ChatGPT

Qubes OS Policy System Tutorial (2025 Edition)

Introduction to Qubes OS Policies

Qubes OS uses strong isolation: each “qube” (VM) is a separate domain for security. Yet qubes often need to interact – for example, copying files between qubes, opening links in a browser qube, sharing clipboard text, or attaching a USB device. The Qubes policy system governs **which interactions are allowed, denied, or require user approval**, enforcing security boundaries between qubes [1]. This policy system is implemented via Qubes’ qrexec framework (remote procedure calls between qubes) and was overhauled in Qubes 4.1 to a *new unified format* that is used in Qubes 4.2 (and beyond). We will focus **only on the new policy system** introduced in Qubes 4.1+, and **not** the old per-service files from Qubes 4.0. All examples and guidance here reflect how Qubes policies work as of 2025.

How Qubes RPC policies work: When one qube requests an action involving another, qrexec checks the policy rules in Dom0 (the administrative domain) to decide if that action is allowed. Every such cross-qube action is associated with a **service name** (like `qubes.Filecopy` for file transfers or `qubes.OpenURL` for opening a URL in another qube). The policy rules determine, based on service and the source/destination qubes, whether to `allow` the action, `deny` it, or `ask` the user for consent [1] [2]. By default, Qubes ships with a set of *restrictive* or *prompting* rules for safety. For instance, copying a file between qubes by default will prompt you (the `ask` action) for approval [3]. Some particularly risky actions are flat-out denied by default – e.g. a qube trying to directly open a raw network socket into another qube (`qubes.ConnectTCP`) is denied system-wide [4]. This ensures that unless you explicitly change the rules, qubes cannot abuse powerful services without your knowledge.

“First match wins” principle: The policy engine goes through the rules in order and **applies the first rule that matches** the requested action [5] [6]. If no rule matches, the action is denied by default. This means rule *ordering* is very important. Qubes handles ordering both by the sequence of rules in a file and by the file naming (which we’ll explain shortly). Essentially, more specific or higher-priority rules should come before general catch-all rules. We will see examples of this in practice (e.g. allowing certain qubes to interact freely while forbidding others).

Policy Files and Syntax (New Unified Format)

In Qubes 4.1+ all RPC policies are defined in files under the directory `/etc/qubes/policy.d/` in Dom0 [7] [8]. Unlike older versions (which had one file per service), the new system treats the entire policy set as one unified table – you can define rules for any service in any policy file. By convention, Qubes provides default policy files numbered high (e.g. `90-default.policy` and `90-admin-default.policy`), and you as the user can create your own policy file(s) with a lower number (like `30-user.policy`) to override defaults [9] [10]. All files with the `.policy` extension in that directory are merged together; **rules in files with a lower number take precedence** over those in higher-numbered files [11]. This means you should **never edit the `90-default.policy` file directly** – instead, put your custom rules in a new file like `30-`

`user.policy` (or `20-something.policy`). Qubes will evaluate your file first, and any rule you define there will "win" over the generic defaults [9].

Policy rule format: Each rule is one line with **five columns**, separated by whitespace:

```
service-name  +argument   source-qube   target-qube   action   [options]
```

For example, a rule might look like:

```
qubes.Filecopy    *    work    personal    ask
```

This would match any `qubes.Filecopy` request (file copy) from the qube named "work" to the qube "personal" and require the user to approve (`ask`). Let's break down the columns:

- **Service name:** This corresponds to the RPC service being invoked (e.g. `qubes.Filecopy`, `qubes.OpenURL`, `qubes.StartApp`, etc.). You can use a wildcard `*` to match any service, or a prefix to group services. For instance, an `*` in the service field means "any service" [12]. (Be cautious with broad wildcards; usually you'll specify the service or use groups of services via includes/tags, explained later.)
- **Argument:** Some services take an argument (for example, `qubes.StartApp+Firefox` passes the application name "Firefox" as an argument, or a device attach service might pass a device ID as argument). In the policy, an argument is denoted by a `+`. You can put an `*` here to match any argument or an empty argument [12]. We'll cover below how arguments enable fine-grained rules. If a service doesn't use arguments, you can just put `*` in this column.
- **Source qube:** The name of the qube *requesting* the action (the caller). This can be an explicit qube name (e.g. `work`), or a special keyword:
- `@anyvm` (or the older `$anyvm`) means "any qube" [13] (excluding dom0, since dom0 isn't allowed to initiate RPC requests in Qubes' model).
- `@dispvm` means a DisposableVM (a one-time throwaway qube) – more precisely, it matches a qube that was created as a disposable for this request [14]. (There's also `@dispvm:NAME` to refer to a disposable based on a specific template; more on that later.)
- `@adminvm` refers to dom0 (the admin domain) – though normally dom0 initiating a call is always allowed by Qubes, so you don't often see it in source field.
- You can also match qubes by *type*, using `@type:TYPE` (for example `@type:TemplateVM` to match all Template VMs, or `@type:AppVM` for regular app qubes) [15].
- Or by *tag*, using `@tag:TAG` (match any qube that has a given tag) [15]. Tags are user-defined labels you can assign to qubes (e.g. tag several VMs as "work" or "untrusted"). Using tags is very powerful for policy (as we'll see in examples).
- **Target qube:** The name of the qube that is the *target* of the action. This might be the qube where a file is to be copied **to**, where a program should be launched, which qube should handle a request, etc. The target can likewise be a specific name or use the special placeholders above (`@anyvm`, `@dispvm`, tags, types, etc.). There is also `@default` which is used when the source did not explicitly specify a target [16]. For example, if you run `qvm-copy somefile` in a source qube

without naming a destination, the policy will look for a rule with target `@default` to decide how to handle that situation (usually it will prompt you to choose a VM).

