Introduction

Effective policy administration is critical for using Ping Authorize to control access to resources based on fine-grained attribute based access control (ABAC) policies. Ping Authorize provides a powerful policy engine and flexible policy modeling to handle complex, real-world authorization scenarios. However, this power and flexibility also introduces management challenges as the number of policies and complexity of the authorization model grows.

This document provides a set of best practices for efficiently and reliably administering Ping Authorize policies at scale. It covers recommendations for organizing policies, defining attributes, using policy testing and deployment techniques, and leveraging Ping Authorize's features for manageability. The practices are derived from Ping's extensive experience working with customers to deploy dynamic authorization in production.

By adopting these practices, policy administrators can:

- Implement clear, consistent policies aligned to business requirements
- Manage policies and attributes effectively as complexity increases
- Deliver policy changes rapidly and safely using automation
- Provide business stakeholders visibility and confidence in the authorization model

The guide is intended for IAM architects, developers and administrators responsible for deploying Ping Authorize and defining the policies that enforce authorization. Familiarity with Ping Authorize concepts and configuration is assumed. Refer to the Ping Authorize Administration Guide for more details on specific features and procedures referenced.

Overview of Policy Administration with Ping Authorize

Policy Components

Ping Authorize policies are built from the following main components:

- **Policy Sets** - Top level containers that group related policies together. Commonly used to represent domains like Banking or HR.
- **Policies** - Containers that aggregate policy rules implementing a related set of access requirements, like Customer Access to Accounts
- **Rules** - Logical expressions that specify the criteria for permitting or denying access, based on combinations of attributes

- **Attributes** - Named values representing characteristics of the user, resource, action and environment, used in composing rules
- **Attribute Resolvers** - Logic for retrieving attribute values from PingAuthorize's local trust framework or external policy information points (PIPs) like directories and APIs

Policies are evaluated for incoming authorization requests containing attributes that identify the subject, resource, action and other contextual data. The policy engine processes the policies to render a Permit or Deny decision which is returned to the enforcement point.

Trust Framework

Policies retrieve most of the attributes they need to make decisions from Ping Authorize's local Trust Framework. The Trust Framework provides a unified repository where administrators can define the attributes, resolvers and other data needed for policies.

Items defined in the Trust Framework include:

- **Attributes** - Attribute definitions including name, data type, resolvers, and other properties
- **Domains** - Hierarchical categorization of resources that policies protect
- **Services** - REST API endpoints used as PIPs to retrieve external attribute values at policy evaluation time
- **Identity Providers** - Sources of user authentication mapped to trust levels
- **Tokens** - Token types and claims representing calling applications

The Trust Framework allows complex authorization data to be modeled in business-friendly terms, independent of the technical implementation. This enables policy administrators to author policies against clean, consistent abstractions.

Policy Administration Lifecycle

Policy administration with Ping Authorize typically follows an iterative lifecycle:

1. **Gather Requirements** - Work with business and application teams to understand the resources that need protecting and the rules governing their usage
2. **Model Attributes** - Identify the subject, resource, action and environmental attributes needed to express the policy rules and define them in the Trust Framework
3. **Author Policies** - Compose policy sets, policies and rules using the Trust Framework attributes to implement the access requirements
4. **Test and Refine** - Exercise the policies using Ping Authorize's testing tools to verify they make the right decisions for representative access requests, refining as needed

5. **Deploy to Production** - Use deployment packages to promote final policies to production PingAuthorize servers and enable them
6. **Monitor and Audit** - Review PingAuthorize's decision logs to validate enforcement and identify any unexpected results
7. **Extend and Evolve** - Enhance policies over time as business requirements change and new use cases emerge

The rest of this guide provides practices for optimizing each stage of the lifecycle, making policy administration more efficient, consistent and reliable.

Modeling Attributes Effectively

Well-designed attributes are the foundation of good policy administration. Keep the following practices in mind when modeling attributes in the Trust Framework:

Naming Conventions

- Use clear, concise names that reflect the business purpose of the attribute, avoiding technical jargon
- Establish naming conventions and apply them consistently across all attributes to avoid ambiguity
- Use namespaces to group related attributes and avoid naming collisions between different attribute sources
- Provide descriptions for attributes to document their intended usage and value

Attribute Granularity

- Model attributes at the right level of granularity to support policy rules without unnecessary complexity
- Avoid bundling multiple access control values into complex multi-valued attributes
- Use parent-child relationships to derive fine-grained attributes from more coarse-grained ones
- Strike a balance between granularity and maintainability - prefer fewer, more generic attributes vs many special-purpose ones

Attribute Resolvers

- Use attribute resolvers to virtualize attributes - pull values from PIPs on-demand during policy evaluation vs syncing

