Linux Automotive Security

"Safer and more secure"

Authors

Fulup Ar Foll fulup@iot.bzh
José Bollo jobol@iot.bzh

Abstract

Cars are expensive pieces of equipment, yet they represent a huge mass market. Adding Internet connectivity to previous elements generates perfect conditions for the growth of a viable business model on attacking "Connected Cars". It is already well understood that cars will be connected and connected cars will be attacked.

While it's still too early to predict with certainty how "Connected Cars" will be impacted by security flaws, making the assumption that a car should be at least as secure as a TV, a set-top-box or a smart phone should make sense to everyone.

This white paper focuses on how Linux best practices security mechanisms that could be used today and within the next couple of years to make connected cars safer and more secure.

Version 1.0

January 2016

Table of contents

1. Introduction

Analysts predict that by 2020 almost every new car will have some form of connectivity. While the "Connected Car" is already a reality for most high-end automakers, Gartner Research predicts that by 2020, over 250 million vehicles in circulation will be connected.

It's common knowledge that connected equipment can be attacked. Last summer the attack on Chrysler/Fiat Jeeps established some concrete numbers on the cost of ignoring security flaws. While the Fiat incident was quite visible with the recall of 1.4M vehicles, it wasn't the first. A few months earlier, BMW had 2.2M vehicles impacted by a security flaw that left locks open to hackers.

Because cyber-crimes against cars are obvious, attacks will continue to increase. There are multiple viable potential businesses related to car attacks. Stealing money from their owners is the most obvious: vehicles are expensive and if a hacker succeeds in "bricking" a car, most owners may expect to pay a significant amount of money to retrieve usage of their beloved car. Stealing private information may also be very profitable. Vehicles contain a lot of information about the owner's private life. This information is very valuable and many people are willing to pay for it. And finally, there is potential activism for terrorism. Sadly, there are many organizations around the globe that would love to block modern economies with impenetrable traffic jams.

Before summer of 2015, too many people were able to elude security questions with the argument that: "*It will not happen to us*"; "My subcontractor *endorses all security liabilities*"; "*Attacking cars is too much effort versus the reward*", etc. If the only argument is that attacking cars is time-consuming, difficult, and expensive, well, unfortunately, hackers are very smart and have both money and time.

The most common motivation for ignoring security is cost reduction and time to market. In order to have developers implementing secure systems, security needs to be baked into the operating systems and must be harder to remove than to work with. One risk when talking about security is the "boiling the ocean" syndrome. The fact that security has an impact on every single component of the system does not mean that the effort is going to be endless.

This paper talks about existing mechanisms that can be used by developers today to make connected cars safer and more secure. We should assume that cars need to be at least as secure as a TV, a set-top-box, or a smart phone. A system claiming to be "Automotive Grade" should be able to easily answer questions such as: "How do I protect my system boot?", "How do I guarantee my software integrity?", "How do I update the system?", and "How do I prevent untrustworthy applications from violating their permissions?"

2. Make Sure You Run the Right Code

If a hacker succeeds in replacing any of the system or application code, then the complete security of the system is compromised. With all cars becoming connected, we need to make sure their systems only boot approved initial software. Then, when upgrading software, we need to make sure alien pieces of code are not introduced.

Furthermore, the right software to upload may depend on usage context. What is acceptable during the development phase is very different from what is acceptable in production when the car is on the road. While the system should probably enable some special feature during factory testing and later on during development, those options should be irreversibly removed when running in production. For example: during production, it might be required to run tests that probe hardware interfaces pin by pin to check for short circuits. During development phase, we may want to accept self-signed software, but when in production we can only accept proved software that's been officially signed.

2.1. Before Booting

Those who have physical access to a system are well placed for finding an entry door. While it is possible to restrict access to an Internet server by locking it in a secure data center, trying to use the same model and lock connected cars in a secure garage is not going to work.

While it might not be easy to get physical access to an IVI/ADAS board, underestimating the skills of "black hats" is a very risky bet. Automotive systems should be protected even against people who have full physical access to the device.

- Protecting the hardware: In production interfaces like JTAG or others, low level mechanisms to introduce code in RAM need to be disabled through an irreversible fuse feature. Allowing JTAG is allowing anyone to do anything from inside the processor. Securing the system is simply not possible if development/debug capabilities remain open.
- Making sure the bootloader is the expected one: In production, a safe option is to permanently fuse the bootloader code in a ROM directly on the SoC and forbid updates. Alternative boot methods should be forbidden too, as soon as the boards quit factory or development labs.

Note that enforcing boot protection does not prevent booting alternative operating systems for test or maintenance purposes. Only one constraint should remain: those systems should have a valid signature. Even if such a scenario is not recommended, a secure boot model could be implemented in such a way that a garage could boot a special signed version of an IVI/ADAS system for testing or maintenance purposes, even from a USB stick.

2.2. When Booting

Be sure to get the right kernel with the right options. The boot-loader should use cryptography to assert kernel validity. This technique is known as secure boot or trusted boot.

Make sure the system is the right one. Essential components in the system should be cryptographically checked using hashes. Linux integrates a module known as "IMA[1]" for Integrity Measurement Architecture. This module is intended for checking if files have been accidentally or maliciously altered. This can be completed with the Extended Verification Module (EVM[2]). EVM checks if extended attributes of files have been accidentally or maliciously altered. Another possibility, although less practical, is the use of the "verity" feature of device-mapper kernel module.

