

CLOUDNATIVE
SECURITYCON
NORTH AMERICA 2024

JUNE 26-27 | SEATTLE, WA #CNSCon

Accelerating AI Securely with gVisor



Lucas Manning

Why Sandbox?

Containers rely on kernel features for isolation

+

All the containers share the same kernel

+

The kernel has vulnerabilities

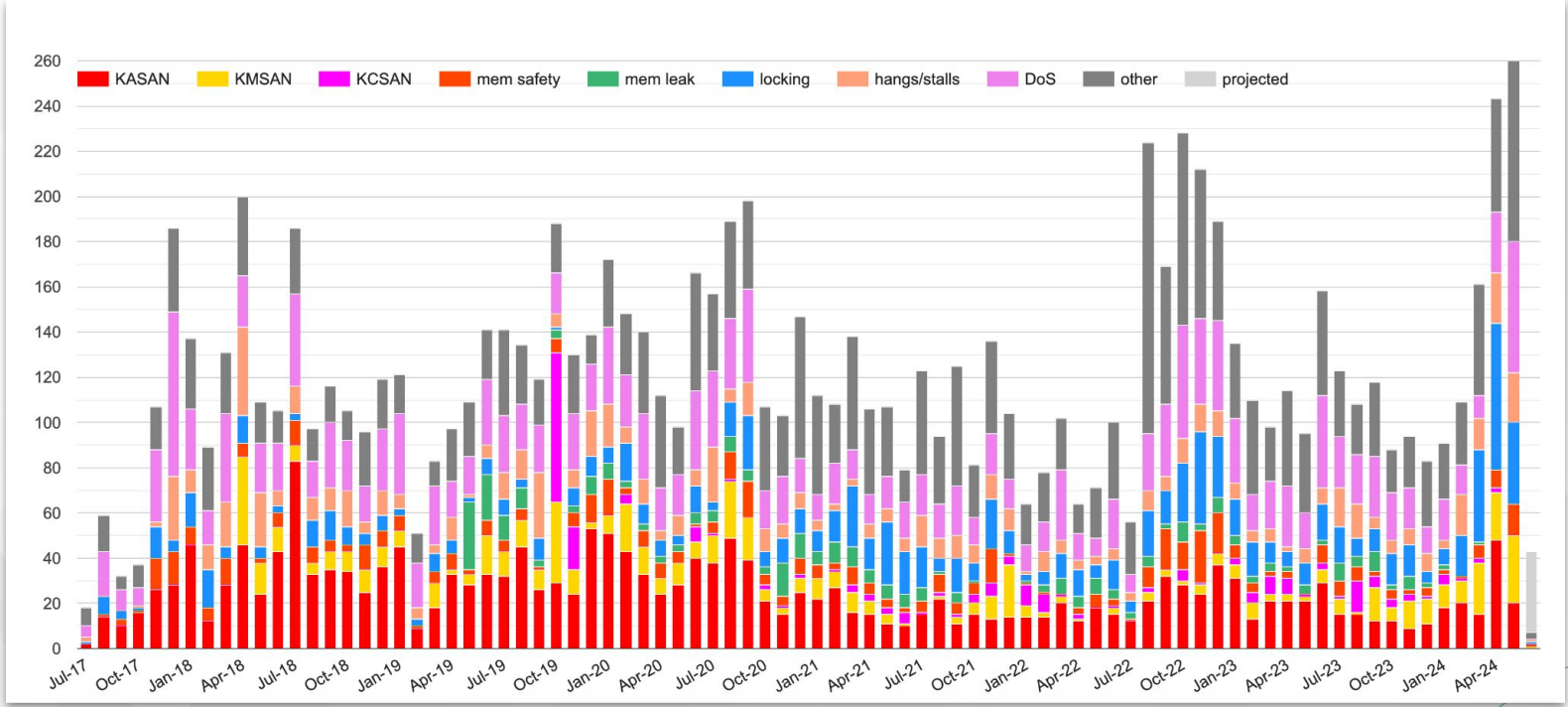
+

Container escape can give you every container on the host

Why Sandbox?

