DOI: 10.3745/KIPSTC.2011.18C.2.089

무차별 공격에 효과적인 다중 Address Space Randomization 방어 기법

박 수 현†·김 선 일††

요 약

Address Space Randomization(ASR)은 성능 부하가 없고 광범위한 데이터 메모리 영역의 보호가 가능한 우수한 방어 기법이다. ASR은 사용 가능한 데이터 메모리 영역 내에서 변수를 재배치 함으로써 공격자에게 변수의 주소를 숨기는데, 데이터 메모리 영역의 크기가 한정되어서 무차별 공격에 취약한 단점이 있다. 본 논문은 기존 ASR의 단점을 제거하기 위한 다중 ASR 기법을 제시한다. 다중 ASR 기법은 데이터 메모리 영역을 원본 및 복사 영역으로 나누고 각 메모리 영역의 변수 값을 비교함으로써 공격을 탐지하고 방어한다. 다중 ASR에서 각 데이터 메모리 영역의 변수는 서로 다른 순서로 배치되므로 한 번의 공격을 통해 동시에 동일한 변수 값을 조작하는 것은 불가능하다. 다중 ASR이 적용된 프로그램은 중복 수행으로 인해 비교적 높은 성능 부하를 보이나, 실제 공격 대상이 되는 웹서버 등 I/O 처리가 많이 요구되는 프로그램의 경우 40%~50% 정도의 성능 부하를 보인다. 아울러 본 논문에서는 프로그램에 다중 ASR을 적용하기 위한 변환프로그램을 개발하였다.

키워드 : 프로그램 보안, ASR(Address Space Randomization), 버퍼오버플로, 프로그램 변환

Multiple ASR for efficient defense against brute force attacks

Soohyun Park† · Sunil Kim††

ABSTRACT

ASR is an excellent program security technique that protects various data memory areas without run-time overhead. ASR hides the addresses of variables from attackers by reordering variables within a data memory area; however, it can be broken by brute force attacks because of a limited data memory space. In this paper, we propose Multiple ASR to overcome the limitation of previous ASR approaches. Multiple ASR separates a data memory area into original and duplicated areas, and compares variables in each memory area to detect an attack. In original and duplicated data memory areas variables are arranged in the opposite order. This makes it impossible to overwrite the same variables in the different data areas in a single attack. Although programs with Multiple ASR show a relatively high run-time overhead due to duplicated execution, programs with many I/O operations such as web servers, a favorite attack target, show 40~50% overhead. In this paper we develop and test a tool that transforms a program into one with Multiple ASR applied.

Keywords : Program Security, ASR(Address Space Randomization), Buffer Overflow, Program Transformation

1. 서 론

프로그램 공격은 프로그램이 가진 보안 취약점 등을 이용하여 공격자가 원하는 대로 프로그램의 실행 흐름을 조작하는 것을 의미한다. 보안 취약점은 입력 데이터의 크기를 검사하지 않고 메모리 영역으로 복사하는 이른바 버퍼 오버플로 취약점이 대표적으로, 이를 이용한 공격은 프로그램의 의미(semantic)을 벗어나서 비정상적인 방법으로 메모리 영역의 값을 덮어쓸 수 있다. 프로그램 공격은 이러한 프로그램의 취약점을 이용하여 스택 스매싱[1], 힙 오버플로우[2], 포맷 스트링[3]와 같이 다양한 공격 기법을 사용하여 프로그램을 장악하고 조작한다. 공격자가 원하는 대로 프로그램이 실행되는 경우 시스템 전체로 피해가 확산될 수 있으며, DDoS[4]의 좀비 프로세스[5]와 같이 다른 공격에 사용되기도 한다.

프로그램 공격을 막기 방어 기법은 대표적으로 ASR[6], 스택 가드[7], 포인트 가드[8]를 들 수 있다. 스택 가드는 스택 스매싱 공격을 막기 위해 고안되었으며 스택에 저장되는 반환 주소 앞뒤에 난수 값(Canary Words)을 배치한다. 스택 스매싱은 버퍼 오버플로 취약점을 이용하는데 반환 주소

※ 이 논문은 2010학년도 홍익대학교 학술연구진흥비에 의하여 지원되었음.
† 준 회 원 : 홍익대학교 컴퓨터공학과 박사과정
†† 종신회원 : 홍익대학교 컴퓨터공학과 교수
논문접수 : 2010년 10월 27일
수 정 일 : 1차 2011년 1월 24일
심사완료 : 2011년 3월 10일

를 변경하기 위해 취약점이 있는 변수 지점부터 반환 주소를 포함한 영역을 원하는 값으로 바꾸어 쓴다. 따라서 반환 주소 값의 앞 뒤에 위치한 난수 값도 함께 바뀌게 된다. 스택 가드에서는 함수 반환 직전 난수 값이 변경되었는지 검사함으로써 공격 여부를 탐지할 수 있다. 포인트 가드는 프로그램에서 사용되는 포인터 변수 값을 특정 암호 키를 사용하여 암호화하고, 사용하기 전에 복호화하는 방식이다. 공격자가 암호화 된 포인터 값을 공격 코드의 주소로 덮어쓰게 되면 복호화 과정에서 전혀 다른 값으로 바뀌게 되며 공격이 실패하게 된다.