- **Action:** This is what to do if the rule matches. It can be:
- `allow` – permit the action with no prompt.
- `deny` – block the action. (The calling application will usually get a "Request refused" error.)
- `ask` – open a confirmation dialog asking the user to approve or reject the action in real time.
- **Options (optional):** In brackets at the end of the line, you can specify additional parameters. Common options include:
- `target=<vm>` – *redirect* the action to a different target qube than the one requested. For example, the UpdatesProxy policy uses `target=sys-net` to transparently send update checks to the "sys-net" qube, no matter which qube initiated them (unless overridden) [17] . You might use this to enforce a certain proxy or firewall VM for network services.
- `default_target=<vm>` – when using `ask`, this sets a default pre-selected VM in the confirmation dialog [18] . For instance, if a source qube asks to open a file and you want the prompt to default to a particular "viewer" qube, you could use `default_target=viewer-vm` on the `ask` rule.
- `user=<username>` – to specify which user account the service should run as on the target side. By default, most services run as a normal user. Some admin services or special cases might need root. For example, the policy comments note that if you ever allow the `qubes.VMRootShell` service (which is normally denied), you should include `user=root` so that it runs with root privileges as intended [19] . Generally, you won't use this option unless dealing with low-level admin calls.
- `notify=yes` or `notify=no` – controls desktop notifications. By default, Qubes will show you a notification if an action is denied (unless it was an action you explicitly clicked "deny" in a prompt) [20] . You can set `notify=yes` on an `allow` rule to get a pop-up whenever that rule allows something [21] (useful for auditing). Or set `notify=no` on a deny rule if you want to suppress the "denied" notifications for that particular case.

  Note: Qubes policy syntax replaced the old `$` prefix with `@` in Qubes 4.1. For example `$anyvm` is now `@anyvm`, `$dispvm` is `@dispvm`. The old `$` forms still work for now (for backward compatibility), but you'll see `@` in all modern rules [22] . It's best to use the `@` syntax going forward.

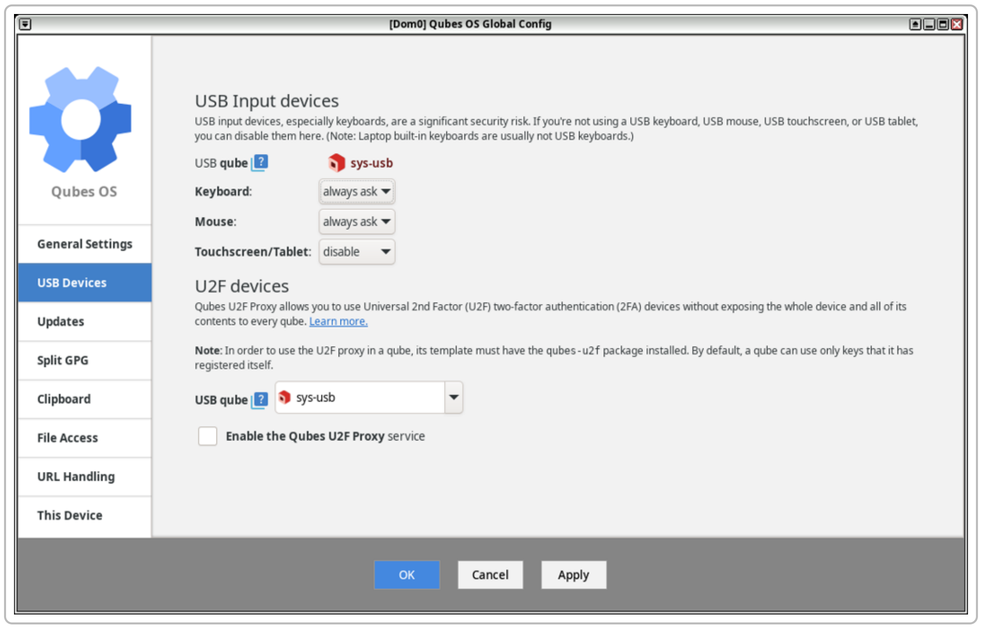
How rules are evaluated: When a qube calls an RPC service, the policy daemon in dom0 scans the combined policy (all those files in `/etc/qubes/policy.d`) from top to bottom (honoring the numeric file order) and stops at the first matching rule [6] . "Matching" means the service name (and argument, if specified) matches the rule's service column, the source qube matches the source column, and the target matches the target column. If that rule says `deny`, the action is blocked; if `allow`, it proceeds; if `ask`, a prompt is shown to you. If no rule matches at all, the default is to deny the request [6] – Qubes prefers to *fail closed* for safety.

Because the first match wins, you should order rules from most-specific to least-specific. Often you'll have a "catch-all" rule at the bottom like `* * @anyvm @anyvm deny` or `ask` to cover anything not covered above. The Qubes default policy file has such catch-all entries (for example, the very last rule in `90-default.policy` is typically an `@anyvm @anyvm ask` for many services, meaning "if nothing earlier matched, ask the user") [23] . When you add your own rules in a lower-numbered file, you can override the defaults by matching earlier.

Let's illustrate the matching with a quick example: Suppose Qubes receives a request "Qube A wants to copy a file to Qube B". It will look for a rule with service `qubes.Filecopy` (or `*` wildcard), source matching A, and target matching B or `@anyvm`. If you have a rule specifically for A→B, that will match first. If not, maybe you have a broader rule like `@tag:work @anyvm deny` (meaning "no Work-tagged VM can copy to any VM") – if A is tagged "work", that catches it. If no tag rules, maybe the default `@anyvm @anyvm ask` catches it, resulting in a prompt. Once a rule is found and applied, it stops; it **does not** continue scanning lower rules [5]. This is why ordering and specificity matter so much.

Managing Policies: GUI Tools vs Text Editor

Qubes OS (especially version 4.2) provides some user-friendly GUI tools to manage policies, which is great for newcomers. Under the hood, these tools are editing the same policy files, but they present common settings in a more visual way.



USB Input devices

USB input devices, especially keyboards, are a significant security risk. If you're not using a USB keyboard, USB mouse, USB touchscreen, or USB tablet, you can disable them here. (Note: Laptop built-in keyboards are usually not USB keyboards.)

