ISSN 1598-0170 (Print)
ISSN 2287-1136 (Online)
http://www.jksii.or.kr

안드로이드에서 어플리케이션의 메모리 보호를 위한 연구

A memory protection method for application programs on the Android operating system

김 동 율[1] 문 종 섭[2*]
Dong-ryul Kim Jong-sub Moon

요 약

안드로이드 폰이 점점 대중화됨에 따라 많은 어플리케이션이 제작자의 이윤과 직결되는 데이터나 스마트 폰 사용자의 민감한 데이터를 다룬다. 이러한 중요 데이터는 당연히 보호받아야 하지만 안드로이드에서는 악의적인 사용자에 의해 조작되거나 공격자에 의해 유출될 수 있다. 이런 일이 발생하는 이유는 안드로이드의 근간인 리눅스의 디버깅 기능이 악용되기 때문이다. 리눅스의 디버깅 기능을 이용하면 다른 어플리케이션의 가상 메모리에 접근하는 것이 가능하다. 이 기능이 악용되는 것을 방지하기 위해선 해당 기능을 제공하는 주체인 리눅스의 커널에서 기존의 접근 제어를 더욱 강화해야 한다. 하지만 현재 이 기능은 안드로이드 환경의 특성을 반영하지 않은 채 기존의 리눅스와 동일한 접근 제어를 사용하고 있다. 이에 본 논문에서는 리눅스가 제공하는 다른 어플리케이션의 가상 메모리에 접근할 수 있는 기능을 분석하고, 분석 결과와 안드로이드 환경을 고려하여 스레드 그룹 ID를 검증하는 새로운 계층을 추가하는 방안을 제시한다. 이 방안을 적용함으로써 접근 제어를 더욱 강화할 수 있다. 실제로 본 논문이 제안한 방법이 접근 제어를 강화할 수 있는 지 확인하기 위해, 다른 어플리케이션의 메모리를 수정할 수 있는 메모리 조작 어플리케이션으로 자체 제작한 어플리케이션의 데이터 수정을 시도한다. 접근 제어를 강화하기 전에는 메모리 조작 어플리케이션이 자체 제작한 어플리케이션의 메모리에 있는 데이터를 수정할 수 있었지만, 접근 제어를 강화한 후에는 데이터 수정에 실패하는 것을 확인할 수 있다.

☞ 주제어 : 안드로이드 메모리 보호, ptrace 시스템 콜, /proc/PID/mem 가상 파일, 가상 메모리 보호, 메모리 조작 어플리케이션

ABSTRACT

As the Android smart phones become more popular, applications that handle users' personal data such as IDs or passwords and those that handle data directly related to companies' income such as in-game items are also increasing. Despite the need for such information to be protected, it can be modified by malicious users or leaked by attackers on the Android. The reason that this happens is because debugging functions of the Linux, base of the Android, are abused. If an application uses debugging functions, it can access the virtual memory of other applications. To prevent such abuse, access controls should be reinforced. However, these functions have been incorporated into Android O.S from its Linux base in unmodified form. In this paper, based on an analysis of both existing memory access functions and the Android environment, we proposes a function that verifies thread group ID and then protects against illegal use to reinforce access control. We conducted experiments to verify that the proposed method effectively reinforces access control. To do that, we made a simple application and modified data of the experimental application by using well-established memory editing applications. Under the existing Android environment, the memory editor applications could modify our application's data, but, after incorporating our changes on the same Android Operating System, it could not.

☞ keyword : Android memory protection, ptrace system call, /proc/PID/mem virtual file, Virtual memory protection, memory editor

1. 서 론

안드로이드는 리눅스를 근간으로 하는 운영체제로써 어플리케이션(이하 앱)을 하나의 리눅스 프로세스로 관리한다. 따라서 안드로이드의 앱에서 접근하는 가상 메모리는 자신만이 접근할 수 있는 독립적인 공간이어야 하지만, 리눅스의 커널이 다른 프로세스의 가상 메모리에 접근할 수 있는 기능을 제공하기 때문에 안드로이드에서도 제 3의 앱이 다른 앱의 가상 메모리에 접근하는 것이 가능하다. 이것은 중요 데이터의 변조 및 유출로 이어져 앱 개발자에게 재산상의 피해를 입거나 안드로이드 사용자의 민감한 개인 정보가 유출될 수 있음을 의미한다.

[1] GRADUATE SCHOOL OF INFORMATION SECURITY, KOREA UNIVERSITY., Seoul, 136-701, Korea.
[2] ELECTRONICS & INFORMATION ENGINEERING, KOREA UNIVERSITY., Sejong, 339-700, Korea.
[*] Corresponding author (jsmoon@korea.ac.kr)
[Received 10 October 2016, Reviewed 11 October 2016, Accepted 26 October 2016]

http://dx.doi.org/10.7472/jksii.2016.17.6.93

이런 일이 발생하는 가장 큰 원인은 리눅스 커널이 사용하는 다른 프로세스의 가상 메모리에 대한 접근 제어를 안드로이드 환경의 특성을 고려하지 않고 동일하게 사용하기 때문이다. 이에 본 논문에서는 리눅스 커널의 다른 프로세스의 가상 메모리에 대한 접근 제어 체계를 분석하고, 분석 결과를 바탕으로 안드로이드 환경에 맞게 접근 제어를 강화할 수 있는 방안을 제시한다. 또한 제시한 방안의 타당성을 검증하기 위해 실험을 수행한다.

