Path Traversal in Container Migration Solution Accelerator

מאת יובל אלבר

הקדמה

בשנים האחרונות הפך הענן לתשתית שעליה רצות כמעט כל המערכות שאנו מכירים, וחברות הענן הגדולות החלו לספק לא רק שירותי תשתית אלא גם Solution Accelerators - ערכות קוד פתוח מוכנות שנועדו להדגים כיצד לבנות פתרון שלם מעל שירותי הענן שלהן. מטרתם להאיץ פיתוח: במקום להתחיל מאפס, המפתח מקבל ארכיטקטורה עובדת, קוד לדוגמה ותסריטי פריסה, ועליו רק להתאים אותם לצרכיו.

בדיוק כאן נולד גם הסיכון. קוד שנכתב כ-"הדגמה" נוטה לקצר פינות באבטחה, אך מתגלגל לסביבות ייצור אמיתיות שכן הוא נושא את חותמת היצרן. במאמר זה נצלול לחולשת Path Traversal שמצאתי ב-Container-Migration-Solution-Accelerator - פרויקט קוד פתוח של Microsoft שמטרתו להגר הגדרות של שירותי קונטיינרים אל Azure Kubernetes Service. נראה כיצד שורה אחת תמימה למראה ברכיב ה-frontend פתחה דלת לקריאת קבצים שרירותיים מהשרת, נבין מדוע ההגנות "המתבקשות" אינן מספיקות, ונראה כיצד Microsoft תיקנה את החולשה. [(1)]

המאמר מיועד לקהל מגוון: מי שכבר מכיר את Path Traversal לעומק יוכל לדלג על פסקאות הרקע ולצלול ישר לניתוח הקוד, ומי שפוגש את המושג לראשונה יקבל את כל ההסברים הדרושים בדרך.

מהי חולשת Traversal Path?

חולשת Path Traversal (הידועה גם כ-Directory Traversal, ומסווגת תחת CWE-22 [(3)]) מתרחשת כאשר אפליקציה בונה נתיב לקובץ במערכת הקבצים מתוך קלט שהמשתמש שולט בו, מבלי לוודא שהנתיב הסופי אכן נשאר בתוך הספרייה המורשית. התוקף מנצל רצפים מיוחדים כמו /.. ("רמה אחת מעלה") כדי "לטפס" אל מחוץ לספרייה המיועדת ולהגיע לקבצים רגישים במערכת. [(4)]

דוגמה קלאסית: שרת מגיש קבצים מהספרייה /var/www/files/ לפי שם שהמשתמש מספק. אם המשתמש מבקש את report.pdf השרת יחזיר /var/www/files/report.pdf - בדיוק כמצופה. אך אם הוא מבקש /etc/passwd/../../../.., והשרת פשוט משרשר את הקלט לנתיב הבסיס, מתקבל הנתיב /etc/passwd - קובץ מערכת רגיש לחלוטין מחוץ לספריית ההגשה. לדוגמא:

הספרייה המורשית: /var/www/files/

קלט תקין: "report.pdf", מוביל להגשת הקובץ: /var/www/files/report.pdf

קלט זדוני: "/etc/passwd/../../..", מוסיל להגשת הקובץ: /etc/passwd

ב-Linux ו-macOS המפריד הוא "/", וב-Windows גם "\". מעבר לכך, קיימים וקטורים מתוחכמים יותר: קידוד URL (למשל 2e%2e%2f% במקום /..), קידוד כפול, ושימוש בקישורים סימבוליים (symlinks). לכן הגנה מבוססת "רשימה שחורה" של תווים נכשלת כמעט תמיד, והגישה הנכונה היא לאמת את הנתיב לאחר שפותר (resolved) במלואו.[5]

חשוב להבחין בין שני סוגי נתיב. **נתיב לקסיקלי** הוא המחרוזת כפי שנכתבה, על כל רצפי ה-/.. שבה, ואילו **נתיב פתור** (resolved) הוא הנתיב הקנוני שאליו מצביעה מערכת הקבצים בפועל, לאחר כיווץ הרצפים ופתירת הקישורים. ההבחנה הזו היא לב העניין: בדיקה המתבצעת על הנתיב הלקסיקלי ("האם המחרוזת מכילה /..?") ניתנת לעקיפה במגוון דרכים, בעוד בדיקה על הנתיב הפתור היא חסינה. בהמשך נראה כיצד התיקון הרשמי נשען בדיוק על העיקרון הזה.

