CCY4304 · 12TH PROJECT · RISC-V XV6

Securing the Kernel, Protecting Lives

A medical-device security layer for xv6-riscv: user authentication, file access control, and a syscall audit log.

Ahmed Walid Ibrahim
ID 221011183

Jana Ashraf Ali
ID 221010291

Course CCY4304: Operating Systems Security
Lecturer Prof. Dr. Ayman Adel Abdel-Hamid
Teaching Assistant Abdelrahman Solyman

Project Report 2026

Table of Contents

Abstract 3

Environment and Build 4

Phase 1: User Authentication 5

Phase 2: File Access Control 7

Phase 3: Syscall Audit Log 9

Compliance Testing 11

Regulatory Context 13

Appendix A: Source Code 14

Abstract

This report describes a security layer we added to xv6-riscv, the teaching kernel from MIT. The setting is a medical wearable, an insulin pump, that has to decide who is using it and what they are allowed to do. We built three things into the kernel: user authentication at boot, Unix-style file permissions on every file, and an audit log that records security-relevant system calls.

The motivation is a real failure. In 2019 the FDA issued a Class I recall for the Medtronic MiniMed 508 insulin pump. Researchers showed that anyone within radio range could send commands to the pump and change the insulin dose, with no authentication at all (CVE-2019-10964). The root cause was an operating system with no idea of user identity, no file access control, and no record of what happened. Any process could do anything. Our project answers a narrower question: what is the smallest set of OS-level controls a device like that needs before it should be trusted with a dose.

We kept the code small on purpose. xv6 is about ten thousand lines, so every change we made stays readable and every security decision can be traced from the system call down to the kernel check. The work is split into three phases. Each phase is a self-contained kernel change, and together they cover identity, access control, and accountability. An 18-test compliance program runs the whole thing on real xv6 inside QEMU and reports the result.

The three default accounts model the three roles a clinic deals with:

Username	Password	Role	UID
`admin`	`admin123`	Administrator	0
`patient1`	`patient123`	Patient	1
`doctor1`	`doctor123`	Doctor	2

Environment and Build

The kernel targets the RISC-V 64-bit `virt` board and runs under QEMU. Two tools are needed: a RISC-V cross compiler and `qemu-system-riscv64`. On Fedora the packages are `gcc-riscv64-linux-gnu` and `qemu-system-riscv`. On Debian or Ubuntu they are `gcc-riscv64-linux-gnu` and `qemu-system-misc`.

Building and booting is three commands:

```
cd xv6-security
make clean
make
make qemu
```

`make` cross-compiles the kernel and the user programs, then builds the filesystem image with `mkfs`. `make qemu` boots that image. The console drops straight into the login program, not a shell, which is the first visible result of Phase 1.

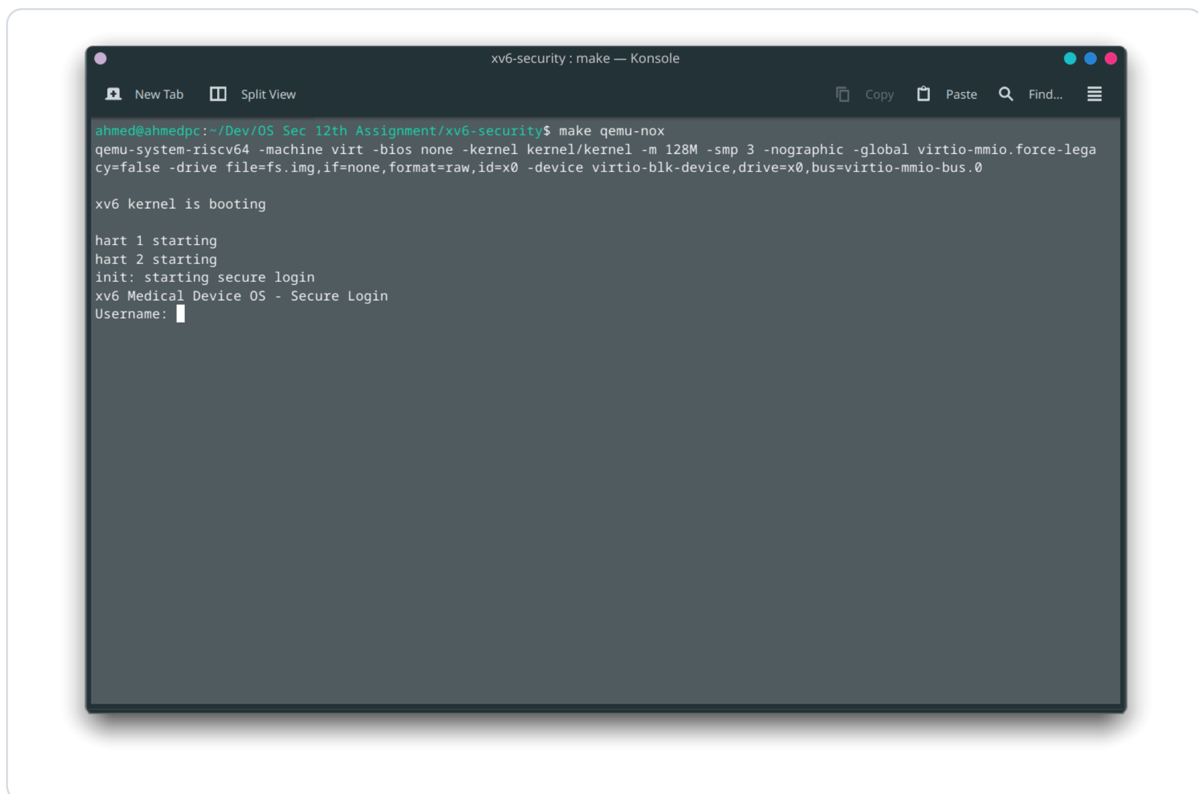

Figure 1. xv6 boots under QEMU and stops at the secure login prompt instead of a shell.

