MCP SECURITY IMPLEMENTATION KIT

Treat MCP Like SSH

Red Asgard Security Research

April 2026

Contents

INTRODUCTION

This kit provides practical templates, policy examples, and checklists for securing MCP (Model Context Protocol) deployments.

Core Principle: Treat MCP Like SSH

MCP is a privileged protocol that bridges AI-generated intent with system execution. It deserves the same security rigor as SSH:

- Require authentication on every MCP server
- Log all MCP sessions comprehensively
- Apply least-privilege access controls
- Monitor for anomalous usage patterns

What's Included

1. **Implementation Templates** — Copy-pasteable code for authentication, policy gates, validation
2. **Policy Examples** — YAML configurations for read-only, scoped-write, and deny-by-default policies
3. **Logging Checklist** — Essential fields for MCP audit logs
4. **Quick Start Guide** — 5-step implementation path
5. **Production Readiness Checklist** — Pre-deployment verification

IMPLEMENTATION TEMPLATES

> ⚠️ **Security Notice:** These are starter templates requiring customization and security review before production use.

1. Authentication

API key authentication for MCP servers with timing-attack resistant comparison:

```
import hashlib
import secrets
from typing import Optional, Dict

class MCPAuthenticator:
    """Authentication for MCP server requests."""

    def __init__(self, api_keys: Dict[str, str]):
        """Initialize authenticator with API keys."""
        self.api_key_hashes: Dict[str, str] = {}

        for actor_id, key in api_keys.items():
            if key.startswith("sha256:"):
                self.api_key_hashes[actor_id] = key[7:]
            else:
                self.api_key_hashes[actor_id] = \
                    hashlib.sha256(key.encode()).hexdigest()

    def authenticate(self, api_key: str) -> Optional[str]:
        """Verify API key and return actor ID if valid."""
        if not api_key:
            return None

        key_hash = hashlib.sha256(api_key.encode()).hexdigest()

        for actor_id, stored_hash in self.api_key_hashes.items():
            if secrets.compare_digest(key_hash, stored_hash):
                return actor_id

        return None

# Usage
authenticator = MCPAuthenticator({
    "assistant-service": "secret_key_here"
})

actor = authenticator.authenticate(
    request.headers.get("X-API-Key")
)
if not actor:
    return {"error": "authentication_failed"}, 401
```

2. Policy Gate

External policy gate that evaluates tool calls before execution:

```
import json
import logging
from pathlib import Path
from typing import Dict, Any

logger = logging.getLogger(__name__)

class PolicyGate:
    def __init__(self, policy_config_path: str):
        """Initialize policy gate from JSON config."""
        with open(policy_config_path, 'r') as f:
            self.config = json.load(f)

        if "allowed_tools" not in self.config:
            raise ValueError("Missing: allowed_tools")

        self.allowed_tools = self.config.get(
            "allowed_tools", {}
        )

    def evaluate_tool_call(self, actor: str,
                           tool_name: str,
                           arguments: Dict[str, Any]) -> str:
        """Returns ALLOW or DENY."""

        self._log_request(actor, tool_name, arguments)

        # Check allowlist
        if tool_name not in \
           self.allowed_tools.get(actor, []):
            return "DENY"

        # Validate by tool type
        if tool_name == "read_file":
            if not self._validate_file_path(
                arguments.get("path", "")
            ):
                return "DENY"

        if tool_name == "query_database":
            if not self._validate_sql_query(
                arguments.get("query", "")
            ):
                return "DENY"

        return "ALLOW"

    def _validate_file_path(self, path: str) -> bool:
        """Prevent path traversal attacks."""
        try:
            base = Path("data").resolve()
            requested = (base / path).resolve()
            requested.relative_to(base)
            return True
        except (ValueError, RuntimeError):
            return False

    def _validate_sql_query(self, query: str) -> bool:
```

```
        """Basic validation - defense in depth only.

        This is NOT a substitute for parameterized
        queries or constrained query APIs. Real
        protection requires using query builders or
        stored operations that don't accept raw
        model-authored SQL.
        """
        import re

        query_clean = re.sub(r'--[^\n]*', '', query)
        query_clean = re.sub(
            r'/\*.*?\*/', '',
            query_clean,
            flags=re.DOTALL
        )
        query_clean = ' '.join(
            query_clean.split()
        ).upper()

        if not query_clean.startswith("SELECT"):
            return False

        dangerous = [
            "DROP", "DELETE", "INSERT",
            "UPDATE", "ALTER", "CREATE"
        ]
        for keyword in dangerous:
            if re.search(rf'\b{keyword}\b',
                         query_clean):
                return False

        return True
```