본 논문은 6장으로 이루어져 있다. 2장에서는 안드로이드에서 앱의 메모리를 덤프 하여 개인 정보 및 기타 민감한 데이터를 추출한 기존 연구와 다른 앱의 가상 메모리에 있는 데이터를 변조하는 메모리 조작 앱에 대해 알아본다. 3장에서는 다른 앱의 가상 메모리에 접근할 수 있는 기능에 대해 분석을 수행하며, 4장에서는 3장의 분석을 바탕으로 접근 제어를 강화하는 방안을 제시한다. 5장에서는 제안한 방안의 타당성을 검증하기 위해 메모리 조작 앱을 대상으로 실험을 수행한다. 마지막 6장에서는 결론을 맺음으로써 논문을 마무리한다.

2. 배 경

2.1 관련 연구

최근의 몇몇 연구에 따르면 안드로이드에서 앱의 메모리 영역을 덤프하면 민감한 데이터를 추출할 수 있다고 한다. 이것은 만약 공격자가 앱의 메모리 영역을 덤프하는 악성 앱을 유포한다면, 연구에서와 마찬가지로 민감한 데이터를 추출할 수 있음을 의미한다.

P. Stirparo et al. 은 15개의 은행 앱과 페이스 북, 드랍박스와 같이 사람들이 널리 사용하는 11개의 앱에 대해서 메모리로 부터 개인 정보(credentials)를 얻을 수 있는지를 실험하였다 [1, 2]. 이 실험은 앱이 실행 중일 때와 종료 직후에 수행되었다. 이들의 결과에 따르면 은행 앱의 경우에는 USERNAME과 PASSWORD를 무려 각각 12개(80%), 11개(73.3%)의 앱에서 추출하였으며, 일반적으로 많이 사용하는 앱의 경우에는 USERNAME은 7개(63.6%), PASSWORD는 2개(18.2%)의 앱에서 추출하였다. 특히 은행 앱은 개인 정보를 유출하면 금전적인 피해와 직결되기 때문에 [2]에서 제작한 trojan 앱과 같은 악성 앱이 개인 정보를 유출하지 못 하도록 막을 수 있어야 한다.

Z. Ding et al. 은 중국의 최대 인터넷 기업인 텐센트가 제공하는 모바일 메신저인 WeChat의 메모리를 덤프하면 사용자가 주고받은 메시지나 삭제한 메시지를 얻을 수 있음을 보여 준다 [3]. 특히 이 연구는 WeChat과 같이 데이터를 스마트 폰에 저장할 때에는 암호화를 수행하는 앱도 암호화한 데이터를 사용할 때에는 메모리에서 복호화를 수행하기 때문에 메모리 덤프로부터 중요 데이터를 보호하는 것은 어렵다는 것을 보인다.

안드로이드의 메모리 덤프를 기반으로 한 연구는 안드로이드의 근간인 리눅스의 메모리를 덤프하는 연구를 바탕으로 진행되었다. 리눅스의 메모리를 덤프하는 방법은 크게 2가지가 존재한다. 첫 번째는 물리 메모리를 덤프하는 것이다. 물리 메모리를 덤프하는 방법 중 가장 널리 사용하는 것은 Lime [4]이다. Lime은 Loadable Kernel Module(LKM)로써 리눅스뿐만이 아니라 안드로이드에서도 널리 사용하며, J. Sylve et al. 이 제안한 Android memory acquisition module(DMD)를 확장한 오픈 소스 툴이다 [5]. Lime의 결과물은 메모리 분석 툴인 Volatility [6]와도 연동이 가능하며, [1, 3] 에서도 Lime과 Volatility를 사용하였다. 하지만 Lime은 공격자 입장에서는 사용이 제한된다. 공격자가 Lime을 이용해서 메모리를 덤프하기 위해선 Lime 커널 모듈을 대상 스마트 폰의 리눅스 커널에 삽입해야 하는데, Major 커널은 외부 코드가 커널에 적재되는 것을 막고 있을 뿐만 아니라 모듈을 삽입하고자 하는 커널의 버전에 맞춰서 컴파일 해야 하는 제약사항이 존재하기 때문이다. 다른 물리 메모리 덤프 방법을 살펴보면 예전의 리눅스는 /dev/(k)mem 파일을 이용해서 직접적으로 물리 메모리에 접근할 수 있었다. 하지만 최근의 Major 리눅스는 /dev/(k)mem 파일에 대한 접근을 통제하고 있다. 이에 대한 대안으로 나온 방법이 fmem [7]이다. 그러나 fmem도 Lime과 마찬가지로 커널 모듈을 적재해야할 뿐만 아니라, fmem이 사용하는 일부 커널 함수는 스마트 폰이 사용하는 ARM 아키텍처를 지원하지 않는다.

리눅스에서 메모리를 덤프할 수 있는 두 번째 방법은 프로세스의 가상 메모리를 덤프하는 것이다. 가상 메모리를 덤프하는 방법은 memfetch [8]에서 소개하고 있다. 이 방법은 Lime과 달리 커널 모듈을 적재할 필요가 없고 사용법도 간단하기 때문에 악성 앱들이 충분히 악용할 수 있다. 실제로 P. Stirparo et al. 도 [2]에서 이 방법을 이용하여 trojan 악성 앱을 제작하였다.

