

ARM

Reverse Engineering

singi@hackerschool

Facebook : @sjh21a

<http://kernelhack.co.kr/netsec-singi.zip>

Table of Contents

- **First Phase**

- About ARM
- ARM Operating Mode and Registers
- Basic ARM Instruction
- Thumb Mode

- **Second Phase**

- Configuration of Reverse Engineering
- Function Calling Convention
- Analysis C Syntax to ARM Assembly
- Real world Example #1 – for third party app
- Real world Example #2 – for Default Browser
- Reference

First Phase

- ARM CPU의 동작 모드
- ARM CPU의 Register와 용도
- 자주 사용되는 ARM Instruction 습득
- ARM CPU만의 특징 파악
- ARM Assembly 예제

About ARM

- ARM is **A**dvanced **R**ISC **M**achine

- 32비트의 명령어로 구성되어 있음. (ARM 서버용으로 64비트 출시 됨)
- RISC는 명령어가 CISC보다 간단하고, 수가 적음. (x86 계열은 CISC 사용)
- ARM Core? ARM Processor?

- ARM Architecture / Processor 종류

Architecture	Processor
v4	ARM7TDMI, ARM720T, ARM940T, ARM920T, ARM922T
v5TE	ARM946E-S, ARM926E-S, Xscale
v5TEJ	ARM926EJ-S
v6	ARM1136JF-S
v7	Cortex A, M, R

T : Thumb, D : Debug Port (JTAG),

M : 8비트 곱셈기, I : Break Point나 Watch Point 설정 가능, 'D'와 사용.

-E : DSP 연산 명령어 추가, -S : VHDL, Verilog로 회로도가 제공됨.

-J : Java Byte code 해석 가능.
















ARM Operating Mode and Registers

■ ARM Operating Modes

Mode	설명	
Supervisor(SVC)	Reset이나 SWI 명령이 실행 될 때	Privileged mode
FIQ	Fast Interrupt가 발생 되었을 때	
IRQ	일반적인 Interrupt가 발생 되었을 때	
Abort	Data 나 instruction fetch에 실패 할 때	
Undefined	정의되지 않은 instruction일 때	
System	User 모드와 같지만, 특권 모드임	Unprivileged mode
User	응용프로그램이나 OS 실행 할 때	

- Privileged mode에선 Interrupt의 사용유무 설정 가능
- Privileged mode 에서 서로 변경이 자유롭지만, Unprivileged mode에선 변경이 불가능함.

ARM Operating Mode and Registers

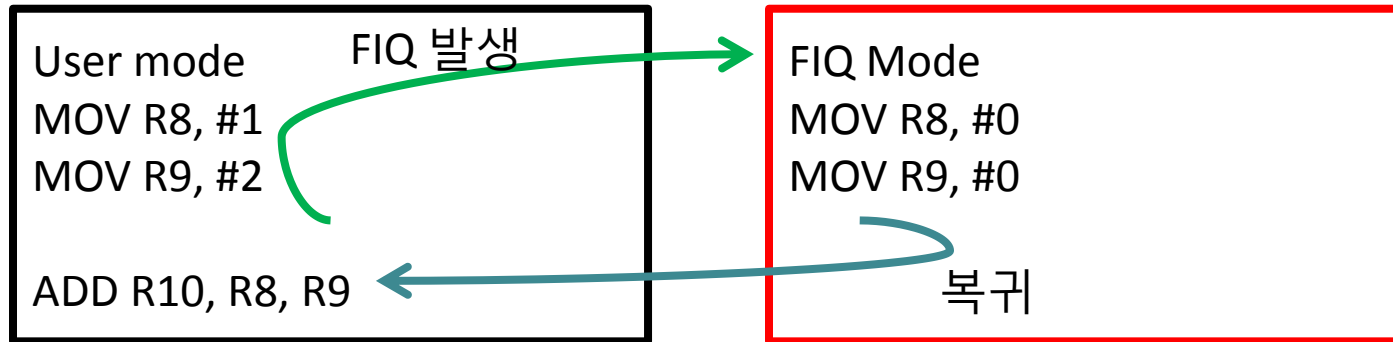
System & User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	 R8_fiq	R8	R8	R8	R8
R9	 R9_fiq	R9	R9	R9	R9
R10	 R10_fiq	R10	R10	R10	R10
R11	 R11_fiq	R11	R11	R11	R11
R12	 R12_fiq	R12	R12	R12	R12
R13	 R13_fiq	 R13_svc	 R13_abt	 R13_irq	 R13_und
R14	 R14_fiq	 R14_svc	 R14_abt	 R14_irq	 R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

ARM State Program Status Registers

CPSR	CPSR  SPSR_fiq	CPSR  SPSR_svc	CPSR  SPSR_abt	CPSR  SPSR_irq	CPSR  SPSR_und
------	--	--	---	--	--

ARM Operating Mode and Registers

- Example #1



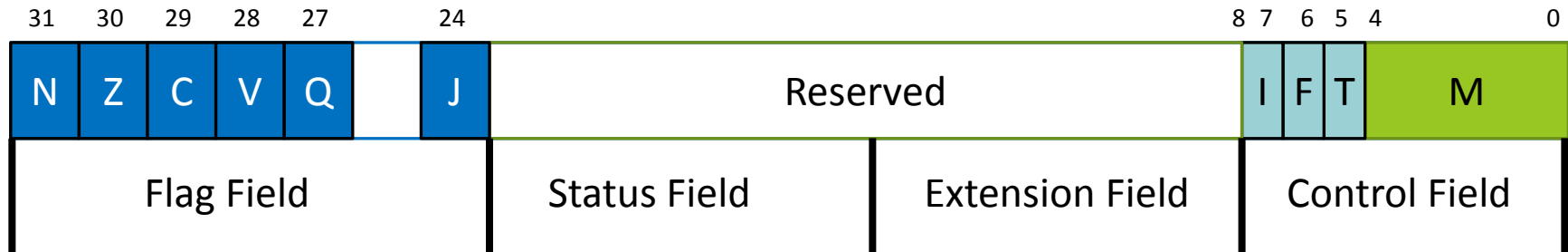
- R10에 저장되는 값은?

