Analysis of pipeline and it's vulnerability based on DirtyPipe(CVE-2022-0847)

ABSTRACT

우리는 모든 유저들에게 강력한 편의성을 제공하는 파이프라인이 메모리에 접근하는 기능이기에, 만약 이 기능에 취약한 점이 존재한다면 아주 강력한 공격 수단이 될 것이라는 생각을 했다. 따라서 주제를 선정하기 위한 조사를 진행했으며, DirtyPipe(CVE-2022-0847)를 발견하였고, 우리는 이 취약점에 기반을 두어 Pipe 코드를 분석하는 것을 주제로 선정하였다.

우리는 파이프가 어떻게 구현되고 동작하는지, DirtyPipe 취약점이 왜 발생 하는가와 어떻게 공격에 이용되는지, 취약점을 막기 위해서는 코드를 어떻게 변화시켜야 되는지를 중점적으로 분석하였으며, 이를 Ubuntu 환경에서 실험하는 과정 또한 함께 진행하였다.

1. INTRODUCTION

- Reason for Selecting topic

In Linux-based operating systems, the pipeline function ('|') in the terminal provides users with tremendous convenience. This is because it connects the output of the first command to the input of the next command when executing two or more commands sequentially. A typical use case is to find a specific character 'kim' in the text data of a file, such as "cat file.txt | grep kim". In this example, the cat command will fetch file.txt and connect/store the data loaded in the page cache to the buffer, and pass it to the grep command as the input value.

Here, we noticed the following two points.

・Pipes are a very convenient feature that is available to all users.

・If there is a vulnerability in the pipeline that accesses memory, it will be a very powerful attack vector.

Therefore, we searched for actual vulnerabilities related to pipes, and found the "CVE-2022-0847" document. We selected the analysis of pipes centered on this vulnerability as the topic of this project.

- What we want to do in this analysis

As the main subject of the assignment is the analysis of the Linux kernel, we first analyze what a pipeline is and how it is implemented. After that, we will analyze what the vulnerability "CVE-2022-0847", which is also the reason for the topic selection, is, its cause, and the attack process. Finally, we want to analyze the patch for the vulnerability, apply it, and check if the attack is blocked.

- 선정 이유

리눅스 기반의 OS들을 사용하는 것에 있어서, 터미널의 파이프라인 기능('|')은 사용자에게 엄청난 편의성을 제공해 준다. 두 개 이상의 명령어를 순차적으로 실행할 때, 앞 명령어의 출력을 뒤의 명령어의 입력으로 제공하여 연결해 주기 때문이다. 대표적인 사용 예로는 "cat file.txt | grep kim" 과 같이 파일의 출력에서 특정 문자를 찾아내는 것이 있다. 이 예시에서는 cat 명령어가 file.txt를 가져와 페이지 캐시에 로드되어 있는 데이터들을 버퍼에 연결/저장, grep 명령어의 입력 값으로 넘겨주게 된다.

우리는 여기서 다음 두 가지 부분에 주목하게 되었다.

"파이프는 매우 편리한 기능으로 모든 사용자에게 제공되는 기능이다"

"메모리에 접근하는 파이프라인에서 취약점이 존재한다면 아주 강력한 공격수단이 될 것이다"

따라서 우리는 실제로 파이프와 관련된 취약점이 존재하는지를 검색하였고, "CVE-2022-0847" 문서를 발견하여 해당 취약점을 중심으로 파이프를 분석하는 것을 이번 프로젝트의 주제로 선정하였다.

- 본 분석에서 하고자 하는 것

리눅스 커널에 대한 분석이 과제의 대주제인 만큼, 우선 파이프라인이 어떤 것인지와 그 구현이 어떻게 되어있는지를 분석하고자 한다. 그 이후에 주제선정의

이유이기도 한 “CVE-2022-0847”이 어떤 취약점이며, 그 원인과 공격과정을 분석할 것이다. 최종적으로는 해당 취약점의 패치를 분석하여 적용하고 공격이 막히는 지를 확인해보고자 한다.

2. BACKGROUND

- 시스템 콜

시스템 콜은 운영 체제의 기능을 호출하기 위한 인터페이스입니다. 사용자 프로그램은 시스템 콜을 사용하여 파일 시스템에 접근하고, I/O를 수행하고, 프로세스를 관리하는 등의 작업을 수행할 수 있습니다.
시스템 콜은 사용자 모드에서 커널 모드로의 전환을 통해 수행됩니다. 사용자 프로그램은 시스템 콜을 호출하기 위해 특수한 명령어를 사용합니다. 이 명령어는 CPU에 의해 인터럽트 신호를 발생시키고, 커널은 이 신호를 처리하여 시스템 콜을 수행합니다.
파이프라인 기능 또한 시스템 콜로 구현되어 쉘 프로그램이 이를 사용하는 것으로, 사용자에게 제공됩니다.

