Introducing the Next Generation Qubes Core Stack

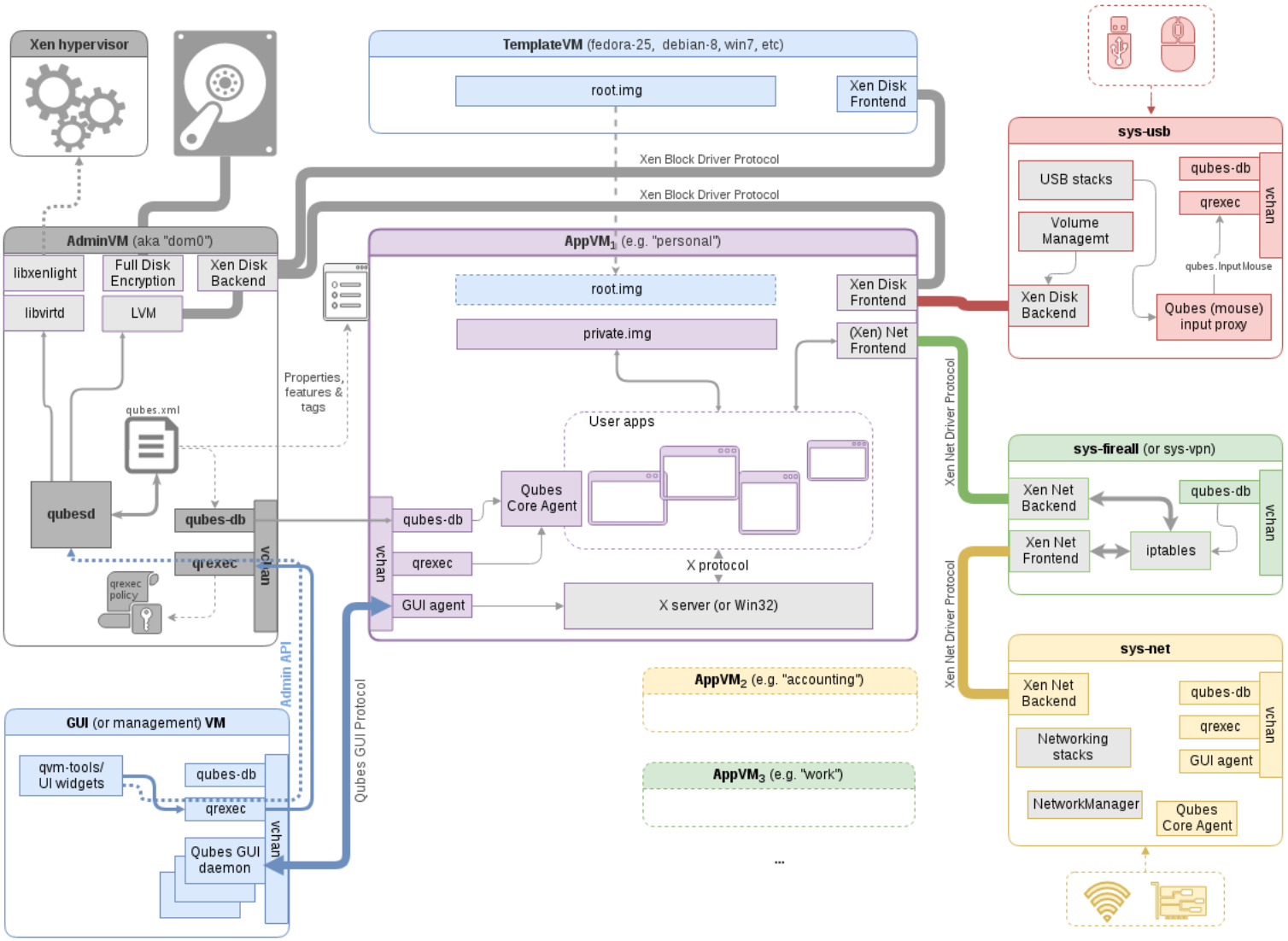
39-49 minutes ⋮ 10/2/2017

This is the 2nd post from the “cool things coming in Qubes 4.0” series, and it discusses the next generation Qubes Core Stack version 3, which is the heart of the new Qubes 4.x releases. The previous part discussed the Admin API which we also introduced in Qubes 4.0 and which heavily relies on this new Qubes Core Stack.

Qubes Core Stack is, as the name implies, the core component of Qubes OS. It’s the glue that connects all the other components together, and which allows users and admins to interact with and configure the system. For the record, the other components of the Qubes system include:

- VM-located core agents (implementing e.g. qrexec endpoints used by various Qubes services),
- VM-customizations (making the VMs lightweight and working well with seamless GUI virtualization),
- Qubes GUI virtualization (the protocol, VM-located agents, and daemons located in the GUI domain which, for now, happens to be the same as dom0),
- GUI domain customizations (Desktop Environment customizations, decoration coloring plugin, etc),
- The AdminVM distribution (various customizations, special services, such as for receiving and verifying updates, in the future: custom distro),
- The Xen hypervisor (with a bunch of customization patches, occasional hardening) or - in the future - some other virtualising or containerizing software or technology,
- Multiple “Qubes Apps” (various services built on top of Qubes qrexec infrastructure, such as: trusted PDF and Image converters, Split GPG, safe USB proxies for HID devices, USB proxy for offering USB devices (exposed via qvm-usb), Yubikey support, USB Armory support, etc)
- Various ready-to-use templates (e.g. Debian-, Whonix-based), which are used to create actual VMs, i.e. provide the root filesystem to the VMs,
- Salt Stack integration

And all these components are “glued together” by the Qubes Core Stack. The diagram below illustrates the location of all these components in the overall system architecture. Unlike many other Qubes architecture diagrams, this one takes an AppVM-centric approach. (Click the image for the full size version.)



There are also a bunch of additional components not shown on the diagram above, and which technically are not part of a Qubes system, but which are instrumental in *building* of the system:

- qubes-builder,
- template-builder,
- tons of automatic tests,
- as well as all the supporting infrastructure for building, testing and distributing Qubes and the updates, and hosting of the website and documentation.

