情報保護學會論文誌
第15卷 第6號, 2005. 12

리눅스 커널 변수 취약성에 대한 소스레벨 발견 방법론*

김 재 광,[1†] 고 광 선,[1] 강 용 혁,[2] 엄 영 익[1‡]
[1]성균관대학교, [2]극동대학교

A Source-Level Discovery Methodology for Vulnerabilities of Linux Kernel Variables*

Jaekwang Kim,[1†] Kwangsun Ko,[1] Yong-hyeog Kang,[2] Young Ik Eom[1‡]
[1]Sungkyunkwan University, [2]Far East University

요 약

오늘날 리눅스 운영체제는 임베디드 시스템, 라우터, 대규모 서버에 이르기까지 다양한 분야에 사용되고 있다. 이는 리눅스 운영체제가 추구하는 커널 소스 공개 정책이 시스템 개발자들에게 여러 가지 이점을 주기 때문이다. 하지만 시스템 보안 측면에서 볼 때, 리눅스 커널 소스 공개는 보안상 문제점을 발생시킬 수 있는데, 만일 누군가가 리눅스 기반의 시스템을 공격하려 한다면 그 공격자는 리눅스 커널의 취약성을 이용하여 쉽게 시스템을 공격할 수 있기 때문이다. 현재까지 소프트웨어의 취약성을 분석하는 방법은 많이 있었지만 기존의 방법들은 방대한 크기의 리눅스 커널 소스에서 취약성을 발견하기에 적합하지 않다. 본 논문에서는 소스레벨 리눅스 커널 변수 취약성을 발견하는 방법론으로 Onion 메커니즘을 제안한다. Onion 메커니즘은 두 단계로 이루어져 있는데, 첫 번째 단계는 패턴매칭 방법을 이용하여 취약 가능성이 있는 변수들을 선정하는 단계이고, 두 번째 단계는 선정된 변수들의 취약 여부를 시스템 콜 트리를 이용해 검사하는 단계이다. 또한 본 논문에서 제안한 방법론을 이미 알려진 두 가지 소스레벨 취약성에 적용한 결과를 보인다.

ABSTRACT

In these days, there are various uses of Linux such as small embedded systems, routers, and huge servers, because Linux gives several advantages to system developers by allowing to use the open source code of the Linux kernel. On the other hand, the open source nature of the Linux kernel gives a bad influence on system security. If someone wants to exploit Linux-based systems, the attacker can easily do it by finding vulnerabilities of their Linux kernel sources. There are many kinds of existing methods for finding source-level vulnerabilities of softwares, but they are not suitable for finding source-level vulnerabilities of the Linux kernel which has an enormous amount of source code. In this paper, we propose the Onion mechanism as a methodology of finding source-level vulnerabilities of Linux kernel variables. The Onion mechanism is made up of two steps. The first step is to select variables that may be vulnerable by using pattern matching mechanism and the second step is to inspect vulnerability of each selected variable by constructing and analyzing the system call trees. We also evaluate our proposed methodology by applying it to two well-known source-level vulnerabilities.

Keywords : *Linux Kernel, Vulnerability, Source-Level Discovery Methodology*

접수일 : 2005년 7월 12일 ; 채택일 : 2005년 11월 25일
* 본 연구는 정보통신부 대학 IT연구센터 육성, 지원사업의 연구결과로 수행되었습니다.
† 주저자 : linux@ece.skku.ac.kr
‡ 교신저자 : yieom@ece.skku.ac.kr

Ⅰ. 서 론

최근 리눅스 운영체제는 임베디드 시스템에서부터 하이엔드 서버에 이르기까지 다방면에 사용되기 알맞은 형태로 포팅 되어 가고 있다. 이처럼 다른 운영체제에 비하여 리눅스 운영체제가 다방면에서 주목받는 이유는 리눅스 운영체제의 소스가 공개되어 새로운 기술을 빠르게 적용할 수 있다는 점 때문이다. 하지만 소스의 공개는 양날의 검과 같아서 악의적인 사용자에 의해 리눅스 커널 소스의 작은 취약성 하나가 악용될 경우 큰 재앙을 불러올 수 있다. 최근엔 리눅스 커널 개발이 빠르게 진행됨에 따라 리눅스 커널에 각종 디바이스를 비롯한 여러 가지 기능들이 추가되는 주기가 짧아지고 있고 새로 추가된 코드로부터 보고되는 취약성의 수도 증가하고 있다[1]. 이에 따른 위험 요소를 줄이기 위해서는 리눅스 커널 소스의 취약성을 조기에 발견하고 보완하는 연구가 필요하다[2,3].

방대한 크기의 리눅스 커널 소스에서 수작업으로 취약성을 찾는다는 것은 상당히 난해한 일이기 때문에 패턴매칭 기술이나 구문분석 기술을 이용하여 취약성을 발견하는 연구가 진행되고 있다. 패턴매칭 기술은 기존의 취약성 패턴을 벗어나는 취약성을 발견하는데 한계가 있었고, 구문분석 기술은 구현 자체가 너무 어렵고 제한된 범위에만 사용 되는 한계를 가지고 있다[4-6].

본 논문에서는 리눅스 커널 소스의 커널 영역 변수 취약성을 발견하는 발견 방법론으로 Onion 메커니즘을 제안한다. Onion 메커니즘을 사용하면, 기존의 공개된 취약성 유형 중에서 미 발견된 커널 변수 취약성을 발견할 수 있다. Onion 메커니즘은 두 단계로 구성되어 있다. 첫 번째 단계는 리눅스 커널 소스에서 취약 가능 변수를 선별 하는 단계이고, 두 번째 단계는 취약 변수에 대한 취약여부를 검사하는 단계이다. 또한 기 공개된 리눅스 커널 소스 취약성을 발견하는데 적용함으로써 Onion 메커니즘의 타당성을 보인다.

본 논문의 구성은 다음과 같다. 2장에서 관련 연구를 보이고, 3장에는 Onion 메커니즘을 자세히 설명하며, 4장에서 Onion 메커니즘을 이용하여 기 발견된 리눅스 커널 소스 취약성에 적용한 결과를 보인다. 마지막으로 5장에서 결론을 보인다.

Ⅱ. 관련 연구

본 절에서는 기존에 연구되었던 취약성 발견 방법에 대하여 설명한다. 취약성 발견 방법은 일반적으로 패턴매칭 기술을 이용한 방법과 구문분석 기술을 이용한 방법으로 나뉜다.

1. 패턴매칭 기술을 이용한 취약성 발견 방법

패턴매칭 기술은 미리 알려진 취약성 패턴을 이용하여 검사하는 기술로서 이 기술을 이용하여 리눅스 커널 소스가 가지고 있는 발견되지 않은 새로운 취약성을 발견할 수 있다. 패턴매칭 기술을 사용하여 소프트웨어의 취약성을 검사하면 빠르고 일관된 검사가 가능하다는 장점이 있는 반면 검사에 사용하는 패턴이 고정되어 있어 새로운 종류의 취약성을 발견하는데 어려움이 있다. 대표적인 패턴매칭 도구로는 ITS4와 RATS가 있고, 관련 연구로는 D. Wagner가 제시한 연구가 있다[7-10].

