버퍼오버플로우 취약성과 인프라 공격

충북대학교 조은선*

1. 서 론

인프라 상의 각종 소프트웨어들의 취약성을 이용하여 침투하는 사례[Krsul98]는 인터넷 활용이 급증하면서 증가하고 있으며, 이는 최근 들어 잇따라 발생되고 있는 대규모 침해 사고들과 분산 서비스 거부 공격 사건들에서도 잘 드러나고 있다[안02,안03]. 특히, 버퍼오버플로우 취약성은 이중 가장 많은 공격에 이용되고 있는 대표적인 것으로서, 보안 관련 게시판들에 자주 회자되고 있으며, 마이크로소프트사의 보안 패치의 대부분을 차지하고 있기도 하다 [Howard02].

버퍼오버플로우란 운영체제로부터 할당된 메모리 공간을 벗어난 부분에 자료를 덮어쓰는 것을 말한다. 만일 덮어쓰게 된 부분이 함수 리턴 주소나 함수 포인터 값 등의 프로그램의 제어 흐름을 다루는 값이 있는 위치라면, 수행중인 프로그램의 제어도 바꿀 수 있고 특히 어딘가에 심어져 있는 악성 코드를 수행하게 할 수가 있다. 버퍼오버플로우를 허용하는 소프트웨어에 대해서는 버퍼오버플로우 취약점을 가지고 있다고 한다. 일반적인 C 기반 소프트웨어들이 이러한 취약점에 노출되어 있다고 볼 수 있다.

버퍼오버플로우의 가능성은 비교적 오래전부터 지적되어 1989년 제작된 Morris 웜에서도 이미 사이버 공격의 침투, 전파 단계에서 사용된 바 있지만, 현재까지도 완벽한 대응책에 대한 해답이 나오지 않고 있다. 본 논문에서는 버퍼오버플로우에 대해 소개하고 그동안 제시되어 온 대응 기법들에 대해 정리한다.

2. 버퍼오버플로우 취약성에 대한 소개

가장 전형적인 버퍼오버플로우의 공격의 형태는

* 종신회원

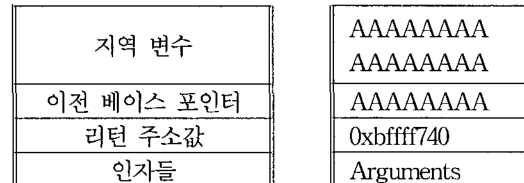

그림 1 버퍼오버플로우에 의한 스택의 변화

스택에 있는 버퍼에 오버플로우를 일으켜 동일 프레임 내에 저장된 리턴 주소값이나 기반 포인터 (base pointer) 값을 바꾸어 버리는 것이다.

Unix를 기준으로 한다면 수행중의 프로그램은 그림 1(a)와 같은 스택 구조를 가지게 되는데 이 때 지역 버퍼를 매우 긴 값(여기서는 반복되는 'A')으로 채우도록 한다면 그림 1(b)와 같은 모습으로 리턴 값이나 기반 포인터의 값이 비정상적으로 바뀌게 된다.

그림 2는 이를 유발하는 프로그램의 일부분을 나타낸다. 이 프로그램을 수행시킬때 인자 argv[1]으로 "AAAAAAAAAAAAAAAA.....AAAAAA"와 같은 8글자 이상의 데이터로 주게 되면 버퍼 buf에서 오버플로우가 일어난다. 만일 넘친 부분이 리턴시 복귀할 위치값 주소를 덮어쓰게 되면, 리턴 시에 원하지 않은 메모리 주소로 제어가 넘어가게 된다. 따라서 이 긴 인자 중 리턴 주소를 덮어 쓰게될 위치에 악성 코드의 메모리 주소(예: 0xbffff740)를 넣어두게 되면 정상적인 리턴 대신 해당 악성 코드 주소로 이동하여 수행된다. 이 때 악성 코드 자체도 주로 긴 인자의 일부에 위치시켜 주소값 파악이 용이하도록 한다. 공격자는 실제와 동일한 컴파일러로 컴파일된 취약한 코드에 대해, 공략할 대상과 동일한 환경에서 여러번 실험해 보는 과정을 거치게 된다.

```
int main(int argc, char ** argv[])
{
        int buf[8];
        strcpy(stackbuf, argv[1]);
        ...
}
```

그림 2 버퍼오버플로우 취약성을 가지는 코드

리턴 주소값을 변용하는 스택 버퍼오버플로우는 현재까지 가장 빈번하게 시도되고 있는 취약점 공격 방식이다. 이는 중요한 제어 관련 데이터인 리턴 주소값의 위치가 비교적 쉽게 알려질 수 있기 때문으로 볼 수 있다. 리턴 주소값 외에도 버퍼오버플로우 취약점을 이용하는 공격자가 제어에 영향을 미치기 위해 노리는 대상들은 다음과 같은 것들을 들 수 있다.