2.2 메모리 조작 앱

개발자는 앱을 개발할 때 데이터를 저장하기 위해서 변수를 사용한다. 변수는 사용할 때마다 동적 할당을 하

거나 다시 생성하지 않는 이상 메모리에서 동일한 곳에 위치한다. 메모리 조작 앱은 다른 앱의 가상 메모리에서 특정 데이터를 저장하는 변수 위치를 찾음으로써 데이터를 조작하는 앱이다. 대표적인 예는 Cheat Engine [9], Game Guardian [10], SB Game Hacker [11] 등이 있다.

안드로이드 폰 사용자들이 메모리 조작 앱을 주로 사용하는 곳은 게임 앱이다. 게임 앱의 대부분은 수익을 거두기 위해 부분 유료화 정책을 사용한다. 하지만 사용자가 메모리 조작 앱을 이용하면 게임 내의 금전 및 아이템을 수정할 수 있을 뿐만 아니라 게임 캐릭터와 관련된 각종 주요 데이터를 수정할 수 있다. 게임 서버 측에서 중요 데이터에 대해서 검증을 수행한다면 사용자의 메모리 조작 앱 사용을 예방할 수는 있지만, 게임 상에서 발생하는 모든 데이터를 검증하기란 쉽지도 않을뿐더러 불가피하게 오버 헤드가 발생한다.

3. 가상 메모리 접근 기능에 대한 분석

3.1 가상 메모리

가상 메모리란 운영 체제가 프로세스에게 제공하는 추상화 한 메모리 공간 내에서 실제로 프로세스가 접근할 수 있는 영역을 의미한다. 운영 체제는 가상 메모리를 통해서 프로세스가 물리 메모리의 어느 위치든 적재될 수 있는 특성인 재배치와 다른 프로세스의 메모리 공간을 침범하지 않는 보호 특성을 제공하며, 이상의 두 가지 특성을 바탕으로 사용자가 여러 프로세스를 동시에 실행시키는 것처럼 보이게 하는 멀티태스킹 기능을 지원한다. 즉, 가상 메모리의 가장 큰 특성 중 하나가 각 프로세스 별로 독립적인 추상화 된 메모리의 제공인데, 리눅스에서는 디버깅 등의 편의를 위해 다른 프로세스의 가상 메모리에 접근할 수 있도록 두 가지 기능을 제공한다.

3.2 접근 기능 Ⅰ - ptrace 시스템 콜

첫 번째 기능은 한 프로세스가 다른 프로세스를 추적할 때 사용하는 ptrace 시스템 콜(이하 ptrcae)을 이용하는 것이다. ptrace는 process trace의 약자로써, 다른 프로세스의 실행을 제어할 수 있도록 중지점(break point)을 설정하거나 시그널을 통제할 수 있으며, 다른 프로세스의 메모리 및 레지스터에 존재하는 데이터를 확인하고 수정할 수 있다. 따라서 ptrace는 디버깅 용도로 사용하며, 리눅스의 대표적인 디버거인 GDB에서도 이 시스템 콜을 사용한다.

ptrace는 인자로 전달하는 플래그에 따라 수행하는 역할이 달라진다. 다른 프로세스의 가상 메모리에 접근할 때 사용할 수 있는 플래그로는 4가지가 있다. PTRACE_PEEKTEXT나 PTRAE_PEEKDATA 플래그는 다른 프로세스의 메모리에서 데이터를 읽을 때 사용하고, PTRACE_POKETEXT나 PTRACE_POKEDATA 플래그는 데이터를 쓸 때 사용 한다 [12].

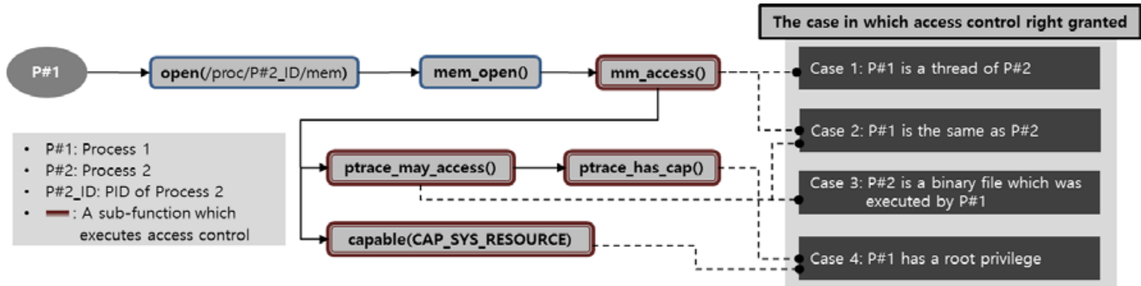
하지만 ptrace를 악용하여 다른 앱의 가상 메모리를 조작하는 것은 한계가 있다. ptrace의 플래그 중에는 PTRACE_TRACEME라는 플래그가 존재한다. 프로세스가 이 플래그로 ptrace를 호출하면 리눅스는 해당 프로세스가 부모 프로세스에 의해 추적당하는 것으로 인식한다. 리눅스에서는 한 번에 하나의 프로세스만이 다른 프로세스를 추적할 수 있기 때문에 자신의 가상 메모리를 보호하려는 앱이 PTRACE_TRACEME 플래그를 이용하면 다른 앱의 추적을 피할 수 있다.

