به‌نام خدا

دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

امنیت داده و شبکه

تمرین ۳: رمزنگاری

مهراد میلانلو

۹۹۱۰۵۷۷۵

۱ کلاسیک

متن رمز شده: «**قنفریفتفنیخحدارزجتثخمظیقتفضبفرذمعسفلاگدثژ**»

در این سوال نکته‌ای که به شکستن رمز Vigenere کمک می‌کند، این است که می‌دانیم طول کلید ۹ است و عبارت ۱۰ حرفی باران‌بهاری در متن آشکار آن وجود دارد. بنابراین حرف اول و آخر این عبارت یعنی «ب» و «ی» با یک حرف (عدد) رمز شده‌اند و به تعداد یکسانی شیفت خورده‌اند. بنابراین، کافیست در متن رمز شده، به دنبال حروفی با تعداد ۹ حرف فاصله میانشان بگردیم که در الفبای فارسی دو عدد فاصله دارند. (دقیقا مانند «ب» و «ی») برای حل این سوال از شماره‌گذاری الفبای فارسی به‌ترتیب استفاده کردم:

الف=۱	ب=۲	پ=۳	ت=۴	ث=۵
ج=۶	چ=۷	ح=۸	خ=۹	د=۱۰
ذ=۱۱	ر=۱۲	ز=۱۳	ژ=۱۴	س=۱۵
ش=۱۶	ص=۱۷	ض=۱۸	ط=۱۹	ظ=۲۰
ع=۲۱	غ=۲۲	ف=۲۳	ق=۲۴	ک=۲۵
گ=۲۶	ل=۲۷	م=۲۸	ن=۲۹	و=۳۰
ه=۳۱	ی=۳۲			

با این روش، با گردش روی عبارت رمز شده، به کاراکتر چهارم یعنی «ر» بر می‌خوریم که ۹ حرف بعد از آن در متن «د» است و در حروف الفبا ۲ عدد فاصله دارند. بنابراین حدس می‌زنیم عبارت بین این دو حرف همان «باران‌بهاری» در متن آشکار باشد. یعنی:

باران‌بهاری → ریفتفنیخح

بنابراین حرف به حرف، اختلاف این دو عبارت را به‌دست می‌اوریم تا به کلید رمزنگاری برسیم. بعد از انجام این کار، به کلید **دهذبگلاحم** می‌رسیم. سپس با این کلید سعی می‌کنیم حروف متن رمز شده را برعکس شیفت بدهیم تا به متن آشکار برسیم. (دقت کنید کلید را طوری قرار می‌دهیم که حرف «د» متناظر با حرف چهارم عبارت رمزشده قرار بگیرد. یعنی کلید **احمدهذبگل** را ابتدای متن رمزشده قرار داده و برعکس شیفت می‌دهیم. (به‌جای جلو رفتن روی حروف الفبا، به عقب برمی‌گردیم). در نهایت متن آشکار به‌دست آمده، چنین است: (فاصله‌ها را برای خوانا شدن اضافه کردم). این اعمال را به‌راحتی می‌توان با کد هم انجام داد اما با توجه به کوتاه بودن متن رمزشده، آن را دستی انجام دادم.

متن آشکار: «**فعل باران بهاری با درخت آید از انفاسشان در نیکبخت**»

۲ درمیان AESها

کدهای این قسمت در پیوست قرار داده‌شده اند.

(آ)

در این قسمت، چون تنها یک مرحله رمزگذاری انجام شده‌است، نیازی به حمله‌ی ملاقات در میانه نیست و با یک حمله‌ی فراگیر ساده می‌توان کلید رمزگذاری را پیدا کرد. دقت کنید که تنها دو بایت ابتدایی کلید تصادفی است و بقیه‌ی بایت‌ها همگی ۰ هستند. بنابراین برای دو بایت

ابتدایی، دو حلقه قرار می‌دهیم که تمام حالات از ۰ تا ۲۵۵ را برای آن‌ها امتحان کنیم. هر کلیدی که متن آشکار داده شده را به متن رمزشده تبدیل کند، به‌عنوان کلید خواسته شده برمی‌گردانیم.

