

הצפנה מקצה לקצה

מאת עידן שכטר

הקדמה

עוד מימי העת העתיקה, בני אדם פיתחו שיטות להעביר ביניהם מסרים בצורה שתקשה על צד שלישי, כזה שלא אמור לקבל את אותם מסרים, להסיק את התוכן המקורי שלהם. אותן שיטות אשר השתכללו לאורך השנים ולצד צמיחתה הבלתי פוסק של הטכנולוגיה, הפכו לאלגוריתמים מורכבים העושים שימוש במודלים מתמטיים שונים, המהווים סטנדרט בעולמות האינטרנט והתקשורת.

בעולם בו הזיקה לפרטיות הולכת וגוברת, אפליקציות רבות עושות מאמצים להטמיע מנגנונים שונים כדי להבטיח לקהל המשתמשים שלהן שהמידע שלהם מוגן ופרטיותם מובטחת.

אחד המנגנונים הרלוונטיים ביותר, המאפשר למשתמשים להחליף ביניהם מידע מוצפן מבלי שהשרת המתווך ביניהם (לדוגמה - שרת של אפליקצית מסרים מידיים) יוכל לפענחו, הינה הצפנה מקצה לקצה. מאחר והמאמר לא עוסק בהצפנה על בוריה, אלא בא להסביר קונספט ספציפי, לא נצלול לעומק כל הביטויים והמנגנונים אותם נפגוש לאורך המאמר. לכן, אמליץ לקורא לצאת למסע ולחקור את עולם ההצפנה כדי להבטיח הבנה עמוקה של הנושאים השונים שנזכיר.

האינטרנט של פעם

לפני שהפרוטוקול SSL נכנס לחיינו ושינה את הצורה בה אנו מתקשרים מעל האינטרנט, ישויות אינטרנטיות החליפו ביניהן מידע בצורה גלויה, ללא הצפנה. מצב זה איפשר לתוקף מתוחכם להתגנב אל נתיב תקשורת, להעביר את תעבורת הרשת דרכו (MITM) ולהיחשף לכמויות גדולות של מידע חיוני. אין ספק שמנקודת מבט עכשווית, מצב זה נשמע הזוי לחלוטין.

אך אם ניקח צעד אחורה ונבחן את הצורה בה האינטרנט עובד כיום, נגלה כי למרות אותם פרוטוקולי הצפנה מתקדמים, המידע שלנו עדיין נמצא בסיכון.

החברה בה אנו חיים צורכת שירותים רבים המתבססים על האינטרנט: רשתות חברתיות, אפליקציות מסרים, אתרי קניות, בנקים ועוד רבים אחרים. כל אלה זקוקים לאמון של המשתמש הפשוט, אותו משתמש



שבוחר להפקיד לידי אותם שירותים פרטים אישיים ומידע רב נוסף הנובע מאופי השירות הנצרך (לדוגמא, רשתות חברתיות - מעגלי חברים ותמונות, אנשי קשר ואלפי הודעות אישיות וקבוצתיות) שהשירות שאותו הוא צורך עושה שימוש בפרטים שמסר אך ורק בכדי לספק את השירות הרלוונטי.