ARM Operating Mode and Registers

- R0 ~ R12 : 일반 연산 및 임시 저장 장소 등으로 사용.
- R13
 - **Stack Pointer** (SP)
 - 동작 모드 별로 별도로 존재함.
- R14
 - **Link Register** (LR)
 - 함수 호출 시 리턴 될 주소를 가지고 있음
 - 동작 모드 별로 별도로 존재 함.
 - 스택에 접근 해서 Return Address에 접근 하는 것은 레지스터보다 상대적으로 느림.
- R15
 - **Program Counter** (PC)
 - 다음에 실행 될 명령어를 가지고 메모리로부터 가지고 옴.
 - 모드 별로 별도로 존재하지 않고, 하나의 R15 레지스터만 존재함.
- CPSR
 - **Program Status Register** (CPSR)
 - Mode가 변경 되면 H/W적으로 변경되기 전의 CPSR이 SPSR(Saved ...)에 저장됨.
 - User/System Mode를 제외하고 각 모드 마다 하나씩 존재함.

ARM Operating Mode and Registers

CPSR Register 구조



Flag Field	
N	연산 결과가 마이너스인 경우에 set
Z	연산 결과가 0인 경우에 set
C	연산 결과에 자리 올림이 발생 했을 때 set
V	연산 결과가 overflow 됐을 경우 set
Q	포화가 발생되면 set, 반드시 clear
J	자바 바이트 코드 실행 상태

Control bits	
I	1인 경우 : IRQ 비활성화
F	1인 경우 : FIQ 비 활성화
T	1인 경우 : Thumb, 0인 경우 : ARM

Mode bits	
10000	User
11111	System
10001	FIQ
10010	IRQ
10011	SVC
10111	Abort
11011	Undefined

Basic ARM Instruction

32bit ARM Instruction

구분		명령어
1	분기 명령	B, BL
2	데이터 연산 명령	ADD, ADC, SUB, SBC, RSB, RSC, AND, ORR, BIC, MOV, MVN, CMP, CMN, TST, TEQ
3	Multiply 명령	MUL, MLA, SMULL, SMLAL, UMULL, UMLAL
4	Load/Store 명령	LDR, LDRB, LDRBT, LDRH, LDRSB, LDRSH, LDRT, STR, STRB, STRBT, STRH, STRT
5	Load/Store Multiply 명령	LDM, STM
6	Swap 명령	SWP, SWPB
7	Software Interrupt 명령	SVC (기존 SWI에서 변경)
8	PSR 전송 명령	MRS, MSR
9	Co-Processor 명령	MRC, MCR, LDC, STC
10	Branch Exchange 명령	BX
11	...	ARM Architecture 별로 명령어들이 추가 존재.

Basic ARM Instruction

MOV R0, #31337

Op-code

Destination Register

MOV R0, R1

Source Register

- ARM은 메모리 내에 직접 데이터를 쓰거나, 가져올 수 없음!
- **특정 명령**을 통해 메모리 값을 레지스터에 가져오거나, 레지스터 값을 메모리에 써야 함. (LDR, STR 명령어)
- 이것은 RISC 구조의 대표적인 특징이고, **Load/Store 구조**라 함.
- 또한, 32비트 상수 값은 Operand로 사용 할 수 없음.

Basic ARM Instruction

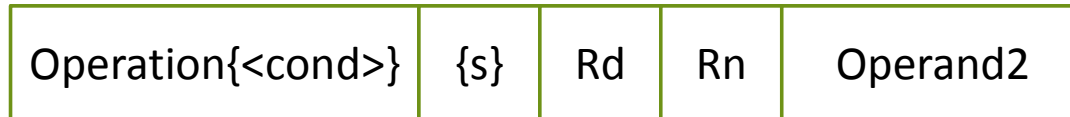
■ 조건부 실행 {<cond>}

- 파이프라인 구조를 가지는 CPU의 지연을 줄이기 위한 것.
- 명령어의 조건필드와 CPSR Register의 N, Z, C, V값을 비교하여 수행 됨.

접미사	CPSR Flag	의미
EQ	Z Flag set	같다.
NE	Z Flag Clear	같지 않다.
CS	C Flag Set	크거나 같다.(unsigned)
CC	C Flag Clear	작다(unsigned)
MI	N Flag Set	음수
PL	N Flag Clear	양수 또는 0
VS	V Flag Set	오버플로우 발생
VC	V Flag Clear	오버플로우 X
HI	C Flag Set, Z Flag Clear	크다.(Unsigned)
LS	C Flag Clear	작다.(Unsigned)
GE	N Flag = V Flag	크거나 같다.
LT	N Flag != V Flag	작다
GT	Z Flag Clear AND (N = V)	크다
LE	Z Flag Set OR (N != V)	작거나 같다
AL	Ignore	무조건 실행

Basic ARM Instruction

- 산술 연산 명령어 형식



- Operation**

- ADD, SUB, ADC, ... 등

- {<cond>}**

- 조건부 실행을 위한 조건

- {s}**

- 연산 결과로부터 CPSR Register의 Flag를 set 함.

- Rd**

- Destination Register

- Rn**

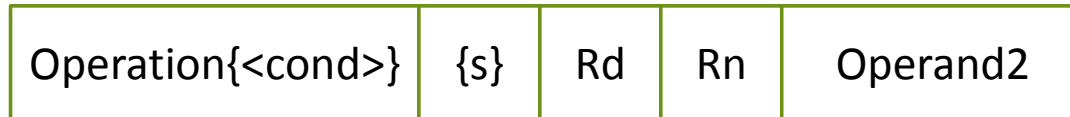
- Source Register

산술 연산 명령어 예제

ADD	R0, R1, R2	R1과 R2을 더하여 R0에 저장.
SUBEQ	R0, R1, #8	EQ조건이면, R2에서 8을 빼서, R0에 저장하고, 결과에 따라 CPSR Flag를 설정함.
ADDS	R1, R2, R0	R2과 R0를 더하여 R1에 저장하고, 결과에 따라 CPSR Flag를 설정함.

Basic ARM Instruction

- 논리 연산 명령어 형식



- **Operation**

- AND, ORR, EOR, BIC

- **{<cond>}**

- 조건부 실행을 위한 조건

- **{s}**

- 연산 결과로부터 CPSR Register의 Flag를 set 함.

