BLUE COAT

ProxySG/
Policy

Policy Best Practice Guide

Document Revision: 03/Date: 05/16/16

Copyrights

Americas	Rest of the World
Blue Coat Systems, Inc. 384 Santa Trinita Avenue Sunnyvale, CA 94085	Blue Coat Systems International SARL 3a Route des Arsenaux 1700 Fribourg, Switzerland

Policy Best Practices

This document provides best practices to consider and follow when you are creating policy for Blue Coat ProxySG appliances. The document contains the following major topics.

Learn

Conceptual information about policy functionality and components.

"About Blue Coat Policy" on page 9

Construct

Best practices to consider when creating policy layers and rules.

"Construct Policy" on page 29

Optimize

Describes how to configure the ProxySG appliances and policy components for optimal policy performance.

" Optimize Policy Performance" on page 36

Advanced Policy

Provides best practices for experienced policy users.

"Advanced Policy" on page 53

Troubleshoot

Provides suggestions on how solve policy problems.

"Troubleshoot Policy" on page 56

Table Of Contents

About Blue Coat Policy

The following sections discuss the fundamentals of policy for the ProxySG appliance:

- "About Policy" on page 10–Provides an introduction to Blue Coat policy, including default policy options, layer descriptions, and the available creation methods.
- "About Policy Evaluation" on page 16–Describes how policy layers function together.
- "About Policy Actions " on page 21–Describes various actions available in policy rules.

About Policy

Blue Coat policy is a powerful tool that allows you to create and apply flexible rules to users and content. Policy, with its various functions, enables you to control communications across users, content types, and applications. The role of policy is similar to that of a gate guard; a gate guard intercepts all incoming traffic, evaluates it against the security policy, and applies the proper action. Policy in its most basic form evaluates the rules you create and implements the actions associated with those rules.

Policy files manage every aspect of the ProxySG appliance, from controlling user authentication and privileges to disabling access logging. Specifically, policy makes decisions for authentication, authorization, malware scanning, content filtering, DLP, and other functions.

This flexibility, and the ability to create almost limitless rule combinations, is possible because of the extensive set of available conditions, properties, and actions. The policy for an appliance can contain several files with many layers and rules in each. Policies are configured through the Visual Policy Manager (VPM) or composed in Content Policy Language (CPL). The appliance's VPM is a GUI for CPL; however, some policy gestures are CPL-only. Policy evaluates layers and rules based on the configured default policy, Allow or Deny, and either allows the transaction or denies the transaction. However, you can create exceptions to the configured default policy in policy rules. The versatility of the ProxySG appliance gives you the ability to adjust policy according to your company's security needs.

The following topics provide more details.

- "Types of Policy Files" on page 11–Multiple sources can provides policy.
- "Standard Default Policy: Allow or Deny ?" on page 12–Which is the best default setting for your environment?
- " Policy Layers" on page 13–What are policy layers and how do the function?
- "VPM or Local Policy File?" on page 15–What is the Visual Policy Manager and how does it relate to the policy language?

Types of Policy Files

Manage policy through four file types.

Visual Policy File

VPM policy can either supplement or override the policies created in the Local, Central, and Forward Policy Files. The Visual Policy File contains VPM-created policies specific to the ProxySG appliance on which it was created.

Local Policy File

A file you create yourself. When the VPM is not the primary tool used to define policy, the Local file contains the majority of the policy rules specific to the ProxySG appliance it was created in. If the VPM is the primary tool, this file is either empty or includes rules for advanced policy features that are not available in VPM.

Central Policy File

Contains common policies the administrator wants to enforce across all proxy devices; each proxy in a large network of proxies subscribes to this file and is updated as changes occur by either the administrator or an automated process. The Central policy centrally governs a set of policy generated by either the administrator or automated process.

Forward Policy File

Usually used to supplement any policy created in the Visual, Central, and Forward policy files. The Forward policy file forwards decisions, helping the ProxySG appliance reach its origin servers.

Policy File Selection – Best Practices

Blue Coat recommends using the Visual and Local policy files for policies directly associated with a single appliance. Implement the Central and Forward policy files as needed with larger networks of appliances.

Standard Default Policy: Allow or Deny ?

The default policy is the primary action applied on all decisions unless later policy rules override it. The default policy is to either Deny proxied transactions or to Allow proxied transactions.

- **Deny**–A default policy of Deny prohibits all proxied transactions; you must then create policies to explicitly grant access on a case-by-case basis; this is called whitelisting.
- **Allow**–A default policy of Allow permits all proxied transactions. You must create policies to explicitly deny access on a case-by-case basis; this is called blacklisting.

Default Policy Configuration – Best Practices

Set the default policy (Allow or Deny) to align with your corporate security policy – and then use blacklist or whitelist approaches as appropriate.

Blue Coat recommends the following settings:

- Secure Web Gateway: Allow or Deny
  - If security is a priority, set the default proxy policy to Deny.
  - If service is a priority, set the default proxy policy to Allow.
- WAN Optimization Deployments: Allow
- Reverse Proxy Deployments: Deny

Policy Layers

VPM and CPL contain a set of layers, as displayed in the figure below. Each layer refers to a specific area where related policy rules can be entered. For example, if you want to implement a rule allowing access to certain web content, you would place that rule in the **Web Access Layer**.

VPM layers:

- Are logically named–The layer names are indicative of the action taking place in the layer. For example, the **Admin Authentication Layer** indicates that the layer is for actions associated with accessing administrative consoles. This allows you to group related actions in a related layer to make a single decision.
- Can be renamed and renumbered.
- Can be disabled and reordered.

As you can see from the table below, CPL layers are similar to VPM layers; however, they are in CPL.

To maintain optimized evaluation, include policy in the specified order according to the following.

VPM Layer	CPL	Description
Web Authentication	`<proxy>`	User authentication and allowed triggers
Web Access	`<proxy>`	Access control and general transaction testing
SOCKS Authentication	`<proxy>`	Controls SOCKS client authentication
Web Content	`<cache>`	Object store behavior modification and malware scanning
Forward	`<forward>`	Request destination control
DNS Access	`<dns-proxy>`	DNS proxy specific control and transaction testing
Admin Authentication	`<admin>`	User authentication when accessing the proxy's administrative console
Admin Access	`<admin>`	Access control and Read/Write privilege control for the proxy's administrative console
SSL Intercept	`<ssl-intercept>`	Controls HTTPS and s-tunnel proxy (what to intercept)
SSL Access	`<ssl>`	Controls SSL certificate validation behaviors
Not Applicable	`<exception>`	Control of exception responses including header set/rewrite/delete
CPL	All of the above	CPL layer allows users to write policy using CPL without having to switch from VPM to a local policy file

Work with Layers – Best Practices

As the number and complexity of policy rules grows, it can be difficult to keep track of your policy logic. Because of that Blue Coat recommends the following:

- Name the layer according to the purpose of the layer.
- Number the layer to avoid confusion over multiple same type layers.

- Place a three-letter acronym at the beginning of the layer name to ensure original layer name and type are not lost (see the following table).

Acronym	Layer Name
WAL	**Web Access Layer**
WCL	**Web Content Layer**
WAU	**Web Authentication Layer**
FWD	**Forwarding Layer**
SAU	**SOCKS Authentication Layer**
DNS	**DNS Access Layer**
AAU	**Admin Authentication Layer**
AAL	**Admin Access Layer**
SIL	**SSL Intercept Layer**
SAL	**SSL Access Layer**
CPL	**CPL Layer**
NAL	N/A Layer

Using the recommendations, the following are sample layer names:

- **WAL Finance Rule(1)**
- **AAL Seattle Center(1)**
- **AAU Branch HR(1)**

VPM or Local Policy File?

The Local policy file, CPL, mainly controls the following operations:

- User authentication requirements
- Access to web-related resources
- Cache content
- Various aspects of request and response processing
- Access logging

The Visual Policy Manager (VPM) is a user interface launched from within the ProxySG appliance Management Console. On this graphical user interface, the VPM controls the same operations as the Local file. The VPM allows you to quickly create and install policy. You can add VPM objects to policy and display the generated CPL. The VPM provides graphical layers that you can use to create rules corresponding to the logic of the layer. For example, you can create a deny rule in the **Web Access Layer** to deny specific web access.

VPM and Local Policy File – Best Practices

Use the VPM whenever possible. If you need to implement advanced policy features not available in the VPM or you want to optimize efficiency, use the Local policy file.

While the VPM generates CPL, do not edit the generated CPL. Editing the generated CPL can have unintended consequences, for example, overriding a previously desired rule in the layer.

Blue Coat recommends maintaining separate policy files: VPM for general policy and Local policy file for layers and/or rules that require more advanced CPL not available in the VPM.

About Policy Evaluation

Policy evaluation goes through all applicable layers for a transaction from top to bottom. When the ProxySG appliance receives a transaction request, the appliance evaluates layer guards first, if any, followed by the layer's rules until a match is made.

For each layer, policy evaluation starts with layer guards if there are any. If the layer guard condition is satisfied, policy evaluates the layer next. The layer's rules are evaluated from top to bottom until a rule match is made or all the rules have been tried and no match is found, ending the evaluation of the layer. This process is repeated until the last applicable layer is evaluated. After the last applicable layer is evaluated, whether a matched rule is found or not, the evaluation ends.

This order of evaluation occurs on a continuous cycle, matching in every section, until the last-matched rule is read or no match is found.

When the last-matched rule is read, or no match is found in the layer, the transaction is complete and no further evaluation is done.

The following sections cover key aspects of evaluation.

- "Layer Guards " below
  - What are layer guards.
  - How layer guards work.
- "Layer Evaluation Order" on the next page
  - How the evaluation of layers occurs.
- " Rule Evaluation" on page 18
  - How rule evaluation occurs.
- "Enforcement of the Last Matched Rule " on page 18
  - The importance of rule order in rule evaluation.
  - How to maintain the integrity of policy with policy evaluation.

Layer Guards

The more policy you create, the more you are likely to have rules related by a common condition within a layer. A common condition is a classifying agent that recurs in the set of rules and can be anything from IP addresses to subnets. In these cases, implement *layer guards*. The purpose of a layer guard is to test a condition before evaluating the following rule or set of rules in the layer. This avoids unnecessary processing of rules in a layer.

