OS A1

Anuj Naval - 2021CS50591

March 2025

1 Introduction

This assignment involves enhancing the xv6 shell by implementing several new features through system calls and user-level commands. The modifications include authentication mechanisms, command history tracking, system call blocking, and file permission management.

- **User Authentication:** The shell is protected with a username-password-based login system. The credentials are defined in the Makefile, and users have a maximum of three attempts to log in before access is denied.
- **Command History:** A new system call, `sys_gethistory`, is introduced to retrieve and display a history of executed commands. Each entry includes the process ID (PID), process name, and memory utilization.
- **System Call Blocking:** The `sys_block` and `sys_unblock` system calls are implemented to allow blocking and unblocking specific system calls for processes spawned by the current shell.
- **File Permission Management:** The `chmod` command is added through the `sys_chmod` system call, which modifies file permissions based on read, write, and execute bits. Unauthorized operations result in appropriate error messages.

This document details the implementation methodology, including modifications to the xv6 kernel, system call handling, and user-space commands. Code snippets or pseudocode are provided to illustrate key steps.

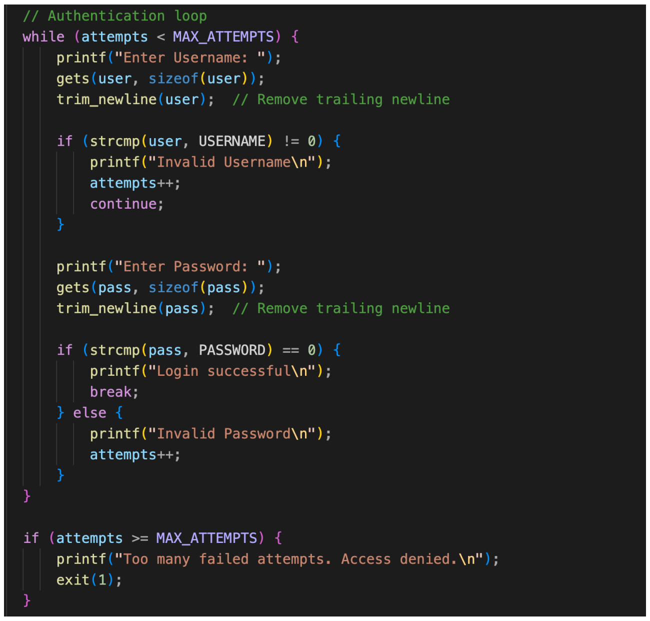
2 Enhanced Shell for xv6: User Authentication

2.1 Implementation Methodology

To implement user authentication in xv6, modifications were made to the `init.c` file to require a valid username and password before starting the shell. The authentication logic was implemented in a loop that allows up to three attempts before denying access.

2.2 Modifications to init.c

The following modifications were made to `init.c`:



```c
// Authentication loop
while (attempts < MAX_ATTEMPTS) {
    printf("Enter Username: ");
    gets(user, sizeof(user));
    trim_newline(user);  // Remove trailing newline

    if (strcmp(user, USERNAME) != 0) {
        printf("Invalid Username\n");
        attempts++;
        continue;
    }

    printf("Enter Password: ");
    gets(pass, sizeof(pass));
    trim_newline(pass);  // Remove trailing newline

    if (strcmp(pass, PASSWORD) == 0) {
        printf("Login successful\n");
        break;
    } else {
        printf("Invalid Password\n");
        attempts++;
    }
}

if (attempts >= MAX_ATTEMPTS) {
    printf("Too many failed attempts. Access denied.\n");
    exit(1);
}
```

Figure 1: Authentication Loop

- Open the console and set up standard input, output, and error.
- Prompt the user for a username and password.
- Compare the user input with predefined credentials.
- Allow a maximum of three login attempts before terminating access.
- If authentication is successful, proceed to start the shell.

The authentication mechanism ensures that only authorized users can access the xv6 shell. The login prompt repeatedly asks for credentials until valid input is provided or the maximum attempts are exceeded.

2.3 Defining Macros in the Makefile

To configure the authentication system, the username and password were defined as macros in the `Makefile`:

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

These macros are passed as preprocessor directives during compilation, embedding the credentials in the xv6 kernel.

