ksmbd 버그 헌팅 프로젝트

개인 프로젝트 보고서 중간 제출

서울여자대학교 정보보호학과 육은서

시스템보안 정성훈 교수님

제출일: 2025년 10월 22일

1. Linux ksmbd SMB 서버

Ksmbd란 리눅스 커널 내에서 SMB3 프로토콜을 구현하여 네트워크 파일 공유를 지원하는 서버이다. 기존에는 Samba라는 windows와 파일 공유를 위한 user space 데몬 프로세스가 존재하였다. 아직 Samba가 널리 사용되고는 있지만 ksmbd는 Samba와 다르게 kernel space에서 동작하는 SMB 서버이기에 데이터 복사가 아닌 파일 시스템에 직접 접근할 수 있고 속도가 훨씬 빠르다는 이점이 있다. 임베디드 환경에서 도입하고 있다.

SMB는 윈도우에서 채택한 핵심 네트워킹 프로토콜이다. 보통 파일 공유를 위해 사용된다. SMB 1.0 같은 경우, 이터널블루라는 치명적 취약점으로 인해 현재는 사용하지 않고 SMB 2.0와 가상화를 위한 SMB 3.0를 같이 혼용중이다.

Windows의 프로토콜로 채택되고 Linux는 이 Windows와의 호환을 위해 뒤늦게 구현되었다. 앞서 언급하였지만, Samba를 통해 사용하였다가 최근 ksmbd가 추가되었다. Windows는 핵심 프로토콜이 SMB이기 때문에 단순 파일 공유뿐 아니라 다른 소스, 기능과도 연관이 많이 되어있다. 때문에 비교적 attack surface가 많다 할 수 있다. 반면 ksmbd는 고성능 파일 전송에 집중하였기 때문에 보다 더 컴팩트하다.

본 보고서에서는 kernel 단의 코드 리뷰를 진행하고 커널 디버깅 실습 진행을 하며 커널 수준 취약점에 대해 기술할 예정이다.

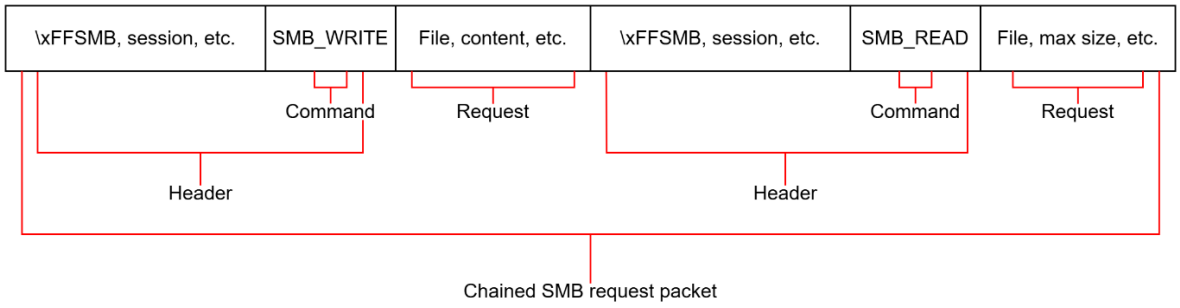
---

2. SMB 프로토콜 포맷

SMB가 프로토콜이기도 하고 이후에 나올 취약점을 설명하려면, Connection과 Session의 정의를 짚고 넘어가야 한다.

	정의
연결(Connection)	클라이언트와 서버 간의 통신 채널을 의미한다. SMB의 경우 TCP 포트 445를 통해 socket이 생성되며 이 socket을 일컫는다.
세션(Session)	그리고 그 socket에 누가(user) 인증하였고 요청하였는지 식별할 수 있는 리소스를 말한다. 클라이언트가 SMB_SESSION_SETUP으로 인증하게 되면, 이 세션을 식별하기 위한 SessionID가 발급된다.

SMB는 서버 리소스를 과도하게 사용되는 것을 방지하기 위해 Credit System을 도입하였다. 단순 연결만으로 요청을 수행하는 것이 아니라, 세션마다 credit만큼의 작업 한도가 있다. 이 credit이 0에 도달하면 명령을 처리할 수 없다.



