ARM RE

“ARM basic reverse engineering과 ARM용 Packer의 이해” V0.1

SeungJin Beist Lee

ARM Assembly

- RISC(ARM) is more simpler than CISC(x86)
- Non-aligned vs aligned instruction
- More easy to understand

ARM Assembly

- ARM CPU
  - ARM mode (4byte instruction)
  - Thumb mode (2byte instruction)
  - Thumb-2 mode (2 or 4byte instruction) - from ARMv7

ARM Assembly

- As a reverse engineer
  - Branch instructions are many many in a program
  - You can’t catch up what the program is doing if you’re not familiar with branch instructions
  - Which means understanding “*Branch*” is important

Branches

- There are 2 kind of branches
  - Same level branch
    - Ring3 to Ring3 or Ring0 to Ring0
    - EX) User to User or Kernel to Kernel
  - Different level branch
    - Ring3 to Ring0
    - EX) User to Kernel

Branch

- Same level branch
  - Usually, function to function
    - And returning
  - Or jumping to an address which is in the same function
  - On x86: call, *jmp, jz, jne, jl and etc*
  - On ARM: b, bl, bx, blx and etc

Branch

- Different level branch
  - Ring3 to Ring0
  - User level to Kernel level
    - And back to User level, usually
  - On x86: *sysenter*, *int 0x80* and etc
  - On ARM: *svc* and etc

Registers

- On x86
  - General registers: eax, ebx, ecx, edx and +++
  - Special registers: esp, ebp, eip and +++
  - Segment registers (SS, DS and etc)
  - eip is special, you can't set a value to eip for example
  - eax - 4byte, ax - 2byte, al -1byte

Registers

- On ARM
  - From R0 to R15
  - R15 is used like EIP (called PC)
  - R14 is Link-Register (called LR)
  - R13 is ESP (called SP)
  - But all registers including R13 to R15 are completely general
  - You can set a value to R15 directly (Impossible on x86)

User level jump practice

- Generally, there are 2 ways for this
  - "*bl*" (or "*blx*") and "*bx lr*" pair
  - "*push*" and "*ldm*" pair

Different level jump practice

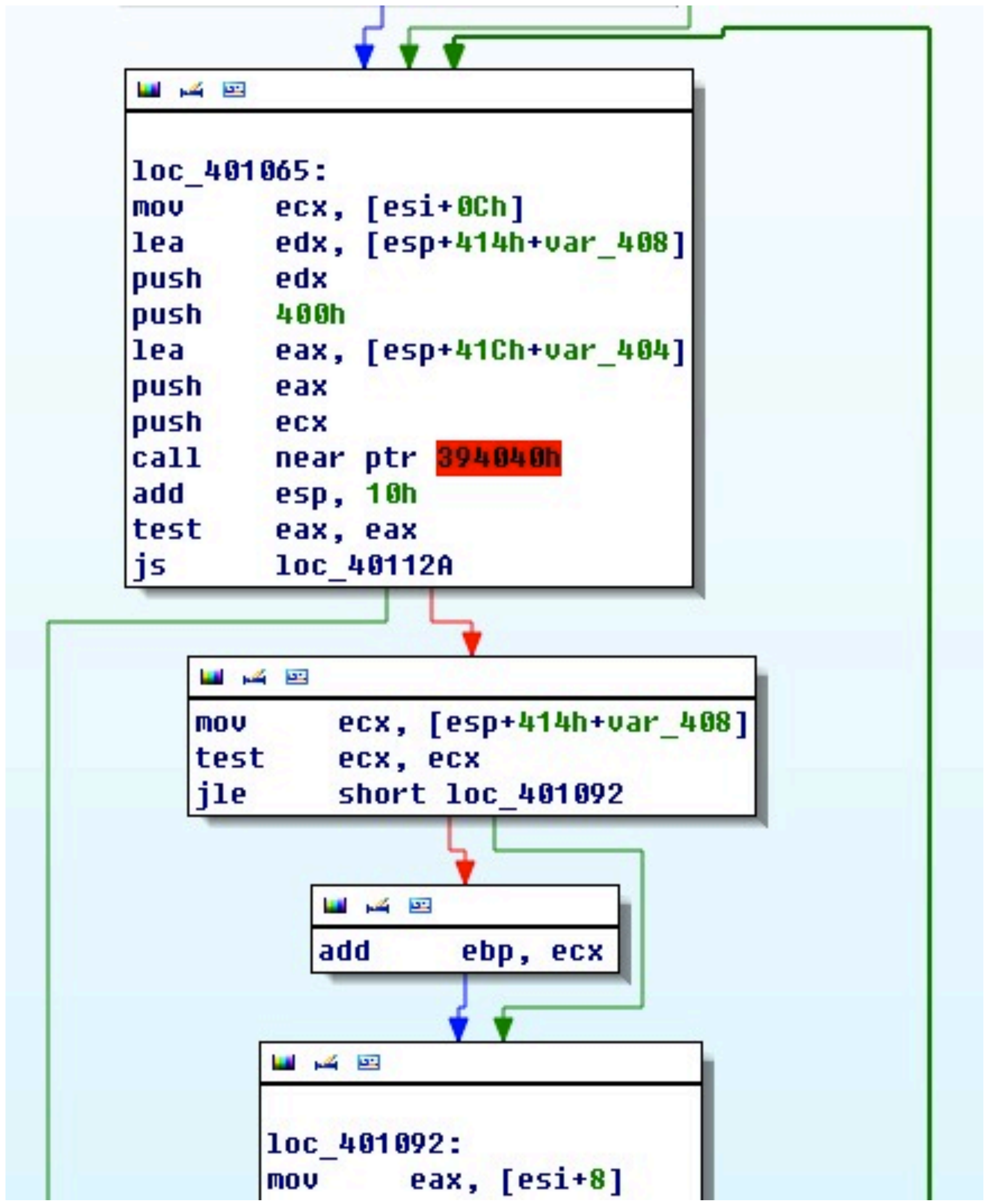
- When you want to execute system calls
  - General way - executing "*svc*" or "*swi*" instruction
    - They are the same
  - read, write, exec, exit, mmap and etc
  - r0 to r3 are used for arguments
    - stack is used if there are more

IDA time

- IDA is the industry standard tool
  - Official website: http://www.hex-rays.com
  - For figuring out how a program works
  - Can be used for reversing, hunting bugs and etc
  - Powerful tool for both static and dynamic guys
  - The best IDA book: "*The IDA pro book*" - by Chris Eagle

IDA screen shot

- Easy to understand code
- Control flow graphs



```
loc_401065:
mov     ecx, [esi+0Ch]
lea     edx, [esp+414h+var_408]
push    edx
push    400h
lea     eax, [esp+41Ch+var_404]
push    eax
push    ecx
call    near ptr 394040h
add     esp, 10h
test    eax, eax
js      loc_40112A
```

```
mov     ecx, [esp+414h+var_408]
test    ecx, ecx
jle     short loc_401092
```

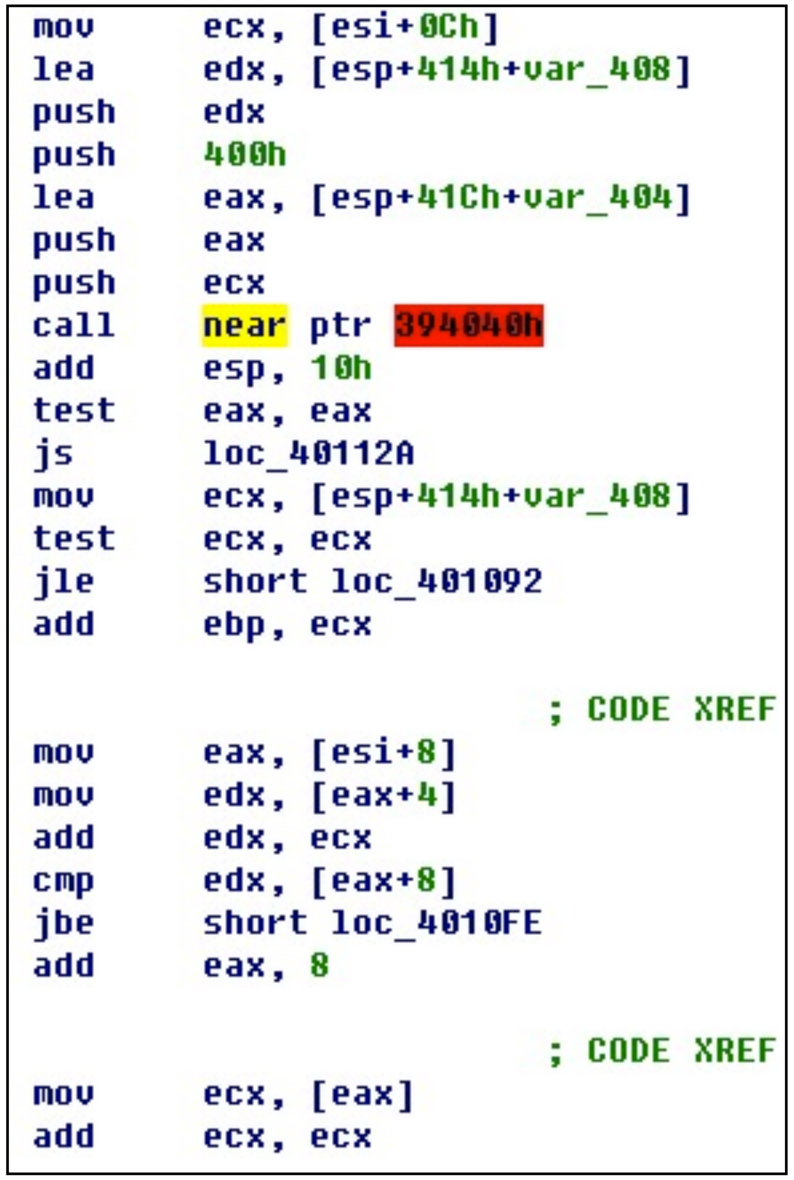
```
add     ebp, ecx
```

```
loc_401092:
mov     eax, [esi+8]
```

What we should do with IDA

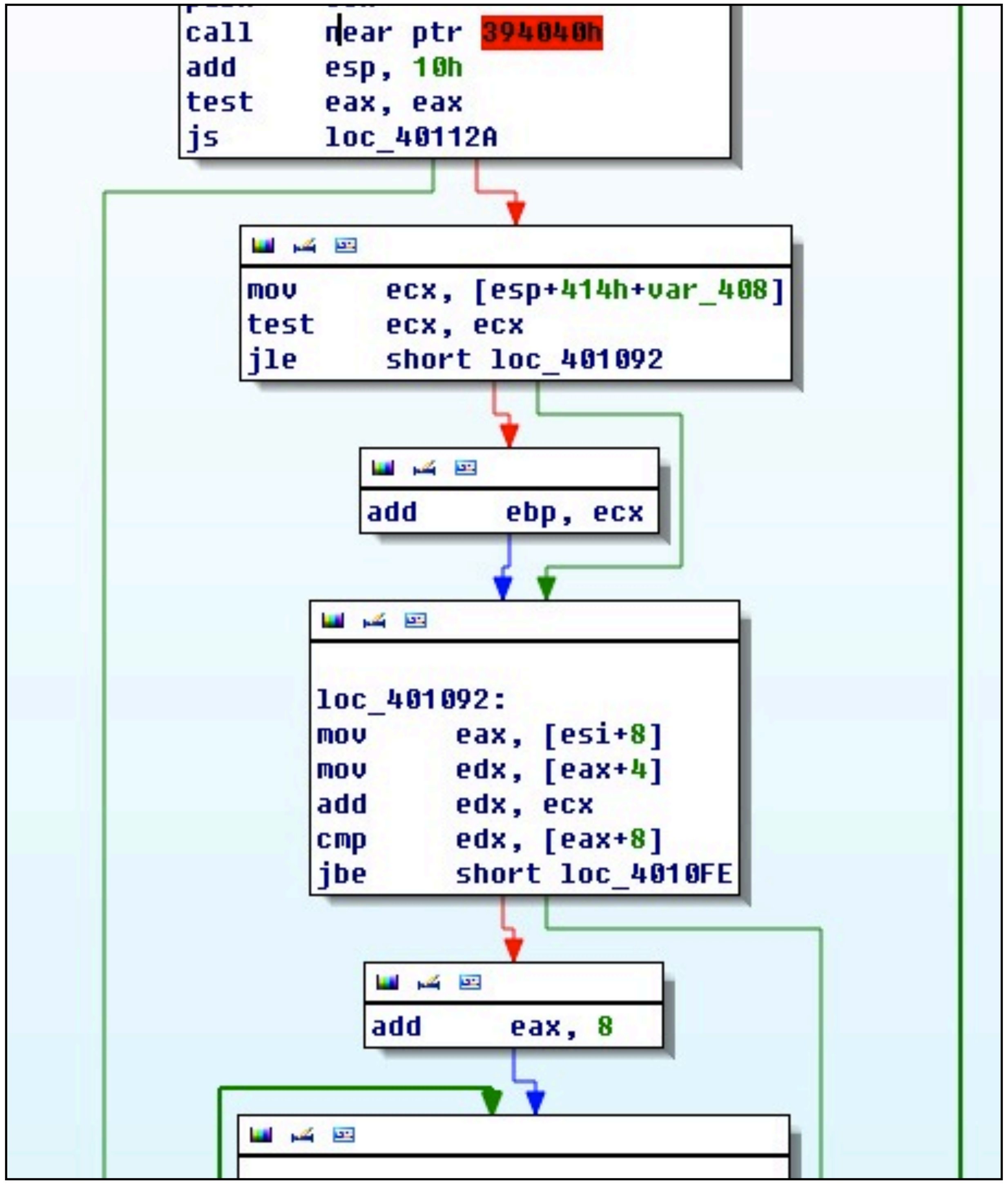
- Hackers like IDA because
  - The graphs (To understand code)
  - The note and edit feature (Can be shared as well)
  - Plugins (You can make your own)
  - Script (Automation makes our life easier)

The graphs



```
mov     ecx, [esi+0Ch]
lea     edx, [esp+414h+var_408]
push    edx
push    400h
lea     eax, [esp+41Ch+var_404]
push    eax
push    ecx
call    near ptr 394040h
add     esp, 10h
test    eax, eax
js      loc_40112A
mov     ecx, [esp+414h+var_408]
test    ecx, ecx
jle     short loc_401092
add     ebp, ecx

                        ; CODE XREF
mov     eax, [esi+8]
mov     edx, [eax+4]
add     edx, ecx
cmp     edx, [eax+8]
jbe     short loc_4010FE
add     eax, 8

                        ; CODE XREF
mov     ecx, [eax]
add     ecx, ecx
```

vs

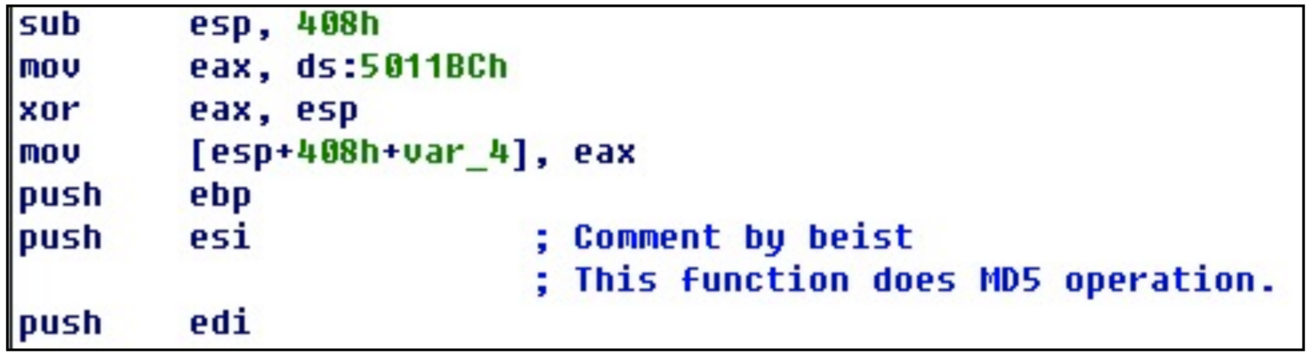


```
call    near ptr 394040h
add     esp, 10h
test    eax, eax
js      loc_40112A

mov     ecx, [esp+414h+var_408]
test    ecx, ecx
jle     short loc_401092

add     ebp, ecx

loc_401092:
mov     eax, [esi+8]
mov     edx, [eax+4]
add     edx, ecx
cmp     edx, [eax+8]
jbe     short loc_4010FE

add     eax, 8
```

The graphs

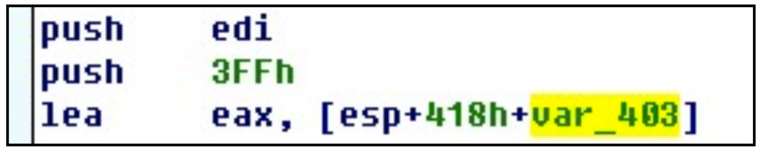
- It depends on a hacker
  - Some hackers more like the *text* view
  - But at least for beginners, CFG is definitely helpful
  - You can easily recognize the flows
    - Green and Red lines
  - You should try to use some useful features
    - Set colors on basic blocks for example

The note and edit feature

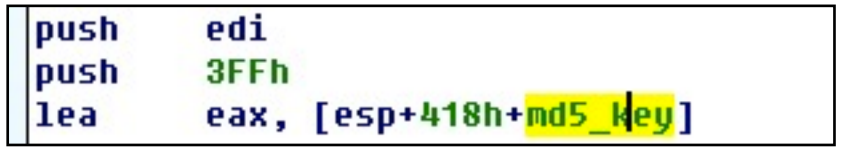


```
sub     esp, 408h
mov     eax, ds:5011BCh
xor     eax, esp
mov     [esp+408h+var_4], eax
push    ebp
push    esi                 ; Comment by beist
                            ; This function does MD5 operation.
push    edi
```

[comment]



```
push    edi
push    3FFh
lea     eax, [esp+418h+var_403]
```



```
push    edi
push    3FFh
lea     eax, [esp+418h+md5_key]
```

[variable rename]

The note and edit feature

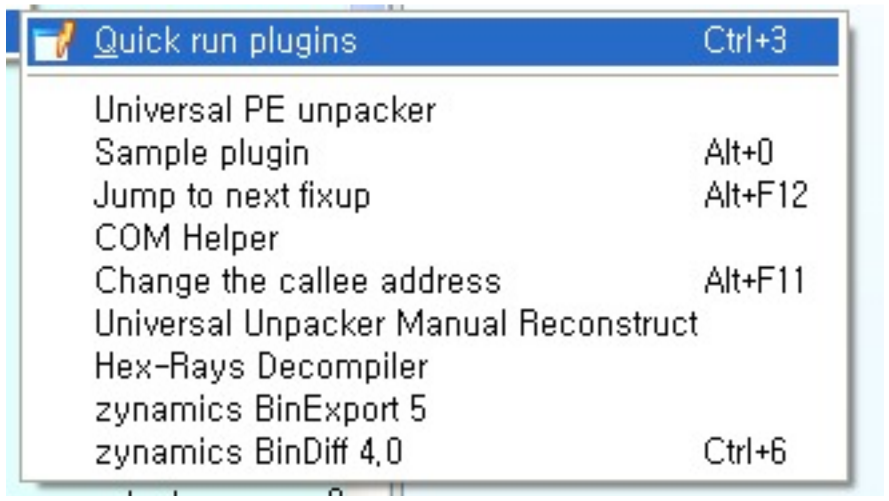
- We can note/edit almost everything
  - local variable, argument, function rename
  - Every time when you note a new thing which you just found, it'll be cleared more and more
  - This is extremely useful if you co-work with somebody

The note and edit feature

- Comment is more important than you think
    - Again, when you co-work
    - Modern programs are huge
    - Update_md5() is much better than sub_0x401323()

IDA Plugins

- Hackers are all around the world
  - They make a lot of tools
  - The tools are also very useful for developers


Quick run plugins Ctrl+3
Universal PE unpacker
Sample plugin Alt+0
Jump to next fixup Alt+F12
COM Helper
Change the callee address Alt+F11
Universal Unpacker Manual Reconstruct
Hex-Rays Decompiler
zynamics BinExport 5
zynamics BinDiff 4.0 Ctrl+6

- You can make your own plugin using IDA SDK by the company

Script feature

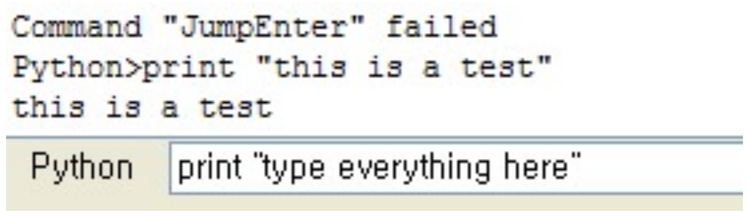
- Script is useful for tedious jobs
- 2 scripts are popular on IDA
  - IDC (by IDA company, it’s like C style)
  - IDAPython
    - Literally, it is python
    - Python interpreter is integrated

