Australian Government

Cyber Security Challenge Australia 2014
www.cyberchallenge.com.au

CySCA2014 Reverse Engineering Writeup

Background: Fortcerts has been contracted by Terribad Corp. to analyse a number of binaries. Fortcerts doesn't have anyone experienced in reverse-engineering so they have asked you to do it for them.

Reverse Engineering 1 - U JAD BRO?

Question: Staff from Terribad Corp have forgotten the password for their proprietary data protection Java application. They need you to retrieve the data stored in the application and submit it as the flag.

Designed Solution: Players use JAD to decompile the supplied Java code. They then use a simple xor operation with a hardcoded string to recover the poorly protected data from the PRAuthService class.

Write Up:
In order to decompile the java program, we will use the JAD decompiler.

Firstly, we extract the contents of the Java program

```
#> jar -xf pr.jar
```

After file extraction, the contents of the jar file are stored in the com directory. We run jad over the directory to decompile all of the class files.

```
#> ./jad -o -r -sjava -dsrc 'com/**/*.class'
```

We will now analyse the decompiled files in the src directory.

We find the user authentication code in PRAuthService.java, the stored password is a SHA-256 hash and the code doesn't seem to have any obvious flaws.

We decide to target the stored data directly. We find this stored data in PRContentService.java.

We can see the content is stored as a byte array that is XOR'ed with the username (hardcoded as 'Hero33' in the PRAuthService class) and is de-XORed before being presented to the user.

We reuse the XOR code in the PRContentService.java to decode the stored data.

```
public class Solution {
    private static final byte CIPHERED_BYTES[] = {
        30, 0, 21, 8, 90, 86, 4, 0, 10, 6,
        80, 92, 38, 53, 30, 26, 80, 88, 113, 87,
        67
    };
    public static String cipherString(String text, String key) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < text.length(); i++) {
            sb.append((char)(text.charAt(i) ^ key.charAt(i % key.length())));
        }
        return sb.toString();
    }

    public static void main(String[] args) {
        String result;
        result = cipherString(new String(CIPHERED_BYTES), "Hero33");

        System.out.println(result);
    }
}
```

We execute this code and get the de-xored data (which is the flag for this question)
VeggieLexiconPluck921

Reverse Engineering 2 - Knock Knock

Question: Terribad Corp has provided a binary which they think is the "next big thing" in security. They would like to get it certified as a secure product. We need you to reverse engineer the algorithm to understand what it does. Once you have done this, a test server is running at 172.16.1.20:3422 to allow you to prove you completely recovered the algorithm.

Designed Solution: Players need to reverse engineer the provided binary to determine that it is a port knocking application. Players then need to determine the ports used by the application and the challenge algorithm used when generating the challenge. They then create a simple client that will emulate the program and receive the flag.

Write Up:
In our study of this binary we will use IDA to perform perform a static analysis of the provided binary.

We start by setting up the the environment to run the provided binary on our Kali workstation.

```
#> mkdir /chroots/2011
#> mkdir -p /chroots/2011/dev
#> mknod /chroots/2011/dev/urandom c 1 9
#> cd /chroots/2011
#> ./488f866ad090d0843657f322e516168a-re02
Currently running as user 0 and group 0
Moving into chroot jail for user 2011. Path = '/chroots/2011'
Changing group from 0 to 1012
Changing user from 0 to 2011
Forking....
Child process 22216 spawned. Original process quitting
Running portknockd
```

With the binary running we can try a connection using netcat in another terminal.

Terminal 2:

```
#> nc localhost 3422
F
<enter>
```

Original terminal running supplied binary:

```
portknockd: New client connected
portknockd: New Client. waiting for knocks
```

The binary output in our original terminal leads us to think that we're dealing with a port knocking server. Generally this means we can trigger some behaviour from the server by accessing a list of ports in the correct order.

Apart from the “portknockd” banner in the binary terminal we have received “F” in netcat and a failure message. Connecting multiple times reveals that the “F” changes each time to a seemingly random value.

Lets open the binary in IDA and look at the behaviour between the welcome banner and failure message. There must be something we need to do between these events.

In IDA, when looking at the strings listing we can see the string “portknockd: New Client. Waiting for knocks” written out with _puts at 0x8049207 (you can jump to this address in IDA by pressing ‘g’ and entering the address in hex).

At 0x8049239 the server calls a function, following the call reveals a function that generates a random number by opening “`/dev/urandom`”. The random number is written to the socket by _write at 0x8049256. This explains the random character we’re seeing between the server banner and failure message.

At 0x80492A3 the server reads 4 bytes from a socket. Ok, so we need to send something to the server. As an aside, note that IDA adds names for library functions it can identify (e.g. static libc calls). Consider the following assembly leading up to the _read call:

```
mov [ebp+userInput], 0
mov dword ptr [esp+8], 4   ; nbytes
lea eax, [ebp+userInput]
mov [esp+4], eax           ; buf
mov eax, [ebp+fd]
mov [esp], eax             ; fd
call _read
```

IDA recognizes the “`mov [esp+x]`” instructions leading up to “`call _read`” as arguments being pushed onto the stack. Because the calling convention of _read by is known by IDA it annotates the argument names with a comment. We check the arguments of read by referring to its man page.

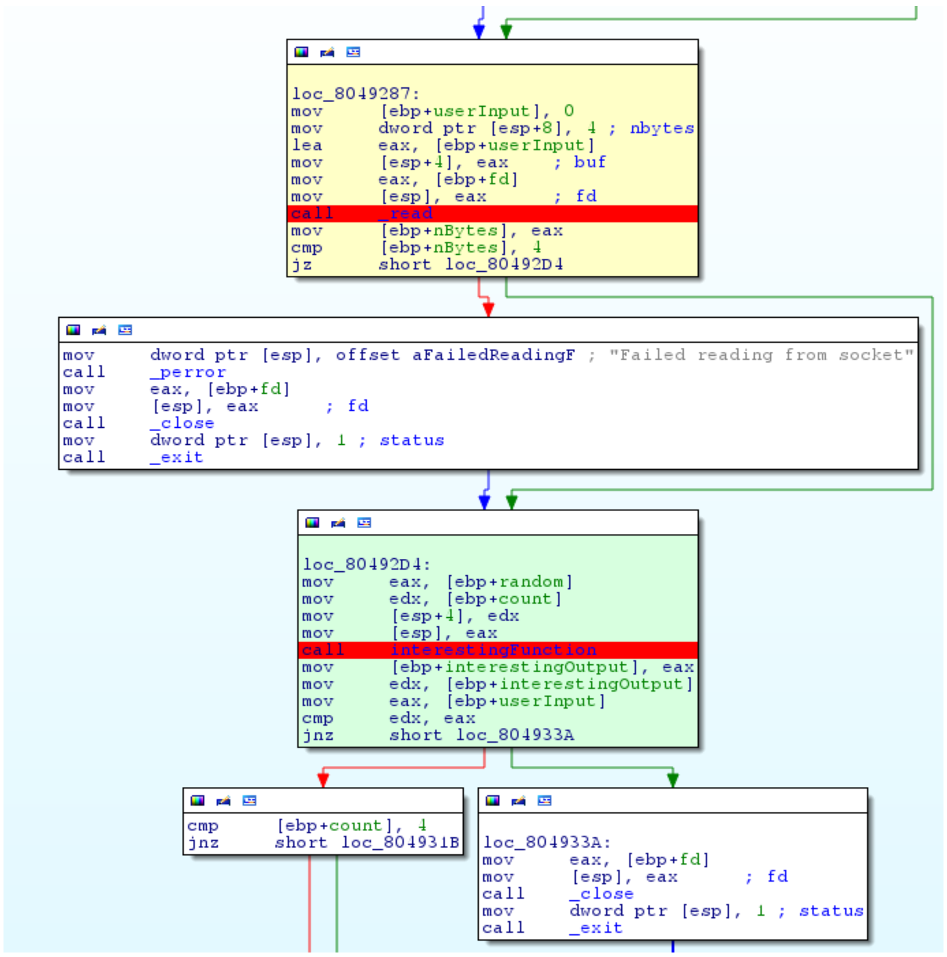
```
#> man read
<snip>
SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read()  attempts to read up to count bytes from file descriptor fd into the
       buffer starting at buf.
```

Funtip: Checking the man page is a valuable exercise for any function you don’t understand.

Further down at 0x80492E1, there is a function call that takes the random value generated from urandom and an unidentified variable as arguments.

To try determine the purpose of the unknown variable we start looking back for references to the unknown variable, it's first set to 0 at 0x804920C, compared with 4 at 0x8049351 and incremented at 0x8049334. The comparison at 0x80492F3 is followed by a "`jbe`", suggesting a loop that iterates 5 times. We can assume that our unknown variable is the loop counter "count".

Within the loop body, the server reads a value, calls a function (we'll call it "interestingFunction" for now) and compares the return value with 4 bytes sent by the client.



```
loc_8049287:
mov     [ebp+userInput], 0
mov     dword ptr [esp+8], 4 ; nbytes
lea     eax, [ebp+userInput]
mov     [esp+4], eax    ; buf
mov     eax, [ebp+fd]
mov     [esp], eax      ; fd
call    _read
mov     [ebp+nBytes], eax
cmp     [ebp+nBytes], 4
jz      short loc_80492D4
```

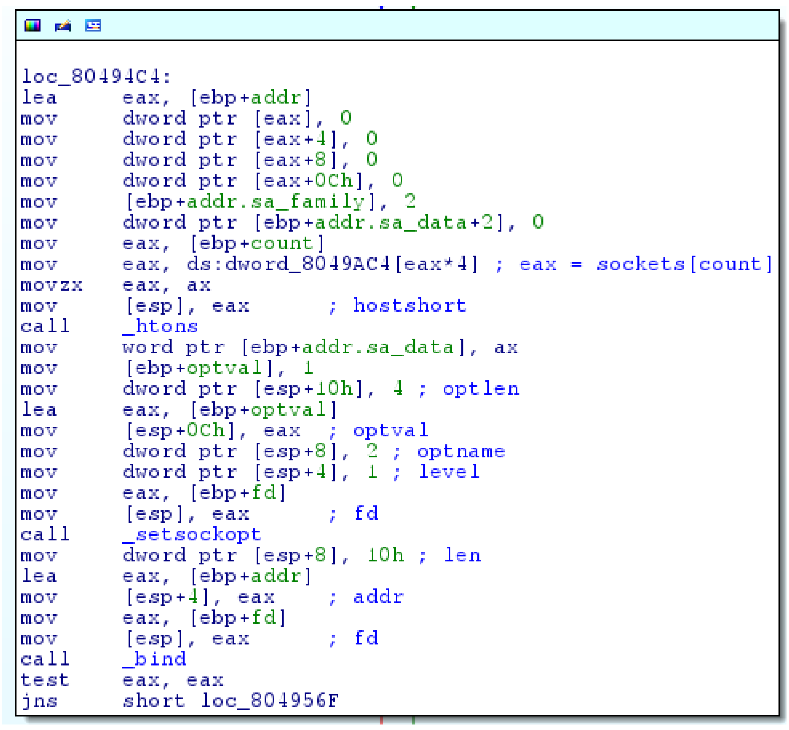
```
mov     dword ptr [esp], offset aFailedReadingF ; "Failed reading from socket"
call    _perror
mov     eax, [ebp+fd]
mov     [esp], eax      ; fd
call    _close
mov     dword ptr [esp], 1 ; status
call    _exit
```

```
loc_80492D4:
mov     eax, [ebp+random]
mov     edx, [ebp+count]
mov     [esp+4], edx
mov     [esp], eax
call    interestingFunction
mov     [ebp+interestingOutput], eax
mov     edx, [ebp+interestingOutput]
mov     eax, [ebp+userInput]
cmp     edx, eax
jnz     short loc_804933A
```

```
cmp     [ebp+count], 4
jnz     short loc_804931B
```