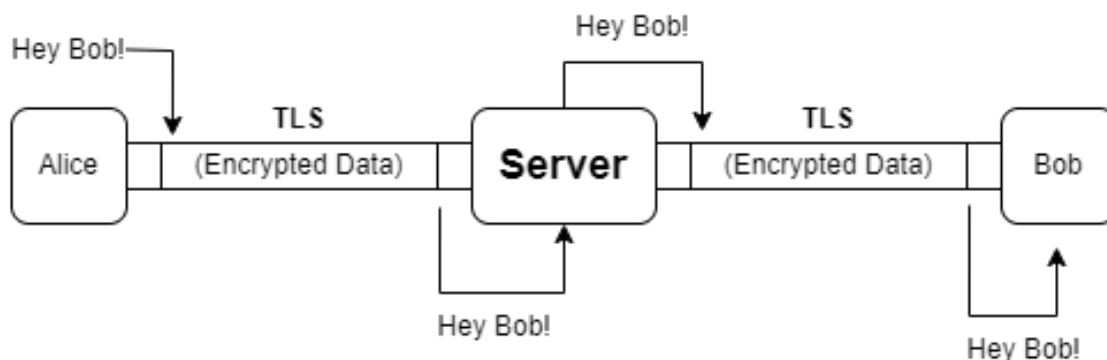
לדוגמא, משתמש ברשת חברתית המנהל שיחות אישיות עם חבריו באמצעות צ'אט, סומך על אותה רשת חברתית שלא תבחן את הודעותיו האישיות ותעשה בהן שימוש שיפגע בפרטיות שלו לצורך מקסום רווחים (לדוגמא, טירגוט פרסומי של משתמש על פי תוכן ההודעות ששלח לחברו). בנוסף, אותו משתמש מצפה שהמידע האישי שלי לא יימסר לגורם שלישי למטרות כאלה ואחרות, לדוגמא - לצורך האזנת סתר או ריגול, תופעה שקיימת בעיקר במדינות בעלות משטר עם אופי דיקטטורי בהן חופש הביטוי והזכות לפרטיות נעקרים מן היסוד (יש לציין שאותו עיקרון יכול לקבל משמעות הפוכה כאשר מדובר במידע שיכול למנוע פגיעה בחפים מפשע - חרב פיפיות).

גם אם השירות הרלוונטי אכן מבטיח את פרטיותו של המשתמש, אין הדבר מונע ממידע אישי של משתמש ליפול לידיים הלא נכונות במקרה של מתקפת סייבר או הדלפה. כלומר, עצם הימצאות המידע במאגרי נתונים של השירות בצורתו המקורית והלא מוצפנת היא בעיה בפני עצמה.

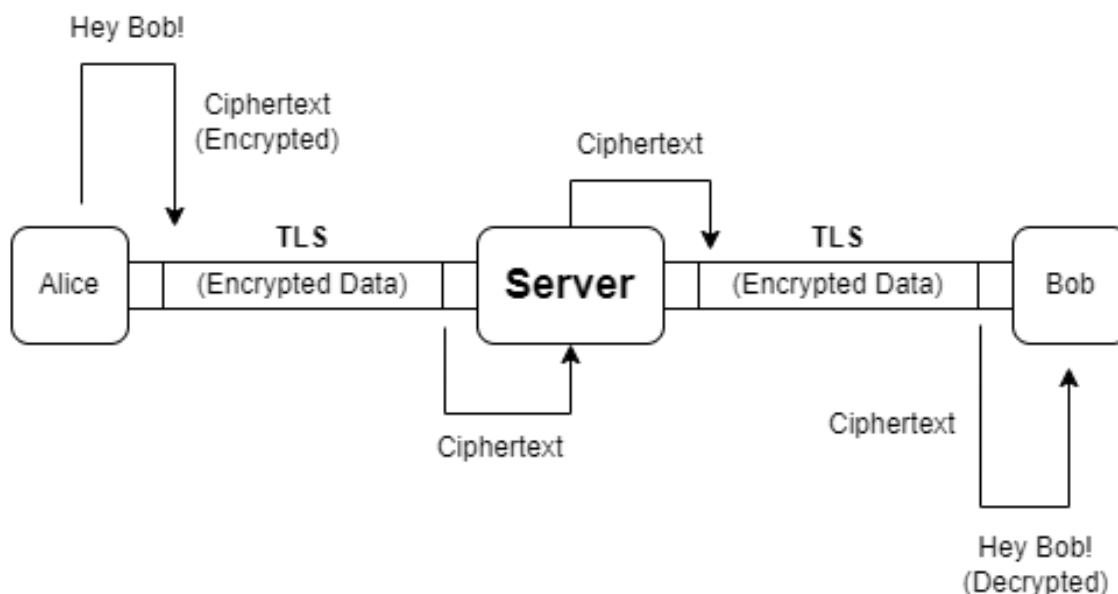
מי יוכל להבטיח לנו שהמידע שלנו לעולם לא יודלף? או שתוכן השיחות האישיות שלנו לא עוברות לגורם שלישי? שימוש בהצפנה מקצה לקצה פותר את הבעיה הזו.

