버퍼 오버플로우 공격에 대한 마이크로구조적 방어 및 복구 기법

(Microarchitectural Defense and Recovery Against Buffer Overflow Attacks)

최 린 † 신 용 †† 이 상 훈 ††
(Lynn Choi) (Yong Shin) (Sanghoon Lee)

요 약 버퍼 오버플로우 공격은 Code Red나 SQL Slammer와 같은 최근의 웜의 발발에서 알 수 있는 것과 같이 가장 강력하고 치명적인 형태의 악성 코드 공격이다. 버퍼 오버플로우 공격은 일반적으로 시스템에 비정상적인 증상들을 유발한다. 버퍼 오버플로우 공격에 대한 기존의 대처방안들은 심각한 성능 저하를 초래하거나, 다양한 형태의 버퍼 오버플로우 공격을 모두 방지하지 못했으며, 특히 일반적으로 사용되는 소프트웨어 패치를 사용하는 방법은 버퍼 오버플로우 웜의 확산을 효과적으로 차단하지 못한다. 이러한 문제를 해결하고자 본 논문에서는 적은 하드웨어 비용과 성능 저하만으로 거의 모든 악성 코드 공격을 탐지하고 피해를 복구할 수 있도록 하는 복귀 주소 포인터 스택 (Return Address Pointer Stack: RAPS) 과 변조 복구 버퍼 (Corruption Recovery Buffer: CRB)라는 마이크로 구조 기술들을 제안한다. 버퍼 오버플로우 공격으로 인한 비정상적인 증상들은 RAPS를 통해 프로세스 실행 중 메모리 참조의 안전성을 점검함으로써 쉽게 탐지될 수 있으며, 이는 그러한 공격들에 의한 잠재적인 데이타 혹은 제어 변조를 피하는 것을 가능하게 한다. 안전 점검 장치의 사용으로 인한 하드웨어 비용과 성능 손실은 거의 발생하지 않는다. 또한, RAPS에 비해 더욱 강도 높은 방법인 CRB를 이용하여 보안 수준을 더욱 향상시킬 수 있다. 변조 복구 버퍼는 안전 점검 장치와 결합되어 버퍼 오버플로우 공격에 의해 발생했을 가능성이 있는 의심스러운 쓰기들을 저장함으로써 공격이 탐지되는 경우 메모리의 상태를 공격 이전의 상태로 복구시킬 수 있다. SPEC CPU2000 벤치마크 중에서 선정한 프로그램들에 대해 상세한 시뮬레이션을 수행함으로써, 제안된 마이크로구조 기술들의 효율성을 평가할 수 있다. 실험 결과는 안전 점검 장치를 사용하여 공격으로 인한 복귀 주소 변조로부터 스택 영역을 방어하는 것이 시스템의 이상 증상들을 상당 부분 감소시킬 수 있다는 것을 보여준다. 또한, 1KB 크기의 작은 변조 복구 버퍼를 안전 점검 장치와 함께 사용할 경우 스택 스매싱 공격으로 인해 발생하는 추가적인 데이타 변조들까지 막아낼 수가 있는데, 이로 인한 성능 저하는 2% 미만에 불과하다.

키워드 : 버퍼 오버플로우, 보안, 컴퓨터 구조, 시뮬레이션

Abstract The buffer overflow attack is the single most dominant and lethal form of security exploits as evidenced by recent worm outbreaks such as Code Red and SQL Slammer. In this paper, we propose microarchitectural techniques that can detect and recover from such malicious code attacks. The idea is that the buffer overflow attacks usually exhibit abnormal behaviors in the system. This kind of unusual signs can be easily detected by checking the safety of memory references at runtime, avoiding the potential data or control corruptions made by such attacks. Both the hardware cost and the performance penalty of enforcing the safety guards are negligible. In addition, we propose a more aggressive technique called corruption recovery buffer (CRB), which can further increase the level of security. Combined with the safety guards, the CRB can be used to save suspicious writes made by an attack and can restore the original architecture state before the attack. By performing detailed execution-driven simulations on the programs selected from SPEC CPU2000 benchmark, we

† 종신회원 : 고려대학교 전자컴퓨터공학과 교수
lchoi@korea.ac.kr
†† 학생회원 : 고려대학교 전자컴퓨터공학과
sy3620@korea.ac.kr
smile97@korea.ac.kr
논문접수 : 2004년 12월 7일
심사완료 : 2005년 11월 25일

evaluate the effectiveness of the proposed microarchitectural techniques. Experimental data shows that enforcing a single safety guard can reduce the number of system failures substantially by protecting the stack against return address corruptions made by the attacks. Furthermore, a small 1KB CRB can nullify additional data corruptions made by stack smashing attacks with only less than 2% performance penalty.

Key words : Buffer Overflow, Security, Computer Architecture, Simulation

1. 서 론

1988년 Morris 웜의 출현 이후로, 인터넷의 급속한 성장과 함께 인터넷 상에서의 악성 코드 공격들은 양적인 면뿐만 아니라 그 종류도 증가하였다. 그림 1은 1988년 이후 발생한 보안 사고에 대한 CERT의 통계자료이다[1]. 그림에서 볼 수 있는 바와 같이, y축은 로그 스케일로 그려져 있는데 이는 그 수가 1988-2003년 사이에 기하급수적으로 늘어나고 있음을 의미한다. 이러한 악성 코드 공격들은 '취약점'이라고 불리는 프로그램 내의 결함을 통해 이루어지며, 이를 이용해 공격자들은 불법적인 접근 권한을 얻어 임의의 코드를 삽입하거나, 또는 서비스 거부 상태와 같은 불안정한 행위들을 유발하게 된다. 또한 그림 1은 매년 보고되는 취약점들의 수를 보여주고 있는데, 이 역시 빠른 증가 추세를 보이고 있다.

그러한 취약점들과 공격들 중에서, 버퍼 오버플로우 취약점 및 이를 이용한 공격들은 가장 심각한 보안 문제로 인식되고 있다. 이것은 여러 요인에 의한 것으로 첫째, 버퍼 오버플로우는 버퍼 인근의 데이타만 변조하는 것이 아니라, 프로그램의 제어를 가로채 악의적인 목적을 가진 임의의 코드를 실행하는 것을 가능하게 한다. 둘째로, 이러한 악성 코드들은 어떠한 수동 조작 없이도 스스로 복제되고 전파될 수 있기 때문에, 모든 형태의 악성 코드 공격들 중에서 가장 빠른 전파 속도를 가지고 있다. 한 예로, CAIDA의 보고에 따르면, SQL Slammer 웜은 약 8분 만에 전 세계 수십만 대의 패치가 되어있지 않은 SQL 서버들을 감염시킬 수 있었다[2]. 셋째로, 버퍼 오버플로우 공격은 악성 코드 공격의 여러 형태들 중에서 가장 널리 사용되고 있다. 지난 3년간 모든 보안 취약점들의 약 4분의 1가량[3], 그리고 SANS Institute와 FBI에 의해 선정된 가장 심각한 20

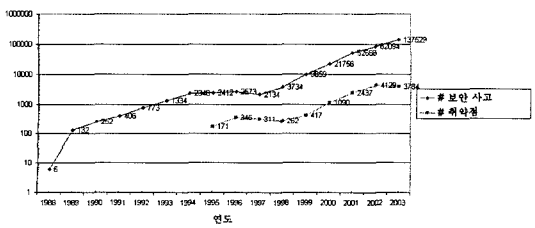

그림 1 보안 사고 및 취약점에 관한 CERT의 통계

가지 인터넷 보안 취약점들 중에서 14가지가 버퍼 오버플로우 취약점을 사용하였다[4]. 넷째로, 모든 악성 코드 공격들 중에서 가장 지속적인 형태라고 할 수 있다. 버퍼 오버플로우 공격은 1988년에 Robert Morris가 작성한 인터넷 웜 이후로 오랜 기간 알려져 왔지만, Code Red I/II, SQL Slammer와 W32/Blaster와 같은 최근의 웜의 발발에서 알 수 있듯이 계속해서 심각한 보안 위협을 제공하고 있다. 게다가, 버퍼 오버플로우 공격은 향후 20년간 계속해서 문제가 될 것으로 예측되고 있다[5].

이러한 버퍼 오버플로우 취약점을 해결하기 위하여 운영 체제의 수정[6], 컴파일러 도구[7-10], 디버깅 툴[11], 실시간 라이브러리[12] 등 여러 소프트웨어적인 해결 방안들이 다양한 형태로 제안되었지만, 이러한 기술들은 근본적으로 기존 응용 프로그램에는 속수무책이거나 또는 심각한 성능 장애를 유발시키는 문제점 때문에 실효를 거두지 못하고 있다. 여전히 대응책으로 가장 많이 사용되고 있는 방법은, 취약점이 발견된 부분의 소스 코드를 수정하고 이를 다시 컴파일 하여 패치 및 수정 사항들을 제작한 후, 이를 해당 프로그램의 사용자들로 하여금 직접 내려 받게 하는 것이다. 그러나 이는 취약한 소스가 공개적으로 알려진 후에만 적용 가능하며 특정 제품의 특정 취약성에만 효과적이기 때문에 근본적인 대책이라고 볼 수는 없다.

본 논문에서는, 버퍼 오버플로우 문제에 대한 새로운 하드웨어적인 해결 방안을 제안한다. 그러므로, 소프트웨어적인 해결 방안에서 발생하는 재 컴파일이나 경계 영역 검사 수행을 위한 별도의 코드 실행을 필요로 하지 않는다. 뿐만 아니라, 하드웨어 수준에서의 방어는 프로세서에서 실행중인 모든 시스템 및 응용 프로그램들에 적용할 수 있어서, 각 제품에 맞는 특정 패치를 일일이 설치해야 하는 기존의 방법에 비해 보다 안전하게 보호해줄 수 있다. 버퍼 오버플로우 공격은 일반 프로그램 실행에서는 발생하지 않는 이상한 증상이나 흔적을 나타내는 것이 일반적이다. 예를 들어, 스택 스매싱이라고 불리는 가장 흔한 형태의 버퍼 오버플로우 공격은 프로세스의 로컬 스택 프레임의 복귀 주소를 변경하게 되는데 이러한 동작은 일반적인 프로그램 수행 중에는 일어나지 않는다. 또한, 악성 코드들은 종종 프로그램 주소 공간의 스택 또는 데이타 영역으로부터 명령어를

가져오려 하는데, 이 역시 보통의 프로그램 수행에서는 잘 일어나지 않는 동작이다. 본 논문에서는 이러한 비정상적인 증상들을 검출하기 위해 명령어와 데이타 참조들의 주소 영역을 검사하여 그 안전성을 검증하는 '안전 점검 장치(safety guard)'라 불리는 새로운 마이크로구조 기술을 제안한다. 이러한 안전 점검 장치로 인한 성능 손실은 없으며 무시할 수 있는 아주 적은 하드웨어 비용으로 구현될 수 있다. 또한 본 논문에서는 이러한 안전 점검 장치에 추가하여, 의심스러운 메모리 쓰기들을 임시 저장하여 공격이 탐지될 경우 시스템을 공격 이전의 상태로 되돌릴 수 있는 '변조 복구 버퍼(Corruption recovery buffer; CRB)'라는 더욱 강화된 보안 기술을 제안한다.

