

KDAB's Software Development Best Practices

Software Updates Outside the App Store

Andreas Holzammer | Senior Software Engineer, KDAB

Leon Matthes | Software Engineer, KDAB

Lieven Hey | Software Engineer, KDAB

Eystein Stenberg | Chief Technology Officer, Mender



When building an application distributed through the Google or Apple app stores, much of the heavy lifting in delivering updates is handled for you. However, there are many circumstances where these consumer app stores don't apply: embedded devices, industrial desktop applications, and dedicated-use tablets, among others. If you're building one of these systems, your software will necessarily live outside of these app stores. So how do you design and build a system that can effectively deliver and update your software?

The challenge isn't just updating the application itself. Often, the entire execution environment needs to be replaced, including the operating system, drivers, firmware, libraries, FPGA programming files, DSP firmware, system configuration files, and any other application-dependent components. The update procedure must be designed very carefully to keep customer downtime to a minimum and avoid situations that could disable the system.

To learn what engineering teams should consider when planning their update strategy, we've consulted with our internal experts as well as our partner Mender that specializes in over-the-air (OTA) updates. Each of us has contributed best practices and insights gleaned from many customer engagements and difficult engineering challenges. This document primarily focuses on embedded devices because they often require the most work to address. However, many of these considerations is also relevant to other software systems that operate outside the main app stores.

1. Update Content

1.1. Packages

One of the first decisions when designing an application update scheme is determining the scope of the updates. A tempting choice might be package-style updates. In other words, you could set up your system to periodically run an [apt](#) or [yum](#) script to pull down the most recent packages. This method is attractive since the technology is pre-existing, reliable, free, and seems to do what you need: updating

binaries and configuration data with the most recently tested packages for in-field systems.

However, some significant drawbacks come with using package managers for updates, which is why we don't recommend using them.

- **Fragmented updates.** Package managers, by design, update portions of the system, rather than the entire system at once. While this might save on download time, this approach makes it extremely difficult to ensure consistency across in-field devices. Given that there's no certainty about which devices update which packages and when, it introduces unpredictability. For instance, if you've deployed a dozen packages on the system, the potential combinations of the system image can grow exponentially. Consequently, it is nearly impossible to test all possible variants of the system. Replicating issues that match a user's specific configuration also becomes unreliable.
- **Limited reach.** While package managers work well at updating software that's deployed within a package, they cannot manage the many parts of a running image that are outside – board support package, operating systems, and device drivers. You could attempt to wrap these parts with a custom-built package, but even so, updating certain files with a package manager will remain unfeasible.
- **Challenging rollback.** What happens if an update causes problems and the package must be rolled back? While package managers can downgrade by asking for a specific version of the package, interwoven package dependencies make this a relatively delicate operation that cannot be thoroughly tested. Considering that rollbacks are rare and emergency driven, they must be dead-simple and reliable. Relying on a package manager to rollback is risky and testing it is impractical.
- **Lack of segmentation.** Updates through package managers will refresh every single device. However, in practice, there's often a

need to segment your updates based on different geographies, user categories, product models, beta testers, etc. The one-size-fits-all approach of package managers doesn't accommodate this granularity.

1.2. Containers

Containers offer a broader scope for system updates compared to packages, which can increase their reliability and testability for software updates. However, just like packages, containers suffer from similar issues such as multiplying permutations of the system and an inability to update the entire system image.

However, there are circumstances where containers might be an appropriate solution. For instance, if you're using a single container on a stable OS installation that seldom changes, a container-based update could replace all the components that need to be regularly updated. Containers might also be the best alternative when you don't have full control over the system, like on an application-dedicated tablet.

However, a container still can't update the full software stack, and because of this inability to update the OS or other low-level components – [despite being great developer tools](#) – containers aren't well suited to be the sole component of a robust system update strategy.

1.3. Firmware Images

Our preferred way for delivering updates involves sending a complete bootable firmware image to the device. This is especially the case for embedded systems and other devices where you have full control. A complete image update lets you test the entire system without worrying about the many permutations of the running software components. Since the whole firmware image is replaceable, this method also offers access to low-level components that other

methods don't, such as patches to the operating system or updating device drivers. Implementing complete firmware image updates also makes it straightforward to tag the full set of files in your version control system associated with a particular build.

