

# 5LD\_PRELOAD를 이용해 MBR locker 우회

## Assignment 1

김가람 단국대학교 산업보안학과



Dankook univ.

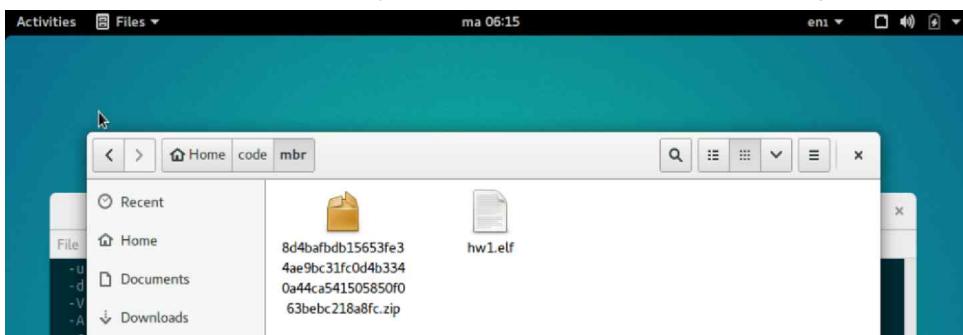


과목명 : 시큐어코딩  
담당교수 : 이승광  
학과 : 산업보안학과  
학번 : 32230324  
이름: 김가람  
제출일 : 2025/11/07

# I. 분석

## 1. 압축 해제 & 이름 변경

다운로드한 MBR locker의 압축을 해제한 후, 이름을 hw1.elf로 변경한다.



## 2. 실행

chmod 명령어를 통해 실행 권한을 부여한 후, user 권한으로 실행할 경우, 아래와 같이 root 권한으로 실행하여야 한다는 메시지가 출력되는 것을 볼 수 있다.

```
binary@binary-VirtualBox:~/code/mbr$ ./hw1.elf
You need to run as root user. Remember that running this binary can lead to LOSS OF THE CURRENT SYSTEM, run it in a virtual machine.
```

따라서 sudo 명령어를 이용해 root 권한으로 mbrlocker를 실행한 결과, /dev/sda에 payload를 쓰는 것에 성공하였고, 5초 후에 재부팅을 시작한다는 메시지가 출력된다. 5초 후 재부팅이 시작되지만 제대로 수행되지 않고 고양이 영상이 나오는 것을 볼 수 있다.

```
binary@binary-VirtualBox:~/code/mbr$ chmod 777 hw1.elf
binary@binary-VirtualBox:~/code/mbr$ sudo ./hw1.elf
Starting...
Writing payload inside /dev/sda
Payload successfully written! The machine will restart in 5 seconds
```



### 3. 어셈블리 코드

objdump -M intel -d hw1.elf

#### 1) <\_start>

- 4006ad : 해당 코드에서 rdi 레지스터에 0x40077d를 적재
- 4006b4 : \_\_libc\_start\_main@plt를 호출

따라서 main의 시작 주소는 0x40077d임을 알 수 있다. 따라서 해당 주소로 가 main의 흐름을 살펴보자 한다.

```
Disassembly of section .text:  
0000000000400690 <_start>:  
400690: 31 ed          xor    ebp,ebp  
400692: 49 89 d1        mov    r9,rdx  
400695: 5e              pop    rsi  
400696: 48 89 e2        mov    rdx,rsp  
400699: 48 83 e4 f0        and   rsp,0xfffffffffffffff0  
40069d: 50              push   rax  
40069e: 54              push   rsp  
40069f: 49 c7 c0 e0 08 40 00  mov   r8,0x4008e0  
4006ab: 48 c7 c1 70 08 40 00  mov   rcx,0x400870  
4006ad: 48 c7 c7 7d 07 40 00  mov   rdi,0x40077d  
4006b4: e8 77 ff ff ff        call   400630 <__libc_start_main@plt>  
4006b9: f4              hlt  
4006ba: 66 0f 1f 44 00 00  nop    WORD PTR [rax+rax*1+0x0]
```

