Qubes OS Architecture

Version 0.3

January 2010

Joanna Rutkowska
Invisible Things Lab
joanna@invisiblethingslab.com

Rafal Wojtczuk
Invisible Things Lab
rafal@invisiblethingslab.com

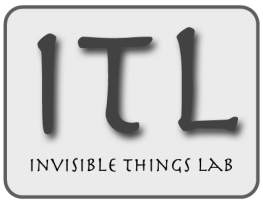

Table of Contents

1. Introduction

Project Qubes aims at building a secure operating system for desktop and laptop computers. The stress is on security, which is achieved by exploiting the isolation capabilities of the bare-metal hypervisor (Xen), together with modern hardware technologies, such as Intel VT-d and Trusted Execution Technology.

This document presents an architecture overview for the upcoming system. Some parts of this architecture will already be implemented by the time this document is made public, while others are still planned for future releases.

The first two introductory chapters of this document should be read by all users of the system. The following chapters dive into various technical details of the system architecture and are intended for system developers and advanced users of the system.

1.1. Problems with current Operating Systems

Current mainstream operating systems that are used for desktop computing, e.g. Windows, Mac OS X, or Linux-based systems, proved unsatisfactory when it comes to security. The major problem with current systems is their inability to provide effective isolation between various programs running on one machine. E.g. if the user's Web browser gets compromised (due to a bug exploited by a malicious web site), the OS is usually unable to protect other user's applications and data from also being compromised. Similarly, if certain system core components get compromised, e.g. WiFi driver or stack, none of the mentioned above systems can defend themselves from a complete compromise of all the user's applications and data.

The above situation is a direct result of certain architectural design decisions, which include over-complexity of the OS API, insecure GUI design, and, last but not least, the monolithic kernel architecture. It is the opinion of the authors that systems that are based on such insecure architecture simply cannot be made secure.

One can, of course, try to take the reactive approach, as many vendors do today, and try to patch every single known security bug. But such approach not only doesn't scale well, it simply doesn't work. Especially for users who require more than average security. Patching can only address known and popular attacks, but offers no protection against new, or less popular, more targeted, threats.

1.2. Why new OS?

Authors of this document do not believe that, at any time in the foreseeable future, we would be able to patch all the bugs in the software we use, as well as detect all the malicious software. Consequently we need a different approach to build secure systems.

Obviously building a new OS can be a very time consuming task. That's why, with Qubes OS, we reuse as much ready-to-use building blocks as possible, in particular the Xen hypervisor. This in turn implies the use of virtualization technology, which is used for two primary reasons: offering excellent security isolation properties, as well as the ability to reuse the large amount of software, including most of the applications and drivers written for mainstream OSes, like Linux or Windows.

1.3. How does virtualization enable security?

Virtualization allows to create isolated containers, the Virtual Machines (VM). VMs can be much better isolated between each other than standard processes in monolithic kernels of popular OSes like Windows or Linux. This is because the interface between the VM and the hypervisor can be much simpler than in case of a traditional OS, and also the hypervisor itself can be much simpler (e.g. hypervisors, unlike typical OS kernels, do not provide many services like filesystem, networking, etc). Additionally modern computers have hardware support for virtualization (e.g. Intel VT-x and VT-d technology), which allows for further simplification of the hypervisor code, as well as for creating more robust system configurations, e.g. using so called driver domains -- special hardware-isolated containers for hosting e.g. networking code that is normally prone to compromise.

1.4. Qubes security model: a practical example

The more systematic introduction into Qubes architecture is presented in the next chapter. Here, we present the practical view of the Qubes security model from the user's perspective.

Virtual Machines (VMs), also called *domains*, are the primary building blocks in the Qubes OS security model. The primary job of Qubes OS is to isolate VMs from each other, so that if one of the VMs gets compromised, the others are still safe to use. To see how this could improve the security of a typical user, let's consider a practical example with the following hypothetical VMs (domains) defined by the user:

1. The "**Random**" VM, which is to be used for all the usual Internet browsing, googling, news reading, watching You Tube funny movies, etc. Also the user would use this Random VM to install some more or less random applications that perhaps the user cannot fully trust, but which might be fun enough to try, e.g. some games. What defines the Random VM is that there is no sensitive data there at all, so the potential compromise of this VM doesn't present any security problem for the user. If the machine starts misbehaving (because perhaps the Web browser running there got compromised when the user was browsing the Web) the user can simply restart it, and Qubes would make sure to revert it to the known good state.

2. The "**Social**" VM, which the user might decide to use e.g. to host an email client for his or her personal email account (i.e. not a corporate one), and also for updating the user's private blog, twitter, facebook, and god-knows-what-other Web social service. The common denominator of all the things that the user does in this VM is that they are all connected with the user's social identity online (email, blog, twitter, etc), and that it is this identity that is the most sensitive resource to be protected in this particular VM. Specifically, if any of the applications running in this VM gets compromised, this would likely lead to compromise of the user's social identity on the Web. But, if we think about, this would happen, even if the user kept all the applications, used for different social services, each in a different VM. E.g. if the attacker gained control over the user's email, it's likely that the attacker could easily gain control over e.g. the user twitter account, etc. Consequently, because all those applications have access to the same sensitive resources, in that case the user's social identity, there is little point into further separating them from each other, and is thus reasonable to host them all in the same VM.

3. The "**Shopping**" VM, which would be used for all the Internet shopping, e.g. Amazon, eBay, etc. The common sensitive resource in this case is the user's credit card number (and perhaps the user's address, phone and name). If this machine gets compromised, this would likely happen because one of the shopping sites turned out to be malicious and was hosting a driver-by exploit. This might be because the owner of the website is a criminal, or perhaps because a legitimate site got compromised (due to server side bug) and the attacker modified the website so that it now try to infects all the visitors. It is however unimportant. What is important, is that if the user credit card got stolen by one of the shopping websites it's already compromised, and the fact that another website might also steal it later, is really irrelevant. So, there is no point in further separating e.g. the Amazon shopping activities from the shopping done in some little local department store if the user uses the same credit card to pay in both stores. Perhaps a more paranoid user might use two different credit cards for different types of shopping: one card with very low limit for all the usual common things, like ordering food, buying books, etc, and the other one with very high limit, e.g. for buying all the latest computer hardware. In that case it would make sense for the user to have two different shopping VMs, e.g. "Shopping-low", and "Shopping-high". In most civilized countries, the widespread credit card transaction insurance would not make it worth the effort though.

4. The "**Bank**" VM, which the user might exclusively use only for accessing his or her bank website, making wires, etc. Such machine might be allowed only HTTPS access to the bank website, and nothing except it.

5. The "**Corporate**" VM, which the user might use only to host the corporate email client, as well as for accessing the corporate intranet. So, this machine would have a VPN client and would only allow VPN connections to the corporate VPN gateway. This is also where all the user's job-related files, like reports, spreadsheets, databases will be kept and used (so to work on a report the user would use a word processor application running in this "Corporate" VM).

   Of course, it might happen that this corporate VM gets compromised, e.g. because the colleague from work, working on the same project, sends the user an infected PDF document, that contains an exploit for the PDF viewer that is used by the user. Qubes would not stop such a hypothetical exploit from compromising the VM (but will stop it from compromising other VMs). On the other hand, it's very likely the

secrets from the user's corporate VM would get compromised anyway, because the colleague has a compromised machine, so if they work in the same team, on the same project, all the project-related documents would get compromised anyway.

A user that interacts with more than one team (with different clearance levels), or different departments, would likely need to define more than one VMs, e.g. "Corporate-research", "Corporate-accounting", "Corporate-customers", etc.

Some other users might work as consultants, and e.g. on two different projects for two different companies. The user might then want to create two different VMs: "Corporate-A" and "Corporate-B", to fully isolate those two job engagements from each other.

The above is just an example of how a user might divide tasks and resources into several VMs (security domains). Other users might choose a different scheme. E.g. government users might instead prefer to use the VMs in a way it follows the well known information classification systems used by many governments e.g.: "Non Restricted", "Restricted", "Confidential", "Secret", and "Top Secret".

Additionally there are some default system VMs that are provided automatically. One example is the "**Network**" VM, which isolates all the world-facing networking code into an unprivileged VM. This protects the system when using the network in a potentially hostile environment, such as a hotel or an airport. Even if an attacker exploits a potential bug, e.g. in the WiFi driver, this doesn't allow to compromise the system. At worst, the user would simply lose network connection, but the user's VMs and data should be safe.

Qubes makes it easy for the user to define and manage many VMs, so that the user doesn't practically need to be aware that all of the applications are executing in various VMs. Qubes provides a secure GUI subsystem, which makes it even simpler for the user to use and manage all those VMs. Qubes GUI makes it possible to display various applications, even if they are hosted in different VMs, on the same common desktop (e.g. an email client from the "Social" VM next to the Web browser from the "Random" VM), and yet this doesn't weaken the isolation between the VMs. The user also has the ability to copy and paste and exchange files between VMs.

Secure GUI is a key element to make Qubes useful in everyday desktop computing. One can simply come up with many examples. For instance the user receives an email from a friend with a HTTP link to "cool new movie". The user copies the link from the "Social" VM, where the email client runs, to the Web browser running in "Random" VM, and can safely see the "cool new video", regardless of whether the website hosting them also doubles as a malware infector or not.

2. System architecture overview

The figure below shows the Qubes OS architecture from 30,000 feet. Virtual Machines (VMs) are the primarily building blocks for the system. Qubes architecture optimizes disk (and in the future also memory) usage, so that it's possible to run many VMs in the system, without wasting precious disk resources. E.g. file system sharing mechanisms allow to reuse most of the filesystem between VMs, without degrading security isolation properties. Intel VT-d and TXT technology allows for creating safe driver domains, that minimize system attack surface.

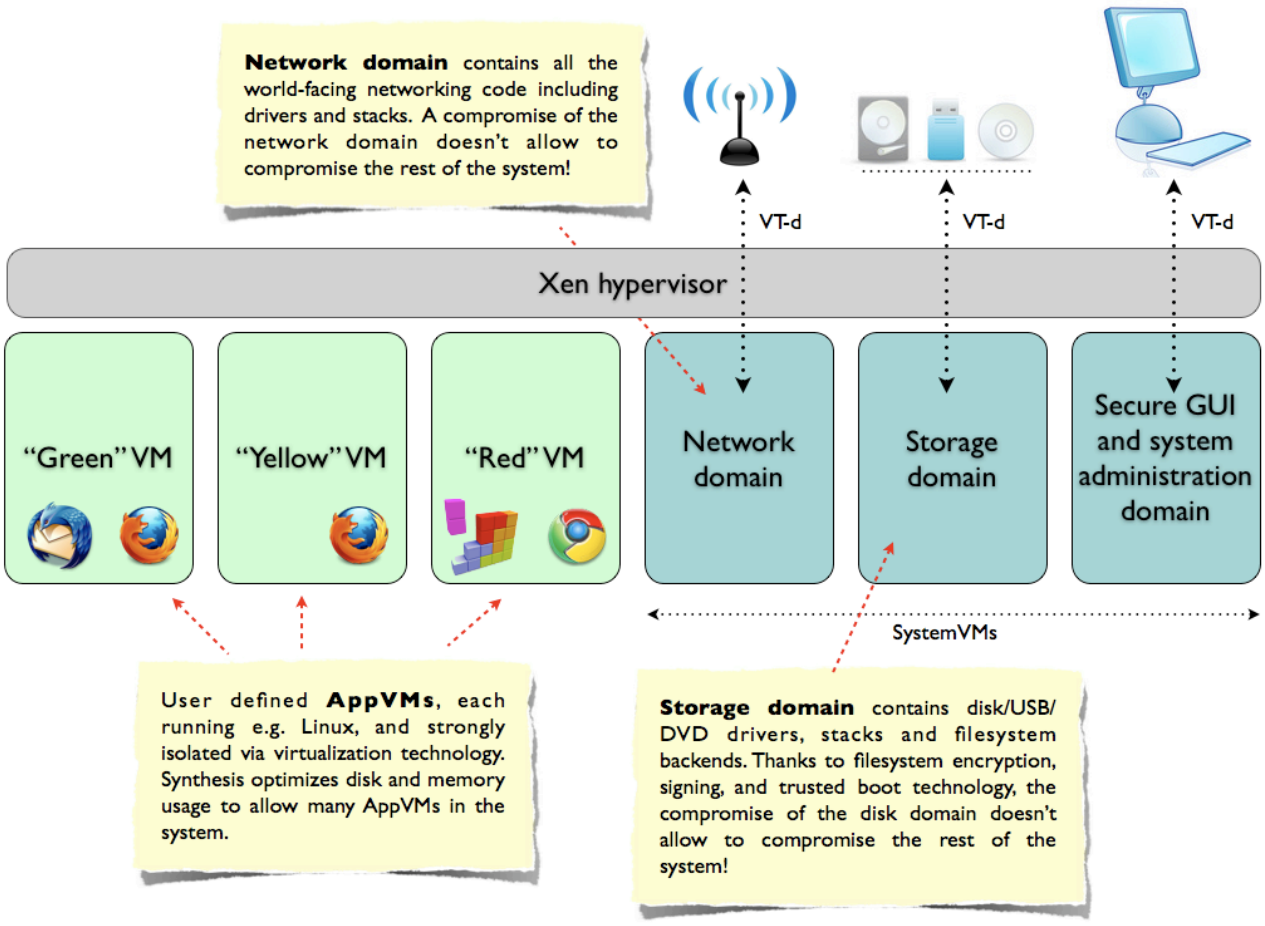

Figure 1. Qubes architecture overview.

One can divide the VMs used by the system into two broad categories: the AppVMs, that are used to host various user applications, such as email clients, web browsers, etc, and the SystemVMs (or ServiceVMs) that are special in that they are used to provide system-wide services, e.g. networking or disk storage.

Below we describe the functions and basic characteristics of each type of VM. For more detailed technical discussion the reader might want to consult further chapters in this document.

2.1. Common desktop and secure GUI

One of the main goals of Qubes OS is to provide seamless integration of the various applications hosted in different VMs onto a common user desktop, so that it is easy for the not-technical user to work with all the applications, regardless of what VMs they are hosted in. Specifically, we assume that all the user applications, except for X Window Manager, and management applications, would be hosted in AppVMs.

Another important goal is to provide a near-native performance, especially for displaying applications that use rich graphics and multimedia, like e.g. Web browsers. It's a natural requirement that the user should e.g. be able to comfortably watch movies in applications running in AppVMs.

Detailed discussion and description of of the secure GUI subsystem implementation is described later in this document.

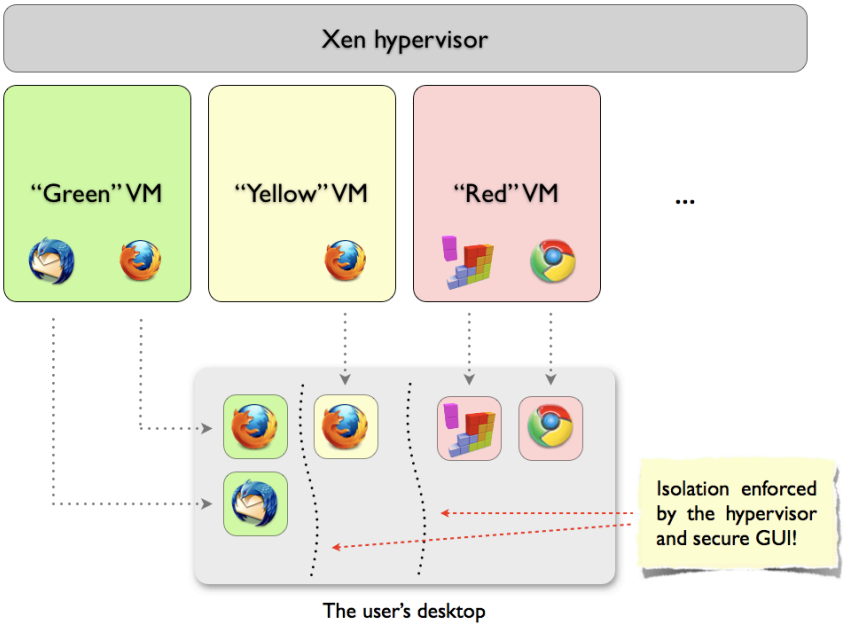

Figure 2. Qubes common desktop concept.

2.2. The AppVMs

AppVMs are the virtual machines used for hosting user applications. For the first releases of Qubes we assume all AppVMs to run Linux OS. But there is nothing that would prevent extending Qubes OS to support other OS in AppVMs, e.g. Windows-based.

Qubes makes special effort to save on the disk space used by the virtual machines. E.g. all the AppVMs based on the same system distribution (currently Linux) should share the same read-only file system, and only use separate disk storage for user's directory and per-VM settings. This allows to save tens of gigabytes of disk space, which otherwise would be wasted to replicate full OS image for each VM.

We realize that such a design introduces some complexity to the filesystem backend and creates potential points of attacks. To address this problem, Qubes architecture introduces dedicated storage domain, that sandboxes all the disk and file system storage code, so that even in case it got compromised, no harm can be done to the rest of the system.