ASR(Address Space Randomization) 기법은 실행 시 데이터 메모리 영역의 주소를 임의로 재배치하여 프로그램이 사용하는 변수의 주소를 숨긴다. 공격자는 특정 변수의 주소를 알고 그 값을 바꾸어 프로그램을 공격하는데 ASR에서는 실행 시 변수가 재배치 되기 때문에 원하는 변수의 값을 바꾸는 것이 어렵게 된다. ASR는 스택 가드나 포인트 가드와 달리 공격 탐지 검사를 위한 연산이나 암호화 및 복호화 과정이 없어서 실행 시간 동안의 부하가 없고, 스택이나 데이터, 힙과 같이 광범위한 메모리 영역의 보호가 가능하다. 따라서 매우 적은 부하로 버퍼 오버플로우, 스택 스매싱, 포맷 스트링과 같은 다양한 공격을 방어할 수 있으며 최근 운영체제 다수가 방어 기법으로 채택하고 있다[9][10]. 하지만 ASR 기법은 재배치에 사용할 수 있는 메모리 공간이 한정되기 때문에 무차별 공격에 취약하다. 무차별 공격은 공격 대상이 되는 변수가 위치 할 수 있는 모든 메모리 영역의 값을 차례로 하나씩 바꾸어 보는 것이다. 실제로 특정 변수가 위치 할 수 있는 영역은 프로세서가 사용할 수 있는 전체 데이터 영역보다 작고 따라서 무차별 공격이 성공할 확률은 상당히 높다[11].

본 논문은 기존 ASR의 단점을 보완하여 효과적으로 공격을 탐지하고 방어할 수 있는 다중 ASR 기법을 제시한다. 다중 ASR 기법에서는 모든 변수가 데이터 메모리상의 원본 장소와 복사본 장소 두 위치에 존재하게 된다. 프로그램이 실행되면서 각 변수의 값은 동일하게 변한다. 공격자는 변수의 값을 바꾸고자 할 때 해당 변수의 두 위치를 모두 알더라도 다른 위치에 있는 변수를 동일한 값으로 바꿀 수 없다. 변수의 원본 장소와 복사본 장소의 값을 비교하여 공격이 탐지된다. 따라서 다중 ASR에서는 변수의 주소가 알려져도 방어가 가능하고 무차별 공격도 탐지 된다.

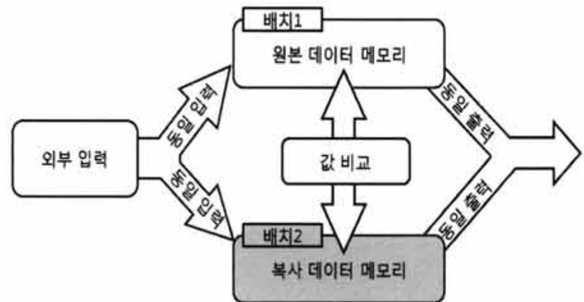
본 논문은 다음의 순서로 구성되었다. 2장은 다중 ASR이 기법을 설명하고 3장에서 프로그램에 다중 ASR을 적용하기 위해 개발된 RETMAS를 살펴본다. 4장은 다중 ASR을 통한 공격 방어 및 성능 평가를 살펴본 뒤 5장에서 결론을 서술하고 마친다.

2. 다중 ASR

다중 ASR은 기존 ASR과 달리 변수의 배치가 알려지거나 무차별 공격이 있어도 이를 방어 할 수 있다. 공격이 있을 시 원본 데이터 영역과 복사본 데이터 영역의 변수 값은 서로 다르게 변하게 되고 변수 값을 비교하여 공격을 탐지한다. 이를 위하여 RETMAS는 프로그램 코드 영역을 수정하여 본 절에서는 다중 ASR이 어떻게 작동하며 공격을 어떻게 탐지 하는지 살펴 본다.

2.1 다중 ASR 동작 과정

다중 ASR에서는 데이터 메모리 영역을 원본 및 복사본 영역으로 나누고, 각 영역에서 변수들의 순서를 반대로 배치한다. 원본 데이터 메모리 영역의 변수(원본 변수)및 복사 데이터 메모리 영역의 변수(복사 변수) 값을 동일하게 바꾸어 주기 위해 기존 프로그램의 연산과 같지만 복사 변수에 대해 수행되는 연산을 추가한다. 이렇게 수정된 프로그램은 실행 과정에서 원본 및 복사 변수에 동일한 연산을 번갈아 가며 수행한다. 실행 도중 외부로부터 데이터가 입력되면 원래의 연산과 추가된 연산에 전달하고 각각의 연산을 거쳐 원본 및 복사 변수에 동일하게 전달 된다. (그림 1)은 이러한 과정을 보여주고 있다.