#### 2) <main> : 0x40077d

- 40077d ~ 400781 : 스택 준비 (0x210 바이트 할당)
- 400788 ~ 40079f : 0x400a40에 저장된 데이터를 스택 버퍼(rbp-0x210)로 복사
- 4007a2 : geteuid()을 호출하여 UID 검사
- 4007a7 : test eax, eax를 통해 UID == 0(루트)인지 확인
- 4007a9 : root일 경우, 0x4007bf로 점프

```
000000000040077d <main>:  
40077d: 55              push   rbp  
40077e: 48 89 e5          mov    rbp,rsp  
400781: 48 81 ec 10 02 00 00  sub   rsp,0x210  
400788: 48 8d 95 f0 fd ff ff  lea    rdx,[rbp-0x210]  
40078f: be 40 0a 40 00        mov    esi,0x400a40  
400794: b8 40 00 00 00        mov    eax,0x40  
400799: 48 89 d7          mov    rdi,rdx  
40079c: 48 89 c1          mov    rcx,rax  
40079f: f3 48 a5          rep    movs QWORD PTR es:[rdi],QWORD PTR ds:[rsi]  
4007a2: e8 79 fe ff ff        call   400620 <geteuid@plt>  
4007a7: 85 c0              test   eax,eax  
4007a9: 74 14              je    4007bf <main+0x42>
```

#### 2-1) root가 아닌 경우

- 4007ab ~ 4007b0 : edi 레지스터에 0x400920을 적재 후 puts를 호출하여 문자열 출력
- 4007b5 ~ 4007ba : eax 레지스터에 1(종료) 적재 후 0x40085f(leave)로 점프

```
4007ab: bf 20 09 40 00        mov    edi,0x400920  
4007b0: e8 2b fe ff ff        call   4005e0 <puts@plt>  
4007b5: b8 01 00 00 00        mov    eax,0x1  
4007ba: e9 a0 00 00 00        jmp   40085f <main+0xe2>
```

gdb를 통해 확인한 결과(x/s 0x400920), 0x400920 위치의 값은 “You need to run as root user...”이라는 root 권한으로 실행하여야 한다는 뜻의 메시지가 있는 것을 확인할 수 있었다. 따라서 root가 아니면 경고문을 출력하고 종료한다는 것을 알 수 있다.

```
Reading symbols from hw1.elf...(no debugging symbols found)...done.  
(gdb) x/s 0x400920  
0x400920:      "You need to run as root user. Remember that running this binary can lead to LOSS OF  
THE CURRENT SYSTEM, run it in a virtual machine."  
(gdb) █
```

## 2-2) root인 경우

- 4007bf ~ 4007c4 : edi 레지스터에 0x4009a5 적재 후 puts를 호출하여 문자열 출력
- 4007c9 ~ 4007ce : edi 레지스터에 1 적재 후 sleep 호출하여 1초 대기

```
4007bf:      bf a5 09 40 00      mov    edi,0x4009a5  
4007c4:      e8 17 fe ff ff      call   4005e0 <puts@plt>  
4007c9:      bf 01 00 00 00      mov    edi,0x1  
4007ce:      e8 ad fe ff ff      call   400680 <sleep@plt>
```

gdb를 통해 확인한 결과(x/s 0x4009a5), 0x4009a5 위치의 값은 “Starting...”이라는 메시지가 있는 것을 확인할 수 있었다.

따라서 시작한다는 메시지를 출력 후 잠시 대기한다는 것을 알 수 있다.

```
(gdb) x/s 0x4009a5  
0x4009a5:      "Starting..."  
(gdb) █
```

- 4007d3 ~ 4007e2 : esi 레지스터에 0x4009b1, edi 레지스터에 0x4009ba, eax 레지스터에 0을 적재 후 printf를 호출하여 문자열 출력

```
4007d3:      be b1 09 40 00      mov    esi,0x4009b1  
4007d8:      bf ba 09 40 00      mov    edi,0x4009ba  
4007dd:      b8 00 00 00 00      mov    eax,0x0  
4007e2:      e8 29 fe ff ff      call   400610 <printf@plt>
```

gdb를 통해 확인한 결과, printf를 통해 Writing payload inside /dev/sda을 출력하고 있음을 확인할 수 있었다. 이를 통해 페이로드를 /dev/sda에 쓴다는 것을 알 수 있다.

```
(gdb) x/s 0x4009b1  
0x4009b1:      "/dev/sda"  
(gdb) x/s 0x4009ba  
0x4009ba:      "Writing payload inside %s\n"
```

