Digital Whisper
Technologic Papers

Trust No One
עוברים לפדרציה? אל תשאירו את המפתחות בדלת

מאת הילה קוזוקרו

הקדמה

במשך תקופה ארוכה, סיסמאות ומפתחות סטטיים היו עבורי סוג של "ברירת מחדל". לא באמת אהבתי אותם, אבל היה נדמה שאין להם תחליף אמיתי. הכל השתנה כשגיליתי שאפשר להתנהל כמעט בלעדיהם בזכות מנגנון עם שם קצת טכני מדי - **Federation**.

המעבר לפדרציה הרגיש כמו קפיצת מדרגה, זה פתרון אלגנטי שפותר בעיות אבטחה מהשורש, אבל ככל שהעמקתי בו, הבנתי שהוא דורש מאיתנו משהו אחר: דייקנות. במאמר זה אני רוצה לשתף אתכם למה פדרציה היא כלי כל כך עוצמתי, ודווקא בגלל זה למה היא דורשת מאיתנו תשומת לב מקסימלית לפרטים הקטנים.

בחלק הראשון של המאמר נתמקד בהגדרות ולאחר מכן אדגים איך טעות הגדרה קטנה, כזו שקל מאוד לפספס, יכולה להפוך את המנגנון הזה לדלת פתוחה לשליטה מלאה ברשת הארגונית - חולשה שפגשתי לא פעם "בשטח", גם בסביבות Production של חברות ענק!

עולם ללא מפתחות

דמיינו עולם שבו אין מפתחות. לא לבית, לא לרכב, ולא לאופניים. אתם מתקרבים לדלת והיא נפתחת מעצמה. לא כי הקשתם קוד, לא כי החזקתם צ'יפ, אלא כי המערכת מזהה אתכם מעצם היותכם אתם. הדרך שבה אתם נעים, הקצב שלכם, זו החתימה הדיגיטלית והייחודית שאתם משדרים - כל אלו מייצרים זהות ייחודית שקשה מאוד לזייף. אין מפתח לאבד, אין קוד לשכוח.

זה נשמע מושלם, נכון? ובדיוק מהסיבה הזו, זה גם נשמע too good to be true. הרעיון של גישה המבוססת על אמון בזהות ולא על חפץ מוחשי (כמו מפתח או סיסמה), הוא הבסיס למה שנקרא בעולם

האבטחה **Federation**. במקום להחזיק מפתח, מישהו אחר מעיד עליכם. המערכת לא צריכה להכיר אתכם מראש, היא פשוט צריכה לסמוך על הגורם שהנפיק את האישור שאתם מחזיקים.

כדי להבין למה המודל הזה מחליף את עולם הסיסמאות, ולמה הוא עלול להישבר בצורה מסוכנת, אנחנו צריכים לחזור רגע לבסיס המעיק של חיינו: הסיסמה הסטטית.

למה המודל הישן נשבר?

סיסמאות הן מנגנון האימות המוכר ביותר. במשך עשורים זה היה פשוט: מי שמחזיק במפתח, נכנס. מי שלא, נשאר בחוץ. המודל הזה עבד כשהיו לנו שלוש מערכות וארבעה משתמשים. היום? העולם הזה השתנה, מספר השירותים גדל, מספר המשתמשים הלא-אנושיים (Non Human Identity - NHI) זינק, ומספר המקומות שבהם סיסמאות ומפתחות API נשמרים, מועתקים או נשכחים הפך להרבה יותר שכיח ממה שהיינו רוצים להאמין.

הבעיה היא לא רק שסיסמאות נגנבות. הבעיה היא שהן **סטטיות**. מפתח API שנשמר ב-GitHub Secret נשאר שם לנצח. אם הוא דלף, התוקף מחזיק בסיסמא שלכם עד שתבצעו רוטציה ידנית. אפילו עם MFA, הבעיה הבסיסית נשארת: יש "סיסמא" מוחשית, ואם מישהו אחר מחזיק בה, הוא בפנים. פדרציה באה לשנות את הפרדיגמה הזו מהיסוד.

