Linux Kernel KeyLogger

Dennis Cornwell and Brett Lesnau

***Abstract*—This paper presents an implementation of a loadable Linux kernel module which captures various forms of user input, including keyboard key activity and TTY device activity, and how to prepare that input to be consumed by a user-level process. Existing functionality provided by the kernel to receive keyboard notifications is explained to capture the simple use case where a physical keyboard is connected to a computer. More advanced techniques of capturing keyboard input are also explored such as intercepting function calls to insert new functionality at runtime. The resulting implementation of the keylogger is examined attempting to determine if it is a viable solution for the intended use cases.**

I. Introduction

Our general idea that we began exploring when thinking of ideas for this project was that we wanted to monitor or capture some type of user activity on a computer running Linux. This could be for record-keeping purposes, a more malicious snooping purpose, or to use the captured information to provide additional useful functionality.

We ended up creating a keylogger to keep track of user keyboard input. An interesting motivation for this was to spy on a user of a Linux machine without their knowledge. A less malicious use for this kernel module is to simply monitor user activity for a variety of reasons including record keeping and monitoring employees' computer use. It wasn't as simple as just registering for keyboard event notifications. When using a remote or virtual terminal like an SSH client, keyboard events are not sent. A method of intercepting write operations to TTY device file buffers was used to capture input in these scenarios

II. Background

Various other Linux key loggers exist with varying implementations. In general there are certain distinctions that separate how each of these keyloggers work. They run in either user-space or in the kernel and some only support listening to physical keyboard input while others also support capturing input from virtual or remote terminals.

LKL Linux KeyLogger [1] is a keylogger that runs as a user-space process. It monitors hardware port 0x60 specifically, which is reserved as a hardware keyboard port. Keyboard input is then translated to ASCII strings and written to a file.

THC-vlogger [2] is another Linux keylogger with many more advanced features when compared to LKL. It runs in the kernel and allows configuration to capture only administrator user keyboard input or only non-administrator user input. It can capture input from physical keyboards as well as remote and virtual terminal sessions. This keylogger is also smart enough to automatically detect password prompts so that snoopers can more easily find sensitive information such as passwords.

The developers who created the previously mentioned THC-vlogger have provided a great overview [3] of how keyboard drivers work in Linux along with many different approaches of how to capture keyboard input. These approaches include registering for a keyboard interrupt handler and hijacking various function calls. Hijacking function calls allows you to point a function call to a new function that you have created. You can even choose to still call the preexisting implementation of those functions within the newly defined functions so that existing functionality remains intact.

III. Design and Software Development

A. Overview

Our solution for capturing keyboard input includes two separate pieces of functionality; a loadable Linux kernel module and a user-level process. These two entities communicate and work together to get keyboard activity information out of the kernel and consumable by others on the outside.

The kernel module code is written in C and the user-level process is written in C++. All of the code was written against version 2.6.32.5 of the Linux kernel. All of the code for our project, as well as documentation and instructions describing how to use our code is available in our GitHub repository [A.1].

B. KeyLogger Kernel Module

We created a loadable kernel module named *keylogger*. This module's job is to capture any hardware keyboard or terminal keyboard activity and notify one or more registered user-level processes of any buffered keyboard input.

This keyboard input is currently captured using two different methods exposed by the Linux kernel. One involves using a well-defined interface to register for keyboard notifications and the other requires the *keylogger* module to intercept function calls which write data to TTY device files.

C. Kernel Registration

The kernel registration method of capturing keyboard input involves using a well-defined interface to register for

keyboard notifications. The *register_keyboard_notifier* and *unregister_keyboard_notifier* functions are exported to kernel code as symbols in *drivers\char\keyboard.c* and allow a kernel module to register and unregister for keyboard activity.