3.3 접근 기능 Ⅱ - /proc/PID/mem 가상 파일

다른 프로세스의 가상 메모리에 접근할 수 있는 두 번째 방법은 /proc/PID/mem 가상 파일(이하 mem 가상 파일)을 이용하는 것이다. 여기서 PID는 리눅스 커널이 프로세스를 관리하기 위해 부여한 ID를 의미한다. mem 가상 파일은 ptrcae로 다른 프로세스의 가상 메모리에 접근하면 한 번에 읽거나 쓸 수 있는 데이터의 양이 unsigned long 크기에 불과하다는 단점을 보완한다. 프로세스가 mem 가상 파일을 이용하기 위해선 일반 파일에 접근하는 것과 동일하게 open(), read(), write() 함수를 사용한다. 하지만 프로세스는 운영체제가 제공하는 메모리 공간의 모든 영역을 가상 메모리로 사용하는 것이 아니기 때문에 mem 가상 파일을 이용하여 전체 메모리 공간을 덤프하면 오류가 발생한다. 따라서 프로세스가 사용 중인 가상 메모리 영역에 대한 정보가 필요한데, 이 정보는 /proc/PID/maps 가상 파일에서 확인할 수 있다.

3.3.1 mem 가상 파일의 접근 체계 요약

안드로이드에서 앱의 mem 가상 파일에 접근하기 위해 open() 시스템 콜을 호출하면 리눅스 커널은 그림 1과 같은 흐름으로 함수를 호출한다. 접근 제어는 mm_access() 함수에서 이루어지며, 이 함수는 mem 가상 파일에 접근하는 프로세스가 권한을 가지고 있는 지를 확인



(그림 1) /proc/PID/mem 가상 파일에 접근할 때 호출되는 함수의 흐름
(Figure 1) The relations of Kernel Calls for accessing the /proc/PID/mem virtual file

한다. mm_access() 함수는 자체에서 수행하는 접근 제어 외에도 ptrace_may_access() 함수와 capable() 함수를 호출하여 접근 제어를 수행하는데, 이를 통과할 수 있는 경우는 그림 1에서 볼 수 있듯이 4가지가 존재한다.

3.3.2 접근 체계 분석 I - mm_access()

mm_access() 함수가 접근 제어를 수행하기 시작한 것은 리눅스 커널 3.2.2이후부터이다. 리눅스 커널 2.6.39부터 3.2.2까지는 권한이 없는 사용자가 mem 가상 파일을 이용하여 루트 권한을 획득할 수 있는 취약점 [13]이 존재했었는데, 이를 보완하기 위해서 mm_access() 함수가 등장하였다. 안드로이드는 4.1까지 리눅스 커널 3.2.2보다 낮은 버전을 사용했었는데, 2016년 9월 17일을 기준으로 [14]에 따르면 이들의 점유율은 10% 미만이다. 따라서 접근 체계 분석은 점유율이 가장 높은 안드로이드 4.4의 리눅스 커널인 3.4를 기준으로 한다. 최근의 스마트 폰은 안드로이드 6.0을 사용하는데, 이들은 대부분 리눅스 커널 3.18을 사용한다. 하지만 이 버전도 접근 체계가 3.4버전과 크게 다르지 않다.

mm_access() 함수는 세 가지 접근 제어를 수행한다. 첫 번째는 mem 가상 파일에 접근 하는 프로세스(이하 접근자)와 mem 가상 파일을 소유하는 프로세스(이하 소유자)의 메모리 공유 여부를 통해 검증하는 것이고, 두 번째와 세 번째는 ptrace_may_access() 함수와 capable() 함수를 호출하는 것이다. 두 함수는 이후의 절에서 살펴볼 것이므로 본 절에서는 첫 번째 접근 제어에 대해서 분석한다.

리눅스 커널은 동작중인 프로세스에 대한 정보를 task_struct 구조체에 저장한다. task_struct 구조체의 내부에는 mm이라는 변수가 존재하는데, 이 변수는 프로세스가 사용 중인 메모리를 관리하는 mm_struct 구조체에 대한 포인터 변수이다. mm_access() 함수에서는 접근자와 소유자의 mm 포인터 변수가 동일한 mm_struct 구조체를 가리키는지를 확인함으로써 접근 제어를 수행한다. mm_strcut 구조체를 공유하는 경우는 자기 자신의 메모리에 접근하는 경우이거나, 프로세스를 복제하는 clone() 시스템 콜을 호출할 때 CLONE_VM 플래그를 지정하는 경우이다. CLONE_VM 플래그를 사용하는 경우는 프로세스가 스레드를 생성할 때이므로, 안드로이드에서 mm_access() 함수의 첫 번째 접근 제어를 통과하는 경우는 앱 자신이나 앱의 스레드가 mem 가상 파일에 접근할 때이다.

3.3.3 접근 체계 분석 II - ptrace_may_access()

mm_access() 함수의 두 번째 접근 제어는 ptrace_may_access() 함수를 통해 이루어지는데, ptrace_may_access() 함수도 세 가지 접근 제어를 수행한다. 안드로이드에서 이 접근 제어를 통과할 수 있는 경우 중 첫 번째는 앱이 자신의 mem 가상 파일에 접근하는 경우이다. 이 접근 제어는 접근자의 task_struct 구조체와 소유자의 task_struct 구조체가 동일한 지를 비교함으로써 이루어진다.

두 번째는 안드로이드 앱이 자식 프로세스에 접근하는 경우이다. 리눅스는 각 프로세스의 보안 정보를 관리하기 위해 task_struct 구조체내에 cred 구조체에 대한 포인터 변수를 저장한다. ptrace_may_access() 함수는 cred 구조체가 가지는 정보 중 이름 공간 [15]과 uid, gid를 이용해서 접근자와 소유자가 부모-자식 관계를 이루는지 확인한다. 하지만 안드로이드에서는 앱이 Dalvik-VM (Virtual Machine)의 인스턴스로 실행될 때 각각 다른 사용자 ID를 가지기 때문에 앱 간에는 부모-자식 관계를 형성할 수 없고, 앱이 실행한 시스템의 바이너리 파일에 대해서만 부모-자식 관계를 형성할 수 있다.

