print | close

Automotive Security and the Chain of Trust

Electronic Design

Justin Moon

Tue, 2015-11-17 14:20

Automotive systems are inherently complex and becoming even more so. Increasingly, vehicle manufacturers are consolidating disparate functional components, including infotainment (navigation, multimedia, speech recognition, content management), telematics (remote diagnostics, occupant safety and security, tracking), driver information (instrument clusters, heads-up displays), connected services (OTASL, V2X), and advanced driver assist.

Such an approach makes for a more integrated cockpit experience, but also ratchets up the complexity of a vehicle's architecture. Add in the growing connectedness of cars, and you have greater potential for security vulnerabilities and greater access to a large number of attack surfaces. Automotive subsystems are exposed to potential security exploits, including rendering system services unavailable, unauthorized access to data at rest and in motion, and unsolicited remote access to system services.

Related

What's the Difference Between Secure Comms and Secure Systems?

Can The Internet Of Things Be Secure?

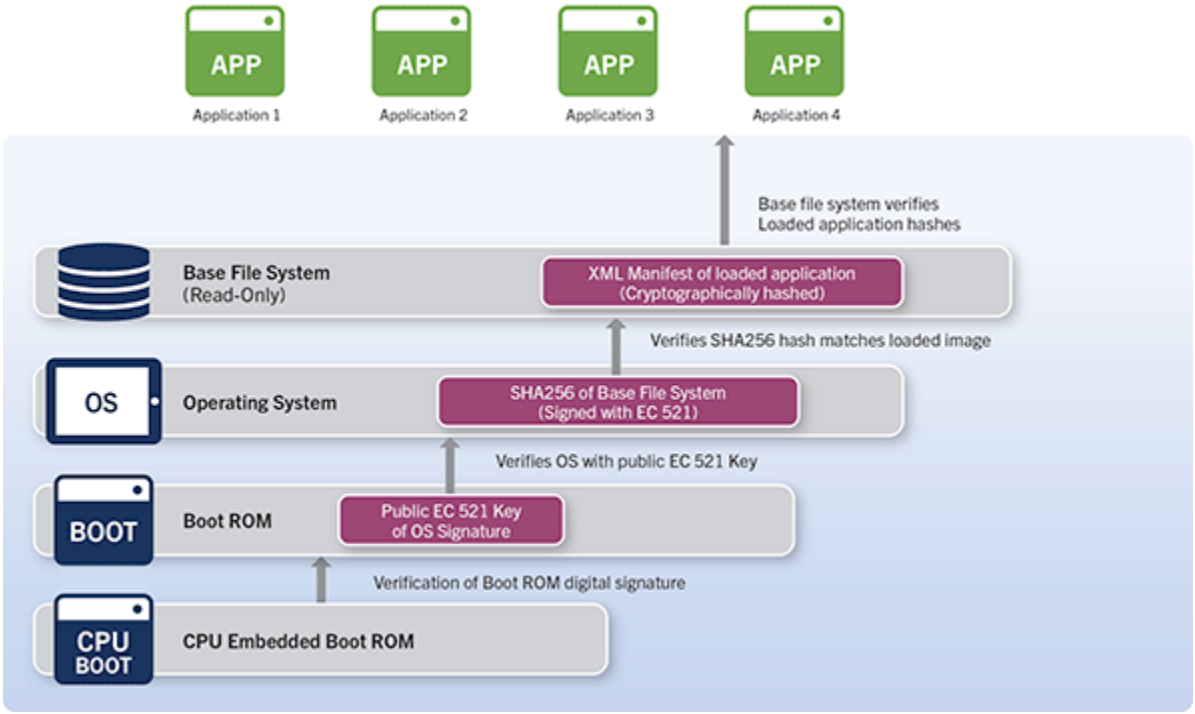
The Many Faces of Embedded Security

Recent research papers have identified theoretical and practical vulnerabilities stemming from native code exploits, most commonly buffer overflows and issues with authentication and authorization management. Hardening of attack surfaces—not just those identified in research papers and the mass media, but in all requisite subsystems—can mitigate the risk of exploit.