- **Rd**

- Destination Register

- **Rn**

- Source Register

논리 연산 명령어 예제

AND	R0, R1, R2	R1과 R2을 AND연산하여 R0에 저장.
-----	------------	-------------------------

ANDEQS	R0, R1, R2	EQ조건이면, R1과 R2를 AND연산 하여 R0에 저장하고, 결과에 따라 CPSR Flag를 설정함.
--------	------------	---

BICLE	R1, R2, R0	LE조건이면, R2와 R0를 XOR 하고 결과를 R1에 저장.
-------	------------	------------------------------------

Basic ARM Instruction

■ 비교 명령어 형식

Operation{<cond>}	Rn	Operand2
-------------------	----	----------

■ Operation

- CMN, CMP, TEQ, TST

■ {<cond>}

- 조건부 실행을 위한 조건

■ Rn

- Source Register

■ Destination Register가 존재 하지 않음.

■ CPSR Flag를 설정하기 위한 명령이 없어도 항상 설정함.

비교 명령어 예제

CMP	R0, R1	R1과 R2를 비교하여, 그 결과로 CPSR에 Flag 설정.
TSTEQ	R2, #8	EQ조건이면, R2와 #8을 비교 후, 결과로 CPSR Flag 설정.
CMN		LE조건이면, R2와 R0를 XOR 하고 결과를 R1에 저장.

Basic ARM Instruction

■ MOVE 명령어 형식

Operation{<cond>}	Rd	Operand2
-------------------	----	----------

■ Operation

- MOV, MVN

■ {<cond>}

- 조건부 실행을 위한 조건

■ Rd

- Destination Register

■ Source Register가 존재 하지 않음.

비교 명령어 예제

MOV	R0,	R1	R1을 R0로 옮긴다.
-----	-----	----	--------------

MOVL	R0,	R2	LE조건이면, R2를 R0로 옮긴다.
------	-----	----	----------------------

MVN	R0,	R2	R2 XOR 0xFFFFFFFF 한 값을 R0에 옮긴다. (Negative로 변경)
-----	-----	----	--

Basic ARM Instruction

■ 분기 명령어 형식

B{L}	{cond}	<expression>
------	--------	--------------

■ {L}

- Branch with link로 R14에 PC값을 저장 함.

■ {<cond>}

- 조건부 실행을 위한 조건

■ <expression>

- 위치 정보

비교 명령어 예제

BL	somewhere	Somewhere로 분기, LR에 돌아올 주소 저장. (함수 호출에 사용) 함수 호출 후, 되돌아 갈 때, MOV PC, LR 사용함.
B	somewhere	Somewhere로 분기
BL	somewhere+1	계산된 위치로 분기.
CMP	R1, #0	R1이 0이면 (EQ조건이면) success로 분기.
BEQ	success	

Basic ARM Instruction

■ 데이터 전송 명령어 형식 – **Pre-index** 방식

LDR STR	{cond}	{B}	Rd	[Rn, <offset>]	{!}
---------	--------	-----	----	----------------	-----

■ {cond}

- 조건부 실행을 위한 조건

■ {B}

- Unsigned Byte 단위의 Access를 할 때 사용 됨.

■ Rd

- LDR의 경우 Destination Register가 되고, STR의 경우 Source Register로 사용 됨.

■ Rn

- Base Register로 사용 됨.

■ Offset

- 12비트 상수 또는 레지스터가 올 수 있음.

■ {!}

- Pre-Index 방식에서 Base Register 값을 자동으로 업데이트 할 때 사용 됨.

데이터 전송 명령어 형식 – Pre-index 방식

LDR R1, [R2, R4] R2+R4 위치에서 데이터를 워드만큼 읽어서, R1에 저장.

LDR R1, [R2, R4]! R2+R4 위치에서 데이터를 워드만큼 읽어서, R1에 저장.
전송 후, R2의 값은 R2+R4 값으로 변경됨.

STR R1, [R2, R4] R1 값을 R2+R4 위치에 워드 만큼 저장 함.

LDR R1, [R2, #8] R2+8 위치에서 워드만큼 읽어서 R1에 저장 함.

Basic ARM Instruction

■ 데이터 전송 명령어 형식 – **Post-index** 방식

LDR STR	{cond}	{B}	{T}	Rd	Rn	offset
---------	--------	-----	-----	----	----	--------

■ {cond}

- 조건부 실행을 위한 조건

■ {B}

- Unsigned Byte 단위의 Access를 할 때 사용 됨.

■ {T}

- 데이터 전송 시에 Unprivileged mode로 전송 함.

■ Rd

- LDR의 경우 Destination Register가 되고, STR의 경우 Source Register로 사용 됨.

■ Rn

- Base Register로 사용 됨.

■ Offset

- 12비트 상수 또는 레지스터가 올 수 있음.

데이터 전송 명령어 형식 – Post-index 방식

LDR R1, [R2], #4 R2에서 워드만큼 읽어서 R1에 저장 후, R2+4값으로 R2 변경 함.

LDR R1, [R2], R4 R2에서 워드만큼 읽어서 R1에 저장 후, R2+R4 값으로 R2 변경 함.

STR R1, [R2], R4 R1 값을 R2 위치에 워드 만큼 저장 후, R2+R4 값으로 R2 변경 함.

Basic ARM Instruction

■ 데이터 전송 명령어 형식 – PC-Relative 방식

LDR	{cond}	{size}	Rd	Label =DATA
-----	--------	--------	----	---------------

■ {cond}

- 조건부 실행을 위한 조건

■ {size}

- Unsigned Byte 단위의 Access를 할 때 사용 됨.

■ Rd

- Destination Register.

■ Label

- PC 위치에서 4K byte 내에 있는 Label 참조

■ =DATA

- 32비트 데이터를 레지스터에 쉽게 넣기 위한 어셈블러 기능.

데이터 전송 명령어 형식 – PC-Relative방식

LDR R1, label

label: Label 위치에서 데이터 0xdeadbeef를 워드만큼 읽어서, R1에 저장.

DCD 0xdeadbeef

LDR R1, =0xdeadbeef 데이터 0xdeadbeef를 R1에 저장한다.

Basic ARM Instruction

■ 다중 데이터 전송 명령어 형식

STM	{cond}	<addressing mode>	Rn{!}	<register_list>
-----	--------	-------------------	-------	-----------------

■ {cond}

- 조건부 실행을 위한 조건

■ {addressing mode}

- 어드레스를 만드는 방법을 나타냄.

■ Rn

- Base Register로 이 위치로부터 데이터가 저장.

■ !

