Technologic Papers

פתרון אתגרי הקריפטוגרפיה מגמר תחרות CSAW2018

מאת מתן אלפסי ואלון בן-צור

הקדמה

בין התאריכים 8-10 בנובמבר 2018 התקיים גמר תחרות CSAW בחיפה וארך כ-36 שעות. האתגרים להם ציפינו מכל היו אתגרי V35 - סדרת אתגרים המוסווים בתוך משחק מטעם הקבוצה VECTOR35. הקבוצה מוכרת במיוחד בזכות סדרת המשחקים שלה - Pwn Adventure ומבעוד מועד התגלה לנו שהם הכינו משחק גם לתחרות הזו. למרבה הצער השרתים של המשחק נפלו בשלב מוקדם בתחרות וחזרו רק בשלב מאוחר - אז החלטנו לעבוד על אתגרי קריפטוגרפיה.

אתגרי קריפטוגרפיה לרוב אינם דורשים ידע גבוה במתמטיקה, ולאחר התנסות בשיטות הצפנה ואימות מודרניות כאלו ואחרות - אתגרים מהסוג מרגישים דווקא ברורים למדי. לאחר עבודת מחקר פשוטה תמיד מצליחים ללמוד משהו חדש, בין אם בפן הטכנולוגי ובין אם בפן המתמטי, אם אמרה זו לא ברורה בינתיים, תראו דוגמא בהמשך. האתגרים הקריפטוגרפים בתחרות היו בנושאים מאוד מוכרים: DSA ו-RSA, אך בשניהם וקטורי ההתקפה היו קשורים דווקא במימוש ובמתן השירותים הלקוי של השרת.

פתרון אתגר א': Disastrous Security Apparatus

תיאור האתגר

באתגר זה קיבלנו קוד מקור של שרת HTTP מבוסס Flask. ראשית השרת מאתחל את עצמו:

1. מאתחל אובייקט "מפתח פרטי" של DSA מתוך קובץ מפורמט pem.
2. מאתחל אובייקט הצפנה ואות'נטיקציה מסוג Fernet המסוגל להצפין הודעות עם AES128 ולאמת אותן.

לנו כמובן אין גישה למפתחות של שני האובייקטים:

```
ctf_key = load_pem_private_key(
    pem_data, password=None, backend=default_backend()
)
CSAW_FLAG = os.getenv("CSAW_FLAG")
FERNET = Fernet(Fernet.generate_key())
```

ה-route הראשון של השרת התופס את העין הוא capture, אשר מקבל שני ארגומנטים: "challenge" - הודעה מוצפנת ומאומתת מהשרת, ו-"signature" - חתימה דיגיטלית שיוצרה באמצעות DSA ומספקת אותנטיות למסר המוצפן שנשלח בפרמטר הראשון.

עושה רושם שמתן challenge וחתימה אותנטית יגרמו לשרת לגלות את הדגל, זאת על פי תוכן הפונקציה:

```
@app.route("/capture", methods=["POST"])
def capture():
    sig = binascii.unhexlify(request.form["signature"])
    challenge = request.form["challenge"].encode("ascii")

    # error unless successfully decrypted w/ fernet
    FERNET.decrypt(challenge)

    # error unless challenge verified against public key & sha256
    ctf_key.public_key().verify(sig, challenge, hashes.SHA256())
    return "flag{%s}" % CSAW_FLAG
```

לאחר בדיקת פענוח ואימות תוכן האתגר, השרת בודק שהחתימה נעשתה בשימוש המפתח הפומבי של השרת עם שימוש בפונקציית גיבוב sha256.

