Security Testing The O-RAN Near Real Time RIC & A1 Interface

Kashyap Thimmaraju
kashyap.thimmaraju@tu-berlin.de
Technische Universität Berlin
Berlin, Germany

Altaf Shaik
a.shaik@tu-berlin.de
Technische Universität Berlin
Berlin, Germany

Sunniva Flück
fluecks@tu-berlin.de
Technische Universität Berlin
Berlin, Germany
ETH Zürich
Zürich, Switzerland

Pere Joan Fullana Mora
perejfm@gmail.com
Technische Universität Berlin
Berlin, Germany

Christian Werling
cwerling@sect.tu-berlin.de
Technische Universität Berlin
Berlin, Germany

Jean-Pierre Seifert
jean-pierre.seifert@tu-berlin.de
Technische Universität Berlin
Berlin, Germany

ABSTRACT

Open-Radio Access Network (O-RAN) is the next evolutionary step in mobile network architecture and operations and the Near-Real Time RAN Intelligent Controller (Near-RT RIC) plays a central role in the O-RAN architecture as it interfaces between the orchestration layer and next generation eNodeBs.

In this paper we highlight the architectural weakness of a centralized controller in O-RAN by first drawing parallels with the Software-Defined Networking (SDN) controller. We then present a two part security evaluation of two open-source Near-RT RICs (μONOS and OSC), focused on the newly introduced A1 interface of the Near-RT RIC. In the first part of our evaluation, we evaluate the supply-chain risks of μONOS and OSC using off-the-shelf open-source dependency analysis and configuration file analysis tools. In the second part, we present our run-time security testing of the A1 API implemented by μONOS and OSC using our custom O-RAN A1 Interface Testing Tool (OAITT).

Our supply-chain risk analysis shows that both the open-source Near-RT RICs we evaluated have multiple dependency risks and weak or insecure configurations. We identified 211 and 285 known dependency vulnerabilities in μONOS and OSC respectively of which 82 and 190 dependencies were rated as high CVSS respectively. The A1 interface contributed to a majority of the dependency risks in both Near-RT RICs. From a security misconfiguration perspective, we identified issues concerning access control, lack of encryption and poor secret management.

Our run-time testing of OSC and μONOS revealed the following. First, both Near-RT RICs lack TLS for the A1 interface. Second, malicious Non-Real Time RAN Intelligent Controller (Non-RT RIC)s or rApps that reside in the Non-RT RIC could tamper with policies installed in the Near-RT RIC which can impact the availability of the O-RAN. Third, the A1 protocol could be exploited by Non-RT RICs for covert communication via the Near-RT RIC. Fourth, the A1 implementation by μONOS was vulnerable to degradation of service attacks (10–60s response time for GET requests) and a denial of service attack, the latter has been ethically reported and a fix is underway.

WiSec '24, May 27–30, 2024, Seoul, Republic of Korea
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0582-3/24/05.
https://doi.org/10.1145/3643833.3656118

CCS CONCEPTS

• **Networks** → **Network architectures**; **Mobile networks**; • **Security and privacy** → **Software and application security**.

KEYWORDS

O-RAN; Software Security; Security Testing; Near-Real Time RIC

ACM Reference Format:
Kashyap Thimmaraju, Altaf Shaik, Sunniva Flück, Pere Joan Fullana Mora, Christian Werling, and Jean-Pierre Seifert. 2024. Security Testing The O-RAN Near Real Time RIC & A1 Interface. In *Proceedings of the 17th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '24), May 27–30, 2024, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3643833.3656118

1 INTRODUCTION

The introduction of O-RAN by the O-RAN alliance presents new opportunities and challenges to the telecommunication industry. O-RAN is designed to open up the RAN market place and avoid vendor lock-in by having interoperability, open interfaces, programmability and virtualization as central design goals [7]. The O-RAN marketplace is active with several vendors (e.g., Nokia, Ericsson, VMware, Juniper Networks, etc.) and operators (ATT, Orange, Deutsche Telekom, NTT Docomo, Rakuten, etc.) making contributions to the specifications and software community.

A high level architecture of O-RAN illustrated in Figure 1 shows two key and new components introduced to the RAN in O-RAN (in addition to others): the *Non-RT RIC* and the *Near-RT RIC*. These components are interconnected through the A1 REST API based on the O-RAN A1 protocol [8, 10].

The Near-RT RIC plays a critical role in O-RAN [11] as it implements a number of critical functions: network information, topology, routing, policy management, conflict mitigation and messaging infrastructure. Its central role in O-RAN is further exemplified by the four different interfaces it implements: A1, E2, O1 and Y1.

The Near-RT RIC can be viewed analogous to the centralized controller in SDN [28]. In SDN the chief function of the controller is to receive configuration and policy information from a service and management orchestration layer, compute topology and routing information and configure the switches (data plane) with flow-rules. Similarly, in O-RAN the Near-RT RIC interfaces with the SMO (which also includes the Non-RT RIC) from which it receives configurations and policies, maintains a topology database and configures the various E2 nodes (gNB, CU, DU, etc.).

Previous SDN research [38] has shown that a centralized network controller can be compromised and results in compromising the entire SDN setup. Therefore, if the Near-RT RIC is compromised in an O-RAN deployment, it can potentially result in the entire deployment being compromised. Furthermore, the Near-RT RIC being a shared resource amongst the Non-RT RIC and E2 nodes introduces the possibility for covert channels as demonstrated in SDN [23, 37]. Covert channels pose a threat as malicious entities could exfiltrate information such as private keys/certificates, or coordinate an attack that is part of a stealthy operation.

Indeed, the complex architecture of O-RAN and the sheer diversity in software introduce concerns regarding the security and privacy of O-RAN deployment. The O-RAN Alliance security working group have analyzed the threats, risks of the O-RAN architecture [6] and also identified multiple threats and solutions for the Near-RT RIC, Non-RT RIC and the newly introduced O-RAN interfaces, e.g., A1 and E2. The security working group particularly highlight the consequences of an attacker exploiting the Near-RT RIC and the RAN via false policies [3]: by installing false policies or manipulating the topology within the Near-RT RIC, an attacker could cause denial of service attacks, degradation of services or even isolate User Equipment (UE).

From a theoretical standpoint, there have been numerous reports on the security challenges O-RAN introduces. For example, Mimran et al. [29] point out concerns on the O-RAN ecosystem being a major threat, Shen et al. [35] emphasize authentication and authorization vulnerabilities, Liyanage et al. [27] provide a comprehensive risk assessment of O-RAN, Klement et al. [21] describe solutions for security challenges in O-RAN, and Köpsell et al. [22] assessed the risk of O-RAN with respect to confidentiality, integrity, accountability, availability and privacy.

On the contrary, security research practically oriented towards O-RAN and the Near-RT RIC has been sparse. Groen et al. [17] measured the overhead of using IPsec to secure the E2 interface between the Near-RT RIC and E2 nodes. Tiberti et al. [39] demonstrated a man-in-the-middle attack on the O-RAN Software Community (OSC) Near-RT RIC and A1 interface in a default OSC Kubernetes setup. Furthermore, the authors provided a first glimpse into the potential for tampering with A1 policies installed in the Near-RT RIC. The latter is discussed in detail in this paper.

