Rooting(hw3)

1. 준비단계
2. 첫번째 루팅
3. Dev/mem 의 open
4. Memory base 0x40000000 과 page offset 0xc0000000
5. Code 분석

* Kallsyms 를 통해서 setresuid 주소값 알아내기

1. Setresuid 변형

* setresuid함수
* 왜 첫 번째 if문에 걸리지 않는가?
* 0xe3500000 자세히 알아보기
* Ns\_capable 함수

1. 또다른 system call patch
2. 한단계 더
3. 실패이유 분석
4. 눈높이를 낮춘 또 다른system call patch
5. 결과
6. Setresuid를 이용한 rooting 화면 C0041e08
7. Setuid를 이용한 rooting 화면 C0041d40

시스템 프로그래밍 유혁 교수님

정보대 컴퓨터학과 2010210107 홍보의

2014.12.19

1. 준비단계

준비단계에서 android 개발자 모드를 켜지 않아서 adb가 연결되지 않는 상황이 일어났었다. 개발환경 준비에서는 수많은 문제들이 발생 할 수 있다. 내가 개발환경을 구성하는 것이 아니라면, 결국 다른 사람이 구성해 놓은 환경을 재구성 하는 것이기 때문에, 매뉴얼을 확실히 숙지하는 것이 중요하다는 생각을 했다.

조금은 고생했지만, android 컴파일에 대해서 조금이나마 감을 잡아보는 기회가 되었다. 이론으로만 배웠던 linux 커널과 그 위에 구성되는 android middle ware의 실체에 대해서 알 수 있었다. 제대로 하지는 못했지만 많은 고민을 할 기회가 되었다.

1. 첫번째 rooting
2. /dev/mem 의 open

Dev/mem을 open 한다는 것. Dev/mem이라는 것은 RAM에서 사용하고 있는 정보를 debugging 목적으로 볼 수 있도록 해 놓은 것이다. 이 파일의 권한이 666으로 되어 있어서 감사하게도 read write 를 root가 아님에도 할 수 있다. 조교님의 코드를 받기 이전에 가장 먼저 고민하게 된 것은 dev/mem을 open하고 read 했을 때 무엇이 나타나는 가 였다. Procfs 를 만들어 보았으니, 이에 대한 추론은 쉽게 할 수 있었다. 프록시 파일에 open read write 명령어가 각기 정의되어 있듯이. Dev/mem 파일에 대해서도 read write open 동작들이 정의되어 있을 것이라 생각되었다.

|  |
| --- |
| [**devlist**](http://lxr.free-electrons.com/ident?i=devlist)**[] = {**  **[785](http://lxr.free-electrons.com/source/drivers/char/mem.c" \l "L785) [1] = { *"mem"*, 0, &[mem\_fops](http://lxr.free-electrons.com/ident?i=mem_fops), &[directly\_mappable\_cdev\_bdi](http://lxr.free-electrons.com/ident?i=directly_mappable_cdev_bdi) },** |

위와 같이 /drivers/char/mem.c에는 여러가지 char형 메모리 접근 디바이스 드라이버가 정의되어 있었다. 위는 “mem” 에 mem\_fops operation이 정의되어 있는 모습이다. 해당 커널의 경우에는 이 “mem”의 접근권한이 0666으로 되어 있었을 것으로 예상된다. 위의 “mem” 다음에 나오는 0 이 접근권한을 뜻하는데, 이것이 새로 만든 system image 의 경우에 0666으로 할당 되어 있을 것으로 예상된다. 본래 가지고 있었던 linux-odroid-3.4.y 를 보면 0으로 되어 있는데, 아마 system 이미지 파일을 바꾸면서 바뀌는 것이 아닌가 싶다. 위의 파일의 행동을 정의하는 Mem\_fops는 같은 파일에 정의되어 있다.

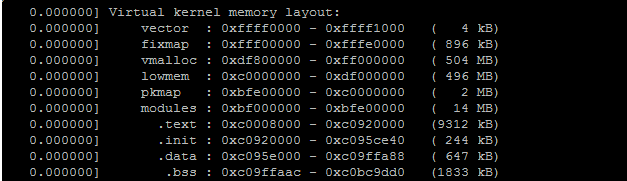
|  |
| --- |
| static const struct file\_operations mem\_fops = {  .llseek = memory\_lseek,  .read = read\_mem,  .write = write\_mem,  .mmap = mmap\_mem,  .open = open\_mem,  .get\_unmapped\_area = get\_unmapped\_area\_mem,  }; |

mmap는 mmap\_mem으로 정의되어 있다. Mmap\_mem도 같은 파일 내에 정의되어 있다.