As you can see Qubes is a pretty complex creature, and the Qubes Core Stack is central to its existence.

Qubes VMs: the building blocks for Explicit Partitioning Model

Qubes implements *explicit partitioning* security model, which means that users (and/or admins) can define multiple *security domains* and decide what these domains can and cannot do. This is a different model than the popular sandboxing model, as implemented by increasingly many applications and products today, where every application is automatically sandboxed and some more-or-less pre-defined set of rules is used to prevent behaviour considered “unsafe” (whatever that might mean…). I believe the explicit partitioning model provides many benefits over the sandboxing model, among the most important one being that it is information-oriented, rather than application-oriented. In other words it tries to limit damage to the (user’s) data, rather to the (vendor’s) code and infrastructure.

There have always been a few different kinds of VMs in Qubes: AppVMs, Template VMs, Standalone VMs, NetVMs, ProxyVMs, DispVM, etc. In Qubes 4 we have slightly simplified and cleaned up these categories.

First we’ve hidden the PV vs HVM distinction. Now each VM has a property named `virt_mode` which is used to decide whether it should be virtualized using Xen para-virtualization (PV), full virtualization with auxiliary qemu in isolated “stub domain” (HVM), or – in the near future – as full virtualization *without* qemu and the additional stub domain (PVH). This means we no longer classify VMs as PV vs HVMs, because every VM can be easily switched

between various modes of virtualization with a flip of a property.

We also no longer distinguish between AppVMs, NetVMs and Proxy VMs. Instead, each VM has a property `provides_network`, which is false by default, except for the VMs which we want to expose networking to other VMs (e.g. because they might have some networking device assigned, such as cellular modem or WiFi device, or because they act as VPNs or proxies of some sort).

We discuss what the properties are and how to check/set them, later in this article.

So, to recap, starting with Qubes 4.0 we have only the following classes of VMs:

- `AppVM` - covers what we called AppVMs, NetVMs, and ProxyVMs in earlier Qubes releases,
- `DispVM` - defined by not having a persistent private image across VM reboots (see also below),
- `TemplateVM` - these provide root filesystem for AppVMs, the semantics haven't changed in Qubes 4,
- `StandaloneVM` - these are like AppVMs, but are *not* based on any TemplateVM,
- `AdminVM` a singleton (i.e. a class which has only one instance), which represents the Administrator VM (Up until recently called `dom0`, a term we're departing from, given it is Xen-specific).

One can list all the VM classes known to the stack using the following command:

```
[user@dom0 ~]$ qvm-create --help-classes
```

BTW, we've been recently promoting the use of alternative names to "VM(s)", such as domain(s), and - more recently - "qube(s)". This is to stress the connection between Qubes and virtualization technology is less tight that many might perceive. Another reason is that, we believe, for many users these alternative terms might be more friendly than "VMs", which apparently have strong technical connotation. The author is quite used to the term VM, however, so should be excused for (ab)using this term throughout her writings.

Each VM (domain) in Qubes can have a number of properties, features and tags, which describe both how it should behave, as well as what features or services it offers to other VMs:

- **Properties**, as the name suggests, are used to change the behaviour of the VM and/or how it is treated by the Qubes Core Stack. The `virt_mode` property mentioned above is a good example. For a list of other properties, take a look here. A command which can be used to list, read and set properties for a VM is `qvm-prefs` (see below).

- **Features** are very similar to properties, except that they are mostly opaque to the Core (unlike properties, which are well defined and sanitized). Features are essentially a dictionary of 'key=value' pairs assigned to each VM. They can be used in various places outside of the main logic of the Core Stack, specifically in Core Extensions. A new tool in Qubes 4 used to inspect or set features is called `qvm-features`.

- A good example of a mechanism implemented on top of features are **services**, which have been reimplemented in Qubes 4. Services are used to let the VM agents know about whether various additional services, such as Network Manager, should be enabled or not. Indeed, the `qvm-service` tool now (i.e. in Qubes 4.0) internally simply creates features named `service.XYZ`, where `XYZ` is the name of the service passed to `qvm-service`. It also takes care of interpreting the true/false values.

- Finally, each VM can also have some **tags** associated with it. Unlike features, tags do not have a value – a VM can either have a specific tag ("can be tagged with it") or not. Unlike properties, they are not interpreted by any core logic of the Core Stack. The sole purpose of tags is that they can be used for qrexec policy rules, as discussed below. (This is also the reason why we wanted to keep them absolutely simple – any complexity within qrexec policy parsing code would definitely be asking for troubles…).

Last but not least, we should mention that each of the VM has a **unique name** associated with it. VM names in Qubes are like filenames in the filesystem – not only they are unique, but they also are used as a primary way of identifying VMs, especially for security-related decisions (e.g. for qrexec policy construction, see below).

And just as one can get away with using generic tagging schemes in place of referring to paths and filenames in some security systems, similarly in Qubes OS one can refer to tags (mentioned above and discussed later) in the policy (but currently not in firewalling rules).

Internally, a VM's name is implemented as the property `name` and thus can be read using `qvm-prefs`. It cannot be changed, however, because, starting with Qubes 4.0, we treat it as an immutable property. User-exposed tools, however, do have a "VM rename" operation, which is implemented as creating a copy of the VM with the new name and removing the old VM. Thanks to the new volume manager we also introduced in Qubes 4 (and which will be the topic of another post), this operation is actually very cheap, disk-wise.

So, let us now start a console in AdminVM (`dom0`) and play a bit with these mechanisms.