- 데이터 전송 후, Base Register를 변경한다.

■ {register_list}

- 저장할 데이터를 가지고 있는 레지스터.

다중 데이터 전송 명령어 형식

STMIA R0, {R1,R2,R3} R1,R2,R3 데이터를 R0 위치 부터 저장함.

STMIA R0!, {R1,R2,R3} R1,R2,R3 데이터를 R0 위치에 저장하고, R0 값에 12가 더해짐.
 $12 = R1(4) + R2(4) + R3(4)$

Basic ARM Instruction

다중 데이터 전송 명령어 형식

LDM	{cond}	<addressing mode>	Rn{!}	<register_list>^
-----	--------	-------------------	-------	------------------

{cond}

- 조건부 실행을 위한 조건

{addressing mode}

- 어드레스를 만드는 방법을 나타냄.

Rn

- Base Register로 이 위치로부터 데이터가 읽혀짐.

!

- 데이터 전송 후, Base Register를 변경한다.

{register_list}

- 저장할 데이터를 가지고 있는 레지스터.

^

- Privileged mode 에서 CPSR를 복원.

다중 데이터 전송 명령어 형식

LDMIA R0, {R1,R2,R3} R0 위치에서 데이터를 차례대로 읽은 후, 각각 R1, R2,R3에 저장.

LDMIA R0!, {R1,R2,R3} R0 위치에서 데이터를 차례대로 읽은 후, 각각 R1, R2, R3에 저장,
그 후, R0에 R1(4)+R2(4)+R3(4)가 증가.

LDMIA R0, {R0-R7} R0 위치에서 데이터를 읽은 후, R0~R7까지 각각 저장.

LDMIA R0!, {R0-R2, PC} R0위치에서 데이터를 읽은 후, R0~R2까지 그리고 PC에 저장.

Basic ARM Instruction

■ ARM<->Thumb mode 명령어 (Inter-working)

BLX	{cond}	Rm
BLX	Label	

■ {cond}

- 조건부 실행을 위한 조건

■ Rm

- 분기 하고자 하는 주소 정보와 변환하고자 하는 상태 정보가 저장.

■ Label

- PC-relative 방식으로 분기, **반드시 조건 없이 사용** 해야 하고, 무조건 Thumb로 전환 됨.

Inter-working 명령어 형식

BLX	thumb_Func	thumb_Func로 분기하고, Thumb mode로 전환하고 R14 Register에 돌아올 주소 저장.
BLXEQ	R0	EQ조건이면, R0가 지정하는 위치로 분기하고, 상태 정보에 따라 상태를 전환하고, R14 Register에 돌아올 주소 저장.

Basic ARM Instruction

- Software Interrupt 명령어 형식

SVC	{cond}	<expression>
-----	--------	--------------

- **{cond}**

- 조건부 실행을 위한 조건

- **<expression>**

- 인터럽트 번호

Software Interrupt 명령어 형식

SWI	0x80	CPSR을 SPSR_svc에 저장 후, PC를 0x08(SWI의 예외 처리 Handler)로 분기 후, 0x80을 보고 정의된 동작 실행.
-----	------	---

Basic ARM Instruction

■ ARM<->Thumb mode 명령어 (Inter-working)

BLX	{cond}	<addressing mode>	Rn{!}	<register_list>^
-----	--------	-------------------	-------	------------------

■ {cond}

- 조건부 실행을 위한 조건

■ Rm

- 분기 하고자 하는 주소 정보와 변환하고자 하는 상태 정보가 저장.

■ Label

- PC-relative 방식으로 분기, **반드시 조건 없이 사용** 해야 하고, 무조건 Thumb로 전환 됨.

Inter-working 명령어 형식

BLX	thumb_Func	thumb_Func로 분기하고, Thumb mode로 전환하고 R14 Register에 돌아올 주소 저장.
BLXEQ	R0	EQ조건이면, R0가 지정하는 위치로 분기하고, 상태 정보에 따라 상태를 전환하고, R14 Register에 돌아올 주소 저장.

Basic ARM Instruction

```
.global _start
```

```
_start:
```

```
    MOV    r0, #1
```

```
    MOV    r1, pc
```

```
    ADD    r1, #24
```

```
    MOV    r2, #13
```

```
    MOV    r7, #4
```

```
    SVC    1
```

```
    SUB    r0, r0, r0
```

```
    MOV    r7, #1
```

```
    SVC    1
```

```
.ascii "Hello NetSec\n"
```

Thumb Mode

■ Thumb mode 도입 이유

- 기존 ARM 명령을 사용하여 만들어진 **바이너리 보다 크기가 약 75% 적음**. (Thumb-2)
- 크기가 적어지면, Flash나 Rom 같은 **저장 장치의 단가를 줄일 수 있음**.
- 16비트 메모리 인터페이스를 사용하면, 가격과 **전력 소모를 줄일 수 있음**.

	ARM Mode	Thumb Mode
명령어 크기	32bit	16bit
사용 Register 수	R0~R15	R0~R7
조건부 실행 가능	가능	불가
실행 파일 크기	100%	65%

Thumb Mode

```
.global _start
```

```
_start:
```

```
    .code 32
```

```
    ADD    r3, pc, #1
```

```
    BX     r3
```

```
    .code 16
```

```
    MOV    r0, pc
```

```
    ADD    r0, #10
```

```
    STR    r0, [sp, #4]
```

```
    ADD    r1, sp, #4
```