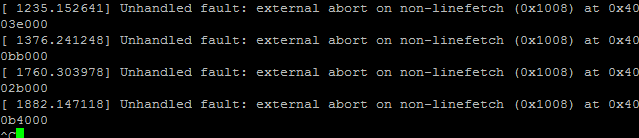
|  |
| --- |
| static int mmap\_mem(struct file \*file, struct vm\_area\_struct \*vma)  {  size\_t size = vma->vm\_end - vma->vm\_start;  if (!valid\_mmap\_phys\_addr\_range(vma->vm\_pgoff, size))  …  if (!private\_mapping\_ok(vma))  return -ENOSYS;  if (!range\_is\_allowed(vma->vm\_pgoff, size))  return -EPERM;  if (!**phys\_mem\_access\_prot\_allowed**(file, vma->vm\_pgoff, size,  &vma->vm\_page\_prot))  return -EINVAL;  vma->vm\_page\_prot = phys\_mem\_access\_prot(file, vma->vm\_pgoff,  size,  vma->vm\_page\_prot);  **vma->vm\_ops = &mmap\_mem\_ops;**  /\* Remap-pfn-range will mark the range VM\_IO and VM\_RESERVED \*/  if (remap\_pfn\_range(vma,  ….  }  return 0;  } |

모든 동작을 확인 할 수는 없지만, 뭔가 vma라는 단어가 계속 나오는 것으로 보아 혹시 virtual memory 의 주소를 가져와서 그것을 memory에 올리고 이를 리턴해 주는 것인가 싶었다. 하지만 그렇다면, 물리 메모리를 디버깅 한다는 dev/mem의 의미가 무엇인지 궁금했다. 물리 메모리에 올라와 있는 자체를 볼 수 있어야 하는 것 아닌가 싶었다. 조사결과 붉은색으로 표시한 phys\_mem\_access\_prot\_allowed 함수를 통하여 mem에 접근하는 권한을 얻어오고 다음 mmap\_mem\_ops주소를 할당해 준다. 이 mmap\_mem\_ops가 바로 유저공간에 커널 메모리를 매핑하는 함수가 정의되어 있는 부분이다. 동작들이 어떻게 되고 있는지 또한 결과적으로 어느 offset부터 어느 length 까지를 mmap 해야 하는지 알 수가 없었다.

일단은 생각대로 virtual memory의 주소 공간 내에서 kernel 이 존재하는 부분을 mmap 해 오겠다는 생각을 하였다. Odroid가 부팅 될 때 제일 처음화면을 보면 다음과 같은 부분을 확인 할 수 있다.

****

번개처럼 지나가는 부팅화면에서 해당 부분을 본 나는 virtual kernel memory? 이거다. 싶었다. 그래서 커널의 text가 존재한다는 0xc0008000 에서부터 0xc0920000 만큼을 맵핑 하려고 하였다. 하지만 bus error 가 났다.



그리고 시리얼 창에는 다음과 같은 에러가 발생하였다. 무언가 mmap 영역이 잘못된 것 같았다. 장시간 고민을 했지만 해결을 보지 못하고 조교님의 code 강의를 기다렸다.

1. Memory base 0x40000000 과 page offset 0xc0000000

많은 고민에 쌓여있던 나는 코드를 받고 0x40000000 영역부터 mapping 을 한다는 데에서 많은 놀라움을 받았다. 이 영역은 메모리가 매핑되어 있는 주소공간이다. 내가 1GB 램을 사용하던지 4GB 램을 사용하던지 0x40000000 으로부터 메모리가 매핑되어 있다. 이 영역의 정의는 /arch/arm/configs/odroidxu-deconfig 파일에 되어 있다. 파일 상단에

# Automatically generated file; DO NOT EDIT.

라고 되어 있는 것으로 보아 이 파일은 커널 컴파일 시에 커널 형성 정보를 기록해 놓은 파일 으로 추정된다. 이곳에 exynos\_mem\_base라고 선언되어 있는 부분이 물리메모리를 매핑해 놓은 주소 공간이다.

CONFIG\_EXYNOS\_MEM\_BASE=0x40000000

다음과 같이 선언되어 있다. 좀 더 자세히 알아보면, 이것은 수업시간에 배운 MMAP I/O의 다름 아니다.