- 4007e7 ~ 4007f6 : esi 레지스터에 0x4009d5, edi 레지스터에 0x4009b1을 적재 후 fopen을 호출 후 FILE\*(반환 값)은 rax에 저장하고, 스택에 저장

```
4007e7:      be d5 09 40 00      mov    esi,0x4009d5  
4007ec:      bf b1 09 40 00      mov    edi,0x4009b1  
4007f1:      e8 6a fe ff ff      call   400660 <fopen@plt>  
4007f6:      48 89 45 f8      mov    QWORD PTR [rbp-0x8],rax
```

gdb를 통해 확인한 결과, /dev/sda를 wb의 권한으로 열고 있음을 알 수 있었다.

```
(gdb) x/s 0x4009d5  
0x4009d5:      "wb"  
(gdb) x/s 0x4009b1  
0x4009b1:      "/dev/sda"
```

- 4007fa : 스택에 저장해둔 FILE\*을 rdx에 로드
- 4007fe : rax에 버퍼 주소 적재
- 400805 ~ 400815 : rcd 레지스터에 FILE\*, edx 레지스터에 1, esi 레지스터에 0x200(512), rdi 레지스터에 버퍼 주소를 적재 후 fwrite 호출

즉, 버퍼(0x210)로부터 512바이트를 FILE\*에 기록하는 것을 확인할 수 있다.

이때, 512바이트는 MBR의 크기이다.

```
4007fa: 48 8b 55 f8      mov    rdx,QWORD PTR [rbp-0x8]  
4007fe: 48 8d 85 f0 fd ff ff  lea    rax,[rbp-0x210]  
400805: 48 89 d1      mov    rcx,rdx  
400808: ba 01 00 00 00    mov    edx,0x1  
40080d: be 00 02 00 00    mov    esi,0x200  
400812: 48 89 c7      mov    rdi,rax  
400815: e8 56 fe ff ff    call   400670 <fwrite@plt>
```

- 40081a ~ 400837 : 파일 포인터 위치를 조정 후 파일 닫기
- 40083c ~ 400841 : edi 레지스터에 0x4009d8 적재 후 puts를 호출하여 문자열 출력
- 400846 ~ 40084b : edi 레지스터에 5 적재 후 sleep을 호출하여 5초 대기
- 400850 ~ 400855 : edi 레지스터에 0x400a1c 적재 후 system 호출
- 40085f ~ 500862 : eax 레지스터에 0(성공) 적재 후 스택 정리(프로그램 종료)

```
40081a: 48 8b 45 f8      mov    rax,QWORD PTR [rbp-0x8]  
40081e: ba 00 00 00 00    mov    edx,0x0  
400823: be 00 00 00 00    mov    esi,0x0  
400828: 48 89 c7      mov    rdi,rax  
40082b: e8 20 fe ff ff    call   400650 <fseek@plt>  
400830: 48 8b 45 f8      mov    rax,QWORD PTR [rbp-0x8]  
400834: 48 89 c7      mov    rdi,rax  
400837: e8 b4 fd ff ff    call   4005f0 <fclose@plt>  
40083c: bf d8 09 40 00    mov    edi,0x4009d8  
400841: e8 9a fd ff ff    call   4005e0 <puts@plt>  
400846: bf 05 00 00 00    mov    edi,0x5  
40084b: e8 30 fe ff ff    call   400680 <sleep@plt>  
400850: bf 1c 0a 40 00    mov    edi,0x400a1c  
400855: e8 a6 fd ff ff    call   400600 <system@plt>  
40085a: b8 00 00 00 00    mov    eax,0x0  
40085f: c9                leave  
400860: c3                ret
```

gdb를 통해 확인한 결과, 페이로드 쓰기에 성공했다는 메시지를 출력하고 5초 대기 후 system("reboot")를 통해 재부팅을 시도하는 것을 알 수 있었다.

```
(gdb) x/s 0x4009d8  
0x4009d8:      "Payload successfully written! The machine will restart in 5 seconds"  
(gdb) x/s 0x400a1c  
0x400a1c:      "reboot"  
(gdb) █
```

### 3) 결론

해당 elf 파일은 루트 권한 여부를 확인한 뒤 512바이트 페이로드를 /dev/sda에 기록(fopen + fwrite)하고 재부팅(system)한다. MBR의 크기인 512바이트만큼 블록 디바이스 파일인 /dev/sda를 덮어쓰는 동작으로 보아 MBR을 덮어쓰는 악성 행위를 수행하는 MBR locker로 추정된다.

