("abc");
9:      figure7("abcde");
10:     return 0;
11: }
```

[그림 7] 함수 호출 문맥을 고려한 분석이 중요한 예

있다. 정적 분석이 안전하다는 것은 정적 분석을 통해 대상 프로그램에 버퍼 오버플로우 취약점이 존재하지 않는다는 결론을 얻었다면, 실제로 대상 프로그램을 어떠한 방법으로 수행하더라도 버퍼 오버플로우가 발생하지 않는다는 것이 보장된다는 의미이다. 정적 분석을 통해 프로그램 실행 중에 발생하는 일을 완전하게 파악하는 것이 불가능하다는 것은 수학적으로 증명된 사실이다. 정적 분석 도구는 대상 프로그램의 실행 중에 발생하는 일을 대략적으로 파악할 수밖에 없으며, 그 결과로 오류를 발견하지 못하거나 오류가 아닌 부분을 오류라고 판단하는 경우가 발생한다. 안전한 정적 분석 도구는 오류를 놓치지 않도록 설계된 분석 기법을 사용하므로 오류가 아닌데도 오류라고 판단하는 경우만 발생하는데, 본 고에서는 이런 경우를 잘못된 경고 메시지(false alarm)라고 부른다.

실제적으로는 안전하지 않은 정적 분석 기법이 많이 사용되는 두 가지 이유가 있다. 하나는 안전한 정적 분석 기법이 다수의 잘못된 경고 메시지를 출력한다는 것이고, 다른 하나는 안전한 정적 분석 기법을 충분히 효율적으로 수행하기 위한 연구 성과가 부족하다는 것이다.

6. 종합: 요약 해석(Abstract Interpretation)

다양한 정적 분석 기법들을 살펴보고 버퍼 오버플로우 검출이라는 목적에 효과적일 것으로 판단되는 기법들을 선택하여 정적 분석을 설계한다. 안전한 정적 분석을 체계적으로 설계하는 좋은 방법 중 하나는 요약 해석 프레임워크[16]에 맞추어 설계하는 것이다. 요약 해석 프레임워크의 요구 조건에 맞추어 설계한 정적 분석 기법은 정적 분석의 올바름이 자동으로 증명된다는 장점이 있다. 다음 장에서 소개할 PolySpace C Verifier와 Airac은 모두 요약 해석을 기반으로 한 안전한 정적 분석 도구에 속한다.

다만 요약 해석에 기반을 두고 개발된 정적 분석 도구는 아직 충분한 성능을 보여주지 못하고 있다. 그러나 요약 해석에 기반을 둔 정적 분석 도구를 실제로 구현하면서 얻은 연구 결과가 국외 학회에 종종 발표되고 있는 것으로 미루어 보면, 최근에는 요약 해석에 기반을 둔 정적 분석 도구를 효율적으로 구현하기 위한 연구가 활발히 진행되고 있는 것으로 판단된다.

Ⅳ. 정적 분석 도구 현황

인자의 길이를 검사하지 않는 strcpy 등의 함수를 사용하는 것이 버퍼 오버플로우 취약점의 원인이 된다는 점에 착안하여, 구문 분석을 통해 보안에 취약한 함수의 사용 여부를 검사하는 도구들이 있다. 이러한 유형에 속하는 분석 도구로는 ITS4, FlawFinder, RATS 등이 있다. 이러한 분석 도구들도 초보적인 단계의 정적 분석 도구라고 말할 수 있지만, 프로그램의 의미를 고려하지 않으므로, 본 논문에서 다루려는 정적 분석 도구와는 거리가 있다.

M. Zitser 등은 이미 알려진 버퍼 오버플로우 보안 취약점들을 단순화한 테스트 케이스를 설계하고, 이를 이용해 BOON, Splint, ARCHER, PolySpace C Verifier 등의 정적 분석 도구들을 비교 평가하는 연구를 수행했다[8]. 본 고에서는 이들 정적 분석 도구들의 단순한 성능 비교를 넘어 각각의 정적 분석 도구가 선택한 분석 기법을 자세히 살펴보고, 각 분석 기법이 가지는 장점과 더불어 근본적인 한계점에 대해 생각해 보려고 한다.