- **함수포인터 값[Wilander03]** : C에서는 함수에 대한 포인터를 사용하여 함수 자체를 값처럼 다루어 다른 함수의 인자로 넘겨주고 리턴 받을 수 있도록 하고 있다. 결과적으로 호출될 함수를 수행 중에 결정할 수 있도록 해준다. 이러한 함수 포인터 변수 값은 버퍼오버플로우로 인해 변용이 가능한데, 특히 인자로 전달될 때 스택의 다른 지역 변수에서 일어난 오버플로우로 인해 다른 값으로 덮어 씌여지는 일이 발생할 수 있다.
- **longjump() 버퍼[Wilander03]** : setjump()와 longjump()는 여러 함수 호출 후 한꺼번에 리턴하는 효과를 가지게 하는 장치이다. 예를 들어, 함수 f0()에서 setjump()를 하고 나서 f1()를 호출하였는데, f1() 내부에서 다시 f2()를 호출한 상황을 생각해 볼 때, 만일 f2()에서 longjump()를 불렀다면, f1()으로 돌아가지 않고 직접 setjump()를 한 f0() 위치로 복귀하게 된다. 즉 f2(), f1() 호출 스택을 일순간에 팝업 (pop-up) 해버리는 효과를 갖게 되는데, 이때 메모리에 f0()내의 setjump() 한 위치를 저장하게 된다. 이것을 스택 및 힙 버퍼 오버플로우에 의해 다른 값으로 바꿀 수 있다.
- **쉘(shell) 변수** : 프로그램 수행 중에 쉘 명령을 수행시키기 위해 준비된 명령 문자열은 C 프로그램 내부에서는 문자열 포인터에 의해 접근된다. 역시 메모리 주소값으로 준비되게 되므로 공격자는 악성 프로그램 구동을 내용으로 가지는 공간의 주소 값으로 변경시킬 수 있다.

오버플로우가 되는 버퍼의 위치는 스택과 힙 메모리 모두 가능하고, 변용 대상이 반드시 동일 종류의 메모리에 있을 필요는 없다. 즉, 위 리턴 주소값 변용 예에서는 스택 버퍼의 넘침으로 스택에 있는 리턴 주소값이 변경되었지만, 힙 메모리 상의 버퍼가 넘침으로 인해 스택의 리턴 주소값이 변경되는 것도 가능하다[Howard02].

3. 버퍼오버플로우 취약성에 대한 대응 기법

본 장에서는 코드의 개발 과정에 따라 필요로 하게 되는 버퍼오버플로우 완화 기법들에 대해 소개한다. 여기서 다루는 것은 C/C++에 관련한 버퍼오버플로우 대응 방안에 대한 것으로서 기타 언어나 다른 취약성들에 관한 대응책은 배제하였다.

3.1 소스 코드 스캐닝

소스 코드가 주어진 개발자들이 제작중인 소프트웨어에 버퍼오버플로우 취약성이 있는지에 대해 검증하는 것을 도와주는 도구들에 대해 적지 않은 연구가 이루어져 왔다. 이러한 도구들은 프로그램의 소스 코드를 적어도 한 번 이상 스캔함으로써 취약성을 점검하게 되는데, 그 분석의 면밀함에 따라 다양한 기법들이 있다.

미 Cigital사의 IT4S[Viega00]는 취약한 함수 호출에 관한 사전 자료를 가지고 비교를 통해서 취약한 함수 사용 여부를 파악하고 위험의 정도 (NO_RISK, LOW_RISK, MODERATE_RISK, VERY_RISKY, MOST_RISKY) 를 알려주는 방식을 사용하고 있다. 또한 버퍼오버플로우 외에도 경쟁 조건 (race condition)과 다른 취약성들에 대해서도 데이타베이스화하여 가지고 있다. 분석의 정확성은 다소 떨어지나 속도가 빠르고 코딩 환경에 포함가능한 형태로 지원되고 있어 실제 활용성이 매우 높다는 장점을 가진다. 동일한 방식의 코드 스캐너로는 D. Wheeler의 Flowfinder[Wheeler03a]와 Security Software사의 RATS(Rough Auditing Tool for Security)[Secure Soft]가 있다[1]. 이들 스캐너에서 다루고 있는 버퍼

1) Flawfinder와 RATS는 오픈 소스로서, 독립적으로 개발되어 완성되었지만, 향후 버전에는 두 스캐너

오버플로우에 취약한 함수들은 strcpy()를 포함하여 gets(), cuserid(), scanf() 계열 함수들, sprintf(), strcat(), streadd(), strecpy(), vsprintf(), strtrns() 등이 있다[Wheeler03a].

