Vehicle systems provide a challenging use case for security, where physical and virtual access is essentially unlimited, and reverse engineers have easy access to replacement parts, components, and even whole vehicles – enabling multiple tries to perfect attacks on the system. Given these challenges in the vehicle deployment model, the goal shifts from denying an attacker access, to limiting the damage an attacker can do once access is achieved. Specifically, the integrity and availability of the system (vehicle) must be maintained even after a successful compromise. Security solutions built into Automotive Grade Linux (AGL), such as Smack, Cynara and AppFW can help increase the integrity of software applications, but are not sufficient by themselves to close significant vulnerabilities in the platform.

This talk will take a hands-on approach to securing an AGL system, using the CES 2017 reference platform, application stack, and peripherals as an exemplar. The talk will show how to apply Smack, Cynara and AppFW to reduce the attack surface and enhance the security of the IVI platform. With existing security solutions applied, the talk will switch to attack flows, vulnerabilities that remain in the system, and suggestions for continuing to improve the overall security of an automotive software stack such as AGL.

System integrators and developers generally fail to understand the scope of these security solutions, and what the overall threats to the system are. Security solutions such as Smack and Cynara generally assume a closed system, and assume an attacker won't be able to achieve direct execution access on the system. Unfortunately, this assumption is incorrect. Direct execution access can be achieved through a compromised root or administrative account, application vulnerability, shared library, latent privilege escalation vulnerability or even external peripheral hardware. Some of the vulnerabilities remaining in the system after the application of tools such as Smack, Cynara and AppFW include:

1. The reliance on a privileged account capable of modifying access permissions, policy databases, and mandatory access control mechanisms.
2. No protection of shared library replacement, addition, reversion, validation, etc.
3. The ability to load kernel modules
4. The ability for an attacker to bring their own tools (i.e. executables, libraries, etc.)
5. Limited protections on application configuration
6. No protection around secure update mechanisms including keys, certificates, block devices, etc.
7. No system call enforcement
8. Extraneous kernel functionality (i.e. kexec, drivers, etc.)

The talk will focus on applying additional in-kernel, and user-space capabilities within the AGL environment to address unmet threats and vulnerabilities. Additionally, the talk will discuss integrating security controls into the development, build, and test environments based on several real-world use cases. The talk will help take the application of security on AGL as an afterthought, bolt-on solution, to a well thought, full-spectrum, fully integrated solution which is integrated early in the development process. The talk will also focus on

helping developers and integrators identify the full spectrum of attacks their systems may be vulnerable to.

AGL-based systems are vulnerable to a variety of attacks, including: physical, over the wire, and through the interface. The talk will primarily be focused on over the wire and through the interface attacks, as they present the greatest opportunity for an attacker to achieve attacks at scale. However, it should be noted that an attacker with physical access can (and will) use software access and reverse engineering techniques to help develop attacks at scale. As an example, the highly-publicized Jeep attack, made use of physical access to the platform, to develop their attacks for use "at scale" and remotely.

Existing AGL security solutions attempt to raise the bar for an attacker, but they need to be augmented with additional kernel functionality to achieve the maximum desirable effect. These security solutions should be implemented with other low-hanging fruit and industry best-practices such as: lockdown/removal of engineering and flash update tools, complete removal of unnecessary tools and libraries (attack surface), secure boot founded within a hardware root of trust, and having the system evaluated by an external entity.

Attack Flow

Regardless of whether an attacker is attacking a system through virtual access (over the wire, software updates, etc.), or physically they will generally follow the same steps to gain access to the system. Using the Jeep hack as an example, the talk will follow the attack flow through the system, identify security components that could make the system stronger with a special emphasis on areas where existing AGL security capabilities fall short. The general flow for an attack starts with system reconnaissance. Next, an attacker will exploit a vulnerability to gain execution on the system. After achieving initial access, they can escalate privileges if necessary, or continue to exploit the system to achieve their individual goals. The attack process is often circular, and cycles between reconnaissance, gaining access, performing system analysis, and increasing access. It is often necessary to perform several iterations of the process to achieve the intended system exploitation goals. Existing AGL security solutions, coupled with other solutions to address residual vulnerabilities, provides opportunities to mitigate or prevent attacks at each step of the attack flow.

Mandatory Access Control (MAC)

MAC enables system developers and integrators to restrict access to system resources and data to specific users, classes of users, applications, hardware/software objects, and sub-systems. MAC provides logical isolation between applications running on the same system and it limits what an attacker is able to do once initial access has been achieved. This is true whether the access is through a vulnerable application, system bus, or shell console. When implemented correctly, MAC restricts what an attacker can do once they have already gained access, and provides continued integrity of the rest of the system. MAC (in the traditional

sense), is primarily implemented using OS and kernel extensions.

Fundamentally, AGL security tools such as Smack, Cynara or AppFW implement MAC. These security tools enable system integrators to limit access to resources, communication mechanisms, OS facilities, hardware, and separate applications, libraries, and data. MAC can even limit what the *root* or *administrator* user can do and access on the system, making it even more difficult for an attacker to gain total control of the system and exploit the rest of the vehicle systems. MAC can be thought of as a series of explicit policies, governing what exactly a user, application or defined role can do on the system. MAC policies are generally either established during a learning phase, in which normal system activity is recorded over a set interval, or the policy is explicitly defined by the system developer. In the case where the policy is developed based on actual use of the system over a defined interval, the established policy can be tailored to provide additional protection, and address paths or access vectors not exercised during the learning cycle. The MAC policies are used to identify permitted activities and accesses on the system, under the premise of that which is not explicitly permitted is denied.