Given the chief role the Near-RT RIC plays in O-RAN, the velocity with which mobile network operators are looking to deploy O-RAN in production environments, the limited practical security research on Near-RT RICs and the consequences of an attacker compromising the Near-RT RIC, in this paper we focus on security testing open-source Near-RT RIC software namely, µONOS and OSC, that are highly likely to be used in production setups. Our testing methodology consists of two parts. We first assess the security risks of open-source Near-RT RICs using static program analysis techniques: dependency analysis and configuration analysis; and then test the O-RAN A1 protocol run-time implementation on the Near-RT RICs using dynamic program analysis. To that end, this paper makes the following **contributions:**

- We describe the relationship between O-RAN and SDN and the security implications of using a centralized controller in O-RAN.
- We conduct a security risk assessment of two state-of-the-art open-source Near-RT RICs for known security vulnerabilities via dependency and configuration analysis.
- We design and implement an O-RAN A1 interface security testing tool (OAITT) which currently includes SSL/TLS, OAuth, Denial of service, Policy fuzzing, URL fuzzing and Policy tampering tests.
- We evaluate the A1 protocol implementation using OAITT on two open-source Near-RT RICs.
- We present novel timing and storage covert channels between Non-RT RICs and the Near-RT RIC that exploit the A1 protocol.
- We introduce Threat "T-A1-04" in the *O-RAN Study on Security for Non-RT-RIC* (to be published in 2024).
- We identified and reported a denial of service vulnerability in the µONOS (version 1.4) A1 interface implementation.

Paper Structure. The remainder of this paper is organized as follows. In Section 2 we provide background information on O-RAN followed by our threat model in Section 3. In Section 4 we describe our security risk assessment of two open-source Near-RT RIC implementations. In Section 5 we present our O-RAN A1 Interface Testing Tool and the results from testing the A1 interface. Section 6 outlines recommendations for the O-RAN community, followed by related work in Section 7 and then our concluding remarks in Section 8.

2 OPEN-RADIO ACCESS NETWORKS

We start by providing an architectural overview of O-RAN followed by specifics on the A1 interface and end with a note on the software architecture of two open-source Near-RT RICs.

2.1 Architectural Overview

The high level architecture of an O-RAN is illustrated in Figure 1. Broadly speaking it comprises of three major components: Service Management and Orchestration Framework (SMO) which also includes the so-called Non-RT RIC, the Near-RT RIC and E2 nodes which include the next generation eNodeB (O-eNB), Central Unit (O-CU), Distributed Unit (O-DU) and Radio Unit (O-RU).

The SMO handles RAN domain management and provides Fault, Configuration, Accounting, Performance and Security (FCAPS) interfaces to O-RAN network functions. The Non-RT RIC in the SMO supports RAN optimization via policies, enrichment information and machine learning management which are sent to the Near-RT RIC via the A1 interface. The new configuration received by the Near-RT RIC is then transformed into E2 specific configuration and installed on the respective nodes. The Non-RT RIC can provide radio resource management in time intervals that are greater than 1s, therefore the name 'Non Real Time' [7]. The Non-RT RIC supports so-called rApps, which are applications meant to offer more flexibility in the RAN, e.g., mobility load balancing [13]. rApps

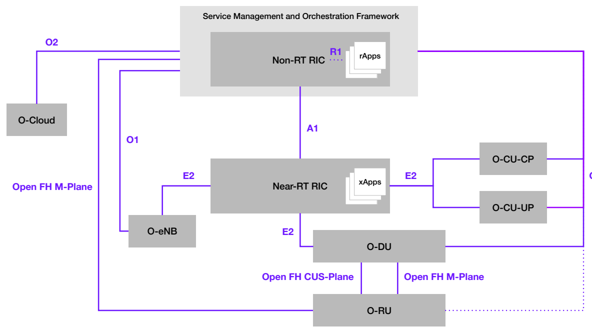


Figure 1: High-level architecture of O-RAN that illustrates the newly introduced O-RAN components and interfaces. The 3GPP components and interfaces are not illustrated here.

can create or update policies or enrichment information and forward it to the A1 termination in the Non-RT RIC.

The Near-RT RIC is designed for RAN optimization and hosts multiple functions, including routing and topology, policy management, management of the xApps, conflict mitigation and messaging infrastructure. The Near-RT RIC is near real time as it is designed to have use-case specific control loops of 10ms−1s over E2 nodes. The xApps within a Near-RT RIC are applications designed to provide optimization and control of the E2 nodes over the E2 interface or collect data from the E2 nodes in real time. Example xApps include traffic steering, dynamic spectrum sharing, interference management and smart handover. The xApps can also receive optimization information from an rApp/Non-RT RIC via the A1 interface.

The E2 Nodes are the logical endpoints of the E2 interface and facilitate communication and coordination between the O-CU and the O-DU. E2 nodes are responsible for radio resource management, mobility management, and other coordination tasks essential for efficient RAN operation.

2.2 A1 Interface

As described previously, the A1 interface connects the Non-RT RIC in the SMO and the Near-RT RIC. It enables the Non-RT RIC to provide i) policy-based optimization and management information and; ii) enrichment information and machine learning data. The A1 interface is set up as a REST API with client/server endpoints in the Near-RT RIC and in the Non-RT RIC [9]. The A1 interface is meant to be secured via TLS for confidentiality and mutual authentication between the Non-RT RIC and Near-RT RIC [10] and OAuth 2.0 for authorization. Since we only focus on policy information in this paper, we merely describe that.

Policies. The policies sent over the A1 interface can be for a variety of functions: Radio Resource Management (RRM), Load Balancing, Handover, Interference Management, Resource Allocation, Quality of Service, Energy Efficiency and Traffic Management. More technically, the A1 policies are represented as JSON objects strictly defined by schemes that are provided by the xApps deployed in the Near-RT RIC. All policies have a *policy type identifier* and a *policy identifier* to uniquely identify the respective policy type and the policy within a policy type. Furthermore, each policy also specifies a *scope identifier* and *policy statements*. The scope identifier defines the target for the policy, e.g., UEs, quality of service (QoS) flows or cells. The policy statements are the objectives to be achieved upon deploying the policy, e.g., UE-cell preferences or flow priority. The policies, as soon as they are accepted by the A1 server in the Near-RT RIC and the xApp, are then communicated to the E2 nodes, where feedback is returned if the enforcement is successful [9]. Policies on the Near-RT RIC can be deployed, altered or deleted via HTTP GET, PUT and DELETE messages.

2.3 Near-RT RIC Software Architecture

In general, the O-RAN Near-RT RIC architecture comprises of three components: RIC platform, APIs and xApps. The RIC platform provides the underlying infrastructure, services and functions to enable programmatic control over the Near-RT RIC and E2 nodes as well as the ability to host multiple xApps. Some of the key components of the RIC platform are the O1, A1, Y1 and E2 termination interfaces (see Section 2.1), a database (to store RAN information, network state, E2 configuration, etc.), management, AI/ML support, security and a messaging infrastructure. The Near-RT RIC APIs enable the xApps to communicate with the Near-RT RIC so that developers can implement business/platform specific logic in an xApp to control the RAN.

Upon surveying the Near-RT RIC open-source software landscape, we identified µONOS and OSC. µONOS is an initiative by the Open Networking Foundation and Linux Foundation while OSC is a community effort by O-RAN vendors, operators and academics. Both Near-RT RICs follow a micro-service architecture where each micro-service corresponds to a specific RIC component. The micro-services are run as containers using Docker or Containerd. Helm charts are used as the package and resource manager for the aforementioned containers while Kubernetes is used for container management and orchestration.

The platform aspect of µONOS comprises of the `onos-topo` (for routing information), `onos-uenib` (for managing user equipment related information), `onos-e2t` (the E2 termination interface), `onos-a1t` (the A1 termination interface) and `onos-config` (for configuration and management). It also has a command line interface micro-service `onos-cli` that can be used to configure the Near-RT RIC. For an xApp that uses the A1 interface in µONOS, we have the Rimedo Labs Traffic Steering xApp [30], and for key performance indicator (KPI) monitoring, we have the `onos-kpimon`.

OSC on the other hand comprises of several components, some of which we highlight here. The platform related components of OSC typically have the `ric-plt-` prefix while the xApp components have the `ric-app-` prefix. The platform and xApp related components are shown in the X-axis of Figure 3. Compared to µONOS, OSC attempts to fulfill the Near-RT RIC architecture requirements. Another obvious difference between the two is in the implementation of the E2 interface: OSC splits it up into `ric-plt-e2` and `ric-plt-e2mgr`, the former is responsible for establishing the SCTP connection with an E2 node while the latter is for managing connection establishment and API management of the E2 node.