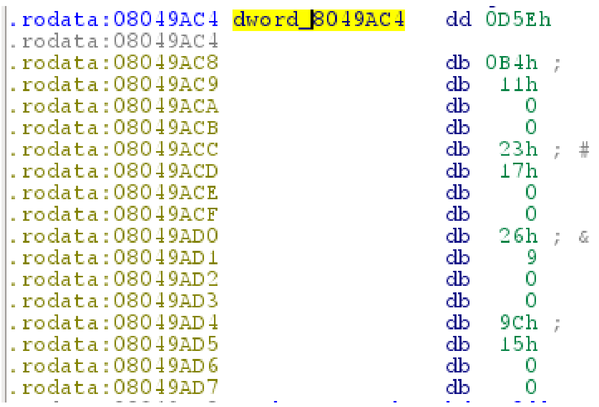
```
loc_804933A:
mov     eax, [ebp+fd]
mov     [esp], eax      ; fd
call    _close
mov     dword ptr [esp], 1 ; status
call    _exit
```

If all of the values supplied by the client match the interestingFunction output, the server calls a function at 0x80492FF to print the flag (we determine this by following the function to the "`/flag.txt`" reference. If the loop isn't finished (i.e we haven't received 5 client inputs) , the server closes the current socket (0x8049321) and calls a function at 0x804932C. If we look at the function at 0x804932C we see that it is responsible for setting up a socket for the next port in the knock list.



```
loc_80494C4:
lea     eax, [ebp+addr]
mov     dword ptr [eax], 0
mov     dword ptr [eax+4], 0
mov     dword ptr [eax+8], 0
mov     dword ptr [eax+0Ch], 0
mov     [ebp+addr.sa_family], 2
mov     dword ptr [ebp+addr.sa_data+2], 0
mov     eax, [ebp+count]
mov     eax, ds:dword_8049AC4[eax*4] ; eax = sockets[count]
movzx   eax, ax
mov     [esp], eax      ; hostshort
call    _htons
mov     word ptr [ebp+addr.sa_data], ax
mov     [ebp+optval], 1
mov     dword ptr [esp+10h], 4 ; optlen
lea     eax, [ebp+optval]
mov     [esp+0Ch], eax  ; optval
mov     dword ptr [esp+8], 2 ; optname
mov     dword ptr [esp+4], 1 ; level
mov     eax, [ebp+fd]
mov     [esp], eax      ; fd
call    _setsockopt
mov     dword ptr [esp+8], 10h ; len
lea     eax, [ebp+addr]
mov     [esp+4], eax    ; addr
mov     eax, [ebp+fd]
mov     [esp], eax      ; fd
call    _bind
test    eax, eax
jns     short loc_804956F
```

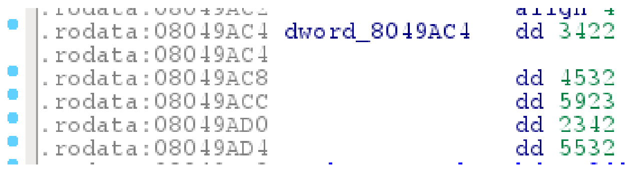
By renaming the stack arguments to this function we see that the loop counter has been passed as an argument. The counter is used at 0x80494EF as the offset into a data structure in .rodata. Jumping to 0x8049AC4 we see the following data.



```
.rodata:08049AC4 dword_8049AC4   dd 0D5Eh
.rodata:08049AC4
.rodata:08049AC8                 db 0B4h ;
.rodata:08049AC9                 db  11h
.rodata:08049ACA                 db    0
.rodata:08049ACB                 db    0
.rodata:08049ACC                 db  23h ; #
.rodata:08049ACD                 db  17h
.rodata:08049ACE                 db    0
.rodata:08049ACF                 db    0
.rodata:08049AD0                 db  26h ; &
.rodata:08049AD1                 db    9
.rodata:08049AD2                 db    0
.rodata:08049AD3                 db    0
.rodata:08049AD4                 db  9Ch ;
.rodata:08049AD5                 db  15h
.rodata:08049AD6                 db    0
.rodata:08049AD7                 db    0
```

Note the blue “db/dd” identifiers, dd meaning “double-word”, db meaning “byte”. The mov instruction at 0x80494F2 referred to double-words so lets try changing the type of this data. Pressing ‘d’ toggles between byte, word and double-word types. We can change the base of these values by selecting a value and pressing ‘h’ in IDA. Toggle the values to decimal double-words to reveal the port numbers. We now have an ordered list of the ports to knock.

We can see the port we’re already connecting to, 3422, is the first entry in the list.

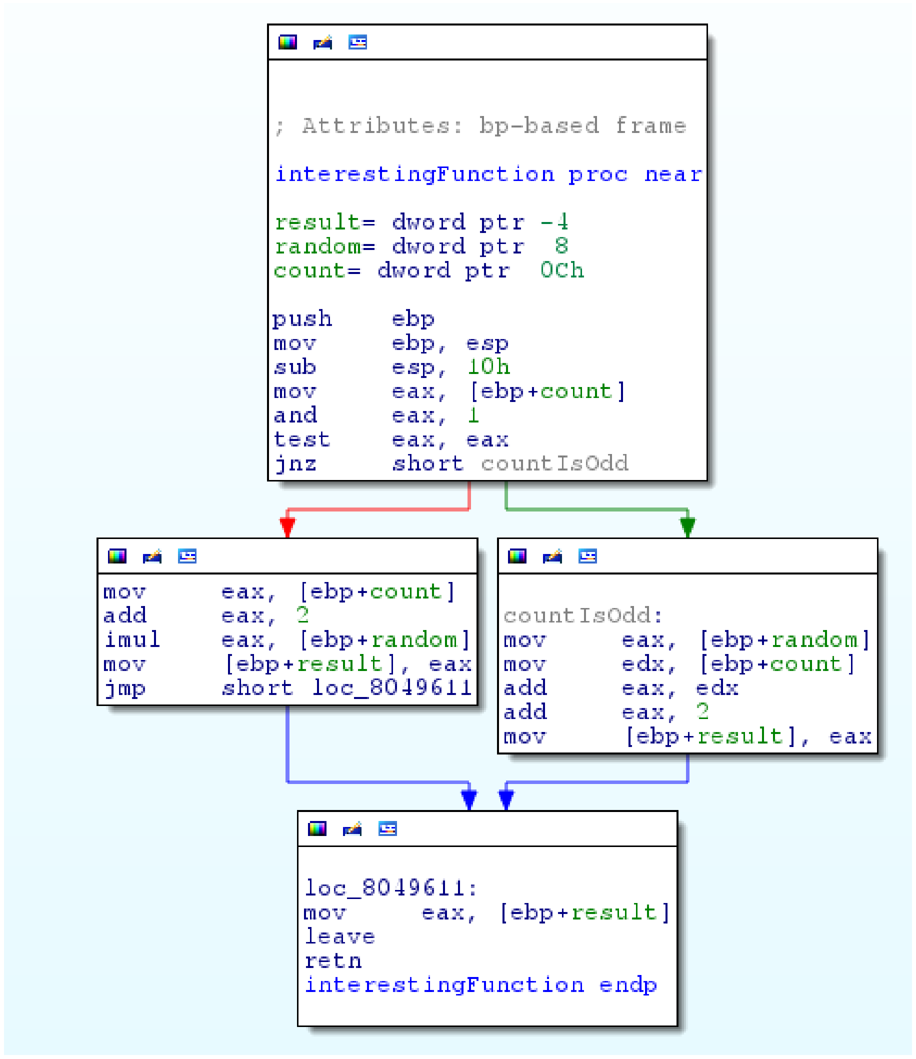


```
.rodata:08049AC4 dword_8049AC4   dd 3422
.rodata:08049AC4
.rodata:08049AC8                 dd 4532
.rodata:08049ACC                 dd 5923
.rodata:08049AD0                 dd 2342
.rodata:08049AD4                 dd 5532
```

We'll need to connect to each of these ports in order to print the flag.

So at this point, we've discovered a challenge/response mechanism and a list of ports to connect to.

