MENDER

A MENDER WHITE PAPER

Software updates for IoT and the Hidden Costs of Homegrown Updaters

How hard can software updates for IoT be?

The embedded systems industry is struggling to properly bring devices online due to the security implications of connecting devices at mass scale. The Internet of Things is not new -- it was previously referred to as cyber-physical systems or machine-to-machine (M2M): bringing devices online is not a novel development. But due to the scale of connecting devices and the numerous hacks that has already affected health-sensitive target devices as in the case of medical devices and connected cars, it is rightfully receiving renewed scrutiny.

One specific area receiving attention is the ability to update and patch vulnerable IoT devices. Unfortunately, the software update mechanism is typically a rudimentary homemade endeavor for many embedded development teams. A common thread from these homegrown updaters was it largely lacked the many capabilities to ensure security and robustness.

There were two primary causes for this:

1. Building a software update mechanism for network- connected embedded systems was outside the core competency of most embedded developers
2. The update mechanism was often an afterthought and with time-to-market pressure and product release deadlines, the OTA mechanism was hastily assembled and did not consider many of the requirements to ensure a robust and secure update process

The industry clearly needs a better approach with something so critical to the security of the growing connectivity of devices. Rather than having individual embedded teams each develop their own software update mechanism, there should be an end-to-end open source project with a permissive license the entire embedded community can get behind. Developers of various connected device products and solutions can draw upon this open source project and it will support the growing needs of the market. For example, it should support popular boards such as the Raspberry Pi, which has grown from a hobbyist board initially dismissed as a prototyping board to one disrupting the traditional embedded world with its low cost and easy availability at scale. Another benefit of a community-backed project is it will have many more eyes on the code which will increase the security and the quality of the code. And while a reliable OTA mechanism is a requirement for connected devices, spending time building an updater will distract already limited time and resources from focusing on the actual features of your product that will provide direct value to customers and differentiation from competitors. An update mechanism requires attention to many security nuances and is crucial to a product's viability given the potential repercussions of a compromised device, whether it has been bricked in the field or malware has compromised it.

As detailed in Figure 1, the costs are largely in the lower layers where there are many open source options available that are community-driven and freely available. Building your own OTA updater should no longer be an option. We will cover the specific requirements under four broad categories often overlooked in a homegrown OTA solution.

> Many client-only solutions exist but an update mechanism should have the ability to manage the entire fleet of devices.

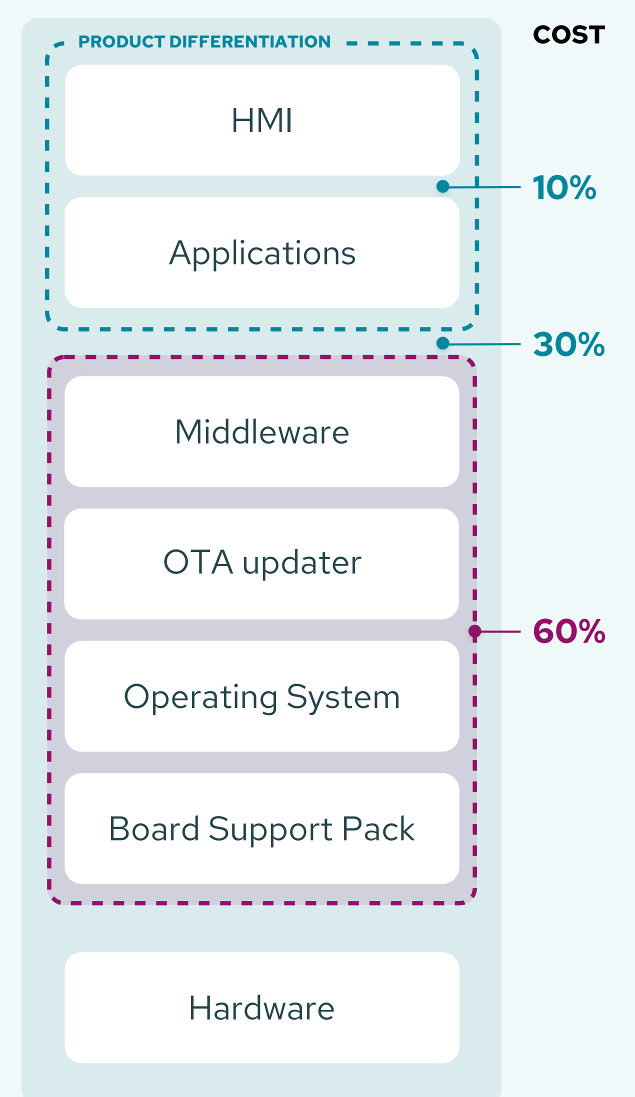

Figure 1: Embedded Stack

Fleet management

Manual one-by-one updating of devices should be a remnant of the past even with smaller fleets of just 10 devices. Many client-only solutions exist but an update mechanism should have the ability to manage the entire fleet of devices. Fleet management requirements can be extensive and have many different requirements depending on the specific scenario and environment. To be concise, we have limited the list of requirements for fleet management to six key capabilities.

