xRAC: Execution and Access Control for Restricted Application Containers on Managed Hosts

Frederik Hauser, Mark Schmidt, and Michael Menth
Chair of Communication Networks, University of Tuebingen, Tuebingen, Germany
Email: {frederik.hauser,mark-thomas.schmidt,menth}@uni-tuebingen.de

***Abstract*—We propose xRAC to permit users to run special applications on managed hosts and to grant them access to protected network resources. We use restricted application containers (RACs) for that purpose. A RAC is a virtualization container with only a selected set of applications. Authentication verifies the RAC user's identity and the integrity of the RAC image. If the user is permitted to use the RAC on a managed host, launching the RAC is authorized and access to protected network resources may be given, e.g., to internal networks, servers, or the Internet. xRAC simplifies traffic control as the traffic of a RAC has a unique IPv6 address so that it can be easily identified in the network. The architecture of xRAC reuses standard technologies, protocols, and infrastructure. Those are the Docker virtualization platform and 802.1X including EAP-over-UDP and RADIUS. Thus, xRAC improves network security without modifying core parts of applications, hosts, and infrastructure. In this paper, we review the technological background of xRAC, explain its architecture, discuss selected use cases, and investigate on the performance. To demonstrate the feasibility of xRAC, we implement it based on standard components with only a few modifications. Finally, we validate xRAC through experiments. We publish the testbed setup guide and prototypical implementation on GitHub [1].**

I. Introduction

In this paper we consider the problem of permitting users to run special applications on managed hosts and to grant them access to protected network resources. This is an important challenge in practice as applications communicate with multiple peers and have multiple, possibly a priori unknown flows characterized by 5-tuples. Moreover, common packet filters such as firewalls or deep packet inspectors lack knowledge of allowed flows, and identifying traffic from specific applications becomes more difficult with traffic encryption using TLS.

We address this challenge by running applications in containers as so-called restricted application containers (RACs) on managed hosts. RACs provide selected sets of applications including their dependencies and configuration. The managed host gives users only limited freedom, e.g., they can download and launch RACs. We propose authentication and authorization (AA) for RACs so that their execution is restricted to authorized users. That means, the user identity, the integrity of the RAC's image, and the permission of the user to execute the RAC are verified before a RAC is launched. We suggest to extend this authorization also to protected network resources required by the RAC, e.g., to internal networks, servers, or Internet access. That means, appropriate network control elements, e.g., firewalls or SDN controllers, may be informed about authorized RACs and their needs. The authorized traffic can be identified by the RAC's IPv6 address. We call this concept xRAC as it controls execution and access for RACs.

We mention a few use cases that may benefit from xRAC. RACs may allow users to execute only up-to-date RAC images. Execution of a RAC may be allowed only to special users or user groups, e.g., to enforce license restrictions. Only selected users in a high-security area may get access to the Internet through a RAC-based browser which is isolated from the remaining infrastructure. Only selected users may be able to execute administration software with access to servers providing confidential material. RACs may be used for applications with increased quality of service (QoS) requirements, e.g., voice-over-IP, video conferences, or games. Their traffic may be preferentially treated by network elements.

To facilitate deployment of xRAC, we reuse and adapt standard technologies, protocols, and infrastructure. We leverage the Docker platform to create and deploy containerized applications. When the container management daemon (CMD) is requested to launch a RAC, it first issues an AA request and launches the RAC only after successful AA. We adopt 802.1X for AA purposes and adapt its components 802.1X supplicant (802.1X S), 802.1X authenticator (802.1X A), and 802.1X authentication server (802.1X AS). The CMD interfaces with an 802.1X container supplicant (802.1X CS) on the host, the 802.1X CS with a 802.1X container authenticator (802.1X CA), and the 802.1X CA with an 802.1X authentication server (802.1X AS). The 802.1X AS holds RAC-specific AA data, performs authentication, and returns authorization data to the 802.1X CA. The 802.1X CA interfaces with network control elements and configures access to network resources depending on authorization data. It also forwards the authorization data to the 802.1X CS. EAP-over-UDP and RADIUS are utilized for communication. To demonstrate the feasibility of xRAC, we provide a prototype based on existing 802.1X components, implement the 802.1X CS as plugin for the Docker authorization framework and the 802.1X CA as part of an SDN controller. We use a RADIUS server for 802.1X AS and extend its data structures to store AA data for users and RAC. We use this prototype to experimentally validate xRAC in a testbed.

This work was supported by the bwNET100G+ project which is funded by the Ministry of Science, Research and the Arts Baden-Württemberg (MWK). The authors alone are responsible for the content of this paper.

The rest of the paper is structured as follows. In Section II, we review technical background and related work on container virtualization. In Section III, we review technical background and related work on 802.1X and AA for applications. In Section IV, we present the architecture of xRAC in detail. Section V discusses use cases along benefits and limitations of xRAC. Section VI describes the prototypical implementation of xRAC which is used for its experimental validation in Section VII. In Section VIII, we briefly investigate on performance considerations of xRAC. Section IX concludes this work.

II. Container Virtualization: Technical Background and Related Work

We first introduce the concept of container virtualization, describe advantages and give an overview of Docker as a widespread implementation. Then, we review container security platforms and containers for GUI applications.

A. Overview

Containers implement virtualization on the operating system (OS) level. They provide virtualized OS environments that are isolated with regard to hardware resources and security. They share the OS kernel and may include binaries and libraries that are required to run one or several enclosed applications. In xRAC, we enclose only one special application with its dependencies and configuration in a container. Containers run on top of a container runtime and are managed by a container management daemon (CMD) that creates, starts and suspends containers. Examples for container platforms are Docker [2], Kubernetes [3], BSD jails [4], and Solaris containers [5].

B. Advantages

Virtualization facilitates efficient and flexible usage of hardware resources, improves security through isolation, and provides fault-safety and scalability through simple migration processes. Containers in particular have the following additional advantages. Due to the shared OS, containers require less CPU, memory, and hard disk resources. Container images are much smaller than virtual machines, which simplifies distribution among many recipients. Containers simplify application deployment. Instead of providing support for complex combinations of applications, dependencies, and user configurations, administrators just deploy containers that are tested prior to release. Containers have no bootup times, which makes them especially suitable for short-lived applications.

