ESP Educational Guide

Learning the Endpoint State Policy Language

A Complete Hands-On Tutorial

Version 1.0
November 2025

Table of Contents

Part 1: Introduction & Setup

1.1 Welcome to ESP

Welcome to the ESP (Endpoint State Policy) Educational Guide! This comprehensive tutorial will take you from complete beginner to proficient ESP policy writer through hands-on examples and practical exercises.

What is ESP?

ESP is a **compliance-as-data fabric language** designed to express security and compliance rules in a structured, machine-readable format. Unlike traditional compliance tools that mix policy and execution code, ESP treats policies as *pure data definitions* that can be:

- Validated automatically by compliance scanners
- Versioned and tracked like any other data
- Reused across different platforms and environments
- Audited and reviewed like any other document

Why Learn ESP?

If you work in cybersecurity, compliance, or IT operations, ESP provides several key advantages:

Benefit	Description
Universal Language	Write once, apply everywhere - Linux, Windows, cloud, containers
Declarative	Define WHAT should be true, not HOW to check it
Version Control	Track policy changes over time like code
Auditable	Human-readable policies that can be reviewed and approved

> ***Note:*** *ESP is platform-agnostic, meaning you write policies once and apply them across Linux, Windows, cloud environments, and containers.*

1.2 How This Guide Works

This guide is designed for **progressive learning**. Each section builds on the previous one, introducing concepts incrementally with plenty of hands-on practice.

Your Learning Path

Part	Time	What You'll Learn
Part 1	30 min	Introduction and environment setup
Part 2	1 hour	Core concepts: Objects, States, Criteria
Part 3	1.5 hours	Building your first complete policy with exercises

Part	Time	What You'll Learn
Part 4	2 hours	Variables, multiple checks, logic operators
Part 5	2 hours	Advanced: Sets, filters, runtime operations
Part 6	2 hours	Real-world STIG and CIS implementations
Part 7-8	1 hour	Troubleshooting and quick reference

> ***Time Commitment:*** *Plan for 6-10 hours to complete this guide thoroughly. You can work through sections at your own pace.*

1.3 Prerequisites

To get the most out of this guide, you should have:

- **Basic understanding of IT security concepts** (file permissions, services, packages)
- **Familiarity with compliance frameworks** (STIG, CIS, or NIST - helpful but not required)
- **Visual Studio Code** with the Dev Containers extension installed
- **Docker Desktop** or Docker Engine running on your system
- **Git** for cloning the repository

> ***New to compliance?*** *Don't worry! This guide explains compliance concepts as we encounter them.*

1.4 Setting Up Your Environment

The ESP project uses a **devcontainer** for a consistent development environment. This ensures everyone has the same tools and dependencies.

Installation Steps

Step 1: Install Prerequisites

1. Install Docker Desktop: https://www.docker.com/products/docker-desktop
2. Install Visual Studio Code: https://code.visualstudio.com/
3. Install the Dev Containers extension in VS Code

Step 2: Clone the Repository

```
git clone https://github.com/CurtisSlone/Endpoint-State-Policy.git
cd Endpoint-State-Policy
```

Step 3: Open in VS Code

```
code .
```

Step 4: Start the Dev Container

When VS Code opens, you'll see a prompt to 'Reopen in Container'. Click it, or:

1. Press F1 (or Ctrl+Shift+P / Cmd+Shift+P)
2. Type 'Dev Containers: Reopen in Container'
3. Press Enter

The container will build (first time takes a few minutes) and you'll have a complete ESP development environment.

Step 5: Verify Installation

Open a terminal in VS Code and test with a sample policy:

```
cd esp_scanner_sdk
cargo run -- esp/set_test.esp
```

You should see the scanner compile and execute the policy, showing compliance results.

Scanner Usage

The ESP scanner accepts two types of arguments:

Single Policy Scan:

```
cargo run -- path/to/policy.esp
```

Batch Directory Scan:

```
cargo run -- path/to/policies/
```

This scans all .esp files in the directory.

Configuring Logging Levels

ESP provides detailed logging to help you understand what's happening during compilation and scanning. You can control the verbosity using the **`ESP_LOGGING_MIN_LEVEL`** environment variable.

Level	What You See
`debug`	Everything - parser tokens, symbol tracking, validation steps (verbose)
`info`	Phase completions, scan results, major events (default)
`warning`	Potential issues, deprecated features, non-critical problems
`error`	Only critical errors that prevent execution

Setting the Log Level

Linux/Mac:

```
export ESP_LOGGING_MIN_LEVEL=debug
cargo run -- esp/my-policy.esp
```

Windows PowerShell:

```
$env:ESP_LOGGING_MIN_LEVEL="debug"
cargo run -- esp/my-policy.esp
```

Or set it permanently in your .env file:

```
# In the project root
```

```
echo 'ESP_LOGGING_MIN_LEVEL=debug' >> .env
```

Example: Debug Output

With `ESP_LOGGING_MIN_LEVEL=debug`, you'll see detailed compiler phases:

```
[DEBUG] D000 - Expecting identifier
[DEBUG] D000 - Identifier matched
[DEBUG] D000 - Parser advanced
[INFO] I040 - ESP file parsing completed successfully
[DEBUG] D000 - Symbol discovery using global logging
[INFO] I050 - Symbol collection completed successfully
[DEBUG] D000 - Starting cycle detection
[INFO] I060 - Reference validation completed
```

This visibility into the compiler's multi-pass process helps you understand:

- Which compilation pass is running
- How symbols are being discovered and resolved
- Where validation checks occur
- Performance timing for each phase

Ready to Go! *Your environment is set up. Let's start learning ESP in the next section.*

Part 2: ESP Fundamentals

In this section, you'll learn the core building blocks of ESP. We'll start with simple concepts and gradually build your understanding through hands-on examples.