מילון מושגים

לפני שנצלול לקרביים של המנגנון, בואו ניישר קו על המושגים. אם הם מוכרים לכם מוזמנים להמשיך לפרק הבא:

1. **זהות (Identity):** בענן, זהות היא כל ישות שמבקשת לבצע פעולה. זה יכול להיות בן אדם (Human), אבל בעולם המודרני זה לרוב **Workload** - תהליך אוטומטי, קוד שרץ ב-Kubernetes, או Pipeline של CI/CD. תחשבו על identity כעל ה"תעודת זהות" של המבקש.
2. **Service Account:** הוא "חשבון משתמש" שאינו מיועד לבני אדם, אלא לתוכנות (אפליקציות, שרתים או סקריפטים). בעוד שמשתמש אנושי מזדהה עם אימייל וסיסמה, ה-Service Account הוא ישות בתוך הענן שמחזיק הרשאות (Roles) לביצוע פעולות על משאבים..
3. **Workload Identity:** זהו המנגנון שמאפשר לזהויות מחוץ לענן (כמו GitHub Actions) להזדהות מול הענן בלי סיסמה. איך? במקום להחזיק מפתח סטטי בכיס, ה-Workload מציג הוכחת זהות קריפטוגרפית זמנית (OIDC Token המבוססת על ההקשר שבו הוא רץ. למשל: "אני ה-Pipeline שרץ עכשיו על ה-Main Branch במאגר X"). הענן בודק את ההוכחה הזו מול ספק הזהות (GitHub) ומחליט אם להכניס אתכם פנימה.

4. **ה-Trust Policy:** זה המושג הכי חשוב **במאמר**. בעוד ש-**Permission Policy** מגדירה **מה** מותר לעשות (הרשאות גישה למשאבים), ה-**Trust Policy** מגדירה **מי** רשאי בכלל להשתמש בזהות של ה- Service Account (ה-Principal).

ואם נפשט זאת:

- **Permission Policy:** רשימת החדרים שמותר לך להיכנס אליהם בבניין.
- **Trust Policy:** רשימת האנשים שהשומר בכניסה מוכן בכלל לתת להם מפתח.

5. **Authentication vs. Authorization. (או: מי אתה ומה מותר לך?)** כדי לא ללכת לאיבוד, נפריד בין שני שלבים שביניהם עובר ה-**Trust Boundary** (גבול האמון). גבול האמון הוא הנקודה שבה המערכת מפסיקה לחשוד ומתחילה לסמוך:

- **Authentication (אימות):** בשלב הזה הענן שואל "מי אתה?". הוא בודק את ה-OIDC Token ומוודא שהוא אמיתי וחתום קריפטוגרפית על ידי מישהו שהוא מכיר (כמו GitHub). זהו תהליך של **זיהוי** - הענן מוודא שהתעודה שהוצגה אינה מזויפת, אבל הוא עדיין לא יודע אם הישות היא "משלנו" או סתם עובר אורח עם תעודה תקפה.
- **Authorization (הרשאה):** כאן נחצה ה-**Trust Boundary**. אחרי שספקית הענן השתכנעה שהישות אכן מגיעה מ-GitHub (האימות עבר), היא שואלת: "אוקיי, אז מה מותר לך לעשות אצלי?". בשלב הזה נבדקים המאפיינים הספציפיים (כמו שם ה-repo) מול המדיניות שהוגדרה. זהו השלב שבו נקבע האם מותר לישות "להיכנס לנעליים" של ה-Service Account ולפעול בתוך הסביבה.

לפדרציה שלי שלוש פינות

נתמקד בשלושה מימושים לפדרציה:

1. **IdP - Identity Provider . הארגוני: המקור לזהויות אנושיות:** במודל הזה, הארגון מחזיק ספק זהויות מרכזי כמו Entra ID או Okta, המהווה את 'מקור האמת' (Source of Truth). זהו המאגר המרכזי שבו נשמרות ומנוהלות כל הזהויות הארגוניות במקום אחד. במקום שכל אפליקציה תנהל מסד נתונים נפרד של משתמשים וסיסמאות, האימות מתבצע אך ורק ב-IdP. איך זה עובד? כשמשתמש ניגש למערכות כמו Jira או ה-AWS Console, הן מפנות אותו ל-IdP. המערכות הללו לא יודעות מה הסיסמה של המשתמש, הן פשוט סומכות על ה"אישור" (Assertion) שהן מקבלות מה-IdP.

