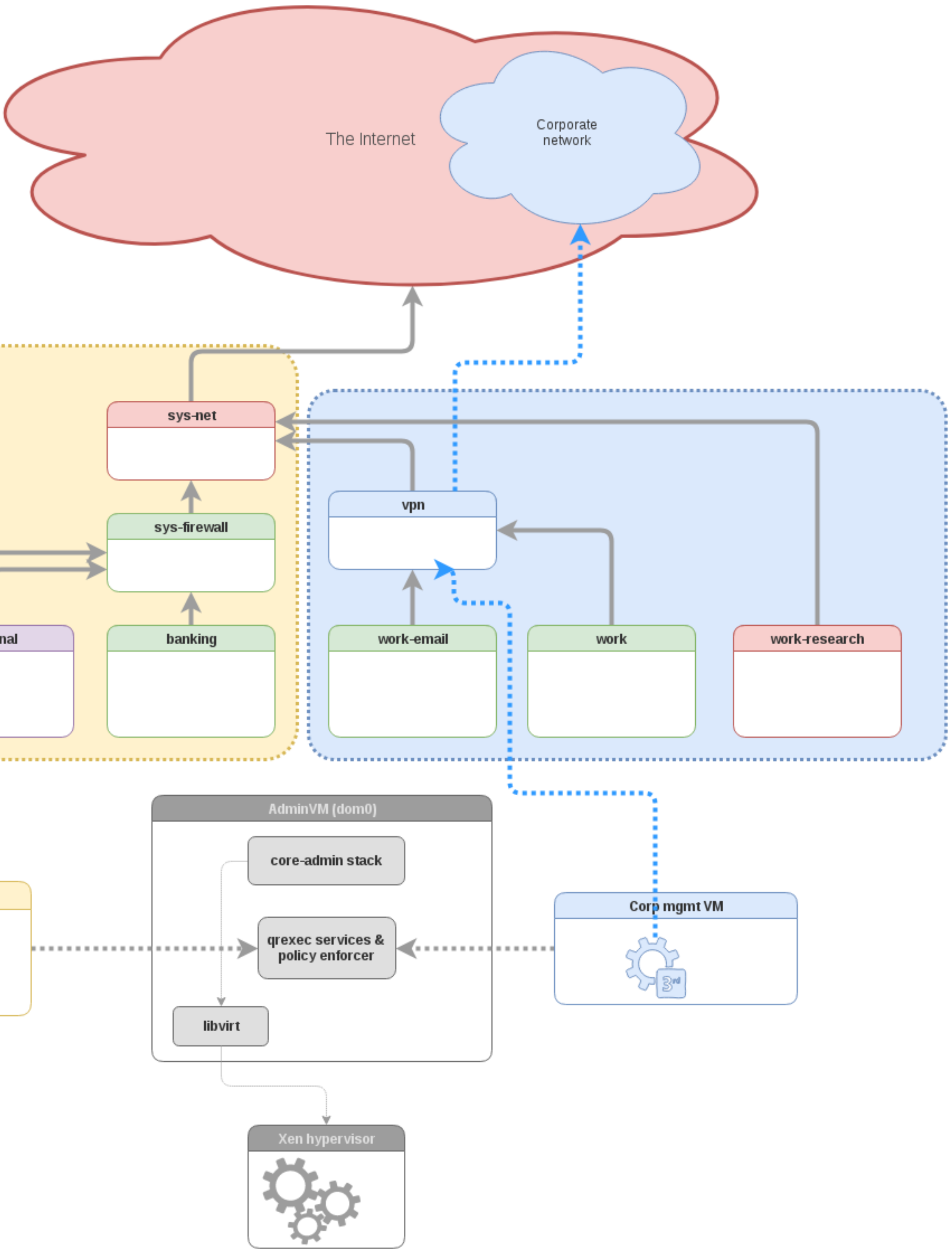
Introducing the Qubes Admin API

30-38 minutes ⋮ 6/26/2017

This post starts the "cool things coming in Qubes 4.0" series and focuses on what we call the "Qubes Admin API." This should not be confused with Qubes Salt Stack integration, which we have already introduced in Qubes 3.2.