또한, packet과 request/command 개념을 구분하는 것이 중요하다. Packet은 네트워크로 전송되는 한 묶음을 뜻한다. 무작정 하나의 요청==하나의 패킷으로 통용되지는 않는다. SMB 2.0부터는 command chaining이 도입되어 여러 개의 request가 하나의 packet에 담겨 전송된다.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
ProtocolId																															
StructureSize																CreditCharge															
(ChannelSequence,Reserved)/Status																															
Command																CreditRequest/CreditResponse															
Flags																															
NextCommand																															
MessageId																															
...																															

AsyncId
…
SessionId
…
Signature
…
…
…

SMB2 packet header 부분이다. 이중에서 chaining packet인지 확인하는 부분은 Nextcommand이다. 이부분이 0이 아닌 양수 N이라면 N바이트 오프셋 뒤에 다음 request가 있다는 뜻이고, 0이라면 NextCommand가 없다는 뜻이다. 이후 설명할 OOB read 취약점은 이에 대한 검증이 부족한 경우이다.

---

3. ZDI-23-979 분석 (NULL 포인터 역참조)

CVE ID	CVE-2023-3866
CVSS	5.9, AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:N/A:H
AFFECTED VENDORS	Linux
AFFECTED PRODUCTS	Kernel

해당 취약점은 SMB request를 연속적으로 처리하는 세션 과정에서 발생한다. Chained SMB request packet을 받을 때 세션 검사를 request마다 수행하지 않는다.

```
static void __handle_ksmbd_work(struct ksmbd_work *work,
                                struct ksmbd_conn *conn)
{
        u16 command = 0;
        int rc;
```

```
	// [snip] (initialize buffers)

	if (conn->ops->check_user_session) {
		rc = conn->ops->check_user_session(work);

		// if rc != 0 goto send (auth failed)
		if (rc < 0) {
			command = conn->ops->get_cmd_val(work);
			conn->ops->set_rsp_status(work,
					STATUS_USER_SESSION_DELETED);
			goto send;
		} else if (rc > 0) {
			rc = conn->ops->get_ksmbd_tcon(work);
			if (rc < 0) {
				conn->ops->set_rsp_status(work,
					STATUS_NETWORK_NAME_DELETED);
				goto send;
			}
		}
	}

	do {
		rc = __process_request(work, conn, &command);
		if (rc == SERVER_HANDLER_ABORT)
			break;

	    // [snip] (set SMB credits)
	} while (is_chained_smb2_message(work));

	if (work->send_no_response)
		return;

send:
	// [snip] (send response)
```

```
}
```

매 패킷을 연결할 때마다 위 __handle_ksmbd_work 함수가 호출된다. 이때 chaining packet 또한 `if (conn -> ops -> check_user_session )`을 통해 단 한 번의 세션 검사만 수행한다. 이후 do-while문에서 체이닝된 각 request를 처리한다.

1. 세션이 필요없는 SMB2_ECHO 먼저 요청
2. 이 다음 세션이 필요한 SMB2_WRITE 요청

위와 같이 chained packet을 구성하면 1번에서 통과한 if문에 의해 session 자리는 여전히 NULL일 것이다. 하지만 do-while문에서 두번째 request를 처리할 때 이 session을 역참조하게 되면서 패닉이 일어난다.

---

4. ZDI-23-980 분석 (OOB read)

CVE ID	CVE-2023-3865
CVSS	7.1, AV:N/AC:H/PR:L/UI:N/S:C/C:L/I:N/A:H
AFFECTED VENDORS	Linux
AFFECTED PRODUCTS	Kernel

이번 취약점의 경우, network-based SMB request 처리를 제대로 하지 않아 kernel information leak이 발생하는 취약점이다.

Root cause는 ksmbd_smb2_check_message이다. 여기서 hdr->NextCommand에 대한 길이 검사가 제대로 되어 있지 않다.

```
int ksmbd_smb2_check_message(struct ksmbd_work *work)
{
        struct smb2_pdu *pdu = ksmbd_req_buf_next(work);
        struct smb2_hdr *hdr = &pdu->hdr;
        int command;
        __u32 clc_len;  /* calculated length */
        __u32 len = get_rfc1002_len(work->request_buf); //원래는 이렇게 len 설정
```

```
        if (le32_to_cpu(hdr->NextCommand) > 0)
                len = le32_to_cpu(hdr->NextCommand); //여기서 overwrite
        //그리고 이에 대한 유효성 검사가 없음
        else if (work->next_smb2_rcv_hdr_off)
                len -= work->next_smb2_rcv_hdr_off;

        // [snip] check flag in header

        if (hdr->StructureSize != SMB2_HEADER_STRUCTURE_SIZE) {
                // [snip] return error
        }

        command = le16_to_cpu(hdr->Command);
        // [snip] check if command is valid

        if (smb2_req_struct_sizes[command] != pdu->StructureSize2) {
                // [snip] return error (with exceptions)
        }

        if (smb2_calc_size(hdr, &clc_len)) {
                // [snip] return error (with exceptions)
        }

        if (len != clc_len) {
                // [snip] return error (with exceptions)
        }

validate_credit:
        // [snip] irrelevant credit check

        return 0;
}
```