C. Docker Container Virtualization Platform

Docker [2] is one of the most popular container platforms today. Figure 1 provides a simplified overview of the Docker platform including its operations. A host with a common OS runs the Docker CMD that generates and manages containers and container images. Container images are write-protected templates that include applications with their dependencies. Containers are runtime instances that extend the write-protected container images by a writeable layer. Therefore, multiple container instances may share a common image. This introduces scalability with low hard disk requirements. The Docker CMD is controlled by the Docker client via a REST interface. The Docker client can be located on the host running the Docker CMD or on a remote host. The Docker command line interface (CLI) is an example for user control through CLI calls. The Docker CMD may connect to Docker registries that allow users to upload (push) or download (pull) container images. Those registries are either private or publicly available. Docker Hub [6] with more than 100.000 container images is an example for the latter. Common operations are build (1), pull (2), and run (3). With build, users may create individual container images. With pull, users may download container images from a repository to become part of the set of local images. With run, container images from the set of local images can be executed on the host system.

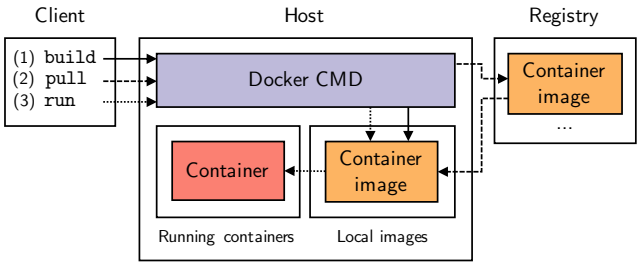


Fig. 1: The Docker architecture consists of the Docker client, the Docker CMD, and the Docker registry [7].

Docker uses several functions of the OS kernel [8]–[10] to provide isolation and resource emulation. It supports storage drivers, e.g., AuFS [11], OverlayFS [12], and ZFS [13], to enable file system stacking. The container format and runtime environment of Docker were adopted by the Open Container Initiative [14] as open industry standard.

D. Container Security Platforms

Container security platforms extend the CMD by security functions. Twistlock [15], [16] and the Aqua Container Security Platform [17] provide a runtime engine based on machine learning mechanisms to permanently monitor containers for detecting fraudulent behavior and special network firewalls to filter container traffic. The Sysdig Secure [18] platform allows the formulation of service-aware policies, i.e., policies that are based on applications, containers, hosts, or network activities. The platform provides alerts and actions based on policy violations, an event history, and incident captures. The Atomicorp Secure Docker Kernel [19] is a hardened Linux kernel that provides security-related features such as breakout protection, memory corruption protection, or prevention of direct userland access by the kernel. All platforms focus on monitoring and controlling potentially untrustworthy containers that are executed on a shared container runtime. Features for AA for users, containers, and their network flows are not part of those platforms.

The Docker Authorization Framework [20] is part of Docker since Version 1.10 [21]. It extends the Docker CMD by a REST interface to external authorization plugins. Requests

from the Docker CMD, e.g., to start a container, are forwarded to an authorization plugin that implements mechanisms to decide whether to allow or deny the request. The Docker Authorization Framework does not implement security functions but provides a base for implementing such security concepts. xRAC leverages this framework.

E. Containers for GUI Applications

Containers typically deploy applications or services without graphical user interfaces (GUIs) that run in the cloud or on data center infrastructures. Examples are containers that enclose web applications with their requirements, e.g., an nginx web server with a PHP runtime and a MySQL database server. The idea of leveraging Docker containers to deploy desktop applications with a GUI was first presented in [22]. The author proposes to mount X11 sockets for GUI presentation and hardware devices of the host system, e.g., an audio card or a web camera, to the Docker container. Thereby, even more complex GUI applications such as the Chrome web browser, the Spotify music player, or the Skype video chat application can be run in containers and be deployed as container images. Today, many Docker container images with GUI applications can be downloaded from Docker Hub.

III. 802.1X: Technical Background and Related Work

We give an overview of 802.1X and explain how it supports AA. We present EAPoUDP which is an alternative protocol to carry AA data in 802.1X. We summarize how AA for applications is currently performed in practice and review another research work that provides AA for applications based on 802.1X.

A. Overview

IEEE 802.1X [23]–[25] introduces port-based network access control in wired Ethernet networks. However, it is mainly known from wireless 802.11 networks today. An example is Eduroam [26], a federation of wireless university campus networks. Participants can connect to the Internet no matter if located at their home institution or at a foreign university campus, e.g., while attending a conference.

Figure 2 depicts the three components of 802.1X and the principle of port-based network access control. The supplicant system is a network host that contains the 802.1X supplicant (802.1X S). The authenticator system contains the 802.1X authenticator (802.1X A) and controls network access of network hosts. Examples are access switches that connect network hosts to the main network. Prior to authorization, the supplicant system can reach the 802.1X A but not the network. The 802.1X AS is an authentication, authorization, and accounting server. It stores authentication data to verify user identities and authorization data to grant permission to the network. It authenticates the 802.1X S and delivers authorization information to the 802.1X A.

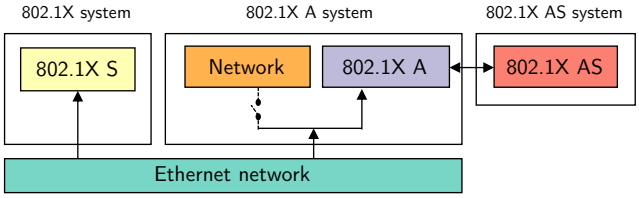


Fig. 2: Port-based authorization model of 802.1X [24].