- Chain together multiple resolvers to aggregate values across different PIPs into consolidated business attributes
- Leverage different resolver types - LDAP/HTTP for external PIP attributes, Groovy for deriving custom attributes
- Cache attribute values judiciously to boost performance for frequently accessed, slowly changing attributes

Attribute Data Types

- Assign the appropriate data type for each attribute based on its range of values - string, numeric, boolean, date, etc.
- Avoid defaulting to string type for non-textual data - use structured types like numbers and dates for more precise matching
- Use null/empty checks in policies for optional attributes to avoid malformed request errors
- Be aware of data type differences when combining attributes from multiple PIPs in a single policy rule

Secret Attributes

- Mark attributes holding sensitive data like OAuth token values as Secrets to prevent leakage in logs
- Avoid using Secret attributes in policy rules unless absolutely required to minimize exposure
- Rotate secrets periodically and after any suspected compromise using new resolver configurations

Documenting Attributes

- Document all attributes used in policies in an attribute dictionary for policy admins and business stakeholders
- Include attribute name, description, data type, resolver config, owning domain and example values
- Call out any attribute dependencies, limitations or other usage constraints
- Keep the dictionary in sync with the Trust Framework as attributes evolve over time

Investing effort up-front to model clean, consistent attributes will make policy authoring and maintenance much smoother by providing a stable, well-understood foundation to build upon.

Organizing Policies Effectively

As the number of applications and resources governed by Ping Authorize grows, so does the number of corresponding policies. Organize the growing policy footprint with the following practices:

Policy Hierarchy

- Use a hierarchical policy model aligned to your business domains to encapsulate related policies
- Start with a top-level Policy Set for each major organizational unit or business function
- Nest sub-Policy Sets beneath to further segregate policies by application, resource type or policy target
- Limit Policy Set nesting depth to 3-4 levels to avoid over-complexity
- Use a common hierarchical template across all major domains to enforce consistency

Policy Granularity

- Structure individual policies around a specific access control decision - "Can user X do action Y on resource Z?"
- Include all relevant rules for a decision in a single policy vs spreading across multiple policies
- Keep policies cohesive - if rules don't decision the same access decision, break them into separate policies
- Balance policy granularity against proliferation - consolidate related rules to simplify management
- Leverage policy inheritance for shared rules - put common rules in parent policy and specialize in child policies

Policy Naming Standards

- Define descriptive, intention-revealing names for Policy Sets and Policies
- Encode key policy target, effect type and intended action in policy names for quick readability
- Follow consistent naming conventions across the policy hierarchy for cohesiveness
- Avoid names that are too long or include volatile details that change frequently
- Document naming conventions in a wiki or team site for all policy authors to reference

Policy Targets

- Use the "Applies To" section of policies to explicitly target the resources and actions they govern
- Build targets from well-defined Trust Framework attributes representing Domains, Services, Users, etc.
- Avoid catch-all policies with no explicit targets - such policies are hard to understand and manage
- Minimize conditional targets in policies - extract conditional logic into rules or PIPs where possible
- Leave "Applies To" empty for abstract, reusable policies designed to be attached to other policies

Policy Reuse

- Encapsulate rule logic that recurs frequently across policies into Condition attributes in the Trust Framework
- Create skeleton "template" policies for standard access patterns (e.g. role-based, relationship-based, etc.)
- Copy policies or rules to accelerate authoring when creating policies for similar use cases
- Attach parameterized rules to multiple policies to enforce shared access constraints in a consistent way
- Reuse tested policies and rule idioms when possible to avoid reinventing the wheel and introducing errors

Policy Documentation

- Write a concise summary statement for each Policy Set, Policy and Rule to convey the intent at a glance
- Reference the corresponding business/regulatory requirements driving each policy artifact
- Document any prerequisite data, configuration or timing dependencies required for policies to evaluate correctly
- Call out any known issues, edge cases or limitations in policies that policy consumers should be aware of
- Maintain policy documentation in-line using policy descriptions and rule comments where supported

Organizing policies effectively takes careful forethought and ongoing discipline as the authorization layer grows. But it pays off in spades by making the policy structure more transparent, adaptable, and efficient to manage and change safely over time.

