

운영체제 Assignment 4

이름 : 이준휘

학번 : 2018202046

교수 : 최상호 교수님

강의 시간 : 금 1, 2

While 문을 통해 vm이 유효한 동안 출력을 수행한다. 현재 file 정보를 file->vm file에서 가져


온다. 만약 가져온 파일 정보가 유효하지 않은 경우 다음 vm을 탐색한다. 만약 유효한 경우 D_path를 통해 해당 파일의 path를 가져와 file_path에 저장한다. 이후 mem 주소는 vm->vm_start, vm->vm_end로, code의 주소는 mm->start_code, mm->end_code로, data의 주소는 mm->start_data, mm->end_data로, heap의 주소는 mm->start_brk, mm->brk로 출력하며, file_path에 저장된 파일 위치를 출력한다.

Make_rw()함수는 system call table에 쓰기 권한을 부여한다.

Make_ro()함수는 system call table에 쓰기 권한을 뺀다.

Module initialize 시 sys_call_table을 가져와 쓰기 권한을 부여한다. 이후 현재 ftrace 위치의 함수를 real_trace에 저장하고, file_varea 함수를 해당 위치에 hooking한다. 이후 쓰기 권한을 뺀다.

Module exit 시 sys_call_table에 쓰기 권한을 부여하고, 원래 ftrace로 해당 system call을 변경한다. 이후 쓰기 권한을 뺀다.



```
root@kali:~/kernel# cat Makefile
KDIR := /lib/modules/$(uname -r)/build
PWD := $(shell pwd)
CFLAG := -DEXPORT_SYMTAB

all:
    $(MAKE) $(CFLAG) -C $(KDIR) SUBDIRS=$(PWD) modules
clean:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean
```

The screenshot shows a terminal window with a dark background. The prompt is root@kali:~/kernel#. The user has entered the command cat Makefile, and the output of the Makefile is displayed. The Makefile defines KDIR as /lib/modules/\$(uname -r)/build, PWD as \$(shell pwd), and CFLAG as -DEXPORT_SYMTAB. It also defines targets for 'all' and 'clean'.

해당 파일은 모듈을 만들기 위한 Makefile이다. 이전 2, 3 과제와 같은 방법으로 파일을 작성하였다.