Let's go back to "interestingFunction" and reverse the challenge/response checking algorithm.



```
; Attributes: bp-based frame

interestingFunction proc near

result= dword ptr -4
random= dword ptr  8
count= dword ptr  0Ch

push    ebp
mov     ebp, esp
sub     esp, 10h
mov     eax, [ebp+count]
and     eax, 1
test    eax, eax
jnz     short countIsOdd
```

```
mov     eax, [ebp+count]
add     eax, 2
imul    eax, [ebp+random]
mov     [ebp+result], eax
jmp     short loc_8049611
```

```
countIsOdd:
mov     eax, [ebp+random]
mov     edx, [ebp+count]
add     eax, edx
add     eax, 2
mov     [ebp+result], eax
```

```
loc_8049611:
mov     eax, [ebp+result]
leave
retn
interestingFunction endp
```

After renaming the arguments much of the functionality becomes clearer. The test in the first block checks if the loop counter is odd or even:

```
mov eax, [ebp+count];      if(count%2 == 0):
and eax, 1
test eax,eax
```

If the loop counter is even, the left code branch is executed. If the loop counter is odd, the right branch is executed. Thus, we can derive the following pseudo code for “interestingFunction”:

```
if count%2 == 0:
        result = (count+2)*random
    else:
        result = (count+2)+random
return result
```

We now have sufficient information to create a simple client that reads the challenge and sends back a response. A Python implementation of the client is listed below:

```
#!/usr/bin/python

import struct
import socket
import time

server = '172.16.1.20'
ports = [3422,4532,5923,2342,5532]

for i in range(5):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((server,ports[i]))
    challenge = s.recv(4096)[0]

    if i%2 == 0:
        response = (i+2)*ord(challenge)
    else:
        response = (i+2)+ord(challenge)

    print "ROUND:{0}".format(i)
    print "CHALLENGE:{0}({1})".format(challenge,ord(challenge))
    print "RESPONSE:{0}".format(response)
    print ""

    s.send(struct.pack('<I',response))
    flag = s.recv(4096)
    if len(flag) > 0:
        print "FLAG: {0}".format(flag)
    # Only necessary with local binary.
    # Possible to send next response before
    # socket has been opened.
    time.sleep(1)
```

Note the use of “`struct.pack`” with modifier “`<I`”, this just reorders the bytes of the response as little-endian.

```
#> python soln.py
ROUND:0
CHALLENGE:s(115)
RESPONSE:230

ROUND:1
CHALLENGE:O(79)
RESPONSE:82

ROUND:2
CHALLENGE:8(56)
RESPONSE:224

ROUND:3
CHALLENGE:(127)
RESPONSE:132

ROUND:4
CHALLENGE:-(45)
RESPONSE:270

FLAG: DemonViolatePride346
```

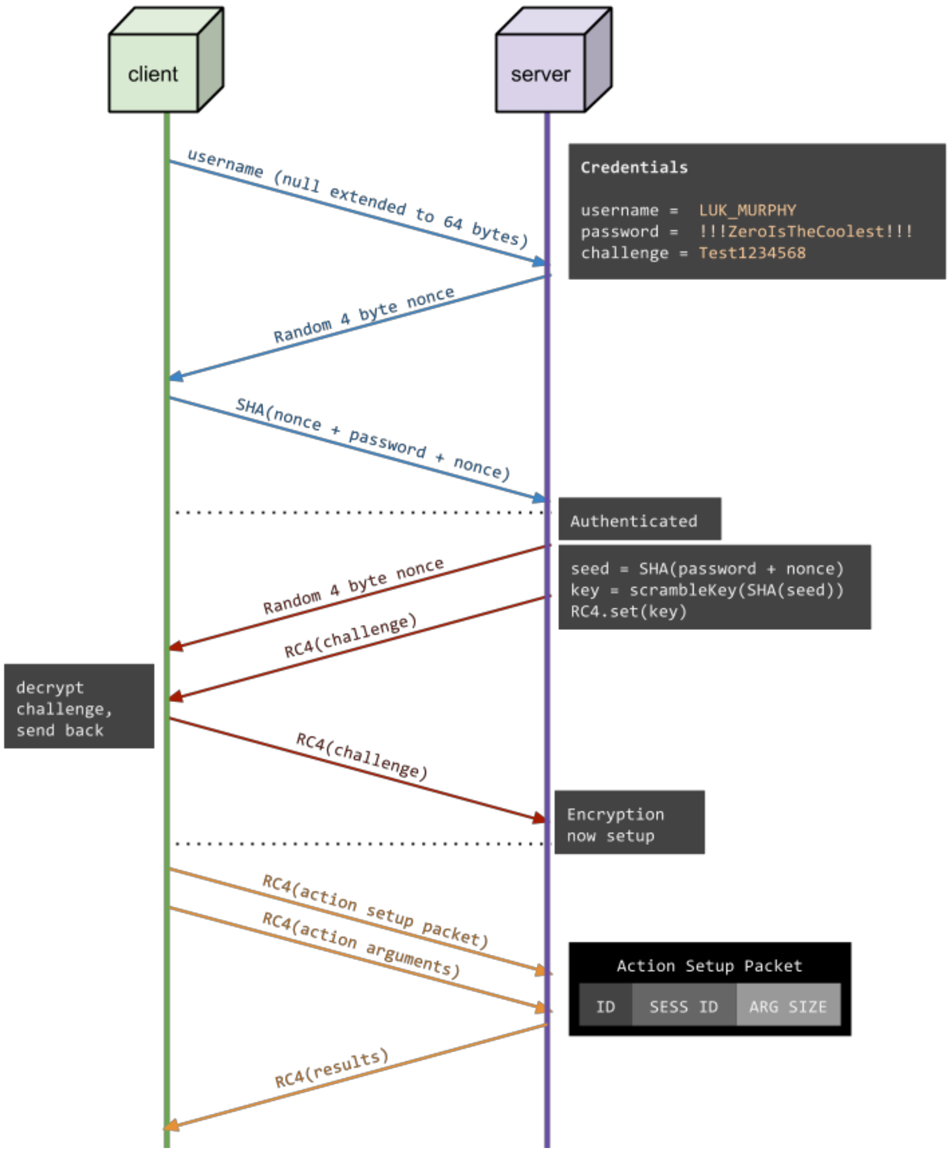
We run our client which iterates through 5 challenge/response rounds and then returns the flag for this challenge. **DemonViolatePride346**.

Reverse Engineering 3 - Forever Alone

Question: Terribad Corp has lost the client component of a legacy application that they no longer have the source code for. They want you to reverse engineer the provided server binary and build a client to interact with the server. Once you have done this, the server binary is running on 172.16.1.20 to test your client implementation against.

Designed Solution: Players need to reverse engineer the supplied C++ binary to determine the authentication and encryption protocol used by the server. Players also need to determine the commands available to the client program. Once they know these details, players need to implement a client to interface with the server to retrieve the flag. The following diagram illustrates the protocol that players need to reverse.

.



Write Up:
Note that throughout this write up “SHA” refers to SHA1**.**

Given a server binary, we are tasked with reversing the server’s communication protocol and writing a client capable of extracting the flag. Before studying the binary we run file and strings to gain a cursory understanding of the supplied binary.

```
#> file 10cb2de913234d75c8aa0d1d6219afec-re03
re03: ELF 32-bit LSB  executable, Intel 80386, version 1 (SYSV), dynamically
linked (uses shared libs), for GNU/Linux 2.6.24,
BuildID[sha1]=9ae210bbdc76ed78e5247554702a71b42a05c91d, stripped

#> strings 10cb2de913234d75c8aa0d1d6219afec-re03
**** SNIP ****
LUK_MURPHY
!!!Zer0IsTheCoolest!!!
RE03: Authentication failed for user '
RE03: User '
' Authenticated
RE03: Scrambling Key
Test12345678
RE03: Encryption setup failed for authenticated user
' encryption setup
7CClient
RE03: Unknown Command Type:
**** SNIP ****
```

Strings reveals a number of messages associated with setting up ports, chrooting (as we’ve seen in other challenges) etc. We also discover three unusual strings: LUK_MURPHY, !!!Zer0IsTheCoolest!!! and Test12345678. Other server messages indicate authentication, encryption and “command” functions. Perhaps we’ll need to negotiate user authentication and encryption!

To study the binary we’ll want a static disassembly in IDA and a local copy of the server binary to run and debug. We need to set up the server to run from /chroots/2013 and make it executable. We also need to note the process PID to attach a debugger to.

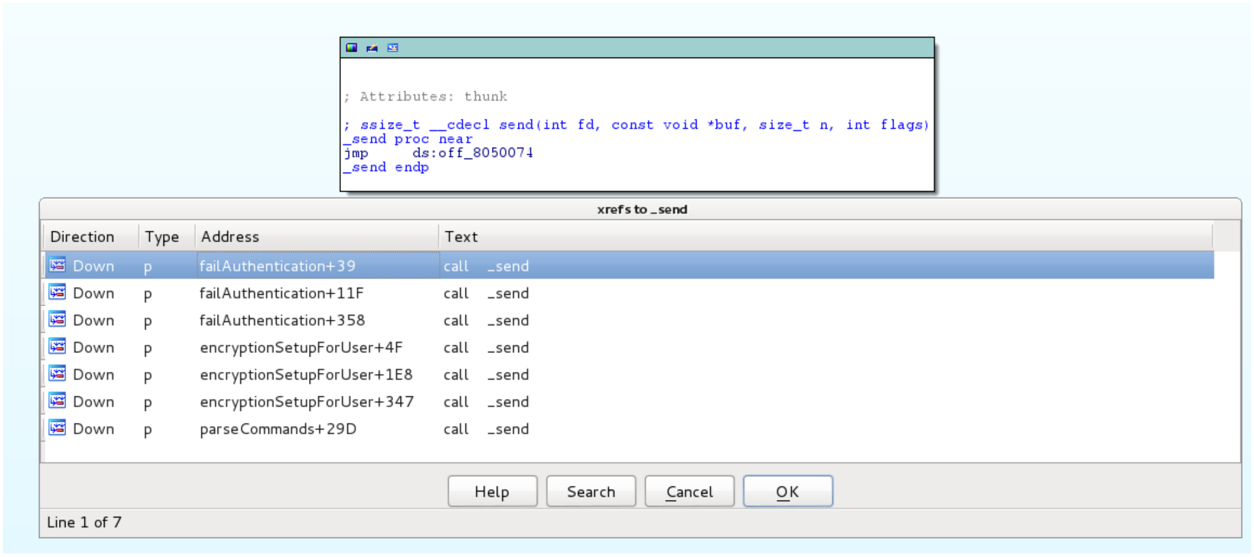
```
#> mkdir /chroots/2013
#> mkdir -p /chroots/2013/dev
#> mknod /chroots/2013/dev/urandom c 1 9
#> ./10cb2de913234d75c8aa0d1d6219afec-re03
Currently running as user 0 and group 0
Moving into chroot jail for user 2013. Path = '/chroots/2013'
Changing group from 0 to 1014
Changing user from 0 to 2013
```

```
Forking....
Child process 4171 spawned. Original process quitting
RE03: Waiting for connections on port 7821
```

Sending an arbitrary string to the server via netcat confirms connectivity. We also notice that the server returns a single hex value: 0xFF. Perhaps an error code? Although we don't know anything about the return value, we could use the server's "send" function as a starting point for our reversing.

```
#> echo "AAAAAAAAAA" | nc localhost 7821 | hexdump
RE03: Connection recieved on port 7821 from client 127.0.0.1
0000000 00ff
0000001
```

Loading the binary in IDA (MetaPC with default options will suffice for processor type) we see a number of pre-populated function names in IDA's function list, including a function **"_send"**, we'll start there. (**Note:** the xref function names below **won't** be populated by IDA straight away, we named them in later in our analysis)



```
; Attributes: thunk

; ssize_t __cdecl send(int fd, const void *buf, size_t n, int flags)
_send proc near
jmp     ds:off_8050074
_send endp
```