Since you must always have a bootable image even when a new update fails or is in progress, you typically need two bootable partitions. Once an update is downloaded and ready-to-go, a flag is toggled to indicate the system needs to boot into the freshly downloaded partition on next reboot. This is called an A/B system, since the two bootable partitions are usually labeled "A" and "B". Using an A/B strategy allows the system to revert to a known working state in case of a failure (we'll discuss how in section 2.1), and it's what we recommend.

1.4. Hybrid system

If you have an A/B update system to address the entire firmware, you can also consider adding the flexibility of container-based updates on top. Container update payloads are smaller, saving transmission cost, time, and bandwidth. Furthermore, new applications can be pushed out with less disruption to the user since a full reboot isn't required to activate the new load. A hybrid update scheme lets you roll out more agile application updates every couple of weeks while reserving full system updates to once a quarter or twice a year. These full updates refresh the lower-level components and bring all in-field systems to the same baseline version. The disadvantage to a hybrid approach is the necessity to develop (and/or acquire) and support two independent update mechanisms.

1.5. Compression vs Deltas

To minimize the amount of time and bandwidth required to download update payloads, it's recommended to use some type of compression scheme on your update files. Employing methods

such as compress, gzip, or bzip2 can reduce file size by at least 20-30%, depending upon how much redundancy is in your image. It might also be beneficial to try some trials using your specific image files to see [which algorithms achieve the best compression](#) in both average size reduction as well as decompression speed.

A delta algorithm calculates the differences between the last image and the update, creating significantly smaller files than standard file compression does. Although the payload size differs depending on the size and number of changes between the old and new image, Mender estimates that a delta file can typically achieve a compression ratio of 85 - 90%. Even though delta technology isn't widely available in open-source platforms, the potential savings in bandwidth, update download time, and application update speed may make it an avenue worth exploring.

2. File Systems

2.1. A/B Boot Partitions

As discussed, an A/B boot partition is a straightforward and reliable way to structure full system updates. In the A/B boot setup, the system has two identically sized boot partitions. The bootloader has a flag outside of these partitions (in either dedicated persistent storage or a mutable data partition) to indicate which partition to boot from. New updates are downloaded into the inactive partition. Once an image is successfully downloaded and validated, the boot flag is toggled to indicate that the newly updated image should be used for subsequent reboots. This approach offers several benefits:

- New updates can be downloaded and written in the background while the system is fully operational.
- The downloaded image can be validated before the system attempts to use it.

- There should always be enough space to store new update payloads.
- If any complications arise with an aborted, incomplete, or corrupt image download, the system will revert to rebooting from the verified stable image.

The biggest limitation is that each partition contains a complete image of the executable code, so this configuration requires up to twice as much flash. However, this isn't a significant drawback; to deliver a robust and reliable update procedure, an equivalent amount of extra flash is essential regardless of how you structure your system.

Background Writes

Downloading new update images in the background minimizes user interruption. While this works great for phones and laptops, this is something worth double-checking in an embedded system. It's advisable to carefully test the device's ability to receive updates, especially while it's heavily loaded with other tasks.

This is because the default scheduler for most Linux variants is "fair", which means it can inadvertently interrupt critical tasks, especially when network and/or disk I/O bandwidth is consumed by heavy update downloads. To make the whole system resilient under fluctuating loads, it's vital to consider thread priorities and algorithms on a system-wide basis. Linux offers tools like SCHED_FIFO, SCHED_RR, and SCHED_DEADLINE for this. Real-time operating systems (such as QNX, VxWorks, or FreeRTOS) can set critical tasks to a higher priority to avoid this type of background thread interruption.

2.2. Data Location

When using an A/B dual boot configuration, the best location for the data is a dedicated partition shared between the A and B partitions. This partition should have all data that's modified at run-time, including items like user configurations, accounts and passwords, databases, and logs. Leave out static data that your application uses, like default configurations, initial databases, help systems, multimedia files, neural network models, and so on. All

your read-only data should reside in the executable A/B partitions.

Do you need two data partitions – one used by the A partition and one used by the B partition? Not really, and it doesn't offer any benefit. Two partitions would require duplicating data from the live configuration to the new one once you've successfully downloaded a new image. Otherwise, any user changes would be lost. Maintaining identical content in two places just takes up space and adds time to the update process.