XX 형태로 add와 비슷하게 이루어져 있다. Imul의 경우 06 C0 XX의 형태로 이 또한 add와 sub과 유사한 형태로 동작한다. 마지막으로 div의 경우 f6 f2로 d의 값을 나누는 모습을 보인다. 이를 통해 x86에서 assembly language가 어떻게 생겼는지 확인하여 패턴을 특정할 수 있었다.

```
cs2018202046@ubuntu:~/csk_4_2018202046_0/Assignments_2
//for using MAP_ANONYMOUS
#define GNU_SOURCE

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <assert.h>
#include <fcntl.h>
#include <time.h>

uint8_t* Operation; //memory mapping pointer
uint8_t* compiled_code; //shared_memory pointer

int segment_id;

void sharedmem_init(); // Shared Memory attach
void sharedmem_exit();
void decompile_init(); // memory mapping(ANONYMOUS and PRIVATE)
void decompile_exit();
void* decompile(uint8_t* func); //Assembly Optimization

int main(void)
{
    int (*func)(int a);
    int output;
    struct timespec start_time, end_time;
    double t;

    sharedmem_init(); //attach shared_memory
    decompile_init(); //Memory Allocation

    //Assembly Optimize
    func = (int (*)(int a))decompile(Operation);

    //Check start time
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    //run
    for(int i = 0; i < 10000000; i++)
        output = (*func)(i);

    //Check end time
    clock_gettime(CLOCK_MONOTONIC, &end_time);

    //Calculate time and print
    t = (end_time.tv_sec - start_time.tv_sec) + (double)(end_time.tv_nsec - start_time.tv_nsec) / 1000000000;
    printf("Total execution time: %f\n", t);
    printf("Output : %d\n", output);

    //decompile_exit(); //Memory deallocation
    sharedmem_exit(); //detach shared_memory
    return 0;

    void sharedmem_init()
    {
        //get shared memory using key
        segment_id = shmget((key_t)1, (size_t)PAGE_SIZE, 0);
        assert(segment_id != -1);

        //attach shared memory using segment id
        compiled_code = (uint8_t *)shmat(segment_id, NULL, 0);
        assert(compiled_code != (uint8_t *)-1);
    }

    void sharedmem_exit()
    {
        //remove shared memory using segment id
        int shmctl_code = shmctl(segment_id, IPC_RMID, NULL);
    }
}

cs2018202046@ubuntu:~/csk_4_2018202046_0/Assignments_2

void decompile_exit()
{
    //Memory Unmapping
    int munmap_code = munmap(Operation, PAGE_SIZE);
    assert(munmap_code != -1);

    void* decompile(uint8_t* func)
    {
        int i = 0;
        int protect_code;

        #ifdef dynamic
        //dynamic option(optimize)
        uint8_t prev_op[] = { 0 }, op_size = 0, dl = 0;

        //save dl
        if(compiled_code[i] == 0x80) dl = compiled_code[i + 1];

        //MOV SIB IMUL case
        if(compiled_code[i] == 0x80) || compiled_code[i] == 0x80){
            //current limit
            prev_op[0] = compiled_code[i++];
            prev_op[1] = compiled_code[i++];
            prev_op[2] = compiled_code[i++];
            prev_op[3] = compiled_code[i++];

            //optimize
            for(; prev_op[0] == compiled_code[i] && compiled_code[i + 1]; i += 2){
                if(prev_op[0] == 0x80) prev_op[1] = compiled_code[i + 2];
                else prev_op[2] = compiled_code[i + 2];
            }

            //Insert last
            *func++ = prev_op[0];
            *func++ = prev_op[1];
            *func++ = prev_op[2];
            *func++ = prev_op[3];
        }

        //DIV case
        else if(compiled_code[i] == 0x90){
            //current limit
            prev_op[0] = compiled_code[i++];
            prev_op[1] = dl;

            //optimize
            for(i++; prev_op[0] == compiled_code[i]; i += 2){
                prev_op[1] = dl;
            }

            //MOV dl prev_op[i]
            *func++ = 0x80;
            *func++ = prev_op[1];
            //DIV dl
            *func++ = prev_op[0];
            *func++ = prev_op[1];

            //Not AND SIB IMUL DIV
            else *func++ = compiled_code[i++];
        } while(compiled_code[i - 1] != 0x00);

        #else
        //default option(just copy)
        *func++ = compiled_code[i];
        while(compiled_code[i++] != 0x00);
        #endif

        //change to Rx
        protect_code = mprotect(Operation, PAGE_SIZE, PROT_READ | PROT_EXEC);
        assert(protect_code != -1);

        return (void *)Operation;
    }
}
```

위의 파일은 shared memory에서 컴파일된 코드를 가져와 조건에 따라 재컴파일을 진행하는 D_recompile.c이다. 해당 파일에서는 전역변수로 저장할 재컴파일할 함수를 저장할 포인터 Operation과 shared memory의 데이터를 가리킬 compiled_code 포인터가 존재한다. 그리고 compiled_code의 shm id를 저장할 segment id가 있다.

Sharedmem_init() 함수에서는 shmget() 함수를 통해 D_recompile_test.c 파일에 key값으로 설정된 1234를 통해 shared memory의 id를 가져온다. 이후 compiled code가 가리키는 값을

shmat() 함수를 통해 segment_id가 가리키는 메모리로 attach한다.

Sharedmem_exit() 함수는 segment_id를 shmctl()의 IPC_RMID 명령을 사용하여 없애준다.

Drecompile_init() 함수는 mmap()함수를 사용하여 Operation에 임의의 메모리 공간을 mapping 해준다. 이 때 MAP_PRIVATE을 통해 해당 프로세스만 단독으로 사용하며, MAP_ANONYMOUS를 통해 다른 file descriptor를 사용하지 않고 생성할 수 있다. 권한은 RW로 제한한다.

Drecompile_exit()함수에서는 Operation의 위치를 활용하여 munmap()함수로 unmapping시켜준다.

Drecompile() 함수는 shared_memory의 영역에서 mapping된 영역으로 데이터를 복사하는 과정이 있다. #ifdef 문을 사용하여 dynamic 옵션이 존재하는 경우에는 optimized된 컴파일 복사를 제공하며, 옵션이 없을 경우에는 단순히 0xC3(종료 지점의 값)까지 Operation에 해당하는 func에 복사한다.