2.1 The Big Picture: How ESP Works

Before diving into syntax, let's understand *what ESP does* and *how it works*.

The ESP Workflow

Step	What Happens
1. Write Policy	You define what should be checked (file permissions, service status, etc.)
2. Parse	The scanner reads your ESP file and validates syntax
3. Collect Data	The scanner gathers the actual state of your system
4. Compare	The scanner compares actual state against your policy requirements
5. Report	You get a compliance report showing PASS or FAIL for each check

> ***Key Insight:*** *ESP separates WHAT you want to check (the policy) from HOW to check it (the scanner). This separation makes policies portable and reusable.*

2.2 Your First ESP Policy

Let's write the simplest possible ESP policy that actually does something useful. We'll check if the `/etc/passwd` file has secure permissions.

Example 1: File Permission Check

Create a file called `passwd-check.esp`:

```
DEF
  STATE secure_permissions
    permissions string = `0644`
  STATE_END

  OBJECT etc_passwd
    path `/etc`
    filename `passwd`
  OBJECT_END

  CRI AND
    CTN permission_check
      TEST all all
```

```
        STATE_REF secure_permissions
        OBJECT_REF etc_passwd
      CTN_END
    CRI_END
DEF_END
```

Let's Break It Down

Every line in this policy has a purpose. Here's what each part does:

Code Part	Purpose
`DEF...DEF_END`	Wraps the entire policy definition
`STATE...STATE_END`	Defines the expected condition (permissions = 0644)
`OBJECT...OBJECT_END`	Identifies what to check (/etc/passwd file)
`CRI...CRI_END`	Groups criteria together with logic (AND, OR)
`CTN...CTN_END`	A single compliance test that connects STATE + OBJECT
`TEST all all`	How to evaluate (check all objects, all must match)

Try It Yourself

Run this policy with the scanner:

```
cargo run -- passwd-check.esp
```

You should see output like:

```
✓ PASS: /etc/passwd has correct permissions (0644)
```

> ***Exercise:*** *Modify the policy to check for permissions 0640 instead. What happens when you run it?*

2.3 Understanding Objects

Think of **OBJECT** as your *target*. It's the thing on your system you want to check - a file, a service, a package, or a configuration setting.

Object Structure

Every object has an identifier and fields that help locate it:

```
OBJECT identifier_name
  field_name `value`
  another_field `another_value`
OBJECT_END
```

Common Object Examples

Example 1: A File Object

```
OBJECT ssh_config
  path `/etc/ssh`
  filename `sshd_config`
OBJECT_END
```

Example 2: A Package Object

```
OBJECT openssh_package
  package_name `openssh-server`
OBJECT_END
```

Example 3: A Service Object

```
OBJECT firewall_service
  service_name `firewalld`
OBJECT_END
```

> ***Key Point:*** *The fields inside an object depend on what TYPE of object it is. Files use path/filename, packages use package_name, services use service_name.*

2.4 Understanding States

A **STATE** defines *what should be true* about your object. It's your **expected condition**.

State Structure

States contain field checks with operators:

```
STATE identifier_name
  field_name type operator `value`
STATE_END
```

Understanding Operators

Operator	Meaning	Example
`=`	Equals	`owner string = `root``
`!=`	Not equals	`status string != `disabled``
`contains`	String contains	`content string contains `error``
`>`	Greater than	`size int > 1000`
`<`	Less than	`size int < 5000`
`>=`	Greater or equal	`version string >= `2.0``

State Examples

Example 1: File Permission State

```
STATE secure_file
  permissions string = `0600`
  owner string = `root`
STATE_END
```

Example 2: Service State

```
STATE service_running
  status string = `active`
  enabled boolean = true
STATE_END
```

Example 3: File Content State

```
STATE required_config
  content string contains `PermitRootLogin no`
STATE_END
```

> ***Exercise:*** *Write a state that checks if a file size is greater than 1000 bytes. Answer at end of section.*

2.5 Connecting Objects and States with CTN

Now comes the magic: **`CTN (Criterion)`** connects an OBJECT with a STATE to create an actual compliance check.