The following example shows a common condition, `group=hr`, without a layer guard.

```
<proxy>
authenticate(myrealm)

<proxy>
group=hr user=bluecoat\bob.kent OK
group=hr url.domain=www.mercurynews.com/hotjobs/ OK
```

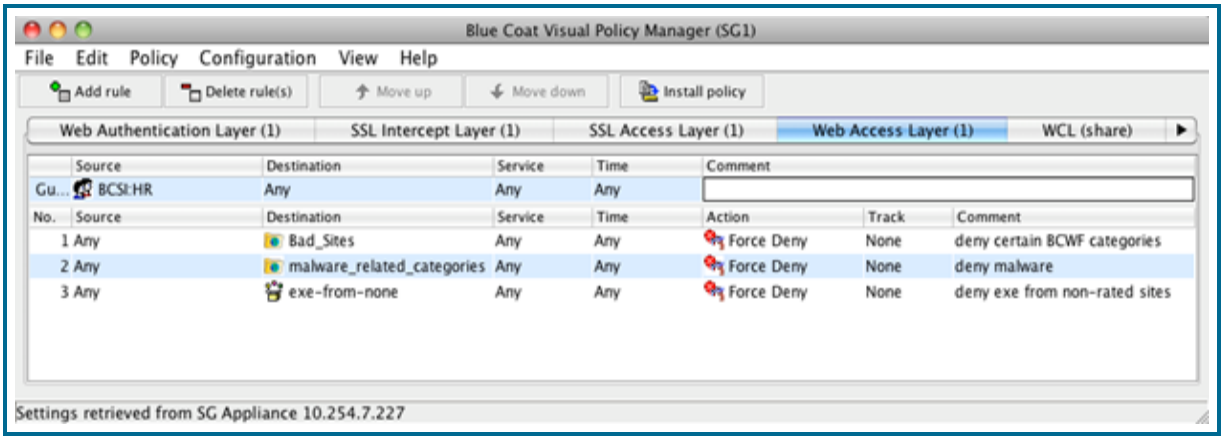
```
group=hr url.domain=sfgate.com/jobs/ OK
group=hr url.address=192.168.23.5 DENY
group=hr category=(news/media)
```

The following example shows an improved implementation with a layer guard.

```
<proxy>
authenticate(myrealm)
```

```
<proxy>group=hr
user=bluecoat\bob.kent OK
url.domain=www.mercurynews.com/hotjobs/ OK
url.domain=sfgate.com/jobs/ OK
url.address=192.168.23.5 DENY
category=(news/media) exception(content_filter_denied)
```

The following image shows how the same layer guard appears in the VPM.



Layer guards test the common condition before evaluating the subsequent rule or set of rules in the layer. In the preceding example, policy examines the user request and tests it with the layer guard condition `group=hr`. If the user request matches the layer guard, the rest of the layer is evaluated until a match is found. If the user request does not match the layer guard, the layer is not evaluated and policy continues on to the next layer, saving CPU. In cases where no layer guard is in place, the entire layer is evaluated until there is a match.

Work with Layer Guards — Best Practices

For organization and optimization purposes, Blue Coat recommends using layer guards to prevent unnecessary evaluation within layers and to improve performance. Without the use of layer guards, policy evaluates every rule in the layer, increasing CPU utilization and lengthening evaluation time.

Next Evaluation

See "Layer Evaluation Order" below.

Layer Evaluation Order

Policy layer evaluation occurs in the following order:

1. Layer guards (if any) (see "Layer Guards " on the previous page)
2. Rules (top to bottom)

Evaluation of a layer ends when a matched rule is found and executed. When evaluation of a layer is completed, policy moves to evaluate the following layer.

This evaluation order repeats until the last applicable layer is evaluated, completing the requesting user's transaction.

For example, consider when a user clicks on a web page. The ProxySG appliance examines the installed policy to evaluate the request. It reads and defines layer guard first; if layer guards exist, the appliance determines if the request matches the conditions of the guard.

- If there *is* a layer guard and:
  - The request does *not* match the layer guard conditions, the appliance skips the layer and evaluates the next layer.
  - The request matches the layer guard conditions, the appliance evaluates the layer from top to bottom until there is a rule match. The appliance executes the rule action if applicable and skips any remaining rules in the layer.

    The appliance evaluates the next layer, repeating the process of layer guards and top-to-bottom rule evaluation until it matches the last possible rule, or no match rule is found.
- If there is *no* layer guard, the appliance evaluates every rule in the layer from top to bottom until it finds a match or it reaches the end of the layer. The appliance executes the rule action if applicable and skips any remaining rules in the layer.

  The appliance evaluates the next layer repeating the process of layer guard and top-to-bottom rule evaluation until it reads the last-matched rule, or no matched rule is found.

Next Evaluation

See " Rule Evaluation" below.

Rule Evaluation

Blue Coat policy evaluates rules in a layer from top to bottom until there is a match. A policy rule consists of one or more conditions and some number actions. When the appliance evaluates a rule, it tests the conditions for the current transaction. If all of the conditions evaluate to true, the rule is said to match. When there is a match, all of the listed actions are executed (though later layers can potentially override the action) and evaluation of the current layer ends. If one or more of the conditions evaluate to false for that transaction, it is a miss, and policy evaluation continues to the next rule in the layer and the process is repeated.

Next Evaluation

See "Enforcement of the Last Matched Rule " below.

Enforcement of the Last Matched Rule

The ProxySG appliance evaluates rules in the order they are listed in policy. Policy evaluates rules row by row and enforces the matched rule, skipping the remaining rules and proceeding to the next policy layer where the process is continued until the last matched rule is read.

For any conflicting policy actions in matched rules, the last matched rule prevails. For example, the action of the last rule matched can override all conflicting actions in previous rules.

As policy grows and becomes more complex, maintaining the integrity of the rules becomes difficult. Due to evaluation order, it is possible to unintentionally override previous layer rules. To prevent this from happening, ensure that you do the following in both the Local file and VPM:

- Adjust the rule order.
  - Specific rules to general rules.
- Understand and use the appropriate action:
  - Ok
  - Allow
  - Deny
  - Force Deny
  - Exception
  - Authenticate
  - Force Authenticate
- Use Boolean operators to provide further regulation.

These topics are described in the next section.

General Rule Order – Best Practices

Adjust the rule order from specific to general to create a match earlier in a layer.

Within a layer, use the sequence of most-specific to most-general policy.

Because the appliance evaluates policy rules from top to bottom, and the first rule that matches ends further evaluation of the layer, place specific conditions or exceptions to global rules near the top of the layer to establish a match sooner and prevent a performance impact. When policy rules within a layer are evaluated, remember that evaluation is from the top down, but the first rule that matches ends further evaluation of that layer. Therefore, the most specific conditions, or exceptions, should be defined first to establish a match sooner and save CPU.

Understand and use the appropriate rule actions.

Understanding and using the appropriate rule actions, as well as utilizing Boolean operators, allows you to create and understand specific and general rules.

Use the appropriate rule action based on what you intend to authorize and decline.

- **OK**–No action taken
- **Allow**–Permits access
- **Deny**–Restricts access
- **Force Deny**–Restricts all access (cannot be overridden and committed immediately)
- **Exceptions**–Allowed or denied access based on the rule condition
- **Authenticate**–Controls authentication for the transaction
- **Force Authenticate**–Controls the relation between authentication and denial

Regulate rules further with Boolean operators.

Boolean operators provide additional conditions for rules. Use them to create specific rules.

- **And**–Used to provide more than one trigger; conditions must match both trigger conditions to match this rule
- **Negate**–Used to specify rule matches only when the condition is not satisfied.
- **Or**–Used to signify the rule can match on either of the triggers

These rules can be general or specific depending on the conditions of the rule and the default policy configuration.

About Policy Actions

The configured action for rules determine how transactions are enforced in policy. For example, a user wants to access a news website and the rule action is Allow. This result is user access to the news website because the Allow action in the rule. If the action is Deny, the user cannot access the website.

Actions allow you to create a limitless combination of rules to fit your company's security needs. Blue Coat recommends reviewing the following policy actions reference topics.

Ok Action

Ok actions prevent the overruling of a previous Deny or Allow action. Policy reads the Ok action as a *no action* rule. When policy reads through a rule, finds a match, and applies the indicated Ok action, the result is no action for the matched rule and skipping the rest of the layer.

Consider the following example. This policy implementation has three layers:

- Layer 1: Explicitly states a Deny rule
- Layer 2: States an exception rule
- Layer 3: Explicitly states an Allow rule

```
<proxy>
url.extenstion=.exe DENY

<proxy>
category=(sports) exception(content_filter_denied)

<proxy>
client.address=192.168.15.252/30 ALLOW
```

As discussed previously, the action of the last rule matched is the action that is enforced. In this example, the action of the last-matched rule is Allow.

The administrator intended to do the following:

- Deny ALL .exe files
- Allow a specific subnet to watch sports content

However, because the last rule matched is Allow, the matched subnet (12.168.15.252/30) can also access content with .exe files.

To avoid overriding previously desired rules, use the Ok action as follows:

```
<proxy>
url.extenstion=.exe DENY

<proxy>
client.address=192.168.15.252/30 OK
category=(sports) exception(content_filter_denied)
```

Because the match has an Ok action (no action) and no other rules apply to the Allow action, no previous actions are reversed.

Allow Action

The Allow action permits transactions. If policy evaluates a rule match, and the indicated action is Allow, policy permits the transaction. While most administrators are comfortable using the Allow action, many do not fully understand the effect it can have on policy. Specifically, the Allow action, depending on where it is being used in policy, can unintentionally reverse a previous denial the administrator did not intend to reverse. For example, an administrator may create a rule denying .exe files. If this Deny policy rule precedes an Allow rule, it could unintentionally allow a request that would have been denied based on some other request criteria.

In the CPL the administrator intended the third layer to allow specific users to access watch movie content, but actually, it is allowing everyone to access .exe files, reversing the previous denial.

```
<proxy>
url.extension=.exe DENY

<proxy>
category=(movies) exception (content_filter_denied)

<proxy>
client.address=192.168.15.252/30 ALLOW
```

Allow Action Usage – Best Practices

The best way to avoid having an Allow action reverse a previous action is to arrange your policy in the following way: Instead of having two layers – one with the general rule and another with the exceptions – create one layer in which the rules are arranged so that the exception group never matches on the original action at all. Because a layer evaluates until the request matches a rule, matching a condition (even one with no action) is still a match.

Policy rules with no action are legal syntax, but for those who find that it makes the policy harder to read, the Ok action can be used.

Using the recommendation, here is the proper implementation of the example used above:

```
<proxy>
url.extenstion=.exe DENY

<proxy>
client.address=192.168.15.252/30 OK
category=(movies) exception (content_filter_denied)
```

Placing the client and category rule in the same layer, and replacing the Allow with an Ok (no action), orders the rules so that the clients in the indicated subnet result in a match before the content-filtering category condition is evaluated. Because the match has no action, no previous Deny actions are reversed.

Deny and Force Deny Action

Although Deny is one of the most common policy actions used, it is often used incorrectly. Therefore, it is beneficial to understand the difference between Deny and Force Deny:

- Deny:
  - Restricts access based on the request
  - Can be overruled by a following Allow action on subsequent layers
- Force Deny:
  - Denies all access on the request
  - Cannot be overturned by a following Allow action on subsequent layers

Using Force Deny can be crucial, for not just for preventing a later policy from accidentally overturning a policy rule, but also for preventing any unnecessary processing request that the administrator does not intend to allow.

The following policy shows a typical incorrect implementation of Deny. Assume that a user, who is not in the subnet `my_users` makes a request for a non-approved executable. Also assume that the user-defined condition "executable" tests response headers and/or response data.

Layer One

Evaluation of the client address temporarily marks a user for denial by the policy processing engine since they are not a member of *my_users.*

Layer Two

Evaluation results in a category look up to determine if the user should receive a `content_filter_denied` exception.

Layer Three

Evaluation results in a request for the object from the origin server so that the policy-processing engine can determine whether or not the object satisfies the `condition=executable` and `conditions=!approve_application` conditions, and can then determine if it should return to the `too_risky` exception.

```
Define subnet_my users
192.168.23.5/8
192.168.0.0/16
end

<proxy>
client.address=!my_users DENY
<proxy>
category=(pornography, gambling) exception(content_filter_denied)
<proxy>
condition=executable condition=!approved_application exception(user_defined.too_risky)
```

In this case, a better policy implementation would be to replace Deny with Force Deny, which results in the immediate denial of any clients not in the `my_users` subnet for that layer rule, before continuing to evaluate on the next layer. This prevents any unnecessary processing of the user's request – a category lookup and server-side request in this particular example.