2.3. The network domain

In a typical operating system like Window or Linux, the core networking code, that includes network card drivers and various protocol stacks (802.11, TCP/IP, etc), runs in the kernel. This means that a potential bug in the networking code, when exploited by an attacker, always results in full system compromise.

In the Qubes architecture we try to limit potential attack vectors on the system as much as possible. That's why the whole networking code, including drivers and protocol stacks, has been moved out to an unprivileged VM, called the Network Domain.

The network domain has direct access to the networking hardware, i.e. the WiFi and ethernet cards. The access is granted via Intel VT-d technology, that allows for safe device assignment to unprivileged code.

When a potential bug in the networking code, e.g. in the WiFi driver, is exploited by an attacker, all that the attacker can gain is the control over the network domain, but not on any other VM. This doesn't give the attacker any advantage comparing to what he or she can already do when controlling e.g. the hotel LAN or WiFi network (which assumes all the usual networking attacks like sniffing of unencrypted traffic, or performing man-in-the-middle attacks). Particularly, it's expected that all security-sensitive applications (that are hosted in other domains) use some form of cryptography-protected network protocols, such as SSL, VPN, or SSH.

2.4. The storage domain

As mentioned earlier, Qubes tries to optimize disk usage by virtual machines. The sophisticated file sharing mechanisms (e.g. copy-on-write filesystem) require complex code on the backend side. Having such a code in the administrative domain (Dom0) would increase the attack surface on the system. To mitigate this risk, the filesystem sharing code, the backends, together with disk drivers and stack, are moved to dedicated unprivileged VM, the so called storage domain. The storage domain has direct access to the disk controller, granted via Intel VT-d technology, similarly as the network domain has access to the networking hardware. Additionally, the storage domain has access to USB and CD/DVD controllers to allow it to handle all the removable storage devices, such as USB flash drivers or CD/DVD discs. This also allows to mitigate attacks originating from removable devices, e.g. specially formatted USB drive that tries to exploit a potential bug in the USB driver.

The biggest challenge with storage domain is how to make it security non-critical. In Qubes architecture we solve this problem by using cryptography to protect the filesystems, so that the storage domain cannot read confidential data owned by other domains, nor it could modify (in a meaningful way) the shared root filesystems. Additionally we make use of Intel Trusted Execution Technology (TXT) in order to prevent modification of the system boot code. This means that the storage domain, if compromised by the attacker, can, in the worst case, make the system, or select AppVMs, unbootable but cannot compromise the system, e.g. by installing backdoors or rootkits and steal the user data.

The more in-depth description of the storage domain architecture is provided later in the document.

2.5. The GUI/administrative domain (Dom0)

The GUI domain is the one that has direct access to the graphics device, as well as input devices, such as keyboard and mouse. The GUI domain runs the X server which displays the user desktop, and the Window Manager that allows to start and stop the applications and manipulate their windows. The most common application that runs in the GUI domain is the Application Viewer, a little stub application that is used to launch and then display the content of the actual application hosted in some AppVM. AppViewer provides an illusion for the user that the application executes natively on the desktop, while in fact it is hosted (and isolated) in an AppVM.

The GUI domain is security critical, because if one controls this domain, one can mimic any action that the legitimate user could also perform, including starting and operating all the applications in all AppVMs.

Because the GUI domain is security-sensitive, there is no world-facing code running in this domain. E.g. there is no networking code in the GUI domain. Also it has a very simple and slim interface used to communicate with other domains (in order to display windows of application running in AppVMs), to minimize the possibility of an attack originating from one of the AppVMs.

Also, because the GUI domain is so security-sensitive, it's unclear whether there is any benefit of hosting the GUI subsystem outside of Dom0. In the first releases of Qubes OS we assume the GUI domain to be the same as the administrative domain (Dom0). This might change in the future. See also the detailed discussion about this problem in the chapter dedicated to the GUI.

3. The hypervisor and the administrative domain

3.1. The role of the hypervisor

The hypervisor is the single most security critical element in the system. It is the hypervisor that provides isolation between different virtual machines.

A single bug in the hypervisor can result in full system compromise. It's not possible to sandbox the hypervisor and thus, special care should be paid when choosing the best hypervisor for Qubes OS.

There are currently two popular, open source hypervisors actively being developed: the *Xen hypervisor*[1] and the *Linux Kernel-based Virtual Machine* (KVM)[2]. Below we discuss why we believe that the Xen hypervisor architecture is better suited for Qubes OS.

3.2. Xen vs. KVM security architecture comparison

Most Xen advocates would dismiss KVM immediately by saying that it's not a true bare-metal hypervisor, but rather more of a type II hypervisor added on top of (fat and ugly) Linux kernel. The KVM fans would argue that while Xen itself might indeed be much smaller than Linux with KVM, however that it's not a fair comparison, because one needs to also count Xen's Dom0 code as belonging to TCB, and that Dom0 uses a pretty much regular Linux system, which in the end comes down to pretty comparable code bases we need to trust (Linux + KVM vs. Xen + Linux).

Both of those arguments are oversimplifications, however.

The thin vs. fat hypervisor argument

In KVM architecture each VM is just another type of a Linux usermode process. The exception being that such a "VM process" doesn't have access to the standard Linux system call interface, but instead can interact with the kernel via VMX or SMX intercepts. In that case the kernel actually becomes the hypervisor for all the VM processes. One should note, however, that the VM↔hypervisor interface in that case is much simpler than in case of a regular process↔kernel interface.

However it's not entirely true. Particularly the hypervisor still uses (or is free to use) the whole Linux kernel infrastructure, with all its drivers and internal interfaces. This makes the line between what code in the Linux kernel is, and what is not, used for handling various VM-generated events, to be blurry. This is not the case when we consider a true bare-metal hypervisor like Xen. In Xen, at no point does the execution path jump out of the hypervisor to e.g. Dom0[3]. Everything is contained within the hypervisor. Consequently it's easier to perform the careful security code audit of the Xen hypervisor, as it's clear which code really belongs to the hypervisor.

At the same time the above argument cannot be automatically transfered to Xen's Dom0 code. The main reason is that it is possible to move all the drivers and driver backends out of Dom0[4]. The same is true for moving the IO Device Emulator (ioemu) out of Dom0[5].

The only element (that is accessible to VMs) that must be left in Dom0 is the XenStore daemon, that is responsible for managing a directory of system-wide parameters (e.g. where is each of the backend driver located). That represents a very minimal amount of code that needs to be reviewed.

---

[1] http://xen.org

[2] http://linux-kvm.org

[3] Of course there are hypercalls that result in triggering events in Dom0, so code execution in Dom0, but that is clear from their definition, e.g. hypercalls used to deliver events to guests.

[4] In general this is a non-trivial task. E.g. special care should be paid when creating a disk driver domain. See the discussion later in this document on how to create a truly non-security-critical disk driver domain.

[5] If one doesn't need to run HVM guests, e.g. Windows, and runs only PV guests in Xen, then it's possible to get rid of the I/O Emulator entirely. Otherwise, one can use the "stub-domain" mechanism introduced in Xen 3.3, that allows to host the I/O emulator in a special VM dedicated to each HVM machine.

The I/O Emulation vs. PV drivers

KVM uses full virtualization approach for all its virtual machines. This means that every virtual machine in KVM must have an associated I/O emulator. KVM uses the open source *qemu* emulator for this purpose. In KVM architecture the I/O emulator is running on the host, although as an unprivileged non-root process.

Xen, on the other hand, allows for both fully virtualized, as well as para-virtualized virtual machines. This offers an option of either removing the I/O emulator completely from the system (in case the user wants to use only PV guests, as they don't need an I/O emulation), or to host the I/O emulators in dedicated minimal PV domains. Xen provides even a dedicated mechanism for this, the so called "stub-domains".

The I/O emulator is a complex piece of software, and thus it is reasonable to assume that it contains bugs and that it can be exploited by the attacker. In fact both Xen and KVM assume that the I/O emulator can be compromised and they both try to protect the rest of the system from a potentially compromised I/O emulator. They differ, however, in the way the try to protect the system from a compromised I/O emulator.

KVM uses standard Linux security mechanisms to isolate and contain the I/O emulator process, such as address space isolation and standard ACL mechanisms. Those can be further extended by using e.g. SELinux sandboxing. This means that the isolation quality that KVM provides cannot be significantly better than what a regular Linux kernel can provide to isolate usermode processes[6].

Xen uses virtualization for the I/O emulator isolation, specifically para virtualization, just in the very same way as it is used for regular VM isolation, and doesn't relay on Linux to provide any isolation.

Driver domains support

A very essential feature for the Qubes OS is the ability to sandbox various drivers, so that even in the case of a bug in a driver, the system could be protected against compromise. One example here could be a buggy WiFi driver or 802.11 stack implementation that could be exploited by an attacker operating at an airport lounge or in a hotel. Another example could be a bug in the disk driver or virtual filesystem backend, that could be exploited by the attacker from one of the (lesser-privileged) VM in order to compromise other (more-privileged) VMs.

In other to mitigate such situations, Qubes architecture assumes existence of several so called driver domains. A driver domain is an unprivileged PV-domain that has been securely granted access to certain PCI device (e.g. the network card or disk controller) using Intel VT-d. This means that e.g. all the networking code (WiFi drivers, stack, TCP/IP stack, DHCP client) is located in an unprivileged domain, rather than in Dom0. This brings huge security advantages -- see the specific discussions about network domain and disk domain later in this document.

KVM, on the other hand, doesn't have support for driver domains. One could argue it would be possible to add a support for driver domains (that is for hosting PV driver backends in unprivileged domains) using Linux shared memory, because KVM allows to use VT-d for secure device assignment to unprivileged VMs, but that would probably require substantial coding work. More importantly, because KVM doesn't support PV domains, each driver domain would still need to make use of the I/O emulator running on the host. As explained in the previous paragraph, this would diminish the isolation strength of a driver domain on a KVM system.

Summary

We believe that the Xen hypervisor architecture better suits the needs of our project. Xen hypervisor is very small comparing to Linux kernel, which makes it substantially easier to audit for security problems. Xen allows to move most of the "world-facing" code out of Dom0, including the I/O emulator, networking code and many drivers, leaving very slim interface between other VMs and Dom0. Xen's support for driver domain is crucial in Qubes OS architecture.

KVM relies on the Linux kernel to provide isolation, e.g. for the I/O emulator process, which we believe is not as secure as Xen's isolation based on virtualization enforced by thin hypervisor. KVM also doesn't support driver domains.

[6] Assuming the I/O emulator is subject to compromise, which is a reasonable, and backed up in its history, assumption.

3.3. Securing the hypervisor

Formal security proofs?

It would be nice if it was possible to formally prove correctness of the hypervisor code. Unfortunately this doesn't seem to be feasible for software that interacts with commodity PC hardware (x86 architecture). Particularly in order to prove correctness of the hypervisor, it seems like one would first need to have complete models of all the hardware that the hypervisor can interact with, including e.g. the CPU and the chipset (MCH), which seems highly nontrivial to create.

For instance, the recently published paper on formally proving the correctness of well known seL4 microkernel[7], explicitly states that only the ARM port (and only for non-SMP systems) has been proven, while the x86 port is still left to be formally verified in the future. More over, the paper states that e.g. the memory management has been pushed of out of the microkernel to the usermode process, and that this process has not be contained within the proof. They admit, however, that this process is part of the TCB, and so eventually would have to also be proved. One can easily imagine that in case of a general purpose OS, based on such a microkernel, there would be more usermode processes that would also be part of the TCB and hence would have to be formally proved, e.g. the filesystem server.

In case of the x86 hardware, one would also assure that the drivers cannot program devices to issue malicious DMA transactions that could potentially compromise the microkernel or other parts of the system. This would require the microkernel to support programming of the IOMMU/VT-d, as otherwise one would need to prove correctness of every single driver, which is unfeasible in reality. However, addition of IOMMU/VT-d support to the seL4 microkernel might likely result in rendering the formal proving much more difficult.

Additionally certain peculiarities of the x86 platform, like the SMM mode, might also be difficult to account for[8].

Thus, we don't expect any bare-metal hypervisor or microkernel for commodity x86 PC hardware to be formally proved secure anytime in the near future. Consequently other, more pragmatic, approaches are needed in order to secure the hypervisor. The best approach we can think of is the manual audit of the hypervisor code and to keep the hypervisor as small and simple as possible. Additionally one might apply some anti-exploitation mechanisms, like e.g. non-executable memory.

In the future, when formally proved hypervisors become a reality, there is no reason why Qubes should not switch to such a hypervisor. Particularly, the rest of the architecture proposed in this document (secure GU, secure storage domain, etc) would still be required even if the system used formally verified hypervisor.

Reducing hypervisor footprint

In order to reduce the hypervisor footprint, one might consider to remove certain features from the hypervisor. We expect that Qubes OS might be distributed with different versions of the hypervisor, depending on the specific needs of different user groups. E.g. the government users might be interested in using the most secure version of the hypervisor with minimal amount of extra features (e.g. with no HVM support, no ACPI support), while other users with not so strict security constrains might want to use a fully-featured hypervisor, even though its code footprint might be twice as big, and consequently the risk of a hidden security bug higher[9].

Anti-exploitation mechanisms in the hypervisor

Currently Xen doesn't make use of any well known anti-exploitation techniques, like Non-Executable memory (NX) or Address Space Layout Randomization (ASLR).

Adding proper NX markings on all the pages that do not contain code is usually an obvious first step in making potential bugs exploitation harder. Particularly the combination of NX and ASLR is used most often, because NX protection alone can easily be circumvented using the so called return-into-lib exploitation technique, where the attacker jumps into the code snippets that are already present (as they are parts of the legal code) in the address space of the target being exploited.

---

[7] http://ertos.nicta.com.au/publications/papers/Klein_EHACDEEKNSTW_09.pdf

[8] A buggy SMM handler might e.g. overwrite the microkernel memory, perhaps as a result of some malicious triggering actions by some usermode process.

[9] Although it's currently not clear if e.g. removing the HVM code from Xen really reduces the effective footprint, understood as the code reachable from the gust, if the user only uses the PV guests. To Be Determined.

However, in case of Xen, the potential benefits of using NX marking are questionable. This is because IA32 architecture, as implemented on modern Intel and AMD processors, allows the CPU that executes in ring0 to jump and execute code kept on usermode pages. So the attacker can always keep the shellcode in the usermode, in this case, e.g. in the VM's kernel or process, and can bypass all the NX protections implemented in the Xen hypervisor. The only solution to this problem would be to modify the IA32 architecture so that it would be possible to disable this mode of operation (e.g. via some MSR register).

ASLR does make sense though. Particularly, one might modify all the memory allocation functions and also attempt to make the Xen code relocatable, so that each time the Xen is load it gets loaded at a different address. On the other hand such changes might be non-trivial, and perhaps might introduce some more complexity to the hypervisor. Further research is needed to decide if addition of any anti-exploitation mechanisms is worth the effort.

Reducing Inter-VM covert channels

User with very high security requirement might not only be concerned about the "classic" security bugs (which lead to system compromises, e.g. installing malware), but also about more subtle security problems related to inter-VM covert channels.

We can divide possible covert channels in a system like Qubes into two broad categories:

1. Covert channels that require cooperation on both ends, e.g. one (malicious) application running in a more privileged VM sniffs data and "leaks" them via a covert channel to another (malicious) application running in some other VM, that later sends them out through the network.
2. Covert channels that don't require cooperation on both ends, e.g. a (non-compromised) process in one VM generates a cryptographic key, while another (malicious) process, running in a different VM, sniffs this key, e.g. by observing the CPU cache characteristics.

Out of the two classes mentioned above, the second class is of a much bigger concern in Qubes OS.

The cooperative covert channels (first class mentioned above) are not so much of a concern, because, if we assume that the attacker managed to insert a malicious application into one of the user more privileged VMs, it means that the attacker already broke Qubes isolation mechanism (and there are many other, simpler, ways to steal sensitive data from the VM), or, more likely, that the software in the VM got somehow exploited. E.g. the corporate email client could be exploited while connecting to a corporate server (which was itself exploited and modified to serve exploits to visitors). But this means that the attacker can leak the information from the infected VM using much simpler ways than via covert channels to other VMs. Most likely the attacker would simply send the information back over the same connection from which the attack came.

The second class of inter-VM covert channels presents, however, a real security concern.

The discussion about potential mitigation for the above classes of inter-VM covert channels is beyond the scope of this document. In general this might be a very non-trivial task, especially given a commodity PC hardware. We expect that in order to eliminate certain classes of inter-vm covert channels, special restrictions would have to be applied, e.g. two different VMs would be forbidden from being scheduled to execute on two different cores of the same physical processor, as otherwise there the shared CPU cache might be exploited to leak some information from one VM to another, etc.

This again, we expect, will come down to allowing the user to choose the appropriate version of the hypervisor. The users with highest security requirements would install the most restrictive hypervisor, sacrificing perhaps their system performance, in return for better covert channels prevention.

