ממ"ן 16

שאלה 2:

בקוד המצורף בשם Q2.py בחרתי בסיסמה נפוצה password123 על מנת להמחיש את המתקפה.

תיאור המתקפה:

1. תחילה ניצור מילון/אוסף של סיסמאות נפוצות, בחרתי ב42 סיסמאות נפוצות ע"פ דירוגים שמצאתי באינטרנט, נעשה בהמשך גם שילוב של הסיסמאות עם עצמם על מנת ליצור עוד שילוב של סיסמאות נפוצות ובכך נגדיל את המילון.
2. "נשלוף" את הערכים מהתקשורת בין שרת האימות ללקוח שיהיו רלוונטים למתקפה ובעזרתם יאמתו לנו שאכן עלינו על הסיסמא של הלקוח:

* Nonce - שהלקוח שולח לשרת האימות (לא מוצפן).
* IV – שחוזר משרת האימות ללקוח שבעזרתו שרת האימות מצפין את הNonce.
* Encrypted Nonce – הNonce המוצפן שחוזר משרת האימות ללקוח שהוצפן באותו IV שהזכרתי ועם סיסמת hash של הלקוח.

1. נשתמש בפוקנציית Hash sha256, אותה פונקציה שבעזרתה שרת האימות יוצר מפתח סימטרי בינו לבין הלקוח באמצעות הסיסמה שלו.
2. כעת נרוץ בלולאה על כל ה42 סיסמאות הנפוצות ששמרנו נחשב לכל סיסמא את הhash sha256 שלה, עם התוצאה המתקבל בפעולה זו שמסמלת עבורנו את המפתח הסימטרי בין הלקוח לשרת האימות, נשתמש כמפתח ונצפין את Nonce המקורי נבדוק שהוא שווה ל Encrypted Nonce, אם הם שווים זה אומר לנו שאכן עלינו על המפתח הסימטרי האמיתי בין הלקוח לשרת האימות, נעשה עוד בדיקה שאכן אם אנחנו מפענחים את Encrypted Nonce שהצפנו נקבל את ה- Nonce המקורי, אם כן אכן עלינו על מפתח האמיתי ועל הסיסמא של הלקוח ונדפיס אותה למסך.
3. נעשה את אותו התהליך רק שהפעם נשלב את הסיסמאות הנפוצות זו בזו, כלומר אם לדוגמה ישנה סיסמה – "123" וישנה סיסמה "password", אז השילוב יעבוד כך:
4. "password123"
5. "123password"

נעשה כך לכל סיסמה וסיסמה.

1. לאחר שעשינו זאת הגדלנו את המילון של הסיסמאות הנפוצות בעוד 84 סיסמאות (42 X 2) ונחזור על שלב 4, בסך הכל השתמשנו ב126 סיסמאות.

אם מצאנו את הסיסמא נדפיס אותה למסך ובכך בעצם המתקפה הצליחה אם לא יודפס למסך שלא מצאנו את הסיסמא ובעצם המתקפה לא הצליחה.  
  
פתרון למניעת המתקפה:

נשתמש בפונקציה הבאה:

1.def pbkdf\_password(hash\_password,salt,key\_length=32, iterations=600000):  
 hashed\_password = PBKDF2(hash\_password, salt, dkLen=key\_length, count=iterations)  
 return hashed\_password

עם הפונקציה:

2.def hash\_password(password):   
 hashed\_password = hashlib.sha512(password.encode('utf-8')). digest()  
 return hashed\_password

בממ"ן נדרשנו את לחשב הhash sha256 של כל סיסמא עבור הלקוח כדי ליצור ממנה מפתח, כעת נחשב את הhash sha512 של כל סיסמא בפונקציה 2, את הערך המתקבל נשלח כפרמטר לפונקציה 1, ניצור salt אקראי וייחודי ונשלח גם כפרמטר לפונקציה 1.

באמצעות הפונקציה הראשונה, אנו יכולים להאט משמעותית את החישוב של כל סיסמא שהתוקף עושה על מנת למצוא את המפתח הסימטרי של הלקוח בין שרת האימות, עד כדי כך, שהמתקפה לא תהיה אפקטיבית ולא יעילה ותמשך המון זמן, למשתמש ההבדל של החישוב לא יורגש משמעותית הבדל של שניות בודדות, אבל ככל שהתוקף יעשה את זה על מאגר גדול של סיסמאות, זה יקח המון זמן.

ההבדל בין הפונקציות שגורם לקושי חישוב המפתח סימטרי של כל סיסמא נובע מכך שמוסיפים ערך אקראי וייחודי של salt שמתווסף לסיסמה, ובכך גודל הסיסמא גדל יותר והכי חשוב מתקבל תוצאת hash שונה בעקבות זאת גם אם הסיסמא זהה בעקבות התוספת של המלח שהוא ייחודי.

בנוסף תפקידו של הפרמטר iterations=600000 - ככל שהמספרו גבוה יותר, כך יש יותר קושי לתוקפים לנסות לעשות חישוב על כל סיסמא ובכך להאט את התוקף ולהפוך את המתקפה לאיטית, לא יעילה ולא אפקטיבית.

אפשר לבחור במספר יותר גבוהה ממה שרשמתי, אני בחרתי לפי ההמלצה של OWASP .

בנוסף בפונקציה השנייה אנו מבצעים חישוב של sha512 במקום sha256 כפי שהתבקשנו בממ"ן, דבר זה מוסיף זמן לחישוב הhash של הסיסמה.

נשלב בנוסף עוד מנגנון לחיזוק ההגנה מפני מתקפת מילון לא מקוונת:

נשלב בקוד בדיקה שכאשר המשתמש בוחר סיסמא נבדוק אם הסיסמא שלו היא סיסמא נפוצה או מורכבת מסיסמא נפוצה, אם הוא בחר בסיסמא נפוצה או שהיא מורכבת מסיסמא נפוצה אז לא נקבל אותה ונאלץ אותו לבחור סיסמא אחרת יותר "חזקה" (בדומה למנגנון שיש בהרבה אתרים באינטרנט כאשר נרשמים).

בנוסף נוסיף סט חוקים לבחירת סיסמא למשל: שתהיה מאורך מסויים (מעל 8 תווים), שתכלול תווים מיוחדים(@#$%&\*^ וכו'), מספרים, אותיות גדולות וקטנות. מה שיאלץ את המשתמש לבחור סיסמה חזקה מאוד ולא סיסמה נפוצה שתהיה במילון של התוקף.

בכך אנחנו בעצם מונעים שהסיסמאות של המשתמשים שלנו לא יופיעו במאגר המילון של הסיסמאות הנפוצות של התוקפים ואם הסיסמאות שלהם לא מופיעים שם, אזי מתקפת המילון לא תעבוד בשל כך.

לסיכום: עם ההצעה שהצעתי, יהיה אפשר למנוע מתקפת מילון לא מקוונת.