COL331 - Assignment 1

Nikhil Zawar - 2022CS11106

March 6, 2025

Contents

1 Introduction

In this implementation, three new system calls—block, unblock, and history—enhance the functionality and security of xv6. The block and unblock system calls allow processes to control access to specific system calls using a per-process bit-vector. The history system call maintains a log of executed processes, storing details such as process ID, name, and memory usage. These details are recorded upon process termination and can be retrieved in chronological order.

2 Authentication for xv6

Objective: to enhance the security of the xv6 kernel by implementing a basic authentication mechanism

2.1 Makefile Configuration:

In the Makefile, specific credentials were defined as preprocessor macros to ensure they are accessible during compilation.

```
USERNAME = \"nikhil\"
PASSWORD = \"nikhil\"
CFLAGS += -DUSERNAME="$(USERNAME)" -DPASSWORD="$(PASSWORD)"
```

2.2 Implementation Details:

- *compare_strings* Function: compare two strings, taking into account the difference in termination characters between terminal and Makefile input.
- *authenticate* Function: implementd in init.c responsible for user authentication
- If the username does not match the predefined USERNAME, an attempt is counted. Incorrect password attempts are counted, allowing for up to MAX_ATTEMPTS (set to 3 in the implementation)

Conclusion: In conclusion, the implementation of authentication in the xv6 kernel enhances its security by requiring users to authenticate with a specific username and password combination. The solution addresses key security concerns while ensuring compatibility and usability within the xv6 environment. Managing string comparisons with different termination characters posed an initial challenge, addressed by the compare_strings function.

3 History Functionality

Objective: implementing a new system call, ‘history‘, in the xv6 operating system. This system call retrieves the process execution history and sorts it by process ID (PID), which effectively orders the history chronologically. The integration is of multiple files within the xv6 - ‘syscall.h‘, ‘syscall.c‘, ‘proc.h‘, ‘proc.c‘, ‘sysproc.c‘, ‘usys.S‘, and ‘sh.c‘.

3.1 System Call Declaration

The system call number is defined in ‘syscall.h‘: This ensures that ‘gethistory‘ is recognized by the kernel.

```
#define SYS_gethistory 22
```

3.2 System Call Mapping

- In ‘syscall.c‘, the system call is mapped to the function ‘sys_gethistory‘:[SYS_gethistory] sys_gethistory,
- This associates the system call number 22 with the function ‘sys_gethistory‘.
- In ‘proc.h‘, the history structure and tracking variables are defined: #define MAX_HISTORY 100 extern struct history_entry process_history[MAX_HISTORY];

3.3 Recording & Reordering Process History

In ‘proc.c‘, the ‘process_history‘ array and ‘history_count‘ variable are implemented: The function ‘add_to_history‘ records a process’s details before it exits.

```
struct history_entry process_history[MAX_HISTORY];
int history_count = 0;

void add_to_history(struct proc *p) {
  if (history_count >= MAX_HISTORY) {
      return;
  }
  struct history_entry *entry = &process_history[history_count];
  entry->pid = p->pid;
  safestrcpy(entry->name, p->name, sizeof(entry->name));
  entry->mem_usage = p->sz;
  history_count++;
}
```

The function ‘add_to_history‘ is invoked in the ‘exit()‘ function within ‘proc.c‘, ensuring that every terminated process is recorded.
In ‘sysproc.c‘, ‘sys_gethistory‘ is defined to retrieve and sort the process history. Sorting by PID ensures that the processes are ordered chronologically.

```c
struct history_entry process_history[MAX_HISTORY];
int history_count = 0;

void add_to_history(struct proc *p) {
  if (history_count >= MAX_HISTORY) {
    return;
  }
  struct history_entry *entry = &process_history[history_count];
  entry->pid = p->pid;
  safestrcpy(entry->name, p->name, sizeof(entry->name));
  entry->mem_usage = p->sz;
  history_count++;
}
```

3.4 System Call handling at user level & File interconnections

In ‘usys.S‘, the system call is declared: **SYSCALL(gethistory)**
To detect when the ‘gethistory‘ system call should be executed, ‘sh.c‘ is modified. Idea: check if the command ‘history‘ has been entered. If detected, the ‘gethistory‘ system call is triggered. If the call fails, an error message is printed. This ensures that the user can retrieve process history by typing ‘history‘ in the shell.

The following diagram illustrates how different files interact in implementing the ‘gethistory‘ system call:

- ‘syscall.h‘ defines the system call number.
- ‘syscall.c‘ maps the number to the function.
- ‘proc.h‘ declares the history structure and function prototype.
- ‘proc.c‘ implements process history tracking.
- ‘sysproc.c‘ implements the system call logic.
- ‘usys.S‘ links user-space to kernel-space.
- ‘sh.c‘ detects user input and invokes ‘gethistory‘.

Conclusion: By recording terminated process details and sorting them based on PID (effectively a timestamp), this is a simple yet effective history tracking mechanism.

4 block & unblock

4.1 syscall function in syscall.c

The syscall function is responsible for handling system calls made by user processes.

- The function starts by retrieving the system call number from the eax register of the current process (curproc). This is how system calls are invoked in xv6—by placing the syscall number in the eax register before triggering a software interrupt.

- **Checking for System Call Blocking via Bit Vector:** If parent_bit == 0, the function checks whether the direct parent of the process has blocked the system call using a bit vector mechanism. The function is_syscall_blocked checks whether the parent process has disabled the requested system call. If blocked, the function prints a message and denies the syscall by setting eax = -1.
- **Checking if the Process is Marked for Parent-Based Restriction :** A flag parent_bit is used to determine if the current process (curproc) is subject to system call blocking rules inherited from its grandparent (parent_pro-¿parent). If parent_bit == 1, the function checks whether the grandparent process has blocked the syscall. If the syscall is blocked, an error message is printed.

4.2 modification to prco.h

This is a key modifcation in the task_struct which is

```c
struct proc {
    ...
    char blocked_syscalls[MAX_BLOCKED_SYSCALLS];
    int parent_bit;
};
```

Listing 1: Definition of struct proc

4.3 Implementation of the 'block' & 'unblock' System Call

The 'block' system call allows a process to disable a specific system call for itself. The function is defined in 'sysproc.c'. The function retrieves the **syscall ID** from user input.It prevents blocking critical system calls ('syscall_id == 1' or 'syscall_id == 2'). It sets 'blocked_syscalls[syscall_id] = 1', marking the system call as blocked.

The 'unblock' system call allows a process to re-enable a previously blocked system call: The function retrieves the syscall ID from user input. It sets 'blocked_syscalls[syscall_id] = 0', allowing the system call to be executed again.

4.4 Summary

This implementation allows processes in xv6 to **block and unblock system calls dynamically** using a **bit vector**. The modifications enhance **process security** by:

- **Preventing unauthorized syscalls** at both **process and grandparent levels**.
- **Ensuring system stability** by restricting critical syscalls (`fork`, `exit`).
- **Providing a flexible mechanism** for controlling system call access.

```c
int sys_block(void){
    int syscall_id;
    if(argint(0, &syscall_id) < 0)
        return -1;
    ...l_id == 1 || syscall_id == 2)
        return -1;
    struct proc *curproc = myproc();
    curproc->blocked_syscalls[syscall_id] = 1;
    return 0;
}

int sys_unblock(void){
    int syscall_id;
    if(argint(0, &syscall_id) < 0)
        return -1;
    struct proc *curproc = myproc();
    curproc->blocked_syscalls[syscall_id] = 0;
    return 0;
}
```