Sentry: Kernel Extension을 위한 바이너리 수준의 인터포지션 기법*

김세원° 황재현, 유혁
고려대학교 컴퓨터 학과
{swkim°, jhhwang, hxy}@os.korea.ac.kr

Sentry: a Binary-Level Interposition Mechanism for Kernel Extension

Se-Won Kim°, Jae-Hyun Hwang, and Hyuck Yoo
Dept. of Computer Science and Engineering, Korea University

요 약

현재 사용되고 있는 운영체제들은 그들의 기능을 확장하거나 교체하기 위해서 kernel extension을 사용해 왔다. 일반적으로 이러한 kernel extension들은 커널과 같은 주소공간에서 실행하기 때문에, 그것에서 발생하는 오류(fault)로 인해 전체 시스템이 망가지는 결과를 초래할 위험이 있다. 그래서 kernel extension의 안전한 실행에 관한 연구들은 kernel extension에서 발생한 오류를 전체 시스템으로부터 고립시키는 것이 주 목적이었다. 하지만 이러한 방법들은 kernel extension의 어셈블리어로 된 코드를 분석하거나 사용하고 있는 커널의 소스 코드를 수정을 필요로 한다.

본 논문은 Sentry라는 kernel extension을 감시하기 위한 인터포지션 서비스를 제안한다. Sentry를 사용하기 위해서 별도의 커널 코드를 수정할 필요도 없으며, 이미 사용하고 있는 리눅스와 호환될 수 있는 특징을 지니고 있다. 그리고 kernel extension의 소스코드 및 어셈블리 코드에 대한 분석 없이 바이너리 파일을 직접 수정하여 kernel extension을 모니터링 할 수 있도록 한다. 게다가 Sentry는 재구성이 가능하기 때문에 얼마든지 kernel extension에 대한 보호 정책을 동적으로 바꿀 수 있다.

1. 서 론

현재 사용되는 운영체제들은 그들의 기능을 확장하거나 추가하기 위해 kernel extension을 사용해왔다. Kernel extension의 대표적인 예로 장치 드라이버를 들 수 있겠다. 새로운 장치가 시스템에 추가될 경우, 커널을 새로 컴파일 하거나 다시 설치할 필요 없이, 장치 드라이버를 커널 주소공간에 로드함으로써, 커널은 새로운 장치를 사용할 수 있게 된다. 이와 같은 장치 드라이버의 예처럼, kernel extension은 커널을 보다 유연하고 확장 가능하게끔 한다. 그러나 이와 같은 kernel extension은 커널을 불안정한 상태로 만드는 원인이 되기도 한다. 예를 들어, 로드된 kernel extension이 올바르지 않은 주소에 접근하여 오류가 발생한다면, 커널 전체가 망가지는 현상이 발생할 수 있다. 불행히도, 대부분의 운영체제는 커널 주소 공간내의 보호 메커니즘을 제공하기 않기 때문에 이와 같은 문제를 막는 것은 쉬운 일이 아니다.

인터포지션 서비스는 kernel extension의 동작을 감시하는 환경을 구현하기위한 방법으로 잘 알려져 있다. 인터포지션 서비스를 하기위해, 로드할 Kernel extension이 호출하는 함수 전에 훅(hook)을 삽입한다. 삽입된 훅은 약속된 wrapping stub을 호출하여 kernel extension이 호출하고자 하는 함수를 허용할지 안할지 결정한다. 이와 같은 방법은 단순하면서도 직관적인 반면, 두 가지 문제점을 가지고 있다. 첫 번째는 인터포지션 서비스를 하기위해서는 kernel extension코드를 수정하거나 커널 소스 코드를 수정해야 한다. Heck[1]에서는 kernel extension의 어셈블리 코드에 훅을 삽입한다. 하지만, 어셈블리 코드 자체에 새로운 코드를 삽입하면, jmp와 같이 상대 주소로 분기하는 명령어에 대해 주소 계산을 다시 해야 하는 복잡한 과정이 필요하게 된다. 또한 Nooks[2]는 인터포지션 서비스를 제공하기 위해 커널 코드인 kernel module loader를 수정해야만 했다. 두 번째는 새로운 wrapping stub을 추가하거나 이미 사용되는 wrapping stub을 교체하기 위해서는 전체 인터포지션 서비스를 새로 컴파일해야 해야 한다. 이 문제는 wrapping stub과 나머지 인터포지션 서비스가 분리된 구조를 통해서 해결할 수 있을 것이라 본다.

본 논문은 Sentry라는 kernel extension을 감시하기 위한 인터포지션 서비스를 제안한다. Sentry를 사용하기

* 본 연구는 한국과학재단 특정기초연구 (R01-2004-000-10588-0)의 지원으로 수행되었음

위해서 별도의 커널 코드를 수정할 필요도 없으며, 이미 사용하고 있는 리눅스와 호환될 수 있는 특징을 지니고 있다. 그리고 kernel extension의 소스코드 및 어셈블리 코드에 대한 분석 없이 바이너리 파일을 직접 수정하여 kernel extension을 모니터링 할 수 있도록 한다. 게다가 Sentry는 재구성이 가능하기 때문에 얼마든지 kernel extension에 대한 보호 정책을 동적으로 바꿀 수 있다.