2.3. Data Structures

No matter where you put your data, it's imperative to maintain backward and forward compatibility in your data structures, including database schemas, configuration files, data formats, and data filenames. It is impossible to avoid all data format changes when modifying code to fix bugs or adding new features, but make sure that any data changes are compatible between software versions. To start addressing potential issues:

Create classes that wrap data files so that software is insulated from the data's on-disk representation.

When reading JSON or XML files, don't error out when encountering unrecognized tags, just ignore them. This gives you the flexibility to add new data types and fields in the future that are still compatible with older code.

The same applies to proprietary binary files. Include features like version numbers, field type identifiers, and size fields that allow data chunks to be skipped over if they aren't understood by the code.

Add code that can recognize and convert older data structures to a newer format after they have been read.

Changing data formats and structures significantly complicates testing, so in many cases it's probably better to defer outright

changes to data formats.

2.4. Switching the Bootable Partition

Only flip the bootable partition from A to B or vice versa once you've fully validated the downloaded image. Your download images should incorporate a verifiable hash or checksum to ensure the content was downloaded correctly. Flipping the bootable partition from A to B or vice versa must be atomic (like a single flag in a dedicated flash sector) so that if the process is interrupted, the system will still boot into the old partition.

What if something goes wrong in the new partition? Our preferred solution is to allow the boot logic three attempts to successfully reboot; if the new image can't be booted in any of those attempts, then revert back to the original partition. One way to do this is to use a hardware watchdog. The bootloader programs the watchdog length for just beyond the maximum duration of a successful boot under normal circumstances. The application turns the watchdog off once it starts executing. This way boot failures (detected by the application failing to reset the watchdog) can be automatically caught by the hardware, forcing a system reset.

2.5. Rescue Partitions

Sometimes you may need to clear the device to a "factory reset" state. This acts as a "safe boot" option, allowing the user to restart a faulty device regardless of corruption or faults in the update partitions. This can be done with a rescue partition, which provides a static bootable image untouched by updates. Booting into the rescue partition wipes the device data and downloads the latest update.

While this functionality is a bit out of scope from most OTA implementations, it's still an important part of the whole update

lifecycle. Whether or not you need a rescue partition depends on various factors:

- Availability of reserve flash
- A user-initiated way to invoke it
- Potential resale of the device
- Presence of sensitive user data
- Support reasons for adding this capability

Booting into a rescue partition can prevent the device from bricking if something goes severely wrong with the bootable images. This is often a last-ditch effort since it wipes off any user data. This also makes it an effective way to remove all private data off the device for resale or disposal. Support staff can also leverage a factory reset option if the device is malfunctioning in a way that can't be corrected.

For user-initiated factory resets, a dedicated, recessed switch might be required – something you want to determine in the initial stages of hardware and enclosure design.

3. Hardware

3.1. Write cycles and wear-leveling

Solid-state drives use NAND Flash, which has a limited number of program-erase cycles per block before it can start to fail. Under normal circumstances this isn't a cause for concern as the flash hardware controller takes care of wear leveling, automatically shifting blocks into new locations to evenly distribute wear and proactively prevent issues.

However, this results in a gradual loss of free space as blocks become unusable. This is something to consider if your application is very actively writing to disk. Although most file

and database operations are buffered in memory by default, continuous flushing of data to disk – like when continually logging or capturing data – can accelerate the degradation process. Logging is an instance where this happens frequently (since you want the on-disk content to reflect the last thing the application did before it crashes), so many loggers will flush after every new log. Consider disabling this “always flush” behavior in release builds.

For applications with constant disk writes, consider reserving plenty of spare provisioning room in an untouched partition. This ensures the flash controller has an exclusive store of blocks for wear leveling. Another alternative, albeit more costly, is to place highly active files on a separate disk backed by a NOR flash chip, which can support a much higher number of program-erase cycles.

3.2. SD cards

While it might work for DIY applications using a Raspberry Pi, we don't recommend placing your boot image on an SD card due to potential complications:

- Cards may not be inserted fully.
- Vibrations may loosen the card and cause intermittent connections.
- Users can use low-speed or insufficiently sized cards.
- Hackers can easily remove the card for reverse-engineering.

To build the most robust device, we recommend using on-board flash or dedicated flash chips.