מעבר על 2 הראוטים הבאים מראים לנו שהשרת מספק בשמחה שירות של הצפנת הודעה קבועה מראש עם Fernet ושירות חתימת DSA באמצעות המפתח הפרטי של השרת לכל הודעה שנחפוץ בה:

```
@app.route("/challenge")
def challenge():
    return FERNET.encrypt(b"challenged!")


@app.route("/sign/<data>")
def signer(data):
    r, s = sign(ctf_key, data)
    return json.dumps({"r": r, "s": s})
```

מצוין, עוד לפני שנסתכל במימוש של הפונקציה החותמת - sign (פונקציה לוקאלית), עושה רושם שיש לנו בדיוק את שני המצרכים להם אנו זקוקים בכדי לקבל את הדגל. לכאורה, ניתן לבקש מהשרת את ה-"challenge", לאחר מכן לחתום עליו עם המפתח הפרטי של השרת באמצעות DSA ולקבל את הדגל משום שהפענוח של האתגר והאימות אמורים להיות לגיטימיים.

שלחנו ל-capture את המצרכים שקיבלנו והשרת "עף". נזרקה חריגה באימות מול המפתח הפומבי, הסיבה טמונה בהבדל בין המימוש של פונקציית החתימה, לבין הוריפיקציה של החתימה. נסתכל על פונקציית המימוש:

```
def sign(ctf_key, data):
    data = data.encode("ascii")
    pn = ctf_key.private_numbers()
    g = pn.public_numbers.parameter_numbers.g
    q = pn.public_numbers.parameter_numbers.q
    p = pn.public_numbers.parameter_numbers.p
    x = pn.x
    k = random.randrange(2, q)
    kinv = _modinv(k, q)
    r = pow(g, k, p) % q
    h = hashlib.sha1(data).digest()
    h = int.from_bytes(h, "big")
    s = kinv * (h + r * x) % q
    return (r, s)
```

עוד מלפני שבדקנו את המימוש אל מול הספציפיקציה (המימוש בסדר גמור), ניתן לזהות את הבעיה בקלות, פונקציית החתימה משתמשת באלגוריתם גיבוב מסוג SHA-1 (ניתן לראות בהשמה של הפרמטר h) ואילו הפונקציה שמאמתת משתמשת ב-SHA-2 מה שהופך את פונקציית החתימה (כרגע) ללא שימושית.

המשכנו להסתכל על שני ה-routes הנותרים, עושה רושם בתחילה שהם מיותרים כמעט לחלוטין, אך הם מתבררים כנקודת החולשה של התוכנית:

```
@app.route("/forgotpass")
def returnrand():
    random_value = binascii.hexlify(struct.pack(">Q",
random.getrandbits(64)))
    return random_value.decode("ascii")

@app.route("/public_key")
def public_key():
    pn = ctf_key.private_numbers()
    return json.dumps({
            "g": pn.public_numbers.parameter_numbers.g,
            "q": pn.public_numbers.parameter_numbers.q,
            "p": pn.public_numbers.parameter_numbers.p,
            "y": pn.public_numbers.y
    })
```

ה-route הראשון הוא המוזר מכולם, עושה רושם שהוא מחזירמחרוזת של תווים אקראיים בהקסאדצימל. ה-route השני מייחצן את g, q, p ו-y, ארבעת הפרמטרים המהווים את המפתח הפומבי של התוכנית, זה יעזור בהמשך.

מבט חטוף ברשימת הספריות שהתוכנית מייבאת וניתן לראות שהספריה בה התוכנית משתמשת כדי לייצר מספרים אקראיים היא הספרייה הפנימית של פייתון הידועה לשמצה:

> **Warning:** The pseudo-random generators of this module should not be used for security purposes. Use `os.urandom()` or `SystemRandom` if you require a cryptographically secure pseudo-random number generator.

פיצוח DSA

למעשה נקודת החולשה של DSA טמונה בבחירת הפרמטר k עם ערך אקראי. אפשרות לנבא ערך זה מסוגלת להביא לשבירת יכולת האימות באמצעות גילוי של המפתח הפרטי.

כעת, לפי ההגדרות ב-DSA:

$$s = k^{-1}(SHA1(m) + xr)\%q$$

כאשר q הוא מידע ציבורי, m בשליטתנו, את s, r אנחנו מקבלים (זאת בעצם החתימה) ואת k אנחנו חוזים. לכן נקבל:

$$(sk - SHA1(m)) \cdot r^{-1}\%q = x$$

צד שמאל במשוואה מורכב ממידע שיכול להיות לנו (את k נוכל לגלות בהמשך), כלומר בר חישוב.