הערך: מניעת כפילות זהויות, אכיפת MFA וניהול כניסת / יציאת עובדים (Onboarding/Offboarding) מהיר ובטוח.

2. **פדרציית ענן ו-CI/CD: מלחמה ב-Secrets סטטיים:** זהו המודל שבו נתמקד, והוא מיועד לזהויות לא-אנושיות (Workloads). כאן, מערכת ה-CI/CD (לדוגמא: GitHub Actions או GitLab) מתפקדת כספק הזהויות.

איך זה עובד? עבור כל ריצה של Workflow, המערכת מנפיקה OIDC Token קצר-מועד. ספקית הענן בודקת את הטוקן מול ה-Trust Policy שהוגדר מראש.

הערך: במקום לשמור מפתחות API קבועים בתוך ה-GitHub Secrets (שעלולים לדלוף או להישכח), הגישה מבוססת על ההקשר (Context) של הריצה. זהו פתרון מודרני ובטוח, שהפך להיות הפרקטיקה המקובלת בקרב צוותי **DevOps ו-Cloud Engineering**, שמעדיפים להימנע מניהול ידני של מפתחות. אך השיטה הזו רגישה מאוד לטעויות קונפיגורציה וזו נקודת התורפה שננתח בהמשך.

3. **Cross-Cloud Federation: ספקית ענן שסומכת על שירות אחר:** מודל זה נפוץ בסביבות Multi-Cloud או בארגוני Enterprise גדולים. כאן, ספקית ענן מסוימת (או פרויקט אחד) בוחרת לסמוך על זהויות שנוצרו בספקית ענן אחרת. לדוגמא, יש שרת ב-**AWS** שצריך לגשת למסד נתונים ב-**Google Cloud**. במקום לייצר למכונה משתמש וסיסמה בתוך Google, מגדירים ש-Google פשוט "תסמוך" על הזהות שהמכונה כבר קיבלה מ-AWS.

הערך: הפרדת סביבות (Segregation) ושיתוף משאבים ללא צורך בשכפול משתמשים או העברת סיסמאות בין סביבות ענן שונות.

כדי להבין איך פדרציה נראית בפועל, ואיפה היא עלולה להישבר, נתמקד כעת במימוש אחד ספציפי - **ספקית ענן שסומכת על מערכת CI/CD וספציפית נפרט על Workload Identity**. זהו מימוש שממחיש היטב את עקרונות הפדרציה, ואת האופן שבו Trust Policy מגדירה לא רק מי מאומת, אלא מי בכלל רשאי להפוך לזהות לגיטימית במערכת.

איך פורטים את המודל הזה לפרקטיקה? כדי לראות את זה קורה בעיניים, אנחנו צריכים לצלול אל תוך המערכת שמנהלת את הדיאלוג הזה בין העולמות. ב-Google Cloud המשימה הזו מוטלת על שירות ה-**Workload Identity Federation**.

Workload Identity Pool: חשבו על זה כעל "קבוצת אמון". זהו ה-Namespace שבו מנהלים את הזהויות החיצוניות. מומלץ ליצור Pools נפרדים לסביבות שונות (Prod/Dev) כדי למנוע זליגת הרשאות.

1. **Workload Identity Provider:** זהו הרכיב בתוך ה-Pool שמגדיר את הקשר הטכני עם ה-IdP (כמו GitHub). כאן אתם מגדירים את ה-Audience (מי אמור לצרוך את הטוקן) ואת ה-Mapping (איזה שדה בטוקן הופך לאיזה Attribute בענן).

איך זה עובד בפועל?