Smb2_hdr 구조체는 아래와 같다.

```
hdr->StructureSize == 64
pdu->StructureSize2 == smb2_req_struct_sizes[command]   // SMB2_WRITE: 49, SMB2_ECHO: 4
hdr->NextCommand == pdu->StructureSize2 + hdr->StructureSize   // SMB_ECHO
hdr->NextCommand == hdr->DataOffset + hdr->Length   // SMB_WRITE
```

다시 설명하자면 첫번쨰 hdr->StructureSize는 64이다(헤더 구조의 크기) 이때 hdr->NextCommand의 값을 아주 큰 값(0x10000)으로 설정한다면 len 값이 overwrite되면서 뒤따라 오는 함수가 조작된 값만큼 커널 메모리를 읽을 수 있다.

이 취약점은 두 가지 방법으로 악용할 수 있다. 첫번째는 SMB_ECHO 요청을 활용해서 인증 없이 익스플로잇 할 수 있는 방법이다. 두번쨰는 SMB_WRITE 요청을 활용하는 방법이다. 후자는 인증이 필요하지만, ECHO 요청은 인증 없이도 가능하다. 다만 후자가 hdr->Length를 조작하여 최대 65536 바이트를 leak할 수 있어서 더 악용될 가능성이 높다.

```
struct smb2_echo_req {
        struct smb2_hdr hdr;
        __le16 StructureSize;     /* Must be 4 */
        __u16   Reserved;
} __packed;
```

ECHO의 경우 위 구조체에 맞춰서 패킷을 보내면 2바이트 누출이 가능하다. StructureSize에는 p16(4)로 넣고, hdr.NextCommand에는 sizeof(smb2_echo_requ->hdr)+smb2_echo_req->StructureSize로 요청을 보낸다. 이러면 원래 패킷 총 크기는 66 바이트인데 68로 설정되어 2바이트를 leak할 수 있다.

```
static int smb2_get_data_area_len(unsigned int *off, unsigned int *len,
                                  struct smb2_hdr *hdr)
{
        int ret = 0;
```

```
        *off = 0;
        *len = 0;

        switch (hdr->Command) {
        // [snip] not reached
        case SMB2_WRITE:
                if (((struct smb2_write_req *)hdr)->DataOffset ||
                    ((struct smb2_write_req *)hdr)->Length) {
                        *off = max_t(unsigned int,
                                     le16_to_cpu(((struct smb2_write_req *)hdr)-
>DataOffset),
                                     offsetof(struct smb2_write_req, Buffer));
                        *len = le32_to_cpu(((struct smb2_write_req *)hdr)->Length);
                        break;
                }

                *off = le16_to_cpu(((struct smb2_write_req *)hdr)-
>WriteChannelInfoOffset);
                *len = le16_to_cpu(((struct smb2_write_req *)hdr)-
>WriteChannelInfoLength);
                break;
        // [snip] not reached
        default:
                // [snip] not reached
        }

        // [snip] return error if offset > 4096

        return ret;
}
```

위 코드에서 볼 수 있듯이 SMB_WRITE request의 경우, hdr->Length와 hdr->NextCommand 변수를 설정할 수 있다.

```
struct smb2_write_req {
        struct smb2_hdr hdr;
        __le16 StructureSize; /* Must be 49 */
        __le16 DataOffset; /* offset from start of SMB2 header to write data */
        __le32 Length;
        __le64 Offset;
        __u64   PersistentFileId; /* opaque endianness */
        __u64   VolatileFileId; /* opaque endianness */
        __le32 Channel; /* MBZ unless SMB3.02 or later */
        __le32 RemainingBytes;
        __le16 WriteChannelInfoOffset;
        __le16 WriteChannelInfoLength;
        __le32 Flags;
        __u8    Buffer[];
} __packed;
```

위 요청 포맷을 보면 알겠지만 메모리 내용을 저장할 파일 write 권한이 이미지 있어야 한다(일종의 인증) 이게 있어야 위 포맷의 PersistentFileId를 채울 수 있다. Id가 있는 파일에 메모리 info를 옮겨 적는 것이 목적이다.

