Automatic Isolation and Analysis Environment

Naor Maman, Shachar Markovich

Jerusalem College of Technology

Department of Computer Science and Software Engineering

Fundamentals of Software Security – Arie Haenel

2021, semester B

Abstract

Today, in the security world we often deal with files that we are not sure if they are malicious or not. Until this day, in order to analysis and research those kinds of files, we would set up a virtual machine, usually slow and nerve-wracking, in which we would install the software and research it. We were developed a much more effective method to do this – with an isolated Docker.

In this article we aim to answer our research questions:

1) What is Docker and how it is different from Virtual Machine?
2) What are files' dependencies and how to discover them?
3) How to set up an isolated Docker environment which give the opportunity of safe malware research in an efficient way?
4) Why Docker is more efficient from a Virtual Machine?

תוכן עניינים

מבוא – טרמינולוגיה

Virtual Machine

מכונה וירטואלית (VM) היא הדמיה של מערכת מחשב. מכונות וירטואליות מבוססות על ארכיטקטורות מחשב ומספקות פונקציונליות של מחשב פיזי. היישומים שלהם עשויים לכלול חומרה מיוחדת, תוכנה או שילוב.

מכונות וירטואליות של המערכת מספקות תחליף למכונה אמיתית. הם מספקים פונקציונליות הדרושה לביצוע מערכות הפעלה שלמות. Hypervisor, אשר הינה תוכנה המיועדת לטפל בהרצת מכונות וירטואליות וניהולן, משתמש בביצוע מקומי כדי לשתף ולנהל חומרה, מאפשר יצירת סביבות רבות המבודדות זו מזו, אשר קיימות באותה מכונה פיזית.

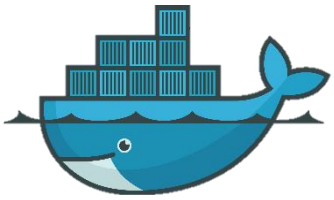
ישנם אמולטורים של מכונות וירטואליות, כגון QEMU, שנועדו לחקות (או "לחקות" כמעט) ארכיטקטורות מערכת שונות וכך לאפשר ביצוע יישומי תוכנה ומערכות הפעלה שנכתבו עבור מעבד או ארכיטקטורה אחרת. וירטואליזציה ברמת מערכת ההפעלה מאפשרת לחלק את משאבי המחשב דרך הליבה.

Container

וירטואליזציה ברמת מערכת ההפעלה הוא שיטת וירטואליזציה בשרתים שבה הליבה של מערכת ההפעלה מאפשרת את קיומם של מספר מרחבי משתמש מבודדים, במקום רק אחד. מופעים כאלה, אשר נקראים Container (קונטיינר), עשויים להיראות כמו שרת ממשי מנקודת מבט של משתמשיו.

בנוסף לבידוד מנגנונים, ליבת מערכת ההפעלה, ה-Kernel, מספקת לעיתים קרובות יכולות של ניהול משאבים כדי להגביל את ההשפעה של הפעולות בקונטיינר אחד על קונטיינרים אחרים.

Docker



הוא פרויקט קוד פתוח של תוכנה המספקת שכבת הפשטה ואוטומציה להתקנת והרצת יישומים בתוך container המנוהלים על ידי וירטואליזציה ברמת מערכת ההפעלה על גבי מגוון מערכות הפעלה, ביניהם Windows ו-Linux. המערכת פותחה על ידי חברת Docker Inc. בעיקר עבור לינוקס, ועושה שימוש בגישה לקרנל לינוקס וכן

במערכת טעינת קבצים, כדי לאפשר לקונטיינרים לרוץ כמופע אחד של לינוקס, וזאת כדי למנוע תקורה של הקמה ותחזוקה של מכונות וירטואליות (VM).

ה-Kernel של לינוקס מבודד יישום זה בסביבת מערכת ההפעלה, כולל כל התהליכים שרצים, הרשת, שמות משתמשים ומערכת הקבצים. ואינו דורש שימוש רב במשאבי זיכרון ומשאבי המעבד. בעצם ניתן לממש על מחשב אחד, מערכת ומסביבה כל המערך הנדרש דוגמת עם שרת אינטרנט, מסד נתונים וכדומה, והכל בתוך קונטיינר אחד. כמו כן, מספר קונטיינרים יכולים לרוץ על מחשב אחד בו-זמנית וגם ליצור תקשורת אחד מול השני.