High-level overview

Let's start with a high-level architecture picture of how the Admin API fits into the Qubes OS architecture:



As we can see, the main concept behind the Admin API is to let *select* VMs preform various *select* administrative functions over the Qubes OS system.

If this idea scares the hell out of you, then, my dear reader, we're on the same side. Indeed, if we're not careful, we can use the Admin API to shoot ourselves in the foot. Moreover, it might look like we're actually adding complexity and enlarging the amount of trusted code (TCB) in Qubes OS. All good intuitions. But below I argue that the opposite actually holds, i.e. that the Admin API allows us to actually *shrink* the amount of trusted code, *simplify* trust relationships in the system, and ultimately to improve the overall security at the end of the day. It's a bit like comparing SSH to Telnet. Admittedly, at first sight, the SSH protocol has much more complexity than Telnet, yet no one questions today that SSH is actually significantly more secure than the much simpler Telnet.

So, let's first quickly look at examples of why we have introduced the Admin API, i.e. what problems it helps to solve and how.

Management VMs

For Qubes OS to become suitable for use in large organizations and/or corporate environments, it inevitably must become remotely manageable by entities such as corporate IT departments. There are, of course, many ways to implement this, but most would punch too many holes in the Qubes security model. For example, if we wanted to run some management agent in dom0, this would not only open up possible ways of attacking the whole system by exploiting potential bugs in the agent itself, but it would also require us to allow networking in dom0, exposing it to a number of additional attacks.

The Admin API solves this problem elegantly without requiring network access to dom0, and we show exactly how below.

Additionally, the Admin API nicely complements our existing Salt Stack integration. While the latter is perfect for pre-configuration at install time, the former is ideal for ongoing system maintenance, monitoring, and on-demand provisioning of VMs.

Last but not least, we've designed the Admin API with the goal of allowing very strict "need to know" (and "need to do" for that matter) rules. This means that it should be possible to have admin roles (implemented as specific VMs) that would be able to e.g. provision and manage a user's AppVMs, **but not be able to read the user's data!** Of course, this is more tricky than it might seem when we look at the diagram above, and I discuss some of the catches and solutions below. We hope this will pave the way for organizations to embrace the idea of **non-privileged admins**.

The GUI domain

In the current Qubes architecture we've combined two different subsystems: 1) the (trusted) GUI subsystem and 2) the Admin stack. They're both in the same VM: the almighty dom0.

This has somewhat made sense, since the GUI subsystem, which comprises the Window Manager, X server and graphics drivers, must be trusted anyway. After all, the GUI subsystem "sees" all your sensitive data, and it can also mimic any user actions by injecting keystrokes and/or mouse movements.

Also, in contrast to traditional desktop systems, the Qubes GUI subsystem is very well isolated from the external world (user apps, USB devices, networking, etc.), which means there is very little chance of compromising it from the outside.

Yet, there are several good reasons to move the GUI code away from the Admin VM (i.e. dom0) to a separate, *less* trusted and *less* privileged VM:

1. The possibility of efficiently separating the user and admin roles. This is somewhat complementary to the admin de-privileging mentioned above and discussed in more detail at the end of this post.

2. Protection against hypothetical attacks compromising the GPU via DP/HDMI ports.

3. The option to use a much smaller, less bloated, more difficult to backdoor-by-the-vendor (ideally reproducibly buildable) distro for dom0.

4. More freedom in choosing the software stack for the GUI: the actual OS (e.g. Windows instead of Linux?); more recent, less trusted GPU drivers (e.g. proprietary instead of from the Linux tree); more choices with regard to desktop environments and window managers; and an easier upgrade cycle for this whole stack.

