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

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

***Abstract*—Restricted application containers (RACs) encapsulate applications with their dependencies and configuration for execution on a hypervisor host. xRAC [1] is a novel approach for execution control and network access control (NAC). That is, a RAC can be executed only after successful authentication and authorization (AA) and obtain limited access to network resources. A RAC has a unique IPv6 address so that its traffic is identifiable and controllable by network components. For AA, xRAC adopts and extends components and procedures of 802.1X. We publish its source code and a testbed setup guide on GitHub [2]. In this paper, we give a brief overview on the architecture and functionality of xRAC, describe the prototypical implementation, a testbed, and four demo scenarios.**

***Index Terms*—Network Access Control, Application Execution Control, 802.1X, SDN**

I. Overview on xRAC

In today's networks, traffic is increasingly encrypted so that traffic control becomes hard, e.g., for Quality of Service (QoS) or security purposes, since the content of the traffic is unclear. xRAC [1] tackles this problem. Applications are run in restricted application containers (RACs) on managed hosts and are executed only after successful AA. As each RAC has its own IPv6 address, the traffic of such applications can be easily identified and appropriately treated.

With xRAC, users are assigned permissions to run special RACs on managed hosts. When a user starts a RAC on a managed host, an AA server is contacted to authenticate the user and grant authorization to launch the RAC. Only in case of success, the RAC is executed. Managed hosts may run multiple RACs in parallel to host-native applications.

For authentication, the managed host sends user authentication data (UAND) and container authentication data (CAND) to the AA server. UAND may be a user identity with a password and CAND may be the integrity checksum of the RAC. If the user is permitted to run the RAC, the AA server returns container authorization data (CAZD) to the managed host and network control elements. The former permits execution of the RAC, the latter grants access to protected network resources.

For applying this AA procedure with 802.1X on RACs, we adopt the three 802.1X components with the following modifications to the original standard. First, we introduce a softwarized 802.1X container supplicant (802.1X CS) that runs on the managed host with an interface to the container management daemon (CMD). Second, we introduce a softwarized 802.1X container authenticator (802.1X CA) that is deployed as network application connecting the 802.1X CS and 802.1X authentication server (802.1X AS). Third, we substitute EAP-over-LAN (EAPoL) by EAP-over-UDP (EAPoUDP) for the communication between the 802.1X CS and 802.1X CA. Last, we configure the 802.1X AS to support CAND validation and CAZD. More technical details can be found in the paper [1].

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.

Figure 1 depicts AA of RACs with 802.1X. A user attempts to start a RAC via the CMD (1). The start request includes the RAC name and UAND. The CMD calculates the image checksum as CAND and requests the 802.1X CS for permission to run the RAC (2). The 802.1X CS initiates and performs authentication with the 802.1X AS via the 802.1X CA (3). In case of successful authentication and execution permission, the 802.1X AS responds with CAZD (4) to the 802.1X CA. The 802.1X CA forwards CAZD to the 802.1X CS (4a), the 802.1X CS permits the RAC launch permit of the CMD (4b). In parallel, the 802.1X CA forwards CAZD to network control elements (4c). Here, it instructs an SDN controller to install rules so that the RAC can communicate with the protected server. Now, the RAC but not the managed host or other RACs can communicate with the protected server (5).

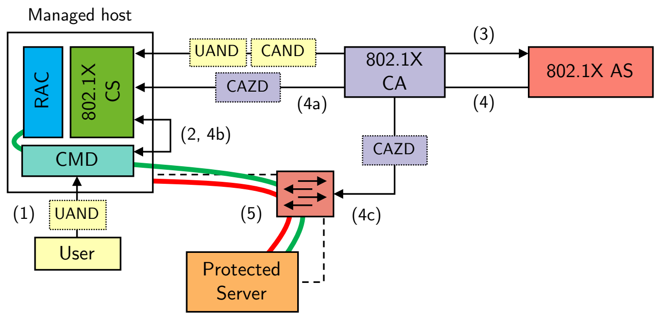

Fig. 1. Components and process of AA in xRAC (similar to [1]).

II. Prototypical Implementation

The prototypical implementation of xRAC comprises three parts. First, the managed host runs RACs with the help of the CMD and 802.1X CS. We use Docker (19.03.5) as virtualization platform, enable IPv6 networking, and create a fixed subnet with globally routeable IPv6 addresses in the CMD configuration. Each RAC receives a dedicated IPv6 global unicast address, the CMD manages the routing table of the host system and enables IPv6 forwarding so that other hosts can reach the RACs. We run the NDP Proxy Daemon (NDPPD) [3] that responds to neighbor solicitation request for RAC addresses with the MAC address of the managed host. We implement the 802.1X CS as plugin for the Docker Authorization (AuthZ) framework [4]. The AuthZ framework provides a REST API that allows individual authorization plugins to approve or deny requests to the CMD. We implement the 802.1X CS as Flask [5] web application and run it with with the uWSGI [6] application server. Second, the 802.1X CA is implemented as application for the Ryu SDN controller framework [7] (v4.34). Therefore, we extend the 802.1X Authenticator from our previous work [8] by AA for RACs using EAPoUDP. As example for network access control (NAC), we extend a L2 switch by access control lists. Static white-list entries that allow communication with public hosts are defined by the administrator. Dynamic white-list entries are added by the 802.1X CA when receiving CAZD from the 802.1X AS. Last, we use FreeRADIUS (v3.0.16) as 802.1X AS. We extend its data model with vendor-specific attributes (VSAs) for CAND/CAZD and add unlang [9] control sequences to validate CAND within AA.