In the end, there will probably be just two versions of the hypervisor available:

- the general purpose hypervisor based on mainstream Xen, offering wide range of features, such as support for different AppVMs (e.g. Linux, Windows, etc), sophisticated power management (ACPI support), and good performance,
- and the ultra-secure hypervisor for the most security demanding users. This hypervisor would again be based on Xen, but with most of the features that are absolutely not necessary removed (e.g. support only for PV Linux), and with additional restrictions to prevent potential inter-VM communication.

3.4. The administrative domain (Dom0)

The administrative domain, also referred to as Dom0, is almost as privileged as the hypervisor[10]. Consequently we took special care to limit the amount of world-facing and VM-facing code running in Dom0. In fact there is no world-facing code in Dom0 at all (no networking) and the amount of VM-facing is kept at an absolute minimum (see below), drastically minimizing likelihood of an attack reachable from other VMs.

Dom0 hosts the so called XenStore daemon, which is a system-wide registry used for system management, which contains e.g. information where various backend devices are located.

Dom0 also hosts the GUI subsystem. While this might seem like lots of code, the interface between other VMs and the GUI daemon is extremely slim. The reason to host the GUI subsystem in Dom0, and not in a separate domain, comes from the fact that the attacker who compromised the GUI subsystem can effectively compromise all the user's VM. This makes the GUI subsystem, by the very definition, almost as privileged as Dom0[11].

Besides the Xen Store and GUI daemon, there is no other code in Dom0 that could be interacted with from other VMs. This particularly means that there are no backend drivers in Dom0 -- those have been moved out to unprivileged VMs: the network domain and the storage domain.

So, even though Dom0 runs a regular Linux OS (with a kernel that supports Dom0 pvops), the interface to Dom0 is very minimal (just GUI and Xen Store), and hence the potential attack surface (and the amount of code to audit) if also very minimal.

Additionally, because in the Qubes architecture the hypervisor set strict VT-d protections, Dom0 (as well as any other AppVM) gets automatic DMA protection from physical attacks originating from bus mastering devices, like e.g. FireWire or CardBus/PCMCIA devices. This is very important for the laptop users, who otherwise might be subject to a trivial physical attacks[12].

Dom0 has access to all secrets used for block device (filesystem) encryption and signing (see the chapter about Storage domain for more details). Those secrets are released to Dom0 by the TPM, only if the correct (i.e. non-tampered) hypervisor image has been booted, and only if Dom0 kernel and filesystem has been non-tampered with. Otherwise Dom0 will not have access to VM's private date.

3.5. Power management and ACPI support

ACPI security concerns

Every modern operating system contains support for power management -- this includes putting and resuming the system to and from various sleep states, and also on-the-fly adjustment of e.g. the CPU frequency, or various fans speeds, in order to maintain proper power consumption and to prevent over-heating of the system.

The power management subsystem is usually placed in the OS kernel. In case of Qubes that would be either the hypervisor, or Dom0, or both. The rationale for such a placement is that the power management code should have a full glimpse over the whole system, and also because it should be allowed I/O interactions with many devices.

ACPI is a widely adopted standard for power management interface on PC platform. However, ACPI, due to its complexity and design, might also cause potential security problems for the platform.

Particularly dangerous is the ACPI's interpreted language, ACPI Machine Language, or AML, that is used to describe how certain power-management operations should be accomplished by the underlying OS.

---

[10] Dom0 cannot modify the hypervisor memory, while the hypervisor can modify Dom0 memory. Still, Dom0 can access and modify all the memory of all the other VMs in the system.

[11] In the chapter dedicated to GUI subsystem, there is a detailed discussion about potential benefits of having the GUI subsystem hosted in a separate domain, and why those benefits are, in the authors opinion, not worth the effort.

[12] E.g. in case of most mainstream OSes, which do not use VT-d, it is possible to prepare a specially programmed PCMCIA card that, when inserted into the laptop, can automatically compromise the machine, e.g. by installing a rootkit or backdoor. In that case any laptop left unattended for just a few minutes is subject to such an attack.

It has been demonstrated first by John Heasman[13], and later by Loic Duflot[14]. Malware that is capable of re-flashing the BIOS, or gain control early in the boot process, e.g. as a result of MBR infection, can replace the original ACPI tables in memory with malicious ones, so that when later the OS will be interpreting (executing) the AML methods, the attacker's code will compromise the OS.

Such an attack could thus theoretically be used by an attacker who gained control over the storage domain (see later chapter for details about storage domain implementation). The attacker could then modify the boot loader, e.g. the MBR or GRUB, so that the code modify the ACPI tables that are later passed on to Xen and Dom0 kernel. Xen doesn't interpret ACPI AML methods, but Dom0 kernel does. Thus the attacker can thus potentially use this attack to compromise Dom0, despite Intel TXT being used to assure trusted boot (see later for trusted boot description in Qubes)[15].

Preventing ACPI abusing attacks

Even though the attacker would need to first gain control over the storage domain in order to proceed with this attack, in Qubes architecture we would like to assume that the storage domain compromise is security non-critical, so it's important to consider workarounds to prevent this attack.

1. **Disable ACPI processing in Dom0.** This is the most straightforward solution, but might not be suitable for most users, especially for laptop users. After all power management is an important feature of modern platforms.
2. **Verify ACPI methods.** Theoretically it might be possible to somehow verify if the particular ACPI method is not malicious before executing it, e.g. if it only access the power-management related I/O registers. Creating such an ACPI verifier might however be non-trivial and we are not aware of any research in this area.
3. **Measure ACPI methods as part of the TXT launch.** The TXT loader (e.g. `tboot.gz`) can be modified so that it first copies the platform ACPI tables onto the TXT heap [16], and later measures them (with the results places in e.g. PCR18 register). This would prevent any modifications of the ACPI tables after the system initial installation (we naturally assume that the platform's original ACPI tables were not malicious). However, as pointed out by Loic Duflot[17], effective measuring of ACPI tables might be non-trivial, because of their highly nested nature (one table pointed out by a pointer in another table, etc).
4. **Provide good known ACPI tables.** Instead of measuring the ACPI tables, we might copy the original ACPI tables (that we observed during the installation of the system, again we assume the system is not-compromised at this stage), and then use this "known good copy" of the ACPI table on each system boot. This would require slight modifications to the Dom0 starting scripts (possibly also to the kernel), but seems rather straightforward.

We believe the last solution (providing good known ACPI tables) is the simplest and most effective to implement, and thus is recommended as a solution to the ACPI attacks problem.

[13] Implementing and Detecting an ACPI BIOS rootkit, http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Heasman.pdf

[14] ACPI Design Principles and Concerns, http://www.ssi.gouv.fr/IMG/pdf/article_acpi.pdf

[15] One should note that, strictly speaking, this is not an attack against TXT -- rather it's an attack on one of the entities loaded via TXT (Dom0 in that case) that chooses to trust unverified input (ACPI tabled provided by BIOS), that turns out to be malicious.

[16] This is already being done by tboot for ACPI DMAR tables.

[17] In private conversation.

4. The AppVMs

The AppVMs are the VMs used to host user's applications. We assume that all of the user's applications run in AppVMs and none of them run in Dom0. The only user-visible applications running in Dom0 are the Window Manager, the AppViewers (used to interact with the applications hosted in various AppVMs) and some system management applications, like those monitoring the state of all VMs, memory utilization, etc.

All the user files are kept in one or more of the AppVMs. E.g. all the user email will be kept in the AppVM where the user runs the email client (perhaps the user might want to have different email clients in different AppVMs, one for personal email, the other for corporate email, and in that case the emails are kept in the corresponding AppVMs). The user is offered, of course, an ability to move the files from one AppVM to another one, as well as the ability to store them on external storage devices like USB flash drives.

It is assumed that most of the AppVMs would be based on the same Linux distribution, which makes it possible share the root file system among them all (in a read-only manner, of course).

Qubes provides a special stub-application called AppViewer, that is used to bring all the user applications from various AppVMs onto a common desktop in Dom0 and allows the user to interact with them just as if the applications were natively running there, and not in other VMs. For this to work, each AppVM must run a special agent called QubesWM, that can be thought of as of a dedicated Window Manager that the AppViewers interact with. The AppViewer and QubesWM are described in more detail in the next chapter devoted to secure GUI implementation.

4.1. Sharing the Root FS among VMs

It is expected that the user will use many AppVMs: at least three in case of most users, and probably over 10 in case of more security paranoid users. It's thus very important to use an effective scheme for sharing the common root filesystem among the AppVMs in a secure way.

In Qubes architecture we achieve this by dividing the file system used by each AppVM into two parts:

1. The root file system, that we want to share among all the AppVMs (e.g. `/boot`, `/bin`, `/usr`)
2. The private data that are VM-specific (e.g. `/home`, `/usr/local`, and `/var`).

All the files belonging to the first group are shared among all the AppVMs via a read-only block device that, when mounted by each AppVM provides the root filesystem directories. Most of the Linux distributions would, however fail to use a root filesystem that is backed by a read-only device. To solve this problem, we use device mapper (dm), an integral part of any Linux 2.6 system, to create a copy-on-write (COW) device backed by the previously mentioned read-only block device (with root filesystem) and by another block device, the COW block device, that is writable by each AppVM and that is backed up by a separate per-VM file in the storage domain (the per-VM COW file). The scripts for creating this RW device with root filesystem executes in the AppVM's initrd script.

Additionally, the VM-specific private data are kept on a per-VM block device that is backed up by a per-VM image file in the storage domain (this device is read-write, of course).

As it will be explained in more detail later in this document, in a chapter dedicated to the storage domain, the VM-specific block devices (used for COW device and private data) are encrypted with AppVM-specific key, known only to the AppVM and Dom0 (encryption is done by LUKS). The read-only block device used for the root filesystem, on the other hand, is signed, and each AppVM verifies this signature when using the block device (this is done on the fly on a per-block basis). Those extra security measures are needed to make the storage domain non-security-critical, so that a potential compromise of the storage domain doesn't allow to compromise the rest of the system[18].

The diagram below represents the architecture of Filesystem sharing among AppVMs.

[18] Additionally, also Intel(R) TXT is needed to make the storage domain security non-critical. See the appropriate chapter later in this document for more details.

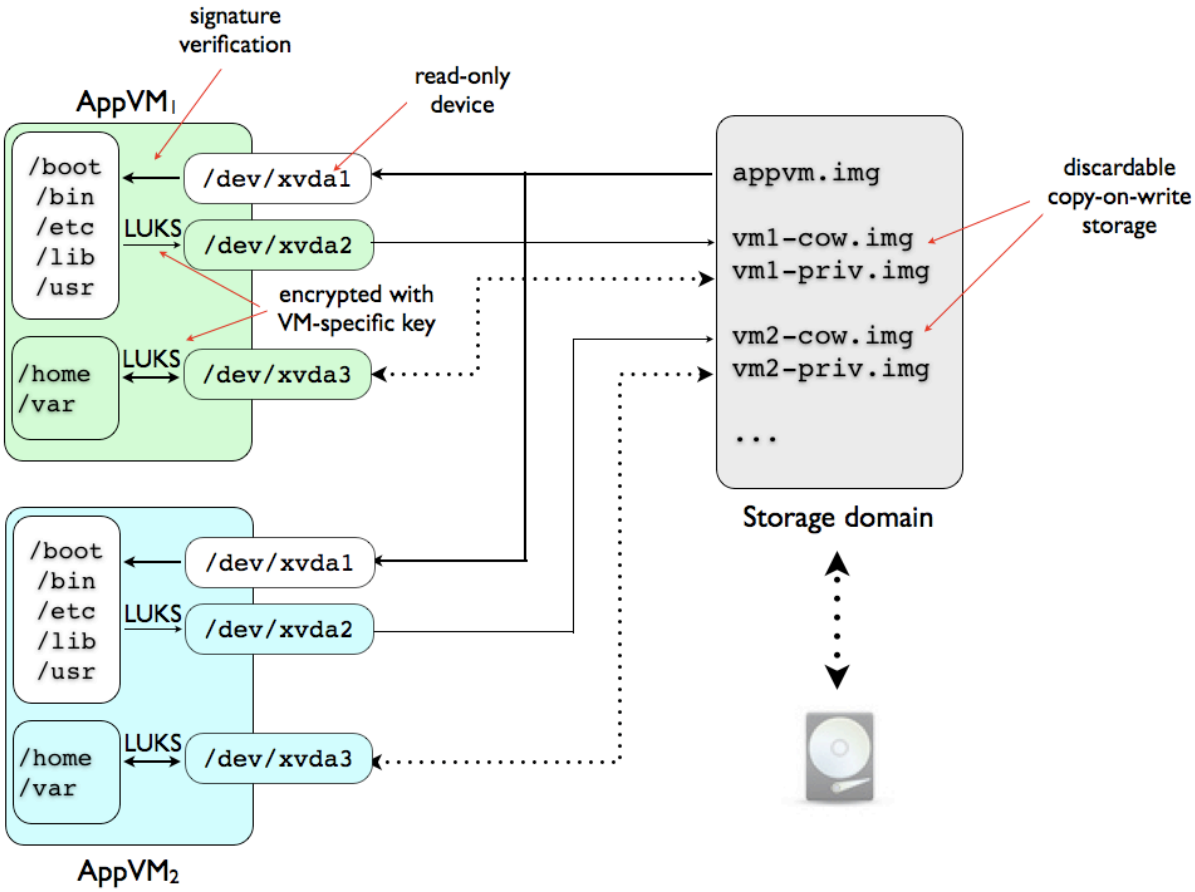

Figure 1. Secure file system sharing among many AppVMs.

It might be tempting to allow the user to explicitly mark certain VM's COW files as persistent. Such an operation would allow to preserve changes to the root filesystem to only certain AppVMs. E.g. the user might want to install a certain specialized application only in one AppVM, rather than on a common root filesystem (perhaps because the user has limited trust into this particular application, e.g. that it doesn't contain some backdoor).

Unfortunately allowing the user to preserve per-VM COW files, would break the automatic centralized updates (see later in this chapter), as it might cause desynchronization of the underlying read-only root filesystem and the COW filesystem -- the later would suddenly find itself applied to a modified root filesystem.

So, giving the user an option to preserve any of the COW files is not recommended. The user should instead install custom applications in e.g. `/usr/local` directory, that should be kept on the per-VM device (together with `/home` directory).

It is also worth mentioning that we should not use neither the PyGRUB or PVGRUB to start the AppVM. Instead an explicit specification of the AppVM kernel and initrd file should be used. Otherwise the attacker that controls the storage domain could provide a modified kernel or initrd files for each VM, so that e.g. the initrd file provided to each AppVM would not verify the signature on the root filesystem device. This in turn would effectively allow the storage domain to modify the root filesystem at will. By specifying the kernel and initrd file explicitly in Dom0, we avoid this attack, and ensure that each AppVM would verify the signature of the root filesystem (provided by the storage domain) before mounting it (and would not mount it, if the signature is wrong or missing).

4.2. Inter-VM file exchange

It is essential to provide the user with a simple and effective, yet secure, way of exchanging files between various AppVMs. Generally, there are two, independent, security challenges that should be considered here:

1. The security policy that tells which VMs can exchange files, and which direction this is allowed,
2. The actual implementation of the file exchange mechanism, so that one VM (e.g. lesser privileged) could not exploit a potential flaw in the other (more privileged) VM's code used for file exchange.

Qubes architecture focuses on how to implement the mechanism for file exchange in a secure way (so, point #2 above), and leaves it up to the user to define the policy of what VMs can and cannot exchange files.

In the future, Qubes might provide a special tool that would help the user to reinforce the defined policy, but this is optional.

More essential problem, as mentioned above, is how to securely implement the file sharing mechanism.

We currently believe that the best way to implement inter-VM file sharing, would be via a special block device, exported by the storage domain. The process would be the following (domain A wants to send a file to domain B):

1. Domain A sends a request to domain B: I have a file X that I would like you to copy from me. Additionally, domain A sends a secret key K that will be used for encrypting the file in transfer, so that the storage domain could not sniff it or modify it. This communication between domain A and B happens over a dedicated directory in the Xen Store.
2. Domain B either agrees to receive the file or not. Lack or response from domain B in certain defined time is treated as a negative response.
3. If domain A agrees, again by writing to certain Xen Store directory, then the special daemon running in the storage domain starts the exchange process by first assigning a dedicated block device to domain A.
4. Domain A mounts the exchange block device and copies the encrypted file to be transfered onto this device. Then, domain A unmounts the devices, and let the daemon in the storage domain know about it (again by writing into a special Xen Store directory).
5. The daemon in the Storage domain now assigns the exchange block device to domain B, that again mounts it and copies the encrypted file to its own local filesystem. Domain B can also decrypt the file, because domain A wrote the encryption/decryption key K into the Xen Store (with permissions set so that only domain B could read it, and not e.g. the Storage domain). When finished it lets the Storage domain know about it, again by writing to Xen Store directory.
6. The daemon in the Storage domain can now destroy the exchange block device used for file transfer and free the memory or disk file used to back this block device.

