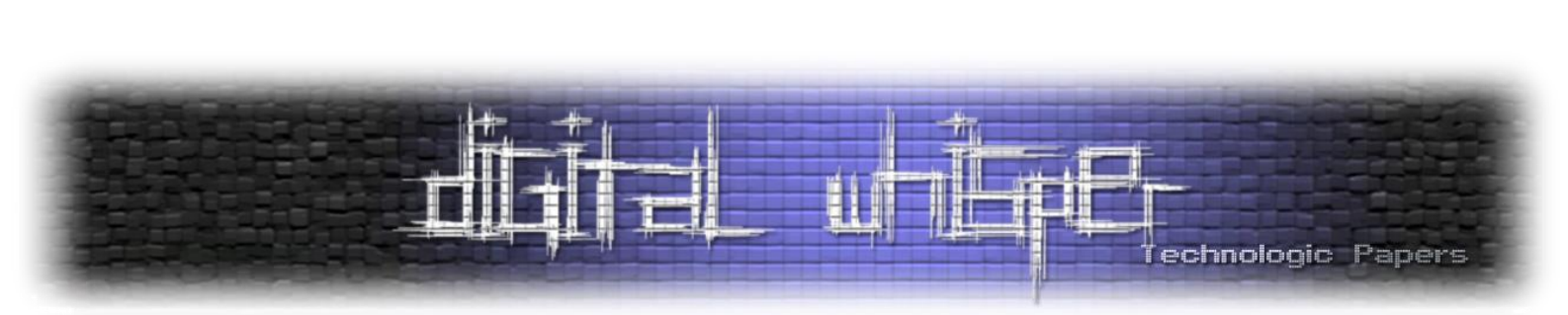


Docker - מנפצים את הקופסה השחורה

מאת עמית בוצר

הקדמה

בין אם אתם מפתחים, חוקרים, מנהלי מערכת, בודקי תוכנה או אנשים טכנולוגיים אחרים, בוודאי שמעתם על Docker. אעיז ואגיד שסביר כי הרבה מכם אף השתמשתם בכלי זה. אך האם אי פעם עצרתם לחשוב איך הדבר המופלא הזה פועל? במאמר זה, "ננפץ" את הקופסה השחורה, ונבין באופן פרקטי כיצד החלקים המרכיבים את Docker פועלים.

מאמר זה לא דורש ידע מקדים רב. אני יוצא מנקודת הנחה שהקורא מכיר את לינוקס בצורה בסיסית, וגם את Docker. אם אתם לא בטוחים לגבי אחד מהשניים, מוזמנים לקרוא את המאמר מאת ניצן שולמן.

במאמר זה, נסקור את הטכנולוגיה מאחורי Docker: הפיצ'רים השונים בקרנל והכלים אשר בשימוש על ידי Docker.

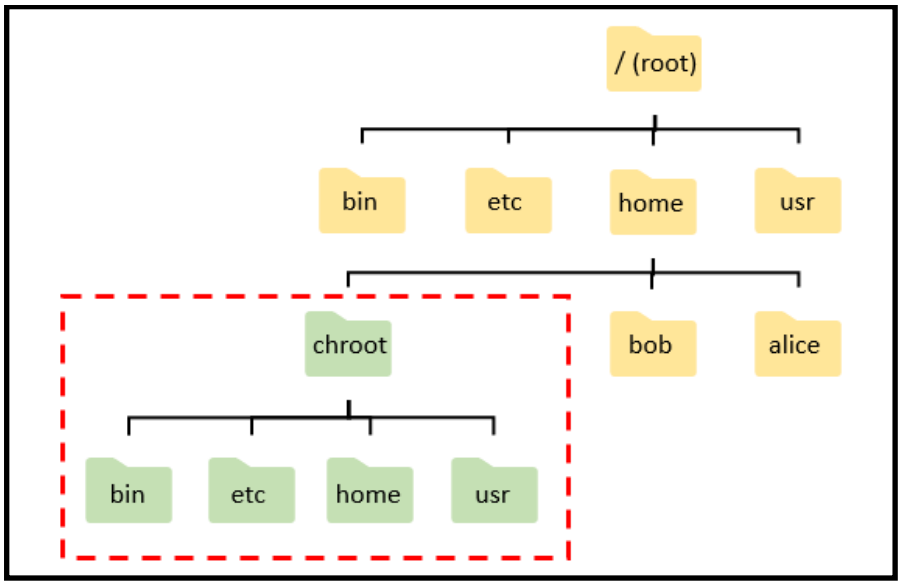
בסיום מאמר זה, יהיה בידכם הבנה מוצקה לגבי האופן שבו Docker פועל מאחורי הקלעים. תוכלו להשתמש בידע זה כדי להשתמש ב-Docker בצורה יעילה יותר, וגם לפתור בעיות שונות שעולות תוך שימוש בו.