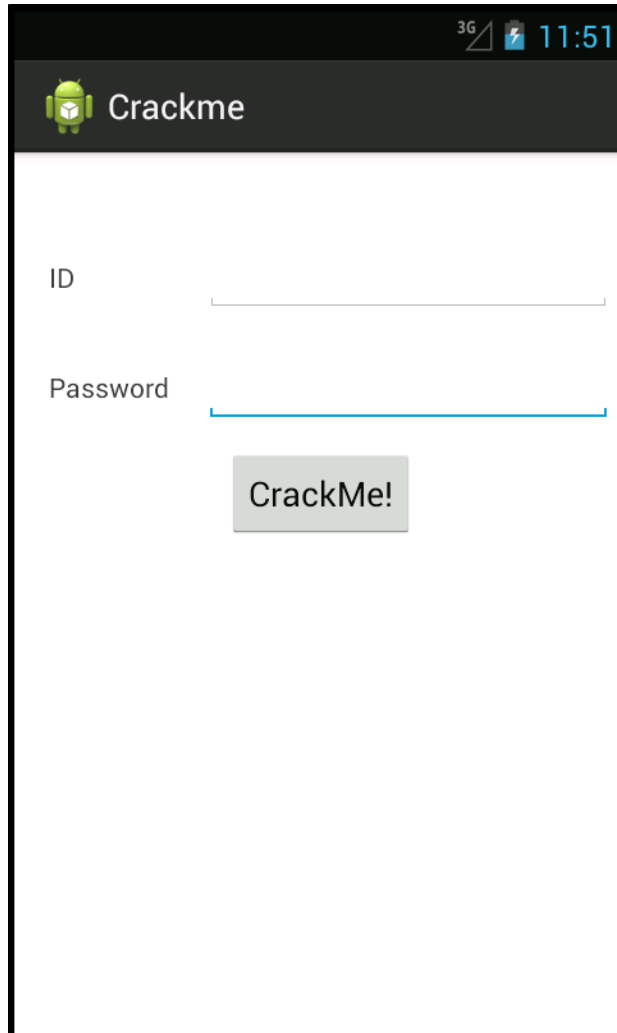
```
    SUB    r2, r2, r2
```

```
    MOV    r7, #11
```

```
    SVC    1
```

```
.ascii "/system/bin/sh"
```

Let's Do it! CrackMe #1



The screenshot shows the Crackme application interface on an Android device. The status bar at the top indicates 3G connectivity, battery level, and the time 11:51. The app's title bar features the Android logo and the text "Crackme". The main interface consists of two input fields: "ID" and "Password". The "ID" field has a light blue underline, and the "Password" field has a light blue underline. Below these fields is a grey button labeled "CrackMe!".

Crackme.apk

Let's Do it! CrackMe #1

```
-----
! Compression-Level: 9 ! Heap Size: 512mb ! Decompile : Sources and Resources Files ! Current-App: None !
-----

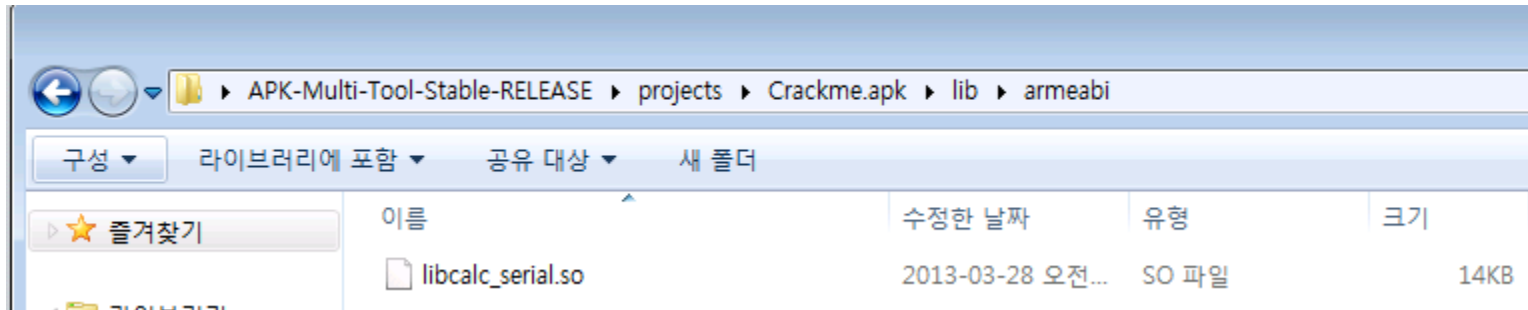
Simple Tasks Such As Image Editing      Advanced Tasks Such As Code Editing      Themers Conversion Tools
-----
0  Adb pull                             9  Decompile apk                          16  Batch Theme Image Transfer
1  Extract apk                           10  Decompile apk <with dependencies>      <Read the Instructions before
2  Optimize images inside                 <For proprietary rom apks>              using this feature>
3  Zip apk                               11  Compile System APK files
4  Sign apk <Dont do this if its          12  Compile Non-System APK Files
   a system apk>                         13  Sign apk
5  Zipalign apk <Do once apk is           14  Install apk
   created/signed>                       15  Compile apk / Sign apk / Install apk
      Install apk <Dont do this if        <Non-System Apps Only>
      system apk, do adb push>
7  Zip / Sign / Install apk
   <All in one step>
8  Adb push <Only for system apk>
-----

tools Stuff
-----
17  Batch Optimize Apk <inside place-apk-here-to-batch-optimize only>
18  Sign an apk<Batch support><inside place-apk-here-for-signing folder only>
19  Batch optimize ogg files <inside place-ogg-here only>
20  Clean Files/Folders
21  Select compression level for apk's
22  Set Max Memory Size <Only use if getting stuck at decompiling/compiling>
23  Read Log
24  Set current project
25  About / Tips / Debug Section
26  Switch decompile mode <Allows you to pick to fully decompile the APK's or JAR's
   or to just decompile Sources or just the Resources or do a raw dump allowing you
   to just edit the normal images>
00  Quit
-----

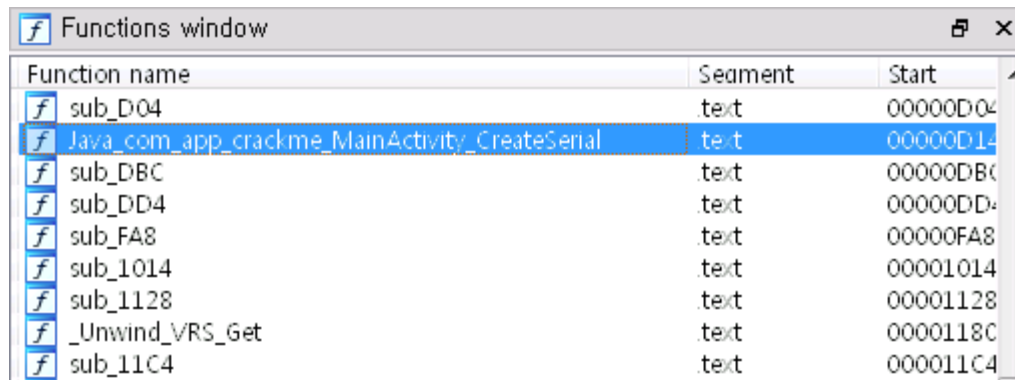
Please make your decision:
```

APK-Multi-Tool을 이용해 디 컴파일

Let's Do it! CrackMe #1



APK 안에 있던 JNI 파일 추출.