B. AA with 802.1X

802.1X leverages the Extensible Authentication Protocol (EAP) [27] and the Remote Authentication Dial In User Service (RADIUS) [28] to exchange AA data. Both provide a fixed request and response scheme to exchange AA data. The Diameter protocol [29] is a less widespread alternative. Authentication data is transmitted in Ethernet frames as EAP-over-LAN (EAPoL) encapsulation between the 802.1X S and 802.1X A and as EAP-over-RADIUS (EAPoRADIUS) between the 802.1X A and 802.1X AS. Figure 4 depicts the packet structure of EAPoL. Authorization data is transmitted in RADIUS frames between the 802.1X AS and 802.1X A.

We explain the details of 802.1X with the four-step process of AA as depicted in Figure 3. In the first step (1), the 802.1X S initializes authentication by sending an EAPOL-Start message to the 802.1X A. In the second step, the 802.1X A requests the identity from the 802.1X S (2a) and forwards it to the 802.1X AS (2b). RADIUS supports large domains that consist of many hierarchically organized RADIUS servers. Each identity is associated with a domain and known by the RADIUS server of that domain so that AA attempts can be forwarded in RADIUS infrastructures. In the third step (3), authentication is performed between the 802.1X S and 802.1X AS. The authenticator decapsulates EAP packets from EAPoL frames and reencapsulates them as EAPoRADIUS frames and vice versa. The flexible message structure of EAP allows the use of different authentication procedures. Simple approaches carry plain-text identity information or simple MD5-hashed passwords, but more secure authentication procedures like EAP Tunneled TLS [30] and EAP-TLS [31] are also supported. The authenticator only relays EAP messages in pass-through manner. Therefore, new EAP types only need to be implemented on the 802.1X S and 802.1X AS but not on the 802.1X A. In the fourth step, the RADIUS server may return authorization data after successful authentication to the 802.1X A (4a). It can be coarse-granular, e.g., a binary access decision whether the supplicant system gets access or no access, or fine-granular, e.g., VLAN tags [32] to be set for prospective user traffic or filter rules [33] that are applied by the authenticator. The authenticator applies the authorization data on the particular physical port of the switch, e.g., it sets a VLAN tag. Afterwards, the authenticator confirms successful AA to the supplicant with an EAP-Success message (4b).

C. EAP-over-UDP (EAPoUDP)

EAPoUDP is a variation of EAP that allows transmission of EAP data over UDP and IP. Figure 4 depicts its packet

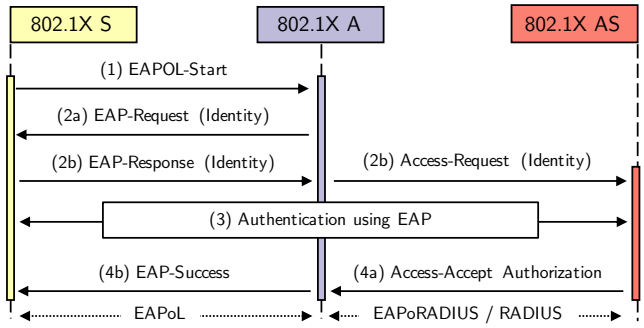


Fig. 3: Communication example of 802.1X based AA.

structure in comparison to EAPoL. In contrast to EAPoL, EAPoUDP can be used to authenticate multiple applications that run on a network host. Also, UDP packets can be transmitted over any link layer technology or even routed within multi-domain networks. EAPoUDP was introduced as Internet draft [34] that expired without standardization in the PANA working group of the IETF in 2002. Cisco leveraged EAPoUDP in its Trust Agent [35] tool that runs on network hosts and interacts with Cisco NAD, a prorietary network control system. The Trust Agent collects host system information, interfaces host software, and delivers notifications to network hosts within EAPoUDP frames. xRAC leverages EAPoUDP for frontend authentication.

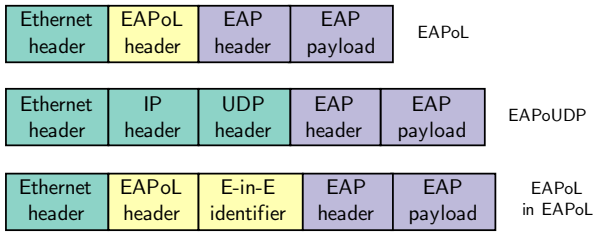


Fig. 4: EAPoUDP (a) in comparison to EAPoL (b). EAPoUDP uses UDP as transport protocol, EAPoL leverage Ethernet frames along an EAPoL header.

D. AA for Applications in Practice

802.1X focuses on port-based access control for network hosts. In practice, AA for applications is implemented as part of the application or with the help of Kerberos.

Most network applications implement some form of AA mechanism. Examples are login forms in the launch window where users are required to enter valid credentials to start using the application. Other examples are client certificates that are used in conjunction with TLS and a public key infrastructure. However, AA functionalities that are part of the application have an impact that is limited to the client and server side of the network application. Neither the start of the application nor the network infrastructure in between can be controlled.

Kerberos [36], [37] is a network authentication protocol that provides mutual authentication for clients and servers over an insecure network. Clients are entire hosts, users, or applications; servers represent hosts that offer particular network applications. Kerberos adapts user tickets for authentication for various network applications. Kerberos needs to be implemented by applications on client and server side, which prevents its application for legacy applications that cannot be modified. Again, neither the start of kerberized applications nor the network infrastructure in between can be controlled.

E. AA for Applications with FlowNAC

FlowNAC [38] introduces a fine-granular SDN network access control system that adapts 802.1X for AA of applications on network hosts. To enable multiple AA for different applications on a network host, the authors introduce EAPoL-over-EAPoLAN encapsulations. As depicted in Figure 4, FlowNAC introduces another variation of EAPoL. An EAPoL-in-EAPoL packet field identifies up to 64K different EAP processes that are transmitted as encapsulated EAP payloads. However, this deviation from legacy 802.1X requires major changes of the 802.1X S and 802.1X A. The 802.1X S is part of an OS's kernel, the 802.1X A is part of network switches so only open source OSs and firmwares allow modifications. Nevertheless, it is difficult to carry on the modifications in new versions of the OS's kernel or firmware image. The authors rely on EAPoL, i.e., AA data transfer is limited to the Ethernet link. EAPoUDP would solve those shortcomings but was not considered in the work. Unlike xRAC, FlowNAC neither introduces IP addresses for applications nor restricts the start of applications by AA.