다음과 같이 각각의 기기에 대해서 주소가 할당이 되어 있고, 해당 주소에의 접근은, 해당 기기의 레스지터와 road/store 명령을 주고받는 것을 의미한다. 위의 그림은 LPC214x 프로세서의 VLSI peripheral bus address range 안에 있는 모든 기기들의 주소가 할당된 모습이다. 이러한 peripheral 들은 configure 해 줄 수 있고, 기기에서 사용하는 레지스터들에 접근 할 수 있다. RAM은 0x40000000에 매핑이 되어 있는 것이고 이 주소로의 접근은 RAM과 데이터를 주고 받을 수 있게 한다. General purpose RAM의 경우에는 위의 표에서 보는 것과 같이 0x40000000 으로부터 0x40003FFF 까지 되어있다. 내가 접근하고자 하는 어떠한 데이터던지 이 범위 안에서 접근 가능하다. 0x4000은 4KB를 의미하는 공간이다. 이 공간에 RAM의 모든 데이터가 들어간다는 말은, 이 공간을 이용하여 RAM의 모든 address에 접근 할 수 있다는 말로 이해하였다. 하지만 의문이 들었다.

|  |
| --- |
| len = page\_size \* page\_size; |

다음은 kallsyms 소스를 찾기 위하여 0x40000000으로부터 mmap을 하는 공간의 길이이다. 페이지 사이즈가 4KB 인데 왜 16MB만큼의 주소를 mmap 하는 걸까? Dev 파일의 mmap의 정의를 해석하지 못해서 그 동작을 정확히 알지는 못하겠다. 하지만 그 결과로 mmap이 리턴하는 주소는 0x40000000 ( RAM의 베이스 주소) 로부터 16MB를 mmap된 공간의 주소일 것 이다.

이후에 setresuid systemcall의 주소를 kallsyms procfs를 통해서 찾게 되는데, 이때 얻게 되는 주소는 virtual memory 기준의 주소이다. Virtual memory에서 page가 할당되기 시작하는 부분을 page\_offset이라고 하는데, 이 주소 또한 같은 파일에 선언되어 있다.

CONFIG\_VMSPLIT\_3G=y

CONFIG\_PAGE\_OFFSET=0xC0000000

위와 같이 선언되어 있는데, virtual memory에서 커널 이미지의 시작 지점을 0xc0000000으로 한다는 의미이다. 이것이 바로 커널이 virtual address로 물리 메모리를 보는 관점의 시작지점이다. 즉, 물리 메모리는 가상 메모리 상에서 0xc0000000 이 스타팅 포인트 이고 이로부터 위로 매핑 한다. 부팅시에 보았던 virtual kernel memory layout 이란 바로 이것에 대한 정보가 나열되어 있는 것이었다. 이 부분부터 위로(주소 값이 증가하는 방향으로), 페이지 테이블이 존재하고 커널 소스도 존재한다.

1. Code 분석

소스코드에서는 먼저 /dev/mem 을 open 한 이후에 특정 영역을 mmap을 한다.

|  |
| --- |
| dev\_fd = open("/dev/mem",O\_RDWR);  hw3\_map = (unsigned long\*)mmap(NULL,len,PROT\_READ | PROT\_WRITE,MAP\_SHARED,dev\_fd,KERNEL\_ADDR))==MAP\_FAILED) |

위의 open 에서도 처음에 고생을 조금 했는데, 실패하면 NULL( 0 ) 값일 것이라고 가정하고 0일 경우 바로 종료를 시켰다. 그러고 계속 open이 안 된다고 고민 하였다. 실패시 -1을 리턴하고 성공시 FILE 구조체의 주소를 리턴하는 함수 open 이다.

아래의 mmap에서 첫번째 인자는 mmap을 한 이후에 가져온 데이터를 저장할 메모리 공간을 의미한다. Mmap을 해서 특정 영역을 가져왔다면 그곳에 접근할 주소가 필요 할 것이다. (물론 virtual memory address) 이때 이 주소를 정해 줄 수도 있지만, kernel에게 알아서 정하고 시작 주소를 return 요청 할 수 있다. 이 kernel에게 정해달라 요청하는 것이 0을 할당 했을 때의 작동이다. 이에 대한 내용은 mmap의 linux man page 에 나와있는 정보를 이용했다. 두 번째 인자는 mapping 을 할 영역의 크기이다. 이 영역의 크기는 BYTE 단위 이지만 PAGE단위의 배수로 할당해야 한다. 페이지의 크기는 sysconfig(\_SC\_PAGE\_SIZE)로 확인할 수 있다. 이 length는 마지막 변수인 offset 으로 부터의 값이다. 세번째 인자는 mmap 되어서 접근 가능하게 된 주소의 권한을 설정한다. PRTO\_EXEC(실행) PROT\_READ(읽기) PROT\_WRITE(쓰기) PROT\_NONE(암것도 x) 등의 권한 설정이 가능하다. | 기호를 통해서 중복설정이 가능하다. 그 다음의 flag 값에는 대표적으로 MAP\_SHARED MAP\_PRIVATE 의 설정이 가능한데, 이를 통해서 다른 process에서 mmap된 영역에 접근이 가능한지 아닌지를 설정 할 수 있다. 그다음 fd는 mmap을 할 file의 file descripter이다. 현재는 /dev/mem 파일이 된다. Offset은 위에서 설명 했듯이 시작 주소의 값이다. A에서 살펴 보았듯이 /dev/mem 파일의 경우 mmap의 동작이 따로 정의되어 있다. 리눅스의 file 이라는 개념이 얼마나 광범위하게 사용되는지를 다시 한번 확인 할 수 있었다.

