ARMOUR: Smart Secure Policies

1 Brief

This project will develop an infrastructure to provide data protection for loosely coupled networked applications. The infrastructure will be supported by a security policy language and a runtime control-plane to realize data-driven application level access control.

2 Motivation

Arm is heavily invested in the internet services and IoT lines of business. This is founded on a strong reputation for managing hardware/software security. As illustrated by the Healthcare PoC, data management and data security are fundamental requirements to enable the development of secure IoT applications. Developing advanced, Arm-centred solutions for managing the security of software services will provide Arm with a competitive advantage, or at the very least avoid being at a competitive disadvantage with respect to emerging x86-based cloud solutions.

3 Problem Statement

Modern software development makes use of technologies (such as web-frameworks) that were developed for cloud computing. Traditional monolithic applications are being replaced with micro-service style applications, which communicate using protocols such has REST over HTTP, gRPC, MQTT and ZeroMQ. Services are often containerized and managed through sophisticated orchestration infrastructure, using tools such as Docker and Kubernetes. Crucially, these services might be the gatekeepers for high value data assets, such as financial and medical records. The use of distributed service style architectures leads to complex and challenging security issues, especially in multi-tenant and multi-actor scenarios, where there will be interactions with communicating services that could be compromised and/or malicious. Furthermore, services can be internet facing, which means that they are potentially exposed to highly-sophisticated attacks from well-resourced parties.

3.1 Cloud-based Solutions Assumptions

A number of network security solutions exist within the cloud micro-services space. Importantly, most cloud-based applications make certain trust assumptions:

- Trusted infrastructure: In most cloud-based micro-services applications the execution platform is trusted. This is often achieved by having in-premises clouds or by relying on well-established cloud providers like Amazon AWS or Microsoft Azure.

- Cluster Orchestration: Typically the management of deployment and maintenance of cloud micro-services relies on orchestration software. Examples are Kubernetes **?** for containerized micro-services, or Amazon Elastic Container Service (ECS) **?**, Azure Kuberentes Service (AKS) **?**, Amazon CloudFormation **?** for infrastructure, etc. Among other issues these frameworks have to provide facilities for network communication and some of them support plugins for network security. Considering only the Kubernetes orchestration as a notable example we find a multitude of network Container Network Interface (CNI) plugins, including among others Cilium **?**, Calico **?**, Flannel **?**, etc.

- Network Security: As mentioned above, most of these network plugins provide some degree of security enforcement. These range from simple network connectivity (Layers 2 and 3 of the OSI stack), to application level APIs (Layer 7) such as the case of Cilium **?**.

3.2 Representative Cloud Network Security Solutions

Notable examples of cloud Network Security Solutions are are Cilium, Envoy and Istio.

Cilium: Cilium **?** focuses on high performance API/network-based security; however, it has a few shortcomings generally shared by all existing solutions: 1. The policies are relatively simple, with rules being limited to the selection of end-points (service entry-points) and the API method called called, as well as role-based access control. 2. It does not natively support popular publish-subscribe protocols such as MQTT. (For instance extensively used in the Healthcare PoC.) 3. The codebase for Cilium is large and complex, and there are many software dependencies. Currently it has not been made to work on Arm hardware. 4. Being a CNI plugin for Kuberenetes, it shares its control-plane/data-plane model, whereby a master controls an agent in each node, and it centralizes the configuration of security among other services.

Istio: This **?** is another interesting framework. Istio provides a mesh-like network infrastructure with emphasis on security. In particular Istio enforces the security of the underlying communication channels, and it provides implementations of secure naming, communications backed by strong cryptography, key and certificate management, etc. Unlike Cilium, Istio network security policies are limited to L2/L3. Istio is a mature and widely deployed tool, but much like Cilium, Istio is complex and it is not currently Arm compatible. Like the Kubernetes master in the case of Cilium, Istio's control-plane centralizes the configuration and deployment of policies, which are used to set up the proxies in the nodes of the mesh implementing the policy.

ENVOY: Envoy **?** is an open source edge and service (reverse)-proxy focused on information security. Importantly, both Cilium and Istio use Envoy to implement their security policies. Envoy can route HTTP methods, configure routing and enforce L2/L3 policies. However, Envoy is only a proxy, so there is no control-plane to manage a complete distributed system.

OTHER SOLUTIONS: There is a multitude of solutions in this space, but most of them have similar features and shortcomings. In particular, it is not clear that these solutions are well adapted to the case of edge/IoT applications, where the assumptions on trust on the infrastructure are not necessarily well established.