## II . 코드 작성

LD\_PRELOAD를 이용해 앞서 설명한 악성 행위(MBR locker)를 우회하는 코드를 작성하고자 한다.

### · 전처리 & 헤더

코드 작성에 필요한 헤더 파일들을 불러온다. LD\_PRELOAD 기능 구현을 위해서는 dlfcn.h 헤더가 필수적이다. 이 헤더는 dlsym()과 RTLD\_NEXT를 제공한다. 또한 \_GNU\_SOURCE가 없으면 RTLD\_NEXT 심볼을 인식하지 못해 컴파일 에러가 발생할 수 있다. 그 외 헤더들은 파일 입출력(stdio.h, fcntl.h), 문자열 처리(string.h), 시스템 호출(unistd.h, sys/stat.h), 에러 처리(errno.h) 등의 기능을 제공한다.

```
#define __GNU_SOURCE
#include <dlfcn.h>           // dlsym, RTLD_NEXT
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <errno.h>
#include <sys/types.h>
```

### · 위험 경로 판별 함수 (is\_dangerous\_path)

is\_dangerous\_path() 함수는 프로그램이 접근하려는 파일 경로가 위험 경로인지 판별한다. 앞서 분석해본 결과에 따르면 악성 프로그램은 /dev/sda에 접근하여 데이터를 기록하려는 행위를 수행하고 있었으므로 이 함수에서는 해당 경로 및 그 하위 경로에 대한 접근을 모두 위험 행위로 간주한다.

경로가 존재하지 않거나 인자로 전달되지 않은 경우에는 0을 반환하고, 경로가 존재하면서 /dev/sda를 포함한 문자열이라면 1을 반환한다. 이를 위해 strstr() 함수를 사용하여 /dev/sda만 검사하는 것이 아니라 /dev/sda1, /dev/sda2와 같은 하위 경로까지도 포괄적으로 탐지할 수 있도록 하였다.

```
// 위험 경로 판별
// /dev/sda
static int is_dangerous_path(const char *path) {
    if (!path)
    {
        return 0;
    }

    if (strstr(path, "/dev/sda"))
    {
        return 1;
    }

    return 0;
}
```

#### · fopen 흑 (파일 열기 차단 및 리다이렉트)

원본 fopen() 함수 심볼을 가로채는 흑이다. 초기화 시 dlsym(RTLD\_NEXT, "fopen")으로 원본 fopen()의 주소를 획득해 orig\_fopen에 저장한다. 이후 흑 안에서 원본 함수를 호출하려면 이 포인터를 사용해야 한다. 만약 path가 /dev/sda로 시작하는 위험 경로라면 해당 경로를 여는 대신 사용자의 HOME 환경변수를 확인하여 더미 파일 경로를 대신 열고 fopen을 위험 경로에서 더미 파일 경로(\$HOME/code/mbr/dummy.img)로 리다이렉트 했다는 메시지를 출력한다. 이때, HOME이 없으면 임시 경로(/tmp/dummy.img)를 사용하고, 더미 파일이 존재하지 않으면 생성한다. 이 과정을 통해 프로그램은 정상적으로 파일을 연 것처럼 동작하지만 실제로는 더미 파일을 열게됨으로써 MBR을 손상시키는 것을 차단할 수 있다. 반면 안전한 경로에 대해서는 원본 fopen을 그대로 호출한다.

```
/* fopen : 프로그램이 위험 경로를 fopen하려 하면 더미 파일로 우회
typedef FILE *(*orig_fopen_t)(const char*, const char*);
static orig_fopen_t orig_fopen = NULL;

FILE *fopen(const char *path, const char *mode) // 가짜 fopen
{
    if (!orig_fopen)
    {
        orig_fopen = (orig_fopen_t)dlsym(RTLD_NEXT, "fopen"); // 원본 심볼 획득
    }

    if (path && is_dangerous_path(path)) // 위험 대상이면 더미 파일로 우회
    {
        const char *home = getenv("HOME");
        static char dummy[512];

        if (home)
        {
            snprintf(dummy, sizeof(dummy), "%s/code/mbr/dummy.img", home);
        }
        else
        {
            snprintf(dummy, sizeof(dummy), "/tmp/dummy.img");
        }

        // 더미 파일 없으면 생성
        int fd = open(dummy, O_RDWR | O_CREAT, 0600);
        if (fd >= 0)
        {
            close(fd);
        }

        printf("[SAFE] fopen redirect: %s -> %s\n", path, dummy);
        // 더미 파일 반환
        return orig_fopen(dummy, mode);
    }
}

// 안전 경로면 원본 fopen
return orig_fopen(path, mode);
}
```