본 논문의 2장에서는 Sentry 두 부분인 Sentry Run-time System과 Kernel Extension Pre-Processor의 관계를 살펴본다. 3장에서 각 구성 요소에 대한 설계와 이러한 설계를 바탕으로 선행 연구의 발견된 문제점을 해결책을 제시하고 4장을 마지막으로 본 논문의 결론을 제시한다.

2. Sentry의 구조

그림 1은 Sentry의 구조를 보인다. Sentry는 Sentry Run-time System과 Kernel Extension Pre-Processor로 나뉘어 진다. Sentry Run-time System은 리눅스의 LKM(Loadable Kernel Module)으로 구현되어 커널의 주소공간에서 실행한다. 이것은 kernel extension의 동작을 실행 시간에 감시하는 역할을 담당한다. Kernel Extension Pre-Processor는 유저 프로세스로 로드하기 위한 kernel extension의 ELF[3]정보를 수정한다. 이렇게 수정된 kernel extension을 Sentry Run-time System의 wrapping stub을 호출하여 Sentry로부터 감시를 받게 된다.

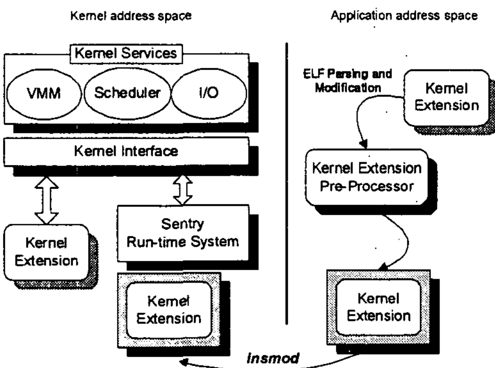

그림 1 Sentry의 구조

3. Sentry의 설계

3.1. Sentry Run-Time System

Sentry Run-time System은 wrapping stub, run-time system engine, policy로 구성된다. Wrapping stub은 수정된 kernel extension이 호출하는 함수들의 집합이다. 모든 wrapping stub은 Sentry에 빌트인(built-in)되어 있지 않다. 즉, 각각의 wrapping stub역시 LKM으로 구현되어 있기 때문에, 리눅스의 kernel module loader의 도움으로 Sentry와 별도로 컴파일 되고 Sentry와 같은 주소 공간에 로드되어 사용된다.

Run-time System Engine은 앞서 설명한 wrapping stub을 관리하고 kernel extension의 실행에 대한 로그를 생성한다. wrapping stub을 관리하기 위해, 우리는 두 개의 interface인 *register_wstub()*과 *unregister_wstub()*을 정의했다. 이 두 함수는 각각 wrapping stub을 Run-time System Engine에 wrapping stub을 등록 하고 해제하는 역할을 수행한다. Wrapping stub은 리눅스의 LKM형태로 구현되기 때문에, init_module()과 cleanup_module() 두 함수[4]가 있다. 따라서 wrapping stub을 작성할 때, init_module()함수에서 *register_wstub()*을 그리고 cleanup_module()함수에서는 *unregister_wstub()*을 호출하도록 한다면, kernel module loader에 의해 자동적으로 위 두 인터페이스가 호출된다.

Policy는 kernel extension에게 허용할 커널 함수 및 시스템 자원을 명세 한다. Wrapping stub이 policy에 기반 하여 동작할 수 있도록, *query_permission()*이라는 별도의 인터페이스를 제공한다. 이를 이용하여, wrapping stub은 kernel extension이 요청한 함수를 허용할지 말지 결정할 수 있게 된다.

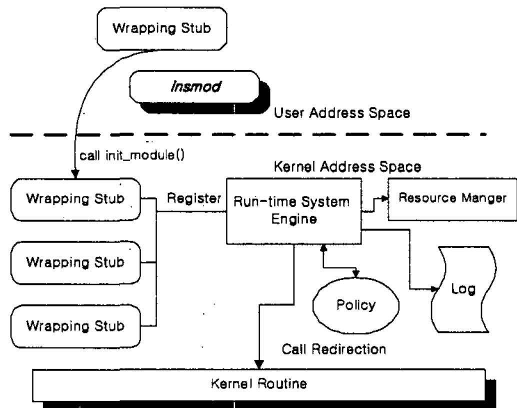

그림 2 Sentry Run-time System

3.2. Kernel Extension Pre-Processor

그림 3은 Kernel Extension Pre-Processor가 Relocatable ELF 오브젝트 파일인 kernel extension을 바이너리 수준에서 수정하는 과정을 보인다. 이러한 작