- خروجی و زمان اجرا

```
KEY:  9b3b000000000000000000000000000
Runtime of the program is (in seconds):  0.23363685607910156
```

- پیچیدگی زمانی

$$\mathcal{O}(۲۵۶^۲ \times t_{AES})$$

- پیچیدگی حافظه

$$\mathcal{O}(۱)$$

(ب)

در این قسمت چون در دو مرحله‌ی رمزگذاری و رمزگشایی انجام می‌شود، با انجام حمله‌ی فراگیر نمی‌توانیم کلیدها را پیدا کنیم زیرا به‌ازای هر رمزگذاری با زیرکلید دلخواه k_i، باید رمز به‌دست آمده را با زیرکلید دلخواه k_j رمزگشایی کنیم و پیچیدگی زمانی این کار به‌طور نمایی زیاد می‌شود. بنابراین از حمله‌ی ملاقات در میانه استفاده می‌کنیم. به این صورت که متن آشکار را با تمام کلیدهای ممکن (۲۵۶^۲ حالت) رمزگذاری می‌کنیم و در حافظه نگه می‌داریم. سپس متن رمزشده را با تمام کلیدهای ممکن رمزگذاری می‌کنیم (چون در جهت عکس داریم حرکت می‌کنیم) و هر عبارت به‌دست آمده را در حافظه بررسی می‌کنیم که وجود دارد یا نه. اگر وجود داشته باشد، زیرکلیدهای دلخواهمان را پیدا کرده‌ایم.

- خروجی و زمان اجرا

```
SUB-KEY1:  599300000000000000000000000000
SUB-KEY2:  d760000000000000000000000000000
Runtime of the program is (in seconds):  0.40400075912475586
```

- پیچیدگی زمانی

$$\mathcal{O}(۲۵۶^۲ \times t_{AES} + ۲۵۶^۲(t_{AES} + t_{hach_check}))$$

- پیچیدگی حافظه

$$\mathcal{O}(۲۵۶^۲)$$

(پ)

حمله‌ی این بخش نیز درست مانند بخش قبلی است و از ملاقات در میانه استفاده می‌کنیم. از آن‌جایی که در تابع رمز، ابتدا دوبار متوالی با استفاده از زیرکلید $k_۱$ رمزگشایی و سپس با زیرکلید $k_۱$ رمزگذاری انجام شده‌است، این عملیات را انجام می‌دهیم: متن آشکار را با هر کلید از تمام کلیدهای ممکن دو بار متوالی رمزگشایی می‌کنیم و سپس در حافظه ذخیره می‌کنیم. سپس عبارت رمزشده را با تمام کلیدهای ممکن رمزگشایی کرده و بررسی می‌کنیم در حافظه وجود دارد یا نه. اگر وجود داشته باشد، کلیدهای موردنظر را پیدا کرده‌ایم.

- خروجی و زمان اجرا

```
SUB-KEY1:  ffff00000000000000000000000000
SUB-KEY2:  a3c30000000000000000000000000000
Runtime of the program is (in seconds):  0.5717573165893555
```

- پیچیدگی زمانی

$$\mathcal{O}(۲۵۶^۲ \times ۲t_{AES} + ۲۵۶^۲(t_{AES} + t_{hach_check}))$$

- پیچیدگی حافظه

$$\mathcal{O}(۲۵۶^۲)$$

۳ RSA با OpenSSL

(آ)

کلید خصوصی داده‌شده با کلمه‌ی عبور DNS14022 رمزگذاری شده‌است و افراد بدون داشتن این کلمه‌ی عبور تنها می‌توانند به عبارت رمزشده‌ی کلید دسترسی داشته‌باشند که بی‌فایده است و برای استفاده از آن توسط OpenSSL نیازمند کلمه‌ی عبور هستیم تا رمزگشایی کنیم.

(ب)

با استفاده از دستور زیر می‌توانیم به تمام اجزای سازنده‌ی این کلید خصوصی دسترسی داشته باشیم: (بعد از وارد کردن کلمه‌ی عبور)

```
openssl rsa -in private.pem -text
```