The need for a management server

One of the more complicated aspects of an OTA solution is the fleet management capabilities of the management server. During our initial investigation interviewing embedded device owners who built their own updater, many lacked the capability for fleet-wide rollouts of updates properly. There was no way to do rollouts with certain groups or locations of the device population or do the rollouts in phases to mitigate risks of widely deployed bad updates. Many homegrown updaters were also restricted to updating devices one-by-one, which easily becomes unmanageable as the device population grows. The increase of manual tasks will increase the probability of mistakes, which becomes highly risky given the sensitive nature of the update process. One mistake can lead to unusable devices in the field, an expensive and potentially unrecoverable situation.

The increase of manual tasks will increase the probability of mistakes, which becomes highly risky given the sensitive nature of the update process.

Basic inventory list of every device

Another key capability in the management server is the ability to list all the devices with key inventory information such as network addresses, product version, model, and when it last connected to the management server. The inability to see this information can easily lead to security breaches of forgotten devices as there is no visibility to ensure all devices are actively maintained and patched if a vulnerability is discovered.

Overview of running software versions

The ability to see the current software version installed on each device is also important in planning which updates you will be rolling out. This is required to understand which devices are vulnerable to known vulnerabilities and CVEs and won't be left outdated and insecure. This will also make it possible to know if the installed software is compatible with the desired update.

Software compatibility

It is also required to understand what device types are compatible with what available updates. A recent example of this having catastrophic results is a smart lock company widely used by Airbnb[1]. There were many reasons - which will be covered - that lead to the smart locks being bricked by a software update in August 2017. A key reason was that a software update was deployed to the incorrect devices -- a previous version of the smart lock. And it rendered those devices unable to receive another OTA update to fix the issue. Unfortunately, automatic rollback was not a capability built into the devices which lead to a PR disaster that required a public apology from their CEO.

[1] Lomas, Natasha (2017, August 14) *Update bricks smart locks preferred by Airbnb.* Retrieved from https://techcrunch.com/2017/08/14 wifi-disabled/

Log management

Other capabilities needed from the management server are deployment logs to properly diagnose any issues arising from a failed deployment. Deployment status reports are required for visibility into the overall status of an update, such as how many devices succeeded and failed, and specifying which devices failed. Many servers lack this capability and thus bricked devices go unnoticed.

Building a management server is costly

In conversations with several embedded consultants who have built a management server with all of these capabilities, the lines of code range from 40,000 to 80,000 just for the backend. This will typically take up the time of two developers for over a year -- an expensive endeavor. And while the management server is a critical part of an OTA mechanism, this does not include the client-side component that exists on the embedded system, which can range from 10,000 to 20,000 lines of code.

Robust

One of the most important characteristics of an OTA solution is to ensure the update process is robust. This has many elements to ensure the resiliency of the embedded systems. One of the worst possible scenarios is to have devices remotely deployed and then due to an interruption during an update become unusable and bricked. The resiliency and reliability of the update process should be a central concern given the potential consequences.

The resiliency and reliability of the update process should be a central concern given the potential consequences.

Power or network loss

A common scenario causing devices to brick is when a loss of power or network occurs during the update. It is common for embedded systems to experience this when they are remotely deployed, thus it is necessary to have atomic installations where an update is fully installed or not at all. Partial installations create inconsistency in remotely deployed devices and can quickly veer into chaos when the production devices do not match the test environment. It is generally a best practice in embedded systems to avoid non-atomic updates due to the lack of integrity it can produce.

The risk of package management

While package-based updates are common in traditional Linux software (e.g. apt or yum), it is typically avoided in embedded Linux due to many issues. It is difficult to guarantee a consistent set of packages installed across a fleet of devices. Additionally, the number of combinations of installed packages can make for a QA nightmare. Package systems are fragile and if a post-trigger/script fails the package install itself fails. And it will likely not be a clean fail, where the partially installed package can block the install of a new package designed to fix the fail. The attention required to tend to individual packages easily becomes unmanageable and the need to test all combinations of installed packages for each release can become a very large task -- affecting how timely you can even deploy updates.