코드를 단순하게 스캔하는 렉시컬 분석 (lexical analysis)외에도 파싱 트리 (parsing tree)를 만들고 프로그램의 의미 구조적인(semantic) 내용을 정밀하게 분석하는 방법들이 있다. Univ. of Virginia의 Splint(Secure Programming Lint) [Larochelle01]는 개발자가 함수의 전/후 조건 ('@requires'/'@ensures')을 프로그램의 주석 형태로 기술하도록 하여 이것을 검증하고 있다. 예를 들어 strcpy(char *dst, char *src) 함수의 경우 버퍼가 입력 값보다 크게 할당되어 있어야 한다는 수행 전 조건은 /* @requires maxSet(dst) >= maxRead(src) @*/ 이고, 결과 조건은 /* @requires maxRead(src) == maxRead(src) @*/으로 나타낸다. 그 외에도 프로그래머에 의한 별도의 타입이나 조건 등의 표기를 통해 비교적 간단한 공정으로 보다 정교하게 C 문자열 함수들을 분석하는 연구들이 있어 왔다[Dor01,Shankar01]. U.C. Berkeley의 BOON(Buffer Overrun detectiON) [Wagner00]은 프로그래머에 의한 별도의 주석 표시 없이 현재 사용되고 있는 버퍼의 실제 할당량과 현재 자료 길이의 쌍으로 버퍼를 추상화하여 오버플로우 가능성에 대한 정밀한 분석을 하고 있다. 이러한 방식들을 사용하여 정교하게 의미 구조적으로 분석하게 되면, 정확한 분석에 의해 거짓 긍정(false-positive) 오류가 낮아지는 반면, 단순한 코드 스캐너에 비해 속도 및 활용성이 떨어지는 단점이 있다.

3.2 라이브러리

버퍼오버플로우에 취약한 strcpy(), strcat(), sprintf() 대신 C 표준 라이브러리의 strncpy(), strncat(), _snprintf() 등의 복사 대상의 크기를 적는 함수를 사용할 수가 있다.

F. Cavalier III에 의해 제작된 라이브러리인 libmib[Cavalier98]는 malloc(), free() 등과 호환이 되면서도 동적으로 버퍼를 재할당 받을 수 있는 astrcpy(), avsprintf() 등의 함수를 제공하여 안전한 코드 제작에 도움을 주고 있다.

Avaya사의 libsafe[Baratloo00]는 stycpy()와 같은 몇 개의 함수에 대해 둘러싸기(wrapping)를 하여 수행전에 입력 인자 길이와 할당된 버퍼 크기를 비교하는 검사를 동적으로 하도록 한다. 기존의 취약한 함수에 앞서 소개된 Splint나 BOON 등 정적 분석 기법을 통한 조건 분석에 해당 되는 코드를 둘러싼 새로운 함수를 제공한다. 이 때 자료를 저장할 버퍼의 최대 크기는 정확하게 분석하지 않고 리턴 주소값이 저장된 위치를 넘어가지 않는지 여부로 어림잡아 검사한다. libsafe는 동적 라이브러리 형태로 사용된다.

가 병합될 계획이라고 발표된 바 있다.

3.3 컴파일러 개조

컴파일러가 스택을 생성하는 방식을 바꾸고, 제반 점검에 필요한 코드를 삽입하여 생성하는 기법들은 제어 값에 대한 방어를 효과적으로 할 수 있다. 본 절에서는 리턴 주소 값과 함수 포인터 값에 대한 방어에 대해 다룬다.

■ 리턴 주소 값 덮어쓰기에 대한 방어 - 경계 설정

실제 공격에서 가장 많이 보이는 예는 스택 영역에 버퍼오버플로우를 발생시킴으로써 리턴될 주소 값을 바꾸는 것이다. 따라서 많은 연구들이 스택의 지역 변수 영역에서 넘친 데이터에 의한 리턴 주소 값의 손상을 막거나, 손상된 리턴 주소 값을 보존하기 위해 컴파일러를 개조하는 것에 초점을 맞추고 있다.

