Typical product development process

- Prototyping
- Production design
- Mass production
- Release deadline panic
Updater is too often an afterthought

Prototyping → Production design

Mass production → Release deadline panic

Oh no! There will be bugs in production!
Updater is too often an afterthought

This is where you *should* design for OTA updates.
Homegrown updater #1: The bricker

- Developed in a hurry, not well tested
- Typically user-space shell scripts, e.g. using ipkg/rpm or application “self update”
- Lacks update compatibility checks
- No sanity checking / rollback
- Does not meet robustness & atomicity requirements for embedded
- If the device loses power during update process it is likely bricked
Homegrown updater #2: The honeypot

- Developed without sufficient skills of security
- Gets updates over *plaintext protocols* or misuses crypto
- Lacks update signing / encryption
- Makes it trivial for (wireless) attacker to “pwn” the devices; install any malware on them
Homegrown updater #3: The needy

- Lacks *update server*; cannot automated updates across devices
- Manual operation to do 1-on-1 updates to devices, e.g. USB stick or remote 1-on-1 connection
- Updates become so frustrating and expensive that it is not used except during disasters
The embedded environment

- Remote
  - Expensive to reach physically

- Long expected lifetime
  - 5 - 10 years

- Unreliable power
  - Battery
  - Suddenly unplugged

- Unreliable network
  - Intermittent connectivity
  - Low bandwidth
  - Insecure

What can go wrong?
Key criteria for embedded updates

1. Robust and secure
2. Integrates with existing environments
3. Easy to get started
4. Bandwidth consumption
5. Downtime during update
6. Update server enabling mass updates
**Generic embedded updater workflow**

1. **Detect update** (secure channel)
2. **Compatibility check**
3. **Download** (secure channel)
4. **Integrity** (e.g. checksum)
5. **Authenticate** (e.g. signature)
6. **Decrypt**
7. **Extract**
8. **Pre-install actions**
9. **Install**
10. **Post-install actions**
11. **(re)Start**
12. **Sanity checks**
13. **Failure recovery** (e.g. roll back)

---

- **Must-have**
- **Environment-specific**

*E.g. reboot, restart service, start container

---

Choose a strategy
Mender provides integrated client and update server

- Client-server model
  - Mender provides both
  - Easy integration: No need to “glue” several projects
  - Server can integrate with 3rd party clients through its REST API

- Dual A/B rootfs partition layout
  - Atomic deployments
  - Deploy to inactive partition
  - Robust update process

- Supports updating
  - Kernel, device tree
  - Applications

Apache License v. 2.0
Mender demo!
2018 focus: simplify device integration (1)

- Automate U-Boot patching (for rootfs partition selection)
  - Almost done!

- Mender community integrations & free CI service
2018 focus: simplify device integration (2)

- No U-Boot/boot command patching
  - UEFI binary support

- Alternative: “POC mode” vs. production integration

- Binary post-process images to integrate Mender
2018 focus: simplify device integration (3)

- x86
- Debian, Ubuntu, Raspbian
- Buildroot
Feedback? What is missing for you?

- Is simplifying device integration worthwhile? How?
- Other product-related items?
- Areas for community & contributions?
Join OTA updates sessions

- Comparing and Contrasting Embedded Linux Build Systems and Distributions
  - Drew Moseley, Mender.io
  - Tuesday 10:50am, Pavilion East

- Update My Board!
  - Mirza Krak, Endian Technologies AB
  - Tuesday 10:50am, Grand Ballroom II

- Mender booth in expo hall
  - Attendee reception Tuesday, 5:10pm - 7pm

- The IoT Botnet Wars, Linux Devices, and the Absence of Basic Security Hardening
  - Eystein Stenberg, Mender.io
  - Wednesday 3:30pm, Broadway I/II
Installer strategy 1: run-time installation

- Robustness is hard
  - Atomicity: Hard or impossible
  - Consistency (dev=test): Hard

- Integrates well
  - May already have packages
  - Some userspace tools

- Low bandwidth use (<1mb)

- Short downtime (seconds)

- Updater deploys to running environment
  - Package managers (ipkg, rpm, deb...)
  - OSTree
  - Many homegrown (tar.gz)
Installer strategy 2: boot to maintenance mode

- Robustness is hard
  - Not atomic (can get partial update)
  - Consistent on success (image)

- Integrates fairly well
  - Bootloader features & intelligence

- High bandwidth use*
  - Whole image

- Long downtime
  - Whole image install
  - 2 reboots

*Can mitigate: compressed/delta

- Updater deploys “up the stack” while running in bootloader
  - Used in older Androids (before ‘N’)
  - “Rescue environment” common in embedded
Installer strategy 3: dual A/B rootfs layout

- Very robust
  - Fully atomic and consistent

- Integrates fairly well
  - OS, kernel, apps unchanged
  - Needs bootloader “flip” support
  - Partition layout, requires 2x rootfs storage

- High bandwidth use*
  - Whole image

- Fairly short downtime (minute)
  - 1 reboot

*Can mitigate: compressed/delta

- Updater deploys to inactive partition, then reboots into it
  - Used in newer Androids (‘N’ and later)
  - Common in “mid/high-end” embedded

<table>
<thead>
<tr>
<th>Updater A</th>
<th>User space A</th>
<th>Kernel, initramfs A</th>
<th>Bootloader</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>User space B</td>
<td>Kernel, initramfs B</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Updater B</th>
<th>User space B</th>
<th>Kernel, initramfs B</th>
<th>Bootloader</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Installer strategy 4: proxy

- **Updater deploys to remote system**
  - Used on smaller devices (sensors, ECUs, etc.), such as in Smart Home or Automotive
  - Requires intelligent gateway to manage

- **Slightly different scenario**
  - Smaller devices (no client)
  - Complements the others

- **Suited for closeby installations only, not internet**
  - Robustness (e.g. connection/power loss)
  - Security