To leave QEMU, press `Ctrl-A` then `x`.

Phase 1: User Authentication

Stock xv6 runs `sh` the moment it boots. Every process is trusted the same, which is exactly the hole the MiniMed pump had. Phase 1 puts an identity check in front of the shell.

What we added to the kernel

Each process now carries an identity. We added five fields to `struct proc`:

```
int uid;              // 0 = admin, 1 = patient, 2 = doctor
int gid;              // group id, mirrors the uid here
int role;             // ROLE_ADMIN, ROLE_PATIENT, ROLE_DOCTOR
char username[16];    // for whoami and the audit log
int authenticated;    // 0 until login() succeeds
```

A forked child inherits all five fields, so every program a logged-in shell starts runs under the same identity.

Credentials live in `/etc/passwd`, one account per line:

```
username|uid|gid|role|hash
```

Passwords are never stored in the clear. We hash them first. We use SHA-256, the same cryptographic hash behind TLS certificates, Bitcoin, and Git. The implementation is self-contained and needs no external library. About 80 lines of C inside `pw_hash()` produces a 64-character hex digest. The hash still lacks a salt and a tunable work factor, which a production device should add via bcrypt or Argon2id, but the core primitive is now a real standard rather than a teaching approximation.

The login flow

`init` no longer runs the shell. It runs `login`. The login program asks for a username and password, calls the `login()` system call, and only runs `sh` once the call returns success. After three failed attempts it stops and locks the device until reboot, which models a wearable that should not let an attacker keep guessing.

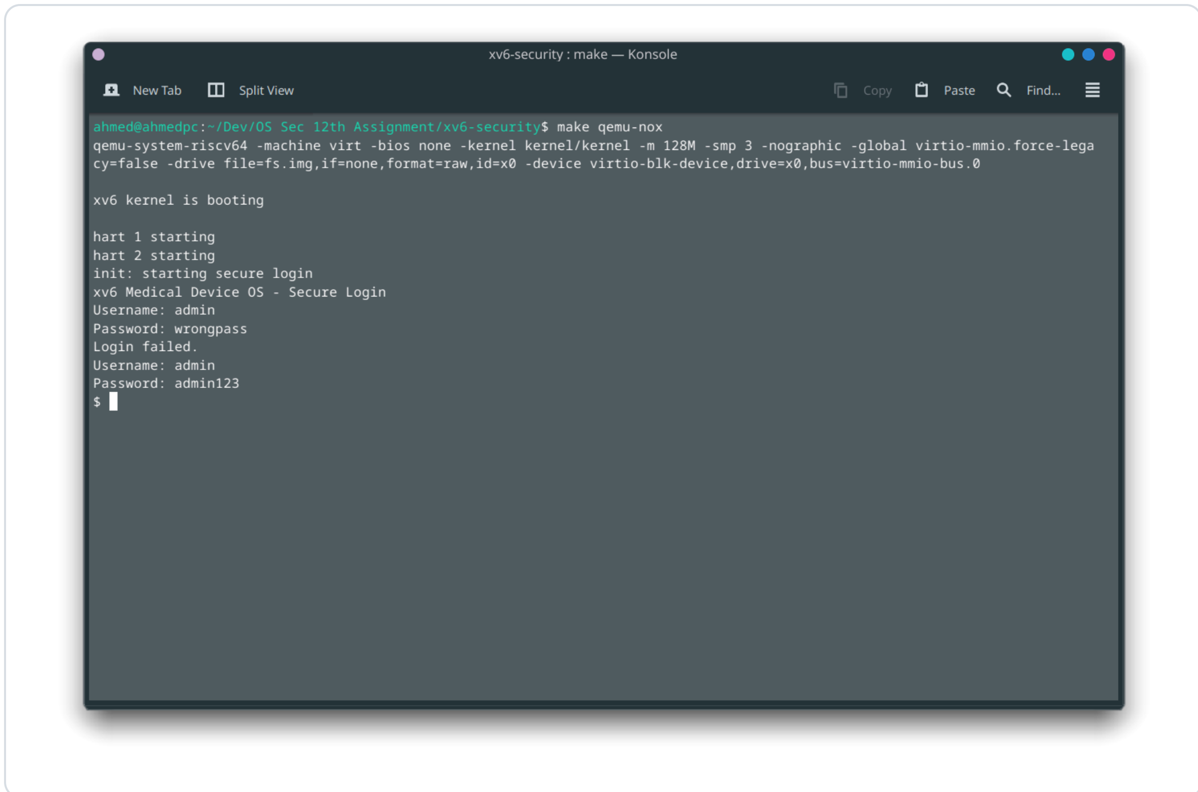

Figure 2. A wrong password is rejected, then a correct admin login opens the shell.

The account commands

Four commands manage identity. They map to system calls that enforce their own rules inside the kernel, so a patient cannot bypass them by crafting a raw call.

```
whoami                        # print name, uid, gid, role
useradd nurse nurse123 1      # admin adds a patient-role account
passwd doctor1 doctor123 newpw # doctor1 changes its own password
userdel nurse                 # admin removes the account
```

`useradd` and `userdel` are admin only. The `admin` account cannot be deleted. `passwd` lets a normal user change their own password if they give the old one, and lets the admin change anyone's. `whoami` works for any logged-in user.