ITS4는 C 또는 C++ 언어로 작성된 소스 코드에서 잠재적인 취약성을 가지고 있는 함수 호출을 검사하는 도구로 유닉스와 윈도우 플랫폼에서 커맨드라인으로 동작한다[7,8]. ITS4가 검사할 수 있는 취약성의 종류는 버퍼 오버플로우를 발생시킬 수 있는 스트링 관련 함수들과 레이스 컨디션을 일으킬 수 있는 함수들이다. ITS4는 소스 코드를 검사할 때 파일에 대해서만 검사가 가능하기 때문에 디렉토리 전체에 대한 검사를 실시하고자 할 경우 쉘 스크립트를 이용해야 한다. ITS4는 변수명과 함수명을 상세하게 설정하여 검사할 수 있기 때문에, 소스의 세밀한 분석이 가능하다는 특징이 있으며, 사용자가 필요에 따라서 취약성 패턴을 추가할 수도 있다.

RATS는 Security Software사에 의해 개발된 오픈 소스 도구로서 C 또는 C++ 언어로 작성된 소스 코드의 취약성을 검사한다. C/C++ 언어 이외에 PHP 또는 Perl과 같은 다양한 언어로 작성된 소스에 대한 검사도 가능하다[9]. RATS가 검

사할 수 있는 취약성의 범위는 버퍼 오버플로우를 발생시킬 수 있는 스트링 관련 함수들과 레이스 컨디션을 발생시킬 수 있는 함수들, 난수 생성 관련 함수들, 시스템 콜 함수들이다. RATS는 내부적으로 관리하는 취약성 패턴이 XML로 관리되기 때문에 사용자가 필요에 따라서 취약성 패턴을 확장하는 것이 매우 쉽다. 또한 검사결과를 HTML 형식으로 보여줌으로써 사용자가 결과를 쉽게 확인할 수 있다.

D. Wagner는 소프트웨어의 보안과 오류에 관한 취약성을 발견하는 절차적인 방법에 대한 연구를 실시하였다. 이 방법은 먼저 안전한 프로그래밍을 할 수 있는 규칙을 정해놓고, 이러한 규칙에 맞는지 검사하는 것이다. 이 방식은 알려진 특정 종류의 취약성뿐 아니라 알려지지 않은 취약성에 대하여도 검증이 가능하다는 장점이 있다[10].

2. 구문분석 기술을 이용한 취약성 발견 방법

구문분석 기술은 문장의 문맥을 분석하는 기술로서 문장의 전후 문맥을 따져 오류를 검출하는 방법이다. 구문분석 기술을 사용하여 소프트웨어의 취약성을 발견할 경우, 구분문석 기술을 이용하여 취약성을 발견할 때에는 특정 패턴이 없더라도 변수 값의 관계나 소스의 논리적인 오류로 인한 취약성에 대한 발견이 가능하며, 패턴매칭 기술로는 찾을 수 없는 문맥상의 오류를 찾아낼 수 있다는 장점이 있는 반면, 구문분석을 기술의 구현 자체가 어렵다는 단점이 있다[11]. 대표적인 구문분석 도구로는 Splint와 Cqual이 있고, 관련 연구로는 D. Engler의 연구가 있다[12-16].

Splint는 C 언어에서 사용되는 구문분석 도구인 LCLint에다가 취약성 검사 기능을 추가한 도구이다. Splint를 사용하면 검사 대상 소스에 Splint 명령어를 기술함으로써 사용되지 않는 변수, 변수 간 타입 불일치, 반환 값 불일치, 무한루프의 반복 등과 같은 취약성을 검사할 수 있다[12]. Splint가 검사할 수 있는 취약성의 범위는 사용되지 않는 변수, 변수 간 타입 불일치, 반환 값 불일치, 무한루프의 반복, 버퍼 오버플로우이다. Splint는 검사 대상이 되는 소스 코드와 검사 규칙에 해당하는 헤더 파일을 입력으로 받은 후, 소스 코드에서 호출되는 함수들의 타당성 여부를 검사한다.

Cqual은 C 언어로 작성된 소스 코드의 데이터 타입을 분석하는 도구이다[13,14]. 사용자는 C 언어에서 제공하는 데이터 타입을 확장한 형태 수식자를 소스 코드에 추가하고, 미리 정의한 형태 수식자와 데이터 타입의 변화가 일치하는지를 확인하여 취약성을 검사한다. Cqual이 검사할 수 있는 취약성은 사용자가 검사하고자 하는 변수가 잘못된 변화를 일으키는 경우이다. Cqual를 사용하여 취약성을 검사할 때에는 qualifier interface라는 인터페이스를 통하여 검사 결과를 추적할 수 있다.

D. Engler는 특정 시스템에서 정적 분석을 통해 보안 오류 또는 정수의 잘못된 사용과 같은 취약성들을 찾는 시도를 하였다[15,16]. 여기서 사용한 취약성 발견 방법은 컴파일러를 이용하여 컴파일 시 취약성을 발견하는 것인데, 이 컴파일러는 각 시스템에 맞게 수정된 것으로 특정 시스템에 맞는 확장된 다단계 컴파일러이다. D. Engler는 컴파일러를 이용하여 리눅스와 FreeBSD의 100가지 보안 오류들을 검사하고 그 결과를 공표하였는데, 이러한 접근을 통해 정적인 시스템의 오류를 찾아낼 수 있다고 주장하였다[17].

Ⅲ. Onion 메커니즘

본 절에서는 리눅스 커널 변수의 취약성 발견 방법론으로 Onion 메커니즘을 제안한다. Onion이라는 명칭은 단계적으로 취약성 여부를 밝혀낸다는 의미를 지닌다. Onion 메커니즘은 아래와 같이 두 단계로 구성되어 있다.

- 1 단계: 취약 가능 변수의 선별
- 2 단계: 선별된 변수의 취약 여부 검사
  - 2-1. 시스템 콜 트리의 구성
  - 2-2. 사용자로부터 선별된 커널 변수까지 도달하는 경로 선정
  - 2-3. 사용자의 입력 값 범위 계산

Onion 메커니즘의 첫 번째 단계는 커널 영역

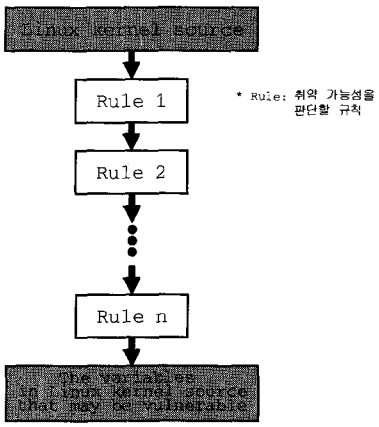


그림 1. 취약 가능 변수의 선별 단계

변수 중 정수 오류가 발생 할 가능성이 있는 소스 변수들을 선별하는 단계이고, 두 번째 단계는 선별된 변수가 취약 변수인지 검사하는 단계이다. 여기서 두 번째 단계는 다시 세 가지 세부 과정으로 구성되며, 각각 시스템 콜 트리를 구성하는 과정, 사용자로부터 선별된 변수까지 도달하는 경로 선정 과정, 사용자의 입력 값 범위 계산 과정이다.