USB **qube**: sys-usb

Keyboard: always ask

Mouse: always ask

Touchscreen/Tablet: disable

U2F devices

Qubes U2F Proxy allows you to use Universal 2nd Factor (U2F) two-factor authentication (2FA) devices without exposing the whole device and all of its contents to every qube. Learn more.

Note: In order to use the U2F proxy in a qube, its template must have the `qubes-u2f` package installed. By default, a qube can use only keys that it has registered itself.

USB qube: sys-usb

☐ **Enable the Qubes U2F Proxy** service

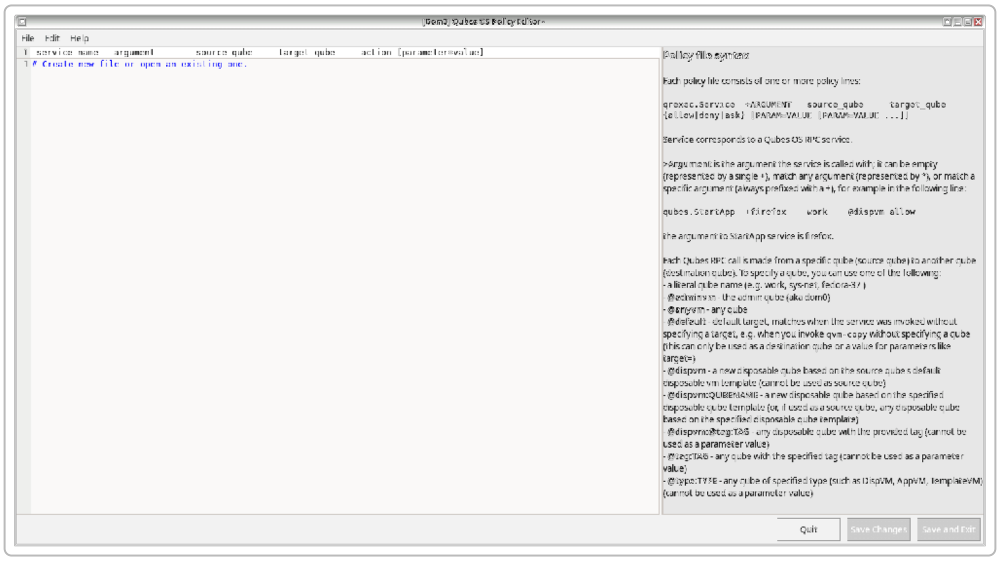
Screenshot: The Qubes Global Config tool in Qubes 4.2, showing the "USB Devices" section. Note the sidebar with various sections ("General Settings", "USB Devices", "Updates", "Split GPG", "Clipboard", "File Access", "URL Handling", etc.). These GUI screens allow you to configure corresponding policy rules or system settings without directly editing text files.

Qubes Global Config: This is a central GUI accessible from dom0's start menu (or via the Qubes toolbar gear icon). Go to **Qubes Tools → Qubes Global Config** [24]. In Qubes 4.2, the Global Config window has multiple tabs or sections, each relating to certain policies or settings: - **USB Devices:** Control how USB devices are assigned. For example, you can choose what Qube is used as the "USB qube" and whether keyboards/mice are always asked, allowed, or disabled (since USB input devices can be a security risk if compromised). This section essentially tweaks policies for the USB device attachment services or uses Qubes-specific settings for USB passthrough. - **Updates:** Configure how templates get updates (e.g. you could set all updates to go through a specific UpdateVM or allow direct). - **Split GPG:** Set up which qube will serve as your GPG keyholder (Vault) and which qubes are allowed to use it – this in turn adds the appropriate `qubes.Gpg` policy rules for you. - **Clipboard:** Manage the global clipboard behavior (how copy/paste between qubes is handled). By default Qubes uses an inter-qube clipboard with manual steps, but here you might impose additional restrictions if desired (for example, disallowing clipboard transfer from less trusted qubes to more trusted ones). - **File Access:** Controls the "Open in another qube" functionality. For instance, when you right-click a file in one qube and choose "View in dispVM" or "Open in qutebrowser (another qube)", these actions are governed by `qubes.OpenInVM` policy – the GUI lets you set default destinations or restrictions. - **URL Handling:** Similar to File Access, but for opening URLs in other qubes (governed by `qubes.OpenURL` policy). - **And more:** General Settings (some global defaults), "This Device" (hardware-related settings), etc.

These UI sections provide drop-downs and checkboxes for common policy tweaks. For example, in the **Clipboard** tab you might see options like "Allow copy from any qube to any qube (default)" or the ability to make clipboard one-directional for certain qubes. In **File Access/URL**, you might choose a default disposable VM for untrusted file viewing, etc. When you adjust these and hit Apply, the tool is editing the policy files in the background. In fact, a great tip for learning is: make a change in Global Config, then click a "view policy file" or open the corresponding policy to see what rule was added [25]. For instance, toggling an option in the Split GPG tab and then viewing the policy will show you the new `qubes.Gpg` rule that was inserted. This helps you connect the high-level setting to actual policy syntax [25].

Importantly, the Global Config and the new **Policy Editor** perform syntax validation when saving changes [24]. This means they won't let you save a rule with a typo or wrong format – a lifesaver for avoiding broken policy files that could block important functionality. If you're new, using these tools can simplify policy editing and reduce errors (you can always refine the text files as you get comfortable).

Qubes Policy Editor (GUI): Qubes 4.2 introduced a dedicated Policy Editor application (accessible via Qubes Global Config or the applications menu) for directly editing policy files in a controlled environment. This editor is essentially a text editor tailored for Qubes policies: it shows the rules in a table, and often provides a side-by-side reference of syntax and common variables (so you don't have to memorize every keyword). It has a minimal interface with all necessary options and does on-the-fly validation of your input [26] [24]. You can create a new policy file or open an existing one, edit rules, then save. It's a nice middle-ground between hand-editing with `nano` / `vim` and using only checkboxes – you get full control but with some guidance.



```
service name    argument        source qube     target qube     action [parameter=value]
# Create new file or open an existing one.
```

Policy file syntax

Each policy file consists of one or more policy lines:

```
qrexec.Service  +ARGUMENT   source_qube     target_qube
{allow|deny|ask} [PARAM=VALUE [PARAM=VALUE ...]]
```

Service corresponds to a Qubes OS RPC service.

