Source Code Audit on CRI-O Runtime for the OSTIF

Final Report and Management Summary

2025-12-03

X41 D-Sec GmbH
Soerser Weg 20
D-52070 Aachen
Amtsgericht Aachen: HRB19989

`https://x41-dsec.de/`
`info@x41-dsec.de`

Organized by the Open Source Technology Fund

Revision	*Date*	*Change*	*Author(s)*
1	2025-11-10	Draft Report and Management Summary	Alexander Schloegl, Christian Mayr and Hannes Moesl-Canaval
2	2025-11-21	Final Draft Report	Hannes Moesl-Canaval
3	2025-11-21	Final Report	Eric Sesterhenn and Yasar Kla-wohn
4	2025-12-03	Public Report	Eric Sesterhenn and Yasar Kla-wohn

Contents

1 License **4**

2 Executive Summary **5**

3 Introduction **7**
- 3.1 Methodology 8
- 3.2 Scope 8
- 3.3 Coverage 9
- 3.4 Recommended Further Tests 11

4 Rating Methodology **12**
- 4.1 CVSS 12
- 4.2 Severity Mapping 15
- 4.3 Common Weakness Enumeration 15

5 Results **16**
- 5.1 Findings 16
- 5.2 Informational Notes 16

6 About OSTIF **21**

7 About X41 D-Sec GmbH **22**

Dashboard

Target	
Customer	OSTIF
Name	CRI-O Runtime
Type	Runtime
Version	301eb72ed2cad2feac49631aee778be757b78b31
Engagement	
Type	Source Code Audit
Consultants	3: Alexander Schloegl, Christian Mayr and Hannes Moesl-Canaval
Engagement Effort	20 person-days, 2025-10-27 to 2025-11-10
Total issues found	0

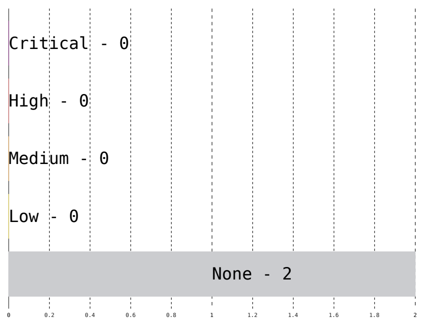

Figure 1: Issue Overview

1 License

2 Executive Summary

In October and November 2025, X41 D-Sec GmbH performed a Source Code Audit against the CRI-O Runtime OCI-based container runtime. During the audit, X41 identified no vulnerabilities. Instead, two information findings were spotted where one concerns the use of outdated dependencies that include known vulnerabilities. While it could not be conclusively verified whether these issues are directly exploitable through *CRI-O*, the fact that the affected component involves *runc* – a critical dependency in the container runtime stack – warrants prompt remediation. Updating the dependency to the latest upstream version is recommended to mitigate potential exposure and ensure consistency with current security patches. X41 additionally recommends establishing a robust update process for dependencies to ensure relevant security updates are identified and applied in a timely fashion.

The second informational finding pertains to an input validation weakness in the sandbox configuration of CRI-O. While this issue does not present a direct exploitation path, addressing it is recommended as a defense-in-depth measure to improve the robustness and reliability of container runtime isolation.

X41 conducted a comprehensive source code review against the CRI-O Runtime OCI-based container runtime.

The test was performed by three experienced senior security experts between 2025-10-27 and 2025-11-10.

Overall, the quality of the code base is outstanding. The functionality exposed through the APIs is implemented in a deliberately minimalistic fashion, effectively minimizing the potential attack surface. The use of Go's native constructs for structured parameter handling and safe data (un)marshalling reflects a high level of proficiency in secure coding practices within the Go ecosystem. This provides X41 with the impression of a robust, intentionally engineered code base that demonstrates a clear and consistent security-oriented design philosophy.

During the audit, it was observed that input validation generally follows a narrowly scoped approach – values are primarily checked to prevent runtime errors or panics within the *CRI-O* code base itself. While efficient, this approach can introduce risks in scenarios involving untrusted data flowing through multiple integrated components. In such cases, limited validation may allow malformed or malicious input to propagate into dependent systems, increasing the likelihood of indirect vulnerabilities.