IV. xRAC Architecture

In this section, we first explain RACs and give an overview of xRAC. Then, we explain the operation of its three control components in details.

A. Restricted Application Containers (RACs)

RACs are executable container images that enclose a single application, its dependencies, and configuration data such as program settings or software licensing information. As depicted in Figure 5, RACs are executed on a container runtime in parallel to OS-native applications. The CMD controls the execution of RACs and provides an interface for users to create, delete, start, or stop RACs. Each RAC has a unique IPv6 address so that its traffic can be easily identified in the network. RAC images are created by network administrators and deployed via RAC registries. They are either downloaded from those registries by users or automatically synchronized to managed hosts.

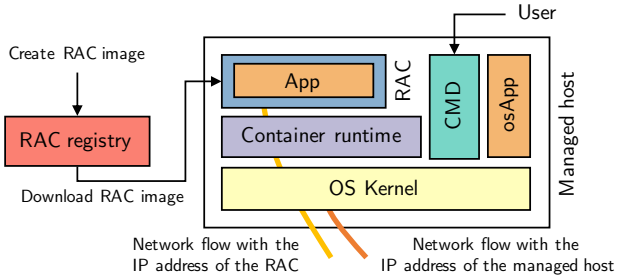


Fig. 5: The managed host executes a RAC and a OS application (osApp).

B. Overview of xRAC

xRAC provides execution and access control for RACs on managed hosts. A RAC needs to be authenticated and authorized before launching. Figure 6 depicts the AA process for RACs with 802.1X. First, a user attempts to start a RAC via the CMD (1), and the CMD requests the 802.1X CS for AA (2). After successful authentication (3), the 802.1X AS responds with authorization data via the 802.1X CA (4) to the 802.1X CS (4a). The 802.1X CS notifies the CMD to launch the RAC (4b). In addition, the 802.1X CA informs network control elements about the authorized RAC. In our example, it configures the firewall to permit access to the proteced server (4c). Other examples are SDN controllers that program SDN switches. Now, the authorized RAC but not the managed host or other RACs can communicate with the protected server (5).

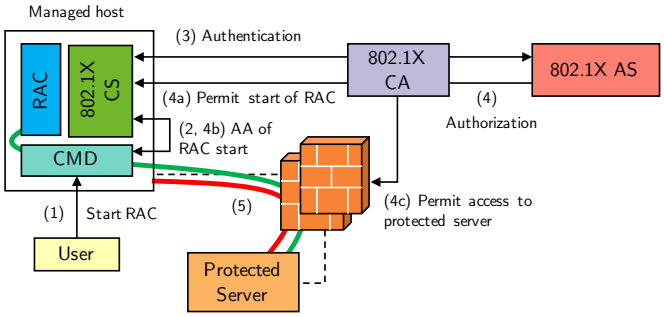


Fig. 6: Adaption of 802.1X for AA of RACs. The 802.1X CS authenticates the RAC using the 802.1X AS. The 802.1X CA receives authorization data and forwards them to the 802.1X CS and to a firewall that secures a protected server from unauthorized traffic.

AA for RACs introduces two additional advantages to common application deployment and network security. First, AA of RACs restricts RAC launches on managed hosts to predefined RAC images and permitted users. This allows network operators to ensure that only up-to-date and unmodified RAC images can be launched. This improves computer and network security as only valid RAC images can be executed on the managed hosts. In addition, network operators may deploy RAC images to managed hosts in advance, e.g., by synchronizing their set of RAC images with an internal RAC repository in the background. Users have all available RAC images on managed hosts but are only able to start them if they become authorized after AA. Last, each RAC has a globally unique IPv6 address that can be used to identify and steer all traffic originating from a particular RAC. RAC authorization data on the 802.1X AS includes information on how the RAC's traffic should be steered by network elements that can be applied by network control elements.

C. 802.1X Authentication Server (802.1X AS)

The authentication request from the 802.1X CS to the 802.1X AS contains user and container authentication data (UAND, CAND). The 802.1X AS authenticates the user and verifies the integrity of the RAC image. If the RAC image is valid, and if the user is authenticated, and if the user has permissions to run the RAC, the 802.1X AS responds to the 802.1X CA with authorization data. To perform that decision, the 802.1X AS requires a new data model which is depicted in Figure 7. It consists of user profiles, RAC profiles, and groups that define whether a particular user is permitted to run a particular RAC. User profiles contain user authentication data (UAND) that is used to authenticate the user. Examples are user credentials, e.g., user names and passwords. RAC profiles contain container authentication data (CAND) and container authorization data (CAZD). The first is used to verify the integrity of the RAC through calculating the cryptographic hashing function over the RAC image. CAZD include all permissions of a RAC, i.e., to be started by the requesting user and to utilize network resources. In the depicted example, the RAC is allowed to access a specified network resource. The AA data for the described model is stored on the 802.1X AS. The data model is an example that can be easily extended to support other requirements.

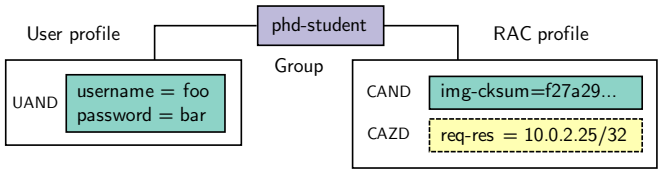


Fig. 7: The AA data model for RACs consists of user profiles, RAC profiles, and group mappings. User profiles include UAND, RAC profiles contain CAND for authentication and CAZD for authorization.

D. 802.1X Container Supplicant (802.1X CS)

The 802.1X CS authenticates RACs with the 802.1X AS via the 802.1X CA. It transmits UAND and CAND to the 802.1X AS and receives CAZD from the 802.1X CA.