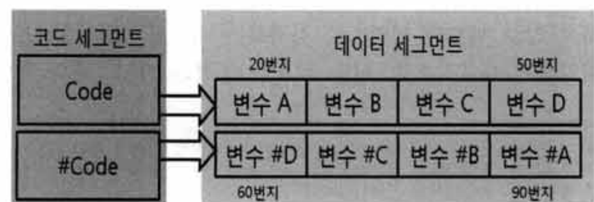


(그림 1) 두 개의 데이터 메모리 영역을 가지는 프로그램

스택 스매싱과 같은 공격이 입력 데이터로 주어지면 데이터 메모리의 특정 주소에 있는 값이 조작되나 각 영역의 변수 주소가 서로 다르기 때문에 한 번의 공격으로 동일한 변수 값을 모두 변경할 수 없다. 공격에 의해 변경된 변수를 사용하기 전에 각 변수 값이 같은지 검사해서 공격을 탐지하고 방어할 수 있다. 다음 2.3절에서는 다중 ASR에서 공격을 탐지하는 방법을 구체적으로 살펴본다.

2.2 원본 및 복사 변수의 배치

공격 발생시 각 변수 값이 반드시 서로 다르도록 하기 위해 다중 ASR 기법은 각 영역의 변수 배치 순서를 서로 반대로 한다.



(그림 2) 데이터 메모리의 분할과 변수의 재배치

다중 ASR을 통해서 각 데이터 메모리 영역은 (그림 2)와 같이 메모리 내의 실제 주소를 의미하는 절대 주소와, 변수 간의 위치 관계를 의미하는 상대 주소에 있어서 서로 다르게 된다. 공격자가 함수 반환 주소와 같은 특정 변수의 값을 바꾸려고 해도 입력이 동일하게 전달되기 때문에 공격은 한 번에 하나의 주소에 있는 값만 조작할 수 있다. 이로 인해 각 변수 값이 서로 다르게 되어 변수 값을 사용하기 전에 비교하여 공격을 탐지하고 방어할 수 있다. 예를 들어 변수 B에서 버퍼 오버플로우 취약점을 이용한 공격을 통해서 메모리 영역을 덮어쓰게 되면 변수 B에서 시작하여 메모리 주소가 증가하는 방향의 값들이 모두 바뀌는데, 결과적으로 원본 변수 C, D의 값이 덮어쓰여진다. 마찬가지로 복사 영역에서 변수 #B에서 버퍼 오버플로우 취약점을 이용한 공격을 통해 덮어쓰게 되면 원본 영역과 마찬가지로 메모리 주소가 증가하는 방향의 값이 바뀌에 되어 결과적으로 복사 변수 #A의 값이 바뀌게 된다. 따라서 원본 변수 D와 복사 변수 #D는 그 값이 서로 다르며 변수 D를 사용하기 전 비교함으로써 공격을 탐지할 수 있다. 원본 변수 B와 복사 변수 #B는 이 경우 똑 같은 값을 갖게되나 변수 B 값의 수정은 프로그램의 동작에 있어서 의도된 부분이기 때문에 변수 B 값을 수정하여 공격이 가능한 취약점은 일종의 프로그램 버그(bug)로 볼 수 있으며, 다중 ASR에서 검출할 수 있는 공격 대상에서 벗어난다. 메모리 20번지에서 90번지 까지의 값을 모두 동일하게 덮어 쓰는 공격이 발생하면 변수 값이 동일하게 변해서 공격을 탐지할 수 없으므로, 다중 ASR은 원본 및 데이터 메모리 영역 사이에 운영체제가 제공하는 mprotect() 시스템 콜 등을 사용하여 쓰기 금지 영역을 설정하고 이러한 공격을 막는다. 공격이 쓰기 금지 영역의 값을 조작하려고 시도할 경우 운영체제에서 세그먼트 폴트(segment fault)[12] 를 발생시키고 공격을 탐지할 수 있다.

2.3 변수 값의 비교를 통한 공격 탐지 및 방어

위에서 살펴본 것 처럼 공격이 발생할 경우 원본 및 복사 변수의 값은 서로 일치하지 않게 된다. 따라서 변수 값의 비교를 통해 공격을 탐지하고 방어할 수 있다. 변수 값의 비교 시점은 <표 1>과 같이 세 가지를 고려할 수 있다.

〈표 1〉 상태 변화 감지 시점의 비교

비교 시점	장점	단점
변수를 이용하는 모든 연산시	메모리 상의 모든 변수에 대한 세밀한 변경 추적이 가능	부하가 매우 큼
조건 분기문과 같은 제어 흐름 변화시	제어 흐름에 관련된 데이터 변화를 추적하고 공격자가 임의로 제어 흐름을 변경하는 것을 막을 수 있음	제어 흐름에 관계되지 않은 데이터 변화를 추적할 수 없음
시스템 콜 호출시	가장 적은 부하로 공격 탐지 가능 시스템 상태의 변화 추적 가능	시스템 콜이 사용하지 않는 메모리나 레지스터 값 변화를 추적할 수 없음

프로그램 실행 과정에서 변수를 사용하는 경우는 ADD, SUB와 같은 단순한 연산에서 조건 분기문의 검사 값, 함수 호출 시 사용되는 함수 포인터 값 사용 등 매우 다양하나 <표 1>에서와 같이 크게 세 가지로 나눌 수 있다. 변수를 이용하는 모든 연산은 명령어에 변수가 포함되는 모든 경우를 의미하며, 이 경우 거의 대부분의 명령어를 실행하기 전 변수 값을 비교해야 하므로 부하가 매우 크다. 따라서 다중 ASR은 제어 흐름이 변화하는 명령어 실행 또는 시스템 콜 호출 이전에 사용되는 변수 값을 비교한다. 제어 흐름 변경 시 사용되는 변수 값을 비교함으로써 제어 흐름에 대한 공격 탐지가 이루어지며, 시스템이나 프로그램의 직접적인 상태 변경에 관계된 시스템 콜 호출 시 인자 값의 비교를 통해 비 제어 흐름 공격에 대한 탐지가 이루어진다. 제어 흐름 공격 및 비 제어 흐름 공격 탐지는 3장에서 자세히 설명한다.