The CTN Pattern

Every CTN follows this pattern:

```
CTN identifier_name
  TEST existence state_logic
  STATE_REF state_identifier
  OBJECT_REF object_identifier
```

```
CTN_END
```

Understanding TEST

The **`TEST`** line tells ESP *how* to evaluate the check. It has two parts:

Part	Options
Existence	**`all`** - All objects must exist **`any`** - At least one object must exist **`none`** - No objects should exist **`at_least_one`** - One or more must exist
State Logic	**`all`** - All state conditions must match **`any`** - At least one condition must match

Complete CTN Example

Let's put it all together:

```
# Define what we're checking
OBJECT shadow_file
  path `/etc`
  filename `shadow`
OBJECT_END

# Define what should be true
STATE locked_down
  permissions string = `0000`
  owner string = `root`
STATE_END

# Create the check
CRI AND
  CTN shadow_check
    TEST all all
    STATE_REF locked_down
    OBJECT_REF shadow_file
  CTN_END
CRI_END
```

What This Checks: The /etc/shadow file exists, has permissions 0000, and is owned by root.

Exercise Answer: *STATE file_size_check size int > 1000STATE_END*

Part 3: Building Your First Policy

Now that you understand the core concepts, let's build a complete, real-world compliance policy from scratch. We'll tackle a common security requirement: SSH hardening.

3.1 The Security Requirement

Scenario: Your security team requires that SSH on all Linux servers must:

4. Have the OpenSSH package installed
5. Have the SSH service running
6. Disable root login in the configuration
7. Use protocol version 2

Your Mission: Create an ESP policy that validates all four requirements.

3.2 Planning the Policy

Before writing code, let's plan what we need:

Requirement	Object Needed	State Needed
Package installed	Package object	`installed = true`
Service running	Service object	`status = 'active'`
Disable root login	Config file	`contains 'PermitRootLogin no'`
Protocol 2	Config file	`contains 'Protocol 2'`

3.3 Writing the Policy Step-by-Step

Step 1: Add Metadata

Start with metadata to document your policy:

```
META
  version `1.0.0`
  author `security-team`
  platform `linux`
  description `SSH hardening policy`
  severity `high`
META_END
```

Step 2: Start the Definition

```
DEF
```

Step 3: Define Objects

Create objects for the package, service, and config file:

```
# Package object
OBJECT openssh_pkg
  package_name `openssh-server`
OBJECT_END

# Service object
OBJECT sshd_service
  service_name `sshd`
OBJECT_END

# Config file object
OBJECT sshd_config
  path `/etc/ssh`
  filename `sshd_config`
OBJECT_END
```

Step 4: Define States

Create states for each condition:

```
# Package must be installed
STATE package_installed
  installed boolean = true
STATE_END

# Service must be active
STATE service_active
  status string = `active`
STATE_END

# Config must disable root login
STATE no_root_login
  content string contains `PermitRootLogin no`
STATE_END

# Config must use protocol 2
STATE protocol_two
  content string contains `Protocol 2`
STATE_END
```

Step 5: Create the Criteria

Now connect everything with CTN blocks inside a CRI. Since **all** requirements must pass, we use **`CRI AND`**:

```
  CRI AND
    # Check 1: Package installed
    CTN pkg_check
      TEST all all
      STATE_REF package_installed
      OBJECT_REF openssh_pkg
    CTN_END

    # Check 2: Service active
    CTN service_check
      TEST all all
      STATE_REF service_active
      OBJECT_REF sshd_service
    CTN_END

    # Check 3: Root login disabled
    CTN root_login_check
      TEST all all
      STATE_REF no_root_login
      OBJECT_REF sshd_config
    CTN_END

    # Check 4: Protocol version 2
    CTN protocol_check
      TEST all all
      STATE_REF protocol_two
      OBJECT_REF sshd_config
    CTN_END
  CRI_END
```

Step 6: Close the Definition

```
DEF_END
```

3.4 The Complete Policy

Here's the complete ssh-hardening.esp file:

```
META
  version `1.0.0`
  author `security-team`
```

```
  platform `linux`
  description `SSH hardening policy`
  severity `high`
META_END

DEF
  OBJECT openssh_pkg
    package_name `openssh-server`
  OBJECT_END

  OBJECT sshd_service
    service_name `sshd`
  OBJECT_END

  OBJECT sshd_config
    path `/etc/ssh`
    filename `sshd_config`
  OBJECT_END

  STATE package_installed
    installed boolean = true
  STATE_END

  STATE service_active
    status string = `active`
  STATE_END

  STATE no_root_login
    content string contains `PermitRootLogin no`
  STATE_END

  STATE protocol_two
    content string contains `Protocol 2`
  STATE_END

  CRI AND
    CTN pkg_check
      TEST all all
      STATE_REF package_installed
      OBJECT_REF openssh_pkg
```

```
    CTN_END

    CTN service_check
      TEST all all
      STATE_REF service_active
      OBJECT_REF sshd_service
    CTN_END

    CTN root_login_check
      TEST all all
      STATE_REF no_root_login
      OBJECT_REF sshd_config
    CTN_END

    CTN protocol_check
      TEST all all
      STATE_REF protocol_two
      OBJECT_REF sshd_config
    CTN_END
  CRI_END
DEF_END
```

3.5 Testing Your Policy

Save the policy and run it:

```
cargo run -- ssh-hardening.esp
```

Expected output:

```
✓ PASS: Package check - openssh-server is installed
✓ PASS: Service check - sshd is active
✓ PASS: Root login disabled in configuration
✓ PASS: Protocol 2 configured

Overall: COMPLIANT (4/4 checks passed)
```

Challenge Exercise: *Add a fifth check that verifies the sshd_config file has permissions 0600. Try it before looking at the solution in the next section!*

Part 3: Building Your First Policy

[Content to be added: Step-by-step guided tutorial with exercises]

Part 4: Intermediate Patterns

Now that you can build basic policies, let's level up with intermediate techniques: variables for reusability, multiple checks with logic operators, and more sophisticated compliance patterns.