1. BOON [5]

BOON은 체계적인 프로그램 정적 분석 기법을 활용하여 버퍼 오버플로우 취약점을 분석하려고 시도한 최초의 정적 분석 도구라고 할 수 있다. 프로그램 정적 분석 기법을 사용함으로써 단순히 strcpy 함수가 사용된 것을 문제 삼지 않고, 실제로 대상 버퍼의 크기가 복사하려는 문자열의 길이보다 작은 경우만을 지적할 수 있다. BOON은 라이브러리 함수에 의해 발생하는 버퍼 오버플로우 취약점을 보다 정확하게 검출하려는 시도인 것으로 생각된다. 구간 도메인을 사용하여 버퍼의 크기나 문자열의 길이 등 정수 값을 분석하고, 포인터의 사용은 무시한다. 제어 흐름 및 함수 호출 문맥을 무시하는 분석 기법을 이용하여 분석의 효율을 높이려고 시도했다. BOON은 FlawFinder 류의 분석 도구에 비해 우수하지만, 정적 분석 도구로서는 초보적인 수준이다. 특히 포인터를 분석하지 않는다는 점에서 존재하는 취약점을 놓칠 가능성이 높다.

논문의 실험 결과에 따르면 펜티엄3 PC를 사용하여 소스 코드가 3만 줄 정도인 Sendmail 8.9.3 버전을 15분 안에 분석할 수 있었다. 분석 결과로 44개의 경고 메시지가 출력되었는데, 경고 메시지를 분석하여 새로운 버퍼 오버플로우 오류를 발견할 수 있었다고 발표했다.

2. Splint [6]

Splint는 가벼운(light-weight) 정적 분석이라고

부르는 정적 분석 기법을 사용하여 버퍼 오버플로우 취약점을 검출하려고 시도한 것이다. 프로그램의 의미를 이해하고 제약 식을 이끌어내기도 하지만, 기본적으로는 사용자가 특별한 형식의 주석(annotation)을 작성하여 많은 정보를 제공해 줄 것을 전제로 하여 개발된 분석 도구이다. Splint와 같이 특별한 주석을 요구하는 정적 분석 도구들은 정적 분석에 지식이 없는 사용자가 사용하기는 어렵다는 단점이 있다. 물론 주석을 작성하지 않아도 기본적인 분석을 수행할 수 있지만, 다수의 잘못된 경고 메시지가 출력되므로 분석 결과가 유용하다고 말하기 어렵다. Splint는 안전하지 않은 분석 기법을 사용한다. 충분한 주석을 작성하여 대상 프로그램에 버퍼 오버플로우 취약점이 없다는 분석 결과를 얻었다 하더라도 실행 중에 버퍼 오버플로우가 발생하지 않는다는 것을 보장하지 못한다.

Splint를 사용하여 [그림 7]의 프로그램을 분석하면 넷째 줄에서 버퍼 오버플로우가 발생할 수 있다는 경고 메시지를 출력한다. Splint는 내부적으로 strcpy 함수를 안전하게 호출하기 위해 만족시켜야 하는 제약 식을 알고 있는데, 버퍼 buf의 크기가 문자열 src의 길이보다 커야 한다는 것이다. 이 제약식과 첫째 줄에서 얻은 버퍼 buf의 크기가 5라는 정보를 결합하여, src의 길이가 4 이하여야 한다는 제약 식을 이끌어낸다. Splint는 주어진 프로그램에서 문자열 src의 길이가 4 이하라는 정보를 이끌어낼 수 없기에 경고 메시지를 출력하는 것이다. Splint가 출력하는 경고 메시지는 아홉째 줄의 함수 호출과는 아무런 관련이 없다. Splint는 각 함수를 별도로 분석하며, 함수 호출을 분석할 때에는 사용자가 작성한 주석에만 의존하기 때문이다. strcpy와 같은 라이브러리 함수를 처리하는 것은 제작자가 미리 작성해 둔 주석에 의한 것이다. 실제로 아홉째 줄의 함수 호출이 없어도 동일한 경고 메시지가 출력되는데, 함수의 인자가 길이 4 이하의 문자열이어야 한다는 주석을 작성하면 보다 정확한 분석을 수행할 수 있다.