- 파일시스템 - 구조와 파일디스크립터

유닉스 시스템에서는 파일, 디렉토리, 소켓, 파이프 등 모든 객체들을 파일로 관리합니다. 그리고 시스템 내부적으로 현재 오픈 되어 있는 파일들의 목록을 테이블 형태로 관리하여 각 파일에 인덱스 값(음이 아닌 정수 값)을 부여합니다. 이 값이 바로 ‘파일 디스크립터’이며, 프로세스는 파일에 접근이 필요한 경우 이러한 파일 디스크립터를 이용하여 접근하게 됩니다. 윈도우의 핸들(Handle)과 유사한 개념이라 볼 수 있습니다.

- 파일시스템 - 페이지캐시

리눅스는 파일 I/O의 성능 향상을 위해 페이지 캐시라는 메모리 영역을 만들어서 사용합니다. 한번 읽은 파일의 내용을 페이지 캐시에 저장해 두었다가 다시 한 번 동일한 파일에 대한 접근이 발생하면 디스크에서 읽지 않고 페이지 캐시에서 읽습니다. 파일의 내용을 수정하는 경우에도 마찬가지로 페이지 캐시의 데이터를 수정하며, 변경된 사항은 일정 조건에 따라 디스크로 동기화(Write-Back) 됩니다.

3. PIPELINE

파이프란, 프로세스간 통신을 위해 사용되는 IPC(Inter Process Communication) 기법 중 하나이며, 프로세스 간 단방향 통신을 지원합니다.

파이프는 시스템 콜로 정의되어 사용되고, 실제 작동은 모든 프로세스에 대응 가능하도록 커널에서 만들어져 기동합니다. 구조와 기능 구현에 관한 코드들은 “/include/linux/pipe_fs_i.h”, “/fs/pipe.c”, “/fs/splice.c” 에 있으며, 커널에서 생성되고 작동하는 코드는 “/lib/iov_iter.c” 에 있습니다.

다음 내용들은 순서대로 파이프라인의 특징과 작동과정을 설명하는 절, 파이프 구조와 동작에 관련하여 구현된 소스코드를 분석하는 절, 커널에서 파이프의 생성을 분석하는 절입니다. 이 때 분석에 사용된 소스코드들의 커널 버전은 취약점이 발견된 버전인 linux kernel 5.8 버전을 중심으로 분석하였습니다.

a. Feature and Operation Process

앞서 설명한 것처럼 파이프는 둘 이상의 프로세스 간의 연결을 지원해주는 기능으로, 한 프로세스의 출력이 다른 프로세스의 표준입력이 됩니다. 즉 IPC(Inter-Process Communication)의 하나로 볼 수 있습니다.

또한 파이프는 다음과 같은 특징들을 가집니다.

・단방향 통신

하나의 프로세스는 파이프버퍼에 write를 하고, 다른 하나의 프로세스는 파이프에서 read를 합니다. 즉 write를 하는 주체와 read를 하는 주체가 해당 시점에는 고정적으로 단방향 통신만이 가능합니다.

・생성 프로세스와 모든 하위 프로세스가 공유 가능

・한 프로세스는 write, 한 프로세스는 read를 하기 때문에, Read-Write Sync. Problem이 존재합니다.

Read-Write Sync. Problem이 존재하기 때문에 Write 전(버퍼가 비어있는 상태) 읽기를 시도한다면, 기록 완료까지 대기를 하는 등, wait_queue와 같은 동기화 방법으로 이를 해결합니다.

・프로세스의 Open file table에서 사용가능한 첫 두 개의 위치를 찾고 이를 파이프의 읽기 끝 / 쓰기 끝으로 할당합니다.

[4]와 [5]에서 확인할 수 있듯이 하나의 프로세스에서 만들어진 pipe 객체는 fork()를 통해서 부모프로세스와 자식 프로세스 간에 공유가 가능합니다. 때문에 실제 리눅스에서는 모든 프로세스의 부모 프로세스인 init 프로세스에서 kernel 내부에 pipe 객체를 만들어서 이를 공유하고 사용하게 됩니다. 이를 그림으로 나타내면 다음 두 그림과 같습니다.

[pipe process 그림]

[kernel process - parent-child 그림]

b. Source Code Analysis regarding structure and operation

본 절은 /include/linux/pipe_fs_i.h 파일과 /fs/pipe.c 파일을 분석하여 파이프가 어떤 구조로 이루어져 있고 어떤 함수들로 구성되어 있는지를 설명하는 절입니다.

(i) 매크로 및 변수

(a) _LINUX_PIPE_FS_I_H

#ifndef 와 함께 헤더의 중복 로드를 방지하기 위한 테크닉으로 사용된다.

(b) PIPE_DEF_BUFFERS

#define PIPE_DEF_BUFFERS 16

파이프의 기본 버퍼 크기를 16으로 정의한다.