4.1 Using Variables

Variables (**`VAR`**) let you define values once and reuse them everywhere. This makes policies easier to maintain and adapt to different environments.

Why Use Variables?

- **Consistency:** Use the same value in multiple places without typos
- **Maintainability:** Change once, update everywhere
- **Environment Adaptation:** Easily switch between dev/prod settings

Variable Syntax

```
VAR variable_name type value
```

Examples:

```
VAR base_path string `/etc/security`
VAR max_size int 1000
VAR must_be_enabled boolean true
```

Using Variables in States and Objects

Reference variables using the `VAR` keyword:

```
DEF
  # Define variables
  VAR config_dir string `/etc/app`
  VAR required_owner string `appuser`
  VAR min_perms string `0640`

  # Use variables in object
  OBJECT app_config
    path VAR config_dir
    filename `config.ini`
  OBJECT_END

  # Use variables in state
  STATE secure_config
    owner string = VAR required_owner
```

```
    permissions string = VAR min_perms
  STATE_END

  CRI AND
    CTN config_check
      TEST all all
      STATE_REF secure_config
      OBJECT_REF app_config
    CTN_END
  CRI_END
DEF_END
```

Best Practice: *Define all variables at the top of your DEF block for easy visibility and maintenance.*

4.2 Logic Operators: AND vs OR

The **`CRI`** block can use **`AND`** or **`OR`** logic to combine multiple checks.

Operator	Logic	When to Use
`AND`	**All** checks must pass	Strict requirements - everything must be correct
`OR`	**At least one** check must pass	Alternative options - any valid approach works

Example: AND Logic

All checks must pass:

```
CRI AND  # ALL of these must be true
  CTN pkg_check
    TEST all all
    STATE_REF package_installed
    OBJECT_REF security_pkg
  CTN_END

  CTN service_check
    TEST all all
    STATE_REF service_running
    OBJECT_REF security_service
  CTN_END
CRI_END
```

Result: PASS only if both package is installed AND service is running

Example: OR Logic

At least one check must pass:

```
CRI OR  # ANY of these can be true
  CTN firewalld_check
    TEST all all
    STATE_REF service_active
    OBJECT_REF firewalld_service
  CTN_END

  CTN iptables_check
    TEST all all
    STATE_REF service_active
    OBJECT_REF iptables_service
  CTN_END
CRI_END
```

Result: PASS if either firewalld OR iptables is active (at least one firewall)

4.3 Checking Multiple Files

Real compliance policies often check multiple related files. Let's build a policy that validates several critical system files at once.

Example: Critical Files Policy

Check that /etc/passwd, /etc/shadow, and /etc/gshadow all have proper permissions:

```
DEF
  # Define the expected permissions
  VAR passwd_perms string `0644`
  VAR shadow_perms string `0000`
  VAR gshadow_perms string `0000`

  # Define objects for each file
  OBJECT passwd_file
    path `/etc`
    filename `passwd`
  OBJECT_END

  OBJECT shadow_file
    path `/etc`
    filename `shadow`
  OBJECT_END
```

```
  OBJECT gshadow_file
    path `/etc`
    filename `gshadow`
  OBJECT_END

  # Define states
  STATE passwd_secure
    permissions string = VAR passwd_perms
  STATE_END

  STATE shadow_secure
    permissions string = VAR shadow_perms
  STATE_END

  STATE gshadow_secure
    permissions string = VAR gshadow_perms
  STATE_END

  # All files must have correct permissions
  CRI AND
    CTN passwd_check
      TEST all all
      STATE_REF passwd_secure
      OBJECT_REF passwd_file
    CTN_END

    CTN shadow_check
      TEST all all
      STATE_REF shadow_secure
      OBJECT_REF shadow_file
    CTN_END

    CTN gshadow_check
      TEST all all
      STATE_REF gshadow_secure
      OBJECT_REF gshadow_file
    CTN_END
  CRI_END
DEF_END
```

Exercise: *Add a fourth check for /etc/group with permissions 0644. Hint: Follow the same pattern as the other three files.*

4.4 Nested Logic (CRI inside CRI)

You can nest `CRI` blocks to create complex logic like: "(A AND B) OR (C AND D)"