در تصویر زیر، مقادیر عامل‌های اول استفاده شده در مبنای ۱۶ مشاهده می‌شود:

```
prime1:
    00:d5:69:61:2f:59:a4:0a:3a:b8:60:51:3b:36:09:
    3f:94:35:99:c6:e7:86:39:17:d9:25:05:a9:b2:b3:
    6a:fc:b8:9b:ab:78:65:92:a2:bb:9f:93:c0:f0:4e:
    cf:fb:21:a0:07:2b:22:ab:44:be:fe:b0:00:ce:02:
    66:af:7d:28:df:19:52:55:ca:a0:e7:55:44:81:b7:
    9e:31:7b:ec:59:75:1d:5c:21:d5:28:28:51:2a:e8:
    c9:bc:a5:02:62:0a:0e:b0:81:f0:5e:69:d0:c0:d0:
    7d:dc:ec:bf:54:33:d9:42:71:59:a4:16:30:dc:74:
    62:eb:bc:15:83:dd:b9:dc:0a:a6:de:e9:53:98:18:
    d7:63:b1:22:97:5b:d6:dc:f4:8e:27:d8:a1:5a:ef:
    60:f1:71:b1:af:67:28:ad:be:68:4b:9c:fe:2c:e9:
    43:3f:4c:99:e5:73:f1:06:40:6f:a3:0e:16:51:d6:
    e0:74:57:23:91:8e:2d:56:5e:52:23:32:8d:8a:2f:
    1f:21:a6:fc:03:f6:11:63:88:93:3e:91:b4:c2:f7:
    c9:74:42:4b:c9:76:d3:8f:46:b1:22:fa:b2:5e:35:
    98:6b:b9:7a:b3:ac:18:26:c3:d8:b5:a6:f1:3f:63:
    bb:44:91:12:18:69:ce:8e:30:31:99:e8:89:7f:37:
    04:77
prime2:
    00:cc:bf:32:97:39:75:1d:9d:45:00:74:2a:30:a8:
    61:ec:d4:fc:8f:25:2f:b7:4f:f1:f0:bd:62:d4:cd:
    05:98:65:8e:cc:b0:72:77:5b:70:dc:7f:78:7f:4a:
    33:89:7c:7f:26:49:9a:3f:10:1f:8f:dd:77:a1:ea:
    71:4b:b3:63:48:c3:99:47:d8:ce:18:12:a3:f1:ac:
    cd:d3:5c:89:c6:1b:0e:38:8c:03:70:39:21:5f:cb:
    98:1e:1b:c8:cc:f5:a7:4a:57:76:6e:f7:69:41:78:
    4f:f1:5d:85:2f:df:eb:de:2e:a9:e0:fb:ec:c9:07:
    77:35:b3:e4:80:77:21:59:22:52:3d:0d:68:14:32:
    7d:0f:b2:ef:75:77:85:2b:37:ce:cc:8c:3e:5b:18:
    bf:33:75:ef:95:99:41:6a:8e:99:9f:b9:a9:ff:37:
    be:77:81:5e:be:7b:6e:1f:bc:da:86:db:11:00:0c:
    8e:34:9f:9e:88:91:0c:5b:ca:b9:c6:4b:09:86:fc:
    c0:18:a5:cf:d4:8b:c1:72:de:73:3d:b8:67:00:04:
    ee:32:e0:69:18:af:2a:0d:f6:a5:1b:8e:f5:15:3e:
    90:1d:88:52:28:f2:ef:92:6a:ce:85:6d:41:5c:52:
    9e:43:9d:0d:c6:82:bb:3b:16:f7:c1:df:3c:10:ba:
    5f:61
```

از آن‌جایی که می‌دانیم: $\varphi(n) = (p-1) \times (q-1)$، با استفاده از کد زیر، مقدار φ را به‌دست می‌آوریم:

```
p = '00d569612f59a40a3ab860513b36093f943599c6e7863917d92505a9b2b3
6afcb89bab786592a2bb9f93c0f04ecffb21a0072b22ab44befeb000ce0266af7
d28df195255caa0e7554481b79e317bec59751d5c21d52828512ae8c9bca50262
0a0eb081f05e69d0c0d07ddcecbf5433d9427159a41630dc7462ebbc1583ddb9d
c0aa6dee9539818d763b122975bd6dcf48e27d8a15aef60f171b1af6728adbe68
4b9cfe2ce9433f4c99e573f106406fa30e1651d6e0745723918e2d565e5223328
d8a2f1f21a6fc03f6116388933e91b4c2f7c974424bc976d38f46b122fab25e35
986bb97ab3ac1826c3d8b5a6f13f63bb4491121869ce8e303199e8897f370477'

q = '00ccbf329739751d9d4500742a30a861ecd4fc8f252fb74ff1f0bd62d4cd0
598658eccb072775b70dc7f787f4a33897c7f26499a3f101f8fdd77a1ea714bb36
348c39947d8ce1812a3f1accdd35c89c61b0e388c037039215fcb981e1bc8ccf5a
```

```
    74a57766ef76941784ff15d852fdfebde2ea9e0fbecc9077735b3e480772159225
    23d0d6814327d0fb2ef7577852b37cecc8c3e5b18bf3375ef9599416a8e999fb9a
    9ff37be77815ebe7b6e1fbcda86db11000c8e349f9e88910c5bcab9c64b0986fcc
    018a5cfd48bc172de733db8670004ee32e06918af2a0df6a51b8ef5153e901d885
    228f2ef926ace856d415c529e439d0dc682bb3b16f7c1df3c10ba5f61'

    phi = (int(p, 16) - 1) * (int(q, 16) - 1)
    print(hex(phi))
```