5. Reliable logging, which can be separated and protected from both user and admin interference.

6. The possibility of implementing a bunch of cool features, which otherwise (i.e. if the GUI were in dom0) would be too risky to even consider, such as:

    - Exposing the GUI domain through VNC or RDP to other computers, allowing for screen sharing or a Qubes network appliance.

    - GPU/OpenGL virtualization for AppVMs.

    - Perhaps even Qubes-as-a-Service running in the cloud?

I should note that, as of Qubes 4.0, we still don't have a separate GUI domain, but we plan to implement this in the upcoming 4.1 release (and some work has already been done here, in fact). The Admin API will be instrumental in this undertaking.

Safe third-party templates

One of the "bottlenecks" we've been experiencing recently with Qubes is that there is a growing number of people interested in creating and maintaining various, more-or-less exotic TemplateVMs for Qubes. But there is a problem with allowing (untrusted) third parties to maintain Qubes templates: the templates need to be installed in dom0.

The primary "carriers" for bringing things into dom0 are digitally-signed RPMs. Of course, the digital signatures are always checked by code in dom0 before the RPMs are processed. While this is an important checkpoint, let there be no misunderstanding: this cannot do anything to stop malicious RPMs if the author has signed, either intentionally or accidentally, a malicious RPM. And RPMs can be malicious very easily, as they contain lots of wrapping scripts, such as pre- and post-install scripts. We have thus been always very careful with enabling third-party repository definitions in dom0.

The Admin API comes to the rescue. You can now process the template's RPM (check signature, unpack, execute scripts) within a normal, possibly untrusted VM (even a Disposable VM), and then use the Admin API to request installation of the template *files* (not the RPM!) in dom0. The difference is huge, because now we're just asking the Qubes Core Stack to use the content of an opaque file, i.e. the VM image, to fill a newly-created LVM volume (which is done by executing dd in dom0, so without parsing the untrusted volume content), instead of performing a very complex package installation operation (i.e. `rpm -i some-community-template.rpm`). (In Qubes 4.0, we switched to using LVM storage by default for all VM images; more details in an upcoming post.)

Needless to say, not every VM would be able to request template installation in dom0. For this to work, the qrexec policy would need to allow the set of Admin API calls needed for this (see below for practical examples).

Also, let's be clear: if the third-party template maintainer turns out to be malicious (or the computer on which she builds the template turns out to be compromised), then the user will end up with a compromised template, and each and every AppVM based on or created from this template will be compromised also. But the whole point is that *other* AppVMs, those based on *other* templates (e.g. the built-in ones) won't be compromised. Furthermore, the user might, in some cases, prevent the compromised AppVMs from doing any significant damage, e.g. by using the Qubes firewall to limit their ability to leak out data, or even by running the AppVMs completely disconnected from all networking.

Sandboxed Paranoid Backup Restores

Very similar to creating a new template is a backup restore operation. And just as we want to allow less-trusted third-parties to provide templates, in the case of backups we want to be able to restore backups made on a potentially compromised machine (e.g. an older version of Qubes OS that contained a Xen bug vulnerability or perhaps a backdoored BIOS) without compromising the rest of our new system. This concept, known as **Paranoid Backup Restore mode** has been described in more detail in a separate post.

As described in that post, one of the problems with the present implementation of the Paranoid Backup Restore Mode is the need to parse the complex backup format. And this very action can now be done from within a Disposable VM using the Admin API.

Policing the Admin API (and catches!)

Because the Admin API has been implemented on top of qrexec, it can be policed using the standard qrexec policy.

This might sound easy, but, as usual, the devil's in the details. Take a look again at the high-level diagram above. We see that, not surprisingly, all the Admin API-related qrexec services have the same destination: dom0. While this allows us to make somewhat meaningful policies (e.g. mgmtvm1 can invoke `admin.vm.List` and `admin.property.List`, but not `admin.vm.Create`), this is far from what one might like to have. Indeed, we would like to say that a given VM can invoke certain Admin API calls *only for specific AppVMs* and cannot touch other sets of AppVMs.