Hardening via a Chain of Trust

The best way to harden attack surfaces is to apply a "chain of trust" — fundamentally designing systems that leverage security resources (from the reset vector through the boot process to the fully initialized system) and employ a secure operating environment.

The first link in the chain of trust involves silicon vendors and hardware-enforced security. Securing the boot process, both in terms of trusted code execution and proving the authenticity of the boot chain, is key to ensuring a fundamentally secure environment. Hardware-enforced secure software execution environments such as ARM TrustZone or Intel Trusted Execution Technology can provide secure key stores (keys injected at manufacture to guarantee device validity).



Securing the boot process can come with tradeoffs, including lower performance. It will take longer for a system to boot in a secure fashion. Why? Because the system must follow a number of steps to ensure the boot process is in fact secure *(Fig. 1)*. For example:

- Upon application of power, the system must verify the boot ROM's digital signature to ensure that the mechanism for loading the OS and the rest of the system software is intact.
- Using a public/private key verification (preferably stored in a hardware-enforced secure environment), the system can verify the initial OS software as correct. Not all components need to be verified as a single blob of code. Individual components can be verified in stages to satisfy boot performance requirements.
- Prior to loading the contents of a file system (if present), the system can also verify the contents, through a hash.
- Once the file system is available, the system can initialize and execute additional system services.

While hardware is certainly the starting point for secure design, it alone doesn't ensure a secure operating environment. An operating system (OS) that's certified in the realms of both safety (IEC 61508 or ISO 26262) and security (Common Criteria EAL) offers a significant advantage. Security qualifications such as CC EAL 4+ provide a structured approach for evaluating various aspects of security, including user data protection, identification and authentication, and resource utilization. The chain of trust is reinforced by the use of software components with both a security pedigree and a safety pedigree, preferably proven with a certificate.

Safe and Secure

Is safety really an element to a secure system design? Functional safety requirements will become commonplace outside the realm of function-specific engine controllers, as digital instrument clusters and advanced driver assistance systems (ADAS) become more complex.

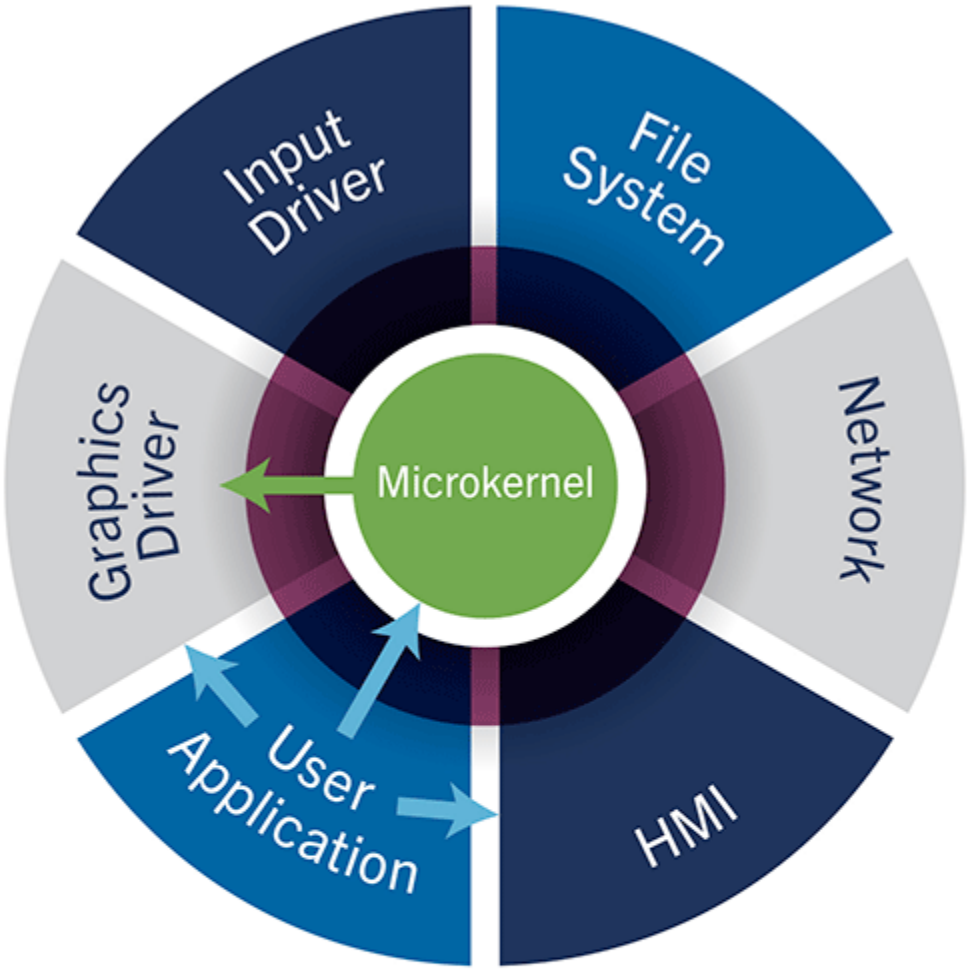
Safe system design dictates the system's ability to meet functional requirements, handling the possibility of random faults in the safest manner possible. *Secure* system design dictates the system shall meet functional requirements, handling the possibility of directed and or malicious attempts at failure in the safest manner possible. A system needs to react to failures, or to protect from the presence of said failures, in the safest