Let's start with something very simple and let us take a look at the properties of the AdminVM (in the current Qubes implemented by Xen's `dom0`):

```
[user@dom0 ~]$ qvm-prefs dom0
default_dispvm  D  fedora-25-dvm
label           -  black
name            D  dom0
qid             D  0
uuid            D  00000000-0000-0000-0000-000000000000
```

As we can see, there aren't many properties for the AdminVM, and none of them can be modified by the user or admin. While we're here, we should mention there exists a similar tool, `qubes-prefs` which allows one to view and modify the **global system properties**, and these properties should not be confused with the properties of the AdminVM:

```
[user@dom0 ~]$ qubes-prefs
check_updates_vm       D  True
clockvm                -  sys-net
default_dispvm         -  fedora-25-dvm
default_fw_netvm       D  None
default_kernel         -  4.9.45-21
default_netvm          -  sys-firewall
default_pool           D  lvm
default_pool_kernel    -  linux-kernel
default_pool_private   D  lvm
default_pool_root      D  lvm
default_pool_volatile  D  lvm
default_template       -  fedora-25
stats_interval         D  3
updatevm               -  sys-firewall
```

Now, let's inspect an ordinary AppVM, say `work` (which is one of the default VMs created by the installer, but, of course, the user might have it named differently):

```
[user@dom0 ~]$ qvm-prefs work
autostart             D  False
backup_timestamp      D
debug                 D  False
default_dispvm        D  fedora-25-dvm
default_user          D  user
gateway               D
include_in_backups    D  True
installed_by_rpm      D  False
ip                    D
kernel                D  4.9.45-21
kernelopts            D  nopat
label                 -  blue
mac                   D  00:16:3E:5E:6C:00
maxmem                D  4000
memory                D  400
```

```
name                -  work
netvm               -  None
provides_network    D  False
qid                 -  8
qrexec_timeout      D  60
stubdom_mem         U
stubdom_xid         D  6
template            -  fedora-25
template_for_dispvms  D  False
updateable          D  False
uuid                -  fab9a577-2531-4971-bce1-ca0c9b511f27
vcpus               D  2
virt_mode           D  hvm
visible_gateway     D
visible_ip          D
visible_netmask     D
xid                 D  5
```

We see lots of properties which have a `D` flag displayed, which indicates the property uses the core-provided default value. In most cases the user is able to override these values for specific VMs. For example the `virt_mode` property is by default set to `hvm` for all AppVMs, which means that full virtualization mode is used for the domain, but we can override this for specific VM(s) with `pv` and thus force them to be virtualized using para-virtualization mode, which is what the installer scripts do for the `sys-net` and `sys-usb` VMs in Qubes 4.0-rc1 in order to work around problems with PCI passthrough support that we've observed on some platforms.

As a simple exercise with changing the property, we can switch off networking for one of the AppVMs by setting its `netvm` property to an empty value:

```
[user@dom0 ~]$ qvm-prefs --set work netvm ""
```

We can now confirm that indeed the VM named `work` has no network. E.g. we can use the `qvm-ls` command (it should print `-` in the column which indicated the netvm), or open a terminal in the `work` VM itself and try to see if there is a virtual networking device exposing the network (we can e.g. use the `ifconfig` or `ip a` commands).

Instead of setting an empty value as the netvm, we could have also set some other VM, thus forcing the networking traffic to pass through that other VM, e.g. to route all the VM's traffic through a Whonix Tor gateway:

```
[user@dom0 ~]$ qvm-prefs --set work netvm sys-whonix
```

The VM which we want to use as the provider of networking services to other VMs, such as `sys-whonix` in the example above, should have its `provides_network` property set to true.

To revert back to the system-default netvm (as specified by `qubes-prefs`), we can use the `--default` switch:

```
[user@dom0 ~]$ qvm-prefs --default work netvm
```

Now let us take a look at the features and services. It might be most illustrative to look at both of these mechanisms together. Let's suppose we would like to enable the Network Manager service in some VM. Perhaps we would like to use it to create a VPN connection terminated in one of the AppVMs, say in the `work` VM. (An alternative option would be to create a separate VM, say `work-vpn`, set its `provides_network` property to true, enable the network-manager service in there, and use it to provide networking to the other VMs, e.g. `work`.)

To enable the Network Manager service and a handy widget that shows the status of the VPN, we can do:

```
[user@dom0 ~]$ qvm-service --enable work network-manager
```

This should result in the following output:

```
[user@dom0 ~]$ qvm-service work
network-manager  on
```

We see the service is enabled, and if we restart the `work` VM, we should see the Network Manager widget appear in the tray.

Now let's view the features of this same VM:

```
[user@dom0 ~]$ qvm-features work
service.network-manager  1
```

We can now clearly see how the services are internally implemented on top of features. This is done internally by the `qvm-services` command and by the scripts in the VMs, which interpret specifically-formatted features as services. The `qvm-service` tool also takes care about properly setting and interpreting the true/false and empty/non-empty strings to make sense for services states (on/off), eliminating user mistakes.

One thing left for us to explore are tags, discussed above, but we will defer this discussion until a later chapter, which talks about qrexec policy, as tags are exclusively for use within policies. For now, suffice to say, that there is a dedicated tool, `qvm-tags` used to check and set/modify the tags for each of the VM (in addition to some tags being automatically created by the Core Stack, such as e.g. the `created-by-XYZ` tag, discussed in the previous article on Admin API).

As a final note we should stress one more time that VM names, despite being normal property, are special in that the are immutable. In order to change the name one needs to 1) create a new VM, 2) import the volume from the original VM (using `admin.vm.volume.CloneFrom` and `admin.vm.volume.CloneTo` calls), as well as 3) copy all the properties, tags, and features from the original VM, and 4) remove the original VM.

Each of these operations can be controlled independently via qrexec policy. While this might seem like a superficial complication at first sight, we believe it allows to simplify the implementation, as well as to minimize the amount of potential mistakes when creating policies, notably when there is more than one management VMs in the system, such as e.g. the (de-privileged) GUI domain and some corporate-owned (but also semi-privileged) management VM. This has been discussed in more details in the previous post on Admin API.

Disposable VMs redesigned

We have also redesigned how Disposable VMs (DispVMs) work in Qubes 4.0. Before 4.0, DispVMs had two kinds of largely unrelated characteristics: 1) being disposable, i.e. having no persistent private image, and 2) booting from pre-created savefiles in order to save on startup time.

In Qubes 4.0 we have redefined DispVMs solely by this first property, i.e. private-image-not-having, which we believe to be the essential characteristics of a Disposable VM. The underlying mechanism, e.g. whether the VM is restored from a savefile to speed up the startup time, might or might not be implemented for any VM and should not concern the user.