که خروجی آن این‌گونه است: (در مبنای ۱۶)

```
    aaaf5d3de3cc447757400eb7589d8b34652d6fbee750d45d70d2f0de2092c15e5
    511f0878ae0a76df95ada07bc04275b7e0469badaea59d792038ce1363e07174a
    048305c177ad185b3477b2028ace3615bd324e2d2ba6cbd7e34e8c61b629b0a64
    ddb174fb8f11d46269ae3aa3fe3110ae9024ecce6dbe83bc18dd5881e9d370f06
    1d4612ae634d09705c73a5722d030c361eae1d56527b1a23fcb5e2bfbe0085678
    da70a63867542e510211cc9b6e59eb706fee843bd12062c8ff54d618cefebf33a
    f584a55afd21c913aadacf3be023b72997a01a0bc4e392a07443326eaf3171d70
    e51dc0ef8514915d262a88a6751000f64303844716d23d497baf4da340e7832db
    7b1a5e8535a2436faa485bd056d32be08167876575f98ddc6b3faa708ab66d8de
    cf6008521f03782a82a8a57bb2ed59f7910d8aa7c1e7daffecd56830c40d1bcb7
    9feb8799e479da2851fd19f585307a595483b07e66ea99ac609b3cb808646462d
    cb2e2da76a38e3507a3100d4fd8e473960127bf088f4b61033d1f5ee7c47fccf1
    d155c78d57e39d496b47abda7d5fb2bf3f698d48c60474ea0ed36c288adeb9de8
    edc2bef8f850a0f274968b50abbe686b6e856dede6c0ea39ed29cecaf7e7fdec1
    50bebb6b1c6f4939a04ab8cfd6930b34ed69d828f41e52ec7bc8305ae39613628
    c94b86a3c4e8ddda1ddeecf221c53b4cd3799edadbd057640
```

(پ)

برای رمزگشایی فایل و چاپ خروجی آن، از دستورات زیر استفاده می‌کنیم:[۱]

```
    openssl rsautl -decrypt -inkey private.pem -in enc.bin > dec.bin
    cat dec.bin
```

بعد از وارد کردن کلمه‌ی عبور در دستور اول، محتوای فایل رمزگشایی شده و سپس می‌توانیم خروجی آن را چاپ کنیم.

- خروجی: `https://xkcd.com/538/`

(ت)

از آن‌جایی که متن آشکار در قسمت‌های قبل با استفاده از کلید عمومی رمزگذاری شده‌است، ابتدا با دستور زیر، از روی کلید خصوصی، کلید عمومی را به‌دست می‌آوریم:

[۱] https://opensource.com/article/۲۱/۲/openssl-decryption-encryption/

```
openssl rsa -in private.pem -pubout > public.pub
```

سپس با دستور زیر، فایل متن آشکار را با استفاده از کلید عمومی رمز می‌کنیم:

```
openssl rsautl -encrypt -inkey public.pub -pubin -in dec.bin -out enc2.bin
```

سپس دو فایل رمزشده را چاپ می‌کنیم:

همان‌طور که مشخص است، خروجی‌ها متفاوت هستند. در الگوریتم RSAای که OpenSSL پیاده‌سازی می‌کند، تعدادی صفر و رشته‌ی تصادفی به‌عنوان padding اضافه می‌شوند. این رشته‌های رندوم با استفاده از تعدادی عملیات در تمام رشته توزیع می‌شوند تا قابل تشخیص نباشند. به همین دلیل، هربار که با استفاده از RSA فایل را رمز کنیم، خروجی متفاوتی می‌گیریم. اما در رمزگشایی این رشته‌های تصادفی بازیابی شده و برداشته می‌شوند تا متن آشکار صحیح به‌دست کاربر برسد.

(ث)

الگوریتم RSA می‌تواند فایلی حداکثر به طول کلید که در این‌جا ۴۰۹۶ است را رمز کند. بنابراین برای فایل‌هایی با طول بیش‌تر از این، نمی‌توان از RSA استفاده کرد. حتی اگر امکان‌پذیر هم بود، به دلیل سربار زیاد الگوریتم‌های نامتقارن، این کار بهینه نبود. بهترین کار استفاده‌ی همزمان از الگوریتم‌های متقارن و نامتقارن است. به این شکل که فایل بزرگ را با استفاده از الگوریتم‌های متقارن رمز می‌کنیم و سپس کلید این رمزگذاری را با استفاده از الگوریتم‌های نامتقارن نگه‌داری و ارسال می‌کنیم.