Nevertheless, given the open-source nature of *CRI-O* and its surrounding ecosystem, responsibility for robust input validation can be shared and improved collaboratively across related projects. Enhancing input verification mechanisms – whether within *CRI-O* or its upstream dependencies – would contribute significantly to overall system resilience. In conclusion, X41 assesses the security posture of *CRI-O* as well-designed and effectively executed, striking a sound balance between minimalism and practical robustness.

3 Introduction

X41 conducted a security review of the *CRI-O* runtime – an OCI-compliant implementation of the Kubernetes Container Runtime Interface (CRI) responsible for creating, running, and supervising container workloads on Kubernetes nodes. At a functional level, *CRI-O* provides the runtime plumbing between the Kubernetes control plane and OCI-format container images: it pulls and verifies images from registries, instantiates containers according to pod specifications, sets up namespaces, cgroups and networking for each container, enforces configured resource limits and security options, and integrates with image and registry signing/verification mechanisms.

Compromise of the container runtime or its misconfiguration allows attackers to bypass intended isolation and access cluster resources. A successful compromise can enable attackers to: extract secrets used by applications or the control plane, deploy persistent malicious containers, manipulate workloads or cluster state, and move laterally to other nodes or control-plane components. Given *CRI-O*'s privileged position between Kubernetes and the host, weakness at this layer carries high potential for privilege escalation and broad impact to confidentiality, integrity, and availability.

For example, an attacker who gains code execution inside a container could attempt a container escape by exploiting weaknesses in isolation or misconfiguration. A plausible non-exhaustive scenario is a workload started with excessive privileges (for instance, a privileged container, host PID[1] namespace, or host file system mounts) or with access to host control sockets; from such a foothold an attacker could interact with host resources, access host process information, or manipulate the container runtime to start additional containers with elevated capabilities. Alternatively, an attacker could target a vulnerability in the runtime itself (or its use of kernel features) to break namespace or cgroup isolation and achieve access to the underlying Kubernetes node. Any of these outcomes would permit access to host-level secrets, persistence mechanisms, or the ability to disrupt or propagate across the cluster.

The interface between workloads and the runtime could potentially be exploited for DoS[2] at-

[1] Process ID
[2] Denial of Service

tacks. Data that originates from an unprivileged container, such as debug info, program logs, and return values, can be used to crash the runtime – and thus also other workloads – if not properly sanitized and length checked. Such an attack has been demonstrated in a previous audit with a bug tracked as CVE-2022-1708[3][4].

This assessment concentrated on identifying weaknesses in configuration, privilege management, and implementation within *CRI-O* that could lead to breaches of confidentiality, integrity, or availability as previously described. It further aimed to provide recommendations to strengthen the runtime deployment and minimize the potential impact, or blast radius, of a container compromise.

3.1 Methodology

X41 performed a security code audit of the source code provided via git.

The assessment was conducted as a comprehensive source code review of the public repository available at `https://github.com/cri-o/cri-o`.

3.2 Scope

Prior to the commencement of the project, a kick-off meeting was conducted to establish the objectives of the assessment. During this session, the overall architecture of the *CRI-O* environment was presented to the testing team. This included an overview of the *CRI-O* runtime architecture itself, as well as its interactions with other core components such as *runc*.

Additionally, the security objectives of *CRI-O* were discussed. It was communicated to X41 that, while *CRI-O* does not provide explicit security *guarantees*, its fundamental design intent is that an unprivileged container should neither influence nor obtain information about other containers or the host system.

Not all functionalities are handled by *CRI-O* itself, which informs the scope of the ongoing assessment. The following aspects were discussed as being relevant to security and should be considered part of the attack surface:

- image downloading and verification
- container image management
- container lifecycle management

[3] `https://nvd.nist.gov/vuln/detail/CVE-2022-1708`
[4] `https://nvd.nist.gov/vuln/detail/cve-2022-1708`

- resource isolation as required by the CRI[5]

With this scope in mind, the security review was performed on the following repository:

- `https://github.com/cri-o/cri-o`
  - Commit: 301eb72ed2cad2feac49631aee778be757b78b31 (Oct 24, 2025)

The *CRI-O* project is implemented in Go and consists of approximately $60{,}000$ lines of code, excluding its external dependencies.