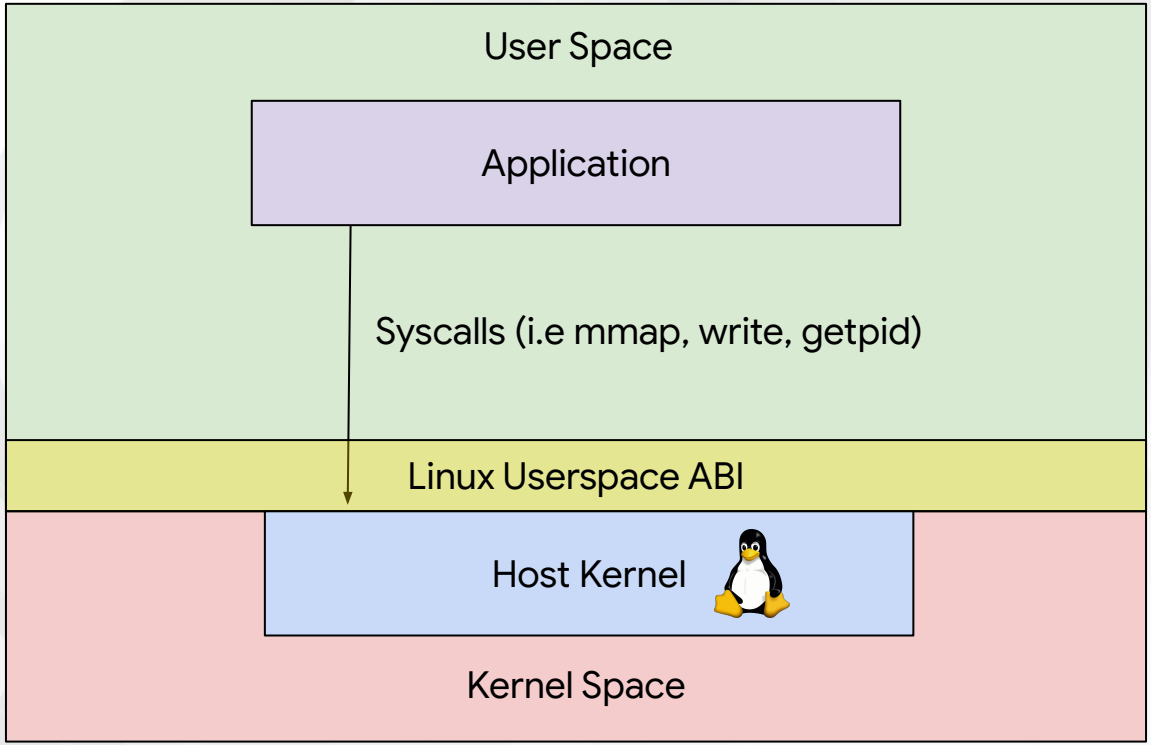
- Native containers rely on kernel features for security
- Linux vulnerabilities are constant
  - Syzkaller (fuzzing tool) found 240 bugs in May 2024
  - These are public and easy to find
  - Not all are fixed, and you only need one
- We need to run containers that pose risk to host systems
  - Parsing/storing/running user-generated content
  - Working with sensitive datasets
  - Running third party code
  - Software supply chain attacks
- **Since 2020, gVisor has protected against 98% of Linux pod-to-guest escalation vulnerabilities**

Why Sandbox?