File Dependencies

File Dependencies מאפשרת לקשר קובץ אחד או יותר לקובץ אחר אשר נעשה בו שימוש. זה מקל על ניהול הקשרים בין קבצים שמשתפים נתונים. כאשר קובץ מסוים רץ, הוא עושה שימוש בקבצים נוספים אשר שמה קיימות, לדוגמא פונקציות, אשר נעשה בהם שימוש. דבר זה קורה עקב הרצון לחסוך בקוד – דמיינו לעצמם בראש שבפרויקטים השונים שלכם, בכל פרויקט שלכם ובכל פעם שתרצו, לדוגמא, לשמור נתונים בקובץ log תצטרכו לממש בעצמם את הקוד שפותח קובץ וכותב אליו מידע. שיטת עבודה זו אינה נוחה ואינה אפשרית עקב העובדה שלא כל המפתחים יודעים כיצד לעשות זאת, מה גם שצורת הכתיבה לקובץ שונה בין מערכות הפעלה שונות.

בדיוק מסיבה זו קיימים קבצים אלה אשר תוכנות משתמשות בפונקציונאליות שהם מאפשרים, לדוגמא הדפסה למסך וכתיבה לקובץ.

מטרת הפרויקט

כיום, בעולם האבטחה לעיתים קרובות אנו מתמודדים עם קבצים שאיננו בטוחים אם הם זדוניים או לא. עד היום, כדי לנתח ולחקור קבצים מסוג זה, היינו מקימים מכונה וירטואלית (VM), בדרך כלל איטית ומורטת עצבים, בה התקנו את התוכנה הזדונית ובה חקרנו אותה. מטרת פרויקט זה היא לפתח שיטה חדשה, שיטה יעילה ומהירה יותר משימוש ב-VM. לכן בחרנו לעשות זאת עם Docker.

Docker Vs Virtual Machine

Docker הוא טכנולוגיה מבוססת Container, והן קיימות במרחב המשתמש (user space) של מערכת ההפעלה. ברמה הנמוכה, Container הוא רק קבוצה של תהליכים המבודדים משאר המערכת, הפועלים מתמונה ברורה המספקת את כל הקבצים הדרושים לתמיכה בתהליכים. הוא בנוי להפעלת יישומים. ב- Docker, הקונטיינרים הפועלים חולקים את ליבת מערכת ההפעלה של המכונה המארחת.

מכונה וירטואלית, לעומת זאת, אינה מבוססת על טכנולוגיית זו. היא מורכבת משטח משתמש (user space) בתוספת שטח ליבות של מערכת הפעלה (kernel space). תחת מחשבי VM, חומרת השרת עוברת וירטואליזציה. לכל VM יש מערכת הפעלה (OS) ואפליקציות. מכונה וירטואלית חולקת את משאבי החומרה מהמארח.

כל אחד מהם כולל יתרונות וחסרונות. בסביבת VM, כל מופע זקוק למערכת הפעלה מלאה. אך עם Container, עומסי עבודה מרובים יכולים לפעול עם מערכת הפעלה אחת. בנוסף, Container לא מחייב הפעלת Hypervisor (כגון VMWare או VirtualBox). ניתן להשתמש ב- Container לבידוד יישומים בודדים, ולהשתמש במכונות וירטואליות לבידוד מערכות שלמות. בכך Container מביא יתרונות נוספים כמו גודל מצומצם של תמונות, אפליקציות מהירות יותר, עדכוני אבטחה מופחתים ופשוטים ופחות קוד להעברה.

לסיכום שתי שיטות וירטואליזציה אלו פועלות ברמות שונות של הפשטה. כעט אנחנו מבינים למה הפעלת מכונה וירטואלית יקרה יותר מבחינת זמן מאשר הפעלת קונטיינר. כך גם לגבי חלק ההפצה והבנייה. לכן, עקב הדברים שהסברנו לעיל, אנו מבינים שהשיטה היעילה והטובה יותר עבור חקר של תוכנה בודדת אחת יהיה באמצעות שימוש ב-Docker.

מחקר

בפרק זה נסביר על תהליך המחקר שביצענו – החל מהבנת צורת העבודה של כלים ושיטות שקיימים היום בשוק, דרך ניסוי וטעייה של כלים שונים ועד יצירת קוד עובד שעומד בדרישות.