Although a matched Force_Deny action cannot be overruled by subsequent layers, the action does not always or automatically discontinue evaluation. If there is a subsequent layer with another subnet/IP- address condition, or no condition, that layer may still be evaluated, but the configured action can not override the Force Deny.

Similarly, using force_exception( ) instead of exception( ) results in users of the my_users subnet being immediately denied if they attempt to access pornography or gambling sites. This prevents any server-side request to determine which error page to present to the user. In this particular example, a force_exception( ) in layer three does not change the behavior of policy because the executable condition is the last rule in the policy. If there was additional policy, however, you would also want to use force_exception( ) to ensure the executables of non-approved applications are immediately denied with the indicated exception.

The following example illustrates the proper usage:

```
Define subnet_my users
192.168.23.5/8
192.168.0.0/16
end

<proxy>
client.address=!my_users FORCE_DENY
<proxy>
category=(pornography, gambling) force_exception(content_filter_denied)
<proxy>
condition=executable condition=!approved_application exception(user_defined.too_risky)
```

Deny and Force Deny Usage – Best Practices

When using the Deny action, proper organization is crucial. If placed before an Allow, the Deny may be overridden. If your intention is to immediately deny, replace the Deny action with a Force Deny. If your intention is not to immediately deny, place the Deny action at the bottom of the layer, removing any chance of it being overturned.

The same recommendations apply for exceptions and Force Exceptions. If placed before a Deny or Allow action, the exception may be overridden. If your intention is to focus on the exception, use a Force Exception. If that is not your intention, place the exceptions *after* Allow actions and *before* the Deny actions.

Authentication and Force Authentication

Do not confuse Authentication and Force Authentication with Deny and Force Deny.

- *Authenticate* either allows users to a specified realm or disables authentication for the policy transaction.
- *Force Authenticate* controls the relation between authentication and denial in a transaction.

Authenticate

When authentication is dependent on any condition that is not part of the clients identity, then some transactions from the client are authenticated while some are not. For example, if a client requests access to a website and policy knows the user request is set for denial, policy does not prompt the user to authenticate. On the other hand, if policy knows the user's request is allowed, policy prompts the user to authenticate.

However, the browser offers some credential types pro-actively. The default behavior of the ProxySG appliance is to forward any proxy credentials that it does not consume.

To prevent forwarding of proxy credentials in situations where there is no upstream proxy authentication, use the no_upstream_authentication option.

This example implements the following policy:

- Rule One: All traffic to `a.com` is authenticated.
- Rule Two: All traffic to `b.com` is authenticated by an upstream proxy.
- Rule Three: All other traffic is unauthenticated, and proxy credentials are not forwarded.

```
<proxy>
url.domain=//a.com/ authenticate(localr)
url.domain=//b.com/ authenticate(no)
authenticate(no, no_upstream_authentication)
```

Force Authenticate

Using the Force Authenticate action forces the user to authenticate even if the user request is going to be denied.

This can be useful for some administrators. For example, in cases concerning accounting and logging where a user is denied access to a blacklisted site, prompting the user to log in allows you to have a log entry of their authentication.

Force Authenticate in its' proper CPL syntax is indicated below.

```
authenticate.force(yes|no)
```

Force Authenticate has two settings: yes or no. The default configuration is no unless otherwise configured.

`authenticate.force(yes)`	Makes `authenticate()` a higher priority than `deny()` or `exception()`
`authenticate.force(no)`	`Deny()` and `exception()` have a higher priority than `authenticate()`

Authenticate and Force Authenticate Usage – Best Practices

- Use Authenticate and Force Authenticate in the `<proxy>` and `<admin>` layers and in proxy and administrative transactions. However, Force Authentication does not apply to `<cache>` layers or transactions.
- Using Force Authenticate slows down performance to send out authentication requests for every transaction. For this reason, Blue Coat recommends only using Force Authenticate when accounting and logging are beneficial. For example, you want to have a log entry of all users who attempt access to a blacklisted site.
- Use `authenticate.force(yes)` to ensure that the user IDs are available for access logging (including denied requests).
- Use `authenticate.force(no)` to configure an early denial.

For more information on Authenticate and Force Authenticate usage, see What is the difference between Authenticate and Force Authenticate policy actions.

Boolean Operators

The VPM and CPL utilize Boolean operators. The expressions used are

- `and`
- `or`
- `negate`

Each Boolean operator adjusts the rule in a specific way. See the table below.

Boolean Operators	CPL Syntax	CPL Example	Description
And	None	`client.address=1.1.1.1 url.domain=abc.com url.extension=(exe, com) DENY`	A match occurs only if the request meets both domain and extension triggers.
Negate	!	`client.address=1.1.1.1 url.domain=abc.com url.extension=!(exe, com) DENY`	A match occurs only if the request meets the domain trigger and *not* extension triggers.

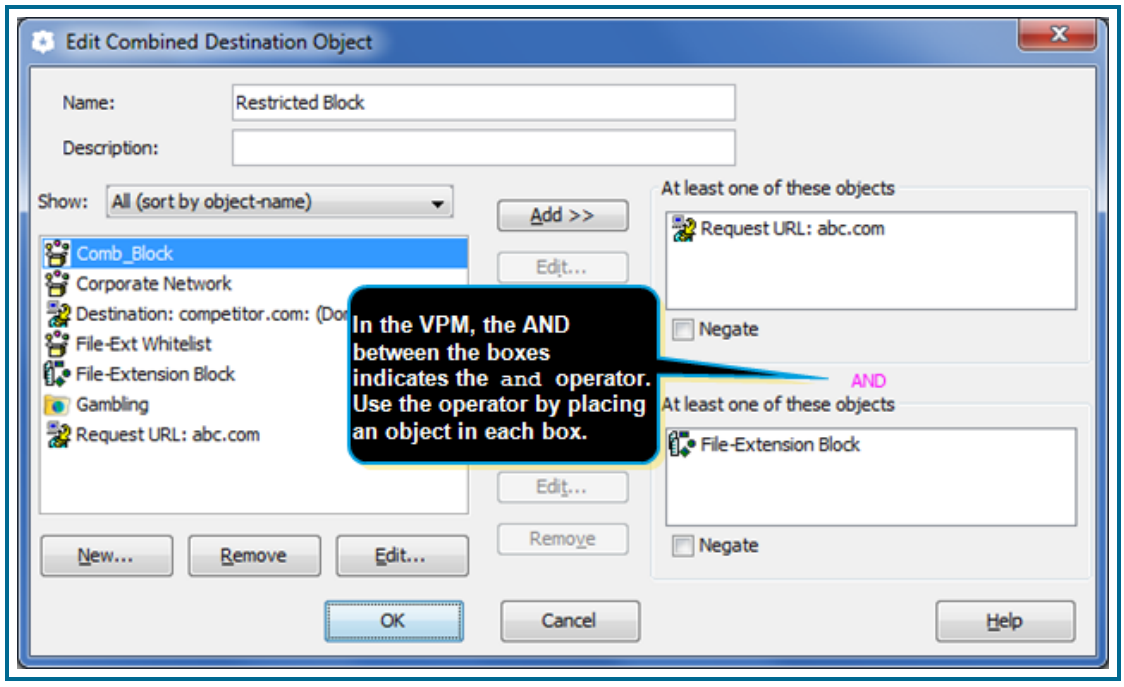
Or	||
	or
	,
		`client.address=1.1.1.1 url.-domain=abc.com time= (0900..1200 || 1300..1700)`
		or
		`client.address=1.1.1.1 url.extension=(exe, com)`
			A match occurs only if the request meets the time trigger `0900..1200` or `1300..1700`.
			or
			If the request meets the extension trigger `.exe` or `.com`.

And Operator

The and operator indicates a match on not one, but *both* triggers. In the following example, a match occurs only if the transaction meets both the domain and extension triggers. If the transaction matches only one of the conditions, the rule does not match.

```
client.address=1.1.1.1 url.domain=abc.com url.extension=(exe, com) DENY
```

The following screenshot demonstrates how the VPM displays the and operator.

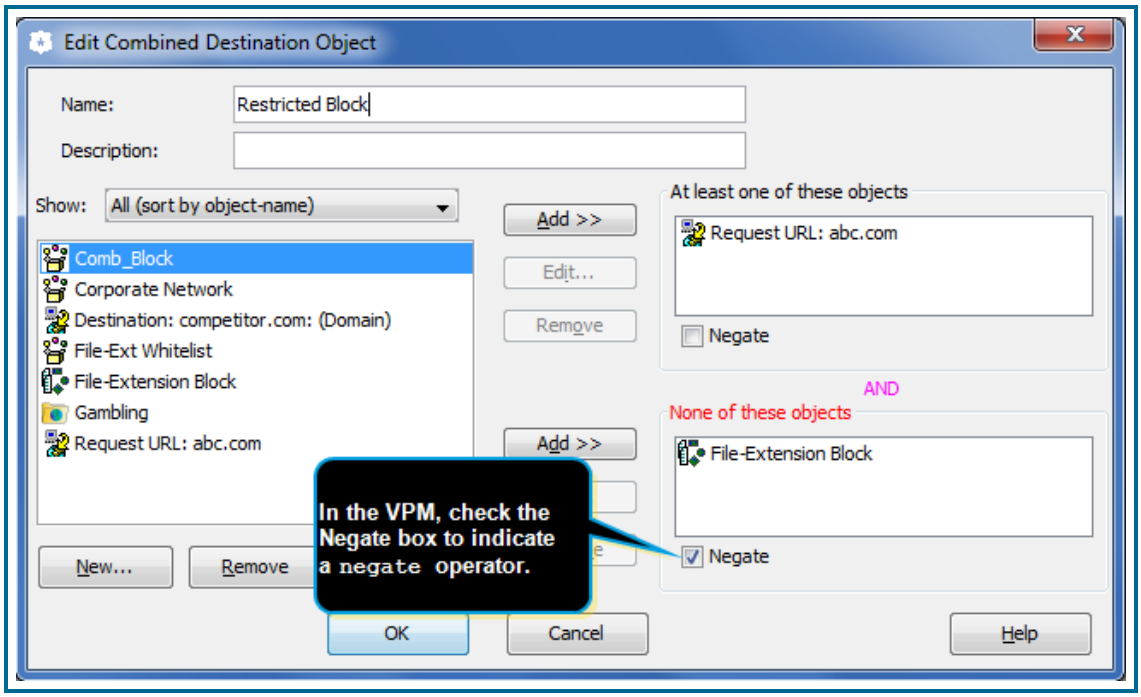


Negate Operator

The negate operator, as defined by a ! in the rule, indicates that the match must be made on both of the specified conditions. In this rule, a match only occurs if the request meets the domain trigger abc.com *and not* the extension triggers (exe, com).

```
client.address=1.1.1.1 url.domain=abc.com url.extension=!(exe, com) DENY
```

The following screenshot demonstrates a negated and operator. Selecting the **Negate** option negates the corresponding condition; that is, negate only matches when the condition is *not* satisfied.



To prevent unintended logical consequences, Blue Coat recommends using the negate operator sparingly.