Authoring Policies Safely

Ping Authorize's policy language is very expressive, allowing complex authorization logic to be modeled. However, this flexibility can lead to policies that are hard to understand, test and debug when something goes wrong. Follow these practices to make policies more robust and maintainable:

Policy Clarity

- Write policies for readability using a consistent, easy-to-follow structure and flow
- Organize rules into logical groups with clear headings to convey their purpose
- Keep rule criteria concise by extracting complex logic into custom attributes or external PIPs
- Use optional braces, extra spaces and line breaks to visually clarify compound rule logic
- Add comments to explain the rationale and expected behavior for any non-obvious rules

Safe Policy Logic

- Avoid double negatives and complex compound statements in rules that are hard to mentally parse
- Use positive logic ("employee has manager role") vs negative logic ("employee does not have non-manager role")
- Explicitly handle empty/null attribute values in rules to avoid unexpected evaluation errors
- Validate attribute values from untrusted sources to screen out malformed data before using in rules
- Use parentheses liberally to make operator precedence explicit vs relying on default precedence rules

Policy Defaults

- Write policies to fail safely - start with a default Deny effect and only Permit when criteria are met
- Include a final "catch-all" rule to handle unmatched requests and log for visibility
- Set "missing attribute" policy to Deny vs NotApplicable to fail closed and log for investigation
- Return "Obligations" from policies to enforce additional constraints beyond binary permit/deny

- Use Permit Overrides combining algorithm for policies - a single permit trumps any denies

Attribute Validation

- Validate any attributes retrieved from external sources before using in policy logic
- Check for expected data types, ranges and formats using explicit comparisons or regular expressions
- Constrain enumerated attributes to an explicit list of allowed values to prevent injection
- Escape any user-defined attributes in rules to prevent embedded scripting expressions
- Reject any attribute values that don't match expected patterns to fail safely

Handling Errors

- Decide on the appropriate error handling behavior when an invalid attribute or PIP failure occurs during evaluation:
  - Deny the request
  - Allow the request
  - Continue evaluating rules as if the attribute doesn't exist
- Use rule conditions, obligations or error handling policies to enforce the desired behavior consistently
- Log unexpected errors with context about the request to aid in troubleshooting
- Alert on error threshold breaches to proactively detect authorization issues before end users

Policy Hygiene

- Peer review all policy changes before moving to production using merge requests and approval workflows
- Evaluate policies with static analysis tools to identify common mistakes and anti-patterns
- Regularly audit policies against business requirements to identify redundant, conflicting or outdated logic
- Refactor policies continuously to keep them concise, clear and easy to maintain
- Deprovision policies when the corresponding application is retired to keep the policy set clean

Authoring policies is part engineering, part art. It requires balancing precision, flexibility and usability to model the desired authorization semantics without overcomplicating the implementation. By adopting the above practices, policy authors can produce policies that are easier to understand, test and safely evolve over time as requirements change.

Testing and Deploying Policies

Policies don't provide value until they are promoted to production and used to authorize real application requests. But deploying untested policies is risky - buggy policies can block legitimate access or expose sensitive data. Use the following practices to test and deploy policies safely:

Unit Testing

- Author policies against realistic sample data to model the types of requests they will process
- Develop a suite of test cases that exercise the main policy path and edge cases
- Organize test cases by policy outcome (permit, deny, etc.) and test data setup required
- Use Ping Authorize's Test Suite to execute test cases and render a pass/fail for each
- Integrate policy unit testing into the CI/CD pipeline to catch regressions early

Vendor Testing

- Work with application teams to define the expected behavior for a representative set of privileged and non-privileged requests
- Implement end-to-end tests using vendor test frameworks (e.g. Postman, SoapUI, etc) to drive application requests through PingAuthorize
- Assert PingAuthorize returns the expected permit/deny decision and any Obligations or Advice for each test case
- Include vendor tests in CI/CD pipeline and gate deployments on tests passing

Manual Testing

- Manually exercise the application UI to test policies for user acceptance before releasing
- Ensure policies enforce the expected user experience for different user roles and permission levels
- Perform exploratory testing using different user accounts to identify gaps in policy coverage
- Document and share any bugs or unexpected behaviors discovered in manual testing
- Perform manual regression testing when policies or dependent applications change

Deployment Automation

- Package policies as versioned artifacts and manage in a central repository
- Use PingAuthorize's policy migration features to automate deploying policies from lower environments to production
- Implement deployment smoke tests to quickly assess the overall health of the authorization layer after each deployment
- Integrate policy deployment with application release process so policies and apps are in sync
- Decouple policy deployment from application deployment where possible to support independent release velocity

Partial Rollouts

- Roll out policies gradually using PingAuthorize's policy branching feature to limit blast radius
- Deploy new policies to a small set of pilot users first and monitor for correct behavior
- Progressively deploy to larger user populations as confidence grows
- Support fast rollback by deploying new policies side-by-side with old and swapping when ready
- Use traffic splitting and blue/green deployment patterns to shift traffic safely to new policies

Policy Metrics

- Track policy deployment frequency, lead time and rollback rates to assess agility and quality
- Monitor policy evaluation outcomes (permit/deny/error rates) to detect unexpected changes in authorization patterns
- Alert on spikes in authorization failures or latency to proactively detect issues
- Track metrics over time and perform trend analysis to spot opportunities for optimization
- Use metrics to inform policy refactoring and drive continuous improvement

Testing and deployment rigor is critical for delivering reliable authorization decisions in production. By adopting a structured testing methodology, automating policy deployments, and monitoring key metrics, enterprises can ship policy changes frequently without compromising security or user experience.