3.3. Manufacturing

When your device is being built, your manufacturer must have a

starting image to flash into your products. They will also do basic provisioning, such as generating unique keys for each device. Our recommendation is to develop an automated process for regular updates of images to your hardware manufacturer. Try to work this out ahead of time, not after the production line is already rolling. Additionally, assume that every device will need an OTA update immediately “out of the box”. This lets you load the latest software version before it starts being used and reduces the number of software variants in the field.

Device-Unique Keys

Using one key for all devices simplifies the system build since the key will be part of the device firmware image. However, this approach is also the most prone to serious repercussions if the key gets compromised. Placing a private key in the image poses risks as hackers can reverse engineer the image to extract it. Additionally, obtaining factory-flashed chips from a silicon vendor becomes challenging because blacklisting the key invalidates all devices, not just one.

To use device-unique keys, allocate persistent storage on the device for the manufacturer to load with a unique private key. They should concurrently send the public part of the key to your cloud-based certificate management service. Importantly, while the device’s private key should be read-only under most circumstances, it must be alterable for key replacement – so don’t place it in write-once memory.

4. Delivery

4.1. Network

While it’s common to assume devices have constant connectivity, this is not always the case. Even when your devices do have connectivity, there are a couple considerations that can impact your update strategy:

- **Sporadic connection.** Devices that only connect while docked or have unreliable coverage.
- **Costly connection.** Devices using expensive cellular, roaming, or satellite technology.

- **Limited bandwidth.** Devices with only (or often) low-bandwidth connections.

In these cases, you can't just assume that firmware images are downloadable at any moment. Managing them requires minimizing update payloads and implementing a very robust retry strategy that should be interruptible and able to pick up exactly where it left off.

There are two other possible strategies to handle these low or sporadic connection situations, but both have drawbacks:

- **Update cadence.** Opting for a less frequent update schedule can be one solution. For more details, see section 5.2 on Security Advisories.
- **User initiated updates.** Another approach is to let users initiate or control updates. However, users may consistently skip pending updates, thereby missing out on crucial security updates or bug fixes. To avoid this, flag updates as either optional or mandatory and only allow users to skip optional updates. Another alternative is to allow users to skip optional updates but be restricted after a certain number of skips or days before updates become mandatory.

4.2. Hosting

Should you host your own OTA server? Using a SaaS solution will probably provide you and your customers with a better experience than a self-hosted solution. Generally, we think it's better to rely on someone who does this for a living. However, industry requirements or privacy regulations may make hosting your own on-prem OTA server necessary. If that's the case, make sure it runs on a dedicated system and assign a maintenance team.

4.3. Certificates

SSL certificates allow you to ensure that your devices are being updated with genuine content. That's why we recommend using them as part of the protocol connecting to the update server or within the update payloads. They provide a mechanism that authenticates your company as the valid source of the update binaries.

Certificates can be created with different cryptographic algorithms. Since continual advancements in computing power shorten the investment necessary by hackers to decrypt keys, it is prudent to use algorithms with the longest bit length keys (or those of equivalent strength). This maximizes the dependable lifespan of the certificate before needing to transition to an even more secure cryptographical algorithm and update all devices and servers.

Certificates still have a specified validity date though, and outside that time range they won't work. Plan to update all user devices as well as your update server with plenty of advance notice.

What if a certificate you're relying on expires anyway? The implications vary depending on whether the expired certificate is on the server or device side. If the server's certificate expires, devices connecting to the server will reject the expired authorization and fail. However, the resolution is simple – just update the certificate on the server. Eventually when the devices retry the connection, updates will proceed normally.

If the device certificates expire, you may have to have the user manually intervene. This situation is similar to circumstances where certificates become invalidated for other reasons – see the sidebar, *When Certificates Go Bad*. Such scenarios, while mere inconveniences for desktop systems, can be significantly challenging for embedded devices. If your update mechanism

relies on properly working certificates, then certificate problems will disrupt deployed devices from getting any further updates. As a precaution, have an emergency update method that doesn't require a valid certificate on the device, like a USB stick update.

When Certificates Go Bad

It's common knowledge that certificates expire. But they can also become blacklisted. When a certificate authority faces a security breach, it might become compromised, which in turn will invalidate all of the certificates issued by that authority. Additionally, certain attacks can result in fraudulent certificates being issued, allowing attackers to gain access to devices by faking authentication credentials. Granted, these are infrequent situations, but it pays to evaluate your OTA scheme against these events to be prepared should they arise.