۴ کلید بدشانس

با توجه به این که n_1 و n_2 عامل اول مشترک دارند، پس کافیست gcd آن‌ها را پیدا کنیم. عامل‌های اول دیگر با تقسیم n_1 و n_2 بر عامل اول مشترک به‌دست می‌آید. بعد از به‌دست آوردن این عوامل اول، به‌راحتی طبق الگوریتم RSA و با داشتن کلیدهای عمومی، کلیدهای خصوصی را پیدا می‌کنیم:

```
from math import gcd

n1 = 882389665577830838482125131852013816279695311
n2 = 726247788835915752041026275800104626981008161
e1 = 65537
e2 = 5

p = gcd(n1, n2)

q1 = n1 // p
q2 = n2 // p
```

```
    phi1 = (p - 1) * (q1 - 1)
    d1 = pow(e1, -1, phi1)

    phi2 = (p - 1) * (q2 - 1)
    d2 = pow(e2, -1, phi2)

    print("PRIVATE KEY 1: ", d1)
    print("PRIVATE KEY 2: ", d2)
```

دقت کنید تابع `pow(e, -1, phi)` در نسخه‌ی `Python 3.8.1` به بعد کار می‌کند و وظیفه‌ی آن پیدا کردن وارون ضربی e در پیمانه‌ی φ است.

- خروجی

```
PRIVATE KEY 1:  824507962534983366448927554628504418529409649
PRIVATE KEY 2:  145249557767183150408191326356348342055363665
```

DH ۵ کوچک

با توجه به این که می‌دانیم مقدار خصوصی یکی از طرفین یعنی X_A یا X_B در اجرای الگوریتم دیفی‌ـهلمن کوچک انتخاب شده‌است، برای حدس زدن آن می‌توانیم از اجرای حمله‌ی فراگیر استفاده کنیم و مقدار آن را پیدا کنیم. در هر مقداری که α^{X_A} یا α^{X_B} برابر یکی از مقادیر داده شده در صورت سوال شود، یعنی مقدار خصوصی را پیدا کرده و مقدار دیگر α^{X_A} یا α^{X_B} را به‌توان مقدار خصوصی یافته شده می‌رسانیم تا کلید مشترک را بیابیم:

```
    q = 28891853952108934833679324067849349771
    a = 3
    aXA = 12782377710547948619020211758683185425
    aXB = 18336445517324902159800604412589181711

    for i in range(1, q):
        Y = pow(a, i, q)
        if Y == aXB:
            print("Shared key: ", aXA ** i % q)
            break
        elif Y == aXA:
            print("Shared key: ", aXB ** i % q)
            break
```

- خروجی

```
Shared key:  216993463219521037996220823391497386611
```

۶ ضد نشت

(آ)

- (u, p)

  - **بررسی صحت رمز عبور:** در این حالت کافیست رمز گرفته شده از کاربر را با p ذخیره شده در پایگاه داده مقایسه کنیم.
  - **امنیت بعد از نشت:** ناامن ترین روش ممکن برای ذخیره‌ی اطلاعات، ذخیره‌ی متن آشکار است. زیرا در صورت نشت، به‌راحتی مهاجم به تمام رمزها دسترسی دارد.

- $(u, h(p))$

  - **بررسی صحت رمز عبور:** در این حالت کافیست رمز گرفته شده از کاربر را با تابع درهم‌ساز h، هش کرده و خروجی را با $h(p)$ ذخیره‌شده در پایگاه داده مقایسه کنیم.
  - **امنیت بعد از نشت:** این روش از ذخیره‌ی متن آشکار رمزها امن‌تر است، اما امنیت کامل را فراهم نمی‌کند. زیرا با وجود rainbow tableها می‌توان به رمز کاربران دسترسی پیدا کرد. در این جدول‌ها، تعداد زیادی از رمزهای معروف و رایج را با توابع درهم‌ساز معروف مانند Sha-256 هش کرده و رمزها را به خروجی درهم آن‌ها نگاشت می‌کنند. بنابراین مهاجم می‌تواند با جست‌وجوی هرکدام از رمزهای ذخیره‌شده در پایگاه داده در این جدول‌ها، به برخی از رمزها و نام‌کاربری‌های کاربران دسترسی داشته باشد.