הצפנה מקצה לקצה

בשונה מהפרוטוקול TLS שמצפין מידע בין שרת ללוקח ברמת התעבורה (הצד שכותב ל-Socket מצפין את המידע, הצד שקורא מה-Socket מפענח את המידע):



במימוש הצפנה מקצה לקצה, נצפין את המידע ברמת האפליקטיבית (ללא קשר להצפנה שהפרוטוקול TLS יבצע עבורנו) כך שרק הצד המקבל (זה שההודעה הרלוונטית מיועדת לו) יוכל לפענחו:



אך איך הדבר מתבצע בפועל? בואו נצלול לעניינים!

פרקטיקה

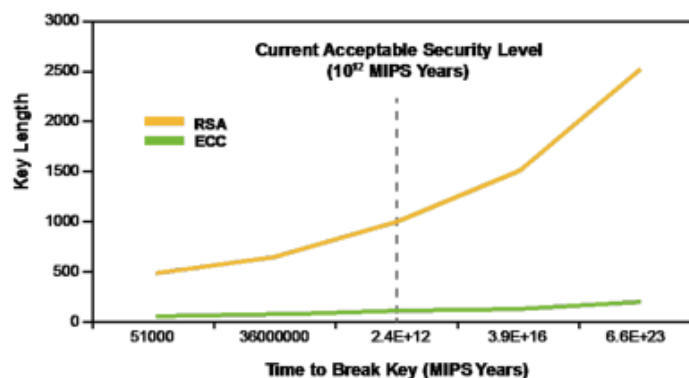
על מנת להבין איך הצפנה מקצה לקצה ממומשת בפועל, בדגש על אפליקציות מסרים מידיים, נפתח מנגנון שכזה ב-Python. אליס ובוב הם משתמשים באפליקצית המסרים המידיים Zubian (אפליקציה מומצאת) המממשת הצפנה מקצה לקצה. אליס מעוניינת לשלוח לבוב הודעה סודית וחשובה, המנגנון שלנו יפעל בצורה הבאה:

1. אליס תבקש משרת האפליקציה של Zubian את המפתח הפומבי של בוב (לצורך החלפת מפתחות)
2. אליס תצפין את ההודעה הסודית באמצעות הצפנת AES
3. אליס תייצר ערך Shared Secret על ידי שימוש ב-Diffie-Hellman
4. אליס תגזור מפתח מה-Shared Secret שיצרה, על ידי שימוש בפונקציה גזירת מפתח (KDF)
5. אליס תצפין את מפתח ה-AES (פעולה המכונה עטיפה) בו השתמשה כדי להצפין את ההודעה, על ידי המפתח שגזרה (סעיף 4)
6. אליס תשלח לבוב את ההודעה
7. לאחר שבוב יקבל את ההודעה, הוא יבצע את אותן פעולות בדיוק, יפענח את מפתח ה-AES שאליס הצפינה וישתמש בו כדי לפענח את תוכן ההודעה המוצפנת.

```
# Create a keypair for Alice and Bob
alice_keypair = generate_ec_keypair()
bob_keypair = generate_ec_keypair()
```