Note that the above scheme doesn't let the Storage domain, if compromised, to read or modify the file in transfer. Also, the additional code needed to implement this mechanism, i.e. the daemon that the two domains communicate with, is running in the unprivileged Storage domain, so even if one of the domains is malicious and could exploit a potential bug in this daemon, this doesn't threaten the security of the rest of the system, including the other domain taking part in the file exchange.

4.3. Network policy enforcement

Rationale for per-VM network policy enforcement

It's up to the user to define the policy of how to use AppVMs. E.g. the user might want to dedicate one VM to use for corporate email and file exchange, and perhaps a different VM to use only for internet banking. Qubes architecture doesn't enforce any particular policy.

However, once the user defined the policy of how the AppVMs should be used, it might be desirable to have some mechanisms built into the system that would help the user to enforce the policy. E.g. the user might want to expect that the AppVM that is to be used for corporate email only is only allowed to initialize IMAP and SMTP connections to the corporate email server and nothing elsewhere. Similarly, the AppVM assigned for internet banking might be allowed to only establish HTTPS connections only to the user's banking web

server. Perhaps the more paranoid user would like to add server-side SSL certificate verification to those restrictions.

The above network policy enforcement is in fact very essential, as it's not only designed to protect against user accidental mistakes (use of a wrong browser in a wrong AppVM to connect to the bank website), but also to protect against applications that would like to make internet connections without getting clear user consent. Today more and more applications behave like this. E.g. any more complex email client can also automatically turn into a almost fully-featured web browser, only because somebody sent an email with a few hyperlinks embedded in the body. Or, perhaps, because the user wanted to open the Help document, which just turned out to be implemented as a server-provided HTML file, and the application, that normally is not a web browser, suddenly spawns a web browser to let the user view the help file. Or, yet another example, the wide-spread auto-update mechanism, that so many applications implement by their own.

These are all examples of legitimate applications sometimes initiating an internet connection (most often unsecured HTTP connection) to some more or less random servers on the internet. Obviously such behavior can easily break any security policy that the user would like to implement with the help of Qubes OS. E.g. if the user's "corporate" email client, that is hosted in a special AppVM, and normally allowed to only connect to the corporate email client via an SSL-secure connection, if this email client suddenly opened a plain HTTP connection to load the help file from some server, this would create a very serious security threat. Particularly, if the user was at the airport, or in a hotel, where one should assume all the HTTP traffic is subject to malicious manipulation, that would allow the attacker to try to attack the user's application via some potential bug in the web rendering engined used by the help viewer.

Such an attack would clearly never be possible if the system properly enforced the network policy and not allow the help browser to initiate this insecure connection, and only allow the email client to use its secure connection to the email client (which is the main reason for having this application).

It's thus essential to have a mechanism to enforce the user defined network policy in a per-VM manner.

Network policy enforcer location

One might think that the Network domain is the right place where the per-VM network policy enforcement should be implemented. After all the assumption is that all the VMs would be connecting to the outside world thorough the Network domain. Unfortunately, having the network policy enforcement logic in the Network domain would have a side effect of making the Network domain more privileged than we would like it to be, because if the attacker compromised the Network domain this would mean that the attacker could tamper with the network policy, and this might later lead to compromising the security of one of the VMs because of the security problems outlined in the paragraph above.

Consequently, we believe that it is better from the security stand point to place the network policy enforcement logic in each AppVM. More specifically, assuming that AppVMs are based on Linux, we can make use of the standard Linux built in packet filter configurable via iptables.

One might be suspicious about placing the enforcement logic into the same VM on which this policy is about to be enforced. However, this doesn't present a conflict for our threat model. One should note that the purpose of having the network policy enforced is not to prevent against potentially malicious applications running in the particular AppVM, but rather to prevent the legitimate, but perhaps not-so-securely written, applications from being compromised when they try to accidentally establish a connection. Particularly, if we assumed that one of the applications in the VM (that we want to protect with the network policy enforcement) is malicious, then even having the network policy enforcement in an isolated trusted VM would not help much, as there are many ways to leak the information out via TCP/IP-based networking using various more or less advanced covert channels.

So there is no benefit of having the policy enforcer isolated from the VM it is supposed to protect, because if it is already compromised, then the policy enforcement would hardly provide any additional security. At the same time, by having the policy enforcer located in each AppVM (each configured accordingly to what policy one wants to enforce on particular VM) we do not add a target that the attacker could attack and compromise, as it would be the case if we had the central policy enforcer for all VMs in a separate domain, e.g. in the Network domain.

Of course, we can imagine that in the future versions of Qubes OS, the user would have an easy-to-use tool with simple user interface that would be used to help the user to configure the enforced in each AppVM, e.g.

by automatically generating all the needed iptables rules in a form of a ready-to-use script that would be later copied to each AppVM.

4.4. AppVM installation and initial setup

It is expected that users will be provided with the AppVM ready-to-use images, as part of the Qubes OS distribution. However, some more advanced users might want to have an option to install custom operating systems as their AppVMs. This might include a different Linux distribution, or, in the future, a different OS, assuming it would be supported by the Qubes OS (e.g. GUI agent will be available for this OS).

In any case the procedure for creating a new image of AppVM should be the following.

1. Boot the OS installer into a “full screen mode”.

   The term “full screen mode” used here, means that the OS installer should be provided with VGA emulation (like for a HVM guest) and we would like to see the whole screen of this VM in a window in Dom0’s AppViewer. This step provides potential security weakness if the OS we are trying to install was malicious, e.g. if the OS installer tried exploit the VGA emulator. To account for such an attack, we should run the VGA emulator (and perhaps all other I/0 emulation if wanted to run the guest in HVM mode) in a separate PV domain, similar to Xen stub domain, with an exception that such domain should also have an X server and our GUI agent running, to make it possible to display the guest’s emulated VGA in AppViewer as if it was a normal X application.

2. Boot the newly installed OS

   Once the system has been installed using the regular installer (see previous point), the next step is to boot the system, also in “full screen mode” and get administrative (root) access to the system, e.g. by opening a console.

3. If the OS to be installed is Linux based, and we plan to use root filesystem sharing, to be able to easily create instances of AppVMs based on this OS, then we need to make sure that the system will use the kernel and initrd files provided by Dom0, as part of domain building process. Particularly, the initrd file to be used by this system should take care about verifying and mounting the root filesystem, as it has been described earlier in this chapter.

   It’s currently not clear how to support shared root filesystem in case of other-than Linux OSes, like e.g. Windows. This is left to be determined later. Until this is resolved, installation of AppVMs based on other OSes could probably skip this step (the root filesystem for an ApPVM that doesn’t use sharing, can be simply encrypted with a per-VM key, so no need to verify any signature by the early boot script -- the storage domain would not be able to modify this filesystem anyway).

4. The next step is to install the GUI agent, configure the X server (e.g. to use the dummy driver, and to use the XComposite extension), and to install additional scripts that would be starting the GUI agent on each boot of the AppVM.

5. The last step includes some final configuration activities that would depend on the actual OS being installed and its purpose. E.g. Linux-based AppVMs with shared root filesystem might require disabling of access time recording on the root filesystem, to avoid unnecessary growth of the COW files and to optimize performance. Also, one would most likely want to disable automatic updates, as they can be handled in a centralized way, as described below.

6. Finally we should reboot the VM, and since now on it should be possible to use it in a AppVM-fashion, i.e. via the AppViewer application in Dom0.

4.5. AppVM centralized updates

Because all of the AppVMs (or at least most of) share the same root filesystem, it’s desirable to have a simple method to perform automatic updates of this file system, which will result in all the applications being automatically updated in All AppVMs at once.

In order to perform such global AppVMs update, we need to first shut down all the VMs and then start a special UpdateVM that will have read-write access to the root filesystem block device, and also to the signing key so that it can resign this device.

Obviously this allows the UpdateVM to potentially compromise all the AppVMs based on the root filesystem that the UpdateVM was granted update access. This means that the UpdateVM should be strictly restricted only to secure connections to the update server (e.g. certain Linux repository server) and should perform signature verification on the updates.

The update process should always be initialized by the user.

One important thing to note is that the UpdateVM should not attempt to upgrade the VM's kernel, if the AppVM uses root filesystem sharing. There are two reasons for this:

1. The AppVMs that use shared root filesystem need to use kernels and initrd files "injected" by the Dom0 domain builder to avoid attacks originating from the storage domain (e.g. disabling of root filesystem signature verification, as described above).
2. Each AppVM must use a customized initrd script that e.g. knows how to verify the root filesystem, and how to create a R/W COW-based device out of this script.

For the reasons given above, the kernel of AppVMs that use filesystem sharing should not be updated automatically, but instead manually -- e.g. via a dedicated script in the UpdateVM that would take care about customizing the initrd script and also copying the kernel and initrd files to Dom0 and updated the domain configuration file to account for the updated versions.

We should realize, however, that kernel updates for AppVM should very rarely be required, except perhaps for the network domain, that sometimes must be updated to support new hardware. The regular AppVMs however, do not deal with any real hardware, as they use Xen-provided frontend drivers to access disk and network virtual devices.

4.6. AppVM trusted booting

One interesting side effect of having the common read-only root filesystem shared among all the AppVMs, is that each time the VM boots, it boots into a "clean", i.e. non compromised, state. Indeed, even if the VM got somehow compromised, e.g. via a web browser exploit, the next time it boots all the potential changes to the root filesystem will be discarded (the COW backing file will be discarded by storage domain[19]), and everything, starting from the kernel, through the initrd script, and all the system binaries will be automatically resorted to the state as they were after the system was installed (or upgraded by the UpgradeVM). This brings a very desired security feature for the user.

However, as soon as the user-provided contents becomes executing, this property is lost. For instance, the user might have installed some plugins to the commonly used applications, e.g. a flash player plugin for the Web browser. Those plugins will, of course, be kept in the user's home directory (it cannot be otherwise, as the VM is not allowed to write to the read-only shared root filesystem; whatever it would write there, even if the user had root access in the VM, it would get discarded after the next reboot). If one of the plugins was malicious, the plugin could compromise the system each time the user starts the application that uses them, which might end up being almost always.

Similarly, especially in case of UNIX-based systems, it is possible to modify configuration files or startup scripts used by the commonly used applications, such as `bash`. This way the attacker can introduce potentially malicious behavior (e.g. stealing the user sensitive data), by only modifying the contents of the user home directory.

Qubes cannot protect the particular VM from becoming compromised by the user's actions. Qubes can only assure that the VM would be clean directly after boot, and, of course, to make sure that the other VMs would not be infected, even if particular VM is. Everything that happens inside the VM is beyond the Qubes security model.

However, Qubes should offer the user some options to deal with such scenarios. One possibility to deal with such problems is to allow the user to create AppVMs that would have non-persistent home directory, i.e. the file used to backup the home directory would be discarded after every reboot of this particular AppVM. The user might then just use such a one-time AppVM to do a particular task, and later just discard everything, so that the next time the user starts the machine is clean.

[19] Or reverted to the user's chosen "last known good state".

Another option would be let the user mount the AppVM's home directory into another AppVM, say the "investigation AppVM", that would contain various tools that an experienced user might use in order to see if anything is suspicious in the home directory and remove the offending files if needed.

This scenario with compromised home directory presents a much easier task for the investigator, than traditional investigation into a compromised system, where virtually every component might be compromised, starting from the BIOS an PC devices firmware, through boot loader an kernel, system root fileystem, and on the home directory ending. Here we "only" need to investigate the home directory for potentially malicious modifications.

4.7. Runtime integrity verification of running AppVMs

It would be desirable if Qubes could also provide a mechanism for runtime verification of the running kernel in the AppVMs, to detect potential runtime attacks, e.g. installation of a kernel keylogger or other form of a rootkit. One could attempt to create such a detector, by adding a special "Integrity Scanning" VM, that would be granted read-only access to all the other VMs (except for Dom0) and no network connectivity (to avoid potential leaks, in case the integrity verification engine was compromised).

Such an Integrity Scanning VM might then attempt to periodically verify all the memory pages marked as executable used by the given VM's kernel. If only we could enforce each VM's kernel to allow only signed code, and also to make sure that all the non-code pages are marked as non-executable (NX), and also make sure that the kernel doesn't use any self-modifying code, and also, through the use of virtualization, ensure that the VM's kernel execution patch cannot jump out to a usermode page, then such a scheme should be rather straightforward.

However, besides the problems mentioned above, there remains a problem of the so called Return-oriented exploits/rootkits, that could introduce arbitrary programs without injecting any foreign code[20]. Another problem that needs to be answered is how much performance impact the introduction of such a Integrity Scanning VM would cause.

In other words, this subject is left for further research.

[20] The reader is references to the Hovav Shacham's Black Hat presentation on *Return-Oriented Programming* (http://cseweb.ucsd.edu/~hovav/talks/blackhat08.html), and also to the Ralf Hund's, Thorsten Holz's, and Felix C. Freiling's paper on *Retun-Oriented Rootkits* (http://www.usenix.org/events/sec09/tech/full_papers/hund.pdf).

5. Secure GUI

This chapter describes the details of the Qubes graphics (GUI) subsystem architecture. Secure and efficient GUI is one of the most important elements of the system.

5.1. Secure GUI challenges

While technically the GUI subsystem is less privileged than the hypervisor, a compromise of this subsystem still allows for very severe system compromise. If the attacker gains control over the GUI subsystem (e.g. the Window Manager, or the X server), one can interact with all the applications in all user's virtual machines. Thus, special care should be paid into making the GUI interface, which is used to communicate with other VMs, very simple and secure.

Another important property of the GUI subsystem is the ability to always provide the user with information about the currently active (focused) application, together with clear description of the VM that hosts the application. This is to prevent various spoofing attacks, where some untrusted application (e.g. from "Red VM") could e.g. create a window covering full screen and display imitation of other windows, this time from more trusted VMs, and trick the user e.g. into entering a secret that the user normally would only enter when prompted by a more trusted application (e.g. credit card number, that the user would only enter into the "Yellow VM").

Finally, it is important to prevent applications that run in different VMs from being able to e.g. take a snapshot of the screen occupied by applications belonging to other VMs, or sniff or inject events to the other applications.

At the same time we would like the GUI subsystem to all the near-native user experience, so that e.g. it was possible to watch movies played by applications executing on different VMs.

5.2. GUI subsystem location: DomX vs. Dom0

Before we move on to describe the details of the GUI subsystem architecture in the Qubes OS, we should first discuss where the GUI subsystem should be located. We should consider two possible places: in the administrative domain (Dom0), or in the unprivileged domain with direct access to the graphics and audio hardware (which we call DomX, to stress that the X server runs in this domain).

The strongest argument for keeping the GUI subsystem in Dom0 is that the attacker, by gaining control over the GUI subsystem, effectively gains control over the whole system anyway. After all the attacker can now interact with all the user AppVMs, stealing all their secrets, executing programs there, etc.

On the other hand, if the interface between DomX and Dom0 is well designed, such a separation might at least prevent the attacker, who gained control over DomX, from introducing any system-wide persistent modifications. E.g. the attacker should not be able to re-seal secrets in TPM after replacing the hypervisor or Dom0 kernel image. The attacker should also not be able to modify the root filesystem used by all the VMs, as only Dom0 should have the signing key, and could allow modifications to this filesystem only when booted in special mode (not to be confused with previously mentioned maintenance mode, that is actually a less privileged mode for running Dom0).

On yet another hand though, we already tried to make the VM↔GUI interface as secure as possible (see the following paragraphs), and it's not immediately clear if the potential DomX↔Dom0 interface could be significantly more secure, as it would be based on the same Xen inter-VM communication mechanisms, i.e. the Xen shared memory and Xen event channels. At the same time, separating GUI from Dom0 would complicate the design of the system, e.g. it's unclear how to do "PS/2 keyboard and mouse passthrough" from DomX.

Consequently, we believe it is not worth the effort to move the graphics subsystem out of the Dom0, as the potential security benefits are very minimal, if any.

5.3. Qubes GUI architecture

The figure below shows the basic idea of how Qubes implements secure GUI. In each AppVM there is Qubes agent running (think of it as of a Window Manager), that is responsible for the following tasks:

1. Sending notifications to the AppViewer about new windows appearing on the AppVM's virtual desktop. For each newly created window, the agent obtains the content of this window (or just the address of the composition buffer) and sends this to AppViewer.
2. Sending notifications to AppViewer whenever the content of any of the windows changes (relaying XDamage notifications).
3. Focus management -- whenever the user changes focus to another AppViewer window in Dom0, the AppVM Window Manager does the same in the AppVM, so the focus in Dom0 and AppVM is always synchronized.
4. Relaying all the keyboard and mouse input, entered into the corresponding AppViewer window in Dom0, to the focused local application.