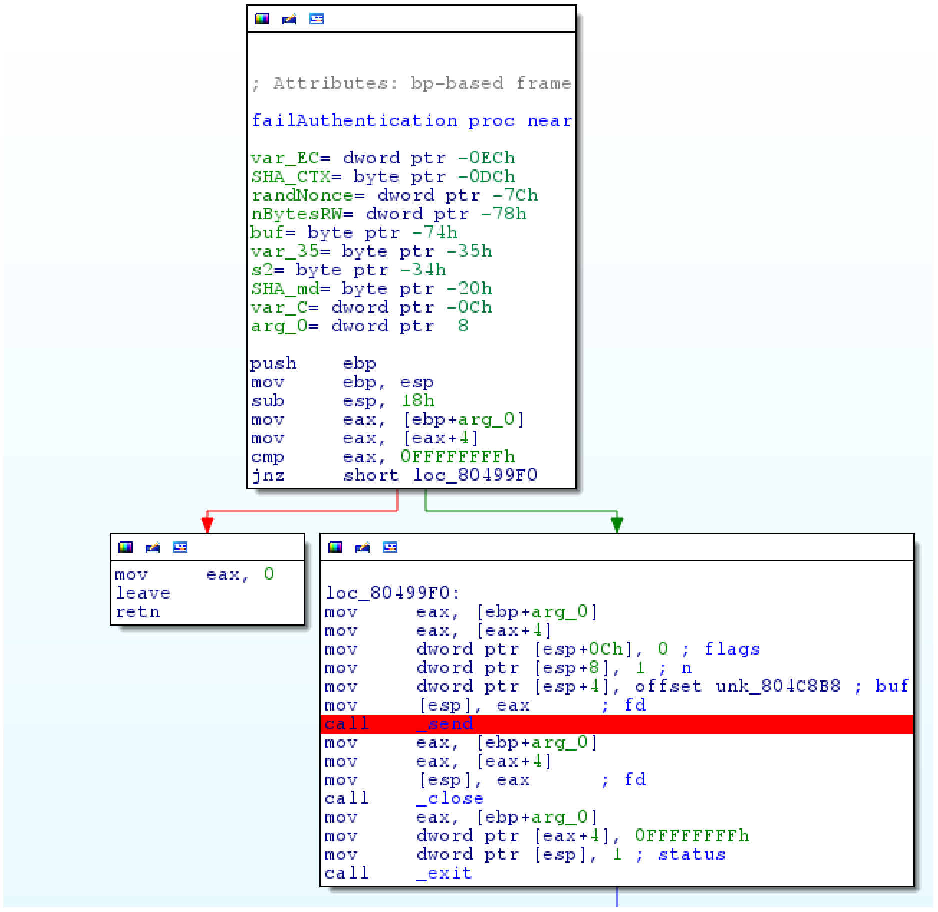
xrefs to _send

Direction	Type	Address	Text
Down	p	failAuthentication+39	call _send
Down	p	failAuthentication+11F	call _send
Down	p	failAuthentication+358	call _send
Down	p	encryptionSetupForUser+4F	call _send
Down	p	encryptionSetupForUser+1E8	call _send
Down	p	encryptionSetupForUser+347	call _send
Down	p	parseCommands+29D	call _send

Following the first xref, we come to a _send call (0x8049A11) and want to set a breakpoint to establish if this call is sending us the failure code. We can accomplish this with gdb as follows:

```
#> gdb --pid 4171 --eval-command "set follow-fork-mode child" --eval-command
"handle SIGALRM ignore" --eval-command "break *0x8049a11" --eval-command "c"
GNU gdb (Ubuntu 7.7-0ubuntu3.1) 7.7
**** SNIP ****
0xf76f5430 in __kernel_vsyscall ()
Signal        Stop  Print  Pass to program     Description
SIGALRM       No    No     No                 Alarm clock
```

```
Breakpoint 1 at 0x8049a3a
Continuing.
```



```
; Attributes: bp-based frame

failAuthentication proc near

var_EC= dword ptr -0ECh
SHA_CTX= byte ptr -0DCh
randNonce= dword ptr -7Ch
nBytesRW= dword ptr -78h
buf= byte ptr -74h
var_35= byte ptr -35h
s2= byte ptr -34h
SHA_md= byte ptr -20h
var_C= dword ptr -0Ch
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 18h
mov     eax, [ebp+arg_0]
mov     eax, [eax+4]
cmp     eax, 0FFFFFFFFh
jnz     short loc_80499F0
```

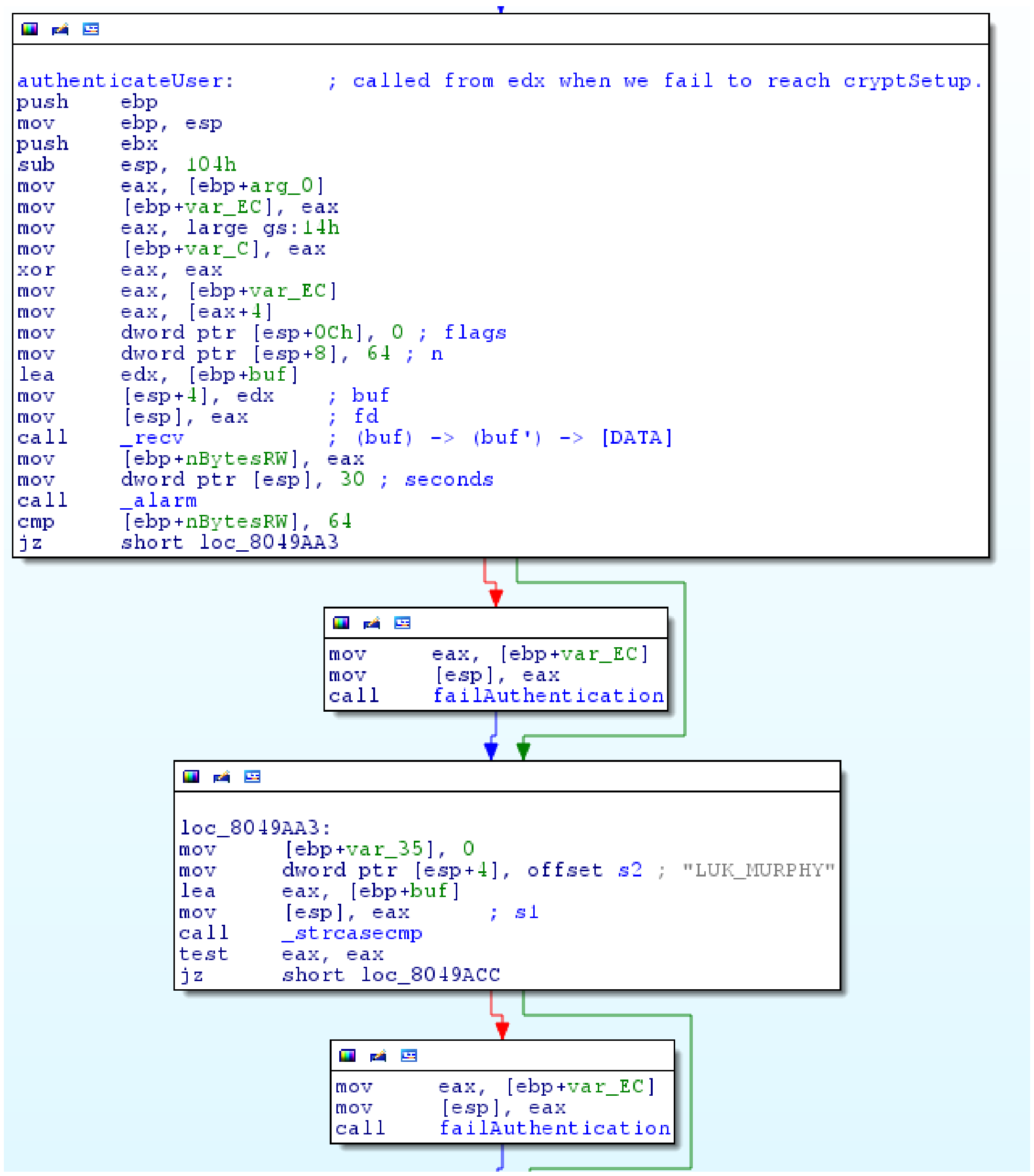
```
mov     eax, 0
leave
retn
```

```
loc_80499F0:
mov     eax, [ebp+arg_0]
mov     eax, [eax+4]
mov     dword ptr [esp+0Ch], 0 ; flags
mov     dword ptr [esp+8], 1 ; n
mov     dword ptr [esp+4], offset unk_804C8B8 ; buf
mov     [esp], eax      ; fd
call    _send
mov     eax, [ebp+arg_0]
mov     eax, [eax+4]
mov     [esp], eax      ; fd
call    _close
mov     eax, [ebp+arg_0]
mov     dword ptr [eax+4], 0FFFFFFFFh
mov     dword ptr [esp], 1 ; status
call    _exit
```

GDB confirms that we reach this _send call. Clearly this isn't the path we want to be on; shortly after _send we see calls to _close (closing the server socket) and _exit (ending the program). We can also see number of auto-populated C++ style names (namespace scope etc.) and deeply nested function calls, from these we can assume that the server is written in C++.

Looking further down IDA's output, we find a call to _recv, breaking here (0x8049A7B) confirms that this is the call responsible for reading our 'AAAAAAAAAA' string. At 0x8049AB5 we see a comparison between our input and the string "LUK_MURPHY" (a username perhaps?). We also see the server checking that it received 64 bytes and terminating the connection if it didn't.

We try sending the 'LUK_MURPHY' followed by 54 null bytes to the server and observe the check passing in GDB. The server then returns a random 4 byte nonce value.