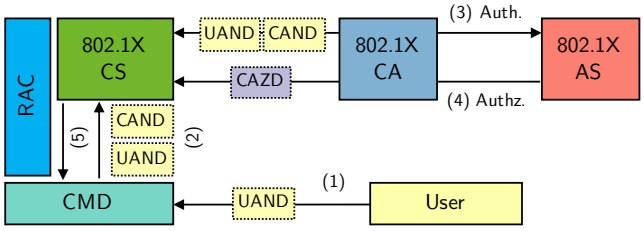
Figure 8a illustrates the process of AA from the perspective of the 802.1X CS. It runs on the managed host, interfaces the CMD, and is configured with the IP address or URL of the 802.1X CA so that it can initiate AA. In (1), the user requests the CMD to start a particular RAC on the managed host. The request includes UAND. The CMD requests the 802.1X CS to permit the user's demand (2). The request includes UAND and CAND collected by the CMD. The 802.1X CS initiates AA by establishing an EAPoUDP session with the 802.1X CA. Afterwards, it performs authentication with the 802.1X AS via the 802.1X CA (3). Backend authentication is performed via EAPoRADIUS while frontend authenticated is performed via EAPoUDP as in legacy 802.1X. In case of successful authentication, the 802.1X CS receives CAZD from the 802.1X CA (4). Then, the 802.1X CS permits the CMD to start the RAC (5).

E. 802.1X Container Authenticator (802.1X CA)

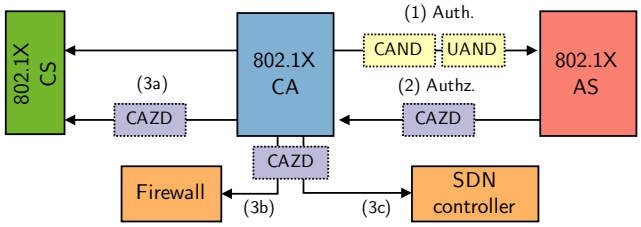
The 802.1X CA relays AA data between the 802.1X CS and the 802.1X AS. Moreover, it informs network control elements about authorized RACs.

In step (1) of Figure 8b, authentication data are transported over EAP between 802.1X CS and 802.1X AS. Between

802.1X CS and 802.1X CA, the EAP data are carried over UDP (EAPoUDP) and between 802.1X CA and 802.1X AS, they are carried over RADIUS. Thus, one task of the 802.1X CA is to modify the tunnel for EAP data. Moreover, after successful authentication the 802.1X AS returns CAZD over RADIUS to the 802.1X CA (2) which then informs the 802.1X CS about successful authorization (3a). While the conventional 802.1X A just opens ports on a switch for authorized devices, the 802.1X CA may also inform other network control elements about authorized RACs. Those may be ports on a switch, firewalls (3b), or SDN controllers (3c). The firewall is then programmed to pass through all outbound traffic with the RAC's IP address and the SDN controller instructs SDN switches to forward all traffic with the RAC's IP address appropriately. More specific flow descriptors are not needed.



(a) 802.1X CS receives UAND and CAND from the CMD. It then performs frontend authentication via EAPoUDP and backend authentication via EAPoRADIUS with the 802.1X AS. In case of successful authentication, the 802.1X CA and the 802.1X CS receive CAZD.



(b) The 802.1X CA performs frontend authentication with the 802.1X CS and backend authentication with the 802.1X AS. In case of successful authentication, it receives CAZD from the 802.1X AS that is forwarded to the 802.1X CS and other network components.

Fig. 8: AA from the perspective of the 802.1X CS and the 802.1X CA.

V. Use Cases and Discussion

We discuss two exemplary use cases of xRAC and discuss its benefits and limitations.

A. Use Case I: Web Browsers in High-Security Areas

Research departments, state departments, or clinics dealing with highly sensitive data isolate their internal networks from the Internet. However, web browsers are still required for online research activities. We propose to deploy web browsers as RACs on managed hosts. The isolation of RACs prevents that malicious users may misuse the Internet access to leak internals or contaminate the internal network through infectious downloads, e.g., PDF documents that include a trojan or virus. The network flow control of RACs ensures, that only the web browser can reach the Internet. If the RAC's traffic is encrypted, e.g., DNS queries and web site data, network control elements can still perform packet filtering based on the IP addresses of the RAC.

B. Use Case II: Confidential Data Access

Applications dealing with confidential data, e.g., research activities, medical documentation, or law enforcement, often access servers with confidential data. If such applications are deployed as RACs on managed hosts, only legitimate users have access to those servers. The isolation feature of RACs prevents remote hackers from attacking the server. Normally, they get system access through gateways provided by viruses or trojans received as browser downloads or e-mail attachments which is not possible with RACs. Furthermore, malicious users of legitimate applications may use hacker tools to gain access to the server and to leak information from it, which is not possible in the digital domain with an isolated application. The isolation of RACs prevent malicious users or applications from attacking the server. The network flow control ensures that the server can be reached only by legitimate RACs and users but not by other RACs or the managed host itself.

C. Benefits of xRAC

xRAC inherits the advantages generally known from virtualization and container virtualization that we have discussed in Section II-B. In addition, xRAC can guarantee that only valid containers are launched on managed hosts and that they can be used only by legitimate users. Thus, xRAC performs AA for applications without the need to modify them, which is a particular benefit for legacy applications. Moreover, the 802.1X CA can configure network control elements such that authorized RACs have access to protected network resources. RACs facilitate this control as all traffic of a RAC is identified by a single IPv6 address. This is a particular benefit as in today's networks there is no information about legitimate flows, many application flows may have the same IP address, and applications may even be invisible due to encryption using TLS. Thus, steering traffic from legitimate or trusted applications is a tough problem for which xRAC provides a solution. xRAC is flexible as it implements software-defined network control by interacting with other network control elements. In particular, it does not depend on and is not limited to specific technologies.