2.4 Challenges and Enhancements

- Preventing buffer overflows by ensuring input does not exceed predefined sizes.
- Removing newline characters from input to avoid mismatches.
- Using `fork()` and `exec()` to launch the shell after successful authentication.
- A future improvement could involve hashing passwords instead of storing them as plaintext.

This implementation successfully restricts access to the shell and improves security within xv6.

3 Shell Command: `history`

3.1 Introduction

The `history` command provides a way to track and display all executed processes in xv6. It lists the process ID (PID), process name, and memory usage of each process. This feature is implemented using a new system call, `sys_gethistory`, and a history management function, `add_to_history`.

3.2 Process Execution History

A global array in the kernel maintains a history of process executions. This history is updated whenever a new process starts. The key functions involved in this implementation are:

- `add_to_history`: Adds process details to the global history array.
- `sys_gethistory`: Retrieves process history for user-space programs.

3.3 Adding Processes to History

The function `add_to_history` is defined in `proc.c` and is called whenever a process is created. It stores the following details:

- Process ID (PID)
- Process Name
- Memory Usage (Text, Data, BSS, Heap, Stack)

3.4 Kernel Modifications

The kernel is modified to support process history tracking:

- A global data structure is introduced to maintain history.
- The function `add_to_history` is implemented to populate this structure.
- A new system call, `sys_gethistory`, retrieves history data.

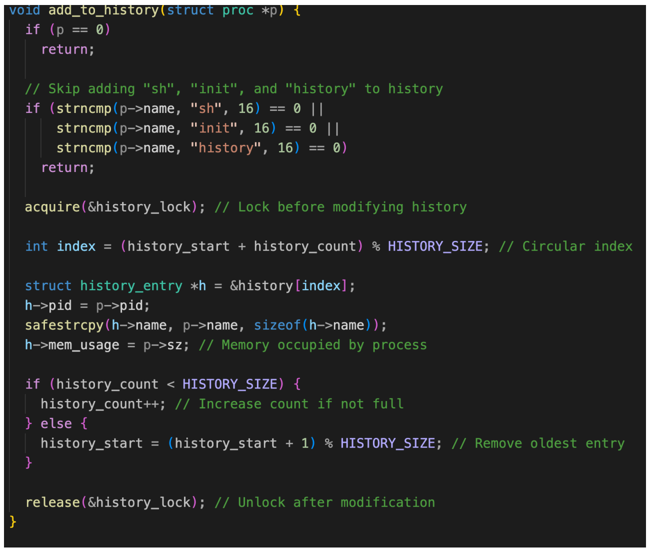
3.5 User-Space Implementation (`history.c`)

The user-space program `history.c` is responsible for displaying the process history. It does the following:

- Calls the `sys_gethistory` system call.
- Sorts the process history based on execution order.
- Displays the output in a tabular format.