3.3 Coverage

A security assessment attempts to find as many of the existing problems as possible, though it is practically never possible to rule out the likelihood of additional weaknesses being found in the future. Further, where applicable, suggestions of more resilient design patterns are given.

The time allocated to X41 for this assessment was sufficient to yield a good coverage of the given scope.

The following paragraphs outline the security assessment procedures and corresponding tests that were conducted:

1. Package dependencies were examined for publicly disclosed vulnerabilities.
2. In addition to the manual source code review, multiple state-of-the-art static analysis tools were employed, including semgrep[6].
3. Particular attention was paid to the CRI-O sandbox implementation, with a focus on the role of the infra container in establishing and maintaining pod-level namespaces. The analysis included the configuration and behavior of user namespaces, the handling of shared memory segments between the host and containers, and the network and DNS[7] configurations managed through CNI[8] plugins. Furthermore, the seccomp profile configuration and enforcement for unprivileged containers were assessed to determine compliance with CRI-O's default security policies and to identify deviations introduced by custom runtime or container engine configurations.
4. The enforcement of Linux capabilities within CRI-O was reviewed, as excessive or misconfigured capabilities can considerably elevate the risk of container escape or host privilege

[5] Container Runtime Interface
[6] `https://github.com/semgrep/semgrep`
[7] Domain Name System
[8] Container Network Interface

escalation. The review examined CRI-O's management of default and runtime-assigned capability sets, differentiating between allowed, additive, and dropped capabilities as defined in the container runtime configuration and pod security context. Special focus was placed on CRI-O's interaction with the OCI[9] runtime to enforce these restrictions and on any deviations from Kubernetes' default capability profiles or PodSecurity standards that could expose the host to elevated privilege operations.

5. The state of fuzzing integration within CRI-O was evaluated. It was confirmed that fuzzing is actively performed under the CNCF's OSS-Fuzz initiative, primarily maintained by Ada-Logics. However, the current fuzzing targets lack dedicated or manually curated seed corpora, limiting initial input diversity and slowing coverage expansion during early fuzzing runs. Although OSS-Fuzz automatically preserves and reuses crash-inducing inputs and evolved corpora over time, the absence of representative seed corpora reduces the efficiency of localized or short-term fuzzing efforts. Incorporating realistic configuration files, CRI request payloads, and serialized runtime objects as initial corpora would significantly improve fuzzing depth and coverage efficiency.

6. The CRI server implementation was tested for potential DoS vectors resembling CVE-2022-1708[10]. This included evaluating all data originating from unprivileged containers that is processed within the runtime environment. The original CVE[11] exploited an unchecked logging mechanism in the ExecSync endpoint to exhaust runtime memory. During this test, similar attack surfaces – such as writing to the termination-log file or altering container status responses – were assessed. Exploitation was deemed infeasible, as all relevant read operations were properly bounded by length.

7. Additional potential DoS vectors via CRI server requests were also investigated but were found to be unexploitable.

8. The process of container sandbox creation underwent detailed manual code review to ensure clean and correct value handling, eliminating opportunities for injection or parameter confusion. The parsing and validation of configuration parameters were examined thoroughly. It was observed that CRI-O performs only minimal validation of parsed values, placing reliance on downstream components to defend against malformed or malicious inputs.

9. As CRI-O supports user and group ID mapping limitations, these mechanisms were analyzed for robustness. No injection, overflow, underflow, or TOCTOU[12] vulnerabilities were identified.

10. Attempts to escalate sandbox privileges via additional container annotations were successfully blocked through a combination of container specification validation and implemented

[9] Open Container Initiative
[10] `https://nvd.nist.gov/vuln/detail/CVE-2022-1708`
[11] Common Vulnerabilities and Exposures
[12] Time of Check Time of Use

filtering logic. During this evaluation, X41 revisited the previously disclosed CVE-2022-0811[13][14], given the continued use of the *pinns* utility. It was determined that the current approach – constructing individual key/value pairs for each *sysctl* parameter and passing them to *pinns* using Go's *exec.Command* functionality – provides significantly stronger resilience against malicious input compared to the former implementation.