Dynamic 옵션이 적용되었을 때 0xC3까지 다음 명령을 반복한다.

이전 코드를 보았을 때 0xB2를 입력받은 경우 다음 값은 dl에 입력할 값이기 때문에(mov) 해당 값을 임시로 저장한다.

만약 현재 컴파일 코드의 위치의 값이 0x83 또는 0x68인 경우 add sub imul 명령이기 때문에 3가지 인자를 저장할 필요가 있다. 이에 prev_op에 해당 3가지 상태를 모두 저장한다. 그 후 반복문을 통해 만약 기존 명령과 중복되는 명령일 경우 반복을 수행한다. 이 때 0x83은 덧셈 연산을(add or sub), 아닌 경우 곱셈 연산을 통해 prev_op[2] 값을 업데이트한다. 반복문을 탈출한 경우 해당 코드를 func에 넣어준다.

만약 현재 컴파일 코드의 위치의 값이 0xF6인 경우 div 명령이기 때문에 2가지 인자를 저장할 필요가 있다. Prev_op[1]에 dl의 값을 저장하며 이전과 마찬가지로 명령이 중복될 때 마다 dl의 값을 곱 누적하여 업데이트를 진행한다. Sub 명령의 경우 dl 레지스터의 값을 나누기 때문에 mov 명령을 통해 현재 prev_op[1]의 값을 dl reg에 넣어두며 이후 기존 명령과 같은 나눗셈 연산을 진행한다.

이외의 경우에는 기존의 코드를 단순 복사한다.

#endif 이후 Operation의 권한을 RX로 변경하기 위해 mprotect() 함수를 사용한다. 그 후 Operation의 주소를 반환하며 함수를 종료한다.

Main 함수의 동작은 다음과 같다. Func 포인터는 실행 시에 사용할 포인터이며, struct에 변수의 값을 저장할 예정이다. Sharedmem_init()과 drecompile_init()을 통해 shared memory와 이를 저장할 memory 공간을 준비한다. 이후 drecompile(Operation) 함수를 수행하며 이 결과를 func이 가리키는 값으로 한다. 그 후 clock_gettime() 함수를 사용하여 시작 시간을 체크하며, 1,000,000회 동안 func을 수행한다. 이후 해당 종료 시간을 체크하고, 시작시간과 종료 시간의 차이를 이용하여 수행 시간을 계산하여 출력한다. 이후 dercompile_exit()과 sharedmem_exit() 함수를 사용하여 각 메모리 공간을 할당 해제한다.

```
os2018202046@ubuntu:~/os_4_2018202046_0/assignment4_2
$ gcc -o D_recompile
$ cc -o gcc

default:
$ (cc) -o D_recompile D_recompile.c -lrt

dynamic:
$ (cc) -dynamic -o D_recompile D_recompile.c -lrt

clean:
rm -rf D_recompile $(CXXC)

"makefile" [noel] 91, 174C
```

해당 파일은 Makefile로 기존의 주어진 형식 예시와 동일하게 작성되었다.

```
os2018202046@ubuntu:~/os_4_2018202046_0/assignment4_2
$ gcc -o test2 D_recompile_test.c
os2018202046@ubuntu:~/os_4_2018202046_0/assignment4_2$ make
gcc -o D_recompile D_recompile.c -lrt
os2018202046@ubuntu:~/os_4_2018202046_0/assignment4_2$ ./D_recompile
D_recompile: D_recompile.c:64: sharedmem_init: Assertion 'segment_id != -1' failed.
Aborted (core dumped)
os2018202046@ubuntu:~/os_4_2018202046_0/assignment4_2$
```

위의 결과는 shared memory를 할당하지 않고 수행할 경우 해당 영역이 잡히지 않아 failed가 뜨는 모습을 보인다.

```
ms2018202046@ubuntu:~/hw_4_2018202046_0/assignment4_2
ms2018202046@ubuntu:~/hw_4_2018202046_0/assignment4_2$ sync
ms2018202046@ubuntu:~/hw_4_2018202046_0/assignment4_2$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3
ms2018202046@ubuntu:~/hw_4_2018202046_0/assignment4_2$ ./test2
Data was filled to shared memory. page size = 4096
total execution time: 0.391709
output = 15
ms2018202046@ubuntu:~/hw_4_2018202046_0/assignment4_2$
```