4.4. Certificates and Time

SSL certificates require the device has reliable timekeeping in order to check certificate validity. Your product needs to keep time – both with an on-board real-time clock and by regularly consulting a time server. Irregular time may present problems with authenticating your certificates between device and OTA server.

If your device permits adjusting the time manually, setting the time outside a certificate's valid date range can result in unexpected errors. This can impact secure protocols, and any other process that relies on a valid SSL certificate. If you grant users the capability to adjust the device's time, it's crucial to understand these potential implications.

4.5. Payload Encryption

Sending updates (including compressed or delta files) without encryption can expose device software to reverse-engineering risks or tampering with the contents. We recommend encrypting the update payload to prevent any misuse, as well as any USB or offline update packages. Similar to the rationale provided in

section 4.3 on certificates, use the strongest keys that you can manage on the device.

4.6. Server Protocol

If you use a commercial or open-source solution, the protocol between the devices and update server will be already decided for you. If you're creating your own update server however, then you need to decide what protocol is necessary. If you've already encrypted and authenticated the update payload, then it might be acceptable to use plain HTTP.

Another option is to set up a dedicated server and develop your own protocol. This gives you complete control over the entire system, eliminating the need to rely on third-party web server code or external signing authorities. An advantage of this approach is that external entities can't force you to update your server or certificates when you don't want to. However, this can lead to a false sense of security. While avoiding software or protocol updates reduces the danger of externally forced updates, it doesn't necessarily enhance security. That said, creating your solution from scratch is a lot of work. A home-grown solution is probably overkill, especially if your team lacks cryptographic experience.

4.7. USB

Do you need to allow updates from a USB stick? For devices without network access, it might be the only choice. A USB update process (a "standalone update") can use most of the same mechanics as an OTA update, making it a reasonably low-effort addition to an existing OTA process. And many embedded devices will have a USB port either as part of an off-the-shelf board or already used for other purposes.

USB updates are particularly useful in certain situations:

- **Unavailable Internet.** In cases where connectivity isn't reliable or affordable (the problem cases in the above Network section), USB updates are practical.
- **Emergencies.** When updates fail due to a disrupted server connection (as discussed regarding HTTPS in the Encryption section), USB updates can provide a work around.
- **Field support.** Field support teams can use USB software updates to help customers recover from problems when OTA updates are not working.
- **Testing.** Testers benefit from USB updates as it allows them to easily switch between builds for bug reproduction and verification purposes.
- **Manufacturing.** If it's impractical to set up a local update server at a manufacturing plant, USB updates can be a viable alternative.

However, USB updates come with their challenges. The process has several technical steps that might make it overwhelming for average users. As a result, regular customers might only resort to this method in emergencies, often with help from support staff. It is much more useful for trained personnel – developers, testers, and support staff.

4.8. Right to Repair

The right to repair is increasingly a significant consideration. This allows device owners to update their devices outside of your controlled process. Though not universally mandated, certain jurisdictions may require it for specific classes of device, and the consumer-driven push for more right-to-repair legislation is growing. Even if it's not currently affecting your industry, incorporating these sorts of features can reflect a commitment to customer service and corporate responsibility.

So, what does accommodating the right to repair look like for an update solution?

- **Providing a bypass to the process.** Any method to bypass the server authentication or signed-image validation process should be a manual one. If your device can accept unsigned updates over-the-air, this capability will be certain to be abused by hackers. A USB key update that requires physical device access is a safer method for user-initiated side-loads.
- **Voiding the device warranty.** Personally adjusting the software can compromise device functionality, making support unfeasible or impractical. Thus, user-initiated software installations should void the warranty.
- **Warning the user.** Given the warranty implications and potential loss of support, users should be well-informed before making personal modifications. Issue a clear warning (and/or disclaimer) during the update process, ensuring users understand the ramifications before proceeding.
- **Tracking user software.** User installed software should be tracked so that your support team knows if a warranty still applies and whether they can update the firmware remotely. To signify an unsigned update is installed, you could blow a hardware fuse, making the transition to a user-maintained state irreversible.