Example: Firewall Compliance

Pass if EITHER firewalld OR iptables is properly configured:

```
CRI OR  # Either of these two groups must pass
  # Option 1: firewalld is installed AND active
  CRI AND
    CTN firewalld_pkg
      TEST all all
      STATE_REF pkg_installed
      OBJECT_REF firewalld_package
    CTN_END

    CTN firewalld_svc
      TEST all all
      STATE_REF svc_active
      OBJECT_REF firewalld_service
    CTN_END
  CRI_END

  # Option 2: iptables is installed AND active
  CRI AND
    CTN iptables_pkg
      TEST all all
      STATE_REF pkg_installed
      OBJECT_REF iptables_package
    CTN_END

    CTN iptables_svc
      TEST all all
      STATE_REF svc_active
      OBJECT_REF iptables_service
    CTN_END
  CRI_END
CRI_END
```

Logic Flow:

4. Outer CRI uses OR: at least one nested CRI must pass
5. First nested CRI: firewalld package AND service must both be good
6. Second nested CRI: iptables package AND service must both be good
7. Result: PASS if either firewall solution is fully operational

Part 5: Advanced Techniques

You're ready for advanced ESP features! In this section, we'll explore Sets for grouping objects, Filters for selective checking, and pattern matching for sophisticated validation.

5.1 Introduction to Sets

A **`SET`** lets you group multiple objects together and perform operations on them as a collection. This is powerful when you need to check many similar objects at once.

Set Operations

Operation	Description
`union`	Combine multiple objects into one set (A + B + C)
`intersection`	Find objects that exist in all sets (A AND B)
`complement`	Remove objects from a set (A - B)

Example: Union - Combining Objects

Check multiple configuration files as a group:

```
DEF
  # Define individual file objects
  OBJECT ssh_config
    path `/etc/ssh`
    filename `sshd_config`
  OBJECT_END

  OBJECT sudoers_file
    path `/etc`
    filename `sudoers`
  OBJECT_END

  OBJECT hosts_file
    path `/etc`
    filename `hosts`
  OBJECT_END

  # Combine them into a set
  SET critical_configs union
    OBJECT_REF ssh_config
    OBJECT_REF sudoers_file
    OBJECT_REF hosts_file
```

```
  SET_END

  # Check that all files in the set exist
  STATE files_exist
    exists boolean = true
  STATE_END

  CRI AND
    CTN set_check
      TEST all all
      STATE_REF files_exist
      OBJECT
        SET_REF critical_configs
      OBJECT_END
    CTN_END
  CRI_END
DEF_END
```

Result: The scanner checks all three files at once and reports if any are missing.

5.2 Using Filters

A **`FILTER`** narrows down which objects in a set should be checked. Think of it as a "WHERE clause" for your objects.

Filter Types

- **`include`** - Only check objects that match the filter state
- **`exclude`** - Skip objects that match the filter state

Example: Filter Large Files

Check only files larger than 1000 bytes:

```
DEF
  # Define size threshold
  STATE is_large
    size int > 1000
  STATE_END

  # Create a set of files with a filter
  SET large_log_files union
    OBJECT_REF log_file_1
    OBJECT_REF log_file_2
    OBJECT_REF log_file_3
    FILTER include
```

```
      STATE_REF is_large
    FILTER_END
  SET_END

  # Now check permissions only on large files
  STATE secure_perms
    permissions string = `0640`
  STATE_END

  CRI AND
    CTN filtered_check
      TEST all all
      STATE_REF secure_perms
      OBJECT
        SET_REF large_log_files
      OBJECT_END
    CTN_END
  CRI_END
DEF_END
```

How It Works:

1. Scanner examines all three log files
2. Filter includes only files > 1000 bytes
3. Permission check applies only to the filtered large files

5.3 Pattern Matching with Regex

ESP supports regular expressions for advanced string matching. Use the **`pattern_match`** operator to check if content matches a pattern.

Common Pattern Examples

Use Case	Pattern
Email address	`^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`
IPv4 address	`^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$`
Phone number (US)	`^\d{3}-\d{3}-\d{4}$`
Hexadecimal color	`^#[0-9A-Fa-f]{6}$`
Date (YYYY-MM-DD)	`^\d{4}-\d{2}-\d{2}$`

Example: Validate IP Addresses

Check if a configuration file contains valid IPv4 addresses:

```
DEF
  STATE valid_ip_format
    # Pattern for IPv4: xxx.xxx.xxx.xxx
    content string pattern_match `^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$`
  STATE_END

  OBJECT network_config
    path `/etc/network`
    filename `interfaces`
  OBJECT_END

  CRI AND
    CTN ip_check
      TEST all all
      STATE_REF valid_ip_format
      OBJECT_REF network_config
    CTN_END
  CRI_END
DEF_END
```

Important: *Regex patterns in ESP use the platform's regex engine. Test patterns on your target system to ensure compatibility.*

5.4 Record Checks for Structured Data

When checking configuration files with structured content (like INI, YAML, or JSON), use **`record`** blocks to validate specific fields.