הכרת המטרה: Accelerator-Solution-Migration-Container

הפרויקט Container-Migration-Solution-Accelerator הוא אפליקציה מרובת-שירותים מבית Microsoft, המספקת פתרון הגירה מבוסס סוכני AI להעברת תצורות של שירותי קונטיינרים מפלטפורמת ענן אחת אל Azure Kubernetes Service. הארכיטקטורה כוללת רכיב Frontend, שירות Backend API, רכיב Processor שמושך משימות מתור ב-Azure Storage Queue, ושירותי נתונים כגון Azure Blob Storage ו-Cosmos DB. כל הרכיבים רצים כ-Azure Container Apps.

הרכיב שבו נתמקד הוא ה-Frontend. בניגוד לרבים מפרויקטי ה-React שמוגשים על-ידי שרת סטטי כמו nginx, כאן בחרו המפתחים לכתוב שרת Python קטן מבוסס FastAPI שמגיש את קבצי ה-build של האפליקציה. השרת הזה, frontend_server.py, הוא לב העניין שלנו.

FastAPI - FastAPI היא מסגרת עבודה (framework) פופולרית לכתיבת שירותי Web ו-API ב-Python, הבנויה מעל Starlette ו-Pydantic. היא נפוצה מאוד בפרויקטי AI בזכות ביצועים גבוהים ותחביר פשוט. בפרויקט שלנו היא משמשת להגדרת הנתיבים (routes) שמגישים את קבצי ה-frontend ואת הגדרות התצורה ללקוח.

הקוד הפגיע: שורה אחת ששוברת הכול

הבעיה מתמקדת בנתיב ה-catch-all של השרת - ה-route שתופס כל בקשה שלא הותאמה לנתיב ספציפי אחר, ומיועד להגיש קובץ סטטי מתוך ספריית ה-build. להלן הקוד הפגיע מתוך src/frontend/frontend_server.py, לפני התיקון:

```python
BUILD_DIR = os.path.join(os.path.dirname(__file__), "dist")
INDEX_HTML = os.path.join(BUILD_DIR, "index.html")

@app.get("/{full_path:path}")
async def serve_app(full_path: str):
  file_path = os.path.join(BUILD_DIR, full_path)
  if os.path.exists(file_path):
    return FileResponse(file_path)
  return FileResponse(INDEX_HTML)
```

[קוד 1: ה-route הפגיע ב-frontend/frontend/src/server_frontend.py (שורות 65-69)]

שימו לב להגדרת ה-route: {full_path:path}. הסיומת path: היא מומר נתיב (path converter) של Starlette, והיא קריטית כאן: בניגוד לפרמטר רגיל ש-FastAPI עוצר בתו /, המומר path: **"בולע" גם את לוכסני הנתיב.** כך ערך כמו /../../etc/passwd מגיע אל הפונקציה כמחרוזת שלמה, על כל ה-/.. שבתוכה.

מכאן הדרך קצרה. השורה os.path.join(BUILD_DIR, full_path) נראית תמימה, אך טומנת בחובה מלכודת ידועה לשמצה ב-Python: כאשר אחד הרכיבים שמועברים ל-os.path.join הוא נתיב מוחלט (מתחיל ב-/), **הפונקציה מתעלמת** מכל הרכיבים שקדמו לו. וגם ללא נתיב מוחלט, רצף של /.. פשוט מטפס מעלה במערכת הקבצים. התוצאה זהה: הנתיב הסופי יוצא מחוץ ל-BUILD_DIR.[6]

נדגים את מלכודת ה-os.path.join בקונסולת Python:

```
>>> import os
>>> os.path.join("/app/frontend/dist", "index.html")
'/app/frontend/dist/index.html'
>>> os.path.join("/app/frontend/dist", "../../../etc/passwd")
'/app/frontend/dist/../../../etc/passwd'
>>> os.path.join("/app/frontend/dist", "/etc/passwd")
'/etc/passwd'
```