+Argument is the argument the service is called with; it can be empty (represented by a single +), match any argument (represented by *), or match a specific argument (always prefixed with a +), for example in the following line:

```
qubes.StartApp  +firefox    work    @dispvm allow
```

the argument to StartApp service is firefox.

Each Qubes RPC call is made from a specific qube (source qube) to another qube (destination qube). To specify a qube, you can use one of the following:
- a literal qube name (e.g. work, sys-net, fedora-37)
- @adminvm - the admin qube (aka dom0)
- @anyvm - any qube
- @default - default target, matches when the service was invoked without specifying a target, e.g. when you invoke qvm-copy without specifying a qube (this can only be used as a destination qube or a value for parameters like target=)
- @dispvm - a new disposable qube based on the source qube's default disposable vm template (cannot be used as source qube)
- @dispvm:QUBENAME - a new disposable qube based on the specified disposable qube template (or, if used as a source qube, any disposable qube based on the specified disposable qube template)
- @dispvm:@tag:TAG - any disposable qube with the provided tag (cannot be used as a parameter value)
- @tag:TAG - any qube with the specified tag (cannot be used as a parameter value)
- @type:TYPE - any qube of specified type (such as DispVM, AppVM, TemplateVM) (cannot be used as a parameter value)

Screenshot: The Qubes OS Policy Editor tool. On the left is the editable policy (here it's blank with a hint "Create new file or open an existing one"), with columns for service, argument, source, target, action, etc. On the right is a helpful reference of policy file syntax, listing keywords like @anyvm, @dispvm, tag usage, etc., to assist the user.

To use the Policy Editor, open **Qubes Global Config → (menu) Open Policy Editor**, or find "Qubes Policy Editor" in the dom0 applications menu. When editing, remember to **save** changes and consider that changes take effect immediately once the policy daemon loads them (the policy daemon auto-reloads when files change). The Policy Editor will prevent saving if there's a syntax mistake (for example, wrong number of columns), which is very helpful.

Manual editing via text files: The GUI tools are convenient, but you can always directly edit or create policy files in a dom0 terminal or text editor. Your custom files belong in `/etc/qubes/policy.d/` and should be named with a number prefix lower than 90. A common practice is to have one file named `30-user.policy` for all your tweaks [9], but you can split them (e.g. `20-work.policy` for work VM rules, `25-vault.policy` for vault rules, etc.) – the ordering is what matters. Use a text editor in dom0 (like `nano` or `vim`) to edit these files. After saving, the changes apply immediately (the policy daemon logs any syntax errors to `journalctl -u qubes-qrexec-policy-daemon`, which can help debug if something isn't working). If you make a mistake that locks things down (e.g. you accidentally denied something critical), you can always open the file again in dom0 and fix it – Qubes doesn't require a reboot for policy changes, and the worst-case if you deny all qrexec is that some Qubes features won't work until fixed.

When editing manually, follow these best practices:
- **Never remove the default files.** It's fine to override specific rules in your own file, but don't delete `90-default.policy` or `90-admin-default.policy`. They contain many essential rules (for example, rules that allow the Update Proxy or certain inter-qube notifications). If you don't want some default behavior, just override it by adding a more specific rule earlier. You can comment out lines in your own files with `#` if needed (in `90-default.policy` all lines are active unless commented by `#`; but again, edit a custom file rather than modifying 90-default in place).
- **Use specific matches for what you intend to allow.** Broad `@anyvm allow` rules can punch big holes in isolation if not careful. Always try to scope rules by source, target, service, or tags so they apply only where needed. We'll see examples.
- **Test after changes:** If you add a new rule, test the intended action to confirm it behaves (e.g. if you added a rule to allow file copy from VM A to B without ask, actually try copying a file to see if it now goes through quietly). If something fails, check the logs or notifications – Qubes will usually notify "Denied: <service> from X to Y" which tells you a rule is missing or not matching what you thought.

One more thing: in addition to normal "RPC" policies, there are also **Admin API policies** (for services that start with `admin.*`). These govern administrative actions (like creating qubes, changing settings, etc.) that can be invoked via qrexec. By default, **no qube is allowed to use admin services** except dom0, *unless* you set up a dedicated AdminVM or GUI domain. Admin policy rules live in `90-admin-default.policy` and use include files like `admin-local-rwx` etc. [27]. For a beginner, you typically won't touch these unless you are configuring something advanced (like a separate GUI VM that needs permission to control other qubes). Just be aware that if you ever see policy lines involving `admin.vm.List` or `admin.vm.Volume` etc., those are part of the Admin API. They should generally remain "deny" for all external qubes by default (for security, since they can control your system) [28]. We'll not dive deep into admin policies here, but know that they exist and follow the same format (with the requirement that any allow/ask for admin services must include `target=dom0` because those services run in dom0) [29] .

Common Policy Scenarios and Examples

Now let's walk through real-world examples of policy configurations. These illustrate how you might tweak Qubes OS policies in practice and what other users in 2025 are doing to balance security and convenience. Each example will explain the *why*, *how*, and use of the new policy syntax.

Example 1: Isolating a Vault – Deny All Incoming Requests

Many Qubes users maintain a highly sensitive **Vault qube** (for things like password storage or keys) that should never be impacted by less-trusted qubes. A good policy is to **deny all RPC calls going *into* the Vault**. This means no other qube can, say, copy a file into the vault or request something from it. The vault will only ever send data out (and even that you might handle carefully). We can implement this with a simple rule:

```
*    *    @anyvm    vault    deny
```

This uses `*` wildcards for service and argument (meaning **any service**), source `@anyvm` (any qube), destination `vault`, and action `deny`. Placing this rule in your `30-user.policy` effectively erects a firewall around the vault. Now, if you try from another VM to copy a file to `vault` or send it a message, the first matching rule will be this deny [30] , and the action will be blocked (likely you'll see "Request refused" immediately with no prompt).