nner possible.

Critical systems of a vehicle are typically safety-related. Access to these systems needs to be granted in the most secure way possible. Although the methods and philosophies of security and safety differ, in fact one may not be an element of the other, they should be considered synonymous when dealing with any system (in-vehicle or otherwise) where human interaction is possible.

Exploiting OS Resources

A system's security can be further strengthened by making judicious use of the resources available through the OS *(Fig. 2)*. The OS kernel needs to be as small as possible, providing only core services (such as memory management and process management), with all other services (such as file systems and device drivers) running as memory-protected processes in user space. The OS needs to keep these processes isolated from one another in terms of runtime environment, memory, and CPU utilization. Process isolation helps avoid vulnerabilities by limiting access to functional blocks (system services) on a given device.



Contains less code

- Avoid vulnerabilities and attack surfaces
- Simplify security verification and testing

Designed for resiliency

- Kernel isolates processes in user space
- Restart unresponsive processes without affecting others

Minimize number of root processes

- Run only essential tasks as root

This spatial separation is accomplished through comprehensive use of the memory-management unit (MMU) integrated into a large number of popular embedded CPUs employed in the automotive industry today. Memory-management routines implemented within the core of the OS map each process's virtual-memory addresses to physical memory, providing full memory protection. Temporal separation, or the management of CPU time, also helps to ensure the security of the running process in a given system. The OS needs to schedule

cesses, both periodic and aperiodic, while at the same time ensuring system deadlines are maintained. licious code has the potential to change the system's functional behavior.

The operating environment should also provide the ability to guarantee minimum budgets of CPU time to defined groups of threads, without wasting unused processing time. This effectively accomplishes two things: system functionality can be separated at the thread level into designated "buckets," and the scheduling of these buckets can ensure that unexpected system loads (like malicious code) don't affect the secure operation of the system.

Furthermore, the OS needs to support stack properties, such as cookies, that can help identify and trap foreign code. The compiler randomly generates stack cookies during the build process, which are then placed on the stack upon function (source code) entry and validated upon function exit. Attempts to write beyond the limits of the function's frame on the stack will result in the cookie being overwritten, causing the offending process to terminate. The APIs and tooling provided with the OS should make it possible to minimize the potential for buffer overflows. Common API exploitation occurs through various functions that perform memory access and string manipulations. Buffer overflows are a common exploit point at the system service level.