Record Syntax

```
STATE config_requirements
  record config record_end
    FieldName type operator `value`
    AnotherField type operator `value`
STATE_END
```

Example: SSH Configuration Record

Validate multiple SSH settings in structured format:

```
DEF
  STATE ssh_secure_config
    record config record_end
      Protocol string = `2`
      PermitRootLogin string = `no`
      PasswordAuthentication string = `no`
```

```
    PubkeyAuthentication string = `yes`
  STATE_END

  OBJECT sshd_config
    path `/etc/ssh`
    filename `sshd_config`
  OBJECT_END

  CRI AND
    CTN ssh_config_check
      TEST all all
      STATE_REF ssh_secure_config
      OBJECT_REF sshd_config
    CTN_END
  CRI_END
DEF_END
```

What This Does:

The scanner parses sshd_config and validates that all four fields have the correct values. If even one field is wrong, the check fails.

> ***Practice Challenge:*** *Create a policy that uses a SET of three files, filters for files owned by root, and checks their permissions using pattern matching. Solution in the Real-World Examples section!*

Part 5.5: Advanced Concepts and Architecture

Understanding how ESP works under the hood will help you write better policies and troubleshoot issues more effectively. This section covers the compiler, scanner architecture, and advanced language features.

5.5.1 Separation of Duties: Compiler vs Scanner

ESP uses a **two-phase architecture** that separates policy validation from policy execution. This separation is fundamental to ESP's design philosophy.

Component	Compiler	Scanner
Role	Validates policy syntax and structure	Executes policy on target system
Checks	Grammar rules Type compatibility Reference resolution Circular dependencies	Platform-specific semantics Resource accessibility Actual system state Value format validity
When	Policy authoring time	Policy execution time

Why This Matters

- **Portability:** The compiler ensures your policy is syntactically valid everywhere
- **Early Detection:** Catch syntax errors before deployment
- **Platform Flexibility:** Same policy can be scanned on different platforms
- **Clear Boundaries:** Grammar rules vs business logic are separate concerns

5.5.2 The Multi-Pass Compilation Process

The ESP compiler uses a **six-pass architecture** to thoroughly validate your policy before the scanner ever sees it.

Pass	Name	Purpose
1	Lexical Analysis	Break input into tokens (keywords, identifiers, strings)
2	Syntax Parsing	Build AST following grammar rules
3	Symbol Discovery	Collect all VAR, STATE, OBJECT, SET declarations
4	Reference Resolution	Validate STATE_REF, OBJECT_REF, SET_REF, VAR references
5	Semantic Analysis	Type checking, dependency cycles, operator compatibility

Pass	Name	Purpose
6	Structural Validation	Logical patterns, completeness, optimization opportunities

Example: How the Compiler Catches Errors

Consider this policy with several errors:

```
DEF
  VAR my_path string `/etc`
  VAR my_path int 100  # Error: Duplicate identifier

  STATE check_perms
    permissions int = `0644`  # Error: Type mismatch
  STATE_END

  CRI AND
    CTN test
      TEST all all
      STATE_REF nonexistent  # Error: Undefined reference
      OBJECT_REF my_obj
    CTN_END
  CRI_END
# Missing DEF_END  # Error: Missing terminator
```

Compiler Output:

```
ERROR at line 3: Duplicate identifier 'my_path'
ERROR at line 6: Type mismatch - 'permissions' expects string, got int
ERROR at line 11: Undefined STATE reference 'nonexistent'
ERROR at line 15: Missing DEF_END terminator
Compilation failed: 4 errors found
```

5.5.3 Understanding TEST Existence Checks

The **`TEST`** directive has two parts. We've covered the second part (state logic), but the **first part (existence check)** is equally important.

Existence Check	When It Passes
`all`	Every object must exist and pass state checks
`any`	At least one object exists and passes state checks
`none`	No objects exist (forbidden resources check)
`at_least_one`	One or more objects exist (may or may not pass state checks)

Existence Check	When It Passes
only_one	Exactly one object exists and passes state checks

Example: Checking for Forbidden Files

Ensure backup files are NOT present in production:

```
DEF
  OBJECT backup_files
    path `/var/www/html`
    filename `*.bak`
  OBJECT_END

  STATE should_not_exist
    exists boolean = false
  STATE_END

  CRI AND
    CTN backup_check
      TEST none all  # Pass only if NO backup files exist
      STATE_REF should_not_exist
      OBJECT_REF backup_files
    CTN_END
  CRI_END
DEF_END
```

Example: At Least One Administrator

Ensure there's at least one admin user:

```
CTN admin_exists
  TEST at_least_one all  # Need at least one admin
  STATE_REF is_admin
  OBJECT_REF user_accounts
CTN_END
```

5.5.4 BEHAVIOR Directives: Controlling Scanner Behavior

BEHAVIOR blocks give you fine-grained control over *how* the scanner collects data, without changing *what* you're checking.

Common BEHAVIOR Options

Behavior	Purpose
`recursive_scan`	Scan directory contents recursively

Behavior	Purpose
`max_depth N`	Limit recursion depth (default: 3)
`include_hidden`	Include hidden files/directories (dotfiles)
`follow_symlinks`	Follow symbolic links during scan
`timeout N`	Command timeout in seconds (default: 5)
`cache_results`	Cache collection results for batch operations

Example: Recursive Directory Scan

```
OBJECT log_directory
  path `/var/log/app`
  BEHAVIOR recursive_scan
    max_depth 3
    include_hidden false
  BEHAVIOR_END
OBJECT_END
```

5.5.5 RUN Operations: Runtime Transformations

RUN blocks allow you to compute values at runtime using arithmetic, string operations, or field extraction from objects.