Policy Configuration (policy.json):

```
{
  "allowed_tools": {
    "assistant-service": [
      "read_file",
      "query_database"
    ],
    "admin-service": [
      "read_file",
      "write_file"
    ]
  },
  "tool_policies": {
    "read_file": {
      "allowed_paths": [
        "data/public/",
        "docs/"
      ]
    },
    "query_database": {
      "allowed_tables": [
        "users",
        "products"
      ]
    }
  }
}
```

3. Schema Validation

Strict schema validation with Pydantic:

```
from pydantic import BaseModel, field_validator
from pathlib import Path

class FileArgs(BaseModel):
    path: str

    @field_validator('path')
    @classmethod
    def validate_path(cls, v: str) -> str:
        """Prevent path traversal."""
        if len(v) > 500:
            raise ValueError("Path too long")

        if not v:
            raise ValueError("Path empty")

        try:
            base = Path("data").resolve()
            requested = (base / v).resolve()
            requested.relative_to(base)
            return v
        except (ValueError, RuntimeError):
            raise ValueError("Invalid path")

class QueryArgs(BaseModel):
    query: str
    max_rows: int = 100

    @field_validator('query')
    @classmethod
    def validate_query(cls, v: str) -> str:
        """Enforce read-only queries."""
        if len(v) > 5000:
            raise ValueError("Query too long")

        query_upper = v.strip().upper()

        if not query_upper.startswith("SELECT"):
            raise ValueError("Only SELECT")

        dangerous = [
            "DROP", "DELETE", "INSERT"
        ]
        for keyword in dangerous:
            if keyword in query_upper:
                raise ValueError(
                    f"{keyword} not allowed"
                )

        return v
```

4. Rate Limiting

Sliding-window rate limiter for MCP tool calls:

```
from collections import defaultdict
from datetime import datetime, timedelta, timezone
from typing import Dict, Tuple

class RateLimiter:
    def __init__(self,
                 calls_per_minute: int = 10,
                 calls_per_hour: int = 100):
        self.calls_per_minute = calls_per_minute
        self.calls_per_hour = calls_per_hour
        self.minute_buckets: Dict[str, list] = \
            defaultdict(list)
        self.hour_buckets: Dict[str, list] = \
            defaultdict(list)

    def check_rate_limit(
        self,
        actor: str,
        tool_name: str
    ) -> Tuple[bool, str]:
        """Check if request is within limits."""
        now = datetime.now(timezone.utc)
        key = f"{actor}:{tool_name}"

        # Clean old entries
        self.minute_buckets[key] = [
            t for t in self.minute_buckets[key]
            if now - t < timedelta(minutes=1)
        ]

        if len(self.minute_buckets[key]) >= \
           self.calls_per_minute:
            return False, "rate_limit_minute"

        self.minute_buckets[key].append(now)
        return True, "ok"
```

5. Network Policy

Enforce network-level access controls:

```
import ipaddress
from urllib.parse import urlparse

class NetworkPolicy:
    """Starter network policy enforcement.

    This is a starter control, not complete SSRF
    protection. Production systems need additional
    controls for redirects, hostname re-resolution,
    protocol restrictions, and allowlist enforcement.
    """

    METADATA_SERVICES = {
        "169.254.169.254",
        "metadata.google.internal"
    }

    def validate_url(self, url: str) -> bool:
        """Validate URL against policy."""
```

```
        parsed = urlparse(url)

        # Block metadata services
        if parsed.hostname in \
           self.METADATA_SERVICES:
            return False

        # Block private IPs
        try:
            ip = ipaddress.ip_address(
                parsed.hostname
            )
            if ip.is_private or ip.is_loopback:
                return False
        except ValueError:
            pass

        return True
```

6. PII Sanitization

Sanitize sensitive data before logging:

```
import re
import copy
import hashlib

class LogSanitizer:
    @staticmethod
    def sanitize_arguments(
        arguments: dict
    ) -> dict:
        """Redact sensitive data."""
        sanitized = copy.deepcopy(arguments)

        def _redact_recursive(obj):
            if isinstance(obj, dict):
                for key, value in obj.items():
                    if key.lower() in [
                        'password', 'api_key',
                        'token', 'secret'
                    ]:
                        obj[key] = "[REDACTED]"
                    elif isinstance(value, str) \
                         and '@' in value:
                        obj[key] = hashlib.sha256(
                            value.encode()
                        ).hexdigest()[:16]

        _redact_recursive(sanitized)
        return sanitized
```