3. ARCHER [7]

ARCHER는 Splint와 비슷하게 복잡도가 높지 않은 분석 기법을 사용하여 유용한 정적 분석 도구를 개발하려는 시도로 볼 수 있다. 사용자가 주석을 작성할 필요가 없다는 점과 함수 호출을 자동으로 분석한다는 점에서 Splint와 차별화된다. ARCHER는 비교적 단순한 버퍼 오버플로우 취약점을 가능한 빠르고 정확하게 찾아내는 것에 중점을 두어 개발된 도구라고 판단된다. M. Zitser 등의 실험에서 ARCHER는 BOON, Splint, PolySpace C Verifier와 비교할 때 오류를 발견하지 못하는 경우가 가장 많은 것으로 나타났다 [8].

ARCHER를 통해 [그림 7]의 프로그램을 분석하면 아홉째 줄의 함수 호출이 문제가 된다는 것을 정확하게 지적할 수 있다. ARCHER의 분석 방식은 함수 figure7을 분석하여 인자 문자열 src의 길이가 5 이상인 경우 버퍼 오버플로우가 발생한다는 조건을 이끌어내고, 함수 figure7을 호출할 때마다 버퍼 오버플로우 발생 조건을 만족하는지 확인하는 방식이다.

ARCHER는 분석 속도와 잘못된 경고 메시지의 비율 면에서 매우 우수한 성능을 보여주고 있다. 소스 코드의 크기가 160만 줄에 달하는 리눅스 2.5.53 버전을 Xeon 2.8 GHz CPU와 512 MB RAM을 탑재한 시스템에서 4시간 정도에 분석하여 118개의 실제 오류를 찾았는데, 잘못된 경고 메시지는 39개에 불과했다는 실험 결과가 발표되었다.

4. Coverity [11]

미국의 Coverity 사는 프랑스의 PolySpace Technologies 사와 더불어 정적 분석 기술을 상용화시킨 대표적인 기업이다. Coverity 사의 정적 분석 도구는 Stanford 대학의 D. Engler 등이 개발한 메타컴파일(metacompilation) 기법과 xgcc [9]에 기반을 둔 것으로 알려져 있다. 위에서 설명한 정적 분석 도구들과 비슷하게 안전성을 희생하고 복잡도가 높지 않은 분석 기법을 사용하여 개발된 정적 분석 도구인데, 상용화된 제품이니만큼 유사한 정적 분석 도구들 중에서는 가장 성능이 우수한 것으로 보인다.

Coverity 사에서 공개한 보고서에 따르면 소스 코드의 크기가 550만 줄에 달하는 리눅스 2.6.5 버전을 분석할 수 있었다. 버퍼 오버플로우와 관련된 경고 메시지는 총 124개였는데, 이들을 분석하여 15개의 버퍼 오버플로우 관련 오류를 발견했다고 발표했다.

5. PolySpace C Verifier [12]

안전한 정적 분석 도구로는 프랑스 PolySpace Technologies 사의 PolySpace C Verifier가 대표적이다. PolySpace C Verifier는 요약 해석에 기반을 둔 정적 분석 기법을 사용하여 개발한 정적 분석 도

구로 알려져 있다.

요약 해석에 기반을 둔 안전한 정적 분석 도구들은 아직 분석 속도 면에서 만족할 만한 성능을 보여주지 못하고 있다. M. Zitser 등은 PolySpace C Verifier를 사용하여 소스 코드가 15만 줄 정도인 Sendmail 8.12.4 버전을 분석하려고 시도했는데, 4일 동안 분석을 수행해도 결과를 얻을 수 없었다고 발표했다[8]. 속도 문제를 제외하면 M. Zitser 등의 실험에서도 PolySpace C Verifier가 가장 높은 정확도를 보여 주는 정적 분석 도구로 판명되었다. 요약 해석에 기반을 둔 안전한 정적 분석 도구가 실제로 유용하게 사용되기 위해서 분석 속도는 필수적으로 개선되어야 할 주요 문제점이다.