Phase 2: File Access Control

Phase 1 says who you are. Phase 2 says what you can touch. Without it, a logged-in patient could open the device config or overwrite the insulin log, which would defeat the login entirely. We added Unix-style discretionary access control to every file.

What we added to the inode

The on-disk inode (`struct dinode`) gained three fields, mirrored on the in-memory inode:

```
ushort mode;    // permission bits, e.g. 0640
ushort uid;     // owner user id
ushort gid;     // owner group id
```

The permission bits follow the standard owner, group, other layout. A single helper, `perm_check`, reads the caller's identity and the inode's owner, group, and mode, then returns allow or deny. The admin (uid 0) bypasses the check. Everyone else is classified as owner, group, or other, and the matching read, write, or execute bit decides the outcome.

The protected medical files

`mkfs` assigns ownership when it builds the image, so the files are protected from the first boot:

Path	Mode	Owner	Who can do what
`/patient/records`	`0400`	uid=1, gid=1	Patient reads, admin override
`/dosage/insulin.log`	`0640`	uid=2, gid=1	Doctor (owner) writes, patient (group) reads, admin override
`/device/config`	`0600`	uid=0, gid=0	Admin only
`/audit/syscall.log`	`0400`	uid=0, gid=0	Admin read-only

The insulin log is the interesting one. The doctor owns it and writes new doses. The patient sits in the file's group with read-only group bits, so they can see their dose history but never change it. Nobody else gets in.

Where the check runs

Checking only at open time is not enough, because a process could hold an open handle after its rights change. So `perm_check` runs at four points: `sys_open`, `fileread`, `filewrite`, and `sys_exec`. Read needs the read bit, write needs the write bit, and running a program needs the execute bit.

Changing permissions

```
chmod /device/config 0600       # owner or admin may change the mode
chown /dosage/insulin.log 1 1   # admin only: change owner uid and gid
```

`chmod` is allowed for the file owner or the admin. `chown` is stricter and requires the admin.

Walkthrough: a patient opens the device config

Say `patient1` (uid 1) runs `cat /device/config`. The file is owned by admin (uid 0) with mode `0600`. The kernel resolves the path, then calls `perm_check` for read before it hands back a file descriptor. The patient is not admin, so there is no bypass. The file owner is uid 0, not 1, so the patient is not the owner. The file group is 0, the patient's group is 1, so they are not in the group either. That leaves the "other" class, and the other-read bit in `0600` is clear. The check returns deny, the open returns -1, and the read fails. The attempt is also written to the audit log, which Phase 3 covers.

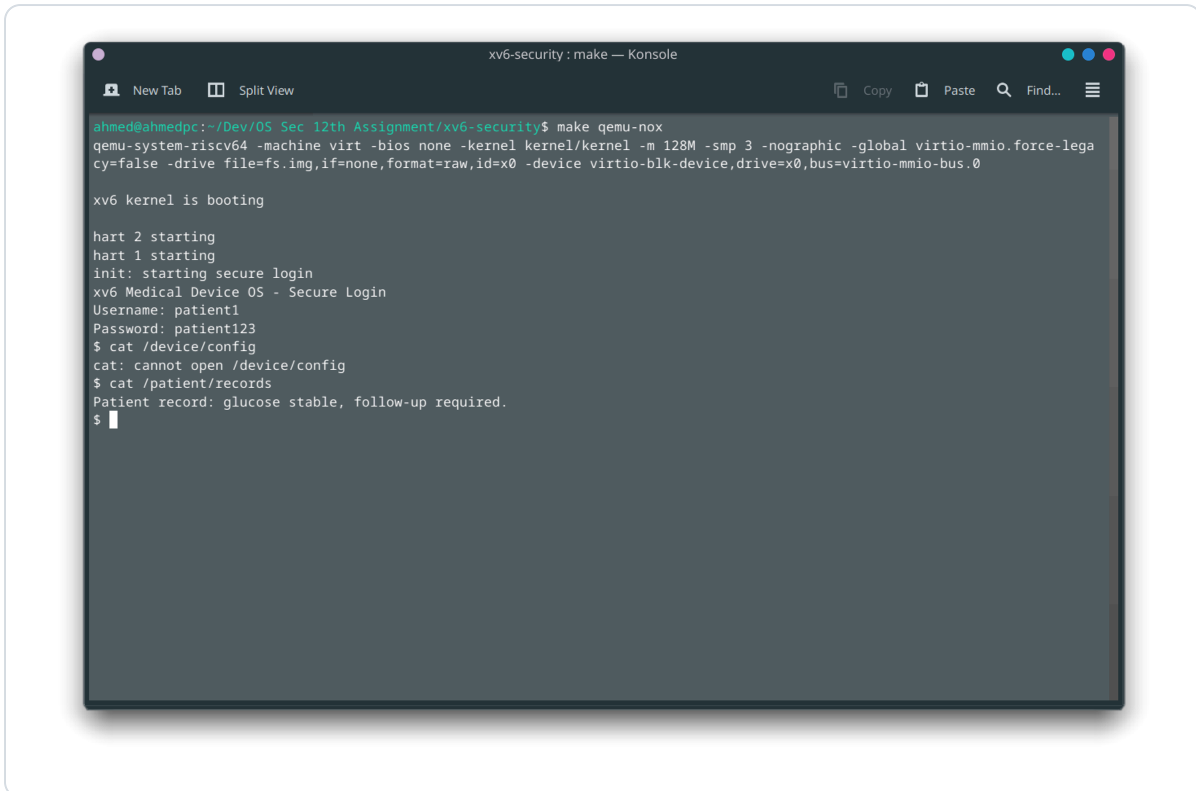

Figure 3. The patient is blocked from the device config but can read their own records.