1. 취약 가능 변수의 선별 (1단계)

취약 가능 변수를 선별하기 위하여 패턴매칭 기술과 구문분석 기술을 혼합하여 사용하며, 이에 대한 구성은 그림 1과 같다.

그림 1에서 보이는 바와 같이 취약 가능성이 존재하지 않은 커널 변수들을 단계적으로 제외하여, 최종적으로 취약 가능성이 존재하는 리눅스 커널 변수들을 얻는다. 이때, 각 단계에서 취약 가능성이 존재하지 않는 커널 변수를 제외하는 방법은 규칙을 기반으로 하며, 이 규칙들은 '취약 원인에 따른 리눅스 커널 취약성 분류법'을 참조한다[1,18-21]. 추가적으로, 리눅스 커널 소스에는 정수형 변수만 존재하므로 정수형 변수의 취약여부를 판별하는 규칙만을 적용한다. 이와 같이 취약 가능 변수를 선별하면 조사 대상이 되는 커널 소스의 범위를 축소하여 취약성 발견 가능성을 높일 수 있다[22]. 한 예로써 4개의 리눅스 커널 변수(map_count, i_count, IP_MSFILTER_SIZE, GROUP_FILTER _SIZE)가 전체 커널 소스에서 차지하는 비율은 표 1과 같다.

또한, 리눅스 커널 소스에서 사용되는 자료 형식(__u8, __u16, __u32, __s8, __s16, __s32)으로 선언된 변수가 전체 커널 소스에서 차지하는 비율은 표 2와 같다.

표 1과 표 2의 분석 결과가 보이는 바와 같이 Onion 메커니즘의 1 단계를 통해 효과적인 패턴매칭 규칙을 적용할 경우, 취약성 여부를 확인해야 하는 커널 소스의 양을 효과적으로 줄일 수 있음을 알 수 있다. 이러한 과정을 통해 선별된 리눅스 커널의 취약 가능 변수는 Onion 메커니즘의 두 번째 단계에서 취약성 여부가 검사된다[23][24].

2. 취약 여부 검사 (2단계)

Onion 메커니즘의 두 번째 단계인 선별된 정수형 변수의 취약 여부를 검사 하는 단계는 다음의 두 가지를 확인하는 것을 목적으로 한다.

- 선별된 취약 가능성 있는 커널 영역 변수에 영향을 미치는 사용자의 입력 경로가 존재하는지

표 1. 4개의 리눅스 커널 변수가 전체 커널 소스에서 차지하는 비율 (단, 리눅스 커널 버전 2.4.23 기준)

변수명	특정 변수가 사용된 라인 수	전체 커널 소스에서 차지하는 비율 (%)
map_count	146	0.0004870
i_count	183	0.0006100
IP_MSFILTER_SIZE	8	0.0000267
GROUP_FILTER_SIZE	11	0.0000367
합 계	348	0.0011600

표 2. 리눅스 커널 소스에서 사용되는 자료형식으로 선언된 변수가 전체 커널 소스에서 차지하는 비율 (단, 리눅스 커널 버전 2.4.23 기준)

변수 타입	특정 형식의 변수가 선언된 라인 수	전체 커널 소스에서 차지하는 비율(%)
__u8	2,284	0.0007610
__u16	1,644	0.0054800
__u32	2,723	0.0090800
__s8	5	0.0000167
__s16	37	0.0001230
__s32	95	0.0003170
합 계	6,788	0.0226000

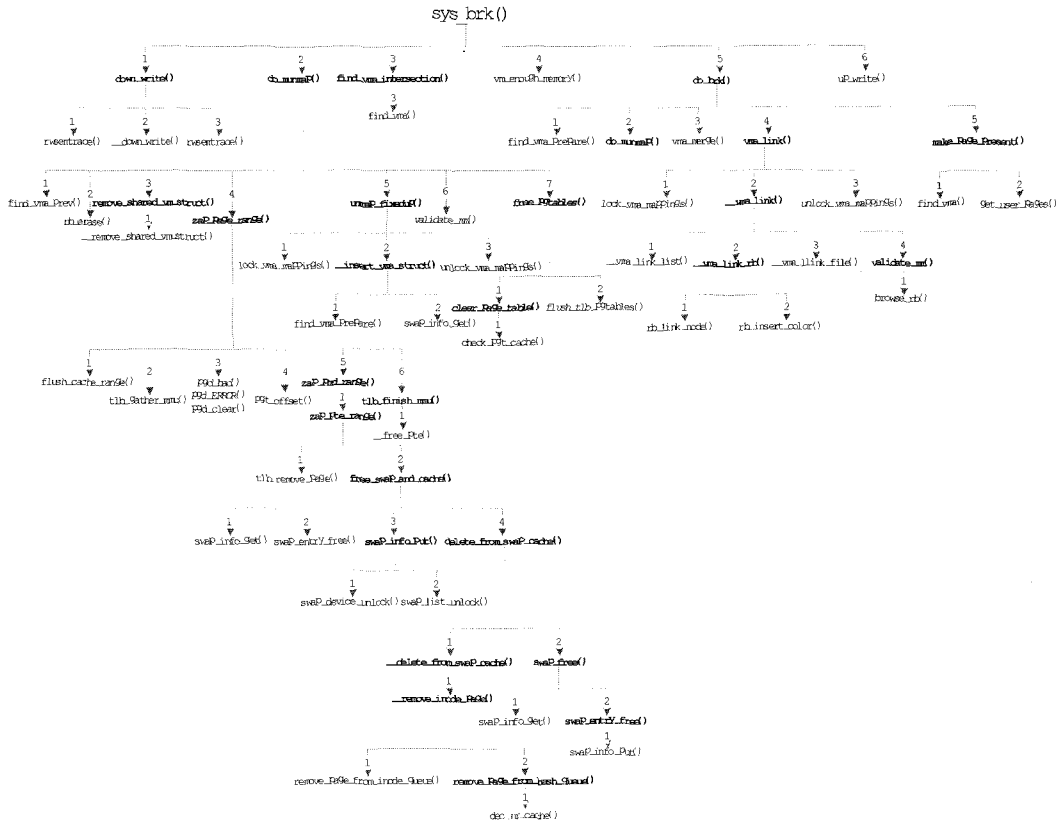


그림 2. sys_brk() 시스템 콜 트리

확인

· 커널 영역 변수의 오류를 일으키는 사용자의 입력 범위가 존재하는 확인

위의 두 가지, 즉 경로의 확인과 사용자의 입력 범위를 확인하기 위해 가장 먼저 시스템 콜 트리를 구성한다.

