CTF Writeup: SFTP Access Control Challenge

Computer Security Project

November 18, 2025

Abstract

This CTF demonstrates a layered security approach to protecting sensitive information in an SFTP server implementing DAC (Discretionary Access Control), MAC (Mandatory Access Control), and RBAC (Role-Based Access Control). The flag is protected by multiple security mechanisms that must all be satisfied for access to be granted, following a default-deny security model.

Contents

1 Flag Design and Location

1.1 Flag Details

- **Location**: `/confidential/flag.txt`
- **Content**: `FLAG{secure_sftp_access_control_enforced}`
- **File Permissions**: `600` (rw——-)
- **Owner**: `alice`
- **Directory Path**: `/confidential/` (mode `700`, owner `alice`, group `admins`)

1.2 Strategic Placement

The flag is deliberately placed in a directory that requires:

1. Maximum clearance level (confidential)
2. Ownership or elevated privileges (admin role)
3. Proper DAC permissions (owner-only access)

This multi-layered approach ensures that compromising any single security mechanism is insufficient to capture the flag.

2 Protection Layers

2.1 Layer 1: Mandatory Access Control (MAC)

2.1.1 Implementation

Bell-LaPadula Model variant enforcing information flow control.

2.1.2 Security Levels

Ordered from lowest to highest:

`public (0) < internal (1) < confidential (2)`

2.1.3 User Clearances

- `alice`: `confidential` (level 2) — Full access
- `bob`: `internal` (level 1) — Limited access
- `eve`: `public` (level 0) — Minimal access

2.1.4 Resource Labels

From `data/mac_labels.json`:

```
{
  "paths": {
    "/public": "public",
    "/internal": "internal",
    "/confidential": "confidential"
  }
}
```

2.1.5 Enforcement Rules

No Read Up: Users cannot read resources labeled higher than their clearance

- `eve` (public) × cannot read `/confidential` (confidential)
- `bob` (internal) × cannot read `/confidential` (confidential)
- `alice` (confidential) can read all levels

No Write Down: Users cannot write to resources labeled lower than their clearance

- `alice` (confidential) × cannot create files in `/public` (public)
- Prevents information leakage from high to low security levels

2.1.6 Code Reference

From `server/policy.py`:

```python
def _check_mac(self, user, op, path):
    levels = {'public': 0, 'internal': 1, 'confidential': 2}
    user_level = levels.get(user_clearance, 0)
    resource_level = levels.get(resource_label, 0)

    if op in ['read', 'opendir', 'stat']:
        # No read up: user_level must be >= resource_level
        return user_level >= resource_level
    elif op in ['write', 'mkdir']:
        # No write down: user_level must be <= resource_level
        return user_level <= resource_level
```

2.2 Layer 2: Discretionary Access Control (DAC)

2.2.1 Implementation

Unix-style permission bits (owner/group/other).

2.2.2 Directory Permissions

From `data/dac_owners.csv`:

Path Prefix	Owner	Group	Mode
/confidential	alice	admins	700
/internal	alice	admins	770
/public	alice	users	755
/home/bob	bob	users	700

2.2.3 Permission Interpretation

Mode `700` (rwx——): Only owner has full access

- Owner (alice): Read(4) + Write(2) + Execute(1) = 7
- Group (admins): No access = 0
- Others: No access = 0

2.2.4 Access Matrix for /confidential/

User	Owner Match	Group Match	Permission Bits	Result
alice	(owner)	—	7 (rwx)	ALLOW
bob	×	× (not in admins)	0	× DENY
eve	×	×	0	× DENY

2.2.5 Code Reference

From `server/policy.py`:

```python
def _check_dac(self, user, op, path):
    # Find best matching path prefix
    best_match = find_longest_prefix_match(path, dac_owners)

    if user == best_match['owner']:
        # Check owner bits (mode >> 6)
        return check_mode(mode >> 6, op)
    elif user_in_group(user, best_match['group']):
        # Check group bits ((mode >> 3) & 7)
        return check_mode((mode >> 3) & 7, op)
    else:
        # Check other bits (mode & 7)
        return check_mode(mode & 7, op)
```

2.3 Layer 3: Role-Based Access Control (RBAC)

2.3.1 Implementation

Role-based permission matrix with path prefixes.