Phase 3: Syscall Audit Log

After an incident, you need to know what happened, when, and who did it. A clinician should not be able to deny writing a dose, and an intrusion should leave a trail. Phase 3 adds a kernel audit log that records security-relevant system calls.

The ring buffer

The log is a fixed ring buffer of 256 entries (`AUDIT_BUF_SIZE`). A ring buffer fits kernel space well: it needs no dynamic allocation, each write is a couple of pointer moves, and when it fills the oldest entry is dropped so the system keeps running. A spinlock protects it from concurrent writes.

Each entry holds:

```
struct audit_entry {
  int  pid;          // process id of the caller
  int  uid;          // user id of the caller
  int  syscall_no;   // which syscall
  int  result;       // return value (-1 means denied or failed)
  uint tick;         // kernel tick, for ordering
  char comm[16];     // process name
};
```

The hook lives in `syscall.c`, right after each system call returns, so both allowed and denied calls are recorded by the same mechanism. We do not print audit lines from the trap handler. Printing there would loop, because `printf` itself calls `write`, and each character written would log another trap that prints again.

Reading the log

Reading the log is privileged. The `audit_read` system call returns -1 for any caller that is not the admin (uid 0). It does not even reveal the buffer size to an unprivileged process. Audit data is treated as sensitive on its own. The `audit_dump` tool prints the buffer for the admin:

```
$ audit_dump
tick pid uid syscall result comm
12 3 0 login 0 login
45 4 1 open 3 sh
67 4 1 open -1 compliance
89 4 0 audit_read 0 compliance
```

The columns are tick, pid, uid, syscall, result, and process name. A `-1` on `open` is a denied access. The line with uid 1 and result -1 is a patient who tried to open a file they do not own.

xv6-security : make — Konsole

New Tab Split View Copy Paste Find...

```
Login failed.
Username: admin
Password: admin123
$ audit_dump
tick pid uid syscall result comm
1 2 0 write 1 login
124 2 0 read 1 login
124 2 0 read 1 login
124 2 0 read 1 login
124 2 0 read 1 login
124 2 0 read 1 login
124 2 0 read 1 login
124 2 0 write 1 login
124 2 0 write 1 login
124 2 0 write 1 login
124 2 0 write 1 login
124 2 0 write 1 login
124 2 0 write 1 login
124 2 0 write 1 login
124 2 0 write 1 login
124 2 0 write 1 login
124 2 0 write 1 login
182 2 0 read 1 login
182 2 0 read 1 login
182 2 0 read 1 login
182 2 0 read 1 login
182 2 0 read 1 login
182 2 0 read 1 login
182 2 0 read 1 login
182 2 0 read 1 login
182 2 0 read 1 login
182 2 0 read 1 login
182 2 0 login -1 login
182 2 0 write 1 login
182 2 0 write 1 login
182 2 0 write 1 login
182 2 0 write 1 login
182 2 0 write 1 login
182 2 0 write 1 login
182 2 0 write 1 login
182 2 0 write 1 login
182 2 0 write 1 login
182 2 0 write 1 login
```

Figure 4. The admin dumps the audit ring and sees the denial recorded with its uid and tick.

Compliance Testing

`compliance_test` is a single user program that exercises all three phases on real xv6 inside QEMU. It logs in as each role, tries actions that should pass and actions that should fail, and checks the audit log for the right entries. Run it from the shell after logging in:

```
compliance_test
```

The 18 tests and what each one checks:

Test	What it checks
T01	Valid admin login succeeds
T02	Valid patient login succeeds
T03	Valid doctor login succeeds
T04	Wrong password is rejected
T05	Non-admin cannot call `useradd`
T06	`whoami` returns the correct username
T07	Patient cannot open `/device/config`
T08	Patient can read `/patient/records`
T09	Patient cannot write `/patient/records`
T10	Doctor can write `/dosage/insulin.log`
T11	Doctor cannot read `/device/config`
T12	Admin can open all protected files
T13	`audit_read` by a non-admin returns EPERM
T14	`audit_read` by the admin returns data
T15	The log contains the denial event
T16	The log contains the successful write event
T17	An attack is denied and detected in the audit log
T18	All three phases are active at once

All 18 pass.

xv6-security : make — Konsole

```
init: starting secure login
xv6 Medical Device OS - Secure Login
Username: admin
Password: admin123
$ compliance_test
[PASS] T01 valid admin login succeeds
[PASS] T02 valid patient login succeeds
[PASS] T03 valid doctor login succeeds
[PASS] T04 wrong password is rejected
[PASS] T05 non-admin cannot call useradd
[PASS] T06 whoami returns correct username
[PASS] T07 patient cannot open /device/config
[PASS] T08 patient can read /patient/records
[PASS] T09 patient cannot write /patient/records
[PASS] T10 doctor can write /dosage/insulin.log
[PASS] T11 doctor cannot read /device/config
[PASS] T12 admin can open all protected files
[PASS] T13 audit_read by non-admin returns EPERM
[PASS] T14 audit_read by admin returns data
[PASS] T15 log contains EPERM denial event
[PASS] T16 log contains successful write event
[PASS] T17 attack denied and detected in audit
[PASS] T18 all three phases active simultaneously

==========================================
  COMPLIANCE REPORT - CCY4304 12th Project
  Students: Ahmed Walid Ibrahim - 221011183
            Jana Ashraf Ali - 221010291
  Passed: 18 / 18
==========================================
$
```

Figure 5. The compliance run reports 18 of 18 tests passed.