3 THREAT MODEL

In our attack model targeting the Near-RT RIC we consider the attacker as an insider in the operational O-RAN environment with access to the A1 interface, as O-RAN is designed to operated in a zero-trust environment. This insider could take the form of a cloud provider, O-RAN operator, or O-RAN software supplier. A cloud provider manages the cloud infrastructure hosting O-RAN components, while an O-RAN operator has complete control over the RAN. On the other hand, a software supplier (or vendor) to operators, providing software, e.g., Near-RT RIC, xApps or rApps, can communicate directly with the Near-RT RIC via standard APIs.

We assume that the attacker's goal is to compromise the operation of the O-RAN via the Near-RT RIC: from the Near-RT RIC the attacker potentially has access to and control over the entire O-RAN, e.g., UE information, traffic steering, etc. Although, the attacker can launch an attack from the A1, Y1 consumers or the E2 nodes, in this paper, we restrict the scope to that of the A1 interface as it is newly introduced by the O-RAN alliance and the policy server on the Near-RT RIC combined with xApps, impacts the performance, security and resources of E2 nodes: CU, DU or RU [3].

As described in Section 2, the A1 interface is protected via TLS and OAuth. However, TLS can only prevent external threats, insiders are presumed to have TLS connectivity to the A1 interface. Additionally, OAuth requires additional resources, mechanisms and configuration, it does not ship with the Near-RT RIC. We assume the attacker is resourceful (e.g., has computing power and financial support), and is also active and willing to manipulate data beyond her controlled system parts. Since the A1 interface is not accessible via external entities, e.g., handset users (UEs), attacks from the radio interface are not considered.

Having described our attacker, in this paper, we take the role of a white hat hacker by evaluating the security of the Near-RT RIC to reduce the chance of an attack occurring via the A1.

4 SECURITY RISK ASSESSMENT OF OPEN SOURCE NEAR-REAL TIME RICS

From an attacker's perspective on compromising the Near-RT RIC, identifying known vulnerabilities or weak configuration patterns within the code-base (which includes infrastructure as code) of the Near-RT RIC provides a starting point to design and launch an attack. The vulnerability discovered need not necessarily be present within the Near-RT RIC's code but within one of the dependency libraries/packages/images used by the Near-RT RIC. This is highly likely as modern software is typically an assembly of numerous third-party, open-source software [36]. Therefore, a vulnerability in the dependency could be as disruptive and risky as a vulnerability within the application-specific code. The log4j [18] and Heartbleed [41] incidents are prime examples of how dependencies pose critical and costly supply-chain risks.

To perform our risk assessment of theNear-RT RICs (μONOS and OSC), we utilize static application security testing (SAST) as it enables us to evaluate the security of potentially the entire Near-RT RIC code-base statically. In particular our SAST employs dependency analysis and configuration file analysis.

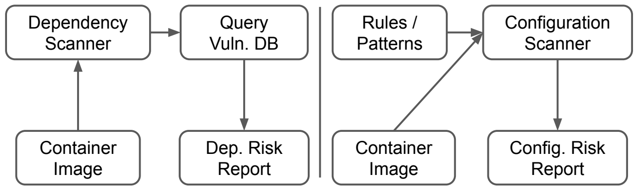


Figure 2: A high-level overview of dependency analysis on the left and configuration file analysis on the right.

4.1 Methodology

Our choice of SAST tools is based on two primary factors. Firstly, both Near-RT RICs extensively incorporate third-party libraries and dependencies. Secondly, they are designed to operate in a microservice architecture which require numerous API specification files, and configuration files tailored for the seamless deployment of containers into a Kubernetes environment. As a result, we have chosen tools capable of delving into code bases, automatically tracking all dependencies, and effectively scanning the relevant configurations files. In the following we describe how dependency analysis and configuration file analysis works and then then the repositories we scanned.

Dependency Analysis. In general, dependency analysis works as shown in Figure 2. The tool scans a particular container image or repository to identify the Open-Source Software (OSS) components and their respective versions. It then queries a vulnerability database, e.g., National Vulnerability Database (NVD), MITRE, GitHub, for Common Vulnerability Exposures (CVEs) for the identified OSS versions and finally reports the findings to the user/analyst. In our evaluation we used Grype [12], Snyk [36] and Trivy [40]. The aforementioned tools support Golang and remote repository scans making them ideal and straightforward to use for OSC and μONOS are written in Golang and are available on GitHub.

Configuration File Analysis. Configuration file analysis on the other hand typically uncovers insecure configurations by employing a rule-based approach where every rule specifies a pattern to look for in configuration files (see Figure 2). For instance, a container running as root is a rule executed against Kubernetes configuration files. The findings are then reported to the user/analyst. We used KICS [20] for our evaluation as it supports Kubernetes YAML and Docker manifests which are used by the two Near-RT RICs. As input KICS requires API specification files, container deployment files, and docker image creation files. For each asset, KICS formulates queries aligned with their respective security standards, best practices, and known vulnerabilities. These queries are then executed against the provided files and configurations of each RIC, leveraging pattern matching to pinpoint particular constructs, configurations, or practices that could pose security risks.

Repositories scanned. We scanned 52 repositories in total: 9 from μONOS (version 1.4) and 43 from OSC (F release). These repositories are continually under active development by diverse members of the O-RAN Alliance. The list of repositories scanned is shown in the X-axis of Figure 3.

Near-RT RIC	Total Vulnerabilities	CVSS Low	Medium	High	Critical
µONOS	211	20	99	82	10
OSC	285	2	86	190	7

Table 1: Distribution of CVSS V3S for µONOS and OSC using Grype, Trivy and Snyk. Note that the same vulnerability might be present in different repositories within the Near-RT RIC.

Near-RT RIC	Dependency	# Of Occurrences
µONOS	github.com/containerd/containerd	29
	golang.org/x/net	28
	github.com/docker/docker	28
	github.com/opencontainers/runc	26
	helm.sh/helm/v3	25
	k8s.io/kubernetes	19
	github.com/docker/distribution	9
	golang.org/x/text	9
	gopkg.in/yaml.v3	8
	golang.org/x/net/http2	7
	github.com/prometheus/client_golang	7
	helm.sh/helm/v3/pkg/strvals	6
	golang.org/x/net/http2/hpack	5
	github.com/opencontainers/image-spec	5
OSC	golang.org/x/net	89
	golang.org/x/text	40
	golang.org/x/crypto	39
	github.com/prometheus/client_golang	17
	golang.org/x/sys	16
	gopkg.in/yaml.v2	12
	go.mongodb.org/mongo-driver	10
	helm.sh/helm/v3	8
	github.com/labstack/echo/v4	6
	PyYAML	6
	github.com/docker/docker	6
	github.com/containerd/containerd	6
	github.com/dgrijalva/jwt-go	5
	github.com/gogo/protobuf	5
	github.com/gorilla/websocket	5
	urllib3	5
	Werkzeug	5

Table 2: Distribution of vulnerable dependencies used in µONOS and OSC with occurrences of five or more.

4.2 Dependency Risks

Data Analysis Notes. On analyzing the data from the scanned repositories, we identified two instances of duplicates: i) the tools might report the same vulnerability within a repository and; ii) the same vulnerability might exist in multiple repositories. We eliminated duplicates across tools within the same repository but counted them across repositories because the latter represents another instance of a vulnerability within the Near-RT RIC regardless of it being the same CVE. Next, we also observed that Snyk and Trivy, reported GitHub and Gitlab security advisories, or CVEs with CVSSs from GitHub and NVD. For such cases, we excluded Gitlab security advisories and only included CVEs with CVSS V3S from NVD making our analysis more conservative.