SPEC CPU2000 벤치마크로부터 선정된 다수의 프로그램들에 대한 상세한 시뮬레이션을 수행함으로써, 제안된 마이크로구조 기술들의 효용성을 평가할 수 있었다. 본 실험에서는 시뮬레이션 도중에 프로그램 주소 공간의 다양한 데이타 영역들을 임의로 변조함으로써, 스택 스매싱 공격은 물론 실제 상황에서 아직 발생하지 않은 다양한 형태의 새로운 버퍼 오버플로우 시나리오들을 발생시켰다. 실험 결과에 따르면 안전 점검 장치 만을 사용하여 스택 스매싱 공격의 상당 부분을 감소시킬 수 있다는 것을 확인하였다. 또한, 1KB 크기의 작은 CRB를 안전 점검 장치와 함께 사용할 경우 2% 이하의 성능 저하만으로 스택 스매싱 공격으로부터 발생하는 모든 추가적인 데이타 변조들을 막아낼 수 있었다.

2장에서는 버퍼 오버플로우 공격이 어떻게 데이타를 변조시킬 수 있으며, 결국 실행중인 프로세스의 제어 흐름을 바꿀 수 있는지 논의하고 이 버퍼 오버플로우 공격과 관련된 기존의 보안 연구들을 소개하고 문제점을 논의한다. 3장에서는 이러한 버퍼 오버플로우 공격에 대한 해결책으로 먼저 아주 적은 비용으로 제어 변조를 막을 수 있는 안전 점검 장치를 소개하고 더 나아가 데이타 변조 역시 방어할 수 있는 변조 복구 버퍼라 불리는 새로운 하드웨어 기술을 소개한다. 4장에서는 먼저 다양한 제어 및 데이타 변조를 발생시키는 실험 환경을 설명하고 그러한 공격들로부터 제안된 안전 점검 장치와 CRB가 어떻게 효과적으로 시스템을 보호할 수 있는지 그 효용성을 평가한다. 마지막으로 5장은 논문을 요약하고 앞으로의 계획에 관해 논의한다.

2. 공격 패턴 분석 및 관련 연구

2.1 공격 패턴 분석: 데이타 및 제어 변조

버퍼 오버플로우 공격은 오버플로우 된 변수 근방의 데이타 변조에서 시작된다. 프로세스의 주소 공간에서 텍스트 영역은 쓰기가 금지되어 있기 때문에 스택, 힙(Heap), 혹은 정적 데이타와 같은 데이타 영역만이 변조 가능하다. 그림 2는 x86계열 프로세서에서 스택 스매싱 공격이 어떻게 복귀 주소를 변경할 수 있는가를 보여준다. strcpy와 같이 취약점이 있는 함수가 지역 변수에 입력을 읽어 들일 때 외부 입력은 함수의 스택 프레임 내에서 해당 변수뿐만 아니라 인근 영역의 데이타로 오버플로우 되며 이는 결과적으로 복귀 주소를 변조시킬 수 있다.

그림 3은 버퍼 오버플로우 공격에 의해 만들어진 제어 및 데이타 변조의 종류들을 분류한 것이다. 언제 그리고 어디에서 데이타가 변조되었는가에 따라 프로그램 실행 도중 에러 발생 여부가 결정된다. 변조된 데이타 항목이 그 이후로 전혀 참조되지 않는다면 데이타 변조는 프로그램 실행에 영향을 미치지 않을 것이다. 이하에서는 이런 경우를 '비활동성 데이타 변조' 라고 한다. 만일 같은 위치에서 더 이른 시점에 동일한 공격이 가해지면 그 데이타는 참조될 것이고 심각한 문제를 일으킬 수 있다. 그러므로 데이타 변조의 효과는 그 위치만이 아니라 공격의 시점에도 영향을 받게 된다. 변조된 데이타 항목이 나중에 참조되는 경우를 이하에서는 '활동성 데이타 변조'라고 언급할 것이다.

활동성 데이타 변조는 다음과 같이 3가지 종류로 분류될 수 있다. 첫 번째로, 변조된 데이타가 복귀(return) 명령 혹은 간접 분기(indirect branch) 명령과 같은 분기 명령어의 목적 주소(target)로 사용되는 경우 이를 '분기 주소 변조'라 부른다. 이런 경우 변경된 분기 목적

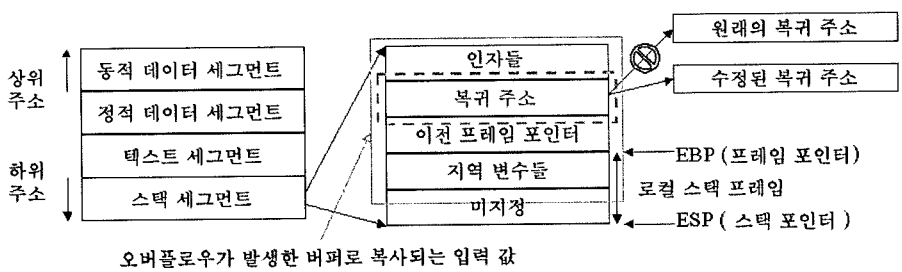

그림 2 메모리 주소 지도 및 x86 프로세서에서 스택 프레임의 구조

주소로부터 명령어를 가져올 경우 프로그램의 실행 흐름이 변경된다. 대부분의 버퍼 오버플로우 공격은 스택 프레임에 저장된 복귀 주소 혹은 스택 또는 힙 영역에서의 함수 포인터들을 목표로 하여 이와 같은 제어 데이타에 대한 변조를 통해 악성 코드 공격을 시도한다.

데이타 변조의 두 번째 유형은 공격이 분기 명령의 조건을 변경할 때에 발생한다. 이것 역시 복귀 주소나 함수 포인터를 변경하지 않더라도 프로그램 제어 흐름을 변경할 수 있다. 이하에서는 이와 같은 것을 '분기 조건 변조'라고 할 것이다. 예를 들어 아래와 같은 코드는 a의 값이 변경될 경우 제어흐름이 변경될 수 있다. 즉, 변조된 데이타 항목이 명령어 포인터를 변경하지 않고 제어흐름을 바꿀 수 있는 코드의 예이며 이러한 형태의 공격은 아직 보고된 바가 없다[1].

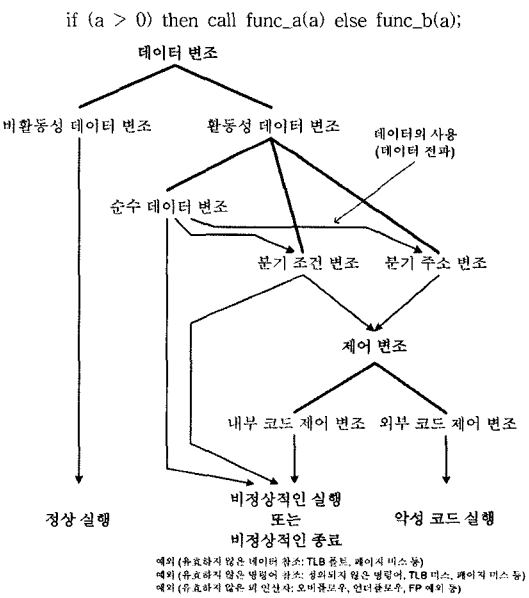

그림 3 버퍼 오버플로우 공격으로 인해 발생하는 데이타 및 제어 변조의 분류

이하에서는 활동성 데이타 변조의 다른 모든 경우를 '순수 데이타 변조'라고 할 것이다. 이 경우 제어흐름을 직접적으로 바꾸지는 않지만 이후 프로그램에 의해 참조될 활동성 데이타를 변경한다. 이런 경우 프로그램이 정상적으로 실행되는 것처럼 보이지만 잘못된 결과나 상태를 초래할 수 있다. 덧붙여 데이타 변조는 다른 데이타 위치들로 전파될 수 있으며, 그 결과 분기 주소 변조 혹은 분기 조건 변조로 이어질 수 있다.

상기의 활동성 데이타 변조의 3가지 경우 모두 비정상적인 실행 또는 종료를 일으킬 수 있다. 공격을 당한 프로세스가 잘못된 결과를 보여주며 실행을 마치게 되는 경우, 이를 '비정상적인 실행'이라 할 것이다. 이보다 자주, 공격을 당한 프로세스는 활동성 데이타 항목을 사용하여 제대로 완료되지 못하고 갑작스럽게 종료되곤 한다. 이러한 경우를 '비정상적인 종료'라고 할 것이다. 변조된 데이타 항목을 참조하는 것은 잘못된 피 연산자들에 의해 발생되는 에러와 관련된 데이타 참조 예외(Exception) 혹은 실행으로 이어지기 쉽다. 예를 들면 변조된 데이타 주소로부터 로드 하는 것은 TLB 미스(miss), 페이지 미스 혹은 접근 위반으로 이어질 수 있다. 비록 변조된 데이타 항목이 성공적으로 참조된다 하더라도 이후의 연산 과정에서 오버플로우나 부동 소수점 연산 예외 등을 일으킬 수 있다. 이러한 종류의 비정상적인 종료는 서비스 거부 상태를 유발하게 되는데, 이는 악성 코드의 실행보다는 덜 위험하지만 여전히 중요한 보안 문제라고 할 수 있다.