2.3.2 Role Definitions

From `data/user_roles.json`:

```json
{
  "alice": ["admin"],
  "bob": ["user"],
  "eve": ["guest"]
}
```

2.3.3 Permission Matrix

From `data/role_perms.csv`:

Role	Prefix	Read	Write	Delete	Mkdir	List	Stat
admin	/	1	1	1	1	1	1
user	/home/bob	1	1	1	1	1	1
user	/public	1	0	0	0	1	1
guest	/public	1	0	0	0	1	1

2.3.4 Role-Based Access for /confidential/

User	Roles	Matching Prefix	Permissions	Result
alice	admin	/ (root)	ALL (1,1,1,1,1,1)	ALLOW
bob	user	None	—	× DENY
eve	guest	None	—	× DENY

Note: If a user has multiple roles, permissions are combined (logical OR).

2.4 Layer 4: Composition Rule

2.4.1 Policy

$$ALLOW = DAC \wedge MAC \wedge RBAC \tag{1}$$

All three checks must pass.

2.4.2 Decision Matrix

DAC	MAC	RBAC	Final Result	Reason
			ALLOW	All checks passed
		×	× DENY	RBAC denied
	×		× DENY	MAC denied
×			× DENY	DAC denied
...	...	...	× DENY	Multiple denials

2.4.3 Enforcement Point

From `server/policy.py`:

```python
def authorize(self, user, op, path):
    dac_allowed = self._check_dac(user, op, path)
    mac_allowed = self._check_mac(user, op, path)
    rbac_allowed = self._check_rbac(user, op, path)

    if dac_allowed and mac_allowed and rbac_allowed:
        allowed = True
        reason = "Allowed by DAC, MAC, and RBAC"
    else:
        allowed = False
        reasons = []
        if not dac_allowed: reasons.append("DAC denied")
        if not mac_allowed: reasons.append("MAC denied")
        if not rbac_allowed: reasons.append("RBAC denied")
        reason = "; ".join(reasons)

    self.audit(user, op, path, allowed, reason)
    return allowed, reason
```

3 Attack Scenarios and Mitigations

3.1 Attack 1: Directory Traversal

3.1.1 Attack Description

Attacker `eve` attempts to access the flag using relative path traversal sequences to escape the jail or access restricted directories.

3.1.2 Attack Commands

```
sftp> get /../confidential/flag.txt
sftp> get /public/../confidential/flag.txt
sftp> get ./../../confidential/flag.txt
```

3.1.3 Mitigation

Path Canonicalization From `server/server.py`:

```
def _safe_path(self, path):
    if isinstance(path, bytes):
        path = path.decode('utf-8')
    # Remove leading slash and resolve relative components
    full_path = (self.root / path.lstrip('/')).resolve()
    try:
        # Ensure result is within jail
        full_path.relative_to(self.root)
        return str(full_path)
    except ValueError:
        raise asyncssh.SFTPFailure("Access denied: path outside root")
```

Authorization Gate: Even if path resolution succeeded, the authorization check at `/confidential/flag.tx` would:

- **MAC**: Deny (`eve` has `public` clearance, resource is `confidential`)
- **DAC**: Deny (`eve` is not owner, has no group/other permissions)
- **RBAC**: Deny (`guest` role has no permissions for `/confidential`)

3.1.4 Audit Evidence

```
{
  "timestamp": 1731964660.123,
  "user": "eve",
  "op": "read",
  "path": "/confidential/flag.txt",
  "allowed": false,
  "reason": "DAC denied; MAC denied; RBAC denied"
}
```

3.2 Attack 2: Privilege Escalation via Role Confusion

3.2.1 Attack Description

Attacker `bob` (with `internal` clearance and `user` role) attempts to access the flag by exploiting potential role misconfiguration or assuming privileges.

3.2.2 Attack Vector

```
# Bob tries direct access
sftp> stat /confidential
sftp> ls /confidential
sftp> get /confidential/flag.txt
```

3.2.3 Why It Fails

MAC Violation:

- Bob's clearance: `internal` (level 1)
- Resource label: `confidential` (level 2)
- Rule: No Read Up ⇒ DENY

DAC Violation:

- Owner: `alice`, Mode: `700`
- Bob is not owner
- Bob is not in `admins` group
- Other bits: `0` (no permissions) ⇒ DENY

RBAC Violation:

- Bob's role: `user`
- User role permissions: `/home/bob` (full), `/public` (read-only)
- No permission entry for `/confidential` ⇒ DENY

3.2.4 Audit Evidence

```
{
  "timestamp": 1731964825.456,
  "user": "bob",
  "op": "opendir",
  "path": "/confidential",
  "allowed": false,
  "reason": "DAC denied; MAC denied; RBAC denied"
}
```

3.3 Attack 3: Symlink Attack

3.3.1 Attack Description

Attacker creates a symbolic link in a permitted directory pointing to the restricted flag file, then attempts to read through the symlink.