3. RETMAS(Redundant Execution Translator for Multiple ASR)

다중 ASR을 적용하기 위해 프로그램을 분석하고, 프로그램의 코드 세그먼트를 수정하고, 두 개의 데이터 메모리 영역을 만들기 위해 프로그램 변환하는 툴인 RETMAS (Redundant Execution Translator for Multiple ASR)를 구현하였다. RETMAS는 리눅스 기반 시스템에서 목적 파일이나 실행 파일의 처리를 위한 규격인 ELF(Executable and Linkable Format)[13]를 분석해서 코드 세그먼트에 연산을 추가하고, 모든 연산이 각각의 데이터 메모리 영역에 주어진 변수 배치를 유지할 수 있도록 수정한다. 연산에 사용되는 명령어는 Intel Architecture[14]를 기준으로 만들었다.

3.1 코드 세그먼트 분석 및 수정

ELF 규격에 따르면 목적 파일이나 실행 파일은 실행 가능한 명령어 집합을 가지는 섹션(section)을 가지고 있으며, 섹션이 모여서 단일 세그먼트를 이룬다. RETMAS는 이러한 섹션을 분석하여 목적 파일이나 실행 파일이 가지고 있는 명령어 목록을 만든다. 또한 새로운 명령어 생성 및 배치를 위해 필요한 재배치 정보(relocation information), 심볼 정보(symbol information)도 분석한다.

RETMAS는 코드 세그먼트의 분석이 완료되면 명령어 목록을 순회하면서 연산은 동일하고 연산에 사용되는 레지스터나 메모리 주소를 조정하여 복사 데이터 메모리 영역에서 수행되는 명령어를 새로 생성하고 삽입한다. 수정된 코드 세그먼트는 순차적으로 명령어를 수행할 경우 원본 및 복사 변수에 동일한 연산을 번갈아 가며 수행하게 된다. 생성 및 수정의 대상이 되는 명령어는 MOV, XOR, ADD, PUSH와 같이 레지스터 혹은 메모리 주소를 지정해야 하는 모든 명령어가 된다. 인터럽트[15] 발생 명령어인 INT와 같이 레지스터 혹은 메모리 주소가 지정되지 않는 명령어나 FPU(Floating Point Unit)과 같이 내부 메모리 스택을 사용

〈표 2〉 명령어에 대한 처리 예

변경 전	변경 후	비고
MOV EAX, 4(EBP)	MOV EAX, 8(EBP) *MOV ECX, 36(EBP)*	스택 크기가 20인 경우, 이를 40만큼 확장하며 각 영역의 변수 배치 순서가 반대이기 때문에 원본 변수의 EBP+4 주소는 복사 변수의 EBP+24 주소와 대응한다.
XOR VAR, EBX	XOR VAR, EBX *XOR #VAR, EDX*	복사 변수 #VAR은 원본 변수 VAR에 대응한다. EAX, EBX 레지스터는 추가되는 연산에서 각각 ECX, EDX에 대응된다.
ADD EBX, ECX	ADD EBX, ECX *PUSH ECX* *MOV VIRT_ECX, ECX* *ADD EDX, ECX* *MOV ECX, VIRT_ECX* *POP ECX*	기존 코드가 ECX 레지스터를 사용하고 있기 때문에 레지스터가 부족하다. 따라서 추가 생성되는 코드는 가상의 레지스터 변수인 VIRT_ECX를 만들어서 사용한다. 추가되는 코드는 ECX 레지스터 값을 대피시킨 후, VIRT_ECX 값을 복구시켜 사용한 뒤 저장하고 다시 ECX 값을 대피된 값으로 복구한다.

〈표 3〉 조건 분기문의 처리 예

변경 전	변경시 문제점	올바른 처리
CMP i, 0 BEQ LABLE …… LABLE: ……	CMP i, 0 BEQ LABLE ***CMP #i, 0 // 위의 CMP i, 0 결과에 따라 실행 여부가 결정됨*** ***BEQ LABLE*** …… LABLE:	CMP i,0 ***BEQ COMPARE_#i*** ***JMP COMPARE_NOT_#i*** ***COMPARE_#i:*** ***CMP #i, 0*** BEQ LABLE ***JMP DETECT_ATTACK*** …… ***COMPARE_NOT_#i:*** …. LABLE:

하는 명령어는 추가 생성 및 수정의 대상이 되지 않는다. 아래 <표 2>는 명령어에 대한 추가 및 생성 방법을 설명한다. 생성되는 명령어는 기존의 명령어와 다른 레지스터나 메모리 주소, 즉 복사 변수를 대상으로 연산을 수행하게 된다.