Regulatory Context

This is a teaching model, not a compliance claim. Still, the three phases line up with what real guidance asks for. The FDA's 2023 premarket cybersecurity guidance expects authentication, access control, and event logging on a connected medical device. IEC 62443, the industrial security standard often applied to medical devices, frames the same ideas as identification and authentication control, use control, and the recording of security events. Phase 1 covers identity, Phase 2 covers use control, and Phase 3 covers the audit trail. The MiniMed 508 recall is what happens when all three are missing.

Appendix A: Source Code

The core of each phase is listed below. The full source, including the user-space tool wrappers (`useradd`, `userdel`, `passwd`, `whoami`, `chmod`, `chown`, `audit_dump`) and the compliance test, is on GitHub at https://github.com/ahmeddwalid/OSSec12th.

Phase 1: kernel/auth.c

Listing: `auth.c`

```
#include "types.h"
#include "riscv.h"
#include "defs.h"
#include "param.h"
#include "stat.h"
#include "fs.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "file.h"
#include "proc.h"
#include "auth.h"

#define PASSWD_BUF_SIZE 2048 // enough for MAX_USERS (16) pipe-delimited credentials
#define MAX_PASSWORD    64

// constant-time-safe: checks length before comparing bytes
static int
streq(const char *a, const char *b)
{
  return strlen(a) == strlen(b) && strncmp(a, b, strlen(a)) == 0;
}

static int
emptystr(const char *s)
{
  return s == 0 || s[0] == 0;
}

static void
append_char(char *buf, int *off, int max, char c)
{
  if(*off < max - 1)
    buf[(*off)++] = c;
  buf[*off] = 0;
}

static void
append_str(char *buf, int *off, int max, const char *s)
{
  while(*s)
    append_char(buf, off, max, *s++);
}

// writes decimal integer into buffer. tmp[16] holds "-2147483648" + nul
static void
append_int(char *buf, int *off, int max, int v)
{
  char tmp[16];
```

```
  int i = 0;

  if(v == 0){
    append_char(buf, off, max, '0');
    return;
  }
  if(v < 0){
    append_char(buf, off, max, '-');
    v = -v;
  }
  while(v > 0 && i < (int)sizeof(tmp)){
    tmp[i++] = '0' + (v % 10);
    v /= 10;
  }
  while(i > 0)
    append_char(buf, off, max, tmp[--i]);
}

static int
parse_int(char **cursor, int *out)
{
  int v = 0;
  char *p = *cursor;

  if(*p < '0' || *p > '9')
    return -1;
  while(*p >= '0' && *p <= '9'){
    v = v * 10 + (*p - '0');
    p++;
  }
  *out = v;
  *cursor = p;
  return 0;
}

static int
parse_field(char **cursor, char *out, int outsz, char delim)
{
  int n = 0;
  char *p = *cursor;

  while(*p && *p != delim && *p != '\n'){
    if(n < outsz - 1)
      out[n++] = *p;
    p++;
  }
  out[n] = 0;
  if(*p != delim)
    return -1;
  *cursor = p + 1;
  return 0;
}

static int
parse_credentials(char *data, struct credential creds[], int max)
{
  int count = 0;
  char *p = data;

  while(*p && count < max){
    if(*p == '\n'){
```

```
      p++;
      continue;
    }
    if(parse_field(&p, creds[count].username, sizeof(creds[count].username), '|') < 0)
      return -1;
    if(parse_int(&p, &creds[count].uid) < 0 || *p++ != '|')
      return -1;
    if(parse_int(&p, &creds[count].gid) < 0 || *p++ != '|')
      return -1;
    if(parse_int(&p, &creds[count].role) < 0 || *p++ != '|')
      return -1;
    if(parse_field(&p, creds[count].hash, sizeof(creds[count].hash), '\n') < 0){
      int n = 0;
      while(*p && n < HASH_LEN){
        creds[count].hash[n++] = *p++;
      }
      creds[count].hash[n] = 0;
      if(*p != 0)
        return -1;
    }
    count++;
  }
  return count;
}

static void
append_credential(char *buf, int *off, int max, struct credential *c)
{
  append_str(buf, off, max, c->username);
  append_char(buf, off, max, '|');
  append_int(buf, off, max, c->uid);
  append_char(buf, off, max, '|');
  append_int(buf, off, max, c->gid);
  append_char(buf, off, max, '|');
  append_int(buf, off, max, c->role);
  append_char(buf, off, max, '|');
  append_str(buf, off, max, c->hash);
  append_char(buf, off, max, '\n');
}

static struct inode*
auth_create(char *path, short type)
{
  struct inode *ip, *dp;
  char name[DIRSIZ];

  if((dp = nameiparent(path, name)) == 0)
    return 0;
  ilock(dp);
  if((ip = dirlookup(dp, name, 0)) != 0){
    iunlockput(dp);
    ilock(ip);
    return ip;
  }
  if((ip = ialloc(dp->dev, type)) == 0){
    iunlockput(dp);
    return 0;
  }
  ilock(ip);
  ip->nlink = 1;
  iupdate(ip);
```

```
  if(type == T_DIR){
    if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
      goto fail;
  }
  if(dirlink(dp, name, ip->inum) < 0)
    goto fail;
  if(type == T_DIR){
    dp->nlink++;
    iupdate(dp);
  }
  iunlockput(dp);
  return ip;

fail:
  ip->nlink = 0;
  iupdate(ip);
  iunlockput(ip);
  iunlockput(dp);
  return 0;
}

static int
read_passwd(struct credential creds[], int max)
{
  static char buf[PASSWD_BUF_SIZE]; // kept off the 1-page kernel stack
  struct inode *ip;
  int n;

  begin_op();
  if((ip = namei(PASSWD_FILE)) == 0){
    end_op();
    return -1;
  }
  ilock(ip);
  n = readi(ip, 0, (uint64)buf, 0, sizeof(buf) - 1);
  iunlockput(ip);
  end_op();
  if(n < 0)
    return -1;
  buf[n] = 0;
  return parse_credentials(buf, creds, max);
}

static int
write_passwd(struct credential creds[], int count)
{
  static char buf[PASSWD_BUF_SIZE]; // kept off the 1-page kernel stack
  int off = 0;
  struct inode *ip;
  int wrote;

  memset(buf, 0, sizeof(buf));
  for(int i = 0; i < count; i++)
    append_credential(buf, &off, sizeof(buf), &creds[i]);
  if(off >= sizeof(buf) - 1)
    return -1;

  begin_op();
  if((ip = namei(PASSWD_FILE)) == 0){
    end_op();
    return -1;
```

```c
  }
  ilock(ip);
  itrunc(ip);
  wrote = writei(ip, 0, (uint64)buf, 0, off);
  iunlockput(ip);
  end_op();
  return wrote == off ? 0 : -1;
}

static void
seed_credential(struct credential *c, char *username, int uid, int role, char *password)
{
  safestrcpy(c->username, username, sizeof(c->username));
  c->uid = uid;
  c->gid = uid;
  c->role = role;
  pw_hash(password, c->hash);
}

// sha-256 per FIPS 180-4. runs in kernel context — no malloc, only stack vars.
// outputs 64 hex chars + nul into out_hex.
void
pw_hash(const char *password, char *out_hex)
{
  static const char hex[] = "0123456789abcdef";
  static const uint K[64] = {
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
    0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
    0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
    0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
    0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
    0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
    0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
    0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
    0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
  };

  uint H[8] = {
    0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