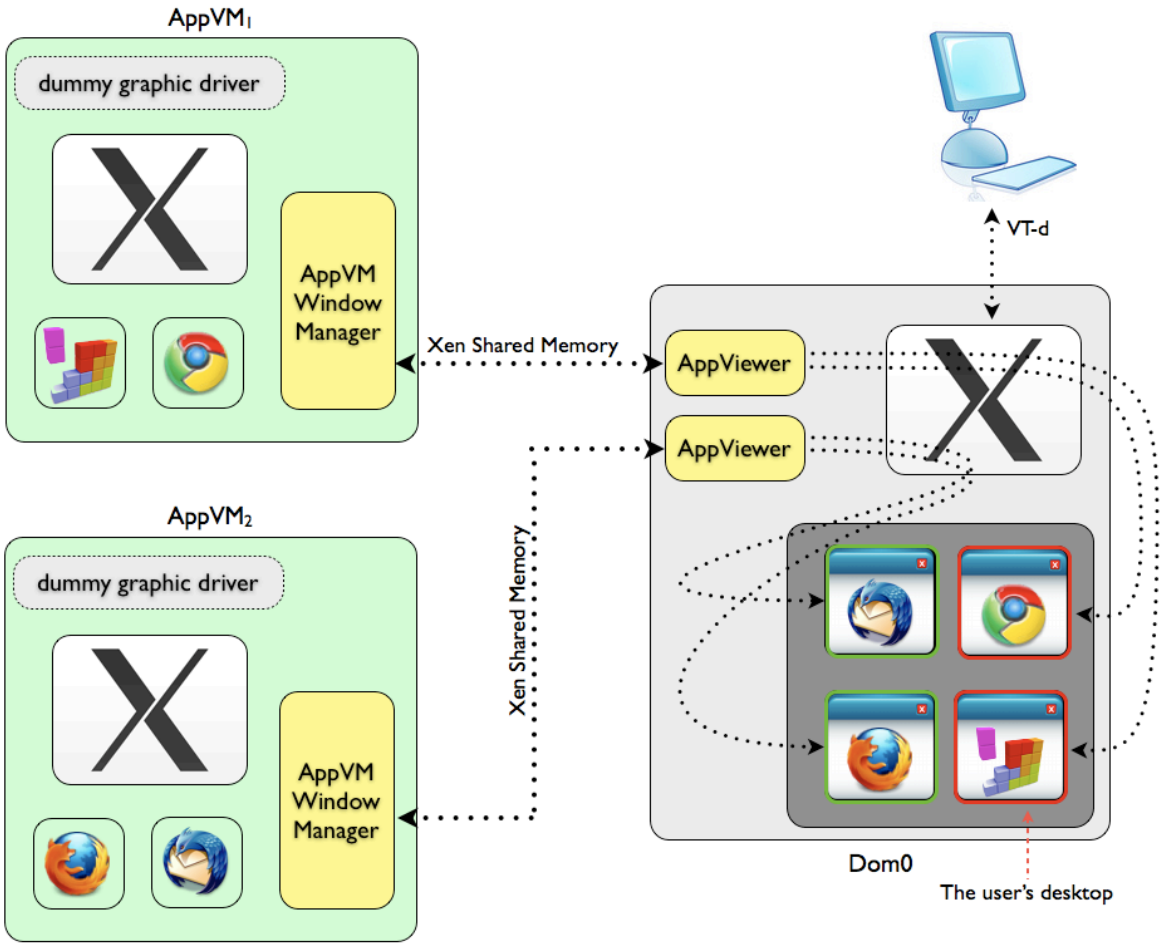

Figure 1. GUI subsystem design overview.

The communication protocol between the Qubes agent in AppVM and the AppViewer in Dom0 is a simple message-based protocol, implemented using the standard Xen Ring buffer protocol (the same that is used by VMs to communicate e.g. with the Xen Store Daemon).

We assume that the X server running in the AppVM uses *composition buffer* to keep track of the content of each window (even if its obstructed by other windows). In practice most of the Linux distributions today use the Xorg's X servers, and all the recent versions of Xorg server have native support for composition mode

provided by the *XComposite* extension[21]. Interestingly other major operating systems, like Mac OS X, and Windows[22], also use composition buffers for window content tracking.

The composition mode support is essential, because it allows our agent to get unobstructed image of each window content, no matter whether they obstruct each other on the virtual desktop or not.

In the simplest form the agent in AppVM can obtain the contents of each application window using the regular `XGetImage()` API function, that copies from the composition buffer, if composition mode is enabled in the X server. However, a more effective implementation, that is currently being investigated, would use the agent to only get the address of the composition buffer (virtual address in the local X process), send this address to the AppViewer running in Dom0, who, in turn, will map the composition buffer into its own address space using the Xen shared memory mechanism. This would save on unnecessary buffer copying, as sharing pages between VMs does not involve a copy operation.

The AppViewer, having the buffer (pixmap) with application content mapped in its own space, can now display it in one of its windows using the very efficient `XRenderComposite()` function.

Even when the composition buffer is directly shared with the AppViewer, the AppViewer needs to know when the buffer changes, so it can inform the X server running in Dom0 about the need to re-render the AppViewer window content (via `XRenderComposite()` call). The Qubes agent running in AppVM registers to receive XDamage notifications from the local X server, whenever any of the window on the virtual desktop gets updates (even if it is obstructed). The agent passes those notifications to the AppViewer, using the ring buffer protocol mentioned earlier.

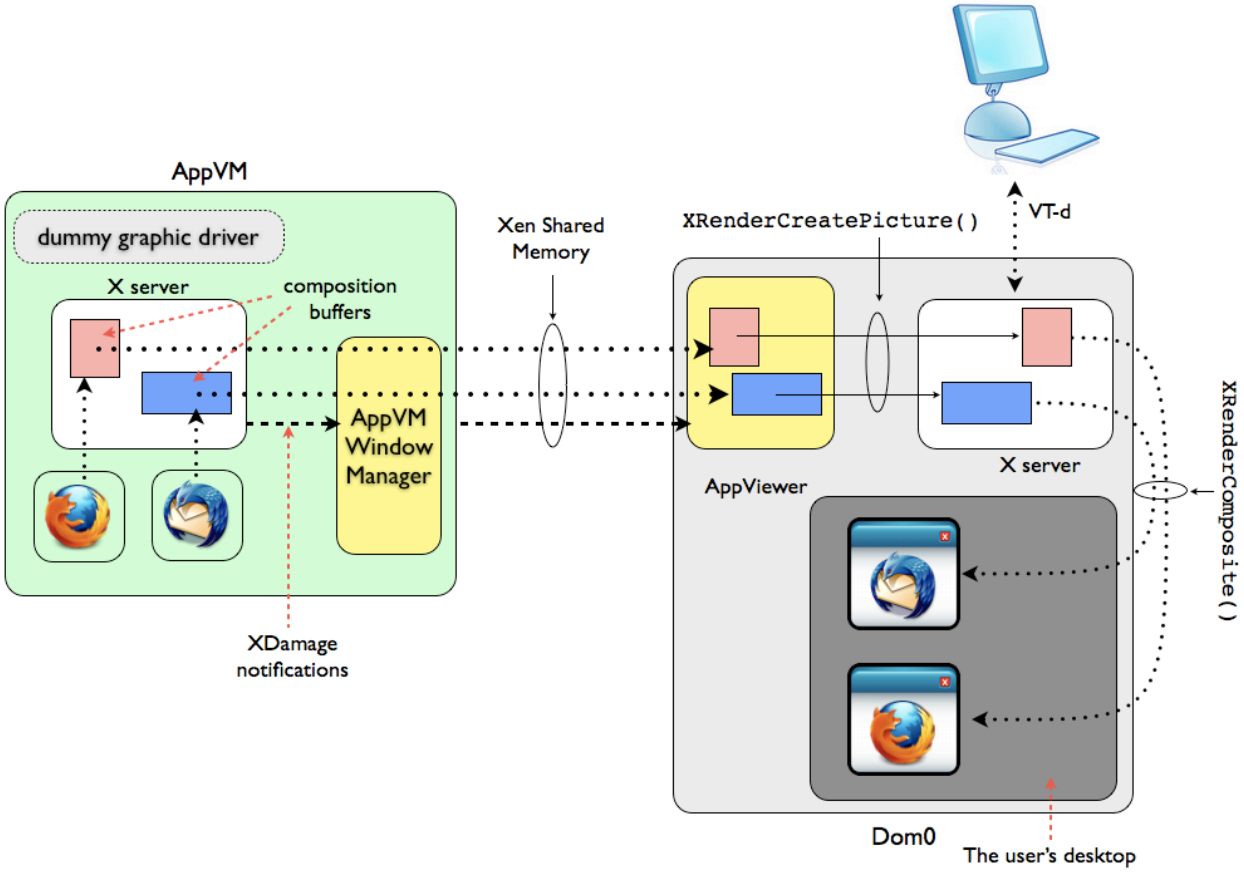

Figure 2. Efficient implementation of AppViewer using composition buffers (requires XComposite extension to be enabled in the AppVM's X server).

[21] http://www.x.org/releases/X11R7.5/doc/compositeproto/compositeproto.txt

[22] Starting from Windows Vista.

5.4. Application stubs

In order to start a new application in AppVM the special program called the AppStub is executed in Dom0. This program should take care about first checking if the VM where the particular application is hosted has already been started, as well as the corresponding AppViewer in Dom0, and later to send a command to the GUI agent in the AppVM to start the particular application (e.g. `/usr/bin/firefox`) and make sure the application started successfully, or otherwise report a problem to the user.

5.5. Secure clipboard

Secure clipboard that could allow to copy and paste block of data between applications executing in various AppVMs is essential for effective use of the system.

Qubes doesn't enforce any policy on inter-VM data flows, leaving this decision up to the user. However, it should not be difficult to add an additional policy monitor to the GUI daemon in Dom0, which could enforce some formally defined policy on clipboard and file exchange (e.g. which VMs can exchange data, which cannot, perhaps also specifying the allowed directions). The specification of such a policy monitor, is, however, outside the scope of this document.

The algorithm specified below should allow for secure clipboard exchange between AppVMs.

Clipboard algorithm for the "copy" operation

1. The copy operation is initialized by the magic key combination for the copy operation (e.g. Shift-Ctrl-C) entered in Dom0 into the focused AppViewer
2. The AppViewer does not pass the keystrokes to the AppVM. Instead it builds and sends the `get_clipboard` command over the GUI communication channel to the GUI agent running in the AppVM.
3. The Agent in the AppVM receives the `get_clipboard` command and, in response creates a response command (`clipboard`) where it copies whatever is currently "marked" in the AppVM, limited to the max allowed buffer size.
4. AppViewer receives the `clipboard` message and copies the content into the defined storage space in Dom0. This storage space could be e.g. a dedicated file in the Dom0's `/tmp` or `/dev/shm` directory. Particularly this should not be placed into the Dom0's Window Manager's clipboard buffer, to make sure that the user doesn't mistakenly paste the buffer into any of the Dom0's applications. The user should only be allowed to paste the contents of the clipboard buffer into another AppViewer window.

Clipboard protocol for the "paste" operation

1. The paste operation is initialized by the magic key combination for the paste operation (e.g. Shift-Ctrl-V) entered in Dom0 into the focused AppViewer
2. The AppViewer does not pass the keystrokes to the AppVM, instead the AppViewer sends the `clipboard` message to the agent in the AppVM.
3. The agent copies the contents from the `clipboard` packet to the VM's Window Manager clipboard buffer and simulates and initializes the paste operation into the focused application (as if the user pressed Ctrl-V locally in the AppVM).

Why is the clipboard protocol secure?

The proposed clipboard exchange protocol seems to be secure, because the paste operation (so, retrieving the contents of the clipboard buffer) is always initialized explicitly by the user in Dom0. Consequently the unauthorized VMs do not have a chance to steal the contents of the clipboard buffer.

5.6. Audio support

The applications executing in AppVMs might generate audio, e.g. the Web browser might generate audio as a result of the user watching a You Tube video. It is thus important to bring the sound from the VMs to Dom0, where it could be played on the physical audio card, that is assigned to Dom0.

The most straightforward way for implementing audio support, seems to be to extend the role of the GUI agent running the AppVMs, so that it also reported where is the audio output buffer located, so that this buffer could be shared with Dom0, using the same mechanism as the graphics buffers are shared.

Then, on the Dom0 side, each AppViewer would be responsible to “play” the provided audio buffer. The standard sound mixing application in Dom0 would be then responsible of creating the final audio stream, composed of all the streams provided by each of the running AppVM (mixed according to the user’s preferences).

The exact details of implementation of Audio sharing between AppVMs and Dom0 are currently left for further research.

6. Secure networking

Network should always be considered hostile. This is especially true for laptop users who need to connect to the Internet via public WiFi hotspots, e.g. at the airport or in the hotel. Today networking software is very complex - it comprises the network card drivers (e.g. WiFi drivers), the various protocol stacks (e.g. the 802.11 very complex stack), the TCP/IP stack, the firewalling software, and often applications such as DHCP clients, etc. Most of this software runs with kernel or system privileges on typical OS like Windows or Linux, which means that the attacker who exploited a bug e.g. in the WiFi driver or stack can compromise the whole system, including all user data and applications.

It's an obvious attack surface and Qubes OS architecture aims at eliminating it, by moving all the world-facing networking code into a dedicated network domain. Another reason to move the networking code out of Dom0 is to prevent potential attacks originating from other VMs and targeting potential bugs in Xen network backends, or Dom0 minimal TCP/IP stack.

6.1. The network domain

The network domain is granted direct access to the networking hardware, e.g. the WiFi or ethernet card. Besides, it is a regular unprivileged PV domain.

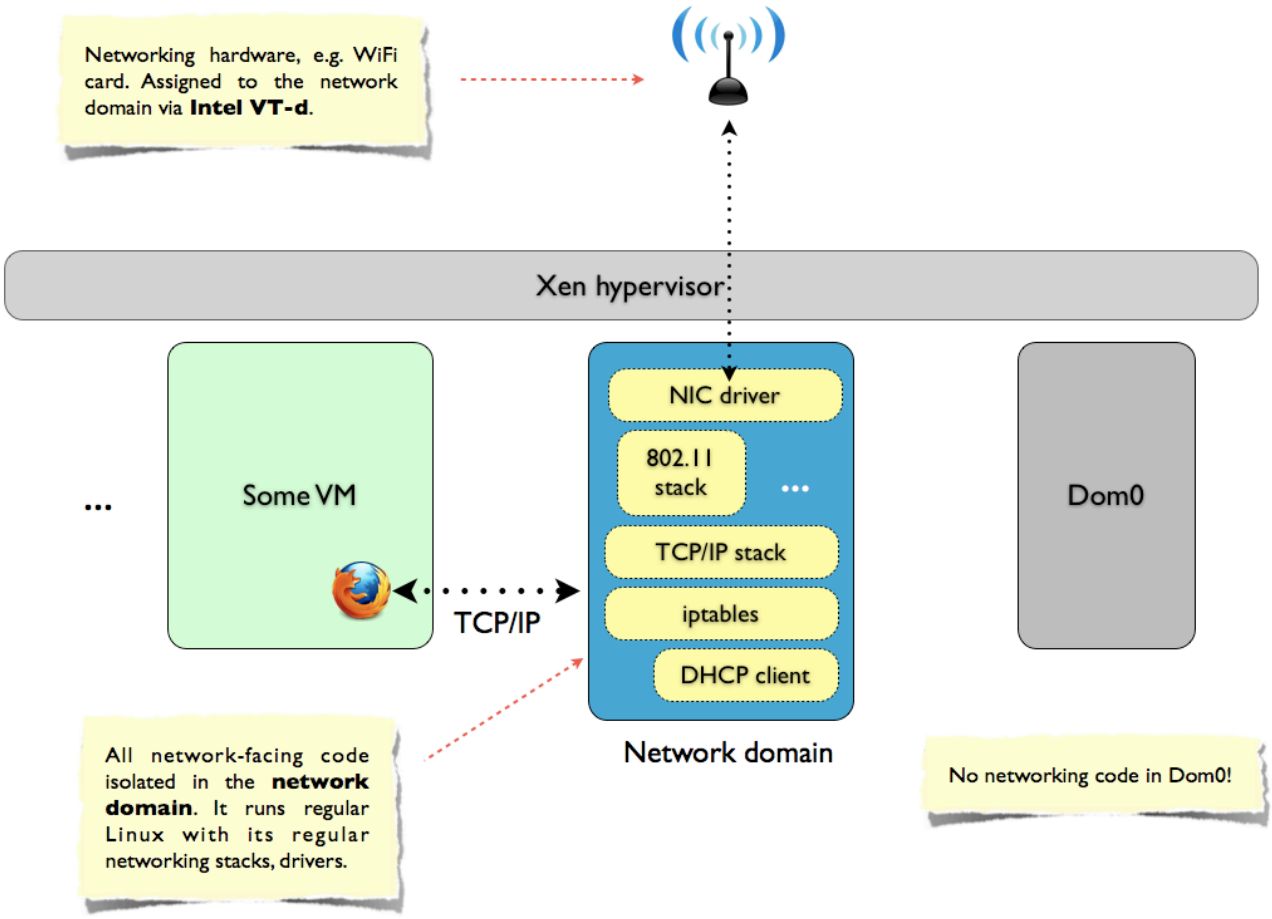

Figure 1. The network domain.