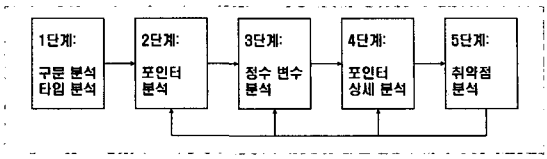
6. Airac5 [13]

서울대학교의 정영범 등은 요약 해석에 기반을 둔 정적 분석 도구인 Airac을 개발했다[10]. 최근에는 Airac을 더욱 개선한 버전인 Airac5를 개발했고, 실험결과를 홈페이지를 통해 공개하고 있다. Airac은 버퍼 오버플로우 취약점 검출에 특화된 정적 분석 도구라는 점에서 PolySpace C Verifier와 차별화된다. 통계적인 방법으로 잘못된 경고 메시지를 걸러내는 기법을 사용한다는 것도 특징적이다.

홈페이지를 통해 공개되어 있는 실험 결과에 따르면 3.2 GHz CPU와 4 GB RAM을 탑재한 펜티엄4 PC를 사용하여 소스 코드의 크기가 3만 라인 정도인 GNU bison 1.875 버전을 6 시간 이내에 분석할 수 있다. 다섯 개의 GNU S/W 실험 결과를 종합하면 Airac5는 소스 코드 374줄 당 1개의 비율로 잘못된 경고 메시지를 출력하고 있는데, 안전한 정적 분석 도구로서는 비교적 높은 정확도를 보여주고 있다고 판단된다. M. Zitser 등은 PolySpace C Verifier가 12줄 당 1개의 잘못된 경고 메시지를 출력할 것으로 추정하였고, Splint가 46줄 당 1개의 잘못된 경고 메시지를 출력할 것으로 추정하였다[8].

V. 우리의 연구 성과

우리는 모든 버퍼 오버플로우 취약점을 검출할 수 있는 정확하고 효율적인 정적 분석 도구를 개발하려는 목표를 가지고 있다. 4 장에서 살펴보았듯이 기존의 안전한 정적 분석 도구들은 아직 만족할 만한 성능을 보여주지 못하고 있다. 이 장에서는 우리가 개발하고 있는 버퍼 오버플로우 정적 분석 도구인 Raccoon을 소개하려고 한다.



〔그림 8〕 Raccoon의 구조

〔그림 8〕과 같은 구조로 정적 분석 도구를 구현함으로써 기존의 한계를 어느 정도 극복할 수 있을 것으로 기대하고 있다. 버퍼 오버플로우 검출을 위해 분석해야 하는 정보들을 몇 가지로 분류하고, 〔그림 8〕과 같이 여러 종류의 분석을 완전히 독립된 단계로 수행하는 구조의 정적 분석 도구를 설계했다. 정적 분석 도구를 이와 같은 구조로 설계한 이유는 다음과 같은 세 가지 이점을 기대하기 때문이다. 첫째, 각 단계에서 서로 다른 정확도 및 속도를 가진 분석 기법을 사용할 수 있는 유연성이 있다. 둘째, 보다 정확한 결과를 위해 추가적인 정보를 분석할 필요가 있다면 새로운 단계를 추가하는 것으로 목적을 이룰 수 있다. 셋째, 분석 과정에서 메모리의 최대 사용량(memory peak)을 줄일 수 있다. 대규모 소프트웨어를 분석하는 경우에는 메모리 최대 사용량이 분석 도구의 성능을 좌우하는 중요한 기준이 될 수 있다.