다음은 이전의 gcc -o test2 D_recompile_test.c 명령을 통해 test2 프로그램을 생성하여, 이를 먼저 수행한 후, 단순한 make를 통해 ./drecompile을 생성한 모습이다. ./test2를 통해 shared_memory에 공간이 할당되었으며 이후 ./drecompile 함수에서 정상적으로 함수가 수행되어 결과가 나온 모습을 볼 수 있다. 수행 시간은 약 0.391709초 소요되었으며 결과는 15로 나왔다.

```
ms2018202046@ubuntu:~/hw_4_2018202046_0/assignment4_2
ms2018202046@ubuntu:~/hw_4_2018202046_0/assignment4_2$ make dynamic
gcc -Ddynamic -o drecompile D_recompile.c -lrt
ms2018202046@ubuntu:~/hw_4_2018202046_0/assignment4_2$ sync
ms2018202046@ubuntu:~/hw_4_2018202046_0/assignment4_2$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3
ms2018202046@ubuntu:~/hw_4_2018202046_0/assignment4_2$ ./test2
Data was filled to shared memory. page size = 4096
total execution time: 0.099794
output = 15
ms2018202046@ubuntu:~/hw_4_2018202046_0/assignment4_2$
```

다음은 make dynamic를 통해 ./drecompile을 생성한 모습이다. ./test2를 통해 shared_memory에 공간이 할당되었으며 이후 ./drecompile 함수에서 정상적으로 함수가 수행되어 결과가 나온 모습을 볼 수 있다. 수행 시간은 약 0.099794초 소요되었으며 결과는 15로 나왔다. 이는 이전의 결과보다 시간이 매우 단축된 모습을 보였다.

Default									
0.391709	0.390154	0.392357	0.391474	0.392061	0.395001	0.390843	0.392604	0.391756	0.392202
0.39485	0.394798	0.393754	0.400061	0.393499	0.394841	0.395131	0.394317	0.40101	0.402936
0.397048	0.394051	0.395587	0.39319	0.397692	0.395443	0.395986	0.395117	0.398752	0.394438

0.393827	0.392655	0.393901	0.397765	0.393109	0.394664	0.392269	0.393231	0.389908	0.396107
0.399879	0.393079	0.395849	0.394019	0.391127	0.394237	0.394473	0.392911	0.391879	0.393636
Dynamic									
0.099794	0.098883	0.098023	0.099004	0.099363	0.099327	0.099502	0.1002	0.099207	0.100603
0.099495	0.098289	0.08798	0.099492	0.100736	0.100321	0.098316	0.098352	0.10014	0.099183
0.098585	0.099772	0.099452	0.099204	0.09956	0.098614	0.098699	0.102665	0.098196	0.099704
0.099576	0.099824	0.09848	0.099702	0.098717	0.099296	0.099042	0.099934	0.098974	0.099452
0.099777	0.099972	0.098465	0.098662	0.10131	0.099784	0.097939	0.09886	0.100616	0.099409
Default의 50 set의 값은 다음과 같이 나왔으며 평균 0.39442374초가 소요되었다. 이에 반해 Dynamic 50 set의 값은 평균 0.09916904초가 나와 약 3.977287배의 성능 개선이 이루어졌다. 이에 해당 목표에 맞게 정확히 동작함을 확인하였다.									

3. Consideration

해당 과제를 통해 task_struct 내에 존재하는 메모리가 어떠한 구조로 이루어져 있으며, 이를 통해 가상 메모리 주소와 실제 위치, 파일 경로 등을 추적하는 방법을 알 수 있었다. 또한 shared memory와 관련한 함수에 대하여 익힐 수 있는 과제였다. 그리고 mmap 함수를 통해 특정 파일 혹은 가상의 메모리 공간을 할당하는 방식을 직접 실습할 수 있었다. 마지막으로 프로그램을 objdump를 통해 자세히 확인하여 recompile을 하는テクニック을 익히 수 있는 과제였다. 다만 과제 중 기존에 주어진 Operation 함수가 1이 아닌 값에서는 동작하지 않는다는 사실이 주어지지 않아 해당 부분에서 Floating Point Exception으로 인해 많은 시간이 소요되었다.

4. Reference

- 강의자료만을 참고