To make this finer-grained policing possible, we need a simple trick: the `destination` argument for all Admin API calls should be interpreted on a slightly higher level than the qrexec protocol level. Our qrexec policy allows us to easily achieve this approach, thanks to the `target=` keyword used in the policy for all of the Admin API calls, as shown below. This also explains why the Admin API table has a column titled "dest".

We can thus easily express policies for the Admin API with much more granularity and flexibility. For example:

```
$ cat /etc/qubes-rpc/policy/admin.vm.Create
mgmt-corpo  $adminvm    allow

$ cat /etc/qubes-rpc/policy/admin.vm.property.Set+netvm
mgmt-corpo  $tag:created-by-mgmt-corpo allow,target=$adminvm
```

The first rule above allows the VM named `mgmt-corpo` to create new VMs in the system. Each such VM will automatically get tagged with the `created-by-mgmt-corpo` tag. We can further refer to this tag in the policy, as shown in the second rule above, to allow said management VM to **operate only on the VMs that it created**. This rule allows the management VM to change the `netvm` property on any of the VMs that it has created. Note that we have introduced a new keyword: `$adminvm`, which is the new preferred way to refer to dom0 in both policies and qrexec calls.

In Qubes, VMs are identified by their names, which the Qubes Core Stack ensures are unique within the system. (This is similar to how paths and filenames uniquely identify files on any filesystem). The use of names, instead of GUIDs, makes it easier for humans to write and audit policies. This means that if the user decides to rename the management VM, the above example rules will need to be adjusted. Luckily, there is no Admin API call to request the renaming of the VM. (We specifically block setting of the name property via the Admin API).

Indeed, in Qubes 4.0 we have decided to make the VM's name an immutable property of the VM object. Once the user creates a VM, the name is set in stone.

About the only way to effectively change the name of a VM, starting from Qubes 4.0, is to perform the clone operation. It's important to note that the clone operation in Qubes 4.0 differs from Qubes 3.x and earlier. In 4.0, it might be more accurate to say that we're cloning the VM's *volume*, rather than cloning the entire VM. This is because the new clone operation can only be performed between two already existing VMs, and the operation requires two steps realized by two separate calls: `admin.vm.volume.CloneFrom` and `admin.vm.volume.CloneTo`. The first call is typically called by the VM that manages the source VM, i.e. the one whose volume we want to clone, and the second call is expected to be called by whatever VM manages the destination VM (of course these might be the same management VM, e.g. the GUI domain in a single-user, unmanaged system).

This ensures that one management VM (perhaps one running some corporate management agent) cannot impersonate one of the VMs managed by another management VM (perhaps the user-operated GUI domain).

Simple monitoring VM demo

Alright, enough theory and hand-waving. Let's now get our hands dirty with some command-line, shall we?

All the examples below have been run on a pre-rc1 release of Qubes 4.0, but they should work just the same on the "final" 4.0-rc1, which is expected to be released "very soon" (TM) after the publication of this post.

First, let's start with something very simple:

```
[user@dom0 ~]$ qvm-create --label yellow test-mon
```

Edit `/etc/qubes-rpc/policy/admin.vm.List` and add the following two rules:

```
test-mon $adminvm allow,target=$adminvm
test-mon $anyvm allow,target=$adminvm
```

The first rule is required to get a list of all the VMs in the system, while the second is required to query about the state of each of those VMs. Note that the `$anyvm` keyword does *not* include dom0 (aka the `$adminvm`).

If we also want to allow the monitoring VM to query various properties of the VMs in the system, we should also add the second rule to the `admin.vm.property.Get` file (or else `qvm-ls` run in our VM won't display any more info besides the name, type, and power state of the other VMs in the system).