[קוד 2: התנהגות os.path.join מול קלט זדוני]

הניצול: מבקשה תמימה לקריאת קבצים שרירותית

מכיוון שמדובר בבקשת GET פשוטה ללא צורך באימות, הניצול טריוויאלי. כל מה שצריך הוא לשלוח בקשה HTTP עם רצף טיפוס מקודד-URL (..%2F במקום ../), שמטפס מספיק רמות כדי להגיע לשורש מערכת הקבצים ומשם אל הקובץ הרצוי. להלן מבחר מטענים שנבדקו מול הפרויקט המקורי, שנבנה והורץ באמצעות ה-Dockerfile של הפרויקט עצמו:

```
# 1) Read /etc/passwd via path traversal
curl "http://localhost:3000/..%2F..%2F..%2F..%2F..%2Fetc%2Fpasswd"
  → HTTP 200

# 2) Read the application source code:
curl "http://localhost:3000/..%2Ffrontend_server.py"

# 3) OS Fingerprinting:
curl "http://localhost:3000/..%2F..%2F..%2F..%2F..%2Fetc%2Fos-release"
  → NAME="Microsoft Azure Linux"
# 4) Leak the config via /config endping
  → MSAL client IDs, API URLs, auth authority
```

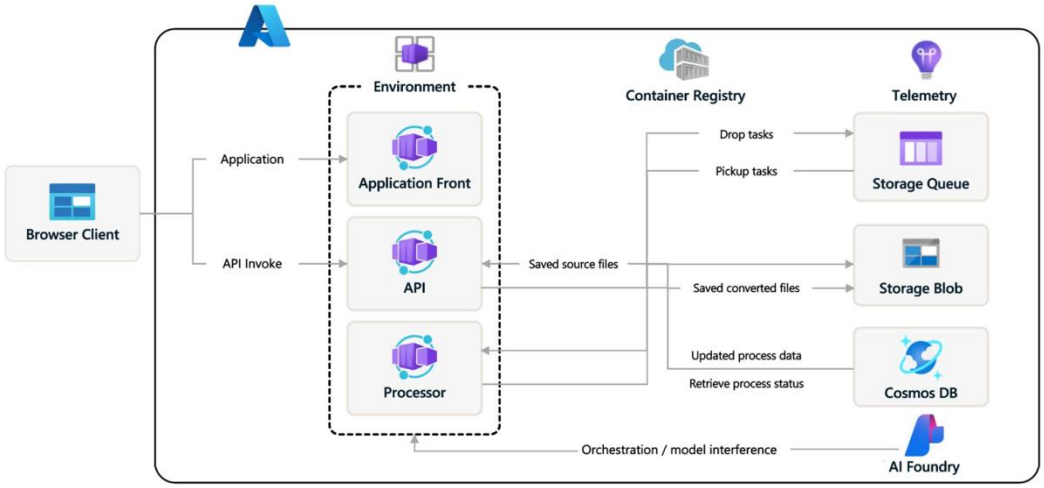
[קוד 3: מטעני הניצול (6/6 אומתו מול הפרויקט הלא-מותאם)]

שימו לב לקידוד URL של רצף הטיפוס. שליחת /.. "נקי" עלולה להינרמל על-ידי הלקוח או על-ידי שכבת ביניים עוד לפני שהיא מגיעה לאפליקציה, אך 2F%.. עובר את שכבות הביניים כפי שהוא ומפוענח רק בתוך Starlette - בדיוק במקום הלא נכון. נקודה טכנית נוספת: הקוד הפגיע משתמש ב-os.path.exists ולא ב-os.path.isfile, כך שהבדיקה עוברת גם על נתיבים שאינם קבצים רגילים - למשל הקובץ הווירטואלי /proc/self/environ.

