

P1: System Calls

Prologue

You work for a top-secret shadow government organization dedicated to the rise of the Silurian overlords. You, as a faithful member of the Lizard Legion, are part of the team charged with improving data storage and handling, particularly tracking *metadata* – that is, data about data – within the organization’s computer systems. You have been tasked to build a coded message subsystem under the guise of process logging for kernels running in “Sky Skink”, the cloud computing system. Naturally, the Legion uses the superior Reptilian operating system distribution.

Overview

In this project, you will implement 3 system calls in Reptilian along with 3 static library functions that allow the system calls to be invoked from a C API. These custom system calls will *get* and *set* a custom process log level that will sit atop the standard Linux kernel’s diagnostic message logging system (**dmesg**) and allow processes to submit log entries along with a log level. If the log level for the message is more severe (lower than) the current log level, the message will be added to the kernel log. Log levels and names will correspond to those in the Linux kernel. We, as your benevolent lizard overlords, will provide a program that exercises and demonstrates the new calls. You create a short video to demonstrate your code. (Our masters will be most pleased.) You will submit the project via Canvas so as not to invite suspicion.

Table 1. Kernel Log Levels and Corresponding Process Log Levels

Kernel Level Name	Description	#	Process Level Name
KERN_EMERG	Emergency / Crash Imminent (no process logging)	0	PROC_OVERRIDE
KERN_ALERT	Immediate Action Required	1	PROC_ALERT
KERN_CRIT	Critical/Serious Failure Occurred	2	PROC_CRITICAL
KERN_ERR	Error Condition Occurred	3	PROC_ERROR
KERN_WARNING	Warning; recoverable, but may indicate problems	4	PROC_WARNING
KERN_NOTICE	Notable, but not serious (e.g., security events)	5	PROC_NOTICE
KERN_INFO	Informational (e.g. initialization / shutdown)	6	PROC_INFO
KERN_DEBUG	Debug messages	7	PROC_DEBUG

NOTE: Take snapshots of your VM! You will probably brick your machine at some point during this or other projects, and you will not want to start from scratch. No, seriously – take snapshots!

Structure

The project is broken into four main parts:

- 1) Create a kernel-wide *process log level* variable.
- 2) Create system calls that allow a process to *get* or *set* the *process log level* of the system.
- 3) Create system call that allows a process to add a process log message at a defined log level.
- 4) Create static library functions that allow the system calls to be invoked via a C API.



Figure 1: A system call invoked from a user program

While exact implementation may vary, the library functions must match the signatures laid out in this document, and the system calls must apply the security model properly. Logged messages are formatted “`$log_level_name [$executable, $pid]: $message`”, e.g.:

`PROC_ERROR [bacon_pancakes, 21]: Life is scary & dark. That is why we must find the light.`

NOTE: Your output must match the format exactly, including whitespace and semicolon location, failure to do will result in point deductions.

System Calls

The system will have a single, kernel-wide *process log level* which should initialize on boot in the kernel and must be stored persistently (until shutdown / reboot). The rules for logging are as follows:

- 1) The system-wide *process log level* should be initialized to 0 — i.e., override logging only.
- 2) Any process can read (get) the *process log level*.
- 3) Only a process running as the *superuser* may write (set) the *process log level*.
- 4) Log levels are values between 0–7. An invalid level results in call failure.
- 5) Any process may send a log message to the kernel.
- 6) If a message’s log level is higher than the *process log level*, the message is ignored.
- 7) If a message’s log level is lower than or equal to the *process log level*, the message will be logged.
- 8) Any successfully logged message should be logged at the corresponding kernel log level.
- 9) Messages must have a maximum length of 128 characters. Longer messages will be truncated.

System calls are called via `syscall(call_num, param1, param2)`. To log a message, the call should be `syscall(PROC_LOG_CALL, msg, level)`. *Call parameters are limited to two at most!*

Static Library

You will create a static library to invoke the system calls in a directory named **process_log**. This includes a header, **process_log.h** (prototypes and level symbols), and a static library file named **libprocess_log.a**. You will also need to provide a Makefile for the library. All sources must be contained within the **process_log** directory. Please note, these filenames must match exactly!

You will create a tarred gzip file of the **process_log** directory with name **process_log.tar.gz**. When testing, we will decompress the archive, enter the **process_log** directory, and build. All functions enumerated below must be made available by including “**process_log.h**”. See *Submission* for details.

Library Functions

`int get_proc_log_level()`

Invokes system call which reads system-wide process log level. Returns the process log level.

`int set_proc_log_level(int new_level)`

Invokes system call which attempts to change the system-wide process log level to **new_level**. Returns **new_level** on success, and **-1** otherwise.

`int proc_log_message(int level, char *message)`

Invokes system call to log a message for this process. Note that the order of parameters differs from the system call. Returns **-1** for invalid log level, and **level** otherwise.

Harness Functions

In addition to the standard library functions, you will implement testing harness functions (also in the library code). The testing harness functions are used to verify security of the system calls from the system library (and are required for full credit on this assignment). We will call these functions to retrieve the information needed to make a system call. We will then call the system call within our own program. This ensures that no security checks are being done in the user-level library.