The need for atomic updates with built-in rollback

It is very common for the output of an embedded Linux CI build to be a complete root filesystem, thus having a dual rootfs approach is one of the simplest and most reliable ways to ensure the embedded device is robust with automatic rollback to the other rootfs partition. A common approach is to have the update written to the inactive rootfs partition while the bootloader is configured to boot from it when the embedded system reboots. Once the updater comes up it will report the success of the deployment to the management server. If this fails, it will automatically rollback to ensure the device will remain updateable. This was the challenge with the smart locks previously mentioned - it was not able to communicate to the server after receiving the wrong update and did not automatically rollback, thus it became bricked. The dual rootfs approach also simplifies the build system by building all the packets in a reliable and predictable way. The deployments will be reproducible as all the devices will get the same version of all the subcomponents.

Integrity checks

Other capabilities to ensure a robust update mechanism is the ability to have integrity checks in order to avoid corruption of the update artifact. There should not be any corruptions end-to-end, from build system to device due to any transfer or hardware issues. Random modifications of the update could lead to applications malfunctioning, applications not starting properly, or even the device bricking.

Compatibility checks

Compatibility checks are another requirement and will verify that software updates can run on the target device. For example, the CPU architecture of the devices needs to match and be supported by the software update - and a compatibility check will ensure this. Risks of not having this include embedded applications not being able to start or the device even being bricked due to the software being built for the wrong device type.

Update success validations

The ability to deploy another update is required by making sure the deployment server can be reached after every update - otherwise a rollback should be performed. If an update effectively makes a device lose network connectivity and unable to communicate with the deployment server, it is effectively bricked and will require a field technician to repair the device, driving up costs. Custom sanity checks should also be in place, as the embedded applications will need to pass a series of checks to ensure all the applications are properly functioning.

Security

The increase of connected devices being compromised had a direct result in the 91% growth of DDoS attacks in 2017[2], which was attributed to poor IoT security. The trend is profoundly disturbing and requires security to no longer take a backseat to time-to-market pressures for product development teams.

Code signing

A key security feature is code signing (cryptographic validation) of an update to ensure tight control over who can reprogram sensitive components. Researchers at a Chinese firm, Tencent, revealed they could remotely activate a moving vehicle's brakes by exploiting security vulnerabilities such as the lack of code signing of the Tesla Model S in 2016, which has since been rectified by Tesla: "Cryptographic validation of firmware updates is something we've wanted to do for a while to make things even more robust, "said Tesla's Chief Technical Officer JB Straubel[3].

Both the Chevy Impala that researchers hacked via OnStar in 2010[4] and the 2014 Jeep Cherokee that hackers hijacked on the highway in 2015 lacked code signing[5], demonstrating that even large automotive manufacturers lack basic security to ensure the authenticity of their firmware updates.

Encrypted communication only

Another key security requirement is secure communications between the management server and the client running on the device. There should be bi-directionally authenticated communication between the client/server to avoid the risk of the update being modified while in transit and allowing an attacker to inject malicious code and take over the device. This type of attack is particularly susceptible over wireless networks as they can imitate the update server and force a malicious update to be installed.

Integration into existing environments

While there are times embedded teams are deploying a greenfield project without the burden of an existing fleet of devices remotely deployed, that is usually not the case. In many situations where you have existing devices in the field and are deploying a new product version, you will need

[2] Rayome, Alison DeNisco (2017, November 20) *DDoS attacks increased 91% in 2017 thanks to IoT*. Retrieved from https://www.techrepublic.com/article/ddos-attacks-increased-91-in-2017-thanks-to-iot/

[3] Greenberg, Andy (2016, September 27) *Tesla responds to Chinese hack with a major security upgrade*. Retrieved from https://www.wired.com/2016/09/tesla-responds-chinese-hack-major- security-upgrade/

[4] Greenberg, Andy (2015, September 10) *GM took 5 years to fix a full-takeover hack in millions of Onstar cars*. Retrieved from https://www.wired.com/2015/09/gm-took-5-years-fix-full-takeover-hack-millions-onstar-cars/

[5] Greenberg, Andy (2015, July 21) *Hackers remotely kill a Jeep on the highway - with me in it*. Retrieved from https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/

to consider how it integrates into your existing environment and will not rip-and-replace your established infrastructure and require reworking of your existing processes.

Integration into existing development tools

Your existing development, build, and continuous integration system is key, and the software update mechanism should integrate into your existing operating and build systems. The risk is the pushback from the development team if an update mechanism would dictate the tools they need to use or a solution that requires you to adopt a specific embedded OS, language, or how updates are packaged.