Distribution of Vulnerabilities Found. To obtain an overview of the dependency risks across µONOS and OSC and their respective components, we first count the total number of identified vulnerabilities for each Near-RT RIC and then the number of vulnerabilities within the respective Near-RT RIC repository. Table 1 tabulates the data for total number of vulnerabilities found in µONOS and OSC. We have 211 and 285 vulnerabilities for µONOS and OSC respectively. Figure 3 plots the distribution of unique vulnerabilities within a Near-RT RIC's individual repositories/micro-service component. For µONOS, we can see that the A1 (`onos-a1t`) termination, the KPI Monitor (`onos-kpimon`) and UE Network Information Base (`onos-uenib`) have around 60 vulnerabilities each whereas the E2 (`onos-e2t`) and the Config (`onos-config`) have zero vulnerabilities. For OSC, the KPI monitor (`ric-app-kpimon-go`), A1 (`ric-plt-a1`), xApp manager (`ric-plt-appmgr`), E2 manager (`ric-plt-e2mgr`), O1 (`ric-plt-o1`), Deployment Management Service (`ric-plt-ricdcms`) and Routing manager (`ric-plt-rtmgr`) have approximately 20–40 vulnerabilities whereas 28 repositories had zero vulnerabilities. Comparing the two Near-RT RICs we observe that in both cases, the A1 interface and the KPI monitor xApp is particularly risky. In OSC, we observe that even the E2 and O1 have a number of security risks.

Distribution of CVSS. To understand the severity of the identified vulnerabilities, we leverage the Common Vulnerability Scoring System (CVSS) version 3 (V3) that is reported along with each CVE. The distribution of low, medium, high and critical CVSSs across µONOS and OSC are tabulated in Table 1. We can see that µONOS has 82 high and 10 critical vulnerabilities whereas OSC has 190 high and 7 critical vulnerabilities. Due to the limited space, we only highlight repositories with high or critical vulnerabilities here. µONOS had four critical vulnerabilities in `onos-a1t` and two in `onos-kpimon, onos-operator, onos-uenib` while OSC had five critical vulnerabilities in `ric-plt-appmgr`. With respect to high CVSS, `onos-a1t, onos-kpimon, onos-uenib` in µONOS had 18–24 vulnerabilities; in OSC we highlight `ric-app-kpimon-go, ric-plt-a1, ric-plt-appmgr, ric-plt-e2mgr, ric-plt-o1,` to have 13–24 high vulnerabilities. Again, we observe that the A1, E2 and O1 interface implementations pose high to critical security risks to the Near-RT RICs.

Distribution of Dependencies. To uncover the dependencies that introduce the most risk, we extract the vulnerable dependency and count the number of occurrences of that dependency which is tabulated in Table 2. For conciseness, we only show dependencies that have five or more occurrences for the two Near-RT RICs. With µONOS, we discern that a majority of the dependency risks are micro-service infrastructure related: (`containerd, docker, runc, helm, kubernetes, opencontainers`) and then networking related (`x/net, x/net/http2, x/net/http2/hpack`. For OSC, we have networking (`x/net, echo/4, gogo/protobuf, gorilla/websocket, urllib3`), crypto (`x/crypto`) and system (`x/sys`) related dependencies as major risks. Unlike µONOS, in OSC micro-service infrastructure related issues are a minority. Delving into the dependencies per repository, we uncover the previously described pattern where the A1, E2, O1 as well as the KPI monitor, UE NIB, Routing Manager, xApp Manager and DMS repositories have considerable dependency risks in terms of quantity (i.e., several dependencies) and frequency (i.e., multiple occurrences of the same dependency).

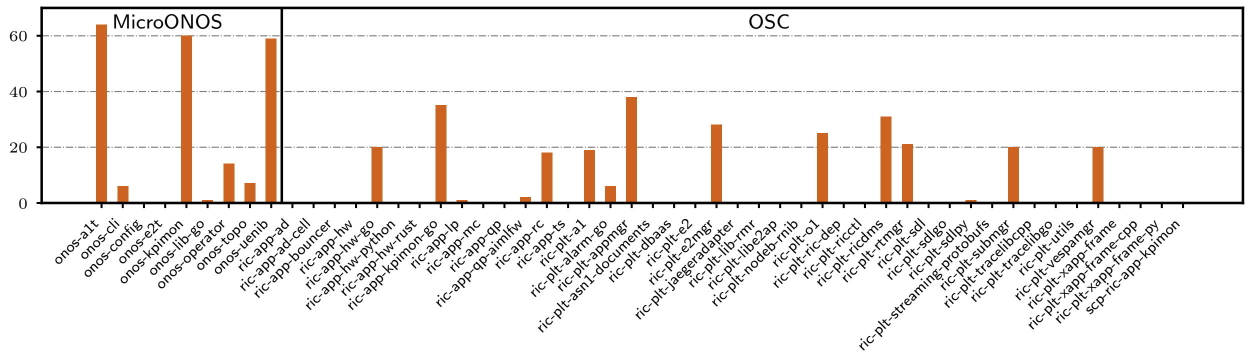


Figure 3: Distribution of potential vulnerabilities in µONOS and OSC reported by Grype, Snyk and Trivy.

4.3 Configuration Risks

In total, KICS (v1.7.11) completed a scan of 695 files and nearly 90,000 lines of code within a span of 20 minutes. The scan revealed 820 vulnerabilities with a high severity rating, 9,685 with a medium rating, and 350 with a low severity rating. For detailed insights into the individual heuristics and reports of each repository, please see the Appendix A.

While KICS discovered numerous vulnerabilities in the OpenAPI, Kubernetes, and Docker technologies used in the RICs, for brevity we focus on the most critical security concerns in this paper. The emphasis is on clarifying the key issues found in the configuration files, along with providing an example of the affected software component (file/configuration) and its impact on the overall system.

Access control. The Near-RT RIC has the capability to employ the "security schemes" attribute to restrict user access based on authentication and authorization. Commonly available schemes comprise OAuth, HTTP basic auth, API keys, etc. Our examination brought to light that security attributes are not utilized in the API specification files for all the APIs employed in Near-RT RIC. As an illustration, the RIC DMS (Deployment Management Services) in OSC lacks the "security schemes" attribute, consequently rendering it unable to enforce access control. This loophole with high severity allows any arbitrary user to deploy an xApp or similar applications in the O-RAN environment.

Insecure Configuration. KICS detected insecure configurations in various repositories that feature xApp deployments through Kubernetes. For example, `onos-a1t` repository in µONOS offers sample configuration templates (as YAML) files to specify various rules and settings including its privileges, access controls, etc. Precisely, in defining a security context for deploying an xApp as a pod or a container, the templates overlooked setting crucial attributes such as "allowPrivilegeEscalation" and "runAsRoot". Kubernetes' best practices [24] recommend explicitly disabling such attributes, because their absence could result in automatic privilege escalation. Thus, an xApp provider can gain control over the Near-RT RIC, and interact with many other parts of the system.

Build process. With KICS we made another critical observation in the container image build process: all provided Docker manifests files used for build, deployment, and execution lack specification of a "user" to run the image. This omission could result in the image running as root, potentially gaining unnecessary higher privileges than required.

Encryption. OpenAPI specification files contain the *schemes* attribute for indicating the transport mode for communication between various entities. Our observation shows that OSC uses the "http" scheme, permitting any xApp to subscribe and register, thereby exposing the communication to interception. It is recommended that OSC should exclusively support the "https" scheme to enhance security.

Secret management. The Near-RT RIC repositories leverage the Kubernetes *secrets* [25] feature to safeguard highly sensitive information within applications, encompassing API keys, tokens, and database passwords. *secrets*, comprising key-value pairs and metadata, are defined in a manifest file and exposed as environment variables for a container. For instance, during the scan of the `ric-plt-ric-dep` repository from OSC, KICS flagged a severe issue: the secrets were published to the Git repository, posing a concern of "Hard-coded secret key appears in source".

Resource management. When creating a container template, an optional configuration is specifying the resources each container can use on a Kubernetes node, commonly CPU and memory (RAM). However, all configuration files lack the specification of required resources or defined resource limits. Consequently, it fails to enforce limits on the containers, potentially leading to resource exhaustion by an xApp or other components.