- $(u, h(h(p)))$

  - **بررسی صحت رمز عبور:** در این حالت کافیست رمز گرفته شده از کاربر را با تابع درهم‌ساز h، دو بار متوالی هش کرده و خروجی را با $h(h(p))$ ذخیره‌شده در پایگاه داده مقایسه کنیم.
  - **امنیت بعد از نشت:** این روش هم مانند روش قبلی امنیت بیش‌تری فراهم می‌کند اما همچنان ممکن است توسط rainbow tableها امنیتشان به‌خطر بیفتد. این روش نسبت به روش قبلی صرفا randomness بیش‌تری دارد.

(ب)

- $(u, h(p))$

  - **بررسی صحت رمز عبور:** در این حالت کافیست رمز گرفته شده از کاربر را با تابع درهم‌ساز h، هش کرده و خروجی را با $h(p)$ ذخیره‌شده در پایگاه داده مقایسه کنیم.
  - **امنیت بعد از نشت:** این روش از ذخیره‌ی متن آشکار رمزها امن‌تر است، اما امنیت کامل را فراهم نمی‌کند. زیرا با وجود rainbow tableها می‌توان به رمز کاربران دسترسی پیدا کرد. در این جدول‌ها، تعداد زیادی از رمزهای معروف و رایج را با توابع درهم‌ساز معروف مانند Sha-256 هش کرده و رمزها را به خروجی درهم آن‌ها نگاشت می‌کنند. بنابراین مهاجم می‌تواند با جست‌وجوی هرکدام از رمزهای ذخیره‌شده در پایگاه داده در این جدول‌ها، به برخی از رمزها و نام‌کاربری‌های کاربران دسترسی داشته باشد.

- $(u, salt, h(salt||p))$

  - **بررسی صحت رمز عبور:** در این حالت باید $salt$ ذخیره شده در ردیف نام کاربری مربوط را از پایگاه داده بازیابی کنیم. سپس رمز گرفته شده از کاربر را با $salt$ بازیابی شده می‌چسبانیم و با استفاده از تابع درهم‌ساز h، هش کرده و خروجی را با $h(salt||p)$ ذخیره‌شده در پایگاه داده مقایسه کنیم.

- **امنیت بعد از نشت:** این روش از روش قبلی امنیت بیش‌تری را تامین می‌کند. زیرا وجود $salt$ تصادفی در کنار رشته،باعث می‌شود رشته‌های درهم‌سازی شده را نتوانیم در rainbow tableها پیدا کنیم. با این حال مهاجم می‌تواند بعد از نشت اطلاعات، رمزهای رایج را در کنار $salt$ به‌دست آورده شده قرار بدهد و حمله‌ی فراگیر را انجام دهد. اما احتمال موفقیت آن بسیار کم‌تر است زیرا به‌ازای هر کاربر باید این حمله را انجام بدهد.

(پ)

- $(u, salt, h(salt||p))$

- **بررسی صحت رمز عبور:** در این حالت باید $salt$ ذخیره شده در ردیف نام کاربری مربوط را از پایگاه داده بازیابی کنیم. سپس رمز گرفته شده از کاربر را با $salt$ بازیابی شده می‌چسبانیم و با استفاده از تابع درهم‌ساز h، هش کرده و خروجی را با $h(salt||p)$ ذخیره‌شده در پایگاه داده مقایسه کنیم.
- **امنیت بعد از نشت:** این روش از روش قبلی امنیت بیش‌تری را تامین می‌کند. زیرا وجود $salt$ تصادفی در کنار رشته،باعث می‌شود رشته‌های درهم‌سازی شده را نتوانیم در rainbow tableها پیدا کنیم. با این حال مهاجم می‌تواند بعد از نشت اطلاعات، رمزهای رایج را در کنار $salt$ به‌دست آورده شده قرار بدهد و حمله‌ی فراگیر را انجام دهد. اما احتمال موفقیت آن بسیار کم‌تر است زیرا به‌ازای هر کاربر باید این حمله را انجام بدهد.

- $(u, salt, E_{salt}(p))$

- **بررسی صحت رمز عبور:** در این حالت باید $salt$ ذخیره شده در ردیف نام کاربری مربوط را از پایگاه داده بازیابی کنیم. سپس رمز گرفته شده از کاربر را با الگوریتم E و کلید $salt$ رمزگذاری کرده و خروجی را با $E_{salt}(p)$ ذخیره‌شده در پایگاه داده مقایسه کنیم.
- **امنیت بعد از نشت:** این روش امن نیست! زیرا بعد از نشت اطلاعات، مهاجم با داشتن کلید $salt$ می‌تواند $E_{salt}(p)$ را رمزگشایی کند و عملا تفاوتی با نگه‌داری متن آشکار در پایگاه داده ندارد.