בידיעת המספר החד-פעמי האקראי k והמפתח הפומבי, יש בידינו את היכולת לשבור את אותנטיות החתימה ולהכין חתימה משלנו עם המפתח הפרטי של השרת. כעת כל שעלינו לעשות הוא לנבא את

המספר החד-פעמי k. ניתן לעשות זאת משום שהמודול random אינו בטוח מבחינה קריפטוגרפית וכותבי השרת היו נחמדים מספיק כדי לספק לנו endpoint שיחזיר לנו מחרוזת אקראית.

כיצד ניתן לשבור את random? הספרייה המובנית של פייתון משתמשת באלגוריתם הידוע Mersenne Twister, או בגרסה היותר ספציפית שלו - MT19937 שמסוגל לייצר מספרים פסאודו-אקראיים בדיוק בגודל 32 ביט. גם כאשר תוכניות בפייתון מבקשות מספרים בגדלים גבוהים יותר, מתבצע שרשור וחיתוך של מספרים אקראיים בגודל 32-ביט, לכן כשבתוכנית k מיוצר באמצעות getrandbits עם הארגומנט 64 בכדי לייצר מספר אקראי בגודל 64-ביט, למעשה האלגוריתם פשוט נקרא פעמיים ומתבצע שרשור.

הכנת האקספלויט

השתמשנו בספרייה חיצונית[1] שמצאנו ב-GitHub שבהינתן 624 מספרים אקראיים שנוצרו ברצף על ידי האלגוריתם הנ"ל, היא תהיה מסוגלת למצוא את המצב הפנימי של המודול random ולפיכך לנחש את המספרים הבאים בדיוק קרוב מאוד ל-100%.

בכדי למצוא את k, נעשה 312 קריאות לשרת, שיגרמו לו לייצר 624 מספרים אקראיים בגודל 32 ביט:

```
print '[-] fetching random numbers from server'
for i in range(312):
    response = get_random_from_server()
    r1, r2 = response[:8], response[8:]

    random_numbers.append(int(r2, 16))
    random_numbers.append(int(r1, 16))

print '[-] submitting numbers to rand cracker'
rc = RandCrack()
for n in random_numbers:
    rc.submit(n)

prediction = rc.predict_getrandbits(64)
real = int(get_from_server(), 16)
print '[-] testing prediction {} against server {}'.format(prediction,
real)

assert prediction == real
print '[+] prediction seems good!'
```

[1] https://github.com/tna0y/Python-random-module-cracker

כעת כשהצלחנו לנחש את המצב הפנימי של ספריית הראנדום בשרת, נוכל לחתום את האתגר מול ראוט החתימה של השרת, לחלץ את המפתח הפרטי כאשר יש לנו ניחוש די טוב של k, ולחתום בעצמנו שוב על האתגר עם SHA-256 במקום SHA-1:

```
predicted_k = rc.predict_randrange(2, q)
challenge = get_challenge()
signed_response = server_sign(challenge)
r, s = signed_response['r'], signed_response['s']
h = int(hashlib.sha1(challenge).digest().encode('hex'), 16)
x = ((s * predicted_k) - h) * _modinv(pow(g, predicted_k, p) % q, q) % q
```

וכעת כשיש בידינו את המפתח הפרטי (בהנחה כמובן שצדקנו בחיזוי של k), נוכל לייצר חתימת SHA-256. ניתן לבצע שימוש חוזר ב-r (חלק מתוצאת החתימה) משום שאינו מושפע מהערך של h (תוצאת פונקציית הגיבוב):

```
correct_h = int(hashlib.sha256(challenge).digest().encode('hex'), 16)
correct_s = _modinv(predicted_k, q) * (correct_h + r * x) % q
fake_signature = encode_dss_signature(r, correct_s).encode('hex')
print '[+] flag=%s' % server_capture(challenge, fake_signature)
```