Another major change to DispVMs semantics in Qubes 4 (largely allowed by this relaxation of DispVM definition) is that any of the AppVMs can be now used as a template for the DispVM.

This provides lots of flexibility, e.g. it's easy to create a customized DispVM for signing PDF documents, or various disposable service VMs, such as Net- and VPN- VMs. One just creates a normal AppVM, sets everything up there, e.g. loads keys, configures VPNs, connects to known WiFi networks, and then shuts down the AppVM and in place of it starts a DispVM based on that AppVM.

This way, whenever one has a gut feeling that something's wrong with the VM (e.g. because one just connected a USB stick to copy some slides), it's easy to just restart the DispVM in question and benefit from a new clean root filesystem! Admittedly sometimes this might not be enough to get rid of an infection, as discussed in the recent post on Compromise Recovery, but for many cases this feature is highly desirable.

But opening up this flexibility comes at a price. We have to be careful about which VMs can create which DispVMs. After all, it would be a disaster to allow your casual Internet-browsing AppVM to spawn a DispVM based on your sensitive `work` AppVM. The casual Internet-browsing VM could have the new DispVM open a malicious file that compromises the DispVM. The compromised DispVM would then be able to leak sensitive work-related data, since it uses the `work` VM as its template.

There are two mechanisms in place to prevent such mistakes:

1. Each AppVM has a property called `template_for_dispvms`, which controls whether this VM can serve as a template for Disposable VMs (i.e., whether any DispVMs based on this VM are allowed in the system. By default, this property is false for all AppVMs and needs to be manually enabled.

2. The choice of the template (i.e. specific AppVM) for the Disposable VM must be provided by the qrexec policy (which the source VM cannot modify), and defaults to the source VM's `default_dispvm` property, which by default has a value as specified via `qubes-prefs`. The resulting AppVM must have the `template_for_dispvms` property set, or otherwise an error will occur.

Here we will take a look at how the template can be specified explicitly when starting the DispVM from dom0:

```
[user@dom0 ~]$ qvm-run --dispvm=work --service qubes.StartApp+firefox
Running 'qubes.StartApp+firefox' on $dispvm:work
$dispvm:work: Refusing to create DispVM out of this AppVM, because
template_for_dispvms=False
```

As mentioned above, we also need to explicitly enable use of the `work` AppVM as Disposable VMs templates:

```
[user@dom0 ~]$ qvm-prefs --set work template_for_dispvms True
[user@dom0 ~]$ qvm-run --dispvm=work --service qubes.StartApp+firefox
Running 'qubes.StartApp+firefox' on $dispvm:work
```

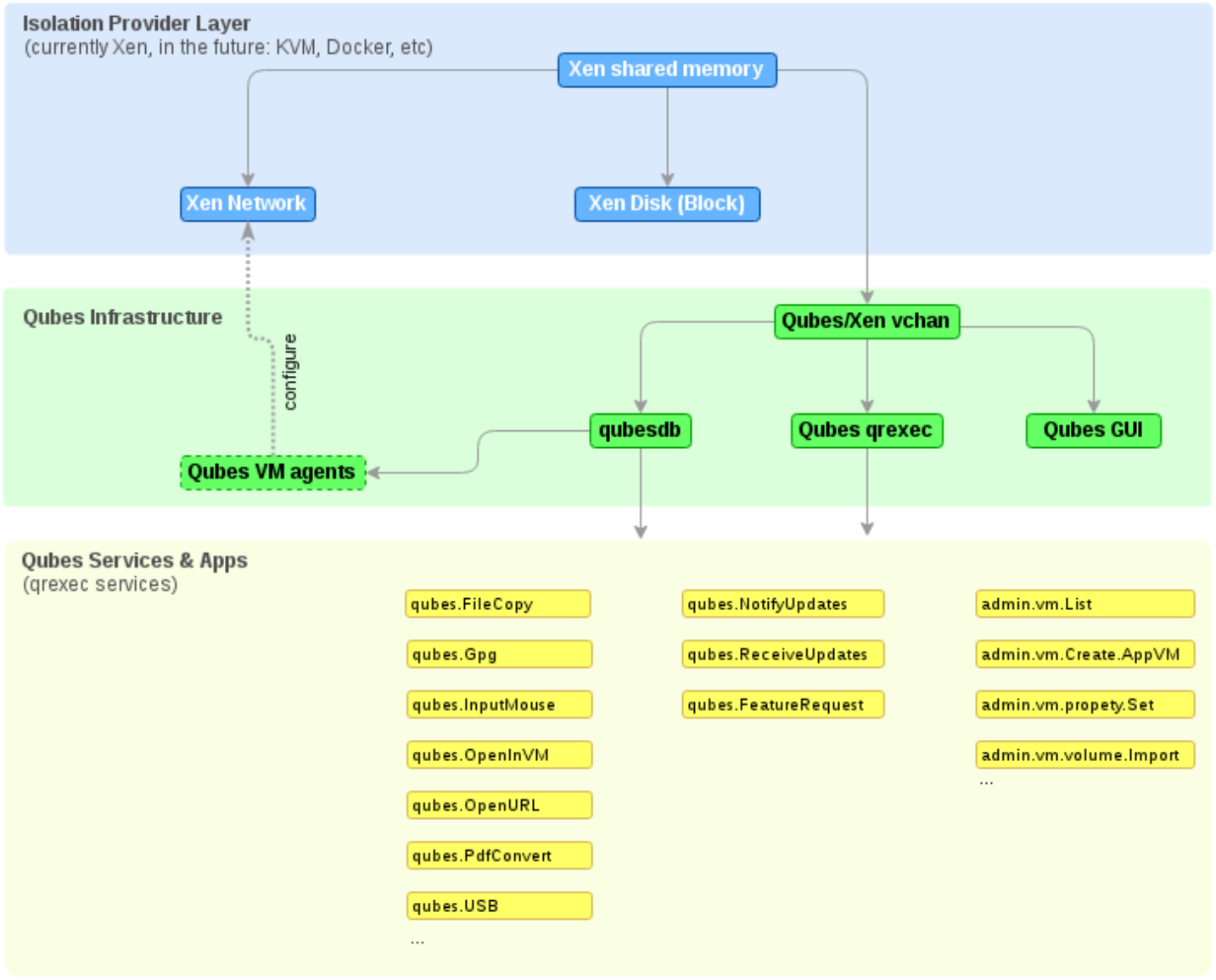
We will look into how the DispVM can be specified via qrexec policy in a later chapter below.