D. Limitations of xRAC

xRAC requires a managed infrastructure where the managed host, its CMD, and the 802.1X components, i.e., 802.1X CS, 802.1X CA, and 802.1X AS, are trusted. Encapsulating applications in RACs complicates access to shared resources so that xRAC may be cumbersome or infeasible for some use cases.

VI. Prototypical Implementation

We describe a prototypical implementation of xRAC and publish its source code with a testbed setup guide on GitHub [1]. In the following, we give an overview on the testbed environment and describe all components in detail.

A. Testbed Environment

Figure 9 depicts the testbed environment. The managed host executes RACs. The SDN switch connects the managed host, the protected server, and the public server and is controlled by an SDN controller. The SDN controller runs the 802.1X CA as SDN application that communicates with an 802.1X AS.

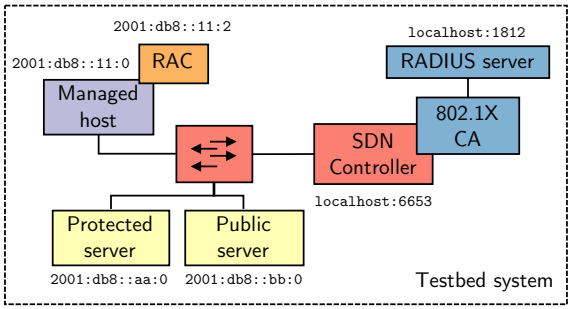


Fig. 9: Testbed environment.

We execute the testbed on a ThinkPad T460s with an i5-6200U CPU, 20GB RAM, SSD, and running Ubuntu 18.04.3 LTS. The managed host and both servers are Virtual Box virtual machines (VMs), each running Ubuntu 18.04.3 LTS. Open vSwitch [39] serves as SDN switch that is controlled by the Ryu SDN controller [40] (v4.34). The FreeRADIUS 802.1X AS (v3.0.16) is executed directly on the testbed host.

B. Docker as Container Virtualization Platform for RACs

We use Docker [2] (v19.03.5) as container virtualization platform to implement RACs. We configure the Docker CMD so that each RAC gets a dedicated IPv6 global unicast address that is reachable by other network hosts. Figure 10 depicts the applied networking configuration that follows the approach presented in [41]. By default, RACs only receive a link-local IPv6 address. Therefore, we set up a fixed IPv6 subnet with routable addresses for RACs. The managed host is configured with the IPv6 subnet `2001:db8::11:0/116` and the RACs receive an IPv6 address from that range. The first IPv6 address is reserved for the `docker0` interface. Therefore, the first RAC receives `2001:db8::11:2` and the second RAC `2001:db8::11:3`, respectively. The Docker daemon automatically adds routes to the routing table of the system and enables IPv6 forwarding so that all traffic to the IPv6 subnet will be routed via the `docker0` interface. To make the RACs reachable from other network hosts, we leverage the NDP proxy daemon [42]. It forwards L2 address resolution for IPv6 addresses of the RACs, i.e., it listens to neighbor solicitation requests for the RACs addresses and answers with the MAC address of the managed host. Afterwards, packets that address a RAC are received and forwarded through the Docker host via the `docker0` device to the particular RAC.

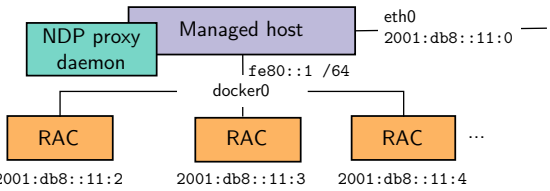


Fig. 10: Network configuration of Docker in the testbed environment. Each RAC gets an IPv6 address of the IPv6 subnet that is assigned to the managed host. The NDP proxy daemon resolves L2 addresses for the RACs.

C. 802.1X Container Supplicant (802.1X CS)

We implement the 802.1X CS as plugin for the Docker Authorization Framework introduced in Section II-D. We program the plugin in Python and leverage the Flask [43] library to implement its REST interface. Figure 11 depicts the authorization process. In (1), the user requests the CMD to start a container. The request includes UAND, e.g., a user name and a password. The Docker Authorization Framework predefines a two-step authorization process, but we only require the second step. The first authorization request (2) includes only minimal data, e.g., the name of the RAC image. As we solely rely on the second authorization step, the 802.1X CS responds with a permit by default. The second authorization request (3) includes UAND and CAND. The 802.1X CS performs authentication with the 802.1X AS through the 802.1X CA (3) as discussed before. In (4), 802.1X AS returns CAZD that is forwarded to the 802.1X CS in case of successful AA.

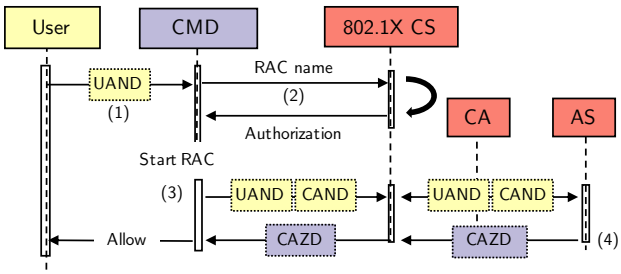


Fig. 11: Two-step authorization process in the Docker Authorization Framework [20]. The CMD requests the 802.1X CS to perform AA.

D. 802.1X Container Authenticator (802.1X CA)

We implement the 802.1X CA as SDN application for the Ryu SDN controller framework [40]. We extend the 802.1X A of [44] by adding support for authentication with the 802.1X CS using EAPoUDP. The 802.1X CA opens a UDP socket on port 5995 and waits for connections from the 802.1X CS. The 802.1X CA still may act as legacy 802.1X A. As example for network control with xRAC, we implemented a restricted MAC-learning switch. It learns MAC addresses from connected hosts but only forwards packets if the IP addresses of both sender and receiver are in a whitelist. The whitelist contains static entries, e.g., for public servers, and dynamic entries that can be modified by the 802.1X CA after receiving CAZD from the 802.1X AS. We implement the restricted MAC-learning switch by extending the L2 switch [45] from the Ryu SDN controller framework.