Likewise we should also allow the VM to query the list of labels defined in the system, by inserting the first rule above (i.e. `test-mon $adminvm allow,target=$adminvm`) in the `admin.label.List` policy.

Now, let's start our new monitoring VM, switch to its console, and run `qvm-ls`:

```
[user@dom0 ~]$ qvm-run -a test-mon gnome-terminal
[user@test-mon ~]$ qvm-ls
```

Above we 1) created a new VM named `test-mon`, 2) added a qrexec policy to allow this VM to issue the `admin.vm.List` Admin API call, 3) started the VM, and finally 4) ran `qvm-ls` inside it. It should show the same output as when you run `qvm-ls` in your dom0 console!

One cool thing you can use this for is to provide IP address resolution for Qubes AppVMs in some ProxyVM, such as `sys-firewall`. Then, when running a network monitoring tool, such as `tcpdump`, `wireshark` or `iftop`, you would get nicely resolved IP addresses into Qubes VM names. :)

Take a look at the Admin API calls table to see what other calls your monitoring VM might want to use. We might add more in the future, e.g. to allow measuring (i.e. hashing) of various things, such as the policy used, template root images, etc.

Simple management VM demo

Now, let's move on to something more serious. We shall now create a real management VM:

```
[user@dom0 ~]$ qvm-create --label green test-mgmt
```

We will first create a rule to allow our VM to create new VMs in the system by adding the following rule to the `admin.vm.Create.AppVM` policy:

```
test-mgmt $adminvm allow,target=$adminvm
```

We can actually be even more picky and add this rule to the file: `admin.vm.Create.AppVM+fedora-25`, which would allow the creation of templates based only on the (already installed) template named "fedora-25". Similarly, we could allow only select VM properties to be read or modified. You can read more about qrexec policy arguments and policing them here.

Additionally, we will add the following rule to `/etc/qubes-rpc/policy/include/admin-local-rwx`:

```
test-mgmt $tag:created-by-test-mgmt allow,target=$adminvm
```

This rule is where the Admin API really shines. As previously discussed, it allows our VM to manage only those VMs that it previously created, and not any other VM in the system!

Notice the file in which this rule was placed: `include/admin-local-rwx`. This is a special *include* file. Before we explain what these new files are and how they work, let's briefly discuss how Admin API calls are classified in Qubes 4.0.

Each Admin API call can be classified as having either global or local scope. Global scope means that it can somehow affect (or query) the whole system state, while local scope means that it can only operate on some subset of the VMs (e.g. those granted via the `created-by-` tag mentioned above). If a call seems like it fits into both the local and global scope categories, then it is classified as a global call.

Additionally, each call can be assigned one or more of these three attributes:

- R(eading)
- W(riting)
- X(ecuting)

The R-calls do not affect the state of the system. They only return information about the system (or specific VMs). Both the W- and X-calls affect the state of the system, but while the W-calls change the persistent state (e.g. create new VMs, adjust properties of the VMs, or change firewall policies), the X-calls affect only the volatile state (e.g. start/stop VMs).

Now back to our special include files. We provide 4 predefined includes (all in `/etc/qubes-rpc/policy/include/`):

- `admin-global-ro` – for all R-calls with global scope
- `admin-global-rwx` – for all RWX-calls with global scope
- `admin-local-ro` – for all R-calls with local scope
- `admin-local-rwx` – for all RWX-calls with local scope

Each individual policy file (e.g. `/etc/qubes-rpc/admin.vm.List`) includes one of the above files at the beginning. Thus, by adding a rule to one of these default include files, we conveniently apply the rules for all the calls mentioned above (unless the user modifies the specific per-call policies not to include these default files for some reason). In addition, both `admin-local-ro` and `admin-global-ro` also include `admin-local-rwx` and `admin-global-rwx`, respectively.