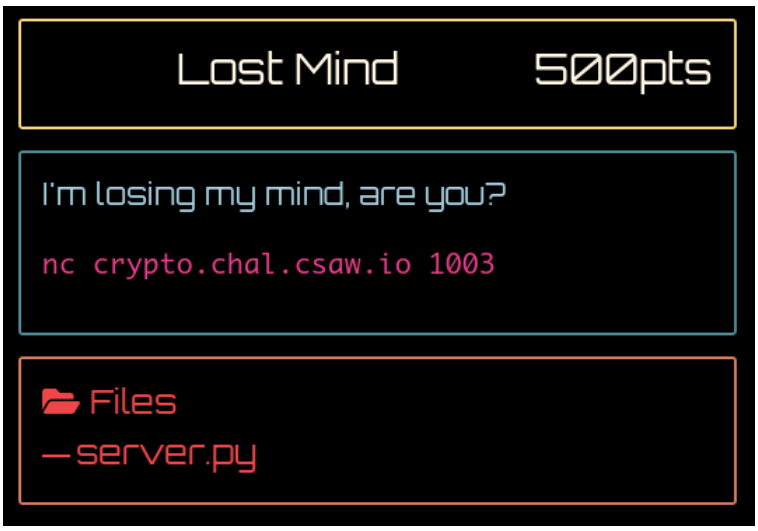
ואכן הדגל התקבל:

```
[+] flag=flag{NowyourereadytocrackthePS3YeahSonydidthiswithECDSA}
```

הדגל הוא אכן רפרנס לפעם שסוני השתמשה במנגנון ECDSA (גירסה של DSA) בפלייסטיישן-3 וערך ה-k שבו נעשה שימוש היה סטטי וחזר על עצמו עבור כל חתימה[2], רעיון שכמובן גרם לחתימה חסרת שימוש ולפריצת הקונסולה באופן גורף (מביך).

[2] https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm#cite_ref-2

פתרון אתגר ב': Lost Mind



ניתן לקרוא את המאמר הבא כהקדמה לחלק זה:

https://www.digitalwhisper.co.il/files/Zines/0x19/DW25-4-MathBaseRSA.pdf

תיאור האתגר

באתגר קיבלנו שרת TCP שמוגדרת אצלו מחלקת RSA סטנדרטית. כל מופע של מחלה זו מייצג מימוש של מערכת RSA עם ראשוניים p,q המוזנים כפרמטרים, כאשר המפתח הפומבי e מוגרל בעת יצירת כל מופע. המחלקה מממשת פונקציות הצפנה ופענוח.

בתחילת כל חיבור, השרת מגריל ראשוניים p,q מסדר גודל של 512, ויוצר אובייקט RSA מתאים. לאחר מכן, הלקוח מבקש "חתיכה" מהדגל בגודל ובהיסט כרצונו (הנתונים בבתים). לאחר מכן, לחלק זה משורשר רצף בתים אקראי כך שהאורך של התוצאה הוא 123 בתים. תוצאה זו מוצפנת על ידי המערכת RSA ונשלחת ללקוח:

```
def get_flag(off, l):
    flag = open('flag', 'r').read().strip()
    init_round = 48
    t_l = 123

    if off + l > len(flag) or off < 0 or l < 1:
            exit(1)

    round = init_round + len(flag) - l
    flag = flag[off:off+l] + os.urandom(t_l - l)

    return flag, round
```

לאחר מכן, ללקוח ניתנות שתי אפשרויות:

1. קבלת תוצאת ההצפנה של כל הודעה שיבחר.

2. קבלת הבית הראשון (LSB) של פענוח טקסט מוצפן כרצונו.

ניתנת לו כמות סיבובים התלויה בגודל החתיכה (כפי שמוצגת בפונקציה במשתנה round), כך שבכל סיבוב ניתן לו לבצע אחת מפעולות אלה. לבסוף, השרת מנתק את החיבור.

RSA LSB Oracle Attack

נתבונן לרגע בבעיה אחרת.