Visibility and Audit

Policies only deliver their intended value if they are evaluated correctly for all requests. Even small gaps in policy enforcement can have big security or compliance consequences. Use the following practices to gain visibility into policy decisions and audit their usage:

Decision Logging

- Configure PingAuthorize to log detailed info about each policy transaction, including:
    - Request attributes (subject/resource/action/env)
    - Decision rendered (permit/deny/notApplicable)
    - Policies and rules evaluated and their outcomes
    - Any obligations or advices returned
- Log additional context (timestamp, request ID, app name, etc) to correlate decisions with app transactions
- Mask any sensitive data (SSN, CC#, etc.) in the logs to avoid leakage
- Securely store policy decision logs with encryption at rest and strict access control
- Retain logs for a sufficient period to satisfy compliance and forensics requirements

Decision Monitoring

- Stream policy decision logs to a SIEM or log analysis tool for centralized monitoring
- Parse logs and extract key data elements into a structured format for analysis
- Define dashboards to track policy decision trends and summary statistics in real-time
- Configure alerts to notify admins when unusual authorization patterns occur (e.g. spikes in failures)
- Investigate any unexpected results and adjust policies and/or policy data as needed

Enforcement Auditing

- Periodically audit PingAuthorize policy decisions against application and API access logs to verify policies are being enforced end-to-end
- Identify any resources accessed without a corresponding Permit decision from PingAuthorize (may indicate policy gaps)
- Identify any resources not accessed that have Permit decisions (may indicate stale or over-permissive policies)
- Reconcile any discrepancies between policy decisions and actual resource access with app teams
- Report on enforcement audit findings and track remediation items to closure

Access Reviews

- Perform periodic access reviews with resource owners and auditors to verify the right users have the right permissions
- Extract permission data from PingAuthorize policies and present in user-friendly views by role, attribute values, etc.
- Collect review decisions (certify, revoke, etc.) from reviewers and feed back into PingAuthorize to update policies
- Use PingAuthorize policy versioning to generate audit trail of permission changes over time
- Provide evidence of review process and outcomes to satisfy compliance requirements

Policy Explainability

- Educate business stakeholders on how to interpret PingAuthorize policies and decision logs
- Provide self-service tools to look up why specific access was granted or denied based on policies and request attributes
- Enable dynamic policy evaluation in test environments to interactively explore decision outcomes
- Embed policy explainability features into applications to provide transparency to end users
- Train support staff on troubleshooting common authorization issues using PingAuthorize explainability tools

Compliance Automation

- Codify relevant regulatory and industry compliance requirements (HIPAA, PCI, etc.) into authorization policies
- Map PingAuthorize policy decisions to specific compliance controls to show coverage
- Integrate compliance policies into unit and integration testing to automatically assess adherence
- Generate compliance reports showing all resources and actions explicitly permitted and prohibited by policies
- Attest to compliance status using PingAuthorize audit trails as evidence

Providing robust visibility and auditing for authorization decisions is essential for security assurance and compliance. By instrumenting PingAuthorize with rich decision logging, actively monitoring decisions in production, and instituting regular enforcement auditing and access review processes, enterprises can be confident their authorization policies are being applied completely and correctly across their environment.

Conclusion

This guide provides a set of best practices for efficiently and reliably administering authorization policies using PingAuthorize at enterprise scale. From modeling clean, consistent policy attributes to organizing policies for manageability, authoring policies safely, and deploying and auditing policies effectively, it offers concrete recommendations for each stage of the policy administration lifecycle.

Incorporating these practices into your IAM operating model will help drive better authorization outcomes for your business in multiple dimensions:

- **Security** - consistent, fine-grained enforcement of authorization policies across apps and APIs
- **Agility** - faster time-to-market for new apps and features through self-service policy management
- **Usability** - seamless, personalized user experiences based on contextual access decisions
- **Compliance** - auditable enforcement of regulatory and business compliance requirements

IDPartners is committed to continually make policy administration even more powerful and productive for our customers. We welcome your feedback on how we can further simplify dynamic authorization and enforce the right access at scale in your environment.