טכנולוגיה

בפרק זה של המאמר, אסקור את הפיצ'רים השונים בקרנל הלינוקסי והכלים אשר בשימוש על ידי Docker. עבור כל נושא, אסביר כיצד הוא פועל ולאחר מכך אדגים בעזרת ניסוי כדי ליצור הבנה עמוקה יותר. בחלק מן הפרקים גם אדגים כיצד ניתן להשתמש בפיצ'ר ב-Docker.

Chroot

לצורך בידוד בקונטיינר, הדבר הראשון שאנו צריכים לעשות הוא להשיג מערכת קבצים שונה משל המחשב המארח. הפקודה Chroot מאפשרת להריץ פקודה מסוימת עם root directory מוגדר. בעזרת פקודה זו, ניתן להיכנס למערכת קבצים מבודדת:



[מקור]

לצורך ההדגמות הקרובות, הורדתי מערכת קבצים מכווצת, ועשיתי לה Decompress לתוך התיקייה FS:

```
$ ls fs
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
```

אגדיר את רצף הפעולות הבא "כניסה לקונטיינר":

```
$ sudo chroot fs
[chroot]# export PATH=/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

הערה #1: כעת, נסמן תהליך בתוך ה-chroot עם prefix של [chroot].

הערה #2: מתבצע שינוי של המשתנה PATH כדי שיתאים למערכת הקבצים הספציפית שהורדתי.

Namespaces

פיצ'ר זה של הקרנל יוצר הפרדה בין משאבים של כל המערכת. תהליך אשר נמצא בתוך Namespace מסוים מרגיש כאילו יש לו את הסביבה המבודדת שלו מבחינת משאבי מערכת. כל שינוי בסביבה זו גלוי לכל תהליך אחר אשר נמצא איתו באותו ה-Namespace. אך שינויים ב-Namespace אחד לא גלויים לתהליך הנמצא ב-Namespace אחר.

קיימים 8 Namespaces נכון להיום: Cgroup, IPC, Network, Mount, PID, TIME, User, UTS.

מערכת לינוקס מתחילה עם Namespace יחיד מכל סוג. כל תהליך, אם לא הוגדר אחרת, נוצר בתוך ה-Namespace הראשוני. אך תהליך רשאי ליצור Namespaces חדשים או להיכנס אל תוך Namespaces אחרים.

נדגים את השימוש ב-PID Namespace.

אם נבצע "כניסה לקונטיינר", ונספור את כמות התהליכים, נראה כי כמות התהליכים בתוך הקונטיינר זהה לכמות התהליכים מבחוץ לקונטיינר.

מבחוץ לקונטיינר:

```
$ ps aux | wc -l
177
```

בתוך הקונטיינר:

```
[chroot]# ps aux | wc -l
177
```

הסבר הפקודה: `ps aux` מחזיר לנו רשימה של תהליכים. אנחנו מעבירים את הפלט ל `wc -l` אשר מחזיר את כמות השורות.

כרגע, אין הפרדה ב-PIDs כאשר אנחנו בתוך הקונטיינר ומחוצה לו. אז בואו ננסה מהתחלה, עכשיו נשתמש בפקודה unshare כדי להריץ תוכנית עם PID Namespace חדש, ונראה את ההשפעה:

```
$ sudo unshare --pid --fork /bin/bash
```

נבצע "כניסה לקונטיינר", ולאחר מכן נציב מחדש את `/proc`, מערכת הקבצים הוירטואלית (לא נמצאת בדיסק הקשיח) אשר שומרת מידע על תהליכים ועוד:

```
[chroot]# mount -t proc proc /proc
```

וכעת נבדוק את כמות התהליכים בתוך הקונטיינר:

```
[chroot]# ps aux | wc -l
5
```

נבחין כי בתוך הקונטיינר יש רק כמה תהליכים בודדים. הצלחנו לבודד את ה-PID של התהליכים שבתוך הקונטיינר!

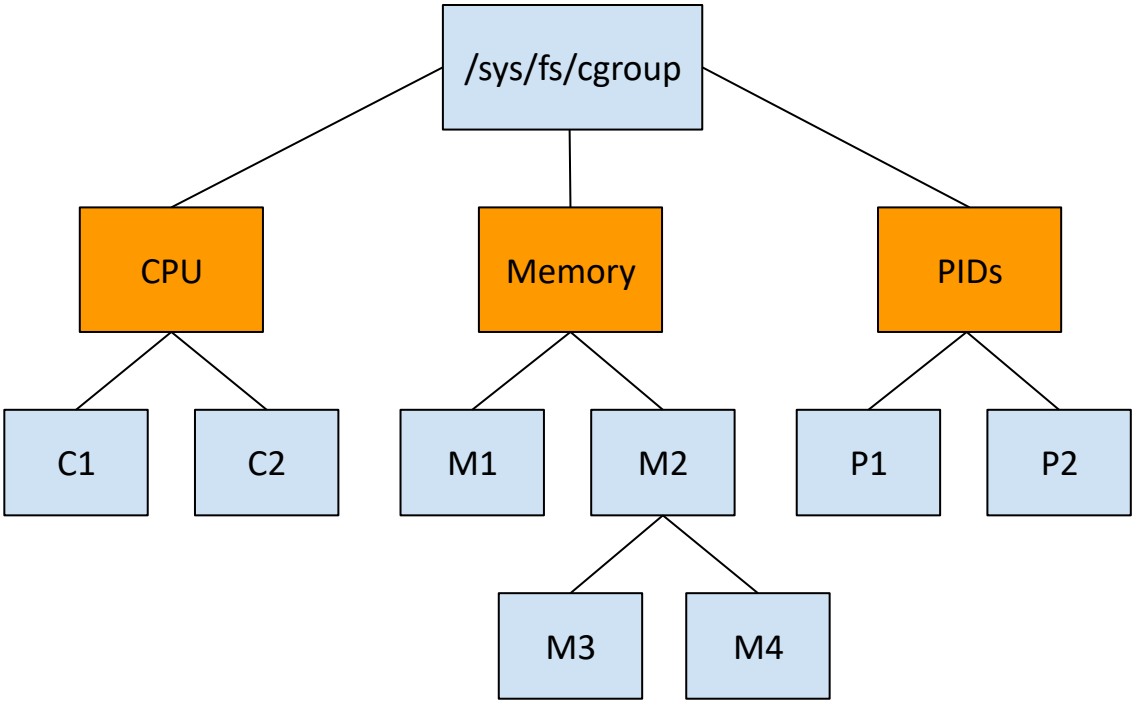
Cgroups

פיצ'ר זה מאפשר להגביל ולנטר את השימוש של תהליכים במשאבי מערכת (כגון זיכרון, מעבד, כמות תהליכים). החלוקה של קבוצות cgroups התהליכים פועלת על פי היררכיה, כך שניתן ליצור רמות שונות.

תיקיית הבסיס של cgroups היא:

```
/sys/fs/cgroup
```

אילוסטרציה:



למה זה יכול להיות שימושי? דמיינו שרת שמריץ כמה קונטיינרים, אשר כל אחד מהם מטפל באתר אחר. אתר יחיד עובר התקפת מניעת שירות. אם לא נגביל את כמות המשאבים של קונטיינר זה, הקונטיינר ישתמש בכל המשאבים של המערכת שברשותו כדי להתמודד עם כל הבקשות, והשרת כולו יעבור מניעת שירות (כולל קונטיינרים של אתרים אחרים בכלל!).

בואו ננסה לראות כיצד אפשר להשתמש ב-cgroups. למען ההדגמה, ננסה לבדוק האם cgroups יוכל להגן עלינו מהתקפת fork bomb. קודם כל נסתכל על תיקיית הבסיס של cgroups:

```
$ ls /sys/fs/cgroup/
blkio       cpu,cpuacct     freezer   net_cls           perf_event   systemd
cpu         cpuset          hugetlb   net_cls,net_prio  pids         unified
cpuacct     devices         memory    net_prio          rdma
```

ב-fork bomb המטרה היא ליצור הרבה מאוד תהליכים.

כדי למנוע התקפה זו, נשתמש בקבוצת ה-cgroups של PIDs כדי להגביל את מספר התהליכים שתהליך יכול ליצור:

```
$ ls /sys/fs/cgroup/pids/
Docker        init.scope        system.slice      user.slice
cgroup.clone_children           cgroup.procs      cgroup.sane_behavior
notify_on_release               release_agent     tasks
```

כעת ניצור קבוצת cgroups חדשה בה נגדיר חוקים חדשים. כל מה שאנחנו צריכים לעשות זה ליצור תיקייה חדשה, וכל הקבצים הרלוונטים יווצרו לבד:

```
$ sudo mkdir /sys/fs/cgroup/pids/fork-bomb-defense/
$ ls /sys/fs/cgroup/pids/fork-bomb-defense/
cgroup.clone_children   cgroup.procs      notify_on_release
pids.current            pids.events       pids.max                tasks
```

הקבצים הרלוונטיים לנו כרגע הם tasks ו-pids.max. קודם כל נגדיר את כמות התהליכים המקסימלית:

```
$ sudo echo "10" > /sys/fs/cgroup/pids/fork-bomb-defense/pids.max
```

לאחר מכן אנחנו צריכים לשייך את התהליך אותו אנחנו רוצים להגביל לקבוצת ה-cgroups החדשה שיצרנו. נשתמש בקובץ tasks המחזיק רשימה של תהליכים אשר שייכים ל-cgroup הנוכחי. נכתוב לקובץ זה את המספר של התהליך הנוכחי:

```
$ sudo echo $BASHPID > /sys/fs/cgroup/pids/fork-bomb-defense/tasks
```

עכשיו נריץ את ה-fork bomb, נקווה שהמחשב לא יקרוס (אל תנסו בבית):

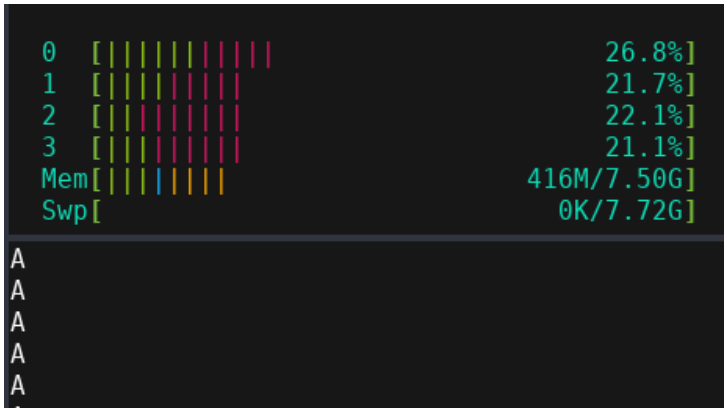
```
# :(){ :|:& };:
[1] 40047
bash: fork: retry: Resource temporarily unavailable
bash: fork: retry: Resource temporarily unavailable
bash: fork: retry: Resource temporarily unavailable
bash: fork: retry: Resource temporarily unavailable
bash: fork: retry: Resource temporarily unavailable
bash: fork: retry: Resource temporarily unavailable
```

כפי שניתן לראות, התהליך מנסה לשכפל את עצמו, אך הוא לא מצליח! הצלחנו למנוע את ההתקפה! המחשב שלי לא קרס ☺.

בוא נדגים שימוש של cgroups ב-Docker. נגדיר שהקונטיינר יוכל להשתמש לכל היותר ב-20% מתוך ה-cpu שלנו, ובתוך הקונטיינר נדפיס "A" בלולאה אינסופית (פעולה אשר צורכת הרבה cpu):

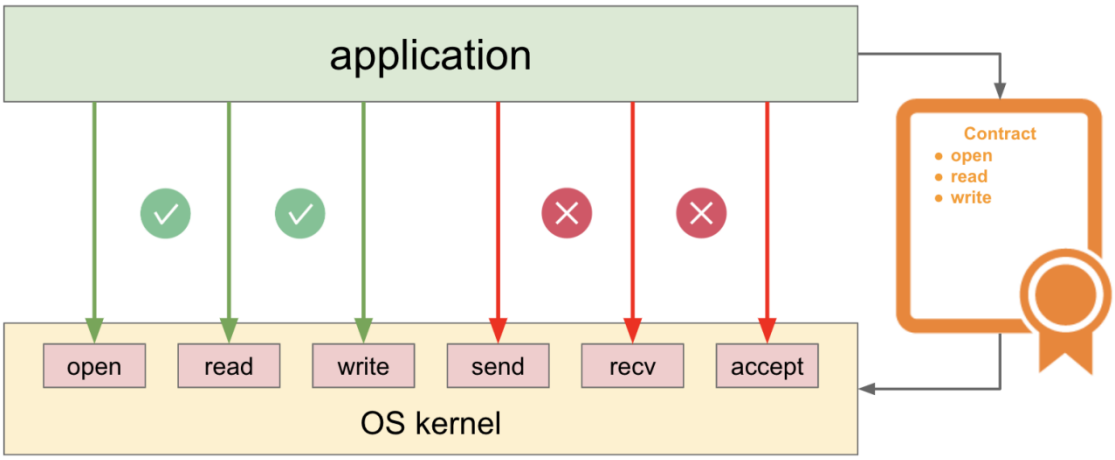
```
$ sudo docker run --rm -it --cpus 0.2 bash
bash-5.0# while true; do echo "A"; done;
```

מצורפת תמונה כתוצאה מההרצה. בתמונה 2 טרמינלים. בטרמינל התחתון הקונטיינר אשר בו מתרחשת ההדפסה בלולאה אינסופית. בטרמינל העליון הפלט של process viewer, ניתן לראות כי כל ליבה של ה-cpu מגיעה עד 20% בערך:



Seccomp

באופן דיפולטיבי, תהליך פשוט מ-userland רשאי לבצע הרבה syscalls. הפיצ'ר Seccomp מגדיר אילו syscalls תהליך רשאי לבצע ואילו syscalls לא רשאי לבצע. ניתן להגדיר מה יקרה עבור תהליך שיחרוג מחוקי ה-Seccomp, לדוגמה, יהרג על ידי SIGKILL שנשלח מהקרנל. בזכות Seccomp ניתן למזער את החשיפה של הקרנל לתהליכים:



[מקור]

למה זה חשוב? ישנם syscalls מסוימים הנחשבים מסוכנים יותר. חלקם מוגדרים כך משום שמאפשרים פוטנציאלית לבצע שינויים במחשב המארח (מחוץ לקונטיינר!). למשל, syscalls הקשורים במניפולציות של הקרנל בעזרת kernel modules, והם: init_module, finit_module, delete_module. בנוסף, ה-syscall בשם kexec_load נחסם משום שמאפשר להחליף את הקרנל הנוכחי (אשר יטען לאחר reboot).

למען ההדגמה הקרובה, אנחנו נעשה שימוש בפונקציונליות המתאפשרת בעזרת libseccomp. ניתן להתקין ממנהל החבילות השגרתי, למשל בעזרת Pacman:

```
$ sudo pacman -S libseccomp
```

נכתוב תוכנה קצרה, seccomp.c:

```
#include <seccomp.h>
#include <unistd.h>
#include <string.h>

#define STDOUT 1

int main(int argc, char* argv[]) {
    // block all syscalls
    scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_KILL);

    // allow basic needed syscalls
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(fstat), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit_group), 0);

    // no arguments
    if(argc-1 == 0) {
        // allow write syscall
        seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 0);
    }

    seccomp_load(ctx);

    const char* msg = "Hello there!\n";
    write(STDOUT, msg, strlen(msg));

    return 0;
}
```

בתוכנה זו אנחנו קודם כל יוצרים seccomp filter שחוסם את כל ה-syscalls. לאחר מכן מאשרים 2 syscalls אשר הכרחיים לריצה תקנית של התוכנית. אחר כך יש בדיקה - אם לתוכנית אין arguments, אנחנו מאשרים גם את write. לאחר מכן אנחנו טוענים את הפילטר שיצרנו (מפעילים אותו). ולסיום, אנחנו מדפיסים ל-STDOUT הודעה בעזרת הפונקציה write (המהווה wrapper ל-write syscall).

נקמפל:

```
$ gcc seccomp.c -lseccomp -o seccomp
```

כאשר נריץ את התוכנית **בלי** ארגומנטים, נקבל:

```
$ ./seccomp
Hello there!
```

כאשר נריץ את התוכנית **עם** ארגומנטים, נקבל:

```
$ ./seccomp ARG
Bad system call (core dumped)
```

נראה שהפילטר עבד.

בואו נסתכל בדיוק מה קרה בעזרת strace - פקודה שעוקבת אחרי syscalls של תוכנית:

```
$ strace ./seccomp ARG
...
write(1, "Hello there!\n", 13)          = ?
+++ killed by SIGSYS (core dumped) +++
```

ניתן לראות כי התהליך השתמש ב-write syscall, ואז נהרג על ידי הסיגנל SIGSYS. עכשיו כשאנחנו מבינים איך Seccomp עובד, בוא נשתמש בזה ב-Docker. ב-Docker ניתן להגדיר פרופיל Seccomp - אשר מגדיר את החוקיות של ה-Syscalls בקונטיינר. אם לא הוגדר אחרת, כאשר מריצים את Docker הוא משתמש בפרופיל הדיפולטיבי.

בואו נשנה את הפרופיל הדיפולטי, ונאפשר את ה-syscall שפגשנו קודם - unshare, אשר חסום מברירת מחדל. נשמור בקובץ custom.json. החלק הרלוונטי מתוך הקובץ:

```
"syscalls": [
    {
        "names": [
            "unshare",
```

נבדוק מה קורה כאשר מריצים את הפקודה unshare בתוך Docker עם הפרופיל הדיפולטיבי:

```
$ sudo docker run bash unshare
unshare: unshare(0x0): Operation not permitted
```

לא ניתן להשתמש ב-syscall כפי שניתן לראות. ועכשיו עם הפרופיל החדש שיצרנו:

```
$ sudo docker run --security-opt seccomp=./custom.json bash unshare
$
```

אין שגיאה, הצלחנו להוסיף את ה-syscall.

Capabilities

בעבר, תהליך בלינוקס היה יכול לרוץ כ-root (עם uid 0), או לרוץ לא כ-root (עם uid שונה מ-0). לתהליך שרץ כ-root היה הרשאות לבצע כל פעולה שרצה, והיה יכול לעקוף את רוב ההגבלות הקיימות, בניגוד לתהליך שרץ לא כ-root. כדי לאפשר למשתמשים רגילים (שונים מ-root) להריץ תהליכים אשר דורשים הרשאות גבוהות, ניתן להשתמש ב-setuid bit בבינארי. ה-attribute הזה מאפשר לתהליך לרוץ עם הרשאות של יוצר הבינארי, כך משתמש יכול לרוץ עם הרשאות גבוהות מאלו שיש לו. עם זאת, תהליכי setuid מוגדרים מסוכנים יותר, משום שבמקרה וינוצלו בהם חולשות, ניתן להשיג privilege escalation.

אז הוצע הרעיון של capabilities. מפרקים את כל ההרשאות של root, לקבוצות קטנות יותר - ואלו בעצם capabilities. לכל capability פונקציונליות שונה. תהליך אשר צריך הרשאות מסוימות בשביל לרוץ, יכול

לקבל רק אותן, הוא לא צריך לקבל הרשאות אחרות אשר לא רלוונטיות אליו. כאשר תהליך רץ עם uid של 0 (במילים אחרות, root), הוא רץ עם כל ה-capabilities.

ניתן נבדוק מה ה-capabilities של תהליך מסוים בעזרת הקובץ:

```
/proc/pid/status
```

קודם כל נבדוק כאשר אנחנו non-root:

```
$ grep Cap /proc/$BASHPID/status
CapInh:    0000000000000000
CapPrm:    0000000000000000
CapEff:    0000000000000000
CapBnd:    0000003fffffffff
CapAmb:    0000000000000000
```

ולאחר מכן כאשר אנחנו root:

```
$ sudo su
# grep Cap /proc/$BASHPID/status
CapInh:    0000000000000000
CapPrm:    0000003fffffffff
CapEff:    0000003fffffffff
CapBnd:    0000003fffffffff
CapAmb:    0000000000000000
```

נשתמש בפקודה capsh כדי להבין מה המספרים האלו אומרים:

```
$ capsh --decode=0000003fffffffff
0x0000003fffffffff=cap_chown,cap_dac_override,cap_dac_read_search,cap_fo
wner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_imm
utable,cap_net_bind_service,cap_net_broadcast,cap_net_admin,cap_net_raw,
cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,c
ap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_
sys_resource,cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_aud
it_write,cap_audit_control,cap_setfcap,cap_mac_override,cap_mac_admin,ca
p_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read
```

אלו כל ה-capabilities. ההבדל הוא שב-root ה-capabilities מופיעים ב-CapPrm, אשר המשמעות Permitted Capabilities, וגם ב-CapEff, אשר המשמעות Effective Capabilities. זאת אומרת שאלו capabilities שמותרים לשימוש והם בשימוש. זה מסתדר עם מה שאמרנו קודם, ל-root יש את כל ה-capabilities.

כעת, בואו נראה שימוש פרקטי ל-capabilities. למען ההדגמה כתבתי שרת פשוט שמאזין בפורט 80:

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello there from port 80\n")
}

func main() {
    http.HandleFunc("/", handler)
    err := http.ListenAndServe(":80", nil)
    if err != nil {
     log.Fatal(err)
    }
}
```

כאשר ננסה להריץ את השרת:

```
$ ./server
listen tcp :80: bind: permission denied
```

אין לנו הרשאה להאזין בפורט 80. הסיבה היא שדרוש ה-capability בשם CAP_NET_BIND_SERVICE בשביל להאזין בפורט נמוך מ-1024. אז בואו ניתן לקובץ את ה-capability:

```
$ sudo setcap cap_net_bind_service+ep server
```

נבדוק שאכן הוא קיבל את ה-capability:

```
$ getcap server
server = cap_net_bind_service+ep
```

ועכשיו בואו נריץ את השרת:

```
$ ./server
```

מתהליך אחר:

```
$ curl localhost:80
Hello there from port 80
```

הצלחה! הוספנו לשרת את ה-capability הבודד. זה יותר בטוח מאשר להריץ את השרת עם root.

בואו נדגים שימוש ב-Docker ל-Capabilities. בואו קודם כל נבדוק מה ה-capabilities הדיפולטיביים:

```
$ sudo docker run bash
bash-5.0# grep Cap /proc/$$/status
CapInh:    00000000a80425fb
CapPrm:    00000000a80425fb
CapEff:    00000000a80425fb
CapBnd:    00000000a80425fb
CapAmb:    0000000000000000
```

נבדוק את המשמעות בעזרת capsh:

```
$ capsh --decode=00000000a80425fb
0x00000000a80425fb=cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_
kill,cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,
cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap
```

בואו נדגים מה קורה כאשר נוריד את ה-capability בשם CAP_SYS_CHROOT:

```
$ sudo docker run --cap-drop sys_chroot bash
bash-5.0# chroot /
chroot: can't change root directory to '.': Operation not permitted
```

הפקודה נכשלה, הוצאנו את האפשרות לביצוע chroot בתוך הקונטיינר.

Mandatory Access Control

לפני שאפרט על MAC, נצטרך להכיר את ה-framework בשם Linux Security Models. ה-framework מאפשר תמיכה של הקרנל בהרחבות אשר נוגעות באבטחה שלו. השימוש העיקרי ב-LSM הוא של MAC, אשר מהווים שכבת הגנה, ומספקים מדיניות אבטחה לגבי עצמים במערכת. ללא שיטת MAC בקרנל, השיטה הדיפולטית היא capabilities.

אז איך בעצם MAC עובד? MAC מגדיר עבור כל עצם במערכת רמת אבטחה, מדיניות מסוימת. העצם לא יכול לחרוג מהמדיניות שלו, רק administrator יכול לשנות את רמות האבטחה.

במאמר זה נסקור את AppArmor, שיטת ה-MAC הנפוצה ביותר בקרב קונטיינרים. אם אתם מעוניינים ב-SELinux, אתם מוזמנים לקרוא את המאמר מאת עדי מליאנקר.

ב-AppArmor, ניתן להגדיר עבור תהליך פרופיל, אשר בו חוקים שונים. תהליך רשאי לבצע רק מה שהחוקים מתירים לו. עיקרון ה-blocklist הוא השיטה לפיה פועל AppArmor. בפרופיל ריק, הכל חסום. לאחר מכן ניתן להוסיף חוקים אשר מאשרים פעולות מסוימות. עם זאת, ניתן לכתוב פרופיל AppArmor בצורת allowlist, וכך נהוג ב-Docker. יש לכך סיבות, למשל, לא ניתן לצפות אילו קבצים הולכים להיווצר בתוך קונטיינר, לכן אי אפשר מראש להכין allowlist לאותם הקבצים.

