Project #2: 보호모드 전환 및 Secure Boot 단순구현

신정은, 조재호, 조한주
Soongsil University

1. 구현 목표

리얼모드에서 MINT64OS가 동작하는 주요 모드인 1A-32e 모드로 전환하기 위해서는 반드시 32비트 보호모드를 거쳐야 한다. 따라서 이번 프로젝트에서는 리얼모드에서 보호모드로 전환하는 과정을 구현하도록 한다.

리얼 모드에서 보호 모드로 전환하기 전에 해야할 일이 있다. 보호모드 전환에 필요한 자료구조인 세그먼트 디스크립터와 GDT정보를 생성하고, 생성된 자료구조를 프로세서에 설정하는 것이다. 이 두가지 자료구조는 보호모드로 전환하는 즉시 프로세서에 의해 참조되므로 위의 과정을 수행한 후에 보호모드로 전환해야 한다.
보호모드로 전환하는 과정 후에는 [BITS 32]를 통해 어셈블리어에게 보호 모드용 코드임을 알린다. 리얼모드에서 보호모드로 전환 되면 스택의 크기가 2바이트에서 4바이트로, 범용 레지스터의 크기가 32비트로 증가한 것과 세그먼트 레지스터(셀렉터)의 기능이 변한 것을 이해한다.
이렇게 자료구조를 준비하고 설정하는 과정을 거치면 커널 이미지가 생성된다. 이로써 우리는 가상 이미지 대신에 실제 커널 이미지를 메모리에 올려 우리의 목표에 한발 다가서게 된다.

보호모드로 전환한 후부터는 C언어로 커널을 작성한다. 보호모드 엔트리 포인트의 뒷부분에 C언어로 작성한 커널을 연결하고 엔트리 포인트에서 C커널 시작 부분으로 이동시켜 C커널이 동작하게 한다. 이때 링커스크립트를 활용하여 C커널이 동작하도록 제약 조건에 맞게 빌드 하도록 한다.
또한 커널 크기가 변경될 때마다 부트로더 코드의 TOTALSECTORCOUNT 값을 수정하는 것이 아니라 빌드시에 이를 자동으로 업데이트 되게 하여 작업환경을 개선하도록 한다.

감염된 커널이 실행되면 악성코드가 설치되거나 부팅자체가 무력화되므로 보안 측면에서 다소 위험하다. 이를 방지하기 위해 Secure boot 기능을 추가한다. Secure boot는 부팅 시 사용되는 부트로더 및 커널코드가 인증된 상태인지 확인하고 부팅을 허용하는 기능이다. 이 기능을 구현해 봄으로써 Secure boot가 작동하는 과정을 직접 확인한다.

따라서 본 프로젝트에서 구현해야 할 구현 목표는 다음과 같다.

1) 리얼 모드에서 보호 모드로 전환
2) c언어로 커널 작성
3) Secure Boot를 단순화하여 구현

최종 구현 결과 화면에는 다음과 같이 출력되도록 한다.
-인증 성공 시 결과 화면

```
MINT64 OS Boot Loader Start~!!
OS Image Loading··· Complete~!!
OS Image Checking··· Okay~!!
Switch To Protected Mode Success~!!
C Language Kernel Started~!!
```

-인증 실패 시 결과 화면

```
MINT64 OS Boot Loader Start~!!
OS Image Loading··· Complete~!!
OS Image Checking··· Fail~!!
Loaded Hash Value: (커널이미지에 저장된 해쉬값)
CalcuatedHash Value:
                    (로딩된 커널이미지에서 계산한 해쉬값)
```

2. 구현 내용

2.1 리얼모드에서 보호모드로 전환

보호모드로 전환하는 즉시 프로세서에 의해 참조되므로 세그먼트 디스크립터와 GDT를 생성해야 한다. 우리가 구현할 MINT64 OS는 세그먼트 디스크립터에 대해서 아래 4가지 조건이 만족하도록 설정해야 한다.

```
1.커널 코드, 데이터용 세그먼트 디스크립터 필요
2.커널세그먼트 디스크립터는 4GB 전체에 접근가능
3.오퍼랜드 크기를 32비트로 설정
4.최상위 권한으로 설정
```