נפרק את ה-OIDC לארבעה שלבים:

שלב 1: הנפקת ה-JWT (המאני-טיים מתחיל)

ב-GitHub Actions, כשה-Workflow רץ, הוא מנפיק טוקן זמני. וכך זה נראה:

```
{
  "iss": "https://token.actions.githubusercontent.com",
  "sub": "repo:my-org/my-app:ref:refs/heads/main",
  "repository": "my-org/my-app",
  "repository_owner": "my-org",
  "workflow": "Deploy to Prod"
}
```

אלו הן בעצם **הצהרות** (Claims) ש-GitHub מעיד על הישות: מי היא, מאיזה **מאגר (Repository)** הגיעה ואיזה Workflow היא מריצה. כדי שאף אחד לא יוכל לזייף את המכתב הזה, GitHub חותם עליו קריפטוגרפית.

שלב 2: ה-Handshake (אימות קריפטוגרפי ללא החלפת סיסמאות)

בשלב הזה, ה-Workload שולח את ה-JWT שקיבל מ-GitHub אל ה-Workload Identity Provider ב-GCP. כאן קורה פלא טכנולוגי: GCP צריך לוודא שהטוקן הזה באמת הגיע מ-GitHub ולא זויף על ידי תוקף, אבל הוא עושה זאת **בלי שהגדרנו מראש שום מפתח סודי**. **איך זה עובד?** GCP משתמש בכתובת ה-Issuer שמוגדרת ב-Provider (למשל: https://token.actions.githubusercontent.com). הוא פונה לכתובת הזו ומושך ממנה קובץ ציבורי שנקרא **JWKS** (JSON Web Key Set) - המכיל את המפתחות הציבוריים של GitHub. באמצעותם, GCP מוודא שהחתימה על הטוקן תקינה. אם מישהו שינה אפילו אות אחת ב-Payload של הטוקן, החתימה תישבר והתהליך ייעצר כאן.

שלב 3: ה-Mapping וה-Condition - מי אתה ומה אתה מביא איתך?

לאחר הווידוא בשלב הקודם שהטוקן אמיתי וחתום על ידי GitHub (ה-Authentication עבר), עוברים לשלב ה-Authorization -השלב שבו מחליטים "מה הישות הזו יכולה לעשות?". בשלב זה קורים שני דברים קריטיים:

1. **Attribute Mapping (תרגום שפות):** הטוקן שמגיע מ-GitHub מכיל "הצהרות" (או בשפה המקצועית: **Claims**) - אלו הם פרטי מידע ש-GitHub חתם עליהם, כמו שם המאגר, שם הארגון או המשתמש שהריץ את הקוד.
2. **ה-Attribute Condition: זה הסלקטור שנמצא בכניסה למסיבה,** הוא בעצם הפילטר שבודק את התוכן של ה-Mapping. אם לא נגדיר תנאי (Condition) מפורש, ספק הענן יסתפק בכך שהטוקן הגיע מ-GitHub, בלי לבדוק **מי** בתוך GitHub שלח אותו.

הנה כמה Attributes מרכזיים שחובה להכיר:

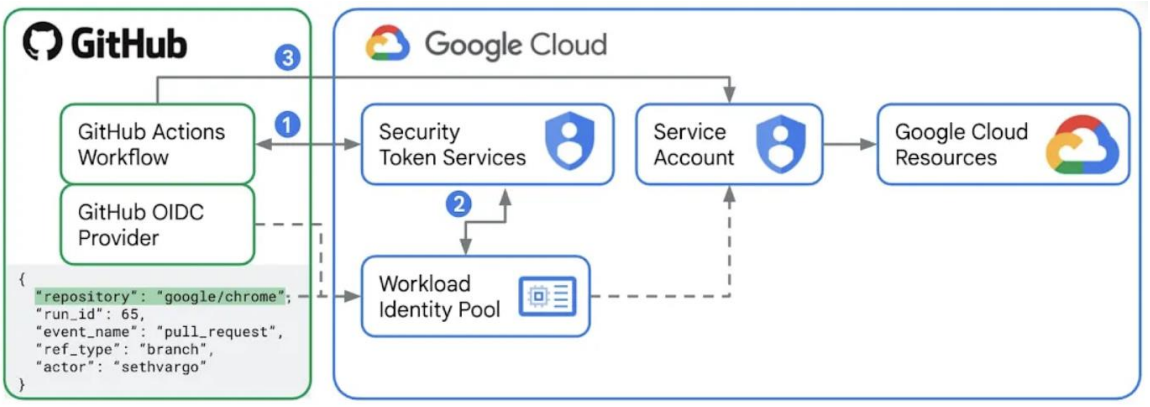
- **sub (Subject):** המזהה המדויק של ה-Workflow. הוא כולל בד"כ את שם הארגון, ה-Repository וה-Branch (לדוגמה: repo:my-org/my-repo:ref:refs/heads/main). **זהו הקריטריון החשוב ביותר.**
- **repository / repository_owner:** מאפשר להגביל את הגישה רק לארגון שלכם או לפרויקט ספציפי, כדי למנוע מצב שבו חשבון GitHub אחר לגמרי מקבל גישה לענן שלכם.
- **actor:** המשתמש שהפעיל את ה-Workflow. שימושי אם רוצים לוודא שרק פעולות של משתמשים ספציפיים מורשות לבצע שינויים קריטיים.
- **בשורה התחתונה:** בלי Condition שבודק לפחות את ה-sub או ה-repository, השער שלכם פתוח לכל מי שיש לו חשבון GitHub בעולם.