(ت)

- $(u, E_k(salt), h(salt) \oplus p)$

- **بررسی صحت رمز عبور:** در این حالت با استفاده از کلید خصوصی سرور یعنی k، $E_k(salt)$ را رمزگشایی کرده و به $salt$ می‌رسیم. سپس هش آن را با استفاده از تابع درهم‌ساز h محاسبه می‌کنیم و با رمز واردشده توسط کاربر یعتی p `xor` کرده و خروجی را با $h(salt) \oplus p$ ذخیره‌شده در پایگاه داده مقایسه کنیم.
- **امنیت بعد از نشت:** این روش امن نیست! زیرا اگر نشت کامل اطلاعات را متصور شویم، مهاجم می‌تواند به کلید خصوصی سرور نیز دسترسی داشته باشد. پس به‌راحتی می‌تواند $E_k(salt)$ را رمزگشایی کرده و به $salt$ دست بیابد. سپس آن را هش می‌کند و $h(salt)$ را به‌دست می‌آورد. آن را با $h(salt) \oplus p$ ذخیره‌شده در پایگاه‌داده `xor` می‌کند و می‌تواند به p کاربر دلخواهش دسترسی داشته باشد.

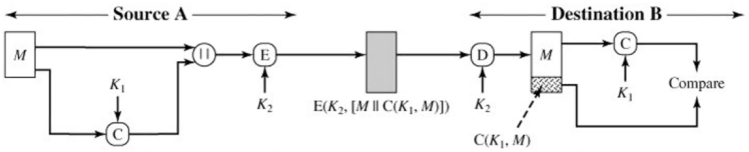
- $(u, h(salt), E_k(salt) \oplus p)$

- **بررسی صحت رمز عبور:** در این حالت رمز واردشده توسط کاربر را با $E_k(salt) \oplus p$ xor کرده و به $E_k(salt)$ می‌رسیم. بعد از آن، با کلید خصوصی سرور آن را رمزگشایی کرده و به $salt$ می‌رسیم. در نهایت آن را با استفاده از تابع h هش کرده و خروجی را با $h(salt)$ ذخیره‌شده در پایگاه داده مقایسه کنیم.
- **امنیت بعد از نشت:** این روش امن است. حتی بعد از نشت اطلاعات مهاجم نمی‌تواند به هیچ جزئی از اطلاعات محرمانه‌ی کاربر دسترسی داشته باشد.

OFB-ENC+CBC-MAC ۷

(آ)

در کد enc_mac داده‌شده در صورت سوال، معماری شبیه به مورد زیر استفاده شده‌است که برای احراز صحت پیام و محرمانگی است:



بنابراین برای رمزگشایی و بررسی اعتبار کد اصالت‌سنجی، ابتدا پیام رمزشده را با همان مد استفاده‌شده در رمزگذاری، رمزگشایی می‌کنیم

```
m = AES.new(mode=AES.MODE_OFB, key=k, iv=k).decrypt(c)
```

سپس با همان مد به‌دست آوردن کد اصالت‌سنجی، سعی می‌کنیم کد را از روی متن آشکار به‌دست آمده بسازیم و بررسی کنیم که با کد ارسال شده یکسان است یا خیر. اگر این دو کد برابر نباشند، یعنی پیام ارسال شده معتبر نیست.

```
if AES.new(mode=AES.MODE_CBC, key=k, iv=c[:BLOCK_SIZE]).encrypt(m)[-BLOCK_SIZE:] == t:
```

اگر پیام ارسال‌شده معتبر باشد، دنباله‌زنی قراردادشده در انتهای آخرین بلوک را برداشته و پیام را به‌عنوان پیام معتبر چاپ می‌کنیم:

```
from Crypto.Cipher import AES
BLOCK_SIZE = 128 // 8

k = '875faffbaeea63eb878613b98460f4d2'
c1, t1 = 'd8b8239628a3f44c81e50cbd57aaac62586cdf1376c25fa8c23e8becf6be4688',
'abb859c60dd1450bd789a40bc3638f4e'
c2, t2 = 'dfb3319a23e6bf4d88b20cf342a9ac62447cc04770dd2cd2bc5b87e0fab24a84',
'b893a8d5032f5c004f11543626fc942e'

def check_mac(k, c, t):
    m = AES.new(mode=AES.MODE_OFB, key=k, iv=k).decrypt(c)
    if AES.new(mode=AES.MODE_CBC, key=k, iv=c[:BLOCK_SIZE]).encrypt(m)[-BLOCK_SIZE:] == t:
        pad_size = m[-1]
        if pad_size < 1 or pad_size > BLOCK_SIZE:
```