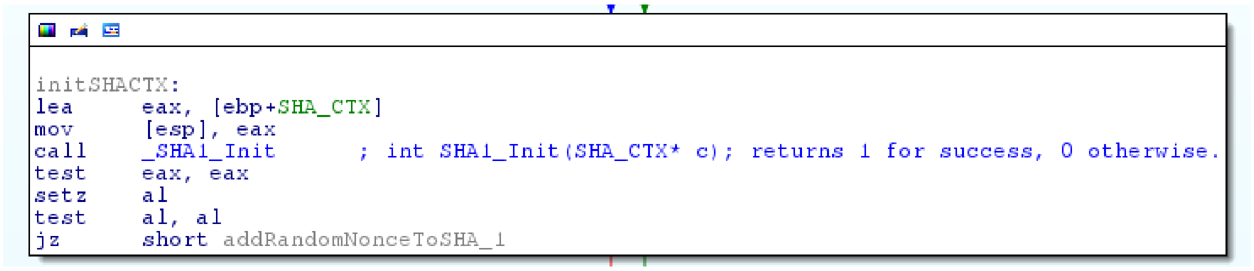

```
authenticateUser:        ; called from edx when we fail to reach cryptSetup.
push     ebp
mov      ebp, esp
push     ebx
sub      esp, 104h
mov      eax, [ebp+arg_0]
mov      [ebp+var_EC], eax
mov      eax, large gs:14h
mov      [ebp+var_C], eax
xor      eax, eax
mov      eax, [ebp+var_EC]
mov      eax, [eax+4]
mov      dword ptr [esp+0Ch], 0 ; flags
mov      dword ptr [esp+8], 64 ; n
lea      edx, [ebp+buf]
mov      [esp+4], edx     ; buf
mov      [esp], eax       ; fd
call     _recv            ; (buf) -> (buf') -> [DATA]
mov      [ebp+nBytesRW], eax
mov      dword ptr [esp], 30 ; seconds
call     _alarm
cmp      [ebp+nBytesRW], 64
jz       short loc_8049AA3
```

```
mov      eax, [ebp+var_EC]
mov      [esp], eax
call     failAuthentication
```

```
loc_8049AA3:
mov      [ebp+var_35], 0
mov      dword ptr [esp+4], offset s2 ; "LUK_MURPHY"
lea      eax, [ebp+buf]
mov      [esp], eax        ; s1
call     _strcasecmp
test     eax, eax
jz       short loc_8049ACC
```

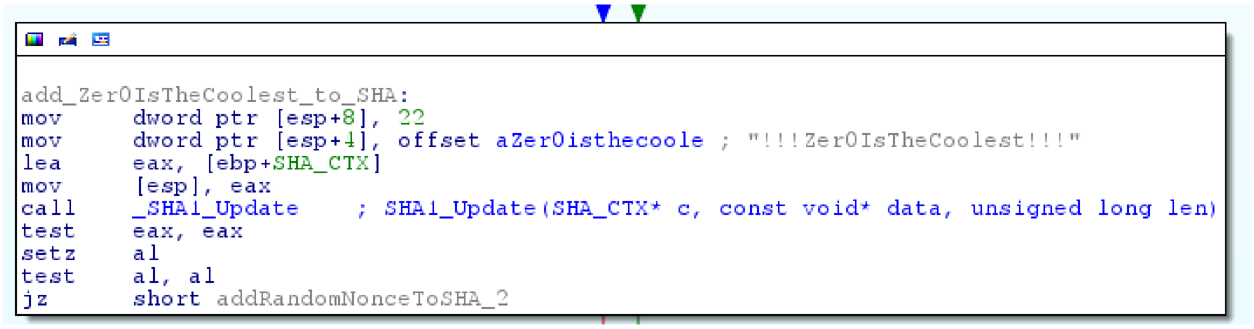
```
mov      eax, [ebp+var_EC]
mov      [esp], eax
call     failAuthentication
```

Further down at 0x8049B5B we see the server initialising a SHA digest control structure(SHA_CTX), appending data to the structure's message 3 times, then calculating the digest of the message. This digest is compared against a value sent to the server. Tracking the function variables and the order of update calls, we deduce that we need to send the server the SHA digest of [<nonce> + <password> + <nonce>].



```
initSHACTX:
lea     eax, [ebp+SHA_CTX]
mov     [esp], eax
call    _SHA1_Init      ; int SHA1_Init(SHA_CTX* c); returns 1 for success, 0 otherwise.
test    eax, eax
setz    al
test    al, al
jz      short addRandomNonceToSHA_1
```

Looking near the second sha1_update function we can see that the password "!!!Zer0IsTheCoolest!!!" is hardcoded and plainly visible in the server binary (referenced at 0x8049BB3).



```
add_Zer0IsTheCoolest_to_SHA:
mov     dword ptr [esp+8], 22
mov     dword ptr [esp+4], offset aZer0isthecoole ; "!!!Zer0IsTheCoolest!!!"
lea     eax, [ebp+SHA_CTX]
mov     [esp], eax
call    _SHA1_Update    ; SHA1_Update(SHA_CTX* c, const void* data, unsigned long len)
test    eax, eax
setz    al
test    al, al
jz      short addRandomNonceToSHA_2
```

We choose to start writing our client in Python to capture what we know about the authentication process so far. The Python packages "socket" and "sha" make interacting with the server and calculating the necessary SHA digest very simple.

Running the current iteration of our python client against the local server causes the following messages to be output in the terminal running our local server binary.

Terminal 1:

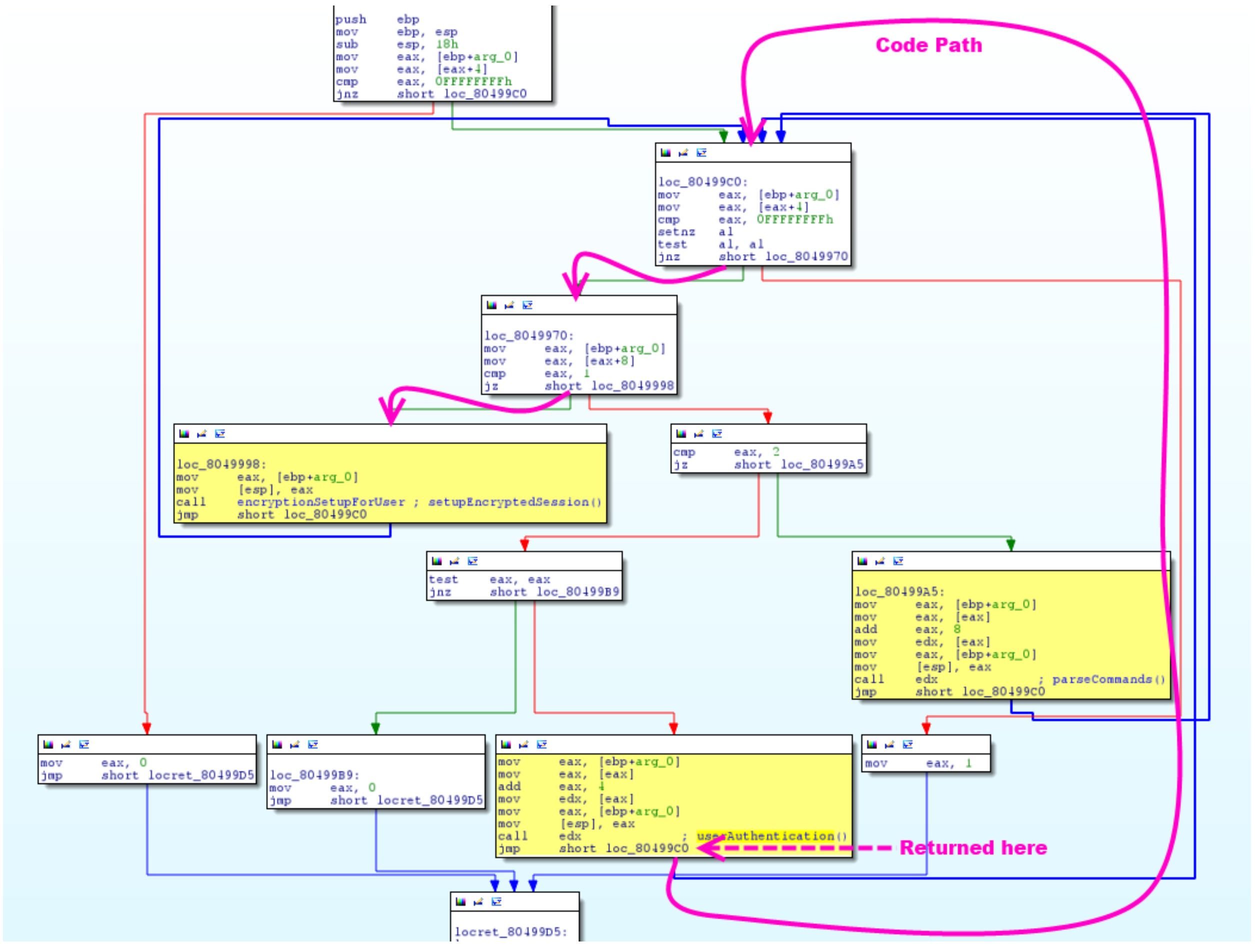
```
#> python client.py
```

Local Server Terminal:

```
RE03: User 'LUK_MURPHY' Authenticated
RE03: Scrambling Key
RE03: Encryption setup failed for authenticated user LUK_MURPHY
```

It seems from the output that we are successfully authenticating as user LUK_MURPHY but the encryption setup is failing. Clearly there's more to be done.

In GDB, we set a breakpoint at the end of the discovered authentication function and step out of the function. We now find ourselves in a higher level loop containing three call functions (including calling our authentication function), beginning at 0x8049958. Stepping through the server loop with GDB, we find that the server now takes the branch to loc_8049998 (the top-left highlighted code block displayed below).



```
push    ebp
mov     ebp, esp
sub     esp, 18h
mov     eax, [ebp+arg_0]
mov     eax, [eax+4]
cmp     eax, 0FFFFFFFFh
jnz     short loc_80499C0
```

```
loc_80499C0:
mov     eax, [ebp+arg_0]
mov     eax, [eax+4]
cmp     eax, 0FFFFFFFFh
setnz   al
test    al, al
jnz     short loc_8049970
```

```
loc_8049970:
mov     eax, [ebp+arg_0]
mov     eax, [eax+8]
cmp     eax, 1
jz      short loc_8049998
```

```
loc_8049998:
mov     eax, [ebp+arg_0]
mov     [esp], eax
call    encryptionSetupForUser ; setupEncryptedSession()
jmp     short loc_80499C0
```

```
cmp     eax, 2
jz      short loc_80499A5
```

```
test    eax, eax
jnz     short loc_80499B9
```

```
loc_80499A5:
mov     eax, [ebp+arg_0]
mov     eax, [eax]
add     eax, 8
mov     edx, [eax]
mov     eax, [ebp+arg_0]
mov     [esp], eax
call    edx             ; parseCommands()
jmp     short loc_80499C0
```

```
mov     eax, 0
jmp     short locret_80499D5
```

```
loc_80499B9:
mov     eax, 0
jmp     short locret_80499D5
```

```
mov     eax, [ebp+arg_0]
mov     eax, [eax]
add     eax, 4
mov     edx, [eax]
mov     eax, [ebp+arg_0]
mov     [esp], eax
call    edx             ; userAuthentication()
jmp     short loc_80499C0
```

```
mov     eax, 1
```

```
locret_80499D5:
```

Next we will analyse the function that we have named encryptionSetupForUser(0x8049f32) in the diagram above. A quick glance over this function allows us to see the use of SHA functions in addition to RC4 functions.

Up to the SHA_Final call at 0x804A050 this function looks similar to the authentication function although it only adds one nonce to the password before hashing; the server calculates the SHA digest of [<password> + <nonce>]. At 0x804A086 this digest and the nonce are passed as arguments to a function at 0x8049D54. After this function call, the server uses the SHA digest buffer as the RC4 key.

