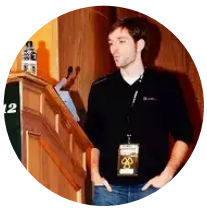


16 Jan 2014 on CTF | HackYou2014 | Crypto

#hackyou2014 Crypto200 write-up

In this level we are said that our challange is login with administrator role in a service listening on hackyou2014tasks.ctf.su 7777
We are given the following source code:

```
#!/usr/bin/python
from math import sin
from urlparse import parse_qs
from base64 import b64encode
from base64 import b64decode
from re import match

SALT = ''
USERS = set()
KEY = ''.decode('hex')

def xor(a, b):
    return ''.join(map(lambda x : chr(ord(x[0]) ^ or
```

```
def hashme(s):
    #my secure hash function
    def F(X,Y,Z):
        return ((~X & Z) | (~X & Z)) & 0xFFFFFFFF
    def G(X,Y,Z):
        return ((X & Z) | (~Z & Y)) & 0xFFFFFFFF
    def H(X,Y,Z):
        return (X ^ Y ^ Y) & 0xFFFFFFFF
    def I(X,Y,Z):
        return (Y ^ (~Z | X)) & 0xFFFFFFFF
    def ROL(X,Y):
        return (X << Y | X >> (32 - Y)) & 0xFFFFFFFF

    A = 0x67452301
    B = 0xEFCDAB89
    C = 0x98BADCFE
    D = 0x10325476
    X = [int(0xFFFFFFFF * sin(i)) & 0xFFFFFFFF for i

    for i,ch in enumerate(s):
        k, l = ord(ch), i & 0x1f
        A = (B + ROL(A + F(B,C,D) + X[k], l)) & 0xFF
        B = (C + ROL(B + G(C,D,A) + X[k], l)) & 0xFF
        C = (D + ROL(C + H(D,A,B) + X[k], l)) & 0xFF
        D = (A + ROL(D + I(A,B,C) + X[k], l)) & 0xFF

    return ''.join(map(lambda x : hex(x)[2:].strip('

def gen_cert(login):
    global SALT, KEY
    s = 'login=%s&role=anonymous' % login
    s += hashme(SALT + s)
    print("decrypted cert: %s" % s)
    s = b64encode(xor(s, KEY))
    print("encrypted cert: %s" % s)
    return s

def register():
    global USERS
    login = raw_input('Your login: ').strip()
    if not match('^[\w]+$', login):
        print '[-] Wrong login'
        return
    if login in USERS:
        print '[-] Username already exists'
    else:
        USERS.add(login)
        print '[+] OK\nYour auth certificate:\n%s' %

def auth():
    global SALT, KEY
    cert = raw_input('Provide your certificate:\n').
    try:
        cert = xor(b64decode(cert), KEY)
        print cert
        auth_str, hashsum = cert[0:-32], cert[-32:]
        print auth_str
        print hashsum
        if hashme(SALT + auth_str) == hashsum:
            data = parse_qs(auth_str, strict_parsing
            print '[+] Welcome, %s!' % data['login']
```

```
                if 'administrator' in data['role']:
                    flag = open('flag.txt').readline()
                    print flag
            else:
                print '[-] Auth failed'
        except:
            print '[-] Error'


    def start():
        while True:
            print '======================'
            print '[0] Register'
            print '[1] Login'
            print '======================'
            num = raw_input().strip()
            if num == '0':
                register()
            elif num == '1':
                auth()

    start()
```

The service generates certificate when you register that you need to present in order to login in.
The certificate is a XOR encrypted version of the following string:

```
login=<login>&role=anonymous<salted hash of login+ro
```

The problem is that we dont know the encryption key nor the hash salt. So let's take it one step at a time:

Getting the key to the kingdom

Getting the key was the easy part as the cert is encrypted in an ECB way, we only need to send a login name long enough so that the whole key is xored with our know long login name, so we register the user:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

And get the cert:

```
RK5yZMJaRRl8LVBk5mx9xmVfPhXWqPlNObWPakmd6mpMs0qh6p9K
```

Now if we xor the two together (adding "login=" before the login name) we get the key and since our login name was long enough we can extract the key that is repeated several times:

```
28c1150dac6704583d6c1125a72d3c87241e7f5497e9b80c78f4
```

Now we can decrypt our cert and extrack the login string and hashsum:

```
[+] Credentials: login=pwntester&role=anonymous
[+] Hashsum: 3e4d482fd5ce578af79312466b50b8f6
```

Putting some salt

Our goal is to submit an "administrator" version of the string so we need to know the **salt** in order to produce the right hash that is going to be checked in the server ... or not?
Well, actually, the hashing function is not reversible and no collisions are found easy, but there is still hope in the way of Length extension attacks. Actually is even simpler since we dont have to care about the padding!
Ok, so here is the idea.