Be sure not to slow down system performance and lower user experience. Ensuring that the right code is executed implies intensive cryptographic computation. While in the past such operations without crypto-coprocessor might have degraded end-user experience to a hardly acceptable level, today modern hardware such as ARM-V8 with cryptographic extension do not face those limitations.

2.3. After Booting

After the system has successfully booted an approved operating system, it's time to run applications. Most user applications will not be bundled with the baseline operating system and should be downloaded independently. Many applications may also be provided and maintained by third parties with an independent life cycle.

On one hand, applications are easier to control than the base operating system. It is possible to upload them one by one on demand, and the installation process can rely on a working and secure operating system. On the other hand, there are hundreds of potential applications that a car owner may want to run and many of those applications may come from a "half-trusted" zone.

Before accepting an application, the system should:

- Check the code. Applications should be downloaded within a sealed package containing not only the application, but also its configuration, security rules, attached data. Application packages should embed cryptographic signatures to allow integrity checks, typically based on some hashes on content, with guaranteed origin and global consistency.
- Handle dependencies. Installing or upgrading software usually requires checking that its dependencies *(the other software components that are required for proper operation)* are satisfied by the system. A problem occurs when these dependencies cannot be satisfied. Two policies may then apply: either the entire

1 http://sourceforge.net/p/linux-ima/wiki/Home
2 http://sourceforge.net/p/linux-ima/wiki/Home/#linux-extended-verification-module-evm

software install/upgrade is aborted, or dependencies are installed/upgraded within a single extended transaction. Upgrading the base system should not raise dependency issues. It also should not introduce breakage to any existing applications. Ensuring that existing applications will never be broken over a ten-year period might be harder than expected. If a dependency issue is raised when installing an application, it is generally safer to cancel full installation and suggest a system wide upgrade. Applications can avoid such issues by embedding their own critical dependencies within their package. This may significantly increase package size and require more resources to download/install/upgrade. Nevertheless, doing so should lower the global effort to keep the system secured.

- Install the application. Installation processes should activate security rules into the file system. Until the full installation process is terminated, the application should not be launched. Adding new applications should impose the update of a trusted repository with new corresponding file properties and hashes. Doing so requires using a secret key which can be specific to the hardware (usually the SoC[3]).
- Start the application. The system should check application integrity before starting. This check has a performance cost and should take into account initial user experience. To limit impacts on performance, this operation could be cached to speed up any further attempt to restart the application.

Ensuring that the right code runs while still being able to upgrade it relies intensively on cryptography and requires keeping the system's "secret key" truly secret. Furthermore, although hardware cryptography is theoretically not mandatory, it keeps both performance and user experience at acceptable levels. In the real world, it is a must-have feature.

3. Keeping Secrets Secret

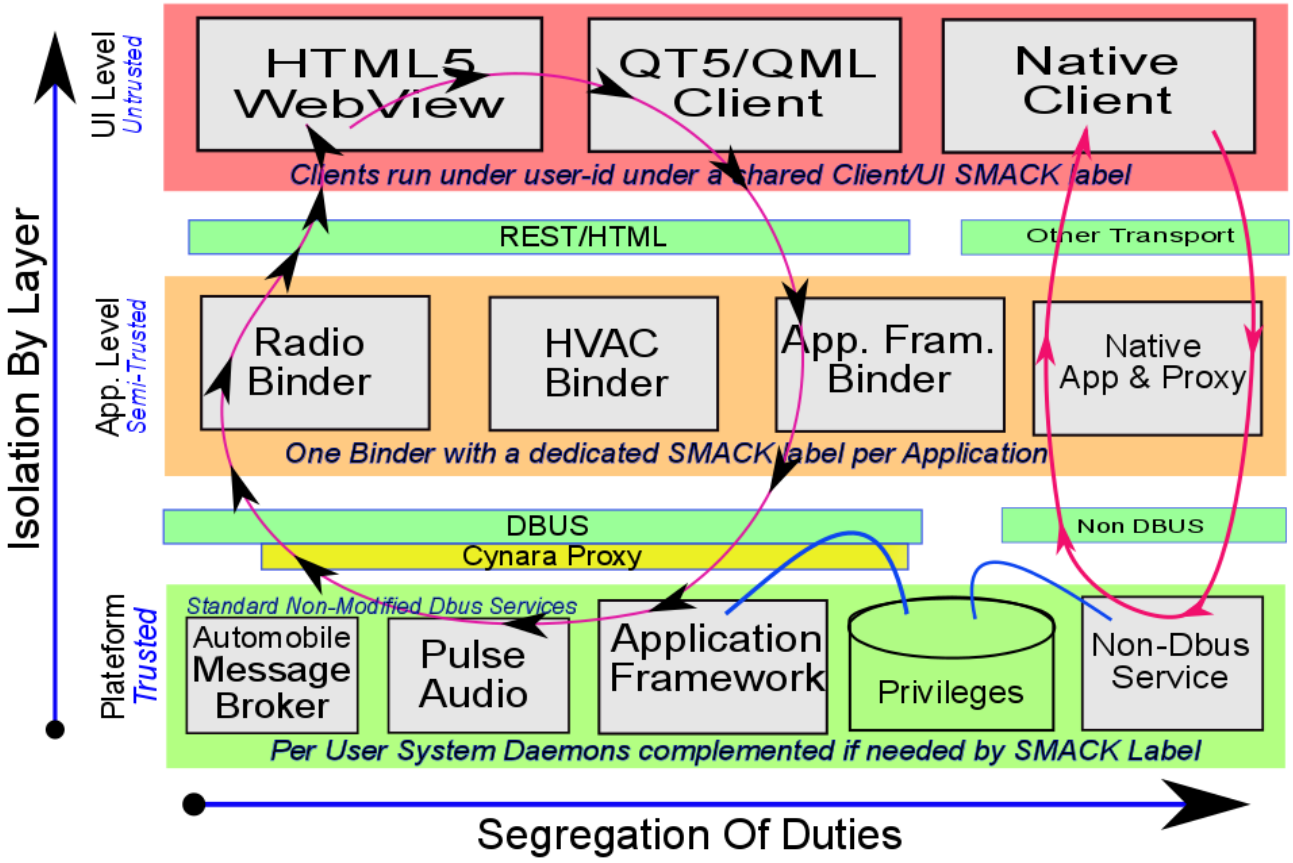
The only secret that will never leak is the one we do not know. In a perfect world, we would have no secrets. Unfortunately, this is not possible. A car system has many secrets: for DRM[4], software integrity or access to external services.