(c) PIPE_BUF_FLAGS

파이프버퍼의 메모리 관리, 데이터 전송방식, read()/write() 함수의 동작 제어를 위한 플래그 값을 정의한다.

i) #define PIPE_BUF_FLAG_LRU 0x01

해당 페이지가 LRU(Least Recently Used) 캐시에서 관리되고 있을 경우 설정되는 값.

ii) #define PIPE_BUF_FLAG_ATOMIC 0x02

해당 페이지가 atomically mapping(단일 CPU 명령) 되었을 때, 파이프버퍼의 설정되는 값.

iii) #define PIPE_BUF_FLAG_GIFT 0x04

해당 페이지가 gift page(다른 프로세스가 파이프버퍼에 제공한 페이지) 일 때 설정되는 값.

iv) #define PIPE_BUF_FLAG_PACKET 0x08

해당 페이지를 패킷으로 읽어야 할 때, 설정되는 값.

v) #define PIPE_BUF_FLAG_CAN_MERGE 0x10

파이프 버퍼가 다른 버퍼와 병합이 가능할 때, 설정되는 값, 본 플래그 값이 설정된 경우 커널은 메모리 절약을 위해 여러 개의 작은 버퍼를 더 큰 단일 버퍼로 병합하는 등의 행위가 가능하다.

vi) #define PIPE_BUF_FLAG_WHOLE 0x20

read()함수가 전체 버퍼를 읽어야 할 때, 설정되는 플래그 값. 본 플래그 값이 설정된 경우, read() 함수는 해당 버퍼의 전체 내용을 무조건적으로 읽어야하고, 일부만 읽고 함수를 종료하는 것이 불가능하다.

vii) #define PIPE_BUF_FLAG_LOSS 0x40

CONFIG_WATCH_QUEUE와 함께 사용되어 프로세스가 파일시스템 이벤트를 모니터링 하는 것이 가능하다면, 메모리 로스가 발생했을 때, 본 플래그를 세팅한다.

(d) PIPE_SIZE

#define PIPE_SIZE PAGE_SIZE

실제 메모리 할당의 길이이다.

(e) 외부변수

/fs/pipe.c에서 정의되는 변수들로, 본 헤더파일을 include한 모든 소스코드가 본 변수를 사용할 수 있도록 전역화 해주는 선언이다.

i) extern unsigned int pipe_max_size;

루트가 아닌 사용자가 파이프를 확장할 수 있는 최대 크기이다.

ii) extern unsigned long pipe_user_pages_soft;

사용자 당 할당 가능한 최대 페이지 수로 soft는 시스템의 기본 값과 일치한다.

iii) extern unsigned long pipe_user_pages_hard;

사용자 당 할당 가능한 최대 페이지 수로 hard는 기본적으로 설정되지 않는다.

(ii) 구조체

(a) struct pipe_buffer

파이프 버퍼를 나타내는 구조체로 다음의 멤버 변수를 포함한다.

i) struct page *page;

데이터가 저장된 페이지를 가리키는 포인터다.

ii) unsigned int offset, len;

페이지 내에서 데이터의 시작부를 나타내는 변수 offset과 데이터의 길이를 나타내는 변수 len 이다.

iii) const struct pipe_buf_operations *ops;

이 버퍼와 연결된 작업을 정의하는 pipe_buf_operations 구조체에 대한 포인터다.

iv) unsinged int flags;

파이프버퍼의 플래그를 나타내는 값이다.

v) unsinged long private;

연관된 작업(ops)에 의해 사용되는 개인 데이터를 저장하는 변수이다.

(b) struct pipe_inode_info

파이프의 i-node 정보를 나타내는 구조체로 파이프에 대한 정보 저장과 동작제어를 위한 함수를 제공한다.

i) struct mutex mutex;

파이프에 대한 동시성 접근 제어를 위한 mutex.

ii) wait_queue_head_t rd_wait, wr_wait;

파이프에 데이터가 없을 때, reader가 기다리기 위한 wait_queue와 파이프에 데이터가 가득 찼을 때, writer가 기다리기 위한 wait_queue.

iii) unsigned int head;

파이프 버퍼의 head 위치를 저장하는 변수로, 다음에 읽을 데이터의 위치를 가리킨다.

iv) unsigned int tail;

파이프 버퍼의 tail 위치를 저장하는 변수로, 다음에 데이터를 쓸 위치를 가리킨다.

v) unsigned int max_usage;

파이프 버퍼의 최대 사용량을 저장하는 변수로, 파이프에 저장할 수 있는 데이터의 최대 크기를 결정한다.

vi) unsigned int ring_size;

파이프 버퍼의 ring_size를 저장하는 변수로 파이프에 사용할 수 있는 버퍼 개수를 결정한다.

vii) bool note_loss;

(CONFIG_WATCH_QUEUE 구성이 활성화 되어 모니터링이 가능 할 때) 다음 read() 호출 시 데이터 손실 메시지를 삽입해야 함을 나타내는 플래그이다.

viii) unsigned int nr_accounted;