이러한 기법의 효시중 하나인 Stackguard[Cowan98]는 리턴 주소 값과 스택 사이에 '캐너리(canary)'라고 불리는 임의의 값을 끼워 두는 새로운 스택 모양을 제안하였다. 캐너리 값이 리턴 직전에 변했는지, 즉 새로운 값으로 덮어 씌여졌는지를 검사함으로써 스택 버퍼오버플로우 발생 여부를 확인한다. 캐너리 값은 공격자가 미리 예측하지 못하도록 수행 중에 임의의 숫자로 결정되는데, 공격이 일어난 경우 정상적인 복귀를 하기 위해 리턴 주소 값과 캐너리 값을 XOR 한 값도 캐너리와 더불어 저장하게 된다. '터미네이터 캐너리(terminator canary)'는 C에서 문자열의 끝을 나타내는 문자를 캐너리 값으로 하여 만일 공격자가 캐너리 값을 알아내어 이를 반영한 버퍼오버플로우를 시도한 경우에도, strcpy() 등의 문자열 연산 수행시 입력 값 중간에 문자열 끝을 만나게 하여 연산을 중단시킨다. 즉, 공격자 의도대로 긴 문자열의 복사가 일어나지 않게 되므로 결국 버퍼오버플로우를 막

을 수 있다. 이러한 StackGuard 기법은 현재 GCC 3.*에 포함되어 발표되었다[Immunix03]. 또한 IBM 에서도 Stackguard의 캐너리와 비슷한 개념의 가드(guard)를 사용하는 GCC 확장을 제공하고 있다[Etoh03]. 여기서는 그밖에도 모든 문자열 버퍼들을 스택 프레임의 가장 밑바닥에 저장하도록 변수 배치를 재조정함으로써 리턴 주소 뿐 아니라 문자열 버퍼를 제외한 다른 지역 변수들에 대한 보호도 가능하도록 하고 있다. Microsoft의 Visual C++.NET 컴파일러에서는 /GS 옵션을 통해 쿠키(캐너리) 값을 리턴 주소값과 XOR로 하여 저장하는 개념을 도입할 수 있도록 하고 있다[Bray02].

Stack Shield[Angelfire00]는 리턴 주소 값을 저장하는 별도의 스택을 배열 형태로 두고 있다. Global Ret Stack이라 불리는 이 자료구조를 써서 함수 리턴 직전 해당 리턴 주소 값을 읽어 사용한다. 만일 Global Ret Stack에 저장된 리턴 주소값과 실제 스택의 리턴 주소 값이 다르다면 공격임을 판단할 수 있다. 또는 Global Ret Stack의 값을 실제 스택의 리턴 주소값에 그대로 복사하여 사용함으로써 버퍼오버플로우 공격의 의도를 무산시킬 수도 있다[Wilander03]. 이와 관련하여 전체 스택 구조를 별도로 유지하지 않고 현재 함수의 리턴 주소 값만을 보관하여 사용하는 'Ret Range Check'가 제안되었다. 이와 비슷한 개념을 사용하는 RAD(Return Address Defender) [Chiueh01]는 RAR(Return Address Repository) 영역에 리턴 주소를 따로 저장하고 읽기 전용(Read- only)로 만드는 코드를 각 함수의 앞뒤에 삽입하는 GCC 컴파일러를 제안하였다.

기타 앞서 살펴본 동적 라이브러리 libsafe, libverify[Baratloo00]와 뒤에 살펴볼 StackGhost [Frantzen01] 등도 캐너리나 가드와 동일한 개념을 기본적으로 사용하고 있다. 이와 같은 방식들에서는 보관해 둔 비교 대상인 값 또는 범위를 지정하는 값이 공격자에 의해 변경되지 않도록 노력이 필요하다.

■ 포인터의 적정 영역 계산

일반 함수 포인터의 덮어쓰기를 막으려는 시도도 있어 왔다. StackShield는 모든 함수 포인터 값들이 정상적으로는 프로세스의 코드 영역, 즉 텍스트 세그먼트 내부를 가리킨다는 점에 착안하여, 이를 벗어나면 프로세스를 중지시키는 방법을 제안하고 있다[Wilandler03]. 이를 구현하기 위해 프로그램에 시작 부분에 새로운 전역 변수를 선언을 하나 추가하여 데이터 세그먼트와 텍스트 세그먼트의 경계를 표시한 후, 함수 포인터 값이 그 전역 변수 주소 값 보다 큰 주소 값 (데이터 세그먼트 부분)이면 프로그램을 중지시킨다. 이 방식은 정상적인 프로그래머가 의도적으로 동적인 코드 수행을 시도하는 경우에도 프로그램이 중지되는 거짓 긍정 오류를 발생시키는 단점이 있다.