변조된 데이타 항목이 조건 분기의 분기 조건으로 혹은 간접 분기의 분기 주소로 사용되는 경우 제어 흐름을 변경시킨다. 특히 잘못된 목표 주소로부터 명령어를 가져오는 경우를 '제어 변조'라고 부른다. 이것은 버퍼 오버플로우 공격에 의해 발생한 잘못된 제어 흐름을 의미한다. 분기 목표가 외부에서 들어온 웜 코드에 대한 포인터로 변경될 경우 악성 코드 실행으로 이어질 수 있다. 이하에서는 이와 같은 제어 변조를 '외부 코드 제어 변조'라고 할 것이다. 악성 코드는 스스로 복제되어 다른 취약한 호스트들로 전파되기 때문에 버퍼 오버플로우 공격의 가장 심각한 결과라 할 수 있다. 또한 이것은 버퍼 오버플로우 공격의 가장 흔한 형태이기도 하다. 제어 변조의 또 다른 형태는 변경된 분기 목표가 텍스트 영역의 정상적인 코드를 가리킬 때이다. 이것을 '내부 코드 제어 변조'라고 부를 것이다. 이러한 형태는 버퍼 오버플로우 사고의 몇몇 경우들에서 보고된 바 있다[1].

2.2 관련 연구들

복귀 주소를 수정하고 악성 코드를 스택 세그먼트에 삽입하는 스택 스매싱은 가장 흔한 버퍼 오버플로우 공격 형태이다. 현존하는 공격들의 99% 이상이 이러한 형태의 스택 스매싱 공격을 이용하고 있다[1]. 그러므로, 대부분의 기존의 연구들이 이러한 형태의 공격에 초점을 맞추고 있다. 이러한 연구들은 구현 수준에 따라 소스 수준, 컴파일러 수준, 운영체제 수준과 아키텍처 수준의 네 가지 유형으로 분류해볼 수 있다.

2.2.1 소스 수준 방어 기법

컴파일러의 일반적인 정적 분석 방법이나 포인터 및 배열 경계 검사 방법을 사용하여 변수 경계 검사 코드를 자동으로 생성하거나 취약한 코드들의 위치를 정확

하게 알아낼 수 있다. 발견된 취약한 코드들은 수동적으로 혹은 자동적으로 패치가 될 수 있다. 배열 경계 검사의 여러 구현들이 이미 예전에 소개된 바 있다. 그 중에서, Richard Jones와 Paul Kelly[11,13]에 의해 제안된 GCC 패치 기법은 포인터 변경 없이 완전한 검사를 제공한다. 그래서, 이 패치를 이용하여 검사된 코드는 기존의 다른 기법들의 경우와는 달리 일반 코드들과의 호환이 가능하다. 이 기법은 각각의 포인터 관련 연산으로부터 기본 포인터를 이끌어낸 후, 기본 포인터의 저장 영역을 찾아내고 해당 연산의 결과가 이것과 동일한 저장 영역을 가리키는지를 검사함으로써 그 연산에 대한 검사를 수행하게 된다.

하지만, C 언어에서의 완전한 배열 경계 검사는 성능 오버헤드가 너무 커서 사용하기가 어렵다. 특히 포인터가 많이 사용된 프로그램의 경우에는 30배 정도의 성능 감소가 보고된 바 있다. 결과적으로 이러한 툴들은 단지 디버깅 시에만 사용이 될 뿐, 실제 배포 시에는 사용되지 않는다. 또한, 이러한 소스 수준의 방어 기법들은 소스 코드의 변경을 필요로 하는데 이는 언제나 가능한 일이 아니다.

2.2.2 컴파일 수준 방어 기법

Immunix Inc.에서 만든 GCC 패치인 StackGuard[8, 14, 15]는 다른 컴파일러 기반 방어 기술들의 토대를 제공하였으며 스택 canary를 이용하여 저장된 제어 정보들을 덮어쓰는 것을 방어하는 기법을 개척하였다. 이것은 컴파일러 확장 기법이며, 그림 4에서 보여지는 바와 같이 함수가 호출될 때 복귀 주소의 아래쪽에 보안 센서로서 canary word를 삽입하게 된다. 함수가 복귀하기 전에, 그 시점의 canary값과 그것의 원래 값을 비교해 봄으로써 복귀 주소의 변조 여부를 알아낼 수 있다. 버퍼 오버플로우 공격이 일어나는 동안에, canary 센서를 덮어쓰지 않고는 복귀 주소의 변조가 일어날 수 없다는 가정이 전제되어야 한다. 정밀한 경계 검사와 비교해볼 때, StackGuard는 성능 오버헤드가 적은 편이다.

StackShield[16,15]는 GNU C 컴파일러 확장 기법으로서 복귀 주소를 보호한다. 함수를 호출할 때, StackShield는 복귀 주소를 오버플로우가 불가능한 다른 영역에 복사하며, 함수가 복귀할 때 이 값을 이용하여 복귀 주소를 복구한다. 스택 내에 저장되어 있는 복귀 주소가 바뀌더라도, 원래의 복귀 주소는 다른 곳에 저장되어 있으므로 아무런 영향을 미치지 못한다. 또한 StackShield는 포인터의 값이 텍스트 영역을 가리키는 것만을 허용함으로써 함수 포인터에 대한 방어 기능도 제공한다. 이 방법은 단순하지만 임의로 삽입된 코드가 텍스트 영역 외의 다른 영역에 위치하게 되기 때문에 메모리를 동적으로 할당하는 프로그램들의 경우 이상을 초래할 수 있다. StackShield의 함수 포인터에 대한 방어는 텍스트 영역에 쓰여질 수 없는 악성 코드를 동작시키려는 시도들을 완전하게 차단하지만, StackGuard와 마찬가지로, 프로그램들은 다시 컴파일 되어야 하며, 여전히 서비스 거부 공격들에 대해 취약하다. 또한 스택 영역 이외의 다른 영역들에 대한 공격은 탐지할 수 없다.

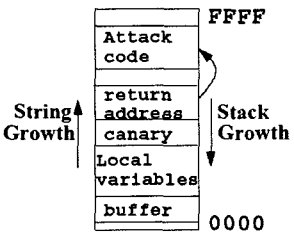

그림 4 스택 스매싱 공격에 대한 StackGuard의 방어

복귀 주소 방어(Return Address Defender: RAD)[7]는 동적 탐지를 수행하는 또 다른 컴파일러 패치 기법이다. 이 기법은 복귀 주소 값을 복귀 주소 저장소(Return Address Repository: RAR)라 불리는 분리된 영역에 저장하며, 복귀 명령어가 실행될 때, 스택 프레임의 복귀 주소를 함수 호출 시 저장해 놓은 값과 비교하게 된다. RAD는 MineZone RAD와 Read-Only RAD의 두 가지 방법으로 구현될 수 있는데, 이들은 RAR에 저장되어 있는 복귀 주소들을 각각 다른 방법으로 보호하게 된다. *MineZone* RAD가 보다 효율적인 방어를 수행하는 반면 Read-Only RAD는 보다 안전한 방어를 가능하게 한다.

PC-암호화[9]는 경계 검사 및 동적 탐지 어느 쪽도 필요로 하지 않는다. 함수가 호출될 때, 복귀 주소는 키를 이용하여 암호화된 후 스택 프레임에 저장된다. 함수가 복귀할 때, 암호화된 복귀 주소의 복호화가 이루어진다. 정상적인 실행이 이루어지려면 함수 호출 및 복귀 시 반드시 적절한 키를 이용하여 암호화 및 복호화가 이루어져야 한다. PC-암호화는 어떠한 동적 탐지도 수행하지 않기 때문에, 다른 기법들에 비해 성능 오버헤드가 덜한 편이다. PC-암호화는 하드웨어 또는 컴파일러 수준에서 구현될 수 있다. PC-암호화 기법은 복귀 주소 포인터뿐 아니라 함수 포인터를 보호하는 것으로도 확장될 수 있다[10]. 하지만, PC-암호화는 단지 제어 변조만 막아줄 뿐 여전히 모든 종류의 데이타 변조에 대해서는 취약하며 이는 서비스 거부 상태로 이어질 수 있다.

2.2.3 운영 체제 수준 방어 기법

Libsafe[12]는 보호를 필요로 하는 모든 프로세스와 함께 동적으로 사용 가능한 라이브러리로 구현되며, 프로그램 코드와 동적으로 사용 가능한 표준 C 라이브러리 함수들의 중간에 삽입된다. 그림 5에서 보여지는 바

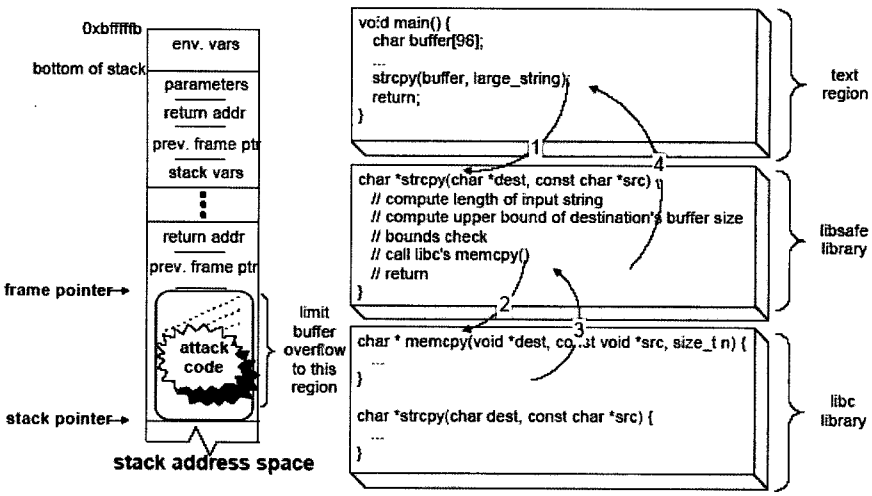

그림 5 버퍼 오버플로우에 대한 libsafe의 방어

와 같이 strcpy, gets, scanf, sprintf와 같은 취약한 표준 C 라이브러리 함수들에 대한 호출을 가로채어 그것을 동적으로 사용 가능한 라이브러리 내의 안전한 함수로 돌려주게 되는데, 이곳에서는 지역 버퍼의 크기가 현재 스택 프레임의 끝을 넘어가지 못하도록 한다. 라이브러리 및 그것을 사용하는 프로그램들을 다시 컴파일 하는 것은 필요하지 않다. 링커 및 동적 링커들은 바이너리 내에 탐지 메커니즘을 설치하는 실제적인 역할을 담당한다. Libsafe 라이브러리는 리눅스 상에서 구현되었다. 하지만 libsafe 기법을 사용하는 데에는 몇 가지 문제점이 따르게 된다. 첫째로 대상 버퍼의 정확한 크기를 계산하는 것이 불가능하다. 가장 좋은 방법은 버퍼의 크기를 버퍼의 시작 지점과 프레임 포인터의 차이와 동일하게 설정하는 것이다. 따라서 목적 버퍼 주변의 데이타 변조를 막을 수 없으므로 서비스 거부 공격에 취약하다. 또한 GCC로 컴파일 할 때 -fomit-frame-pointer 플래그를 사용한 프로그램들의 경우 스택 내에 프레임 포인터를 삽입하지 않고 최적화를 수행하므로 libsafe가 방어를 수행할 수 없게 된다.