While it might be tempting to base the network domain on a regular AppVM root filesystem that all other VMs share, this might not be the optimal solution from the security point of view. Namely, there exist a slight chance of a bug in the regular Linux TCP/IP stack (in contrast to a more likely bug e.g. in the WiFi driver or WiFi stack). If such a bug existed and the attacker used it to exploit the stack in the network domain, then the

attacker could automatically use this same exploit to further exploit any network-connected AppVM that used this very network domain.

Thus, it's better to use a different OS in the network domain, e.g. FreeBSD, instead of Linux, so that the attacker couldn't reuse the same hypothetical bug in the TCP/IP stack -- quite obviously two different OSes would have very different TCP/IP stack implementations.

6.2. No Inter-VM networking

Each AppVM that uses networking uses a virtual network interface created by Xen network frontend, that appears as `eth0` in the VM. The other side of the interface, in the network domain, is named `vifX.Y`, where `X` is the VM's ID number, while `Y` the per-VM number of the interface (0 in case there is only one interface per domain, which is the case here). It is a common practice to bridge together all the `vif*` interfaces on the backend side (in our case network domain side) and connect them to the outgoing interface (e.g. `wlan0`) via the regular Linux packet filter.

Such a configuration is however not optimal in case of Qubes. It's because it is not desirable to allow one VM (perhaps of of lower privilege) to interact with the TCP/IP stack of another VM (perhaps more sensitive), because there is a slight chance of an exploitable bug in the TCP/IP stack of the OS running the VM.

Consequently it is recommend to not bridge the `vif*` interfaces together, but rather connect them via the packet filter to the outgoing interface, making sure that the packet filter blocks any inter-VM traffic. Alternatively one can use bridging, but with explicit filter rules to forbid inter-VM traffic. The latter approach might even be more desirable, because it would allow to easily support other protocols than ipv4, and would also allow to make exceptions and allow selected AppVMs to have direct networking connectivity (e.g. for development and testing purposes).

The figure below illustrates the above.

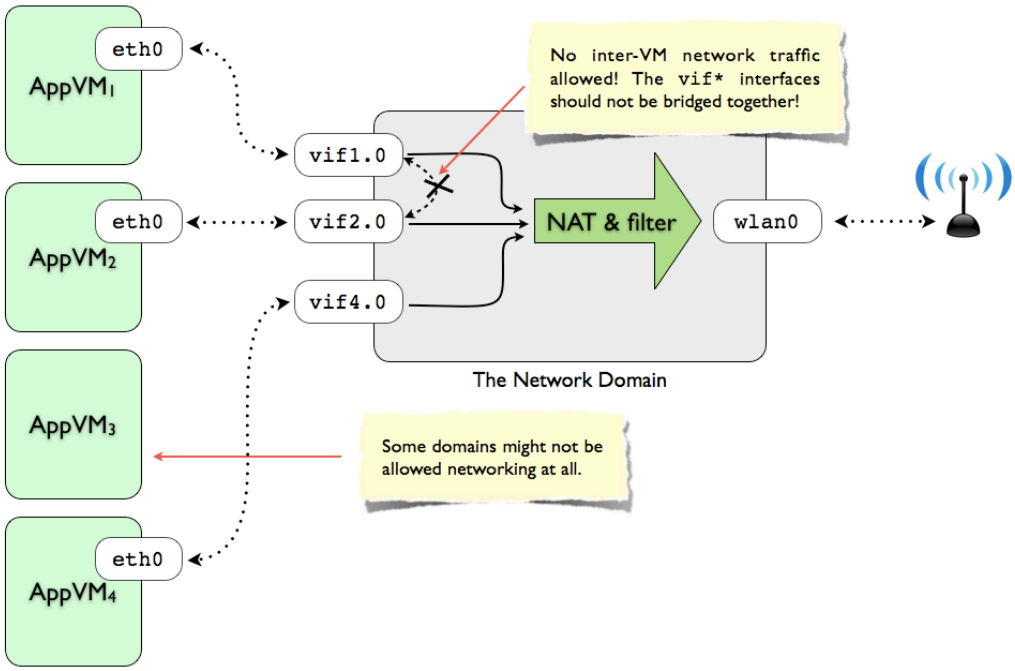

Figure 2. Networking overview.

6.3. The Net domain user Interface

The user should be allowed to somehow interact with the networking domain. The user input is necessary to e.g. choose the desired WiFI SSID to connect to, and to provide the optional WEP/WPA password. The more advanced users might would like to have even more control of the networking, e.g. be able to manually configure the outgoing interface IP address, etc.

The control over the network domain should be provided via a GUI application running in the network domain. This might be something as simple a an `xterm` application with root access, or something much more user friendly like e.g. the Network Manager GUI.

In both cases the applications should be made accessible to the user using the regular Qubes GUI mechanism for AppVMs, i.e. the GUI agent in the VM and the AppViewer in Dom0 displaying the application.

This places a requirement on the OS used for the network domain, that the GUI agent should support this OS. Currently this would mean that the OS runs an X server (with a dummy graphics driver, as usual).

6.4. The optional VPN domain

Normally it is assumed that all the network traffic encryption is performed by the AppVM that initiates and uses the connection. For instance if the user uses a Web browser in the AppVM to connect to the web server over an encrypted SSL channel (HTTPS), then the SSL engine is running in this very AppVM, not in the network domain. Thus, the network domain can be considered security non-critical, as all the sensitive traffic traversing through it is encrypted and the network domain doesn't know the keys.

In some situations, however, the user might want to use a VPN connection to provide automatic encryption for several (or all) AppVMs at once. For example a user at the airport might prefer to use VPN, rather than allowing all the other people in the lounge to watch his or her traffic, even if it is considered non sensitive. Another scenario might be e.g. the user having a few different "corporate" AppVMs that all should use the same corporate VPN.

It might be tempting to place the VPN client software in the network domain. However such an approach would not be optimal from the security point of view, because we would like to keep the network domain as security non-critical.

Thus, it's better, from the security standpoint, to have a separate "VPN domain", where the VPN client software can be run, isolated from all the world-facing networking code, that runs in the network domain. The VPN domain should expose virtual network interfaces to other AppVMs, just like the network domain. So, the VPN domain "looks like" a network domain to all the AppVMs that the user chooses to use this VPN. On the other hand, the VPN domain should use the networking provided by the network domain, just like any other VM in the system. Thus, from the network domain point of view, the VPN domain "looks like" an AppVM.

The user might want to use the VPN networking provided by the VPN domain in all AppVMs, or just a few selected AppVM, while having the other AppVM to use the regular network domain-provided networking.

The VPN domain should be understood in a broader sense than just a VPN protocol. Perhaps some user would like to use a "VPN domain" that would be running *tor* client and provide anonymous networking to all the AppVM that use it. Such approach would be very desirable, because all the AppVMs connected through such a "tor domain" would be using anonymous networking whether they want it or not, i.e. it would not be necessary to assure cooperation of the applications (e.g. via specifying tor as a proxy).

The exemplary setup with VPN domain is illustrated on a figure below.

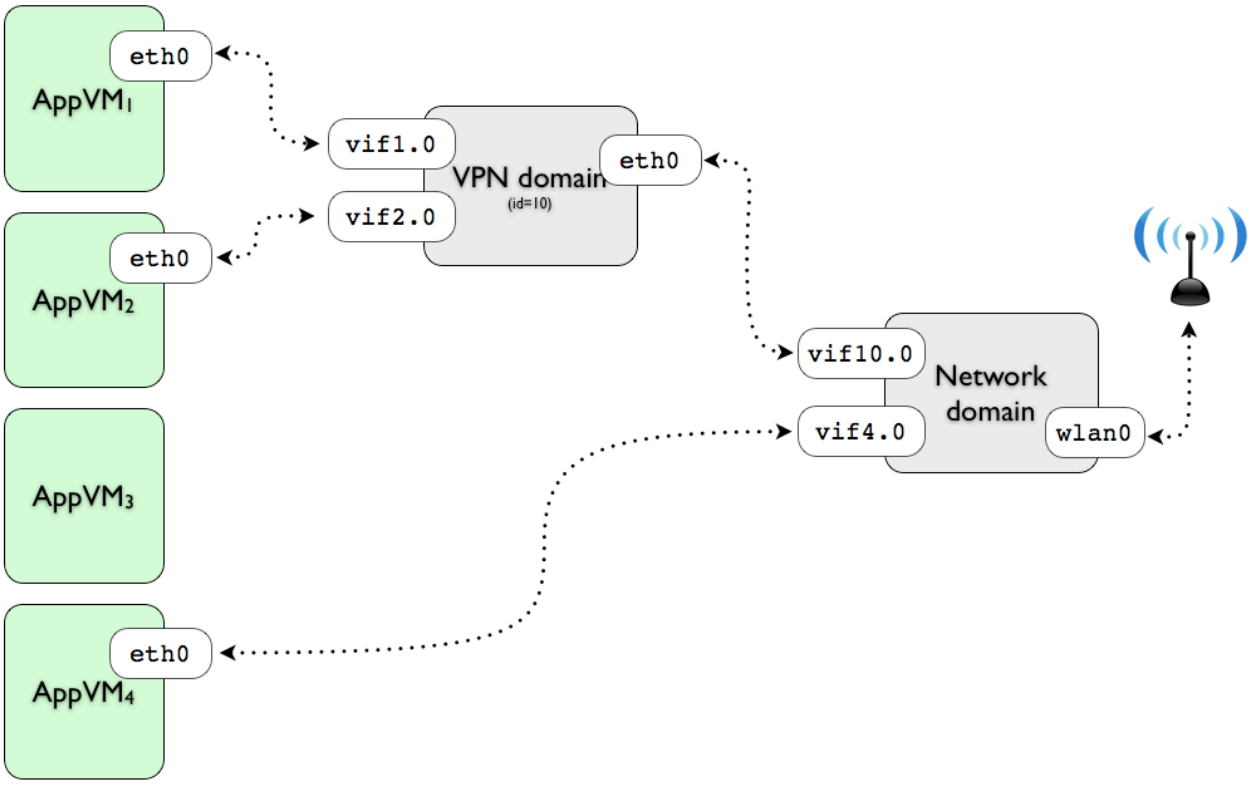

Figure 3. Exemplary VPN usage.

7. Secure storage

Secure storage subsystem is essential for any system. There are several things that all together make the storage secure in the Qubes architecture:

1. Confidentiality, understood as preventing one VM from reading other VMs data
2. Confidentiality, understood as preventing access to the data when the machine is left unattended (full disk encryption, resistance to Evil Maid attacks, etc)
3. Integrity, understood as preventing one VM from interfering with the filesystem used by other VMs
4. Security non-critical role: a potential compromise of the storage subsystem doesn't result in other system components, like other VMs, compromise. Storage subsystem is not part of the TCB in Qubes OS.

7.1. The Storage Domain

Qubes architecture implements secure storage by having a separate, unprivileged, domain which has access to the disk and other mass storage devices, and that also hosts the Xen block backend that are used to export the virtual block devices to all other VMs, including the Dom0. Cryptography is used to assure that the storage domain cannot compromise the filesystems used by other domains in any meaningful way, which makes the storage domain security non-critical. This means that if the attacker managed to compromise the storage domain, this would not automatically let the attacker to compromise any other VM.

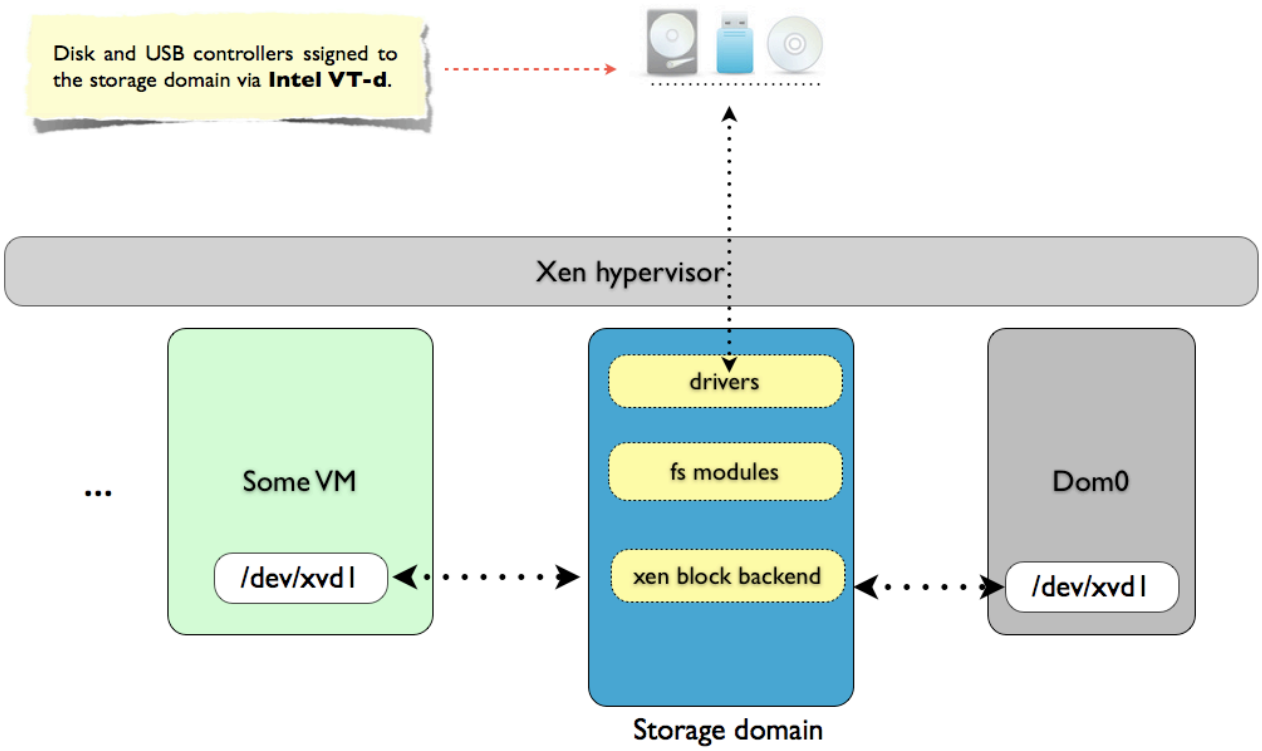

Figure 1. The storage domain.

Unlike in case of the Network domain, the Storage domain does not contain any world-facing code, so the need to create an isolated domain for "disk drivers" might seem a bit of an overkill at first look. However, we

should remember that there is lots of code running to provide all the subsystem services: block device backends, file exchange daemon (described earlier), USB PV backends, etc. By isolating all this code in a separate, unprivileged domain, we don't need to worry about potential bugs that might be present in this code. Indeed, Qubes architecture assures, that even if the attacker compromised the storage domain, this doesn't automatically let the attacker to compromise the rest of the system, including the VMs that have their filesystem provided via the Storage domain.

The property of the storage domain being security non-critical is achieved with the help of three mechanisms:

1. Encrypted per-VM private storage devices (only given AppVM and Dom0 know the key)
2. Signed block device with root filesystem used by AppVMs (only the UpdateVM and Dom0 know the signing key)
3. TPM-based trusted/verified boot process implemented with the help of Intel TXT

The sharing of the root file system among AppVM has already been discussed in the chapter devoted to AppVM architecture.

One thing that might require clarification is why it needs to be stored on a signed device, instead of an encrypted device. If the shared root filesystem was stored on an encrypted block device, then the decryption key would have to be shared among all the AppVMs (but not with the Storage domain). However, this would mean that if the attacker compromised only one of the AppVMs (perhaps the one that was least privileged) _and_ the storage domain, then the attacker could compromise all the other AppVMs, because now the attacker would both know the encryption key and would have access to the Storage domain. In case of a signed block device the above scenario isn't possible, because only the UpdateVM and Dom0 know the signing key, and all the other AppVMs know only how to verify the signature (public portion of the signing key).

Another thing worth explaining in more detail is the need for verified boot process, in that case based on Intel Trusted Execution Technology. Without trusted boot mechanism, the attacker who compromised the storage domain could then compromise the hypervisor or Dom0 image stored in plaintext on the boot partition. Indeed nothing could stop the storage domain from doing this, as it has full access to the disk controller (granted via Intel VT-d). The only way to prevent such an attack is by using a trusted boot scheme, like Intel TXT. In that case, the storage domain could still modify the hypervisor image on disk, and the (compromised) hypervisor would still be loaded and started, but TPM would not release the secret needed to decrypt the rest of the filesystems. The system would not be able to boot properly.

It's worth stressing that a compromise of a storage domain could always lead to the Denial-Of-Service attack. As we just explained, any attempt to tamper with the hypervisor or Dom0 image on disk, would render the system unbootable. The attacker, who already compromised the storage domain, could also e.g. delete all the signed and encrypted block devices, again making the system unusable.