### · fwrite 흑 (위험 쓰기 차단)

원본 fwrite() 함수 심볼을 가로채는 흑이다. 초기화 시 dlsym(RTLD\_NEXT, "fwrite")로 원본 fwrite()의 주소를 획득해 orig\_write에 저장한다. 이후 파일 디스크립터를 얻고 /proc/self/fd/<파일디스크립터> 심볼릭 링크를 readlink()로 읽는다. 이를 통해 프로그램이 어떤 경로에 데이터를 쓰려는지 파악할 수 있다. 추출된 경로가 위험 경로로 판단될 경우, fwrite를 성공적으로 차단했다는 메시지와 함께 쓰기를 차단한다. 반면 안전 경로에 대해서는 원본 fwrite 함수를 호출한다. 추가로 이 흑은 앞서 구현된 fopen 흑과 연계되어 동작한다. 따라서 fopen 흑이 정상적으로 이뤄져 위험 경로가 더미 경로로 리다이렉트된 경우에는 fwrite에서 추출되는 경로가 더미 경로로 나타나므로 차단이 발생하지 않고 실제 쓰기는 더미 파일로 수행된다.

```
// fwrite : 위험 경로일 경우 쓰기 차단
[typedef size_t (*orig_fwrite_t)(const void*, size_t, size_t, FILE*);
static orig_fwrite_t orig_fwrite = NULL;

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream) // 가짜 fwrite
{
    if (!orig_fwrite)
    {
        orig_fwrite = (orig_fwrite_t)dlsym(RTLD_NEXT, "fwrite"); // 원본 심볼 획득
    }

    // 총 바이트 수
    size_t total = size * nmemb;

    // 대상 경로 알아내기 위해 /proc/self/fd/<fd> 따라감
    char pathbuf[512] = {0}; // 실제 경로 담을 버퍼
    if (stream)
    {
        int fd = fileno(stream);
        if (fd >= 0)
        {
            char link[64];
            snprintf(link, sizeof(link), "/proc/self/fd/%d", fd);
            ssize_t r = readlink(link, pathbuf, sizeof(pathbuf)-1);
            if (r>0) pathbuf[r]=0;
        }
    }

    // 위험 경로면 쓰기 차단
    if (pathbuf[0] && is_dangerous_path(pathbuf))
    {
        printf("[SAFE] Blocked fwrite to %s (len=%zu)\n", pathbuf, total);
        errno = EACCES;
        return 0;
    }

    // 안전 경로면 원본 fwrite
    return orig_fwrite(ptr, size, nmemb, stream);
}
```