대부분의 운영 체제에서, 스택 영역은 실행 가능하도록 설정되어 있다. 그러므로, 스택 영역을 실행 불가능하도록 설정하는 것은 스택 스매싱을 효과적으로 막아내는 방안이 될 수 있다[6]. 하지만, nested 함수와 같이 GNU C extensions와 몇몇 JIT 컴파일러 구현에서 사용되는 trampoline 함수들의 경우 원활한 동작을 위해 스택 영역이 실행 가능하도록 설정되어야 한다. 게다가, 리눅스의 시그널 핸들러는 일반적으로 실행 가능한 스택에 의존하고 있다. Openwall 프로젝트의 리눅스 커널 패치는 시그널 핸들러를 수정하고 trampoline 함수의 구현을 수정하여 이러한 문제들을 해결하려 하였다. 하지만 이러한 패치는 여전히 데이타 변조에 대해 취약하며 이로 인해 서비스 거부 상태로 이어질 수 있다.

2.2.4 아키텍처 수준 방어 기법

안전한 복귀 주소 스택(Secure Return Address Stack: SRAS)[17,18]는 버퍼 오버플로우 공격을 감지하는 RAS 확장 기법이다. 정확한 실행을 위해 SRAS는 RAS와 달리 commit 단계에서 구현되어야 하며, 제한된 크기로 인한 오버플로우 및 언더플로우를 방지하기 위해, SRAS와 메모리간의 데이타 교환이 필요하다. SRAS가 가득 차는 경우, 가지고 있는 데이타들을 메모리로 보내게 되며, 언더플로우가 발생하는 경우에는 이전에 메모리에 저장해 두었던 데이타들을 다시 SRAS로 가져오게 된다. 이외에도 setjmp, longjmp와 같은 non-LIFO 동작들을 해결하기 위해서 ISA의 수정을 필요로 한다. 이러한 이상의 세 가지 기법들이 SRAS에 모두 적용되는 경우, 버퍼 오버플로우가 발생하는 경우에만 잘못된 예측(miss-prediction)이 발생하게 된다. 하지만 SRAS 역시 공격을 당한 함수가 복귀할 때까지 버퍼 오버플로우를 감지할 수 없으므로 서비스 거부 상태에 대해 취약하다. 또한 ISA의 수정을 필요로 하므로 라이브러리들 및 프로그램들을 다시 컴파일 해야 한다.

이상의 여러 기법들은 '예방 기법'과 '회피 기법'의 두 유형으로 나누어질 수 있다. 실행이 불가능한 스택 및 정적인 분석 기법은 둘 다 취약점들을 제거하고 어떠한 데이타 혹은 제어 변조가 일어나기 전에 가능한 공격들에 대해 미리 방어를 취한다는 점에서 '예방 기법'이라 할 수 있다. 특히, 실행이 불가능한 스택은 제어 변조에 대한 예방 기법이라 할 수 있는 반면, 컴파일시의 분석

표 1 관련 연구들의 비교

	구현 수준				재컴파일	서비스 거부 공격에 대한 방어	변조 데이타 복구
	소스 코드	컴파일러	운영체제	하드웨어			
배열 경계 검사	예	아니오	아니오	아니오	필요함	가능	불가능
StackGuard	아니오	예	아니오	아니오	필요함	불가능	불가능
StackShield	아니오	예	아니오	아니오	필요함	불가능	불가능
RAD	아니오	예	아니오	아니오	필요함	불가능	불가능
PC encoding	아니오	예	아니오	아니오	필요함	불가능	불가능
Libsafe	아니오	아니오	예	아니오	불필요함	불가능	불가능
Non-executable stack	아니오	아니오	예	아니오	불필요함	불가능	불가능
SRAS	아니오	아니오	예	예	필요함	불가능	불가능
Safety Guard with CRB	아니오	아니오	아니오	예	불필요함	가능	가능

과 변환 기법은 데이타 변조에 대한 예방 기법이라 할 수 있다. 다른 방법들은 '회피 기법'을 사용하는데, 이는 버퍼 오버플로우가 발생하는 것을 허용하지만 제어 변조로 이어지기 전에 그것을 탐지하고 실시간에 회피하는 방법이다. 그러므로 실시간에 공격의 가능성을 탐지하기 위해, 이 기법에는 다소의 실행 오버헤드가 따르게 되며 종종 실시간 해결 방안이라 불린다.

위에 소개된 기법들 또는 다른 동적인 기법들의 효용성에도 불구하고, 버퍼 오버플로우 공격은 여전히 영구한 과제로 남아있다. 이것은 주로 실제적인 해결 방안이 존재하지 않기 때문이다. 소스 코드를 다시 컴파일 하는 것은 불가능한 경우가 많은데 이는 소스 코드가 기존의 바이너리 코드들과는 달리 언제나 사용 가능하지는 않기 때문이다. 게다가, 보안에 대한 인식이 부족한 어떠한 프로그래머들도 손쉽게 버퍼 오버플로우 취약점들이 존재하는 프로그램을 개발할 수 있으므로, 모든 취약한 코드들을 다시 컴파일 하는 것은 불가능한 일인지도 모른다. 안전한 커널 패치 또는 실시간 라이브러리들은 유사한 실제적인 문제점들로 인해 제약을 받게 된다. 현재 존재하는 기법들의 또 다른 중요한 문제점은 대부분의 기법들이 복귀 주소의 수정을 금지하고 그로 인한 제어 변조를 막아내는 것에 초점이 맞추어져 있다는 것이다. 이러한 이전의 기법들은 모든 다른 종류의 데이타 변조를 막아낼 수가 없다. 이는 이전의 기법들을 이용하는 시스템들이 여전히 서비스 거부 상태에 대해 취약하다는 것을 의미한다. 이 논문에서는 버퍼 오버플로우 공격으로 인한 데이타 및 제어 변조들을 탐지하고 복구할 수 있는 하드웨어 기법들을 이용함으로써 이러한 문제점들을 다루게 될 것이다.

3. 마이크로구조적 방어

3.1 안전 점검 장치

공격을 당하는 시스템은 데이타와 명령어 참조 시 비정상적인 증상을 보인다. 예를 들면 스택 스매싱 공격은 스택 내부의 지역 변수들 외에 복귀 주소도 덮어쓴다. 게다가 종종 현재의 스택 프레임 밖의 스택 영역으로 자신에게 동반된 악성 코드를 복사하는데, 일반적인 프로그램 실행 중에는 이러한 일은 발생하지 않는다. 또한 공격을 당한 프로그램은 스택 영역으로부터 명령어들을 가져오려 한다. 리눅스의 시그널 핸들러 혹은 gcc의 trampolines 함수들과 같은 아주 드문 경우들 이외에는 스택으로부터 명령어를 가져오는 것은 흔한 일이 아니다. 비정상적인 명령어 또는 비정상적인 데이타 참조는 참조되고 있는 주소에 대한 안전 검사를 통해 실행 도중 쉽게 탐지할 수 있다.

그림 6은 프로세서가 스택 스매싱 공격으로 인해 발생하는 가능한 데이타 혹은 제어 변조들에 대항해서 어떻게 시스템을 방어할 수 있는지를 보여준다. 우선, 스택 스매싱 공격들로 인해 발생하는 외부 제어 변조를 피하기 위해, 프로세서는 프로그램 카운터의 값을 조사할 수 있다. 만일 프로그램 카운터가 스택 내의 한 지점을 가리킨다면, 그 명령어 참조는 안전하지 못한 것으로 판단되어 버려지게 된다. 이러한 방법은 하드웨어 수준의 기법이라는 차이는 있지만 제어 변조를 피할 수 있게 한다는 점에서 실행이 불가능한 스택[6] 및 PC-암호화[9] 기법과 필수적으로 유사한 효과를 가지게 된다. 하지만, 이러한 방법은 gcc trampoline 함수들 또는 리눅스 시그널 핸들러와 같이 스택으로부터 명령어를 가져오는 경우 문제가 될 수 있다. 버퍼 오버플로우를 발생시키는 악성 코드들은 많은 경우 복귀 주소를 덮어쓰기 때문에, 스택 영역으로 제어를 바꾸는 명령어는 복귀 명령어가 된다. 하지만, trampolines는 복귀 명령어가 아닌 호출(call) 명령어에 의해 불려진다. 그러므로 복귀 명령어의 목표 주소가 스택 영역을 가리키는지를 검사함으로써, trampolines 함수들로 인한 정상적인 명령어들과 악성 코드 공격으로 인한 비정상적인 명령어들을 구분할 수 있다. 명령어를 실행하는 동안의 이러한

(a) 명령어 참조 안전 점검 장치:
외부 코드 제어 변조에 대한 방어

복귀 명령어의 목표 주소가 스택을 가리키는 경우, 복귀 명령어는 잘못된 것으로 판단된다.

(b) 데이터 참조 안전 점검 장치:
복귀 주소 (분기 목표) 변조에 대한 방어

스택으로의 저장 연산의 연속적인 실행이 복귀 주소를 수정하는 경우, 복귀 주소의 수정은 잘못된 것으로 판단된다.

그림 6 안전 점검 장치 및 그 동작 방법

종류의 안전 검사를 '안전 점검 장치'라고 하며 특히 복귀 명령어에 대한 참조 안전 검사를 '명령어 참조 안전 점검 장치'라고 부른다. 그림 6(a)는 명령어 참조 안전 점검 장치 및 외부 코드 제어 변조에 대한 그 방어를 보여준다. 이것은 명령어 참조 주소에 대한 간단한 범위 체크로서 포인터가 가리키는 주소에 대한 크기 비교등과 같은 방법을 사용함으로써 하드웨어 비용 또는 성능에 큰 손실 없이 하드웨어로 구현될 수 있다. 외부로부터 들어온 악성 코드는 프로그램의 주소 공간 중 대부분의 경우 스택 영역에만 존재할 수 있기 때문에 명령어 참조 안전 점검 장치는 버퍼 오버플로우 공격에 의해 만들어진 외부 코드 제어 변조를 효과적으로 제거할 수 있다. 그러나 이러한 방법은 텍스트 영역으로부터 명령어들을 가져오는 내부 코드 제어 변조는 막아내지 못한다. 또한 이것은 분기 조건이 아닌 분기 목표 주소만을 검사하므로 분기 조건 변조에 의해 발생되는 제어 변조 역시 막아낼 수가 없다.

