Kubevirt Seccomp Profile Generation

Automated Syscall Auditing

Personal Details **2**
Prologue **3**
Abstract 3
Objectives 4
Auditing Syscalls of the VirtLauncher pod **4**
Auditing Syscalls at Runtime 5
Using eBPF: 5
OCI Hook 5
Working of oci-seccomp-bpf 6
Setting up the oci-seccomp-bpf hook 7
Bpftrace 7
Using Seccomp Profile: 8
Using strace: 9
Kstrace 9
Wrapping the container entrypoint with `strace` 10
Using OCI hooks 10
Falco 10
Comparison of different approaches 11
Integration with Kubevirt CI/CD **12**
1. SSH 13
2. DaemonSet 13
Retrieving Logs from the Nodes 14
SSH 14
DaemonSet 14
Generating the Seccomp Profile 15
Testing the Seccomp Profile 15
Syscalls made by the VM workloads **15**
Project Timeline **16**
Community Bonding Period 16

Week 1: Deciding the Auditing Approach 17
Deliverables: 17
Week 2 & 3: Implementing Syscall Auditing 17
Deliverables: 17
Week 4: Generating the Seccomp profile 17
Deliverables: 17
Week 5: Refactoring, Testing and Documentation 18
Week 6&7: Proposing Integration with Kubevirt CI/CD 18
Deliverables: 18
Week 8: Mid-term evaluation and Feedback on the Proposal 18
Week 9: Set up K8s node to audit syscalls 18
Week 10: Refactoring, Testing and Documentation 18
Week 11&12: Integration with CI/CD pipeline 19
Week 13: Evaluation of the Implementation 19
Week 14: Explore Other Approaches to Improve Accuracy of the Generated Seccomp Profiles 19
Week 15: Implementation of the Plan to Improve the Seccomp Profile Accuracy 19
Week 16: Final Evaluation and Submission 20
Contributions to Kubevirt Organisation 20
Pull Requests 20
Merged 20
Open 20
About Me 20
Motivation for Contributing to Open Source 20
GSOC Related Questions 21
Have you applied to Google Summer of Code before? 21
Did you participate in Google Code-in as a student? 21
Have you ever contributed to open source before GSOC? 21
Are you applying to any other organization in GSOC? 21

Personal Details

Name	Nithish Karthik
Date of Birth	22/09/2001
Country	India
Email ID	nithishkarthik01@gmail.com
GIthub ID	sudo-NithishKarthik
University	Indian Institute of Technology (BHU), Varanasi
Contact Number	+91 99624 92550
Timezone	UTC+05:30 (Asia/Kolkata)
Preferred Language	English

Prologue

Abstract

Project Name	Automated Kubevirt Seccomp Profile Generation
Skills Involved	Linux, Golang, Docker, OCI, eBPF, Kubernetes, Virtualization, Ginkgo
Potential Mentor	Alice Frosi
Expected Size of Project	Large (350 hours, 16 Weeks)

Seccomp is a security facility from the Linux Kernel that prevents processes from executing unauthorised syscalls. By limiting the number of permitted syscalls, seccomp is being utilised in conjunction with Kubernetes to reduce the attack surface of the containers.

KubeVirt already supports custom Seccomp profiles, but that is based on the default seccomp profile that cri-o uses. This static approach leaves room for the profile to block necessary syscalls or allow unnecessary syscalls, hence compromising the security.

Objectives

This project focuses on automating the process of generating a seccomp profile for the VirtLauncher pod.