4.4 Key Takeaways

Our static analysis on the dependencies and configurations used currently by the µONOS and OSC code bases lead to the following risk assessment.

Infrastructure, communication and crypto related dependencies introduce the most risk to µONOS and OSC. The communication and crypto related dependencies are directly linked with the O-RAN interfaces: A1, E2 and O1. For example, the A1 and O1 require http libraries and the E2 requires SCTP, gRPC and protobuf. The A1, E2 and O1 in addition to the xApp and routing are also the repositories reported with the most and highest CVSS rated vulnerabilities.

With almost 820 high severity issues in RICs, the software development in O-RAN raises concerns. KICS has extensively scanned multiple types of configuration files and highlights the absence of standard security practices across various technology platforms used—OpenAPI, Kubernetes, container and Docker. While the reported numbers may appear elevated, their significance is mitigated by the possibility of multiple reports for each type of vulnerability within various configuration files. This can occur when a rule/pattern is violated numerous times or when multiple files infringe upon the same rules/patterns. But, what is most important is that many issues originate from a foundational problem of not adhering to the security by design approach in OSC making A1 and E2 interfaces weaker. With the current software available in OSC and µONOS, our insider type attacker can implant malicious applications into the RIC platform and execute them with highest privileges possible.

5 SECURITY TESTING THE O-RAN A1 API

A static security analysis of the RICs has highlighted concerns related to secure development practices, posing a significant supply chain risk. However, a more in-depth understanding of these risks, especially those related to the logic implementation of the code, requires a dynamic assessment through the exercise of the O-RAN. Additionally, threats like OAuth bypass and policy tampering cannot be verified statically, as they depend on run-time configurations. To address this, we conduct a run-time security evaluation of OSC and µONOS in this section, starting with our testing setup, the O-RAN A1 Interface Testing Tool, and concluding with the results of our evaluation.

5.1 Testing Setup

For the security tests for OAITT, we combined security tests from the O-RAN security requirements, [4], security test specifications [5] and end-to-end test specification [2] as well as our own "corner cases". The currently supported tests are described in the following subsection.

In general, our Near-RT RIC was the device under test (DUT) system which was connected via a Netgear 8x Gigabit switch to another system from which we executed our security tests. The latter system was an Intel(R) Core(TM) i3-6100U Processor @ 2.30 GHz CPU with 16 GB RAM system running the Ubuntu 18.04 operating system.

As Near-RT RICs we used µONOS and OSC. We leveraged SDRAN-in-a-Box version 1.4's implementation of µONOS on an Ubuntu 18.04 OS which had the following hardware specifications: 4 Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz CPUs and 16GB RAM. As, an xApp that consumed A1 policies, we used the Rimedo labs Policy-based Traffic Steering xApp (`make OPT=rimdots VER=1.4.0`) and the SDRAN-in-a-BOX provided RAN-Simulator [31].

For OSC, the *F* release was set up on a UTM virtual machine with Ubuntu 22.04. The setup ran with Docker 20.10.21, Helm 3.5.4 and Kubernetes 1.19.0. We used the provided RAN simulator for the E2 nodes and connected them with the xApp HW-go as demonstrated in the demo [33]. We altered the `bin/install_k8s_and_helm.sh` script to deploy OSC in our environment. For setting up an E2 node with the xApp, we modified the `kpm_callbacks.cpp` C++ file from the sim-e2-interface repository. Since the HW-go app is not a policy app and there was no OSC xApp that used the A1 policies at the time of our testing in 2023, our evaluation of OSC focused on TLS and OAuth 2.0.

5.2 O-RAN A1 Interface Testing Tool

To aid our security evaluation of the Near-RT RIC A1 API, we designed and developed OAITT to operate with different Near-RT RICs, log relevant information and be open-source. OAITT is mostly written by us using Python 3.9 but also includes a third-party tool for TLS testing [15] and a traffic generator [26] to craft our A1 specific traffic patterns. OAITT currently supports the following six security tests.

(1) SSL/TLS: OAITT tests whether the policy server enforces TLS 1.2 or 1.3 connections as specified by the O-RAN Alliance in the test case 15.4.1.STC-15-15.4-001 [5] and security specification 5.1.3.2 [4].
(2) OAuth 2.0: OAITT checks if the interface under test has OAuth 2.0 enforced by issuing a GET request and inspects the underlying authorization requirements. This is required as specified by the O-RAN Alliance in test case 15.4.3.STC-15-15.4-003 [5] and security specification 5.1.3.2 [4].
(3) Policy fuzzing: OAITT checks whether a policy is accepted or rejected depending on alterations of the policy contents. To this end all the keys and values of the provided JSON policy object are mutated in multiple iterations. The policy server is expected to reject all policies with altered keys and; reject all values that are not in the scope of the provided scheme. This test checks if the requirements in 5.2.4.3.1 in the A1 Application Protocol Specification from the O-RAN Alliance are respected [8].
(4) URL fuzzing: OAITT searches for undeclared URLs that can be accessed in the policy server by randomly mutating the length and string of the URL starting with the base URL of `policytypes/`. This evaluates if the server follows the URL structure defined by the O-RAN alliance [8].
(5) Policy tampering: OAITT tests whether a Non-RT RIC can tamper with A1 policies installed by another Non-RT RIC, e.g., by modifying or deleting the policy using a sequence of GET, PUT & DELETE requests.
(6) Denial of service: OAITT tests five A1-specific API traffic patterns to evaluate the A1 implementation.
    (a) Entry-point: The shortest legitimate GET requests to the server which we consider as our baseline.
    (b) Incomplete URL: GET requests for a URL that contains no information (e.g., the URL field is incomplete)
    (c) Asymmetric: GET requests to the server whose response is larger than the requested data.
    (d) Correct Policy: PUT requests to the server with a legitimate policy.
    (e) Faulty Policy: PUT requests to the server with an invalid policy (e.g., incorrect key value).

5.3 A1 API Security Testing Results

We now present the results from our evaluation using OAITT. Table 3 provides an overview of the key results.

Near-Real Time RIC	Security Test									
	TLS	OAuth	Policy Fuzzing	URL Fuzzing	Policy Tampering & Covert Channels	Denial of Service				
						Entrypoint	Asymmetric	Incomplete	Correct Policy	Faulty Policy
OSC	Absent	Absent	NA	NA	NA	NA	NA	NA	NA	NA
μONOS	Absent	Absent	Passed	Passed	Vulnerable	Degradation of service	Degradation of service	Passed	Denial of service	Passed

Table 3: Overview of security results from OAITT.

5.3.1 TLS and HTTPS. SSL/TLS is not supported/implemented on the A1 policy server for μONOS and OSC. As a result an attacker on the path between the Near-RT RIC and Non-RT RIC can eavesdrop on the A1 interface.

5.3.2 OAuth 2.0. OAuth 2.0 is not supported/implemented on the A1 policy server for μONOS and OSC by default. As a result, an arbitrary (authenticated) service in the Near-RT RIC microservice architecture could manipulate the functioning of O-RAN deployments by exploiting the Near-RT RIC A1 implementation.

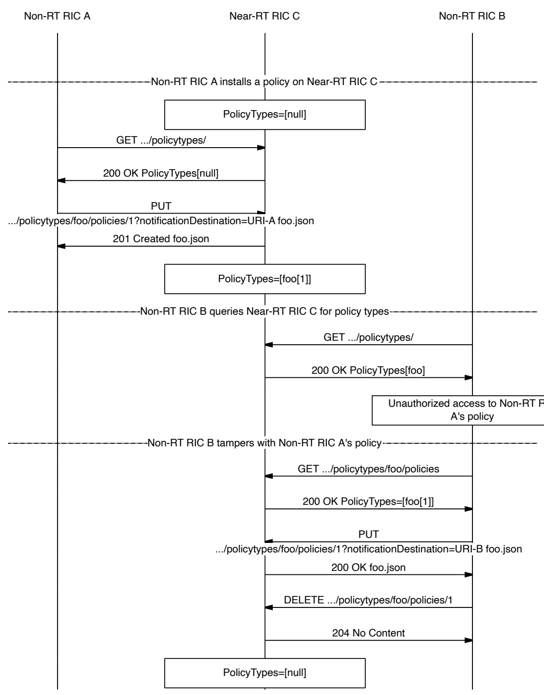
5.3.3 Policy Fuzzing. After mutating the keys and values for the `ORAN_TrafficSteering_Preference_2.0.0` policy inserted by the Rimedo labs traffic steering xApp, OAITT did not uncover any bugs. μONOS had checks in place to verify that all required keys were present and if the policy was specified as per the schema. In the cases where keys did not comply, we received an error message stating that the policy could not be enforced. Similarly, for the values fuzzed, μONOS rejected policies with values that did not meet the expected type or range.