가급적 초기 단계에 공격을 탐지하여 막는 것이 언제나 더 나은 방법이라 할 수 있다. 그러므로 데이타 변조 단계에서 시스템을 방어하는 것이 제어 변조 후에 시스템을 방어하는 것보다 시스템에 가해지는 피해를 줄일 수 있다. 그림 6(b)는 데이타 변조 단계에서 제공될 수 있는 또 다른 안전 점검 방안을 보여준다. 복귀 주소가 저장된 곳의 위치에 대해 연속적인 저장 명령어들의 주소 영역을 검사함으로써 복귀 주소 혹은 프레임 포인터의 변조를 방지할 수 있다. 이를 '데이타 참조 안전 점검 장치'라고 한다. 복귀 주소가 수정되는 것을 방지함으로써 스택 스매싱 공격에 의해 제어 변조가 일어나는 것을 원천적으로 막을 수 있다. 이러한 방법은 하드웨어 수준의 기법이라는 차이는 있지만 복귀 주소의 데이타 변조를 피할 수 있게 한다는 점에서 StackGuard, StackShield, RAD, SRAS와 필수적으로 유사한 효과를 가지게 된다.

이 안전 점검 장치를 사용하기 위해서는, 복귀 주소들이 저장되는 곳의 주소들을 알고 있어야 한다. 호출 명령어의 의미로 인해 복귀 주소 레지스터로서 특별한 레지스터가 제공되기 때문에, 하드웨어는 복귀 주소들이 스택 내의 어느 위치에 저장되어 있는지 알 수 있다. 이러한 복귀 주소를 저장하는 공간을 본 논문에서는 '복귀 주소 포인터 스택(Return Address Pointer Stack: RAPS)'이라 명명한다.

그림 7은 RAPS의 구조를 보여준다. RAPS는 복귀 주소들이 저장되어 있는 스택 내의 위치 정보들을 가지게 된다. 복귀 주소가 스택에 저장될 때마다, 복귀 주소의 저장 위치가 RAPS에 저장(push) 된다. 복귀 주소가 스택으로부터 읽혀질 때, RAPS에서 팝(pop) 연산이 일어난다. RAPS에서 Top은 이전 스택 프레임의 복귀 주소를 가리키며 Bottom은 아래에서 두 번째의 스택 프레임에 저장된 복귀 주소를 가리킨다. x86 프로세서에서 setjmp, longjmp와 같은 함수가 호출될 경우에는 아

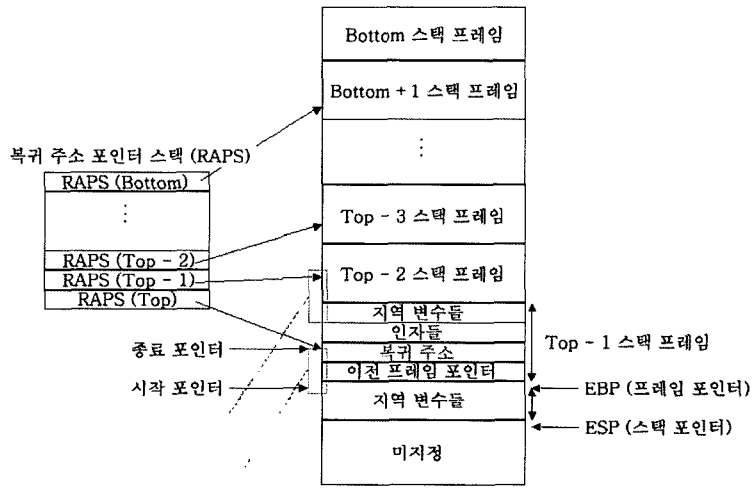

그림 7 복귀 주소 포인터 스택(Return Address Pointer Stack: RAPS)의 구조

래 수식과 같이 여러 번의 팝(pop) 연산이 한번에 일어날 수 있다. setjmp 또는 longjmp가 호출되는 경우의 RAPS의 동작은 아래와 같다.

while(RAPS (top) < ESP) pop RAPS;

RAPS를 이용하여 데이타 참조 안전 점검 장치는 다음과 같이 동작한다. 두 개의 연속적인 저장 연산이 스택 내의 낮은 주소에서 높은 주소 방향으로 이어지는 연속적인 영역에 일어날 때, RAPS 는 그림 7에서 보이는 바와 같이 시작 지점과 끝 지점을 기억하는 두 개의 포인터(각각 시작 포인터와 종료 포인터)를 이용해 이 범위를 기억하게 된다. 만일 연속적인 저장 범위가 RAPS의 Top에 저장된 주소를 침범하게 되면(즉 연속적인 저장 범위와 RAPS에 저장된 주소 사이에 충돌이 발생하면) 데이타 참조 안전 점검 장치는 안전 위반을 알리게 된다.

대부분의 버퍼 오버플로우 공격은 가장 바깥쪽의 스택 프레임 부근의 복귀 주소를 변조시킨다. 만일 오버플로우된 변수가 현재 스택 프레임 내의 지역 변수인 경우, 그림 7에서 보이는 바와 같이 바로 이전 스택 프레임 내의 복귀 주소가 악성 코드에 의해 변조된다. 하지만 오버플로우된 변수가 포인터에 의한 호출(call by reference)로 전달된 인자인 경우, 실제 버퍼 오버플로우는 이전 스택 프레임에서 발생하게 되고, 그림 7에서 보이는 바와 같이 오버플로우가 발생한 스택 프레임 이전의 스택 프레임 내에 있는 복귀 주소가 변조되게 된다. 즉, 오버플로우가 일어나는 변수가 참조에 의한 호출 방식으로 전달될 경우, 어떠한 스택 프레임에서도 버퍼 오버플로우 공격이 발생할 수 있다. 그러므로 RAPS 내의 모든 값들에 대해 연속적인 저장의 범위가 검사되어야 하는데, 이럴 경우 RAPS의 크기에 따라 비용이 커질 수 있다. 하지만 대부분의 공격들은 가장 최근의 스택 프레임 부근의 복귀 주소 즉, 현재 실행 중인 프로세스가 끝난 뒤 복귀할 주소를 목표로 하는데, 이는 그렇게 하는 것이 코드를 삽입하는 것도 쉬울 뿐 아니라 대부분의 취약한 입력 변수들이 지역 변수이거나 참조에 의한 호출 방식을 통해 이전 스택 프레임으로부터 전달된 인자들이기 때문이다. 그러므로, 몇 개의 최근 복귀 주소만 검사하는 것으로 대부분의 공격을 감지할 수 있으며 이는 하드웨어 비용을 고려했을 때, 적은 하드웨어 비용으로 공격 감지를 할 수 있는 충분히 좋은 방법이 될 수 있다. 요약하면, 데이타 참조 안전 점검 장치가 구현되는 경우 하드웨어 비용 면에서는 명령어 참조 안전 점검에 비해 더 비싸지만, 제어 변조는 물론 데이타 변조에 대해서는 더 높은 수준의 방어를 제공하게 된다.

3.2 복구 기법

프로세서가 제안된 안전 점검 장치들에 의해 위험한 메모리 참조들을 탐지하게 되면 몇 가지 대응 방법을 선택할 수 있다. 첫 번째, 공격을 당하는 프로세스를 종료하는 것이다. 그러나 이것은 공격자가 프로세스를 끝내는데 성공하는 것이므로 서비스 거부 상태를 발생시킬 수 있다. 프로세스 풀링(process pooling) 방식을 사용하여 사용자들의 요청을 처리하는 웹 서버들의 경우, 요청을 처리중인 프로세스가 갑작스럽게 종료되어도 아무런 문제가 되지 않지만, 스레드 풀링(thread pooling) 방식을 사용하는 웹 서버들의 경우, 이러한 대응 방법은 더 이상의 정상적인 실행을 어렵게 한다. 두 번째 선택은 현재 공격 당하는 함수의 실행을 중단하고 그 제어 흐름을 강제로 호출자 함수로 돌리는 것이다. 이것을 '강제 복귀(compulsory return)'라고 한다. 이것은 RAPS를 통해 복귀 주소의 위치를 추적하여 강제로 PC 값을 복귀 주소로 바꿈으로써 가능하다. 세 번째는 모든 안전하지 않은 저장 연산들을 무시하고, 원래의 실행 흐름은 변화시키지 않는 방법이다. 이것을 '조용한 복구(calm recovery)'라고 한다. 이 방법은 원래의 실행 흐름은 바꾸지 않지만, 지역 변수들에 대한 데이타 변조들로 인해 비정상적인 실행을 유발할 수 있다. 마지막 방법은 공격 이전의 상태로 복구시킨 후 호출자에게 복귀시키는 것이다. 그러나 이것은 이전 또는 새로운 구조적인 상태를 저장하기 위한 하드웨어적인 저장 수단 혹은 버퍼를 필요로 한다.

3.2.1 강제 복귀

간단한 하드웨어 해결책으로, 공격당한 함수를 끝내고 그 제어의 흐름이 호출자에게 복귀되도록 강제로 변경한다. 이는 공격이 감지된 경우 스택에 저장된 복귀 주소의 값을 이용하여 프로그램 카운터를 갱신하고 현재의 스택 프레임을 팝업(pop up)함으로써 구현될 수 있다. 많은 경우 공격당한 함수의 목적은 단지 외부의 입력을 지역 변수로 복사하는 것이다. 그러므로 오버플로우가 발생한 변수는 악성 코드만을 포함하고 있으므로 오버플로우가 발생했을 때 대부분의 경우 외부 입력을 지역변수로 저장하지 않는 것이 안전하다.