לפני שניגש לחשיבות של AppArmor ודוגמות לשימוש בו, נסתכל על ה-syntax של AppArmor Profiles. ה-syntax די אינטואיטיבי, למשל עבור קובץ, כותבים נתיב, ולאחר מכן הרשאות, בואו נסתכל על כמה דוגמות:

```
/etc/shadow r,
```

בחוק זה, הגדרנו read permission לקובץ /etc/shadow.

```
/dev/tty* rw,
```

בחוק זה, הגדרנו read permission וגם write permission לכל קובץ אשר מתאים לתבנית.

```
/dev/tty[כל דבר]
```

בשביל ה-syntax המלא ניתן לגשת לקישור.

מה החשיבות של AppArmor ב-Docker? שוב, כמו ב-Seccomp, ניתן למנוע גישה למחשב המארח, או ביצוע פעולות מסוכנות. אם נסתכל על התבנית הדיפולטית של AppArmor, נוכל למצוא כמה חוקים החשובים לאבטחה. למשל:

```
deny @{PROC}/kcore rwklx,
```

לפי חוק זה, לא ניתן לקרוא (r), לכתוב (w), לנעול (k), ליצור לינק (k), או להריץ (x) את הקובץ:

```
/proc/kcore
```

קובץ זה מספק dump של הזיכרון הפיזי במערכת, לכן ההגבלות עליו. מידע זה לא צריך להיות נגיש לתהליך שבוך קונטיינר.

למען ההדגמה הקרובה, נשתמש בספרייה apparmor. בואו ניצור פרופיל מסוים עבור bash, בו נתיר הכל חוץ מקריאה של קובץ מסוים:

```
$ cat bash.prof
#include <tunables/global>

"/usr/bin/bash" {
  # Allow all rules
  capability,
  network,
  mount,
  remount,
  umount,
  pivot_root,
  ptrace,
  signal,
  dbus,
  unix,
  file,

  # deny reading /etc/hostname
  deny /etc/hostname r,
}
```

כעת נטען את הפרופיל:

```
$ sudo apparmor_parser -r bash.prof
```

ונבדוק שהוא אכן נטען:

```
$ sudo apparmor_status | grep bash
   /usr/bin/bash
   /usr/bin/bash (710)
   ...
```

לפי הפלט של apparmor_status, ניתן לראות שיש פרופיל נטען עבור הקובץ /usr/bin/bash, וגם הפרופיל טעון עבור התהליך עם pid של 710. עכשיו בואו ננסה לקרוא את הקובץ אשר חסמנו בפרופיל מתוך תהליך bash חדש:

```
$ cat /etc/hostname
cat: /etc/hostname: Permission denied
```

לא ניתן לקרוא את הקובץ! אולי ננסה עם root?

```
$ sudo su
# cat /etc/hostname
cat: /etc/hostname: Permission denied
```

עדיין לא ניתן לקרוא את הקובץ, הפרופיל שלנו עובד!

להסרת הפרופיל:

```
$ sudo apparmor_parser -R bash.prof
```

עכשיו בואו נראה כיצד ניתן להשתמש ב-AppArmor ב-Docker. בואו נריץ את Docker עם הפרופיל שניצור עבורו. נשתמש בפרופיל הקודם, רק נעשה שינוי אחד: נגדיר את הפרופיל לפי שם. מקודם הגדרנו לפי נתיב של קובץ:

```
profile custom-prof {
...
```

נטען את הפרופיל בעזרת apparmor_parser, ולאחר מכן נריץ קונטיינר עם פרופיל זה:

```
$ sudo docker run --security-opt apparmor=custom-prof bash
```

ננסה לקרוא את הקובץ שנחסם בפרופיל:

```
bash-5.0# cat /etc/hostname
cat: can't open '/etc/hostname': Permission denied
```

לא ניתן לקרוא את הקובץ.

UnionFS

UnionFS הוא איחוד של מספר מערכות קבצים לתוך תיקייה אחת, למען יצירת מערכת קבצים וירטואלית חדשה. פעולה זו מתאפשרת בזכות הצבה (Mount). פעולה זו של הצבת מספר תיקיות לתוך תיקייה אחת נקראת Union Mount.

עיקרון חשוב של UnionFS הוא copy-on-write. לפי עיקרון זה, מספר תהליכים שונים יכולים להשתמש בעותקים זהים של משאב, במקום לשכפל אותו. עם זאת, כאשר רוצים לשנות את המשאב החדש (לכתוב), צריך ליצור עותק שלו. מכאן מגיע השם, copy-on-write.