GOING BEYOND API FILTERS: In this project we will cherry-pick some key features of Cilium, Istio and other existing solutions, and at the same time support the use of *more expressive security policies*, allowing rules of the form *"based on what we have observed before, service A is currently allowed to make request R to service B"*.

Beyond the network connectivity and the assumptions of trust on the infrastructure, two important *features lacking* in all of the existing frameworks are the capabilities to

1. track *sessions* and application-level protocols, and
2. to *inspect the data* being protected (information flow audit, leak prevention) to the extent prescribed by the security policy.

In the absence of these features, the enforcement of data and session security has to be relegated to the application(s) developer(s). However this is problematic in the case of applications developed by different parties as illustrated by the security requirements of the health-care PoC. This project will provide the policy writer with the capability to address these issues at the policy level instead of relying on well behaved applications.

INCREMENTAL SECURITY: An important aspect of our proposed solution is that it should not interfere with the security measures put in place in the original application developer. Consider for instance the case where the application developer has implemented secure communication channels among her/his trusted micro-services. It is completely acceptable to have an ARMOUR policy that allows these channels to exist (for which neither data nor session tracking can be implemented by ARMOUR). However, the ARMOUR policy should be permissive enough to allow unrestricted communication between these endpoints, which would be presumably the case if the application developer takes part in the policy specification. On the other hand, if no such security has been implemented by the application developer, but it is required by the security policy, ARMOURs implementation has the capability of enforcing it to achieve the desired policy.

To reiterate, ARMOUR policies do not require that all communication between any two services be made visible to the implementation engine, but if any "encrypted" channels exist, they should not contradict the policy.

	Use-Case	Devs.	Apps.	Policy	App.	Infra.
1.	1 Tenant	1 Org.	1 App.	Centralized	Trusted	Trusted
2.	N Trusted tenants	N Orgs.	M Apps	Centralized	Trusted	Trusted
3.	N Untrusted tenants	N Orgs.	M Apps	Centralized	Untrusted	Trusted
4.	N Untrusted tenants	N Orgs.	M Apps	Decentralized	Untrusted	Untrusted

Table 1: Trust modes

3.3 Tailoring to Different Trust Modes

An important aspect of any security solution for multi-party distributed systems is the consideration of the trust model, as well as the assumptions that can be made of the infrastructure running the application. Here we refer by "trust" to the assumptions that components of an application can make about the intent of other components, that is, whether they are benign or malicious. We do never assume that code is free of bugs or security holes.

Because the capability of ARMOUR to enforce security policies will be restricted by the trust model of the underlying application and infrastructure, we aim to gradually and incrementally consider the use-cases presented in **??**:

1. In the fist case we consider an application written by a group of developers belonging to a single organization. We consider a distributed application for which communication security might not have been put in place by the application developer. In this case, a centralized network security policy would establish the security requirements. This is typically the case of applications running in cloud environments, and are typically the clients of mesh-like security solutions such as Istio **?** and Envoy **?**.

   Importantly, the security assumptions of this kind of application are that the application code is itself trusted by all the components (since there is a single organization), and that the infrastructure is also trusted (in premise, or a trusted provide such as AWS).

   For an application of this type, the use of ARMOUR would provide security to the communication channels, and it would validate that the application communication respects the security policy, for instance to prevent information leaks due to application errors, design inconsistencies, unsafe libraries potentially opening unauthorized communication channels, etc.

2. In the second case we find applications that incorporate components of different organizations or providers. This could for instance be a simplified case of the health-care PoC where all the containerized applications are assumed to trust each other. Importantly, in this case a centralized policy could validate or enforce that security measures are respected. This is particularly important because when combining software components from different providers it is not immediately obvious what the composed behavior of the application is, nor whether it respects high-level policies. For instance, can we easily know when the addition of a new pharmaceutical company to the healthcare PoC will respect the patient's security requirements? This is not trivial even in the case where the pharmaceutical application is trusted since it requires the consideration of the behavior of the whole application.

In this scenario the ARMOUR policy would act as a safety-net ensuring that the composition of different (perhaps dynamic) actors does not violate the high-level security objective.

The trust model for this scenario requires that all applications trust the enforcer of the security policy, and the policy acts as a security monitor to prevent errors stemming from the composition with unknown code.

3. The third use-case we consider best corresponds with the assumptions made by the healthcare PoC application, where we assume that Arm manages the compute infrastructure, and a number of different organizations run different components of the application. We take this case as a typical baseline for IoT-like systems where new devices or services are added on-demand from a set of not necessarily trusted providers.