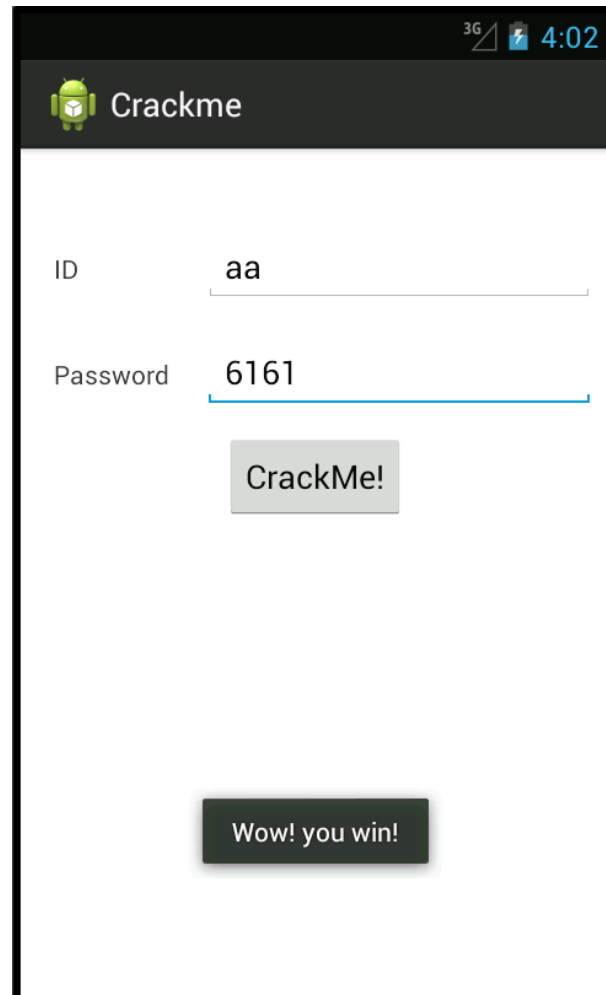
IDA를 이용하면, 함수 이름 보여짐.

함수명 : Java_com_app_crackme_MainActivity_CreateSerial

Let's Do it! CrackMe #1

```
00000D6C          ADD     R6, PC           ; "%02x"
00000D6E          B       loc_D84
00000D70 ; -----
00000D70          loc_D70 ; CODE XREF
00000D70          LDRB    R2, [R7,R4]
00000D72          MOVS    R1, R6           ; format
00000D74          ADD     R0, SP, #0x50+src ; s
00000D76          BLX     sprintf
00000D7A          LDR     R0, [SP,#0x50+dest] ; dest
00000D7C          ADD     R1, SP, #0x50+src ; src
00000D7E          BLX     strcat
00000D82          ADDS    R7, #1
00000D84          loc_D84 ; CODE XREF
00000D84          ADD     R4, SP, #0x50+s
00000D86          MOVS    R0, R4           ; s
00000D88          BLX     strlen
00000D8C          CMP     R7, R0
00000D8E          BCC     loc_D70
00000D90          LDR     R2, [R5]
00000D92          MOVS    R3, 0x29C
00000D96          LDR     R3, [R2,R3]
00000D98          MOVS    R0, R5
00000D9A          LDR     R1, [SP,#0x50+dest]
00000D9C          BLX     R3
00000D9E          ADDS    R4, R0, #0
```

Let's Do it! CrackMe #1



The screenshot shows the CrackMe application interface on an Android device. The status bar at the top indicates 3G connectivity, a battery icon, and the time 4:02. The app's title bar features the Android logo and the text "Crackme". The main interface has two input fields: "ID" with the value "aa" and "Password" with the value "6161". Below these fields is a grey button labeled "CrackMe!". At the bottom of the screen, a dark grey button with the text "Wow! you win!" is visible.

Clear!

Second Phase

- ARM Based Android Device에서 Test 환경 구성
- ARM Assembly로 보는 C언어 함수 호출 규약 파악
- 자주 사용되는 C 언어 문법 형태 파악.
- Webkit One-day 취약점 분석

Configuration of Reverse Engineering

- Test Device/OS : Samsung Galaxy S3 / Android
- Host OS : Ubuntu 12.04
- Cross Compiler : Android-NDK-r8d
- Android Debugger : Android-SDK



Configuration of Reverse Engineering

- NDK : <http://developer.android.com/tools/sdk/ndk/index.html>

```
root@ubuntu:~# cd android-ndk-r8d/
root@ubuntu:~/android-ndk-r8d# ./build/
awk/   core/  gmsl/  tools/
root@ubuntu:~/android-ndk-r8d# ./build/tools/
build-ccache.sh      build-mingw64-toolchain.sh  find-case-duplicates.sh
build-gabi++.sh       build-ndk-stack.sh         gen-platforms.sh
build-gcc.sh          build-ndk-sysroot.sh       gen-system-symbols.sh
build-gdbserver.sh    build-stlport.sh           gen-toolchain-wrapper.sh
build-gnu-libstdc++.sh build-target-prebuilts.sh  make-release.sh
build-host-awk.sh      cleanup-apps.sh            make-standalone-toolchain.sh
build-host-gcc.sh      dev-cleanup.sh             ndk-ccache-gcc.sh
build-host-gdb.sh      dev-platform-compress.sh  ndk-ccache-g++.sh
build-host-make.sh     dev-platform-expand-all.sh package-release.sh
build-host-prebuilts.sh dev-platform-expand.sh    patch-sources.sh
build-host-python.sh   dev-platform-import.sh    rebuild-all-prebuilt.sh
build-host-sed.sh      dev-rebuild-ndk.sh         toolchain-licenses/
build-host-toolbox.sh  dev-system-import.sh      toolchain-patches/
build-llvm.sh          download-toolchain-sources.sh unwanted-symbols/
root@ubuntu:~/android-ndk-r8d# ./build/tools/make-standalone-toolchain.sh --platform=android-14 --install-dir=/tmp
Auto-config: --toolchain=arm-linux-androideabi-4.6
Copying prebuilt binaries...
Copying sysroot headers and libraries...
Copying libstdc++ headers and libraries...
Copying files to: /tmp
Cleaning up...
Done.
root@ubuntu:~/android-ndk-r8d# ls /tmp
arm-linux-androideabi  COPYING      include  lib32     ndk-root  sysroot
bin                   COPYING.LIB  lib      libexec   SOURCES   unware-root
```