11. In addition to container creation, the restoration process from disk was assessed for potential sandbox escape or privilege escalation vectors. The CRI-O runtime reads values directly from stored configuration files, such as the CNI namespace, without validating them for correctness, which could potentially introduce issues under certain circumstances.
12. The image downloading and verification processes were reviewed for proper validation and strict error handling. CRI-O leverages functionality from the *containers/image* package, resulting in a relatively lightweight internal implementation. Security and integrity assurances for pulled images are therefore delegated to this upstream package.

X41 recommends promptly updating the *runc* dependency to the latest secure version and implementing additional validation checks to prevent the restoration of invalid sandbox configurations from disk.

Suggestions for next steps in securing this scope can be found in section 3.4.

3.4 Recommended Further Tests

As noted earlier, the current fuzzing effort for *CRI-O* operates without component-specific seed corpora, resulting in limited initial input diversity and reduced coverage growth during early iterations. The absence of well-structured seeds forces the fuzzers to rely heavily on mutation from trivial or autogenerated inputs, which slows discovery of deeper code paths and increases overall time to meaningful coverage. Introducing targeted, high-quality seed corpora, derived from real-world workloads, substantially improves exploration depth, accelerates coverage expansion, and reduces the likelihood of long plateaus during the fuzzing campaign.

[13] `https://nvd.nist.gov/vuln/detail/CVE-2022-0811`
[14] `https://www.crowdstrike.com/en-us/blog/cr8escape-new-vulnerability-discovered-in-cri-o-container-engine-cve-2022-0811/`

4 Rating Methodology

Security vulnerabilities are given a purely technical rating by the testers when they are discovered during a test. Business factors and financial risks for OSTIF are beyond the scope of a penetration test, which focuses entirely on technical factors. However, technical results from a penetration test may be an integral part of a general risk assessment. A penetration test is based on a limited time frame and only covers vulnerabilities and security issues which have been found in the given time, there is no claim for full coverage.

The CVSS[1] is used to score all findings relevant to security. The resulting CVSS score is mapped to qualitative ratings as shown below.

4.1 CVSS

Testers rate all security-relevant findings using the CVSS industry standard version 4.

Vulnerabilities scored with CVSS get a numeric value based on several metrics ranging from 0.0 (least worst) to 10.0 (worst).

The score captures different factors that express the impact and the ease of exploitation of a vulnerability among other factors. For a detailed description of how the scores are calculated, please see the CVSS version 4.0 specification.[2]

The metrics used to calculate the final score are grouped into three different categories.

[1]Common Vulnerability Scoring System
[2]`https://www.first.org/cvss/v4.0/specification-document`

The *Base Metric Group* represents the intrinsic and fundamental characteristics of a vulnerability that are constant over time and user environments. It captures the following metrics:

- Attack Vector (AV)
- Attack Complexity (AC)
- Attack Requirements (AT)
- Privileges Required (PR)
- User Interaction (UI)
- Vulnerable/Subsequent System Confidentiality (VC/SC)
- Vulnerable/Subsequent System Integrity (VI/SI)
- Vulnerable/Subsequent System Availability (VA/SA)

The *Threat Metric Group* represents the current state of exploit techniques and the availability of proof of concepts. It captures the following metric:

- Exploit Maturity (E)

The *Environmental Metric Group* represents the characteristics of a vulnerability that are relevant and unique to a particular user's environment. It includes the following metrics:

- Confidentiality Requirement (CR)
- Integrity Requirement (IR)
- Availability Requirement (AR)
- Modified Attack Vector (MAV)
- Modified Attack Complexity (MAC)
- Modified Attack Requirements (MAC)
- Modified Privileges Required (MPR)
- Modified User Interaction (MUI)
- Modified Vulnerable System Confidentiality (MVC)
- Modified Vulnerable System Integrity (MVI)
- Modified Vulnerable System Availability (MVA)
- Modified Subsequent System Confidentiality (MSC)
- Modified Subsequent System Integrity (MSI)
- Modified Subsequent System Availability (MSA)

A CVSS vector defines a specific set of metrics and their values, and it can be used to reproduce and assess a given score. It is rendered as a string that exactly reproduces a score.