    0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19
  };

  uchar block[128];
  uint W[64];
  uint a, b, c, d, e, f, g, h, t1, t2, S0, S1, ch, maj;
  uint msglen, i, j, blocks;
  uint64 bitlen;

  msglen = strlen((char*)password);
  bitlen = (uint64)msglen * 8;

  for(i = 0; i < msglen; i++)
    block[i] = password[i];
```

```
    // sha-256 padding per FIPS 180-4: 0x80, zero-fill, 64-bit big-endian bit length.
  // single block if msg < 56 bytes, two blocks if 56–63 bytes.
  block[msglen] = 0x80;

  if(msglen < 56){
    for(i = msglen + 1; i < 56; i++)
      block[i] = 0;
    blocks = 1;
  } else {
    for(i = msglen + 1; i < 64; i++)
      block[i] = 0;
    j = (msglen >= 64) ? 1 : 0;
    for(i = j; i < 56; i++)
      block[64 + i] = 0;
    blocks = 2;
  }

  j = blocks * 64 - 8;
  for(i = 0; i < 8; i++)
    block[j++] = (bitlen >> (56 - i * 8)) & 0xff;

  for(j = 0; j < blocks; j++){
    for(i = 0; i < 16; i++){
      uint off = j * 64 + i * 4;
      W[i] = ((uint)block[off] << 24) | ((uint)block[off+1] << 16) |
             ((uint)block[off+2] << 8) | (uint)block[off+3];
    }

    for(i = 16; i < 64; i++){
      S0 = ((W[i-15] >> 7) | (W[i-15] << 25)) ^
           ((W[i-15] >> 18) | (W[i-15] << 14)) ^ (W[i-15] >> 3);
      S1 = ((W[i-2] >> 17) | (W[i-2] << 15)) ^
           ((W[i-2] >> 19) | (W[i-2] << 13)) ^ (W[i-2] >> 10);
      W[i] = W[i-16] + S0 + W[i-7] + S1;
    }

    a = H[0]; b = H[1]; c = H[2]; d = H[3];
    e = H[4]; f = H[5]; g = H[6]; h = H[7];

    for(i = 0; i < 64; i++){
      S1 = ((e >> 6) | (e << 26)) ^ ((e >> 11) | (e << 21)) ^ ((e >> 25) | (e << 7));
      ch = (e & f) ^ (~e & g);
      t1 = h + S1 + ch + K[i] + W[i];
      S0 = ((a >> 2) | (a << 30)) ^ ((a >> 13) | (a << 19)) ^ ((a >> 22) | (a << 10));
      maj = (a & b) ^ (a & c) ^ (b & c);
      t2 = S0 + maj;

      h = g; g = f; f = e; e = d + t1;
      d = c; c = b; b = a; a = t1 + t2;
    }

    H[0] += a; H[1] += b; H[2] += c; H[3] += d;
    H[4] += e; H[5] += f; H[6] += g; H[7] += h;
  }

  for(i = 0; i < 8; i++)
    for(int shift = 28; shift >= 0; shift -= 4)
      *out_hex++ = hex[(H[i] >> shift) & 0xf];
  *out_hex = 0;
}
```

```c
// called from forkret() during first process init. creates /etc directory and
// /etc/passwd if they don't exist, then seeds admin/patient1/doctor1 accounts.
void
auth_init(void)
{
  struct inode *ip;
  struct credential creds[3];

  begin_op();
  if((ip = namei("/etc")) == 0){
    if((ip = auth_create("/etc", T_DIR)) == 0)
      panic("auth_init: /etc");
    iunlockput(ip);
  } else {
    iput(ip);
  }
  end_op();

  begin_op();
  if((ip = namei(PASSWD_FILE)) == 0){
    if((ip = auth_create(PASSWD_FILE, T_FILE)) == 0)
      panic("auth_init: passwd");
    iunlockput(ip);
    end_op();
    seed_credential(&creds[0], "admin", 0, ROLE_ADMIN, "admin123");
    seed_credential(&creds[1], "patient1", 1, ROLE_PATIENT, "patient123");
    seed_credential(&creds[2], "doctor1", 2, ROLE_DOCTOR, "doctor123");
    if(write_passwd(creds, 3) < 0)
      panic("auth_init: seed");
    return;
  }
  iput(ip);
  end_op();
}

// reads /etc/passwd, hashes input pw, compares against stored hash.
// on success populates proc->uid/gid/role/username/authenticated.
// returns 0 on success, -1 on bad credentials.
int
auth_login(char *username, char *password)
{
  struct credential creds[MAX_USERS];
  char hash[HASH_LEN + 1];
  struct proc *p = myproc();
  int count = read_passwd(creds, MAX_USERS);

  if(count < 0 || emptystr(username) || emptystr(password))
    return -1;
  pw_hash(password, hash);
  for(int i = 0; i < count; i++){
    if(streq(username, creds[i].username) && streq(hash, creds[i].hash)){
      p->uid = creds[i].uid;
      p->gid = creds[i].gid;
      p->role = creds[i].role;
      safestrcpy(p->username, creds[i].username, sizeof(p->username));
      p->authenticated = 1;
      return 0;
    }
  }
  return -1;
}
```

```c
// admin-only (uid==0). appends one credential to /etc/passwd.
int
auth_useradd(char *username, char *password, int role)
{
  struct proc *p = myproc();
  struct credential creds[MAX_USERS];
  int count;

  if(p == 0 || p->uid != 0 || !p->authenticated)
    return -1;
  if(emptystr(username) || emptystr(password) || role < ROLE_ADMIN || role > ROLE_DOCTOR)
    return -1;
  count = read_passwd(creds, MAX_USERS);
  if(count < 0 || count >= MAX_USERS)
    return -1;
  for(int i = 0; i < count; i++){
    if(streq(username, creds[i].username))
      return -1;
  }
  safestrcpy(creds[count].username, username, sizeof(creds[count].username));
  creds[count].uid = role;
  creds[count].gid = role;
  creds[count].role = role;
  pw_hash(password, creds[count].hash);
  return write_passwd(creds, count + 1);
}

// admin-only. compacts credential array in-place, skipping the target entry.
// refuses to delete the admin account.
int
auth_userdel(char *username)
{
  struct proc *p = myproc();
  struct credential creds[MAX_USERS];
  int count, out = 0, found = 0;

  if(p == 0 || p->uid != 0 || !p->authenticated || emptystr(username))
    return -1;
  if(streq(username, "admin"))  // admin account is permanent
    return -1;
  count = read_passwd(creds, MAX_USERS);
  if(count < 0)
    return -1;
  for(int i = 0; i < count; i++){
    if(streq(username, creds[i].username)){
      found = 1;
      continue;
    }
    if(out != i)
      creds[out] = creds[i];
    out++;
  }
  if(!found)
    return -1;
  return write_passwd(creds, out);
}

// non-admin users must supply correct old password; admin can bypass old-pw check
// and reset anyone's password.
int
```

```c
auth_passwd(char *username, char *old_pw, char *new_pw)
{
  struct proc *p = myproc();
  struct credential creds[MAX_USERS];
  char old_hash[HASH_LEN + 1];
  int count;

  if(p == 0 || !p->authenticated || emptystr(username) || emptystr(new_pw))
    return -1;
  if(p->uid != 0 && !streq(username, p->username))  // non-admin can only change own pw
    return -1;
  count = read_passwd(creds, MAX_USERS);
  if(count < 0)
    return -1;
  pw_hash(old_pw, old_hash);
  for(int i = 0; i < count; i++){
    if(streq(username, creds[i].username)){
      if(p->uid != 0 && !streq(old_hash, creds[i].hash))
        return -1;
      pw_hash(new_pw, creds[i].hash);
      return write_passwd(creds, count);
    }
  }
  return -1;
}

// formats "username uid=X gid=Y role=Z\n" into caller-supplied buffer.
int
auth_whoami(char *buf, int bufsz)
{
  struct proc *p = myproc();
  int off = 0;

  if(p == 0 || !p->authenticated || bufsz <= 0)
    return -1;
  append_str(buf, &off, bufsz, p->username);
  append_str(buf, &off, bufsz, " uid=");
  append_int(buf, &off, bufsz, p->uid);
  append_str(buf, &off, bufsz, " gid=");
  append_int(buf, &off, bufsz, p->gid);
  append_str(buf, &off, bufsz, " role=");
  append_int(buf, &off, bufsz, p->role);
  append_char(buf, &off, bufsz, '\n');
  return off;
}
```