■ 암호화

StackGhost[Frantzen01]는 StackGuard의 캐너리 개념과 StackShield의 리턴 주소를 저장해 두는 개념 모두를 제공함과 동시에, 리턴 주소 값을 암호화 시키는 기능을 제공한다. StackGuard를 제안했던 Immunix 사는 임의의 포인터 보호를 위한 Point Guard [Cowan 03]를 발표하였는데, 이것은 리턴 주소 값 외에 일반적인 포인터를 암호화하여 두고 연산할 때만 풀어서 수행하도록 GCC를 확장한 것이다.

■ 소스 코드가 주어지지 않는 경우의 대안

컴파일러를 개조함으로써 버퍼오버플로우 취약성을 완화시키는 방법은 소스 코드가 주어지지 않은 상황에서는 불가능하다. 따라서, COTS (Commercial Off-The Shelf) 제품에 적용하기 위하여 앞서 소개된 개념들을 컴파일러 개조 외의 다른 방법을 통해 지원하는 연구들이 있어 왔다.

Libsafe와 동일한 제작진에 의해 개발된 libverify [Baratloo00]는 동적 라이브러리 형태로 가드 개념을 지원하고 있다. 프로그램이 수행을 시작할 때에 프로세스 메모리에 있는 바이너리를 가드 검사 코드를 포함한 코드로 재작성 (rewrite) 하게 된다. 이 때 실제 수행되는 코드는 힙 내의 재작성 (rewrite) 된 복사본이 되기 때문에 만일 코드 내에 직접 분기가 있다면 다소 문제가 발생될 여지는 있다. 또한 실행 코드가 힙메모리에 복사됨으로써 속도면에서 불리하지만 소스 코드가 없는 경우에도 컴파일러 변경과 동일하게 처리 가능하다는 장점이 있다.

컴파일러의 개조 방법으로 가드 개념을 지원했던 RAD[Chiueh01]는 바이너리 코드를 재작성하는 방법으로도 버퍼오버플로우를 방어하는 기법을 제안하였다[Prasad03]. 이 경우 기계어 코드를 어셈블리어로 바꾸어, 각 함수마다 검사 코드를 앞뒤에 삽입하고 있다. 이 때 코드 영역의 크기를 컴파일이나 링크시 전처리하여 계산한 후 이를 이용하여 정상적인 리턴

주소 값이 가져야 하는 범위를 설정하여 판별하게 되는데, 국내에서도 비슷한 기법이 연구된 바 있다[김02].

앞서 리턴 주소 값을 암호하는 기법을 도입한 것으로 소개했던 StackGhost[Frantzen01]는 실제로는 컴파일러를 변경한 기법이 아니라 Sun Sparc의 레지스터 운영 방식을 이용하는 방법을 사용한다. Sun Sparc은 함수 호출/리턴에 따라 레지스터 윈도우가 슬라이딩하는 기법을 쓰는데, 이 때 레지스터의 재사용을 처리하는데에 있어 운영체제로의 트랩이 걸리게 되는 순간 가드나 캐너리에 관련된 원하는 동작을 수행하도록 하였다. 따라서, 단위 프로그램 레벨이 아닌 하드웨어의 운영체제 호출을 이용하여 구현한 것이므로 호스트 전체 차원에서 버퍼오버플로우를 제어할 수 있다는 특징이 있다.

3.4 실행 중 모니터링 기술

Java 언어의 경우에는 가상 머신이 수행 중에 배열의 인덱스를 올바르게 사용하였는지를 검사하고 있는데, 이것도 버퍼오버플로우를 방지하는 것의 일종으로 볼 수 있다. 그러나 이러한 방법은 다른 모니터링 기법들과 마찬가지로 수행 속도 면에서 다소 불리해 질 수 있다.

프로그램이 가지는 행위를 미리 기술해 놓고 이를 벗어나는지를 확인함으로써 버퍼오버플로우 등으로 인해 예기치 않은 악성 코드가 수행되는지 여부를 점검할 수 있는 도구들도 있다. U.C Berkeley의 D. Wagner는 정적 분석을 통해 프로그램의 성질을 push-down 오토마타로 뽑아내고 이에 맞는지를 점검하는 기법을 제안하였다[Wagner01]. State Univ. of New York at Stony Brook의 R. Sekar는 데이터 마이닝 기법을 통해 프로그램의 성질을 고유한 오토마타 형식으로 표현하는 것을 제안하였다[Sekar01]. 이러한 기법들은 적절한 형태로 프로그램의 수행 성질을 표현하는 것과 수행 중 속도에 지장을 줄이는 것이 중요하다.