For example, the vector `CVSS:4.0/AV:N/AC:H/AT:N/PR:L/UI:A/VC:H/VI:L/VA:N/SC:N/SI:N/SA:N` defines a base score metric with the following parameters:

- Attack Vector: Network
- Attack Complexity: High
- Attack Requirements: None
- Privileges Required: Low
- User Interaction: Active
- Vulnerable System Confidentiality: High
- Vulnerable System Integrity: Low
- Vulnerable System Availability: None
- Subsequent System Confidentiality: None
- Subsequent System Integrity: None
- Subsequent System Availability: None

In this example, a network-based attacker performs a complex attack after gaining access to some privileges, by tricking a user into performing some actions. This allows the attacker to read confidential data and change some parts of that data.

The detailed scores are the following:

Metric	Score
Exploitability	Medium
Complexity	Medium
Vulnerable system	Medium
Subsequent system	Low
Exploitation	High
CVSS Score	5.8 (Medium)

CVSS vectors can be automatically parsed to recreate the score, for example, with the CVSS calculator provided by FIRST, the organization behind CVSS: `https://www.first.org/cvss/calculator/4.0#CVSS:4.0/AV:N/AC:H/AT:N/PR:L/UI:A/VC:H/VI:L/VA:N/SC:N/SI:N/SA:N`.

4.2 Severity Mapping

To help in understanding the results of a test, numeric CVSS scores are mapped to qualitative ratings as follows:

Severity Rating	CVSS Score
NONE	0.0
LOW	0.1–3.9
MEDIUM	4.0–6.9
HIGH	7.0–8.9
CRITICAL	9.0–10.0

4.3 Common Weakness Enumeration

The CWE[3] is a set of software weaknesses that allows vulnerabilities and weaknesses in software to be categorized. If applicable, X41 gives a CWE ID for each vulnerability that is discovered during a test.

CWE is a very powerful method for categorizing a vulnerability. It gives general descriptions and solution advice on recurring vulnerability types. CWE is developed by *MITRE*.[4] More information can be found on the CWE site at `https://cwe.mitre.org/`.

[3] Common Weakness Enumeration
[4] `https://www.mitre.org`

5 Results

This chapter describes the results of this test.

5.1 Findings

No observation with a security impact were made during the audit.

5.2 Informational Notes

The following observations do not have a direct security impact, but are related to security hardening, affect functionality, or other topics that are not directly related to security. X41 recommends to mitigate these issues as well, because they often become exploitable in the future. Doing so will strengthen the security of the system and is recommended for defense in depth.

5.2.1 CRIO-CR-25-100: Incomplete config validation on sandbox restoration

Component:	source/cri-o/internal/lib/container_server.go:295

5.2.1.1 Description

While auditing sandbox creation and restoration, X41 discovered that while it is ensured a sandbox's namespace is not empty during creation, the same validation is lacking for restoration. Restoring a sandbox with an empty namespace would bypass namespace-specific signature policies and could also bypass namespace-specific network policies.

Modifying the namespace via its config file requires access to the control plane, which would allow much more serious attacks. This means the direct security impact is negligible, as the configuration value is handled cleanly during file creation. However, in the interest of defensive programming, additional config validation should occur during sandbox restoration to ensure no vulnerable sandboxes are deployed in case of corrupt config files.