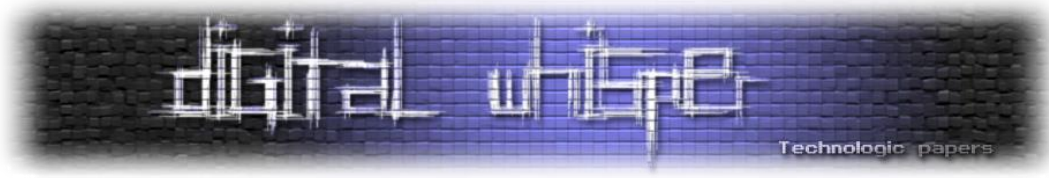
כאשר אליס ובוב התחברו לראשונה למשתמש שלהם באפליקצית Zubian, האפליקציה יצרה עבור כל אחד מהם צמד מפתחות (Keypair) אסימטרי, המשמשים לביצוע [הצפנה מבוססת עקומים אליפטיים \(ECC\)](#) (להצפנה מבוססת עקומים אליפטיים מספר יתרונות על פני הצפנת RSA, מעבר לכך שהיא מהירה יותר, היא מקנה רמת אבטחה גבוהה ביחס לגודל מפתח קטן יחסית, בעוד שהצפנת RSA תצטרך להשתמש במפתח ארוך משמעותית כדי לספק את אותה רמה של אבטחה).

Figure 1. RSA and ECC Performance⁽⁶⁾



This chart presents what key lengths of each algorithm provide a level of security measured in time in MIPS-years to break the security. This illustrates that ECC is more efficient.

[מקור: <https://ww1.microchip.com/downloads/en/DeviceDoc/00003442A.pdf>]



כדי לייצר את המפתחות הרלוונטיים ולצורך ביצוע פעולות קריפטוגרפיות נוספות, נשתמש בספרייה cryptography:

```
from cryptography.hazmat.primitives.asymmetric.ec import EllipticCurvePublicKey, EllipticCurvePrivateKey
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import padding
```

לוגיקת ייצור המפתחות:

```
def generate_ec_keypair() -> dict:
    private_key = ec.generate_private_key(ec.SECP256R1())
    public_key = private_key.public_key()

    return {
        "public": public_key,
        "private": private_key
    }
```

את המפתח הפומבי שלהם ישלחו לשרת האפליקציה Zubian, כך שמשמש שמעוניין לשלוח להם הודעה יוכל לבצע את התהליך נשאר כעת. כמו כן, המפתח הפרטי נשמר במקום בטוח, ולעולם לא ישותף.

כעת, אליס תצפין את המידע הרלוונטי באמצעות הצפנת AES, אשר מסוגלת להצפין כמויות גדולות של מידע במהירות ומגודרת כסטנדרט שנים רבות:

```
message = "this is an important message".encode()
ciphertext_json = aes_encrypt(data=message)

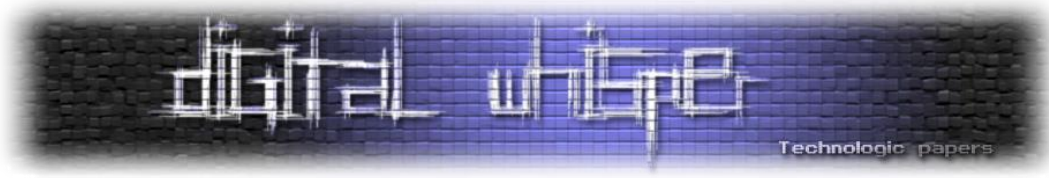
aes_key = ciphertext_json["key"]
iv = ciphertext_json["iv"]
encrypted_message = ciphertext_json["ciphertext"]
```