Phase 1: user/login.c

Listing: `login.c`

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

// reads one line from fd 0 (console). handles backspace (0x7f/DEL) for raw
// terminal editing — xv6 has no cooked mode or readline library.
static void
read_line(char *buf, int max)
{
```

```
  int i = 0;
  char c;

  while(i < max - 1){
    if(read(0, &c, 1) != 1)
      break;
    if(c == '\n' || c == '\r')
      break;
    if(c == '\b' || c == 0x7f){
      if(i > 0)
        i--;
      continue;
    }
    buf[i++] = c;
  }
  buf[i] = 0;
}

int
main(void)
{
  char username[16];
  char password[64];
  int failures = 0;
  char *argv[] = { "sh", 0 };

  printf("xv6 Medical Device OS - Secure Login\n");
  for(;;){
    printf("Username: ");
    read_line(username, sizeof(username));
    printf("Password: ");
    read_line(password, sizeof(password));

    if(login(username, password) == 0){
      exec("sh", argv);   // replace login with shell — discards this process
      printf("login: exec sh failed\n");
      exit(1);
    }

    failures++;
    printf("Login failed.\n");
    if(failures >= 3){
      // deliberate lockout: no reset, device must power-cycle. mimics
      // medical device security best practices for brute-force prevention.
      printf("Device locked after 3 failed attempts.\n");
      for(;;)
        pause(1000);
    }
  }
}
```

Phase 2: kernel/perms.c

Listing: `perms.c`