Script feature

- Can be very complicated
  - Automatic de-obfuscation, unpacking, finding interesting functions, SMT and etc
- We'll try some basic stuff using IDAPython
- Easy, but it will be a good start

IDAPython

- 3 ways to run idapython
  - The interpreter window



```
Command "JumpEnter" failed
Python>print "this is a test"
this is a test
Python  print "type everything here"
```

  - Or via “File - Script file”
  - Or via “File - Script command”

IDA API

- How do you count functions in a program?
  - gdb? ollydbg? and count function by function?
  - objdump could be a good option but
  - in IDAPython, you only need 3 lines
- IDA supports useful APIs for users
  - http://www.hex-rays.com/products/ida/support/idapython_docs/

IDAPython practice

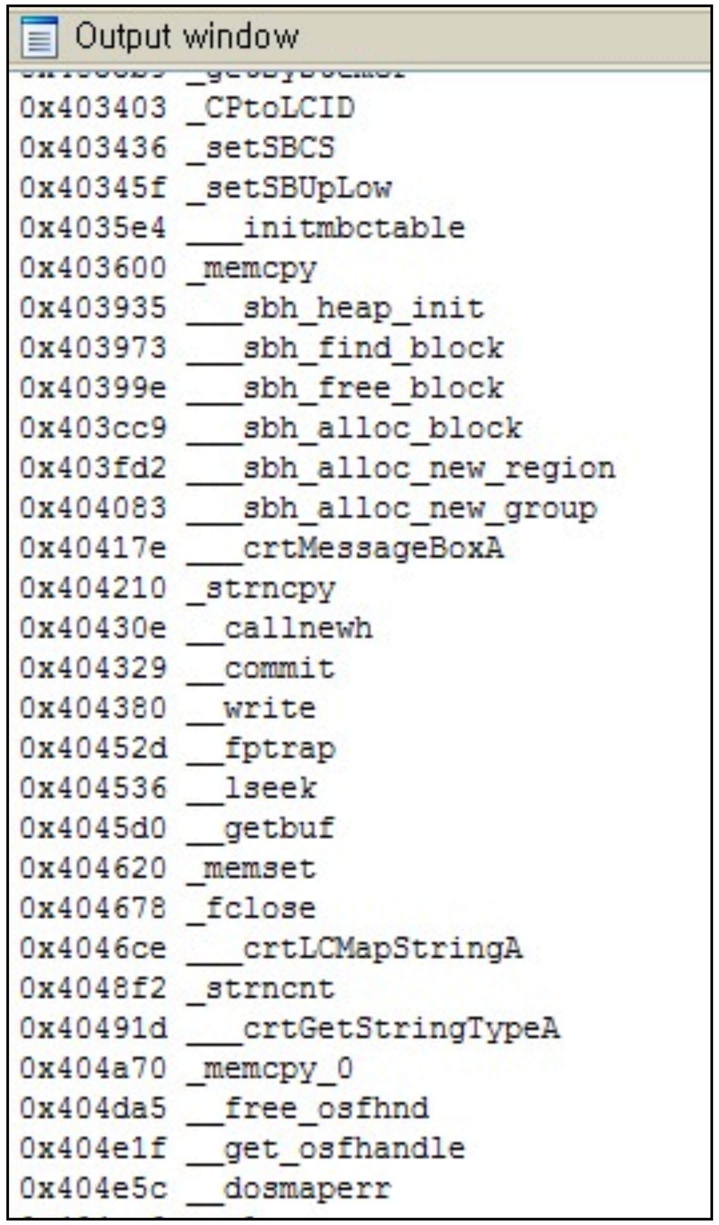
- Again, how do you count functions in a program?
  - in IDAPython, only 3 lines needed

```
ea = ScreenEA()

for function_ea in Functions(SegStart(ea), SegEnd(ea)):
  print hex(function_ea), GetFunctionName(function_ea)
```

- Target
  - c:\windows\system32\calc.exe
  - Load the binary on IDA
- Save the file and execute on IDA

IDAPython practice

- Works like magic!



Output window

```
0x403403 _CPtoLCID
0x403436 _setSBCS
0x40345f _setSBUpLow
0x4035e4 ___initmbctable
0x403600 _memcpy
0x403935 ___sbh_heap_init
0x403973 ___sbh_find_block
0x40399e ___sbh_free_block
0x403cc9 ___sbh_alloc_block
0x403fd2 ___sbh_alloc_new_region
0x404083 ___sbh_alloc_new_group
0x40417e ___crtMessageBoxA
0x404210 _strncpy
0x40430e __callnewh
0x404329 __commit
0x404380 __write
0x40452d __fptrap
0x404536 __lseek
0x4045d0 __getbuf
0x404620 _memset
0x404678 _fclose
0x4046ce ___crtLCMapStringA
0x4048f2 _strncnt
0x40491d ___crtGetStringTypeA
0x404a70 _memcpy_0
0x404da5 __free_osfhnd
0x404e1f __get_osfhandle
0x404e5c __dosmaperr
```

IDA API

- ScreenEA()
  - Get the segment's starting address
- Functions()
  - Get functions
- SegStart(), SegEnd
  - Start of segment and end of segment
- GetFunctionName()
  - It returns a function name of given address

IDAPython practice

- Get to the reference page
    - http://www.hex-rays.com/products/ida/support/idapython_docs/
- Mission: make a script which prints instructions that are used in a program
- Target
    - http://115.68.24.145/armtest/installer_arm_strip
- Hint API:
    - Heads(), isCode(), GetFlags(), GetMnem()

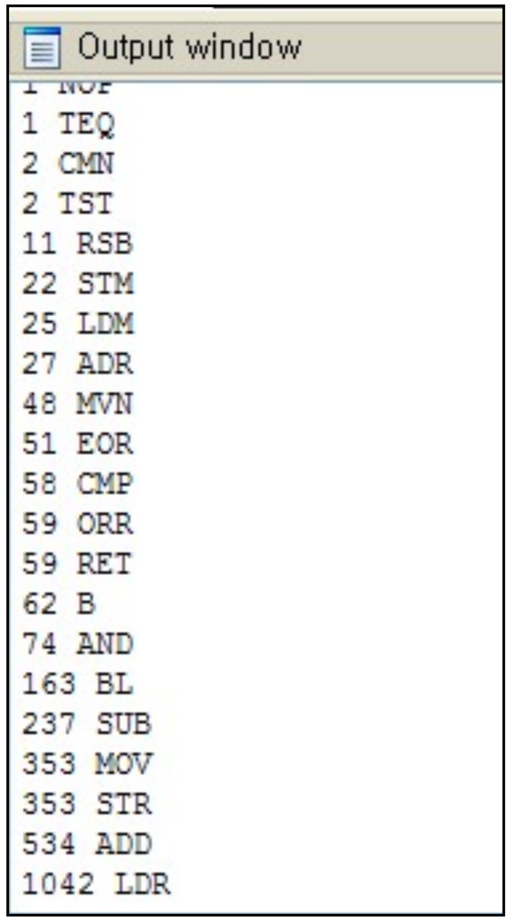
IDAPython practice

```
mnemonics = dict()

# For each of the segments
for seg_ea in Segments():

    # For each of the defined elements
    for head in Heads(seg_ea, SegEnd(seg_ea)):

        # If it's an instruction
        if isCode(GetFlags(head)):

            # Get the mnemonic and increment the mnemonic
            # count
            mnem = GetMnem(head)
            mnemonics[mnem] = mnemonics.get(mnem, 0)+1

# Sort the mnemonics by number of occurrences
sorted = map(lambda x:(x[1], x[0]), mnemonics.items())
sorted.sort()


# Print the sorted list
for mnemonic, count in sorted:
    print mnemonic, count
```

[counting instructions]
from Ero's example script

IDAPython practice

- The output of counting instructions
- Let's back to ARM Assembly



```
1 NOP
1 TEQ
2 CMN
2 TST
11 RSB
22 STM
25 LDM
27 ADR
48 MVN
51 EOR
58 CMP
59 ORR
59 RET
62 B
74 AND
163 BL
237 SUB
353 MOV
353 STR
534 ADD
1042 LDR
```

ARM Assembly basic

- As this is not an arm assembly lecture, we'll cover only popular instructions
- LDR, STR, MOV, PUSH, POP, B, BL, BX, BLX, SUB, ADD, CMP, SVC
- We'll ignore some ARM features like pre-index or some others

ARM Assembly basic

- mov (move)
  - mov r1, r2 // r1 = r2
  - mov r1, #0x80 // r1 = 0x80
- push
  - push 0x10 // push 0x10 onto stack
  - push {r1} // push r1 register onto stack
  - push {r1-r5} // push r1, r2, r3, r4, and r5 onto stack
- pop
  - Same as push

ARM Assembly basic

- LDR (Load)
  - ldr r1, [r2] // r1 = *r2
  - ldr r1, [r2+#0x10] // r1 = *(r2+0x10)
- STR (Store)
  - str r1, [r2] = // *r2 = r1
  - str r1, [r2+#0x1] = // *(r2+1) = r1

ARM Assembly basic

- B / BL (Branch)
  - B 0x8080 (Jump to 0x8080 address)
  - BL 0x8080 (Jump to 0x8080 and save next instruction address of current into *LR* register)
- BX / BLX
  - Same as B/BL but operands are registers

ARM Assembly basic

- ADD
  - add r1, r2 // r1 = r1 + r2
  - add r1, #0x10 // r1 = r1 + 0x10
  - add r1, r2, r3 // r1 = r2 + r3
  - add r1, r2, #0x10 // r1 = r2 + 0x10
- SUB
  - Same as add

ARM Assembly basic

- CMP (Compare)
  - cmp r1, r2 // compare r1 and r2
  - cmp r1, #0x10 // compare r1 and 0x10
  - cmp is mostly used before branch instructions
  - Flags are updated after this instruction
- SVC
  - svc #0x900004 // calling sys_write

ARM Assembly basic

- More about branches
  - ARM instructions have post-fix, “EQ” for example
  - Example)
    - B 0x8080 // Just jump to 0x8080
    - BEQ 0x8080 // Jump to 0x8080 if equal
  - The condition check, “*equal*”, is based on the result of instructions like “*CMP*”

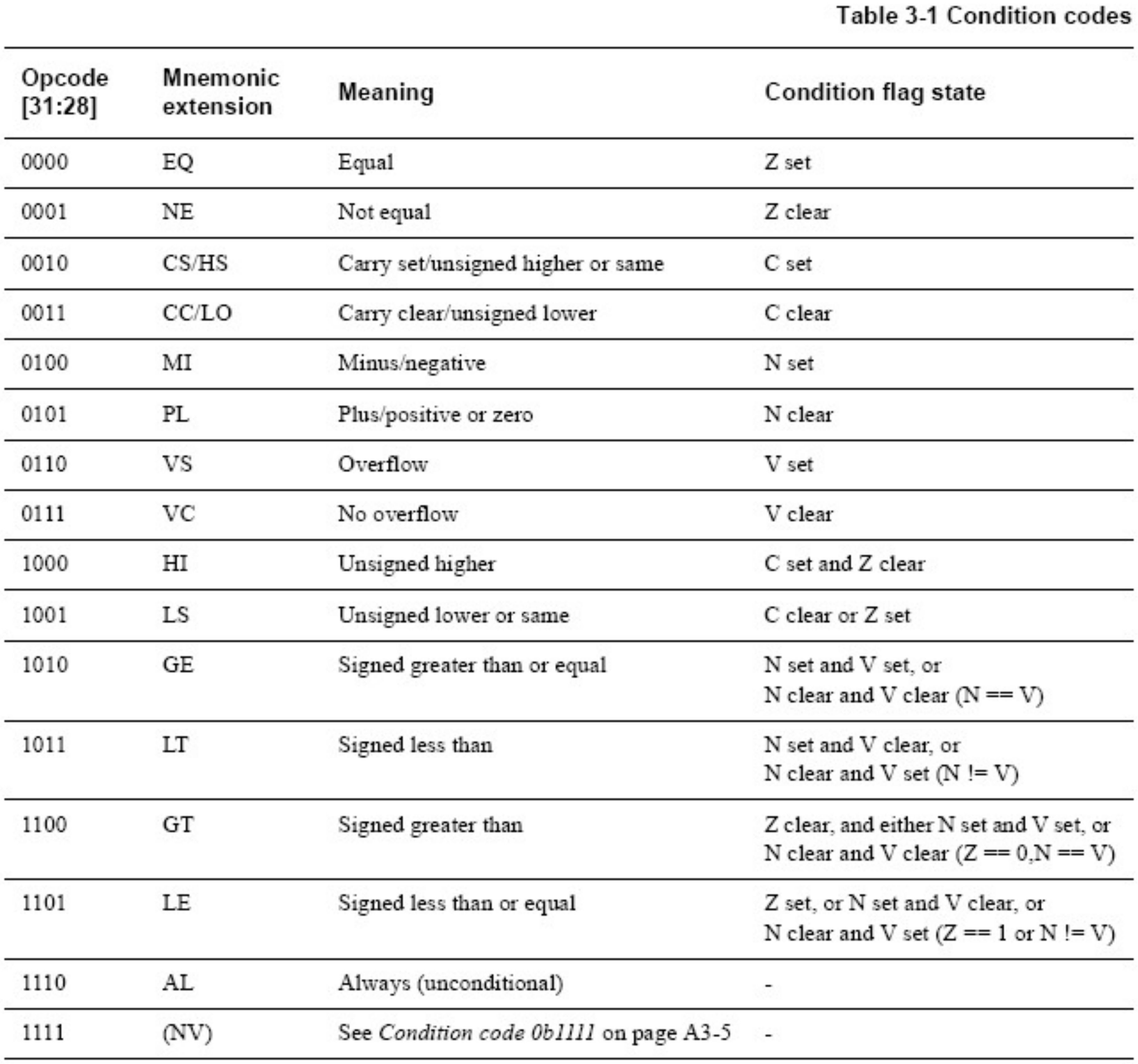


Table 3-1 Condition codes

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	(NV)	See *Condition code 0b1111* on page A3-5	-

[condition codes - from internet]

ARM Warming-up #1

- Make a program that prints “samsung”
- To print the message, you use “*svc*” instruction
- Which means executing system call

Warm-up #1: system call

- Check out: /usr/include/asm/unistd.h
- It contains all system call numbers
- It starts from 0x900000 and every system call has its own number
  - Example) write: 0x900004 (0x900000 + 0x4)
- The system call number is passed to *SVC* instruction as an argument
  - Example) svc 0x900004

Warm-up #1: write function

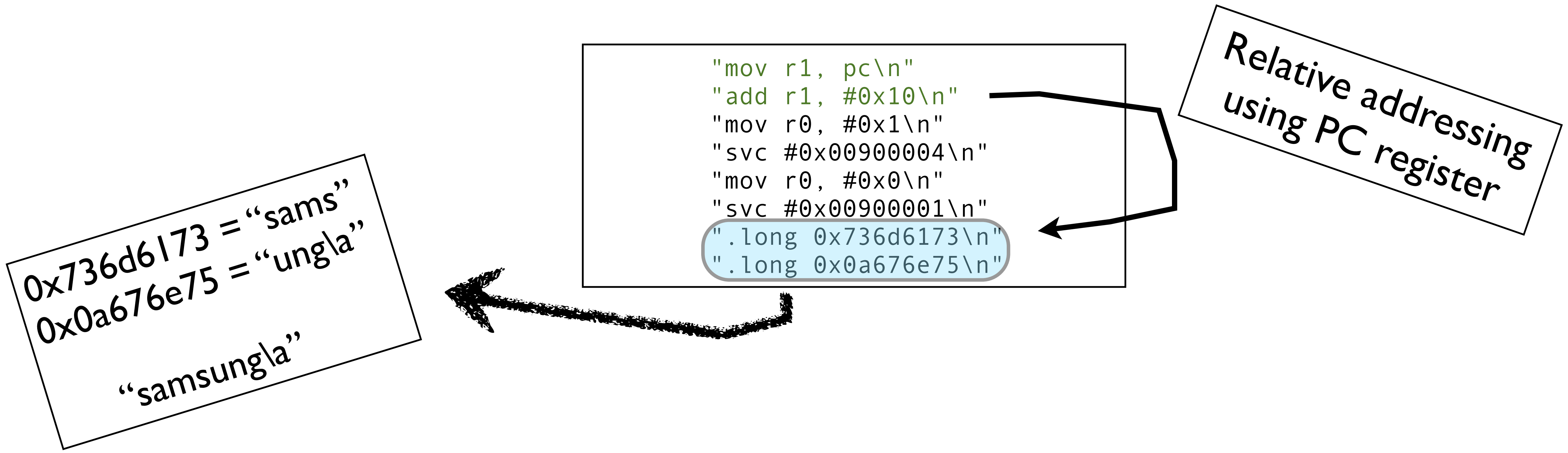
- High level example
  - write(descriptor, buffer, length, 0);
- In assembly level
  - descriptor - r0 (*mov r0, stdout*) (*stdout* is 1)
  - buffer - r1 (*mov r1, pointer_to_buffer*)
  - length - r2 (*mov r2, #0x8*)
  - 0 - r3 (*mov r3, #0x0*)

Warm-up #1: set values

- Setting a constant to a register is easy
  - mov r0, #0x10
- Setting a pointer for character buffer to a register is a little bit tricky
  - write(descriptor, buffer, length, 0);
  - write(constant, ***pointer***, constant, constant);
  - But where we should put our string?

Warm-up #1: character buffer

- Solution
  - push string after code and find them using *PC* register
  - Relative addressing



```
"mov r1, pc\n"
"add r1, #0x10\n"
"mov r0, #0x1\n"
"svc #0x00900004\n"
"mov r0, #0x0\n"
"svc #0x00900001\n"
".long 0x736d6173\n"
".long 0x0a676e75\n"
```

Warm-up #1: The code

- Example

```
main() {
        asm(
        "mov r3, #0x0\n"
        "mov r2, #0x8\n"
        "mov r1, pc\n"
        "add r1, #0x10\n"
        "mov r0, #0x1\n"
        "svc #0x00900004\n"
        "mov r0, #0x0\n"
        "svc #0x00900001\n"
        ".long 0x736d6173\n"
        ".long 0x0a676e75\n"
        );
}
```

[write_samsung.c]

ARM Warming-up #2

- Make two labels at least (example: "*test_label:*")
  - Let's say *A()* and *B()* - *We assume they're functions*
- *A()* calls *B()*
- *B()* does - set 0x1337 to r0
- And back to *A()*
  - Then *A()* does

```
if (r0 == 0x1337)
  write(stdout, "go!\n", 4, 0);
else
  write(stdout, "no!\n", 4, 0);
exit(0);
```

- Use *BL* - *BX* pair for *A()* - *B()* - *A()*

Warming-up #2: Use labels

```
main() {
   asm(
      “label_a:\n”
         “bl label_b\n”
      “label_b:\n”
         “bx lr\n”
)
```

Warming-up #2: ARM limits

- Unfortunately, you are only able to load a limited range of immediate values with *mov*
    - *Reference: http://blogs.arm.com/software-enablement/251-how-to-load-constants-in-assembly-for-arm-architecture/*
- ARM has fixed size instructions (2 or 4 byte)
- Which means you can't move immediate values over the limits
- Also, it's bad that we can't use *movw* and *movt* instructions as our CPU doesn't suppor it

Warming-up #2: ARM limits

- GCC on our machine doesn’t compile this code

```
main() {
   asm(
      “mov r1, #0x1337\n”
   );
}
```

Error: “invalid constant (1337) after fixup”

Warming-up #2: ARM limits

- *ldr* is our friend
- *r1* register will have 0x1337

```
main() {
   asm(
      “ldr r1, [pc]\n”
      “b go_next\n”
      “.long 0x00001337\n”
   );
}
```

- But, using *ldr* or *mov* to get data which is at code text is sometimes annoying if you don’t calculate the offsets well (*be careful!*)

Warming-up #2: The code

```
main() {

        asm(
        "label_a:\n"
                "bl label_b\n"
                "ldr r1, [pc]\n"
                "b go_next\n"
                ".long 0x00001337\n"

        "go_next:\n"
                "cmp r0, r1\n"
                "beq set_yes\n"
                "mov r4, pc\n"
                "add r4, #0x2c\n"
                "mov r1, r4\n"
                "b go_svc\n"

        "set_yes:\n"
                "mov r4, pc\n"
                "add r4, #0x18\n"
                "mov r1, r4\n"
```

```
        "go_svc:\n"
                "mov r3, #0x0\n"
                "mov r2, #0x4\n"
                "mov r0, #0x1\n"
                "svc #0x900004\n"
                "b go_exit\n"
                // go!
                ".long 0x0a216f67\n"
                // no!
                ".long 0x0a216f6e\n"

        "go_exit:\n"
                "mov r0, #0x0\n"
                "svc #0x900001\n"

        "label_b:\n"
                "ldr r0, [pc]\n"
                "bx lr\n"
                ".long 0x00001337\n"
        );
}
```

[warm-up2.c]

messed up, but still works..