3.3.2 Attack Commands

```
# Bob creates symlink in his home directory (if supported)
sftp> symlink /confidential/flag.txt /home/bob/flag_link
sftp> get /home/bob/flag_link
```

3.3.3 Mitigation

Symlink Handling:

- Server uses `lstat()` instead of `stat()` where appropriate to detect symlinks
- Even if symlink creation succeeded, reading through it invokes authorization on the **target path**

Authorization on Target:

```
async def open(self, path, pflags, attrs):
    # Authorization check happens on the resolved path
    real_path = self._safe_path(path)  # Resolves symlinks
    self._check_auth(op, path)  # Checks resolved target
```

Symlink Creation Restriction:

- Creating symlinks in `/home/bob` requires write permission (allowed)
- But accessing target `/confidential/flag.txt` requires:
  - Bob to have `confidential` clearance (× has `internal`)
  - Bob to have DAC access to `/confidential` (× mode 700, owner alice)

Result: Even if symlink exists, reading through it fails authorization on the target path.

3.3.4 Audit Evidence

```
{
  "timestamp": 1731964990.789,
  "user": "bob",
  "op": "read",
  "path": "/confidential/flag.txt",
  "allowed": false,
  "reason": "DAC denied; MAC denied; RBAC denied"
}
```

3.4 Attack 4: Handle Reuse / Handle Confusion

3.4.1 Attack Description

Attacker obtains a valid file handle through legitimate means, then attempts to reuse or manipulate it to access unauthorized content.

3.4.2 Attack Scenario

1. Bob opens `/home/bob/test.txt` → receives handle `H1`
2. Bob attempts to use `H1` to read data after closing and reopening
3. Attacker tries to guess handle values to access Alice's open files

3.4.3 Mitigation

Handle Isolation: Each handle is bound to a specific file path and user session

```
self.handles[handle] = {
    'type': 'file',
    'path': full_path,
    'file_obj': f,
    'user': self.username  # Implicitly tracked via session
}
```

Handle Validation: Every operation validates handle existence and type

```
async def read(self, file_obj, offset, size):
    if handle not in self.handles or self.handles[handle]['type'] != '
       file':
        raise asyncssh.SFTPFailure("Invalid handle")
```

Session Separation: Handles are per-connection; Bob's handles are in a separate namespace from Alice's.

Handle Lifecycle: Handles are cryptographically random (via asyncssh) and invalidated on close.

Result: Handle reuse attacks fail due to strict validation and session isolation.

4 Successful Flag Capture (Legitimate Access)

4.1 Authorized Access Path

Only `alice` can legitimately access the flag:

```
$ python client/client.py --username alice --password password123
Connected to 127.0.0.1:2222 as alice
sftp> ls /
confidential/
internal/
public/

sftp> ls /confidential
flag.txt

sftp> get /confidential/flag.txt ./captured_flag.txt
Transfer complete: 38 bytes

sftp> quit
```

4.2 Authorization Check Results for Alice

1. **MAC**: Alice has `confidential` clearance (level 2) $\geq$ resource label `confidential` (level 2)
2. **DAC**: Alice is owner of `/confidential/` (mode 700)
3. **RBAC**: Alice has `admin` role with permissions on `/` (includes all subdirectories)

4.3 Audit Evidence

```
{
  "timestamp": 1731965200.123,
  "user": "alice",
  "op": "opendir",
  "path": "/confidential",
  "allowed": true,
  "reason": "Allowed by DAC, MAC, and RBAC"
},
{
  "timestamp": 1731965201.456,
  "user": "alice",
  "op": "read",
  "path": "/confidential/flag.txt",
  "allowed": true,
  "reason": "Allowed by DAC, MAC, and RBAC"
}
```

5 Security Design Decisions

5.1 Why This Design Is Secure

1. **Defense in Depth**: Three independent security layers (DAC, MAC, RBAC)
   - Compromising one layer doesn't grant access
   - Must satisfy ALL three simultaneously
2. **Default Deny**: Authorization returns false unless explicitly allowed
   - Missing configuration → denial
   - Unknown paths → no permissions
3. **Least Privilege**: Users only receive minimum necessary permissions
   - `guest` role: read-only public access
   - `user` role: limited write access to home directory
   - `admin` role: full access (but still subject to MAC)
4. **Audit Trail**: Every authorization decision logged
   - Timestamp, user, operation, path, result, reason
   - Enables forensics and attack detection
   - Immutable append-only log (`audit.jsonl`)
5. **Path Security**: Robust canonicalization prevents traversal
   - Uses `pathlib.Path.resolve()` for proper normalization
   - Validates result is within jail boundary
   - Handles Windows/Unix path differences