Computer security relies on isolating software components to prevent them from accessing non-authorized resources. Linux has multiple mechanisms for implementing those protections. They range from the good old file access control layer to newer discretionary and mandatory access control mechanisms implemented through Linux security models, without disregarding virtualization and network firewalls.

3 https://en.wikipedia.org/wiki/System_on_a_chip
4 https://simple.wikipedia.org/wiki/Digital_rights_management

One key issue in security is keeping things simple enough to be manageable. If security becomes too complex, people will either bypass it or introduce some security flaws by misunderstanding concepts or implementation. The only manner to achieve a complete but manageable security system is by working in layers and separating duties.

Example of Layered Security Architecture



Layers allow refinement of security from high-level security to fine grain profile control. At a high level, you may want to make sure that audio applications do not gain access to air conditioning. When at a lower level, you may want to make sure that media player can access phone's streaming service but not its contact book. Separation of duties on the other hand allows different persons to be responsible for different security aspects. A security manager should be able to supervise accessible resources for a given application or group of applications without having to even understand those applications.

3.1. Isolation

Keeping secrets implies building big walls with a few well-guarded doors. This fortress policy is also called isolation and can be implemented using numerous techniques:

- Virtualization:
  - hardware: access is achieved through a hypervisor [eg: KVM[5]]
  - software: relies on Linux namespace & CGroup as Docker or OpenContainer[6]
- Access control:
  - discretionary (regulated by owners)
  - mandatory (regulated by security rules)
- Permissions restrictions:
  - by user
  - by applications

All these techniques are complementary and can be used together or independently.

3.2. Virtualization

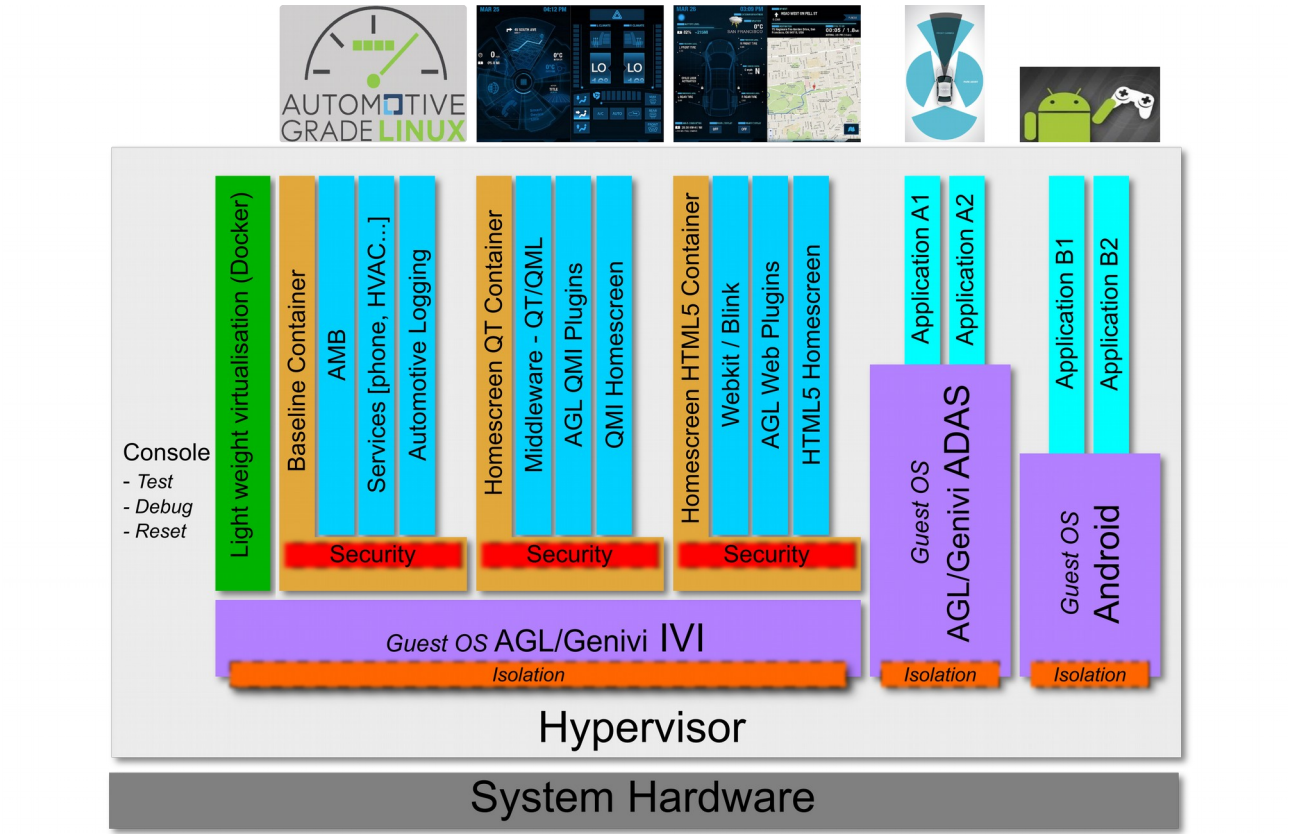
The virtualization goal is to create within a unique host system the illusion of multiple private sub-systems or guest systems. Any virtualization system provides mechanisms to limit resources usage (CPU, Ram, Filesystem...). Depending on the chosen model, virtualization can be more or less complete. In some cases, each virtual instance has a private kernel, with its own dedicated hardware resources. In other cases, almost everything is shared by virtual hosts: a unique kernel, file-system, RAM, I/O system with very minimal isolation such as a private set of libraries or some limits on CGroups to restrain RAM or CPU usage.