נייצר מפתח רנדומלי באורך 32 בתים וערך IV, בהם נשתמש כדי לבצע הצפנה AES במוד CBC (להצפנת AES 5 מצבים שונים, ולכל אחד מהם יתרונות וחסרונות. לדוגמא, הצפנת AES במצב GCM מספקת הגנה מפני שינוי לא רצוי של התוכן המוצפן, ומאפשרת לצד המקבל להבטיח את האותנטיות שלו):

```
def aes_encrypt(data: bytes) -> dict:
    key = os.urandom(32)
    iv = os.urandom(16)

    cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    encryptor = cipher.encryptor()
    pad = padding.PKCS7(128).padder()
    padded_message = pad.update(data) + pad.finalize()
    ciphertext = encryptor.update(padded_message) + encryptor.finalize()

    return {
        "key": key,
        "iv": iv,
        "ciphertext": ciphertext
    }
```



מכיוון שהצפנת AES הינה הצפנה סימטרית, בה מפתח בודד משמש גם להצפנה וגם לפענוח, אליס תצטרך להצפין את המפתח הרלוונטי פעולה המכונה "עטיפה", באמצעות מפתח מיוחד שרק היא ובוב יכולים לייצר. כדי לייצר את המפתח המיוחד, אליס תשתמש בפרוטוקול החלפת מפתחות Diffie-Hellman (בקיצור - DH, ומבקרה שלנו, ECDH - Elliptic Curve Diffie-Hellman).

הפרוטוקול יאפשר לאליס לייצר ערך מיוחד המכונה Shared Secret, אשר תלוי בצורה ישירה במפתחות המזהים שלה ושל בוב. את ערך ה-Shared Secret יהיה ניתן לחשב באמצעות המפתח הפרטי של אליס אל מול המפתח הפומבי של בוב, ולהפך. אליס תייצר את ה-Shared Secret באמצעות המפתח הפרטי שלה, והמפתח הפומבי של בוב. בוב ייצר את אותו ערך בדיוק תוך שימוש במפתח הפרטי שלו, והמפתח הפומבי של אליס:

```
def create_shared_secret(our_private_key:
    EllipticCurvePrivateKey, their_public_key: EllipticCurvePublicKey) -> bytes:
    return our_private_key.exchange(ec.ECDH(), their_public_key)
```

```
shared_secret_alice = create_shared_secret(alice_keypair["private"], bob_keypair["public"])
shared_secret_bob = create_shared_secret(bob_keypair["private"], alice_keypair["public"])
print(f"Shared-Secret, using alice's private key and bob's public key -
    {b64encode(shared_secret_alice).decode()}")
print(f"Shared-Secret, using bob's private key and alice's public key -
    {b64encode(shared_secret_bob).decode()}")
```

התוצאה:

```
Shared-Secret, using alice's private key and bob's public key - yPw64gMJSVrPT+dDedF08hQIxU6gJfYVCz5XP7z7ZLY=
Shared-Secret, using bob's private key and alice's public key - yPw64gMJSVrPT+dDedF08hQIxU6gJfYVCz5XP7z7ZLY=
```

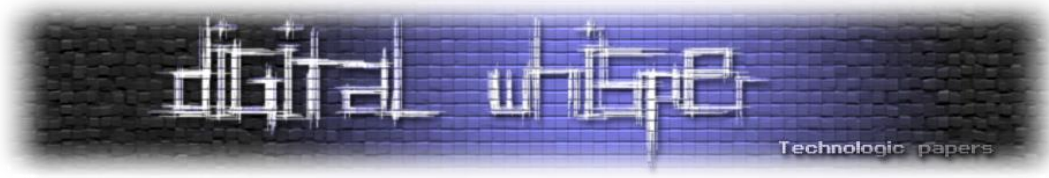
קסמים של מתמטיקה!

חשוב לציין, כי מאחר ולשרת האפליקציה Zubian לעולם לא תהיה גישה למפתח הפרטי, היא לא תוכל לייצר את ערך ה-Shared Secret הרלוונטי, ולכן לא תוכל לפענח את המידע המוצפן.