3.2.2 조용한 복구

또 다른 해결 방안으로, 오버플로우가 발생 가능한 모든 쓰기들을 단순히 무시하는 방법이 있다. 데이타 참조 안전 점검 장치는 복귀 주소를 보호하기 때문에, 공격은 제어를 가로챌 수 없으며, 공격을 당한 프로세스는 공격 이후에 정상적인 실행을 계속하게 될 것이다. 즉, 프로세서는 RAPS를 통해 공격이 감지된 이후의 모든 쓰기들을 무시하여 오버플로우에 의한 제어 변조를 방지한다. 이 때 프로세서에서 공격 당한 프로세스는 원래의 제어 흐름을 바꾸지 않으므로 이것을 '조용한 복구(calm

recovery)'라고 부른다. 이처럼 조용한 복구는 프로세스의 제어 흐름을 강제적으로 변화시키는 강제 복귀 방법에 비해 덜 강제적이다. 하지만 이 방법은 데이타 변조에 취약하며, 데이타 참조 안전 점검 장치가 안전 위반을 알리기 이전에 스택 프레임 내의 지역 변수들이 이미 변조되므로 비정상적인 실행을 유발할 수 있다.

3.2.3 변조 복구 버퍼(Corruption Recovery Buffer: CRB)

앞의 두 방법에 비해 더 적극적인 방법은 공격이 탐지되었을 때 원래의 버퍼 상태를 복구하는 것이다. 이를 위해, 변조 복구 버퍼(CRB)라고 불리는 특별한 하드웨어 버퍼에 안전하지 않은 - 오버플로우가 발생 가능한 - 쓰기들을 저장한다. 데이타 참조 안전 점검 장치 위반이 발생했을 때, 프로세서는 CRB내의 모든 안전하지 않은 값들을 버리고, 조용히 그 이후의 안전하지 않은 쓰기들을 무시한다. 그리고 그 이후 이어지는 모든 연산들을 조용한 복구(calm recovery)에서와 같이 수행하게 된다. 이렇게 함으로써, 메모리의 입력 버퍼는 원래 상태를 유지하게 되며, 어떠한 종류의 스택 영역에 대한 데이타 변조도 피할 수 있게 된다.

그림 8은 FIFO로 구성된 변조 복구 버퍼(CRB)의 구조를 보여준다. CRB는 버퍼 오버플로우 가능성이 있는 의심스러운 쓰기들을 저장하는데 사용된다. 이 의심스러운 쓰기들이 안전한 것으로 판명된 이후 이 값들은 메모리에 전달된다. 그러므로 메모리는 공격 이전의 버퍼의 원래 상태를 항상 유지하게 된다. 버퍼 오버플로우 공격은 메모리의 연속적인 영역에 계속적인 쓰기들을 실행하는 것을 동반한다. 단일 쓰기를 통해서 버퍼 오버플로우는 발생하지 않으므로 연속적인 메모리 영역에 두 번의 계속적인 쓰기가 발생하는 경우를 의심스러운 쓰기의 시작으로 간주한다. 그러므로 두 개의 계속적인 저장 연산만이 아니라 메모리의 연속적인 영역들도 의심스러운 공격의 성립을 위한 필수적인 조건이다. 이하에서는 이와 같은 연속적인 메모리 영역의 두 번째 쓰기를 'CRB 트리거'라고 언급할 것이다. 이러한 연속적인 쓰기들이 RAPS에 의해 유지되고 있는 복귀 주소들 중 하나를 덮어쓰려 할 때, 데이타 참조 안전 점검 장치 위반이 발생하며 CRB에 임시로 저장되어 있던 의심스러운 쓰기들은 공격으로 간주되어 버려지게 된다.

CRB 트리거가 하드웨어에 의해 탐지되면, 다음의 계속적인 쓰기들이 메모리 대신 CRB에 쓰여진다. 만일 이 계속적인 쓰기들이 실행되는 중에 데이타 참조 안전 점검 장치 위반이 발생하지 않는다면, CRB에 저장된 그 계속적인 기록들은 안전한 것으로 여겨지며 메모리로 전달된다. 데이타 참조 안전 점검 장치에 대한 위반이 발생하면 CRB에 저장된 값들은 안전하지 않으며 그 이후의 기록들은 무시될 것이다. 또한 조용한 복구(calm recovery)에서와 마찬가지로 프로세서는 모든 안전하지 않은 쓰기들을 무시한 후, 정상 실행을 계속할 것이다. 이러한 버퍼 오버플로우 공격이 진행되는 동안 실제로 메모리에는 쓰기 연산이 발생하지 않으므로 데이타도 제어도 변조되지 않는다.

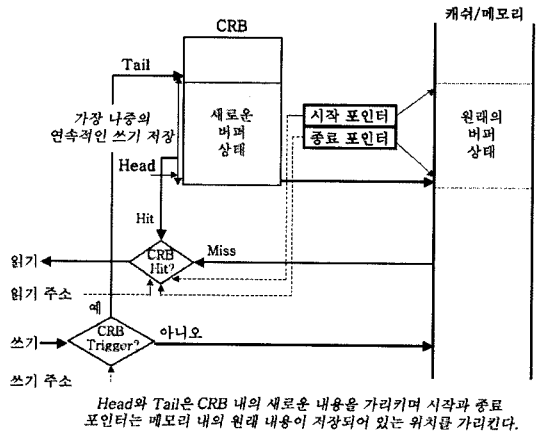

그림 8 CRB의 구조

하지만, CRB를 추가하는 것은 읽기 연산을 복잡하게 한다. 최신의 데이타가 아직 메모리에 존재하지 않을 수 있으므로 읽기 연산은 CRB와 메모리, 즉 캐쉬를 동시에 접근해야 한다. 만일 읽기 연산의 주소가 CRB에 쓰여진 데이타에 있으나 아직 메모리로 전달되지 않았다면 CRB가 그 데이타를 공급해야 한다. 이는 CRB에 의해 유지되는 시작과 끝 주소를 읽기 연산의 주소와 비교하여 간단하게 검사될 수 있다. 그림 8에서 보이는 바와 같이 헤드(Head)와 테일(Tail)은 각각 CRB에서 가장 오래된 기록과 가장 최신의 기록을 가리킨다. CRB의 제한된 크기로 인하여 프로세서 파이프라인의 지연이 발생할 수 있다. 만일 CRB가 가득 차게 되면 의심스러운 쓰기들은 CRB의 일부 데이타가 메모리로 저장될 때까지 지연되어야 한다. CRB가 작을수록 지연은 더 길게 더 자주 발생한다. 또한, CRB가 작을 경우 CRB가 가득 찰 확률이 높아지므로 보안의 수준도 떨어지게 될 것이다. 이는 CRB가 가득 차게 되면, 그곳에 저장되어 있던 이전의 안전하지 않은 쓰기들은 메모리로 저장되어야 하기 때문이다. 이럴 경우 데이타 변조가 발생할 수 있다. 하지만 4장에서 보이는 바와 같이 1KB의 작은 CRB는 2% 이하의 성능 감소로 모든 제어 및 데이타 변조를 제거하는데 충분하다는 것을 알 수 있다.

4. 실험 및 결과

본 논문의 실험에서는 다음의 두 가지 목표를 설정하

였다. 첫째, 버퍼 오버플로우 공격들의 여러 시나리오들에 대해서 알아보려 한다. 특히, 어떻게 공격이 여러 데이타 및 제어 변조들을 일으키고, 결국 잘못된 프로그램 행위를 유발하게 되는지 분석해 보고자 한다. 두 번째 목표는 제안된 마이크로구조 기법들이 얼마나 효율적으로 동작하는지 알아보는 것이다. 버퍼 오버플로우 시나리오의 다양한 상황들을 만들어내기 위해, 응용 프로그램의 스택 또는 데이타 영역에 임의로 쓰레기 값들을 삽입한다. 이를 위해 SPEC CPU2000 벤치마크 중에서 선정한 다섯 개의 응용 프로그램들을 실험에 사용하였다. 실험에 사용된 프로그램은 압축 프로그램인 gzip과 bzip2, 오브젝트 기반 데이타베이스인 vortex, 지진 모델링 프로그램인 equake그리고, 최소 비용 네트워크 흐름 분석기인 mcf이다. 차후 본 논문의 실험을 CPU2000 벤치마크에 포함되어 있는 모든 응용 프로그램들로 확대해 나가려 한다. 본 실험 방법은 이미 보고된 공격은 물론 아직 실제로 보고되지 않은 발생 가능한 모든 공격 시나리오들을 생성할 수 있다. 이는 외부 코드 변조를 유발하는 스택 스매싱 공격이 주를 이루는 실제 상황보다 훨씬 가혹한 환경을 의미한다.

정확한 시뮬레이션을 위해서, SimpleScalar 2.0 tool-set을 사용하였으며 이를 이용해 기본 캐시/TLB를 가지며 4개의 비순차적인(out-of-order) 파이프라인을 가지는 프로세서 모델을 가정하였다. 각 응용 프로그램마다 50번의 독립적인 시뮬레이션을 수행하였으며, 각 시뮬레이션은 256바이트 크기의 쓰레기 값을 갖는 입력 데이타를 실행 도중에 스택에 삽입하는 단일 공격을 동반한다. 총 50번의 공격이 각 응용 프로그램마다 가해지며, 각각의 공격은 모두 다른 시점에 일어나게 된다.

실험 결과를 명확하게 나타내기 위해, 다음의 용어들을 정의한다.

1. 고장: 공격이 성공적일 경우, 이를 고장이라 부를 것이다. 이는 성공적인 공격이 시스템의 고장을 유발하기 때문이다. 비활동성 데이타 변조를 제외한 모든 데이타 또는 제어 변조들은 고장이라 간주되는데, 이는 공격을 당한 프로세스가 비정상적인 제어 변조 또는 변조된 데이타를 사용하게 되기 때문이다.
2. 고장율: 총 공격 횟수 중 고장이 발생하게 되는 비율
3. 고장 유형: 공격에 의해 만들어지는 제어 혹은 데이타 변조들의 유형
4. 입력: 공격 시에 사용되는 쓰레기 데이타. 입력의 90%는 외부 코드 제어 변조를 유발하도록 설정되었으며 나머지 10%는 내부 코드 제어 변조를 유발시키도록 설정되었다. 외부 코드 제어 변조는 스택 스매싱 공격을 통해 복귀 주소를 수정하여 스택 내의 한 지점을 가리키게 하며, 내부 코드 제어 변조는 복귀 주소가 텍스트 영역을 가리키도록 한다.