You would put this rule near the top of your policy file to ensure it matches before any more lenient default rule could allow something. Because we used wildcards for service, it covers *all* cross-qube actions. This might sound overly broad, but for a vault this is usually what you want – it should not accept *anything* from elsewhere. The vault qube can still initiate things that *outbound* from itself; for example, if you (in the vault) run `qvm-copy somefile` to another VM, that is a request *from* vault (source=vault, dest=other). Our rule only blocks source=any → dest=vault. So outbound is still possible (you could tighten that too if desired, but usually the main concern is inbound).

Real-world usage: By 2025, many Qubes users have a vault that's completely isolated. They might even tag the vault with something like `no-net` and have a blanket policy similar to the above using a tag (e.g. `* * @anyvm @tag:no-net deny` to protect all qubes that have the tag "no-net"). Tagging can be handy if you have multiple vault-like qubes. For a single qube, specifying its name as we did is straightforward.

(To apply such a rule via the GUI: There isn't a one-click option for "vault lockdown" in Global Config, but you could manually add it in the Policy Editor as shown above. Always double-check that denying all services to a qube won't break something you need – vaults usually have no network and no needed incoming services, but if yours uses, say, Split GPG, you might allow that one service from specific qubes.)

Example 2: Grouping Qubes by Tag – Trusted Intra-Group, Isolated from Others

A common setup is to have a group of qubes that can freely exchange data with each other, but not with the rest of the system. For instance, suppose you have several qubes related to work projects – let's call them `work-mail`, `work-docs`, `work-web` – and you want them to share files without prompting, yet you want to prevent work qubes from sending anything to non-work qubes (and vice versa, no external qube should send data into your work group without permission).

Qubes makes this easy using **tags** in policy. First, tag all those VMs with "work" (you can assign tags via Qube Settings or Qubes Manager: e.g. open each VM's settings and add tag `work`, or use `qvm-tags work-mail add work` in dom0). Now we can write rules leveraging the `@tag:work` selector:

```
qubes.Filecopy    *    @tag:work      @tag:work      allow
qubes.Filecopy    *    @tag:work      @anyvm         deny
qubes.Filecopy    *    @anyvm         @tag:work      deny
qubes.Filecopy    *    @anyvm         @anyvm         ask
```

Let's break this down (these four lines are adapted from an example in Qubes docs [5] [31] for the file copy service):

- The first rule says: any `qubes.Filecopy` request where the source has tag "work" and the destination has tag "work" is allowed (no prompts) [2]. This means any two work-qubes can copy files among each other freely. This makes working within that group convenient.
- The second rule: if the source is tag "work" and dest is *any other VM*, deny it [32]. So a work VM cannot send a file to a VM that is not tagged work – protecting you from accidentally exfiltrating work data to, say, your personal or untrusted VM.
- The third rule: source any and destination tag "work", deny [31]. This means no outside VM can send a file into your work VMs. (So your work data won't get tainted by something an untrusted VM tries to send; it also mitigates certain social engineering attacks – e.g. malware in an untrusted VM can't push a file to your work VM without you explicitly fetching it in some way.)
- The fourth rule: any to any ask [33]. This is basically the default fallback (Qubes default for Filecopy is ask between arbitrary qubes). We include it here to show completeness – it ensures that if none of the above specific rules matched, the user will be prompted. In practice, this line is probably already in the 90-default policy, so you might not need to add it yourself; but if you're writing a standalone policy file for file copying, you'd end with an ask or deny as a catch-all.

With these rules, *within* the work group, file transfers are seamless, but any transfer that crosses the boundary of that group is automatically blocked (no prompt, just denied) [32] [31]. You as the user can of course override by temporarily changing policy or copying via an intermediate step, but by default this tag-based policy enforces that data stays in its lane.

This tag approach is popular in 2025 because it scales well: if you add a new work VM tomorrow, just tag it "work" and it inherits the same rules – no need to write new per-VM policies. Users also use tags for other trust levels, e.g. an "untrusted" tag to restrict those VMs heavily, or a "dev" tag to allow certain dev VMs to communicate with each other on specific services. The example we gave is for the `qubes.Filecopy` service (copying files). You could replicate similar patterns for other services if needed (e.g. maybe allow `qubes.OpenURL` freely among work qubes but deny from work to others, etc., although that's a less common requirement than file exchange).

Example 3: Enforcing Disposable VM Usage (Open in Disposable by Default)

One of Qubes OS's powerful security features is using **Disposable VMs (DispVMs)** for opening risky content. For example, you might want any PDF or document from an email to open in a disposable VM (to mitigate malware), or any web link you click in a chat app to open in a disposable browser instance. Qubes already provides convenient options for this (you can right-click a file → "View in DispVM", or use the Qubes Menu to open a link in a disposable). The policy system determines what happens by default when such actions are invoked.

In fact, the Qubes default policy already leans toward disposables for *Open in VM* and *Open URL* actions. In `90-default.policy` you'll find rules like:

```
qubes.OpenInVM        *    @anyvm    @dispvm    allow
qubes.OpenInVM        *    @anyvm    @anyvm     ask
```

and similarly for `qubes.OpenURL`:

```
qubes.OpenURL         *    @anyvm    @dispvm    allow
qubes.OpenURL         *    @anyvm    @anyvm     ask
```

These mean: if a qube wants to open a file in a disposable VM, allow it outright [34]; if it wants to open in some specific (non-disp) VM, then ask the user [35]. In practice, this is why when you choose "open in DispVM" it just happens, but if an app tried to open in an existing VM, you'd get a prompt.