RUN ARITHMETIC Example

```
DEF
  # Calculate threshold: (1024 + 512) * 2 = 3072
  RUN computed_threshold ARITHMETIC
    literal 1024
    + 512
    * 2
  RUN_END

  STATE size_check
    size int >= VAR computed_threshold
  STATE_END
DEF_END
```

RUN EXTRACT Example

Extract a value from collected object data:

```
RUN extracted_version EXTRACT
  object_id package_obj
  field version
```

```
RUN_END
```

5.5.6 Type System Deep Dive

ESP's type system ensures that operations are meaningful and catches errors at compile time.

Type	Purpose	Example
`string`	Text values	`` `/etc/passwd` ``
`int`	64-bit integer	`1024`
`float`	64-bit float	`3.14159`
`boolean`	True/false	`true, false`
`version`	Semantic version	`` `2.4.1` ``
`evr_string`	Package version (RPM/Deb)	`` `2:1.8.0-1.el9` ``
`record`	Structured data	Config file fields
`binary`	Raw bytes	Binary file content

Version Comparisons

Version strings use semantic versioning comparison rules:

```
STATE min_version
  version string >= `2.4.0`
STATE_END
```

Comparison logic: `2.4.1 > 2.4.0` but `2.10.0 > 2.9.0` (not string comparison)

5.5.7 Inline vs Referenced Definitions

ESP allows both **global definitions** (referenceable) and **inline definitions** (local to a CTN).

When to Use Each

Aspect	Global/Referenced	Local/Inline
Reusability	Used in multiple CTNs	Single use only
Identifier	Required	Not needed
When to Use	Common checks	One-off checks

Example: Inline Definition

```
CTN one_time_check
  TEST all all
  STATE  # Inline state, not referenceable
```

```
    permissions string = `0600`
  STATE_END
  OBJECT  # Inline object
    path `/tmp/temp.file`
  OBJECT_END
CTN_END
```

5.5.8 Performance Considerations

Writing efficient ESP policies helps scans complete faster and use fewer resources.

Tip	Why It Helps
Use SETs wisely	Reduce duplicate object collection
Filter early	Apply filters before expensive operations
Limit recursion depth	Prevent scanning entire filesystem trees
Reuse definitions	Scanner can cache and optimize referenced definitions
Use TEST any when possible	Early exit when first match found

> ***Key Takeaway:*** *Understanding the compiler/scanner separation and advanced features enables you to write more sophisticated, maintainable, and performant policies.*

Part 6: Real-World Examples

Let's apply everything you've learned to real compliance scenarios from STIG and CIS benchmarks.

6.1 RHEL 9 STIG Example: Password Complexity

STIG Requirement: Ensure password complexity requirements are configured in pwquality.conf

```
META
  version `1.0.0`
  control_framework `STIG`
  control `RHEL-09-611015`
  severity `medium`
  description `Password complexity requirements`
META_END

DEF
  VAR config_path string `/etc/security`
  VAR min_length int 15

  OBJECT pwquality_conf
    path VAR config_path
    filename `pwquality.conf`
  OBJECT_END

  STATE complexity_requirements
    record config record_end
      minlen string >= `15`
      dcredit string = `-1`
      ucredit string = `-1`
      lcredit string = `-1`
      ocredit string = `-1`
  STATE_END

  CRI AND
    CTN password_check
      TEST all all
      STATE_REF complexity_requirements
      OBJECT_REF pwquality_conf
    CTN_END
  CRI_END
```

```
DEF_END
```

6.2 CIS Benchmark: Firewall Configuration

CIS Control: Ensure firewall is enabled and running

```
META
  version `1.0.0`
  control_framework `CIS`
  control `3.5.1.1`
  severity `high`
META_END

DEF
  STATE pkg_installed
    installed boolean = true
  STATE_END

  STATE service_running
    status string = `active`
    enabled boolean = true
  STATE_END

  OBJECT firewalld_pkg
    package_name `firewalld`
  OBJECT_END

  OBJECT firewalld_svc
    service_name `firewalld`
  OBJECT_END

  CRI AND
    CTN package_check
      TEST all all
      STATE_REF pkg_installed
      OBJECT_REF firewalld_pkg
    CTN_END
    CTN service_check
      TEST all all
      STATE_REF service_running
      OBJECT_REF firewalld_svc
    CTN_END
```

```
  CRI_END
DEF_END
```

Best Practice: *Always include metadata with control framework and control ID for traceability and audit purposes.*

Part 7: Troubleshooting Guide

Encountering errors is a normal part of learning ESP. This section covers the most common mistakes and how to fix them.