4.1 시스템 고장 분석: 데이타 및 제어 변조의 분류

그림 9(a)는 각각의 응용 프로그램들에 대해 스택 스매싱 공격이 유발하는 고장율 및 고장 유형의 분포를 보여준다. 그림에서 볼 수 있는 바와 같이, 평균 고장율은 76.0%인데, 이는 네 번의 공격 중 한 번의 공격만이 비활동성 데이타 변조를 일으킨다는 것을 의미한다. vortex나 equake의 경우 모든 스택 스매싱 공격이 고장을 유발하는 반면, mcf의 경우에는 12%에 그치는 것을 알 수 있다. 이러한 시스템 고장 중에서, 80%는 제어 변조로 인해 발생하였다. vortex와 equake를 제외하고는, 스택 스매싱 공격으로 인한 대부분의 이러한 제어

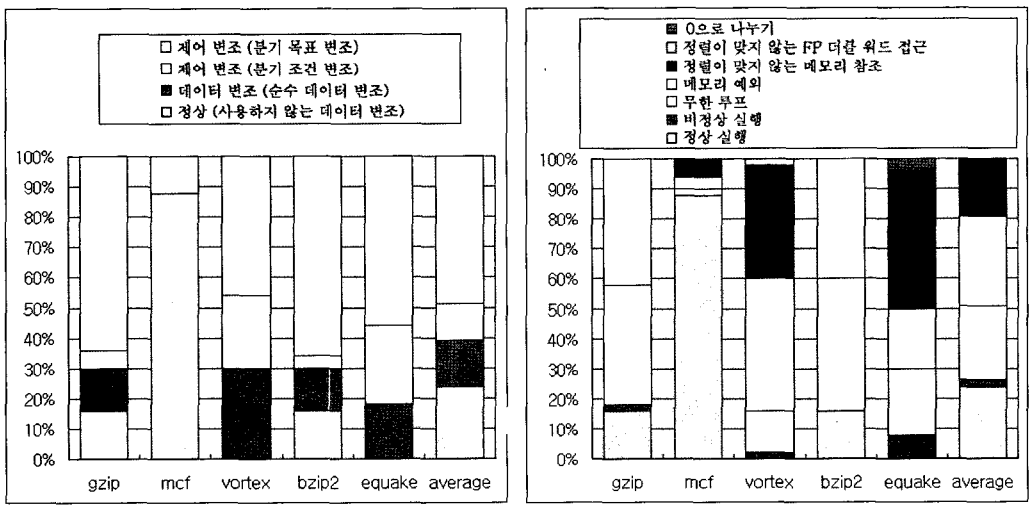

그림 9 스택 스매싱 공격 시 고장 유형 분포

변조는 복귀 주소의 수정으로부터 유발된다.

제어 및 데이타 변조를 비정상 실행 및 비정상 종료로 좀 더 나누어보도록 한다. 그림 9(b)는 비정상 실행의 분포 및 비정상 종료의 상세한 분류를 보여준다. 비정상 종료는 공격을 받은 프로그램이 잘못된 데이타/명령어 참조 혹은 잘못된 피 연산자(operand)로 인해 비정상적으로 종료되는 것을 의미한다. 몇몇 시뮬레이션의 경우, 공격을 당한 프로그램은 무한 루프에 빠져 실행을 계속하게 되는데, 이러한 경우는 비정상적인 종료로 간주되었다. 반면에, 비정상 실행은 프로그램이 잘못된 결과를 보여주긴 하지만 중단 없이 실행을 완료하는 경우를 의미한다. 96.8%의 실패가 비정상 종료인데, 이는 대부분의 실패들이 공격에 의해서 강제로 종료되었으며, 일부 경우만이 잘못된 결과와 함께 중단 없이 실행을 완료했음을 의미한다. 그림에서 보여지는 바와 같이, 접근 위반 및 정렬이 맞지 않는(misaligned) 메모리 참조와 같은 메모리 실행 예외들이 종료의 주된 원인을 차지하고 있으며, 제어 변조로 인한 무한 루프는 비정상 종료의 23%를 차지하고 있다.

그림 10은 입력 크기를 16바이트에서 1킬로바이트까지 변화시킬 때의 고장율 및 고장 유형 분포를 보여준다. 그림에서 볼 수 있듯이, 입력 크기의 증가는 모든 벤치마크 프로그램들의 고장율을 직접적으로 증가시킨다. 또한 데이타 변조 및 제어 변조도 모두 증가한다. mcf의 경우, 입력 크기를 증가시키면 데이타 변조가 제어 변조로 바뀌게 되는데 이는 외부 입력이 스택 내의 복귀 주소를 덮어쓸 정도로 충분히 크다는 것을 의미한다. 또한, equake의 경우 입력 크기가 256 바이트인 경우 약 44%의 제어 변조를 나타내지만 입력 크기가 1024 바이트인 경우 데이타 변조가 발생함을 확인할 수 있다. 이는 equake의 스택 프레임에서 복귀 주소가 스택 포인터에서 약 256 바이트 거리에 저장되어 있기 때문이다. 이 때문에 입력 크기가 1024 바이트가 되면 복귀 주소를 변조할 뿐만 아니라, 다음 프레임의 데이타도 변조하는 것으로 확인된다.

4.2 안전 점검 장치 및 변조 복구 버퍼를 이용한 복구

제안된 마이크로구조 안전 장치들의 효용성을 평가하기 위해, 안전 점검 장치들 뿐만 아니라 CRB를 사용할 수 있도록 프로세서 모델을 수정하였다. SimpleScalar toolset 2.0의 sim-outorder.c와 연관된 파일들의 수정을 통해 각 복구 기법들을 구현하였다. 명령어 참조 안전 점검 장치의 사용은 외부 악성 코드로 인한 제어 변조를 막아주지만 모든 종류의 데이타 변조들에 대해 여전히 취약하다. 또한, 명령어 참조 안전 점검 장치는 내부 코드 제어 변조를 막아내지 못하는데, 이 유형의 변조는 본 실험 환경 및 실제 상황에서도 거의 발생하지 않는다. 데이타 참조 안전 점검 장치를 사용하는 것은 로컬 스택 프레임 외부의 스택 영역으로 이어지는 데이타 변조를 피할 수 있게 해준다. 이를 통해 가장 취약한 데이타인 복귀 주소의 방어를 이용하여 스택 스매싱 공격을 효율적으로 탐지할 수 있다.

그림 11은 다음 네 가지의 서로 다른 복구 기법들에 대해 고장율 및 고장 유형의 분포를 보여준다.

1) 명령어 참조 안전 점검 장치 + 강제 복귀: 제어 변조에 대한 가장 간단한 방어 방법으로, Libsafe, PC-암호화 기법과 같은 방어 수준을 제공한다.

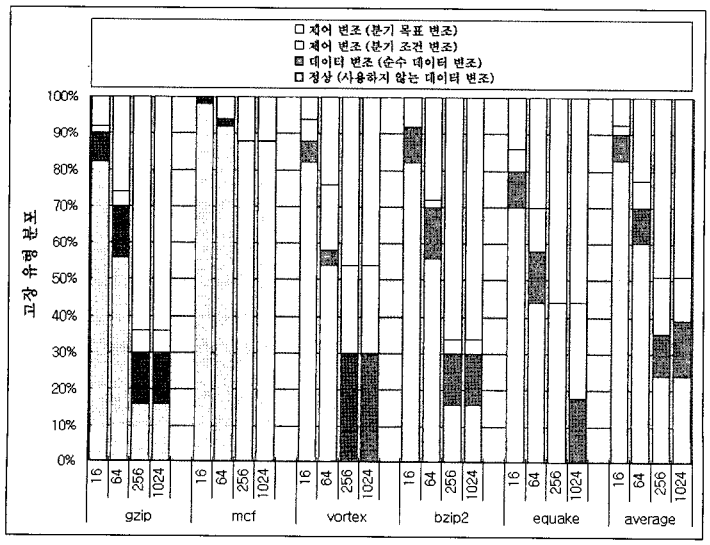

그림 10 입력 크기를 변화시킬 경우 고장 유형 분포

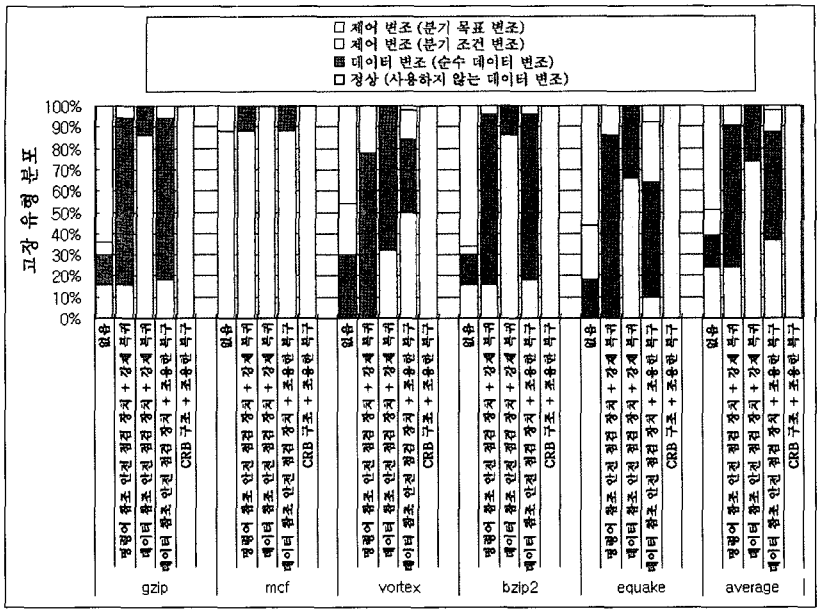

그림 11 4가지 복구 기법들을 이용한 실패 유형의 분포: 1)명령어 참조 안전 점검 장치(IRSG) + 강제 복귀, 2) 데이타 참조 안전 점검 장치(DRSG) + 강제 복귀, 3) DRSG + 조용한 복구, 4) CRB + 조용한 복구

2) 데이타 참조 안전 점검 장치 + 강제 복귀: 복귀 주소 데이타 변조를 방어하는 방법으로, StackGuard, StackShield, RAD, SRAS와 같은 방어 수준을 제공한다.
3) 데이타 참조 안전 점검 장치 + 조용한 복구: 2)와 같이 복귀 주소 변조를 방어하는 방법이지만 데이타 참조 안전 점검 위반이 발생하는 경우에 추후의 데이타 쓰기를 무시하여 제어 흐름을 변경하지 않고 방어하는 방법이다.
4) CRB 구조 + 조용한 복구: 데이타 참조 안전 점검 장치에 추가하여 CRB를 사용하여 데이타 변조를 완전히 차단하는 가장 안전한 방어 방법이다.