Source: https://syzkaller.appspot.com/upstream/graph/found-bugs

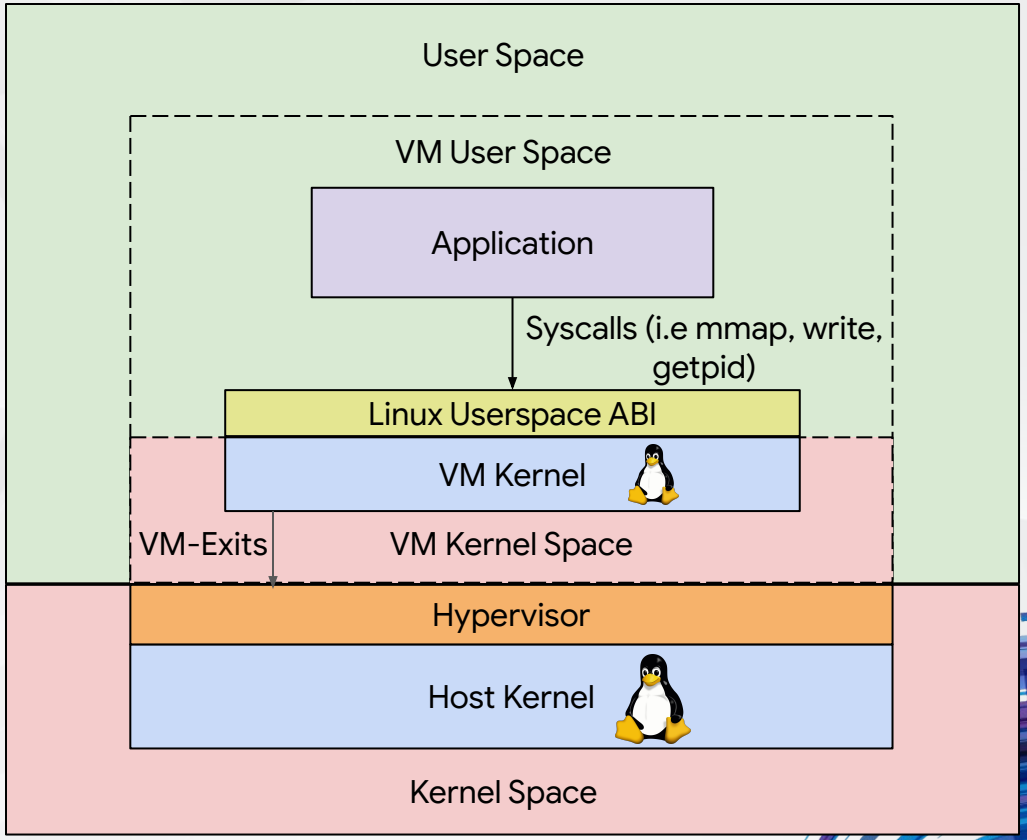
Unsandboxed Containers



📦 Sandboxing Methods 🛠️

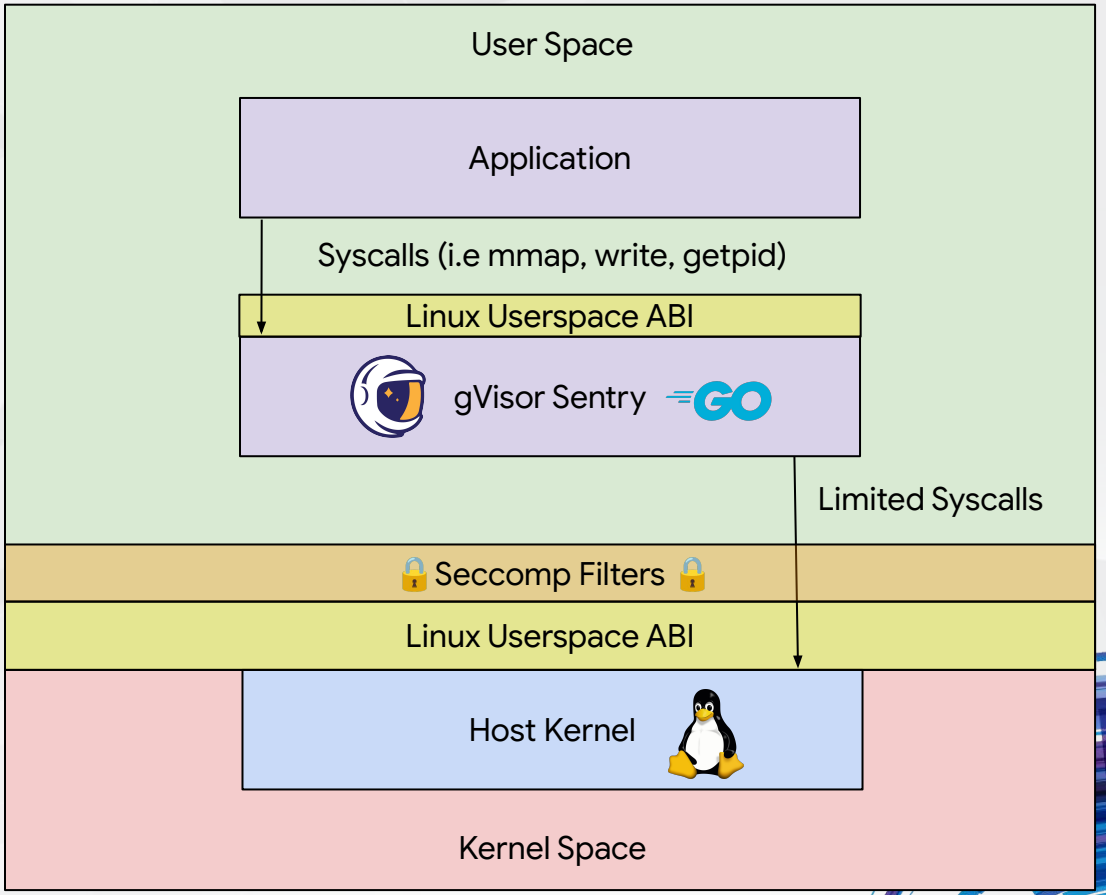
Sandboxing Methods: MicroVMs

- Example: Kata Containers
- Run containers in lightweight VM
- Choose your hypervisor (i.e. Firecracker 🔥)
- Run unmodified binaries
- No more shared kernel



