Assignment 1 – Enhanced Shell for xv6 (Easy)

Vaibhav Katendra
2024MCS2459

March 6, 2025

Contents

1 Introduction

The objective of this assignment is to extend the standard xv6 shell with additional security and functionality features. The enhancements include:

1. A login mechanism that authenticates users via a fixed username and password before any shell operations.
2. A history command that maintains a log of executed external commands, capturing essential details like process IDs, command names, and memory usage.
3. Block/unblock commands that allow the shell to selectively disable certain system calls, thus enabling controlled execution of processes.
4. A chmod command that modifies file permissions directly within the on-disk inode, leveraging a union of bitfields for efficient storage.

This report describes how each feature was implemented, highlighting both the benefits and potential limitations of the chosen approaches.

2 Implementation Methodology

2.1 Enhanced Shell and Login System

The login system is critical for ensuring that only authorized users can access the shell. Its implementation is kept simple yet robust for educational purposes.

2.1.1 Design Overview

- **Fixed Credentials:** The username and password are hardcoded as macros (typically set in the Makefile), which simplifies the authentication process.
- **Attempt Limiting:** To mitigate brute-force attacks, the system allows only three attempts. This is controlled by decrementing an attempt counter on each failure.
- **User Feedback:** Informative messages guide the user through the process. Incorrect entries provide the number of remaining attempts, and upon successful authentication, a confirmation is displayed.
- **Simplicity and Extendability:** Although the current implementation uses fixed credentials, the structure can be extended to support file-based credential storage or integration with external authentication services.

Listing 1: login() in init.c

```
void login() {
  char uname[20], pass[20];
  int attempts = MAX_ATTEMPTS;
  while (attempts > 0) {
      // Prompt user for username
      printf(1, "Enter username: ");
      gets(uname, sizeof(uname));

      // Validate username against predefined macro
      if (strcmp(uname, USERNAME) != 0) {
         attempts--;
         printf(1, "Invalid Username. Attempts left: %d\n", attempts);
         continue;
      }

      // Prompt for password upon correct username
      printf(1, "Enter password: ");
      gets(pass, sizeof(pass));

      // Compare provided password with expected password
      if (strcmp(pass, PASSWORD) == 0) {
          printf(1, "Login successful\n");
          return;
      } else {
          attempts--;
          printf(1, "Incorrect password. Attempts left: %d\n", attempts);
      }
  }

  // Deny access if maximum attempts are reached
  printf(1, "Login failed. Access denied.\n");
  while (1);
}
```

2.2 History Command Implementation

2.2.1 Data Structure and Rationale

A global history array, limited to 64 entries, is defined to store details of external commands executed by the user. Each entry records:

- **Process ID:** Helps in uniquely identifying the process.
- **Process Name:** Captured as a string (up to 16 characters) to identify the command executed.
- **Memory Usage:** The total memory allocated to the process, providing insight into resource utilization.

This structure is designed for efficiency and minimal overhead, considering the limited resources in xv6.

Listing 2: History data structure in proc.h

```
#define MAX_HISTORY 64

struct historyEntry {
  int pid;
  char name[16];
  int mem_usage;
};
```

2.2.2 Logging Mechanism

In the `exit()` function (in `proc.c`), the process details are logged only if the process has successfully executed an external command . This filtering is crucial for:

- Avoiding clutter from internal shell commands.
- Ensuring that only meaningful user actions are recorded.

This selective logging also helps in debugging by providing a clearer history of user-initiated commands.

Listing 3: History logging snippet in exit() in proc.c

```
if(history_count < MAX_HISTORY) {
    history[history_count].pid = curproc->pid;
    safestrcpy(history[history_count].name, curproc->name, sizeof(history[
        history_count].name));
    history[history_count].mem_usage = curproc->sz;
    history_count++;
}
```

2.2.3 System Call Interface

A new system call, `sys_gethistory()`, is introduced in `sysproc.c`. This call iterates through the history array and prints each entry via `cprintf`. Key points include:

- **Iteration:** The system call loops through all recorded entries, ensuring no entry is missed.
- **Output:** Uses formatted printing to display process ID, name, and memory usage.
- **Integration:** This call is integrated into the system call table, making it accessible to user programs.

Listing 4: Implementation of sys_gethistory in sysproc.c

```
int sys_gethistory(void) {
  int i;
  for(i = 0; i < history_count; i++){
    cprintf("%d %s %d\n",
            history[i].pid,
            history[i].name,
            history[i].mem_usage);
  }
  return 0;
}
```

2.2.4 Command

The following code is added in sh.c. It handles error cases and allows users to easily access command history.