Before we test things, we need to allow two more R-calls, which are needed due to a temporary limitation of the current implementation of the `qvm-*` tools, which always attempt to acquire the list of all the VMs in the system. So, we need to either grant access to all the *global* R-calls (note that we added the rule above to `admin-local-rwx`, so this time we would also have to add the rule to `admin-global-ro`), or we need to be more precise by selectively allowing only `admin.vm.List` and `admin.label.List` calls to `$adminvm`:

```
test-mgmt $adminvm allow,target=$adminvm
```

In upcoming releases (beyond 4.0-rc1) we plan to remove this limitation, allowing for the possibility of management VMs that cannot get a complete list of all the VMs in the system.

Now, let's see how our new management VM works:

```
[user@test-mgmt ~]$ qvm-create --label green managed-work
[user@test-mgmt ~]$ qvm-create --label red managed-research
[user@test-mgmt ~]$ qvm-create --label green managed-vpn
[user@test-mgmt ~]$ qvm-prefs managed-work netvm managed-vpn
[user@test-mgmt ~]$ qvm-prefs managed-research netvm sys-whonix
[user@test-mgmt ~]$ qvm-ls
```

```
[...]
```

With the commands above — all executed inside `test-mgmt` — we did a few things: 1) we created three AppVMs: `managed-work`, `managed-research`, and `managed-vpn`, 2) we set `managed-vpn` as the network provider ("NetVM" in Qubes parlance) for `managed-work`, and we also 3) set the system's default `sys-whonix` as a network provider for the other VM.

There are some potential problems with this solution, though. First, `sys-whonix` is not really managed by us, but rather by the user (or perhaps by some other management VM; we — the `test-mgmt` — can't even tell due to the strict policy we used). Second, all the AppVMs we created above are based on the default template, which is also not managed by us. Finally, we would also like to be able to pre-configure some VMs for the user, e.g. configure the VPN client, firewall rules, pre-install some software in the VMs, etc.

One way to solve all these problems in one step is to have our management VM bring its own template(s), which will be used to create the AppVMs that it will subsequently be managing. This is indeed very simple:

```
[user@test-mgmt ~]$ wget https://repository.my-big-organization.com/qubes-work-
template-1.0.0.rpm
# Check the digital signature perhaps? :)
[user@test-mgmt ~]$ sudo dnf install qubes-work-template-1.0.0.rpm
[user@test-mgmt ~]$ qvm-prefs managed-work template qubes-work-template
[user@test-mgmt ~]$ qvm-prefs managed-research template qubes-work-template
```

The template installation step, which is done inside the management VM (rather than in dom0) works because its post-installation scripts automatically execute commands such as `qvm-template-postprocess` (which, in Qubes 4.0, replaces the `qvm-add-template` command along with a bunch of other tasks). These, in turn, are piped over Admin API calls to dom0 (provided that the qrexec policy allows for it, of course). Actually, to support the template installation fully, one would also need to allow the management VM to request the `qubes.PostInstall` call to the newly-installed TemplateVM. (Note this call is not part of the Admin API, because it is implemented by the template itself.) This call wraps many of the actions that must be performed after the template gets installed, such as determining whether it runs the Qubes GUI agent and allowing it to properly set features for the newly-installed template (more about this feature in an upcoming post).

The same could be done, of course, for the templates used by `managed-vpn` and `sys-whonix`. (Presumably, these would be different templates from the "work" template, but they don't have to be.) These custom templates would likely already bring most of the pre-configuration needed for the user. However, if some additional work was needed, e.g. putting in some user credentials for the VPN, this could be achieved using e.g. the standard `qubes.VMShell` service. This service would, however, need to be specifically enabled, as by default **Admin API calls do not imply that the management VM has the ability to execute arbitrary commands in the VMs it manages**.

A better alternative would be to use the upcoming `qubes.InstallPackage` service, which we're planning to introduce shortly after releasing Qubes 4.0. Note that this will not be an Admin API call, because it'll be serviced entirely by the destination VM without any help from the Qubes Core Stack (other than policy enforcement).