이렇게 mmap을 한 이후에 /proc/kallsyms 로부터 주소를 보기 위하여 kallsyms 파일의 %pK %c %s 를 찾는다. %pK %c %s 를 아스키 코드값을 확인해 보면은 0x25704B20 0x25632025 0x73 과 같다. Ppt의 설명과 같이 이 값들은 little endian을 사용하는 odroid에 의하여 32bit 즉 4byte 씩 나누어져서 거꾸로 입력되어 있다. 즉 메모리에는 0x25704B20가 0x204B7025 그리고 0x25632025가 0x25206325 로 바뀌어 입력되어 있다. 그 이후에 %c의 경우 0x73의 값을 갖는데, 소스코드를 보면은 0x730a0000 의 갑을 찾고 있다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0x204b7025 == | 25(%) | 70(p) | 4b(K) | 20( ) |
| 0x25206325 == | 25(%) | 63(c) | 20( ) | 25(%) |
| 0x00000a73 == | 73(s) | **0a(line feed)** | 00 | 00 |

0x0a 의 경우 아스키 코드로 linefeed 를 의미하는데 왜 line feed 가 들어가 있는지 알 수 없었다 다음에는 따움표가 나오는 것이 정상인데 말이다. 그리고 뒤에 00 00 두 개의 null은 또 무엇을 의미하는지 알 수가 없었다. 또한 i를 4씩 증가 시키는 이유는, len의 경우는 byte의 개수이지만 temp\_addr 은 usigned long \* 즉 4byte의 값을 갖기 떄문에 한번 temp\_addr을 증가시키면 4byte씩 증가하는 현상이 일어나기 때문이다.

위의 문자열이 매칭이 되면은 temp\_addr이 가리키는 4byte를 마찬가지로 little endian 방식으로 바꿔서 입력해 준다. 0x20 20 70 25 를 입력하면 K가 20으로 덮어 써 진다. %pK가 되어 있으면 왜 0이 출력될까? %pK는 /proc/sys/krenel/ kptr\_restrict 의 값을 확인해서 이 값이 0이면 %p에 대해서 0을 출력하는 역할을 한다. 이를 그냥 %p로 바꾸었으니, 항상 주소값을 출력하게 된다.

* Kallsyms 를 통해서 setresuid 주소값 알아내기

|  |
| --- |
| str = strtok(ptr, " "); //tokenize  addr\_sym = strtoul(str,NULL,16); //주소값 저장. 주소가 제일먼저 나온다  index = 1;  while(str) {  str = strtok(NULL," "); // tokenize  index++; // 두번째는 심볼타입인데, 사용할 필요 없다.  if(index == 3) {  if(strncmp("sys\_setresuid\n", str,14) ==0) found = true; //마지막 심볼명과 매칭확인!!  break; |

이 부분은 kallsyms의 출력을 공백 기준으로 나누어서 “주소 / 심볼타입 / 심볼명” 형식을 통해서 출력해준다. 이때 첫 번째 공백 이전에 존재하는 주소를 저장해 논 다음에, 두 번째 단어인 심볼타입은 스킵하고 심볼명과 sys\_setresuid와 체크를 해서, 맞으면 true 아니면, 계속 다음 줄로 넘어가서 체크한다.

결국 찾아낸 sys\_setresuid 함수의 주소는 virtual memory 에서의 커널 접근을 위한 주소로서 위에서 설명했듯이 page\_offset인 0xc0000000로부터 시작하여 sys\_setresuid가 어디에 있는지 알려준다. 우리는 dev를 이용하여 물리메모리 기준으로 sys\_setresuid에 접근해야 한다. 그렇기 때문에 해당 주소가 page\_offset으로부터 어느만큼 떨어져 있는지 계산해서, tmp\_addr +=(addr\_sym - PAGE\_OFFSET) 을 한다. Unsigned long 단위로 접근 할 것이기 때문에, 오른쪽으로 두번 shift 연산을 해서, byte 단위인 값을 4byte 단위 값 으로 바꾸어 준다. 이후에 이 값을 128byte 까지 증가시키며 cmp문을 찾는다. 이 과정은 쉽지만은 않은 내용이 숨겨져 있었다. 왜 cmp문을 찾는지, cmp문을 바꾸면 만사 오케이 인지… 알면 알수록 어려워 지는 내용이다. 아래부터 이에 대해 알아보았다.