But besides DoS attacks the attacker cannot achieve anything else, e.g. cannot read or tamper (in a meaningful way) with the VM private data or with the root filesystem used by each VM (or by Dom0). In most cases DoS attacks are not such a big problem and are relatively easy manageable. Particularly the well known solution against this form of attacks is to make regular backups of data.

7.2. System boot process

Qubes boot process comprises several steps.

1. The first stage is the verified boot process, whose goal is to measure the hypervisor and Dom0 image files, stored on the unencrypted boot partition (`/dev/sda2` on the figure below), before they will get loaded and executed. These measurements, done by Intel TXT, are placed into special TPM registers. If the measurements are correct they will later allow to unseal a secret from the TPM that will be needed to get access to various disk encryption keys.
2. Once the TXT boot is finalized, the next step is to load and start the hypervisor and the Dom0 kernel, and also to run the Dom0's initramfs script. One should notice that both the hypervisor and Dom0 kernel will be started regardless of whether their hashes are correct or not (e.g. because they were tampered

by the attacker somehow)[23]. This is however not a problem, because in order to proceed to the next stages of the boot process the Dom0 init scripts would need to unseal a secret from the TPM needed to decrypt the rest of the file systems. If the hypervisor, or Dom0 image, or initramfs script was tampered, this unsealing would not succeed and consequently the system would not be able to proceed with the boot.

3. Next, the Dom0 initramfs script asks user for the secret -- this could be a passphrase entered via regular keyboard, as well as a secret stored on a smart card. This step is necessary to allow only the authorized user to boot the system (because booting the system involves decrypting of many file systems). The advantage of using a secret stored on a smart card (token) is the resistance to keystroke sniffers. This is described in more detail in the last chapter about attack surface analysis.

   It is very important that the system somehow prove to the user that it is non-tampered so that the user enters the passphrase (or inserts the smart card and enters the PIN) only if he or she is sure that the code that asks for the passphrase is not tampered with. This attack is known as Evil Maid attack. We discuss the possible solutions in Qubes OS against this attack in the next chapter.

   Now, only if the correct (non-tampered) hypervisor and Dom0 kernels, and also correct Dom0's initramfs script have been loaded, and also if the correct user passphrase has been entered (alternatively correct secret read from the smart card), only then the system would be able to decrypt the `keys.gpg` file (located in the boot partition) that contains the keys needed for the next boot stages.

4. Having the `keys.gpg` file decrypted into memory[24], makes it now possible, for the Dom0's initramfs script, to create the storage domain.

5. The storage domain root file system is created out of a separate physical partition (`/dev/sda2` on the figure below), which is encrypted with the key stored in the previously mentioned `keys.gpg` file.

   The encryption of the storage domain partition is not absolutely necessary, because there are no security sensitive secrets stored in the storage domain. One theoretical scenario of abusing a non-encrypted storage partition would involve a physical attack similar to Evil Maid attack, i.e. the attacker that has physical access to the laptop (e.g. can boot it via an arbitrary USB stick, or can remove the HDD and connect to another laptop) could manipulate the software stored in the storage domain. While this would not give any immediate benefits to the attacker, as the storage domain is designed to be security non-critical, and also doesn't have access e.g. to the network, still the attacker could use the, now compromised, storage domain to try to attack other VMs in the system. This could e.g. be attempt by modifying the backends and daemons running in the storage domain, so that they try to exploit potential bugs in the other VMs frontends.

   Of course the same theoretical line of attack could be used when the attacker compromises the storage domain via some exploit, coming from one of the VMs (e.g. the low privileged one). But in that case the attacker must have an exploit to get into the storage domain. In case of an Evil Maid-like attack, however, this step is not needed, making it slightly easier (of course only for the attacker that has physical access to the machine). Consequently, because encrypting the storage partition comes at almost no cost (neither in performance, nor in complexity), there is no reason not to do this.

6. The just created storage domain now mounts the unencrypted partition (`/dev/sda3` in the figure below) that contains various files used to back virtual block devices used by other VMs. All the image files there are either encrypted (Dom0 root file system image, per-VM private data) or digitally signed (AppVM root filesystem image). Thus, there is nothing that the attacker could gain by modifying files on this partition.

   The storage domain can now starts its backend drivers and start providing the virtual root block device to Dom0. The Dom0's initramfs script can now mount the root filesystem for Dom0 and complete the Dom0 boot process, including the start of the X server and the Window Manager.

---

[23] Intel Trusted Execution Technology contains a mechanism called Launch Policy, that can be used to prevent a hypervisor from loading, if its hash is incorrect. This mechanism, however, while looking secure at first sight, doesn't really provide any guarantee that the attacker's code will not run, because the attacker can simply choose to never execute the SENTER instruction. TXT-compatible processors allow to lock down the configuration, so that the CPU would refuse to execute any VMX instructions without prior executing SENTER instruction (entering SMX mode). However, there is nothing that would require the attacker to use VMX instructions in his or her modified hypervisor. Consequently, we don't make use of the TXT Launch Policy, as we don't believe it offers any security improvement over the "bare" TXT.

[24] The keys.gpg file would likely have to be part of the initramfs file, to make it possible for the init script to read it. This is a minor technicality though.

7. The storage domain is now ready to provide virtual block devices to any new VM created by Dom0.

This concludes the system secure boot process.

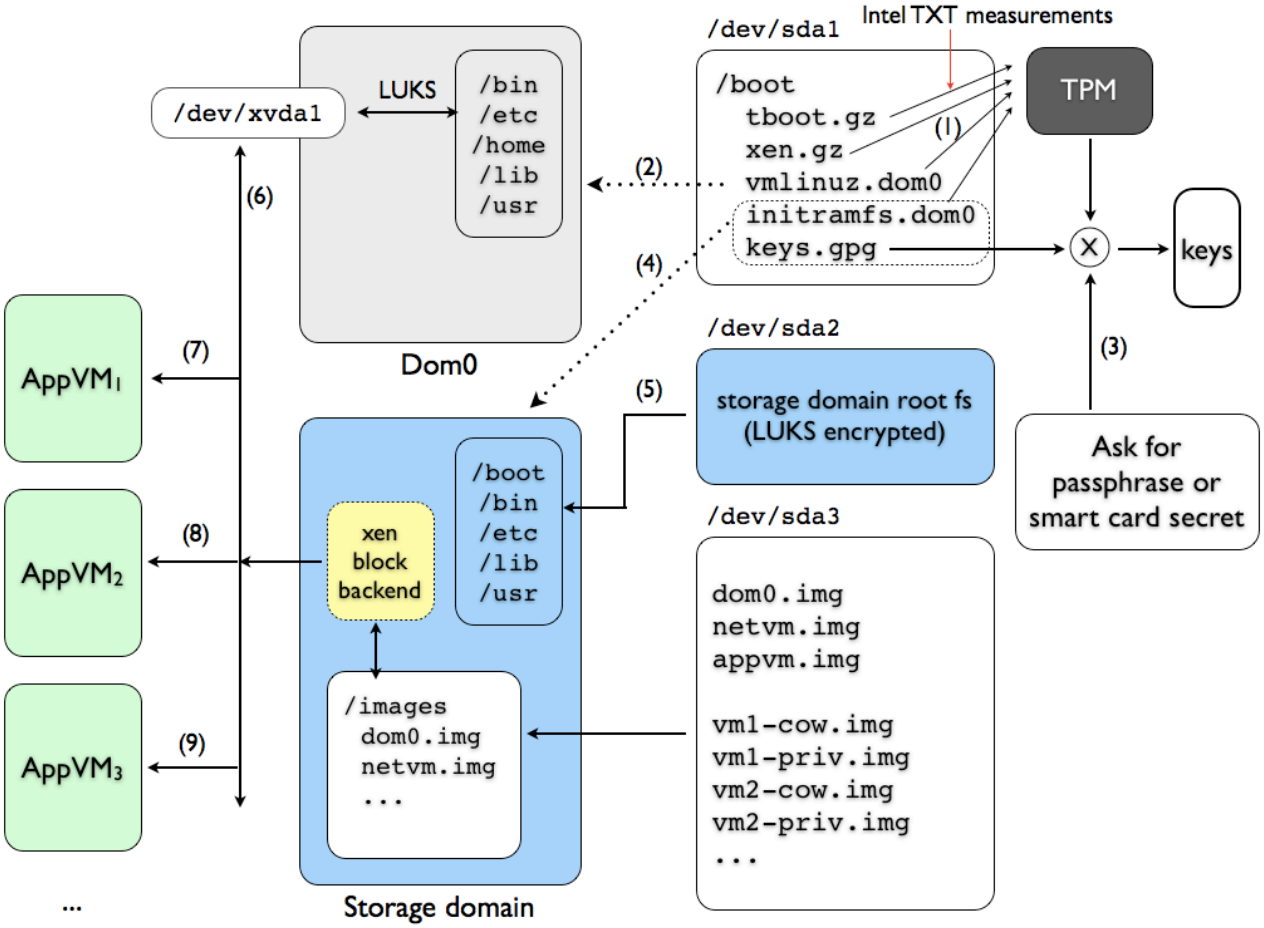

Figure 2. The Qubes OS secure boot process.

7.3. Evil Maid Attack prevention

For laptop users, especially those who travel frequently, it is very important for the system to offer effective prevention against low-cost physical attacks. One such class of attacks, very simple and cheap to conduct, is called Evil Maid Attacks[25]. The attack works by subverting the boot code (e.g. the initramfs script) that asks for the user passphrase or reads the secret from the smart card/token. Even though the TPM-based verified boot process would not allow to later boot the system in case it was “evilmaided”, still the attacker’s code can obtain the user’s passhprase (or smartcard secret) and store it somewhere on disk for later retrieval (e.g. the next time the user leaves the laptop in a hotel room).

The way to prevent the Evil Maid-like attack from succeeding is to provide the user with the ability to find out if the system boot code (the one that asks for the boot passphrase) has been tampered or not. This means the system should somehow authorize to the user, before the user enters the passphrase needed to continue the system boot. This is, however, not such a straight forward task, and several approaches are possible, each providing different tradeoff between user’s conveniance and security.

[25] The term *Evil Maid Attack* has been proposed by one of the authors of this document in a talk about Trusted Computing (http://invisiblethingslab.com/resources/misc09/trusted_computing_thoughts.pdf), and later has started been used by other researchers and vendors to describe this class of attacks. More information about practical Evil Maid Attacks the reader can find in another authors’ article: http://theinvisiblethings.blogspot.com/2009/10/evil-maid-goes-after-truecrypt.html

The “favorite picture” approach

The most obvious approach is for the system to display a secret passphrase on the screen, before asking the user for his or her passphrase. This authentication passphrase would, of course, be normally encrypted, and the decryption key should be sealed into the TPM, so that only if the non-tampered software booted, only this software could unseal the secret from the TPM, decrypt the authentication passphrase, and display it on the screen, proving to the user that it is original software indeed (as it was allowed to retrieve the secret from the TPM) and that the user can safely enter his or her passphrase (this time proving he or she is the authorized user).

The above scheme is however seriously flawed. Namely, if the laptop is left unattended, e.g. in a hotel room, an attacker (e.g. the hotel maid) might simply boot the laptop and she would see the secret authentication passphrase. She could now prepare a version of the Evil Maid attack that would contain this passphrase hardcoded, which would trick the user into thinking that he or she is interacting with an original software, while in fact this would be an Evil Maid password sniffer.

The simple idea of how to improve this scheme is to use a picture (preferably a photo), rather than a passphrase for authenticating. The user could e.g. take a picture of himself or herself during the installation of the system, using the built in laptop camera. This picture could be further encrypted and the decryption key sealed into the TPM. Now, only if the non-tampered software has booted, it would be able to decrypt and display the user’s picture on the screen.

Here, the assumption is that it would be much harder for the attacker to copy the image from the screen. Obviously taking a picture of the laptop screen would not result in the same picture, as many effects like e.g. the scene exposure would likely be imperfect. The attacker might still try to use the VGA/DVI output port in the laptop to capture the “secret” authentication picture, but this makes the attack less trivial. Additionally, it might perhaps be possible on certain system to disable the VGA/DVI port during the boot, so that it could not be used to copy the authentication picture.

USB token

Much more secure solution could be achieved using a USB token or a smart card. Such devices typically expect the authentication PIN to be sent to them first, before allowing the software to e.g. read a secret stored on the smart card file system or to use one of the RSA private keys stored there.

We could combine the secret retrieved from the TPM (key_tpm), that can be retrieved only if correct software has booted, with the passphrase entered by the user (key_user), creating together a PIN used for the smartcard authentication:

$$PIN = H(key\_tpm \oplus key\_user)$$

Now, only if both the software was non-tampered and the user knew the passphrase, only then the secret from the smartcard would be returned to the software. This way we achieve both the system and user authentication with the help of just one additional device and requiring the user to provide only one passphrase.

Please note that even if the system is “evilmaided”, the Evil Maid sniffer cannot sniff anything valuable for the attacker -- it can only sniff the `key_user`, which is not useful alone for the attacker, if the attacker doesn’t also own the token. However, if the attacker can steal the token from the user after performing such an attack, then the attacker would be able to boot the system, as the attacker would know the `key_user`, sniffed by the Evil Maid. Still, this approach provides very reasonable level of security, probably satisfactory for most users.

Additionally, by using a token for authentication, the user can protect against external key stroke loggers, like e.g. hidden cameras.

OTP passwords

Yet another solution against Evil Maid attacks is to use one-time-passwords (OTP) for authenticating the software that asks for the passphrase[26].

The idea is the following. During installation of the system, a list of one time passwords is generated (and saved in a `otp.txt` file). This file is then encrypted with a secret (creating `otp.gpg` file), and the secret is

[26] The reader should note that the OTP passwords can also be used to authenticate the user to the system. Here we focus on the inverse scenario.

sealed into TPM (so that only if the correct software was booted it can retrieve this secret from the TPM). Also a copy of the list in plaintext (`otp.txt`) is given to the user.

Now each time the user boots the machine, the system presents the next password from the OTP list. The user can verify this using the list.

The obvious practical problem is how the user should maintain the `otp.txt` list. One possibility is to use a scratch card, but this has a very serious limitation of allowing just a few tens of OTP passwords per card.

Much better solution seems to be to use OATH-compatible[27] token. The protocol might look like this:

1. User boots the machine
2. The system asks for the next OTP password
3. User enters the password that the token displays (OTP1)
4. The system verifies if OTP1 is within the allowed window[28], and if it is, the system presents the next OTP password, OTP2, that directly follows the OTP1
5. The user generates a new password, e.g. by pressing a button on the token. If this new password is equal to OTP2 that just has been presented by the system, the user assumes the system software is not tampered and can proceed with providing the secret passphrase needed to boot the system.

7.4. Discussion: Intel TXT advantages over Static RTM

In Qubes architecture we decided to relay on the Dynamic Root of Trust for Measurement (DRTM), as implemented in Intel Trusted Execution Technology, rather than on the Static Root of Trust for Measurement (SRTM). There are two primary reasons why we believe that TXT is superior over Static RTM approach: the DMA protection, and the lack of requirement for maintaining the long chain of trust since the system boot.

DMA protection

First, Intel TXT provides automatic DMA protection for the measured code. This is very important, especially in Qubes architecture, where we allow unprivileged domains (the network and storage domains) to directly interfere with the hardware (the network card and the disk and other mass storage devices).

We should anticipate that some of the devices would allow the software that controls them to program them (or perhaps to even reflash the firmware of the device), so that they perform arbitrary DMA transactions. During the normal system operation this is not a problem, because Intel VT-d is used to contain those devices from doing DMA to anything outside the address space of the VM they are assigned to. However, early in the boot process, there is no VT-d enabled. In that case the maliciously programmed device could interfere with the Static RTM measurements, using a race-condition attack (change the code a moment after it got measured but before it gets executed). Of course implementing such an attack is non-trivial, as one need a good timing in order to succeed, nevertheless this seems possible.

Intel TXT prevents such attacks, because the SENTER instruction sets DMA protection for all the code that is about to be measured and later executed.

The shorter chain of trust

Another advantage of using dynamic root of trust for measurement over static root of trust, is that it is not required to maintain a long chain of trust that would cover all possible code ever executed since the start of the machine. This brings the advantage of not being forced to trust e.g. the platform BIOS code, or the PCI EEPROM code.

On the other hand, TXT is also not entirely self contained when it comes to the chain of trust. Particularly, Intel TXT requires that we either trust the platform SMM handler (as it can survive the secure launch process), or alternatively, that we provide a special hypervisor needed to contain the potentially buggy SMM, the so called SMM Transfer Monitor (STM). So, it's either the SMM handler, or the STM hypervisor, that should

[27] The OATH specification can be found at ftp://ftp.rfc-editor.org/in-notes/rfc4226.txt

[28] It's usually assumed that the token OTP counter might get out of sync with the system counter (e.g. because the button on the token used for generating new OTP password might be incidentally pressed by the user). Thus a certain window is assumed, that allows the user to enter any token whose index belong to the following set: N+1, ..., N+n, where *N* is the last used token index, and *n* is the maximum window size.

be included in the chain of trust for TXT. Still, this sounds like less code to trust than in the case of the classic SRTM.