This project involves the following tasks:

1. Auditing the syscalls made by the VirtLauncher pod
2. Generating the Seccomp profile from the audit logs
3. Integrating this procedure with the kubevirt CI/CD pipeline
4. Improving the accuracy of the seccomp profile

Auditing Syscalls of the VirtLauncher pod

There are different methods to analyse what syscalls a program makes. Some of them are:

1. Static code analysis
    a. Complex process
    b. Differs for different programming languages
2. Tracing/auditing syscalls at runtime
    a. Relatively easy to do
    b. All possible scenarios should be covered while testing
3. Trial and error
    a. Requires domain knowledge
    b. Will likely filter too much or too little

c. Time consuming

Auditing Syscalls at Runtime

Auditing syscalls at the runtime is a very good option for us. In this approach, we have to make sure that all the use case scenarios of the VirtLauncher pod are covered while auditing. The current test suite covers all the scenarios the VirtLauncher pod would go through like setting up the VM, launching the VM, migrating the VM etc.

There are different methods we can use to trace the syscalls:

1. Kubernetes seccomp profile (using the Log action)
2. Using strace or similar alternatives
3. Using eBPF

Using eBPF:

OCI Hook

As long as the linux kernel is concerned, there is no entity called `container` as such, according to the kernel, it is just a process. When the container starts, if we can get the PID namespace of the container process, then we can compile and load an eBPF program that hooks into the `sys_enter` tracepoint and logs only the syscalls made by the processes that are within the PID namespace of the container.
But, the eBPF program has to be loaded just before the container starts. We can use the OCI hooks for this. OCI hooks allow us to run binaries at different life cycles of a container. We can use the `preStart` hook which will run a binary when the container namespaces are created, but the container has not started yet.

Our default K8s provider uses CRI-O. We can create an OCI hook in CRI-O by creating a hook configuration that describes what binary to be called and when it should be triggered (https://cloud.redhat.com/blog/extending-the-runtime-functionality for more information). There is already an OCI hook (oci-seccomp-bpf-hook) that generates a

seccomp profile for a container by using the same approach.

```
# [crio.runtime.runtimes.runc]
# runtime_path = ""
# runtime_type = "oci"
# runtime_root = "/run/runc"
# runtime_config_path = ""
# monitor_path = ""
# monitor_cgroup = "system.slice"
# monitor_exec_cgroup = ""
# monitor_env = [
#       "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
# ]
# allowed_annotations = [
#       "io.containers.trace-syscall",
# ]
# privileged_without_host_devices = false
```

This screenshot shows the crio.conf (CRI-O configuration, located at /etc/crio/crio.conf) file taken from the kubernetes node provisioned by Kubevirtci k8s provider. We can see that the default runc configuration allows for this hook to be enabled.

Kubernetes also provides a way to hook into the container lifecycles but it is not good enough for us since the `postStart` hook does not guarantee that it will be run before the entrypoint of the container.

Working of oci-seccomp-bpf

The oci-seccomp-bpf binary will be run whenever a container with the `io.containers.trace-syscall` annotation is created. It will be run when the container namespaces are created, but the container has not started yet. This binary expects the `kernel headers` module to be present. It first parses the annotation to get the input and output file location. Since the binary is run at the `preStart` point of the container lifecycle, the container will not start unless the binary exits/completes. Therefore, in order to make this process non-blocking, the binary creates a child process which will load the eBPF program and start tracing. The parent process will be waiting for a signal from the child process regarding whether or not the child has successfully started tracing. The parent process exits when:

1. The child process has successfully started tracing
2. The child process has returned with exit code not equal to 0
3. When the wait time exceeds a threshold

The runtime provides the container state as a json in the stdin so that the hook binaries can make use of it. The parent binary extracts the PID allotted for the container to be run and sends it to the child process. The child process compiles and loads an eBPF

program on the fly. The child process then starts listening to the events from a perf ring buffer through which the eBPF program will send syscall information. The eBPF program will start tracing the syscalls only when it sees a `prctl()` call, this is to make sure that our syscalls are not polluted by the operations done by runc. The runtimes (runc and crun) guarantees that a `prctl()` call will be made (for setting up the seccomp profile). The eBPF program also uses the `sched:sched_process_exit` tracepoint to trigger a notification when the container has exited to the child process. When the container exits, the child process uses the syscalls stored (by collecting events from the perf ring buffer) to generate a seccomp profile for the container.