GDT는 널 디스크립터를 앞부분을 하여 코드 세그먼트 디스크립터, 데이터 세그먼트 디스크립터를 어셈블리어 코드로 나타내어 생성한다. 그 후 프로세서에 GDT의 시작 어드레스와 크기 정보를 로딩한다. 자료구조가 준비되었으니 이제 GDTR 레지스터와 CR0 컨트롤 레지스터를 설정해준다. CR0 컨트롤 레지스터는 다양한 기능이 있지만 여기서는 세그먼테이션 기

능만 사용하므로 0x4000003B로 설정한다.

이제 32비트 코드를 준비하고 jmp 명령으로 CS 세그먼트 셀렉터를 교체한다.

이 모든 과정을 EntryPoint.s 파일에 작성하고 커널 이미지(Kernel32.bin)를 생성한다.

2.2 c언어로 커널 작성

엔트리 포인트가 C 코드를 실행하려면 아래 3가지 조건을 만족해야 한다.

1.C 라이브러리를 사용하지 않게 빌드해야 한다.
2. 0x10200 위치에서 실행해야 한다.
3. 코드, 데이터만 포함된 바이너리 파일이다.

이 세 조건이 만족하도록 makefile을 작성하고 링커 스크립트를 작성한다. 또한 make의 와일드카드 기능을 사용하여 .c 확장자의 파일만 추가되면 자동으로 포함하여 빌드하게 수정한다.

커널 크기가 변경되더라도 자동으로 TOTALSECTORCOUNT 값이 변경되게 하기 위해 ImageMaker.c 파일을 생성하여 부트로더와 커널 이미지를 통합하여 섹터 크기(512byte)로 정렬하고, BootLoader.bin 파일의 시작으로부터 5byte 떨어진 곳에 섹터 수를 쓴다.

2.3 Secure Boot를 단순화하여 구현

Secure Boot 구현은 3번의 과정을 통하여 구현한다

(1).ImageMaker.c 파일을 수정해 hash 값을 계산하여 커널 이미지 최상단에 4 byte hash 값을 저장한다.
(2). BootLoader.asm 파일을 수정하여 로딩된 커널 이미지로 4 byte hash값을 구한 후 커널 이미지 0번지에 저장된 해쉬값과 비교한다.
(3). 해쉬값이 같으면 커널 이미지 처음으로 jmp하여 32비트 보호모드와 C커널을 실행한다.

Kernel32.bin 파일을 Disk.img 파일에 복사한 후 코드 1에서처럼 섹터 크기로 맞춰진 커널 이미지를 4 byte씩 읽어 hash값을 계산한다.

```
while(1){

    iRead = read(iTargetFd,vcBuffer,sizeof(vcBuffer));

    if(count==0)
        for(int i=0;i<sizeof(vcBuffer);++i)
            byteHash[i] = vcBuffer[i];
    else
        for(int i=0;i<sizeof(vcBuffer);++i)
            byteHash[i] ^= vcBuffer[i];

    if(iRead!=sizeof(vcBuffer))
        break;

    count++;
}
```

코드 1. Kernel Image Hash Calculation

기존의 Kernel32.bin 파일을 4 byte씩 미뤄 최상단에 계산한 해쉬값을 저장한다. 메모리에 로딩될 Disk.img 파일에는 해쉬값이 없는 원본 커널 이미지가 복사되어 있으므로 다시 Kerne32.bin 파일을 Dsik.img 파일의 512byte부터 복사한 뒤 섹터 크기로 맞추기 위해 0으로 값을 채운다.