Smack extends MAC to the network stack, enabling network traffic to be filtered at the per-application level. Additionally, Smack simplifies the implementation of MAC, by enabling default labels or contexts to be applied to filesystem objects that are unlabeled. Smack also provides the concepts of containers, and utilizes several containers to simplify the implementation of Smack on AGL. AGL's implementation of Smack is primarily focused on separating users, from system and IVI processes.

Cynara extends Smack to include Android style, fine grained permissions. Cynara implements a policy check service that can be utilized by a library and dbus components. Cynara enables integrators to extend MAC from the OS-level (ie. files, and sockets) to individual API calls within applications and libraries.

Leveraging MAC on a system such as an Electronic Control Unit (ECU) or In-vehicle Infotainment (IVI) platform, significantly increases the burden for the attacker, and works to limit the exposure once an attacker gains access to the system. During the initial access phase, MAC decreases the available attack surface, and can make it harder to gain execution, alter system firmware, interact with hardware devices, and elevate privileges. Attackers will need to spend more time and resources analyzing the system and preparing attacks, thereby enabling more chances for detection, mitigation and prevention of the attack. MAC can also be used to limit the resources available to an attacker, thereby significantly decreasing the available attack surface, and increasing the burden for analysis as attackers are forced to look for even smaller attack vectors. Once an attacker gains execution on the system, MAC can prevent the attacker from elevating privileges, subverting trusted boot mechanisms, interacting with other components of the system, and transferring aspects of the system for

offline analysis. MAC can also be used to prevent an attacker from bringing their own tools (by limiting access to system calls, not being able to execute untrusted applications, and preventing execution access from directories the attacker controls), which further helps preserve integrity of the system.

Decreasing the Attack Space

In addition to the AGL MAC tools, it is important to decrease the attack surface of both the operating environment and the protection services to minimize the attacker's opportunity to subvert the protections and gain complete control of the vehicle's systems. For example, simply removing binaries (i.e. gdb, strace, kexec) from a system does not mean the functionality has been removed. The functionality may still exist (i.e. in terms of system calls and kernel functionality), and if an attacker can bring their own tools, the latent functionality can then be leveraged to further knowledge of the systems and increase an attacker's access to the system. Functionality should be removed from the host operating environment first, and if necessary, supplemented using a hypervisor and trusted execution environment. Additionally, systems should implement some sort of application whitelist, which works to prevent an attacker from bringing their own tools, exploits, and applications.

Software Partitioning and Containerization

Increasing the protections afforded by MAC, can be achieved through software partitioning and executing it within its own context. Available software partitions mechanisms include separate processes, lightweight containers, and virtualization. Software can be partitioned using logical divisions, such as independent functions, existing IPC and synchronous mechanisms, or resource-based slicing. Software partitioning has numerous benefits including reduced attack surface, better performance, reduced complexity, and better scalability. Software partitioning can be done using both manual and automatic (software-based) means. Like many trade-offs in security, the means available for both slicing and containerization are often limited by existing designs, and whether the systems are being designed from the ground up to support containerization and slicing.

Splitting applications into multiple process spaces is not always feasible, such as is the case for interpreted applications like Java. Additionally, applications running within separate process spaces are still vulnerable to a variety of kernel and user-based attacks including snooping, side channel analysis, and privilege escalation.

Lightweight containers such as docker or chroot environment provide stand-alone execution environments for application slices. These environments execute within the same kernel, but have increased separation and greatly reduced attack space, as they are essentially light-weight virtual machines with only the necessary kernel interfaces, applications, libraries, and network

services running. Lightweight kernels execute using the existing process and resource scheduling making them more highly performant and easier to manage and deploy. Additionally, the provisioning of containers provides another mechanism for enforcing an additional level of security within the AGL environment.

MAC and containerization implemented in the kernel provides application and resource separation, however it also runs at the same achievable-access level as the attacker, providing the attacker a potential mechanism for subverting the afforded protections and escaping the container. Using a hypervisor or separation kernel, not only are the protections removed from the purview of the attacker and moved to a lower level in the system (where they can be monitored and verified by other hardware protections), but additional access controls and separation can be enforced. Using a hypervisor for separation also enables each application or service to be executed in its own context independent of other applications or services on the system. Placing each application in its own execution context not only guarantees resources, but also enables finer grained control over resources, and access to the rest of the system (i.e. peripherals, data, etc.).

Running each application or service in a separate execution context (whether container or virtualization environment), enables the application to be executed in a similar fashion to a unikernel. As such, only the required system services, access mechanisms, and resources are available to the application, significantly decreasing the attack space, and increasing the protections afforded to the system and application.

Separate execution contexts can also be created for encryption services, enabling the encryption of applications, libraries, data, and other resources to happen transparently behind the scenes, without exposing the operations or keys to an attacker. The combination of hardware acceleration, offloading to a hardware security module (HSM) and implementing encryption, key management, and signature checking in a separate execution context or virtual machine, can make it increasingly difficult for an attacker to dump or analyze the keys and subvert the afforded protections. Additionally, a separate execution domain can be established for software and firmware updates. The update and maintenance execution context can be configured as the only environment that can access hardware write mechanisms, security processors, and trusted boot mechanisms, which themselves can be protected by other security mechanisms within the larger integrated platform.

Wrap Up

Using the AGL CES Demo and Jeep hack as exemplars, this talk will take developers and integrators through providing a complete security solution for their AGL platforms. The talk will

focus on identifying classes of threats AGL developers need to be familiar with, the limits of security solutions integrated into AGL, and applying solutions within the Linux ecosystem to address various threats to vehicle platforms. Developers will leave with the skills and experience to develop threat models for systems built using AGL, a better understanding of AGL security solutions and their limitations, and how to use other solutions available within the Linux environment to address various threats to a system, and to integrate security into product life cycle. The talk will focus on identifying vulnerabilities and limitations in AGL security solutions, and layering security solutions to provide more complete protection for AGL platforms.