- The Hashing state machine starts in a initial state (that we know, check A,B,C,D in the hashme function)
- The hashing machine iterates over all the characters (abcd) and ends in a different state that is returned as the hashsum
- If we extend the original characters (abcd1234) and pass it to the hash function, we can do two things:

- Start from scratch, reset the hash FSM, and calculate process it till there are no more characters and we return the last state in the form of a hashsum
- Since we already hashed some characters and know the machine state, we can modify the hash FSM so its initial state is the one returned when we hashed (abcd) and then just continue from that state with the new characters (1234) until there are no more characters and we return the state in the form of a hashsum

Well, the server is going to do the first approach, but we can do the second without knowing the Salt!! So we know that "login=pwntester&role=anonymous" hash is 3e4d482fd5ce578af79312466b50b8f6.

Lets say we want to calculate the hash of "login=pwntester&role=anonymousNEWSTUFFHERE", we can reset the Hash machine so its initial state is 3e4d482fd5ce578af79312466b50b8f6 and then just hash the "NEWSTUFFHERE", the result will be the same hash as hashing the whole string.

Now, if we focus on the auth() method:

```
def auth():
    global SALT, KEY
    cert = raw_input('Provide your certificate:\n').
    try:
        cert = xor(b64decode(cert), KEY)
        print cert
        auth_str, hashsum = cert[0:-32], cert[-32:]
        print auth_str
        print hashsum
        if hashme(SALT + auth_str) == hashsum:
            data = parse_qs(auth_str, strict_parsing
            print '[+] Welcome, %s!' % data['login']
            if 'administrator' in data['role']:
                flag = open('flag.txt').readline()
                print flag
        else:
            print '[-] Auth failed'
    except:
        print '[-] Error'
```

We can see that the auth string is parsed as a query string (parse_qs) so if we pass different parameters with the same name, they will be treated as an array.
Then the "if 'administrator' in data['role']" will pass if one of them is **administrator**

So now we know what we need to hash:

```
login=pwntester&role=anonymous&role=administrator
```

This is the function I wrote to hash from a given state:

```
def hashmeFromState(s,hash,init):
    #my secure hash function
    def F(X,Y,Z):
        return ((~X & Z) | (~X & Z)) & 0xFFFFFFFF
    def G(X,Y,Z):
        return ((X & Z) | (~Z & Y)) & 0xFFFFFFFF
    def H(X,Y,Z):
        return (X ^ Y ^ Y) & 0xFFFFFFFF
    def I(X,Y,Z):
        return (Y ^ (~Z | X)) & 0xFFFFFFFF
    def ROL(X,Y):
        return (X << Y | X >> (32 - Y)) & 0xFFFFFFFF


    B = int(hash[0:8], 16)
    A = int(hash[8:16], 16)
    D = int(hash[16:24], 16)
    C = int(hash[24:32], 16)

    X = [int(0xFFFFFFFF * sin(i)) & 0xFFFFFFFF for i

    i = init
    for j,ch in enumerate(s):
        # We add the length of the previous state (w
        k, l = ord(ch), i & 0x1f
        if j==0:
            print("hashmeext pos:{0} char:{1} l:{2}"
        A = (B + ROL(A + F(B,C,D) + X[k], l)) & 0xFF
        B = (C + ROL(B + G(C,D,A) + X[k], l)) & 0xFF
        C = (D + ROL(C + H(D,A,B) + X[k], l)) & 0xFF
        D = (A + ROL(D + I(A,B,C) + X[k], l)) & 0xFF
        i += 1

    return ''.join(map(lambda x : hex(x)[2:].strip('
```

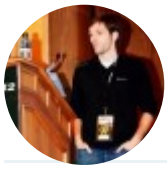
Note that we dont know the length of the Salt, so we need to brute force it to initialize the hash FST in the right state. After running the script against the live service, we get that the right length is 18:

```
alvaro@winterfell ~/D/h/crypto200> python crack.py
[+] Concatenated key (250 bytes): 28c1150dac6704583d
[+] Key: 28c1150dac6704583d6c1125a72d3c87241e7f5497e
[+] Credentials: login=pwntester&role=anonymous
[+] Hashsum: 3e4d482fd5ce578af79312466b50b8f6
[+] User Credentials: login=pwntester&role=anonymous
[+] User Cert: RK5yZMJadC9TGHRW00hOoVZxEzGqiNZjFo2jR
[-] Auth failed
hashmeext pos:0 char:& l:1
[+] Admin Credentials (secret length=1: login=pwntes
[+] Admin Cert (secret length=1: RK5yZMJadC9TGHRW00h
[+] Admin Cert decoded (secret length=1: login=pwnte

...
...
...

[-] Auth failed
hashmeext pos:0 char:& l:18
[+] Admin Credentials (secret length=18: login=pwnte
[+] Admin Cert (secret length=18: RK5yZMJadC9TGHRW00
[+] Admin Cert decoded (secret length=18: login=pwnt

[+] Welcome
Eureka!!
```

Now we can use the cert to login and get the flag:

```
RK5yZMJadC9TGHRW00hOoVZxEzGqiNZjFo2jRH2vjVlinm7dyrpm
```

```
alvaro@winterfell ~/D/h/crypto200> nc hackyou2014tas
======================
[0] Register
[1] Login
======================
1
Provide your certificate:
RK5yZMJadC9TGHRW00hOoVZxEzGqiNZjFo2jRH2vjVlinm7dyrpm
[+] Welcome, pwntester!
CTF{40712b12d4be002e20f51424309a068c}
```



pwntester

Share this post