A *notifier_block* struct must be defined and passed in to register for this functionality. The same *notifier*_block struct is required to be passed when unregistering for keyboard notifications. The *notifier_block* contains a pointer to a function call which is invoked when keyboard keys are pressed or released. This function receives as parameters various pieces of information including the code of a specific keyboard key and whether or not that key was pressed or released.

These keyboard notifications will provide the kernel with information about specific keys that have been pressed or released, but does not directly provide the string representations of those keys. The kernel module needs to handle converting the key inputs into strings, including capitalizing characters manually if the *Shift* key is currently pressed while a different key is also pressed.

One limitation of this kernel registration method is that it only captures keyboard activity from a physical keyboard plugged into a machine or through certain virtualization environments like VNC. Keyboard input will also be captured in a non-terminal instance of the OS such as running an actual visual interface with a mouse and application windows. Keyboard activity when using remote or virtual terminal sessions such as SSH is not captured.

Pseudocode for Kernel Registration:

```
register_keyboard_notifier( notify )

notify( key_code, keyIsDown ) {
  key = key_code
  keyWasPressed = keyIsDown
  ...
}

...

unregister_keyboard_notifier( notify )
```

D. TTY Interception

The TTY interception method of capturing keyboard input involves intercepting a specific function call in the kernel which writes data to TTY device files. TTY device files are associated with and work with the data going in and out of UNIX terminals. In the early days of UNIX, these TTY devices were primarily associated with physical terminal connections to a computer, usually meaning a user was typing on a physical keyboard connected directly to a computer. Over time, the concept and usage of TTY devices has adapted to also support remote and virtual terminal sessions such as SSH.

The function that needs to be intercepted is the *receive_buf* function which is inside of the *tty_struct* struct defined in the kernel to represent a TTY device. The *receive_buf* function is passed a few parameters with information about actual characters output to a terminal including the TTY device associated with the terminal, the characters that were output, and the number of characters that were output. To keep existing functionality intact, the kernel module keeps track of the original *receive_buf* call and calls it inside of the new function call that we have defined. The original call is also restored when the kernel module is unloaded.

Similar to the kernel registration method mentioned previously, the TTY interception method is not able to capture all keyboard input at all times. Actual keyboard key presses and releases do not get written to TTY devices. Any string that is output to a terminal is what gets sent to the TTY device. This means that when only the *Shift* key is pressed, nothing gets written to the TTY device, but when *Shift* and *A* are pressed, a capital 'A' is written to the TTY device. This also means that when pressing keyboard keys in a non-terminal interface, that keyboard activity does not make its way to the TTY device files. The advantage of using the TTY interception method is that keyboard activity is captured when using remote or virtual terminal sessions.

Pseudocode for TTY Interception:

```
file = file_open( "/dev/tty0" )
tty = file->tty_data

old_receive_buf = tty->receive_buf
tty->receive_buf = new_receive_buf

new_receive_buf( tty, chars, charCount) {
  if( tty->charsFromKeyboard ) {
    ...
  }

  old_receive_buf( tty, chars, charCount )
}

...

tty->receive_buf = old_receive_buf
```

E. Module I/O

Module I/O in a similar fashion to that of a character device driver loaded as a kernel module. The module registers a set of I/O operations with a particular node under the */dev* directory. Since there are two keyboard capture implementations differing slightly on the type of input tracked, separate character devices are registered for TTY interception and key notifications. When read, each implementation provides its own stream of buffered characters to the reading application. On write, both implementations accept a process ID (PID) into a global list of registered processes. These processes are later looked up by their PID and notified when lines of text become available. It is good practice to unregister a process from the module when the process is ready to stop reading keys. While the module will simply silently fail if the process leaves memory, the module doesn't waste time and memory

buffering keys while no processes are registered. To unregister a process from listening for lines of key presses, the PID for a listening process may simply be written a second time into either of the device nodes for the module. The module will look up the PID and remove it from the list of registered listeners at this time.