Sandboxing Methods: Application Kernels

- Popular example: gVisor
- Linux system call interception + Userspace API emulation in Go + Host system call filtering
- No more shared kernel
- 2 layers of defense
- Run unmodified binaries
- Sentry implemented in GO



Sandboxing Methods: Summary

- Application kernels (gVisor)
  - Linux system call interception + Userspace API emulation in Go + host system call filtering
- MicroVMs (Kata, Firecracker 🔥)
  - Run container in a lightweight VM

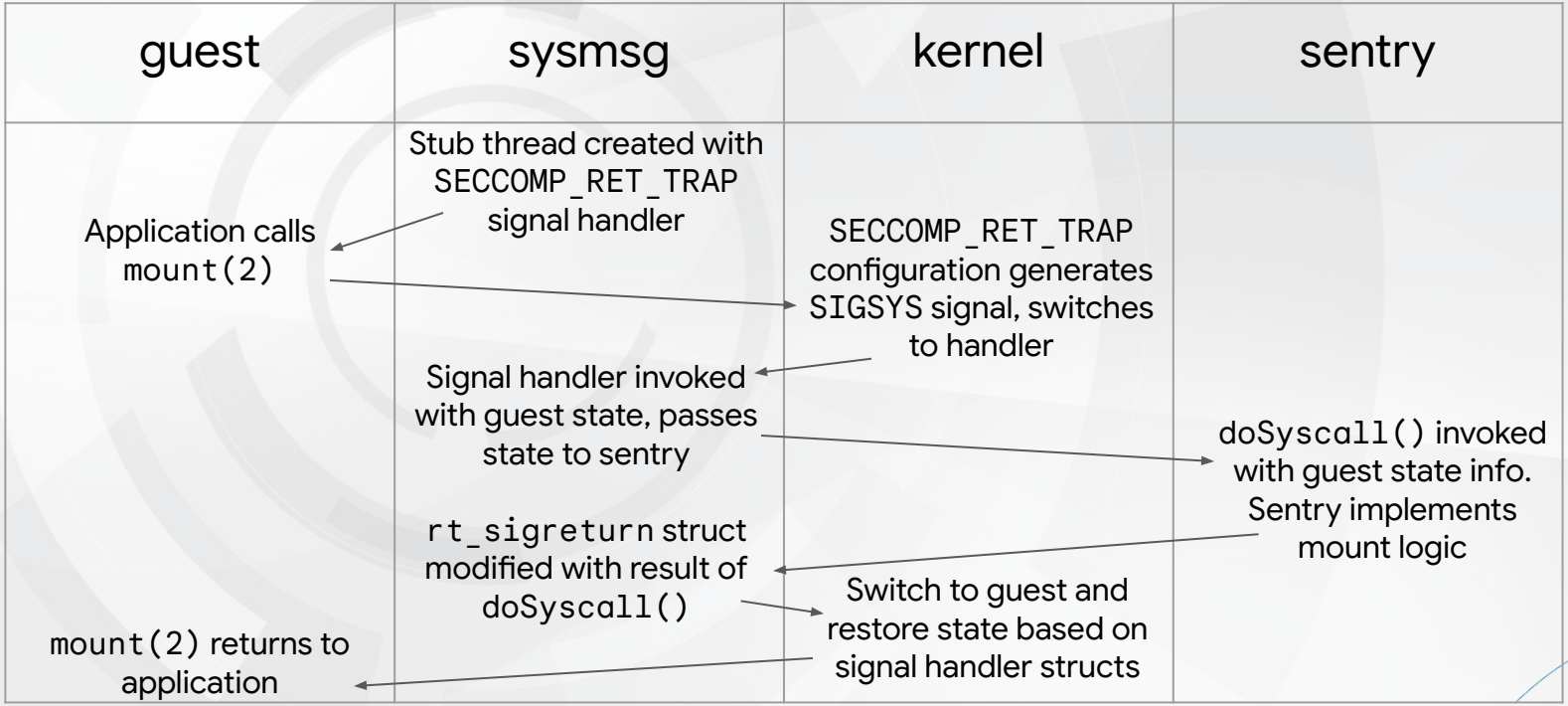
	Startup latency	Steady-state perf	Security	Compatibility	Host Requirements	Mem/CPU footprint
Native					Process	
Application kernel					Process	
MicroVM					Virtualization	