Princeton Univ.가 제안하는 프로그램 목양자(program-shepherd) 기술[Kiriansky02]은 동적 최적화 기법이 가능한 경우 각 분기마다 최적화 알고리즘이 적용될 때에 함께 보안 점검을 하는 기법으로서, 특히 분기의 목적점 주소가 코드 적정 영역인지 파악하는 것을 여러 수준으로 나누어 검사할 수 있도록 하고 있다.

3.5 테스트

코드 개발자나 코드 설치자는 취약성 검사를 통하여 버퍼오버플로우의 가능성을 알아낼 수가 있다. Unix 유틸리티인 Fuzz[Miller95]는 gets() 등의 취약한 함수 사용이나 배열 할당량을 포인터나 첨자로 접근하는 것 등의 버퍼오버플로우 가능성을 검사해 준다.

U.C. Davis에서는 테스트할 대상 성질을 TASPEC 명세로 표현하여 여기서 자동으로 생성된 수행 모니터에 의해 테스트 중 명세의 올바름을 검사하는 프로퍼티 기반 테스트 (property-based test) 방식을 소프트웨어 취약성을 점검하는 도구로 제안하고 있다[Fink97]. 또한, 소스 코드가 주어진 경우에는 검사 코드를 삽입시켜 변형함으로써 소스 코드 스캐너에서 추론했던 결과를 테스트로 얻어내는 방법도 제안하고 있다[Haugh03]. 이로써 소스 스캐너가 수행중에만 결정될 수 있는 부분들에 대해 자세한 결과를 낼 수 없다는 점을 극복할 수 있으므로 함께 사용하면 훌륭한 보완이 될 수 있다.

Cigital사에서는 버퍼오버플로우 취약성에 대하여 수행 모니터 대신에 결함 주입 방식을 사용하는 AVA알고리즘을 사용하여 FIST(fault injection security tool)에서 구현하고 있다[Ghosh01]. FIST에서는 버퍼오버플로우가 일어나 리턴 주소를 덮어쓰게 되는 경우 해당 결함 주입 함수 (buffer overflow perturbation function)로 자동적으로 제어가 넘어가도록 되어 있다.

3.6 안전한 소프트웨어 개발 방법론

앞서 살펴본 바와 같이 다양한 대응 기법들이 있지만 아직도 버퍼오버플로우의 취약성에 따른 피해가 심각하다. 이는 이러한 기법들 각각이 가지는 한계 외에도 각 기법들의 활용도가 낮다는 데에서 기인한다고 볼 수 있다. 특히 현재의 소프트웨어 개발자들은 보안에 대한 인지도가 크지 않은데에다 시간이나 비용을 더 중요시 해야 하는 상황에 있다.

이미 1993년도부터 SSE-CMM (System Security Engineering-Capability Maturity Model)[Ferraiolo 96,SseCmm00]를 통해 소프트웨어를 비롯한 여러 제품의 생명주기 전반을 통해 일반적인 보안 취약성에

대한 검증을 주요 주제로 하는 개발 프로세스가 제안되어 현재까지 연구되고 있다. 이와 더불어 현재 SEI 의해 여러 가지 바람직한 성질의 제품 개발과정을 통합하여 안정성을 꾀하는 CMMI (Capability Maturity Model Integration) 프로세스[Cmmi]에서도 소프트웨어에 대한 일반적인 보안 관련 지침이 언급되고 있다. 또한 취약성을 중점적으로 평가하는 Octave[Alberts01]는 소프트웨어의 취약성에 대한 대비를 개발 프로세스에 흡수하고 있다. 그 밖에도 소프트웨어 개발 프로세스에 보안 강도를 일반적으로 높이는 방안을 강구하는 연구들은 계속 진행되고 있다[Mead02,Usher02].

소프트웨어 개발에 있어서 제시된 각종 세부적인 보안 지침문들은 개발 프로세스와는 다른 접근방법으로 취약성 완화를 보다 적극적으로 유도하고 있다 [Wheeler03b,Howard02,Peteanu01]. 버퍼오버플로우의 대비책에 대해서 이들은 앞절에서 언급된 strncpy등의 라이브러리나 IBM GCC 확장 Microsoft Visual C++.NET 의 /GS 옵션 등의 컴파일러 솔루션 등을 구체적으로 권하고 있다.