GDB practice

- Basic gdb commands
  - b (breakpoint) -- b main 혹은 b *0x83c8
  - run (run the target program)
  - step (step into) -- stepi
  - info (to get information)
  - x (to see memory) -- x/10i 0x83c8 혹은 x/10x 0x83c8
- Compile warm-up2.c and type
  - *# gdb warm-up2*

IDAPython practice

- Mission: Make a script that prints out what functions call *strcpy()* in a program
- Target
  - http://115.68.24.145/armtest/installer_arm_strip
- Hint API:
  - GetFunctionName(), CodeRefsTo()

Update routine bug practice

- Concept
  - It is very often that developers make a security hole in a update module
  - They sometimes use a custom hash algorithm (or slightly different version of popular hash)
- Goal
  - If you solve this challenge, you'll be able to execute your whatever evil program

File identity

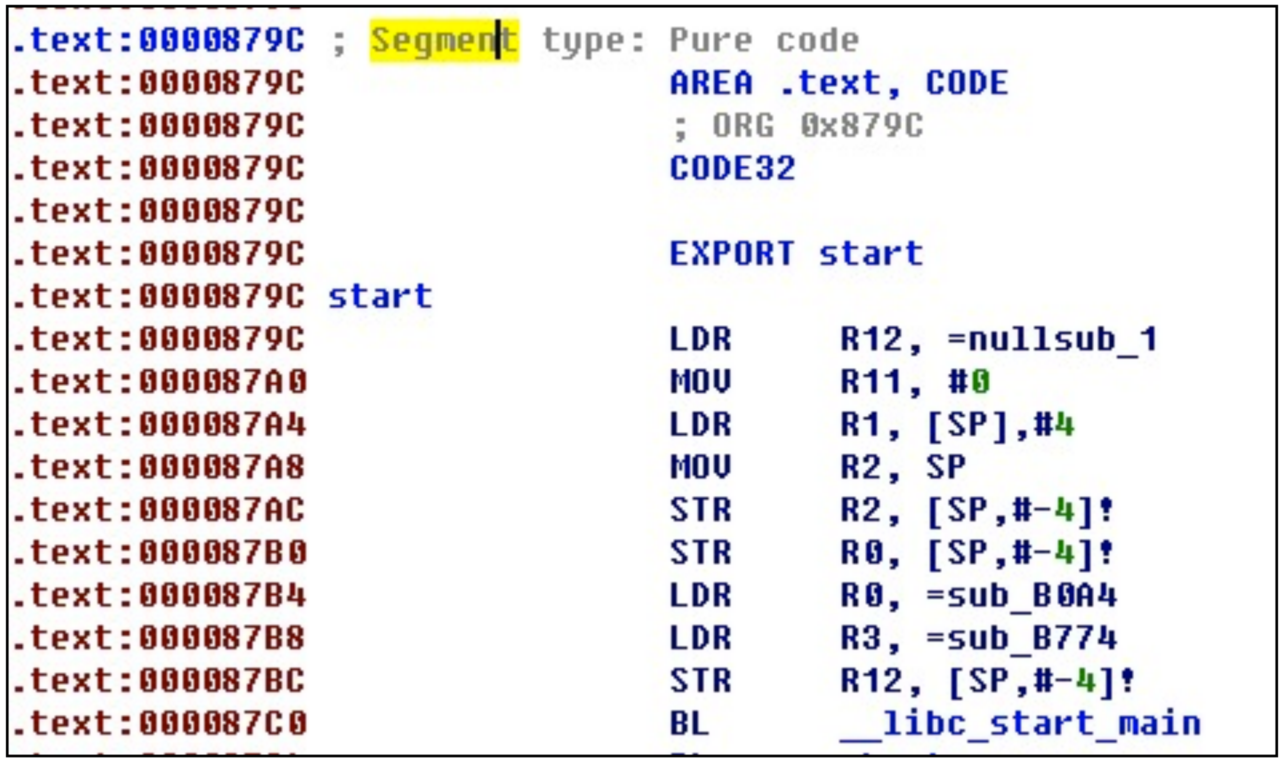
- We don’t know anything about this program
- We are not going to do dynamic-analysis
- debian-arm:~#
    - type - file installer
    - result - “ELF 32-bit LSB executable, ARM, version 1, dynamically linked (uses shared libs), for GNU/Linux 2.6.12, stripped”
- ”Ouch.. Is it a stripped binary..?”

Update routine bug practice

- What should we figure out
  - How the update program works
  - The format of update config file
  - What hash algorithm it uses
  - The integrity check routine
  - How this program tries to hide KEY
- Let’s fire up IDA and load this binary
  - http://115.68.24.145/armtest/installer_arm_strip

Where is the main()?

- So, where is the main, if available?
    - Ok, where we are, now?



```
.text:0000879C ; Segment type: Pure code
.text:0000879C                 AREA .text, CODE
.text:0000879C                 ; ORG 0x879C
.text:0000879C                 CODE32
.text:0000879C
.text:0000879C                 EXPORT start
.text:0000879C start
.text:0000879C                 LDR     R12, =nullsub_1
.text:000087A0                 MOV     R11, #0
.text:000087A4                 LDR     R1, [SP],#4
.text:000087A8                 MOV     R2, SP
.text:000087AC                 STR     R2, [SP,#-4]!
.text:000087B0                 STR     R0, [SP,#-4]!
.text:000087B4                 LDR     R0, =sub_B0A4
.text:000087B8                 LDR     R3, =sub_B774
.text:000087BC                 STR     R12, [SP,#-4]!
.text:000087C0                 BL      __libc_start_main
```

We see *__libc_start_main*, but that's not the *main*.. anyway

__libc_start_main()?

- Let's read an article about "*How main() is executed on Linux*"
  - http://linuxgazette.net/issue84/hawk.html
- So, we can assume that we are at a place that is right before executing *main()*

definition for libc_start_main

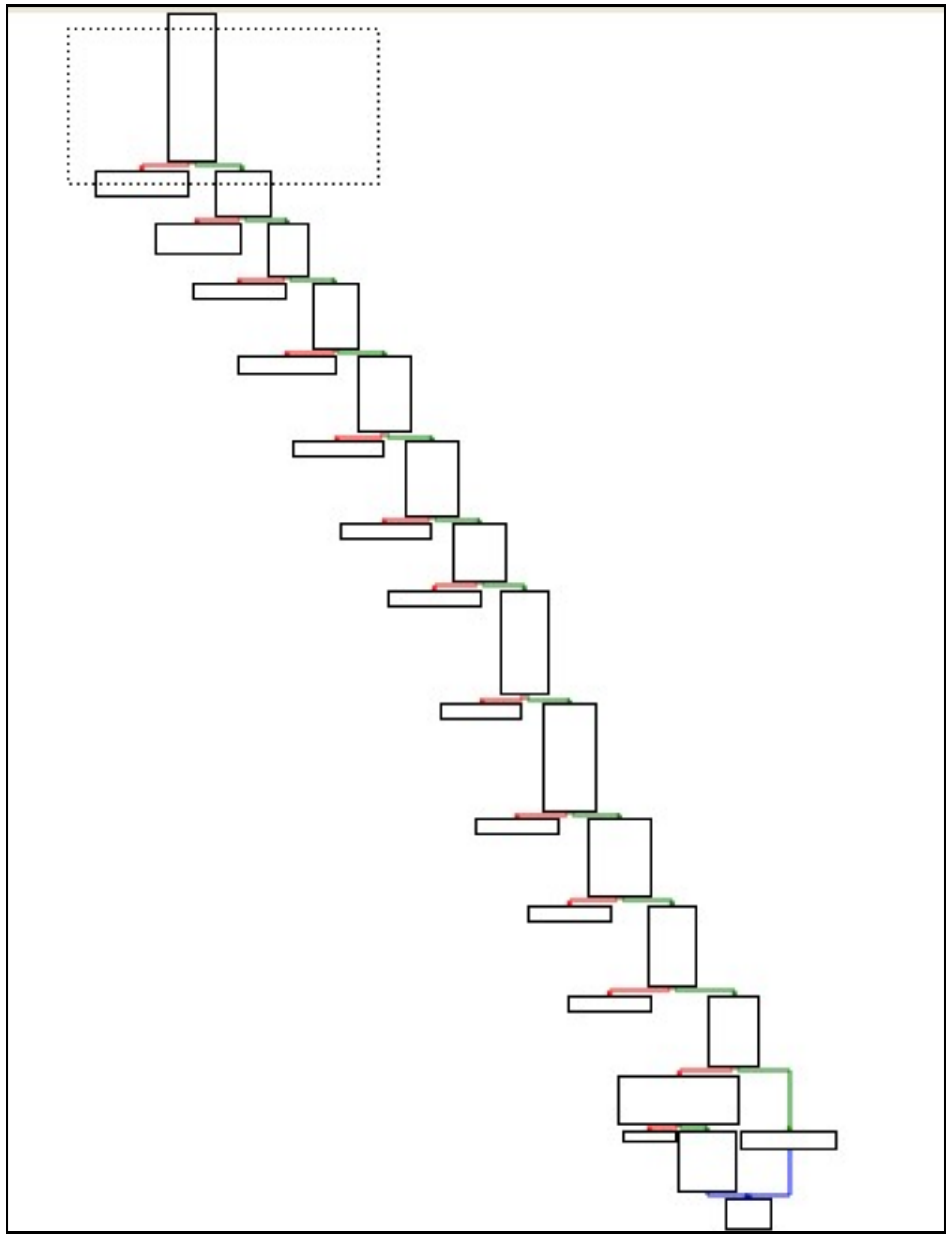
- Check out the web page to see __libc_start_main
    - extern int BP_SYM (__libc_start_main) (int (*main) (int, char **, char **),
- We know that R0 to R3 are used for arguments
- Argument 1 of __libc_start_main() is main's address as we can see
- Which means R0 should have the address
- Let's check out

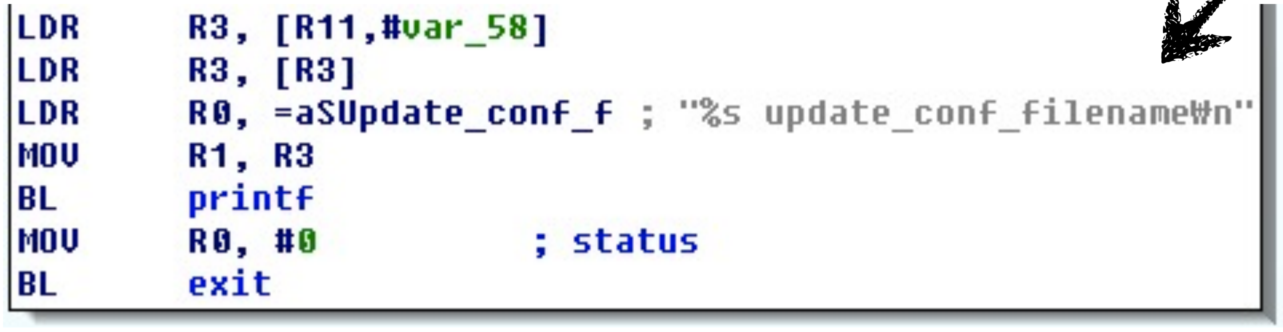
main() address

- .text:000087B4   LDR   R0, =sub_B0A4
  - R0 will be set as sub_B0A4
  - Jump to 0xB0A4
    - Double click “sub_B0A4” on IDA

main()

- The graph overview
- A lot of branches
- Error messages look the binary's, but not library's

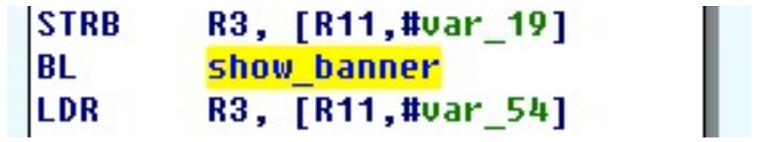




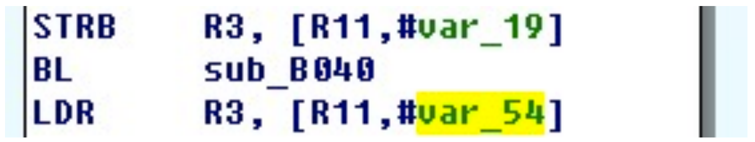
```
LDR     R3, [R11,#var_58]
LDR     R3, [R3]
LDR     R0, =aSUpdate_conf_f ; "%s update_conf_filename\n"
MOV     R1, R3
BL      printf
MOV     R0, #0          ; status
BL      exit
```

Stripped binary

- It seems we've just found the main()
- But what is a stripped binary?
  - A binary which symbol information is removed



```
STRB    R3, [R11,#var_19]
BL      show_banner
LDR     R3, [R11,#var_54]
```



```
STRB    R3, [R11,#var_19]
BL      sub_B040
LDR     R3, [R11,#var_54]
```

- It makes reversing much harder as hackers can't get names of functions
  - EX) 0x8048230() VS SHA256_INIT()

How to make it stripped

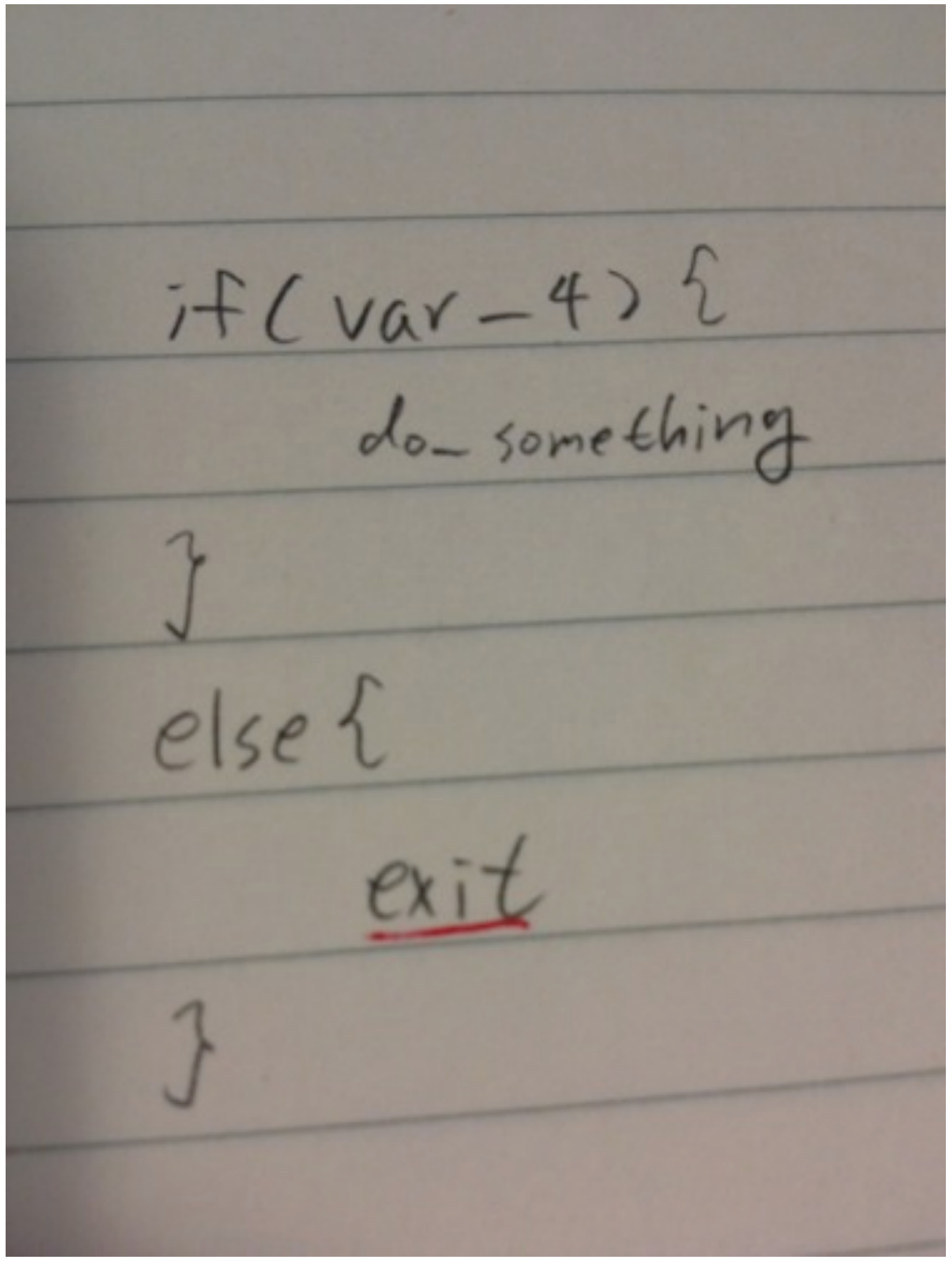
- On Linux for ELF file
  - # /usr/bin/strip ELF_FILE

```
# ls -al strip_test
-rwxr-xr-x 1 beist beist 7123 2012-09-14 05:23 strip_test
# /usr/bin/strip strip_test
# ls -al strip_test
-rwxr-xr-x 1 beist beist 5512 2012-09-14 05:23 strip_test
```

[strip before and after filesize]

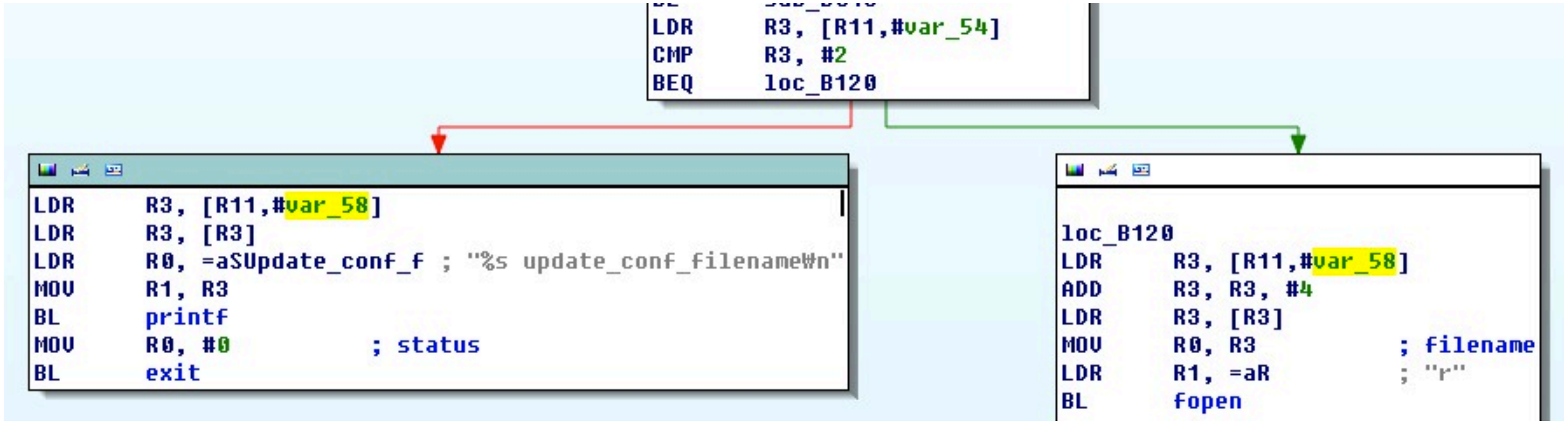
Many branches

- Branches make us crazy sometimes
  - If there are too many
- You don't have to convert the assembly into C, but converting it into pseudo-code is very helpful
- Try to write down pseudo-code when you meet a branch



```
if(var-4){
        do_something
}
else{
        exit
}
```

The first branch



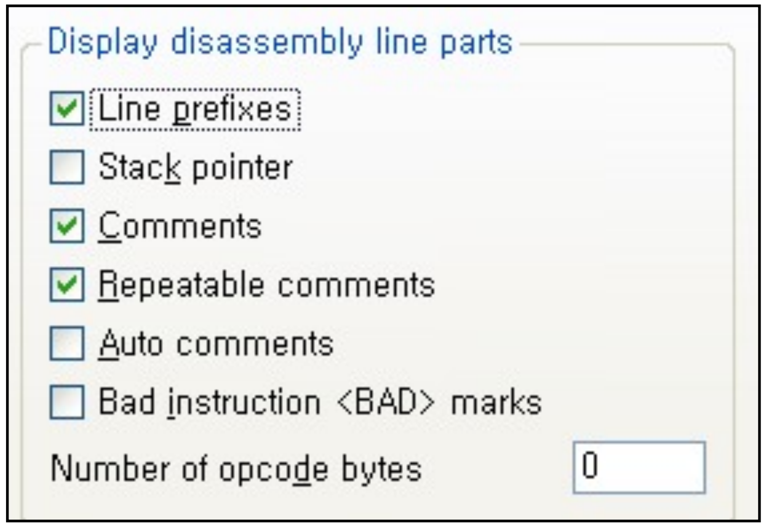
```
LDR     R3, [R11,#var_54]
CMP     R3, #2
BEQ     loc_B120
```

```
LDR     R3, [R11,#var_58]
LDR     R3, [R3]
LDR     R0, =aSUpdate_conf_f ; "%s update_conf_filename\n"
MOV     R1, R3
BL      printf
MOV     R0, #0          ; status
BL      exit
```

```
loc_B120
LDR     R3, [R11,#var_58]
ADD     R3, R3, #4
LDR     R3, [R3]
MOV     R0, R3          ; filename
LDR     R1, =aR         ; "r"
BL      fopen
```