예상했던 것과 같이, 명령어 참조 안전 점검 장치는 목적 주소 변조로 인한 모든 제어 변조를 막아냈는데, 이는 스택 영역으로부터 명령어를 가져와 실행하는 것이 불가능해지기 때문이다. 하지만, 이 방법은 정상 실행 비율을 증가시키지는 못하는데 이는 모든 제어 변조들이 데이타 변조들로 변경되었기 때문이다. 그뿐 아니라, 분기 조건 변조로 인한 제어 변조 역시 피할 수가 없다. 반면, 데이타 참조 안전 점검 장치+강제 복귀 기법은 스택 스매싱 공격이 유발하는 고장들을 제거하는데 있어 매우 효과적이다. 데이타 참조 안전 점검 장치를 이용하는 경우, 고장율은 평균적으로 76%에서 26%로 감소하였다. mcf의 경우에는, 데이타 참조 안전 점검 장치만으로 스택 스매싱 공격으로 인해 발생하는 모든 종류의 시스템 고장들을 제거할 수 있었다. 하지만, 시스템은 여전히 순수 데이타 변조들로 인해 발생하는 서비스 거부 공격에 대해 취약하다. 강제 복귀와 비교해볼 때, 조용한 복구는 vortex의 경우를 제외하고는 순수 데이타 변조를 증가시키며 그리 효율적인 방어를 수행하지 못한다. 하지만, vortex의 경우에는 제어 흐름을 변경시키지 않음으로써 데이타 변조를 확실하게 감소시킨다. 주목할 것은 두 종류의 안전 점검 장치를 함께 사용하는 것은 그다지 효과적이지 못하다. 그 이유는 데이타 참조 안전 점검 장치가 더 이른 시기에 복귀 주소의 변조를 막아내어 외부 코드 제어 변조를 피하게 되기 때문이다.

놀랍게도, 데이타 및 제어 변조들로 인한 모든 종류의 시스템 고장은 데이타 안전 점검 장치와 더불어 제안된 복구 기법인 CRB를 사용함으로써 완전하게 제거된다. 따라서, 버퍼 오버플로우 공격의 가장 흔한 형태인 스택 스매싱 공격에 있어, 악성 코드는, 제안된 안전한 프로세서 모델 내에서 공격을 가하는 프로세스의 제어를 바꾸지 못할 뿐 아니라 서비스 거부 상태 역시 유발하지 못한다.

이상의 실험 외에도 CRB에 의한 성능 오버헤드 측정 실험도 수행하였는데, 특히 CRB가 가득 차거나, 또 다른 CRB 트리거가 발생하는 경우, CRB내의 데이타를

표 2 CRB가 유발하는 성능 오버헤드

응용 프로그램	# 총 사이클	# 지연 사이클
gzip	1249669481	14745699(1.18%)
mcf	822783359	12606216(1.53%)
vortex	1578505170	25099276(1.59%)
bzip2	1582514330	7426505(0.47%)
equake	428596033	476900 (0.11%)

메모리로 보내기 위해 요구되는 추가적인 지연 사이클(stall cycles)을 측정하였다. 프로세서 모델은 두 개씩의 읽기, 쓰기 포트를 가진 nonblocking L1 캐쉬를 사용하며, 매 사이클 당 두 개의 단일 워드 쓰기를 허용한다. 4이슈의 비순차적(out-of-order) 파이프라인이 사용되고 있지만, 시뮬레이션의 모든 경우들에 있어 1KB CRB의 성능 오버헤드는 2% 미만이다. 이는 메모리로 데이타를 전달하는데 필요한 대부분의 쓰기 연산들이 쉬고 있는(idle) 메모리 사이클을 사용할 수 있기 때문이다. 그러므로, CRB는 아주 적은 성능 부담만을 유발한다고 할 수 있다. 이는 표 2에서 확인할 수 있다.

5. 결 론

오랜 기간에 걸쳐 버퍼 오버플로우 문제가 알려져 왔지만, 이는 인터넷 보안에 있어 여전히 가장 큰 위협이 되고 있다. 본 논문에서는 프로세서에 쉽게 적용될 수 있는, 새로운 마이크로구조 기법들을 제안한다. 이 기법들은 실생활의 거의 모든 버퍼 오버플로우 시나리오를 방어하는데 있어 매우 효율적이다.

이 논문이 기여하는 바는 다음과 같다. 첫째, 버퍼 오버플로우 공격으로 인해 발생하는 데이타 및 제어 변조들을 분석하고 그것들을 여러 형태의 데이타 및 제어 변조 시나리오들로 분류하였다. 둘째, 데이타 및 명령어 참조의 안전성을 검사함으로써 프로세서를 이용하여 그러한 공격들로 인해 발생하는 제어 및 데이타 변조들을 탐지하고 막아낼 수 있는 방법을 제안한다. 이러한 안전 보호 점검 장치들은 어떠한 종류의 프로세서들에도 적용될 수 있으며, 하드웨어 비용이 적고 성능 저하가 거의 없다. 또한, 변조 복구 버퍼라 불리는 더욱 강도 높고 안전한 마이크로구조 기법을 제안한다. 이는 버퍼 오버플로우 공격들에 관해 제안된 최초의 복구 기법일 것이다. 그리고 기존 연구에서는 다루어지고 있지 않은 서비스 거부 상태에 대해 다루고 있다. 셋째로, 제안된 기법들을 평가하고, 상세한 구조 시뮬레이션을 수행함으로써 다양한 공격 시나리오들에 대해 알아본다. 실험 환경에서는 프로그램의 주소 공간 내의 임의의 데이타 영역에 쓰레기 데이타를 삽입함으로써 현실에서 보고되지 않은 다양한 종류의 공격 시나리오들을 만들어낼 수 있다. 넷째로, 실험 데이타들은 단일 안전 점검 장치를 사용하는 것이 스택 스매싱 공격들로 인해 발생하는 여러 시스템 실패들을 상당 부분 줄여줄 수 있음을 보여준다. 이에 추가하여, 작은 크기의 CRB를 추가로 사용되는 경우, 제안된 안전한 프로세서 모델은 실생활의 공격 시나리오들 중 대부분을 차지하는 스택 스매싱으로 인한 모든 종류의 제어 및 데이타 변조들을 완전히 제거할 수 있다. 이러한 가능성 있는 초기 결과들을 바탕으로, 힙 영역에서의 함수 포인터 변조와 같은 더욱 희귀한 버퍼 오버플로우 공격들을 탐지하고 복구할 수 있는 보다 효율적인 방법들을 앞으로 연구할 계획이다.

참 고 문 헌

[1] CERT/CC Statistics 1988-2003, http://www.cert.org/stats/cert_stats.html.

[2] Cooperative Association for Internet Data Analysis(CAIDA), Analysis of the Sapphire Worm, http://www.caida.org/analysis/security/sapphier/, Jan. 2003.

[3] Common Vulnerabilities and Exposures(CVE), [TECH] Vulnerability Types Seen in CVE, http://cve.mitre.org/board/archives/2002-10/msg00005.html.

[4] SANS Institute and FBI, The Twenty Most Critical Internet Security Vulnerabilities~The Expert's Consensus, http://www.sans.org/top20, 2003.

[5] Snow, B., Future of Security, Panel Presentation at IEEE Security and Privacy, May 1999.

[6] Openwall Project, Linux kernel patch from the openwall project, http://openwall.com/linux

[7] Chiueh T.-C. and Hsu F.-H., RAD: A Compile-Time Solution to Buffer Overflow Attacks, In Proceedings of the 21st International Conference on Distributed Computing Systems, 2001.

[8] Cowan, C., Beattie, S., Day R. F., Pu, C., Wagle, P., and Walthinsen, E., Protecting Systems from Stack Smashing Attacks with StackGuard, In the Linux Expo, 1999.

[9] Lee, G. and Tyagi, A., Encoded Program Counter: Self-Protection from Buffer Overflow Attack, In Proceedings of the International Conference on Internet Computing, June 2000.

[10] Pyo C. and Lee, G., Encoding Function Pointers and Memory Arrangement Checking against Buffer Overflow Attack, In Proceedings of the Fifth International Conference on Information and Communications Security, October 2003.

[11] Jones R. and Kelly, P., Bounds Checking for C, http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html, July 1995.

[12] Baratloo, A., Singh, N., and Tsai, T., Transparent Run-Time Defense Against Stack Smashing Attacks, In Proceedings of the USENIX Annual Technical Conference, June 2000.

[13] Richard W M Jones and Paul H J Kelly, Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs, May 1997.
[14] Cowan, C., Wagle, P., Pu, C., Beattie S. and Walpole, J., Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade, In Proceedings of the DARPA Information Survivability Conference and Exposition. January 2000.
[15] Peter Silberman and Richard Johnson, A Comparison of Buffer Overflow Prevention Implementations and Weaknesses.
[16] Bulba and Kil3r, Bypassing Stackguard and Stackshield, Phrack, 10(56), May 2000.
[17] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel and Ravishankar K. Iyer, Architecture Support for Defending Against Buffer Overflow Attacks.
[18] John P. McGregor, David K. Karig, Zhijie Shi, and Ruby B. Lee, A Processor Architecture Defense against Buffer Overflow.

최 린

1986년 2월 서울대학교 컴퓨터공학과 졸업. 1988년 2월 서울대학교 컴퓨터공학과 석사. 1988년~1990년 8월 한국통신 연구개발단 전임연구원. 1990년 9월~1996년 3월 미국 일리노이 주립대 컴퓨터학공학 박사. 1996년 4월~1998년 8월 Intel Corporation, Santa Clara, California, Senior Design Engineer, Microarchitecture team of 64-bit CPU,. Itanium project. 1998년 9월~2000년 8월 미국 캘리포니아 주립대 컴퓨터공학 조교수. 2000년 9월~2002년 8월 고려대학교 전기전자전파공학부, 전자공학과 조교수. 2002년 9월~현재 전기전자전파공학부, 전자공학과 부교수

신 용

2003년 2월 고려대학교 전기전자전파공학부 졸업. 2005년 2월 고려대학교 전자컴퓨터공학과 석사. 2005년 3월~현재 삼성전자 연구원

이 상 훈

2004년 2월 고려대학교 전기전자전파공학부 졸업. 2004년 3월~현재 고려대학교 전자컴퓨터공학과 석박사 통합과정 재학