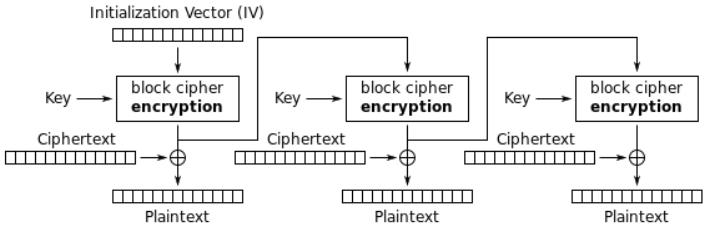
```
                return m, False
            for i in range(1, pad_size + 1):
                if m[-i] != pad_size:
                    return m, False
            m = m[:-pad_size]
            return m, True
        return m, False

    m1, authenticity1 = check_mac(bytes.fromhex(k), bytes.fromhex(c1), bytes.fromhex(t1))
    m2, authenticity2 = check_mac(bytes.fromhex(k), bytes.fromhex(c2), bytes.fromhex(t2))

    if authenticity1:
        print("Message 1: ", m1.decode())
    else:
        print("Message 1: ", "INVALID")

    if authenticity2:
        print("Message 2: ", m2.decode())
    else:
        print("Message 2: ", "INVALID - ", m2.decode())
```

- خروجی

```
Message 1:  enemy knows the system
Message 2:  INVALID - beware of snake oil crypto
```

(ب)

در این قسمت برای پیدا کردن عبارت رمزشده‌ای که متن آشکار آن متن دلخواه `attacker to 99$` باشد، از آسیب‌پذیری OFB استفاده می‌کنیم. به ساختار رمزگشایی این مد نگاه کنید:



Output Feedback (OFB) mode decryption

برای متن آشکار هر بلوک، خروجی ثابتی که از رمز کردن IV نتیجه می‌شود، با متن رمز شده xor شده و متن آشکار را خروجی می‌دهد. بنابراین باید متن رمز را طوری بنویسیم که:

$$fake\_cipher \bigoplus fake\_message = org\_cipher \bigoplus org\_message$$

$$\Rightarrow fake\_cipher = fake\_message \bigoplus org\_cipher \bigoplus org\_message$$

دقت کنید که متن تقلبی دلخواه ما کمتر از ۱۶ بایت که طول یک بلوک است، می باشد. بنابراین تنها کافیست ۱۶بایت اول متن اصلی و رمز اصلی را در نظر بگیریم.

```
from Crypto.Cipher import AES
BLOCK_SIZE = 128 // 8

def xor_strings(xs, ys):
    return bytes(x ^ y for x, y in zip(bytes.fromhex(xs), bytes.fromhex(ys)))

def pad(m):
    r = BLOCK_SIZE - len(m) % BLOCK_SIZE
    pad_size = r if r != 0 else BLOCK_SIZE
    m += pad_size.to_bytes(1, 'big') * pad_size
    return m

c = "ad7fa3468caf0b5c01ec7be9b583fa350d2ce39b8cd57ee26270235cd6598592"
t = "905f6d5d03e5269a52aa3e33b558e764"
org_p_text = '1$ to original_destination'
fake_p_text = '99$ to attacker'
padded_fake_p_text = pad(fake_p_text.encode())
truncated_org_p_text = org_p_text[:BLOCK_SIZE]

c_prime = xor_strings(xor_strings(c, padded_fake_p_text.hex()).hex(),
truncated_org_p_text.encode().hex())
print("c_prime: ", c_prime.hex())
```

که در این‌جا خروجی زیر را به‌عنوان متن رمزشده‌ی تقلبی داریم:

```
c_prime:  a562a71297e0444f1cff73e4bf8ad750
```

برای آزمودن این مورد هم از کد زیر استفاده می‌کنیم که با یک کلید رندوم، رمزگذاری و استفاده از متن رمزشده‌ی جعلی را انجام می‌دهد:

```
# Test
random_key = '875faffbaeea63eb878613b98460f4d2'
c, t = enc_mac(bytes.fromhex(random_key), org_p_text.encode())
c_prime = xor_strings(xor_strings(c.hex(), padded_fake_p_text.hex()).hex(),
truncated_org_p_text.encode().hex())

message = AES.new(mode=AES.MODE_OFB, key=bytes.fromhex(random_key),
iv=bytes.fromhex(random_key)).decrypt(c_prime)
print("Message: ", message.decode())
```

خروجی نیز مطابق انتظار است:

```
    Message:  99$ to attacker
```