Qubes Remote Execution (qrexec): the underlying integration framework

Qubes OS is more than just a collection of isolated domains (currently implemented as Xen VMs). The essential feature of Qubes OS, which sets it apart from ordinary virtualization systems, is the unique way in which it *securely integrates* these isolated domains for use in a single endpoint system.

It's probably accurate to say that the great majority of our efforts is on building this integration in a manner so that it doesn't ruin the isolation that Xen provides to us.

There are several layers of integration infrastructure in Qubes OS, as depicted in the following diagram (click for full size) and described below:



1. First, there is the Xen-provided mechanism based on shared memory for inter-VM communication (it's called "grant tables" in Xen parlance). This is then used to implement various Xen-provided drivers for networking and virtual disk (called "block backends/frontends" by Xen). Generally, any virtualization or containerization system provides some sort of similar mechanisms, and we could easily use of any them on Qubes, because our Core Stack uses libvirt to configure these mechanisms. Qubes does have some specific requirements, however, the most unique one being that we want to put the backends in unprivileged VMs, rather than in dom0 (or host or root partition, however it is called in other systems). This is one of the reasons why Xen still is our hypervisor of choice – most (all?) other systems assume all the backends to be placed in the privileged domain, which clearly is undesirable from the security point of view.

2. Next, we have the Qubes infrastructure layer, which builds on top of the Xen-provided shared memory communication mechanism. The actual layer of abstraction is called "vchan". If we moved to another hypervisor, we would need to have vchan implemented on top of whatever inter-VM infrastructure that other hypervisor was to make available to us. Various Qubes-specific mechanisms, such as our security-optimized GUI virtualization, builds on top of vchan.

3. Finally, and most importantly, there is the Qubes-specific qrexec infrastructure, which also builds on top of vchan, and which exposes socket-like interfaces to processes running inside the VMs. This infrastructure is governed by a centralized policy (enforced by the AdminVM). Most of the Qubes-specific services and apps are built on top of qrexec, with the notable exception of GUI virtualization, as mentioned earlier.

It's important to understand several key properties of this qrexec infrastructure, which distinguish it from other, seemingly similar, solutions. First, qrexec does not attempt to perform any serialization (à la RPC). Instead, it exposes only a simple "pipe", pretty much like a TCP layer. This allows us to keep the code which handles the incoming (untrusted) data very simple. Any kind of (de-)serialization and parsing is offloaded to the specific code which has been registered for that particular service (e.g. `qubes.FileCopy`).

This might seem like a superficial win. After all, the data needs to be de-serialized and parsed *somewhere*, so why would it matter where exactly? True, but what this model allows us to do is to selectively decide how much risk we want to take for any specific domain by allowing (or not) specific service calls from (specific or all) other domains.

For example, by decoupling the (more complex) logic of data parsing as used by the `qubes.FileCopy` service from the core code of qrexec we can eliminate potential attacks on the `qubes.FileCopy` server code by not allowing e.g. any of the VMs tagged `personal` to issue this call to any of the VMs tagged `work`, while at the same time still allowing all of these VMs to communicate with the default clockvm (by default `sys-net`, adjustable via policy redirect as discussed below) to request the `qubes.GetDate` service. We would not have such a flexibility in trust partitioning if our qrexec infrastructure had serialization built in (e.g. if it was implementing a protocol like DBus between VMs). Of course, specific services might decide to use some complex serializing protocols on top of qrexec (e.g. DBus) very easily, because qrexec connection is seen as a socket by applications running in the VMs.

Another important characteristic is the policing of qrexec, which is something we discuss in the next section.

Let's write a simple qrexec service, which would be illustrative for what we just discussed. Imagine we want to get the price of a Bitcoin (BTC), but the VM where we need it has no network connectivity, perhaps because it contains some very sensitive data and we want to cut off as much interfaces to the untrusted external world as possible.

In the `untrusted` VM we will create our simple server with the following body:

```
curl -s https://blockchain.info/q/24hrprice
```

The above should be pasted into the `/usr/local/etc/qubes-rpc/my.GetBTCprice` file in our `untrusted` AppVM (let's name it… `untrusted`). Let's also test if the service work (still from the `untrusted` VM):

```
[user@untrusted ~]$ sudo chmod +x /etc/qubes-rpc/my.GetBTCprice
[user@untrusted ~]$ /usr/local/etc/qubes-rpc/my.GetBTCprice
(...)
```

We should see the recent price of Bitcoin displayed.

Now, let's create a network-disconnect, trusted AppVM called `wallet`:

```
[user@dom0 ~]$ qvm-create wallet -l blue --property netvm=""
[user@dom0 ~]$ qvm-ls
```

And now from a console in this new AppVM, let's try to call our newly created service:

```
[user@wallet ~]$ qrexec-client-vm untrusted my.GetBTCprice
(...)
```

The above command will invoke the (trusted) dialog box asking to confirm the request and select the destination VM. For this experiment just select `untrusted` from the "Target" drop-down list and acknowledge (in case you wonder why the target VM is specified twice – once in the requesting VM and for the 2nd time in the trusted dialog box: we will discuss it in the next section). You should see the price of the bitcoin printed as a result.

Before we move on to discussing the flexibility of qrexec policies, let's pause for a moment and recap what has just happened:

1. The network-disconnected, trusted VM called `wallet` requested the service `my.GetBTCprice` from a VM named `untrusted`. The `wallet` VM had no way to get the price of BTC, because it has no networking (for security).

2. After the user confirmed the call (by clicking on the trusted dialog box, or by having a specific `allow` policy, as discussed below), it got passed to the destination VM: `untrusted`.

3. The qrexec agent running in the `untrusted` VM invoked the handling code for the `my.GetBTCprice` service. This code, in turn, performed a number of complex actions: it TCP-connected to some server on the internet (blockchain.info), performed some very complex crypto handshake to establish an HTTPS connection to that server, then retrieved some complex data over that connection, and finally returned the price of Bitcoin. There's likely hundreds of thousands lines of code involved in this operation.