1. Setresuid 변형

* Setresuid함수

먼저 setresuid system call의 의미에 대해서 알아봐야 한다. 언제나 그러하듯이 linux man page에는 많은 정보가 존재했다. 그중에 유효한 정보는 unprivileged process 가 이 함수를 호출할 때에 대한 정보였다. Root가 아닌 프로세스가 setresuid를 호출하면 해당 프로세스(root가 아닌 프로세스) 의 ruid euid suid 값중에서 내가 세팅 하려는 ruid euid suid 값이 존재해야 한다. 만약에 내가 root 가 아니고, setresuid를 이용해서 root를 설정하고 싶은데, 현재 프로세스의 ruid euid suid 중에 0 값이 없다면, setresuid는 error를 return 할 것이다. 아래 코드에서 굵은 글씨로 표기한 부분이 바로 만약 root 가 아니라면, 세팅하려는 값이 현재의 ruid euid suid중에 존재 하지? 않하면 error 를 리턴하는 부분이다. 즉, 만약 root가 아닌 상태에서 uid 0 을 세팅하려면 붉은색으로 표기한 부분에서 nsown\_capable이 false를 리턴한 상황에서 if문 안으로 들어가려 하는 것을 막고 통과시켜야 할 것이다. 즉 if문을 assembly code로 본다면, cmp문에서 0과 비교를 하고, branch가 되지 않을 것이다. 즉, process가 root가 아니므로, nsown\_capable 이 0(false)를 리턴하면, ! 기호가 이를 1(true)로 만들 것이며, 이것이 cmp문에서 0과 비교되어서 branch equal 혹은 branch zero 문에 의하여 분기에 실패 할 것이다. 우리는 간단하게 cmp문에서 비교하는 상수값을 0에서 1로 바꾸어서 branch 가 되어서 if문 뒤로 instruction pointer를 건너뛰게 만들면 성공이다.

|  |
| --- |
| SYSCALL\_DEFINE3(setresuid, uid\_t, ruid, uid\_t, euid, uid\_t, suid)  {  ……  new = prepare\_creds();  **if (!new)** // 이게 바로 첫 번째 if 문….  return -ENOMEM;  old = current\_cred(); //현재process의 cred 를 받아온다 cred 변수에는 uid가 있다.  retval = -EPERM;  **if (!nsown\_capable(CAP\_SETUID)) {**  **if (ruid != (uid\_t) -1 && ruid != old->uid &&**  **ruid != old->euid && ruid != old->suid)**  **goto error;**  **if (euid != (uid\_t) -1 && euid != old->uid &&**  **euid != old->euid && euid != old->suid)**  **goto error;**  **if (suid != (uid\_t) -1 && suid != old->uid &&**  **suid != old->euid && suid != old->suid)**  **goto error;**  }  …………..  } |

다음과 같은 복잡한 코드가 system의 uid를 바꾼다. 조교님이 주신 source code에서 cmp문을 ‘항상 통과’ 하도록 수정을 한다.

* 왜 첫 번째 if문에 걸리지 않는가?

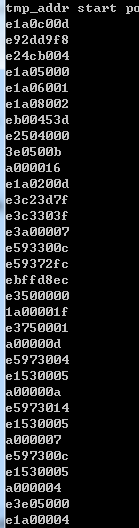
먼저 생각한 이유는 new 는 cred 형 구조체의 주소값 이니, 정수값일 것이고 nsown\_capable은 bool 값을 return 하니, 두 if문이 만드는 assembly code 가 다를 것이고 이 때문에 cmp문이 두번째 if에 걸릴 것 이다. 라는 가설이다.

하지만 boolean ture 를 return 하는 함수 dd()와 정수 int a 에 대해서

If(!dd()) 와 if(!a) 를 만들어서 어셈블리로 보았지만,



다음과 같이 결국에는 0과 비교를 하는 구문이었다. 물론 어셈블리를 만든 컴파일러는 exynods-abuse 를 만들 때 쓰는 컴파일러를 사용했다. 그렇다면 왜 첫 번째 if문이 아닌 두번째 if문을 바꾸는 것인지 이해가 더더욱 되지 않았다.