Listing 5: function sh.c

```
void builtin_history(void) {
  int ret = gethistory();
  if(ret < 0)
      printf(2, "Error getting history\n");
}
```

2.3 Block/Unblock Commands Implementation

2.3.1 Design and Purpose

Block/unblock commands provide a mechanism to disable certain system calls selectively within the shell. This functionality is designed to:

- Allow controlled execution environments.
- Prevent unauthorized or potentially harmful system calls in the shell's own context.
- Maintain normal behavior for child processes executing external commands by clearing blocked states.

These commands modify a per-process `blocked` array (defined in `struct proc` in `proc.h`) that indicates which system calls are blocked.

2.3.2 Implementation in Fork

When a new process is forked, the block settings are copied. However, to ensure that external commands run without restrictions, the block on the `exec` system call is cleared. This decision reflects the balance between process isolation and operational flexibility.

Listing 6: Modified fork() in proc.c

```
for(i = 0; i < NSYSCALLS; i++){
  np->blocked[i] = curproc->blocked[i];
}
```

2.3.3 Error Handling and Future Improvements

- **Error Handling:** The block/unblock system calls check for invalid indices and ensure that the blocked array is not modified beyond its bounds.
- **Future Work:** Consider implementing finer-grained control and logging for blocked system calls to aid in debugging and security audits.

2.4 chmod Command Implementation

2.4.1 Rationale Behind Design Choices

The chmod command directly modifies file permissions by updating the on-disk inode structure. Key points include:

- **Union of Bitfields:** A union is used to combine the number of links and permission bits into a single 16-bit field, preserving the inode size and allowing efficient storage.
- **Backward Compatibility:** The design ensures that the size of the dinode remains unchanged (64 bytes) even after modifications.
- **Direct Disk Modification:** By working directly with the on-disk inode, the chmod system call ensures that permission changes persist across reboots.

Listing 7: Modified dinode in fs.h

```
struct dinode {
  short type;
  short major;
  short minor;
  unsigned nlink:13;
  unsigned mode:3;
  uint size;
  uint addrs[NDIRECT+1];
} __attribute__((packed));

_Static_assert(sizeof(struct dinode) == 64, "dinode size must be 64 bytes");
```

2.4.2 System Call Implementation

The `sys_chmod` system call (in `sysproc.c`) handles updating the inode's permission bits. It carefully locks the inode, updates the mode field, writes the changes to disk, and then releases the lock. This process ensures data consistency and atomicity of the update operation.

Listing 8: Implementation of sys_chmod in sysproc.c

```
int sys_chmod(void) {
  char *path;
  int mode;
  struct inode *ip;
  if(argstr(0, &path) < 0 || argint(1, &mode) < 0)
    return -1;
  begin_op();
  if((ip = namei(path)) == 0){
    end_op();
    return -1;
  }
  ilock(ip);
  ip->mode = mode & 7;
  iupdate(ip);
  iunlockput(ip);
  end_op();
  return 0;
}
```

2.4.3 Considerations and Future Extensions

- **Error Checking:** The system call validates arguments and ensures the target inode exists before proceeding.
- **Potential Extensions:** Future improvements might include detailed logging of permission changes and support for more complex permission models.

3 Extra Details

- **History Filtering:** By recording only processes that successfully execute external commands, the shell maintains a concise and useful history log. Internal commands are deliberately excluded.
- **Block/Unblock Behavior:** The system calls for blocking are carefully designed to affect only the shell's internal operations. The clearing of the exec block in child processes is a key design decision that prevents unintended side effects on externally executed commands.
- **Login Implementation:** The login routine is embedded directly in the initial shell process. Although it uses fixed credentials for simplicity, this modular design allows for future enhancements such as encrypted passwords or integration with external authentication systems.

- **Error Handling:** Throughout the implementation, robust error checking is employed. Each system call verifies inputs and system state before making changes, ensuring system stability. In the future, more granular error messages and logging could be added to aid in debugging.

4 Conclusion

This report provides an in-depth look at the enhancements made to the xv6 shell. The key contributions of this assignment include:

- A secure login mechanism that validates user credentials with informative feedback and controlled access.
- A history command that accurately logs executed external commands along with process-specific data.
- Block/unblock commands that allow selective disabling of system calls in the shell while ensuring that child processes operate normally.
- A chmod command that directly updates file permissions within the on-disk inode using a compact union of bitfields.

The detailed design explanations and code walkthroughs provided in this report highlight the rationale behind each decision, as well as potential avenues for future improvements such as enhanced security measures, refined error handling, and expanded command functionality.