Setting up the oci-seccomp-bpf hook

We would have to set up the hook to start the oci-seccomp-bpf binary in our node. This would require the node to have bcc toolchain and kernel headers. The hook will be configured in such a way that it will run for all the containers with the `io.containers.trace-syscall` annotation. This would mean that we would have to set the hook annotation for the VirtLauncher pod while running tests. Once the container exits, the profile will be outputted to a specific location. Note that if there are multiple containers running in a pod, we have to merge the whitelisted syscalls for all the containers to arrive at the final seccomp profile for the pod.

Some of the requirements for this hook to work is:

1. Root privileges (CAP_SYS_ADMIN) are needed. Hook will not work with rootless containers.
    a. The binary needs to have CAP_SYS_ADMIN to run. We can still start rootless VMs and will not be a problem for us.
2. The bcc tool chain and kernel-headers should be present in the node to be able to compile and load BPF programs

The oci-seccomp-bpf hook is used by container runtimes, such as runc, to apply seccomp filters to container processes. There is no generalised solution that automates the process of setting up this hook in kubernetes, most probably because it is dependent on the container runtime and container engine used. Therefore, we would have to write and maintain code to set up this hook for our kubernetes cluster.

Bpftrace

Bpftrace is a high-level tracing language for eBPF. Bpftrace uses LLVM as a backend to compile scripts to BPF-bytecode and makes use of BCC for interacting with the Linux BPF system, as well as existing Linux tracing capabilities.The bpftrace language is inspired by awk and C, and predecessor tracers such as DTrace and SystemTap.

If we are using bpftrace, we need a parameter that we can use to identify the syscalls made by the Pod in question. For instance, bpftrace has the capability to trace all the syscalls made by a process in a cgroup (https://github.com/iovisor/bpftrace/discussions/1988). But we would have to wait till the container has started and a cgroup is added in the pseudo fs (/sys/fs/cgroup/<docker-container-id>) with the container id to trace every single syscall made by the container.

We can use the OCI hook to tackle this problem. The `preStart` OCI hook is triggered only after the container namespaces and cgroups are created. We can use bpftrace to trace the syscalls made by the processes in the container's cgroup and output it to a log file.

There is already a Kubernetes plugin kubectl-trace for integrating bpftrace with Kubernetes. But we cannot use `kubectl-trace` to start tracing just as soon as the pod starts. We would have to modify the plugin to use OCI hooks or we would have to write our own solution and maintain it.

Using Seccomp Profile:

We can create a seccomp profile that uses the LOG action as the default action. This under the hood uses the `seccomp-bpf` filter mode with `prctl()`. This profile can be applied to the VirtLauncher pod. We can enable the KubevirtSeccompProfile feature gate in KubeVirt to apply this custom seccomp profile to the VirtLauncher pod. This audit seccomp profile (audit.json) can be added to the kubernetes nodes (stored at /var/lib/kubelet/seccomp/kubevirt/audit.json) while we set up the test environment, which is explained in the `Integration with Kubevirt CI/CD` section.

The logs will be stored in either auditd log file (/var/log/audit/audit.log) or syslog file depending on the system configuration. If the system has linux audit daemon running, the logs will be stored in the audit.log file by default, else it will be stored in the syslog file.

We won't have to write or maintain any code in this approach and we can very easily integrate this with kubernetes.

Some of the disadvantages of using this approach is:

1. We cannot differentiate between the syscalls made by different pods
    a. This will become an issue when other pods have seccomp profiles applied as we won't be able to differentiate between the syscalls made by

different pods. This will not be a blocker as long as we can ensure that the VirtLauncher pod will be the only pod with seccomp profile applied. If there are other pods that have a seccomp profile applied, we would have to do some extra work of modifying the profiles to stop logging in order for this approach to work.