- Hardware virtualization: Each virtual machine runs a private operating system using hardware separation inside the processor under the supervision of minimal software called a “hypervisor”. Its potential use in the automotive could be to keep isolated the master ECU from the IVI while using a single hardware board.
- Software virtualization: The soft virtualization, also called containerization, is the application of a Linux mechanism called “unsharing”. This virtualization still has a cost but remains very light compared to hardware virtualization. In automotive it could be used to isolated untrustworthy applications from baseline services.

A key advantage of hardware virtualization is the ability to run different kernels on a single CPU and make sure that no failures can spread from one visualized kernel to

5 *http://www.linux-kvm.org*
6 *https://www.opencontainers.org* *https://www.docker.com*

the other. It is perfectly viable to envision a single system simultaneously running an ADAS system with real-time constraints, an IVI system on central display, and an Android implementation with games for rear seats.



On the other hand, software virtualization is much lighter. While hardware virtualization overhead can easily reach 30% of overall resources, software virtualization is typically below 5%. Software virtualization is typically used to restrain access to resources for untrustworthy applications. In a typical IVI system, any application not part of the certified baseline operating system should probably run within containers.

3.3. Access Controls

Access Controls are used to filter accesses to read, write or execution mode of resources exposed by Linux file-systems. This include files and directories but also other resources like RAM, processes, devices and network interfaces.

There are two main classes of access control:

- Discretionary Access Control (DAC): As the owner of a resource, you choose who is and isn't allowed to use your resource. This is the traditional Unix access control. They define and use the concepts of "user", "group" and "others". Users map to real persons or logical entities and are members of one or more groups.

In this model the owner of a file or directory can choose and change related access rules.

- Mandatory Access Control (MAC): Who can use your resource is decided by a piece of software called "security manager". The mandatory access control philosophy is based on the idea that the owner of a resource may not be a security expert. Today standard Linux kernels supports three security modules (LSM) that implement three different Mandatory Access Control models:
  - SELinux (Security-Enhanced Linux)[7] is the most complete solution but also quite complex. SELinux is used by RedHat, CentOS, and Android.
  - AppArmor (Application Armor)[8] is a light solution with some limitations. It is used by Ubuntu and SUSE.
  - Smack (Simplified Mandatory Access Control Kernel)[9] solution enforces simplicity and is mainly used in the embedded world.

SMACK seems to be the best fit for embedded automotive projects. This is proven as few projects have succeeded in using SELinux/AppArmor. It is light and simple and relies on extended attributes like SELinux without using file paths like AppArmor. Within the embedded world, SMACK was selected for notable projects such as Tizen and a Yocto layer is thus available for "meta-intel-iot-security".

3.4. Restricting Permissions

We can compare strong isolation of applications to running a business inside a private flat with very thick walls. Obviously, thicker walls do not prevent from having a door. From a security point of view, the thickness of the walls doesn't answer the question, "who holds the keys to the door?". Permissions are the keys to open the door.