4. 결 론

버퍼오버플로우는 C나 C++과 같이 융통성(flexibility)과 성능(performance)을 위해 메모리를 직접 접근하는 것을 허용하는 언어에서 피할 수 없는 취약점일 수가 있다. 그러나 이를 완화시키기 위한 다양한 방법들이 제시되어 왔고 실제로 잘 활용된다면 많은 공격들을 막아낼 수 있다고 보아진다. 현재까지 제안된 버퍼오버플로우 공격에 대한 대응 방안으로는 소스 코드 스캐너, 라이브러리, 컴파일러의 개조, 수행 중 모니터링, 테스팅 기법등이 있고, 일부는 이미 GCC나 Microsoft 제품에 도입되어 있다. 그 밖에도 본 글에 미처 정리되지 못한 가치있는 연구들도 다수 있을 것으로 생각된다.

그러나, 이것으로써 버퍼오버플로우 취약점에 대해 안심할 단계로 보기는 어렵다. 현재까지 제안된 기법들은 잘 알려진 스택 오버플로우에 대한 리턴 주소값의 보호나 몇몇 취약한 라이브러리 함수 사용 배제 등에 많은 부분 치우쳐 있었으므로[Wilander03], 향후에는 공격자들이 지능화될 것을 대비하여 폭넓은 공격에 대한 대응책이 연구되어야 할 것이다. 또한 제안된 각 기법은 실효성을 위해 수행 속도나 자원 활용에서 뒷받침 해주는 연구가 뒤따라야 하며, 소프트웨어 개발 프로세스 등에 대응 기법 활용이 구체적으로 강제되어야만 효과가 있을 것으로 본다.

참고문헌

[김01] 김종의 이성욱 홍만표, 버퍼오버플로우 공격 방지를 위한 컴파일러 기법(Improving Compiler to Prevent Buffer Overflow Attack), 한국정보처리학회지(C) 9(4), pp. 453-458, 2002

[안02] Code_Red_II, http://home.ahnlab.com/smart4u/virus_detail_852.html

[안03] SQL_Overflow, http://home.ahnlab.com/smart4u/ virus_detail_1098.html

[Alberts01] C.J. Alberts and A.J. Dorofee, OctaveSM Criteria Version2.0, TECHNICAL REPORT CMU/SEI-2001-TR-016, Carnegie Mellon University, 2001 [Angelfire00] Stack Shield-A "stack smashing" technique protection tool for Linux, www.angelfire.com/sk/stackshield/

[Baratloo00] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In Proceedings of USENIX Annual Technical Conference, June, 2000.

[Bray02] B. Bray, Compiler Security Checks In Depth, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vctchcompilersecuritychecksindepth.asp

[Cavalier98] F. J. Cavalier III, LibMib Allocated String Functions, http://www.mibsoftware.com/libmib/astring/, 1998

[Chiueh01] T. Chiueh and F. Hsu, "RAD: A compile time solution for buffer overflow attacks," in proc. of ICDCS 2001

[Cmmi] Capability Maturity Model Integrated (CMMI) main page, http://www.sei.cmu.edu/cmmi/

[Cowan98] C.Cowan et. al, Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.in proc. of the 7th USENIX Security Conference, January,

1998.

[Cowan03] C. Cowan et. al, PointGuardTM: Protecting Pointers from Buffer Overflow Vulnerabilities In Proc. 12th USENIX Security Symposium, Washington, D.C., August, 2003.

[Dor01] N. Dor, M. Rodeh and M. Sagiv, Cleanness Checking of String Manipulations in C Programs via Integer Analysis, Static Analysis Symposium (SAS01), Paris, France, 16-18 July, 2001

[Etoh03] H. Etoh et. al, GCC extension for protecting applications from stack-smashing attacks, http://www.research.ibm.com/trl/projects/security/ssp/

[Ferraiolo96] K. Ferraiolo, System Security Engineering Capability Maturity Model, Model, in proc. of the 19th national information systems security conference, 1996, now available at http://www.secat.com/download/pdf/ssecmm.pdf

[Fink97] G. Fink and M. Bishop, "Property Based Testing: A New Approach to Testing for Assurance," ACM SIGSOFT Software Engineering Notes, 22(4), 1997

[Frantzen01]M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In Proc. 10th USENIX Security Symposium, Washington, D.C., August, 2001.

[Ghosh01] A. Ghosh, T. O'Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. In Pro.of the IEEE Symposium on Security and Privacy, May, 1998.