### · system 흑 (reboot 차단)

원본 system() 함수 심볼을 가로채는 흑이다. 초기화 시 dlsym(RTLD\_NEXT, "system")로 원본 system()의 주소를 획득해 orig\_system에 저장한다. command의 값이 NULL인 경우 원본 systme()을 호출하고, reboot가 포함되어 있는 경우 위험 명령어를 차단했다는 메시지와 함께 재부팅을 차단한다. 반면 다른 명령어인 경우 원본 system()을 호출한다.

```

// system : reboot 무력화
typedef int (*orig_system_t)(const char*);
static orig_system_t orig_system = NULL;

int system(const char *command) // 가짜 system
{
    if (!orig_system)
    {
        orig_system = (orig_system_t)dlsym(RTLD_NEXT, "system"); // 원본 심볼 획득
    }

    if (!command) // NULL 처리
    {
        return orig_system(command);
    }

    // 재부팅 차단
    if (strstr(command, "reboot"))
    {
        printf("[SAFE] Blocked dangerous system() call: %s\n", command);
        errno = EPERM; // 권한 없음
        return -1; // 실패 반환
    }

    // 나머지는 원본
    return orig_system(command);
}

```

### III . 결과

#### 1. 컴파일 및 실행

작성한 코드를 컴파일하고 실행한 결과, fopen 호출이 위험 경로(/dev/sda) 대신 더미 경로(~/code/mbr/dummy.img)로 리다이렉트되었음을 확인하였다. 이후 프로그램은 더미 파일에 공격 페이로드를 fwrite로 기록하려 시도했지만 앞선 fopen 리다이렉션으로 인해 더미 파일에 기록이 수행되었다. 또한 재부팅을 시도했으나 system() 혹이 이를 차단하여 재부팅이 수행되지 않았음을 확인하였다.

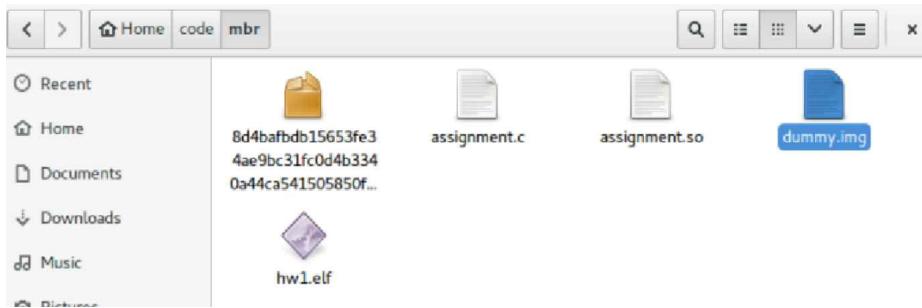
```

binary@binary-VirtualBox:~/code/mbr$ gcc -shared -fPIC -O2 -pthread -o assignment.so assignment.c -l
dl
binary@binary-VirtualBox:~/code/mbr$ sudo LD_PRELOAD=$PWD/assignment.so ./hw1.elf
[sudo] password for binary:
Starting...
Writing payload inside /dev/sda
[SAFE] fopen redirect: /dev/sda -> /home/binary/code/mbr/dummy.img
Payload successfully written! The machine will restart in 5 seconds
[SAFE] Blocked dangerous system() call: reboot
binary@binary-VirtualBox:~/code/mbr$ 

```

#### 2. 공격 페이로드 분석

~/code/mbr/ 디렉터리를 확인한 결과 dummy.img 파일이 생성되어 있음을 확인하였다. 앞서 구현한 코드의 fopen() 혹은 /dev/sda 같은 위험 경로에 대한 열기 요청을 더미 경로로 리다이렉션하므로 프로그램이 수행한 fwrite()는 MBR이 아닌 fwrite가 dummy.img 파일에 기록되었다. 따라서 공격 페이로드는 MBR이 아닌 dummy.img 파일에 남아 있을 것이다.