במאמר זה נתייחס ל-overlay2, סוג מסוים של storage driver אשר מממש UnionFS. הוא גם מומלץ לשימוש לפי Docker. ל-overlay2 יש 4 סוגי שכבות:

- **LowerDir/Base** - אלו שכבות שהן מוצבות כ-Read Only. מהוות שכבת הבסיס של ה-file system שנוצר.
- **UpperDir/Diff** - שכבה אשר שומרת את כל השינויים שנוצרו. כאשר יש שינוי ב-LowerDir (שהוא Read Only), נוצר עותק (copy-on-write) שנשמר בשכבה זו.
- **WorkDir** - לשימוש פנימי על ידי overlay2, נועד כדי להכין (prepare) קבצים כאשר הם מוחלפים בין השכבות.
- **Merged/Overlay** - השכבה בה מוצבות כל השכבות. מוצבת כ-read write.

ניצור את התיקיות:

```
$ mkdir lower1 lower2 upper work merged
```

ניצור קבצים ונשים אותם ב-LowerDirs:

```
$ echo "I'm in lower1" > lower1/file1
$ echo "I'm in lower2" > lower2/file2
```

כעת נבצע Union Mount:

```
$ sudo mount -t overlay overlay -o
lowerdir=lower1/:lower2/,upperdir=upper,workdir=work/ merged/
```

הגדרנו בעזרת הדגל t את הסוג (type) של ה-mount. בעזרת הדגל o את האופציות (options) של ה-mount. ולבסוף כתבנו את התיקייה merged, תיקיית היעד של ה-mount.

בוא נבדוק את תוכן התיקיות:

```
$ sudo tree
.
├── lower1
│   └── file1
├── lower2
│   └── file2
├── merged
│   ├── file1
│   └── file2
├── upper
└── work
    ├── index
    └── work
```

בואו ניצור קובץ חדש, ונסתכל שוב על התיקיות:

```
$ echo "I was created in the merged dir" > merged/file3
$ sudo tree
.
├── lower1
│   └── file1
├── lower2
│   └── file2
├── merged
│   ├── file1
│   ├── file2
│   └── file3
├── upper
│   └── file3
...
```

ניתן לראות כי נוצר עותק ב-Upper Directory, שכבת השינויים. כעת בואו נשנה תוכן של קובץ מהשכבה התחתונה:

```
$ echo "I was changed in the merged dir" > merged/file1
$ sudo tree
.
├── lower1
│   └── file1
├── lower2
│   └── file2
├── merged
│   ├── file1
│   ├── file2
│   └── file3
├── upper
│   ├── file1
│   └── file3
...
```

ניתן לראות כי נוצר עותק ב-Upper Directory. אך התוכן של הקובץ המקורי ב-Lower Directory נשמר:

```
$ cat lower1/file1
I'm in lower1
$ cat upper/file1
I was changed in the merged dir
```

לסיום, נבצע unmount:

```
$ sudo umount merged
```

בואו נראה כיצד השכבות נראות כאשר אנחנו מריצים קונטיינר ב-Docker. נוצרה לנו תיקייה חדשה:

```
$ sudo ls
/var/lib/docker/overlay2/a56f10630dac0486ea26387afc091194da5d6342aa597b1
57eb31ce1d98cfba8
diff  merged  work  link  lower
```

ניתן לראות בתיקייה זו את כל השכבות עליהן דיברנו.

סיכום

במאמר זה סיקרתי המרכיבים של Docker, והסברתי לעומק כיצד כל אחד מהם פועל תוך שימוש בדוגמות.

המאמר כלל הרבה מרכיבים, אסכם בקצרה את כולם:

- **Chroot** - שינוי תיקיית בסיס
- **Namespaces** - הפרדת המשאבים של המערכת מבחינת סביבה
- **Cgroups** - הגבלת השימוש של תהליכים במשאבי מערכת
- **Seccomp** - הגבלת syscalls
- **Capabilities** - פיצול הרשאות
- **MAC** - מדיניות עבור עצמים במערכת
- **UnionFS** - הצבה של מספר מערכות קבצים לתיקייה אחת

מה הדבר הבא שאתם יכולים לעשות? עם הידע שבידיכם עכשיו, תוכלו להשתמש ב-Docker בצורה יותר יעילה, ולהתמודד עם בעיות ביתר קלות. אם תרצו אפילו, אני מאמין שתוכלו לפתח Docker משלכם!

על המחבר

עמית בוצר, בן 18 מלש"ב. בוגר תוכנית מגשימים. בזמני החופשי פותר CTFs, מפתח פרויקטים שונים, ומנהל את ערוץ היוטיוב BigBoost.

- אימייל: botzer.2002@gmail.com
- גיטהאב: https://github.com/BigB00st

למדתי על נושא זה תוך מימוש Docker מאפס ב-Go:

https://gitlab.com/amit-yuval/locker

רשימת מקורות

- https://docs.docker.com/get-started/overview/
- https://ericchiang.github.io/post/containers-from-scratch/
- https://blog.selectel.com/containers-security-seccomp/
- https://www.nccgroup.com/globalassets/our-research/us/whitepapers/2016/june/container_whitepaper.pdf
- https://blog.container-solutions.com/linux-capabilities-in-practice
- https://linux-audit.com/linux-capabilities-101/
- https://www.terriblecode.com/blog/how-docker-images-work-union-file-systems-for-dummies/