2. Less information in the logs
    a. For instance, the logs don't list the arguments of the syscall which might be a limitation for us.
    b. We cannot control what information is exposed in the logs
3. Duplication in the logs
    a. For our use case, we are not really concerned about how many times a syscall is made by a pod. We are only concerned about whether or not the pod makes a particular syscall. When we apply a seccomp profile with LOG as the default action, all the syscalls made by the pod will be added in the log file which will contain a lot of duplicates. This increases the size of the log file. I had observed that the logs from a single VirtLauncher pod is around 24 MBs. This issue magnifies when we have multiple VirtLauncher pods running.

Using strace:

Using tracing tools like autrace, ptrace, sysdig or strace is generally a good option. But we have to analyse how it will be used in our context to arrive at a decision.
Tracing tools like strace, autrace, sysdig or ptrace are generally considered to be a good option for use cases like this.

Kstrace

If we take the kstrace tool (which is a wrapper around strace), it traces the syscalls made by a pod by creating a privileged pod and attaching it to the target pod, which will use strace to trace the system calls of the target pod. But the problem with kstrace is that, we cannot guarantee that kstrace will start tracing simultaneously when the target pod starts, and hence we might lose some syscalls.

If we do not want to use kstrace, if we have access to the kubernetes node, we can use `crictl` to get the PID of the container (since every container is just a linux process at the end of the day), with which we can run strace to trace the syscalls the container is making. But the disadvantage with this approach is that we still cannot guarantee that we can start tracing as soon as the container starts.

Wrapping the container entrypoint with `strace`

Another approach would be to modify the docker image to wrap the entry point command in `strace`. But it is not feasible to modify the docker image and it will be complex to integrate this with our functional tests suite.

Using OCI hooks

Another approach would be to use the OCI hooks as done in the case of eBPF approach. We will have a binary configured to the `preStart` hook of the container lifecycle. This binary will start `strace` for the container process (using the PID which we get from the stdin). Strace output files can be stored in some location and can be retrieved later. A new child process will be created by the `preStart` binary which will start `strace`. This is because if the `preStart` binary is blocking, then the container will not start.

But the problem with this approach is **finding when to stop tracing**. A trivial solution would be to use `(while kill -0 $pid; do sleep 1; done)` where $pid will be the pid of the container process. But this is inefficient compared to what we did in the eBPF approach.

In the eBPF approach, we had the ability to know when the container process exits by using the sched-process-exit tracepoint. There is no direct way to get notified about the exit/completion of a process. Doing a periodic scan of the /proc/ pseudo file system is another approach but it has the same problems we have with the signal 0 kill approach and doesn't make it any better. We cannot use the `postStop` OCI hook since we won't have access to the `preStart` hook binary's child process to signal it to stop tracing.

With some amount of engineering effort, we can make the other aspects of this approach work the same way as the eBPF approach. For instance, we can use the –seccomp-bpf flag to make it faster. And we can make strace stream the logs to a child process (by child process, I mean that this process will be a child of the OCI `preStart` hook binary, this has to be a child process because the container will not start unless the binary is non-blocking) before writing it to avoid duplicate logs. But we have to assess whether this would give any advantage over the eBPF approach.

When we are using OCI hooks, we would have to write and maintain code (it might just be a shell script) that sets up the hook in our kubernetes cluster.

Falco

Falco is a tool created by Sysdig to make the kubernetes clusters more secure by consuming kernel events and enriching those events with information from Kuberentes and the rest of the cloud native stack. Falco is a tool focused more on how we can make

use of the sourced kernel events to make our clusters secure. It supports creation of rules with which we can configure to alert on any unusual events. For our use case, we can write a custom falco rule that just logs the syscalls of the VirtLauncher pod containers. The falco daemon has to be deployed on all the nodes with our custom rule.

Falco has support for different drivers:

1. A kernel module
2. An eBPF probe
3. A ptrace(2) userspace program

Depending on the configuration, the syscalls can be traced by using any of the drivers above. Each driver has its pros and cons and we have to decide which one suits best for our use case. Falco automatically exposes container related information to the rules, so we don't have to do all the low level stuff like creating OCI hooks and all.
We can write a falco rule that can use the kubernetes field class to filter all the syscalls made by the VirtLauncher pod and log the output to a file.