כעת, כדי להוסיף שכבת אבטחה נוספת אליס תשתמש בפונקציית גזירת מפתח (KDF) - פונקציה המקבלת קלט (במקרה שלנו ה-Shared Secret) ומחזירה ערך המשמש כמפתח או כמזהה ייחודי לקלט שקיבלה, בדרך כלל על בסיס פונקציית גיבוב ומספר פרמטרים נוספים (הוספת Salt לערך ה-Shared Secret, גיבוב רב פעמי של הקלט):

```
salt = b"alice-and-bob-4ever"
derived_key = do_kdf(shared_secret_alice, salt)
```

```
def do_kdf(shared_secret: bytes, salt: bytes) -> bytes:
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=1000000,
    )
    return kdf.derive(shared_secret)
```

פעולה זו תאפשר לאליס ולבוב לייצר מפתח זהה, על סמך ה-Shared Secret שייצרו באמצעות המפתחות שלהם. (אף על פי שלגורם שלישי יהיה מאד קשה לנחש את ערך ה-Shared Secret המשותף שלהם, השימוש בפונקציית KDF מעלה את המחיר החישובי במקרה של ניסיון מתקפת brute force, מאפשר יצירה של מפתח ארוך יותר מאורכו של ה-Shared Secret ואף מאפשר ייצור של מספר מפתחות שונים על בסיס אותו Shared Secret).

כעת, תשתמש אליס במפתח שגזרה כדי להצפין את מפתח ה-AES שהשתמשה בו כדי להצפין את המידע. ההצפנה בה תשתמש כדי לעטוף את מפתח ה-AES, תהיה הצפנת AES בעצמה (כעת בטוח להשתמש בהצפנת AES, מאחר ורק אליס ובוב יכולים לייצר את המפתח שגזרנו), רק שהפעם תשתמש בהצפנת AES במוד ECB, כזו שלא דורשת ערך IV.

```
wrapped_aes_key = wrap_aes_key_with_derived_key(aes_key=aes_key, derived_key=
derived_key)

def wrap_aes_key_with_derived_key(aes_key: bytes, derived_key: bytes) -> bytes:
    cipher = Cipher(algorithms.AES(derived_key), modes.ECB())
    encryptor = cipher.encryptor()
    return encryptor.update(aes_key) + encryptor.finalize()
```

ההודעה של אליס מוצפנת ומוכנה לשליחה! אליס תארז את ההודעה המוצפנת לצד המפתח העטוף, ערך ה-IV ששימש להצפנת ההודעה ולצד המפתח הפומבי שלה בפורמט JSON, ותשלח אותו לשרת.

```
message = {
    "public_key": alice_keypair["public"],
    "ciphertext": encrypted_message,
    "wrapped_key": wrapped_aes_key,
    "iv": iv
}
```

המידע שקיבל השרת חסר ערך עבורו, וזאת כי אין ביכולתו לפענח את ההודעה הרלוונטית. במצב זה, השרת משמש כשליח בטוח ואמין, ואליס אף תוכל לסמוך על השרת שישמור עבורה את ההודעה לטווח הארוך, מאחר ודליפה של התוכן המוצפן לא מהווה סיכון. (בפועל, יצירת גיבוי מוצפן הוא אינו דבר של מה בכך, מאחר ועל הלקוח להוכיח לשירות שהוא ראוי לקבל את המידע המוצפן, גם אם הוא לא זה שהצפין אותו. לדוגמא, במצב בו משתמש מתחבר אל החשבון שלו ממכשיר נוסף, המקבל מפתח פרטי חדש, שלא נעשה בו שימוש כדי להצפין את תוכן הגיבוי).