To see addresses

- You need to change an option
  - Menu "Options - General"
  - Check "Line prefixes"
- Now you see addresses of instructions

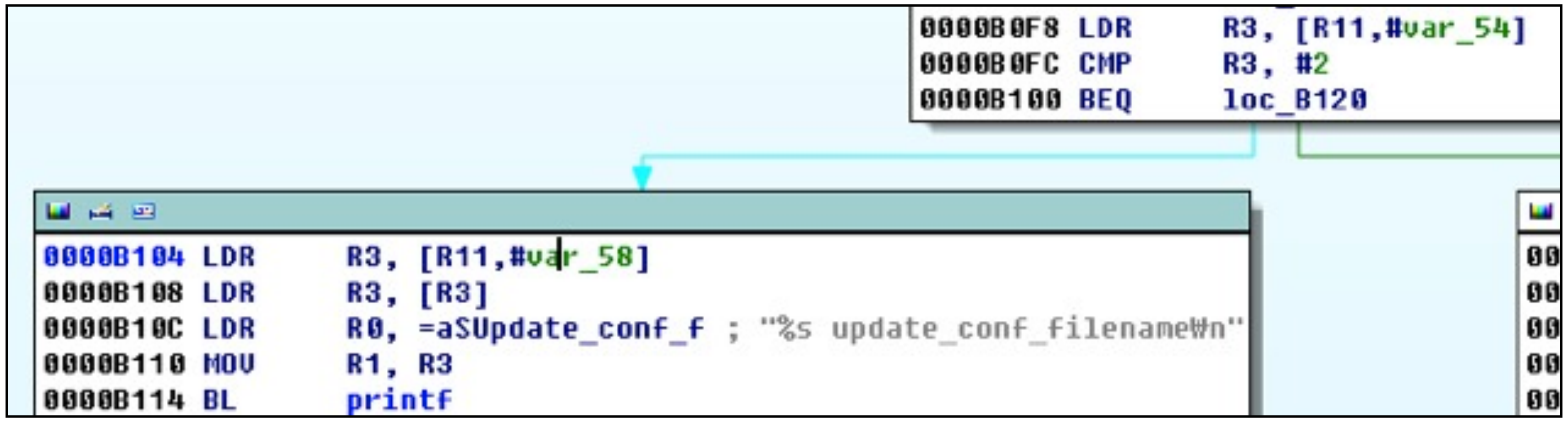


Display disassembly line parts

- [✓] Line prefixes
- [ ] Stack pointer
- [✓] Comments
- [✓] Repeatable comments
- [ ] Auto comments
- [ ] Bad instruction <BAD> marks

Number of opcode bytes 0

[General options]



```
0000B0F8 LDR     R3, [R11,#var_54]
0000B0FC CMP     R3, #2
0000B100 BEQ     loc_B120
```

```
0000B104 LDR     R3, [R11,#var_58]
0000B108 LDR     R3, [R3]
0000B10C LDR     R0, =aSUpdate_conf_f ; "%s update_conf_filename\n"
0000B110 MOV     R1, R3
0000B114 BL      printf
```

Let’s execute the program

- Let’s just execute the program
  - At least it’s free
- # ./installer

```
------------------------------------------------------------------
Welcome to the 'updater' challenge. This challenge is specially
created for SAMSUNG lecture. I hope all you will like this
one.

Cheers
/beist
------------------------------------------------------------------


./installer update_conf_filename
```

Oh?

Figuring out argc and argv

- main()'s definition is - *main(int argc, char *argv[])*
- It's somewhat clear that #var_54 is argc
- And #var_58 is a pointer to argv

if(argc==2)
    jump 0xb120

```
.text:0000B0F4                 BL      sub_B040
.text:0000B0F8                 LDR     R3, [R11,#var_54]
.text:0000B0FC                 CMP     R3, #2
.text:0000B100                 BEQ     loc_B120
.text:0000B104                 LDR     R3, [R11,#var_58]
.text:0000B108                 LDR     R3, [R3]
.text:0000B10C                 LDR     R0, =aSUpdate_conf_f ; "%s update_conf_filename\n"
.text:0000B110                 MOV     R1, R3
.text:0000B114                 BL      printf
.text:0000B118                 MOV     R0, #0          ; status
.text:0000B11C                 BL      exit
.text:0000B120 ; ---------------------------------------------------------------------------
.text:0000B120
.text:0000B120 loc_B120                                ; CODE XREF: sub_B0A4+5Cj
.text:0000B120                 LDR     R3, [R11,#var_58]
.text:0000B124                 ADD     R3, R3, #4
.text:0000B128                 LDR     R3, [R3]
.text:0000B12C                 MOV     R0, R3          ; filename
.text:0000B130                 LDR     R1, =aR         ; "r"
.text:0000B134                 BL      fopen
```

printf("%s ...", argv[1]);

Change variable names

- #var_54 = argc
- #var_58 = argv
- Only argc and argv, but much better than nothing!

The config file

- It’s now clear that the program tries to open argv[1] file

```
.text:0000B120                 LDR     R3, [R11,#argv]
.text:0000B124                 ADD     R3, R3, #4
.text:0000B128                 LDR     R3, [R3]
.text:0000B12C                 MOV     R0, R3          ; filename
.text:0000B130                 LDR     R1, =aR         ; "r"
.text:0000B134                 BL      fopen
```

fopen(argv[1],“r”);

sometimes you should enjoy “*beauty of space*”

File check

```
.text:0000B120                 LDR     R3, [R11,#argv]
.text:0000B124                 ADD     R3, R3, #4
.text:0000B128                 LDR     R3, [R3]
.text:0000B12C                 MOV     R0, R3          ; filename
.text:0000B130                 LDR     R1, =aR         ; "r"
.text:0000B134                 BL      fopen
.text:0000B138                 MOV     R3, R0
.text:0000B13C                 STR     R3, [R11,#var_20]
.text:0000B140                 LDR     R3, [R11,#var_20]
.text:0000B144                 CMP     R3, #0
.text:0000B148                 BNE     loc_B16C
.text:0000B14C                 LDR     R3, [R11,#argv]
.text:0000B150                 ADD     R3, R3, #4
.text:0000B154                 LDR     R3, [R3]
.text:0000B158                 LDR     R0, =aCheckOutSFile_ ; "Check out %s file.\n"
.text:0000B15C                 MOV     R1, R3
.text:0000B160                 BL      printf
.text:0000B164                 MOV     R0, #0          ; status
.text:0000B168                 BL      exit
```

- It's obvious that if the return value of fopen is null, it goes to exit()

The fist function

```
.text:0000B16C                 LDR     R0, [R11,#var_20]
.text:0000B170                 BL      sub_AD8C
```

#var_20 is the return value of *fopen()*. so, it is *sub_AD8C(fp);*

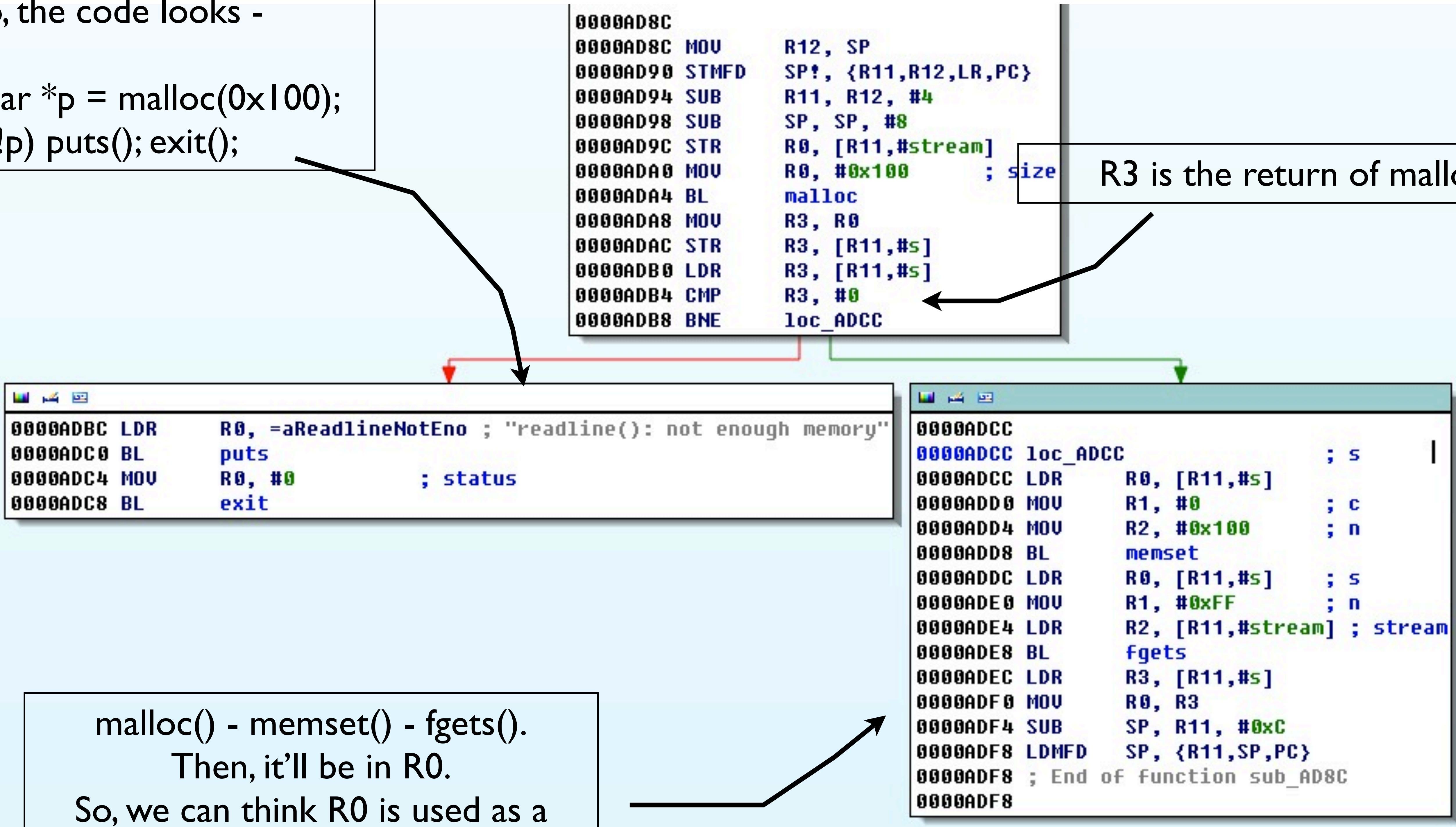
```
.text:0000AD8C                 MOV     R12, SP
.text:0000AD90                 STMFD   SP!, {R11,R12,LR,PC}
.text:0000AD94                 SUB     R11, R12, #4
.text:0000AD98                 SUB     SP, SP, #8
.text:0000AD9C                 STR     R0, [R11,#stream]
.text:0000ADA0                 MOV     R0, #0x100      ; size
.text:0000ADA4                 BL      malloc
.text:0000ADA8                 MOV     R3, R0
.text:0000ADAC                 STR     R3, [R11,#s]
.text:0000ADB0                 LDR     R3, [R11,#s]
.text:0000ADB4                 CMP     R3, #0
.text:0000ADB8                 BNE     loc_ADCC
.text:0000ADBC                 LDR     R0, =aReadlineNotEno ; "readline(): not enough memory"
.text:0000ADC0                 BL      puts
.text:0000ADC4                 MOV     R0, #0          ; status
.text:0000ADC8                 BL      exit
.text:0000ADCC ; ---------------------------------------------------------------------------
.text:0000ADCC
.text:0000ADCC loc_ADCC                                ; CODE XREF: sub_AD8C+2Cj
.text:0000ADCC                 LDR     R0, [R11,#s]    ; s
.text:0000ADD0                 MOV     R1, #0          ; c
.text:0000ADD4                 MOV     R2, #0x100      ; n
.text:0000ADD8                 BL      memset
.text:0000ADDC                 LDR     R0, [R11,#s]    ; s
.text:0000ADE0                 MOV     R1, #0xFF       ; n
.text:0000ADE4                 LDR     R2, [R11,#stream] ; stream
.text:0000ADE8                 BL      fgets
.text:0000ADEC                 LDR     R3, [R11,#s]
.text:0000ADF0                 MOV     R0, R3
.text:0000ADF4                 SUB     SP, R11, #0xC
.text:0000ADF8                 LDMFD   SP, {R11,SP,PC}
```

Function analysis

Very useful message.
So, the code looks -

char *p = malloc(0x100);
if(!p) puts(); exit();

R3 is the return of malloc()



```
0000AD8C
0000AD8C MOV     R12, SP
0000AD90 STMFD   SP!, {R11,R12,LR,PC}
0000AD94 SUB     R11, R12, #4
0000AD98 SUB     SP, SP, #8
0000AD9C STR     R0, [R11,#stream]
0000ADA0 MOV     R0, #0x100        ; size
0000ADA4 BL      malloc
0000ADA8 MOV     R3, R0
0000ADAC STR     R3, [R11,#s]
0000ADB0 LDR     R3, [R11,#s]
0000ADB4 CMP     R3, #0
0000ADB8 BNE     loc_ADCC
```

```
0000ADBC LDR     R0, =aReadlineNotEno ; "readline(): not enough memory"
0000ADC0 BL      puts
0000ADC4 MOV     R0, #0            ; status
0000ADC8 BL      exit
```

```
0000ADCC
0000ADCC loc_ADCC                          ; s
0000ADCC LDR     R0, [R11,#s]
0000ADD0 MOV     R1, #0                    ; c
0000ADD4 MOV     R2, #0x100                ; n
0000ADD8 BL      memset
0000ADDC LDR     R0, [R11,#s]              ; s
0000ADE0 MOV     R1, #0xFF                 ; n
0000ADE4 LDR     R2, [R11,#stream]         ; stream
0000ADE8 BL      fgets
0000ADEC LDR     R3, [R11,#s]
0000ADF0 MOV     R0, R3
0000ADF4 SUB     SP, R11, #0xC
0000ADF8 LDMFD   SP, {R11,SP,PC}
0000ADF8 ; End of function sub_AD8C
0000ADF8
```

malloc() - memset() - fgets().
Then, it'll be in R0.
So, we can think R0 is used as a return value for the caller

Naming is hard

- So, sub_AD8C()'s pseudo code is like

```
char *sub_AD8C(FILE *fp) {
    char *p;
    p = malloc(0x100);
    if(!p) {
        printf("readline(): not enough memory\n");
        exit(0);
    }
    memset(p, 0, 0x100);
    fgets(p, 0xff, fp);
    return p;
}
```

- Now we can give it a name
- Actually, it says "readline()" already
- Change the function name!

sub_AF48() analysis

```
.text:0000B16C                  LDR     R0, [R11,#var_20]
.text:0000B170                  BL      readline
.text:0000B174                  MOV     R3, R0
.text:0000B178                  STR     R3, [R11,#command]
.text:0000B17C                  LDR     R0, [R11,#command]
.text:0000B180                  BL      sub_AF48
```

- Ok - *sub_AF48(buffer)*;

```
.text:0000AF48                  MOV     R12, SP
.text:0000AF4C                  STMFD   SP!, {R11,R12,LR,PC}
.text:0000AF50                  SUB     R11, R12, #4
.text:0000AF54                  SUB     SP, SP, #4
.text:0000AF58                  STR     R0, [R11,#s]
.text:0000AF5C                  LDR     R0, [R11,#s]    ; s
.text:0000AF60                  BL      strlen
.text:0000AF64                  MOV     R3, R0
.text:0000AF68                  SUB     R2, R3, #1
.text:0000AF6C                  LDR     R3, [R11,#s]
.text:0000AF70                  ADD     R2, R3, R2
.text:0000AF74                  MOV     R3, #0
.text:0000AF78                  STRB    R3, [R2]
.text:0000AF7C                  LDMFD   SP, {R3,R11,SP,PC}
```

```
r3=strlen(buffer);
r2=r3 - 1;
buffer[r2]=0x0;
```

sub_AF48() analysis

- The function sets 0x0 to [the length - 1] of buffer
- Remember that the buffer is from fgets()
- Therefore, there might be newline or something
- Looks sub_AF48 wants to delete the newline
- We name it - *delete_linefeed()*

sub_AEF4() analysis

- Before looking at *sub_AEF4()*, see the *puts()* message

```
.text:0000B184                 LDR     R0, [R11,#command]
.text:0000B188                 BL      sub_AEF4
.text:0000B18C                 MOV     R3, R0
.text:0000B190                 STR     R3, [R11,#var_14]
.text:0000B194                 LDR     R3, [R11,#var_14]
.text:0000B198                 CMP     R3, #0
.text:0000B19C                 BNE     loc_B1B0
.text:0000B1A0                 LDR     R0, =aTheMagicIsNotM ; "The magic is not matched."
.text:0000B1A4                 BL      puts
.text:0000B1A8                 MOV     R0, #0          ; status
.text:0000B1AC                 BL      exit
```

- "*The magic is not matched.*"

- Very kind, so, we can guess sub_AEF4() is something that checks a magic value

sub_AEF4() analysis

```
.text:0000AEF4                 MOV     R12, SP
.text:0000AEF8                 STMFD   SP!, {R11,R12,LR,PC}
.text:0000AEFC                 SUB     R11, R12, #4
.text:0000AF00                 SUB     SP, SP, #8
.text:0000AF04                 STR     R0, [R11,#s2]
.text:0000AF08                 LDR     R0, =aVery_nice_frid ; "VERY_NICE_FRIDAY"
.text:0000AF0C                 LDR     R1, [R11,#s2]   ; s2
.text:0000AF10                 BL      strcmp
.text:0000AF14                 MOV     R3, R0
.text:0000AF18                 CMP     R3, #0
.text:0000AF1C                 BNE     loc_AF2C
.text:0000AF20                 MOV     R3, #1
.text:0000AF24                 STR     R3, [R11,#var_14]
.text:0000AF28                 B       loc_AF34
.text:0000AF2C ; ---------------------------------------------------------------------------
.text:0000AF2C
.text:0000AF2C loc_AF2C                                ; CODE XREF: sub_AEF4+28j
.text:0000AF2C                 MOV     R3, #0
.text:0000AF30                 STR     R3, [R11,#var_14]
.text:0000AF34
.text:0000AF34 loc_AF34                                ; CODE XREF: sub_AEF4+34j
.text:0000AF34                 LDR     R3, [R11,#var_14]
.text:0000AF38                 MOV     R0, R3
.text:0000AF3C                 SUB     SP, R11, #0xC
.text:0000AF40                 LDMFD   SP, {R11,SP,PC}
```

Again, R0 is used as a return value

sub_AEF4() analysis

- The function is straightforward
- The pseudo code would be

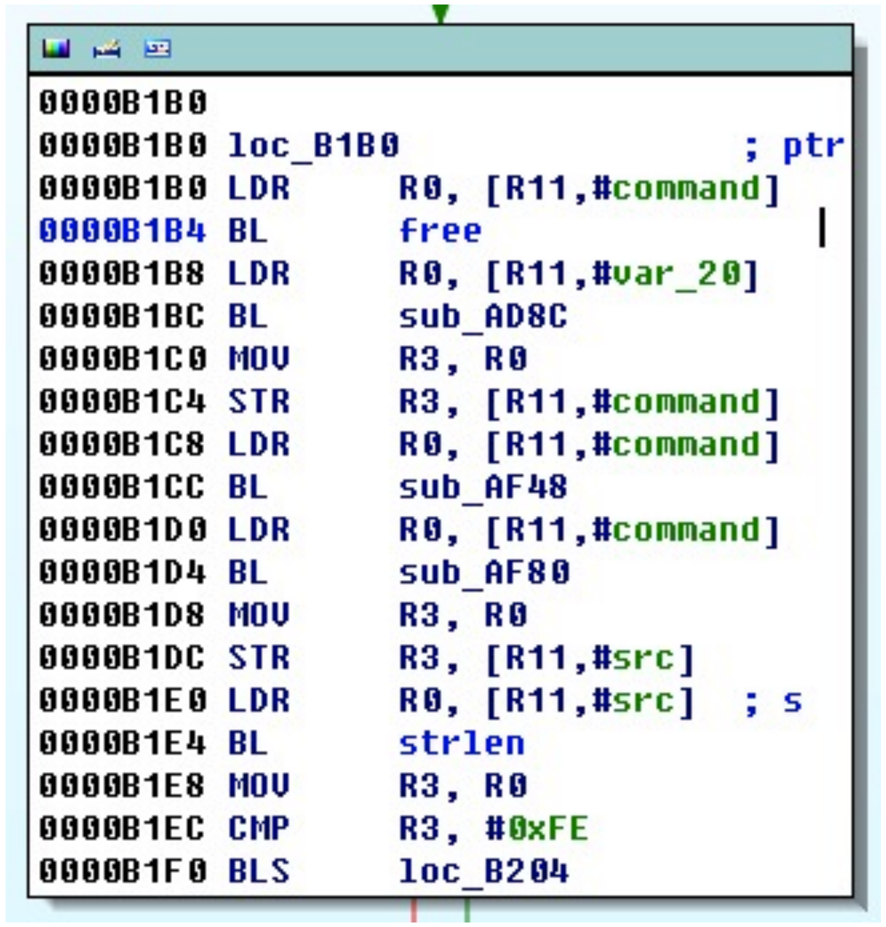
```
if(!strcmp("VERY_NICE_FRIDAY", buffer))
    return 1;
else
    return 0;
```