세 번째는 접근자가 시스템 최고 권한인 루트 권한을

가진 경우이다. 이에 대한 검증은 ptrace_has_cap() 함수를 통해 이루어진다. ptrace_has_cap() 함수는 접근자가 소유자에 대해서 CAP_SYS_PTRACE capability를 가지고 있는 지를 확인한다. 안드로이드에선 일반적으로 앱이 다른 앱에 대해 CAP_SYS_PTRACE capability를 소유하지 않는다. 하지만 루트 권한을 얻기 위해 설치하는 su 바이너리 파일은 capability에 제약을 받지 않기 때문에 루트 권한을 가진 앱은 ptrcae_has_cap() 접근 제어를 통과할 수 있다.

3.3.4 capable()

mm_access() 함수의 마지막 접근 제어는 capable() 함수를 통해서 이루어진다. 이 함수는 capability를 인자로 받는데, 시스템 콜을 호출한 프로세스가 해당 capability를 소유하고 있는 지를 확인한다. mm_access() 함수에서는 capable() 함수의 인자로 CAP_SYS_RESOURCE capability를 전달하는데, 이 capability를 기본으로 가지고 있는 프로세스는 루트 권한을 가진 프로세스이다.

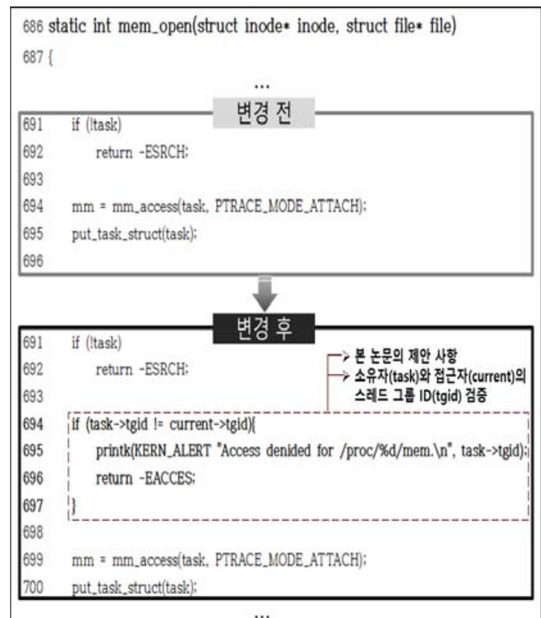
4. 제안 사항

메모리 조작 앱을 이용하는 사용자나 악성 앱이 mem 가상 파일로 다른 앱의 메모리에 접근을 한다면 해당 앱은 ptrcae와 달리 메모리를 보호할 수 있는 방법이 없다. 따라서 본 장의 4.1절에서는 mem 가상 파일의 접근 제어를 강화함으로써 이를 해결할 수 있는 방안을 제시하고, 4.2절에서는 안드로이드 환경의 특성을 고려함으로써 해결 방안의 타당성을 논한다.

4.1 mem_open()의 접근 제어 강화

그림 1에서 보듯이 한 앱의 /proc/PID/mem 가상 파일에 접근할 수 있는 경우는 4가지가 존재한다. 하지만 이 4가지의 경우 중 세 번째 경우와 네 번째 경우는 프로세스가 다른 프로세스의 가상 메모리에 접근하는 경우이기 때문에 독립적인 추상화 된 메모리 공간을 제공하려는 가상 메모리의 개념에 위배된다. 특히 루트 권한을 가진 앱이 다른 앱의 가상 메모리 영역에 접근하는 네 번째 경우는 메모리 조작 앱이나 [2]에서 제작한 trojan 앱에 의해 악용될 수 있다. 이러한 악용 사례를 예방하기 위해선 mem 가상 파일을 제공하는 리눅스의 커널이 접근 제어를 보다 강화해야 한다.

본 논문에서 제안하는 바는 그림 2에서 보듯이 기존의 mem_open() 함수에서 스레드 그룹 ID를 검증하는 계층을 추가하는 것이다. 스레드 그룹 ID란 리눅스에서 한 프로세스가 생성한 스레드를 동일한 그룹으로 관리하기 위해 부여한 ID로써, 리눅스 커널이 프로세스를 관리하기 위해 사용하는 task_struct 구조체 내에 존재한다. 스레드 그룹 ID 검증은 접근자와 소유자의 스레드 그룹 ID를 비교함으로써 이루어지며, 이를 통해 가상 메모리의 개념이 위배되는 것을 막으면서 접근 제어 체계를 강화할 수 있다.



(그림 2) mem_open() 함수의 수정 사항
(Figure 2) Before and after modifying the mem_open() function

4.2 안드로이드 환경의 특성

4.2.1 안드로이드의 디버거

리눅스에서 mem 가상 파일을 사용하는 가장 큰 이유는 디버깅을 위해서이다 [16]. 하지만 이 기능은 [2]에서 소개한 사례나 메모리 조작 앱처럼 다른 용도로도 악용될 수 있기 때문에 리눅스는 LSM [17]이나 YAMA [18]를 통해 합리적으로 제한하려고 노력을 하였다.