The attentive reader will be quick to point out that package installation in a VM is really no different, from a security point of view, from arbitrary code execution, but I argue below that there are important differences that can be used to limit the (ab)use of admin power.

Towards non-privileged admins

As more and more of our activities move to the world of computers, it's becoming increasingly worrying that there is a privileged group of admins who have nearly unlimited power over the (digital) lives of us mere mortal *users*. Interestingly, it is not only users who do not feel comfortable with omnipotent admins, but also the organizations who hire the admins, and even some of the admins themselves. (E.g., some admins may not want the legal liability that comes along with having that power.) Hence, the reduction of admin privileges seems like one of the rare cases in which the interests of individual users and large organizations align.

Let's now think about how can we use the Admin API to make Qubes less prone to admin abuse. First, let's observe that none of the Admin API calls grant direct code execution inside any of the managed VMs or dom0.

So, how can we configure and provision the managed VMs then, if not with `qubes.VMShell`? As already mentioned above, there are a few other, better, options:

1. Bring a pre-configured template.
2. Provision secrets using a dedicated service. In the most trivial case, just `qubes.FileCopy` (typically combined with a pre-configured template, which expects, e.g. VPN credentials, to be copied into: `~/QubesIncoming/mgmt/vpn-credentials/`.
3. Request the `qubes.InstallPackage` service mentioned above (which can only trigger the installation of a package from an *already-defined* repository of the VM).

At the end of the day, of course, admins choose and install software for the user, so they can always decide to install malicious software. But it should be clear that there is a difference between the unconstrained ability to simply read user data or execute arbitrary commands in the user's VM vs. the need to deploy specially backdoored software, potentially also bypassing auditing and signature checking policies. The main goal is to prevent the admin from direct attacks that can copy (i.e. steal) user data or inject backdoors in an unnoticed way.

Additionally, we should point out that, while currently `admin.vm.volume.Import` does *not* enforce any signature checking on the imported VM volumes (e.g. for newly installed templates), this could be relatively easily implemented via a dedicated Qubes Core Stack *extension* (more about extensions in the upcoming post).

Tricky backup operations

Speaking of copying VM images (notably, their private volumes), we cannot avoid mentioning backups. After all, the very essence of a backup operation is to copy user data and store it somewhere in case things go wrong. This operation is, of course, in the domain of admins, not users. So we had to figure out how to allow making backups without revealing user data.

Part of the solution to this problem is something we have long had on Qubes OS. Our unique backup system allows us to write backups into *untrusted* VMs (and, e.g., copy them to an untrusted NAS or USB device). This is possible because we perform the backup crypto operations in dom0, not in the destination VM.

However, when separating the roles of admins and users, we need a way to provide the backup *passphrase* in such a way that the management VM (which ordered the backup creation) will not be able to sniff it.

The way we decided to tackle this problem in the Admin API is through the introduction of backup profiles: simple files that must be created in dom0 and that contain the list of VMs to backup, as well as the command to obtain the passphrase to encrypt the backup. The command might be, e.g., a trivial qrexec-call to some VM (over which the admin doesn't have control!), perhaps extracting the passphrase from some Hardware Security Module.

Towards sealed-off dom0

Ideally, we would like to de-privilege both the admin and user roles enough that neither can interfere with the other. This means, e.g., that the admin will not have access to the user's data, while the user cannot interfere with the admin's policies. Of course, at bottom, there will still be dom0 with its ultimate control over the system, but perhaps it could be sealed off in such a way that neither admins nor users can modify system-wide policies, VM images, or dom0 software.

One approach to this would be to use a digitally-signed filesystem in dom0 (in additional to an encrypted filesystem, which we've been using since the beginning). When (or if) combined with a hardware-enforced root of trust, such as Intel Boot Guard, Intel TXT, or perhaps Intel SGX, this would make it difficult both for the user and for any attacker who might gain physical access to the laptop (e.g. an Evil Maid) to get access to VM images (where user data are kept, or where backdoors might be injected), system configuration (which includes various policies), and other critical files.