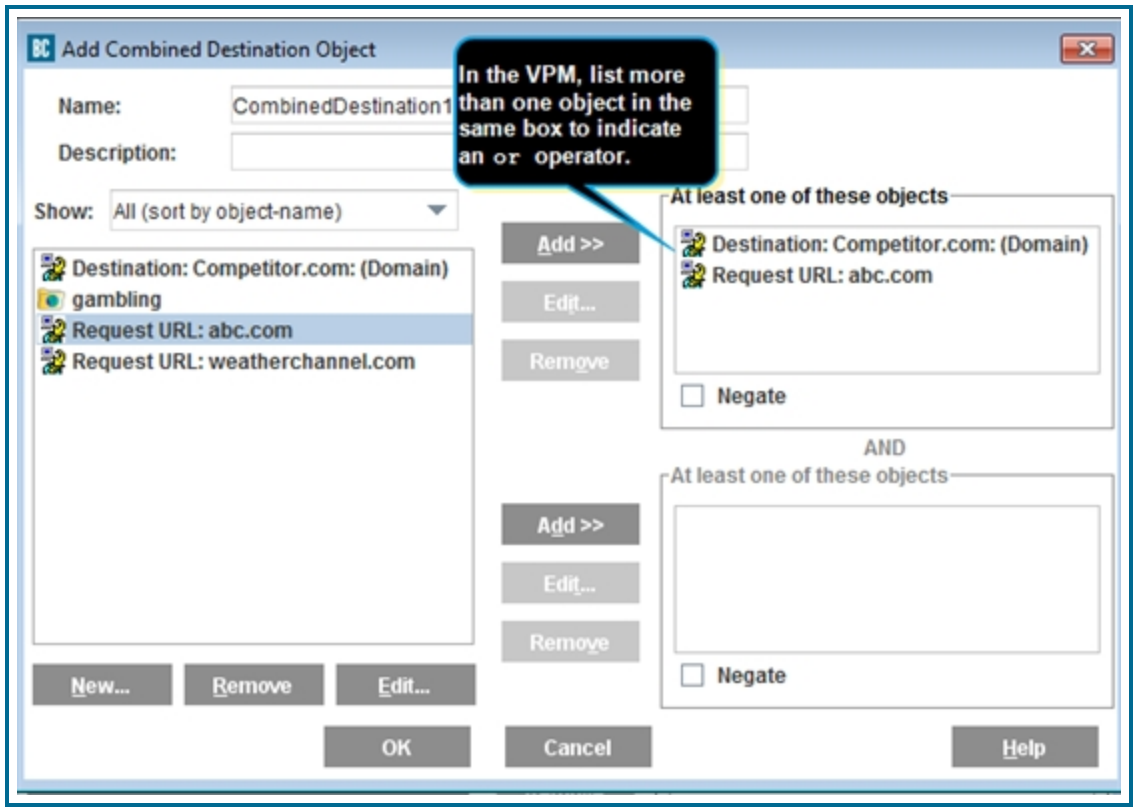
Or Operator

The or operator tells policy that the transaction can match only on either triggers, not both. This operator can be indicated in two ways: comma (,) or double-pipe characters (||). Whichever way it is entered, policy reads it as an or operator. For example, url.extension=(exe, com) indicates a match can only occur if the transaction matches exe or com url.extention triggers.

The following example uses the || syntax option:

```
time=(0900..1200 || 1300..1700)
```

In this example, a transaction only matches if it matches the time frame trigger 0900 to 1200 *or* 1300 to 1700.


Add Combined Destination Object
In the VPM, list more than one object in the same box to indicate an or operator.
Name:
CombinedDestination1
Description:
Show:
All (sort by object-name)
Destination: Competitor.com: (Domain)
gambling
Request URL: abc.com
Request URL: weatherchannel.com
Add >>
Edit...
Remove
At least one of these objects
Destination: Competitor.com: (Domain)
Request URL: abc.com
Negate
AND
At least one of these objects
Add >>
Edit...
Remove
Negate
New...
Remove
Edit...
OK
Cancel
Help

Construct Policy

Policy can become increasingly difficult to maintain as it grows. The following list provides ways in which you can create and maintain well-structured policy:

Configure Policy Rules Within a Layer

- Differentiate between specific and general rules by use of triggers and actions.
- Order rules from specific to general.

Reference: "Configure Policy Rules Within a Layer" on the next page.

Construct Policy Layers

Create policy layers and use descriptive comments.

Reference: "Construct Policy Layers" on page 31.

Separate Policy Layers

Separate layers by each logical function.

Reference: "Separate Policy Layers" on page 32.

Configure policy layers

- Configure policy layers to either align or divert from the default policy setting.
- Differentiate and order policy layers from general to specific.
- Create separate whitelists and blacklists.

Reference: "Configure Policy Layers " on page 33

Order Policy Layers

Use policy's intended order when creating layers.

Reference: " Order Policy Layers" on page 34

Configure Policy Rules Within a Layer

Maintenance is substantially difficult if the rules within a layer are not organized. In addition, rules can potentially override one another if not ordered correctly. Creating a model policy of rules provides an easier way to maintain and configure rules.

To create a model for organizing and structuring policy rules, Blue Coat recommends the following:

- Organize rules in a structured rule order: specific to general.
- Use triggers and actions to define specific policy.(See table below.)

Create Model Policy with Rules – Best Practices

Organize Policy Rules in a Layer from Specific to General

Rules are evaluated from top to bottom until there is a match. With that in mind, Blue Coat recommends ordering rules from most-specific to most-general in a policy layer. Placing rules most likely to match at the beginning of the layer saves CPU, creates a structure benefiting maintenance, and prevents potential policy overrides.

Specific rules are defined by the number of defining triggers and actions.

Utilize Triggers and Actions to create Specific Policy

You are not limited to creating rules with a single trigger and action – general rules. You can use multiple triggers and actions to create *specific* policy rules. The more triggers applied to a rule, the more specific they become.

The following table provides the available triggers and actions.

Triggers	Actions
User	Allow/Deny
URL	Redirect client
Protocol	Block pop-ups
Time/Date	Modify logging

Trigger Examples:

- User: Subnets, IP address
- URL: Host, Port, Path, File, Extension, Query
- Protocol: `http`
- Time/Date: 0900, 1600, 2400

Construct Policy Layers

As policy grows and becomes more complex, maintenance becomes a significant issue. To ensure simpler maintenance, follow these recommendations:

- Use descriptive names and numbers for each layer.
- Use descriptive comments about each layer.
  - State the purpose of the layer.
  - State the date created.
  - State the creators name.

Policy Layers Construction – Best Practices

Use Descriptive Names and Numbers for Each Layer

As complex policy grows, you can lose track of the intention of the layer making it difficult to maintain. Blue Coat recommends the following:

- Name the layer based on the filtering function it provides.
- Number the layer to avoid confusion over multiple same-type layers.
- Place a three-letter acronym at the beginning of the layer name to ensure the original layer type and name are not lost.

For example:

- Web Access Layer for Corporate: **WAL Corporate(1)**
- Web Authentication Layer for the California Branch: **WAU Branch CA(1)**
- Admin Authentication Layer for the HR Department: **AAU HR Department(1)**

Use Descriptive Comments About Each Layer

Blue Coat recommends adding descriptive comments about the layer. Comments include:

- **Layer name**–Defining the purpose of the layer allows you to keep track of what the configured rule actions of the layer are for.
- **Date created**–Placing the date of creation not only informs you when the policy was created but also how up to date the layer rules are.
- **Creator name**–Defining the creator's name informs you of who has created the layer as well as who you should contact in case of discrepancies.

If there is additional information crucial to the layer, use comments to record them.

Enter comments with one of the following methods.

- In CPL–Add a semi-colon (;) after or above the policy created.
- In VPM–Use the **Comment Object Reference** option.

Any comments you write to aid in labeling the policy layer will not impact the policy.

Separate Policy Layers

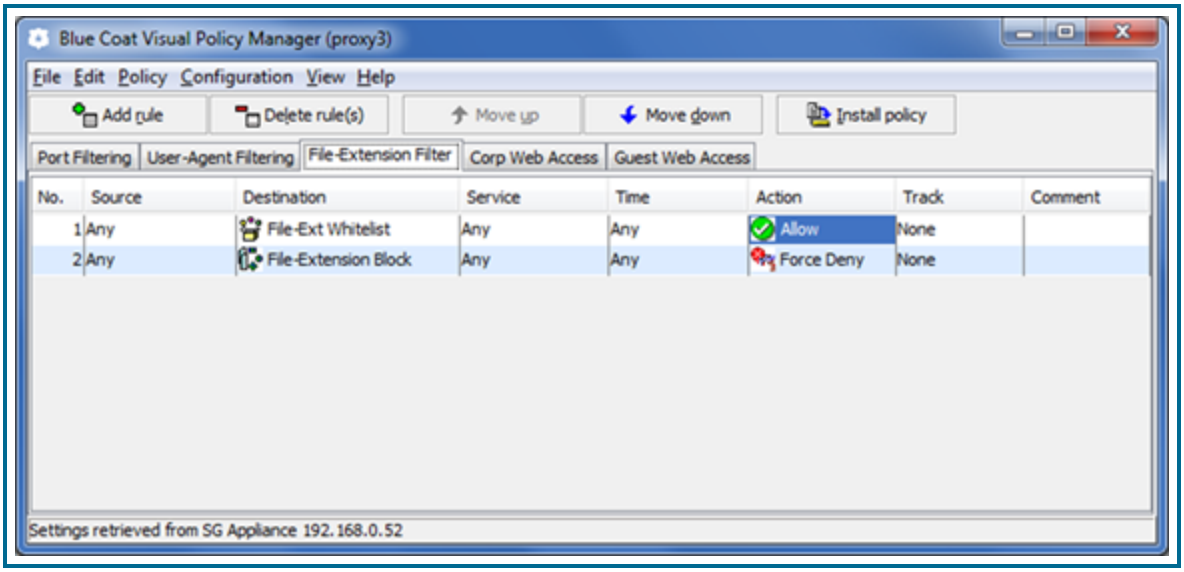
Organization is imperative when creating policy; administrators need to organize their policy in a consistent manner to ensure simple maintenance and optimized policy. This is achieved by using separate layers for each logical function.

Layer Separation—Best Practices

Create Separate Layers for Each Logical Function

Blue Coat recommends that you create separate layers with different functions. Make policy decisions as independent as possible, and express each policy in one layer or multiple adjacent layers. Create one layer per action type; this not only makes it easier to maintain, but helps to prevent you from creating rules that unintentionally cancel out the action of other rules.

For example, in the image below, a user creates separate layers for Port Filtering, User-Agent Filtering, and File Extension. In this scenario, the user placed rules related to Port Filtering in the **Port Filtering Layer**, rules related to User-Agent in the **User-Agent Filtering Layer**, and rules related to File Extension in the **File Extension Layer.**



To sum up, Blue Coat recommends using separate layers for each logical function.

- Port Filtering
- User Agent Filtering
- SSL Interception
- SSL Access
- Corp Web Access
- Group Web Access
- Forwarding
- Admin Auth
- Admin Access
- Web Auth
- Corp Web Filter
- Guest Web Filter

Examples

- Layer 1: **Web Authentication Layer**—Bypass authentication.
- Layer 2: **Web Access Layer**—Deny the **CEO** user access to sport sites.

```
<proxy>
url.domain=yahoo.com authenticate(no) ALLOW
<proxy>
Category=(sports) user="CEO" DENY
```

There is one layer for authentication and one layer for access—one layer per action type.

In addition to these layers, best practices include creating separate layers for the following rule actions:

- Force Deny
- Deny
- Allow

Configure Policy Layers

Policy always aligns with the configured default policy setting (Allow or Deny), unless configured otherwise. You can depart from the default policy by creating separate layers for:

- Deny
- Allow
- Exception
- Global whitelist/blacklist

However, the order in which these layers are arranged is crucial to creating and maintaining policy. Without a structured policy layer order, you might create policy that overrides an earlier policy.