원래 사용자 프로세스가 사용하는 파이프 버퍼의 수를 저장하는 변수이다.

ix) unsigned int readers;

파이프에서 데이터를 읽는 프로세스의 수를 저장하는 변수이다.

x) unsigned int writers;

파이프에서 데이터를 쓰는 프로세스의 수를 저장하는 변수이다.

xi) unsigned int files;

파이프를 참조하는 파일의 개수이다.

xii) unsigned int r_counter;

reader 카운터

xiii) unsigned int w_counter;

writer 카운터

xiv) struct page *tmp_page;

임시적으로 사용되는 페이지이다.

xv) struct fasync_struct *fasync_readers;

reader 측의 fasync 구조체로 비동기 I/O 이벤트의 감지에 사용된다.

xvi) struct fasync_struct *fasync_writers;

writer 측의 fasync 구조체로 비동기 I/O 이벤트의 감지에 사용된다.

xvii) struct pipe_buffer *bufs;

파이프버퍼의 배열

xviii) struct user_struct *user;

파이프를 생성한 사용자 프로세스

xix) struct watch_queue *watch_queue;

(CONFIG_WATCH_QUEUE 구성 옵션이 활성화 되어 모니터링이 가능할 때) 파이프가 watch_queue로 사용되는 경우 watch queue 관련 정보를 저장하는 포인터다.

(c) struct pipe_buf_operations

파이프버퍼에 대한 작업을 정의하는 구조체로 confirm / release / try_steal / get 의 4개의 멤버함수를 포함한다. 이 중 confirm 함수는 try_steal 함수의 실행 전에 반드시 실행되어야 한다. 이는 대상 버퍼의 데이터가 유효하고 사용 가능한지를 판단해야 하기 때문이다.

i) int (*confirm)(struct pipe_inode_info *, struct pipe_buffer *);

파이프버퍼의 데이터가 유효하고 사용 가능한지를 확인하는 함수이다. 파일 시스템에 속하는 페이지를 포함하는 파이프버퍼의 경우 I/O 완료를 기다려야 할 수 도 있다.

유효성 판단 결과 유효한 경우 0을 반환한다.

ii) void (*release)(struct pipe_inode_info *, struct pipe_buffer *);

파이프버퍼의 내용이 완전히 소비되었을 때 호출되는 함수로, 버퍼의 메모리 자원을 해제하는 작업을 수행한다.

iii) bool (*try_steal)(struct pipe_inode_info *, struct pipe_buffer *);

파이프버퍼의 소유권을 독점적 획득하려고 시도하는 함수로 성공 시 버퍼의 내용을 잠그고, 함수 호출자가 완전히 소유하게 한다. 이후 페이지를 다른 매핑으로 전송가능하다. 독적 성공 시 true를 반환한다.

iv) bool (*get)(struct pipe_inode_info *, struct pipe_buffer *);

파이프 버퍼에 대한 참조를 얻는 함수로 버퍼의 사용 횟수를 증가시킨다. 성공하면 true를 반환한다.

(d) 외부구조체

i) extern const struct pipe_buf_operations nosteal_pipe_buf_ops;

/fs/splice.c에서 정의되며, 소켓 등을 위한 pipe_buffer_operations을 정의한다.

(iii) static inline 함수

호출되는 함수가 아닌 매크로처럼 사용된 부분에 스크립트가 붙여 넣어지는 함수로, 함수 호출 스택 등을 아낄 수 있는 구조이다.

(a) bool pipe_empty(unsigned int head, unsigned int tail)

파이프가 비어 있다면 true를 반환하는 함수

(b) unsigned int pipe_occupancy(unsigned int head, unsigned int tail)

파이프가 사용 중인 slot의 개수를 반환하는 함수

(c) bool pipe_full(unsigned int head, unsigned int tail, unsigned int limit)
파이프가 가득 차 있다면 true를 반환하는 함수
(d) unsigned int pipe_space_for_user(unsigned int head, unsigned int tail, sturuct pipe_inode_info *pipe)
파이의 slot 중 user-space로 사용 가능한 개수를 반환하는 함수
(e) __must_check bool pipe_buf_get(struct pipe_inode_info *pipe, struct pipe_buffer *buf)
파이프버퍼에 대한 참조를 반환하는 함수
(f) void pipe_buf_release(sturct pipe_inode_info *pipe, struct pipe_buffer *buf)
파이프버퍼의 메모리 자원을 해제하는 작업을 수행
(g) int pipe_buf_confirm(struct pipe_inode_info *pipe, struct pipe_buffer *buf)
파이프버퍼의 데이터가 유효하고 사용 가능한지를 확인하는 함수로, 유효한 경우 0을 반환한다.
(h) bool pipe_buf_try_steal(struct pipe_inode_info *pipe, struct pipe_buffer *buf)
파이프 버퍼의 소유권을 획들하려고 시도하는 함수로 성공 시 true를 반환한다.