נניח כי נתון לנו מפתח פומבי (N,e) למערכת RSA וכי נתון לנו מסר $C' = M^e \% N$ שהוצפן ע"י מערכת זו. כמו כן, נניח כי נתונה לנו קופסה שחורה Ω המקבל טקסט מוצפן C ומחזירה את הזוגיות של הפענוח של C. באופן מפורש, אם d ההפכי של e מודולו $\varphi(N)$, אז מתקיים: $\Omega(C) = (C^d \% N) \% 2$. האם בתנאים אלו ניתן לשחזר את M בזמן יעיל?

התשובה, באופן מפתיע, היא כן! האלגוריתם הוא פשוט ביותר: ניתן להניח כי N אי זוגי, אחרת קיבלנו את הפירוק לגורמים של N ומצאנו אתM באופן סטנדרטי. אז מניחים כי N אי זוגי. במקרה זה, נחשב את $D = 2^e \% N$. אז מתקיים: $C = DC' \% N = (2M)^e \% N$. נתבונן ב-$\Omega(C)$. זהו הביט הראשון של $2M \% N$. אם $2M < N$, אז $2M \% N = 2M$ ובפרט זהו מספר זוגי. לעומת זאת, אם $2M \geq N$ אז $2M \% N = 2M - N$ כלומר זהו מספר אי זוגי. לכן נקבל את התוצאה הבאה:

$$\Omega(C) = 0 \Leftrightarrow M < \frac{N}{2}, \Omega(C) = 1 \Leftrightarrow M \geq \frac{N}{2}$$

נוכל להמשיך באופן דומה ולקבל חסמים יותר ויותר טובים על M, כאשר בכל שלב אנחנו מקבלים חסם מהצורה:

$$\frac{tN}{2^k} \leq M \leq \frac{(t+1)N}{2^k}$$

כלומר נוכל אחרי $log_2(N)$ פעולות לקבוע את M בוודאות. ליתר דיוק, אם נניח כי קיבלנו $\frac{rN}{2^k} \leq M \leq \frac{(t+1)N}{2^k}$ בשלב k, אז נחשב את הזוגיות של $M \cdot 2^{k+1} \% N$, כלומר נכניס לאורקל את $C' \cdot 2^{k+1} \% N$. אם קיבלנו אפס, נסיק באותו האופן כמו מקודם:

$$M \cdot 2^k \% N \leq \frac{N}{2}$$

אבל הנחנו כי $\frac{tN}{2^k} \leq M \leq \frac{(t+1)N}{2^k}$ ולכן $tN \leq M \cdot 2^k \leq (t+1)N$, כלומר:

$$M \cdot 2^k - tN = M \cdot 2^k \% N \leq \frac{N}{2}$$

ולכן:

$$\frac{2tN}{2^{k+1}} \leq M \leq \frac{(2t+1)N}{2^{k+1}}$$

ובאותו האופן אם האורקל החזיר אחת נסיק כי:

$$\frac{(2t+1)N}{2^{k+1}} \leq M \leq \frac{(2t+2)N}{2^{k+1}}$$

כלומר אכן קיבלנו שיפור אקספוננציאלי (או לינארי באורך הייצוג של N) בחסם על M.

נסכם את החולשה באלגוריתם:

```
def exploit_lsb_oracle(N,e,C):
    U = N, L = 0, k = 0
    while U != L:
        D = (C * pow(2**k, e, N)) % N
        if lsb_oracle(D):
            L = (U+L)/2
        else:
            U = (U+L)/2
    return U
```

הבדלים בין האתגר למתקפה הקלאסית

יש כמה הבדלים בין הגרסה הקלאסית לגרסה שנתונה לנו, אז נעבור עליהם אחד אחד. מה שמוצפן בכל גישה לשרת הוא לא כל הדגל, אלא חתיכה מהדגל המרופדת רנדומלית מימין. אך בעיה זאת לא קריטית, שכן ניתן לבצע מספר גישות לשרת ולהרכיב את הדגל לאט לאט.