하지만 안드로이드의 디버거는 리눅스의 디버거와는 다소 차이가 있다. 안드로이드에서 앱을 개발하기 위해선 구글에서 제공하는 안드로이드 SDK를 사용한다. 안

드로이드 SDK는 개발자에게 앱을 디버깅할 수 있도록 JDWP debugger를 제공하는데, 이 디버거는 리눅스가 제공하는 ptrace이나 mem 가상 파일을 이용하지 않고 JDWP 프로토콜 [19]로 디버깅 기능을 제공한다.

즉, 안드로이드의 리눅스에서 제공하는 디버깅 기능은 안드로이드 환경을 고려하지 못 한 채 과잉 제공되고 있다. ptrace는 안드로이드의 debuggerd [20] 데몬이 처리할 수 없는 예외가 발생한 다른 프로세스의 레지스터와 스택을 덤프할 때 사용하기 때문에 ptrace를 제한하는 것은 바람직하지 않다. 하지만 ptrcae의 경우엔 명백히 앱 스스로가 메모리를 보호할 수 있는 방안이 존재한다. 이에 반해 mem 가상 파일에 대한 접근은 반드시 제한할 필요가 있다. mem 가상 파일은 충분히 악용될 소지가 있지만 메모리를 보호하고자 하는 앱의 입장에서는 ptrace과는 다르게 악성 앱으로부터 자신의 메모리를 보호할 수 있는 방법이 없기 때문이다.

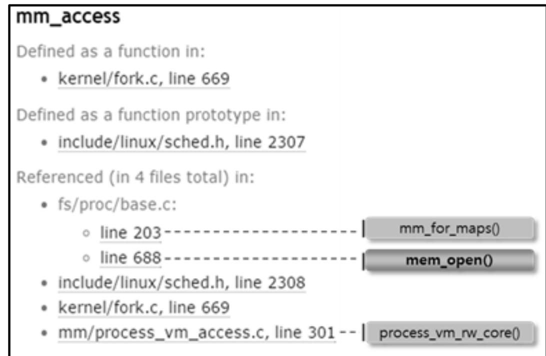
4.2.2 루팅 제한

3장에서 알아본 결과를 바탕으로 시스템의 가장 큰 위협을 제거할 수 있는 또 다른 방법은 루팅을 방지하는 것이다. 루트 권한을 가진 앱이 무분별하게 다른 앱의 가상 메모리에 접근할 경우엔 위험성이 매우 크기 때문이다. 하지만 루팅을 방지하는 것은 쉽지가 않다. 스마트 폰 제조사는 보안을 이유로 루트 권한을 제한한 상태로 안드로이드를 사용자에게 제공하지만, 많은 사용자들이 스마트 폰을 자신의 입맛에 맞게 변형하거나 기기에 이미 설치되어 있던 기본 앱을 제거하는 등의 이유로 루팅을 수행하고 있다 [21]. 안드로이드 루팅 방법은 최근 버전인 안드로이드 6.0을 포함해서 버전별로 인터넷에 공개되어 있기 때문에 사용자는 손쉽게 자신의 스마트 폰에서 루트 권한을 획득할 수 있다. 또한, 안드로이드의 하위 버전에서는 악성 앱이 정상 앱으로 위장하여 강제로 루트 권한을 획득한 사례도 존재 한다 [22]. 사용자의 메모리 조작 앱 남용을 방지하고 악성 앱의 가상 메모리 유출 및 조작을 예방하기 위해선 단순히 루팅을 방지하는 것이 아닌 좀 더 근본적인 대책이 필요하다.

4.2.3 mm_access() 함수 수정

루팅 방지 외에도 mem 가상 파일에 대한 접근 제어를 수행하는 mm_access() 함수를 수정하는 방법도 고려해 볼 수 있다. 하지만 이 방법 또한 적절하지 않다.

그림 3은 리눅스 크로스 레퍼런스 사이트 [23]에서 mm_access() 함수를 검색한 결과이다. mm_access() 함수를 호출하는 곳이 3곳이나 되기 때문에 만약 mm_access() 함수를 수정한다면 예기치 못한 결과를 초래할 수 있다.



(그림 3) mm_access()를 호출하는 함수 목록
(Figure 3) The list of functions which calls mm_access()

5. 실 험

5장에서는 메모리 조작 앱을 이용하여 자체 제작한 앱의 메모리 수정을 시도한다. 이를 통해 본 논문이 제안하는 사항이 앱의 가상 메모리를 보호할 수 있음을 보인다.