gVisor Deep Dives



gVisor Deep Dive: Intercepting with Systrap



gVisor Deep Dive: Implementing mount(2)

```
func (t *Task) executeSyscall(sysno uintptr, args arch.SyscallArguments) (rval uintptr, ctrl
*SyscallControl, err error) {
	s := t.SyscallTable()

	// ... snip ...
		fn := s.Lookup(sysno)
		var region *trace.Region // Only non-nil if tracing == true.
		if trace.IsEnabled() {
			region = trace.StartRegion(t.traceContext, s.LookupName(sysno))
		}
		if fn != nil {
			// Call our syscall implementation.
			rval, ctrl, err = fn(t, sysno, args)
		} else {
			// Use the missing function if not found.
			rval, err = t.SyscallTable().Missing(t, sysno, args)
		}
	// ... snip ...
	return
}
```

//pkg/sentry/kernel/task_syscall.go

gVisor Deep Dive: Implementing mount(2)

//pkg/sentry/syscalls/linux/linux64.go

```
var AMD64 = &kernel.SyscallTable{
      OS:   abi.Linux,
      Arch: arch.AMD64,
      Version: kernel.Version{
            Sysname: LinuxSysname,
            Release: LinuxRelease,
            Version: LinuxVersion,
      },
      AuditNumber: linux.AUDIT_ARCH_X86_64,
      Table: map[uintptr]kernel.Syscall{
            0:   syscalls.SupportedPoint("read", Read, PointRead),
            1:   syscalls.SupportedPoint("write", Write, PointWrite),
            2:   syscalls.SupportedPoint("open", Open, PointOpen),
            3:   syscalls.SupportedPoint("close", Close, PointClose),
            4:   syscalls.Supported("stat", Stat),
            5:   syscalls.Supported("fstat", Fstat),
// ...snip...
            165: syscalls.Supported("mount", Mount),
            166: syscalls.Supported("umount2", Umount2),
// ...snip...
```

gVisor Deep Dive: Implementing mount(2)

```
func Mount(t *kernel.Task, sysno uintptr, args arch.SyscallArguments) (uintptr, *kernel.SyscallControl,
error) {
	sourceAddr := args[0].Pointer()
	targetAddr := args[1].Pointer()
	typeAddr := args[2].Pointer()
	flags := args[3].Uint64()
	dataAddr := args[4].Pointer()

	creds := t.Credentials()
	if !creds.HasCapabilityIn(linux.CAP_SYS_ADMIN, t.MountNamespace().Owner) {
		return 0, nil, linuxerr.EPERM
	}
// ...snip...
	targetPath, err := copyInPath(t, targetAddr)
	if err != nil {
		return 0, nil, err
	}
// ...snip...
	source, err := t.CopyInString(sourceAddr, hostarch.PageSize)
	if err != nil {
		return 0, nil, err
	}

// ...snip...
	_, err = t.Kernel().VFS().MountAt(t, creds, source, &target.pop, fsType, &opts)
	return 0, nil, err
}
```

//pkg/sentry/syscalls/linux/sys_mount.go

gVisor Deep Dive: Implementing mount(2)

```
func (vfs *VirtualFilesystem) MountAt(ctx context.Context, creds *auth.Credentials, source string, target
*PathOperation, fsTypeName string, opts *MountOptions) (*Mount, error) {
	mnt, err := vfs.MountDisconnected(ctx, creds, source, fsTypeName, opts)
	if err != nil {
		return nil, err
	}
	defer mnt.DecRef(ctx)
	if err := vfs.ConnectMountAt(ctx, creds, mnt, target); err != nil {
		return nil, err
	}
	return mnt, nil
}
// ...skipping lots of mount logic...
func (vfs *VirtualFilesystem) connectLocked(mnt *Mount, vd VirtualDentry, mntns *MountNamespace) {
// ...snip...
	mnt.IncRef() // dropped by vfs.umount().
	mnt.setKey(vd)
// ...snip...
	vd.mount.children[mnt] = struct{}{}
	vd.dentry.mounts.Add(1)
	mnt.ns = mntns
	mntns.mountpoints[vd.dentry]++
	mntns.mounts++
	vfs.mounts.insertSeqed(mnt)
	vfsmpmounts, ok := vfs.mountpoints[vd.dentry]
// ...snip...
}
```