2.1 시스템 콜 트리 구성

시스템 콜 트리는 특정 시스템 콜이 호출하는 커널 함수의 이름, 순서, 그리고 호출 관계 등을 명시한 트리이다. 하지만 처음부터 리눅스에 존재하는 모든 시스템 콜에 대해서 모든 호출 경로를 그린다는 것은 비효율적이기 때문에, Onion 메커니즘에서는 첫 번째 단계인 취약 가능 변수 선별 결과를 이용하여 선별된 커널 영역 변수를 호출하는 역방향 시스템 콜 트리를 구성한다. 단, 동작 중인 커널에서는 시스템 콜 트리의 구성이 다양하게 나타날 수 있기 때문에 본 제안 기법에서 구성하는 시스템 콜 트리는 정적인 커널 소스를 적용대상으로 한다[25]. sys_brk() 시스템 콜을 하였을 때, map_count 변수에 미치는 사용자의 영향을 살펴보기 위해 그림 2가 보이는 바와 같이 sys_brk() 시스템 콜이 호출하는 함수 전체에 대한 트리를 구성할 수 있다. map_count 변수는 가상 메모리 영역의 개수를 나타내는 int 형 변수로 -2^{32} ~ $2^{32}-1$의 범위를 초과하는 값을 담을 수 없다. 그러므로 그림 1이 보이는 바와 같이 Onion 메커니즘의 첫 번째 단계에 따라 map_count 변수를 취약 가능성이 있는 커널 영역의 변수로 선별할 수 있다. 그림 2가 보이는 시스템 콜 트리를 볼 때, 선별된 map_count 변수로부터 역으로 호출 관계를 추적하였을 때, sys_

brk() 시스템 콜에 이르는 것을 알 수 있다. 그림 2에서 화살표에 기록된 숫자는 상위 함수에서 호출하는 순서이고, 밑줄과 이탤릭체로 표시된 함수는 종단 함수로써 더 이상 다른 함수를 호출하지 않음을 나타낸다.

2.2 사용자로부터 선별된 변수까지 도달하는 경로 선정

2.1 절에서 구성한 sys_brk() 시스템 콜 트리를 이용하여 사용자의 입력이 커널 영역 변수, map_count까지 영향을 주는 경로를 선정한다. 그림 3은 sys_brk() 시스템 콜 트리에서 사용자의 입력이 map_count 변수에 영향을 미치는 단 1개의 경로를 선정한 것을 보인다. 이 경로는 sys_brk()로 시작하여 do_brk()를 지나 vm_link()에 이르는데, vm_link()가 호출되면 map_count 변수의 값이 1만큼 증가할 수 있다.

2.3 사용자의 입력 값 범위 계산

그림 3에서 보이는 바와 같이 sys_brk() 시스템 콜 함수를 호출하면, do_brk() 함수가 호출되고 다시 vma_link() 함수가 호출된다. vma_link() 함수는 호출 될 때마다 map_count 변수의 값에 영향을 주는데, 정적인 상태의 커널 소스에서 map_count 변수는 vma_link() 함수가 한번 호출될 때마다 1만큼씩 증가할 수 있다. 이것을 통해 sys_brk() 시스템 콜 함수가 한번 호출될 때마다 map_count 변수가 1씩 증가할 수 있다는 호출 관계를

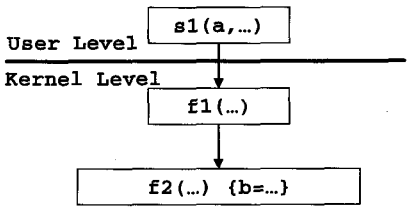


그림 4. 합성함수식으로 구성된 함수 호출관계

알 수 있다. 이 사실을 이용하여 sys_brk() 시스템 콜 함수와 map_count 변수와의 함수식을 구할 수 있다. 함수식은 사용자의 입력 값에 따른 함수 호출 관계를 식으로 표현된다.

그림 3에서 보이는 바와 같이 경로에서 사용자의 입력에 따른 커널 영역 변수의 추이를 구하기 위해서는 첫째로 커널 영역 변수에 영향을 주는 사용자의 입력 값은 무엇인지 정해야한다. 그림 3에서 볼 때, map_count 변수의 변화에 영향을 주는 것은 sys_brk() 시스템 콜 함수의 호출 횟수이다. sys_brk() 시스템 콜의 호출 횟수를 x라고 하고, map_count 변수의 값을 y라고 하면, x의 범위는 $0 \leq x < \infty$, y의 정상적 표현 범위는 $0 \leq y \leq 2^{32}-1$ (map_count는 int 형 변수)이다.

그림 4는 함수 호출 관계를 기반으로 사용자의 입력과 커널 영역 변수와의 관계를 식으로 표현한 것이다. 그림 4에서 보이는 바와 같이 커널 영역 변수를 b라고 정의하고 사용자의 입력을 a라고 정의할 경우, 시스템 콜 함수 호출에 따른 합성 함수

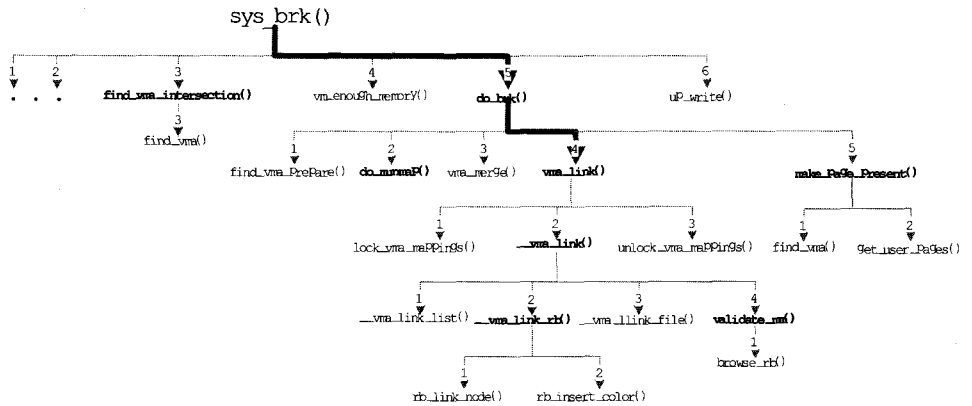


그림 3. 사용자의 입력이 map_count 에 영향을 주는 단 1개의 경로

식은 식 1과 같이 구할 수 있다. 여기서 S1은 유저 애플리케이션 호출 함수인 s1의 함수식이며, F1과 F2는 시스템 콜 트리의 경로 상에 존재하는 호출 함수들의 함수식이다.

$$b = S1 \circ F1 \circ F2(a, \ldots) \quad (1)$$

함수 호출 관계를 기반으로 합성 함수를 구하기 위해서는 다음과 같은 두 가지 전제 조건이 필요하다[24].