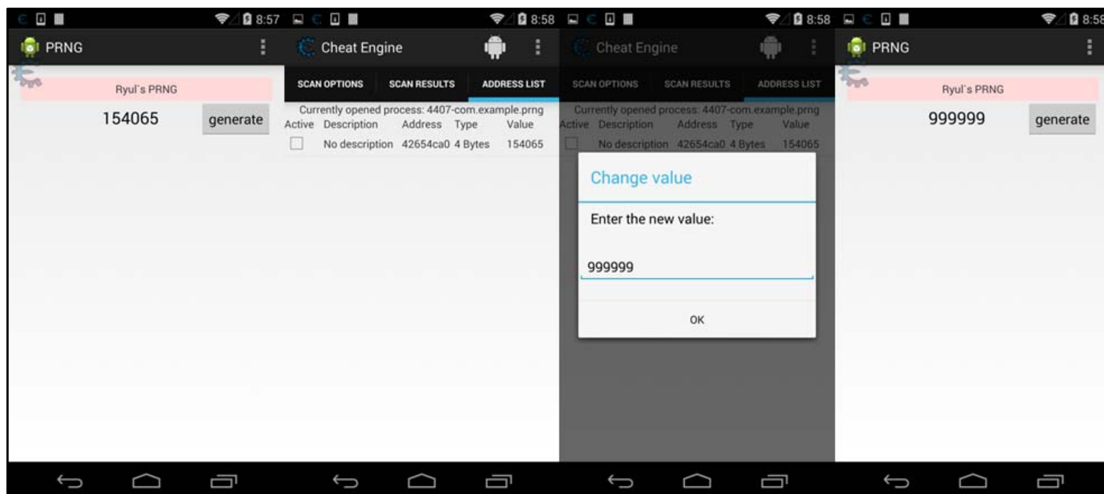
5.1 실험 환경

실험은 [14]에서 볼 수 있듯이 점유율이 가장 높은 안드로이드 4.4와 리눅스 커널 3.4.0을 적재한 Nexus5 레퍼런스 폰에서 진행한다. 메모리 조작 앱이 메모리를 수정할 대상 앱은 자체 제작한 앱으로써, 이 앱은 generate 버튼을 누르면 랜덤 숫자를 출력만 할 뿐 사용자로부터 입력 값을 받을 수는 없다. 메모리 조작 앱은 앞서 소개한 3개의 앱을 이용한다. 또한 3개의 메모리 조작 앱이 실제로 mem 가상 파일을 이용하는지 확인하기 위해 fs/proc/base.c에 존재하는 mem_open() 함수와 mem_read() 함수에서 mem 가상 파일 소유자의 PID를 로그로 남기도록 printk() 함수를 이용하여 수정한다.

5.2 실험 결과

5.2.1 실험용 앱의 메모리 조작

실험은 4단계로 진행한다. 첫 번째 단계에선 루팅을



(그림 4) PRNG앱의 가상 메모리 수정 결과
(Figure 4) The result of modified PRNG app

수행하며, 두 번째 단계에선 실험용 앱을 실행하고 generate 버튼을 눌러 조작할 데이터를 생성한다. 세 번째 단계에선 메모리 조작 앱으로 실험용 앱의 가상 메모리에서 조작할 데이터를 검색하며, 마지막으로 해당 데이터의 조작을 시도한다.

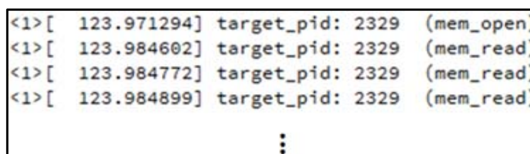
위의 순서로 실험을 진행한 결과는 그림 4와 같다. 실험용 앱의 generate 버튼을 누르자 154065라는 값을 출력한다. 이 값을 메모리 조작 앱인 Cheat Engine으로 검색하면 이 값은 실험용 앱의 가상 메모리에서 0x42654ca0에 위치하는 것을 알 수 있다. 이 값을 999999로 수정하면 실험용 앱은 입력을 받을 수 없음에도 불구하고 999999를 출력한다.

메모리 조작 앱의 mem 가상 파일을 이용 여부를 파악하기 위해 /proc/kmsg 커널 로그를 확인하면 그림 5와 같다. 로그 상의 2329 PID를 가지는 앱을 확인하면 그림 6과 같다.

5.2.2 제안 사항을 적용한 후의 메모리 조작 시도

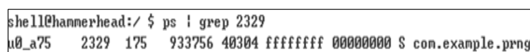
본 논문이 제안한 사항이 실제로 접근 제어를 강화하는지 확인하기 위해 스레드 그룹 ID를 검증하는 코드와 검증에 실패 했을 시에 커널 로그를 남기는 코드를 mem_open() 함수에 삽입한 후에 5.2.1절과 동일한 실험을 수행한다.

실험 결과는 그림 7에서 확인할 수 있듯이 메모리 조작 앱이 6470 PID를 가지는 앱의 가상 메모리 접근에 실패한 것을 알 수 있다. PID가 6470인 앱을 확인하면 그림 8과 같다.



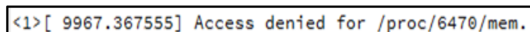
```
<1>[   123.971294] target_pid: 2329  (mem_open)
<1>[   123.984602] target_pid: 2329  (mem_read)
<1>[   123.984772] target_pid: 2329  (mem_read)
<1>[   123.984899] target_pid: 2329  (mem_read)
                    :
```

(그림 5) 메모리 조작 앱의 커널 로그 내역
(Figure 5) The result which shows that /proc/PID/mem was accesses by the memory editor



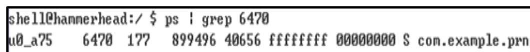
```
shell@hammerhead:/ $ ps | grep 2329
u0_a75    2329  175   933756 40304 ffffffff 00000000 S com.example.prng
```

(그림 6) 메모리 에디터의 커널 로그 내역
(Figure 6) The PID of PRNG is shown as 2329



```
<1>[ 9967.367555] Access denied for /proc/6470/mem.
```

(그림 7) 접근 제어를 강화한 후의 메모리 조작 결과
(Figure 7)The error message shown when accessing the /proc/PID/mem after modifying mem_open())



```
shell@hammerhead:/ $ ps | grep 6470
u0_a75    6470  177   899496 40656 ffffffff 00000000 S com.example.prng
```

(그림 8) PID가 6470인 앱의 정보 확인
(Figure 8)The log left on /proc/kmsg when error occurred after re-enforcing mem_open())