[Haugh03] E. Haugh and M. Bishop, "Testing C Programs for Buffer Overflow Vulnerabilities," in proc. of the Network and Distributed System Security Symposium, San Diego, CA, February, 2003

[Howard02] M. Howard and D. LeBlanc, Writing Secure Code, 2nd, Microsoft Press, 2002

[Immunix03] Stackgurad, http://www.immunix.org/stackgurad.html, 2003

[Kiriansky02] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In 11th USENIX Security Symposium, Aug. 2002.

[Krsul98] I. Krsul, E. Spafford and M. Tripunitara, An Analysis of Some Software Vulnerabilities in proc. of the 21st NIST-NCSC National Information Systems Security Conference, 1998

[Larochelle01] D. Larochelle and D. Evans, Statically Detecting Likely Buffer Overflow Vulnerabilities, in proc. of 2001 USENIX Security Symposium, Washington, D. C., August, 13-17, 2001

[Mead02] N. R. Mead et. al, Life-Cycle Models for Survivable Systems, Technical Report CMU/SEI-2002-TR-026, Carnegie Mellon University, 2002

[Miller95] B. P. Miller et. al, Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical report, Computer Science Department, University of Wisconsin, November, 1995

[Peteanu01] R. Peteanu, Best Practice for Secure Development, avalable at http://members.home.net/razvan.peteanu/

[Prasad03] M. Prasad and T. Chiueh, A Binary Rewriting Defense against Stack-based Buffer Overflow Attacks, in proc. of USENIX Annual Technical Conference, 2003

[Shankar01] U. Shankar et. al, Detecting format string vulnerabilities with type qualifiers. in proc. of the 10th USENIX Security Symposium, 2001.

[SecureSoft] Rats (the Rough Auditing Tool for Security), http://www.securesoftware.com/white_papers.htm, Secure Software

[Sekar01] R. Sekar et. al, "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors," IEEE Symposium on Security and Privacy, Oakland, CA, 2001.

[SseCmm] SSE-CMM (Systems Security Engineering-Capability Maturity Model), http://www.sse-cmm.org/

[Usher92] R. . Usher Improving Software Security

During Development, SANS InfoSec Reading Room, http://www.sans.org/rr/papers/index.php?id=384, submitted to fulfill the practical assignment: SANS Security Essentials GSEC Practical Assignment, Version 1.3

[Viega00] J. Viega, J.T. et. al, ITS4: A static vulnerability scanner for C and C++ code. In Annual Computer Security Applications Conference, 2000.

[Wagner00] D. Wagneret. al, A first step towards automated detection of buffer overrun vulnerabilities. In Proceedings of the Network and distributed system security symposium, February 2000.

[Wagner01] D. Wagner and D. Dean, Intrusion Detection via Static Analysis, in proc of 2001 IEEE Symposium on Security and Privacy, 2001

[Wheeler03a] D. Wheeler, FlawFinder Document , http://www.dwheeler.com/flawfinder/flawfinder.pdf, 2003

[Wheeler03b] D. Wheeler, Secure Programming for Linux and Unix HOWTO, http://www.dwheeler.com/secure-programs/, 2003

[Wilander02] J. Wilander and M. Kamkar, A Comparison of Publicly Available Tools for Static Intrusion Prevention, in proc. of the 7th Nordic Workshop on Secure IT Systems (Nordsec 2002), November, 7-8, 2002, Karlstad, Sweden.

[Wilander03] J. Wilander and M. Kamkar, A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention, in proc. of the 10th Network and Distributed System Security Symposium (NDSS'03) February, 5-7, 2003, San Diego, California.

조 은 선

1991. 2 서울대학교 계산통계학과(학사)
1993. 2 서울대학교 전산과학과(석사)
1998. 8 서울대학교 전산과학과(박사)
1999. 4~2000. 4 한국과학기술원 전임연구원
2000. 5~2002. 2 아주대학교 정보통신전문대학원 조교수 대우
2002. 3~현재 충북대학교 전기전자컴퓨터공학부 조교수
관심분야 : 정보보호, 프로그램 분석
E mail : eschough@chungbuk.ac.kr

The 14th Joint Conference on Communications & Information(JCCI 2004)

- 일 자 : 2004년 4월 28~30일
- 장 소 : 금호 충무마리나리조트(충무)
- 주 최 : 정보통신연구회
- 상세안내 : KAIST 이용훈 교수(Tel. 042-869-4411)

http://www.jcci21.or.kr