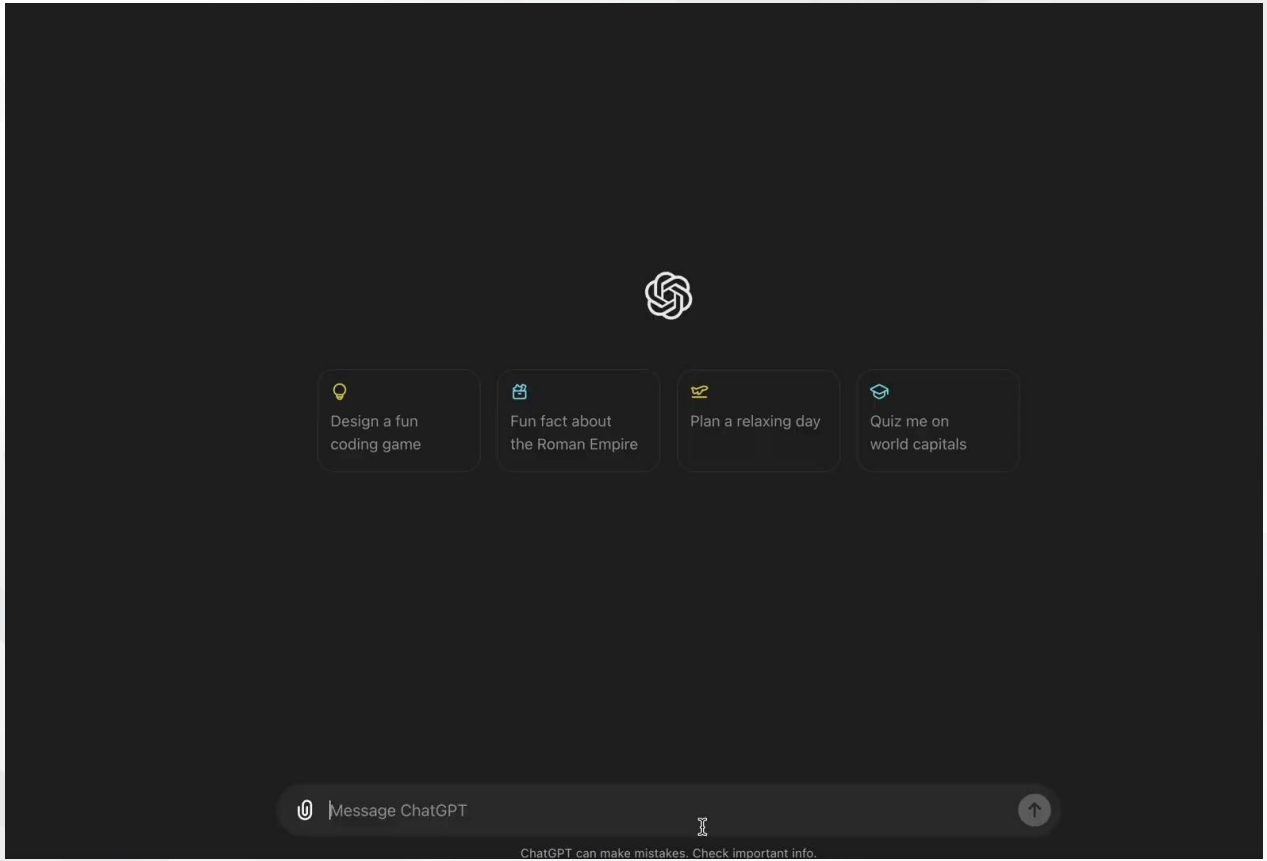
//pkg/sentry/vfs/mount.go

📦 Sandboxing Scenarios 🤔

Scenario: Executing LLM generated code

- Threat landscape?
  - Malicious user gets the LLM to write a container escape
  - Skynet comes online? 💀
- What environment are you running in?
  - Probably a Cloud VM/k8s pod
  - Nested virtualized support not guaranteed on all machine types
- Can you afford higher resource costs?
  - Per container overhead is important, there will be lots of these.
- What kind of work?
  - All kinds theoretically... mostly simple snippets in practice
- How long will containers run for?
  - Very short

Scenario: Executing LLM generated code



Scenario: Protecting Against Software Supply Chain Vulnerabilities During Model Serving

Google » Tensorflow : Product details, threats and statistics

Versions | Vulnerabilities (429) | Product Dashboard | CVSS Report | Metasploit Modules