Stepping into the function at 0x8049D54 we see a reference to the string “RE03: Scrambling Key\n”, which we know occurs before the message “RE03: Encryption setup failed for authenticated user LUK_MURPY”, so scramble key is being called. We can also see the function is performing many bitwise operations.

Without knowing exactly what the scramble function is doing, we can see the result on the SHA digest passed as input (known as the "seed key" from here on). A pointer to the seed key was passed as the 3rd argument to scramble, the nonce was passed as the 2nd argument. In GDB, we break at 0x8049D5B (start of scramble) and 0x8049F24 (end of scramble) and we inspect these argument values.

Nonce @ 0x8049D5B (start of scramble function):

```
(gdb) x/4bx $ebp+0xc
0xffacef34:   0xc0   0xba   0x4c   0xf7
```

Seed Key @ 0x8049D5B (start of scramble function):

```
(gdb) x/wx $ebp+0x10
0xffacef38:   0xffacf038
(gdb) x/20bx 0xffacf038
0xffacf038:   0x2c   0x25   0x25   0x75   0xc6   0x8f   0xbc   0xd9
0xffacf040:   0xc9   0xdc   0x4b   0x98   0xcf   0x84   0x17   0xfe
0xffacf048:   0x55   0x9f   0x51   0xbe
```

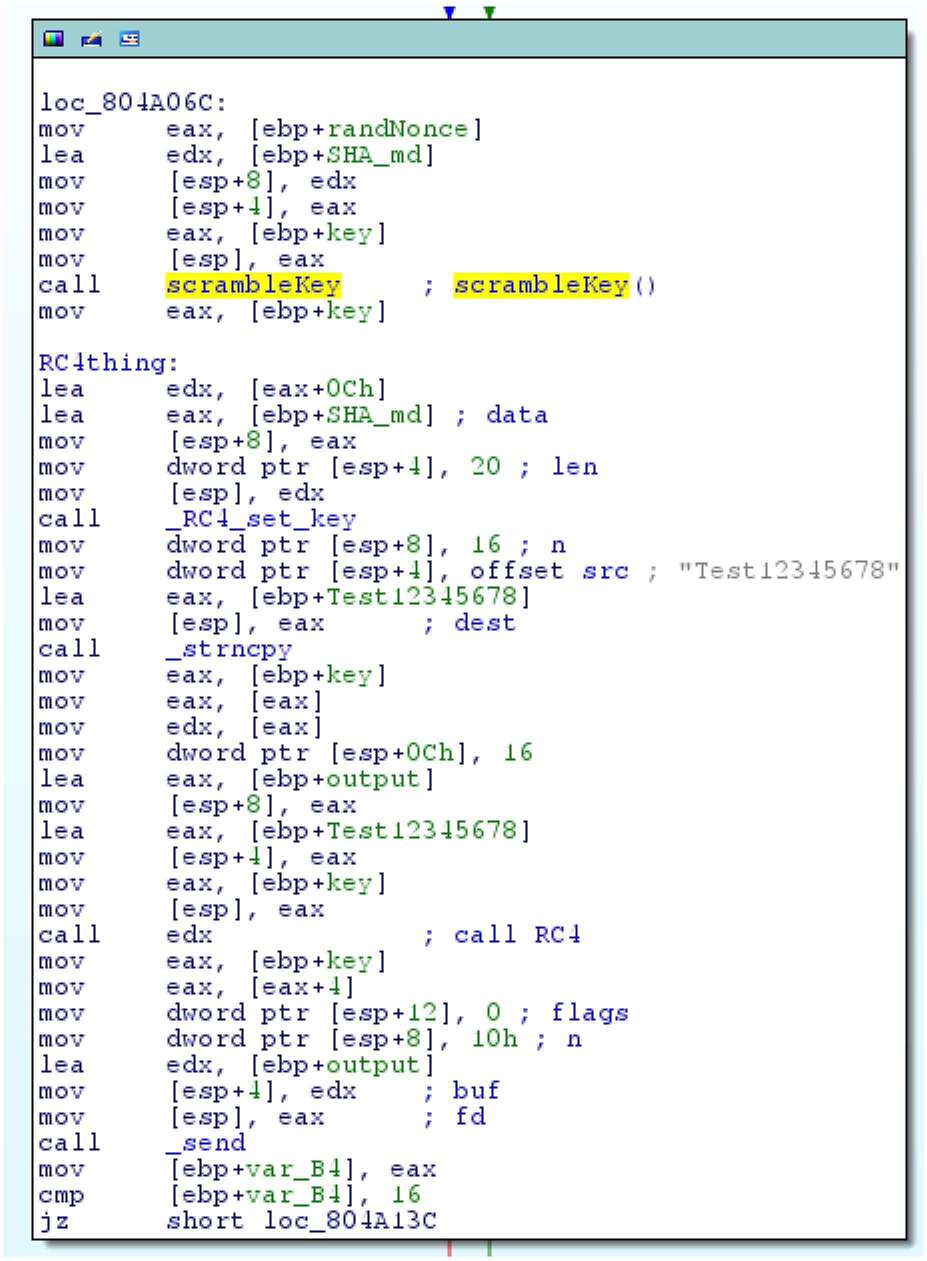
Key @ 0x8049F24 (end of scramble function):

```
(gdb) x/20bx 0xffacf038
0xffacf038:   0x07   0x53   0xbc   0xbd   0x1f   0x32   0xe0   0xdb
0xffacf040:   0x8a   0xa1   0xa6   0x25   0x6f   0x4b   0xd2   0x0a
0xffacf048:   0xc4   0xe1   0x4b   0xb8
```

Ok, so the scramble function is modifying the Seed Key buffer values in place.

We take a quick look back at the encryption setup function that called scramble keys. We observe the scrambled key being used to initialise RC4. The server then encrypts the string "Test12345678" with RC4 and sends it to the client.

The server then receives a response from the client (0x804A162), decrypts it with RC4 (We used GDB to step into the function at 0x804A2CA and see the decryption functions) and compares it to "Test12345678", a classic "challenge response" mechanism.



```
loc_804A06C:
mov     eax, [ebp+randNonce]
lea     edx, [ebp+SHA_md]
mov     [esp+8], edx
mov     [esp+4], eax
mov     eax, [ebp+key]
mov     [esp], eax
call    scrambleKey      ; scrambleKey()
mov     eax, [ebp+key]

RC4thing:
lea     edx, [eax+0Ch]
lea     eax, [ebp+SHA_md] ; data
mov     [esp+8], eax
mov     dword ptr [esp+4], 20 ; len
mov     [esp], edx
call    _RC4_set_key
mov     dword ptr [esp+8], 16 ; n
mov     dword ptr [esp+4], offset src ; "Test12345678"
lea     eax, [ebp+Test12345678]
mov     [esp], eax       ; dest
call    _strncpy
mov     eax, [ebp+key]
mov     eax, [eax]
mov     edx, [eax]
mov     dword ptr [esp+0Ch], 16
lea     eax, [ebp+output]
mov     [esp+8], eax
lea     eax, [ebp+Test12345678]
mov     [esp+4], eax
mov     eax, [ebp+key]
mov     [esp], eax
call    edx              ; call RC4
mov     eax, [ebp+key]
mov     eax, [eax+4]
mov     dword ptr [esp+12], 0 ; flags
mov     dword ptr [esp+8], 10h ; n
lea     edx, [ebp+output]
mov     [esp+4], edx     ; buf
mov     [esp], eax       ; fd
call    _send
mov     [ebp+var_B4], eax
cmp     [ebp+var_B4], 16
jz      short loc_804A13C
```

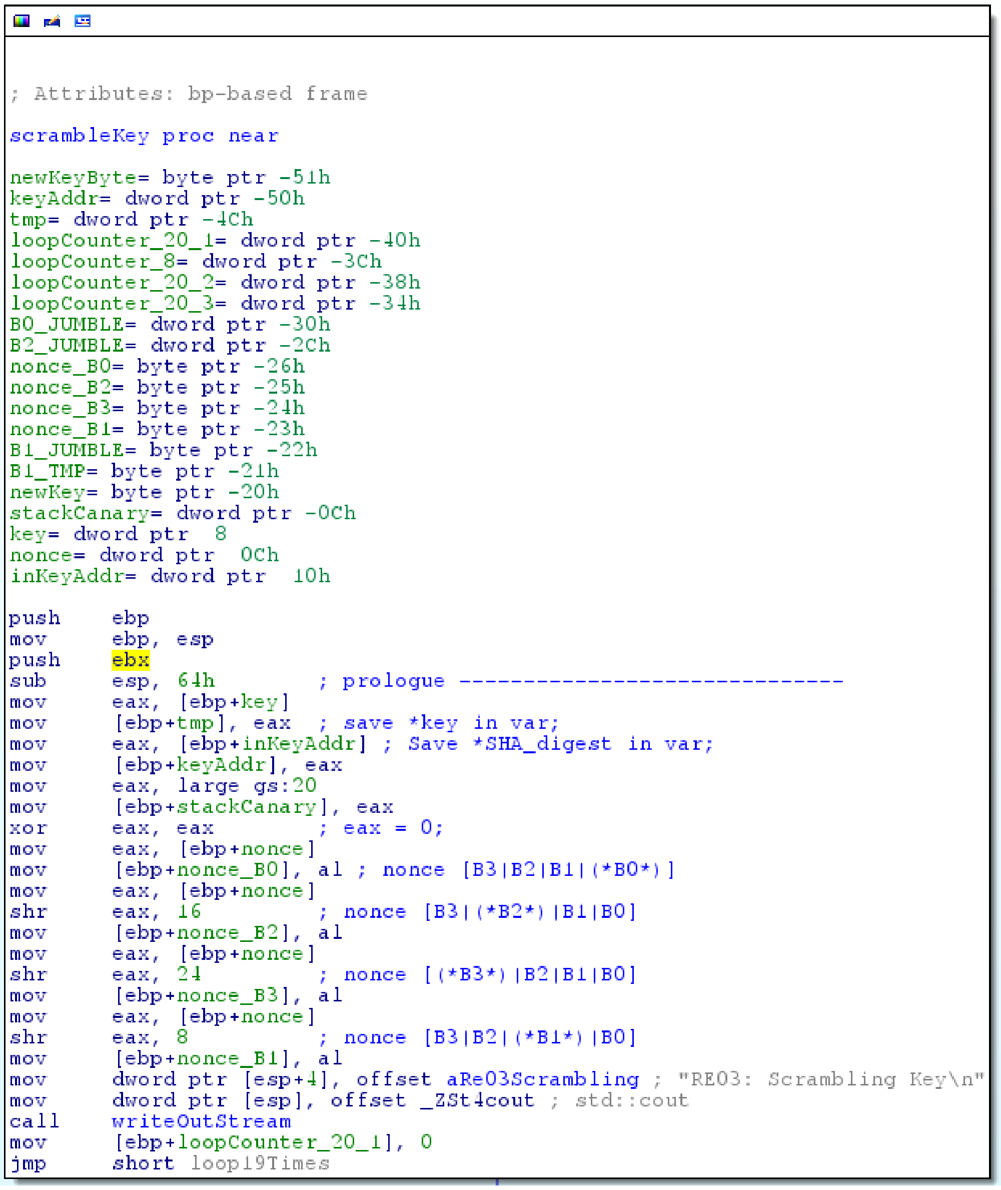
RC4 is a stream cipher, this means that if we don't initialise it with the exact same key as the server, it will output a different keystream and we won't be able to communicate with the server. Additionally, the RC4 keystream needs to be synchronised with the server and this is the reason that in our client we can't simply replay the challenge that we received from the server back to the server.