- 합성 함수식을 계산할 때, b의 초기 값은 동일함
- 변수의 값은 호출된 함수가 반환될 때 변경됨

첫 번째 조건은 커널 영역 변수의 값을 계산할 때, 변수의 초기 값에 따라 최종 값이 달라지기 때문에 이를 명확히 하기 위한 것이며, 두 번째 조건은 합성 함수식을 계산할 때, 함수값이 반환되는 시점에서 계산이 일어나기 때문에 명시한 것이다. 이 두 가지 전제 조건을 통해 사용자로부터 커널 영역 변수에 이르는 함수들의 호출 관계를 합성 함수 식으로 표현할 수 있다.

식 1에서 구한 합성 함수식을 이용하여 정상적인 함수 호출과 변수 b를 오버플로우할 가능성이 있는 함수의 호출 관계를 그림 5와 같이 정리할 수 있다.

그림 5는 사용자의 입력 범위를 집합 A로 표기하고, 사용자의 입력에 따른 변수의 변화 범위를 집합 B로 표기하였을 경우, 두 집합 사이에 성립하는 관계를 보여주고 있다. 따라서 취약성이 있는 변수들을 오버플로우하기 위해서는 집합 A의 부분집합 중 b를 오버플로우 할 수 있는 값인 P의 범위에 해당하는 사용자의 입력이 있어야 하고, 이를 확인함으로써 리눅스 커널 소스의 변수들이 취약성 여부를 확인할 수 있다.

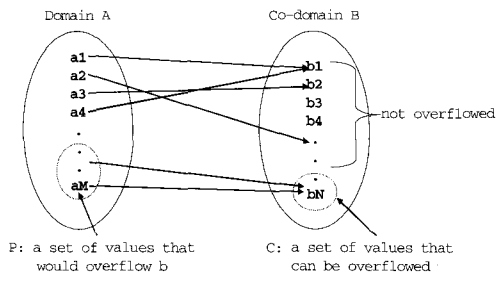


그림 5. 사용자 입력과 커널 영역 변수 사이 관계도

이와 같이 Onion 메커니즘을 적용한 결과 map_count 변수의 경우 sys_brk() 시스템 콜 내에서는 분명히 취약한 변수이다. 다만, 리눅스에서는 가상 메모리 영역을 생성하거나 삭제할 때, -2^{32} ~ $2^{32}-1$ 범위를 초과하는지를 검사하여 초과할 경우 오류 메시지를 보내므로 이 취약성을 이용하여 익스플로잇 할 수는 없다.

Ⅳ. Onion 메커니즘 적용

본 장에서는 2004년 9월 7일 기준으로 Security Focus에 공개된 120가지 커널 소스 취약성을 분석한 결과 중에서 두 가지 취약성, 즉 i_count 오버플로우 취약성, setsockopt MCAST_MSFILTER 정수 오버플로우 취약성에 대해 Onion 메커니즘을 적용한 내용을 보인다[26]. 단, 기 발견된 리눅스 커널 소스의 취약 변수를 사용하기 때문에 취약 가능 변수인지는 알고 있는 상태에서 Onion 메커니즘의 두 번째 단계만을 수행한다.

1. i_count 오버플로우 취약성의 발견 예

본 절에서는 i_count 오버플로우 취약성을 발견하기 위해 Onion 메커니즘을 적용한 결과를 보인다.

1.1 i_count 오버플로우 취약성

리눅스 커널 버전 2.0.33 이하의 버전에서는 inode의 참조 횟수를 의미하는 i_count 필드가 16비트 unsigned short int로 선언되어 있기 때문에 정수 오버플로우를 이용한 공격의 가능성이 존재한다. i_count 변수는 리눅스 커널 변수로서 이를 오버플로우할 경우 다른 프로세스들이 참조하고 있는 동적 라이브러리 파일의 inode가 사용 중인 경우에도 해제 될 가능성이 있으며 공격자는 이 취약성을 이용하여 공격코드를 실행할 수 있다[27].

1.2 Onion 메커니즘 적용 과정

리눅스 커널 버전 2.0.12에서 i_count 변수는 2^{16} 크기의 변수이다. 이는 Onion 메커니즘의 첫 번째 단계에 의해 취약 가능 변수로 판단된다. i

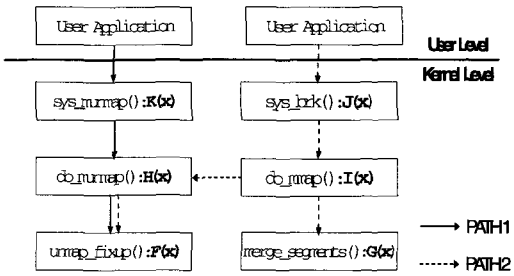


그림 6. 사용자 입력으로부터 i_count 변수까지의 경로

count 변수가 취약 변수인지 정확히 판단하기 위해서는 Onion 메커니즘의 두 번째 단계인 취약 여부 조사 및 검증 단계를 수행해야 한다.

그림 6에서 보이는 바와 같이 i_count 변수를 증가시키는 사용자의 입력 경로는 실선으로 표시된 경로(PATH 1)과 점선으로 표시된 경로(PATH 2), 두 가지 경로로 구분된다. 첫 번째 경로는 sys_munmap() 시스템 콜을 호출하였을 경우이며 두 번째 경로는 sys_brk() 시스템 콜을 호출하였을 경우이다. 그림 7에서 보이는 바와 같이 첫 번째 경로에 나타난 함수 호출 관계를 이용하여 함수식을 구하고, i_count 변수를 오버플로우하기 위한 사용자의 입력, 즉 sys_munmap() 시스템 콜의 호출 횟수의 범위를 구한다[27].

첫 번째 경로(PATH 1)는 그림 7에서 보이는 바와 같이 사용자의 응용 프로그램이 sys_munmap() 시스템 콜을 한번 호출할 때마다 i_count 변수는 T+1만큼씩 증가한다. 여기서 T 값은 한 프로세스가 가지는 가상메모리 영역의 개수를 말하며, 이는 동시에 unmap_fixup() 함수에서 i_count++의 반복횟수를 나타내는 값이다. 한 프로세스가 가지고 있는 가상 메모리 영역의 개수는 1개 이상 6개 이하 이므로 T의 범위는 $1 \le T \le 6$ 이다[27]. 이때 i_count 변수는 unsigned short int 형으로 $2^{16}-1$ 을 초과하는 값을 가지면 오버플로우된다. sys_munmap() 시스템 콜의 호출 횟수(N)와 i_count 값(x)의 관계를 나타내는 함수식을 다음과 같이 정리할 수 있다.

x에 대한 sys_munmap(), do_munmap(), unmap_fixup() 각 함수의 함수식을 $K(x)$, $H(x)$,

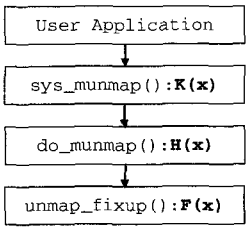


그림 7. 사용자로부터 i_count 변수까지의 도달 경로 (a)