Since tracing is not the main goal of Falco (Falco can be thought of as a framework built on top of the tracing solutions provided by its drivers), we might have to do some experimenting with it to find out how feasible and easy it is to integrate it with our system. We won't be using all the functionalities provided by Falco, therefore one question to think about is whether we want to use a full fledged framework just for a small subset of its features. It would be great if we can install Falco in such a way that we only have the functionalities we need.

There are different approaches for deploying falco like using HELM or directly installing it on the nodes. If we are using Falco, more time will be spent on figuring out how to deploy it in the best way in our system.

Falco is designed keeping kubernetes in mind and therefore it can be easily integrated with our kuberentes clusters. Not much maintenance will be needed from our side.

Comparison of different approaches

We have seen an overview of the different methods that are available to us to audit the syscalls of the VirtLauncher pod at runtime. None of the approaches require any change to the virt-launcher and kubevirt.

The factors that influence our choice are:

1. Ease of integrability of the procedure in our CI/CD flow

2. Compatibility with different kubernetes providers and container engines
3. The solution has to be independent of the container engine and the kubernetes provider used
4. Integrability with kubernetes

A time slice of the project will be dedicated to analyse the advantages and disadvantages of each approach in much more depth and discuss with the community regarding which approach would be the best for our use case. A clear and concise final report will be delivered as to why we chose one approach over the others.

Integration with Kubevirt CI/CD

Once we decide on which approach to use for auditing the sycalls of the VirtLauncher pod, We have to integrate that procedure with our CI/CD pipeline so that the process can be automated. The idea is that, whenever a new release is made, along with the functional tests that run, we also audit the syscalls made by the VirtLauncher pod using any one of the approaches discussed and use the audit data to generate the seccomp profile.

Kubevirt uses Prow, a kubernetes based CI/CD system. Currently, when a release is made in the kubevirt/kubevirt repository, Prow's trigger plugin triggers it to create ProwJob CRDs using the presubmits job configs present in the project-infra repository. Among these jobs are the e2e tests for different kubernetes providers. The `prow-controller-manager` component uses the pod spec in the ProwJob to create pods that run the desired job. For the e2e tests, the quay.io/kubevirtci/bootstrap bootstrap image is used as the container image for the pod. Prow automatically mounts the root of the repository with the working state of the current PR (by rebasing the current commit with master) as the working directory of the container. It then runs the `automation/test.sh` script which sets up the desired kubernetes cluster and runs functional tests on it. The ProwJob pod runs docker inside it (the cluster on which prow is deployed is configured to run docker inside the pods) and hence is able to provision different kubernetes providers like kind and k3d.

Apart from kubevirtci-kubernetes, kubevirtci contains procedures to provision kind and k3d clusters as well, which are useful for some test cases (kind-gpu, kind-sriov, k3d-sriov). For our use case, we need the ability to set up these k8s nodes to be able to audit the syscalls made by the VirtLauncher pod.

There are two ways we can do this:

1. SSH

Kubevirtci provides functions to ssh into the kubernetes nodes irrespective of what providers are used (except external and local providers, but those are not being used in any of the test cases). For kind and k3d providers, we have the _ssh_into_node (kind and k3d) function to ssh into the nodes. These functions use the underlying container engine to exec into the node container to provide this functionality. For kubevirtci kubernetes providers (k8s-1.22, k8s-1.23, k8s-1.26-centos9 etc.), it uses the gocli's ssh command to provide this functionality.

For us to be able to add the functionality to audit the syscalls of the VirtLauncher pod in the kubernetes nodes, we need the ability to execute commands inside the nodes. Commands to be executed are dependent on the approach we are using to audit the syscalls. For instance, if we are using the eBPF approach, we would have to install bcc, download the hook binary, configure the `preStart` hook with this binary etc.