분명 코드 상에는 if문 이후에 if문이 있었는데, 명령어를 보면, 0xe3500000이전에는 cmp문이 눈을 씻고 찾아도 보이지 않는다. 그래서 분기 명령어를 찾아 보았다. 왼쪽의 instruction 에서 opcode가 a 로 시작하는 부분은 모두 분기문이다. 즉 제일 위를 번호 1 로 생각 한다면, 10번째 명령어 0x0a000016은 조건부 분기 같을경우 분기 이므로 0a 이므로 beq 명령어와 같다. 만약 분기한다 가정 했을 때, address offset + 16으로 이동하게 되고, 그곳에는 e5973004 가 있다. 그아래 0xe1530005는 cmp문의 또다른 형태이다. 이러한 분석이 명확성을 지니지는 못하지만 결과가 나와있는 상황에서(첫번째 cmp문이 0xe3500000 이라는 상황에서) 유효한 해석이라고 생각한다.

* 0xe3500000 자세히 알아보기

이제 0xe3500000에 대해서 자세히 알아보겠다. 소스코드에서 0xe3500000을 찾으면 이것을 수정한다. 그렇다면 첫 번째 if문을 ‘항상통과’ 로 바꾸는 것일까? 그런 것으로 보이지는 않는다. 왜냐하면 첫 번째 if문은 -ENOMEM이라는 에러를 리턴하는 if 문 이기 때문이다. 그렇다면 두번째 if문 일까? 그전에 0xe3500000 의 비밀에 대해서 알아보겠다.

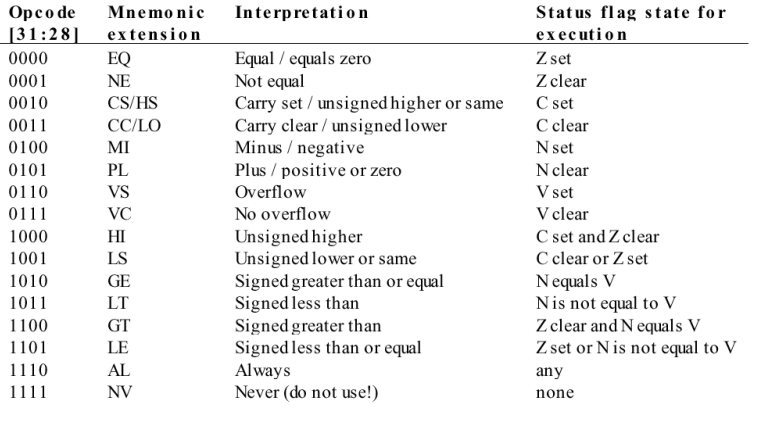


위는 ARM assembly 명령어의 형태이다. 32bit 의 의미에 대해 서술되어 있는데, 0xe3500000은 이진수로 바꾸면 1110 0011 0101 0000 0000 0000 0000 0000 과 같다.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1110 | 001 | 1010 | 1 | 0000 | 0000 | 0000 | | 00000000 |
| cond | cmp의 opcode | | | 첫째레지스터(R0) | sbz = should be zero | rotate?.... | 상수값0 | |

결국 0xe3500000 은 다음과 같은 형태의 cmp 명령어이다. 즉 R0레지스터의 값에다가 0을 빼서 status flag에 반영한다고 한다. 즉 0과 비교하는 것을 1 과 비교할 것이다.

이때 cond 는 무엇을 의미하는가? Cond는 조건부 시행의 경우를 이야기 한다 각 명령의 실행은 flag register의 bit를 변형시키는데, 이 flag register의 어떤 bit 가 설정되어있는 지에 기반하여 시행을 할 수가 있는 것이다. Cond의 가지수는 16가지 이며 다음과 같다.



즉 0xe3500000의 경우 cond 에 1110이 설정되어 있는 것이며, always이다. 즉, flag에 관련없이 명령어를 실행 하라는 이야기 이다. 이에 대한 논의는 뒤의 또 다른 system call을 이용한 rooting을 할 때 필요하다.

* Ns\_capable함수

Nsown\_capable은 ns\_capable을 호출하고 ns\_capable은 다음과 같다.

|  |
| --- |
| bool ns\_capable(struct user\_namespace \*ns, int cap)  {  if (unlikely(!cap\_valid(cap))) {  printk(KERN\_CRIT "capable() called with invalid cap=%u\n", cap);  BUG();  }  if (security\_capable(current\_cred(), ns, cap) == 0) {  current->flags |= PF\_SUPERPRIV;  return true;  }  return false;  } |

가능하면 true를 return 하는 것으로 보인다. 모든 과정을 거쳐서 시나리오 대로 0과 비교를 하게 되지만, 이것을 1과 비교로 patch 해 놓았기 때문에 분기문에서 if뒤로 jump를 하게 된다. 이로서 unprivileged user의 setresuid call 에 대한 처리를 건너뛰게 되었다!! 루팅완료. 쉘을 호출하는 부분은 system("/system/bin/sh"); 로 대체 하였다.