- Name it - *magic_check()*

So far

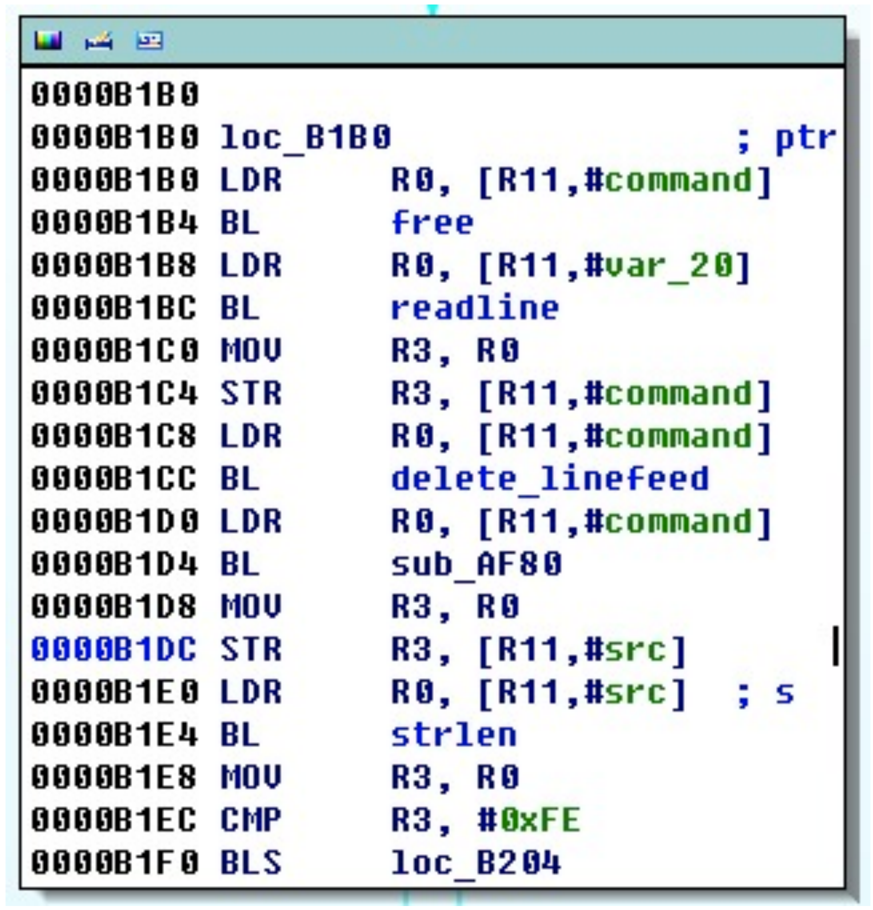
- We have analyzed
  - It shows banner
  - The program opens *argv[1]* file
  - Check the file is available
  - *readline()* and *delete_linefeed()*
  - and *magic_check()*

We’ve cleared a lot!



```
0000B1B0
0000B1B0 loc_B1B0                  ; ptr
0000B1B0 LDR      R0, [R11,#command]
0000B1B4 BL       free
0000B1B8 LDR      R0, [R11,#var_20]
0000B1BC BL       sub_AD8C
0000B1C0 MOV      R3, R0
0000B1C4 STR      R3, [R11,#command]
0000B1C8 LDR      R0, [R11,#command]
0000B1CC BL       sub_AF48
0000B1D0 LDR      R0, [R11,#command]
0000B1D4 BL       sub_AF80
0000B1D8 MOV      R3, R0
0000B1DC STR      R3, [R11,#src]
0000B1E0 LDR      R0, [R11,#src]  ; s
0000B1E4 BL       strlen
0000B1E8 MOV      R3, R0
0000B1EC CMP      R3, #0xFE
0000B1F0 BLS      loc_B204
```

[before]

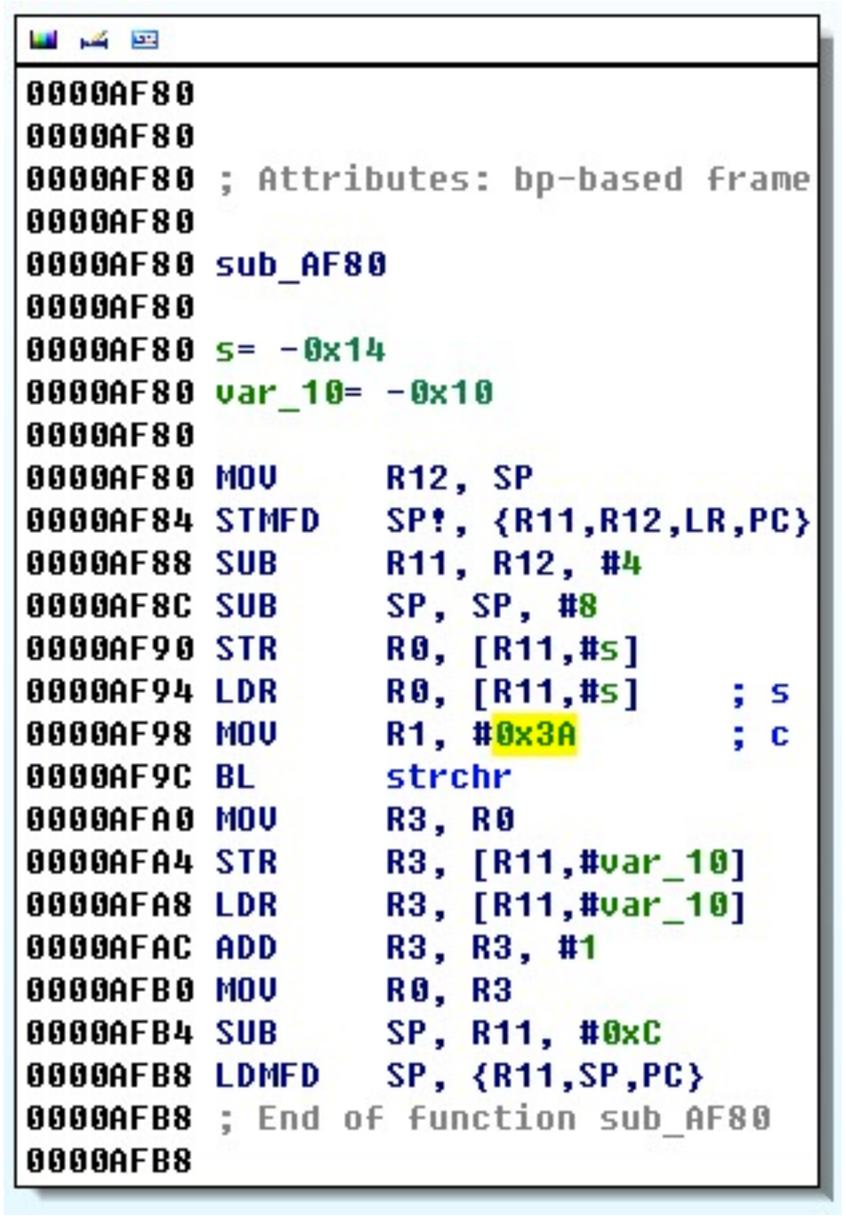


```
0000B1B0
0000B1B0 loc_B1B0                  ; ptr
0000B1B0 LDR      R0, [R11,#command]
0000B1B4 BL       free
0000B1B8 LDR      R0, [R11,#var_20]
0000B1BC BL       readline
0000B1C0 MOV      R3, R0
0000B1C4 STR      R3, [R11,#command]
0000B1C8 LDR      R0, [R11,#command]
0000B1CC BL       delete_linefeed
0000B1D0 LDR      R0, [R11,#command]
0000B1D4 BL       sub_AF80
0000B1D8 MOV      R3, R0
0000B1DC STR      R3, [R11,#src]
0000B1E0 LDR      R0, [R11,#src]  ; s
0000B1E4 BL       strlen
0000B1E8 MOV      R3, R0
0000B1EC CMP      R3, #0xFE
0000B1F0 BLS      loc_B204
```

[after]

sub_AF80() analysis

- It also takes a buffer as an argument
- Then, call *strchr()*
- *strchr(buffer, 0x3A)*
- Click "*#0x3A*" and type "*r*"
- You'll see ":"



```
0000AF80
0000AF80
0000AF80 ; Attributes: bp-based frame
0000AF80
0000AF80 sub_AF80
0000AF80
0000AF80 s= -0x14
0000AF80 var_10= -0x10
0000AF80
0000AF80 MOV     R12, SP
0000AF84 STMFD   SP!, {R11,R12,LR,PC}
0000AF88 SUB     R11, R12, #4
0000AF8C SUB     SP, SP, #8
0000AF90 STR     R0, [R11,#s]
0000AF94 LDR     R0, [R11,#s]     ; s
0000AF98 MOV     R1, #0x3A        ; c
0000AF9C BL      strchr
0000AFA0 MOV     R3, R0
0000AFA4 STR     R3, [R11,#var_10]
0000AFA8 LDR     R3, [R11,#var_10]
0000AFAC ADD     R3, R3, #1
0000AFB0 MOV     R0, R3
0000AFB4 SUB     SP, R11, #0xC
0000AFB8 LDMFD   SP, {R11,SP,PC}
0000AFB8 ; End of function sub_AF80
0000AFB8
```

sub_AF80() analysis

- It simply finds ":" character in the buffer
- And return the pointer to right after ":"
- We name it as - *get_after_colon()*

After get_afer_colon()

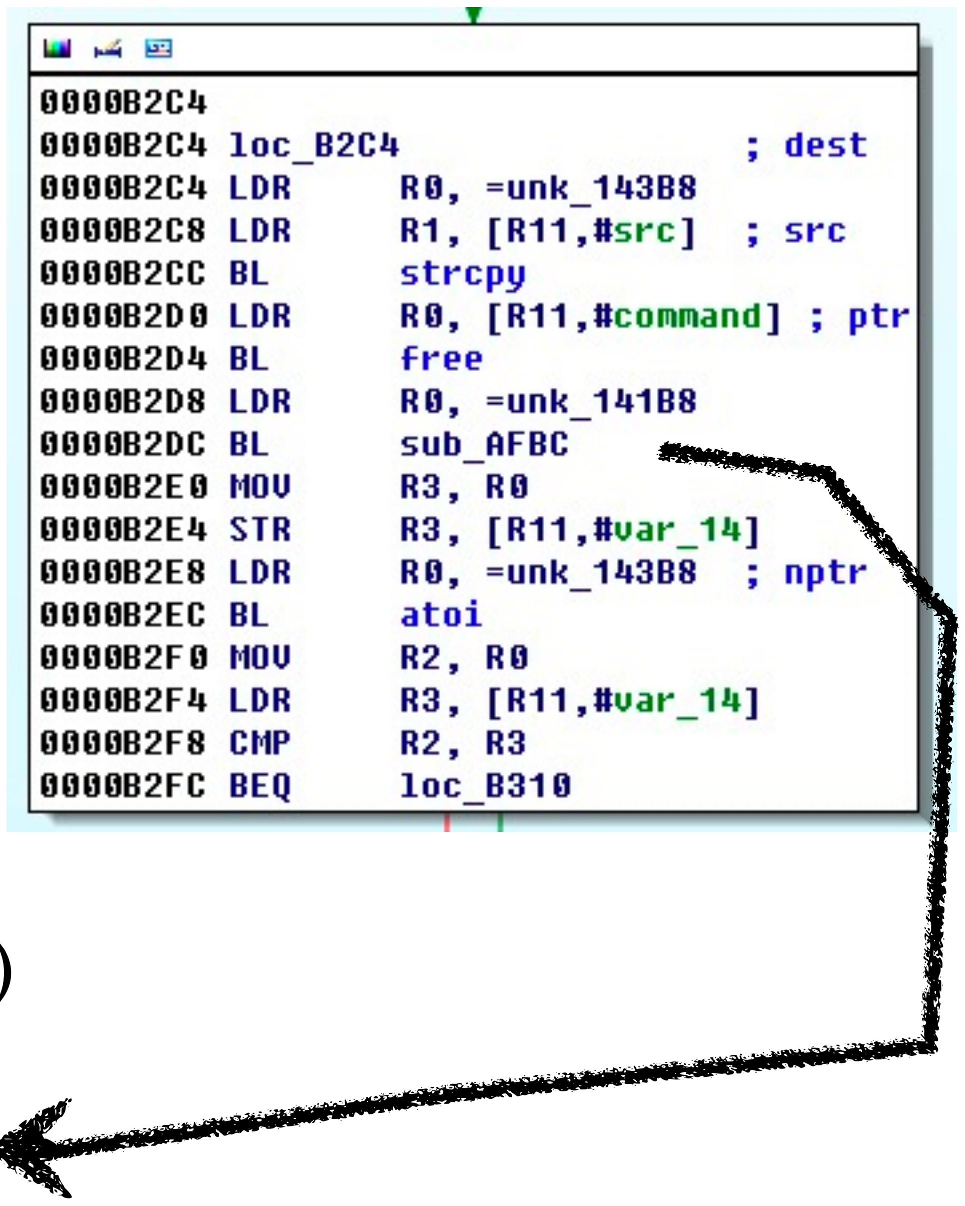
check if it’s lesser than 0xfe,
this is probably for buffer-overflow-check

```
.text:0000B1D4                 BL      get_after_colon
.text:0000B1D8                 MOV     R3, R0
.text:0000B1DC                 STR     R3, [R11,#src]
.text:0000B1E0                 LDR     R0, [R11,#src]  ; s
.text:0000B1E4                 BL      strlen
.text:0000B1E8                 MOV     R3, R0
.text:0000B1EC                 CMP     R3, #0xFE
.text:0000B1F0                 BLS     loc_B204
.text:0000B1F4                 LDR     R0, =aFilenameValueI ; "Filename value is too long."
.text:0000B1F8                 BL      puts
.text:0000B1FC                 MOV     R0, #0          ; status
.text:0000B200                 BL      exit
.text:0000B204 ; ---------------------------------------------------------------------------
.text:0000B204
.text:0000B204 loc_B204                                ; CODE XREF: sub_B0A4+14Cj
.text:0000B204                 LDR     R0, =unk_141B8  ; dest
.text:0000B208                 LDR     R1, [R11,#src]  ; src
.text:0000B20C                 BL      strcpy
```

then, call strcpy(), but dest is *unk_141B8*?

So far

- We’ve figured out
  - VERY_NICE_FRIDAY
  - filename:FILENAME
  - date:DATE
  - size:FILESIZE
    (*date* and *size* are the same with *filename*)
- And sub_AFBC() is coming



```
0000B2C4
0000B2C4 loc_B2C4                 ; dest
0000B2C4 LDR     R0, =unk_143B8
0000B2C8 LDR     R1, [R11,#src]   ; src
0000B2CC BL      strcpy
0000B2D0 LDR     R0, [R11,#command] ; ptr
0000B2D4 BL      free
0000B2D8 LDR     R0, =unk_141B8
0000B2DC BL      sub_AFBC
0000B2E0 MOV     R3, R0
0000B2E4 STR     R3, [R11,#var_14]
0000B2E8 LDR     R0, =unk_143B8   ; nptr
0000B2EC BL      atoi
0000B2F0 MOV     R2, R0
0000B2F4 LDR     R3, [R11,#var_14]
0000B2F8 CMP     R2, R3
0000B2FC BEQ     loc_B310
```

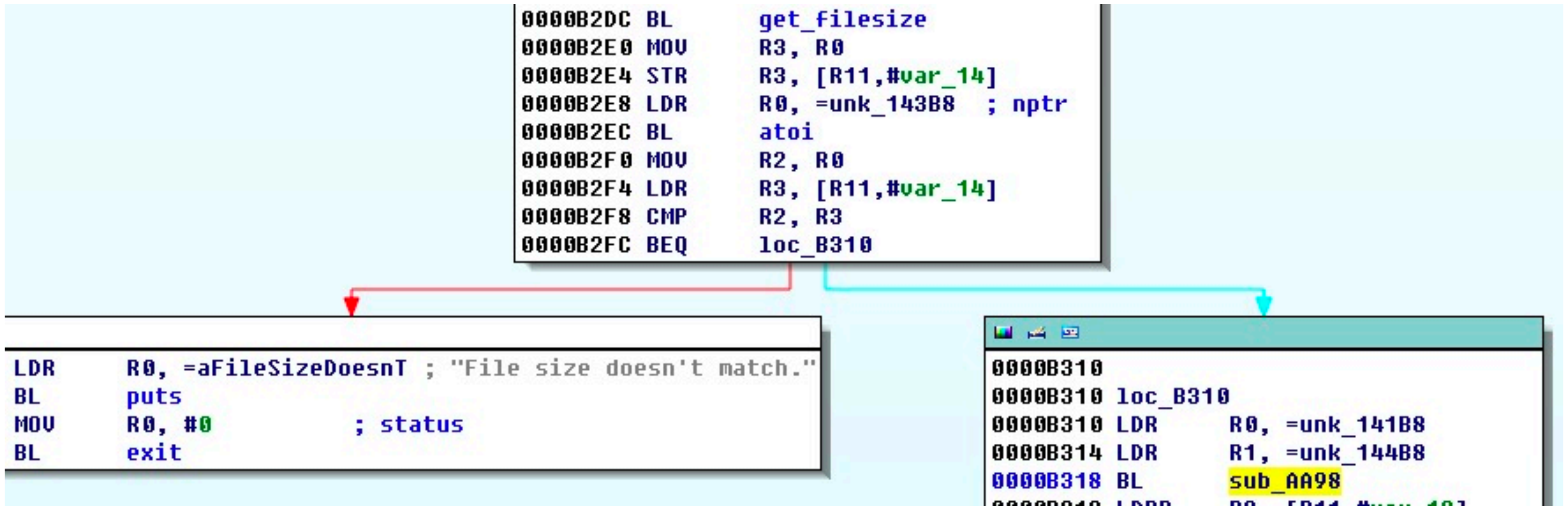
sub_AFBC() analysis

```
.text:0000AFBC                 MOV     R12, SP
.text:0000AFC0                 STMFD   SP!, {R11,R12,LR,PC}
.text:0000AFC4                 SUB     R11, R12, #4
.text:0000AFC8                 SUB     SP, SP, #0xC
.text:0000AFCC                 STR     R0, [R11,#filename]
.text:0000AFD0                 LDR     R0, [R11,#filename] ; filename
.text:0000AFD4                 LDR     R1, =aRb        ; "rb"
.text:0000AFD8                 BL      fopen
.text:0000AFDC                 MOV     R3, R0
.text:0000AFE0                 STR     R3, [R11,#stream]
.text:0000AFE4                 LDR     R3, [R11,#stream]
.text:0000AFE8                 CMP     R3, #0
.text:0000AFEC                 BNE     loc_B000
.text:0000AFF0                 LDR     R0, =aFileIsNotAvail ; "File is not available."
.text:0000AFF4                 BL      puts
.text:0000AFF8                 MOV     R0, #0          ; status
.text:0000AFFC                 BL      exit
.text:0000B000 ; ---------------------------------------------------------------------------
.text:0000B000
.text:0000B000 loc_B000                                ; CODE XREF: sub_AFBC+30j
.text:0000B000                 LDR     R0, [R11,#stream] ; stream
.text:0000B004                 MOV     R1, #0          ; off
.text:0000B008                 MOV     R2, #2          ; whence
.text:0000B00C                 BL      fseek
.text:0000B010                 LDR     R0, [R11,#stream] ; stream
.text:0000B014                 BL      ftell
.text:0000B018                 MOV     R3, R0
.text:0000B01C                 STR     R3, [R11,#var_10]
.text:0000B020                 LDR     R0, [R11,#stream] ; stream
.text:0000B024                 BL      fclose
.text:0000B028                 LDR     R3, [R11,#var_10]
.text:0000B02C                 MOV     R0, R3
.text:0000B030                 SUB     SP, R11, #0xC
.text:0000B034                 LDMFD   SP, {R11,SP,PC}
```

sub_AFBC() analysis

- *fopen()* - *fseek()* - *ftell()* - *fclose()*
- *fseek*(fp, 0, SEEK_END);
  - R2 = #2
  - If argument 2 is #2, it means SEEK_END
- It's a pattern for getting a file size of a given file
- We name it - *get_filesize()*

sub_AA98()


```
0000B2DC BL      get_filesize
0000B2E0 MOV     R3, R0
0000B2E4 STR     R3, [R11,#var_14]
0000B2E8 LDR     R0, =unk_143B8  ; nptr
0000B2EC BL      atoi
0000B2F0 MOV     R2, R0
0000B2F4 LDR     R3, [R11,#var_14]
0000B2F8 CMP     R2, R3
0000B2FC BEQ     loc_B310
```

```
LDR     R0, =aFileSizeDoesnT ; "File size doesn't match."
BL      puts
MOV     R0, #0          ; status
BL      exit
```

```
0000B310
0000B310 loc_B310
0000B310 LDR     R0, =unk_141B8
0000B314 LDR     R1, =unk_144B8
0000B318 BL      sub_AA98
```