(iv) 파이프 함수
(a) lock
파이프의 락 기능 함수들
i) void pipe_lock(struct pipe_inode_info *)
ii) void pipe_unlock(struct pipe_inode_info *)
iii) void pipe_double_lock(struct pipe_inode_info *, struct pipe_inode_info *)

(b) wait 및 정보관리
i) void pipe_wait(struct pipe_inode_info *pipe);
파이프 이벤트를 wait하는 함수로 atomic하게 작동한다.
ii) struct pipe_inode_info *alloc_pipe_info(void);
새로운 pipe_inode_info 구조체를 할당하는 함수로 파이프에 대한 정보 저장과 동작을 관리하는 역할을 수행한다.
iii) void free_pipe_info(struct pipe_inode_info *);
pipe_inode_info 구조체를 해제한다.

(c) 파이프버퍼 동작함수
i) bool generic_pipe_buf_get(struct pipe_inode_info *, struct pipe_buffer *);
파이프에서 사용가능한 버퍼를 가져오는 함수로, 데이터를 쓰기 전에 호출된다.
ii) bool generic_pipe_buf_try_steal(struct pipe_inode_info *, struct pipe_buffer *);
파이프 버퍼를 독점하기 위한 함수로 다른 프로세스의 사용 전에 파이프 버퍼를 가져오는데 사용된다.
iii) void generic_pipe_buf_release(struct pipe_inode_info *, struct pipe_buffer *);
파이프 버퍼를 해제하는 함수로 파이프에 데이터를 쓰기 완료 후, 해당 버퍼를 반환하는데 사용한다.

(d) 파이프 크기 조정 및 파일 관리 함수
파이프 파일의 F_SETPIPE_SZ 및 F_GET_PIPE_SZ 명령을 처리하기 위한 함수들이다.
i) int pipe_resize_ring(struct pipe_inode_info *pipe, unsigned int nr_slots);
파이프의 링 버퍼 크기를 조절하는 함수이다.
ii) long pipe_fcntl(struct file *, unsigned int, unsigned long arg);
파이프 파일의 F_SETPIPE_SZ, F_GET_PIPE_SZ 명령을 직접 처리하는 함수이다.
iii) struct pipe_inode_info *get_pipe_info(struct file *file, bool for_splice);
struct file *file 객체에서 pipe_inode_info 구조체를 가져온다.
iv) int create_pipe_files(struct file **, int);
두 개의 파이프 파일을 생성한다.
v) unsigned int round_pipe_size(unsigned long size);
파이프 크기를 올림처리한다.

(e) 기타함수
CONFIG_WATCH_QUEUE가 활성화 되어 있을 때 정의, 사용되는 함수들이다.
i) unsigned long account_pipe_buffers(struct user_struct *user, unsigned long old, unsigned long new);
파이프 버퍼 사용량을 계산한다.
ii) bool too_many_pipe_buffers_soft(unsigned long user_bufs);
(soft) 사용자 프로세스가 너무 많은 파이프버퍼를 사용하고 있는지 확인하는 함수이다.
iii) bool too_many_pipe_buffers_hard(unsigned long user_bufs);
(hard) 사용자 프로세스가 너무 많은 파이프버퍼를 사용하고 있는지 확인하는 함수이다.
iv) bool pipe_is_unprivileged_user(void);
현재 프로세스가 권한이 없는 사용자인지 확인하는

함수이다.

(f) const struct file_operations_pipefifo_fops
/fs/pipe.c에서 정의 되는 함수 집합 구조체로 파일을 열고 읽고, 데이터를 작성하는 일련의 함수들이 포함되어 있고 각각에 대한 내용은 다음과 같다.
i) fifo_open
파일을 열 때 호출되는 함수입니다.
ii) no_llseek
파일 포인터를 이동하는 함수입니다.
iii) pipe_read
파일에서 데이터를 읽는 함수입니다.
iv) pipe_write
파일에 데이터를 쓰는 함수입니다.
v) pipe_poll
파일의 상태를 검사하는 함수입니다.
vi) pipe_ioctl
파일의 특성을 변경하는 함수입니다.
vii) pipe_release
파일을 닫을 때 호출되는 함수입니다.
viii) pipe_fasync
파일에 대한 파일 엑세스를 동기화하는 함수입니다.

c. Analyzing the creation and use of pipes in the kernel
본 절은 파이프가 동작하는 과정을 설명하고, 이 때 kernel 내부에 만들어지는 파이프가 어떤 로직으로 페이지캐시에 로드되고 파이프버퍼가 초기화 되는지를 설명합니다.

[4]와 [5], [kernel process - parent-child 그림]를 참고하여 "cat file.txt | grep kim" 명령의 실행 순서를 비교하여 보면 다음 그림과 같습니다.

[cat file.txt | grep kim 순서도]

grep 프로세스는 데이터를 읽어오는 read() 작업에서 cat 프로세스가 작동을 완료할 때까지 대기하게 됩니다.