F. Signal Notifications

As was mentioned in the previous section, listening user processes are notified of lines of key presses as they occur. This notification is delivered in the form of an IO signal posted against the registered process. The list of listening processes is iterated and each process's task struct is referenced by the PID supplied.

Pseudocode for Kernel Module Sending IO Signal to Keyreader Application:

```
vpid = find_vpid( registered_pids[i] )
task = pid_task( vpid, PIDTYPE_PID )
send_sig_info( SIGIO, siginfo, task )
```

Since signaling a process in this way doesn't provide a mechanism for supplying any data, this signal being raised simply serves as an indicator that new information is ready to read. The listening process in this case should react by reading the device file from the beginning to retrieve the new set of buffered data. It is important that a process prepare itself for handling IO signals prior to registering itself with the keylogger module since the IO signal notification will otherwise cause the process to die.

G. Special Keys

The methods used to intercept key presses flowing through the kernel simply intercept a series of events that occur when using the keyboard. Without consideration for special keys like delete and shift, the system simply appends another logged key to its current buffer. For accuracy though, it is important to define semantics for these special keys and modify the logged buffer as would be expected in any natural text input. For example, when the delete key is pressed, the last logged key should actually be removed instead of adding another to the end of the buffer. Enter is another important example as it signifies the end of a line of input and triggers notifications to registered listeners. Special keys are considered before each key press is appended to the buffer, allowing them to modify the buffer or overall state of the system in any way they see fit. In the case that a key is determined to be a special key, nothing is appended to the logged key buffer. It is also important to note that since the different methods for intercepting keys receive differing levels of detail about each key press, special key consideration must be defined separately for each implementation.

Pseudocode for Special Key Handling (Kernel Registration Method):

```
int handle_special_reg( int keycode, int down ) {
  switch ( keycode ) {
  ...
  case SHIFT_KEY:
    if ( down ) {
      keymap_for_lookup = SHIFTED_KEYMAP
    else {
      keymap_for_lookup = REGULAR_KEYMAP
    }
    return SPECIAL_KEY_HANDLED
  case BKSP_KEY:
    if ( down ) {
      remove_last_key_from_reg_buffer()
    }
    return SPECIAL_KEY_HANDLED

    ...
 }
```

Pseudocode for Special Key Handling (TTY Interception Method):

```
int handle_special_tty( int keycode ) {
  switch ( keycode ) {
  ...
  case SHIFT_KEY:
    // We don't need to handle this because TTY
    // gives us capitalization for free
  case BKSP_KEY:
    remove_last_key_from_tty_buffer()
    return SPECIAL_KEY_HANDLED

    ...
}
```

H. Keyreader Application

The keyreader packaged with the key logging module serves as a simple example of how to use the module for gathering key press information. It demonstrates many of the core concepts involved in effectively utilizing the functionality that the key logger module provides. The keyreader utilizes and shows the similarities of both the TTY and key registration interfaces provided in the two nodes prepared when installing the module. It displays a simple method for opening the appropriate node, gathering the current process's PID and writing it into the module, registering the application for keylogger kernel module notifications.

The application then demonstrates the method for handling the appropriate signal indicating that key press information is ready to be read. In the signal handler, the application reads the applicable device file to obtain the prepared key press information. The keyreader application also demonstrates a simple method for continuously multiplexing the IO it receives from both user input and the module.

Finally, the keyreader demonstrates the proper way to unregister from the module while cleaning up before exiting the application. This is accomplished by writing the keyreader process's PID into the module a second time, effectively toggling the registered process 'off'. This application serves as a bare minimum implementation of what can be accomplished with the keylogger module. Extensions to this implementation could include interpreting semantics of the keys logged or even going so far as to set up a local key server, sending logged keys from the local machine to remote hosts.

IV. Evaluation

While testing both the kernel registration and the TTY interception method of capturing keyboard input, you see mostly the same results. The results manifest in the text that is output from the *keyreader* user-level application. Both capture methods have their own specific key mapping defined so that the different pieces of capture information map to the same string for the same key. There are a couple of caveats to take note of and differences between the two capture methods though.