III. Testbed Environment

Figure 2 depicts the testbed environment. We execute Virtual Box virtual machines (VMs), an Open vSwitch (OVS) instance, the Ryu SDN controller, and a FreeRADIUS instance on a ThinkPad T460s with an i5-6200U CPU, 20 GB RAM, SSD, and running Ubuntu 18.04.3 LTS. The managed host is deployed as VM running Ubuntu Server 18.04.3 LTS. The managed host VM is configured with the IPv6 subnet `2001:db8::11:0/116`. The `docker0` interfaces receives `2001:db8::11:1`, xRAC containers receive IPv6 addresses starting from `2001:db8::11:2`. Both public and protected servers are VMs running Ubuntu Server 18.04.3 LTS. The public server is configured with the IPv6 address `2001:db8::aa:0`, the protected server with `201:db8::bb:0`. All VMs are interconnected via an OVS bridge with a tap device. The Ryu SDN controller with the 802.1X CA application manages the OVS bridge via OpenFlow. OVS, the Ryu SDN controller, and FreeRADIUS are directly executed on the local testbed system. A GitHub repository [2] includes a complete setup guide and the source code of all software components.

IV. Demo Scenario

We describe the experiments shown in the demonstration. For visualizing xRAC's operations, we open four console windows on the testbed computer that show the CLI of the managed host, the console output of the 802.1X CS application, the console output of the 802.1X CA application, and the console output of the 802.1X AS.

We perform the following four experiments from the managed host that should demonstrate xRAC's functionality. First, we show that only the public, but not the private server are

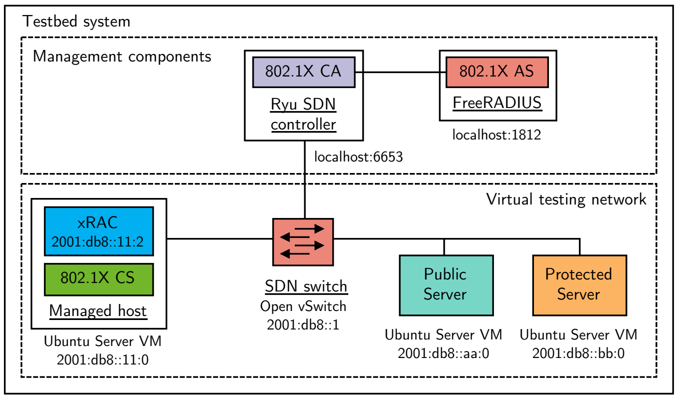

Fig. 2. Testbed environment with xRAC components.

reachable before AA. We send ICMP echo request packets to both servers but only receive ICMP echo response packets from the public server. Second, we show that the latest Busybox [10] container image as example for a RAC can successfully access the protected server after AA. We define UAND (username and password), CAND (checksum of the Busybox image), and CAZD (IPv6 address of the protected server) on the 802.1X AS. We start a new Busybox RAC with the user credentials as UAND. With successful AA, the 802.1X CA adds a dynamic white-list entry so that the RAC can communicate with the protected server. RAC start is granted, and we show successful AA by sending ICMP echo request packets from the RAC to the protected server that are answered with ICMP echo response packets. Third, we show that only the RAC, but not the managed host can access the protected server after AA. While sending ICMP echo requests from the RAC, we start to send ICMP echo requests from the managed host to the protected server. The RAC still receives ICMP echo response packets, but no answers arrive at the managed host. Last, we show that only permitted RACs can be started on the managed host. Instead of Busybox, we try to start an Alpine Linux [11] container image. As expected, execution on the managed host is denied by the 802.1X CS.

Acknowledgment

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

References

[1] F. Hauser, M. Schmidt, and M. Menth, "xRAC: Execution and Access Control for Restricted Application Containers on Managed Hosts," in *IEEE/IFIP Network Operations and Management Symposium 2020.*
[2] "xRAC Repository on GitHub," https://github.com/uni-tue-kn/xrac.
[3] "ndppd," https://github.com/DanielAdolfsson/ndppd.
[4] "Docker AuthZ," https://docs.docker.com/engine/extend/plugins_authorization/.
[5] "Flask," https://palletsprojects.com/p/flask/.
[6] "uWSGI," https://uwsgi-docs.readthedocs.io/en/latest/.
[7] "Ryu SDN Controller Framework," https://osrg.github.io/ryu/.
[8] F. Hauser, M. Schmidt, and M. Menth, "Establishing a Session Database for SDN using 802.1X and Multiple Authentication Resources," in *IEEE International Conference on Communications 2017.*
[9] "unlang," https://freeradius.org/radiusd/man/unlang.html.
[10] "busybox Docker Image," https://hub.docker.com/_/busybox.
[11] "Alpine Linux Docker Image," https://hub.docker.com/_/alpine.
All web resources were last accessed on 01-12-2020.