각 단계에서 수행되는 작업을 간단히 설명하면 다음과 같다. 1 단계는 구문 분석 및 타입 분석 단계이다. 대상 프로그램을 구문 분석하고 각 함수별로 제어 흐름 그래프를 생성한다. 2 단계는 Steensgaard 스타일의 빠른 포인터 분석을 이용하여 함수 포인터와 정수 변수를 가리키는 포인터를 분석한다. 이러한 유형의 포인터 변수들은 프로그램 전체에서 단순하게 사용되는 경향이 있으므로, 정확도를 다소 희생한 분석 기법을 사용해도 전체적으로는 정확도를 크게 잃지 않으리라고 기대했다. 3 단계는 구간 도메인을 사용한 요약 해석으로 정수 변수들의 값을 분석한다. 3 단계는 제어 흐름을 고려하고, 함수 호출 문맥을 고려하는 비교적 정확한 분석 기법을 사용한다. 4 단계는 독자적으로 설계한 도메인을 사용한 요약 해석으로 버퍼를 가리키는 포인터들의 상세 정보를 분석한다. 대상 버퍼의 크기 및 포인터의 오프셋 정보를 분석하는 것이다. 4 단계 분석은 부분적으로 3 단계 분석 결과를 이용하여 진행된다. 5 단계 취약점 분석 단계는 4 단계까지의 분석 결과를 이용하여 프로그램의 각 부분의 버퍼 오버플로우 가능성을 진단하

〔표 1〕 GNU 소프트웨어를 사용한 실험 결과

소프트웨어	크기	포인터 분석	요약 해석	경고 메시지
tar 1.13	14879	0.39s	105.25s	99
bison 1.875	29903	1.51s	201.32s	109
sed 4.08	7464	0.21s	8.75s	5
gzip 1.2.4a	11298	0.23s	47.51s	323
grep 2.5.1	14879	0.37s	76.23s	122

고, 경고 메시지를 출력한다.

Raccoon은 CIL 프레임워크[11]를 활용하여 OCaml 언어로 구현하였다. CIL 프레임워크는 C 언어 프로그램을 단순한 중간 언어로 변환해 주며 구문 트리 수준에서 프로그램을 다루기 위한 다양한 함수를 제공하므로 C 언어 프로그램을 분석하는 도구를 개발하기에 적합한 환경이다.

〔표 1〕은 Raccoon을 이용해 몇 가지 GNU 소프트웨어들을 분석한 결과이다. 크기는 CIL을 이용해 대상 소프트웨어를 하나의 파일로 병합한 후에 측정한 소스 코드의 줄 수이다. 2 단계 분석에 소요되는 시간을 셋째 칸에 나타냈고, 3 단계와 4 단계 분석에 소요되는 시간을 더하여 넷째 칸에 나타냈다. 실험에는 Xeon 3.2 GHz CPU와 4.0 GB RAM을 탑재한 리눅스 시스템을 이용했다. 속도 면에서만 보면 Raccoon은 소스 코드의 크기가 3만 줄에 달하는 프로그램을 4 분 이내에 분석할 수 있었지만, 다수의 잘못된 경고 메시지를 출력하고 있으므로 정확도 면에서는 아직 만족할 만한 결과를 보여주지 못하고 있다. Raccoon이 정확한 분석을 수행하지 못하는 대표적인 원인 중 하나는 C 표준 라이브러리 함수들을 모델링하지 않았다는 것이다. 지금까지는 주로 분석의 속도 향상에 중점을 두어 개발을 진행했는데, 분석의 효율성을 유지하면서 정확도를 개선하기 위한 연구를 계속 수행할 계획이다.

실험 결과는 제안하는 구조를 통해 안전한 정적 분석 도구의 분석 속도를 크게 향상시킬 수 있을 것이라는 추측을 뒷받침하고 있다. 우리는 분석의 속도를 더욱 개선하는 것이 가능할 것으로 보고 있다. 2 단계 분석을 수행할 때는 포인터 연산만 수행하는 함수를 빠르게 건너뛰는 등 각 단계의 분석을 보다 특화시키는 방법과 프로그램의 모듈성을 활용하여 분석의 범위를 줄이는 방법으로 분석의 효율성을 더욱 개선할 것이다.

Raccoon이 잘못된 경고 메시지를 걸러내기 위해 취하는 방식은 분석의 정확도를 조절하여 분석을 반복하는 것이다. 〔그림 8〕의 설계에는 이러한 의도가 반영되어 있지만, 아직 구현이 완료되지는 않았다. 5 단계는 각 경고 메시지의 신뢰도를 평가하고, 잘못된 경고 메시지로 추측되는 경우에는 경고 메시지와 관련된 부분을 보다 정확한 분석 기법으로 분석하도록 각 분석 단계를 조절한다. 경고 메시지와 직접적으로 관련되지 않은 부분에 대해서는 이전의 분석 결과를 재사용하여 추가 분석에 소요되는 시간을 최소화할 수 있을 것이다.