업을 수행하기 위해서는, .symtab으로 나타내어지는 심볼 테이블(symbol table)과 .strtab으로 나타내어지는 문자열 테이블(string table)이 필요하다. 이 두 정보는 역어셈블러나 *readelf*[5]와 같은 특별한 방법 없이, ELF 오브젝트 파일 자체만으로도 알 수 있는 정보들이다. Kernel Extension Pre-Processor는 먼저 ELF 오브젝트 파일을 읽은 후에 ELF 헤더와 섹션 정보들을 파싱(parsing)하여 파일 내의 .symtab과 .strtab의 위치를 찾아낸다. .symtab은 고유의 자료구조인 *Elf32_Sym*의 연속으로 되어 있기 때문에 .symtab의 크기를 *Elf32_Sym*의 단위로 분리하면 각각의 심볼 정보를 알 수 있다. 반면에 .strtab의 경우는 아스키 문자로 인코딩되어 있다.

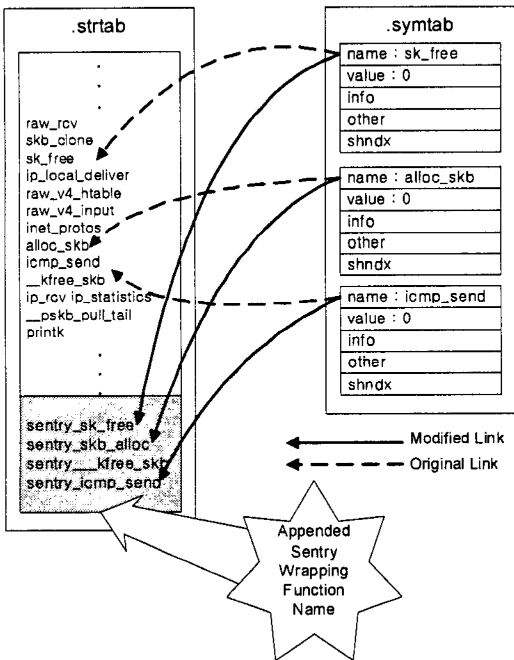

그림 3 Relocatable ELF 오브젝트 파일을 수정하는 과정

Kernel Extension Pre-Processor는 .strtab과 .symtab을 분석하여 kernel extension이 호출할 함수들의 이름을 찾아낸다. 그리고 나면, "sentry_"라는 접두어가 추가된 wrapping stub의 이름을 .strtab에 덧붙인다. .strtab의 추가 과정을 마치고나면 이제 .symtab의 name 필드를 새로운 함수의 이름으로 바꾸고 나면 모든 과정이 끝난다. 이렇게 수정된 kernel extension은 이제 원래의 함수가 아닌 Sentry에서 제공하는 wrapping stub을 호출하게 된다.

이와 같은 간단한 작업만으로 kernel extension의 함수 호출을 Sentry의 warpping stub으로 바꾸는 이유는 다음과 같다. 리눅스의 kernel module loader는 kernel extension내에 있는 심볼에 대한 주소를 결정할 때, 아스키 문자로 된 심볼의 이름을 이용하기 때문이다. 따라서 Kernel Extension Pre-Processor에 의해 바뀐 심볼의 이름으로 비교를 하기 때문에, kernel extension이 로드되기 전에 이미 Sentry의 wrapping stub이 커널에 존재 하면, 수정된 kernel extension의 주소 결정은 Sentry의 wrapping stub의 주소로 치환된다.

4. 결 론

지금까지 Sentry의 구조와 설계를 살펴보았다. Sentry는 wrapping stub과 나머지 run-time system을 분리하여 언제든지 kernel extension에 대한 감시 방법을 교체 할 수 있는 유연한 구조를 갖는다. 게다가 Relocatable ELF를 수정하는 방법은 선행연구에서 제시된 방법보다 간편하면서도, kernel extension의 코드 전체를 분석해야 하는 오버헤드를 갖고 있지 않다. Sentry는 그 구조가 간단하면서 유연하고 커널의 수정을 요구하지 않기 때문에 현재 널리 사용되는 운영체제인 리눅스에서도 쉽게 사용될 수 있을 거라 생각된다.

참고 문헌

[1] Haizhi Xu, Steve C, and Wenliang D, "Detecting Exploit Code Execution in Loadable Kernel Modules", In Proc of the 20th Annual Computer Security Applications Conference, Dec 2004.

[2] Micheal M. Swift, Brian N. Bershad, and Henry M. Levy, "Improving the Reliability of Commodity Operating Systems.", In Proceedings of the 19th ACM Symposium on Operating Systems Principles, October 2003.

[3] Executable and Linkable Format (ELF). Version 1.1, TIS Committee, Oct. 199

[4] J. Corbet, A. Rubini, and G. Kroah-hartman, "Linux Device Drivers", 3rd Edition, O'reilly, Feb 2005.

[5] GNU Binutils http://www.gnu.org/software/binutils/