There seems to be no way around it, we will have to reverse the scramble key function so lets focus back on that function.

We can easily construct the seed key in our client but, because we are using Python we're going to need to reimplement the scramble function. If we were using C or some other compiled language we could extract the entire function and with some small modifications reuse it in our code.

There is a couple of ways we could approach understanding the scramble function. We could apply a decompiler to convert the assembly instructions into a higher level language or we could take the harder, but more personally satisfying option and manually decompile the function by understanding every single operation.

Beginning with the function prologue and setup (see below) we start to populate argument and variable names.



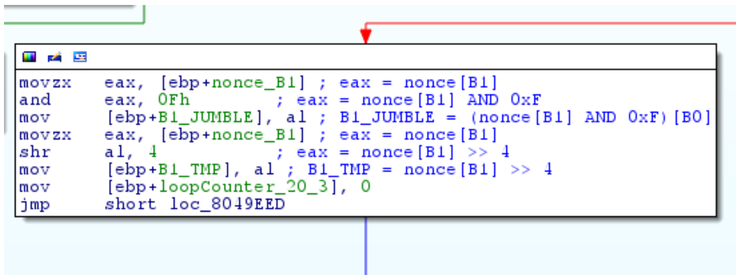
```
; Attributes: bp-based frame

scrambleKey proc near

newKeyByte= byte ptr -51h
keyAddr= dword ptr -50h
tmp= dword ptr -4Ch
loopCounter_20_1= dword ptr -40h
loopCounter_8= dword ptr -3Ch
loopCounter_20_2= dword ptr -38h
loopCounter_20_3= dword ptr -34h
B0_JUMBLE= dword ptr -30h
B2_JUMBLE= dword ptr -2Ch
nonce_B0= byte ptr -26h
nonce_B2= byte ptr -25h
nonce_B3= byte ptr -24h
nonce_B1= byte ptr -23h
B1_JUMBLE= byte ptr -22h
B1_TMP= byte ptr -21h
newKey= byte ptr -20h
stackCanary= dword ptr -0Ch
key= dword ptr  8
nonce= dword ptr  0Ch
inKeyAddr= dword ptr  10h

push    ebp
mov     ebp, esp
push    ebx
sub     esp, 64h          ; prologue ------------------------------
mov     eax, [ebp+key]
mov     [ebp+tmp], eax    ; save *key in var;
mov     eax, [ebp+inKeyAddr] ; Save *SHA_digest in var;
mov     [ebp+keyAddr], eax
mov     eax, large gs:20
mov     [ebp+stackCanary], eax
xor     eax, eax          ; eax = 0;
mov     eax, [ebp+nonce]
mov     [ebp+nonce_B0], al ; nonce [B3|B2|B1|(*B0*)]
mov     eax, [ebp+nonce]
shr     eax, 16           ; nonce [B3|(*B2*)|B1|B0]
mov     [ebp+nonce_B2], al
mov     eax, [ebp+nonce]
shr     eax, 24           ; nonce [(*B3*)|B2|B1|B0]
mov     [ebp+nonce_B3], al
mov     eax, [ebp+nonce]
shr     eax, 8            ; nonce [B3|B2|(*B1*)|B0]
mov     [ebp+nonce_B1], al
mov     dword ptr [esp+4], offset aRe03Scrambling ; "RE03: Scrambling Key\n"
mov     dword ptr [esp], offset _ZSt4cout ; std::cout
call    writeOutStream
mov     [ebp+loopCounter_20_1], 0
jmp     short loop19Times
```

The function of all these variables won’t be clear on the first pass; the variable names above reflect the final result.

By following the behaviour of conditional jumps (in this case **jnz** instructions) we can see 4 loops in the scramble function. The 1st, 3rd and 4th loops iterate 20 times while the 2nd iterates 8 times. Given that the seed key is 20 bytes long we might reasonably assume that the 1st, 3rd and 4th loops are operations performed on each byte in the key.

At the start of the scramble function we saw each byte of the nonce being stored in a stack variable. Tracing these variables through the function we see that the behaviour of each loop is dependent on a different byte of the nonce. Loop 1 is dependent on the 1st byte, loop 2 is dependent on the 3rd byte, loop 3 is dependent on the 4th byte, and loop 4 is dependent on the 2nd byte.

The 4th loop is slightly more interesting in that it relies on a transformation of nonce byte 2.



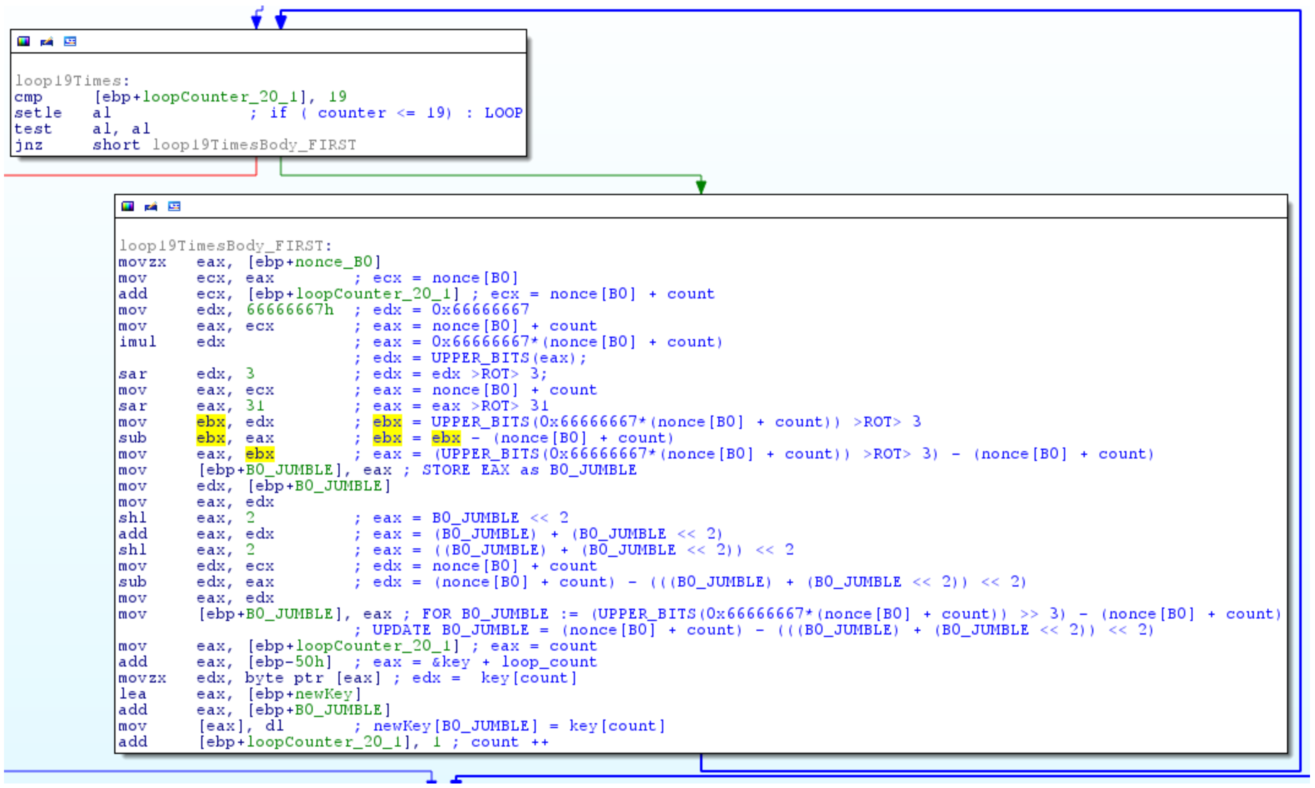
```
movzx   eax, [ebp+nonce_B1] ; eax = nonce[B1]
and     eax, 0Fh          ; eax = nonce[B1] AND 0xF
mov     [ebp+B1_JUMBLE], al ; B1_JUMBLE = (nonce[B1] AND 0xF)[B0]
movzx   eax, [ebp+nonce_B1] ; eax = nonce[B1]
shr     al, 4             ; eax = nonce[B1] >> 4
mov     [ebp+B1_TMP], al ; B1_TMP = nonce[B1] >> 4
mov     [ebp+loopCounter_20_3], 0
jmp     short loc_8049EED
```

At first glance the scramble function is full of bitwise shifts and arithmetic operations and looks like a nightmare to reverse. Don’t despair! These low level operations look significantly more convoluted than the higher level algorithm. By carefully documenting the effect of each line, we can then take a step back and gradually see higher and higher abstractions of the algorithm. The following pages contain line by line comments for each loop and python code that implements the functionality.