System call parameter retrieval data should be returned as a pointer to an **int** array of 2-4 values that can be used to make the system call (which can be cast from other types). It has this format:

```
{ call_number, num_parameters [, parameter1] [, parameter2] }
```

e.g.: { 42, 2, 867, 5309 } → syscall(42, 867, 5309)

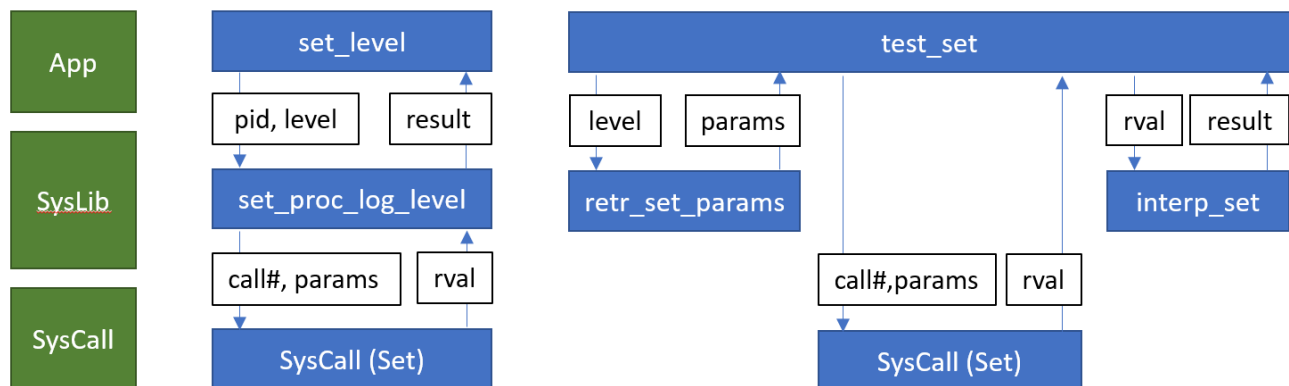


Figure 2: Harness functions can directly invoke system calls without the library functions.

These test harness elements must be implemented *as part of your library* to test your security model:

```
#define PROC_LOG_CALL <number>
```

Definition for the system call number of the log-message system call; should be in header.

```
int* retrieve_set_level_params(int new_level)
```

Returns an **int** array of 2-4 values that can be used to make the set-process-log-level system call.

```
int* retrieve_get_level_params()
```

Returns an **int** array of 2-4 values that can be used to make the get-process-log-level system call.

```
int interpret_set_level_result(int ret_value)
```

After making the system call, we will pass the syscall return value to this function call. It should return **set_proc_log_level**'s interpretation of the system call completing with return value **ret_value**.

```
int interpret_get_level_result(int ret_value)
```

After making the system call, we will pass the syscall return value to this function call. It should return **get_proc_log_level**'s interpretation of the system call completing with return value **ret_value**.

```
int interpret_log_message_result(int ret_value)
```

After making the system call, we will pass the syscall return value to this function call. It should return **proc_log_message**'s interpretation of the system call completing with return value **ret_value**.

Note that there is no **retrieve** function for **log message** as its system call format is defined above.

Submissions

You will submit the following at the end of this project (3 separate files):

- Text file (**screencast.txt**) containing link to unlisted screencast video
- Kernel Patch File (**p1.diff**) on Canvas
- Compressed tar archive (**process_log.tar.gz**) for **process_log** library on Canvas

Screencast

You will record a screencast (with audio) walking through the changes made to the kernel to create the system calls. As you explain something, you should be showing the corresponding file/code. You will upload the recording as an unlisted YouTube video and submit a text file containing a link to this video.

Your screencast should:

- Show each modified kernel file, explaining the changes you made to each and their importance in creating the system calls.
- Show and explain how the system-wide process log level was implemented.
- Show and explain how the executable name and pid were acquired.
- Show and explain how sudo user was checked.
- Show and explain how process message logging was implemented.
- Show all your library code and explain how the system calls were used by the library.
- Mention (you don't have to show) testing patch file in clean kernel.
- Demo library functionality by starting with process_log.tar.gz and following the commands given in the Compressed Archive section to compile and run library_test.cpp and harness_test.cpp
- Be at most **5 minutes** long. (Audio speed-up is prohibited)

Patch File

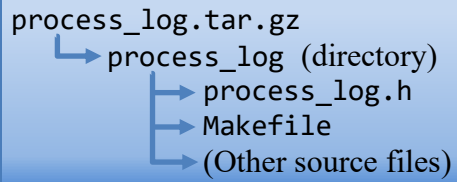
The patch file will include all changes to all kernel files in a single patch. Applying the patches and remaking the necessary parts of Reptilian, then rebooting and then building the test code (which we will also copy over) should compile the test program.

Your project will be tested by applying the patch while in **/usr/rep/src/reptilian-kernel**:

```
$ git apply p1.diff
$ make && sudo make install && sudo make modules_install
```

Compressed Archive (process_log.tar.gz)

Your compressed tar file should have the following directory/file structure:



```
process_log.tar.gz
├── process_log (directory)
│   ├── process_log.h
│   ├── Makefile
│   └── (Other source files)
```

To build the library, we will execute these commands (**from a non-kernel-source directory**):

```
$ tar zxvf process_log.tar.gz
$ cd process_log
$ make
$ cd ..
```

To link against the library, we will execute this command:

```
$ cc -o program_name sourcefile.c -L ./process_log -lprocess_log
```

Please test your library build and linking before submission! If your library does not compile it will result in **zero credit** (0, none, goose-egg) for the library portion of the project.