5. Timing

5.1. Device Reboot

When is it safe to put a newly downloaded software image to work – in other words, reboot the device? This is highly dependent on what you expect from your device. Consider the following:

- How long does the reboot process take?

- Does the device have expected or predictable downtimes?
- Does the device need to meet a specific uptime guarantee?
- What happens when the device becomes unavailable?

To understand how these answers impact your reboot strategy, let's walk through a couple of examples.

Updating a car's software is straightforward as long as it's not driving. This means a car updater can postpone updates until the car is turned off when it can then reflash to be ready for the next trip. Given that cars, even autonomous ones, are not in continuous operation, the delay is minimal. This approach is the same for consumer devices that are turned on and off regularly, like televisions, game consoles, kitchen appliances, and audio systems.

However, many devices need to run constantly and/or be ready at an instant's notice. Examples include factory PLCs, industrial monitoring systems, network routers, HVAC systems, hospital patient monitors, security alarms, and telecommunication relays. These devices typically operate in controlled environments and so have defined downtime periods like factory holidays or planned network outages for equipment installation. For such devices, asking user permission before applying an update may be necessary to ensure adequate preparation for any brief offline periods.

5.2. Security Advisories

Track common vulnerabilities and exposures (CVEs), security patches, and security advisories related to your product's software (and hardware). When you get these advisories, evaluate them, integrate new security patches into your build, and reissue updates. This includes the OS, BSP, drivers, libraries, open-source code, and the tool chain, even from software suppliers. To reduce

update churn, consider batching up a series of patches before pushing new updates out to customers. Whether a batching strategy makes sense depends on the severity of the issues found; critical vulnerabilities may demand immediate updates.

If you're not doing this best practice, at a minimum plan to revisit your CVE exposure every six months. Review all components within your software bill of materials (SBOM) and see if any critical vulnerabilities need to be plugged before they become major problems. It's worth noting the impending [Cyber Resilience Act](#) in the European Union may mandate manufacturers adopt certain requirements for addressing vulnerabilities and OTA, forcing security updates to be rolled out by default while allowing the user to opt-out.

5.3. Update Batches


Releasing updates to all devices at the same time might not be possible due to server bandwidth. Splitting the update into smaller groups can help balance the load and resolve server bandwidth issues. If you have test groups, like in-house testers or beta users, they can receive updates and provide feedback before a broader release. Geographical segmentation is also beneficial, allowing updates to be timed to user time zones, which can be handy for support reasons.

6. Process

6.1. Integration

A core feature of an IoT device is its updatability. However, an update mechanism often isn't added to software until late in the development process. This leaves all of the update-specific testing crammed towards the end of the project.