6. 결 론

본 논문은 안드로이드에서 리눅스의 디버깅 기능이 악용될 수 있음을 소개하고 이 기능에 대한 분석을 수행한 후에, 가상 메모리의 개념에 가장 적합한 방법으로 접근 제어를 강화하는 방안을 제시하였다. 물론 본 논문의 제안 사항을 사용하지 않더라도 보안 솔루션을 이용해서 리눅스의 디버깅 기능이 악용되는 것을 제한할 수 있다. 하지만 해당 기능을 악용하는 앱의 제한을 보안 솔루션에만 의존해선 안 된다. 보안 솔루션은 모든 사용자가 사용한다는 보장이 없을뿐더러 메모리 조작 앱과 같이 자신의 부당한 이익을 추구하는 사용자는 보안 솔루션을 제거해 버릴 수 있기 때문이다. 리눅스는 우리가 익히 알고 있는 것처럼 수많은 기능을 제공한다. 이 기능들 중에는 본 논문이 소개한 디버깅 기능 외에도 사용자와 앱 개발자를 위협할 수 있는 다른 기능이 존재할 수 있다. 따라서 안드로이드 환경에 위협을 가할 수 있는 리눅스의 기능에 대한 연구는 보다 활발하게 진행되어야 한다.

참 고 문 헌 (Reference)

[1] P. Stirparo, I. N. Fovino, and I. Kounelis, "Data-in-use leakages from Android memory-Test and analysis," 2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), Oct. 2013, pp. 701-708. http://dx.doi.org/10.1109/WiMOB.2013.6673433

[2] P. Stirparo, I. N. Fovino, M. Taddeo, and I. Kounelis, "In-memory credentials robbery on android phones," 2013 World Congress on Internet Security (WorldCIS), Mar. 2014, pp. 88-93. http://dx.doi.org/10.1109/WorldCIS.2013.6751023

[3] F. Zhou, Y. Yang, Z. Ding, and G. Sun, "Dump and analysis of Android volatile memory on Wechat," 2015 IEEE International Conference In Communications (ICC), Sep. 2015, pp. 7151-7156. http://dx.doi.org/10.1109/ICC.2015.7249467

[4] 504ensicsLabs/LIME, https://github.com/504ensicslabs/lime

[5] J. Sylve, A. Case, L. Marzlale, and G. G. Richard, "Acquisition and analysis of volatile memory from android devices," Digital Investigation 2011, Vol. 8, no. 3, Feb. 2012, pp. 175-184. http://dx.doi.org/10.1016/j.diin.2011.10.003

[6] volatility, https://code.google.com/p/volatility/wiki/LinuxMemoryForensics

[7] I. Kollar, "Forensic RAM dump image analyser," Master's Thesis, Charles University in Prague, 2010.

[8] lcamtuf-memfetch, https://github.com/citypw/lcamtuf-memfetch

[9] Cheat Engine, http://www.cheatengine.org/

[10] GAMEGUARDIAN, https://gameguardian.net/

[11] SB Game Hacker, http://m.balifornia.store.aptoide.com/app/market/org.sbtools.gamehack/40/3882874/SB+Game+Hacker

[12] PTRACE(2), http://man7.org/linux/man-pages/man2/ptrace.2.html

[13] Linux local privilege escalation via suid/proc/pid/mem write, https://git.zx2c4.com/CVE-2012-0056/about/

[14] Android (operating system) https://en.wikipedia.org/wiki/Android_(operating_system)#Platform_usage

[15] cgroups, https://en.wikipedia.org/wiki/Cgroups#NAMESPACE-ISOLATION

[16] ptrace. https://en.wikipedia.org/wiki/Ptrace#Limitations

[17] LSM, https://www.kernel.org/doc/Documentation/security/LSM.txt

[18] Yama, https://www.kernel.org/doc/Documentation/security/Yama.txt

[19] Java Debug Wire Protocol, http://docs.oracle.com/javase/6/docs/technotes/guides/jpda/jdwp-spec.html

[20] Debugging Native Android Platform Code, http://source.android.com/devices/tech/debug/#debuggerd

[21] Rooting (Android OS), https://en.wikipedia.org/wiki/Rooting_(Android_OS)#Advantages

[22] H. W. Lee, "Android based Mobile Device Rooting Attack Detection and Response Mechanism using Events Extracted from Daemon Processes," Journal of The Korea Institute of Information Security & Cryptology(JKIISC)

2013, Vol. 23, No. 3, Jun. 2013, pp. 479-490.
http://dx.doi.org/10.13089/JKIISC.2013.23.3.479
[23] Linux Kernel Cross Reference,
http://lxr.oss.org.cn/plain/ident?v=3.4.9&a=arm&i=mm_access

◐ 저 자 소 개 ◑

김 동 율 (Dong-ryul Kim)
2015년 홍익대학교 컴퓨터공학과(공학사)
2015~현재 고려대학 정보보호학과 석사과정
관심분야 : 임베디드 보안, 시스템 보안, etc.
E-mail : pixxi242@naver.com

문 종 섭 (Jong-sub Moon)
1981년 서울대학교 계산통계학과(공학사)
1983년 서울대학교 대학원 계산통계학과(공학석사)
1991년 Illinois Institute of Technology 전산학과(공학박사)
2002~현재 고려대학 전자 및 정보공학과 교수
관심분야 : 정보 보안, 패턴 인식, 의공학 etc.
E-mail : jsmoon@korea.ac.kr