Now, you might want to tighten this further. For instance, perhaps you *never* want qubes to open files in each other – you want to **force** usage of disposables. To do that, you can override the policy to remove even the "ask" option for non-disp targets. Essentially, you'd allow disposables and deny everything else. Here's how you could do it:

In your `30-user.policy`, add:

```
qubes.OpenInVM    *    @anyvm    @dispvm    allow
qubes.OpenInVM    *    @anyvm    @anyvm     deny
qubes.OpenURL     *    @anyvm    @dispvm    allow
qubes.OpenURL     *    @anyvm    @anyvm     deny
```

By placing these rules in a lower-numbered file, they will come before the defaults. When a qube asks to open a file or URL: if the target is a DispVM (or no specific target, defaulting to DispVM), it matches the first rule and is allowed. If it's any other target, the next rule matches and it's denied without even a prompt. This guarantees that the only way those actions happen is via disposables. (The user will see “Request refused” if something tries to open in a regular VM – which is your cue that the policy blocked it.)

This setup is used by security-conscious users who want to ensure, for example, that an “Open in VM” button can’t accidentally be pointed to a sensitive VM. In 2025, the default already encourages disposables, so many users leave the default “ask” in place (so they have the option in a prompt to approve a non-disp target if truly needed). But if you prefer a hard line, the above rules accomplish that. It’s a good illustration of how a couple of lines can change a policy from “allow with confirmation” to “forbid entirely except for one case”.

(GUI note: The URL Handling and File Access sections in Global Config let you set a “preferred” disposable VM for opening things. They don’t outright deny non-disp, but they automate using disposables. Our manual rules above are a stricter enforcement. You could combine approaches – for instance, set your default disposable VM for opening PDFs via GUI, and also have a deny rule to be sure.)

Example 4: Fine-Grained USB Device Access

Attaching USB devices in Qubes is mediated by policy as well. In Qubes 4.x, USB devices are typically managed by a dedicated USB qube (say `sys-usb`). When you attach a device to a target VM, what’s happening under the hood is an RPC service call (or admin call) that passes the device identifier and requests attachment. By default, Qubes will prompt you if you try to attach a device from the USB qube to another qube, unless you’ve set it to remember or always allow for that VM. We can customize this behavior – for security or convenience – using arguments in policy.

Why arguments? A USB device attach request includes which device you’re attaching. We can use that as the argument to allow or deny specific devices. Imagine you have two USB drives: one is a trusted backup drive (ID “device1”), the other is an unknown USB stick (ID “device2”). You want to automatically allow the backup drive to be attached to your vault qube, but never allow the unknown stick to attach anywhere (for safety), and for any other devices not explicitly known, just ask.

Suppose the service name for USB attachment is (for example) `qubes.USB` (note: actual Qubes uses something like `qubes.USB+<bus>:<device>` and then a separate `qubes.USBAttach` call – but we’ll simplify). We could write rules:

```
qubes.USB    +device1    sys-usb    vault       allow
qubes.USB    +device2    sys-usb    @anyvm      deny
qubes.USB    *          sys-usb    @anyvm      ask
```

This is analogous to a generic example in Qubes docs[36]. Here `device1` and `device2` would correspond to identifiers for those USB devices. The first rule says requests to attach “device1” from sys-usb to vault are allowed (no prompt). The second says any attempt to attach “device2” from sys-usb to any VM is denied (maybe it’s a rubber ducky or something you’ve deemed malicious). The third rule is a wildcard that matches any other device: it will ask you for confirmation. The source is `sys-usb` in these rules, because generally the USB qube is the one presenting the device to target qubes.

With such rules in place, you could confidently insert your backup drive and attach it to Vault without a prompt (the rule auto-allows it), while any unknown device will trigger an ask (or a deny if it matches a banned device). This is very powerful. Qubes' new policy system encourages using arguments for exactly this reason – it avoids creating overly broad allows. Instead of saying "sys-usb can attach any device to vault" (which might be risky if you someday plug a bad device), we specify the exact device ID we trust.

In practice, how do you get the device identifiers? Usually via Qubes Devices widget or the command line (`qvm-usb` or `qvm-block` will list attached USB devices with identifiers). You'd plug those into the policy line (like `device1` might be something like `04f2_abcdef` for a specific USB).

By 2025, some users have very fine-tuned USB policies – for example, auto-attach Yubikeys to a particular qube for 2FA, allow keyboard/mouse if they're on a known USB controller but deny any new HID devices, etc. Qubes Global Config's **USB Devices** section (see screenshot above) provides a simpler interface: it has drop-downs for keyboard/mouse (always ask vs. allow) and an option to enable the Qubes U2F Proxy service. Those settings under the hood manipulate rules for `qubes.USB` or the `qubes.U2F` service. If you need something more custom (like we did with specific drive IDs), you'd do that in the policy file manually.

(Advanced note: USB attachment in Qubes 4.1+ may actually use admin API calls like `admin.vm.device.usb.Attach` *with arguments for device and target. The same principles apply – you'd match on the device argument and allow/deny accordingly. The exact service name might differ, but the concept of* `service + argument` *holds. The example above illustrates the approach without diving into admin policy includes.)*

Example 5: Controlling Template Updates Proxy

Qubes OS updates for TemplateVMs (the VMs that software is installed in) are often fetched through a designated **Update Proxy**. By default, when a TemplateVM tries to update, it doesn't have network access itself – instead, it contacts a UpdatesProxy service in a networking qube (like sys-net or sys-whonix). The policy governing this is `qubes.UpdatesProxy`. In the default policy, you'll find rules such as [37] [17] :

- Whonix Templates (tagged `whonix-updatevm`) are allowed to use sys-whonix as their update proxy [38] and explicitly denied from using any other VM [39] (to ensure all their updates go through Tor).
- For all other TemplateVMs, the default rule sends updates to sys-net [17] (which in a typical Qubes setup means it goes out to the Internet without Tor by default).
- A final deny rule prevents any qube that didn't match above from using the UpdatesProxy service (so you can't have an AppVM accidentally calling it) [17] .