The keys to dom0 could be generated during installation and kept by some special admins (superadmins?), split between several of an organization's admins (and perhaps also the user), or even discarded after initial provisioning. Different deployment scenarios would have different requirements here.

Skeptics might argue that if one has physical access to the computer, then it's always possible to bypass any sort of secure/trusted boot, or remote-attestation-based scheme. Probably true, but please keep in mind that the primary goal is to de-privilege admins so that they cannot easily steal users' data, not to de-privilege users. If the user is willing to spend tens or hundreds of thousands of dollars on mounting an attack, then this same user will likely also find other ways to leak the crucial data, perhaps using methods not involving computers at all. So this is mostly about keeping users away from admin tasks in order to allow the smooth operation of both user and admin activities.

Similarly, admins (and developers!) can always insert backdoors, but we want to make it as expensive and as auditable as possible.

All this is planned for beyond even Qubes 4.1, most likely for Qubes 5.0. But we've just built the architectural foundation required to implement this new kind of security model for endpoint devices (and perhaps not just for endpoints?).

Power to the (power) users

But what about individual power users? After all, there is a group of people who are not only not interested in having their (endpoint) system managed by someone else, but who actually have an allergic reaction to even the remote possibility of this being an option in their systems. Likewise, any kind of user "lock-down" idea is perceived as taking away their basic freedoms.

The Qubes Project wishes to avoid taking sides in this battle between locked-down computing vs. user-tinkerable computing. Instead, we would like to provide tools that allow both scenarios to be realized, recognizing that there are legitimate scenarios for both approaches.

First of all, it should be clear that Qubes OS, i.e. the software, cannot lock anybody down. This is because the only way to lock down a computer is to have its *hardware* implement the locking.

Admittedly, the system might cooperate with the hardware to sustain the locking, e.g. by not allowing arbitrary programs to be executed. But with Qubes OS being open source this is never going to be a problem, because those who don't agree with our decisions regarding role compartmentalization will always be able to build the whole OS from scratch with whatever modifications they desire.

Second, the separation of admin and user duties should be considered on a logical level. In scenarios where the same person plays all three roles – user, administrator, and super-administrator – it is still preferable for Qubes OS to implement this compartmentalized model of roles. In such scenarios, it just so happens that a single person possesses the keys/passphrases to all of the roles.

But then, one might wonder, what prevents employees from using Qubes systems configured this way (i.e. all power to the user) within organizations, and thus bypassing any organization-imposed policies? That's simple: remote attestation of some sort (Secure/Trusted Boot + TPM, Intel TXT, Intel SGX, maybe even something else…).

So it all comes down to this: either the user "roots" her own Qubes system and has unlimited power (and yet, she might still want to logically separate these powers within the system), or she plays by the rules of some organization (and, at minimum, doesn't have the keys to dom0, while perhaps ensuring that no one else has them either) and is able to pass the organization's remote attestation checks and get access to some (organization-provided) data and services.

Summary

The new Qubes Admin API represents a new approach to endpoint management, which has been optimized for symmetric user/admin role separation. Not only is the user prohibited from interfering with admin-enforced policies; the admin is equally prohibited from stealing or otherwise interfering with the user's data. Furthermore, it is possible to implement multiple mutually distrusting or semi-distrusting management VMs.

The Qubes Admin API has been enabled by the new generation of the Qubes Core Stack, which we're introducing in Qubes 4.0, and which I will describe in further detail in an upcoming post. Meanwhile, we hope to release Qubes 4.0-rc1 in the coming weeks. :)

Finally, I'd like to point out that nothing discussed in this post has been Xen-specific. This is aligned with our long-term vision of making Qubes work seamlessly on many different platforms: different hypervisors, container technologies, and possibly even more!