Ⅵ. 결 론

본 고에서는 버퍼 오버플로우 취약점의 유형과 다양한 공격 방법에 대해 소개하고, 버퍼 오버플로우 취약점을 검출하기 위한 정적 분석 도구들의 현황을 살펴보았다. 다수의 정적 분석 도구들이 안전하지 않은 분석 기법을 사용하여 일부 버퍼 오버플로우 취약점을 빠르게 찾아내는 것에 중점을 두고 있으며, 안전한 정적 분석 도구들은 충분히 만족할 만한 성능을 보여주지 못하고 있다는 것을 알게 되었다.

우리는 안전한 정적 분석 기법을 사용하는 효율적인 버퍼 오버플로우 정적 분석기를 개발하기 위한 연구를 수행하고 있는데, 본 고에서 지금까지의 연구 성과를 일부 소개하였다. 본 고에서 제안하는 방법들을 통해 정확하면서도 효율적인 정적 분석 도구를 개발할 수 있을 것으로 기대하고 있다.

본 고에서 설명한 내용들은 소프트웨어 보안 분야의 연구자들에게 프로그램 정적 분석 분야의 연구 성과를 소개하는 기회가 될 것이고, 동시에 프로그램 정적 분석 분야의 연구자들에게 향후 연구 방향을 제시하는 참고 자료가 될 수 있을 것이다.

참 고 문 헌

〔1〕 R. Secord, *Secure Coding in C and C++*, Addison Wesley, 2006.

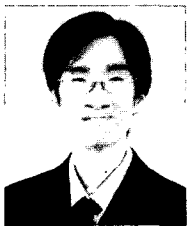
〔2〕 Aleph One, "Smashing the Stack for Fun and Profit", *Phrack*, 49, 1996.

〔3〕 M. Kaempf, "Vudo - An object superstitiously believed to embody magical powers", *Phrack 57*, 2001.

〔4〕 J. Pincus, B. Baker, "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns", *IEEE Security & Privacy*, 2(4), pp. 20-27, Jul/Aug 2004.

[5] D. Wagner, J. Foster, E. Brewer, A. Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities", NDSS'00.
[6] D. Larochelle, D. Evans, "Statically Detecting Likely Buffer Overflow Vulnerabilities", USENIX Security 2001.
[7] Y. Xie, A. Chou, D. Engler, "ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors", ESEC/FSE'03.
[8] M. Zitser, R. Lippmann, T. Leek, "Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code", SIGSOFT'04/FSE-12.
[9] S. Hallem, B. Chelf, Y. Xie, D. Engler, "A System and Language for Building System-Specific, Static Analyses", PLDI'02.
[10] Y. Jung, J. Kim, J. Shin, K. Yi, "Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis", SAS'05.
[11] http://www.coverity.com/
[12] http://www.polyspace.com/
[13] http://ropas.snu.ac.kr/2005/airac5/
[14] http://manju.cs.berkeley.edu/cil/
[15] B. Steensgaard, "Points-to Analysis in Almost Linear Time", PLDI'96.
[16] P. Cousot, R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints", POPL'77.
[17] P. Cousot, R. Cousot, "Automatic Discovery of Linear Restraints Among Variables of a Program", POPL'78.
[18] A. Mine, "Weakly Relational Abstract Domains: Theory and Applications", NSAD'05.

〈著者紹介〉



김 유 일 (Youil Kim)

2001년 : KAIST 전자전산학과 전산학전공 학사
2003년 : KAIST 전자전산학과 전산학전공 석사
2003년~현재 : KAIST 전자전산학과 전산학전공 박사과정
관심분야 : 프로그램 정적 분석

한 환 수 (Hwansoo Han)
정회원

1993년 : 서울대학교 컴퓨터공학과 학사
1995년 : 서울대학교 컴퓨터공학과 석사
2001년 : Ph.D., Computer Science, University of Maryland
2001년~2003년 : Sr. Engineer, Intel Architecture Group, Intel
2003년~현재 : KAIST 전자전산학과 전산학전공 조교수
관심분야 : 프로그램 정적 분석, 임베디드 시스템