   Thus, we can consider that while the application providers do not trust each other, they do trust the infrastructure. In that sense, having a centralized policy run by the infrastructure provider could achieve the desired security objectives for all the parties. This is typically the case of a cloud-provider, where the tenants do not necessarily trust each other, but they do trust the provider (isolation, networking, VPNs, VPCs, etc.). Hence, the security assumptions in this case are: the different applications do not trust each other, the applications do trust the infrastructure, and therefore if the ARMOUR security policy is managed by the infrastructure provider, the applications can rely on this security policy to achieve trust.

4. The final and most adversarial case is one where there is no trust among the applications, nor is there trust in the infrastructure provider. One can argue that the application developers do not need to trust the infrastructure provider with their data, and hence the application of proxies that inspect data and keep track of sessions is not applicable to this case. This is however not incompatible with the objectives of ARMOUR.

   For this kind of adversarial environment we will consider the implementation of a network/communication substrate supported by Trusted Execution Environments (TEEs) to provide end-to-end privacy preservation guarantees. For instance, we could consider TEE-backed proxies or filters such that the different participating parties can remotely attest their correctness (cf. remote attestation). Upon successful attestation the parties can trust that the only function of these proxies is to enforce the agreed upon security policy, without ever deviating from their "verified" expected behavior, and hence not revealing any "secrets". This is the most speculative thrust of the project, and it remains largely an open research problem that we would like to explore, where we believe we can make substantial contributions.

   Since in this case a central policy coordinator might not be achievable, we will consider that the security policies might have to be decentralized, potentially with each party providing its own, and that Trusted Execution Environments (TEEs) will have to be leveraged to ensure security of the individual services. Possible implementations of these TEEs include but are not limited to OS-isolation, SGX Enclaves or Newmore realms, etc.

4 Identity Management and Authentication

This section is motivated (in part) by the Zero Trust Networks (ZT) approach to network design and protection **?**, and it attempts to leverage notions of identity supported by Root of Trust (RoT) mechanisms advocated by Arm and the Platform Security Architecture (PSA) **?**.

ZT networks recommends that all notions of identity should be rooted in PKI certificates. Different entities to which certificates can be associated are: 1. Devices, 2. Users, and 3. Applications.

Device and application trust should be supported by an attestation procedure allowing to associate concrete hardware devices with logical identifiers in ARMOUR for the former case, and associate software measurements provided through attestation to logical application instance identifiers in ARMOUR for the latter case. The notion of a user is dependent on the infrastructure (Argus, AWS, GC, etc.), and it can also be application-specific. Because of that, it is perhaps best to handle user authentication by means of ARMOUR oracles, although this needs to be studied.

While these notions of identity are directly related to a physical/measurable entity, there will be other notions of identity which will be fundamentally logical. Examples of these are - Kubernetes Services, Roles, Pod names, - AWS security groups, AIM, - Unix user and group names, etc.

Another important aspect of authentication is that some logical entities are persistent (devices, users, etc.), while other entities are volatile (workloads, FaaS instances, temporary services, etc.). ARMOUR will requires that all logical notions of identity be backed by entities that have been authenticated using PKI certificates. To that end, ARMOUR will need to provide mechanisms to associate the identity of logical and/or volatile entities to the identity of physical or persistent entities that can be attested through PKI certificates. In some sense, this association propagates the RoT from the attestation mechanisms provided by hardware (eg. PSA) to the high-level identity primitives manipulated by ARMOUR.

4.1 From RoT to ARMOUR IDs

5 Identities and Authorization

Again, following the ZT **?** framework, it would be ideal to make the authorization and authentication management of an ARMOUR policy independent. This however does not mean that they can influence each other. In particular, it is expected that most authorization policies will need to refer to aspects of authentication, even if indirectly through the logical identities established.

GP: TODO

This that should be discussed somewhere. Needs design decisions.

- Through and through encryption, probably supported by mTLS/IPSec,
- Authentication (TODO) vs. authorization (current policy language),
- Do we need network agents a la ZT?,
- From RoT device + user identity to workflow? service? channel?,
- Proxy management, data/control plane API + security,

- Who signs what? Is control plane self-signing and it signs everything else?,
- Implementation of certificate authority a la Citadel?,
- Policy distribution? A la Kubernetes? Replication? Consistency?,
- TLS vs IPSec?, we might be able to chose.

6 Vision

When this project is completed, Arm will have access to prototype proxying middleware that runs on Arm hardware and provides advanced policy-based security features for distributed micro-service style applications.