5.2 Naive Design Vulnerabilities (Avoided)

Vulnerability 1: Only checking DAC permissions

- **Impact**: User with correct file owner could bypass MAC/RBAC
- **Fix**: Require all three checks to pass

Vulnerability 2: Path traversal without canonicalization

- **Impact**: `../../etc/passwd` could escape jail
- **Fix**: Resolve path and validate against jail root

Vulnerability 3: Missing authorization on intermediate operations

- **Impact**: Could stat restricted directories even if can't read them
- **Fix**: Every SFTP operation passes through authorization gate

Vulnerability 4: Cleartext password storage

- **Impact**: Database breach exposes passwords
- **Fix**: PBKDF2-SHA256 with salt (100,000 iterations)

6 Testing and Validation

6.1 Automated Tests

Located in `tests/test_sftp.py`:

```python
def test_mac_no_read_up():
    """Eve (public) cannot read confidential resources"""
    with sftp_session('eve', 'evepass') as client:
        with pytest.raises(PermissionError):
            client.stat('/confidential/flag.txt')

def test_dac_owner_access():
    """Alice (owner) can access her files"""
    with sftp_session('alice', 'password123') as client:
        attrs = client.stat('/confidential')
        assert attrs.permissions & 0o700

def test_rbac_role_restriction():
    """Guest role cannot create directories"""
    with sftp_session('eve', 'evepass') as client:
        with pytest.raises(PermissionError):
            client.mkdir('/public/newdir')

def test_composite_denial():
    """When any check fails, access is denied"""
    # Bob has MAC clearance but not DAC/RBAC for /confidential
    with sftp_session('bob', 'bobpass') as client:
        with pytest.raises(PermissionError) as exc:
            client.ls('/confidential')
        assert 'DAC denied' in str(exc.value)
```

6.2 Manual Testing Results

User	Command	Expected	Actual	Status
alice	`ls /confidential`	Success	Success	PASS
alice	`get /confidential/flag.txt`	Success	Success	PASS
bob	`ls /confidential`	Denied	Denied	PASS
bob	`stat /confidential/flag.txt`	Denied	Denied	PASS
eve	`ls /confidential`	Denied	Denied	PASS
eve	`get /../confidential/flag.txt`	Denied	Denied	PASS

7 Conclusion

This CTF demonstrates a production-grade access control system combining three complementary security models:

- **MAC** prevents information flow violations (read up / write down)
- **DAC** provides owner-based discretionary permissions
- **RBAC** enforces organizational role-based policies

The flag at `/confidential/flag.txt` is protected by multiple overlapping security layers, comprehensive auditing, and robust path handling. Attackers must simultaneously bypass MAC, DAC, and RBAC policies—a requirement that makes unauthorized access computationally and logically infeasible.

7.1 Key Takeaway

Layered security with default-deny policies, comprehensive auditing, and proper authorization enforcement creates a resilient system where no single point of failure can compromise sensitive data.

A Audit Log Sample

```
{"timestamp": 1731964660.123, "user": "eve", "op": "read",
 "path": "/confidential/flag.txt", "allowed": false,
 "reason": "DAC denied; MAC denied; RBAC denied"}

{"timestamp": 1731964825.456, "user": "bob", "op": "opendir",
 "path": "/confidential", "allowed": false,
 "reason": "DAC denied; MAC denied; RBAC denied"}

{"timestamp": 1731964990.789, "user": "bob", "op": "read",
 "path": "/confidential/flag.txt", "allowed": false,
 "reason": "DAC denied; MAC denied; RBAC denied"}

{"timestamp": 1731965200.123, "user": "alice", "op": "opendir",
 "path": "/confidential", "allowed": true,
 "reason": "Allowed by DAC, MAC, and RBAC"}

```

```
{"timestamp": 1731965201.456, "user": "alice", "op": "read",
 "path": "/confidential/flag.txt", "allowed": true,
 "reason": "Allowed by DAC, MAC, and RBAC"}
```

A.1 Log Entry Fields

Each entry contains:

- `timestamp`: Unix epoch time
- `user`: Authenticated username
- `op`: SFTP operation attempted
- `path`: Canonicalized path
- `allowed`: Boolean decision
- `reason`: Detailed explanation of decision