4. Finally, whatever the `my.GetBTCprice` service returned on its `stdout` was automagically taken by qrexec agent and piped back to the requesting VM, our `wallet` VM.

The `wallet` VM got the data it wanted *without* the need to get involved in all these complex operations, which take hundreds of thousands of lines of code talking to untrusted computers over the network. That's how we can improve the security of this process without spending effort on auditing or hardening the programs used (e.g. `curl`).

Of course that was a very simple example. But a very similar approach is used among many Qubes services. E.g. system updates for `dom0` are downloaded in untrusted VMs and exposed to (otherwise network-disconnected) `dom0` via the `qubes.ReceiveUpdates` service (which later verifies digital signatures on the packages). Another example is `qubes.PdfConvert`, which offloads the complex parsing and rendering of PDFs to Disposable VMs and retrieves back only a very simple format that is easily verified to be non-malicious. This simple format is then converted back to a (now trusted) PDF.

More expressive qrexec policies

Because pretty much everything in Qubes which provides integration over the compartmentalized domains is based on qrexec, it is imperative to have a convenient (i.e. simple to use), secure (i.e. simple in implementation) yet expressive enough mechanism to control who can request which qrexec services from whom. Since the original qrexec policing was introduced in Qubes release 1, the mechanism has undergone some slight gradual improvements.

We still keep the policy as a collection of simple text files, located in `/etc/qubes-rpc/policy/` directory in the AdminVM (`dom0`). This allows for automating of the policy packaging into RPM (trusted) packages, as well policy customization from within our integrated Salt Stack via (trusted) Salt state files.

Now, one of the coolest features we've introduced in Qubes 4.0 is the ability to tag VMs and use these to make policy decisions.

Imagine we have several work-related domains. We can now tag all them with some tag of our choosing, say `work`:

```
[user@dom0 user ~]$ qvm-tags itl-email add work
[user@dom0 user ~]$ qvm-tags accounting add work
[user@dom0 user ~]$ qvm-tags project-liberation add work
```

Now we can easily construct a qrexec policy, e.g. to constrain the file (or clipbooard) copy operation, so that it's allowed only between the VMs tagged as `work` (while preventing file transfer to and from any VM *not* tagged with work) – all we need is to add the following 3 extra rules in the policy file for `qubes.FileCopy`:

```
[user@dom0 user ~]$ cat /etc/qubes-rpc/polisy/qubes.FileCopy
(...)
$tag:work   $tag:work   allow
$tag:work   $anyvm      deny
$anyvm      $tag:work   deny

$anyvm      $anyvm      ask
```

We can do the same for clipboard, just need to place the same 3 rules in `qubes.ClipboardPaste` policy file.

Also, as already discussed in the previous post on Admin API, the Core Stack automatically tags VMs with `created-by-XYZ` tag, where `XYZ` is replaced by the name of the VM which invoked the `admin.vm.Create*` service. This allows to automatically constrain power of specific management VMs to only manage its "own" VMs,

and not others. Please refer to the article linked above for the examples.

Furthermore, Disposable VMs can also be referred to via tags in the policy, for example:

```
# Allow personal VMs to start DispVMs created by the user via guidom:
$tag:created-by-guidom $dispvm:$tag:created-by-guidom allow

# Allow corpo-managed VMs to start DispVMs based on corpo-owned AppVMs:
$tag:created-by-mgmt-corpo $dispvm:$tag:created-by-mgmt-corpo allow
```

Of course, we can also explicitly specify Disposable VMs using the `$dispvm:<name_of_appvm_to_use_as_template>` syntax, e.g. to allow AppVMs tagged `work` to request qrexec service from a Disposable VM created off `work-printing` AppVM (as noted earlier, the "work-printing" would need to have it's property `template_for_dispvms` set for this to work):

```
$anyvm    $dispvm:work-printing allow
```

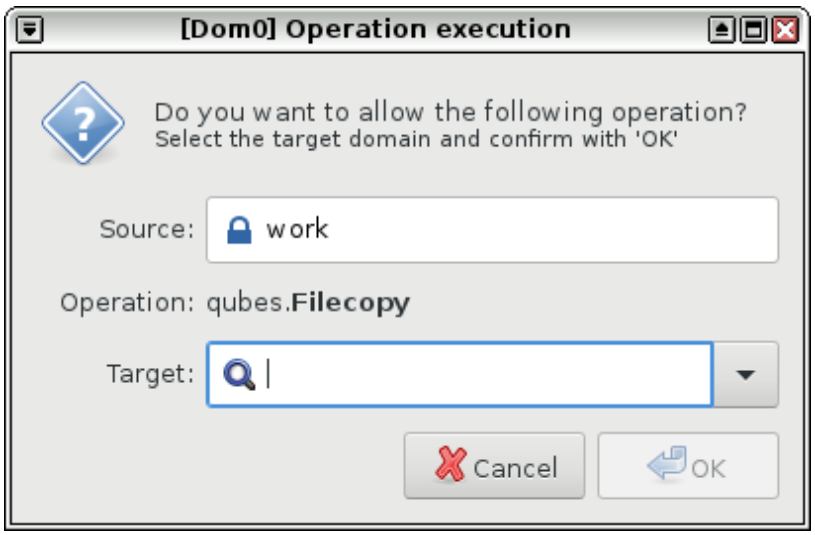
In Qubes 4.0 we have also implemented more strict control over the destination argument for qrexec calls. Until Qubes 4.0, the source VM (i.e., the VM that calls a service) was responsible for providing a valid destination VM name to which it wanted to direct the service call (e.g. `qubes.FileCopy` or `qubes.OpenInVM`). Of course, the policy always had the last say in this process, and if the policy had a `deny` rule for the specific case, the service call was dropped.

What has changed in Qubes 4.0 is that whenever the policy says `ask` (in contrast to `allow` or `deny`), then the VM-provided destination is essentially ignored, and instead a trusted prompt is displayed in dom0 to ask the user to select the destination (with a convenient drop down list).