Listing 5.1 shows the sandbox restoration process, and how annotations are directly loaded from the unmarshalled JSON[1]. Some intermediate code is excluded for layout reasons, but no verification happens in the removed code.

```go
// LoadSandbox loads a sandbox from the disk into the sandbox store.
func (c *ContainerServer) LoadSandbox(ctx context.Context, id string) (sb *sandbox.Sandbox,
↪ retErr error) {
	ctx, span := log.StartSpan(ctx)
	defer span.End()

	config, err := c.store.FromContainerDirectory(id, "config.json")
	if err != nil {
		return nil, err
	}

	var m rspec.Spec
	if err := json.Unmarshal(config, &m); err != nil {
		return nil, fmt.Errorf("error unmarshalling sandbox spec: %w", err)
	}

	labels := make(map[string]string)
	if err := json.Unmarshal([]byte(m.Annotations[annotations.Labels]), &labels); err != nil {
		return nil, fmt.Errorf("error unmarshalling %s annotation: %w", annotations.Labels, err)
	}

	// ...

	sbox.SetLogDir(filepath.Dir(m.Annotations[annotations.LogPath]))
	sbox.SetContainers(memorystore.New[*oci.Container]())
	sbox.SetShmPath(m.Annotations[annotations.ShmPath])
	sbox.SetCgroupParent(m.Annotations[annotations.CgroupParent])
	sbox.SetPrivileged(privileged)
	sbox.SetRuntimeHandler(m.Annotations[annotations.RuntimeHandler])
	sbox.SetResolvPath(m.Annotations[annotations.ResolvPath])
	sbox.SetHostname(m.Annotations[annotations.HostName])
	sbox.SetPortMappings(portMappings)
	sbox.SetHostNetwork(hostNetwork)
	sbox.SetUsernsMode(m.Annotations[annotations.UsernsModeAnnotation])
	sbox.SetPodLinuxOverhead(&podLinuxOverhead)
	sbox.SetPodLinuxResources(&podLinuxResources)
	sbox.SetHostnamePath(m.Annotations[annotations.HostnamePath])
	sbox.SetNamespaceOptions(&nsOpts)
```

[1] JavaScript Object Notation

```
    sbox.SetSeccompProfilePath(spp)
    sbox.SetCreatedAt(created)
    sbox.SetNamespace(m.Annotations[annotations.Namespace])
    sbox.SetKubeName(m.Annotations[annotations.KubeName])
```

Listing 5.1: Sandbox Loading and Lack of Namespace Validation

5.2.1.2 Solution Advice

X41 recommends adding verification logic to the sandbox restoration process, to ensure configuration values have not been corrupted in a way that would leave a sandbox insecure.

5.2.2 CRIO-CR-25-101: Outdated and vulnerable dependencies

Component:	source/cri-o/go.mod

5.2.2.1 Description

During the security assessment, it was identified that the *CRI-O* runtime relies on the *runc* container runtime. However, the deployed version of *runc* was determined to be outdated and exposed to several known security vulnerabilities. The version details referenced here reflect the state observed at the time of testing. Given that *runc* serves as a foundational component of container execution and directly underpins *CRI-O*, any vulnerabilities within this dependency pose a significant and immediate threat to the overall security posture of the containerized environment. Successful exploitation could enable attackers to break out of container isolation, execute arbitrary code on the host, or compromise container workloads. Therefore, ensuring that the *runc* runtime remains securely configured and regularly updated is essential to preserving the integrity, isolation, and trustworthiness of the container infrastructure.

Affected Component	CVE	CVSS Rating
github.com/containerd/containerd@v1.7.28	*CVE-2024-25621*	HIGH
github.com/containerd/containerd@v1.7.28	*CVE-2025-64329*	MEDIUM
github.com/opencontainers/runc@v1.3.2	*CVE-2025-31133*	HIGH
github.com/opencontainers/runc@v1.3.2	*CVE-2025-52565*	HIGH
github.com/opencontainers/runc@v1.3.2	*CVE-2025-52881*	HIGH
github.com/opencontainers/selinux@v1.12.0	*CVE-2025-52881*	HIGH

Notably, the testing team was unable to comprehensively prove any potential impact during the limited time frame granted for this review. As such, the wider implications remain unknown at this point and should be subjected to internal research at the earliest possible convenience for the in-house team.

It must also be noted that, at the time this report was written, no fix was available for the library *github.com/opencontainers/selinux*.

Generally speaking, the provision of robust supply chain security can be considerably challenging to provide to an optimal standard. Oftentimes, an easy or comprehensive solution cannot be offered, while the results and efficacy of the selected protection framework can vary depending on the integrated version of the deployed libraries.