החלק המהותי יותר הוא שלא נתון לנו המידע הציבורי, כלומר N, e. עובדה זאת מעלה שתי בעיות: לא נוכל לבצע פעולות מודולו N וכן לא נוכל להעלות מספרים בחזקת e. הדרך לחשב את N היא טריק ידוע המתבסס על העובדה שאנחנו יכולים להצפין כל מסר נתון. נניח שאנחנו מצפינים את 2 ואת 4, כלומר מקבלים את $C_1 = 2^e\%N, C_2 = 4^e\%N$. אז מתקיים:

$$(C_1)^2\%N = (2^e\%N)^2\%N = 2^e \cdot 2^e\%N = 4^e\%N = C_2$$

כלומר $D_1 = (C_1)^2 - C_2$ מתחלק ב-N.

באותו באופן, אם $C_3 = 3^e\%N, C_4 = 9^e\%N$, נקבל כי $D_2 = (C_3)^2 - C_4$ מתחלק ב-N. כלומר N הוא מחלק משותף של שני המספרים האלו ואם יש לנו מזל הוא המחלק המשותף המקסימלי שלהם:

$$N = gcd\,(D_1, D_2)$$

כמובן שניתן להמשיך את תהליך זה עם D_3, D_4.. ככל שנרצה ולקבל סבירות גבוהה יותר ש-N הוא המחלק המשותף המקסימלי (של כולם ביחד):

$$N = gcd\,(D_1, D_2, D_3, \dots)$$

כמובן שהבעיה עם e היא מלאכותית: אנחנו לא צריכים לדעת אותו כי יש לנו את היכולת להצפין כבר!

```
def go(req):
    r = RSA()
    p = getPrime(512)
    q = getPrime(512)
    r.generate(p, q)

    flag, rounds = get_flag(off, 1)

    def enc_msg():
            p = req.recv(4096).strip()
```

```
            req.sendall('%x\n' % r.encrypt(bytes_to_long(p)))

    def dec_msg():
            c = req.recv(4096).strip()
            req.sendall('%x\n' % (r.decrypt(bytes_to_long(c)) & 0xff))

    menu = {
            '1': enc_msg,
            '2': dec_msg,
    }

    req.sendall('enc flag: %x\n' % r.encrypt(bytes_to_long(flag)))
    for _ in xrange(rounds):
            choice = req.recv(2).strip()
            menu[choice]()
```

הכנת האקספלויט

ראשית הכנו איטרטור שיספק offset ו-length מתאימים כך שנעבוד על כל אות של הדגל בנפרד (יש 43 תווים בדגל):

```
LENGTH = 1
for OFFSET in range(43):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('crypto.chal.csaw.io', 1003))
    # send offset + length
    socket.sendall('%d,%d\n' % (OFFSET, LENGTH))
```

נקרא עבור כל חיבור את הגירסה המוצפנת שלו של הדגל (עם הריפוד של המספרים האקראיים):

```
enc_flag = get_enc_flag()
```

הכנו שתי פונקציות מרכזיות לתקשורת עם השרת: get_enc_msg - פונקציה שמקבלת הודעה להצפנה ומחזירה את המספר המוצפן כפי שהוא מוחזר מהשרת, ופונקציית פיענוח get_dec_msg - ששולחת מסר מוצפן לשרת ומחזירה את המסר המפוענח "והלעוס" שהשרת מחזיר. לכן תחילה נשתמש בהן בכדי לגלות את המפתח הציבורי שיעזור לנו למצוא את החסם העליון:

```
def find_n():
    a = get_enc_msg(2)
    a2 = get_enc_msg(4)
    b = get_enc_msg(5)
    b2 = get_enc_msg(25)
    c = get_enc_msg(7)
    c2 = get_enc_msg(49)

    n1 = GCD(a ** 2 - a2, b ** 2 - b2)
    n2 = GCD( a ** 2 - a2, c ** 2 - c2)
    n = GCD(n1, n2)
    return n, a
```

נאתחל משתנים לפני האיטרציה עד לכינוס הבית השמאלי ביותר של החסם העליון U והחסם התחתון L, כאשר אנחנו 'מנחשים' את החסם החזק יותר $U = N/2^{38}$ כי אנחנו יודעים כי N הוא בסדר גודל של 1024 ביטים וכן המסר המוצפן (הריפוד וחתיכת הדגל ביחד) היא בסדר גודל של 123 בתים, כלומר 984 ביטים ולכן יש, ככל הנראה, חסם עליון בסדר גודל של $N/2^{38}$ (כי 38=984-1024).