프로그램의 모든 명령어를 <표 2>의 방법으로 처리할 수는 없다. 조건 분기문이나 함수 호출과 같이 실행 흐름에 영향을 주는 명령어는 추가로 생성하고 수정할 경우 실행 흐름에 영향을 주거나 실행이 안되는 경우가 있다. 다음 3.2, 3.3에서는 각각 조건 분기문과 함수 호출 및 반환에 대한 다른 처리 방법을 설명한다.

3.2 조건 분기문의 처리

조건에 따라 서로 다른 실행 흐름으로 분기하는 조건 분기문은 3.1, <표 2>의 방법에 따라 명령어가 추가될 경우 조건 검사 구문 바로 다음 명령어가 실행되지 않는 경우가 발생한다. 예를 들어 원본 변수 i의 값이 0인지 검사하는 명령어에 대응하여 바로 다음에 i의 복사 변수 #i가 0인지 검사하는 명령어를 추가할 경우 변수 i와 0의 비교 결과에 따라 #i과 0을 비교하고 분기하는 명령어는 실행되지 않을 수 있다. 따라서 RETMAS는 조건 분기문을 다른 방식으로 처리하게 된다. <표 3>은 변수 i의 값이 0인지 검사해서 일치할 경우 LABLE 구문으로 실행 흐름을 변경하는 조건 분기문을 <표 2>와 같이 변경할 때의 문제점과 올바른 처리 방식을 표현한다.

조건 분기문에서 사용되는 원본 변수와 복사 변수의 값을 비교하여 일치하지 않을 경우 공격으로 간주한다. <표 3> 올바른 처리에서 원본 변수 i값이 0인 경우 복사 변수 #i 값 또한 0이어야 하므로, 이를 COMPARE_#i 구문에서 비교하여 0이 아니면 공격으로 판단한다. i값이 0이 아닌 경우도 동일하게 COMPARE_NOT_#i 구문에서 검사한다. 조건 분기문에서 변수 값을 비교하는 것은 조건 분기문에 사용되는 변수와 같이 간접적으로 제어 흐름에 관계된 데이터 또한 빈번한 공격 대상이 되기 때문이다[16]. 조건 분기문에서 각 변수 값을 비교하고 공격이 탐지되지 않은 경우 정상적인 실행 흐름을 따른다. 프로그램은 조건 분기문에서 변수 비교를 통해 제어 흐름에 관계된 데이터에 공격이 발생하는 것을 탐지하고 방어할 수 있다. 두 값이 동일할 경우 기존 프로그램과 동일한 실행 흐름을 따른다.

3.3 함수 호출 및 반환

함수 내부의 명령어들은 3.1, <표 2>에서 설명한 방법에 따라 내부적으로 추가 연산을 가지게 되므로 함수를 두 번 호출할 필요는 없다. 한 번의 함수 호출만으로 원본 및 복사 데이터 메모리 영역에 대한 연산을 모두 수행할 수 있기 때문이다. 다만 함수 내부에서 처리하는 인자나 반환값 또한 원본 및 복사 메모리 영역으로 나뉘기 때문에 함수 호출부에서 인자로 사용되는 원본 및 복사 변수모두 전달해야 하며, 마찬가지로 함수는 원본 및 복사 반환값을 모두 반환

〈표 4〉 함수 호출 시 인자 전달 처리 예시

	변경 전	변경 후	비고
스택을 통한 인자 전달	PUSH ARG3 PUSH ARG2 PUSH ARG1 CALL FOO	***PUSH #ARG2*** ***PUSH #ARG1*** PUSH ARG2 PUSH ARG1 CALL FOO	원본 변수 ASR1,ARG2과 대응하는 복사 변수 #ARG1, #ARG2도 스택을 통해 전달한다. 함수 내부에서 인자를 사용하는 명령어가 인자의 주소를 반영하기 때문에 인자의 전달 순서는 관계없다.

레지스터를 통한 인자 전달	MOV ARG1, EAX
	MOV ARG2, EBX
	MOV ARG3, ECX
	CALL FOO
		MOV ARG1, EAX
		MOV ARG2, EBX
		MOV ARG3, ECX
		MOV #ARG1, VIRT_EAX
		MOV #ARG2, VIRT_EBX
		MOV #ARG3, VIRT_ECX
		CALL FOO
			기존 함수가 EAX~ECX 레지스터 세개를 사용하기 때문에 레지스터 공간이 부족하다. 이 경우 메모리 영역에 생성된 가상 레지스터 변수(VIRT_EAX~VIRT_ECX)를 사용해서 인자를 전달한다.

〈표 5〉 함수 반환 값 처리 예시

	변경 전	변경 후	비고
함수 반환시 반환값 처리	MOV VAR, EAX LEAVE RET	MOV VAR, EAX ***MOV #VAR, ECX*** LEAVE RET	원본 변수 VAR에 대응하는 복사 변수 #VAR를 EAX에 대응하는 레지스터 ECX에 복사하고 반환한다.
함수호출부의 함수 반환값 처리	CALL FOO MOV EAX, VAR	CALL FOO MOV EAX, VAR ***MOV ECX, #VAR***	변경 후에는 EAX및 ECX로 전달된 두 반환값을 각각 원본 및 복사 데이터 메모리로 복사한다.