לאחר שיקבל את ההודעה, בוב יבצע רצף פעולות דומה. ראשית, ישתמש במפתח הציבורי של אליס כדי לייצר ערך Shared Secret, אותו יכניס כקלט לפונקציית KDF כדי לייצר את המפתח המשותף (המפתח שבאמצעותו אליס הצפינה את מפתח ה-AES):

```
shared_secret_bob = create_shared_secret(bob_keypair["private"], message["public_key"])
derived_key = do_kdf(shared_secret_bob, salt)
```

כעת, יוכל בוב לפענח ("לקלף") את מפתח ה-AES המוצפן:

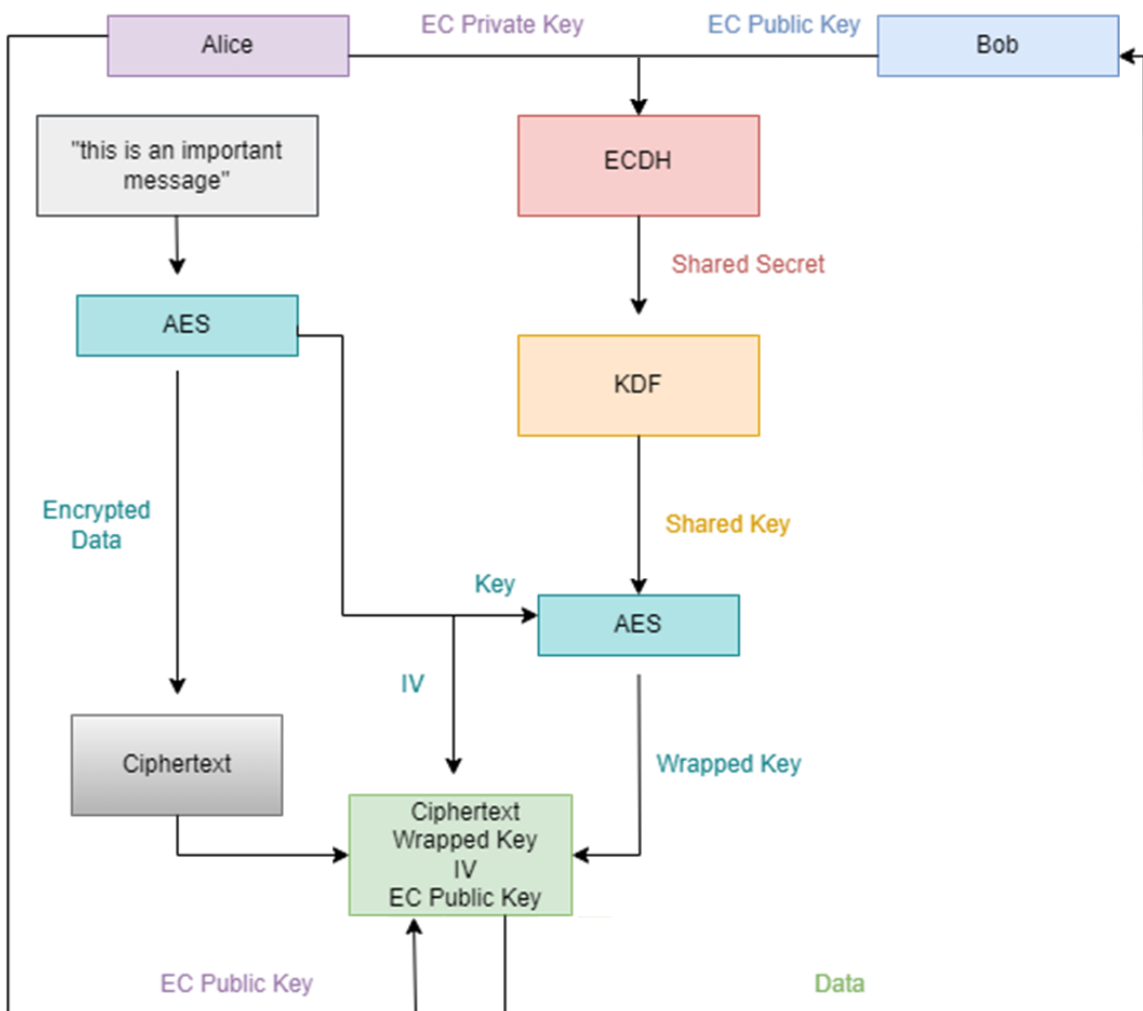
```
def unwrap_aes_key_with_derived_key(aes_key: bytes, derived_key: bytes) -> bytes:
    cipher = Cipher(algorithms.AES(derived_key), modes.ECB())
    decryptor = cipher.decryptor()
    return decryptor.update(aes_key) + decryptor.finalize()
```

```
unwrapped_aes_key = unwrap_aes_key_with_derived_key(message["wrapped_key"], derived_key)
```