For kind and k3d providers, we can use the container engine's exec functionality to achieve this, similar to how it is done for ssh. Another function called `setup_syscall_auditing` can be created in the kind/common.sh and k3d/common.sh.

For normal kubernetes providers (k8s-1.22, k8s-1.23, k8s-1.26-centos9 etc.), we can use gocli's ssh functionality to execute commands on the node.

2. DaemonSet

Another approach that we can use is to use a DaemonSet that will execute the same set of commands on all the nodes of the cluster. The pods of the daemonset have to be privileged (it should have access to the filesystem of the node) in order to be able to set up the node. We will have the ability to monitor and manage logs for daemons in the same way as applications if we are using daemonset. Using a daemonset is a more generalised solution since it abstracts away the need to find a way to communicate with the node.

After the nodes are set up to audit the syscalls made by the VirtLauncher pod, the e2e tests will be run. Once the tests are complete, we have to retrieve the syscall audit logs from the nodes.

Retrieving Logs from the Nodes

Based on the approach we decide to use, we will have the audit logs for all the syscalls made by the VirtLauncher pod present in the node itself. This section describes an approach we can use to retrieve those logs from the nodes, so that they can be used to generate the Seccomp profile.

Since our solution for auditing the syscalls will be independent of the container engine and the kubernetes provider used, the auditing solution will make sure that the audit logs are stored at a specific location on the node.

SSH

This section describes how we can retrieve the logs in the nodes if we are using ssh to set up the nodes to audit the syscalls.
For all the clusters deployed using the kubevirtci kubernetes providers, we can use the gocli's scp command to retrieve the audit logs from the nodes.

For kind and k3d providers, we can create something like the _ssh_into_node (kind and k3d) function to copy the log files. We can use `docker cp` (or `podman cp` in case of podman) to create _scp_node function which we can use to copy the log files from the node to the local system.

For local and external kubernetes providers, we can run the `kubectl debug node` command , which will create a pod with privileged security context on the node, in combination with `kubectl cp` to obtain the log files from the node.
For Openshift clusters, we can use the `oc debug node` in combination with `oc rsync` to achieve the same thing. We would not have to worry about this since we are not provisioning any external or local clusters in the test environment now.

DaemonSet

If we are using daemonset to set up the nodes to audit syscalls, we can retrieve the logs using `kubectl logs`. Each daemonset pod in the node will be configured to audit the syscalls logs made by the VirtLauncher pod and write them to stdout. The container runtime will then take these logs and store them in a file and when we use `kubectl logs`, we get the logs of the pods, which in our case is the syscall audit log. We would have to be careful about not letting kubernetes come in our way. For instance the maximum log files and maximum log file size and log rotation has to be configured properly so that we don't lose our logs.

Generating the Seccomp Profile

The next step is to merge the logs from all the nodes and generate a seccomp profile. This profile should then be pushed as the artifact of the ProwJob.

We can see here that there are more than one presubmit jobs configured for e2e tests alone. There are different ones for testing different k8s providers (kind, k8s-1.26, k3s etc.), testing different configurations (cgroupsv2 etc.) and testing different class of functionalities (sig-compute, sig-network, sig-storage etc.). All these jobs combined give us the status of the full system integration test.

Therefore, we have to use the seccomp profiles generated by each job (which will be present as an artifact of the ProwJob) and merge them to give us a finalised seccomp profile. A new ProwJob config will be added which will do exactly this.

Testing the Seccomp Profile

A full functional retest can be done with the generated seccomp profile to ensure that nothing breaks with the application of this seccomp profile. Once the seccomp profile that is generated has been verified, the profile will be pushed to the kubevirt/kubevirt repository.