To create a well-structured policy, Blue Coat recommends the following:

- Create an organized layer structure: place general layers early on with exception layers later
- Create separate global whitelists and blacklists

Policy Layer Configuration – Best Practices

Place general rule layers early on with exception layers later.

As you create policy aligning with the configured default setting, Blue Coat recommends that you place general policy earlier with exception layers later. This organization allows for easier maintenance and configuration of layers.

The following show a model for layers ordering for each default setting:

Allow

- Layer 1–Impose any authentication requirements (all access must be authenticated)
- Layer 2–Apply general rules specific types of requests
- Layer 3–Create bitrate exceptions to the layers above
- Layer 4–Create exceptions to the layers above

```
define conditions corporate_buddies
 im.buddy_id=nameONE
 im.buddy_id=nameTWO
 ….
 im.buddy_id=nameTHREE
end

<proxy>
authenticate(corp_realm)
<proxy>
streaming.content=yes max_bitrate(56k)
```

```
im.buddy_id=!corporate_buddies im.strip_attatchments(yes)
<proxy>
Group=executive max_bitrate(256k)
Group=marketing max_-bitrate(156k)
<proxy>
Group=public_relations im.strip_attatchments(no)
```

Deny

- Layer 1: Explicitly allow access to the intended user or set of users
- Layer 2: Impose any authentication requirements (all access must be authenticated)
- Layer 3: Exclude any specific type of requests

```
define subnet coporate_subnet
 10.10.12.0/24
end

<proxy>
client.address=corporate_subnet ALLOW
<proxy>
authenticate(corp_realm)
<proxy>
url.domain=playboy.com DENY
category=(gambling, hacking, chat)exception(content_filter_denied)
```

Create separate whitelists and blackslists

Based on policy's configured default setting, create a separate layer for whitelists and blacklists.

- Deny = whitelists
- Allow = blacklists

Blue Coat recommends placing these layers later in policy.

Order Policy Layers

The order of policy layers is important. The ProxySG appliance evaluates policy layers in the order in which they are listed in the VPM. When the ProxySG appliance is evaluating policy layers, it does not execute a given rule as soon as it finds a match. Rather, the appliance compiles a list of all the rules that meet the condition; when it has evaluated all the policy layers, the appliance evaluates the list of matched rules, resolves any apparent conflicts, and then executes the required actions. If there is a conflict between rules in different policy layers, the last-matched rule evaluated takes precedence.

The following are examples for layer no conflict/conflict.

No conflict

Assume there are two **Web Access Layers**; one contains a rule stating that Sales personnel can access certain websites without authentication, and the other rule states that when they do access these websites, limit the available bandwidth. The order is irrelevant because there is no conflict between the rules in the layers.

Conflict

Assume all URL requests from purchasing department members are directed to a single proxy server. To discourage employees from surfing the web excessively during business hours, a company creates a **Web Authentication Layer** rule stating:

```
Whenever a client request arrives, prompt the client to authenticate.
```

Members of the purchasing department, however, need to access specific websites for business reasons, and the company does not want to require authentication every time they do this. So they create a **Web Access Layer** rule that states:

```
If any member of the purchasing department sends a request to a specific URL contained in a
combined-object list, allow access.
```

These rules conflict when placed incorrectly; if the administrator placed the second rule in the first layer, it conflicts with the subsequent layer holding the first rule.

The policy layer with the first rule needs to come first in evaluation order, otherwise it is overridden by the second rule in a subsequent layer.

Policy Layer Order – Best Practices

Layers have an intended order. The VPM lists them in the preferred order (displayed in the **Policy Layer** section menu). Blue Coat recommends using the VPM preferred ordering for both CPL and VPM created layers, as shown below.

CPL Layer Order (From Top Down)	VPM Layer Order (From Top Down)
`<admin>`	Admin Authentication Layer
`<admin>`	Admin Access Layer
`<dns-proxy>`	DNS Access Layer
`<proxy>`	SOCKS Authentication Layer
`<ssl-intercept>`	SSL Intercept Layer
`<ssl>`	SSL Access Layer
`<proxy>`	Web Authentication Layer
`<proxy>`	Web Access Layer
`<cache>`	Web Content Layer
`<forward>`	Forwarding Layer

As you create policy layers, order them from general to specific; that is, establish a general rule in an early policy layer, then create exceptions in later policy layers. This helps to avoid conflicts and unintentionally overriding previous policy.

The following is an example of a model policy.

- Layer 1: Troubleshooting Layer (**Web Access Layer**–Policy tracing, access logging, and so on).
- Layer 2: **SSL Interception**.
- Layer 3: **SSL Access**.
- Layer 4: **Web-Auth**.
- Layer 5: NAME: URL filtering, BW Management, ICAP, and so on (**Web Access Layer**: per action type).
- Layer 6: NAME: ICAP-Response, Caching Rules, and so on (**Web Content Layer**: per action type).

Optimize Policy Performance

Optimization is crucial to saving CPU and speeding up the evaluation process. The following topics provide specific details about how to optimize policy.

- "Prerequisites for Ensuring Proper Policy Function" on page 37–Verify that your ProxySG appliances are configured for optimal policy performance.
- "Rule Placement" on page 38–Place rules most likely to match at the top of the layer.
- "Name Definitions " on page 39–Use definitions and subnets to minimize the number of rules.
- "Policy Coverage Feature" on page 49–Process current policy to find frequently matched rules.
- "URL-Based Rule Optimization" on page 47–Specify the correct URL conditions.
- "Guard Optimization" on page 50–Optimize guard usage with layer and section guards.
- "Local Database: Advanced Policy Optimization" on page 52–Use the Local Database for larger URL lists.

These recommendations are discussed in the following sections.

Prerequisites for Ensuring Proper Policy Function

Blue Coat recommends the following ProxySG appliance configurations to ensure optimized policy performance.

ProxySG Feature	Recommended Configuration
Intercepted Service	Explicit or transparent deployment configured.
Authentication Policy	◦ Authentication realm settings relative to the company's environment ◦ IWA, LDAP, RADIUS and others configured and functioning as needed
SSL Interception Policy	SSL certificate created and verified.
Content Analysis	Required ICAP object configured.
Bandwidth Management	Bandwidth Management Class configured.
Custom-Defined Exception	Custom exception created and configured
Blue Coat Web Filter (BCWF)	◦ Appropriate license acquired. ◦ Database installed.
Classification Policy	◦ Appropriate license acquired ◦ Database installed
Geolocation Service	◦ Appropriate license acquired. ◦ Database installed.
Web Application Protection	WAP subscription obtained and verified.

For more information (setting an authentication realm | configuring ICAP server | enabling a content filter provider | intercepting traffic) see the ProxySG First Steps Guide.

Rule Placement

Place rules most likely to match at the beginning of a layer to speed up policy evaluation. Prompting for a match sooner improves performance; once a match is made, the rest of the layer is skipped. This saves CPU and optimizes policy evaluation, but it also provides simple maintenance.

The following is an example of a typical non-optimized policy:

```
<proxy>
url.domain=www.abc.com DENY
url=http://www.def.com/chatweb/ DENY
url=http://www/hgi/com/finance/ DENY
url.domain=jkl.com DENY
im.buddy_id=john DENY
url.dominan=www/mno.com DENY
im.buddy_id=bill DENY
```

The following is an example of how the same rules were optimized:

```
<proxy>
url.domain=www.abc.com DENY
url.domain=www.mno.com DENY
url.domain=www.jkl.com DENY
url=http://www.def.com/chatweb/ DENY
url=http://www.hgi.com/finance/ DENY
im.buddy_id=john DENY
im.buddy_id=bill DENY
```

This can only be done if the ordering of rules within a layer does not change the intended behavior of the layer.

Name Definitions

You can use definitions to further define and organize rules. By doing so, you are minimizing the amount of rules created. There are various types of name definitions. Each of these definitions is given a user-defined name that is then used in rules to refer to the definitions.

The following are the different types of name definitions.

- "Subnet Definition" on the facing page–Binds a user-defined label to a set of IP addresses or IP subnet patterns
- "Condition Definition" on page 41–Binds a user-defined label to a set of conditions for use in a `condition=` expression
- "Category Definition" on page 41–Enables you to extend vendor content categories or to create your own
- "Action Definition" on page 42 –Binds a user-defined label to a sequence of action statements
- "Transformer Definition" on page 43–Specifies a transformation that is to be applied to an HTTP response.
    - `url_rewrite`
    - `active_content`
    - `javascript`

Definition Usage – Best Practices

Because of the many types of definitions, Blue Coat recommends primarily using category and subnet definitions, and other definitions only as needed. Category and subnet definitions will help condense policy the most.

The following are examples for both category and subnet definitions:

Category Definitions

Consider the following non-optimized policy:

```
<proxy>
url.domain=etrade.com OK
url.domain=nyse.com OK
url.domain=stocktrader.com OK
url.domain=scottrade.com OK
category=(brokerage/trading) exception(content_filter_denied)
```

To minimize the amount of rules and optimize policy, you would use a category definition, as shown below:

```
define category exception_sites
 etrade.com
 nyse.com
 stocktrader.com
 scottrade.com
end

<proxy>
category=exception_sites OK
category=(brokerage/trading) exception(content_filter_denied)
```

The optimized policy, with its defined definition and condensed rule list, is evaluated more efficiently and saves CPU.

Subnet Definitions

The following is a typical non-optimized implementation of policy:

```
<proxy>
client.address=10.0.0.0/8 category=(gambling) OK
client.address=192.168.0.0/16 category=(gambling) OK
client.address=216.52.23.3 category=(gambling) OK
client.address= 216.52.23.5 category=(gambling) OK
category=(gambling) exception(content_filter_denied)
```

This example shows how the subnet definition optimizes the above policy:

```
define subnet test_network
 10.0.0.0/8
 192.168.0.0/16
 216.52.23.3
 216.52.23.5
end
```

```
<proxy>
client.address=test_netowrk category=(gambling) OK
category=(gambling) exception(content_filter_denied)
```

Subnet Definition

A subnet definition binds a user-defined label to a set of IP addresses or IP subnet patterns. Use a subnet definition label with any of the conditions that test part of the transaction as an IP address, including:

- `client.address=`
- `proxy.address=`
- `request.header.header_name.address=`
- `request.x_header_header_name.address=`
- `server_url.address=`

The listed IP addresses or subnets are considered to have a Boolean `OR` relationship, no matter whether they are all on one line or separate lines.

The following is a syntax example:

```
Define subnet label
 {ip_address | subnet} {ip_address | subnet} {ip_address_range}
 {ip_address_wildcards}…
end
```

Where:

- `label`—A user-defined identifier for this subnet definition
- `ip_address`—IP address; for example, `10.1.198.0`
- `subnet`—Subnet specification; for example, `10.26.198.0/16`
- `ip_address_range`—IP address range; for example, `192.0.2.0-192.0.2.225`

- `ip_address_wildcards`—IP address specified using wildcards in any octet(s); for example, `*10.25.*.0 or 10.*.*.0`

For example:

```
define subnet local_net
 1.2.3.4 1.2.3.5 ; can list individual IP addresses
 2.3.4.0/24 2.3.5.0/24 ; or subnets
 2.3.4.0-2.3.4.255 ; or an IP address range
 2.3.*.* ; or IP address wildcards
end

<proxy>
client.address=local_subnet deny
```

Condition Definition

A condition definition binds a user-defined label to a set of conditions for use in a `condition=` expression.