하고 호출부는 반환된 값을 올바르게 처리해야 한다. <표 4>는 함수 호출 시 인자 전달 방법을, <표 5>는 반환부 부분의 명령어 생성과 함수 내부에서 인자 및 반환 값을 처리하는 방법을 설명한다.

RETMAS는 모든 함수 및 함수 호출에 대해서 <표 4>의 방법으로 처리가 가능하나, 시스템 콜의 경우는 다르다. 시스템 콜은 운영체제의 커널 내부에 구현되어 있기 때문에 RETMAS에 의한 수정이 불가능하다. 수정이 되지 않은 시스템 콜을 그대로 사용하면 복사 데이터 메모리 영역의 값이 원본과 동일하게 유지되지 않기 때문에 RETMAS는 시스템 콜 수정 대신 시스템 콜 래퍼(wrapper)를 별도로 정의하여 사용한다.

3.4 시스템 콜 래퍼(System Call Wrapper)

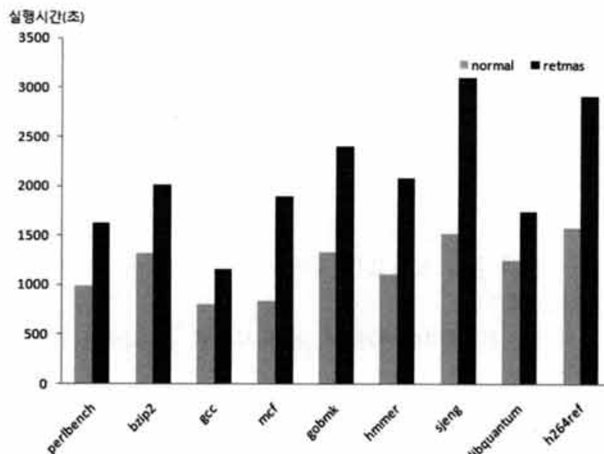
시스템 콜 래퍼는 RETMAS에 의해 수정된 프로그램에서 시스템 콜을 호출 시 운영체제 커널 내부에 정의된 시스템 콜 대신 호출되는 함수다. 시스템 콜 래퍼가 실제 커널 내부의 시스템 콜을 대체할 수 없기 때문에 전달된 인자 값을 변경하고 검사한 뒤 커널 내부의 시스템 콜 호출을 호출하고 그 결과 값을 각 변수에 동일하게 복사해 준다. 따라서 시스템 콜 래퍼는 RETMAS에 의한 변환 대상이 아니며 미리 정의된 라이브러리를 사용하게 된다. 시스템 콜 래퍼에 전달되는 인자는 다른 함수와 마찬가지로 각 데이터 메모리 영역에 대한 인자값이 모두 전달되기 때문에 기존의 시스템 콜 인자보다 두 배 많다. 하지만 내부적으로 해당 인자를 모두 사용하여 연산을 처리하지 않으며, 공격 탐지를 위해서 전달된 각 인자를 비교하고 일치 할 경우 하나의 인자만을 사용하여 실제 시스템 콜을 호출한다. 시스템 콜의 결과값은 래퍼가 각 데이터 메모리 영역으로 복사해준다.

시스템 콜 래퍼의 예로써 read 시스템 콜 래퍼를 살펴보면, 커널에 정의된 실제 read 시스템 콜은 지정된 파일 디스크립트(file descriptor)에서 지정된 최대 사이즈만큼의 입력을 받아들여서 사용자가 지정한 메모리 영역에 저장한다. 따라서 파일 디스크립터, 메모리 주소, 그리고 크기 이 세 개의 인자를 받아들인다. RETMAS가 사용하는 시스템 콜 래퍼는 각 인자를 원본 및 복사 변수에서 받아들이기 때문에 총 여섯 개의 인자를 전달받는다. 여섯개의 인자 중 파일 기술자인 파일 디스크립터 변수와 입력 데이터의 크기를 지정하는 변수는 원본 및 복사 변수의 값이 일치해야 하며, 사용자가 지정한 메모리 영역 주소를 가리키는 변수는 원본 및 복사 데이터 영역에서 그 주소가 항상 다르기 때문에 원본 및 복사 변수의 값이 반드시 서로 달라야 한다. 시스템 콜 래퍼는 이러한 조건을 검사한 후 공격이 탐지되지 않을 경우 원본 데이터 메모리 영역의 인자 만을 사용하여 운영체제의 커널 내부에 정의된 read 시스템 콜을 수행한 뒤, 원본 데이터 메모리 영역으로 전달된 입력 데이터를 지정된 복사 데이터 메모리 영역으로 복사해 준다. 시스템 콜은 read 외에도 다양하며, 각 시스템 콜의 역할이나 인자 처리 방법, 전달되는 인자의 성격이 모두 다르기 때문에 시스템 콜 래퍼는 반드시 모든 시스템 콜 개별적으로 작성되고 처리된다.

이처럼 RETMAS는 코드의 분석을 통해 코드 세그먼트를 수정하고, 제어 흐름이 변경되는 곳과 시스템 콜 래퍼에서 공격을 탐지하게 된다. 기본적으로 약 두 배 정도로 연산이 증가하기 때문에 일반 프로그램에 비해서 그 부하가 크다. 다음 4절에서는 RETMAS를 통해 다중 ASR이 적용된 프로그램의 실행 부하가 어느 정도 되는지 실험을 통해 살펴본다.