Therefore, in essence, the newly created ProwJob config will do the following:

1. Wait till all the e2e jobs complete and merge the seccomp profiles generated by every e2e job to generate a finalised seccomp profile
2. Rerun the whole test suite with the seccomp profile applied to ensure that nothing breaks
3. Push the verified seccomp profile to the kubevirt/kubevirt repository

This is just an overview of what the direction of action will be. More time will be spent going into the details of implementation once the proposal is accepted.

Syscalls made by the VM workloads

So far, we have only been talking about the syscalls the VirtLauncher pod would make for it to support the feature-set advertised by kubevirt (setting up the VM, launching the

VM, migrating the VM etc.), but we cannot predict what syscalls the VM workloads might use and hence we cannot simulate those in the test cases.

There may be some shared resources between the host and guest systems, such as network interfaces or storage devices, which could be subject to the seccomp filters applied to the container process. Additionally, any system calls made by the QEMU/KVM process itself will be subject to the seccomp filters applied to the container.

If a program running in the VM makes a syscall, it will be blocked when:

1. It triggers the `qemu-kvm` process to make a syscall that is blocked by the container seccomp profile (happens when trying to access resources shared by the host kernel)

We can classify the syscalls required by the VirtLaucher pod into two categories:

1. Syscalls required to setup and launch the VM with the feature-set advertised by kubevirt
2. Syscalls required by the VM for it's workloads

Auditing the VirtLauncher pod in the e2e test environment will give us only the syscalls of the first category. As a result of this, by applying the seccomp profile generated by using the syscalls we got by auditing the VirtLauncher pod in the e2e environment, we cannot be 100% sure that it will not block any workloads running in the VM.

If time permits, this project can be extended to implement a solution to this problem by *predicting* what syscalls might be made based on the VMI spec. If that is not good enough, we can use fuzzing as a last option to improve the accuracy of the seccomp profile.

Project Timeline

This week-by-week timeline provides a rough guideline of how the project will be done

Community Bonding Period

Date: May 4 - May 28

- Enhance my understanding of the Kubevirt architecture.
- Get to know my mentor and other community members.
- Participate in community meetings and discussions to get familiar with the project's ecosystem

- Discuss the project plan and objectives with my mentor and gather their insights
- Identify potential challenges and discuss possible solutions with the community
- Review the existing codebase and understand the structure and conventions used in the project

Week 1: Deciding the Auditing Approach

- Analyse the advantages and disadvantages of each approach in much more depth
- Gather feedback from mentors and the community
- Finalize on the approach to use for auditing the syscalls

Deliverables:

- A clear and concise report of why one approach was chosen over the others

Week 2 & 3: Implementing Syscall Auditing

- Write code for implementing the syscall auditing functionality
- Gather feedback from mentors and the community
- Incorporate the feedback and finalize the implementation

Deliverables:

- A binary, eBPF program, an OCI hook or a seccomp profile based on the auditing method that was chosen

Week 4: Generating the Seccomp profile

- Write code for implementing the seccomp profile generation functionality using the audit logs
- Gather feedback from mentors and the community
- Incorporate the feedback and finalize the implementation

Deliverables:

- A binary that can parse the syscall audit log files and generate the seccomp profile

Week 5: Refactoring, Testing and Documentation

- Refactor the code written so far to ensure consistency and readability
- Optimize the code for better performance
- Add test suites for the written code
- Gather feedback from mentors and the community
- Finalize the refactoring and optimization based on feedback received
- Write comprehensive documentation for the code written

Week 6&7: Proposing Integration with Kubevirt CI/CD

- Create a proposal for how the finalized method for seccomp profile generation will be integrated with Kubevirt's CI/CD pipeline
- Involves a more detailed analysis of what is discussed in this doc under the 'Integration with Kubevirt CI/CD' section
- Involves analysis of how the auditing solution can be integrated with different clusters used for testing (like kind, k3d, kubevirtci-kubernetes, etc.)
- Gather feedback from mentors and the community
- Finalize the proposal based on feedback received

Deliverables:

- A design spec that proposes how the seccomp profile generation method can be integrated with our CI/CD pipeline

Week 8: Mid-term evaluation and Feedback on the Proposal

- Submit the work done so far for evaluation
- Gather feedback from mentors and the community
- Make necessary changes and improvements based on feedback.
- Allocate time for additional feedback and adjustments throughout the project.