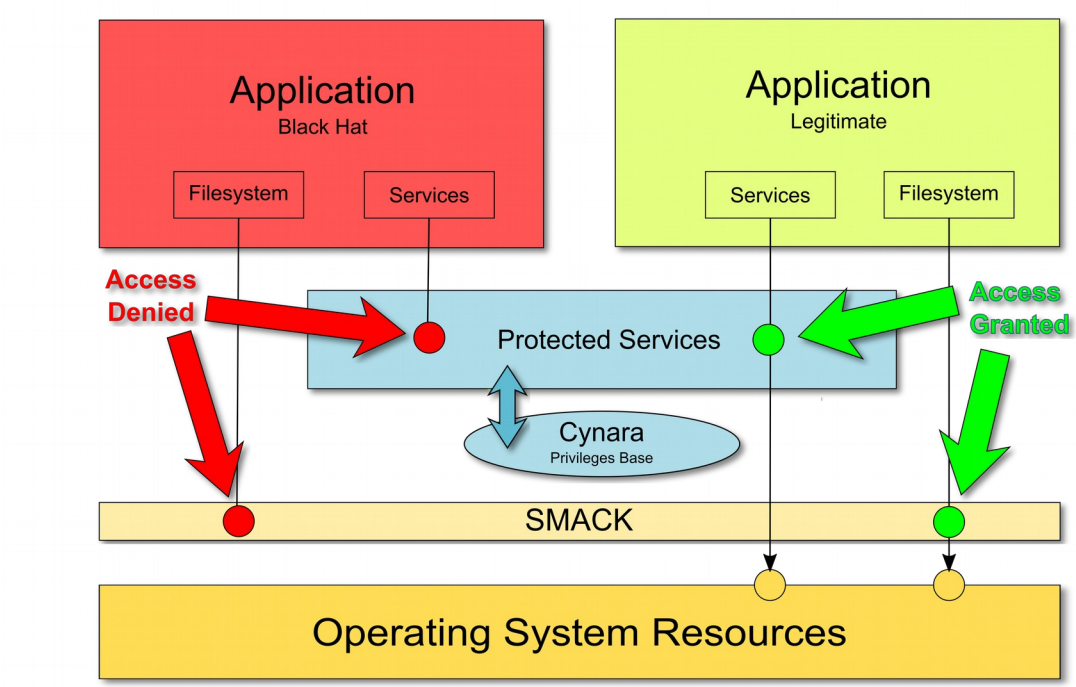
Access control [MAC/DAC] can be used for global security and isolation of a full system:

- Restricting permissions by user: Multiple users can be associated to a given car, such as the owner, the mechanic, the driver, and the passengers. The privileges of these users are not equivalent and some should have more rights than others. This is managed by permissions associated with users or profiles.
- Restricting permissions by application: An application has to request the needed permissions for proper operation. When requested permissions are granted, the application can then access protected services or resources.

7 http://selinuxproject.org
8 http://wiki.apparmor.net
9 http://schaufler-ca.com

Each time an "application" runs for a "user" it uses resources restricted by "permissions". The security framework should check if the triplet (permission, user, application) is valid or not.

SMACK & Cynara privileges handling



3.5. Leverage Hardware Capabilities

In theory most security mechanisms should be independent of hardware. But this is rarely the case in the embedded world where implementing full security without some form of hardware support remains almost impossible.

- ARM TrustedZone: This feature from ARM separates the processor and some peripherals into two zones: a trusted one and a non trusted one. Secrets can then be stored inside the trusted zone to avoid disclosure. This technology deeply inserted within SoC design makes it efficient to keep hardware secret.
- Secure RAM: Any encrypted secrets should be decrypted before usage. Nevertheless, decrypted secrets should be kept invisible on any hardware bus. To achieve this, the SoC provides secured RAM inside the trusted zone.
- Crypto co-processor. Measurements of integrity for installed programs imply computations of hashes like: SHA256 or MD5. Certifications of origin for downloaded packages imply signature checks. Secure transmissions (SSL/TLS) or disk encryptions require encryption like AES. All these cryptographic tasks are highly accelerated by using built-in processor instructions or coprocessors.

- UICC/SIM cards: SIM cards are intended for holding secrets and could be used for that purpose.

Techniques presented above offer a large panel of countermeasures for avoiding threats. With asymmetric cryptography it's obvious that private keys should remain secret. But the entire security technology implemented in a product should be reviewable by experts that have to audit the solution. In that perspective, open-source based technology is far more prepared in theory than proprietary closed sources engineering. Implementing in the open world allows continuous enhancement of the whole system QA, including security aspects.

4. Connectivity with the Outside World

Not only are cars connected to the outside world through Internet and cloud services, but they are connected to local mobile devices through Bluetooth/Wi-Fi and will soon be connected to the environment through ad-hoc networks for vehicle-to-vehicle interactions. This is without disregarding NFC, SD cards and increasingly sophisticated USB sticks.

Keeping a connected car secure is not simple. On one hand, users expect their cars to be more secure than their phones. On the other hand, users do not understand how a vehicle that costs 100 times more than a mobile phone may have less functionality. Users expect automatic updates, the ability to add new applications, automatic discovery and synchronization with their phone and home devices. And they expect their privacy to be respected, integrity of the system to be guaranteed, and all this throughout a car's lifetime.