4. 실 험

RETMAS를 통해 다중 ASR이 적용된 프로그램이 그렇지 않은 프로그램에 비해 어느 정도의 부하를 보이는 지 SPEC CPU 2006을 통해 비교하였다. 실험 환경은 인텔 펜티엄 프로세서 3.0GHz, 2GByte 메인 메모리의 PC에 설치된 페도라 12 리눅스 배포판이다. (그림 3)은 SPEC CPU 2006을 통해 측정된 실험 결과이며, 각 프로그램의 크기 및 특징은 <표 6>에 기술되어 있다.



(그림 3) SPEC으로 측정한 다중 ASR 성능 부하

회색으로 표시된 normal은 다중 ASR이 적용되지 않은 프로그램의 실행 속도 측정 결과이며 검은색으로 표시된 retmas는 다중 ASR이 적용된 프로그램의 실행 속도 측정 결과이다. 다중 ASR이 적용된 프로그램은 기존 프로그램의 연산을 복사 메모리 영역에 동일하게 수행하기 때문에 평균 약 80%의 성능 부하를 보인다. 하지만 몇 가지 경우 더 적은 성능 부하를 보이게 된다. 우선 (그림 3)의 bzip2이나 gcc와 같이 I/O에 관련된 연산이 발생하는 경우, 다중 ASR은 I/O에 대한 처리는 한번만 수행하고 그 결과를 각 데이터 메모리 영역에 복사한다. 즉 I/O와 같은 시스템 콜의 처리는 시스템 콜 래퍼 내부의 인자 검사를 제외하였을 때 기존 프로그램과 동일하게 이루어진다. 따라서 I/O가 많이 발생하는 프로그램인 bzip2 벤치마크에서는 약 52%의 성능 부하를, gcc 벤치마크에서는 약 45%의 성능 부하를 보였다. libquantum과 같이 FPU(Floating Point Unit)에 관련된 명령어를 사용하는 경우, RETMAS는 FPU내부의 스택과 같은 자원을 직접적으로 제어할 수 없는 관계로 FPU 명령어에 대한 추가 생성은 하지 않고 그 결과를 I/O와 마찬가지로 각 데이터 메모리 영역에 복사한다. 이 경우는 I/O와 유사하게 한 번만 실행되기 때문에 libquantum 벤치마크는 약 39%의 성능 부하를 보였다. 그 외 다중 ASR에 의해 추가되는 코드는 기존의 프로그램 연산과 관계없는 메모리 영역 및 레지스터를 사용하기 때문에 데이터 의존(data dependency)가 없어서 CPU에서 파이프라이닝의 효율이 좋아지는 경우가 있다.

반면 (그림 3)에서 mcf, sjeng와 같은 벤치마크의 경우는 최적화가 충분히 이루어 지지 못하기 때문에 두 배 이상의 성능 부하를 보인다. 실험에 사용한 인텔 프로세스는 연산에 주로 네 개의 범용 레지스터를 사용하게 된다. RETMAS는 최적화를 위해서 각 데이터 메모리 영역에 수행되는 명령어들에 두 개 씩의 레지스터만 할당하여 레지스터의 대피와 복구를 최소화한다. 하지만 특정 명령어의 경우 반드시 특정 레지스터를 사용해야 하는 경우가 있으며,

<표 6> SPECINT 벤치마크 프로그램의 비교

이름	실행 파일 크기	변환시간(ms)	비 고
perlbench	1100KByte	8800	스크립터 언어 Perl
bzip2	66Kbyte	450	압축 프로그램
gcc	3200Kbyte	28160	C 컴파일러
mcf	17Kbyte	201	교통 네트워크 스케줄링 시뮬레이터
gobmk	3600Kbyte	30680	인공지능 게임 GO
hmmer	279Kbyte	2456	유전자 정보 검색 프로그램
sjeng	145Kbyte	1276	인공지능 체스 프로그램
libquantum	44Kbyte	370	물리 / 양자 컴퓨팅 시뮬레이터
h264ref	557KByte	4901	비디오 압축 프로그램

이때는 3.1절 <표 1>과 같이 반드시 해당 레지스터 값을 대피시킨 뒤 사용 후 복구해야 하는 과정을 거쳐야 한다. 이로 인하여 mcf 벤치마크는 약 126%, 그리고 sjeng 벤치마크는 약 103%의 성능 부하를 보인다.

RETMAS를 통해 다중 ASR이 적용된 프로그램의 경우 일반 프로그램에 비해 비교적 큰 성능 부하를 보이나 SPEC CPU 벤치마크는 CPU 연산을 많이 요구하는 프로그램으로 이루어져 있으며, 웹서버 데몬과 같이 실제 사용되는 프로그램은 I/O 요청 및 처리가 많이 일어나기 때문에 다중 ASR의 성능 부하가 벤치 마크 프로그램에 비해서 상대적으로 적다. 또한 RETMAS에서 최적화가 이루어질 경우 CPU의 파이프라이닝 등의 이유로 1.5배 이내의 성능 부하를 유지할 수 있다. 이는 대부분의 공격을 탐지하고 방어할 수 있다는 점을 고려하였을 경우 납득할 만한 수준의 부하라 할 수 있다.