즉 파이프로 연결된 명령어 중 제일 마지막에 실행되는 명령어가 pipe_read() 함수 실행 중 다른 프로세스의 파이프버퍼 write() 작업이 완료될 때까지 대기를 하는 것을 알 수 있습니다.
이 함수의 호출 순서를 보게 되면 다음과 같습니다.

[pipe_read() -> written = copy_page_to_iter() -> copy_page_to_iter_pipe()]

위 과정이 수행되면서 파이프버퍼가 copy_page_to_iter_pipe() 함수에 의해서 초기화과정을 거치는 경우가 존재합니다. 이 때 초기화 되는 정보를 확인해 보면 "ops, page, offset, len"의 작동 함수와 참조페이지와 offset, 길이 정보만이 초기화 되는 것을 확인할 수 있습니다.

[초기화 부분 code]

flag 초기화가 되지 않는다는 것을 확인할 수 있는데, 이는 이후에 설명할 CVE-2022-0847의 원인이 됩니다.

4. CVE-2022-0847

- 보고
2022-03-07, 보안연구원 Max Kellermann에 보고된 리눅스 커널 로컬권한상승 취약점입니다.
해당 취약점은 "DirtyPipe(CVE-2022-0847)"로 명명되었으며, 리눅스 파일 시스템 내 읽기 권한이 있는 모든 파일에 임의의 데이터를 쓸 수 있는 취약점입니다. 따라서 /etc/passwd와 같은 민감한 파일에 임의의 계정정보를 삽입하여 루트 권한의 계정을 생성할 수 있습니다.

본 취약점에 관한 타임라인은 다음과 같습니다.

Date	Note
2021-04-26	웹서버 액세스 로그에서 파일 손상 현상 최초 인지
2022-02-19	Linux 커널 버그로 인한 파일 손상으로, 악용 가능 취약점으로 판명
2022-02-20	버그 리포트, 익스플로잇 및 패치를 Linux 커널 보안팀에 공유
2022-02-21	Google Pixel 6에서도 버그 재현 확인, Android 보안팀에 버그리포트 공유
2022-02-23	버그가 수정된 Linux 커널 패치 릴리즈 (5.16.11, 5.15.25, 5.10.102)
2022-02-24	Google은 Android 커널에서 버그를 수정
2022-03-07	CVE-2022-0847 취약점 공개

본 취약점의 적용범위는 다음과 같습니다.

취약 버전	패치 버전
Linux Kernel 5.8 <= version < patch	Linux Kernel 5.16.11 5.15.25 5.10.102

- 공격과정 분석 - PoC 코드 분석

[전체 코드 사진]

공개된 PoC(Proof of Concept)를 기반으로 DirtyPipe 취약점 공격의 구현과정을 분석해 보겠습니다.

[명령 줄 인수 검증 코드]

첫 번째로는, 명령 줄 인수를 검증하는 코드로 취약점 공격을 위해 충족되어야하는 전제조건이 다음 4가지임을 알 수 있습니다.
• 오프셋(대상 파일에서 덮어쓸 데이터의 시작 위치)은 페이지 경계에 위치해선 안됨(PAGE_SIZE = 4069)
• 쓰기는 페이지 경계를 넘을 수 없음
• 대상 파일에 대한 읽기 권한이 있어야 함
• 쓰기는 파일 크기를 초과할 수 없음

[파이프 버퍼 생성 및 세팅 코드]

다음은 파이프를 생성하고 PIPE_BUF_FLAG_CAN_MERGE 플래그를 세팅하는 코드이며 다음 3단계로 나뉩니다.
1) pipe() 함수를 이용하여 파이프를 생성
2) 쓰기전용 파일디스크립터(p[1])를 이용하여 파이프버퍼를 임의의 데이터로 완전히 채우는 것으로, 파이프버퍼의 플래그를 PIPE_BUF_FLAG_CAN_MERGE로 세팅합니다.
3) 읽기전용 파일디스크립터(p[0])를 이용하여 파이프버퍼의 데이터를 모두 읽어들여 비웁니다. 이 때 파이프버퍼의 PIPE_BUF_FLAG_CAN_MERGE 플래그는 초기화되지 않고 유지됩니다.

[파이프버퍼 병합 코드]

다음은 파이프버퍼를 대상 파일의 페이지캐시와 병합하기 위해 사용되는 코드입니다. splice()함수는 인자로 전달받은 두 파일 디스크립터 간에 데이터를 이동하기 위해 사용되는 함수입니다. PoC에서는 대상 파일의 오프셋 직전 1바이트를 파이프로 이동시킵니다. 때문에 대상 파일의 데이터는 페이지캐시에 먼저 로드된 후, 파이프버퍼로 이동하게 됩니다. 이 때, PIPE_BUF_CAN_MERGE로 설정되어 있기 때문에 파이프버퍼는 대상 파일의 페이지캐시와 병합 처리됩니다.