7.1 Common Syntax Errors

Error	Cause	Solution
Missing END marker	Forgot DEF_END, STATE_END, etc.	Add matching END marker
Undefined reference	STATE_REF points to non-existent state	Check identifier spelling
Type mismatch	Using string op on integer	Match operator to field type
Invalid backticks	Unbalanced backticks in strings	Escape backticks with ``

7.2 Common Logic Errors

Error: Policy Always Fails

Symptom: Scanner reports FAIL even though system is configured correctly

Common Causes:

- Using `CRI AND` when one check is impossible to satisfy
- Wrong operator (e.g., `!=` instead of `=`)
- Incorrect `TEST` specification (e.g., `TEST all all` when object doesn't exist)

Error: Policy Always Passes

Symptom: Scanner reports PASS even on misconfigured systems

Common Causes:

- Using `CRI OR` when all checks should be required
- Using `TEST any` when `TEST all` is needed
- State condition is too permissive

7.3 Debugging Tips

1. **Start Simple:** Build policies incrementally - test each CTN individually before combining
2. **Use Verbose Mode:** Run scanner with `--verbose` flag for detailed output
3. **Check References:** Ensure all STATE_REF and OBJECT_REF point to existing definitions
4. **Validate Types:** Make sure operators match field types (string ops for strings, int ops for numbers)
5. **Test Variables:** Print variable values to ensure they're set correctly

Part 8: Quick Reference

Your cheat sheet for ESP syntax and common patterns.

8.1 Syntax Cheat Sheet

Block Type	Syntax
Definition	`DEF name ... DEF_END`
Variable	`VAR name type value`
Object	`OBJECT name ... OBJECT_END`
State	`STATE name ... STATE_END`
Criteria Block	`CRI AND/OR ... CRI_END`
Criterion	`CTN name ... CTN_END`
Set	`SET name union/intersection ... SET_END`
Filter	`FILTER include/exclude ... FILTER_END`

8.2 Common Patterns

Pattern: File Permission Check

```
STATE secure_perms
  permissions string = `0600`
STATE_END
OBJECT file
  path `/etc`
  filename `shadow`
OBJECT_END
```

Pattern: Service Running Check

```
STATE service_active
  status string = `active`
  enabled boolean = true
STATE_END
OBJECT svc
  service_name `sshd`
OBJECT_END
```

Pattern: Package Installed Check

```
STATE pkg_present
  installed boolean = true
```

```
STATE_END
OBJECT pkg
  package_name `openssh-server`
OBJECT_END
```

Pattern: Configuration Content Check

```
STATE required_setting
  content string contains `PermitRootLogin no`
STATE_END
OBJECT config
  path `/etc/ssh`
  filename `sshd_config`
OBJECT_END
```

8.3 Operator Quick Reference

Category	Operators	Types
Comparison	`=` `!=` `>` `<` `>=` `<=`	All types
String	`contains` `starts` `ends`	string only
Pattern	`pattern_match`	string only
Set	`subset_of` `superset_of`	Sets only

Part 9: CTN Type Reference

The ESP Scanner SDK includes 8 CTN types. Each has specific requirements and capabilities documented below.

9.1 Available CTN Types

File System CTN Types:

- **`file_metadata`** - Fast stat() checks: permissions, owner, group, size, existence
- **`file_content`** - Content validation: contains, starts_with, ends_with, pattern_match. Supports recursive_scan, include_hidden, follow_symlinks behaviors
- **`json_record`** - Structured JSON validation using RECORD_CHECK blocks

System CTN Types:

- **`rpm_package`** - Package installation and version checks. Supports timeout, cache_results behaviors
- **`systemd_service`** - Service status: active, enabled, loaded. Supports timeout behavior
- **`sysctl_parameter`** - Kernel parameters: value (string) or value_int (numeric)
- **`selinux_status`** - SELinux enforcement: mode, enforcing boolean

Testing CTN Type:

- **`computed_values`** - Validates RUN operations (testing only, not for production)

9.2 Example Files in Repository

The **`esp/`** directory contains working examples:

File	Description
`set_test.esp`	SET operations (union, intersection, complement)
`ssh_config_check.esp`	SSH hardening validation (file_content)
`passwd_shadow_content.esp`	System file content validation
`missing_file_handling.esp`	Existence checks and error handling
`critical_file_permissions.esp`	File permission validation (file_metadata)
`variable_usage.esp`	Variable declaration and usage examples
`test_behavior*.esp`	BEHAVIOR directive examples

File	Description
`test_filters.esp`	FILTER operations on sets

Test Data Files

The `scanfiles/` directory contains: config.json, ntp.conf, passwd, shadow, sshd_config, sudoers, system.conf, test_data.json

Running Examples

```
# Single file
cargo run -- esp/ssh_config_check.esp

# With debug logging
ESP_LOGGING_MIN_LEVEL=debug cargo run -- esp/set_test.esp

# Batch scan all examples
cargo run -- esp/
```

Congratulations!

You've completed the ESP Educational Guide! You now have the knowledge to:

- Write basic and advanced ESP policies
- Use variables, logic operators, and sets effectively
- Implement real-world STIG and CIS compliance checks
- Debug and troubleshoot policy issues
- Reference common patterns quickly

Next Steps:

1. Practice writing policies for your specific environment
2. Explore the ESP Language Guide for advanced features
3. Review the Scanner Development Guide to extend ESP capabilities
4. Join the ESP community to share policies and get help

Happy Policy Writing!