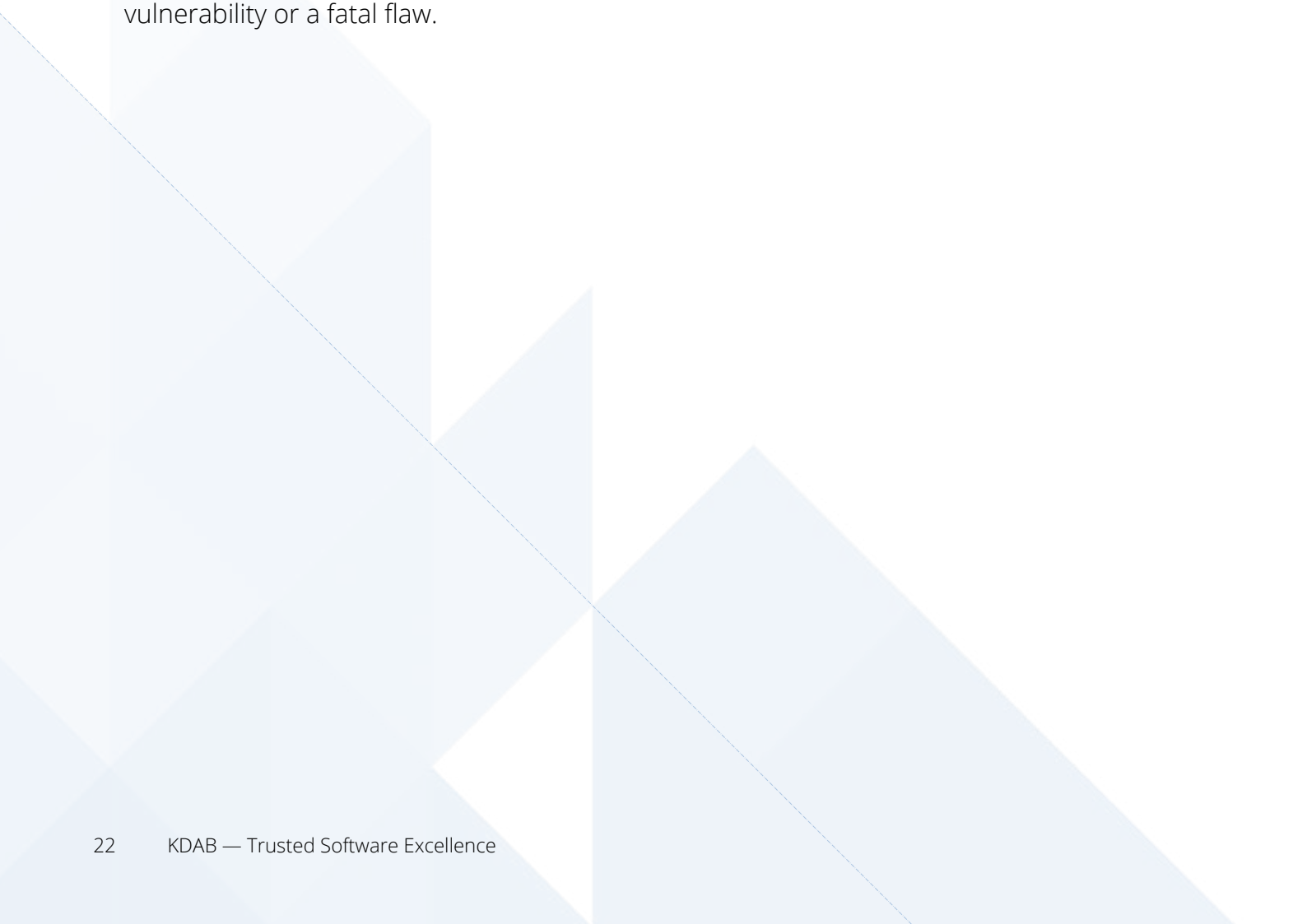
Our advice is to integrate the update system functionality as



early as possible. Early integration lets you get extra milage on update features, ensuring good test coverage. Once the update mechanism is in place, it also facilitates timely delivery of bug fixes and new features to both in-house testers and beta users before the official release.

6.2. Builds

The build process for a new update should do absolutely everything possible automatically. Of course, the build should update version numbers and create the final software image. But it should also sign the image, push it to the server, and kick off the deployment process. Once the software is ready for release, it should be a one-button operation to send it out to your devices. This automation ensures that your team can successfully roll out new software even under high stress situations like a zero-day vulnerability or a fatal flaw.



What is KDAB's Software Development Best Practice series?

This series of whitepapers captures some of the hard-won experience that our senior engineering staff has developed over many years and projects. Offered up as a grab bag of techniques and approaches, we believe that these tips from KDAB engineers and other industry experts have helped us improve the overall development experience and quality of the resulting software. We hope they offer the same benefits to you.

View all the parts of this whitepaper series online at: www.kdab.com/publications/bestpractices/

About the KDAB Group

The KDAB Group is the world's leading software consultancy for architecture, development and design of Qt, C++ and OpenGL applications across desktop, embedded and mobile platforms. KDAB is the biggest independent contributor to Qt and is the world's first ISO 9001 certified Qt consulting and development company. Our experts build run-times, mix native and web technologies, solve hardware stack performance issues and porting problems for hundreds of customers, many among the Fortune 500. KDAB's tools and extensive experience in creating, debugging, profiling and porting complex applications help developers worldwide to deliver successful projects. KDAB's trainers, all full-time developers, provide market leading, hands-on, training for Qt, OpenGL and modern C++ in multiple languages.

www.kdab.com

© 2023 the KDAB Group. KDAB is a registered trademark of the KDAB Group. All other trademarks belong to their respective owners.

The KDAB logo consists of a blue speech bubble shape pointing downwards and to the right. Inside the bubble, the word "KDAB" is written in white, bold, sans-serif capital letters. To the left of the letters is a white icon of a lightning bolt or a stylized 'K'.