Support for offline devices

Another consideration is that while a device fleet is ideally all connected, in many circumstances that is simply not possible and standalone deployments would also need to be supported. Many situations will not allow for certain portions of your fleet to be connected all the time and a standalone update with a USB or SD Card may be required. Devices that cannot be updated both ways will have an increased security risk and be vulnerable to attacks if they cannot be updated flexibly.

Support for embedded storage types

Lastly, embedded systems use a variety of storage and you want an update mechanism to be able to support a variety of them, including raw NAND flash, eMMC, and SPI-NOR. The updater also must not make unnecessary writes to the storage as this will quickly wear out the flash.

Conclusions

The list of requirements needed to have a secure and robust OTA solution is substantial. A complete end-to-end software updater with all the capabilities described in this White Paper, including both the deployment server and the client can range from 50,000-100,000 lines of code. While the cost estimates can be broad, it has been cited that embedded systems software typically range from $15 to $40 per line of code[6]. On the low end, you can expect the cost of a complete update mechanism built from scratch to be a minimum of $750,000 with no sacrifices made to security and robustness.

[6] Koopman, Phil (2010, October 2nd) *Better Embedded System SW*. Retrieved from https://betterembsw.blogspot.com/2010/10/embedded-software-costs-15-40-per-line.html

> **A complete update mechanism built from scratch can be a minimum of $750,000 with no sacrifices to security and robustness.**

Many have ventured into creating a homegrown solution and have discovered the hard way that their updater is lacking many baseline capabilities that leaves their devices at risk of being hacked or bricked. Otherwise, building a feature complete update mechanism with no compromises around security and robustness takes a considerable amount of resources. That translates to high costs to create something that has been done before which does not directly lead to product differentiation.

Yet, having a dependable OTA solution is clearly a necessity with very real ramifications. Car manufacturers have millions in losses due to the inability to update their vehicles remotely and needing to manually bring in vehicle recalls into the dealership that could have been fixed with an OTA software update. Forward-thinking manufacturers such as Tesla have shaken the automotive industry in no small part due to the ability to deploy OTA updates and remove inefficient manual recalls that could be remotely fixed by software. Tesla is also able to generate after-market revenue with almost $20,000 in software-upgradeable options[7].

[7] Lambert, Fred (2016, October 26) *Tesla now offers almost $20,000 in software-upgradeable options when buying a new vehicle.* Retrieved from https://electrek.co/2016/10/26/tesla-now-offers-almost-20000-in-software-upgradable-options-when-buying-a-vehicle/

The smart locks used by Airbnb demonstrate the kind of fallout that can occur from the lack of a secure and robust OTA mechanism, with the CEO publicly apologizing for the bricked door locks with potential sales losses following their PR disaster[8].

DDoS attacks are on the rise, with one of the largest ever attacks causing major Internet outages in October 2016. The Dyn cyberattack took down large swaths of the Internet including sites such as GitHub, Netflix, Amazon, and Verizon Communications[9]. The root cause for this botnet attack were insecure IoT devices. The lessons of 2016 have yet to be learned, as there has been a 91% increase in DDoS attacks in 2017 due to insecure IoT devices[2]. The future is bleak unless the industry takes concrete action in planning and executing a strong security posture. And one of the first major steps is to ensure every connected device is upgradeable and patchable through a secure and robust OTA software updater.

[8] Jeffrey, Cal (2017, August 15) *LockState accidentally bricks hundreds of locks through a failed firmware update*. Retrieved from https://www.techspot.com/news/70588-lockstate-accidentally-bricks-hundreds-locks-through-failed-firmware.html

[9] 2016 Dyn cyberattack. (n.d.) In *Wikipedia*. Retrieved November 29, 2017, from https://en.wikipedia.org/wiki/2016_Dyn_cyberattack

About Mender.io

Mender.io is a leading provider of a secure and robust end-to-end over-the-air (OTA) software update manager for IoT devices. Mender makes it easy to deploy updates to a large number of devices by providing efficient and risk tolerant OTA deployments. Mender enables its customers to stay competitive in a fast-moving market by helping them deliver high-value services on an increasing number of connected devices with growing software complexity. With an active open source community supporting a large number of different hardware and operating systems and growing every day, Mender has quickly become the trusted choice by some of the world's most respected brands.

Mender documentation

https://docs.mender.io/get-started

Mender Hub community:

https://hub.mender.io

Mender on Github:

https://github.com/mendersoftware/

CONTACT

+1 650 670-8600

contact@mender.io

mender.io