Unlike other triggers, the `condition=` trigger can test multiple aspects of a transaction. Because condition definitions can include other triggers, `condition=` triggers can test multiple parts of the transaction state. For condition definitions, the manner in which the condition expressions are listed is significant. Multiple condition expressions on one line, separated by whitespace, are considered to have a Boolean `OR` relationship.

The following is a syntax example:

```
Define condition label
 condition_expression…
 ...
end
```

Where:

- `label` —A user-defined identifier for a condition definition. Used to call the definition from an `action.action_label( )` property.
- `condition_expression` — Any of the conditions available in a rule. The layer and timing restrictions for the defined condition depend on the layer and timing restrictions of the contained expressions.

The `condition=`*`conditions`* is one of the expressions that can be included in the body of a `define condition` definition block. In this way, one condition definition block can call another condition-related definition block, so that they are in effect *nested*. Circular references generate a compile error.

For example:

```
define condition white_list
 condition=extension low_risk
 condition=internal_prescanned
end
```

Category Definition

Category definitions are used to extend vendor content categories or to create your own. These categories are tested (along with any vendor defined categories) using the `category=` trigger.

The following is a syntax example:

```
Define category category_name
 url_patterns
end
```

Where:

- `category_name`—If `category_name` matches the name of an existing category from the configured content filtering service, this is used to extend the coverage of that category; otherwise it defines a new user-defined category. `Category_name` can be used anywhere a content filter category name would normally be used, including any `category= tests`.
- `url_patterns`—A list of URL patterns. A pattern can include the scheme, host, port, path, and query components of the URL. If the pattern does not specify a component the corresponding component of the URL is not tested and can have any value.

For example:

```
define category Grand_Canyon
 Kaibab.org
 ww2.nature.nps.gove/ard/parks/grca/
 nps.gove/grca/
 grandcanyon.org
end
```

Action Definition

An action definition binds a user-defined label to a sequence of action statements. The `action()` property has syntax that allows for individual action definition blocks to be enabled and disabled independently, based on the policy evaluation for the transaction. When the action definition block is enabled, any action statements it contains operate on the transaction as indicated by their respective arguments.

Action statements that must be performed in a set sequence cannot overlap; list them within a single action definition block.

The following is a syntax example:

```
Define action label
 list of action statements
end
```

Where:

- `label` – A user-defined identifier for an action definition. Only alphanumeric, underscore, and dash characters can be used in the label given to an action definition.
- `list of action statements` – A list of actions to be carried out in sequence.

For example:

The following is a sample action given the name `scrub_private_info`, that clears the `From` and `Referer` headers (which normally could be used to identify the user and where they clicked from) in any request going to servers not in the internal domain.

```
<cache>
 url.domain=!my_internal_site.com action.scrub_provate_info (yes)

define action scrub_private_info
 set ( request.header.From, “”)
```

```
 set ( request.header.Referer, “”)
end
```

Notice that the object on which the `set()` action operates is given in the first argument, and then appropriate values follow, in this case, the new value for the specified header. This is common to many of the actions.

Transformer Definition

A transformer definition is a kind of name definition that specifies a transformation that is to be applied to an HTTP response. There are three types:

- `url_rewrite`
- `active_content`
- `javascript`

These types are discussed in the following sections.

`url_rewrite`

A `url_rewrite` definition defines rule for rewriting URLs in HTTP responses. The URLs are either embedded in tags within HTML, CCS, JavaScript, or ASX documents, or they are contained in HTTP response headers. In addition to rewriting URLs, you can also rewrite arbitrary JavaScript.

This transformer takes effect only if it is also invoked by a transform action in a define action definition block, and that block is in turn called from an `action()` property.

For each URL found within an HTTP response, a `url_rewrite` transformer first converts the URL into absolute form, then finds the first `rewrite_url_substring` or `rewrite_url_prefix` statement whose `server_URL_substring` matches the URL being considered. If such a match is found, then that substring is replaced by the `client_url_substring`.

Matching is always case-insensitive.

The following is a syntax example:

```
Define url_rewrite transformer_id
 rewrite_url_substring "client_url_substring" "server_url_substring"
 rewrite_url_substring "client_url_substring" "server_url_substring"
 rewrite_script_substring "client_substring" "server_substring"
 ...
end
```

Where:

- `transformer_id` – A user-defined identifier for a transformer definition block. Used to invoke the transformer using the transform action in a define action definition block.
- `rewrite_url_substring` – Matches `server_url_substring` anywhere in the URL.
- `rewrite_url_prefix` – Matches `server_url_substring` as prefix of the URL.
- `rewrite_script_substring` – Matches and rewrites arbitrary substrings of any pattern inside any unrecognized tag or attribute, including those that cannot validly contain URLs.
- `client_url_substring` – A string that replaces `server_url_substring` when that string is matched for a URL in the retrieved document. The portion of the URL that is not substituted is unchanged.
- `server_url_substring` – A string that, if found in the server URL, is replaced by `client_url_substring`. The comparison is done against original normalized URLs embedded in the document.

Both client _url_substring and sever_url_substring are literal strings. Wildcard characters and regular expression patterns are not supported.

For example:

```
<proxy> ; server portal for example
 url=example.com/ action.example_server_portal(yes)

; this transformation provides server portaling for example non-video content

define url_rewrite example_portal
 rewrite_url_prefix “http://www\.exampleexample\.com/(.*)”,
end

; This action runs the transformation for example server portaling for http content

; Note that the action is responsible for rewriting related headers
```

active_content

An active_content definition defines rules for removing or replacing active content in HTML or ASX documents. This definition takes effect only if it is invoked by a transform action in a define action definition block, and that block in turn enables an action() property as a result of policy evaluation.

Active content transformation acts on the following four HTML elements in documents:

- <applet>
- <embed>
- <object>
- <script>

In addition, a script transformation removes any JavaScript content on the page. For each tag, the replacement can either be empty (thus deleting the tag and its content) or new text that replaces the tag. Multiple tags can be transformed in a single active content transformer. Pages served over and HTTPS tunneled connection are encrypted so the content cannot be modified.

Transformed content is not cached, in contrast with content that has been sent to a virus scanning server. Therefore, a transformer can be safely triggered based on any condition, including client identity and time of day.

The following is a syntax example:

```
Define active_content transformer_id
 Tag_replace HTML_tag_name<<text_end_delimiter
 [replacement_text]
 Text_end_delimiter
 [tag_replace ...]
...
end
```

Where:

- transformer_id–A user-defined identifier for a transformer definition block. Used to invoke the transformer using the transform action in a define action definition block.
- HTML_tag_name–The name of an HTML tag to be removed or replaced, as follows:

- `applet`—Operates on the `<applet>` element, which places a Java applet on a web page.
- `embed`—Operates on the `<embed>` element, which places an object, such as a media file, on a web page.
- `object`—Operates on the `<object>` element, which places an object, such as an applet or media file, on a web page.
- `script`—Operates on the `<script>` element, which adds a script to a web page. Also removes any JavaScript entities, strings, or events that the page might display.

  If the tag_replace keyword is repeated within the body of the transformer, multiple HTML tags can be removed or replaced.

- `text_end_delimiter`—A user-defined token that does not appear in the replacement text and does not use quotes or whitespace. The delimiter is defined on the first line, after the required double angle brackets (<<). All text that follows, up to the second use of the delimiter, is used as replacement text.
- `replacement_text`—Either blank, to remove the specified tag, or new text (including HTML tags) to replace the tag.

For example:

```
<proxy>
url.domain=!my_site.com action.strip_active_content(yes)
define active_content strip_with_indication

tag_replace applet <<EOT
 <B>APPLET content has been removed</B>
EOT
tag_replace embed <<EOT
 <B>EMBED content has been removed</B>
EOT
tag_replace object <<EOT
 <B>OBJECT content has been removed</B>
EOT
tag_replace script <<EOT
 <B>SCRIPT content has been removed</B>
EOT

define action strip_active_content
 transform strip_with_indication
end
```

javascript

A `javascript` definition is used to define a *javascript transformer*, which adds JavaScript that you supply to HTML responses.

The following is a syntax example:

```
transformer_id
 javascript-statement
 [javascript-statement]
 ...
end
```

Where:

- `transfomer_id` – A user-defined identifier for an action definition. Only alphanumeric, underscore, and dash characters can be used in the label given to an action definition.

- A `javascript-statement` has the following syntax:
  ```
  javascript-statement ::= section-type replacement
  section-type ::= prolog | onload | epilog
  replacement ::=<<endmarker newline lines-of-text newline endmarker
  ```

  This allows you to specify a block of java script to be inserted at the beginning of the HTML page (prolog), to be inserted at the end of the HTML page (epilog), and to be executed when parsing is complete and the page is loaded (onload). Each of the section types is optional.

Example

The following is an example of javascript transformer that adds a message to the top of each web page, used as a part of a simple content filtering application:

```
js_transformer
 onload<<EOS
  var msg = “This site is restricted. Your access has been logged.”;
  var p = document.createElement(“P”);
  p.appendChild(document.createTextNode(msg));
  document.body.insertBefore(p, document.body.firstChild);
 EOS
end

define action js_action
 transform js_transformer
end

<proxy>
category=restricted action.js_action(yes)
```

The VPM uses javascript transformers to implement popup-ad blocking.

URL-Based Rule Optimization

Administrators often use regular expressions (regex) when entering URL-based rules into CPL even when there are alternatives and less-intensive conditions. However, regex rules are highly CPU intensive. For that reason, Blue Coat recommends using regex only when absolutely necessary.

If you need to use regex, Blue Coat recommends the following:

- Use the appropriate URL condition; be specific in what you are looking *at* and *for*.
- Use care when using special characters
- Avoid the use of CPU-intensive url.regex

These topics are discussed in the following sections.

Regex Usage – Best Practices

Use the appropriate URL condition

When using regex, use the correct conditions and be specific in what you are looking *at* and *for*. This helps you save CPU as policy evaluates the layers.

The following chart expresses the conditions for what you are looking *at* in this URL:

http://www.example.blah.com:81/path.path_path/demo_file.html?sess=1234

Section	URL Portion	Regex Condition
Protocol	`http`	url.scheme=

Host	`www.example.x.com`
	
	`x.com`
		url.host=
		url.host.regex=
		url.address=
		
		url.domain=

Port	`:81`	url.port=
Path	`/path.path_path/demo_file.html?sess=1234`	url.path= url.path.regex
Extension	`html`	url.extension
Query	`sess=1234`	url.query= url.query.regex=
URL	`http://www.example.x.com:81/path.path_path/demo_file.html?sess=1234`	url= url.regex=

The following chart expresses being specific in what you are looking for:

Regex	Finds
`url.host.regex=\.com$`	Finds host ending in `.com` Note: The period is escaped
`url.host.regex=x+`	Matches x one or more times
`url.host.regex=x{2.5}`	Matches x two to five times
`url.path.regex=^\/bad-directory\/`	Finds all URLs that begin with `/bad-directory/`
`url.query.regex= login=matt`	Finds query strings that contain `login=matt`

Use care when using special characters

Special characters are often misperceived. Be careful when using special characters such as: `\ ^ % . | ? * + ( ) [ ] { }`

Wildcards * and `.*` are CPU intensive and almost always unnecessary.

Regex is more costly than special substring matches. Blue Coat recommends using substrings over regex, if possible. However, when using special characters, consider the following:

More CPU: `url.regex=*yahoo.com*`
A little less CPU: `url.regex=yahoo.com*`
Least CPU: `url.host.substring=yahoo.com`

`url.host.substring=yahoo.com` has the least amount of CPU because it is the most specific. Being specific is crucial in properly using regex.