- If we bypassed this trap, we now go for sub_AA98()

sub_AA98() seems so hard

- It is not like the other functions
  - Very complicated
  - And call other functions inside
- Let's skip now and see below messages

```
.text:0000B384                 LDR     R0, =aHashIsTooLong_ ; "hash is too long."
.text:0000B388                 BL      puts
.text:0000B38C                 MOV     R0, #0          ; status
.text:0000B390
```

We can picture that sub_AA98()
might be related to hash function in any way

sub_AA98() is hard

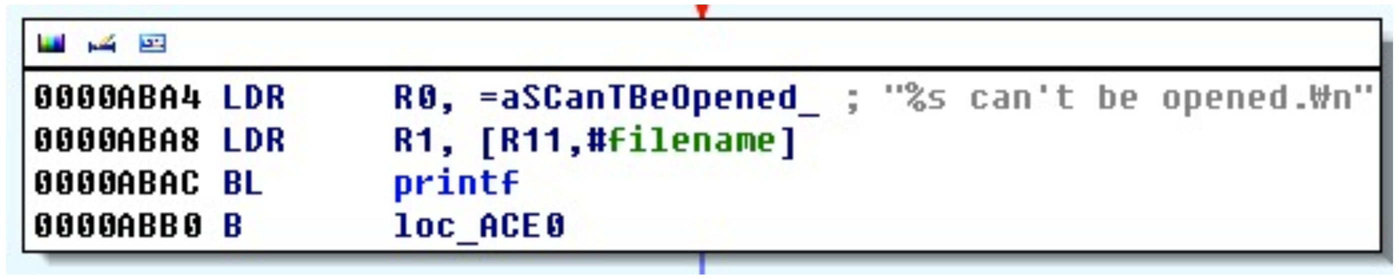
- Ok, AA98() looks a hash function
- If so, it's definitely not easy for hackers to analyze as well
- Usually, hackers try to match them to patterns
  - Example) Flirt
- If it is using a known algorithm, there might be an easy to figure out

Strategy

- Assume that there is a function (algorithm) which is known
- But the function name is stripped
- We don’t want to diff the function with libraries
- Pro tip:
  - Known algorithms are well coded usually
  - It prints out messages for some situations
  - Example) error messages

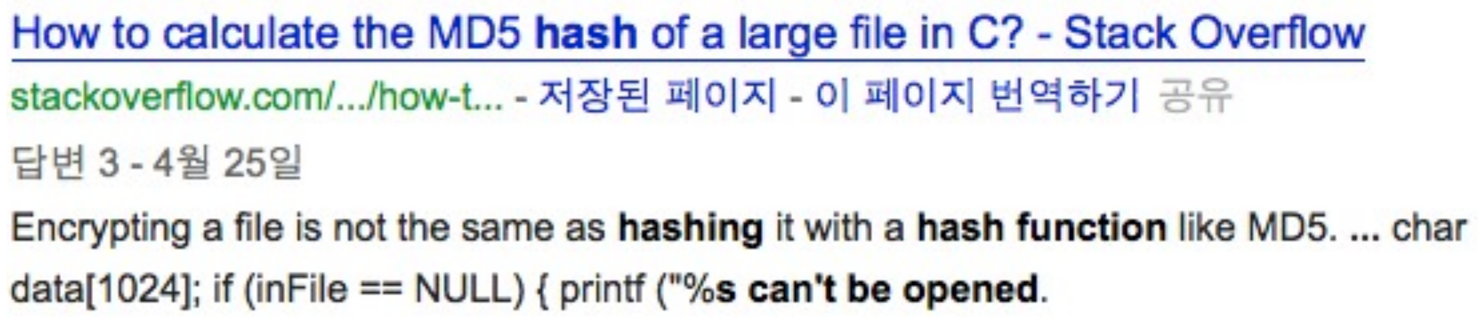
Find messages

- We can find below basic block in sub_AA98()



```
0000ABA4 LDR     R0, =aSCanTBeOpened_ ; "%s can't be opened.\n"
0000ABA8 LDR     R1, [R11,#filename]
0000ABAC BL      printf
0000ABB0 B       loc_ACE0
```

- And search on google
  - "%s can't be opened." hash algorithm



How to calculate the MD5 **hash** of a large file in C? - Stack Overflow
stackoverflow.com/.../how-t... - 저장된 페이지 - 이 페이지 번역하기 공유
답변 3 - 4월 25일
Encrypting a file is not the same as **hashing** it with a **hash function** like MD5. ... char data[1024]; if (inFile == NULL) { printf ("%**s can't be opened**.

Easy solution == win

- We can be sure it uses MD5 libary
  - http://www.phrack.org/issues.html?id=11&issue=55
- As *sub_AA98()* opens a file and generate a hash for that, it's *MDFile()*

So far

- According to messages, the programs calls *strcpy()*
    - filename
    - date
    - size
- Then, call *sub_AA98()* with filename and buffer
- So, it gets MD5 hash of a given file

Drawing is fun

- Art time! Drawing is fun as usual
- Take a break and try to draw the flow of the program roughly
- If a target program is big, it is hard to follow up

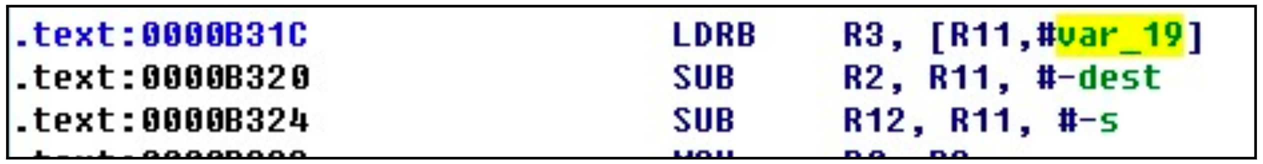
sub_AE00() is not easy

- Arguments are 3 for sub_AE00()

```
.text:0000B31C                 LDRB    R3, [R11,#var_19]
.text:0000B320                 SUB     R2, R11, #-dest
.text:0000B324                 SUB     R12, R11, #-s
.text:0000B328                 MOV     R0, R3
.text:0000B32C                 MOV     R1, R2
.text:0000B330                 MOV     R2, R12
.text:0000B334                 BL      sub_AE00
```

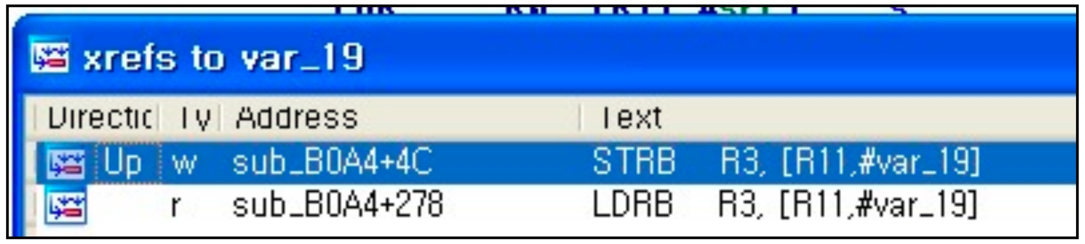
sub_AE00(): xrefs

- R3 is moved into R0
- Where #*var_19* comes from?



```
.text:0000B31C                 LDRB    R3, [R11,#var_19]
.text:0000B320                 SUB     R2, R11, #-dest
.text:0000B324                 SUB     R12, R11, #-s
```

- Mouse on #var_19 and type 'x' key
- You'll see xrefs for #*var_19*



xrefs to var_19

Directic	Ty	Address	Text
Up	w	sub_B0A4+4C	STRB R3, [R11,#var_19]
	r	sub_B0A4+278	LDRB R3, [R11,#var_19]

sub_AE00(): argument 1

```
.text:0000B0EC                    MOV     R3, #0x78
.text:0000B0F0                    STRB    R3, [R11,#var_19]
```

- We see *#var_19* is initialized as 0x78 (alphabet ‘*x*’)
- The first argument is a character and ‘*x*’

sub_AE00(): argument 2

```
.text:0000B0BC                 LDR     R3, =unk_BB6C
.text:0000B0C0                 SUB     R2, R11, #-dest
.text:0000B0C4                 MOV     R12, #0xD
.text:0000B0C8                 MOV     R0, R2          ; dest
.text:0000B0CC                 MOV     R1, R3          ; src
.text:0000B0D0                 MOV     R2, R12         ; n
.text:0000B0D4                 BL      memcpy
```

buffer in *unk_BB6C* is copied into *dest*

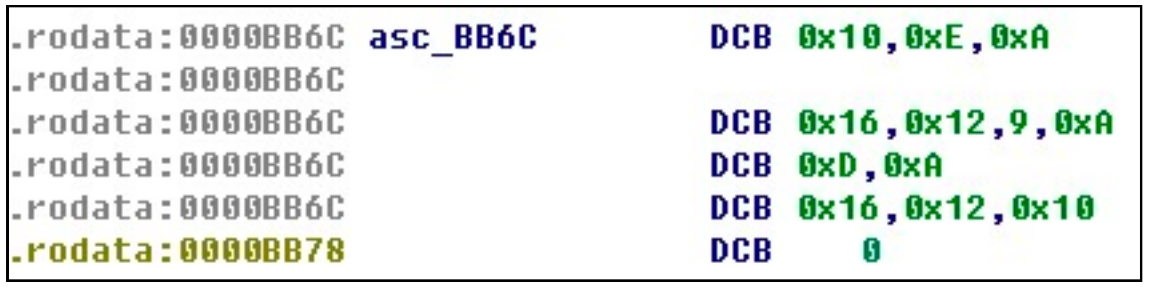
```
.text:0000B31C                 LDRB    R3, [R11,#var_19]
.text:0000B320                 SUB     R2, R11, #-dest
.text:0000B324                 SUB     R12, R11, #-s
.text:0000B328                 MOV     R0, R3
.text:0000B32C                 MOV     R1, R2
.text:0000B330                 MOV     R2, R12
.text:0000B334                 BL      sub_AE00        ;
```

dest is used as argument 2

```
.rodata:0000BB6C asc_BB6C        DCB 0x10,0xE,0xA
.rodata:0000BB6C
.rodata:0000BB6C                 DCB 0x16,0x12,9,0xA
.rodata:0000BB6C                 DCB 0xD,0xA
.rodata:0000BB6C                 DCB 0x16,0x12,0x10
.rodata:0000BB78                 DCB    0
```

sub_AE00(): argument 2, 3

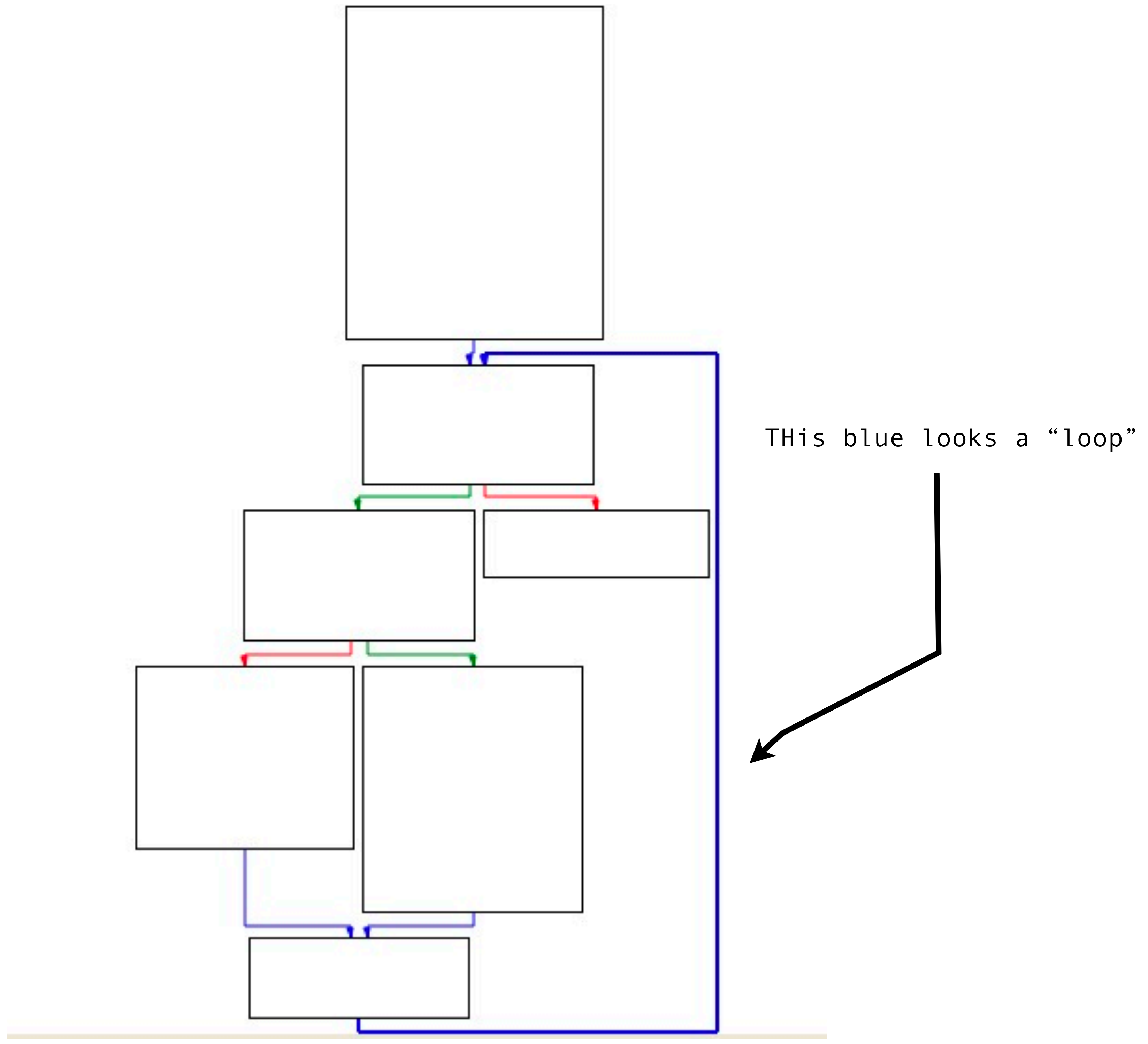
- The string at 0xBB6C looks definitely not readable
  - 0x10, 0xe, 0xa, 0x16, 0x12, 0x9 ... ...
- We smell



```
.rodata:0000BB6C asc_BB6C        DCB 0x10,0xE,0xA
.rodata:0000BB6C
.rodata:0000BB6C                 DCB 0x16,0x12,9,0xA
.rodata:0000BB6C                 DCB 0xD,0xA
.rodata:0000BB6C                 DCB 0x16,0x12,0x10
.rodata:0000BB78                 DCB    0
```

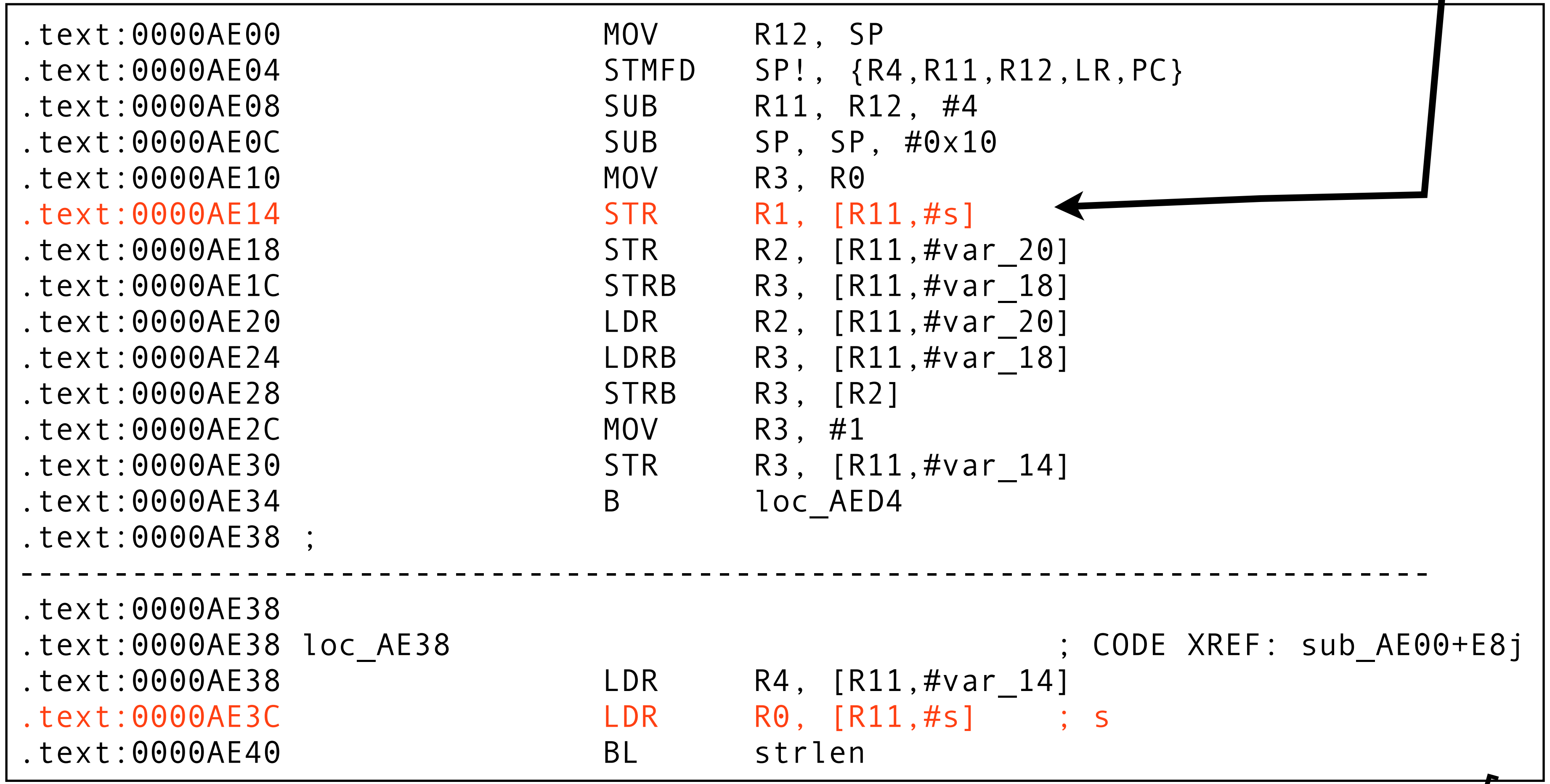
- And argument 3 is just a local buffer
- sub_AE00(a_character, non_readable_string, local_buffer);
- Ok..

sub_AE00(): The graphs

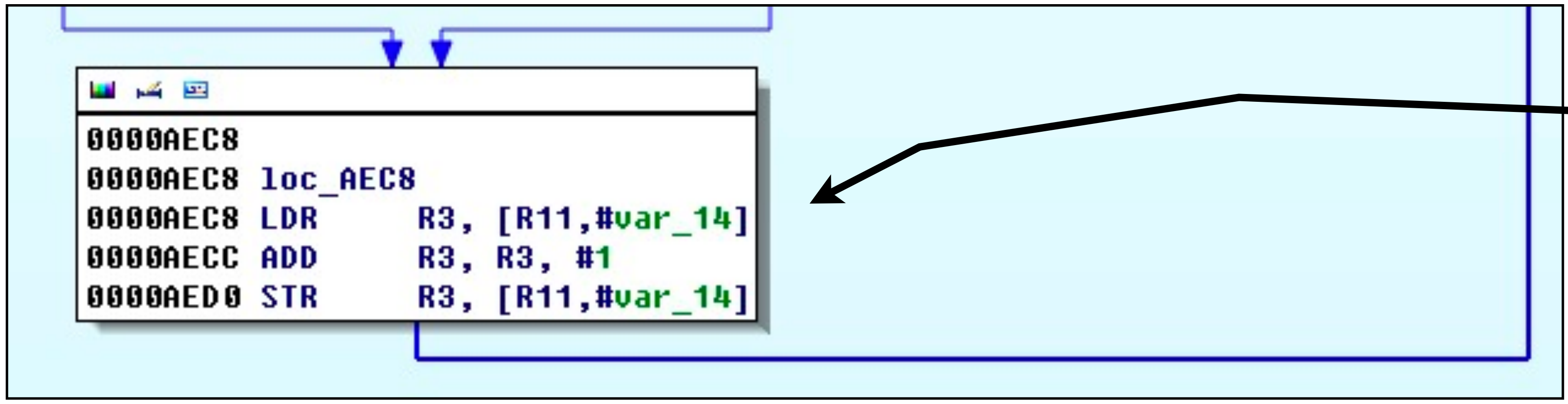


sub_AE00(): loop

[The unreadable string]



```
.text:0000AE00                 MOV     R12, SP
.text:0000AE04                 STMFD   SP!, {R4,R11,R12,LR,PC}
.text:0000AE08                 SUB     R11, R12, #4
.text:0000AE0C                 SUB     SP, SP, #0x10
.text:0000AE10                 MOV     R3, R0
.text:0000AE14                 STR     R1, [R11,#s]
.text:0000AE18                 STR     R2, [R11,#var_20]
.text:0000AE1C                 STRB    R3, [R11,#var_18]
.text:0000AE20                 LDR     R2, [R11,#var_20]
.text:0000AE24                 LDRB    R3, [R11,#var_18]
.text:0000AE28                 STRB    R3, [R2]
.text:0000AE2C                 MOV     R3, #1
.text:0000AE30                 STR     R3, [R11,#var_14]
.text:0000AE34                 B       loc_AED4
.text:0000AE38 ;
---------------------------------------------------------------------------
.text:0000AE38
.text:0000AE38 loc_AE38                                ; CODE XREF: sub_AE00+E8j
.text:0000AE38                 LDR     R4, [R11,#var_14]
.text:0000AE3C                 LDR     R0, [R11,#s]    ; s
.text:0000AE40                 BL      strlen
```



```
0000AEC8
0000AEC8 loc_AEC8
0000AEC8 LDR     R3, [R11,#var_14]
0000AECC ADD     R3, R3, #1
0000AED0 STR     R3, [R11,#var_14]
```