כמובן שניחוש זה גם משפיע באופן כללי על ההסתברות לפענח נכון, ולכן כדאי להשתמש במפת התפלגויות.

```
magic = 38
enc_flag = get_enc_flag()
n, _2e = find_n()
L = 0
U = n / (1 << magic)
_2ei = pow(_2e, magic, n)
```

ונתחיל באיטרציה עד לכינוס הבית השמאלי ביותר של כל חסם:

```
for i in range(magic, magic + 84):
    _2ei = (_2ei * _2e) % n

    dec = get_dec_msg((enc_flag * _2ei) % n)
    if dec & 1:
     L = (L+U)/2
    else:
     U = (L+U)/2

    if (U & (0xff << 122*8)) >> 122*8 == (L & (0xff << 122*8)) >> 122*8:
     break
```

נבדוק את הבית השמאלי של כל חסם, אם הבתים שווים יש סיכוי טוב שהוא מסמל את התו עליו אנחנו מסתכלים מהדגל. אנחנו אומרים סיכוי טוב משום ש-N היה ניחוש מושכל אשר עליו התבססנו במהלך כל החיבור, ויכול מאוד להיות שטעינו בחישוב שלו. שמנו לב כבר בריצה הלוקאלית שאנחנו טועים לעיתים בחישוב של N. לכן במהלך האתגר יצרנו מילון הסתברויות ועבור כל תו בדגל יצרנו חיבור עם השרת 30 פעם ושמרנו את הבית שנספר הכי הרבה פעמים. למרבה שמחתנו שיטה זו הייתה יעילה מספיק, כך שכשהתקבל הדגל טעינו רק באות אחת. הדגל שהתקבל:

```
flag{LSB_4ppr0xim473_4tt4ck_1s_3v3n_b3tt3r}
```

יש מקום להעיר כי לא היינו בהכרח צריכים לחשב את הדגל בית-בית: אם היינו עושים יותר חזרות היינו מקבלים חסמים הדוקים יותר, והיינו מסוגלים למשל לקבל שניים או שלושב בתים. הדבר היחיד שצריך לשים אליו לב היה האיזון בין כמות האיטרציות שהשרת מוכן לתת - לגודל החתיכה מהדגל שמקבלים.

סיכום

אתגרי הקריפטוגרפיה היו מאתגרים למדי, והיוו סיטואציות שאינן בהכרח רחוקות ממקרים בעולם האמיתי, הרעיון מאחורי המתקפה באתגר הראשון שימש לפריצת PS3 ב-2010[3]. זו הייתה הפעם הראשונה שהשתתפנו ב-CTF פיזית (on-site), השתתפנו כחלק מקבוצת טכנולוגיה בשם 0xaa55, ואין ספק שתודה גדולה מגיעה למארגנים מאוניברסיטת חיפה שבזכותם התחרות התקיימה בארץ. פגשנו המון אנשים מהקהילה, וללא ספק נהננו (ואכלנו לא מעט).

מתן אלפסי: matanalfaso@gmail.com

אלון בן-צור: iamgweej@gmail.com

קישורים

- https://github.com/tna0y/Python-random-module-cracker
- https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm#cite_ref-2
- https://github.com/0xaa55-ctf/ctf-writeups/blob/master/2018_csaw_finals/dsa/client.py
- https://github.com/0xaa55-ctf/ctf-writeups/blob/master/2018_csaw_finals/lostmind/client.py

[3] https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm#cite_ref-2