5.3.4 URL Fuzzing. URL Fuzzing of μONOS implementation did not exploit any unknown URLs.

5.3.5 Policy Tampering. μONOS was vulnerable to a (malicious) Non-RT RIC being able to modify and delete a policy installed by another (benign) Non-RT RIC.

Figure 4 illustrates the message sequence pattern of how Non-RT RIC B can read the policies installed by Non-RT RIC A in the Near-RT RIC; and then delete/update Non-RT RIC A's policies without A being notified. Non-RT RIC A first installs a policy object `foo.json` and also sets `notificationDestination=URI-A` so that it gets notified of any changes to the installed policy. Next, not only is Non-RT RIC B able to query A's policies on Near-RT RIC C, but B is also able to first set A's policy `notificationDestination=URI-B` successfully and then delete the same policy without A being notified. Note that the `notificationDestination` is optional (see 5.2.4.8.1 in O-RAN.WG2.A1AP-R003-v04.00 [8]), therefore, unless set, B does not have to change the notification destination. Furthermore, the `notificationDestination` could be further exploited by Non-RT RIC B by overwhelming the Near-RT RIC with update requests on Non-RT RIC A's policies, resulting in notifications being sent to Non-RT RIC A by the Near-RT RIC.

5.3.6 Covert communication. Unauthorized access to policy information as described above can be further exploited by malicious Non-RT RICs or malicious rApps within the same Non-RT RIC to covertly communicate with each other as a timing channel or a storage channel.



Non-RT RIC A
Near-RT RIC C
Non-RT RIC B
Non-RT RIC A installs a policy on Near-RT RIC C
PolicyTypes=[null]
GET .../policytypes/
200 OK PolicyTypes[null]
PUT
.../policytypes/foo/policies/1?notificationDestination=URI-A foo.json
201 Created foo.json
PolicyTypes=[foo[1]]
Non-RT RIC B queries Near-RT RIC C for policy types
GET .../policytypes/
200 OK PolicyTypes[foo]
Unauthorized access to Non-RT RIC A's policy
Non-RT RIC B tampers with Non-RT RIC A's policy
GET .../policytypes/foo/policies
200 OK PolicyTypes=[foo[1]]
PUT
.../policytypes/foo/policies/1?notificationDestination=URI-B foo.json
200 OK foo.json
DELETE .../policytypes/foo/policies/1
204 No Content
PolicyTypes=[null]

Figure 4: Through the described message sequence a malicious Near-RT RIC B can alter or delete a previously deployed policy from Near-RT RIC A.

Timing channel. Assuming both Non-RT RICs have synchronized clocks, share a specific policy id and use rounds to send and receive data, we can have one way communication between a sender and a receiver using a timing channel. The sender Non-RT RIC connects to the Near-RT RIC and installs the pre-shared policy on the Near-RT RIC. Shortly after, the receiver connects to the Near-RT RIC and queries if the pre-shared policy has been installed or not. If it detects the pre-shared policy id has been installed, it has received a 1, if not, it is a 0. To end the round, the sender deletes the policy. In the next round the procedure is repeated to send the next bit of information. By modulating this pattern of installing a pre-shared policy or not and then deleting the pre-shared policy, the two Non-RT RICs can covertly communicate via the Near-RT RIC.
Storage channel. In the storage channel, the sender and receiver require synchronized clocks, a pre-shared policy id and use rounds to send/receive data. Furthermore, they agree upon a particular key-value pair in the policy which will be used for writing and reading data. The sender first connects to the Near-RT RIC and inserts a policy, the receiver then connects to the Near-RT RIC and reads out the policy and inspects the pre-shared key-value for the

covert data. To end the round, the sender deletes the policy. In the next round the procedure is repeated to send the next data string. In doing so, the two Non-RT RICS covertly communicate via the Near-RT RIC.

5.3.7 *Denial of service.*

The tests described earlier in this section can be broadly classified into Read (Incomplete URL, Entrypoint and Asymmetric) and Write (Faulty Policy and Correct Policy) tests. Figure 5a and 5b show the average response times for Read and Write requests respectively. We chose not to show the response rate measurements as they illustrate the same picture as the response time measurements.

With respect to the Read tests we have the following key results. The policy server on the Near-RT RIC responds with a 4xx error code for the Incomplete URL test and a 2xx response code for the other two tests, which are as expected. The response time for incomplete URLs is flat for increasing number of concurrent TCP connections whereas, the response time increases roughly linearly for the Entrypoint and Asymmetric tests with increasing number of concurrent TCP connections. The Incomplete URL test indicates that the Near-RT RIC barely requires any processing time whereas, the Near-RT RIC takes very long (10–60s) to respond to a growing number of concurrent connections for legitimate read requests, thus demonstrating a degradation of service.

With respect to the Write tests we have the following major results. When deploying a policy that does not match the policy scheme we get a 5xx error code whereas a 2xx response code when deploying a legitimate policy. The response time for the faulty policy is small indicating that the Near-RT RIC processing pipeline for such requests is fast and therefore can handle thousands of such requests efficiently. However, when deploying correct policies with merely a single TCP connection and 45 requests per second the Kubernetes A1 pod in μONOS crashes and does not restart on its own, resulting in a denial of service. We have reported our findings to the ONF and a fix is underway. The Kubernetes pod needs to be manually restarted and the previously installed policies need to be reinstalled.

6 RECOMMENDATIONS FOR THE O-RAN COMMUNITY

After conducting both static and dynamic analyses of the critical components within the O-RAN system, we offer security recommendations and mitigations to the O-RAN community, encompassing developers, vendors, and operators.

Security by Design. The initial development of O-RAN specifications in 2018 did not prioritize security by design [16], unlike the 3GPP for its 5G specifications development. Subsequently in 2022, the O-RAN Alliance issued specifications for securing the A1 and E2 interfaces. Given that OSC's first software was in November 2019, it is unsurprising that our static and dynamic Near-RT RIC testing found security features lacking. However, with security specifications already available for an ample amount of time, it is imperative for OSC developers to integrate these features into the mainstream code, e.g., including TLS for the A1 and IPsec for the E2.

Security Mechanisms for the A1 Interface. In the security evaluation of the Near-RT RIC and xApps [3] Solution #9, the working group states: "the Near-RT RIC should not assume that the received policies are trusted and valid and should therefore provide security built-in compliant with a zero-trust architecture ...". In our evaluation of the two open-source Near-RT RICs, we demonstrated that such mechanisms are absent. Therefore, we strongly encourage Near-RT RIC vendors and operators to consider mechanisms to prevent policy-related attacks by including security mechanisms in the Near-RT RIC in addition to OAuth, e.g., mapping policies to the Non-RT RIC identity via TLS certificates for instance.

Adopting the Shift Left Paradigm. In this paper our risk assessment revealed several security concerns that could have been resolved by adopting the shift left paradigm, e.g., including security testing earlier in the software development process. Furthermore, by adopting the shift left paradigm with Software Application Security Testing (SAST) to other O-RAN software components, using other tools, e.g., `oss-scorecard` [32], and integrating SBOMs (See Section 6.3 SBOM requirements for ORAN [4]), we can ensure some degree of secure O-RAN software for production environments.