Week 9: Set up K8s node to audit syscalls

- Implement the procedure to configure the K8s nodes to audit syscalls based on the proposal
- Gather feedback from mentors and the community
- Incorporate the feedback and finalize the implementation

Week 10: Refactoring, Testing and Documentation

- Refactor the code written so far to ensure consistency and readability

- Optimize the code for better performance
- Add test suites for the written code
- Gather feedback from mentors and the community
- Finalize the refactoring and optimization based on feedback received
- Write comprehensive documentation for the code written

Week 11&12: Integration with CI/CD pipeline

- Create a ProwJob as per the design spec to generate the seccomp profiles using the implementation work done on week 4
- Add support for testing the generated profile
- Gather feedback from mentors and the community
- Incorporate the feedback and finalize the implementation

Week 13: Evaluation of the Implementation

- Evaluate how well the implementation works by testing with a bunch of VMs with typical workloads
- This evaluation is mainly to test whether the syscalls made by the workloads are blocked by the seccomp profile thereby breaking the system

Week 14: Explore Other Approaches to Improve Accuracy of the Generated Seccomp Profiles

- Depending on the evaluation result, explore the feasibility of using methods like fuzzing and prediction of syscalls by using the VMI spec
- Discuss with the mentors and community
- Finalize on the plan on action

Week 15: Implementation of the Plan to Improve the Seccomp Profile Accuracy

- Implement the finalized method for improving the accuracy of the profile
- Gather feedback from mentors and the community
- Incorporate the feedback and finalize the implementation

Week 16: Final Evaluation and Submission

- Submit the final work for evaluation.
- Address any last-minute feedback or changes.
- Prepare for the project's completion and handover.
- Allocate a contingency week to address any unforeseen issues or delays that may occur during the project

Contributions to Kubevirt Organisation

I signed the Contributor's Agreement for Kubevirt on 1 March 2023.

Pull Requests

Merged

- [Docs] Update Docs for Docker Authentication in Mac
- [Fix] Get rid of noisy logs in checkVolumesForMigration

Open

- [Refactor] Validate evictionStrategy in KubeVirt Config Update
- [Fix] Avoid nil pointer dereferencing when trying to access evictionStrategy

About Me

I am a passionate software developer who strives to write clean and efficient code. You can get to know more about my experiences through my linkedin.

Motivation for Contributing to Open Source

My motivation for contributing to open-source projects is multi-faceted. I am always looking for new opportunities to explore different codebases and stacks outside my organization and improve my coding abilities.
These projects follow strict code guidelines, often maintained by international organizations or senior developers, which leads to improved developer experience. In addition to enhancing my technical skills, contributing to open-source projects also allows me to connect with other developers in the community.
This collaborative aspect of open-source projects is essential for professional growth

and building a network of like-minded individuals. Finally, I am motivated by the potential to impact the world through my contributions positively. Open-source projects often address critical issues and can significantly impact people's lives.
Effective communication is essential in software development, and contributing to open-source projects provides a platform for inexperienced developers like myself to improve our overall software development skills

GSOC Related Questions

Have you applied to Google Summer of Code before?

No, I haven’t applied to Google Summer of Code before

Did you participate in Google Code-in as a student?

No, I didn’t participate in Google Code-in

Have you ever contributed to open source before GSOC?

Yes, I have had prior experience in contributing to open source before GSOC.

Are you applying to any other organization in GSOC?

No, I am only applying to Kubevirt

Problems with using OCI hooks:

1. Only CRI-O supports OCI hooks, containerd does not support it
    a. We would have to build a wrapper around the runc (runtime) as an alternative step

Using Seccomp:

1. Easy to configure
    a. We don’t have to do much
2.