[병합된 파이프버퍼로 데이터 덮어쓰는 코드]

따라서 이후 해당 파이프버퍼로 전송하는 데이터는 병합된 페이지캐시 영역을 덮어쓰게 됩니다.

단, 디스크를 직접적으로 건들지 않고, 페이지캐시 메모리를 변조하는 것이므로 임의로 캐시메모리를 삭제하거나 커널이 페이지캐시를 삭제하는 경우(메모리 부족으로 인한 회수 등)에는 변경된 사항이 이전으로 되돌려 집니다.

- PoC 테스트
공개된 PoC를 사용하여 /etc/passwd 파일에 루트권한이 없는 사용자 test를 root로 만들어보겠습니다. 테스트 환경은 다음과 같습니다.

환경	VMware Fusion pro ver. 13.0.2
OS	Ubuntu 64-bit ARM Server 20.04.5
kernel	Linux Kernel 5.8.0

우선 커널 버전을 확인하여 취약한 버전인지 확인합니다.

[uname -r]

다음으로 /etc/passwd 파일의 패스워드 필드에 작성할 데이터를 완성합니다. 본 테스트에서는 openssl을 사용하여 다음과 같이 만들었습니다.

algorithm	MD5
salt	linux
password	test

[openssl passwd -1 -salt linux test]

최종적으로 /etc/passwd 파일에 추가할 사용자 정보는 아래와 같습니다.

[payload]

위 과정이 성공적으로 수행되면, 다음과 같이

/etc/passwd 파일에 test 유저가 root 권한의 계정이 된 것을 확인할 수 있습니다.

[./exploit.out /etc/passwd 34 payload]

페이지캐시 메모리를 삭제 후, /etc/passwd 파일을 다시 확인해 보면 PoC를 통해 데이터를 변조하기 이전으로 되돌려진 것을 확인할 수 있습니다.

[echo 1 > /proc/sys/vm/drop_caches]

5. CAUSE ANALYSIS

- v5.8 원인-patch 분석

[취약점 패치]

CVE-2022-0847의 보고문서와 취약점 패치 내용을 보면 struct pipe_buffer 오브젝트를 새로운 내용으로 초기화 해주는 과정에서 flags 필드를 초기화하지 않게 되면서 발생한다고 언급한다. 즉 초기화 이전에 PIPE_BUF_FLAG_CAN_MERGE 플래그로 세팅되어 있었다면, 초기화 이후에도 해당 플래그가 유지되는 오류가 있다는 것이다.

따라서 해당 취약점은 struct pipe_buffer 오브젝트의 초기화 과정에 “buf->flags=0;” 코드 한 줄을 추가하는 것으로 해결된다.

[v5.8에서 해당 부분]
[v5.16.11에서 해당 부분]

실제로 커널 버전 5.8에서 위 패치를 적용하여 재구동한 결과 취약점이 작동하지 않는 것을 확인할 수 있었다.

[path 적용]
[PoC 실행]

- v5.4.247 과의 차이
본 취약점은 리눅스 강의에서 사용한 linux kernel 5.4.247 버전에서는 나타나지 않고, 오히려 더 진보된 버전이라고 할 수 있는 5.8 버전에서 나타난다. 때문에 우리는 더 발전된 버전에서 왜 이러한 문제가 생겼는지를 분석하였고 이에 대한 원인은 PIPE_BUF_FLAG의 세분화 과정을 거치면서 FLAG의 초기화를 수행하는 코드가 고려되지 않았거나 누락되었다고 생각할 수 있다.

실제로 /include/linux/pipe_fs_i.h 파일에 대해 5.4.247 버전과 5.8 버전에 대해 vimdiff를 했을 때의 결과는 다음과 같다.

[vimdiff /include/linux/pipe_fs_i.h 5.4.247 5.8]

이 때 가장 큰 차이는 선언된 PIPE_BUF_FLAG의 세분화 정도가 다른 것을 볼 수 있는데, 5.4.247 버전은 4개의 플래그를 사용하는 반면, 5.8 버전은 3개의 플래그가 추가된 것을 확인할 수 있다. 이 때, 추가 되면서 플래그를 초기화하는 코드가 고려되지 않았거나 누락되었다고 생각할 수 있다.
근거로서 5.4.247 버전에서는 “void pipe_buf_mark_unmergeable(struct pipe_buffer *buf);” 라는 함수가 존재하고, 여러 파이프가 동일한 백업페이지를 공유할 경우 반드시 사용되는 것을 확인 가능하다.

[5.4.247 /include/linux/pipe_fs_i.h pipe_buf_mark_unmergeable(struct pipe_buffer *buf)]

즉, 파이프버퍼의 사용성을 더 높이기 위한 업데이트에서 FLAG 초기화에 대한 부분이 고려되지 않았거나 누락되어 이러한 취약점이 발생하였다고 예상한다.

6. CONCLUSION

We analyzed the pipeline function based on the vulnerability, CVE-2022-0847, which can be a serious security threat, considering that the pipeline function, which provides strong convenience to all users, is a function that accesses memory.

