Nvidiascape Vulnerability Lab – Memory-Based Container Escape Simulation

Author: Robert Blackwell III

Lab Type: Home Lab | Blue Team Security | Container Security | Memory Exploitation

Platform: Ubuntu (Host + Container) | Docker | PyTorch (as attack vector)

# Overview

This lab explores the real-world implications of container breakout attempts, specifically focusing on a simulated version of the Nvidiascape vulnerability. Using a PyTorch container, the objective was to assess how in-memory access can lead to interaction with the host system — even in the absence of mounted volumes or privileged container settings.

# Objective

- Simulate a container escape vulnerability using PyTorch inside Docker  
- Observe whether memory-based access could lead to host interaction  
- Evaluate whether sensitive data within the container and host remain secure  
- Reflect on container hardening best practices and the limits of container isolation

# Lab Setup

|  |  |
| --- | --- |
| Component | Details |
| Host OS | Ubuntu (or compatible Linux distro) |
| Docker Engine | Installed with default settings |
| Container Image | PyTorch with GPU libraries |
| Attack Vector | In-memory inspection + shell access |
| Tools Used | `ps`, `gdb`, `cat /proc`, Python |

# Procedure

1. Deployed PyTorch container with elevated capabilities (e.g., GPU access).  
2. Enumerated memory and process tree using tools like /proc and gdb.  
3. Escalated within the container using default root privileges.  
4. Attempted to access host system features (directories, device mappings, etc.).  
5. Observed what was and wasn’t possible without breaking container isolation directly.

# Key Findings

- Possible to interact with host system metadata without direct file access.  
- Memory exposure and shared kernel increase attack surface.  
- Proper configuration blocks direct access to host files.  
- Privileged containers and mounted host paths increase risk.

# Personal Reflection & Security Takeaway

Even if an attacker gains access to a container — especially when it’s exposed on the organizational network — that doesn’t mean they’ll automatically access sensitive data. What I proved in this lab is that containers can be breached, but proper hardening still protects the “real deal” data. It’s less about what’s stored inside the container and more about the power the attacker has once they’re in. The container may be open to access, but without misconfiguration, the attacker still has to work to reach sensitive assets. This reinforces a key idea in Blue Team defense: breach may be inevitable — but damage is preventable.

One key lesson I had to unlearn was the belief that “not running as root” meant the container wouldn’t contain sensitive data. That was a misconception. The truth is, a container can contain sensitive information, regardless of whether it runs as root. The real purpose of avoiding root is to limit what the attacker can do once inside — not to prevent them from getting in. A skilled attacker may still gain access to the container, but the container’s internal privileges and protections will determine how far they can go.

In this lab, PyTorch was used as the brain to understand the structure of the container, and through that, I was able to simulate a container escape and interact with the host system. Because the container was not running as root, and assuming the environment was properly hardened, I was unable to access privileged or sensitive host files. That’s an important distinction: access to the container doesn't equal access to the host — especially when proper security boundaries are in place.

From a Blue Team perspective, I would deploy runtime security tools to monitor and protect containerized environments. These tools track behavior inside the container while it's running. In the scenario I simulated, a runtime tool like Falco would have detected suspicious activity and raised an alert. However, Falco is a detector — it alerts but doesn’t stop the attack. That’s where tools like AppArmor or Seccomp come in: AppArmor can block access to files or paths, and Seccomp can restrict dangerous system calls.

When alerts are triggered, additional tools — known as automated responders — can quarantine or kill the container automatically. However, the job doesn’t end there. A SIEM (like Splunk, ELK, or Graylog) is used to investigate what happened and how it happened. Even if a threat is blocked by a tool like AppArmor, as an analyst I still have to dig into the logs, verify the source of the breach, and then strengthen the system to recognize and stop similar future threats.

So, the process becomes: detection → response → investigation → prevention.

**Clarified Blue Team Understanding**

A runtime security tool like AppArmor or Seccomp enforces security policies within containers. It does not detect threats by itself, but it can block or deny unsafe behaviors, such as trying to access protected files or perform dangerous system calls.  
  
To detect suspicious behavior in real time, tools like Falco are used. Falco watches for unexpected commands, network activity, or access attempts inside containers. When it notices something unusual (like what I did in this lab), it generates an alert and sends it to a SIEM system, such as Splunk, ELK, or Graylog.  
  
Once the alert is generated and reaches the SIEM, my role as a security analyst is to:  
1. Investigate the event and analyze what container or system was affected.  
2. Understand how the activity happened — was it a vulnerability, a misconfiguration, or a compromised image?  
3. Assess the impact and verify whether any sensitive data or systems were accessed.  
4. Apply hardening strategies to prevent the same type of incident from happening again.  
  
This process of monitoring, blocking, alerting, investigating, and hardening is the full Blue Team loop. While AppArmor may block the behavior, it doesn’t explain why it happened. That’s why the SIEM and the human analyst are still crucial.  
  
So to clarify my earlier confusion: Not running a container as root doesn’t mean there is no sensitive information inside. It means the attacker has limited capabilities once inside. Containers often hold secrets, data, or connections, and an attacker can still reach those. The purpose of hardening is to contain the blast radius and reduce the risk of deeper damage.

# Lessons Learned

- Never run containers as root unless absolutely required.   
- Avoid `--privileged` mode unless the security impact is understood.  
- Use runtime security tools (AppArmor, Seccomp, gVisor).  
- Treat containers as ‘first line exposure’ and isolate access to sensitive systems.

# Notes & Next Steps

- This was one of several home lab experiments exploring container security.  
- Future iterations will test Kubernetes, runtime monitoring, and logging from inside containers.