5. 결 론

본 논문은 데이터 메모리 전반에 걸쳐 발생할 수 있는 프로그램 공격을 효과적으로 탐지하고 방어하기 위한 다중 ASR 기법을 제시하였다. 다중 ASR 기법은 원본 및 복사 데이터 메모리를 통해 제어 흐름 및 비 제어 흐름 공격 모두 탐지하고 방어할 수 있으며, 공격자가 각 데이터 메모리 영역의 동일한 변수를 동일한 값으로 조작할 수 없기 때문에 각 데이터 메모리의 변수 배치 순서와 정확한 주소가 외부에 노출되어도 방어 성능에 전혀 영향이 없다. 이를 통해 무차별 공격에 취약한 기존 ASR 기법의 단점을 제거하였으며 기존 ASR 기법과 마찬가지로 스택 스매싱, 힙 오버플로우, 포맷 스트링과 같은 다양한 공격을 방어할 수 있다. 다중 ASR을 프로그램에 적용하는 RETMAS는 목적 파일 또는 실행 파일을 대상으로 구현되었기 때문에 프로그램 개발 과정에서 다중 ASR의 적용이 필요 없으며, 실제 운용중인 프로그램을 직접적으로 변환할 수 있는 장점이 있다. 다중 ASR이 적용된 프로그램은 SPEC CPU 2006으로 평가해보았을 때 39~120% 가량의 성능 부하를 보이고 이 중 I/O 처리가 필요한 프로그램의 경우 40~50%의 성능 부하를 보인다. 웹서버와 같이 실제 환경에서 공격 대상이 되는 프로그램은 일반적으로 많은 I/O 처리가 요구되어 다중 ASR의 좋은 적용 대상이 된다.

참 고 문 헌

[1] Aleph One, "Smashing The Stack For Fun And Profit", Phrack, 7(49), 1996.

[2] Matt Conover, "w00w00 on Heap Overflows", http://www.w00w00.org/files/articles/heaptut.txt, 1999.

[3] Tim Newsham, "Format String Attacks", Bugtraqmailing list, http://www.securityfocus.com/archive/1/81565, 2000.

[4] Roger M. Needham, "Denial of service", Proceedings of the 1st ACM conference on Computer and communications security, pp.151~153, 1993.

[5] Stephane Racine, "Analysis of Internet Relay Chat Usage by DDoS Zombies", Master's thesis, Swiss Federal Institute of Technology Zurich, April, 2004.

[6] Sandeep Bhatkar, R. Sekar, Daniel and C. DuVarney, "Efficient Techniques for Comprehensive Protection from Memory Error Exploits", 14th USENIX Security Symposium, pp.271~286, 2005.

[7] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, PerryWagle and Qian Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks", 7th USENIX Security Symposium, pp.63~78, 1998.

[8] Crispin Cowan, Steve Beattie, John Johansen and Perry Wagle, "PointGuardTM : Protecting Pointers From Buffer Overflow Vulnerabilities", Proceedings of the 12th USENIX Security Symposium, pp.91~104, 2003.

[9] Michael Howard, Matt Miller, John Lambert and Matt Thomlinson, "Windows ISV Software Security Defenses", Microsoft Corporation, September, 2010.

[10] PaX team, "Documentation for the PaX project", http://pax.grsecurity.net/docs/

[11] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-jin Goh, Nagendra Modadugu and Dan Boneh, "On the Effectiveness of Address-Space Randomization", Proceedings of the 11th ACM Conference on Computer and Communications Security, pp.298~307, 2004.

[12] Randal E. Bryant and David O'Hallaron, 'Computer systems : A programmer's perspective', Prentice Hall, 2003.

[13] John R. Levine, 'Linkers and Loaders', Morgan Faufmann, 1999.

[14] Intel corporation, "Intel® 64 and IA-32 Architectures Software Developer's Manual", http://www.intel.com/Assets/PDF/manual/253665.pdf

[15] Andrew S. Tanenbaum, 'Modern operating systems', 3rd ED, Pearson education, 2009.

[16] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar and Ravishankar K. Iyer, "Non-Control-Data Attacks Are Realistic Threats", 14th USENIX Security Symposium, pp.177~192, 2005

박 수 현

e-mail : ardeness@gmail.com
2007년 홍익대학교 컴퓨터공학과(학사)
2009년 홍익대학교 컴퓨터공학과(석사)
2009년~현 재 홍익대학교 컴퓨터공학과 박사과정
관심분야 : 시스템 소프트웨어, 시스템 보안

김 선 일

e-mail : sikim@cs.hongik.ac.kr

1985년 서울대학교 컴퓨터공학과(학사)
1987년 서울대학교 컴퓨터공학과(석사)
1995년 University of Illinois at Urbana-Champagin(전산학박사)
1995년~1999년 미국 IBM 연구원
1999년~현 재 홍익대학교 컴퓨터공학과 교수
관심분야 : 시스템 소프트웨어, 임베디드 시스템, 시스템 보안