Configuration of Reverse Engineering

```
root@ubuntu:~/netsec# cat write.s
.global _start
_start:
    mov r0, #1
    mov r1, pc
    add r1, #24
    mov r2, #13
    mov r7, #4
    svc 1

    sub r0, r0, r0
    mov r7, #1
    svc 1
.ascii "Hello NetSec\n"

root@ubuntu:~/netsec# arm-linux-androideabi-as -o write.o write.s
root@ubuntu:~/netsec# arm-linux-androideabi-ld -o write write.o
root@ubuntu:~/netsec# file write
write: ELF 32-bit LSB executable, ARM, version 1 (SYSV), statically linked, not stripped
root@ubuntu:~/netsec#
```

설치된 Toolchain을 이용한 ARM 예제 컴파일 방법

Configuration of Reverse Engineering

```
root@ubuntu:~/netsec# cat sh.s
.global main
main:
    .code 32
    add r3, pc, #1
    bx r3
    .code 16

    mov r0, pc
    add r0, #10

    str r0, [sp, #4]
    add r1, sp, #4
    sub r2, r2, r2
    mov r7, #11
    svc 1
.ascii "/system/bin/sh"
root@ubuntu:~/netsec# arm-linux-androideabi-as -mthumb -o sh.o sh.s
root@ubuntu:~/netsec# arm-linux-androideabi-ld -o sh sh.o
root@ubuntu:~/netsec#
```

설치된 Toolchain을 이용한 Thumb 예제 컴파일 방법

Configuration of Reverse Engineering

```
root@ubuntu:~/netsec# arm-linux-androideabi-objdump -d write
```

```
write:      file format elf32-littlearm
```

```
Disassembly of section .text:
```

```
00008074 <_start>:
```

```
8074:      e3a00001      mov     r0, #1
8078:      e1a0100f      mov     r1, pc
807c:      e2811018      add     r1, r1, #24
8080:      e3a0200d      mov     r2, #13
8084:      e3a07004      mov     r7, #4
8088:      ef000001      svc     0x00000001
808c:      e0400000      sub     r0, r0, r0
8090:      e3a07001      mov     r7, #1
8094:      ef000001      svc     0x00000001
8098:      6c6c6548      .word   0x6c6c6548
809c:      654e206f      .word   0x654e206f
80a0:      63655374      .word   0x63655374
80a4:      Address 0x000080a4 is out of bounds.
```

ARM Mode 결과

```
root@ubuntu:~/netsec# arm-linux-androideabi-objdump -d sh
```

```
sh:        file format elf32-littlearm
```

```
Disassembly of section .text:
```

```
00008074 <_start>:
```

```
8074:      e28f3001      add     r3, pc, #1
8078:      e12fff13      bx      r3
807c:      4678          mov     r0, pc
807e:      300a          adds    r0, #10
8080:      9001          str     r0, [sp, #4]
8082:      a901          add     r1, sp, #4
8084:      1a92          subs    r2, r2, r2
8086:      270b          movs    r7, #11
8088:      df01          svc     1
808a:      732f          .short  0x732f
808c:      65747379      .word   0x65747379
8090:      69622f6d      .word   0x69622f6d
8094:      68732f6e      .word   0x68732f6e
```

Thumb Mode 결과

Configuration of Reverse Engineering

```
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Users\Wx> cd .\android-sdks
PS C:\Users\Wx\android-sdks> cd .\platform-tools
PS C:\Users\Wx\android-sdks\platform-tools> .\adb.exe push C:\Users\Wx\Desktop\write /data/local/tmp/write_exam
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
1581 KB/s (4744 bytes in 0.002s)
PS C:\Users\Wx\android-sdks\platform-tools> .\adb.exe shell chmod 755 /data/local/tmp/write_exam
PS C:\Users\Wx\android-sdks\platform-tools> .\adb.exe shell
shell@android:/ $ /data/local/tmp/ex/write_exam
/data/local/tmp/ex/write_exam
/system/bin/sh: /data/local/tmp/ex/write_exam: not found
127!shell@android:/ $ /data/local/tmp/write_exam
/data/local/tmp/write_exam
Hello NetSec
shell@android:/ $
```

컴파일 된 바이너리를 adb를 이용해 test device에 업로드 후 실행.

Function Calling Convention

```

0000830c <main>:
830c: e92d4800 push {fp, lr}
8310: e28db004 add fp, sp, #4
8314: e3a00001 mov r0, #1
8318: e3a01002 mov r1, #2
831c: e3a02003 mov r2, #3
8320: e3a03004 mov r3, #4
8324: ebffffe6 bl 82c4 <func>
8328: e1a00003 mov r0, r3
832c: e8bd8800 pop {fp, pc}
    
```

인자가 4개인 경우

```

00008314 <main>:
8314: e92d4800 push {fp, lr}
8318: e28db004 add fp, sp, #4
831c: e24dd008 sub sp, sp, #8
8320: e3a03005 mov r3, #5
8324: e58d3000 str r3, [sp]
8328: e3a00001 mov r0, #1
832c: e3a01002 mov r1, #2
8330: e3a02003 mov r2, #3
8334: e3a03004 mov r3, #4
8338: ebffffe1 bl 82c4 <func>
833c: e1a00003 mov r0, r3
8340: e24bd004 sub sp, fp, #4
8344: e8bd8800 pop {fp, pc}
    
```

인자가 4개 이상인 경우

Register	용도
R0	1 번째 인자 / 함수 반환 값
R1	2 번째 인자
R2	3 번째 인자
R3	4 번째 인자
R7	Syscall 번호

4개 이상의 인자를 사용하면,
스택을 사용 함.