```
#include "types.h"
#include "riscv.h"
#include "defs.h"
#include "param.h"
```

```
#include "spinlock.h"
#include "sleeplock.h"
#include "fs.h"
#include "file.h"
#include "proc.h"
#include "perms.h"

// maps r/w/x + klass (0=owner,1=group,2=other) to the matching permission bit.
static int
access_bit(char access, int klass)
{
  if(klass == 0){
    if(access == 'r') return PERM_OWNER_READ;
    if(access == 'w') return PERM_OWNER_WRITE;
    if(access == 'x') return PERM_OWNER_EXEC;
  } else if(klass == 1){
    if(access == 'r') return PERM_GROUP_READ;
    if(access == 'w') return PERM_GROUP_WRITE;
    if(access == 'x') return PERM_GROUP_EXEC;
  } else {
    if(access == 'r') return PERM_OTHER_READ;
    if(access == 'w') return PERM_OTHER_WRITE;
    if(access == 'x') return PERM_OTHER_EXEC;
  }
  return 0;
}

// unix-style DAC: resolves caller to owner(0)/group(1)/other(2) and checks the
// matching mode bit. root (uid==0) always passes. resolution order matters:
// owner match takes priority over group match.
int
perm_check(struct inode *ip, char access)
{
  struct proc *p = myproc();
  int class;
  int bit;

  if(ip == 0)
    return 0;
  if(p == 0 || p->uid == 0)  // kernel threads and root bypass dac
    return 1;
  if(ip->uid == p->uid)
    class = 0;       // file owner
  else if(ip->gid == p->gid)
    class = 1;       // group member
  else
    class = 2;       // everyone else
  bit = access_bit(access, class);
  return bit != 0 && (ip->mode & bit) != 0;
}
```

Phase 2: medical file ownership in mkfs/mkfs.c

Listing: `mkfs.c (lines 294-306)`

```
void
add_medical_files(uint rootino)
{
  uint patient = mkdir_meta(rootino, "patient", 0755, 1, 1);
```

```
  uint dosage = mkdir_meta(rootino, "dosage", 0755, 2, 2);
  uint device = mkdir_meta(rootino, "device", 0755, 0, 0);
  uint audit = mkdir_meta(rootino, "audit", 0755, 0, 0);

  file_meta(patient, "records", "Patient record: glucose stable, follow-up required.\n", 0400, 1, 1);
  file_meta(dosage, "insulin.log", "Insulin dosage log initialized.\n", 0640, 2, 1);
  file_meta(device, "config", "Device config: basal_rate=1.0 safety_lock=on\n", 0600, 0, 0);
  file_meta(audit, "syscall.log", "Audit log placeholder; live events are kept in kernel memory.\n", 04
}
```

Phase 3: kernel/audit.c

Listing: `audit.c`

```
#include "types.h"
#include "riscv.h"
#include "defs.h"
#include "param.h"
#include "spinlock.h"
#include "proc.h"
#include "audit.h"

// 256-entry circular buffer. head is write position, tail is oldest valid entry.
// count tracks how many entries are live (clamped at AUDIT_BUF_SIZE).
static struct audit_entry ring[AUDIT_BUF_SIZE];
static int head;
static int tail;
static int count;
static struct spinlock audit_lock;

// called from main() early — before any process or syscall exists.
void
audit_init(void)
{
  initlock(&audit_lock, "audit");
  memset(ring, 0, sizeof(ring));
  head = 0;
  tail = 0;
  count = 0;
}

// called from syscall() on every invocation — success or failure. spinlock protects
// against concurrent writes from multiple cores. wraps around when full.
void
audit_log(int syscall_no, int result)
{
  struct proc *p = myproc();
  struct audit_entry *e;

  acquire(&audit_lock);
  e = &ring[head];
  e->pid = p ? p->pid : 0;
  e->uid = p ? p->uid : -1;
  e->syscall_no = syscall_no;
  e->result = result;
  e->tick = ticks;
  if(p)
    safestrcpy(e->comm, p->name, sizeof(e->comm));
  else
```

```
    e->comm[0] = 0;
  head = (head + 1) % AUDIT_BUF_SIZE;
  if(count < AUDIT_BUF_SIZE)
    count++;
  else
    tail = (tail + 1) % AUDIT_BUF_SIZE;
  release(&audit_lock);
}

// admin-only (uid==0). copies the oldest nentry entries into caller buffer.
// returns bytes written, -1 on permission denied.
int
audit_read(char *buf, int bufsz)
{
  struct proc *p = myproc();
  int copied = 0;
  int idx;

  if(p == 0 || p->uid != 0)
    return -1;
  acquire(&audit_lock);
  int capacity = bufsz / sizeof(struct audit_entry);
  int nentry = count;
  if(nentry > capacity)
    nentry = capacity;
  idx = (head - nentry + AUDIT_BUF_SIZE) % AUDIT_BUF_SIZE;
  for(int i = 0; i < nentry; i++){
    memmove(buf + copied, &ring[idx], sizeof(struct audit_entry));
    copied += sizeof(struct audit_entry);
    idx = (idx + 1) % AUDIT_BUF_SIZE;
  }
  release(&audit_lock);
  return copied;
}
```