מה אפשר לקרוא? כל קובץ שלמשתמש שמריץ את התהליך יש הרשאת קריאה אליו. בסביבת קונטיינר זה כולל בין היתר את /etc/passwd, קבצי קוד המקור של האפליקציה עצמה, ובאופן הקריטי ביותר - את משתני הסביבה של התהליך דרך /proc/self/environ. בקונטיינרים מודרניים, סודות רבים (מפתחות API, מחרוזות חיבור, אסימוני Azure) מוזרקים דווקא כמשתני סביבה, ולכן דליפתם הופכת חולשת קריאה לכאורה "בלבד" לחשיפת אישורים מלאה.

שווה להתעכב על העוצמה של /proc/self/environ. זהו קובץ וירטואלי שמערכת ההפעלה חושפת עבור כל תהליך, והוא מכיל את כל משתני הסביבה שלו בטקסט גלוי. בעוד שקבצי תצורה עשויים להיות מוצפנים או מוסתרים, משתני הסביבה נגישים ישירות. מפתחים נמנעים מהטמעת סודות בקוד (hardcoding) מתוך שיקול אבטחתי נכון, אך מזריקים אותם כמשתני סביבה - וכך, באירוניה, הופכים אותם לקלים יותר לחילוץ דרך חולשת Path Traversal. כך קריאה של קובץ בודד יכולה להתגלגל לתנועה לרוחב (lateral movement) ולפגיעה רחבה בסביבת הענן כולה.[7]

התרשים הבא ממחיש מדוע החשיפה כה משמעותית:



[תרשים 2: ארכיטקטורת הפרויקט - הדפדפן ניגש ל-Front Application ול-API שבסביבת Apps Container Azure (מקור: Microsoft, מאגר הפרויקט ב-GitHub)]

רכיב ה-Frontend אינו רכיב מבודד - הוא רץ באותה סביבת Azure Container Apps לצד ה-API וה-Processor, ובעל גישה לאותם משאבי Azure (Storage Blob, Storage Queue, Cosmos DB). דליפת מחרוזות החיבור והאישורים מתוך משתני הסביבה שלו פותחת דלת לכל אותם שירותי נתונים.

בנוסף לחולשת ה-Path Traversal עצמה, הבחנתי בבעיה משלימה. נקודת הקצה config/ (שורות 38-62) מחזירה לכל קורא לא מאומת ערכי תצורה רגישים: מזהי לקוח של MSAL, כתובת ה-authority של Azure AD, כתובות ה-API וה-scopes. חמור מכך, מדיניות ה-CORS הוגדרה כ-allow_origins=["*"], כך שכל אתר זדוני יכול לקרוא את הערכים הללו ישירות מתוך JavaScript בדפדפן של הקורבן. בשילוב עם קריאת קבצים שרירותית, מתקבל משטח חשיפת מידע רחב על תצורת השירות ואישוריו.

התיקון: אימות הנתיב לאחר פתירה

הכלל הזהב לתיקון Path Traversal אינו "לסנן את ../" - **גישה שכמעט תמיד ניתן לעקוף - אלא לפתור את הנתיב במלואו ואז לוודא שהוא נשאר בתוך הספרייה המורשית**. זה בדיוק מה שעשתה Microsoft ב-commit 3a0ec34:

```
@app.get("/{full_path:path}")
async def serve_app(full_path: str):
  # BUILD_DIR will parse the full path:
  file_path = os.path.realpath(os.path.join(BUILD_DIR, full_path))
  build_dir_real = os.path.realpath(BUILD_DIR)
  if not file_path.startswith(build_dir_real + os.sep) \
      and file_path != build_dir_real:
    return FileResponse(INDEX_HTML)
  if os.path.isfile(file_path):
    return FileResponse(file_path)
```

[קוד 4: ה-route לאחר התיקון]

המפתח הוא os.path.realpath: הפונקציה פותרת קישורים סימבוליים ומכווצת את כל רצפי ה-/.. לנתיב מוחלט וקנוני אחד. רק לאחר הפתירה נבדק שהנתיב הסופי מתחיל ב-build_dir_real + os.sep. הוספת os.sep אינה ניואנס שולי - בלעדיה, ספרייה כמו /app/dist-evil הייתה עוברת את הבדיקה רק משום שהיא מתחילה במחרוזת /app/dist. זוהי מלכודת prefix matching קלאסית שקל לפספס.