E. 802.1X Authentication Server (802.1X AS)

We leverage the widely-used 802.1X AS software FreeRADIUS and extend its AA data model to implement CAND and CAZD. In FreeRADIUS, additional attributes for AA can be implemented using vendor-specific attributes (VSAs) [28], [46]–[48]. Simple policies are defined with the unlang [49] processing language. The defined AA data model can be easily extended and modified by adding more VSAs.

F. Protected and Public Server

We run a public server with the static IPv6 address `2001:db8::aa:0` that is accessible without authorization. As example for a protected network host, we run a protected server with the static IPv6 address `2001:db8::bb:0`.

VII. Experimental Validation

We describe the experiment setup and validation experiments for the testbed from Section VI to validate xRAC.

A. Experiment Setup

The experiments investigate the communication between the managed host, a particular RAC, the protected server, and the public server. We use the latest Busybox [50] image as RAC. In our experiments, we use the RAC to send ICMP echo request packets to both, the protected and the public server. We add UAND, CAND and CAZD on the 802.1X AS that allows a particular user to run the RAC and access the protected server.

B. Validation Experiments

We perform the following experiments as depicted in Figure 12. Before launching the RAC, we validate with ICMP echo requests from the managed host that the public server (1a) but not the protected server (1b) is accessible without authorization. Now, we demonstrate that the integrity of RACs is verified during authentication, i.e., that a RAC with a divergent image checksum cannot be started. We start an Alpine Linux [51] container image as RAC and try to start it using the user credentials as set up on the RADIUS server. Authentication fails, i.e., the RAC cannot be started on the managed host. Now, we demonstrate that the correct RAC can be started and that it can access the protected server after successful AA. After issuing the command to start the RAC, it is authenticated and authorized as described before (2a). The SDN controller receives CAZD and programs the SDN switch to permit packet forwarding between the RAC and the protected server (2c). Now, the RAC is able to exchange ICMP packets with the protected server (2d). Trying to exchange ICMP packets directly from the managed host fails (2e), i.e., the protected server can be reached by the RAC but not by the managed host.

VIII. Performance Considerations

Container virtualization has no remarkable performance overhead compared to native application execution [52]. xRAC adds delay only to the startup time through the CMD. The CMD requires time to calculate the container's integrity checksum and to perform AA using network-based 802.1X. Without

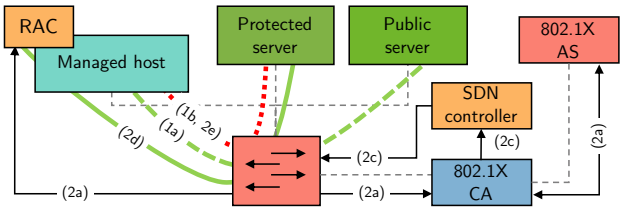


Fig. 12: Experiments to investigate the communication between the managed host, a RAC, and both servers.

xRAC, starting the wget container from Section VII-A takes approximately 0.6 s. With xRAC, the startup time is increased to 1.51 s. The Busybox container image is 6 MB large and the calculation of its SHA256 hash takes 0.12 s on our platform. However, the computation time scales with the image size. For example, a Chrome browser container image with 880 MB takes 3.2 s. The duration of the AA operation depends on the performance of the three 802.1X components and the network in between. The container supplicant is part of the managed host and, therefore, covers only little load. The container authenticator is part of the controller and responsible for many hosts. However, its performance can be scaled up by running multiple instances. The 802.1X AS may be based on RADIUS. This technology is proven to scale well with large deployments and high load by replicating server instances.

IX. Conclusion

In this work we proposed xRAC, a concept for execution and access control for restricted application containers (RACs) on managed clients. It includes authentication and authorization (AA) for RACs such that only up-to-date RAC images can be executed by permitted users. Moreover, authorization is extended to protected network resources such that authorized RACs can access them. Traffic control is simplified through the fact that all traffic of a RAC is identified by its IPv6 address. We presented the architecture of xRAC and showed by a prototype implementation that xRAC can be built from standard technologies, protocols, and infrastructure. Our prototype of xRAC leverages Docker as container virtualization platform, signalling is based on 802.1X components. Modifications were needed to the supplicant, the authenticator, and the authentication server so that both user and container AA data can be exchanged. Moreover, the container authenticator is extended to inform required network control elements about authorized RACs. We used the prototype to experimentally validate xRAC and investigate on the performance. After all, we discussed use cases and showed that xRAC supports software-defined network control and improves network security without modifying core parts of applications, hosts, and infrastructure.

Acknowledgment

The authors thank Julian Rilli for fruitful discussions and programming contributions.