Length에 원래의 파일 크기보다 크도록 설정해준다. Clc_len 검사는 위의 ECHO와 동일하게 우회할 수 있고, 이러면 Dataoffset부터 Length만큼 읽어서 OOB read를 달성할 수 있다.

---

5. 실습 환경 구축

커널 디버깅을 하기 위해 취약한 버전의 커널을 가져와 이미지로 컴파일을 해야 한다.

리눅스 커널 이미지는 컴파일 했을 때 생기는 파일로, 부팅 과정에서 메모리에 로드된다. 커널 이미지 종류는 다양하지만 대표적으로 아래 두 가지가 있다.

vmlinux : 압축되지 않은 리눅스 커널 실행 파일이다. ELF 형식이며 디버깅 심볼이 담겨 있어서 디버깅에 용이하다.

bzImage : 1번보다 더 큰 커널 이미지를 지원하기 위한 형식이다. 이는 압축된 이미지 형이며 커널을 부팅할 때 사용한다.

이러한 커널 이미지는 커널 소스에서 make 명령을 통해 컴파일하여 생성할 수 있다. 이후 QEMU라는 에뮬레이터를 활용하여 빌드를 진행한다.

1. 패키지설치 및 커널 소스 코드 다운
2. 커널 이미지 빌드
3. QEMU 실행을 위한 shell 파일
4. RootFS 빌드

```
mkdir -p ~/kernel-build && cd ~/kernel-build
# 예시: 실제 취약 버전으로 바꿔서 wget 하자
wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.x.y.tar.xz
tar xf linux-6.3.9.tar.xz
cd linux-6.3.9
```

```
cp /boot/config-$(uname -r) .config
make olddefconfig
make menuconfig
make -j8
```

Qemu로 필요한 ksmbd부분만 설치해주고 GDB로 원격 붙을 수 있게 shell 파일을 만들었다.

```
#!/bin/bash

set -e

KDIR=~/kernel-build/linux-6.3.9
KERNEL=$KDIR/arch/x86_64/boot/bzImage
VMLINUX=$KDIR/vmlinux
ROOTFS=~/kernel-build/rootfs.img

qemu-system-x86_64 ₩
    -kernel $KERNEL ₩
```

```
    -hda $ROOTFS ₩
    -append "root=/dev/sda rw console=ttyS0 nokaslr" ₩
    -m 2G ₩
    -smp 2 ₩
    -nographic ₩
    -net nic,model=e1000 ₩
    -net user,hostfwd=tcp::4445-:445 ₩
    -no-reboot ₩
    -s ₩
    -S

# -s: GDB 서버를 localhost:1234에서 시작
# -S: 부팅 시작 시 CPU를 정지 상태로 시작 (GDB 연결 대기)
```

터미널 창을 하나 더 열어서(terminal 2) gdb 디버깅을 붙인다.

```
gdb ~/kernel-build/linux-6.3.9/vmlinux
(gdb) target remote :1234
(gdb) break smb2_echo
(gdb) break ksmbd_smb2_check_message
(gdb) continue
```

PoC 진행하면서 ksmbd.mountd 데몬이 필요하였는데 이를 빌드하는 과정에서 잦은 오류가 생겨 PoC 실습을 끝마치지 못했다. 본 보고서에서 SMB 프로토콜과 ksmbd의 두 가지 취약점에 대해 코드 레벨에서의 root cause를 이해하였다. 기말 제출로는 QEMU 환경 구축을 완성하고 커널 퍼저를 활용하여 버그 헌팅 프로젝트를 진행할 예정이다.

---

6. 참고문헌

V., B. (2023, August 4). *Unleashing ksmbd: crafting remote exploits for the Linux kernel*. Pwning.tech. https://pwning.tech/ksmbd/

Zero Day Initiative. (2023, July 12). *Linux Kernel ksmbd Out-Of-Bounds Read Information*

Disclosure Vulnerability. https://www.zerodayinitiative.com/advisories/ZDI-23-980/

Zero Day Initiative. (2023, July 12). *Linux Kernel ksmbd Use-After-Free Remote Code Execution Vulnerability*. https://www.zerodayinitiative.com/advisories/ZDI-23-979/