$F(x)$이라고 할 때, 각각의 함수식은 $K(x)=x$, $H(x)=x+T$, $F(x)=x+1$와 같이 표현된다. 이때 세 함수의 함성 합수식은 식 2가 보이는 바와 같다.

$$\begin{aligned} & K \circ H \circ F(x) \\ & = K \circ H(x+1) = K(x+T+1) = x+T+1 \\ & = K(x+T+1) = x+T+1 \\ & = x+T+1 \end{aligned} \quad (2)$$

식 2를 이용하여 sys_munmap()함수를 N번 호출하였을 때 합성함수식을 구하면 식 3이 보이는 바와 같다.

$$\begin{aligned} & K \circ K \circ \ldots \circ K(x+T+1) = K^N(x+T+1) \\ & = x+N(T+1) \end{aligned} \quad (3)$$

식 3에서 보이는 합성함수식 $x+N(T+1)$을 이용하여 사용자의 작성한 응용 프로그램이 i_count 변수를 오버플로우하기 위한 sys_munmap()함수의 호출횟수 N을 구하는 식을 식 4가 보이는 바와 같이 구할 수 있다.

$$x+N(T+1) > 2^{16}-1 \text{ (단, } T=1,\ x=0) \quad (4)$$

식 4를 이용하여 N 값을 계산하였을 때, 초기값이 $T=1$, $x=0$ 이었을 때, N이 2^{16} 이상이면 i_count 변수가 오버플로우된다.

두 번째 경로(PATH 2)는 그림 8에서 보이는 바와 같이 사용자의 응용 프로그램이 sys_brk() 시스템 콜을 한번 호출할 때마다 i_count 변수는 T-1만큼씩 증가한다. 역시 T 값은 한 프로세스가 가지는 가상메모리 영역의 개수를 나타내며, unmap_fixup() 함수에서 i_count++를 반복 실행하는 횟수를 나타내는 값이고, T의 범위는 $1 \le T \le 6$ 이

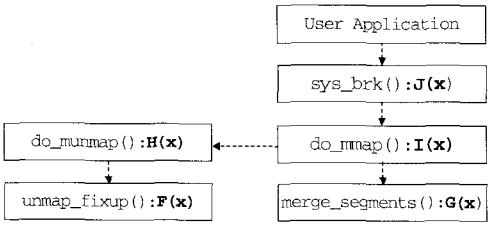


그림 8. 사용자로부터 i_count 변수까지의 도달 경로 (b)

다[27]. 마찬가지로 이때 i_count 변수는 int short 형으로 $2^{16}-1$ 을 초과하는 값을 가지면 오버플로우 된다. 이를 이용하여 이번에는 sys_brk() 시스템 콜의 호출 횟수(N)와 i_count 값(x)의 관계를 나타내는 함수식을 다음과 같이 정리할 수 있다.

x에 대한 sys_brk(), do_mmap(), merge_segment(), unmap_fixup() 각 함수의 함수식을 $J(x)$, $I(x)$, $G(x)$, $F(x)$ 라고 할 때, 각각의 함수식은 $J(x)=x$, $I(x)=x$, $G(x)=x+T-1$, $F(x)=x+T$로 표현된다. 세 함수의 합성함수식은 식 5가 보이는 바와 같다.

$$\begin{aligned} & J\circ I\circ H\circ F(x)+J\circ I\circ G(x) \\ &= J(x+T)+J(x+T-1) \\ &= x+T+x+T-1 \\ &= 2x+2T-1 \end{aligned} \tag{5}$$

또한 sys_brk()함수를 N번 호출하였을 때의 합성함수식은 식 6이 보이는 바와 같다.

$$\begin{aligned} & J\circ \ldots \circ J(x+T)+J\circ \ldots \circ J(x+T-1) \\ &= J^N(x+T)+J^N(x+T-1) \\ &= x+NT+x+NT-N \\ &= 2x+2NT-N \end{aligned} \tag{6}$$

식 6을 이용하면 i_count 변수를 오버플로우하기 위한 sys_brk() 함수의 호출횟수 N을 구하는 식을 식 7과 같이 세울 수 있다.

$$2x+2NT-N>2^{16}-1 \text{ (단, } T=1,\ x=0) \tag{7}$$

식 7을 이용하여 N 값을 계산하였을 때, 초기값이 $T=1$, $x=0$ 이었을 때, N이 2^{16} 이상이면 i_count 변수가 오버플로우된다.

1.3 Onion 메커니즘 적용 결과

리눅스 커널 버전 2.0.12의 커널 소스 중 i_count 변수는 unsigned short int 형으로서 216-1까지의 수를 표현할 수 있는 취약 가능 변수이다. 이 변수는 Onion 메커니즘의 두 번째 단계에 의해 sys_brk() 또는 sys_munmap() 시스템 콜을 이용한 두 가지 경로로부터 사용자의 영향을 받으며 각 경로에서 sys_brk() 또는 sys_munmap() 시스템 콜이 2^{16}번 이상 호출되었을 경우 오버플로우되는 취약성을 가진다.

2. setsockopt MCAST_MSFILTER 취약성 발견 예

본 절에서는 setsockopt MCAST_MSFILTER 정수 오버플로우 취약성을 발견하기 위해 Onion 메커니즘을 적용한 결과를 보인다.

2.1 setsockopt MCAST_MSFILTER 취약성

리눅스에서 multi-source filter 기능을 제공하기 위해 구현된 MCAST_MSFILTER의 경우 사용자 레벨에서 받은 인자에 근거하여 커널 메모리 영역 공간을 할당받는데 사용자 레벨에서 받은 인자에 따라 kmalloc() 함수의 오버플로우를 유도하여 추후 커널 메모리 참조 문제를 발생시킬 수 있다[27]. 이 취약성은 리눅스 커널 버전 2.4.22 이상 2.4.25 이하 혹은 2.6.1 이상 2.6.3 이하의 버전에 존재한다.

2.2 Onion 메커니즘 적용 과정

리눅스 커널 버전 2.4.23에서 사용하는 IP_MSFILTER_SIZE(), GROUP_FILTER_SIZE() 매크로의 반환 값은 각각 32비트 크기의 변수에 담겨지거나 32비트 변수와 크기 비교되는 조건식에 사용된다. 이 두 가지 매크로의 반환 값을 저장하는 변수 및 비교 변수는 Onion 메커니즘의 첫 번째 단계에 의해 취약 가능 변수로 선별 기법에 의하여 취약 가능 변수로 선별 된다. 다음 단계로 Onion 메커니즘의 두 번째 단계인 취약여부 조사 및 검증 단계를 수행한다.

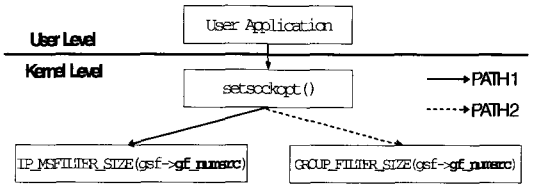


그림 9. 사용자 입력으로부터 IP_MSFILTER_SIZE()와 GROUP_FILTER_SIZE()의 반환 값까지의 경로

먼저 그림 9에서 보이는 바와 같이 사용자로부터 IP_MSFILTER_SIZE()와 GROUP_FILTER_SIZE()의 반환 값 사이의 경로를 구한다.