(One could argue that the VM-provided destination argument in qrexec policy call is not entirely ignored, as it might be used to select a specific qrexec policy line, in case there were a few matching, such as:

```
$anyvm work allow
$anyvm work-web ask,default_target=work-web
$anyvm work-email ask
```

In that case, however, the VM-provided call would still be overridden by the user choice in case the rule #2 was selected.)



A prime example of where this is used is the `qubes.FileCopy` service. However, we should note that for most other services, in a well configured system, there should be very few `ask` rules. Instead most policies should be either `allow` or `deny`, thereby relieving the user from having to make a security decision with every service invocation. Even the `qubes.FileCopy` service should be additionally guarded by `deny` rules (e.g. forbidding any

file transfers between personal and work-related VMs), and we believe our integrated Salt Management Stack should be helpful in creating such policies in larger deployments of Qubes within corporations.

Here comes in handy another powerful feature of qrexec policy: the `target=` specifier, which can be added after the action keyword. This forces the call to be directed to the specific destination VM, no matter what the source VM specified. A good place to make use of this is in policies for starting various Disposable VMs. For example, we might have a special rule for Disposable VMs which can be invoked only from VMs tagged with `work` (e.g. for the `qubes.OpenInVM` service):

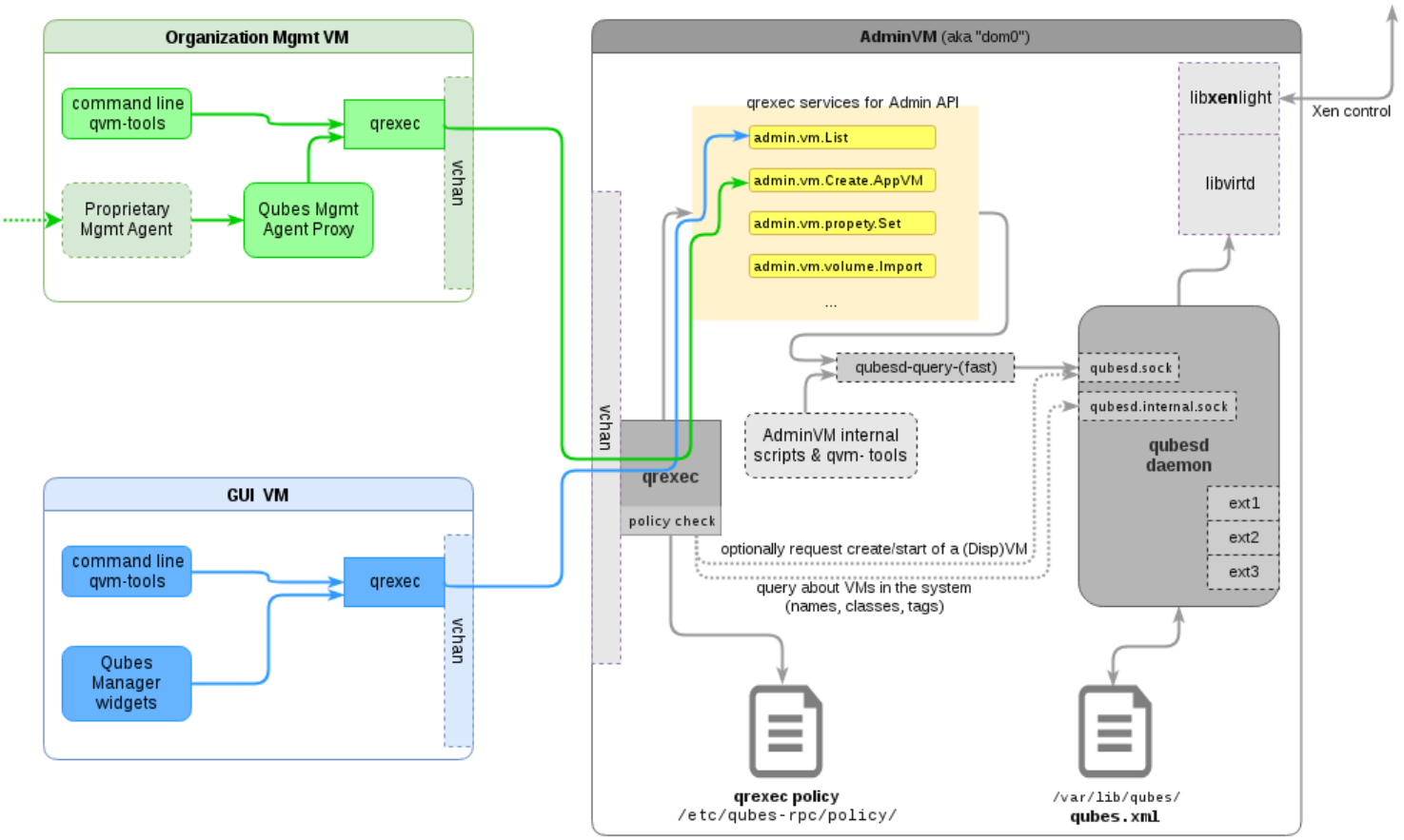
```
$tag:work $dispvm allow,target=$dispvm:work-printing
$anyvm    $dispvm:work-printing deny
```

The first line means that every DispVM created by a VM tagged with `work` will be based on (i.e., use as its template) the `work-printing` VM. (Recall that, in order for this to succeed, the `work-printing` VM also has to have its `template_for_dispvms` property set to true.) The second line means that any *other* VM (i.e., any VM not tagged with `work`) will be denied from creating DispVMs based on the `work-printing` VM.

`qubes.xml`, qubesd, and the Admin API

Since the beginning, Qubes Core Stack kept all the information about the Qubes system's (persistent) configuration within the `/var/lib/qubes/qubes.xml` file. This has included information such as what VMs are defined, on which templates they are based, how they are network-connected, etc. This file has never been intended to be edited by the user by hand (except for rare system troubleshooting). Instead, Qubes has provided lots of tools, such as `qvm-create`, `qvm-prefs` and `qubes-prefs`, and many more, which operate or make use of the information from this file.