부트 로더가 커널 이미지를 메모리에 로딩한 후에 커널 이미지 최상단 4 byte에 있는 해쉬값을 제외한 나머지 커널 이미지를 4 byte씩 xor하여 해쉬값을 계산한다.

```
CHECKOS:
    mov si, 0x1000
    mov es, si
    mov si, 4

    mov cx, word [ es : si ]
    mov dx, word [ es : si+2 ]

.HASHLOOP:
    add si,4

    cmp si, 1024
    je .ENDHASHLOOP

    mov ax, word[es:si]

    mov bx,word[es:si+2]
    xor cx,ax
    xor dx,bx

    jmp .HASHLOOP
```

코드 2. Boot loader hash calculation

코드 2는 커널 이미지 4 byte부터 즉 디스크 이미지 0x10004부터 2 byte씩 읽어 각각 cx,dx 레지스터에 저장한다. si 레지스터로 인덱스 값을 4 만큼 증가시키면서 ax,bx 레스터로 2byte씩 읽어 xor연산을 통한 결과값을 다시 cx,dx 레지스터에 저장하여 최종적으로 cx,dx 레지스터에 메모리에 로딩된 커널 이미지로 계산한 hash값이 저장된다.

이제 계산 한 hash값과 로딩된 커널 이미지 최상단에 있는 hash값을 비교한다.

만약 해쉬값이 같으면 'okay~!' 라는 문자열을 출력하고 커널 이미지로 jmp하여 32비트 보호모드를 실행한다. 하지만 해쉬값이 다르면 'Fail~!' 이라는 문자열을 출력 한 뒤에 커널 이미지에 저장된 해쉬값과 로딩된 커널 이미지에서 계산한 해쉬값을 출력한다.

hash값의 출력은 2가지 함수 STOREHASH와 STORECHA로 구현한다. KernelImage에 쓰여진 hash 값은 메모리 [0x1000:0x00]번지부터 [0x1000:0x04] 번지에 올라가 있고 계산한 hash 값은 cx, dx 레지스터에 들어 있다.

요지는, 4 바이트로 저장된 hash 값(16진수 8자리)을 문자열 8바이트로 변환하여 출력하는 것이다. 이를 위해 데이터 영역에 HASHBUF로 8바이트를 잡아놓는다.

```
push word[ es:si ]
push word[ es:si+2 ]
call STOREHASH
add sp, 4
push cx
push dx
call STOREHASH
add sp, 4
```

es 레지스터(0x1000), si 레지스터(0x00)를 이용해 메모리의 hash 값을 스택에 넣고 STOREHASH 함수를 호출하면 HASHBUF의 8바이트에 hash 값이 저장된다. 이 후 PRINTMESSAGE 함수로 화면에 출력한다. cx, dx 레지스터를 이용해 계산한 hash 값도 같은 방법으로 출력한다.

```
STOREHASH:
    push bp
    mov bp, sp
    push si
    mov si, 0

.STOREHASHLOOP:
    add si, 1
    cmp si, 5
    jge .STOREHASHEND

    mov ax, 0
    mov al, byte[ bp + si + 3 ]
    push ax

    mov ax, si
    sub ax, 1
    mov bx, 2
    mul bx
    push ax

    call STORECHA    ; STORE CHARACTER
    add sp, 4
    jmp .STOREHASHLOOP

.STOREHASHEND:
    pop si
    pop bp
    ret
```

STOREHASH 함수는 위와 같다. 4 바이트의 hash 값이 스택에 저장되어있기 때문에 si 레지스터가 순회하는 loop문을 만들어 hash의 각 byte를 스택에 저장한다. 이 때, 1 byte의 hash 값이 문자 2 byte로 표현되므로 HASHBUF의 2byte를 스택에 저장하기 위해 2 * si의 단위로 스택에 저장한다.

그리고 STORECHA 함수를 호출한다.