1. 또다른 system call patch

이번에는 setresuid와 거의 동일한 구조를 가지고 있는 setuid 를 이용해서 루트를 획득해보자. 어려울 것이 없는 것이 setuid 가 회피해야 할 if문까지의 구조가 setresuid와 setuid 의 경우 같다. 즉, 굵은 글씨로 되어 있는 부분이 같고 회피해야 할 if문이 동일하다. 그러니 setuid를 이용한 patching 은 당연히 된다. 그 이후로는 setresuid와 setuid가 하는 동작이 달라진다. 하지만 그 뒤에는 루트를 획득을 목적으로 하는 입장에서는 고려대상이 아니다.

|  |
| --- |
| SYSCALL\_DEFINE1(setuid, uid\_t, uid)  {  **const struct cred \*old;**  **struct cred \*new;**  **int retval;**  **new = prepare\_creds();**  **if (!new)**  **return -ENOMEM;**  **old = current\_cred();**  **retval = -EPERM;**  **if (nsown\_capable(CAP\_SETUID))** {  ……..  } |

* 한단계 더

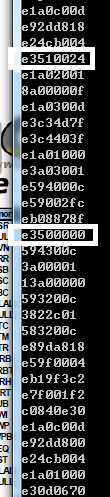
그렇다면 한 단계 더 들어가서 nsown\_capable에게 현재 프로세스의 uid에 관련없이 true를 리턴하도록 하면 되지 않을까? 그렇다면 해당 함수를 이용하여 특권확인을 하는 모든 systemcall이 현재 root 상태와 상관 없이 이용이 가능해 진다. Ns\_capable은 /kernel/capability.c 파일에 정의되어 있고 코드는 위에서 살펴보았듯이. 다음과 같다.

|  |
| --- |
| bool ns\_capable(struct user\_namespace \*ns, int cap)  {  if (unlikely(!cap\_valid(cap))) {….BUG();}  **if (security\_capable(current\_cred(), ns, cap) == 0) {….return true;}**  **return false;**  } |

굵은글씨 처리한 곳에서 ==0 이 되어서 return true를 한다면 모든 프로세스에 대하여 해당 함수는 자신이 root인 것으로 착각하게 될 것이다!

먼저 소스코드를 수정하였다. Kallsyms 기호를 출력하게 하는 곳 까지는 같고, 이후에 “setresuid”에 해당하는 주소를 찾았다면 , 이번에는 “ns\_capable 에 해당하는 주소를 찾는 것으로 변경 하였다. 이후에 두 번째의 if문에서 0과 같으면 true를 리턴하는 것을 통과시키도록 명령어를 patch시키는 과정이 필요하다. Security\_capable 함수는 /security/security.c 파일에 선언되어 있으며 다음과 같다.

|  |
| --- |
| int security\_capable(const struct cred \*cred, struct user\_namespace \*ns,  int cap)  {  return security\_ops->capable(cred, ns, cap, SECURITY\_CAP\_AUDIT);  } |

Security\_ops의 capable함수를 호출하는데, security\_ops는 security\_init함수에서 booting 과정에서 security\_ops = &default\_security\_ops; 를 호출하면서 초기화 한다. 이때 default\_security\_ops의 경우 따로 정의된 부분을 찾을 수 없었다 단지 변수로 선언되어 있었다. 하지만 확실한 것은 security\_capable이 정수값을 return 하고 이것을 0과 비교하여 0이면 우리의 목적인 return true가 달성 된다는 것이다.

다음은 ns\_capable에 해당하는 메모리를 4byte 단위로 나타낸 16진수 이다. 흰색 박스 부분이 2개의 cmp문을 나타내는데, 첫째 명령어는

Cmp r1 #36 이고 두번째는 Cmp r0 #0 이다 0과 비교하고 분기하는 두번째 cmp문을 수정하면 될 것으로 예상한다. 그러면 전 과제와 같게 0을 1로 바꾸면 될 것인가. 수행결과 ‘아니다’ 였다. 왜냐하면 setresuid와 setuid에서는 nsown\_capable 함수의 return 에 대하여 cmp를 수행했는데, 이 값이 bool값을 return 하며, 0 아니면 1을 return 한다 root가 아닌 프로세스는 0을 리턴해서 if문 안으로 들어가게 되어 있는 것을 if문에서 1과 비교를 수행하게 함으로서 if로부터 벗어날 수 있게 한다.