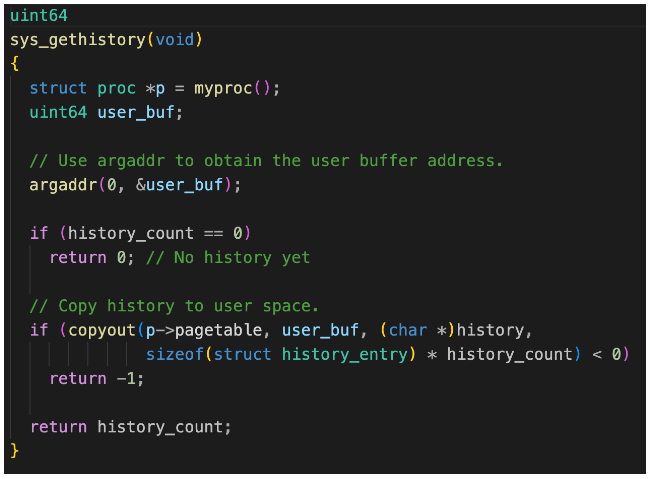
3.6 Code Snippets

The following code snippets show implementation:



```c
void add_to_history(struct proc *p) {
  if (p == 0)
    return;

  // Skip adding "sh", "init", and "history" to history
  if (strncmp(p->name, "sh", 16) == 0 ||
      strncmp(p->name, "init", 16) == 0 ||
      strncmp(p->name, "history", 16) == 0)
    return;

  acquire(&history_lock); // Lock before modifying history

  int index = (history_start + history_count) % HISTORY_SIZE; // Circular index

  struct history_entry *h = &history[index];
  h->pid = p->pid;
  safestrcpy(h->name, p->name, sizeof(h->name));
  h->mem_usage = p->sz; // Memory occupied by process

  if (history_count < HISTORY_SIZE) {
    history_count++; // Increase count if not full
  } else {
    history_start = (history_start + 1) % HISTORY_SIZE; // Remove oldest entry
  }

  release(&history_lock); // Unlock after modification
}
```

Figure 2: Implementation of add_to_history



```
uint64
sys_gethistory(void)
{
  struct proc *p = myproc();
  uint64 user_buf;

  // Use argaddr to obtain the user buffer address.
  argaddr(0, &user_buf);

  if (history_count == 0)
    return 0; // No history yet

  // Copy history to user space.
  if (copyout(p->pagetable, user_buf, (char *)history,
              sizeof(struct history_entry) * history_count) < 0)
    return -1;

  return history_count;
}
```

Figure 3: Implementation of sys_gethistory

3.7 Conclusion

The `history` command enhances process tracking in xv6. By maintaining execution history in the kernel and retrieving it using a system call, this feature provides a simple yet effective way to monitor process activity.

4 Shell command: `block`

4.1 Introduction

System call blocking provides a mechanism to restrict specific system calls for processes spawned by the shell. This is achieved through two new system calls: `sys_block` and `sys_unblock`, which allow the user to disable and enable system calls dynamically.

4.2 Implementation Methodology

The implementation consists of modifying the xv6 kernel to maintain a per-process system call permission table. The key components are:

- A bitmap to track blocked system calls per process.
- The `sys_block` and `sys_unblock` system calls to modify the bitmap.
- Updates to `syscall.c` to check the bitmap before executing system calls.

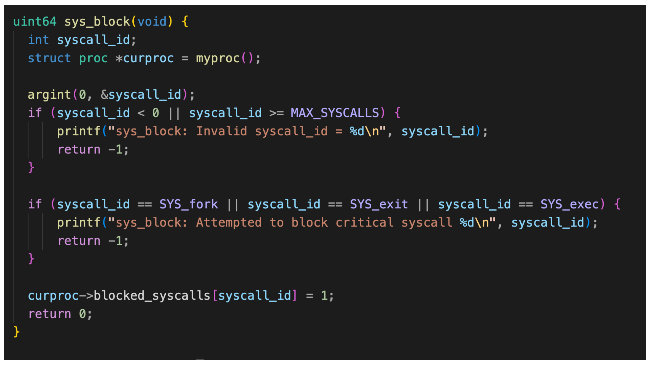
4.3 Kernel Modifications

The following changes were made to the xv6 kernel:

- A new field, `blocked_syscalls`, was added to the `proc` structure to store the blocked system calls.
- The `sys_block` and `sys_unblock` system calls were implemented in `sysproc.c`.
- The `syscall()` function in `syscall.c` was modified to check the `blocked_syscalls` bitmap before executing a system call.

4.4 System Call Implementations

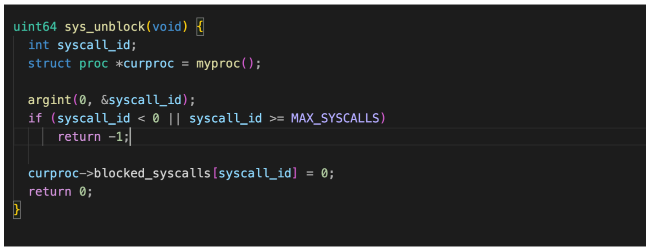
The `sys_block` system call marks a given system call as blocked for the calling process:



```c
uint64 sys_block(void) {
  int syscall_id;
  struct proc *curproc = myproc();

  argint(0, &syscall_id);
  if (syscall_id < 0 || syscall_id >= MAX_SYSCALLS) {
      printf("sys_block: Invalid syscall_id = %d\n", syscall_id);
      return -1;
  }

  if (syscall_id == SYS_fork || syscall_id == SYS_exit || syscall_id == SYS_exec) {
      printf("sys_block: Attempted to block critical syscall %d\n", syscall_id);
      return -1;
  }

  curproc->blocked_syscalls[syscall_id] = 1;
  return 0;
}
```