Avoid using url.regex

Do not use `url.regex=`. The entire URL can contain a large amount of characters, and the wrong use of regex causes policy to evaluate every single character for every transaction; CPU intensive policy.

Policy Coverage Feature

Use the policy coverage feature to review your policy and to determine the frequency with which rules and objects match proxied requests. Having a list of matched requests allows you to organize your policy rules from most likely to match to least likely to match, optimizing policy performance.

Unlike a policy trace, which you must enable and disable through CPL, policy coverage is enabled and running by default. The appliance resets the policy coverage counter whenever a new policy is installed.

To determine which rules match the proxy requests and the frequency with which the rules are "hit," display the current policy coverage in the Management Console (select **Statistics > Advanced** and scroll down to **Policy**. Then click **View current policy coverage**).

The Policy Coverage page displays all policy file types (Visual, Local, Central, and Forward) on the appliance in CPL. The number of times that each rule is hit is listed to the left of each policy item that can be tracked. The following is an example of the output in the Policy Coverage page.

```
: ; Installed Policy -- compiles at: Mon, 03 Mar 2016 14:21:10 UTC

      :; default proxy policy is DENY
      :
      :; Policy Rules
      : <proxy>
      34: Authenticate(local) authenticate.force(no) authenticate.mode(origin)

      : <proxy>
      34: Authenticate.guest("guest", 0, "local")

      0: <proxy> user.is_guest=yes (0)
      0: DENY url.domain=//www.google.com/ (0)
      0: DENY streaming.client=yes (0)

      : <proxy>
      1: DENY url.domain=//www.tinydeal.com/ (1)
```

Domains in a define section do not have conditions, so they are not included in coverage. For example, consider a transaction involving a domain in the following section:

```
define url.domain condition my_domains
 example.com
 company.com
 test.com
end
```

Policy coverage tracks when the `my_domains` hit; however, it does not track transactions involving specific URLs within the `my_domains` condition.

Guard Optimization

Policy is optimal when layer and section guards are defined. Guards essentially prevent unnecessary evaluation of a layer or set of rules by singling out common sets of triggers and properties to avoid having to repeat them each time. The use of guards results in an organized and reduced amount of policy rules and increased policy performance.

There are two types of guards available to optimize policy:

- Layer guards
- Section guards

Layer and section guards are similar in function; however, section guards set conditions for only a grouped set of rules called sections, not the entire layer.

Layer Guards

Layer guards test the common condition before evaluating the subsequent rule or set of rules in the layer; preventing any unnecessary evaluation of subsequent rules in a layer.

Section Guards

The rules in layers can optionally be organized in one or more sections, which is a way of grouping rules together. A section consists of a section header followed by a list of rules.

The section guard condition test rules organized into sections (a grouped set of rules). There are four types of sections:

- `[rule]` - Used to logically organize policy into a section.
- `[url]` - Used to group a number of rules that test the URL, and restricts the syntax of the rules in the section.
- `[url.domain]`- Used to group a number of rules that test the URL domain, and restricts the syntax of the rules in the section.
- `[server url.domain]` - Used to test the domain of the URL used to fetch content from the origin server, and restricts the syntax and rules in the section.

Three of the section types, `[url]`, `[url.domain]`, and `[server_url.domain]`, provide optimization for URL tests.

Just as you can with layers, you can improve policy clarity and maintenance by grouping rules into sections and converting the common conditions and properties into *guard expressions* that follow the section header. A guards expression allows you to take a condition that applies to all the rules and put the common condition next to the section header, as in `[url.domian] group=sales`.

The following example shows policy without section guards.

```
<Proxy>
url.domain=abc.com/sports deny
url.domain=nbc.com/athletics deny
url.regex=”sports|athletics/; access_server(no)
url.regex=”\.mail.\” deny
url=www.bluecoat.com/internal group=!bluecoat_employees deny
url=www.bluecoat.com/proteus group=!bluecoat_development deny
```

The following example shows a section guard implementation for the `[url.domain] group=sales`.

```
<proxy>
[url.domain] group=sales
```

```
abc.com/sports deny
nbc.com/athletics deny
[rule]
url.regex="sports|atheltics/" access_server(no)
url.regex=\.mail.\" deny
[url]
www.bluecoat.com/internal groups=!bluecoat_employees deny
www.bluecoat.com/proteus groups=!bluecoat_development deny
```

Policy only evaluates the section if the user is a part of `group=sales`. If the user is not, policy skips the section and moves on to the next.

The performance advantage of using the `[url]`, `[url.domain]`, and `[server_url.domain]` sections is measurable when the number of URLs being tested reaches roughly 100. Certainly for lists of several hundred or thousands of URLs, the performance advantage is significant.

When no explicit section is specified, the layer rules are assumed to be in a `[rule]` section. For example:

```
<Proxy>
[rule]
url.domain=abc.com/sports deny
url.domain=nbc.com/athletics deny
url.regex=”sports|athletics/; access_server(no)
url.regex=”\.mail.\” deny
url=www.bluecoat.com/internal group=!bluecoat_employees deny
url=www.bluecoat.com/proteus group=!bluecoat_development deny
```

Local Database: Advanced Policy Optimization

The local database is another method of applying policy utilizing CPL.

The VPM and CPL provide similar performances if the URL lists are roughly less than 100,000 entries. While both the VPM and CPL use a hashed lookup approach, the local database is better for scaling larger URL lists such as excessively large blacklists and whitelists.

The category lookup mode sequence is:

- Policy
- Local Database
- IWF
- BCWF
- 3rd part database (if used)

The Local database category has two lookup options:

- **Always**–Always is the default, which means that the database is always consulted for category information. For example, if a URL is categorized under more than one category in different databases, the policy is checked against each category listed.
- **Uncategorized**–Specifies that a database lookup is skipped if the URL match is found in policy.

Using the Local Database – Best Practices

Blue Coat recommends using the local database if you have large whitelists and blacklists or other URL lists.

To use the local database you must enable the feature: **Configuration > Content Filtering > General > Enable**. The local database can then be accessed from the Configuration tab under Content Filtering.

Set up the local database with either FTP or HTTP. Authentication is optional.

The local database is kept and maintained off-appliance on either a web or FTP server, allowing users to maintain it without needing administrative access to the proxy. This can be beneficial in some deployments.

Advanced Policy

The following topics discuss advanced policy techniques to optimize performance.

- "Policy Macros" below–Converting it into a hierarchical policy.
- "Improve Policy Performance with CPU Monitoring" on the next page–Confirming you adhere to the CPU monitoring guidelines for your ProxySG appliance.
- "ICAP Trigger–delete_on_abandonment" on the next page–Avoiding processing of canceled connections by employing the `delete_on_abandonment` tag.

Policy Macros

A policy macro is a sequence of policy layers that can be called by name from other layers. All layers in a policy macro must be of the same layer type, and must be declared on the first line of the definition.

Policy macros are beneficial for performance in large policies. Policy macros function based on hierarchy and allow you to define policy definitions. Because of this, policy macro enables logical branching of policy, resulting in a smaller number of rules evaluated. For example, imagine evaluation descending through a decision tree, where policy evaluates a smaller number of rules at each branch of the tree.

The CPL syntax of a policy macro is as follows:

```
Define LayerType policy Macro
Layer1
Layer2
...
End
```

Consider the following example with the policy macro proxy:

```
Define proxy policy WebAccessPolicy
<proxy>
 DENY hour=9..7 category=NotBusinessRelated
 DENY category=IllegalOrOffensive
End
```

A policy macro call (`policy.MacroName`) is similar to a CPL guard setting: the rule or layer is only evaluated if all the conditions on the guard settings are true. When a macro call is evaluated, all of the layers in the corresponding policy definition are evaluated, setting the configured action. A policy macro that sets no properties has no effect when evaluated (Ok action).

When a rule matches during policy evaluation, all of the configured actions and macro calls in that rule are evaluated from left to right, with later rules overriding earlier property settings. This means that all property setting before a macro call act as defaults, and all property settings after the macro call act as overrides.

A policy macro definition can contain calls to other policy macros. However, reclusive calls and circular call chains are not allowed.

A policy definition cannot contain other definitions.

Improve Policy Performance with CPU Monitoring

If not optimized, policy can be heavily CPU intensive. The higher the CPU, the less efficient policy is. Blue Coat proxies work best with a lower CPU, roughly around 10%-15% CPU. If your policy is running above the ProxySG appliance's specifications, say around 50% CPU, it will not adhere to the sizing guidelines.

Blue Coat recommends optimizing and auditing your policy every so often to adhere to the appliance's threshold. As you optimize and audit policy, CPU monitoring allows you to review CPU utilization and review the functions that are using the majority of CPU. Blue Coat recommends enabling CPU monitoring whenever you want to review the percentage of CPU being used by specific features. For example, if you notice that compression/decompression is consuming most of the CPU, you can change your policy to compress/decompress more selectively.

To view and configure CPU monitoring:

1. Select **Statistics > Advanced**.
2. Click the **Diagnostics** link. A list of links to Diagnostics URLs displays.
3. To enable CPU monitoring, click the **Start the CPU Monitor** link; to disable it, click the **Stop the CPU Monitor** link.
4. To view CPU monitoring statistics, click the **CPU Monitor statistics** link. You can also click this link from either of the windows described in Step 3.

You can configure the interval at which CPU monitoring Statistics refresh in the browser. Enter the CLI commands:

```
#(config diagnostics) cpu-monitor interval seconds
```

The total percentages displayed on the CPU Monitor Statistics page do not always total 100% because the display only shows groups that are using 1% or more of the CPU processing cycles.

ICAP Trigger—delete_on_abandonment

One of the most important triggers available to users through ICAP is `delete_on_abandonment`. This trigger allows policy to terminate abandoned server requests.

For example, a user begins a web request and the file is sent for AV scanning through ICAP. If it is a medium to large file and taking a while to complete scanning, users either stay on the page and wait or abandon the page and start anew. The user's page abandonment does not close the request. Because the ProxySG appliance is unaware that the connection should be torn down, it continues to consume resources that could be used elsewhere.

ICAP monitoring on the ProxySG appliance requires SGOS 5.4 or higher.

The `delete_on_abandonment` trigger is enabled by default when bandwidth gain mode is enabled. However, you can also enable the trigger irrespective of whether bandwidth gain is enabled or disabled. In addition to enabling the trigger, bandwidth gain mode increases bandwidth gain resulting in increased page response time and object freshness.

Trigger Application – Best Practices

Blue Coat recommends using ICAP's `delete_on_abanonment` trigger, otherwise performance problems can occur. Not using the tag allows connection requests to que up, causing performance delays.

The following policy example prevents queuing of duplicate requests for a known aggressive client.

```
<cache>
request.header.User-Agent-"Winamp" delete_on_abandonment(yes)
```

You can enable `delete_on_abandonment` for all clients, using the following code:

```
<proxy>
delete_on_abandonment(yes)
```

`delete_on_abandonment` does not work when patience pages are enabled; it can be used only with data trickling or with no ICAP feedback.

Alternatively, enable `delete_on_abandonment` by enabling bandwidth gain mode.