לאחר שיקלף את המפתח בהצלחה, יוכל להשתמש בו ובערך ה-IV שסיפקה לו אליס כדי לפענח את הטקסט המוצפן:

```
decrypted_message = aes_decrypt(message["ciphertext"], unwrapped_aes_key, message["iv"])
print(decrypted_message.decode())
```

בוב פיענח את המידע וקרא את תוכן ההודעה החשובה! התהליך המתרחש בכל פעם שאליס תרצה לשלוח הודעה לבוב, ולהפך:



בין אם מדובר בהודעה טקסט, קובץ, או כל פורמט אחר שאפליקציית Zubian מאפשרת, הצד השולח יצפין את המידע באמצעות מפתח AES (מפתח AES חדש עבור כל הודעה!) אותו הוא יעטוף באמצעות מפתח משותף (Shared Secret -> KDF) ולבסוף ישלח אותו אל היעד, לצד המידע המוצפן.

אף על פי שרצף הפעולות שתיארנו עשוי להיראות מסורבל וארוך, קסם ההצפנה מאפשר לנו לבצע אותן בזמנים קצרים מאד גם כאשר מדובר בכמויות גדולות של מידע. מאחר ורצף הפעולות שתיארנו נחשב כה טריוויאלי באפליקציות מסרים מיידיות, בהן זמינות ומהירות בעלי עדיפות גבוהה, אלגוריתמים ופרימיטיביים קריפטוגרפיים ימומשו בספריות native לצורך יעילות כך שאפליקציה הכתובה בשפה גבוהה יותר, שלא דווקא רצה כקוד native מקומפל (Java, Python) תוכל "לקרוא" לפונקציות רלוונטיות על ידי טכניקות צימוד כאלה ואחרות. לצורך הדוגמא, ספריית cryptography בה השתמשנו מתבססת בין היתר על ספריית Rust המבצעת צימוד לספריית openssl המוכרת.

אם נקפוץ אל מימוש פונקציה ה-KDF בה השתמשנו, נוכל לראות את הקריאה לפונקציה מספריית rust_ssl:

```
def do_kdf(shared_secret, salt):
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=1000000,
    )
    return kdf.derive(shared_secret)

def derive(self, key_material: bytes) -> bytes:
    if self._used:
        raise AlreadyFinalized("PBKDF2 instances can only be used once.")
    self._used = True

    return rust_openssl.kdf.derive_pbkdf2_hmac(
        key_material,
        self._algorithm,
        self._salt,
        self._iterations,
        self._length,
    )
```

סיכום

במאמר זה הכרנו מקרוב את הדרך בה ממומשת הצפנה מקצה לקצה באפליקציות מודרניות, דיברנו על אלגוריתמי ההצפנה הנפוצים בהם נעשה שימוש במימוש מנגנונים שכאלה, וראינו כיצד שרשור שלהם בפעולות קריפטוגרפיות נוספות, המרגישות כקסם של ממש, מאפשר לנו לתקשר אחד עם השני בצורה בטוחה שמבטיחה את הפרטיות שלנו. את כלל הקוד ניתן להוריד מה-Repo ב-Github:

https://github.com/DWe2ee/dw_e2ee

על המחבר

עידן שכטר, בן 26, אוהב בעלי חיים ואת הים, חוקר בקבוצת NSO.