Figure 4: Implementation of sys_block

Similarly, the `sys_unblock` system call unblocks a previously blocked system call:



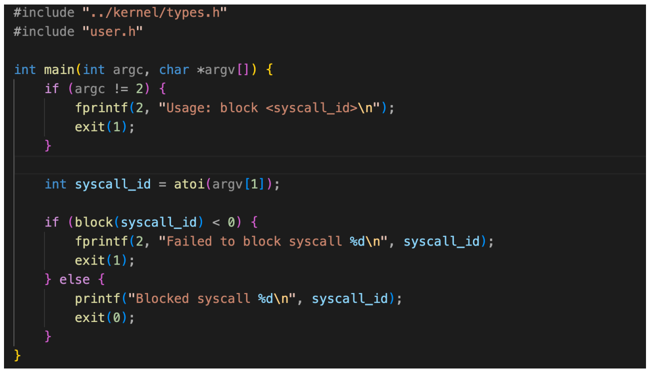
```c
uint64 sys_unblock(void) {
  int syscall_id;
  struct proc *curproc = myproc();

  argint(0, &syscall_id);
  if (syscall_id < 0 || syscall_id >= MAX_SYSCALLS)
      return -1;

  curproc->blocked_syscalls[syscall_id] = 0;
  return 0;
}
```

Figure 5: Implementation of sys_unblock

4.5 User-Space Implementation

User-space programs, `block.c` and `unblock.c`, provides a command-line utility to block and unblock system calls. It takes input from the user and invokes `sys_block` or `sys_unblock` accordingly.

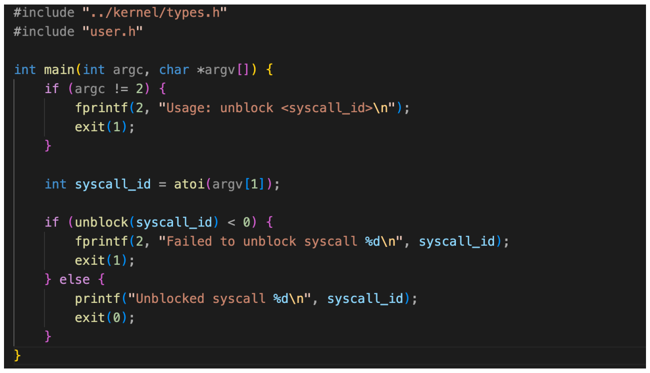
Following is the implementation of `block.c`



```c
#include "../kernel/types.h"
#include "user.h"

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(2, "Usage: block <syscall_id>\n");
        exit(1);
    }

    int syscall_id = atoi(argv[1]);

    if (block(syscall_id) < 0) {
        fprintf(2, "Failed to block syscall %d\n", syscall_id);
        exit(1);
    } else {
        printf("Blocked syscall %d\n", syscall_id);
        exit(0);
    }
}
```

Figure 6: Implementation of `block.c`

Following is the implementation of `unblock.c`



```c
#include "../kernel/types.h"
#include "user.h"

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(2, "Usage: unblock <syscall_id>\n");
        exit(1);
    }

    int syscall_id = atoi(argv[1]);

    if (unblock(syscall_id) < 0) {
        fprintf(2, "Failed to unblock syscall %d\n", syscall_id);
        exit(1);
    } else {
        printf("Unblocked syscall %d\n", syscall_id);
        exit(0);
    }
}
```

Figure 7: Implementation of `unblock.c`

4.6 Challenges and Enhancements

- Ensuring that essential system calls (e.g., `exit`) cannot be blocked.
- Managing blocked system calls across process forking.
- A future enhancement could allow blocking system calls for all child processes of a parent process.

4.7 Conclusion

The system call blocking mechanism enhances security and control within xv6 by allowing users to restrict specific system calls dynamically. This feature is useful for debugging, access control, and security testing.