Whether we like it or not, the next generation of vehicles will have to accept alien devices and applications. While it is still too early to predict the effective interactions between mobile devices and automotive systems, it seems obvious that at least on entry-level cars, users may want to use mobile phones as the primary interface for IVI features. On the other side of the connectivity spectrum, cars will have to synchronize with cloud services, the most obvious benefit being automatic updates of navigation routes or music playlists. Nevertheless, advanced use cases should go much further: uploading usage data, retrieving usage permissions, downloading new applications, etc.

While connecting local devices implies challenges, it remains known as problematic. On the contrary, connecting to the external world and exchanging information with the cloud is completely unknown to car industries. Those Internet communications offer new opportunities, but also introduce new risks. How can we balance the value of sharing data while keeping an acceptable level of privacy?

4.1. Network Connections

Communication with the external world will use existing network connections. Bluetooth to exchange with phones and tablets, Wi-Fi to exchange with user's house and garage, GSM/UMTS/LTE for mobile Internet access and probably NFC for keys, end-user authorizations or preference settings. Car-to-car technologies are still at very early stages and they should use "ad hoc" networks based on some form of Wi-Fi to allow direct connections.

Any connection to the external world introduces new security risks and potential security flaws:

- Tapping/Spying: connections need to be encrypted to avoid spying or identity usurpation (spoofing).
- Privacy enforcement: end-users love automatic pairing, but how can we detect intrusions? Device pairing should include identification to prevent cross-user spying.
- Abnormal behavior: unfriendly applications should not override expected behavior. Access to network resources should be restricted by application using permissions for both incoming and outgoing streams.
- Open software doors: any non-necessary resource should be locked down, especially facilities for debugging, development, and testing. System should not expose unexpected services to their external ends. Programs should not be able to open backdoors. However, there are still cases when allowing programs to create open connection entry points makes sense.
- Denial of Service: automotive systems should resist DoS-attack-like flooding.
- Intrusion detection: Security may lower end-user experience; for convenience they may choose to open multiple doors, reasoning that "I'm not a target" or "It won't happen to me". Security intrusions or abnormal behaviours must be reported for the user to understand risks and take appropriate actions.
- Include a firewall and an updated blacklist: Firewall and blacklist are common techniques for filtering content for child protection. It should also include an updated list of blacklisted servers.
- Implement Content Security Policy: To avoid cross-scripting attack, the W3C defines CSP[10], content security policy. With CSP web pages declare Internet resources as dependencies. Applying such a policy would allow detection of applications requesting access to blacklisted origins.

10 *http://www.w3.org/TR/CSP*

Despite the risks and constraints tied to external connectivity and especially to the Internet, the added value for both end-users and manufacturers is so great that this burden is justified.

4.2. Removable Storages

With the generalization of Internet-connected cars, the need to physically plug in devices may diminish. Nevertheless, removable storages will remain widely spread and used. Users may still want to use SD cards and USB sticks, usually to upload music but also in places where public Internet connectivity is limited or too unstable to upgrade the system (for instance, to download a country map while travelling on vacation).

Usually, removable storage is formatted using FAT file systems because it ensures having a wide range of readers. Unfortunately, FAT was designed far before the generalization of DAC/MAC access control mechanisms, which make it difficult or even impossible to secure such devices.

Even with these limitations, it's still possible to use removable devices in a secure way by:

- Extending system capabilities with a bigger SD-card: In this case the added memory becomes a native extension of the IVI system, formatted using a file system with adequate extended attributes. This type of extension should be transparent to the user, even if some extra care is taken to avoid data corruption when ejecting the card from its slot.
- Using a basic file system to import/export applications, playlists, or even boot in maintenance mode: In this case, security is not provided by the removable storage, but by the structure of the files provided on the device. Typically the application will be under the form of a structured ZIP file with adequate signature and authorization.

Removable devices always create some extra security threats. If an SD card is used as a disk extension, what happens when it is removed? Should it be encrypted to protect private data? Could it create a security flaw within any other IVI systems? In a perfect world, any removable device would be encrypted. But in reality, until there is a generalization of crypto-accelerators and transparently encrypted file systems, removable devices will remain a threat to integrity and their usage should be limited.

4.3. Connectivity to the Cloud

For any cloud exchange route through the Internet, all concerns listed heretofore apply. The cloud comes with an extra set of security risks. Vehicles and Cloud interactions are so new that almost everything still has to be discovered. Nevertheless, looking at other industries helps us to anticipate incoming issues that will need to be addressed:

- How do we protect user privacy? A car collects very valuable data about end-users. How do we protect this data and prevent it from falling into the wrong hands, when most users do not understand the issue and are willing to give up most of their privacy for a few minor privileges or rebate?
- How much should the cloud be trusted? If a user rents a car and downloads his music or preferred applications let alone personal contact lists, how do authorizations apply? When a user returns his car, how do we make sure that the cloud is updated with the right information and that all private data is removed from the rented car?
- What about funding? While some people may argue that funding does not impact security, the controller of a system has a direct impact on its security. The level of trust will vary significantly, depending on whether a user downloads information from his preferred social network or from a car manufacturer's trusted zone.

Connectivity to the cloud has a potential business impact and could become a significant source of revenue. It is a widely unknown field for car manufacturers. Combining money with something new and unknown definitely poses security risks.