In the kernel version that is the target of the vulnerability, pipes can use the PIPE_BUF_FLAG_CAN_MERGE flag, which indicates a state that can be merged with the page cache for flexible use of the pipe buffer. In this case, the vulnerability occurred due to a logical structure error that does not initialize the previous flag value in the initialization process of pipe use.

We found that the vulnerability applies to all versions from Linux kernel 5.8 to the patch, and it is a very serious security threat as it

can be exploited for privilege escalation.

To resolve the vulnerability, it is only necessary to add a single line of code to initialize the flag value in the pipe structure used in the kernel. Therefore, if you are using a vulnerable version of the kernel, we recommend applying the patch or updating to the kernel with the patch applied.

우리는 모든 유저들에게 강력한 편의성을 제공하는 파이프라인 기능이 메모리에 접근하는 기능임에 착안하여, 심각한 보안 위협이 될 수 있는 취약점, CVE-2022-0847에 기반을 두어 파이프라인을 기능을 분석하였다.

해당 취약점 대상인 커널버전에서 파이프는 파이프 버퍼의 유연한 사용을 위해 페이지 캐시와 병합 가능한 상태를 나타내는 PIPE_BUF_FLAG_CAN_MERGE를 사용할 수 있다. 이 때, 파이프 사용에서의 초기화 과정에 이전 Flag 값을 초기화 하지 않는 논리구조상의 오류가 있어 해당 취약점이 나타나게 되었다.

해당 취약점은 리눅스 커널 5.8부터 patch 이전까지의 모든 버전에 적용되는 취약점이며, 권한상승에 악용될 수 있는 취약점으로 매우 심각한 보안 위협이 된다는 사실을 알 수 있었다.

본 취약점의 해결을 위해서는 kernel에서 사용하는 파이프 구조체에서 flag 값을 초기화 해 주는 간단한 코드 한 줄 만을 추가해 주면 되기 때문에, 만약 취약한 버전의 커널을 사용하고 있다면, 해당 패치를 적용하거나 패치가 적용된 커널로의 업데이트를 추천한다.

REFERENCES

[1] The Dirty Pipe Vulnerability, Max Kellermann, https://dirtypipe.cm4all.com/

[2] CVE-2022-0847 Patch note, linux-kernel.vger.kernel.org archive mirror, https://lore.kernel.org/lkml/20220221100313.1504449-1-max.kellermann@ionos.com/

[3] CVE-2022-0847, AKB, https://attackerkb.com/topics/UwW7SVPaPv/cve-2022-0847/rapid7-analysis?referrer=blog

[4] pipe() System call, Kadam Patel, https://www.geeksforgeeks.org/pipe-system-call/

[5] C Program to Demonstrate fork() and pipe(), Kartik Ahuja, https://www.geeksforgeeks.org/c-program-demonstrate-fork-and-pipe/

[6] Linux kernel version 5.8, /include/linux/pipe_fs_i.h, bootlin, https://elixir.bootlin.com/linux/v5.8/source/include/linux/pipe_fs_i.h

[7] Linux kernel version 5.4.247, /include/linux/pipe_fs_i.h, bootlin, https://elixir.bootlin.com/linux/v5.4.247/source/include/linux/pipe_fs_i.h

[8] Linux kernel version 5.16.11, /include/linux/pipe_fs_i.h, bootlin, https://elixir.bootlin.com/linux/v5.16.11/source/include/linux/pipe_fs_i.h

[9] Linux kernel version 5.8, /fs/pipe.c, bootlin, https://elixir.bootlin.com/linux/v5.8/source/fs/pipe.c

[10] Linux kernel version 5.4.247, /fs/pipe.c, bootlin, https://elixir.bootlin.com/linux/v5.4.247/source/fs/pipe.c

[11] Linux kernel version 5.16.11, /fs/pipe.c, bootlin, https://elixir.bootlin.com/linux/v5.16.11/source/fs/pipe.c

[12] Linux kernel version 5.8, /fs/splice.c, bootlin, https://elixir.bootlin.com/linux/v5.8/source/fs/splice.c

[13] Linux kernel version 5.4.247, /fs/splice.c, bootlin, https://elixir.bootlin.com/linux/v5.4.247/source/fs/splice.c

[14] Linux kernel version 5.16.11, /fs/splice.c, bootlin, https://elixir.bootlin.com/linux/v5.16.11/source/fs/splice.c

[15] Linux Filesystems API Chapter 8. pipes API, https://archive.kernel.org/oldlinux/htmldocs/filesystems/index.html

[16] How DirtyPipe Linux Exploit Works and How to Respond, Uptycs Threat Research , https://www.uptycs.com/blog/dirtypipe-linux-exploit

[17] The Dirty pipe vulnerability: Overview,

detection, and remediation, Security Labs, https://securitylabs.datadoghq.com/articles/dirty-pipe-vulnerability-overview-and-remediation/

[18]