· stat -c “%n: %s bytes” dummy.img

파일 크기 : 512bytes

· file dummy.img

DOS/MBR boot sector

· hexdump -C dummy.img

offset 0x1FE ~ 0x1FF를 보면 0x55 0xAA가 존재하는데 이는 MBR 부트 시그니처이다.

```
binary@binary-VirtualBox:~/code/mbr$ stat -c "%n: %s bytes" dummy.img
dummy.img: 512 bytes
binary@binary-VirtualBox:~/code/mbr$ file dummy.img
dummy.img: DOS/MBR boot sector
binary@binary-VirtualBox:~/code/mbr$ hexdump -C dummy.img
00000000  0e 1f 0e 07 fc b9 14 00  bf bb 7d 0f 31 96 31 f0  |.....].1.1.|_
00000010  c1 c6 47 ab e2 f5 b8 13  00 cd 10 68 00 a0 07 6a  |.....h...j|
00000020  04 6a 00 bd 80 02 31 ff  b8 7e 00 31 c9 49 f3 aa  |.j....1..~.I..|
00000030  68 00 50 5f b1 05 57 01  ef f7 dd e8 9c 00 5f 83  |h.P...W.....|
00000040  c7 18 e2 f2 81 ef 68 1b  7d 5e e8 a5 00 e8 9f 08  |[...h.]^.....|
00000050  58 5a a8 01 74 04 f7 dd  f7 d2 f7 dd 40 52 50 29  |XZ..t.....@RP)|
00000060  d7 e8 8b 00 b1 05 e8 4a  00 e2 fb 81 c7 b4 13 e8  |.....J.....|
00000070  a1 00 83 c7 14 e8 9b 00  83 c7 24 e8 95 00 83 c7  |.....$.....|
00000080  18 e8 8f 00 b1 14 bb bb  7d 8b 3f 58 50 c1 e0 04  |.....].?XP...|
00000090  29 c7 31 c0 81 ff 00 e6  77 08 be 9f 7d 53 e8 12  |).1....W...]S..|
000000a0  00 5b 83 c3 02 e2 e2 31  c0 99 b1 02 b4 86 cd 15  |[.....1.....|
000000b0  e9 73 ff 31 db ac 93 ac  92 ad 92 83 fa 01 74 09  |.s.1.....t.|
000000c0  01 d7 89 da e8 03 00 eb  ef c3 51 57 89 d9 f3 aa  |.....QW....|
000000d0  5f 81 c7 40 01 4a 75 f3  59 c3 51 b1 05 6a 28 58  |..@.JU.Y.Q..j(X|
000000e0  6a 18 5b 6a 0c 5a e8 e1  ff 04 04 e2 f3 59 c3 e8  |j.[.j.Z.....Y..|
000000f0  00 00 ad 01 c7 31 c0 ac  91 ac 93 ac 99 92 ac 57  |[.....1.....W|
00000100  52 e8 c6 ff 5a f8 00 ea  08 80 c3 08 81 c7 fc 04  |R...Z.....|
00000110  e2 ed c3 57 56 e8 d7 ff  5e 5f c3 00 fb 03 48 48  |...WV...^.....HH|
00000120  00 0c f6 02 48 40 59 10  fb 03 38 38 3c 38 0f 04  |...H@Y..._88<8..|
00000130  28 28 00 10 f1 03 28 20  19 10 19 04 e2 28 ec 01  |((....(.....(|
00000140  00 08 00 e4 04 1c f6 01  00 04 0f e4 f5 1c fb 01  |.....|
00000150  00 08 41 dc 04 2c f6 01  00 04 00 d0 e1 00 f6 00  |...A.....|
00000160  f6 00 f6 04 f6 04 fb 04  00 00 00 1c fb 00 f6 04  |.....|
00000170  f6 04 fb 04 00 00 00 00  00 00 00 dc 13 00 00 04  |.....|
00000180  fb 04 fb 04 fb 04  fb 04 fb 00 f6 f4 fa 04  |.....|
00000190  f1 01 00 00 00 02 08 0c  00 08 fb 01 08 04 19 04  |.....|
000001a0  0f 00 00 00 00 00 00 00  00 0c ec 04 fb 04 00 00  |.....|
000001b0  00 f4 f5 00 00 04 00 04  fb 01 00 00 00 00 00 00 00  |.....|
000001c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 00 00  |.....|
*
000001f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 55 aa  |.....U..|
00000200
binary@binary-VirtualBox:~/code/mbr$
```

### 3. 결론

ELF 어셈블리 코드 분석에서 확인한 동작(/dev/sda에 512바이트로 쓰기 시도)와 추출된 공격 페이로드(dummy.img)에 대한 분석(512 바이트, 종료 시그니처 0x55AA)을 종합하면 해당 프로그램은 MBR을 512바이트 페이로드로 덮어쓰려 시도하는 악성 행위를 수행하는 MBR locker임을 알 수 있다.