5. Upgrading Software

Software update is a risky business. Either you update your system and take the risk of introducing new security flaws, or you decide not to update and take the risk of leaving some existing security holes open.

Regardless of how smart we are, every automotive system will require regular upgrades throughout the car's life. While it is still too early to predict the exact cycle for car software updates, it is probable that during the first 5-10 years, manufacturers will want to fix security/functional bugs as well as add new features. For older vehicles, they will probably limit updates to security and critical functional issues.

When talking about upgrades, it is important to separate base operating system updates with application updates. Not only are they generally issued from different authorities, but they run under different life cycles. Technically, applications should be upgraded individually when required, while the base image should be upgraded in a single transaction to guarantee global consistency for the full system. In any case, we should make sure that we download and run the right code. Upgrading the baseline system is always more challenging. Not only does the base image include key initial security rules used as a foundation for everything else, but if for any reason this update fails, we risk bricking the entire system. Regardless of the upgrade model we choose, a newly upgraded system will, at worst, restore from an initial or previously stable state.

5.1. Upgrading Baseline System

Common models to upgrade baseline system:

- On desktops and servers, baseline operating systems are upgraded by components such as applications. This "by component" approach has the potential for being very flexible and modular. Unfortunately, it generates significant workload on the target and does not really comply with embedded system capabilities.
- On smaller devices, it is far more common to update the baseline system in a single transaction. Mobile phones typically download new versions of operating systems within user space, and after adequate verification flash it to boot partition. In this model, the boot partition is entirely prepared on server backend and requires only minimal resources on the embedded target. Nevertheless, even if everything is heavily controlled before starting the flash boot operation, theoretically impossible incidents are always possible. A quick Internet survey on "Bricked[11] my phone during update" is enough to confirm that what shouldn't happen inevitably happens from time to time.
- To avoid the "bricking issue" it is possible to set up two boot partitions. The upgrade manager may then use them in master/slave mode. One partition is reserved for a special version of the system dedicated to initialization and test. In theory, with two boot partitions, regardless of what happens during upgrade operation, in case of failure the user should succeed in restarting the last stable version. In reality though, since software generally supports updates and has adequate logic for updating configuration or data files at the first run, moving backwards is rarely well supported.
- The last model is a mix of the two previous ones. It requires a file system that implements snapshots and rollbacks [eg. BTRFS[12]]. It also requires a boot loader that "understands" corresponding file system structure. In this model, a snapshot is done before each new update of the system. In case of failure during an upgrade or an upcoming boot, file system rollback restores the previous working system. Otherwise, the new system is committed.

At least for the near future, the extra resources consumed by snapshot aware file systems makes them harder to select for embedded deployment. Since the market is not ready to accept the risk of bricking a car during an upgrade like this (which happens sometimes with phones), most automotive systems should rely on "two-boot partitions" for baseline upgrades. Nevertheless as using master/slave model to rollback to previous stable version may end up hard or impossible, finally most automotive systems may select the two-boot partitions model with the first one frozen at production time with installation, restoration, debugging, and testing, and the

11 *https://en.wikipedia.org/wiki/Brick_%28electronics%29*
12 *https://btrfs.wiki.kernel.org/index.php/Main_Page*

second one with the active system. If something ever goes wrong, the end-user may always return to the initial state and download the latest stable version. The same model could probably be used when a user sells his or her car and wants to make sure that all private data is removed.

5.2. Upgrading Applications

The mobile world teaches end-users to manage applications through app stores. The final model for automotive is not settled yet, but we already know that initial installations as well as application upgrades should be secure and that no application should run within a vehicle without some form of certification.

From a security point of view, updating or installing an application is a technique equivalent to the one suggested during "Run the Right Code". The main difference with upgrading comes from configuration files and user data. With a new application we start from a blank sheet of paper, but with upgrades we have to reuse existing data. Unfortunately it is quite common that data formats and/or configuration files change from one version to another. One option to solve this issue is to leverage a feature equivalent to "OpenWRT UCI1"[13] where each application's configuration is centralized and rebuilt on demand at the application starting time.

5.3. Very Long Time Support

One very specific issue to the automotive is the very long time support constraint. The common car lifetime is 20 years or so, while the best case scenario for typical long time support in the IT industry is limited to a few years. Within the Linux community, RHEL/CentOS is the only one committing to 10 years long time support. Every other major player has limited themselves to 18-36 months. LTSI[14] kernel follows more or less the same rules and proposes a stable-quality kernel tree for the typical lifetime of a consumer electronics product, typically 2-3 years.

Will it be possible to upgrade the baseline system on a 10-year-old car from kernel 4.x to 8.x? Hardware may or may not support it. CPU/RAM performance may degrade user experience to an unacceptable level. On the other hand, if manufacturers stick to an older version of the kernel, this will translate in back porting hundreds of security patches.

The best practices and working models remain to be discovered for "very long time support". It is probable that during the first years of a car, the system will be fully upgraded including application improvements as well as kernel updates. But for old cars, only major security flaws and dysfunctions should be back ported to previous versions.

13 https://wiki.openwrt.org/doc/uci
14 https://ltsi.linuxfoundation.org/what-is-ltsi

"Very long term support" issues should not introduce any security flaws in the near future. But in the long run it may become a critical issue. In Europe, the average car age is 9.65 years and more than 30% of cars are over 10 years old.