POLICY EXAMPLES

Read-Only Tool Policy

Allow read-only access to approved resources:

```yaml
actors:
  - id: "assistant-service"
    allowed_tools:
      - "read_file"
      - "query_database"

    constraints:
      read_file:
        allowed_paths:
          - "data/public/**"
          - "docs/**"
        denied_paths:
          - "**/.env"
          - "**/*.key"

      query_database:
        allowed_operations: ["SELECT"]
        max_rows: 1000
        timeout_seconds: 30

audit:
  log_all_requests: true
  alert_on_deny: true
```

Scoped Write Action Policy

Permit writes only for approved actors:

```yaml
actors:
  - id: "content-agent"
    allowed_tools:
      - "read_file"
      - "write_file"

    constraints:
      write_file:
        allowed_paths:
          - "content/drafts/**"
        denied_paths:
          - "content/published/**"
        allowed_extensions:
          - ".md"
          - ".txt"
        max_file_size: 1048576

rate_limits:
```

```
  write_file:
    max_calls_per_minute: 10
```

Deny-by-Default Policy

Any tool call without explicit allow is denied:

```
default_action: "DENY"

global_deny_list:
  - "execute_shell"
  - "delete_file"
  - "drop_table"

actors:
  - id: "research-agent"
    allowed_tools:
      - "read_file"
      - "search_documents"

    constraints:
      read_file:
        allowed_paths:
          - "research/**"
        denied_extensions:
          - ".exe"
          - ".sh"

error_handling:
  on_validation_error: "DENY"
  on_policy_error: "DENY"
  on_timeout: "DENY"
```

LOGGING CHECKLIST

Essential Fields for MCP Audit Logs

Identity & Context:

- `actor` — Who/what made the request
- `timestamp` — ISO 8601 with timezone
- `correlation_id` — Related events ID
- `session_id` — AI agent session ID
- `source_ip` — Origin IP address

Tool Invocation:

- `tool_name` — Exact tool invoked
- `arguments` — Parameters (sanitize PII)
- `server_id` — MCP server ID

Policy & Authorization:

- `policy_decision` — ALLOW/DENY
- `policy_rule_matched` — Rule applied
- `auth_method` — Authentication method

Execution:

- `status` — SUCCESS/FAILURE/TIMEOUT
- `error` — Error message if failed
- `duration_ms` — Execution time
- `result_metadata` — Size/count metrics

AI Context:

- `model_output_ref` — AI reasoning hash
- `context_sources` — AI influences
- `end_user_id` — Actual user
- `effective_principal` — Service account

> **Critical:** Track whether operations run as shared service account or preserve user identity.

Example Log Entry

```
{
  "timestamp": "2026-04-19T10:30:00Z",
  "correlation_id": "req-abc-123",
  "actor": "assistant-service",
  "tool_name": "query_database",
  "arguments": {
    "query": "SELECT * FROM users",
    "max_rows": 100
  },
  "policy_decision": "ALLOW",
  "auth_method": "api_key",
  "status": "SUCCESS",
  "duration_ms": 245,
  "end_user_id": "alice@company.com",
  "effective_principal": "mcp-svc"
}
```

QUICK START GUIDE

5 Steps to Secure MCP Deployment

1. Inventory Deployments (15 min)

- List all MCP servers
- Document exposed tools
- Identify backend system access

2. Require Authentication (30 min)

- Require auth on all servers
- Use service accounts
- Rotate credentials regularly

3. Deploy Policy Gate (1-2 hours)

- Use policy gate template
- Start with deny-by-default
- Add explicit allowlist
- Test read-only operations first

4. Add Logging (1 hour)

- Implement logging checklist
- Send logs to SIEM
- Create baseline alerts

5. Validate Input (2-4 hours)

- Apply strict schemas
- Use Pydantic validation
- Test with adversarial inputs

PRODUCTION READINESS CHECKLIST

Before production deployment:

- [ ] Customize authentication
- [ ] Review policy rules
- [ ] Test validation logic
- [ ] Configure logging
- [ ] Set rate limits
- [ ] Enable TLS
- [ ] Test fail-closed behavior
- [ ] Add monitoring
- [ ] Document policies
- [ ] Security review

> **Important:** Customize, test, and review before deployment.

WHAT WE FIND IN ASSESSMENTS

Common findings we observe in MCP security assessments:

- Most MCP servers lack authentication
- Structured audit logging is rarely implemented
- Command injection vulnerabilities are frequent
- Unrestricted URL fetch capabilities create SSRF risk

These issues are fixable. This kit provides the building blocks.

NEED HELP?

MCP Security Assessment

Red Asgard offers specialized assessments:

- MCP server discovery
- Authentication testing
- Input validation analysis
- Logging gap analysis

Contact: contact@redasgard.com
Web: https://redasgard.com

MCP Security Implementation Kit
Red Asgard Security Research
April 2026

For latest version: https://redasgard.com/mcp-security