In Qubes 4.x we've introduced the `qubesd` daemon (service) which is now the only entity which has direct access to the `qubes.xml` file, and which exposes a well-defined API to other tools. This API is used by a few internal tools running in dom0, such as some power management scripts, qrexec policy checker, and `qubesd-query(-fast)` wrapper, which in turn is used to expose most parts of this API to other VMs via the qrexec infrastructure. This is what we call Admin API, and which we I described in the previous post. While the mapping between Admin API and the internal qubesd API is nearly "one-to-one", the primary difference is that Admin API is subject to policy mechanism via qrexec, while the qubesd-exposed API is not policed, because it is only exposed locally within the `dom0`. This architecture is depicted below.



As discussed in the post about Admin API, we have put lots of thought into designing the API in such a way as to allow effective split between the user and admin roles. Security-wise this means that admins should be able to manage configurations and policies in the system, but not be able to access the user data (i.e. AppVMs' private images). Likewise it should be possible to prevent users from changing the policies of the system, while allowing them to use their data (but perhaps not export/leak them easily outside the system).

For completeness I'd like to mention that both qrexec and firewalling policies are *not* included in the central `qubes.xml` file, but rather in separate locations, i.e. in `/etc/qubes-rpc/policy/` and `/var/lib/qubes/<vmname>/firewall.xml` respectively. This allows for easy updating of the policy files, e.g. from within trusted RPMs that are installed in `dom0` and which might be brining new qrexec services, or from whatever tool used to create/manage firewalling policies.

Finally, a note that `qubesd` (as in "qubes-daemon") should not be confused with [qubes-db][https://github.com/QubesOS/qubes-core-qubesdb] (as in "qubes-database"). The latter is a Qubes-provided, security-optimized abstraction for exposing static informations from one VMs to others (mostly from AdminVM), and which is used e.g. for the agents in the VMs to get to know the VM's name, type and other configuration options.

The new attack surface?

With all these changes to the Qubes Core Stack, an important question comes to mind: how do all these changes affect the security of the system?

In an attempt to provide somewhat meaningful answer to that question, we should first observe that there exists a number of obvious configurations (including the default one) of the system, in which there should be no security regression compared to previous Qubes versions.

Indeed, by not allowing any other VM to access the Admin API (which is what the default qrexec policy for Admin API does), we essentially reduce the attack surface onto the Core Stack to what is has been in the previous versions (modulo potentially more complex policy parser, as discussed below).

Let us now imagine exposing some subset of the Admin API to select, trusted management VMs, such as the upcoming GUI domain (in Qubes 4.1). As long as we consider these select VMs as "trusted", again the situation does not seem to be any worse that what it was before (we can simply think of dom0 as having comprised these

additional VMs in previous versions of Qubes. Certainly there is no security benefit here, but likewise there is no added risk).

Now let's move a step further and relax our trustworthiness requirement for this, say, GUI domain. We will now consider it only "somewhat trustworthy". The whole promise of the new Admin API is that, with a reasonably policed Admin API (see also the previous post), even if this domain gets compromised, this will not result in full system compromise, and ideally only in some kind of a DoS where none of the user data will get compromised. Of course, in such a situation there is additional attack surface that should be taken into account, such as the qubesd-exposed interface. In case of a hypothetical bug in the implementation of the qubesd-exposed interface (which is heavily sanitized and also written in Python, but still) the attacker who compromised our "somewhat trustworthy" GUI domain might compromise the whole system. But then again, let's remember that without the Admin API we would not have a "somewhat trustworthy" GUI domain in the first place, and if we assume it was possible for the attacker to compromise this VM, then she would also be able to compromise `dom0` in earlier Qubes versions. That would be fatal without any additional preconditions (e.g. for a bug in qubesd).

Finally, we have the case of a "largely untrusted" management VM. The typical scenario could be a management VM "owned" by an organization/corporation. As explained in the previous post, the Admin API should allow us to grant such a VM authority over only a subset of VMs, specifically only those which it created, and not any other (through the convenient `created-by-XYZ` tags in the policy). Now, if we consider this VM to become compromised, e.g. as a result of the organization's proprietary management agents getting compromised somehow, then it becomes a very urging question to answer how buggy the qubesd-exposed interface might be. Again, on most (all?) other client system, such a situation would be fatal immediately (i.e. no additional attacks would be required after the attacker compromised the agent), while on Qubes this would only be the prelude for trying another attacks to get to `dom0`.

One other aspect which might not be immediately clear is the trade-off between a more flexible architecture, which e.g. allows to create mutually untrusted management VMs on the one hand, and the increased complexity of e.g. the policy checker, which is required to now also understand new keywords such as the previously introduced `$dispvm:xyz` or `$tag:xyz`. In general we believe that if we can introduce a significant new security improvement on architecture level, which allows to further decompose the TCB of the system, than it is worth it. This is because architecture-level security should always go first, before implementation-level security. Indeed, the latter can always be patched, and in many cases won't be critical (because e.g. smart architecture will keep it outside of the TCB), while the architecture very often cannot be so easily "fixed". In fact this is the prime reason why we have Qubes OS, i.e. because fixing of the monolithic architecture of the mainstream OSes has seemed hopeless to us.

Summary

The new Qubes Core Stack provides a very flexible framework for managing a compartmentalized desktop (client) oriented system. Compared to previous Qubes Core Stacks, it offers much more flexibility, which translates to ability to further decompose the system into more (largely) mutually untrusting parts.

Some readers might wonder how does the Qubes Core Stack actually compare to various popular cloud/server/virtualization management APIs, such as OpenStack/EC2 or even Docker?

While at first sight there might be quite a few similarities related to management of VMs or containers, the primary differentiating factor is that Qubes Core Stack has been designed and optimized to bring user one *desktop* system built on top of multiple isolated domains (currently implemented as Xen Virtual Machines, but in the future maybe on top of something else), rather than for management of service-oriented infrastructure, where the services are largely independent from each other and where the prime consideration is scalability.

The Qubes Core Stack is Xen- and virtualization-independent, and should be easily portable to any compartmentalization technology.

In the upcoming article we will take a look at the updated device and new volume management in Qubes 4.0.