Verification of Dependency Vulnerabilities. The vulnerabilities identified by the dependency analysis tools are indeed real, however, verifying the risks they pose to the Near-RT RICs was not verified in this paper. We view future work in this direction as identifying the dependencies that could be exploited, by first distinguishing between direct and transient dependencies. And then, identifying code paths that could lead to exploitation.

7 RELATED WORK

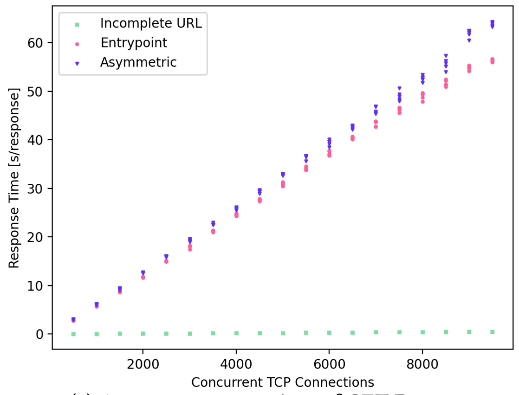
The O-RAN Alliance security working group regularly publishes their assessments of the threats, risks and mitigation strategies [6] posed to O-RAN. With respect to the Near-RT RIC and the A1 interface, the security working group have published security assessments of the Near-RT RIC [3] and the xApps; and have a section on security consideration for the A1 interface [1, 8] in the A1 transport protocol. The end-to-end testing [2] of O-RAN also describes optional security tests to evaluate the A1 interface.

Mimran et al. [29] conducted an elaborate security evaluation of the O-RAN architecture. The paper introduces an ontology specifically designed for evaluating O-RAN's architecture security risks. One of the key insights from the paper is that although the attack surfaces appears to have grown, the fact is that now the attack surface is more clear and cannot hide behind proprietary black box implementations.

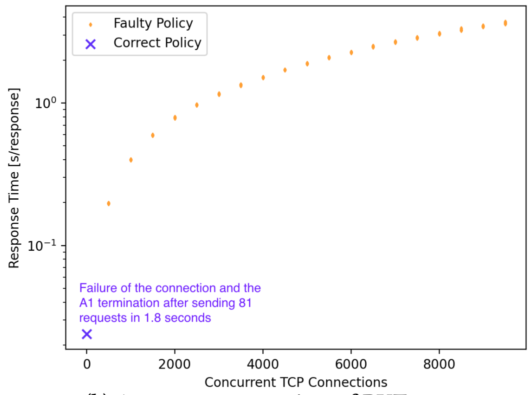
Shen et al. [35] describe the issue of authentication and authorization between the SMO and the Near-RT RIC via the O1 interface and recommend the use of a public key infrastructure to provide authentication and authorization.

Liyanage et al. [27] provide a broad and extensive security analysis of the O-RAN architecture and the associated risks. The findings described in the paper align with our perspective on supply chain security. However, with respect to the the Near-RT RIC, the paper primarily describes risks related to malicious xApps and UE privacy.

Haas et al. [19] proposed a hardware/operating-system based approach to addressing the risks posed by untrusted components in the O-RAN architecture. Their proposed solution relies on a microkernel approach and their evaluation illustrates enhanced security, energy efficiency, low latency, and scalability.



(a) Average response time of GET Requests.



(b) Average response time of PUT requests.

Figure 5: Response time measurements of the A1 policy server on μONOS for GET and PUT traffic patterns.

Dik et al. [14] describe attacks on the O-RAN Fronthaul interface, i.e., between the RU and DU. The use of plain-text Ethernet frames for Layer 2 communication makes the interface vulnerable to man-in-the-middle attacks and severely compromises the operation of the RAN. Their paper also describes MACsec, a secure architecture for the Open Fronthaul interface and proposes and evaluates three hardware architectures for MACsec.

Groen et al. [17] focus on the performance tradeoffs of using encryption (IPsec with ESP in tunnel mode using AES-256 and SHA-256) to secure the O-RAN E2 interface. The evaluation was conducted on the ColO-RAN [34] and used a minimal Near-RT RIC (based on the bronze release of the OSC Near-RT RIC used in this paper). The key takeaway from the paper was that the overhead of IPsec on the E2 interface was low in terms of latency (≤50μs).

Tiberti et al. [39] conducted a man-in-the-middle analysis of the A1 interface using the OSC Kubernetes default setup. The authors were able to poison the ARP entries on the Near-RT RIC and Non-RT RIC pods so that all A1 communication flowed via an attacker controlled pod. Subsequently, the authors observed the ability to tamper with policies on the Near-RT RIC as described in this paper.

8 CONCLUSION

In this paper, we evaluated the security posture of two open-source Near-RT RICs, OSC and μONOS, using static and dynamic application security testing. Our assumption that the O-RAN interfaces are avenues for attackers to exploit the Near-RT RIC was validated through our dependency analysis and configuration file analysis. The dependency analysis tools identified 211 and 285 potential dependency risks in μONOS and OSC respectively and highlighted the security risks associated with the newly introduced O-RAN interfaces namely, A1, E2 and O1. We also uncovered that micro-service infrastructure, networking, configuration and cryptographic dependencies contributed the most risk to the Near-RT RICs.

We conducted a thorough scan of various RIC configuration files using KICS and identified a core issue in OSC's non-compliance with the security by design approach. OSC code development lacks essential security practices from openAPI, Kubernetes, container, and Docker technologies. Specifically, the absence of standard security measures, such as access control and HTTPS for APIs, running containers as root, and the disclosure of secrets in public repositories, are prominent concerns in both the Near-RT RICs. These vulnerabilities strengthen an attacker's ability to execute attacks over the A1 and E2 interfaces.

Our security evaluation of the A1 API implementation uncovered multiple security concerns that require immediate attention. First, OSC and μONOS must add TLS to the A1 interface to meet the minimum security requirements specified by O-RAN. Second, the lack of authorization mechanisms in the Near-RT RIC for A1-related resources can lead to elevated privileges for a malicious Non-RT RIC or rApp. Third, we described how Non-RT RICs can covertly communicate by exploiting the A1 protocol and the lack of authorization of A1 policies in the Near-RT RIC. Fourth, we demonstrated how a malicious Non-RT RIC (or even an rApp) could degrade the performance of the A1 interface and in a worst case cause a denial of service attack by issuing low rate PUT requests.

Based on the above results, we recommend strong operational security for telecommunication operators looking to deploy O-RAN. To reduce the risks associated with vulnerable dependencies and security misconfigurations operators should consider the shift-left paradigm, i.e., integrating security by design and security testing earlier into the software development lifecycle. To ensure confidentiality, authentication and authorization, TLS and OAuth must be implemented and configured on the A1 interface. Furthermore, security mechanisms to prevent attacks arising from tampering with A1 policies or enrichment information should be setup in the Near-RT RIC. And finally, Near-RT RICs require serious security and performance testing before deployment to prevent trivial denial of service attacks from compromising the availability of an O-RAN setup.

ACKNOWLEDGEMENTS

The authors thank the anonymous reviewers for their valuable feedback. This research was funded by the German Federal Ministry for Digital and Transport (BMDV) within the O-RAN Lab Grant 19OL22004C, German Federal Ministry of Education and Research (BMBF) within the 6G Research and Innovation Cluster 6G-RIC Grant 16KISK020K and the Einstein Foundation (Einstein Research Unit on Quantum Devices). The authors express their thanks towards Vincent Ulitzsch, i14y Lab, Petr Jurčík, Andreas Gladisch, Markus Wengel, Ole Reinartz and the μONOS team.

REFERENCES

[1] ORAN Alliance. 2022. O-RAN.SFG.Non-RT-RIC-Security-TR-v01.00. *ORAN Documentation* (March 2022).

[2] ORAN Alliance. 2022. O-RAN.TIFG.E2E-Test.0-v04.00. *ORAN Documentation* (October 2022).