In addition, the OS should provide the ability to build position-independent code and other facilities that make it harder for attackers to locate target processes. For instance, address space randomization is key to ensuring that processes are not running at the same address from boot to boot.

Authorization Management

If implemented correctly, authorization management—effectively carving out system services and resources for a given process—can allow for proper execution while offering fine-grained permission control. The necessity for a process to use and maintain root privilege during runtime can be minimized significantly by implementing an algorithm for authorization management. This algorithm must provide a request/grant system that's more flexible and controllable at runtime than that of the standard POSIX permission models. It would allow processes to drop their "root" privilege and still maintain certain prescribed abilities.

Any task that happens to be compromised through an exploit has only a tiny subset of the privileged operations available, greatly reducing the attack surface. Access control lists (ACLs) could be created to deliver more fine-grained control of file access. Files and file systems should be able to be identified as trusted or not, and programmatically identified as such. This idea of "pathtrust" can be leveraged to ensure that executable files are loaded from only trusted locations on the device, minimizing the ability to inject foreign binaries.

Application sandboxing is a natural extension of authorization management. Applications that are not considered system services, but are relevant to the feature set of the overall in-vehicle system, should be capable of running in their own environment, completely segregated from one another as well as the rest of the system. This sandboxing model should be able to specify the capabilities attained by the application and all resources required to perform the application's function. It should also provide appropriate file access and a directory structure that's isolated from all other applications in the system.

Building upon these isolation methods, the operating environment should provide a mechanism to ensure that data at rest is as secure as possible. An encrypted file system is one method to ensure this. The entire file system does not have to be encrypted. In fact, the file system should be flexible enough to support multiple secure domains, each containing multiple files and directories. These files and directories can be in a locked or an unlocked state. Domains themselves should also be flexible enough to be created or destroyed on demand, with different keys depending on the key management strategy.

Hypervisor Support

ıe isolation methods provided by the OS are considered insufficient, or the system security model calls for n more isolation of safety- and security-critical components, a Type 1 hypervisor may fit the bill. Type 1 hypervisors, which run on the bare metal, provide a level of isolation between operating environments. A hypervisor model can allow system designers to isolate safety-critical elements in a “locked down” OS while running non-safety critical services in a more connected environment.

A well-implemented hypervisor will have minimal impact on overall system performance, while also providing better isolation than the shared kernel solutions used for symmetric multiprocessing or asymmetric multiprocessing environments. A Type 1 hypervisor could also be configured to include a strict firewall between the host and guest operating systems, marshaling any and all cross-boundary accesses. It’s worth noting that a correctly utilized hypervisor solution doesn’t limit architecture or design. It allows system designers to configure individual components of the operating environment, providing the flexibility to design a system with maximum isolation for security elements, without affecting the function of each component. The links in the chain of trust remain strong throughout the operating environment.

Encryption forms a significant part of any device security strategy. A secure operating environment will help protect from external exploits, but encryption can guarantee that, even in the event of a breach, data remains safe. FIPS-certified encrypted file systems for data at rest and the same level of security over the TLS cryptographic protocol for data in motion should be a significant consideration. If the OS selected for the design includes these components, one less piece of the security puzzle needs to be custom-implemented. When available, hardware-based crypto engines can help increase overall performance while maintaining a secure path of execution. Without encryption, it’s impossible to maintain the integrity of a chain of trust.

Clearly, many links can be forged in a chain of trust to help system designers enforce security. The hardening of attack surfaces through hardware-assisted security, a certified OS, and the proper use of system resources is critical. These considerations are, however, best applied as part of a well-rounded security solution that includes capabilities like encryption and key management along with secure inspection models and testing practices. This comprehensive approach is essential to meeting the rapidly increasing demands of device security.

Source URL: http://electronicdesign.com/embedded/automotive-security-and-chain-trust