첫 번째 경로(PATH 1)는 그림 9에서 보이는 바와 같이 사용자의 응용 프로그램이 setsockopt() 시스템 콜을 한번 호출하면 GROUP_FILTER_SIZE() 매크로와 IP_MSFILTER_SIZE() 매크로가 차례로 실행된다. 이때 GROUP_FILTER_SIZE() 매크로와 IP_MSFILTER_SIZE 매크로의 입력 값을 사용자 응용 프로그램으로부터 넘겨받기 때문에 사용자의 입력에 따라 반환 값이 오버플로우될 수 있다.

사용자로부터 넘겨받는 입력 값 numsrc 값을 T, IP_MASFILTER_SIZE()의 반환 값을 x라고 하고, 사용자 응용 프로그램의 함수식을 $H(T)$, setsock opt() 함수의 함수식을 $G(T)$, IP_MS-FILTER _SIZE() 매크로의 함수식을 $F(T)$라고 할 때, 각각의 함수식들은 식 8이 보이는 바와 같다.

$$H(T)=T,\ G(T)=T,$$
$$F(T)=(5\times4)-4+(T\times4)+16$$
$$=4(T+4) \quad (8)$$

식 8은 2^{32}-1을 초과할 경우 오버플로우되기 때문에, 식 8의 내용을 기준으로 IP_MSFILTER_SIZE()의 반환 값을 오버플로우하기 위한 T의 값은 식 9가 보이는 바와 같다.

$$T>2^{30}-4=1{,}073{,}741{,}820 \quad (9)$$

식 9가 보이는 바와 같이 T값이 1,073,741,820 이상이면 IP_MSFILTER_SIZE()의 반환 값은 오버플로우된다.

두 번째 경로(PATH 2)에서 사용자로부터 넘겨받는 입력 값인 numsrc 값을 T, GROUP_FILTER_SIZE()의 반환 값을 x라고 하고, 사용자 응용 프로그램의 함수식을 $H(T)$, setsockopt() 함수의 함수식을 $G(T)$, GROUP_FILTER_SIZE() 매크로의 함수식을 $F(T)$ 라고 할 때, 각의 함수식들은 식 10이 보이는 바와 같다.

$$H(T)=T,\ G(T)=T$$
$$F(T)=268-128+(T\times128)=128T+140$$
$$=4(32T+35) \quad (10)$$

이때 식 10은 2^{32}-1을 초과할 경우 오버플로우 오류가 발생하기 때문에, 식 10의 내용을 기준으로 GROUP_FILTER_SIZE()의 반환 값을 오버플로우하기 위한 T의 값은 식 11이 보이는 바와 같다.

$$T>2^{25}-\frac{35}{32}=335{,}554{,}431 \quad (11)$$

식 11이 보이는 바와 같이 T값이 335,554,431 이상이면 GROUP_FILTER_SIZE()의 반환 값은 오버플로우된다.

2.3 Onion 메커니즘 적용 결과

리눅스 커널 버전 2.4.23의 커널 소스 에서 IP_MSFILTER_SIZE()의 반환 값과 GROUP_FILTER_SIZE() 매크로의 반환 값은 Onion 메커니즘의 첫 번째 단계를 수행한 결과 각각 2^{32} 이하의 크기의 변수에 대입되거나 2^{32} 변수와 비교하는 조건문에 사용되므로 취약 가능 변수로 선별되었고, Onion 메커니즘의 두 번째 단계에 의해 사용자의 입력에 영향을 받는 경로가 존재하는 것과 각 매크로의 입력으로 사용되는 numsrc 값이 각각 1,073,741,820과 335,554,431 이상의 값으로 입력될 경우 오버플로우되는 취약성을 가진다.

V. 결 론

본 논문에서는 리눅스 커널 영역 변수의 알려지지 않은 취약성을 발견하기 위한 발견론으로 Onion

메커니즘을 제안하였다. Onion 메커니즘은 두 단계로 구성되며, 첫 번째 단계는 취약 가능성이 있는 변수를 선별하는 단계이고, 두 번째 단계는 선별된 취약 가능성이 있는 변수의 취약 여부를 검사하는 단계이다. 또한, 제안한 Onion 메커니즘이 적합한지 알아보기 위하여 SecurityFocus에 공개된 리눅스 커널 소스 취약성 중 i_count 오버플로우 취약성, setsockopt MCAST_MSFILTER 정수 오버플로우 취약성에 대하여 Onion 메커니즘을 적용한 결과를 보였다. 이 결과 본 논문에서 제안한 Onion 메커니즘은 정수형 변수에만 적용할 수 있다는 측면에서는 매우 제한적인 발견 방법론일 수 있으나, 정수형 변수로만 구성된 리눅스 커널에서 예상되는 취약성에 대하여는 적용 가능성이 매우 높다.

본 논문에서 제안하는 Onion 메커니즘을 실제로 구현하기 위해서는 각 단계에서 다양한 구문분석도구 및 함수 콜 트리 생성 도구를 사용하여 취약성 발견 방법 과정을 손쉽게 할 수 있다. 예를 들면, 1단계 취약가능 변수의 판별을 위하여 lex[28]나 yacc[29]와 같은 도구를 이용하여 취약가능변수 판별 도구를 구현할 수 있고, 2단계 시스템 콜 트리를 구성하기 위하여 hypersrc[30]과 같은 도구를 사용한 시스템 콜 트리 생성 도구를 구현할 수 있다.

Onion 메커니즘은 기존의 프로그램 오류 검사 도구에서 사용하는 패턴매칭 기술이나 구문분석 기술과 달리 리눅스 커널의 변수 취약성 발견에 한정되어 있지만, 사용자로부터 접근하는 알려지지 않은 입력에 의한 취약성도 발견해 내는 장점이 있다.

본 논문에서 제안한 Onion 메커니즘은 자동화된 리눅스 커널 소스의 취약성 발견 도구 개발에 기여할 것으로 기대되며, 리눅스 커널 취약성에 대한 대응책으로 기대되는 LSM(Linux Security Module) 기술과 접목하여 취약성 발견 및 대응 방법론 연구에 기여할 것으로 기대된다[31].

참 고 문 헌

[1] 장인숙, 남택준, 강정민, 이진석, "공개된 소스레벨 운영체제 취약성 현황 분석," 한국정보처리학회 추계학술발표대회논문집, Vol. 11, No. 2, 2004.

[2] B. Marick, "A survey of software fault surveys," Technical Report UIUCDCS-R-90-1651, University of Illinois at Urbana-Chamaign, Dec. 1990.

[3] I. Arce and E. Leby, "The Rising Threat of Vulnerabilities Due to Integer Errors," IEEE Security & Privacy, pp. 77-82, Jul./Aug. 2003.