REFERENCES

[1] "xRAC Repository on GitHub," https://github.com/uni-tue-kn/xrac.
[2] "Docker," https://www.docker.com/, accessed 01-21-2020.
[3] "Kubernetes," https://kubernetes.io/, accessed 01-21-2020.
[4] "FreeBSD: jails," https://www.freebsd.org/doc/handbook/jails.html, accessed 01-21-2020.
[5] "Oracle Solaris Containers," https://www.oracle.com/technetwork/server-storage/solaris/containers-169727.html, accessed 01-21-2020.
[6] "Docker Hub," https://hub.docker.com/, accessed 01-21-2020.
[7] "Docker Docs," https://docs.docker.com, accessed 01-21-2020.
[8] "cgroups," http://man7.org/linux/man-pages/man7/cgroups.7.html, accessed 01-21-2020.
[9] "LWN.net: PID namespaces in the 2.6.24 kernel," http://lwn.net/Articles/259217/, accessed 01-21-2020.
[10] "LWN.net: Network namespaces," http://lwn.net/Articles/219794/, accessed 01-21-2020.
[11] "AuFS4," http://aufs.sourceforge.net/, accessed 01-21-2020.
[12] "OverlayFS," https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt, accessed 01-21-2020.
[13] "OpenZFS," http://www.open-zfs.org, accessed 01-21-2020.
[14] "Open Container Initiative," https://www.opencontainers.org/, accessed 01-21-2020.
[15] "Twistlock," https://www.twistlock.com/, accessed 01-21-2020.
[16] "Twistlock 18.11 Data Sheet," https://www.twistlock.com/twistlock-data-sheet/, accessed 01-21-20.
[17] "Aqua Container Security Platform," https://www.aquasec.com/products/aqua-container-security-platform/, accessed 06-03-2018.
[18] "Sysdig Secure," https://www.sysdig.com, accessed 06-03-2018.
[19] "Atomicorp Atomic Secured Docker," https://atomicorp.com/features-atomic-secured-docker-kernel/, accessed 06-03-2018.
[20] "Docker Access Authorization Plugin," https://docs.docker.com/engine/extend/plugins_authorization/, accessed 01-21-2020.
[21] "Docker AuthZ Plugins: Twistlock's Contribution to Docker Security," https://www.twistlock.com/2016/02/18/docker-authz-plugins/, accessed 01-21-2020.
[22] "Jessie Frazelle: Docker Containers on the Desktop," https://blog.jessfraz.com/post/docker-containers-on-the-desktop/, accessed 01-21-2020.
[23] IEEE, "802.1X-2001 IEEE Standard for Local and Metropolitan Area Networks - Port-Based Network Access Control," 2001.
[24] ——, "802.1X-2004 IEEE Standard for Local and Metropolitan Area Networks - Port-Based Network Access Control," 2004.
[25] ——, "802.1X-2010 IEEE Standard for Local and Metropolitan Area Networks - Port-Based Network Access Control," 2010.
[26] "Eduroam - About," https://www.eduroam.org/index.php?p=about, accessed 07-19-2018.
[27] B. Aboba, L. Blunk, J. Vollbrecht, J. Carlson, and H. Levkowetz, "Extensible Authentication Protocol (EAP)," RFC 3748 (Proposed Standard), 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3748.txt
[28] C. Rigney, S. Willens, A. Rubens, and W. Simpson, "Remote Authentication Dial In User Service (RADIUS)," RFC 2865 (Draft Standard), jun 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2865.txt
[29] V. Fajardo (Ed.), J. Arkko, J. Loughney, and G. Zorn (Ed.), "Diameter Base Protocol," RFC 6733 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–152, Oct. 2012, updated by RFC 7075. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6733.txt
[30] P. Funk and S. Blake-Wilson, "Extensible Authentication Protocol Tunneled Transport Layer Security Authenticated Protocol Version 0 (EAP-TTLSv0)," RFC 5281 (Informational), RFC Editor, Fremont, CA, USA, pp. 1–51, Aug. 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5281.txt
[31] D. Simon, B. Aboba, and R. Hurst, "The EAP-TLS Authentication Protocol," RFC 5216 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–34, Mar. 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5216.txt
[32] P. Congdon, M. Sanchez, and B. Aboba, "RADIUS Attributes for Virtual LAN and Priority Support," RFC 4675 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–15, Sep. 2006. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4675.txt
[33] ——, "RADIUS Filter Rule Attribute," RFC 4849 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–9, Apr. 2007. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4849.txt
[34] P. Engelstad, "EAP over UDP (EAPoUDP)," Internet Engineering Task Force, Internet-Draft draft-engelstad-pana-eap-over-udp-00, Feb. 2002, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-engelstad-pana-eap-over-udp-00
[35] "Cisco Trust Agent," https://community.cisco.com/t5/security-documents/cisco-trust-agent/ta-p/3113750, accessed 01-21-2020.
[36] "Kerberos: The Network Authentication Protocol," http://web.mit.edu/kerberos/, accessed 01-21-2020.
[37] C. Neuman, T. Yu, S. Hartman, and K. Raeburn, "The Kerberos Network Authentication Service (V5)," RFC 4120 (Proposed Standard), Internet Engineering Task Force, Jul. 2005.
[38] J. Matias, J. Garay, A. Mendiola, N. Toledo, and E. Jacob, "FlowNAC: Flow-based network access control," in *Third European Workshop on Software Defined Networks*, 2014, p. 6.
[39] "Open vSwitch," https://www.openvswitch.org/, accessed 01-21-2020.
[40] "Ryu SDN Framework," http://osrg.github.io/ryu/, accessed 01-21-2020.
[41] "Docker Docs: IPv6 with Docker," https://docs.docker.com/v17.09/engine/userguide/networking/default_network/ipv6/, accessed 01-21-2020.
[42] "NDP Proxy Daemon," https://github.com/DanielAdolfsson/ndppd, accessed 01-21-2020.
[43] "Flask," https://palletsprojects.com/p/flask/, accessed 01-21-2020.
[44] F. Hauser, M. Schmidt, and M. Menth, "Establishing a Session Database for SDN using 802.1X and Multiple Authentication Resources," in *2017 IEEE International Conference on Communications (ICC)*, May 2017, pp. 1–7.
[45] "Examplary MAC-learning Switch for OpenFlow 1.3," https://github.com/osrg/ryu/blob/master/ryu/app/example_switch_13.py, accessed 07-19-2018.
[46] A. DeKok (Ed.) and G. Weber, "RADIUS Design Guidelines," RFC 6158 (Best Current Practice), RFC Editor, Fremont, CA, USA, pp. 1–38, Mar. 2011, updated by RFCs 6929, 8044. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6158.txt
[47] A. DeKok and A. Lior, "Remote Authentication Dial In User Service (RADIUS) Protocol Extensions," RFC 6929 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–68, Apr. 2013. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6929.txt
[48] A. DeKok, "Data Types in RADIUS," RFC 8044 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–35, Jan. 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8044.txt
[49] "FreeRADIUS man pages: unlang," https://freeradius.org/radiusd/man/unlang.html, accessed 01-21-2020.
[50] "busybox Docker Image," https://hub.docker.com/_/busybox, accessed 01-21-2020.
[51] "Alpine Linux Docker Image," https://hub.docker.com/_/alpine, accessed 01-21-2020.
[52] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2015, pp. 171–172.