If you, as a user, want to **change the updates flow**, you can override this. A common tweak in 2025: route *all* template updates through Tor (sys-whonix) for better metadata privacy. The Qubes docs even hint at this with a commented-out line in the default policy that would allow all TemplateVMs to use sys-whonix [40] . To implement it, you'd add to your policy file:

```
qubes.UpdatesProxy    *    @type:TemplateVM    @default    allow    target=sys-
whonix
```

This rule says: any TemplateVM requesting updates (the service argument is not used, so `*`) with target `@default` (meaning the template itself didn't specify a particular proxy VM) is allowed, and it will redirect (`target=sys-whonix`) the request to your Tor proxy VM [37]. You would put this in `30-user.policy` *above* any other UpdatesProxy rules. Because it matches all TemplateVMs, it will catch them before the default sys-net rule in 90-default.policy. The result: now every template update check goes through `sys-whonix` by default. (We also know the default policy denies whonix templates from using others, which is fine – we're using sys-whonix for all anyway.)

If you implement this, test it by doing an update in a Fedora or Debian template – it should succeed (and if you watch the sys-whonix's tor traffic you'll see update connections). If it fails, ensure sys-whonix is running and has network, etc.

Another scenario: some people set up a **caching proxy** (let's say a VM `sys-cacher`) to cache Linux updates and save bandwidth. In that case, they might set the UpdatesProxy target to that VM instead. For example, `target=sys-cacher` in a similar allow rule. The key is you can point the UpdatesProxy to any VM you trust to handle updates. Just make sure that VM is firewalled or isolated appropriately if needed. The Qubes forum has community guides on using a custom update proxy VM – which typically involve adding a rule like the above.

(Note: The UpdatesProxy service is one of those that uses arguments behind the scenes (it passes the URL being fetched, etc.), but as a user you usually don't filter on the argument – you either allow redirect to a specific proxy or not. The default included rules for Whonix templates show how tags and redirection are used in combination – a powerful policy that ensures anonymity for those templates by enforcing Tor.)

Example 6: Split GPG – Using a Vault for GPG operations

"Split GPG" is a popular Qubes feature where your GPG private keys reside in a secure vault VM, and other app VMs (like your email VM) can *ask* that vault VM to sign or decrypt messages via a qrexec service, instead of handling keys directly. This greatly limits exposure of your keys. How does this work? Through a service called `qubes.Gpg`. The policy needs to allow your email qube to invoke `qubes.Gpg` in your vault.

Let's say your vault VM is named `vault` and your email VM is `work-mail` (tagged as "work"). You want `work-mail` to be able to use the GPG service on `vault`, but obviously you don't want *any* VM to do so without permission. You have a few choices: you could allow it outright, or you could require an "ask" every time, depending on your comfort.

A typical policy line would be:

```
qubes.Gpg    *    work-mail    vault    ask
```

This means: when `work-mail` calls `qubes.Gpg` (the argument `*` covers any specific GPG operation), with vault as the target, the system will prompt you to approve or deny [41]. The prompt is nice because you'll see, for example, "work-mail wants to use vault's GPG to sign data" and you can hit OK or Cancel. If you trust your `work-mail` not to misuse GPG (and perhaps you're okay with automatic signing), you might even set it to `allow` instead of ask, for less friction. But many keep it as `ask` for a vault service.

If you had multiple qubes that need GPG, you could use a tag instead of naming them individually. For instance, tag all qubes that are allowed to use GPG as `gpg-client`, and then: `qubes.Gpg * @tag:gpg-client vault ask`. This way, adding the tag to a VM immediately grants it GPG access (with prompt).

In Qubes Global Config's **Split GPG** tab, when you designate a "Split GPG vault" and select which qube to use it, the tool will auto-generate a policy rule like the above for you [25]. It saves you writing it manually. It likely adds an entry in a policy include file that says allow or ask for that specific pairing.

By 2025, many users use Split GPG and even split SSH similarly (though split SSH might require a custom service). The key principle is: a less-trusted VM never sees the private key – it just sends data to the vault via `qubes.Gpg`, the vault does the cryptographic operation, and returns results. The policy ensures only approved qubes can do this, and only to the vault VM.

(Security tip: Even if you allow a service like qubes.Gpg, consider keeping it as `ask` *unless the operations are too frequent, so you have a chance to catch any unexpected usage. The prompt will show* which *command is being run (for qubes.Gpg it might not show granular detail, but for some services like qubes.VMExec, the prompt actually shows the command requested* [42]*). Fewer prompts is nice, but you don't want to become blind to potentially dangerous requests either.)*

These examples scratch the surface, but cover common patterns people use: isolating sensitive qubes, grouping qubes by trust levels, enforcing disposables for risky activities, managing hardware access, and customizing update routes.

Best Practices for Policy Management

To wrap up, here are some **best practices** and tips that Qubes users (new and experienced alike) follow in 2025 when dealing with policies:

- **Least Privilege:** Grant the minimal access needed for things to work. If one qube truly never needs to talk to another, you can explicitly deny it. Conversely, avoid blanket allows like `@anyvm @anyvm allow` – each allow should have a justification. The default policies are quite cautious (lots of "ask" and some denies for dangerous services), which is a good starting point. Only relax a policy to `allow` if you fully understand the implications. For instance, think hard before allowing something like `qubes.VMShell` between qubes – the default policy outright denies it for good reason: if allowed, one VM can execute arbitrary commands in another, effectively **taking full control** of it [43] [44]. Qubes developers warn that instead of allowing such broad services, you should use more specific services that limit what can be done [45] (e.g. use qubes.StartApp for launching only pre-defined applications, rather than a full shell). Keep those warnings in mind.

- **Use "ask" to keep you in the loop:** It's often wise to use `ask` for actions that *could* be risky until you're comfortable. For example, maybe you set up a custom printing service between a VM and sys-firewall – at first, set it to ask so you see when it's triggered and can monitor. If it becomes too chatty and you're confident it's safe, you can flip to allow later. The prompt not only gives you control but also provides a visual audit of what's happening. If you start seeing unexpected prompts ("Untrusted VM wants to send clipboard to Work VM" when you didn't initiate that, for example), it's a red flag.