하지만 이번에 patch해야 할 if문은 int값과 0과의 비교를 수행한다. 이것을 1과의 비교로 바꾼다고 하더라도, ‘정수값’과 0과의 비교이기 떄문에 똑 같은 결과인 false를 리턴할 가능성이 있다. 즉, setresuid와는 다른 경우인 것이다. 아마 setresuid를 exploit한 사람의 경우에도 bool값으로 cmp문을 통과하는 부분을 취약점으로 파악하지 않았을까 싶다. 임의의 정수값과 0이 같은지 판단하는 상황에서 분기 하지 않고 if문 안으로 진입하고자 한다면 cmp문 전체를 바꾸거나 그 이후의 branch명령어의 조건을 바꾸어야 한다. 내가 생각한 의도가 맞는지는 모르겠지만, 저런 사소해 보이는 0 하나에도 모두 의미가 있다고 생각하니 커널에 더더욱 경외감이 느껴졌다.

0xe3500000을 수행한 뒤에 0x3a000010 은 조건부 분기를 한다. 저 명령어의 cond 는 3으로서 0011 (carry clear / unsigned lower)을 의미한다 (carry clear 란 cmp a,b 문에서 a<b일 때 일어나는 것으로서 cmp문이 a-b를 수행할 때 a+ (b의보수) 를 할 때 a<b일 때 만이 가산기에서 carry 가 일어나지 않는 속성을 이용한 것이다. 위의 cmp문에서 비교를 수행한 이후의 작동인데, 그냥 0011 대신에 1110 과 1111 즉, always 와 never condition을 사용하여 시험을 해 보았다. 결과는 꽝 이었다. 둘 다 root가 나오지 않았다. 혹시 위의 cmp문 즉 0xe3510024에서 분기가 일어나지 않고 에러를 출력하는가 하여, 그 아래 0x8a00000f 값을 0xea00000f로 바꾸어 항상 분기가 일어나도록 만들었다. 그랬더니, 커널 패닉이 났다. Console 디버깅 창을 보니, ns\_capable 에서 버그가 났다고 한다. 아마 위에서 내가 강제로 분기하게 한 코드가 error로 분기하는 부분이었나 보다;;



* 실패이유 분석

이유는 위의 cmp 문과 그 뒤에 따라 나오는 3a000010 이 내가 생각하는 if문에 해당하지 않는다는 것이 첫 번째로 생각해 볼 가능성 이다. 왜냐하면 BUG가 호출된 것으로 보아서 첫 번째 if문에서 통과를 하지 못한 것 같기 때문이다. 처음 if문은

아직 ARM assembly 코드가 익숙하지 않아서 그런 것 같기도 하다. 이미 저 바이너리 안에는 내가 하고싶은 모든 것이 들어 있는데 내가 문제인 경우이다. 이 경우가 확률이 가장 높다.

1. 눈높이를 낮춘 또 다른 system call

루팅도 처음 해보는 것이니, 이번에는 조금 수준을 낮추어 진행하여 보기로 하였다. 사실 제일 만만해 보이는 방식으로 위에서 시도한 것이다. 이미 rooting 가능한 함수임을 확인했으니, 한 단계 함수호출을 더 따라 들어가서 루팅을 하는 것이 크게 어렵지 않을 것이라고 생각했다. 하지만 오산이었다. 모든 함수에 접근할 수 있다면 다른 방식의 루팅이 너무나 쉬울 것이라는 것은 오만한 생각이었다. 그래서 내가 알아낸 사실인 setresuid의 특권확인에 착안하여 setresuid와 같은 파일인 /kernel/sys.c 에서 다음 systemcall을 발견하였다.

|  |
| --- |
| SYSCALL\_DEFINE1(setuid, uid\_t, uid)  {  **const struct cred \*old;**  **struct cred \*new;**  **int retval;**  **new = prepare\_creds();**  **if (!new)**  **return -ENOMEM;**  **old = current\_cred();**  **retval = -EPERM**;  **if (nsown\_capable(CAP\_SETUID))** {  new->suid = new->uid = uid;  if (uid != old->uid) {  retval = set\_user(new);  if (retval < 0)  goto error;  }  } else if (uid != old->uid && uid != new->suid) {  goto error;  }  …….  } |

너무 반갑게도 setresuid의 특권확인 nsown\_capable 함수 전까지 모든 코드가 같다!... 바로 작업에 착수했다. 바꾸는 부분은 너무나 간단하다. 딱 두 부분이 존재한다.

|  |
| --- |
| if(strncmp("sys\_setuid\n", str,11) ==0) found = true;  result = setuid(0); |

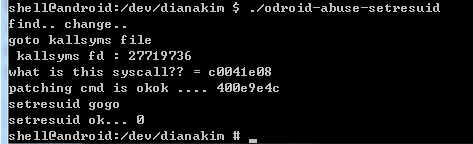
다음과 같이 두 부분을 고치면 모두 해결된다. 의심 반으로 했지만 시행결과 루팅이 잘 되었다.

1. 결과



다음의 두 system call을 통한 루팅

1. C0041e08 setresuid..



1. C0041d40 setuid