Log in to view product risk score details

Vulnerabilities by types/categories

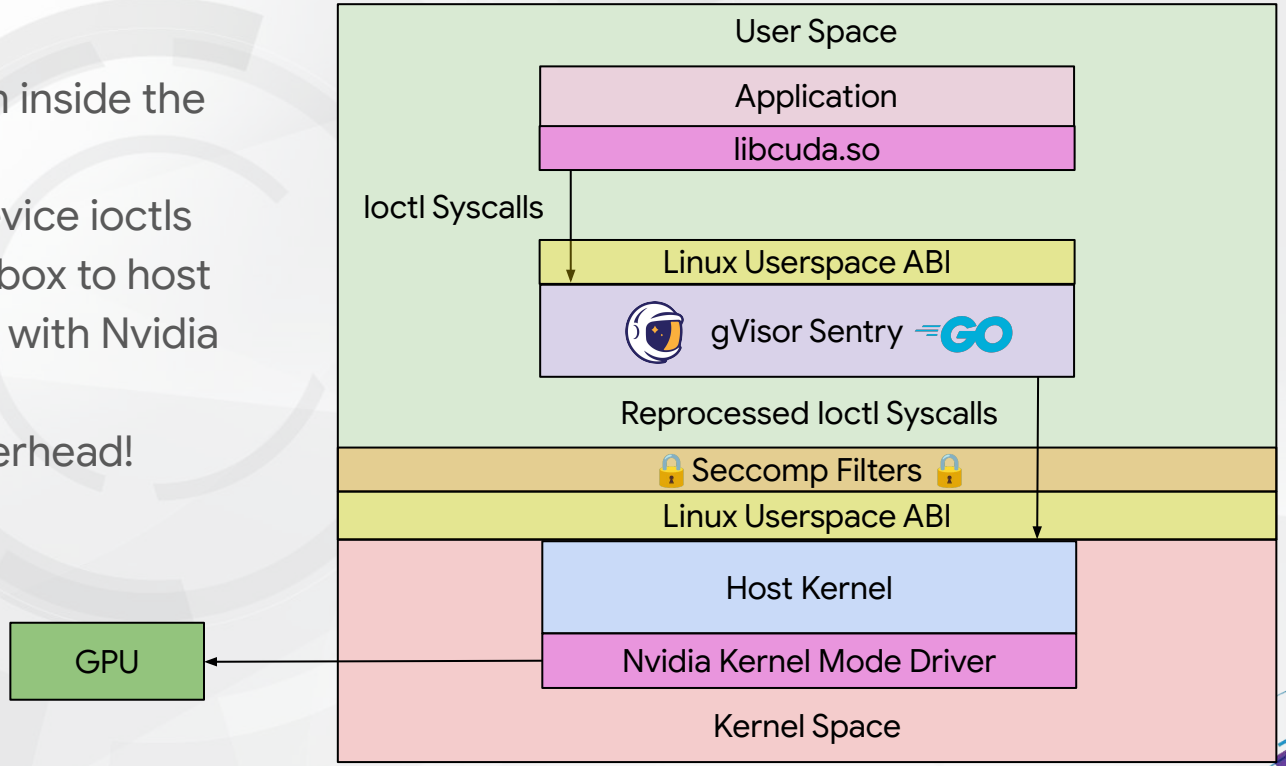
Year	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	File Inclusion	CSRF	XXE	SSRF	Open Redirect	Input Validation
2019	4	2	0	0	0	0	0	0	0	0	1
2020	11	18	0	0	0	0	0	0	0	0	13
2021	32	45	0	0	1	0	0	0	0	0	7
2022	17	15	0	0	0	0	0	0	0	0	39
2023	4	1	0	0	0	0	0	0	0	0	1
2024	0	0	0	0	0	0	0	0	0	0	0
Total	68	81			1						61

Scenario: Protecting Against Software Supply Chain Vulnerabilities During Model Serving