7.5. USB and DVD support

The storage domain, besides the obvious block backends and direct access to the disk controller, also has access to other storage device controllers, in particular to the USB controller (e.g. for USB flash storage) and CD/DVD controllers.

Naturally, it is assumed that whenever the user would decide to record or access security-sensitive data stored e.g. on a USB flash drive, the user would use encryption to protect the data from being potentially viewed or manipulated by the storage domain. This indeed should be an obvious choice, as no security sensitive data should ever be stored on an external storage medium, such as USB flash driver, or DVD disc, in an unencrypted form.

The most straightforward solution here, is for the storage domain to export (via usual backend device) the virtual block device backed up by the physical block device attached to the USB port, e.g. the USB flash drive or a DVD disc. The individual VM, that the user decides to assign the virtual block device to, would then be able to mount the virtual block device, e.g. using dmcrypt in order to encrypt/decrypt its content.

Another solution is for the storage domain to export a USB device, using the USB PV backend and let the VM handle the USB protocol by itself. This solution has an advantage of making some non-storage USB devices available to VMs.

In some cases it might not be preferred to use the USB backend in the storage domain however.

One example of such situation is the use of a smart card for storing the user PGP private keys. Even if the private PGP key never leaves the smart card, still the user that has a control over the USB driver/controller, can sniff the session keys being decrypted by the smartcard, or even mount the MITM attacks, pretending to be a legitimate user (obviously the attacker would know the PIN used for authorization, if the attacker controls the USB driver) the attack would be able to ask the smartcard to decrypt any attacker-provided secret.

We should stress, that such a hypothetical attack as the one described above requires that the attacker gains the control over the storage domain first. In other systems, USB drivers (or backend) are always placed in the privileged domain, enlarging the potential attack surface on the whole system. Qubes places USB backend in an unprivileged domain, hence minimizing the attack surface on the TCB. Still, because we would like to assume that the storage domain compromise to be security non-critical, it's desired to think about possible workarounds against the presented theoretical attacks on the USB.

Fortunately contemporary computers, even laptops, have many USB controllers (each visible as a separate PCI device) and it is not uncommon for different physical USB ports to be connected to different USB buses, which are rooted in different USB controllers.

It is thus possible to assign selected USB controllers (for the select physical USB port(s) to certain AppVMs, e.g. the AppVM that uses that PGP smartcard, using direct device assignment via VT-d, in the same way as the USB controllers are normally being assigned to the storage domain.

7.6. Making backups

Making full backups using storage domain

The storage domain can be used to perform full backup of all the user data. For this, it is just enough to copy all the (encrypted) `vm*-priv.img` files onto a backup medium, e.g. DVD disc[29]. Additionally the user should also keep a backup copy of the keys used to encrypt each of the AppVM's block devices. This cannot be simply a copy of the `keys.gpg` file (from the Dom0's `/boot`), because this file is encrypted with a key that is a product of the TPM secret and the user passphrase. Obviously, the user is only expected to remember the passphrase and not the TPM's secret, which e.g. can be lost if the laptop gets stolen or damaged. It's thus desirable for the Dom0 to maintain an always synchronized copy of the `keys.gpg` file, perhaps named `keys_backup.gpg`, encrypted only with the user passphrase, but preferably with a different passphrase

[29] This should be done after powering down all the AppVMs, or at least after syncing all their filesystems.

than the one used for booting the machine[30]. This (encrypted) file can then be automatically attached to each backup copy made by the storage domain.

Making incremental backups (using Dom0)

The storage domain, however is unable to make an incremental backups of the user data, because the storage domain cannot decrypt the per-VM block devices and cannot see the actual files, that are needed for effective incremental backups, preferably with a history of each file modifications (old revisions of replace files are kept in appropriately named directories, e.g. `backup-YYYYMMDD`).

The only entity that can make incremental backups of all the user data is Dom0, because only Dom0 has access to all the keys used by all the AppVMs. The backup process initiated from Dom0 should be pretty straightforward. It should first create an encrypted filesystem on the backup medium offered by the storage domain (e.g. removable disk connected to the storage domain and exported as block device to Dom0), and later, after powering down all the running AppVMs, mount all the VM private block devices and copy them onto the storage medium, using a popular tool for making incremental backups, like e.g. `rsync`.

The whole backup filesystem should be encrypted with user-provided passphrase.

[30] We would like to avoid the attack when the user's boot passphrase gets somehow recorded, e.g. using the keyboard sniffer, and then the attacker uses it to decrypt the `keys_backup.gpg` file, bypassing the trusted boot protection.

8. Analysis of potential attack vectors

There is no such thing as 100% secure OS, at least on x86 COTS hardware. The software and hardware is simply too complex to analyze for all potential errors in case of x86 platforms. Qubes architecture is all about minimizing the attack surface, so that we could focus on auditing only the most important parts of the system, instead of every single application running there.

Below we present the summary of theoretical attack vectors. We believe all of those vectors are very unlikely and expensive (in terms of developing an exploit for a potential bug). Nevertheless they represent potential attack points. We should stress that other, mainstream OSes, like Windows or Linux, would have such a list too, just much longer -- tens or perhaps hundreds of pages long.

Below we assume the attacker compromised one of the VMs, and discuss potential attack vectors the attacker might use in order to compromise other VMs.

8.1. 1-stage vs. 2-stage attacks

We can divide potential attacks on Qubes OS into two groups:

1. one-stage attacks, that require the attacker find only one bug and exploit it in order to compromise the system (e.g. steal data from other VMs or compromise them),
2. two-(or more)-stage attacks, where the attacker would have to chain two different exploits targeting two different bugs, in two different system components. E.g. one exploit to “get to” the storage domain, and another attack to compromise Dom0 out of the storage domain.

It should intuitively be obvious that the likelihood of a 2-stage attack should be significantly smaller than the probability of a successful 1-stage attack, nevertheless we include them in the attack surface analysis as well.

8.2. Potential attacks vectors from any VM (1-stage attacks)

Potential bugs in the hypervisor

The hypervisor is the most privilege element of the system. Any attack on the hypervisor (e.g. exploiting a buffer overflow in a hypercall) allows to fully compromise the system. As part of the Qubes project we try to minimize the possibility of hypervisor exploitation as it has been described in one of the chapters earlier.

So far there has been only one overflow discovered in the Xen hypervisor, and it wasn’t in the core Xen code, but rather in the optional code in the special extension to the Xen hypervisor called FLASK security module[31]. Incidentally this bug has been discovered and described by one of the authors of this document[32].

We expect the hypervisor code footprint to be between 50,000 - 200,000 LOC (the smaller number is what we expect to have after we start sliming down the current Xen hypervisor, which is closer in size to the latter number).

Potential bugs in the Xen Store Daemon (in Dom0)

Xen Store Daemon runs in Dom0 and its potential compromise can be fatal, as the attacker can later gain full control over Dom0. It should be determined if Xen Store Daemon can be run under restricted user account, rather than using the root account, which might provide a minor level of additional security[33].

There is also a theoretical possibility of logical errors in the Xen Store Daemon implementation. E.g. an unprivileged domain might exploit a hypothetical bug in the daemon to read certain key values that are not to be read by this VM, which might contain e.g. the keys used to encrypt the block device used for inter-VM file exchange, in which case the attacker, that also controls the storage domain, could steal the contents of the files being exchanged between the VMs.

---

[31] This is a good example to support the thesis that the less code in the hypervisor the better.

[32] http://invisiblethingslab.com/resources/bh08/part2-full.pdf

[33] Obviously we don’t assume that OS-level isolation is really that effective, hence the use of the word “minor”. It’s expected that a skilled attacker can find and exploit a kernel overflow to escalate from user to root in Dom0.

Xen Store Daemon uses very simple way of communication, built on top of the Xen shared memory and event channel mechanisms, and also the protocol used is very simple, which should minimize the chances of such attack vectors.

Nevertheless, the Xen Store Daemon, being a very prominent target, should be thoroughly evaluated before commercial use of the system.

Potential bugs in the GUI daemon (in Dom0)

The GUI daemon also runs in Dom0, and its successful exploitation is always fatal (see discussion in the chapter on GUI).

The GUI daemon uses the same underlying communication mechanism as the Xen Store Daemon uses, and similarly simple protocol. It's thus believed that the likelihood of a bug in the GUI daemon implementation should be very small.

Potential CPU bugs

Modern processors are one of the most complex creations made by humans. It is hard not to think that there might be a bug in a CPU that could e.g. allow for unauthorized ring3 to ring0 privilege escalation. Interestingly, no such bugs have ever been publicly disclosed. The closest of what we can think of here would be the SMM caching attack, incidentally discovered by the authors of this document (and also independently by Loic Duflot) a few months ago[34].

One should, however, note that a CPU bug would not only be fatal to Qubes OS -- it would be fatal to any PC-based OS.

There is not much we can do about potential CPU bugs, besides hoping that the CPU vendor did its job well. It's not possible to "audit" the CPU, for obvious reasons.

8.3. Additional potential vectors from driver domains (2-stage attacks)

All the theoretical attacks considered in this paragraph are 2-stage attacks, which means the following attacks require two different exploitable bugs, in two different system components, at the same time, in order to succeed.

Potential VT-d bypass

If the attacker was able to initiate an arbitrary DMA transaction by programing a device assigned to one of the driver domains (e.g. the network card), the attacker would be able to compromise the system. Qubes architecture relies on Intel VT-d to prevent devices assigned to driver domains from performing DMA to memory not belonging to their domain. An attacker that could bypass Intel VT-d technology, e.g. by compromising the Memory Controller Hub, could compromise the system.

So far no attack against Intel VT-d has been publicly presented. We anticipate such an attack would be very non trivial to come up with. Also one should note, that in order to use it against Qubes OS, the attacker would first need to compromise one of the driver domains, by exploiting a bug in the software running there (e.g. a bug in the network driver, or network backend, or disk backend).

Driver domain to MBR attacks

An attacker that controls one of the driver domains can theoretically find a way to re-flash the firmware in the device being assigned to the domain, e.g. the network card. The attacker can then attempt to program the device in such a way that it waited for the system reboot and then, using a malicious DMA, infect the system boot loader or MBR. This, of course, is not enough to subvert the system, as Qubes uses secure boot process implemented with the help of Intel TXT.

Nevertheless, the attacker can now attempt to perform most of the attacks that are otherwise attributed below to storage domain only (except for the attacks on VM storage frontends, see below, that is possible only from the storage domain).

Potential delayed DMA attack on Dom0 (attack mostly from storage domain)

If the attacker controls the storage domain, he or she can infect the MBR or boot loader.

[34] http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf

The infected boot loader can, in turn access the graphics card or the audio card device (because early in the boot, before executing SENTER, there are no VT-d protections in place), and program it in such a way (perhaps by re-flashing its firmware) that it will wait until Dom0 start and then performs a malicious DMA attack, that e.g. installs a backdoor in Dom0.

Such an attack would work because the devices assigned to Dom0, i.e. the graphics card and the audio card, are allowed DMA access to the whole Dom0 memory.

In order to prevent such hypothetical attacks, the hypervisor should allow domains (via an additional hypercall) to set additional per-VM VT-d protections. Obviously the hypervisor should make sure that the domain can only set more restrictive VT-d protections, rather than the other way round.

Potential TXT bypass (attack mostly from storage domain)

Intel Trusted Execution Technology (TXT) allows to have unprivileged security non-critical storage domain. If the attacker was able to bypass Intel TXT (using a software only attack), the attacker, if already compromised the storage domain, could compromise the whole system e.g. by modifying the hypervisor or Dom0 image on disk and rebooting the system.

So far two attacks on Intel TXT have been publicly disclosed. Incidentally both of the attacks have been discovered by the authors of this document[35].

Untrusted input to Dom0 from BIOS, e.g. malicious ACPI (attack mostly from storage domain)

Xen hypervisor and Dom0 kernel process several BIOS-provided inputs during boot. This includes e.g. the BIOS e820 memory map, as well as the BIOS-provided ACPI tables. All the BIOS-provided input should be treated as untrusted and handled with great care. One example of an attack exploiting this vector, is the already mentioned ACPI AML methods that are executed by Dom0 kernel as part of power management activities. The attacker can subvert the original ACPI tables and provide malicious AML methods for Dom0 to execute, which can lead to Dom0 compromise.

More subtle scenarios include exploiting potential input processing errors in the hypervisor or Dom0 kernel, e.g. buffer overflows resulting in the BIOS-provided data being intentionally miss-formatted (e.g. too long fields in the e820 map).

All the Xen and Dom0 kernel code that process BIOS-provided input should be thoroughly reviewed.

All those attacks assume that the attacker is controlling the execution early in the boot process. This can be a result of the attacker gaining control over the storage domain earlier.

Potential encryption scheme bypass (attack from storage domain)

For the storage domain to be security non-critical, it is important to trust the crypto scheme used for VM private block device encryption and the shared root filesystem signing to be correctly implemented. Otherwise, the attacker who gained control over the storage domain, could either steal the VM private data or inject malicious code for other VMs to execute.

Potential attacks on storage frontends (attack from the storage domain)

An attacker that controls the storage domain might attempt to exploit hypothetical bugs in the other VM's block frontends that communicate with the block backend hosted in the storage domain.

The block fontend drivers are very simple and thus it is expected that the likelihood of a potential bug there is rather small.

Potential attacks on VM networking code (attack from the network domain)

An attacker that controls the network domain might attempt to exploit hypothetical bugs in the other VM's network frontends, or their regular TCP/IP stack. The Xen network frontends are very simple and thus the likelihood of a bug there is expected to be rather small. The regular TCP/IP stack (in contrast to e.g. WiFi stack or WiFi driver) is usually a very well tested code, and thus it should be hard to find and successfully exploit a bug in this stack (although more likely than a bug in the frontend).

[35] http://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20paper.pdf and http://invisiblethingslab.com/resources/misc09/Another%20TXT%20Attack.pdf.

8.4. Resistance to attacks from the (local) network

Because all the world-facing networking code executes in the unprivileged network domain, there is no direct attack surface from the local network on the system.

There is, however, a possibility, that the attacker, who already compromised the network domain, e.g. by exploiting a bug in the WiFi driver, can now try to exploit (another) bug in the networking stack of one of the AppVMs, as explained in the previous paragraph. That would however requite a 2-stage attack.

8.5. Resistance to physical attacks

Below we analyze the low-cost physical attacks that could be performed in a matter of minutes against a laptop left e.g. in a conference or hotel room, and how Qubes OS can resist them.

Resistance to Evil Maid attacks (trusted boot bypass)

As it has been described in the previous chapter on storage domain, Qubes boot process is designed to withstand even sophisticated Evil Maid attacks.

Resistance to keystroke sniffing attacks (e.g. hidden camera)

Qubes OS can also resist simple key sniffing attacks that are targeted toward capturing the passphrase used to boot the system. This can be achieved by the already mentioned trusted boot process that uses smartcard to obtain the user secret, rather than the keyboard.

Relaying on OTP passwords to avoid keystroke sniffing attacks can be tricky...

We can imagine a naive implementation, where the disk master encryption key is re-encrypted with next OTP password after each successful login. Thus, in order to unlock the system, the attacker must enter the next OTP password from the list, and the previous OTP password(s) would not allow to decrypt the filesystem.

However, the attacker can, when having physical access to the system for some time (perhaps about 1-2 hours), create a complete copy (dump) of the hard disk content of the target laptop. Then, the attacker, after capturing the next valid OTP password, can use it on the copy of the hard disk he or she just created, effectively “replaying” what the user just did when entered the OTP password. So, even though, on the users laptop the next OTP password would be required to unlock the system, the system running from a copy of the disk that the attacker has requires the same password that the user just entered (and that we assumed has just been sniffed).

One possible solution could be the use of a remote server that would be unlocking the master encryption password and that would be taking care about incrementing the OTP password’s counter. However the requirement of always having network connectivity in order to unlock the system might be unacceptable by many users. Additionally, the need to establish network connection so early in the boot stage would require bringing up the network domain, which in turn would require also starting the storage domain, and this all before even obtaining a passphrase/OTP password from the user. Additionally the client program that would be verifying the OTP password with the remote server (and obtaining the master disk encryption key) might expose system to additional attack vectors from the network (including attacks on the crypto protocol used for this process). For all those reasons the solution with a remote OTP authorization is not recommended for Qubes OS.

Much better solution seems to be to use the TPM’s non-volatile storage to store the disk decryption master key, or the counter for the OTP key. Thus, even if the attacker stole the laptop (which implies the attacker would have access to the TPM), the attacker would be unable to boot the system using the once-used OTP key, even if restored all the disk contents to the previous state, recorded before the user entered the key that was sniffed.

Resistance to DMA attacks (e.g. malicious PCMCIA)

Due to the extensive use of VT-d, Qubes OS should be resistant to many runtime DMA attacks, e.g. when the attacker inserts a specially crafted PC Card (also known as PCMCIA) that is supposed to use DMA in order to compromise e.g. Dom0 and unlock the computer, or just steal some secrets from the memory.