As mentioned in section II.C, the kernel registration capture method cannot capture all keyboard input at all times. It works well and captures all expected key presses when using a physical keyboard connected directly to a computer, but does not capture key presses in virtual or remote terminal sessions such as those connected using SSH. The kernel registration capture method can also work well when connected to a VM using certain virtualization clients such as VNC. These virtualization clients emulate using a real computer as best as they can and simulate real keyboard key presses for the virtualized operating system.

Similarly, as mentioned in section II.D, the TTY interception capture method cannot capture all keyboard input at all times. In an entirely terminal driven environment, this capture method should be able to capture keyboard key presses, but key presses cannot be captured in other situations outside of a terminal such as typing in a word processing program. This is because the TTY capture method does not know about actual individual key presses. It only knows about the strings that a user types into a terminal.

The kernel registration capture method also performs better in certain situations when both capture methods are able to capture something. One notable example is editing a file inside of a terminal editor such as Vim. TTY interception does not always result in very understandable output. Applications like Vim modify what you see in a terminal as you navigate and edit a file in a different way than normal terminal commands write to standard output. Since the TTY interception capture method relies on receiving what is output to the terminal, it is reliant on whatever Vim outputs to the terminal. This results in various character combinations that do not always have a defined string mapping inside of our *keylogger* module. The kernel registration method works much better under the same circumstances. Since this method receives the actual keyboard key presses, it is trivial to map those keys to a human-readable string.

As described above, both capture methods have their own advantages and disadvantages. Neither are able to receive key press information at all times, and at other times, they may receive information that is not easy to decipher and output for a human to read. It is best to use both capture methods simultaneously, so that you get the best of both worlds and can ignore some of the situations in which they fail.

V. Future Research

A. Network Transfer

As described in Section II.H, the *keyreader* application will receive keyboard activity by reading it out of a device file. The application then outputs that information to standard output in a terminal. Ideally, we would also be able to send the information across the network to a remote computer so that a user's keyboard activity can be monitored and recorded remotely.

B. Installing and Running Secretly

Currently, a user would be able to see that the *keylogger* module is installed and running as well as see that the *keyreader* application is running. A step toward making key logging less conspicuous would be to remove the requirement that a user process actively listen for module output while keys are being logged.. Instead of sending the keyboard key presses to a user-level application, they could be sent directly from the kernel module across the network to a remote computer. Taking it even one step further, the *keylogger* could be implemented directly into the kernel code instead of as a loadable kernel module. This obviously brings its own difficulties as it is difficult to give someone a modified kernel for them to use.

C. Capturing All TTY Device Activity

The TTY interception capture method is currently only capturing user input into a terminal, such as typing in a command. It is possible to see all information that is output to terminals, such as the results of a command. Capturing this information will result in a larger amount of information being sent to the *keyreader* application so this may present new problems to be solved. It would provide a much better way to determine what a user is doing though. For example, currently, we can only know that the *ls* command is used. If we were to capture this extra TTY information, would also be able to determine what the contents of the user's current directory were.

APPENDIX

A. *Linux Kernel KeyLogger GitHub Repository*

1. https://github.com/cornwe19/cse812-module

REFERENCES

[1] "LKL Linux KeyLogger" Internet: http://sourceforge.net/projects/lkl/ [Apr. 27, 2013]

[2] "THC-vlogger" Internet: http://www.thc.org/releases.php?q=vlogger [Apr. 27, 2013]

[3] "Writing Linux Kernel Keylogger" Internet: http://thc.org/papers/writing-linux-kernel-keylogger.txt [Apr. 27, 2013]

[4] "Writing device drivers in Linux: A brief tutorial" Internet: http://www.freesoftwaremagazine.com/articles/drivers_linux [Apr. 27, 2013]