[probably #var_14 is a count for the loop]

sub_AE00(): In the loop

[Not sure yet, but we guess R0 will be increased every loop]

```
.text:0000AE88 loc_AE88                                  ; CODE XREF: sub_AE00+50j
.text:0000AE88                LDR     R2, [R11,#var_14]
.text:0000AE8C                LDR     R3, [R11,#var_20]
.text:0000AE90                ADD     R0, R3, R2
.text:0000AE94                LDR     R3, [R11,#var_14]
.text:0000AE98                SUB     R2, R3, #1
.text:0000AE9C                LDR     R3, [R11,#s]
.text:0000AEA0                ADD     R3, R3, R2
.text:0000AEA4                LDRB    R1, [R3]
.text:0000AEA8                LDR     R3, [R11,#var_14]
.text:0000AEAC                SUB     R2, R3, #1
.text:0000AEB0                LDR     R3, [R11,#var_20]
.text:0000AEB4                ADD     R3, R3, R2
.text:0000AEB8                LDRB    R3, [R3]
.text:0000AEBC                EOR     R3, R1, R3
.text:0000AEC0                AND     R3, R3, #0xFF
.text:0000AEC4                STRB    R3, [R0]
```

[Store 1 byte into buffer]

[Exclusive OR]

sub_AE00() is not easy

- It would take time to figure out of *sub_AE00()* if you're not familiar with ARM assembly
- So, we give you the encryption routine in C, but not decryption routine
- This routine will help you analyze *sub_AE00()*

```
void go_enc(char *p_password, char *e_password) {
      int i;
      for(i=0;i<strlen(p_password);i++) {
            if(i == (strlen(p_password) - 1))
                  e_password[i] = p_password[i] ^ p_password[0];
            else
                  e_password[i] = p_password[i] ^ p_password[i+1];

      }
}
```

[encryption function]

decryption routine

```
void go_dec(char key, char *e_password, char *p_password) {
      int i;
      p_password[0]=key;
      for(i=1;i<strlen(e_password);i++) {
            if(i == (strlen(e_password) - 1))
                  p_password[i] = e_password[i] ^ p_password[0];
            else
                  p_password[i] = e_password[i-1] ^ p_password[i-1];

      }
}
```

source code

```
#include <stdio.h>
#include <stdlib.h>
#include "md5.h"
#define DEBUG 0

/*

1. form for update.conf (example)
VERY_NICE_FRIDAY
filename:bin_elf
date:20120910
size:31337
hash:xxxx

2. hash value form
x = md5(A+B+C+D)
y = md5(x)
X = x+y
E = value of 'hash'

3. A+B+C+D
A = date
B = size
C = md5 of 'filename'
D = KEY

*/
```

```
struct update_info {
        char filename[256];
        char date[256];
        char size[256];
        char md5[256];
        char KEY[256];
        char hash[256];
} gogo;

char *readline(FILE *fp) {
        char *p;
        p = malloc(256);
        if(!p) {
                printf("readline(): not enough memory\n");
                exit(0);
        }
        memset(p, 0, 256);
        fgets(p, 255, fp);
        return p;
}
```

source code

```
void go_dec(char key, char *e_password, char *p_password) {
        int i;
        p_password[0]=key;
        for(i=1;i<strlen(e_password);i++) {
                if(i == (strlen(e_password) - 1))
                        p_password[i] = e_password[i] ^ p_password[0];
                else
                        p_password[i] = e_password[i-1] ^ p_password[i-1];

        }
}
```

```
/*
  [how this program works?]

  1. open update.conf file
  2. check the magic number (VERY_NICE_FRIDAY)
  3. A = get date
  4. B = get size and check if the size if correct
  5. C = get md5("bin_elf")
  6. D = get KEY
  7. X = hash(A+B+C+D)
  8. E = get hash
  9. compare X and E
  10. if not, exit
  11. if so, execute "bin_elf"

*/
```

source code

```
int check_magic(char *str) {
        if(!strcmp("VERY_NICE_FRIDAY", str)) {
                if(DEBUG) {
                        printf("check_magic(): The magic is matched.\n");
                }
                return 1;
        }
        else {
                return 0;
        }
}

void delete_newline(char *str) {
        str[strlen(str)-1]='\0';
}

char *get_next(char *str) {
        char *d;
        d = strstr(str, ":");
        if(!d) {
                if(DEBUG) {
                        printf("get_next(): Can't find :\n");
                        exit(0);
                }
        }

        return d+1;
}
```

source code

```c
int get_file_size(char *filename) {

        FILE *fp;
        int len;
        fp = fopen(filename, "rb");
        if(fp == NULL) {
                printf("File is not available.\n");
                exit(0);
        }
        fseek(fp, 0, SEEK_END);
        len = ftell(fp);
        fclose(fp);
        return len;
}

void show_banner() {
        printf("---------------------------------------------------------------\n");
        printf("Welcome to the 'updater' challenge. This challenge is specially\n");
        printf("created for SAMSUNG lecture. I hope all you will like this one.\n");
        printf("\n");
        printf("Cheers\n");
        printf("/beist\n");
        printf("---------------------------------------------------------------\n");
        printf("\n");
}
```

source code

```
int main(int argc, char *argv[]) {
        char *p;
        char *tmp_p;
        char *final;
        FILE *fp;

        // plain key is "xhflzhakflzh"
        char encrypted_key[]="\x10\x0e\x0a\x16\x12\x09\x0a\x0d\x0a\x16\x12\x10";
        char decrypted_key[20]={0,};

        // as plain key's first byte is 'x'
        char key='x';
        int total_len;
        int ret;

        show_banner();

        if(argc != 2) {
                printf("%s update_conf_filename\n", argv[0]);
                exit(0);
        }
        fp = fopen(argv[1], "r");

        if(!fp) {
                printf("Check out %s file.\n", argv[1]);
                exit(0);
        }
```

source code

```
        // Check the magic
        p = readline(fp);
        delete_newline(p);
        ret = check_magic(p);
        if(!ret) {
                printf("The magic is not matched.\n");
                exit(0);
        }
        free(p);

        // Get the filename
        p = readline(fp);
        delete_newline(p);
        tmp_p = get_next(p);
        if(DEBUG) {
                printf("Update filename: %s\n", tmp_p);
        }
        if(strlen(tmp_p) >= 255) {
                printf("Filename value is too long.\n");
                exit(0);
        }
        strcpy(gogo.filename, tmp_p);
        free(p);
```

source code

```
        // Get the date
        p = readline(fp);
        delete_newline(p);
        tmp_p = get_next(p);
        if(DEBUG) {
                printf("Date: %s\n", tmp_p);
        }
        if(strlen(tmp_p) >= 255) {
                printf("Date value is too long.\n");
                exit(0);
        }
        strcpy(gogo.date, tmp_p);
        free(p);

        // Get the size
        p = readline(fp);
        delete_newline(p);
        tmp_p = get_next(p);
        if(DEBUG) {
                printf("Size: %s\n", tmp_p);
        }
        if(strlen(tmp_p) >= 255) {
                printf("Size value is too long.\n");
                exit(0);
        }
```

source code

```
        strcpy(gogo.size, tmp_p);
        free(p);

        ret = get_file_size(gogo.filename);

        if(atoi(gogo.size) != ret) {
                printf("File size doesn't match.\n");
                exit(0);
        }

        // Get the MD5
        // I slightly modified MDFile() to store the md5 result into argument 2
        MDFile(gogo.filename, gogo.md5);

        if (DEBUG)
                printf("MD5 = %s\n", gogo.md5);

        // Get the KEY
        // XXX: this should be encyprted.
        go_dec(key, encrypted_key, decrypted_key);

        if (DEBUG)
                printf("gogo.KEY: %s\n", decrypted_key);
```

source code

```
        strcpy(gogo.KEY, decrypted_key);

        // Get the hash
        p = readline(fp);
        delete_newline(p);
        tmp_p = get_next(p);
        if(DEBUG) {
                printf("Hash: %s\n", tmp_p);
        }
        if(strlen(tmp_p) >= 255) {
                printf("hash is too long.\n");
                exit(0);
        }
        strcpy(gogo.hash, tmp_p);
        free(p);

        // Calculate hash
        // Allocating a heap memory for all attributes.
        // "+10" is dummy
        total_len = strlen(gogo.date) + strlen(gogo.size) + strlen(gogo.md5) + strlen(gogo.KEY)+10;
        p = malloc(total_len);
        if(!p) {
                printf("Not enough memory.\n");
                exit(0);
        }
```

source code

```
	memset(p, 0x00, total_len);
	sprintf(p, "%s|%s|%s|%s", gogo.date, gogo.size, gogo.md5, gogo.KEY);

	if(DEBUG)
		printf("Total: %s\n", p);

	tmp_p = malloc(40);
	if(!tmp_p) {
		printf("Not enough memory.\n");
		exit(0);
	}
	memset(tmp_p, 0x00, 40);

	// tmp = md5(A+B+C+D)
	MDString(p, tmp_p);

	// tmp2 = md5(tmp)
	memset(p, 0x00, total_len);
	MDString(tmp_p, p);

	final = malloc(70);
	if(!final) {
		printf("Not enough memory.\n");
		exit(0);
	}
```

source code

```
        memset(final, 0x00, 70);
        // final hash wil be tmp + tmp2
        sprintf(final, "%s%s", tmp_p, p);

        free(p);
        free(tmp_p);

        if(DEBUG)
                printf("final: %s\n", final);

        if(!strcmp(final, gogo.hash)) {
                printf("Congrats! You've passwed all the conditions, '%s' will be executed, soon.\n", gogo.filename);
                p = malloc(strlen(gogo.filename)+5);
                if(!p)
                        exit(0);
                memset(p, 0x00, strlen(gogo.filename)+5);
                sprintf(p, "./%s", gogo.filename);
                system(p);
        }
        else {
                printf("HASH doesn't match. You failed!\n");
        }
        free(final);
}
```

source code

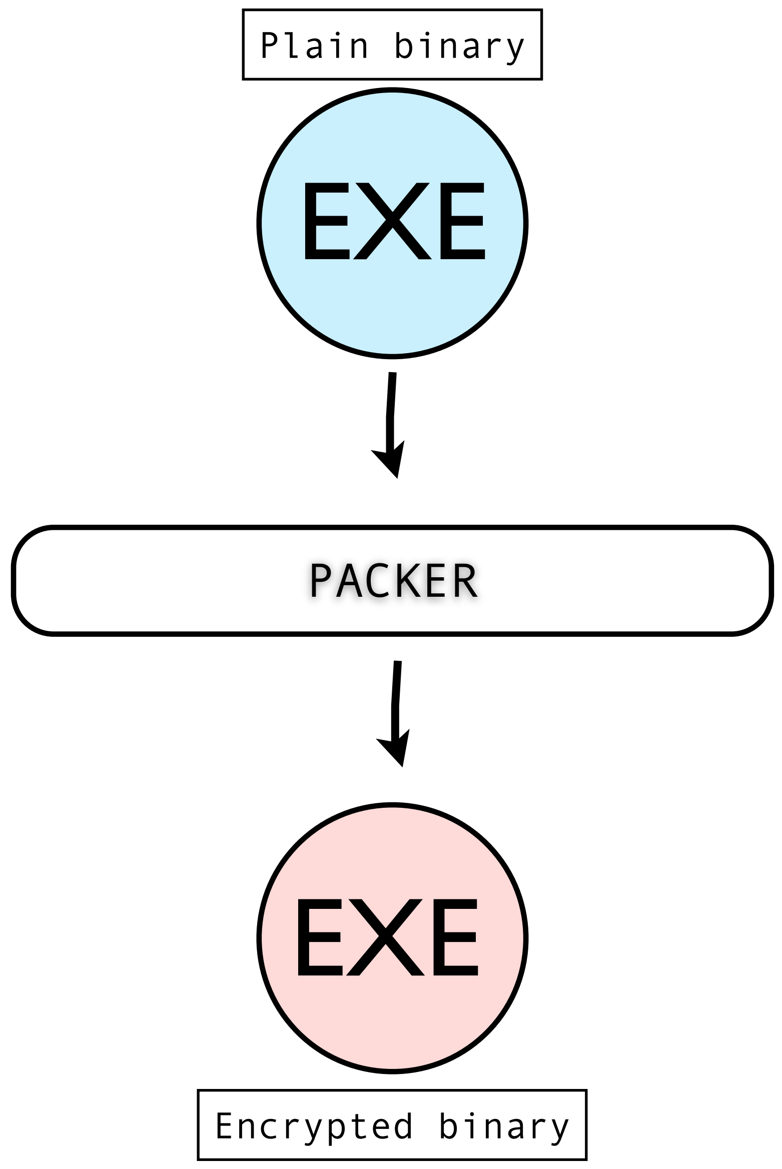
- The full source code is available here
    - http://115.68.24.145/armtest/installer.c

A simple packer

- Packers can reduce your code size
- But it’s mostly used for code-obfuscation
- We’ll make a simple packer and solve it ourself

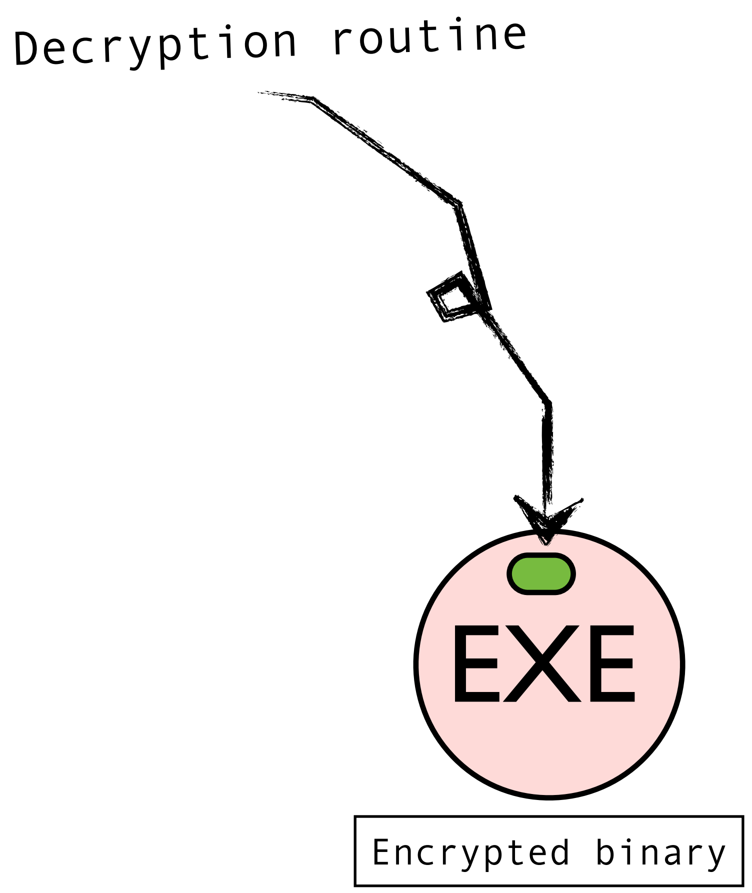
How packers work?

- Your EXE file is encrypted (packed) at static time
- And the encrypted EXE file will be decrypted at run time



How packers work?

- What our packer does
  - Open a file
  - Get the text segment section and code
  - Encrypt the code
  - Put a decryptor for the code into the file
  - Save the file



How packers work?

- When your encrypted file is loaded
  - It first gets to the decryption routine
  - The routine gets the encrypted code
  - Then decrypt the code
  - And write it on memory
  - Finally, the flow is jumping to the original code

Requirements

- What we need to make it
    - Basic ARM Assembly
    - ELF format (We're going to use *readelf* to get info)
    - Encoder/Decoder (or Encryptor/Decryptor)
- We'll only pack the text segment which means it'd be easy to implement

Packer Files

- target binary
  - a target program to pack
- arm_encoder.py
  - packing automatically
  - parse ELF format of the target
  - hook and xoring the main function
- asm.c
  - hooking code
  - mprotect() to execute, recovering the xor’ed main

arm_encoder.py

```
import os
import sys

# check out "XXX" comments in this file

# [TODO]
#
# 1) not changing original data to store our hooking code but adding a new segment and using it
#     - need to take a look at ELF format again (my drunken brain can't remember what i learned
for a long time ago)
# 2) anti-debug routines
#     - checked out some ARM manuals, and i think it's of course possible but
#     - would be platform-dependent? not sure. i'm an ARM newbie.
```

arm_encoder.py

```
# 3) not using 'external programs' like readelf (need to parse ELF format)
#    - easy one if we use ELF library or something
# 4) not injecting a jump code at the beginning of main() but finding a good place to hide
# 5) more obfuscation
#    - obbing

# [README]
#
# this ARM very simple obfuscator is made for training SAMSUNG embedded developers which
don't
# know about computer security well. this program has 2 files. one is this and the other is
# "original_asm.c" which is full of inline assembly and our hooking code.
# this program does xoring main function's code with 'x' and save them into a new file.
# to test this program, you need to get our QEMU + Linux for some reason (checkout 'XXX' in
this file)
```

arm_encoder.py

```
def change(filename, signature, replace_data):
  while 1:
    fp = open(filename)
    data = fp.read()
    fp.close()
    tmp = data.replace(signature, replace_data)
    if tmp == data:
      break
    else:
      fp = open(filename, "w")
      fp.write(tmp)
      fp.close()
      data = tmp


def get_text_start_address(target_program):
  a = "readelf -a %s | grep LOAD | grep \"R E\"| awk -F' ' '{print $3}' >
text_start_address.txt" % (target_program)
  os.system(a)
```

arm_encoder.py

```
def get_main(target_program):
  a = "readelf -a %s | grep \" main$\" | awk -F' ' '{print $2}' > main_address.txt" % (target_program)
  os.system(a)
  a = "readelf -a %s | grep \" main$\" | awk -F' ' '{print $3}' > main_size.txt" % (target_program)
  os.system(a)


def get_text_start_address_int():
  fp=open("./text_start_address.txt", "r")
  data = fp.read()
  fp.close()
  data = data.strip()
  return data
```

arm_encoder.py

```
def get_address_int():
  fp=open("./main_address.txt", "r")
  data = fp.read()
  fp.close()
  data = data.strip()
  row = data
  data = data[-3:]
  return int(data, 16), row


def get_size_int():
  fp=open("./main_size.txt", "r")
  data = fp.read()
  fp.close()
  data = data.strip()
  row = data
  return int(data)
```

arm_encoder.py

```
def do_xor(key, data):
  ret_buf = ""
  for x in data:
    tmp = ord(x) ^ ord(key)
    tmp = chr(tmp)
    ret_buf += tmp
  return ret_buf


if len(sys.argv) != 2:
  print "%s filename" % (sys.argv[1])
  sys.exit(0)

# clean up before starting
os.system("rm -fr %s;cp %s_original %s" % (sys.argv[1],
sys.argv[1], sys.argv[1]))

print "[O] Remove %s file and copy %s_original back
again." % (sys.argv[1], sys.argv[1])
```