ויש כאן מוקש שצריך לשים לב אליו: מכיוון שכל אחד בעולם יכול להנפיק טוקן חתום ותקין מ-GitHub עבור repo הפרטי שלו, ללא ה-Condition הזה, כל Repository בעולם יוכל להזדהות מול ה-Provider שלכם. ה-Condition הוא זה שמוודא שהערך בתוך ה-Mapping הוא אכן שם המאגר שלכם ולא של מישהו אחר.

שלב 4: ה-STS והחלפת הזהות (Impersonation) כאן קורה הקסם הטכני. ה-Workflow פונה לשירות שנקרא STS, או בהרחבה Security Token Service. תפקידו של ה-STS הוא לשמש כ"חלפן זהויות": הוא מקבל את הטוקן החיצוני של GitHub וממיר אותו לטוקן גישה זמני של Google. ברגע שההחלפה מתבצעת, ה-Workflow מבצע **Impersonation** - הוא "נכנס לנעליים" של ה-Service Account המקומי ומתחיל לפעול בענן. מעתה, כל פעולה שיבצע ה-Workflow תתבצע תחת ההרשאות המוגדרות ב-IAM עבור אותו Service Account, כאילו היה ישות פנימית לחלוטין בארגון.

אז למה זה בטוח יותר? בניגוד למפתחות API סטטיים, ה-Access Token שמנפיק ה-STS הוא קצר-מועד (לרוב תקף לשעה אחת בלבד). גם אם תוקף הצליח לשים את ידו על הטוקן הזה, חלון הזמנים שלו לפעולה מצומצם ביותר, והוא אינו יכול לחדש את הטוקן ללא גישה ישירה ל-Workflow המקורי.

ולסיכום הוספתי תמונה שממחישה את התהליך הזה:



רגע לפני שנראה איך הכל קורס, נדגים איך זה נראה במציאות. כשמגדירים את ה-Provider ב-Console, קובעים את חוקי המשחק. בתמונה הבאה תוכלו לראות את ה-Mapping. שימו לב לשדה ה-Condition בתחתית, אם הוא נשאר ריק, הדלת נשארת פתוחה.

הינה דוגמה לקונפיגורציה פגיעה - ללא Condition:

```
{
  "name": "projects/my-project/locations/global/workloadIdentityPools/github-
pool/providers/github-provider",
  "oidc": {
    "issuerUri": "https://token.actions.githubusercontent.com"
  },
  "attributeMapping": {
    "google.subject": "assertion.sub",
    "attribute.repository": "assertion.repository"
  }
}
```

[שימו לב: שדה ה-attributeCondition חסר!]

כעת, כשהבנו איך המנגנון בנוי ואיך הוא אמור להיראות כשהוא תקין (וגם כשהוא לא), נלבש את הכובע של התוקף.