To enable the bandwidth gain mode:

1. Go to **Proxy settings > HTTP > Acceleration Profile**.
2. Select **Enable Bandwidth Gain Mode**.
3. Click **Apply**.

If you are enabling ICAP's trigger after establishing your policy, Blue Coat recommends reducing the amount of queued connection requests. To do this, review the Critical Threshold for ICAP Queued in **Maintenance > Health Monitoring > General**. Select **Edit** and make changes to the Critical Threshold. Then, verify if the current connections are less than the max connections.

To determine how close policy is to running the max connections, navigate to **Management Console > Statistics > ICAP**. Select your preferred service (**Service** or **Service Group**) and **Duration**. The ProxySG appliance displays the current connections in addition to other ICAP statistics.

Troubleshoot Policy

Prevent troubleshooting issues in the ProxySG appliance by the following:

- Access Logging–Track policy function with access logging.
- Object Errors–Avoid the use of server response and DNS-based objects.
- Policy Trace–Debug policy with the policy trace feature.
- Timing Errors–Distinguish possible timing errors and prevent them.

Track Policy with Access Logging

You can use policy to track user requests in a ProxySG appliance access log.

Access logging allows you to track web usage for the entire network or specific information on user or department usage patterns. These logs and reports can be made available in real-time or on a scheduled basis. Blue Coat recommends you create and use access logs to trace policy requests in ProxySG appliance traffic.

When using access logs Blue Coat recommends the following:

- Create log facilities as needed
  - Set a schedule
  - Configure log formats
  - Set passwords
  - Configure log upload time
- Encrypt and decrypt access logs as needed
- Refer to Blue Coat Reporter for log analysis as needed

You can define policy that outputs to a specific log, allowing them to filter what ends up in a new, unique log by detailing policy elements.

For example, your company requires a log indicating which users attempt to access blocked URL categories on your network. Rather than filter the generated main log, you can create a rule in policy that lists *bad categories* in the Destination field and *modify access logging* as the Action. This allows you to designate which access log to write those results to.

See the following Knowledge Base articles for supplementary information on access logs:

- Creating or modifying a log file on the ProxySG
- How to delete access logs on the ProxySG
- How to send access logs to more than one server

Prevent Server Response and DNS-Based Object Errors

Server response and DNS-based objects run the risk of reducing policy performance. Both objects force policy to wait for a response before processing, which can result in a failed request or unexpected behavior and poor policy performance.

Server response-based object–Policy must wait for a response from the rule request before processing, reducing policy efficiency. If a rule request is blocked by an upstream firewall, IPS, or other problems between the proxy and the request site, policy results in a `service unavailable` response instead of an allowed or denied transaction because the request could not complete.

DNS-based object–Policy must wait for a response from DNS requests. If DNS is slow to respond, it results in slow policy performance. If the object requires DNS, such as a hostname, and there isn't a corresponding RDNS entry, the request fails.

Server response and DNS-based objects require policy to wait for a response before finishing the evaluation. The slow response time reduces policy performance and increases the evaluation time.

Server Response and DNS-Based Objects – Best Practices

To avoid performance issues when using response-based rules, Blue Coat recommends the following:

- Use layers and layer guards to reduce traffic checking on response-based rules.
- Minimize the use of server response and DNS-based objects.
- Restrict DNS lookups except for users and groups as needed

These actions will improve policy performance and reduce the risk of unexpected behavior.

Distinguish Transaction and Guard Timing Errors

Timing, which is the sequence of evaluations, occurs as the appliance processes a transaction; for example, the administrator transaction sequence evaluates policy in two stages: before and after the authentication challenge.

Policy timing affects transactions and guards, and it can potentially cause errors.

Policy timing errors, such as transaction and guard timing errors, occur when a rule in policy seeks to match elements of a user request that are either not yet available by the time the specific policy is being evaluated, or are no longer available.

For transaction timing, errors might be linked to:

- The type of transaction
- The timing of a transaction
- The conditions of a transaction
- The layers the transaction occurs in
- The rules of the transaction

Consider the following transaction timing error:

Your users are unable to access a website that uses HTTPS because the site presents a certificate that fails the appliance's certificate validation. To allow users to be able to access the site, you create the following policy, which has an action in the rule that tells the ProxySG appliance not to validate the site's certificate:

```
Web Access/ <proxy> layer
Source: any
Destination: server.certificate.subject.exact="example.com"
Action: server.certificate.validate(no)
```

The policy results in the following error:

```
Error: Late condition guards early action
Condition 'server.certificate.subject.exact=example' vpm-cpl:28
Action 'server.certificate.validate(no)" vpm-cpl:28
```

This error occurs because the destination of the policy is looking at the server certificate for a match for the appliance's certificate validation process, but the action of the policy tells the appliance to not look at the certificate. In this case, change the destination for the rule to `url.host=example.com` to resolve the error in policy.

For guard timing, *late guard early* timing errors might occur. Late guard early timing errors are caused when a rule within a policy specifies that a condition needs to be tested early in the transaction sequence, but the condition can only be tested later in the sequence; for example, a rule states:

```
If the user in in group efg, require authentication
```

The condition `is If the user is in group efg` and it can only be tested after authentication, not before authentication as the rule states. This rule results in a late guard early timing error.

Late guard early timing errors can occur within a layer and across guard expressions. When a trigger in a layer or guard expressions cannot yet be evaluated, policy also has to postpone evaluating all the following rules in that layer; thus, a rule inherits the earliest evaluation-point timing of the latest rule above it in the layer.

For example, the following rule would result in a timing conflict error:

```
group=abc authnticate(MyRealm)
Error: Late guard early action:'authenticate(MyRealm)'
```

The following layer would result in a similar error:

```
<proxy>
 group=abc deny
 authenticate(MyRealm)
Error: Late condition 'group=abc' guard early action:'authenticate(MyRealm)'
```

Policy Timing Error Prevention– Best Practices

To avoid transaction timing errors, Blue Coat recommends you understand and consider the following:

- **Transaction Type**
  - Know the transaction type when troubleshooting transaction errors; for example, is the transaction an <admin>,<proxy>,<cache>, or other type of transaction? Only a subset of layer types, conditions, properties, and actions is appropriate for each of the transaction types.
  - Know whether specific users are associated with a transaction, and consider what content transactions evaluate when troubleshooting transaction errors; for example, no specific user is associated with cache transactions; therefore, cache transactions evaluate content-related policy, but not user-related policy.
- **Transaction Timing**

  Using the Policy Trace Feature, define:
  - When the timing for a given transaction is, such as when the appliance evaluated a policy.
  - How timing affects transactions.
  - When and which portions of the transaction information become available; for example, a <forwarding> transaction is created when the ProxySG appliance needs to evaluate forwarding policy before the appliance accesses a remote host.
  - What the specific requirements are for each decisions that must be made for each protocol.
- **Transaction Conditions**

  Know the conditions a type of transaction supports; for example, <cache> transactions do not support conditions such as client.address= and group=, or authenticate() property.
- **Transactions In Layers**

  Consider how policy in one layer affects transactions in other layer; for example, a cache transaction evaluates policy in <cache> and <forward> layers, but the <forward> layers are only evaluated if an origin server must be contacted to complete the transaction.
- **Transaction Rules**

  Consider what happens when a conflict between a trigger and the property timing exists within a policy rule, for example:

  ```
  If the user is in groups xyz, require authentication
  ```

  The preceding rule is incorrect because group membership can only be determined after authentication.

Other Topics

To avoid layer and guard timing errors, Blue Coat recommends you be aware of the following:

- Layer and rule evaluation order. See "Layer Evaluation Order" on page 17 and " Rule Evaluation" on page 18.
- Guard placement–See "Layer Guards " on page 16.

When you understand these concepts, the potential for guard errors is reduced.

Debug Policy Using Policy Trace

Policy tracing provides debugging information on policy transactions. This can be helpful even when policy is not the issue. Tracing allows you to examine how the appliance's policy is applied to a particular request and how long the request takes.

CPL provides the following trace related properties:

- `trace.request()` – Enables tracing and includes a description of the transaction being processed in the trace. No trace output is generated if this is set to `no`.
- `trace.destination()` – Directs the trace output to a user-named log.

Log Policy Using `trace.request()`

Request tracing logs a summary of information about the transaction:

- Request parameters
- Property settings
- Action effects

This property uses the following syntax:

`trace.request(yes|no)`

Where:

- `yes` – Enables tracing
- `no` – Disables tracing

For example, the following CPL enables full tracing information for all transactions:

```
<cache>
 trace.request(yes)
```

Log Policy Using `trace.destination()`

Use the `trace.destination()` property to configure where the ProxySG appliance saves trace information. You can set and reset the trace destination as needed. It takes effect (and the trace is actually written) only when the ProxySG appliance has finished processing the request and any associated response. Trace output is saved to an object that is accessible using a console URL in the following form:

```
https://appliance_IP_address:8081/Policy/Trace/path
```

Where `path` is a filename, directory path, or both. If you specify a directory only, the default trace file name is used.

For example, the following two destinations are configured for policy tracing information:

```
<proxy>
client.address=10.25.0.0/16 trace.destination(internal_trace.html)
client.address=10.0.0.0/8 trace.destination(external_trace.html)
```

The console URLs for retrieving the trace information are:

```
https://<appliance_IP_address>:8081/Policy/Trace/internal_trace.html
https://<appliance_IP_address>:8081/Policy/Trace/external_trace.html
```

You can view policy statistics through the Management Console: **Statistics > Advanced > Policy > List of Policy URLs**.

Policy Trace Timing Logs

When policy trace is enabled, the request process is logged. This log informs you on how much time the ProxySG appliance spends processing a request.

The timing in a log starts at 0, and the time that follows elapses in milliseconds from the beginning of the request. Any timing under 1ms is rounded down to 0ms.

The following shows a trace log's reported timings from an SSL transaction:

```
total categorization time: 0
 static categorization time: 0
```

In this example, the zeros indicate that URL categorization did not occur because categories are determined by server certificate hostname in SSL transactions.

```
Transaction timing: total-transaction-time 478ms
   Checkpoint timing:
     new-connection: start 1 elapsed 0 ms
     client-in: start 1 elapsed 0 ms
     server-out: start 1 elapsed 0 ms
     server-in: start 250 elapsed 0 ms
     stop-transaction: start 478 elapsed 0 ms
     total policy evaluation time: 0 ms
  ssl server hello complete: 246
  url_categorization complete time: 479
  ssl_server started intercept: 250
  server connection: start 1
     DNS lookup: start 1 elapsed 0 ms
  server connection: connected 83

Total time added: 0 ms
Total latency to first byte: 82 ms
    request latency: 0 ms
  OCS connect time: 82 ms
  Response latency (first byte): 0 ms
    Response latency (last byte): 0 ms
```

Policy Trace Application – Best Practices

Blue Coat recommends using policy tracing for troubleshooting only. Tracing is best used temporarily. If tracing is enabled in a production setting, the ProxySG appliance performance degrades. After you complete troubleshooting, be sure to remove policy tracing. To learn more about policy tracing, refer to the Knowledge Base articles Understanding Policy Trace Checkpoint Timing and How to use Policy Trace to Debug Access Issues.

To configure tracing in a policy file:

- Enable tracing by using several policy language properties.
- Set the verbosity level.
- Specify the path for output.

- Use appropriate conditions to guard the tracing rules; you can be specific about the request for which you gather tracing information.

To configure policy trace in ver 6.6, you only need to enable and disable the policy trace feature.

The following is recommended when using policy trace:

- Use the `trace.request()` property to enable request tracing.
- Use the `trace.destination()` property to configure where the appliance saves trace information.
- Use policy timing to determine how much time it takes for the appliance to serve a request.