arm_encoder.py

```
fp = open(sys.argv[1], "rb")
data = fp.read()
fp.close()


# XXX: todo
# it's shame but the gdb we're using on our QEMU linux makes always
# those string which will be in text area. so, we use those area for
# storing our hooking code. but we should give this away as it's poor.
# solution: make a new segment and store our hooking code there.
first_string = "/build/buildd-glibc_2.7-18lenny7-arm-fLI0zA/glibc-2.7/build-tree/arm-libc/csu/crti.S"
last_string = "/build/buildd-glibc_2.7-18lenny7-arm-fLI0zA/glibc-2.7/build-tree/glibc-2.7/csu"

offset = data.find(first_string)

if offset == -1:
  print "[X] Not matched for the first_string"
  sys.exit(0)
```

arm_encoder.py

```
print "[O] Garbage start_1 point found."

second_offset = data.find(last_string, offset)

if second_offset == -1:
  print "[X] Not matched for the second_string"
  sys.exit(0)

print "[O] Garbage start_2 point found."

print "[O] start_1 offset = %d, start_2 offset = %d" % (offset, second_offset)

aaaa = len(data[offset:second_offset+len(last_string)])
our_string_len = len(data[offset:second_offset+len(last_string)])

print "[O] Garbage buffer length = = %d" % (our_string_len)

# checkout "original_asm.c"
# the c file is full of inline assembly which is our hooking code.
# we make 2 signatures in our original_asm.c file.
# so that we can easily add/modify our hooking code to between those signatures.
```

arm_encoder.py

```
# bytecode for mov r1, r1 * 6
bin_first_string =
"\x01\x10\xa0\xe1\x01\x10\xa0\xe1\x01\x10\xa0\xe1\x01\x10\xa0\xe1\x01\x10\xa0\xe1\x01\
x10\xa0\xe1"
# bytecode mov r2, r2 * 6
bin_second_string =
"\x02\x20\xa0\xe1\x02\x20\xa0\xe1\x02\x20\xa0\xe1\x02\x20\xa0\xe1\x02\x20\xa0\xe1\x02\
x20\xa0\xe1"

get_text_start_address(sys.argv[1])
textstart = get_text_start_address_int()
dummy = "\".long %s\\n\"" % (textstart)
print "\tdummy(text_start_address): " + dummy

very_tmp = textstart[-8:]
very_tmp = int(very_tmp, 16)

get_main(sys.argv[1])
main_address, row_main_address = get_address_int()
```

arm_encoder.py

```
dummy2 = "\".long 0x%08x\\n\"" % (very_tmp + main_address)
print "\tdummy2(main_address): " + dummy2

dummy3 = "\".long 0x%08x\\n\"" % (get_size_int())
print "\tdummy3(main_size): " + dummy3

main_address_process = int(row_main_address, 16)
x = main_address_process + (offset-main_address)
x = hex(x)
x = x[2:]

print "[O] our payload will be at 0x%s at runtime" % (x)

main_address_process = x.decode("hex")

size_value = get_size_int()

print "[O] main function's size = %d" % (size_value)
```

arm_encoder.py

```
main_buffer = get_main_buffer(sys.argv[1], main_address, size_value)

dummy4 = "\".long 0x%02x%02x%02x%02x\\n\"" % (ord(main_buffer[3]),
ord(main_buffer[2]), ord(main_buffer[1]), ord(main_buffer[0]))
print "\tdummy4(original_first_4byte): " + dummy4

dummy5 = "\".long 0x%02x%02x%02x%02x\\n\"" % (ord(main_buffer[7]),
ord(main_buffer[6]), ord(main_buffer[5]), ord(main_buffer[4]))
print "\tdummy5(original_second_4byte): " + dummy5


# the xor key is 'x' for now
encrypted_main_buffer = do_xor('x', main_buffer)

fp = open(sys.argv[1], "rb++")


######## XOR
fp.seek(main_address)
fp.write(encrypted_main_buffer)
######## XOR
```

arm_encoder.py

```
fp.seek(main_address)
# this is: ldr pc, [pc, #-0x4]
fp.write("\x04\xf0\x1f\xe5")
fp.write("%c%c\x00\x00" % (main_address_process[1], main_address_process[0] ))

################### asm start

os.system("rm -fr base_asm.c")
os.system("cp original_asm.c base_asm.c")

change("base_asm.c", "SIG_LONG_BASEADDRESS", dummy)
change("base_asm.c", "SIG_LONG_MAINADDRESS", dummy2)
change("base_asm.c", "SIG_LONG_MAINSIZE", dummy3)
change("base_asm.c", "SIG_LONG_ORIGINAL_ONE1", dummy4)
change("base_asm.c", "SIG_LONG_ORIGINAL_ONE2", dummy5)
```

arm_encoder.py

```
os.system("gcc -o base_asm base_asm.c")

fp_tmp = open("./base_asm", "rb")
data = fp_tmp.read()
fp_tmp.close()

bin_first_offset = data.find(bin_first_string)

if bin_first_offset == -1:
  print "[X] Not matched for the bin_first_string"
  sys.exit(0)

print "[O] Our payload signatare_1 found."

bin_second_offset = data.find(bin_second_string,
bin_first_offset)

if bin_second_offset == -1:
  print "[x] Not matched for the bin_second_string"
  sys.exit(0)
```

arm_encoder.py

```
print "[O] Our payload signatare_2 found."

our_hex_payload = data[bin_first_offset
+len(bin_first_string):bin_second_offset]

print "[O] signature_1 offset = %d, signature_2 offset =
%d" % (bin_first_offset, bin_second_offset)

print "[O] Our payload size = %d" %
(len(our_hex_payload))

if len(our_hex_payload) > our_string_len:
  print "[X] our_hex_payload is too long."
  sys.exit(0)

our_hex_payload = our_hex_payload + ("\x00" *
(our_string_len - len(our_hex_payload)))
#################### asm end
```

arm_encoder.py

```
fp.seek(offset)
fp.write(our_hex_payload)

fp.close()


print "[O] main() in file offset: 0x%x" % (main_address)
print "[O] our_buffer in file offset: 0x%x" % (offset)
print "[O] offset from main() - our_buffer: %d" %
(offset - main_address)


os.system("rm -fr main_address.txt main_size.txt
text_start_address.txt")
print "[O] cleanup."
```

original_asm.c

```
main() {

	// signature 1
	// do not delete this
	asm(
	"mov r1, r1\n"
	"mov r1, r1\n"
	"mov r1, r1\n"
	"mov r1, r1\n"
	"mov r1, r1\n"
	"mov r1, r1\n"

	// main part
	// backup
	"push {r0-r14}\n"

	// this does mprotect(TEXT_ADDR, 0x1000, 0x7)
	// 0x7 is PROT_READ|PROT_WRITE|PROT_EXEC
	"bl TEXT_ADDRESS\n"
	"mov r1, #0x1000\n"
	"mov r2, #0x7\n"
	"svc #0x90007d\n"
```

original_asm.c

```
// Get the original 8byte and recover now
"bl MAIN_ADDRESS\n"
"bl GET_ENC1\n"
"bl GET_ENC2\n"
"str r1, [r0]\n"
"str r2, [r0, #0x4]\n"

// 0x78 = 'x' which is the xor key
"mov r5, #0x78\n"
"bl GET_SIZE\n"
"mov r3, #0x8\n"


// routine for decryping the original byte
// it starts from offset 0x8 as we already
// recovered it above
```

original_asm.c

```
// r0 = main address
// r3 = count
// r4 = size
// r5 = key used for xor
"xoring:"
"ldrb r1, [r0, r3]\n"
"eor r1, r5\n"
"strb r1, [r0, r3]\n"
"add r3, #0x1\n"
"cmp r3, r4\n"
"bne xoring\n"

// restore and back to main()
"pop {r0-r14}\n"
"b BACK_TO_MAIN\n"


// SIG_LONG_BASEADDRESS, SIG_LONG_MAINADDRESS, SIG_LONG_ORIGINAL_ONE1,
// SIG_LONG_ORIGINAL_ONE2, SIG_LONG_MAINSIZE will be automatically
// changed by arm_encoder.py
```

original_asm.c

```
"TEXT_ADDRESS:\n"
"ldr r0, [pc]\n"
"bx r14\n"
SIG_LONG_BASEADDRESS

"BACK_TO_MAIN:\n"
"ldr pc, [pc, #-0x4]\n"
SIG_LONG_MAINADDRESS

"MAIN_ADDRESS:\n"
"ldr r0, [pc]\n"
"bx lr\n"
SIG_LONG_MAINADDRESS

"GET_ENC1:\n"
"ldr r1, [pc]\n"
"bx lr\n"
SIG_LONG_ORIGINAL_ONE1
```

original_asm.c

```
"GET_ENC2:\n"
"ldr r2, [pc]\n"
"bx lr\n"
SIG_LONG_ORIGINAL_ONE2

"GET_SIZE:\n"
"ldr r4, [pc]\n"
"bx lr\n"
SIG_LONG_MAINSIZE
```

original_asm.c

```
        // signature 2
        // do not delete this
        "mov r2, r2\n"
        "mov r2, r2\n"
        "mov r2, r2\n"
        "mov r2, r2\n"
        "mov r2, r2\n"
        "mov r2, r2\n"
        );

}
```

Before packing

```
#(gdb) disassemble main
#Dump of assembler code for function main:
#0x00008474 <main+0>:    mov     r12, sp
#0x00008478 <main+4>:    push    {r11, r12, lr, pc}
#0x0000847c <main+8>:    sub     r11, r12, #4     ; 0x4
#0x00008480 <main+12>:   sub     sp, sp, #24      ; 0x18
#0x00008484 <main+16>:   str     r0, [r11, #-32]
#0x00008488 <main+20>:   str     r1, [r11, #-36]
#0x0000848c <main+24>:   ldr     r3, [pc, #120]  ; 0x850c <main+152>
#0x00008490 <main+28>:   sub     r2, r11, #25     ; 0x19
#0x00008494 <main+32>:   mov     r12, #13         ; 0xd
#0x00008498 <main+36>:   mov     r0, r2
#0x0000849c <main+40>:   mov     r1, r3
#0x000084a0 <main+44>:   mov     r2, r12
#0x000084a4 <main+48>:   bl      0x8334 <memcpy>
#0x000084a8 <main+52>:   ldr     r3, [r11, #-32]
#0x000084ac <main+56>:   cmp     r3, #2  ; 0x2
#0x000084b0 <main+60>:   beq     0x84c4 <main+80>
#0x000084b4 <main+64>:   ldr     r0, [pc, #84]   ; 0x8510 <main+156>
```

Before packing

```
#0x000084b8 <main+68>:   bl      0x8340 <puts>
#0x000084bc <main+72>:   mov     r0, #0  ; 0x0
#0x000084c0 <main+76>:   bl      0x8358 <exit>
#0x000084c4 <main+80>:   ldr     r3, [r11, #-36]
#0x000084c8 <main+84>:   add     r3, r3, #4      ; 0x4
#0x000084cc <main+88>:   ldr     r2, [r3]
#0x000084d0 <main+92>:   sub     r3, r11, #25    ; 0x19
#0x000084d4 <main+96>:   mov     r0, r3
#0x000084d8 <main+100>:  mov     r1, r2
#0x000084dc <main+104>:  bl      0x834c <strcmp>
#0x000084e0 <main+108>:  mov     r3, r0
#0x000084e4 <main+112>:  cmp     r3, #0  ; 0x0
#0x000084e8 <main+116>:  bne     0x84fc <main+136>
#0x000084ec <main+120>:  ldr     r0, [pc, #32]   ; 0x8514 <main+160>
#0x000084f0 <main+124>:  bl      0x8340 <puts>
#0x000084f4 <main+128>:  mov     r0, #0  ; 0x0
#0x000084f8 <main+132>:  bl      0x8358 <exit>
#0x000084fc <main+136>:  ldr     r0, [pc, #12]   ; 0x8510 <main+156>
#0x00008500 <main+140>:  bl      0x8340 <puts>
#0x00008504 <main+144>:  mov     r0, #0  ; 0x0
#0x00008508 <main+148>:  bl      0x8358 <exit>
#0x0000850c <main+152>:  strdeq  r8, [r0], -r4
#0x00008510 <main+156>:  ldrdeq  r8, [r0], -r4
#0x00008514 <main+160>:  ldrdeq  r8, [r0], -r12
#End of assembler dump.
```

After packing

```
#(gdb) disassemble main
#Dump of assembler code for function main:
#0x00008474 <main+0>:    ldr      pc, [pc, #-4]    ; 0x8478 <main+4>
#0x00008478 <main+4>:    andeq    r8, r0, lr, lsl r9
#0x0000847c <main+8>:    bls      0xd3a674
#0x00008480 <main+12>:   bls      0xd72608
#0x00008484 <main+16>:   ldclls   8, cr7, [r3, #-352]!
#0x00008488 <main+20>:   ldclls   8, cr6, [r3, #-368]!
#0x0000848c <main+24>:   stclls   8, cr4, [r7]
#0x00008490 <main+28>:   bls      0xcde61c
#0x00008494 <main+32>:   blls     0xff636670
#0x00008498 <main+36>:   ldmibls r8, {r1, r3, r4, r5, r6, r11, r12, sp, lr}^
#0x0000849c <main+40>:   ldmibls r8, {r0, r1, r3, r4, r5, r6, r11, sp, lr}^
#0x000084a0 <main+44>:   ldmibls r8, {r2, r4, r5, r6, r11, r12, lr}^
#0x000084a4 <main+48>:   orrls    r8, r7, #57147392        ; 0x3680000
#0x000084a8 <main+52>:   stclls   8, cr4, [r3, #-352]!
```

After packing

```
#0x000084ac <main+56>:   blls     0xae669c
#0x000084b0 <main+60>:   rsbsvc   r7, r8, #8060928          ; 0x7b0000
#0x000084b4 <main+64>:   stclls   8, cr7, [r7, #176]!
#0x000084b8 <main+68>:   orrls    r8, r7, #56623104         ; 0x3600000
#0x000084bc <main+72>:   blls     0xff6266a4
#0x000084c0 <main+76>:   orrls    r8, r7, #57671680         ; 0x3700000
#0x000084c4 <main+80>:   stclls   8, cr4, [r3, #-368]!
#0x000084c8 <main+84>:   bls      0xffeda6c0
#0x000084cc <main+88>:   stclls   8, cr5, [r11, #480]!
#0x000084d0 <main+92>:   bls      0xcda65c
#0x000084d4 <main+96>:   ldmibls r8, {r0, r1, r3, r4, r5, r6, r11, r12, sp, lr}^
#0x000084d8 <main+100>:  ldmibls r8, {r1, r3, r4, r5, r6, r11, sp, lr}^
#0x000084dc <main+104>:  orrls    r8, r7, #59244544         ; 0x3880000
#0x000084e0 <main+108>:  ldmibls r8, {r3, r4, r5, r6, r11, lr}^
#0x000084e4 <main+112>:  blls     0xae66cc
#0x000084e8 <main+116>:  rsbsvs   r7, r8, #8060928          ; 0x7b0000
```

After packing

```
#0x000084ec <main+120>:  stclls  8, cr7, [r7, #352]!
#0x000084f0 <main+124>:  orrls   r8, r7, #61341696        ; 0x3a80000
#0x000084f4 <main+128>:  blls    0xff6266dc
#0x000084f8 <main+132>:  orrls   r8, r7, #62390272        ; 0x3b80000
#0x000084fc <main+136>:  stclls  8, cr7, [r7, #464]!
#0x00008500 <main+140>:  orrls   r8, r7, #64487424        ; 0x3d80000
#0x00008504 <main+144>:  blls    0xff6266ec
#0x00008508 <main+148>:  orrls   r8, r7, #61341696        ; 0x3a80000
#0x0000850c <main+152>:  ldmdavc r8!, {r2, r3, r7, r8, r10, r11, r12, sp, lr, pc}^
#0x00008510 <main+156>:  ldmdavc r8!, {r2, r3, r5, r7, r8, r10, r11, r12, sp, lr, pc}^
#0x00008514 <main+160>:  ldmdavc r8!, {r2, r5, r7, r8, r10, r11, r12, sp, lr, pc}^
#End of assembler dump.
```

Download links

- http://115.68.24.145/armtest/encoder/arm_encoder.py
- http://115.68.24.145/armtest/encoder/original_asm.c

Bug hunting and exploiting

- The art of software security assessment
  - http://www.amazon.com/The-Software-Security-Assessment-Vulnerabilities/dp/0321444426/

- A bug hunter's diary
  - http://www.amazon.com/Bug-Hunters-Diary-Software-Security/dp/1593273851/

- The Mac Hacker's Handbook
  - http://www.amazon.com/The-Hackers-Handbook-Charlie-Miller/dp/0470395362/

- Fuzzing: Brute Force Vulnerability Discovery
  - http://www.amazon.com/Fuzzing-Brute-Force-Vulnerability-Discovery/dp/0321446119/

- Fuzzing for Software Security Testing and Quality Assurance
  - http://www.amazon.com/Fuzzing-Software-Security-Assurance-Information/dp/1596932147/

Bug hunting and exploiting

- iOS Hacker's Handbook
  - www.amazon.com/iOS-Hackers-Handbook-Charlie-Miller/dp/1118204123/

- Hunting Security bugs
  - http://www.amazon.com/Hunting-Security-Bugs-Tom-Gallagher/dp/073562187X/

- The Web Application Hacker's Handbook
  - http://www.amazon.com/The-Web-Application-Hackers-Handbook/dp/1118026470/

- The Shellcoder's Handbook: Discovering and Exploiting Security Holes
  - http://www.amazon.com/Shellcoders-Handbook-Discovering-Exploiting-Security/dp/047008023X/

- The Database Hacker's Handbook
  - http://www.amazon.com/The-Database-Hackers-Handbook-Defending/dp/0764578014/ref=pd_sim_b_39

Bug hunting and exploiting

- Hacking: The Art of Exploitation
  - http://www.amazon.com/Hacking-Art-Exploitation-Jon-Erickson/dp/1593271441/

- A Guide to Kernel Exploitation
  - http://www.amazon.com/Guide-Kernel-Exploitation-Attacking-Core/dp/1597494860/ref=pd_sim_b_8

- The Oracle Hacker's Handbook
  - http://www.amazon.com/Oracle-Hackers-Handbook-Hacking-Defending/dp/0470080221/

- The Art of Software Security Testing
  - http://www.amazon.com/The-Art-Software-Security-Testing/dp/0321304861/

Bug hunting and exploiting

- iOS Hacker's Handbook
  - www.amazon.com/iOS-Hackers-Handbook-Charlie-Miller/dp/1118204123/

- Hunting Security bugs
  - http://www.amazon.com/Hunting-Security-Bugs-Tom-Gallagher/dp/073562187X/

- The Web Application Hacker's Handbook
  - http://www.amazon.com/The-Web-Application-Hackers-Handbook/dp/1118026470/

- The Shellcoder's Handbook: Discovering and Exploiting Security Holes
  - http://www.amazon.com/Shellcoders-Handbook-Discovering-Exploiting-Security/dp/047008023X/

- The Database Hacker's Handbook
  - http://www.amazon.com/The-Database-Hackers-Handbook-Defending/dp/0764578014/ref=pd_sim_b_39

Reversing

- Reversing: Secrets of Reverse Engineering
  - http://www.amazon.com/Reversing-Secrets-Engineering-Eldad-Eilam/dp/0764574817/

- Hacker Disassembling Uncovered
  - http://www.amazon.com/Hacker-Disassembling-Uncovered-Kris-Kaspersky/dp/1931769648/

- Rootkits: Subverting the Windows Kernel
  - http://www.amazon.com/Rootkits-Subverting-Windows-Greg-Hoglund/dp/0321294319/

- Gray Hat Python
  - http://www.amazon.com/Gray-Hat-Python-Programming-Engineers/dp/1593271921/

- The IDA Pro Book
  - http://www.amazon.com/The-IDA-Pro-Book-Disassembler/dp/1593272898/

Bug hunting and exploiting

- http://openrce.org
- http://www.reddit.com/r/ReverseEngineering/
- http://woodmann.com
- http://www.crackmes.de/

War game sites

- http://smashthestack.org/
  - http://io.smashthestack.org:84/

- http://www.overthewire.org/wargames/

- http://webhacking.kr

- http://hackerschool.org

- http://codeengn.com/challenges/

Security conferences

- http://blackhat.com
- http://defcon.org
- http://syscan.org
- http://en.avtokyo.org
- http://www.ruxconbreakpoint.com
- http://www.ruxcon.org.au/
- http://hitb.org

Security conferences

- http://www.immunityinc.com/infiltrate
- http://www.ekoparty.org
- http://recon.cx
- http://hackitoergosum.org
- https://events.ccc.de/congress/
- http://xcon.xfocus.org
- http://hack.lu

Security conferences

- http://hitcon.org
- http://www.h2hc.org.br
- https://www.kiwicon.org
- http://www.summercon.org
- http://secuinside.com
- http://codegate.org
- http://isecconference.org
- http://codeengn.com
- http://www.powerofcommunity.net

War game sites

- http://smashthestack.org/
    - http://io.smashthestack.org:84/
- http://www.overthewire.org/wargames/
- http://webhacking.kr
- http://hackerschool.org
- http://codeengn.com/challenges/

Thanks!

- Email: beist@grayhash.com
- Website: http://grayhash.com
- Twitter: http://twitter.com/beist