Attack Path: הפריצה ה"חוקית" שמתחילה ב-Repository שלכם

עד עכשיו הכל נשמע מבטיח: אין מפתחות סטטיים, יש חתימות קריפטוגרפיות, והכל עובר דרך שירות ה-STS המאובטח של גוגל. אבל בעולם ה-Security, ככל שהמנגנון יותר "חכם", כך הכשל שלו יותר שקט ומסוכן.

כדי להבין כמה המנגנון הזה רגיש, נתאר מסלול תקיפה (Attack Path) שמראה איך תוקף יכול להיכנס לסביבה שלכם. הוא לא צריך לנחש סיסמה מורכבת, הוא לא צריך לגנוב מפתח API שדלף, והוא בטח לא צריך לפרוץ את השרתים של גוגל או GitHub. כל מה שהוא צריך זה את **פרטי התשתית שלכם** (שפעמים רבות גלויים בקוד פתוח או פשוטים לניחוש) **ואת הטעות שלכם בקינפוג.**

1. הכנה: התוקף לא צריך אתכם

דמיינו תוקף שאין לו שום גישה לארגון. אין לו משתמש ב-GCP שלכם, אין לו גישה ל-GitHub שלכם, ואין לו מפתח API שדלף. כל מה שיש לו הוא חשבון GitHub אישי וחינמי ויכולת להריץ GitHub Actions - יכולת שזמינה ל**כל אדם בעולם**. התוקף פותח מאגר (Repository) פרטי משלו, למשל:

hacker-dev/malicious-repo

2. הנפקת הטוקן ה"לגיטימי"

התוקף מגדיר Workflow ב-GitHub ומריץ אותו. בזמן ההרצה, GitHub (כספק זהות לגיטימי) מנפיק באופן אוטומטי OIDC Token (JWT) חתום ותקין המייצג את הזהות של ה-Workflow של התוקף. הטוקן הזה חתום על ידי המפתחות של GitHub, בדיוק כמו הטוקן של המפתחים שלכם.

וזה יכול להראות כך:

```
name: The Silent Entry
on: workflow_dispatch
permissions:
  id-token: write
jobs:
  exploit:
    runs-on: ubuntu-latest
    steps:
      - name: Authenticate to GCP
        uses: google-github-actions/auth@v2
        with:
          #כאן התוקף מזין את הכתובת של ה-Provider וה-Service Account שלכם
          workload_identity_provider: "projects/your-project-
id/locations/global/workloadIdentityPools/github-pool/providers/github-
provider"
          service_account: "terraform-admin@your-project-
id.iam.gserviceaccount.com"

      - name: Proof of Concept
        uses: google-github-actions/setup-gcloud@v2

      - name: Who am I?
        run: |
          #הפקודה שמוכיחה שהתוקף מקבל את ההרשאות של ה-Service Account
```

3. המלכודת של ה-Default

התוקף שולח את הטוקן שלו אל ה-Workload Identity Provider בארגון המותקף. כאן קורה הכשל הקריטי: בגלל שה-Provider הוגדר ללא **Attribute Condition**, ה-GCP מבצע את הבדיקות הבאות:

- האם הטוקן הגיע מ-GitHub? **כן.**
- האם החתימה הקריפטוגרפית תקינה? **כן.**
- האם יש תנאי שמגביל איזה Repository מורשה להיכנס? **לא.**

ספקית הענן בודקת דבר אחד בלבד: האם מדובר בטוקן תקף שחתום על ידי ה-Issuer המוכר (GitHub). מכיוון שהתשובה היא **כן**, ה-STS מבצע את ההחלפה (Token Exchange) ומנפיק לתוקף Access Token של ה-Service Account של הארגון.

4. Impersonation התוקף הופך לחלק מהארגון

ברגע הזה, המעבר הושלם. התוקף מחזיק בזהות לגיטימית לחלוטין בענן. הוא לא "האקר מ-GitHub", הוא עכשיו ה-terraform-service-account או ה-prod-deployer. אם ל-Service Account הזה יש הרשאות ל-PROD אז השמיים הם הגבול, התוקף יכול להניח את ידו על כל משאב במערכת.