5.2.2.2 Solution Advice

To mitigate the identified issues effectively, X41 recommends upgrading the *CRI-O* runtime and all associated dependencies, particularly *runc*, to the latest stable versions available from trusted upstream sources. Furthermore, it is advised to establish a structured update and verification policy within the container orchestration environment to ensure that *CRI-O* and its underlying components remain current. Implementing such a process will help ensure that the deployment benefits from timely security patches, reducing exposure to known vulnerabilities and strengthening the overall resilience of the container runtime stack.

6 About OSTIF

The Open Source Technology Improvement Fund (OSTIF) is dedicated to resourcing and managing security engagements for open source software projects through partnerships with corporate, government, and non-profit donors. We bridge the gap between resources and security outcomes, while supporting and championing the open source community whose efforts underpin our digital landscape.

Over the past ten years, OSTIF has been responsible for the discovery of over 800 vulnerabilities, (121 of those being Critical/High), over 13,000 hours of security work, and millions of dollars raised for open source security. Maximizing output and security outcomes while minimizing labor and cost for projects and funders has resulted in partnerships with multi-billion dollar companies, top open source foundations, government organizations, and respected individuals in the space. Most importantly, we've helped over 120 projects and counting improve their security posture.

Our directive is to support and enrich the open source community through providing public-facing security audits, educational resources, meetups, tooling, and advice. OSTIF's experience positions us to be able to share knowledge of auditing with maintainers, developers, foundations, and the community to further secure our infrastructure in a sustainable manner.

We are a small team working out of Chicago, Illinois. Our website is `ostif.org`. You can follow us on social media to keep up to date on audits, conferences, meetups, and opportunities with OSTIF, or feel free to reach out directly at contact@ostif.org or our GitHub.

Derek Zimmer, Executive Director
Amir Montazery, Managing Director
Helen Woeste, Communications and Community Manager
Tom Welter, Project Manager

7 About X41 D-Sec GmbH

X41 D-Sec GmbH is an expert provider for application security and penetration testing services. Having extensive industry experience and expertise in the area of information security, a strong core security team of world-class security experts enables X41 D-Sec GmbH to perform premium security services.

X41 has the following references that show their experience in the field:

- Source code audit of ISC BIND9 DNS server[1]
- Source code audit of the Git source code version control system[2]
- Review of the Mozilla Firefox updater[3]
- X41 Browser Security White Paper[4]
- Review of Cryptographic Protocols (Wire)[5]
- Identification of flaws in Fax Machines[6,7]
- Smartcard Stack Fuzzing[8]

The testers at X41 have extensive experience with penetration testing and red teaming exercises in complex environments. This includes enterprise environments with thousands of users and vendor infrastructures such as the Mozilla Firefox Updater (Balrog).

Fields of expertise in the area of application security encompass security-centered code reviews, binary reverse-engineering and vulnerability-discovery. Custom research and IT security consulting, as well as support services, are the core competencies of X41. The team has a strong technical background and performs security reviews of complex and high-profile applications such as Google Chrome and Microsoft Edge web browsers.

X41 D-Sec GmbH can be reached via `https://x41-dsec.de` or `mailto:info@x41-dsec.de`.

[1]`https://x41-dsec.de/news/security/research/source-code-audit/2024/02/13/bind9-security-audit/`
[2]`https://x41-dsec.de/security/research/news/2023/01/17/git-security-audit-ostif/`
[3]`https://blog.mozilla.org/security/2018/10/09/trusting-the-delivery-of-firefox-updates/`
[4]`https://browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf`
[5]`https://www.x41-dsec.de/reports/Kudelski-X41-Wire-Report-phase1-20170208.pdf`
[6]`https://www.x41-dsec.de/lab/blog/fax/`
[7]`https://2018.zeronights.ru/en/reports/zero-fax-given/`
[8]`https://www.x41-dsec.de/lab/blog/smartcards/`

Acronyms

CNI	Container Network Interface	9
CRI	Container Runtime Interface	9
CVE	Common Vulnerabilities and Exposures	10
CVSS	Common Vulnerability Scoring System	12
CWE	Common Weakness Enumeration	15
DNS	Domain Name System	9
DoS	Denial of Service	7
JSON	JavaScript Object Notation	17
OCI	Open Container Initiative	10
PID	Process ID	7
TOCTOU	Time of Check Time of Use	10