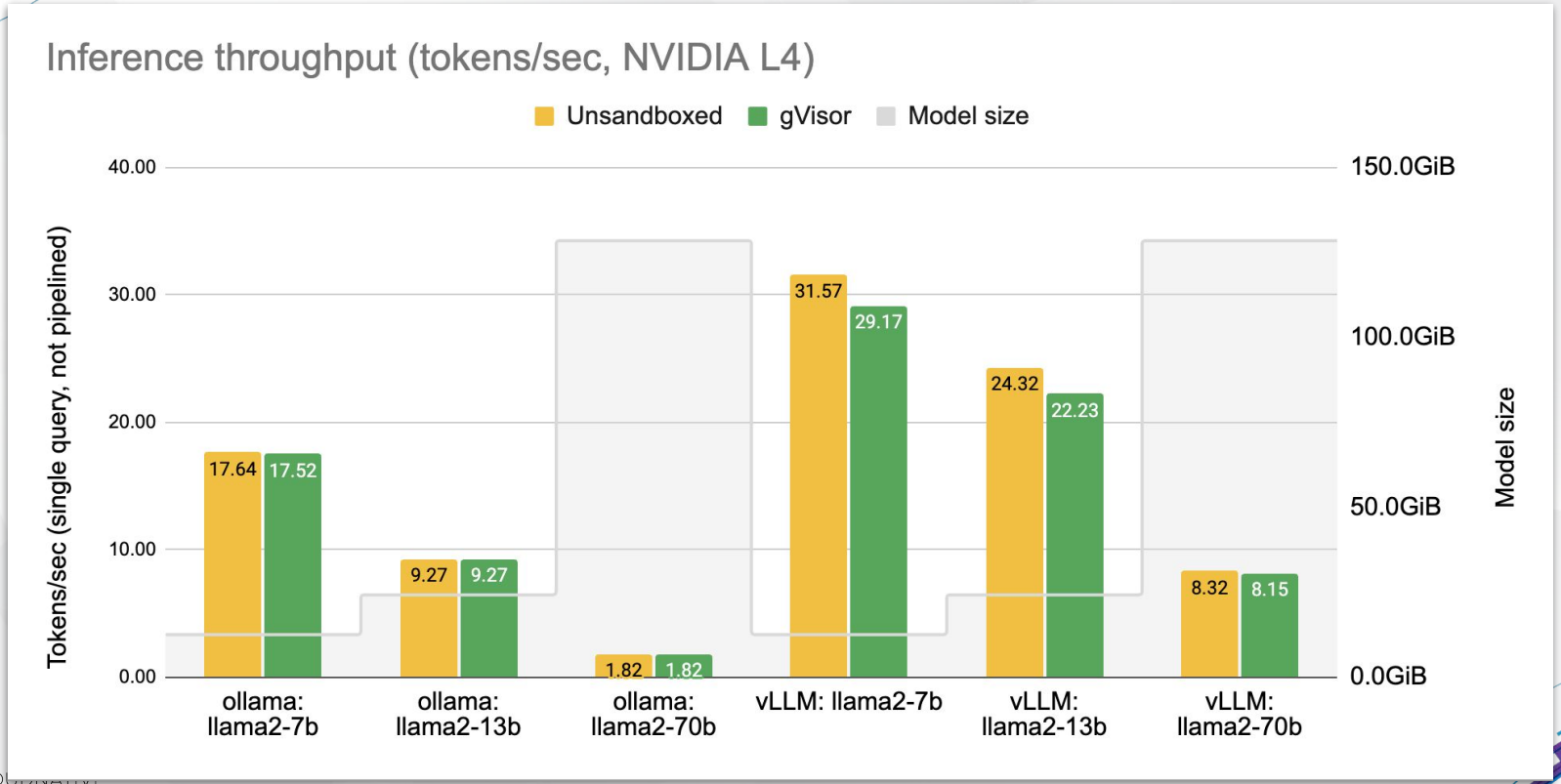
- What's your threat landscape?
  - Vulnerability in pytorch/TF/JAX gives access sensitive data in cluster
- What environment are you running in?
  - Likely some beefy servers for doing inference
- Can you afford higher resource costs?
  - Expensive servers. We want to minimize workload time.
- How long will containers run for?
  - Short, bursty workloads
- What kind of work is your container doing?
  - Lots of allocating, mapping, and moving memory on GPU
  - Spinning, waiting for GPU to finish
  - But gVisor doesn't have a device model, right...?

Introducing: NVProxy

- Use GPU from inside the sandbox
- Proxy GPU device ioctls through sandbox to host
- Compatibility with Nvidia docker tools
- Negligible overhead!



Introducing: NVProxy



NVProxy: Limitations

- Reliant on Nvidia's kernel mode driver
- Userspace/Kernel driver API is not guaranteed to be stable
- Constant maintenance required to keep this working

- It's always still important to protect against "normal" vulnerabilities
- Sentry + Seccomp both block unproxied ioctls
- Recent security analysis: **50% of GPU driver CVEs blocked by gVisor**

NVProxy: Opportunities

- Safely schedule more co-tenant pods on a single node
- Serve untrusted models
- gVisor checkpoint/restore + cuda-checkpoint
  - What can you do with efficient C/R?
    - Fast scale out
    - Eliminate cold starts
    - Live migration
    - Rewind
    - Time slicing
- GPU defense in depth
  - Collect metrics on GPU ioctls



Questions