STORECHA 함수가 한번 호출될 때마다 글자가 HASHBUF에 2 byte씩 저장되고 총 4번 호출하여 hash value 8 글자가 저장된다.

```
STORECHA:
    push bp
    mov bp, sp
    push si

    mov ax, word[ bp + 6 ]
    mov si, 0

.STORECHALOOP:
    add si, 1
    cmp si, 2
    je .SECOND
    cmp si, 3
    jge .STORECHAEND
    shr al, 4
    jmp .COMMON

.SECOND:
    mov ax, word[ bp + 6 ]
    and al, 0xf
.COMMON:
    cmp al, 10
    jge .OVERTEN
.OVERTENRETURN:
    add al, 0x30
    mov bx, word[ bp + 4 ]
    mov cx, si
    mov si, bx
    add si, cx
    dec si
    mov byte[ HASHBUF + si ], al
    mov si, cx
    jmp .STORECHALOOP

.OVERTEN:
    add al, 0x07
    jmp .OVERTENRETURN

.STORECHAEND:
    pop si
    pop bp
    ret
```

STORECHA 함수는 ASCII 문자 2 글자로 바꾸어야 할 1 byte와 문자가 저장될 주소를 인자로 받는다. 먼저 loop를 한 번도 돌지 않았을 때는 bit shift right 이용하여 hash 1 byte의 상위 4 bit만 남긴다. loop를 돌았을 때는 and 연산을 이용하여 하위 4 bit만 남긴다.

그리고 4 bit 값을 10과 비교하여 ASCII '0' ~. '9'와 'A' ~ 'F'를 구분한다. ASCII로 변환하기 위해 기본적으로 0x30을 더해주고 10을 넘는 경우 추가로 0x7을 더해주어 알파벳으로 변환한다. 이제 계산되어진 ASCII 문자를 HASHBUF에 차례대로 저장한다. 이 과정이 loop 문이 2번 돌며 실행된다.

2.4 개발 시 경험한 문제점 및 해결점

- 책의 환경과 개발 환경이 맞지 않았던 문제

책의 코드는 윈도우 환경에서 Cygwin으로 작성되었다. 하지만 우리는 리눅스에서 개발을 하다보니 책의 코드를 참고 할 때 파일의 이름. 경로가 다르다는 문제가 발생하였다. 해당 파일을 리눅스에서는 어떻게 되어있는지 찾아 이름과 경로를 바꿔줘서 이런 문제

를 해결했다. 또한 바이너리 파일을 읽을 때도 윈도우에서는 O_BINARY 키워드를 사용하여 읽었으나 리눅스에서는 바이너리 파일과 텍스트 파일을 동일하게 다루기 때문에 O_BINARY 키워드를 사용할 필요가 없었다.

- 부트로더 메모리의 한계로 인한 메모리 부족 현상

우리가 제작하는 MIN64 OS의 부트로더는 크기가 512 byte로 제한되어 있다. 만약 부트로더의 크기가 고려하지 않아도 될만큼 크다면, hash 값 출력을 비롯한 KernelImage 위변조 검사 과정을 일련의 코드로 작성하여, 차례로 실행되게 작성하여도 그만이다. 하지만 부트로더의 크기는 512 byte로 제한되어 있기 때문에, 반복되는 코드를 최소한으로 줄여 byte를 절약해야 한다. 이를 위해 스택과 함수를 이용하여 과제의 요구사항을 구현했다. 또, 메모리를 절약하기 위해, 필수적이지 않은 문자열 출력 과정을 생략하고 문자열의 길이를 줄여주었다.

- 화면 출력 스위칭 문제

커널 이미지의 최상단에 hash 값을 넣고 메모리에 올려 OS를 실행시키니 'C Language Kernel Started~!! ' 문자열까지 나오지만 qemu 화면이 계속 스위칭되는 문제가 발생하였다. 한 화면은 정상출력되지만 다른 화면은 부트디스크를 읽는 과정에서 문제가 생겼다. 쓰레드가 두 개가 생기는게 아닐까 생각해서 커널이미지로 jmp하기 전에 cli 로 인터럽트가 발생하지 못하도록 막았다. 그래서 해결 할 수 있었다. 오랜시간동안 문제를 고민한 끝에 인터럽트 문제임을 깨달을 수 있었다.

