MENDER

Requirements Checklist for Over-The-Air (OTA) Software Updates for Connected Devices/IoT

Robust

REQUIREMENT	DESCRIPTION	RISKS WHEN NOT IMPLEMENTED
 **Update mechanism is immune to power and network loss mid-update in order to avoid any bricked devices.**	Atomic installs (full image over package- based) where an update is always fully installed or not at all avoids the possibility of a device becoming unbootable due to interruptions of an update. No software component ever sees a partially installed update.	Devices may be bricked when an update is interrupted due to poor network connectivity or power loss and may become unusable. Non-atomic updates also run the risk of inconsistency across a fleet of devices and becomes unmanageable when production does not match test environments.
 **Rollback to previous working version is built-in and automatic with sanity checks post-installation.**	Dual A/B partition is one of the most reliable and simple approaches for built-in rollback where the update is written to the inactive root file-system partition and the bootloader is configured to boot from it and then the embedded system reboots. Once the updater comes up it will try to report the success of the deployment to the management server. If this fails it will automatically rollback.	If an update fails for any reason with no ability to rollback to the last working version, sending a technician to the field where the device resides will be costly.
 **Integrity check to avoid corruption.**	Make sure the update artifact is not corrupted, end-to-end from build system to device, due to any transfer or hardware issues.	Intermittent application malfunction, application not starting or device bricking due to random modifications of the update
 **Compatibility check.**	Verifies that the software update can run on the target devices. For example, the CPU architecture of the devices are supported by the software update.	Application fails to start or the device gets bricked immediately because the software is built for the wrong device type. This is particularly an issue when multiple revisions of devices and software are introduced, and software updates only support a subset of the devices.
 **Ensure another update can be deployed.**	After an update has been deployed, make sure that another update can be deployed by making sure that the deployment server can be reached after the update. Otherwise a rollback should be performed.	If an update makes a device malfunction in certain ways such as losing network connectivity, the device can never be fixed and is effectively bricked.
 **Custom sanity checks after the update.**	Even though a device boots, it does not mean that its application works properly. For example, the UI framework may not be brought up. Custom sanity checks are automatically run after the update and the updater automatically rolls back the update if they do not all pass.	The device may be unusable to the end user because important applications are not brought up after the update or key services are malfunctioning.

Robust

REQUIREMENT	DESCRIPTION	RISKS WHEN NOT IMPLEMENTED
Well maintained code and high code quality.	The updater is a critical component to an embedded system and needs to be well tested and widely used across a wide variety of use cases so the code maintains a high quality level.	Bugs in the updater itself can not only brick devices but also expose serious security vulnerabilities to the devices.

Security

REQUIREMENT	DESCRIPTION	RISKS WHEN NOT IMPLEMENTED
Secure communica- tions between the management server and the client run- ning on the device.	Use of secure and bi-directionally authenticated communication between the client/server (e.g. TLS).	The update can be modified while in transit from the update server to the device, enabling an attacker to inject malicious code, steal data and take over the device. This attack is particularly susceptible over wireless networks. The attacker could also imitate the update server, forcing a malicious update to be installed at their choosing.
Authenticity of update images with code signing.	End-to-end signature management to authenticate it is an authorized and trusted update.	Lack of digital signatures/code signing allows attackers to reprogram sensitive components of the embedded system by modifying a valid update, enabling attackers to inject malicious code or take over the device.

Integration into existing environment

REQUIREMENT	DESCRIPTION	RISKS WHEN NOT IMPLEMENTED
Existing development build system integration.	The update process should integrate with existing OSes, development tools and popular build systems like the Yocto Project.	Pushback from the development team if the updater dictates the tools they need to use, for example a “rip and replace” solution that dictates the embedded OS, language or how updates are packaged.
Standalone deployments.	While OTA is the desired mechanism, many devices will initially require stand- alone deployments with an update via a USB or SD Card.	If devices without wireless network connectivity cannot be updated, they have an increased security risk and will almost certainly be vulnerable to attacks of publicly known vulnerabilities over time.
Support for multiple storage types.	Embedded systems can use a variety of storage, including raw NAND flash, eMMC and SPI-NOR. Besides supporting all these, the updater must not make unnecessary writes to the storage because this will quickly wear out the flash.	If the updater does not support all embedded storage types used, some devices may never be able to update and thus leave a security issue as software vulnerabilities become public over time. If the updater does not take into account the limited wear requirement of flash-based devices, the devices will get bricked over time.

Fleet management

REQUIREMENT	DESCRIPTION	RISKS WHEN NOT IMPLEMENTED
Deployment management server.	Ability to automate deployment across a fleet of devices.	If devices need to be updated one-by-one, deploying updates will be extremely costly or not carried out at all.
Rollout management.	The ability to perform phased rollouts and deployments specific to certain groups or locations of the device population.	Updating fleets without rollout management and device groupings will require more man hours to update entire device fleets. When manual tasks are increased, mistakes increase accordingly. If you can only update to all devices, the risk is high that one mistake can brick the entire fleet.
Device Inventory.	Ability to list all devices, through an UI and REST API, together with key inventory information such as network addresses, product revision and model and when it last connected to the server infrastructure.	Lack of device inventory can easily lead to security breaches as "forgotten" devices are no longer maintained.
Installed software versions.	Ability to see the current software versions installed on each device and query this information.	In order to plan for software updates, you need to know what is out there already. This will make it possible to know if the installed software is compatible with the desired update. It is also critical to know which software is installed on all the devices in order to discover which devices are vulnerable to known vulnerabilities and CVEs so that they can be updated.
Deployment logs.	Diagnostics logs from devices that are captured as a deployment is carried out.	If a deployment fails across one or more devices and there are no logs, issues are very hard to diagnose and fix.
Deployment status reports.	An overall status of the deployment, such as how many devices succeeded and failed the deployment, as well as which devices failed.	If it is not possible to know if a deployment succeeds or fails and on which devices, deploying updates is very risky as bricking devices can go unnoticed for a very long time. The only way to see failures in this case is when end users report them which leads to a very bad customer experience and an expensive support burden.

MENDER

CONTACT

+1 650 670-8600
contact@mender.io
www.mender.io