6. Global Integrity

Even if automotive systems operated correctly under normal conditions, there are cases where they may encounter unexpected bad conditions. Even during those bad scenarios, global coherency should be guaranteed.

Typical "bad conditions" include:

- electrical: sudden power off, lightning, shocks, brownouts
- stress: memory exhaustion, huge workload, incoming traffic
- weak connectivity to networks
- human error or misuse of the system
- component or device failure
- accident: car crash, fire

Technically "bad things" should never happen, but Murphy's Law warns us that not only will they happen, but they will happen at the worst possible time. The only valid option is to also integrate "impossible scenarios" and make sure the system will survive through them.

6.1. Resist Sudden Power Off

Powering off the system is the easiest attack: doing a short cycle of power on/off should never bring the system to an unstable or attackable state.

Most of the bad electrical conditions can be filtered using good electronics. However the overall system must be able to resist sudden power failures. In those situations, file system is always first in line and only a logged or read-only file system may resist the sudden power fail. Nevertheless, protecting the file system is not enough. Power failures should also be integrated within other scenarios, at a system level or within applications:

- firmware, system or application installation, un-installation or upgrading
- system settings edition, user CRUD (Create, Read, Update, Delete) operations, administrative work
- synchronisation of data with network or removable storages
- management of user sessions

In all these situations, the system should survive sudden power failures and in the worst case should be able to recover a functional level from a prior stable state.

6.2. Resist Stressful Conditions

All automotive systems should reserve enough memory and CPU to keep their integrity and deliver a reactive HMI to the driver at any moment. Execution priorities should be managed to ensure that the system is always available for urgent action.

6.3. Weak Connectivity to the Internet

Cloud based services like storage and synchronisation are very valuable for the end-user but may reach a critical situation under low connectivity conditions, which may easily happen in underground parking. The global integrity of the system cannot rely entirely on cloud services and all automotive systems should run smoothly with or without connectivity.

7. Conclusion

Connected car technology is still in its very early days. Most attack patterns or automotive best security practices have not been discovered yet. We can look for existing technologies and leveraging lessons learned from other industries (phone, TV or ADSL). Also, known practices of today's threats will not be applicable in time but new threats will always be tried over time. As connected cars are more exposed to malicious people, they gain new links that help keep their protection up to date. We can assert with a high level of confidence that existing Linux technologies should permit securing connected cars.

Like always with security, the difficulty will remain in the details, and having the right collection of security technologies will not be enough. A smart integration should propose a compromise between simplicity, performance and security, with a comfortable system not only for the end-user, but also for developers or mechanics. If at any point in time users fails to understand how to operate their automotive system with security activated, then security will end up being either partially short circuited or completely deactivated.

Despite existing security technologies, some significant efforts remain to be seen. While we understand attack patterns on mobile or cloud applications, nothing equivalent exists for car attack patterns.

Cars will be connected and connected cars will be attacked. The risk is known, and technology to respond to this risk is available. Building a serious security architecture tailored to the automotive industry is doable, but will require a significant effort.

8. About the Authors & IoT.bzh

IoT.bzh is a partner of Renesas and works with the Open Source community in proposing a "Ready to Go" Linux automotive distribution. IoT.bzh technical team is known in the automotive industry as the formal "Tizen-IVI" team and was responsible for building and maintaining Tizen distributions until Intel released the project. Today, IoT.bzh works with Linux Foundation AGL Project & GENIVI Alliance under the Renesas partnership on proposing a production grade Linux dedicated for the automotive.

Fulup Ar Foll holds a Master degree in Computer Science from the Military French School ESAT in Paris. He started as a research engineer for ten years before joining the Industry. He then took the technical direction of Wind-River in Europe, before moving to Sun-Microsystems where he worked at scaling Internet for mobile and Telecom industries. Fulup regularly talks in multiple international conferences. He was an active member of multiple standardization organizations. He currently works as CEO and Lead Architect for IoT.bzh and is a delegate of Renesas inside AGL & GENIVI.

José Bollo started his career in the industry with ten years as developer for measures automation. He then worked as an engineer-researcher in audio signal processing: 3D rendering of audio signal, noise reduction, echo cancellation, and watermarking. During the last three years, José worked on the Tizen project where he was in charge of the Tizen-3 security model. He currently works as a Security expert for IoT.bzh and is responsible for the next generation of security stack for automotive Linux distributions.

Automotive Grade Linux is a collaborative open source project that aims to accelerate the development and adoption of a fully open software stack for the connected car. Leveraging the power and strength of Linux at its core, AGL is uniting automakers and technology companies to develop a common platform that offers OEMs complete control of the user experience so the industry can rapidly innovate where it counts. The AGL platform is available to all, and anyone can participate in its development. Learn more: http://automotivelinux.org/.

Automotive Grade Linux is a Collaborative Project at The Linux Foundation. Linux Foundation Collaborative Projects are independently funded software projects that harness the power of collaborative development to fuel innovation across industries and ecosystems.www.linuxfoundation.org

Thank you for the contributions, review and support from: Stéphane Desneux, Yannick Gicquel, Manuel Bachmann, Joel Hoffmann and Hisao Munukata.