[4] C. Salter, O. S. Saydjari, B. Schneier, and J. Wallner, "Toward a Secure System Engineering Methodology," New Security Paradigms Workshop, Sep. 1998.

[5] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and A. J. Offutt, "Maintainability of the Linux Kernel," IEE Proc. Software, 2002.

[6] J. J. Tevis and J. A. Hamilton, "Methods for The Prevention, Detection and Removal of Software Security Vulnerabilities," Proc. of the 42nd annual Southeast regional conference, Huntsville, Alabama, 2004.

[7] J. Viega, J. T. Bloch, T. Kohno, and G. McGraw, "ITS4: A Static Vulnerability Scanner for C and C++ Code," ACM Transactions on Information and System Security, 2000.

[8] ITS4, http://www.cigital.com/its4

[9] RATS, http://www.securesoftware.com

[10] H. Chen and D. Wagner, "MOPS: an Infrastructure for Examining Security Properties of Software," Proc. of CCS'02, Washington, DC, USA, Nov. 18-22, 2002.

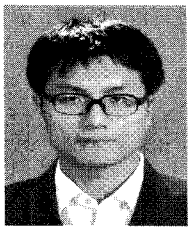
[11] J. S. Foster, M. Fahndrich, and A. Aiken, "A Theory of Type Qualifiers," Programming Language Design and Implementation (PLDI'99), pp. 192-203. Atlanta, Georgia. May 1999.

[12] Secure Programming Lint Specificat-

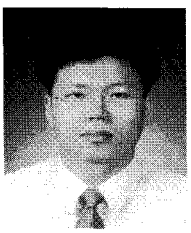
ions Lint, http://www.splint.org
[13] CQual, http://www.cs.umd.edu/~jfoster/ cqual
[14] X. Zhang, A. Edwards, and T. Jaeger, "Using CQUAL for Static Analysis of Authorization Hook Placement," Proc. of the 11th USENIX Security Symposium, San Francisco, California, USA, Aug. 5-9, 2002.
[15] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions," Proc. 4th USENIX OSDI, 2000.
[16] K. Ashcraft and D. Engler, "Using Programmer-Written Compiler Extensions to Catch Security Holes," Proc. of the 2002 IEEE Symposium on Security and Privacy, IEEE Computer Society, Washington, DC, USA, 2002.
[17] R. Dantu, K. Loper, and P. Kolan, "Rist Management using Behavior based Attack Graphs," Proc. of the International Conference on Information Technology: Coding and Computing (ITCC'04), 2004.
[18] 김재광, 고광선, 조은경, 박제호, 강용혁, 장인숙, 엄영익, "취약 원인에 따른 리눅스 커널 취약성 분류법," 한국인터넷정보학회 2004 추계학술발표대회논문집, Vol. 5, No. 2, pp. 127-130, Nov. 2004.
[19] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, "A Taxonomy of Computer Program Security Flaws," ACM Computing Surveys, Vol. 26(3), pp. 211-254, 1994.
[20] M. Bishop and D. Bailey, "A Critical Analysis of Vulnerability Taxonomies," CSE-96-11, Sep. 1996.
[21] K. Jiwnani and M. Zelkowitz, "Maintaining Software with a Security Perspective," International Conference on Software Maintenance(ICSM'02), Montreal, Quebec, Canada, Oct. 03-06, 2002.
[22] T. Jarboui, J. Arlat, Y. Crouzet, and K. Kanoun, "Experimental Analysis of the Errors Induced into Linux by Three Fault Injection Techniques," Proc. of the International Conference on Dependable Systems and Networks (DSN'02), 2002.
[23] J. Viega, J. T. Bloch, T. Kohno, and G. Mcgraw, "Token-Based Scanning of Source Code for Security Problems," ACM Transactions on Information and System Security, Vol. 5, No. 3, pp. 238-261, Aug. 2002.
[24] W. Du and A. P. Mathur, "Testing for Software Vulnerability Using Environment Perturbation," International Conference on Dependable Systems and Networks (DSN 2000), NY, USA. IEEE Computer Society 2000, pp. 603-612, 25-28 Jun. 2000.
[25] M. Bernaschi, E. Gabrielli, and L. V. Mancini, "Operating System Enhancements to Prevent the Misuse of System Calls," Proc. of the 7th ACM conference on Computer and communications security, Athens, Greece, ACM Press, NY, USA, pp. 174-183, 2000.
[26] SecurityFocus, http://www.securityfocus .com
[27] D. P. Bovet and M. Cesati, Understanding the Linux Kernel, O'Reilly, 2003.
[28] S. C. Johnson, "Yacc: Yet Another Compiler Compiler," Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, NJ

07974, 1975.
[29] M. E. Lesk and E. Schmidt, "lex: A Lexical Analyzer Generator," UNIX Programmer's Manual, Vol. 2, pp. 388-400. Holt, Rinehart, and Winston, New York, NY, USA, 1979.
[30] http://www.jimbrooks.org/web/hypersrc/h ypersrc.php
[31] T. Jaeger, A. Edwards, and X. Zhang, "Consistency Analysis of Authorization Hook Placement in the Linux Security Modules Framework," ACM Transactions on Information and System Security (TISSEC) Vol. 7, Issue 2, pp. 175-205, May 2004.

〈著者紹介〉

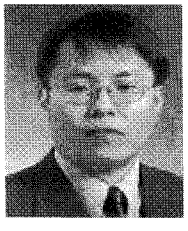


김 재 광 (Jaekwang Kim) 학생회원
2004년 2월: 성균관대학교 정보통신공학부 졸업
2004년 3월~현재: 성균관대학교 컴퓨터공학과 석사과정
〈관심분야〉 시스템 보안, 네트워크 보안, 리눅스



고 광 선 (Kwangsun Ko) 학생회원
1998년 2월: 성균관대학교 정보공학과 졸업
2004년 8월: 성균관대학교 전기전자및컴퓨터공학과 석사
2004년 9월~현재: 성균관대학교 컴퓨터공학과 박사과정
〈관심분야〉 정보보호, 리눅스, 네트워크

강 용 혁 (Yong-hyeog Kang) 정회원
1996년 2월: 성균관대학교 정보공학과 졸업
1998년 2월: 성균관대학교 전기전자및컴퓨터공학과 석사
2003년 8월: 성균관대학교 전기전자및컴퓨터공학과 박사
2003년 3월~현재: 극동대학교 경영학부 정보경영학과 교수
〈관심분야〉 전자상거래, 시스템 보안, 네트워크 보안, 리눅스



엄 영 익 (Young Ik Eom) 종신회원
1983년 2월: 서울대학교 계산통계학과 졸업
1985년 2월: 서울대학교 전산과학과 석사
1991년 8월: 서울대학교 전산과학과 박사
2000년 9월~2001년 8월: Dept. of Info. and Comm. Science at UCI 방문교수
1993년 3월~현재: 성균관대학교 정보통신공학부 교수
〈관심분야〉 분산 컴퓨팅, 이동 컴퓨팅, 이동 에이전트, 시스템 보안, 운영체제,