- 바이너리 파일에서 값의 순서가 바뀌는 문제

우리가 예상하기를 바이너리 파일에서 0x1234 라는 숫자는 해당 주소에12 34 라고 써있기를 생각하고 있었다. 하지만 Hex Editor로 바이너리 파일을 열어보니 34 12 로 저장되어 있었다. 예전에 배웠던 리틀엔디안처럼 값의 저장 방식이 이렇게 되는 것을 알게 되었다.

- 스택 작동을 제대로 파악하지 못한 점

초기에는 단순히 push와 pop만을 사용하면 되는 것으로 오해하여 stack을 정상적으로 사용하지 못했다. 하지만 bp와 sp를 기준으로 하여 스택이 아래로 자라난다는 핵심을 파악하고, 손코딩을 하며 설계하여 스택의 작동을 구현하여 정상적으로 실행되는 것을 확인 할 수 있었다.

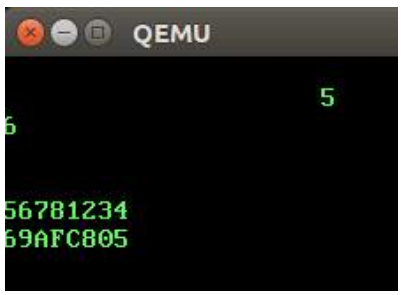
- 함수의 작동 방식을 제대로 이해하지 못한 점, 분기문의 어려움

반복되는 코드를 피하기 위해 많은 LABEL이 코드에 부여되고, jmp 명령이 수시로 일어난다. 따라서 코드의 진행 과정을 철저히 설계하지 않으면 제대로 동작하는 코드를 작성하기가 매우 어려웠다. 예를 들어 hash 값은 8 글자가, 2 번 출력되고, 각 글자가 10이 넘는지 여부를 검사해야 한다. 이를 코드 길이의 제한에 맞추어 어셈블리 코드로 구현하는 것은 생각하는 것보다 난이도가 매우 높은 구현이었다. 하지만 손 코딩으로 함수의 동작을 세세하게 표현하여 2 가지 함수로 나누어 실행되도록 구현했다. 정상적으로 동작하는 코드를 보니 매우 성취감이 들었다.

- ASSEMBLY 코드의 디버깅의 어려움

시행착오 끝에 철저히 설계를 하고 코드 작성에 돌입했지만, 정상적으로 화면에 결과가 출력되지 않았다. 결과적으로 보았을 때 문제는 대부분 레지스터를 중복 사용하여 값이 덮어씌워졌거나 잘못된 레지스터에 접근하였거나 등의 문제였는데 눈으로 레지스터의 값을 확인할 수 없으니 디버깅이 매우 어려웠다. C 언어가 어렵다고 하지만 여러 디버거들이 있고 printf()로 확인할 수 있다. 하지만 디버거 없이 디버깅을 하려니 ASSEMBLY 코드의 작성이 훨씬 어려웠다. 하지만 끝까지 포기하지 않고 노력하여 구현에 성공했다. 모든 과정을 ASSEMBLY로 작성했던 선배들이 대단하다고 느껴졌다.

인증 성공 시 출력결과



인증 실패 시 출력결과
(512 바이트를 오버에 따른 주석처리로 인해 출력 결과가 이렇게 되었습니다.)

3. 조원별 기여도

신정은 33%, 조재호 33%, 조한주 34%
팀원간의 협업을 위해 코드 리뷰 및 관리 용도로 깃허브를 이용한다.
팀 프로젝트 깃허브 주소 :
https://github.com/birdbirdgod/2018_OS