[3] ORAN Alliance. 2023. O-RAN.WG11.Security-Near-RT-RIC-xApps-TR.0-R003-v03. *ORAN Documentation* (June 2023).

[4] ORAN Alliance. 2023. O-RAN.WG11.Security-Requirements-Specification.O-R003-v06.00. *ORAN Documentation* (June 2023).

[5] ORAN Alliance. 2023. O-RAN.WG11.Security-Test-Specifications.O-R003-v04.00. *ORAN Documentation* (June 2023).

[6] ORAN Alliance. 2023. O-RAN.WG11.Threat-Model.O-R003-v06.00. *ORAN Documentation* (June 2023).

[7] ORAN Alliance. 2023. O-RAN.WG1.O-RAN-Architecture-Description-v09.00. *ORAN Documentation* (June 2023).

[8] ORAN Alliance. 2023. O-RAN.WG2.A1AP-R003-v04.00. *ORAN Documentation* (March 2023).

[9] ORAN Alliance. 2023. O-RAN.WG2.A1GAP-R003-v03.01. *ORAN Documentation* (March 2023).

[10] ORAN Alliance. 2023. O-RAN.WG2.A1TP-R003-v02.01. *ORAN Documentation* (March 2023).

[11] ORAN Alliance. 2023. O-RAN.WG3.RICARCH-R003-v04.00. *ORAN Documentation* (March 2023).

[12] Anchore. 2023. Grype: A Vulnerability Scanner for Container Images and Filesystems. https://github.com/anchore/grype. Accessed: September 27, 2023.

[13] Airhop Communications. 2023. *AirHop Launches the Industry's First Comprehensive Portfolio of Field-proven xApps and rApps to Accelerate 4G and 5G Open RAN Deployments.* https://www.airhopcomm.com/news/airhop-launches-the-industrys-first-comprehensive-portfolio-of-field-proven-xapps-and-rapps-to-accelerate-4g-and-5g-open-ran-deployments/ Accessed: 21-08-2023.

[14] Daniel Dik and Michael Stübert Berger. 2023. Open-RAN Fronthaul Transport Security Architecture and Implementation. *IEEE Access* (2023).

[15] drwetter. 2024. testssl.sh. https://github.com/drwetter/testssl.sh. Accessed: 26-01-2024.

[16] Stefan Köpsell et al. 2022. *Open RAN Risk Analysis.* https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/5G/5GRAN-Risk-Analysis.pdf?__blob=publicationFile&v=7

[17] Joshua Groen, Brian Kim, and Kaushik Chowdhury. 2023. The Cost of Securing O-RAN. In *ICC 2023-IEEE International Conference on Communications*. IEEE, 5444–5449.

[18] Ceki Gülcü. 2003. *The complete log4j manual.* QOS. ch.

[19] Sebastian Haas, Mattis Hasler, Friedrich Pauls, Stefan Köpsell, Nils Asmussen, Michael Roitzsch, and Gerhard Fettweis. 2022. Trustworthy Computing for O-RAN: Security in a Latency-Sensitive Environment. In *2022 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 826–831.

[20] KICS. 2024. KICS. https://kics.io. Accessed: 26-01-2024.

[21] Felix Klement, Stefan Katzenbeisser, Vincent Ulitzsch, Juliane Krämer, Slawomir Stanczak, Zoran Utkovski, Igor Bjelakovic, and Gerhard Wunder. 2022. Open or not open: Are conventional radio access networks more secure and trustworthy than Open-RAN? *arXiv preprint arXiv:2204.12227* (2022).

[22] Stefan Köpsell, Andrey Ruzhanskiy, Andreas Hecker, Dirk Stachorra, and Norman Franchi. 2022. Open RAN Risk Analysis. *Federal Office for Information Security, BSI studies* (2022).

[23] Robert Krösche, Kashyap Thimmaraju, Liron Schiff, and Stefan Schmid. 2018. I DPID it my way! A covert timing channel in software-defined networks. In *2018 IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE, 217–225.

[24] Kubernetes. 2023. *Configure a Security Context for a Pod or Container.* https://kubernetes.io/docs/tasks/configure-pod-container/security-context/

[25] Kubernetes. 2023. *Secrets.* https://kubernetes.io/docs/concepts/configuration/secret/

[26] Leeon123. 2024. Stress-tester. https://github.com/Leeon123/Stress-tester. Accessed: 26-01-2024.

[27] Madhusanka Liyanage, An Braeken, Shahriar Shahabuddin, and Pasika Ranaweera. 2023. Open RAN security: Challenges and opportunities. *Journal of Network and Computer Applications* 214 (2023), 103621.

[28] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review* 38, 2 (2008), 69–74.

[29] Dudu Mimran, Ron Bitton, Yehonatan Kfir, Eitan Klevansky, Oleg Brodt, Heiko Lehmann, Yuval Elovici, and Asaf Shabtai. 2022. Evaluating the security of open radio access networks. *arXiv preprint arXiv:2201.06080* (2022).

[30] onosproject. 2022. *Github Rimedo Traffic Steering xApp.* https://github.com/onosproject/rimedo-ts Accessed: 14-08-2023.

[31] onosprojects. 2023. *Installation with RAN-Simulator and RIMEDO Labs Traffic Steering xAPP (rimedo-ts xApp).* https://github.com/onosproject/sdran-in-a-box/blob/master/docs/Installation_RANSim_RIMDEO_TS.md Accessed: 30-08-2023.

[32] OpenSSF. 2024. OpenSSF Scorecard. https://github.com/ossf/scorecard. Accessed: 26-01-2024.

[33] OSC. 2022. *Release F.* https://wiki.o-ran-sc.org/display/RICP/2022-05-24+Release+F Accessed: 25-08-2023.

[34] Michele Polese, Leonardo Bonati, Salvatore D'Oro, Stefano Basagni, and Tommaso Melodia. 2022. ColO-RAN: Developing Machine Learning-based xApps for Open RAN Closed-loop Control on Programmable Experimental Platforms. *IEEE Transactions on Mobile Computing* (July 2022), 1–14.

[35] CT Shen, YY Xiao, YW Ma, JL Chen, Cheng-Mou Chiang, SJ Chen, and YC Pan. 2022. Security threat analysis and treatment strategy for ORAN. In *2022 24th International Conference on Advanced Communication Technology (ICACT)*. IEEE, 417–422.

[36] Snyk. [n. d.]. *Guide to Software Composition Analysis (SCA).*

[37] Kashyap Thimmaraju, Liron Schiff, and Stefan Schmid. 2017. Outsmarting network security with SDN teleportation. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 563–578.

[38] Kashyap Thimmaraju, Bhargava Shastry, Tobias Fiebig, Felicitas Hetzelt, Jean-Pierre Seifert, Anja Feldmann, and Stefan Schmid. 2018. Taking control of sdn-based cloud systems via the data plane. In *Proceedings of the Symposium on SDN Research*. 1–15.

[39] Walter Tiberti, Eleonora Di Fina, Andrea Marotta, and Dajana Cassioli. 2022. Impact of Man-in-the-Middle Attacks to the O-RAN Inter-Controllers Interface. In *2022 IEEE Future Networks World Forum (FNWF)*. 367–372. https://doi.org/10.1109/FNWF55208.2022.00071

[40] Trivy. 2024. Trivy. https://github.com/aquasecurity/trivy. Accessed: 26-01-2024.

[41] Wikipedia. 2023. *Heartbleed.* https://github.com/onosproject/sdran-in-a-box Accessed: 25-08-2023.

A RESULTS FROM DEPENDENCY ANALYSIS AND CONFIGURATION FILE ANALYSIS

The detailed results from Grype, Snyk, Trivy and KICS can be obtained online at the following URL: https://git.tu-berlin.de/hashkash1000/wisec-2024

B O-RAN A1 INTERFACE TESTING TOOL

The source code for OAITT can be downloaded at the following URL: https://git.tu-berlin.de/hashkash1000/wisec-2024