השאלה הראשונה שענינו עליה היא איך אנחנו יודעים באיזה קובץ (או קבצי) dependency תוכנה מסוימת משתמשת, ואיך אנחנו מגלים אותם.

ldd

כדי לענות על שאלה זו, היה עלינו לזהות את ה-dependencies שתוכנה מסוימת משתמשת בהם בזמן הריצה. לאחר חקירה וחיפוש במרשתת מצאנו כי קיימת פקודה בסביבת Linux אשר מספקת מידע באיזה קבצי SO קובץ מסוים תלוי. פקודה זו היא `ldd`. צורת שימוש:

```
user$ ldd /bin/cp
        linux-vdso.so.1 (0x00007ffcc4fcb000)
        libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f2e4b317000)
        libacl.so.1 => /lib/x86_64-linux-gnu/libacl.so.1 (0x00007f2e4b30c000)
        libattr.so.1 => /lib/x86_64-linux-gnu/libattr.so.1 (0x00007f2e4b304000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f2e4b112000)
        libpcre2-8.so.0 => /lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x00007f2e4b082000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f2e4b07c000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f2e4b37a000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f2e4b057000)
```

בפלט של הפקודה, בתחילת כל שורה שמופיע הסימן `=>` אנו יכולים לראות את שמו של ה-dependency שיש בו שימוש, ואחרי הסימן את מיקומו.

בתחילת הפרויקט השתמשנו בפקודה זו, אך אחרי חקירה מעמיקה גילינו כי כדי לגלות מידע זה, `ldd` מריץ את הקובץ(!) ומבחינתנו פעולה זו אסורה בתכלית האיסור, ממש ייהרג ובל יעבור. לכן לא השתמשנו בפקודה זו והמשכנו לחפש פקודות אחרות שיתנו לנו את הפלט הרצוי אך בצורה מאובטחת.

readelf -d

לאחר חקירה נוספת, מצאנו את הפקודה `readelf` אשר מציגה מידע על תוכן של קובץ ELF. צורת שימוש:

```
user$ readelf -d /bin/cp | grep 'NEEDED'
 0x0000000000000001 (NEEDED)             Shared library: [libselinux.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libacl.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libattr.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

פקודה זו היא בטוחה ואינה מריצה את הקובץ.

apt-file

כעט שיש שמו של ה-SO – צריך להוריד את ה- package המתאים שבו הוא כלול. גילינו כי שמו של ה-SO אינו בהכרח זהה לשם ה-package. לכן היה עלינו למצוא שיטה שבאמצעות שם של SO ניתן למצוא את ה-package שבו הקובץ נמצא.

כאן מגיע לעזרתנו פקודת `apt-file`. `apt-file` היא תוכנה שמתייגת תכנים של packages ומאפשרת לחפש קובץ מסוים בין כל החבילות הזמינות.

צורת שימוש:

```
oor@DESKTOP-C1IIBNN:/mnt/c/Users/shachar/project$ apt-file search libc.so.6
libc6: /lib/x86_64-linux-gnu/libc.so.6
libc6-amd64-cross: /usr/x86_64-linux-gnu/lib/libc.so.6
libc6-amd64-i386-cross: /usr/i686-linux-gnu/lib64/libc.so.6
libc6-amd64-x32-cross: /usr/x86_64-linux-gnux32/lib64/libc.so.6
libc6-arm64-cross: /usr/aarch64-linux-gnu/lib/libc.so.6
 ⋮
```

הפלט של פקודה זו ארוך, ואנו רואים שבכל אחד מן השורות כתוב שם הpackage עבור המון גרסאות שונות של מערכות הפעלה שונות. מצאנו כי ה-package הראשון עומד בדרישות כנדרש ולכן השתמשנו בו.

readelf -h

כעט יש בידנו אפשרות להתקין את כל ה-dependencies השונים שהקובץ הזדוני זקוק להם כדי לעבוד בצורה תקינה. השלב הבא והאחרון יהיה מציאת הארכיטקטורה המתאימה עבור הקוד הזדוני.

גם עבור מציאת הארכיטקטורה הנכונה בא לעזרתנו הפקודה `readelf`.

צורת שימוש:

```
user$ readelf -h /bin/cp
        | grep 'Class\|Machine'
  Class:                             ELF64
  Machine:                           Advanced Micro Devices X86-64
```

כעת יש לנו מידע על מספר הסיביות של פקודות מכונה והמעבד המיועד שעליו אמור לרוץ הקוד הזדוני. על פי שני נתונים אלו, אנו יכולים להסיק מה היא הארכיטקטורה המתאימה שתהיה ב-Docker. עשינו זאת ע"י קטע הקוד הבא:

```
if bits == '64:'
    if machine == 'X86-64:'
      arch = 'amd64'
    elif machine == 'ARM:'
      arch = 'arm64v8'
    else:
      print(f"Not supported\nbits:\t{bits}\nmachine:\t{machine}")
      exit()
  elif bits == '32:'
    if machine == 'ARM:'
      arch = 'arm32v7'
    elif machine == '80386:'
      arch = 'amd64'
    else:
      print(f"Not supported\nbits:\t{bits}\nmachine:\t{machine}")
      exit()
  else:
    print(f"Unexpected behavior\nbits:\t{bits}\nmachine:\t{machine}")
    exit()
```

בשלב זה הסתיימה החקירה. כל מה שנותר לעשות הוא לחבר את כל הממצאים לכדי קובץ `dockerfile`. Docker יכול לבנות מופע של container באופן אוטומטי על ידי קריאת ההוראות מקובץ `dockerfile`. `dockerfile` הוא מסמך טקסט המכיל את כל הפקודות שמשתמש יכול לקרוא לשורת הפקודה כדי להרכיב מופע של container.

פיתוח

לאחר שסיימנו לחקור את כל החלקים, השלב שנותר הוא חיבור החלקים לכדי קובץ סקריפט שיקבל את כל הנתונים ויצור עבור המשתמש את ה-Docker עם כל התלויות, כלי הדיבוג והקוד הזדוני בתוכו.

הוראות שימוש

```
user$ python3 run.py
usage: run.py [-h] path tools name
run.py: error: the following arguments are required: path, tools, name
user$ python3 run.py -h
usage: run.py [-h] path tools name
AutoDebugDocker.
positional arguments:
  path        path to the suspicious file
  tools       list of analysis tools name
  name        docker analysis name
optional arguments:
  -h, --help  show this help message and exit
```

כפי שניתן לראות, התוכנית מצפה לקבל שלושה קלטים:

- `path`: מיקום הקובץ החשוד.
- `tools`: כלי הדיבוג והאנליזה שהמשתמש רוצה שיהיו ב-docker על מנת לחקור איתם.
- `name`: השם ל-docker המיועד.

לדוגמא:

```
user$ python3 run.py malware "gcc radare2" docker4hacker
Hit:1 http://archive.ubuntu.com/ubuntu focal InRelease
Get:2 http://archive.ubuntu.com/ubuntu focal-updates InRelease [114 kB]
Fetched 1361 kB in 1s (945 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
82 packages can be upgraded. Run 'apt list --upgradable' to see them.
apt-file update complete
Searching Dependencies
[=============================================================================] 100%
 ⋮
 ⋮
Successfully built c9af1836818d
Successfully tagged malware_analysis:latest
building docker image complete
Process Complete!
 * Starting Docker: docker
2f28610604baa3048642def2616413b52ffb8f6e4592b45409b3ce1c9053cead
```

```
testuser@2f28610604ba:/$ dir
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  malware  media  mnt  opt  proc
root  run  sbin  srv  sys  tmp  usr  var
testuser@2f28610604ba:/$ gcc
gcc: fatal error: no input files
compilation terminated.
testuser@2f28610604ba:/$ r2
Usage: r2 [-ACdfLMnNqStuvwzX] [-P patch] [-p prj] [-a arch] [-b bits] [-i file]
          [-s addr] [-B baddr] [-m maddr] [-c cmd] [-e k=v] file|pid|-|--|=
testuser@2f28610604ba:/$ ./malware
This is a malicious software
MuHaHa!!!
```

כפי שניתן לראות, בצבע רקע כחול בהיר, ה-docker נוצר ורץ!

הדגשנו את החלקים שמראים שכל מה שהמשתמש ביקש להוסיף ל-docker אכן קיימים בו.

עבודה לעתיד

כפי שניתן לראות בקוד שהופיע תחת חלק הארכיטקטורה, בהסבר על הפקודה `readelf -h`, השארנו מקום עבור תמיכה בארכיטקטורות נוספות בעתיד.

נקודות לשיפור

דרוש שעל המכונה של המשתמש יהיה מותקן הספיריות הנדרשות של Docker. במידה והן לא מותקנות הקוד לא יעבוד. המשתמש נדרש שהיהי מותקן מראש Docker על המכונה שלו. שיפור לעתיד יהיה להוסיף לסקריפט שלנו את ההאפשרות שהוא יתקין בעצמו את הסיפריות הנדרשות במידה ואינן קיימות.

אבטחה

כדי למנוע חדירויות וחולשות נקטנו את הפעולות הבאות:

1. אנחנו משתמשים ב-Docker ולא מפעילים קובץ זר מהאינטרנט ישירות על המחשב שלנו. כפי שאמרנו Docker הינו מבודד משאר המערכת שלנו.
2. אנחנו יוצרים משתמש בתוך ה-Docker שכברירת מחדל אינו בעל הרשאות `root`.
3. מתקינים תמונות רשמיות של Ubuntu ולא מגורם צד שלישי לא מוכר.
4. בפקודה `docker run` הגבלנו את כמות הזיכרון של הקונטיינר כדי שהקובץ החשוד לא יעמיס על המחשב המארח. עשינו ע"י הוספת התגית של הגבלת כמות זיכרון: `--memory=256m`.