The Qubes policy system even allows you to get notifications on allowed events if you opt in with `notify=yes`, so you can, say, log whenever a certain service is used [20].

- **Organize with Tags and Multiple Files:** As we demonstrated, tags are your friend for managing groups of qubes. If you find yourself writing many rules for each VM that look similar, consider using a tag to simplify. For instance, rather than have five separate rules for each of 5 work VMs to deny something, tag them and write one rule with @tag. This reduces mistakes and makes the policy easier to read. Similarly, you can split your policy into logical files if it grows large – e.g. a file just for inter-qube cut/paste rules, a file for device rules, etc. This isn't required, but can make it easier to maintain. Just remember the numbering: all files are merged by increasing number, so a rule in `20-policy.policy` will beat one in `30-policy.policy` if they both match.

- **Leverage Qubes Tools for safety:** If you're unsure about syntax, use the Qubes Policy Editor or Global Config to make changes – they will catch syntax errors and often won't let you do something outright dangerous without warning. For example, if you removed all rules for a critical service, the policy daemon would refuse to load (and thus deny everything) which might break functionality. The GUI tries to prevent malformed rules. Always keep in mind that an empty or broken policy means "deny all" by default [46], so it fails safe. If something stops working after your edit, suspect a typo or ordering issue in your rules.

- **Stay updated on community practices:** The Qubes community forum and documentation often share good policy setups. In 2025, users have, for example, posted their hardened policy sets for things like "how to make a vault absolutely quiet" or "using disposable sys-net and firewall – what policies to adjust," etc. Reading those can provide ideas. (Of course, always adjust to your needs; someone else's threat model might differ.)

- **Test in a safe environment:** If you're doing drastic changes, you might test on a non-production Qubes install or at least be prepared to revert. Generally, Qubes is robust against messing up policies since you can always go to dom0 and fix the text. But be careful with Admin API policies – if you were to allow some untrusted VM an admin call, you could inadvertently give it control over your system. So use the same caution you would editing sudoers or a firewall on a normal Linux system.

- **Understand default rules before overriding:** The default `90-default.policy` and `90-admin-default.policy` are well-commented (open them in the editor to read the comments). They explain why certain things are denied or allowed [4] [47]. For instance, they explicitly warn about services like `qubes.VMExec` and `qubes.VMShell`, and they only allow them in the very narrow case of source to its own disposable [48] [44]. When you create an override, ensure you're not inadvertently contradicting a security assumption. If, for example, you allowed `qubes.VMExec * @anyvm @anyvm ask` globally, you'd get a prompt whenever any VM wants to run a command in any other – but the prompt showing the command doesn't guarantee that command is the whole story (a compromised source VM could trick you). The default denies it for a reason [42] [49]. So, when deviating from defaults, be sure you accept the risks.

- **Use notifications/logging for audit:** Consider enabling notifications for critical rules. If you set `notify=yes` on an `allow` rule for, say, your vault, you'll get a pop-up every time that action happens [21]. This can be noisy if frequent, but for something that should rarely happen, it's a great alarm bell. Qubes also logs RPC policy decisions in the system journal (you'll see entries for allowed/denied actions), which you can review if troubleshooting.

By mastering the Qubes policy system, you essentially become the network admin of a mini-network inside your box – dictating which "computer" (qube) can talk to which. It may seem complex at first, but the fundamentals are consistent. Start with small adjustments (like the examples above), and gradually you'll build a policy regime that fits your workflow and security needs.

Recap: Always prefer specific over general rules, keep your high-value qubes locked down, exploit Qubes' features like DispVMs and tags to your advantage via policy, and don't hesitate to use the GUI tools to manage it in a structured way. The new policy system (since Qubes 4.1) is more powerful and easier to manage than ever [50] [51], giving you, the user, fine-grained control over data flows in your Qubes OS. Happy compartmentalizing, and stay safe out there!

Sources: Qubes OS Official Docs and Community Guides have been referenced throughout for accuracy and additional details. Enjoy configuring your Qubes OS policies with confidence, knowing exactly **how, when, and why** your qubes can interact under the rules you set. [1] [35] [43] [52] [24]

---

[1] [7] [12] [20] [21] [22] [30] [46] [50] [51] Qubes Architecture Next Steps: The New Qrexec Policy System | Qubes OS
https://www.qubes-os.org/news/2020/06/22/new-qrexec-policy-system/

[2] [5] [23] [31] [32] [33] [41] RPC policies | Qubes OS
https://www.qubes-os.org/doc/rpc-policy/

[3] [4] [9] [14] [17] [19] [34] [35] [37] [38] [39] [40] [42] [43] [44] [45] [47] [48] [49] Qubes/core-admin - Gogs
https://git.lsd.cat/Qubes/core-admin/src/master/qubes-rpc-policy/90-default.policy

[6] [8] [10] [11] [13] [15] [16] [18] [36] [52] Qrexec: secure communication across domains | Qubes OS
https://www.qubes-os.org/doc/qrexec/

[24] [25] How to use the Qubes Admin Policies/API despite the lack of documentation - WIP - Community Guides - Qubes OS Forum
https://forum.qubes-os.org/t/how-to-use-the-qubes-admin-policies-api-despite-the-lack-of-documentation-wip/29863

[26] Qubes OS 4.2 Releases With PipeWire Support And Refreshed Core Apps
https://news.itsfoss.com/qubes-os-4-2-release/

[27] [28] [29] core-admin/90-admin-default.policy.header at 14e9154e4e283187f893939eea72b294f07ce178 - core-admin - Gitea: Git with a cup of tea
https://git.lsd.cat/Qubes/core-admin/src/commit/14e9154e4e283187f893939eea72b294f07ce178/qubes-rpc-policy/90-admin-default.policy.header