הוא יכול למחוק בסיסי נתונים או לשאוב את המידע שלהם, הוא יכול לשנות הרשאות ולמחוק משתמשים, או אפילו במקרים מסוימים לשנות את הערך של ה claim ולנעול את כל שאר המשתמשים.

ומה אפילו עוד יותר מתוחכם? הלוגים יראו תקינים לחלוטין כי הרי התוקף לא השתמש בזהות לא מאושרת. שום alert או monitoring system לא יצפצף או יתריע על הפשיטה, הכל יראה רגיל, וזה מה שהופך את הפריצה הזאת לעוד יותר מסוכנת כי היא יכולה להישאר לאורך זמן מבלי שיגלו אותה.

זה לא באג - זה כשל לוגי שקט

הכשל המתואר כאן אינו נובע מ**חולשה טכנית ב-GitHub או ב-GCP,** שתי המערכות פועלות בדיוק כפי שתוכננו. הכשל נובע מהגדרת Trust Policy רחבה מדי, שהופכת את כל GitHub למקור אמון אבסולוטי. זהו תרחיש מסוכן במיוחד כי הוא שובר את ההנחה הבסיסית שהיעדר סיסמאות משפר את האבטחה. כאן, דווקא המנגנון המודרני פותח דלת מסוכנת ומתוחכמת.

ולסיכום

פדרציה היא ללא ספק העתיד של ניהול הזהויות בענן. היא מאפשרת לנו להיפרד מהתלות המסוכנת במפתחות סטטיים ומפשטת תהליכי אוטומציה בקנה מידה עצום. אבל עם הכוח הזה מגיעה אחריות מסוג חדש: **ניהול האמון.** כאשר עוברים ממודל של "מפתחות" למודל של "פדרציה", מרכז הכובד של האבטחה עובר לידיים של מי שמגדיר את המערכת. האבטחה היא כבר לא שאלה של "איפה שמרנו את הסיסמה", אלא של "איך הגדרנו את חוקי המשחק".

חשוב להבין: הכשלים האלו לא קורים בגלל חוסר מקצועיות, אלא בגלל **טעות אנוש פשוטה** שיכולה לקרות לכל אחד מאיתנו. קל מאוד להחליק ולוותר על Condition קטן בשם ה"מהירות" או ה"נוחות", וזה קורה גם לארגוני הענק המיומנים ביותר. ארגונים שמתייחסים ליחסי האמון האלו כאל "עניין טכני" בלבד, עלולים לגלות שהפרצה שלהם לא הגיעה מתקיפה מתוחכמת, אלא מהגדרה רחבה מדי שפשוט אפשרה לתוקף להיכנס בדלת הראשית.

בסופו של יום, המנגנון המודרני הזה הוא כלי רב עוצמה, אבל הוא חזק רק כמו ה-Condition הכי חלש שהגדרתם בו.

על המחברת

נעים להכיר! אני רגע לפני קו הסיום של התואר במדעי המחשב וקוגניציה בפתוחה, וביום-יום תמצאו אותי ב-Oasis Security. אני מאוד אוהבת סייבר, ומכורה ללמידה האינסופית של התחום הזה ותמיד מחפשת איפה האתגר הבא מסתתר. מקווה שהמאמר עשה לכם סדר בראש וסגר לכם איזה "חור" או שניים בארגון. נתראה בפריצה הבאה! (מהצד המגן, כן? ☺)

מקורות מידע

GCP Best Practices: המדריך הרשמי של Google לשימוש נכון ב-Workload Identity Federation. חובה לכל מי שמגדיר את זה ב-Production:
https://docs.cloud.google.com/iam/docs/best-practices-for-using-workload-identity-federation?utm_source=chatgpt.com

Deep Dive לסיכונים: מאמר מעולה של Tenable שמנתח לעומק איך תוקפים מנצלים מיסקונפיגורציות במנגנוני Multicloud:
https://www.tenable.com/blog/how-attackers-can-exploit-gcps-multicloud-workload-solution?utm_source=chatgpt.com

הצד של GitHub: איך לאבטח את ה-Cloud Deployment שלכם באמצעות OIDC מהזווית של ה-Repo:
https://docs.github.com/en/actions/concepts/security/openid-connect