Analysis C Syntax to ARM Assembly

```
8340:    ebffffcc    bl    8278 <scanf@plt>
8344:    e51b3008    ldr    r3, [fp, #-8]
8348:    e3530005    cmp    r3, #5
834c:    1a000004    bne    8364 <main+0x44>
8350:    e59f3050    ldr    r3, [pc, #80] ; 83a8 <main+0x88>
8354:    e08f3003    add    r3, pc, r3
8358:    e1a00003    mov    r0, r3
835c:    ebffffc8    bl    8284 <puts@plt>
8360:    ea00000b    b      8394 <main+0x74>
8364:    e51b3008    ldr    r3, [fp, #-8]
8368:    e353000a    cmp    r3, #10
836c:    1a000004    bne    8384 <main+0x64>
8370:    e59f3034    ldr    r3, [pc, #52] ; 83ac <main+0x8c>
8374:    e08f3003    add    r3, pc, r3
8378:    e1a00003    mov    r0, r3
837c:    ebffffc0    bl    8284 <puts@plt>
8380:    ea000003    b      8394 <main+0x74>
8384:    e59f3024    ldr    r3, [pc, #36] ; 83b0 <main+0x90>
8388:    e08f3003    add    r3, pc, r3
838c:    e1a00003    mov    r0, r3
8390:    ebffffbb    bl    8284 <puts@plt>
8394:    e3a03000    mov    r3, #0
8398:    e1a00003    mov    r0, r3
839c:    e24bd004    sub    sp, fp, #4
83a0:    e8bd8800    pop    {fp, pc}
```

If-else 구문

Analysis C Syntax to ARM Assembly

```
8340:    ebffffcc    bl    8278 <scanf@plt>
8344:    e51b3008    ldr    r3, [fp, #-8]
8348:    e3530005    cmp    r3, #5
834c:    0a000002    beq    835c <main+0x3c>
8350:    e353000a    cmp    r3, #10
8354:    0a000005    beq    8370 <main+0x50>
8358:    ea000009    b      8384 <main+0x64>
835c:    e59f3048    ldr    r3, [pc, #72] ; 83ac <main+0x8c>
8360:    e08f3003    add    r3, pc, r3
8364:    e1a00003    mov    r0, r3
8368:    ebffffc5    bl     8284 <puts@plt>
836c:    ea000009    b      8398 <main+0x78>
8370:    e59f3038    ldr    r3, [pc, #56] ; 83b0 <main+0x90>
8374:    e08f3003    add    r3, pc, r3
8378:    e1a00003    mov    r0, r3
837c:    ebffffc0    bl     8284 <puts@plt>
8380:    ea000004    b      8398 <main+0x78>
8384:    e59f3028    ldr    r3, [pc, #40] ; 83b4 <main+0x94>
8388:    e08f3003    add    r3, pc, r3
838c:    e1a00003    mov    r0, r3
8390:    ebffffbb    bl     8284 <puts@plt>
8394:    e1a00000    nop                                ; (mov r0, r0)
8398:    e3a03000    mov    r3, #0
839c:    e1a00003    mov    r0, r3
83a0:    e24bd004    sub    sp, fp, #4
83a4:    e8bd8800    pop    {fp, pc}
```

Switch 구문

Analysis C Syntax to ARM Assembly

```
82f4:      e92d4800      push    {fp, lr}
82f8:      e28db004      add     fp, sp, #4
82fc:      e24dd008      sub     sp, sp, #8
8300:      e3a03000      mov     r3, #0
8304:      e50b3008      str     r3, [fp, #-8]
8308:      ea00000a      b       8338 <main+0x44>
830c:      e51b3008      ldr     r3, [fp, #-8]
8310:      e2033001      and     r3, r3, #1
8314:      e3530000      cmp     r3, #0
8318:      1a000003      bne     832c <main+0x38>
831c:      e59f3030      ldr     r3, [pc, #48] ; 8354 <main+0x60>
8320:      e08f3003      add     r3, pc, r3
8324:      e1a00003      mov     r0, r3
8328:      ebffffca      bl      8258 <printf@plt>
832c:      e51b3008      ldr     r3, [fp, #-8]
8330:      e2833001      add     r3, r3, #1
8334:      e50b3008      str     r3, [fp, #-8]
8338:      e51b3008      ldr     r3, [fp, #-8]
833c:      e3530063      cmp     r3, #99 ; 0x63
8340:      daffffff1      ble     830c <main+0x18>
8344:      e3a03000      mov     r3, #0
8348:      e1a00003      mov     r0, r3
834c:      e24bd004      sub     sp, fp, #4
8350:      e8bd8800      pop     {fp, pc}
```

Analysis C Syntax to ARM Assembly

```
8320:    e92d4800    push    {fp, lr}
8324:    e28db004    add     fp, sp, #4
8328:    e24dd008    sub     sp, sp, #8
832c:    e59f3058    ldr     r3, [pc, #88] ; 838c <main+0x6c>
8330:    e08f3003    add     r3, pc, r3
8334:    e1a00003    mov     r0, r3
8338:    ebffffce    bl      8278 <printf@plt>
833c:    e24b2008    sub     r2, fp, #8
8340:    e59f3048    ldr     r3, [pc, #72] ; 8390 <main+0x70>
8344:    e08f3003    add     r3, pc, r3
8348:    e1a00003    mov     r0, r3
834c:    e1a01002    mov     r1, r2
8350:    ebffffcb    bl      8284 <scanf@plt>
8354:    e51b3008    ldr     r3, [fp, #-8]
8358:    e3530000    cmp     r3, #0
835c:    1a000003    bne     8370 <main+0x50>
8360:    e3a03000    mov     r3, #0
8364:    e1a00003    mov     r0, r3
8368:    e24bd004    sub     sp, fp, #4
836c:    e8bd8800    pop     {fp, pc}
8370:    e51b2008    ldr     r2, [fp, #-8]
8374:    e59f3018    ldr     r3, [pc, #24] ; 8394 <main+0x74>
8378:    e08f3003    add     r3, pc, r3
837c:    e1a00003    mov     r0, r3
8380:    e1a01002    mov     r1, r2
8384:    ebffffbb    bl      8278 <printf@plt>
8388:    eaffffe7    b       832c <main+0xc>
```

Real world Example #1

- Target Third Party Application



- 목적 : Android App에 대한 동적 디버깅 예제.
- 사용 Tool : gdb, gdbserver

Real world Example #2

- Target Application



- 목적 : Android App에 대한 취약성 점검
- 사용 Tool : gdb, gdbserver, IDA



References

1. ARM Architecture Reference Manual ARMv7-A and ARMV7-R / arm.com
2. ARM으로 배우는 임베디드 시스템 / 안효복 저 / 한빛미디어