לצד תיקון ה-route, התיקון הוסיף גם מנגנון CORS מבוסס allow-list ובדיקת מקור (Origin) על נקודת הקצה config/, כך שתצורה רגישה לא תיחשף לבקשות חוצות-מקור. זהו תיקון יסודי שמטפל בשורש הבעיה ולא רק בתסמין.[2]

לקחים: למה זה קורה שוב ושוב?

החולשה הזו אינה ייחודית לפרויקט מסוים. דפוס os.path.join(base, user_input) חוזר באינספור פרויקטים, ובמיוחד בפרויקטי AI ו-MCP צעירים שבהם הקצב מהיר וההתמקדות היא בפונקציונליות. שלושה לקחים מרכזיים:

ראשית, "קוד הדגמה" אינו פטור מאבטחה. ברגע שפרויקט נושא את שם היצרן ומיועד לפריסה, יש להניח שמישהו יריץ אותו בייצור. שנית, אל תסמכו על נרמול בצד הלקוח - דפדפנים ו-curl **אמנם מנקים נתיבים, אך תוקף שולח את הבקשה הגולמית ישירות. שלישית, אמתו אחרי פתירה, לא לפניה**: הבדיקה היחידה שעומדת מול קידודים, קישורים סימבוליים ונתיבים מוחלטים היא השוואת הנתיב הקנוני מול הספרייה המורשית.

נקודה אחרונה שכדאי להפנים נוגעת ל-path converter של Starlette. מפתחים רבים משתמשים ב-{full_path:path} כברירת מחדל נוחה להגשת SPA (Single Page Application), מבלי להבחין שהמומר path: מעביר ביודעין גם תווי / שבדרך כלל נחסמים. כל route כזה שמסתיים בגישה למערכת הקבצים הוא חשוד מיידי, וחובה לזווג אותו עם אימות נתיב קפדני. כאשר ניתן, עדיף בכלל להגיש קבצים סטטיים דרך מנגנון ייעודי ומוקשח כמו StaticFiles של Starlette, ולא דרך route ידני שמרכיב נתיבים בעצמו.

סיכום

במאמר זה ראינו כיצד שורת קוד אחת ב-frontend_server.py - שרשור תמים לכאורה של קלט משתמש לנתיב קובץ - פתחה ב-Container-Migration-Solution-Accelerator של Microsoft חולשת Path Traversal (CWE-22) המאפשרת קריאת קבצים שרירותיים ללא אימות, עד כדי חשיפת אישורים דרך משתני סביבה. ראינו מדוע מלכודת os.path.join כה נפוצה, וכיצד התיקון הנכון - פתירת הנתיב ואימותו מול הספרייה המורשית - חוסם את כל וקטורי הניצול.

המסקנה רחבה יותר מפרויקט בודד: ככל שאנו נשענים על Solution Accelerators ועל קוד שנוצר במהירות סביב שירותי AI, כך גוברת החשיבות של בדיקת קלט שמרנית ושל אימות נתיבים נכון. החולשה דווחה ל-Microsoft, טופלה במהירות, ומדגימה היטב כיצד תהליך Coordinated Vulnerability Disclosure תקין מוביל לתיקון יסודי ובטוח יותר עבור כל המשתמשים.

על המחברת

יובל אלבר היא מהנדסת תוכנה וחוקרת אבטחת מידע עצמאית, עם רקע ביחידה 8200 וניסיון בפיתוח מערכות Python בייצור. בזמנה הפנוי היא בונה פורטפוליו של חולשות CVE במגוון יצרנים גדולים, עם התמקדות גוברת ברכיבים נמוכי-רמה ובהצטלבות שבין AI לאבטחה. ניתן ליצור קשר דרך LinkedIn.

מקורות מידע

- Container-Migration-Solution-Accelerator
- frontend_server.py (המתוקנת הגרסה)
- CWE-22: Improper Limitation of a Pathname to a Restricted Directory (MITRE)
- Path Traversal (OWASP)
- What is path traversal (PortSwigger Web Security Academy)
- os.path.join
- The False Security of AI Containers (Equixly)