Before performing the scrambling operations in our python script we need to transform the seed key to an array of bytes, we create a copy of the seed key array for the scrambled key:

```
# Build byte arrays for seed and scrambled key.
seedKey = [ord(x) for x in inKey]
newKey = [0]*len(seedKey)
keyLen = len(seedKey)
```

Loop 1



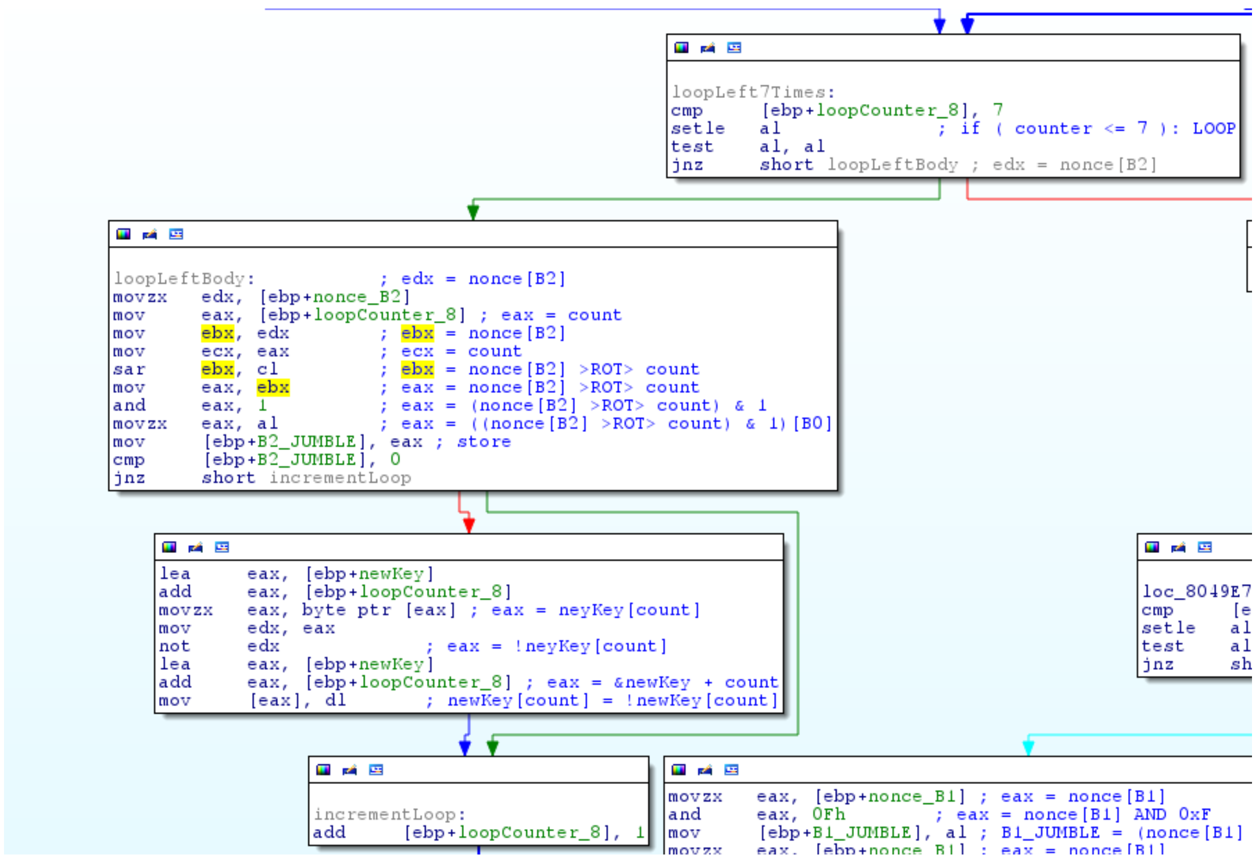
```
loop19Times:
cmp      [ebp+loopCounter_20_1], 19
setle    al             ; if ( counter <= 19) : LOOP
test     al, al
jnz      short loop19TimesBody_FIRST
```

```
loop19TimesBody_FIRST:
movzx    eax, [ebp+nonce_B0]
mov      ecx, eax          ; ecx = nonce[B0]
add      ecx, [ebp+loopCounter_20_1] ; ecx = nonce[B0] + count
mov      edx, 66666667h  ; edx = 0x66666667
mov      eax, ecx          ; eax = nonce[B0] + count
imul     edx               ; eax = 0x66666667*(nonce[B0] + count)
                           ; edx = UPPER_BITS(eax);
sar      edx, 3            ; edx = edx >ROT> 3;
mov      eax, ecx          ; eax = nonce[B0] + count
sar      eax, 31           ; eax = eax >ROT> 31
mov      ebx, edx          ; ebx = UPPER_BITS(0x66666667*(nonce[B0] + count)) >ROT> 3
sub      ebx, eax          ; ebx = ebx - (nonce[B0] + count)
mov      eax, ebx          ; eax = (UPPER_BITS(0x66666667*(nonce[B0] + count)) >ROT> 3) - (nonce[B0] + count)
mov      [ebp+B0_JUMBLE], eax ; STORE EAX as B0_JUMBLE
mov      edx, [ebp+B0_JUMBLE]
mov      eax, edx
shl      eax, 2            ; eax = B0_JUMBLE << 2
add      eax, edx          ; eax = (B0_JUMBLE) + (B0_JUMBLE << 2)
shl      eax, 2            ; eax = ((B0_JUMBLE) + (B0_JUMBLE << 2)) << 2
mov      edx, ecx          ; edx = nonce[B0] + count
sub      edx, eax          ; edx = (nonce[B0] + count) - (((B0_JUMBLE) + (B0_JUMBLE << 2)) << 2)
mov      eax, edx
mov      [ebp+B0_JUMBLE], eax ; FOR B0_JUMBLE := (UPPER_BITS(0x66666667*(nonce[B0] + count)) >> 3) - (nonce[B0] + count)
                           ; UPDATE B0_JUMBLE = (nonce[B0] + count) - (((B0_JUMBLE) + (B0_JUMBLE << 2)) << 2)
mov      eax, [ebp+loopCounter_20_1] ; eax = count
add      eax, [ebp-50h]  ; eax = &key + loop_count
movzx    edx, byte ptr [eax] ; edx =  key[count]
lea      eax, [ebp+newKey]
add      eax, [ebp+B0_JUMBLE]
mov      [eax], dl         ; newKey[B0_JUMBLE] = key[count]
add      [ebp+loopCounter_20_1], 1 ; count ++
```

Translates to:

```
# Loop 1
B0 = nonce[0]
    for i in range(keyLen):
        B0_JUMBLE = (i+B0)%20
        newKey[B0_JUMBLE]=(seedKey[i])
```

Loop 2



```
loopLeft7Times:
cmp     [ebp+loopCounter_8], 7
setle   al              ; if ( counter <= 7 ): LOOP
test    al, al
jnz     short loopLeftBody ; edx = nonce[B2]
```

```
loopLeftBody:               ; edx = nonce[B2]
movzx   edx, [ebp+nonce_B2]
mov     eax, [ebp+loopCounter_8] ; eax = count
mov     ebx, edx            ; ebx = nonce[B2]
mov     ecx, eax            ; ecx = count
sar     ebx, cl             ; ebx = nonce[B2] >ROT> count
mov     eax, ebx            ; eax = nonce[B2] >ROT> count
and     eax, 1              ; eax = (nonce[B2] >ROT> count) & 1
movzx   eax, al             ; eax = ((nonce[B2] >ROT> count) & 1)[B0]
mov     [ebp+B2_JUMBLE], eax ; store
cmp     [ebp+B2_JUMBLE], 0
jnz     short incrementLoop
```

```
lea     eax, [ebp+newKey]
add     eax, [ebp+loopCounter_8]
movzx   eax, byte ptr [eax] ; eax = neyKey[count]
mov     edx, eax
not     edx             ; eax = !neyKey[count]
lea     eax, [ebp+newKey]
add     eax, [ebp+loopCounter_8] ; eax = &newKey + count
mov     [eax], dl       ; newKey[count] = !newKey[count]
```

```
loc_8049E7
cmp     [e
setle   al
test    al
jnz     sh
```

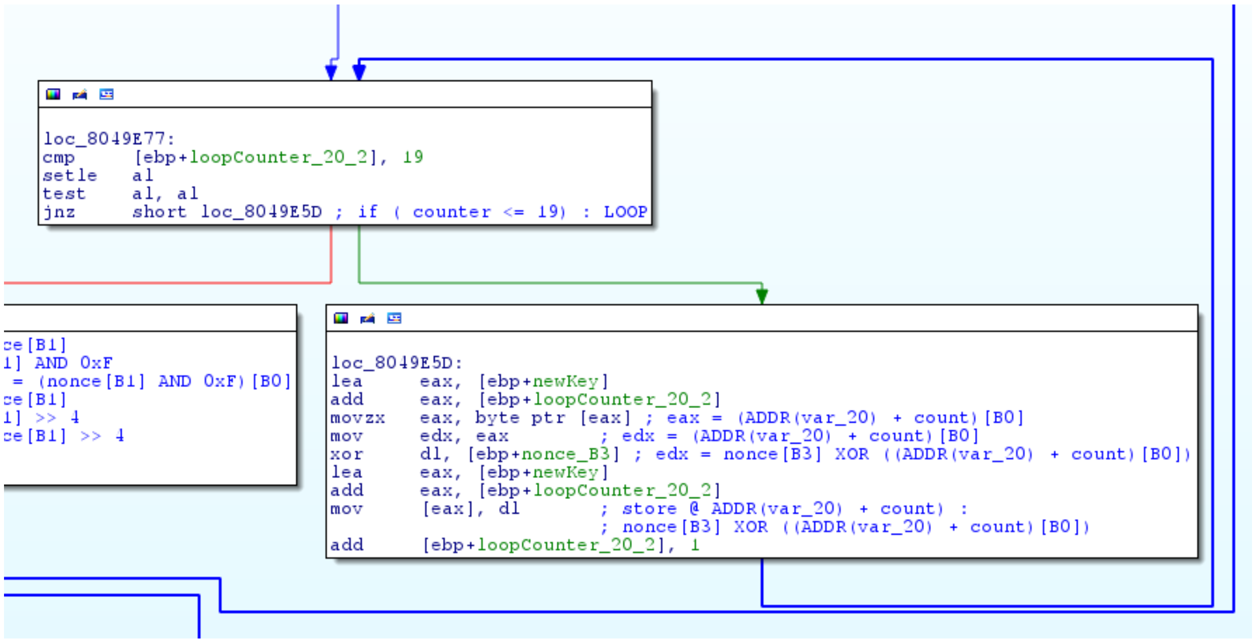
```
incrementLoop:
add     [ebp+loopCounter_8], 1
```

```
movzx   eax, [ebp+nonce_B1] ; eax = nonce[B1]
and     eax, 0Fh        ; eax = nonce[B1] AND 0xF
mov     [ebp+B1_JUMBLE], al ; B1_JUMBLE = (nonce[B1]
movzx   eax, [ebp+nonce_B1] ; eax = nonce[B1]
```

Translates to:

```
# Loop 2
B2 = nonce[2]
for count in range(8):
        B2_JUMBLE = (B2>>count)%2
        if B2_JUMBLE == 0:
                newKey[count] = (~(newKey[count]))
```

Loop 3



```
loc_8049E77:
cmp     [ebp+loopCounter_20_2], 19
setle   al
test    al, al
jnz     short loc_8049E5D ; if ( counter <= 19) : LOOP
```

```
ce[B1]
1] AND 0xF
 = (nonce[B1] AND 0xF)[B0]
ce[B1]
1] >> 4
ce[B1] >> 4
```

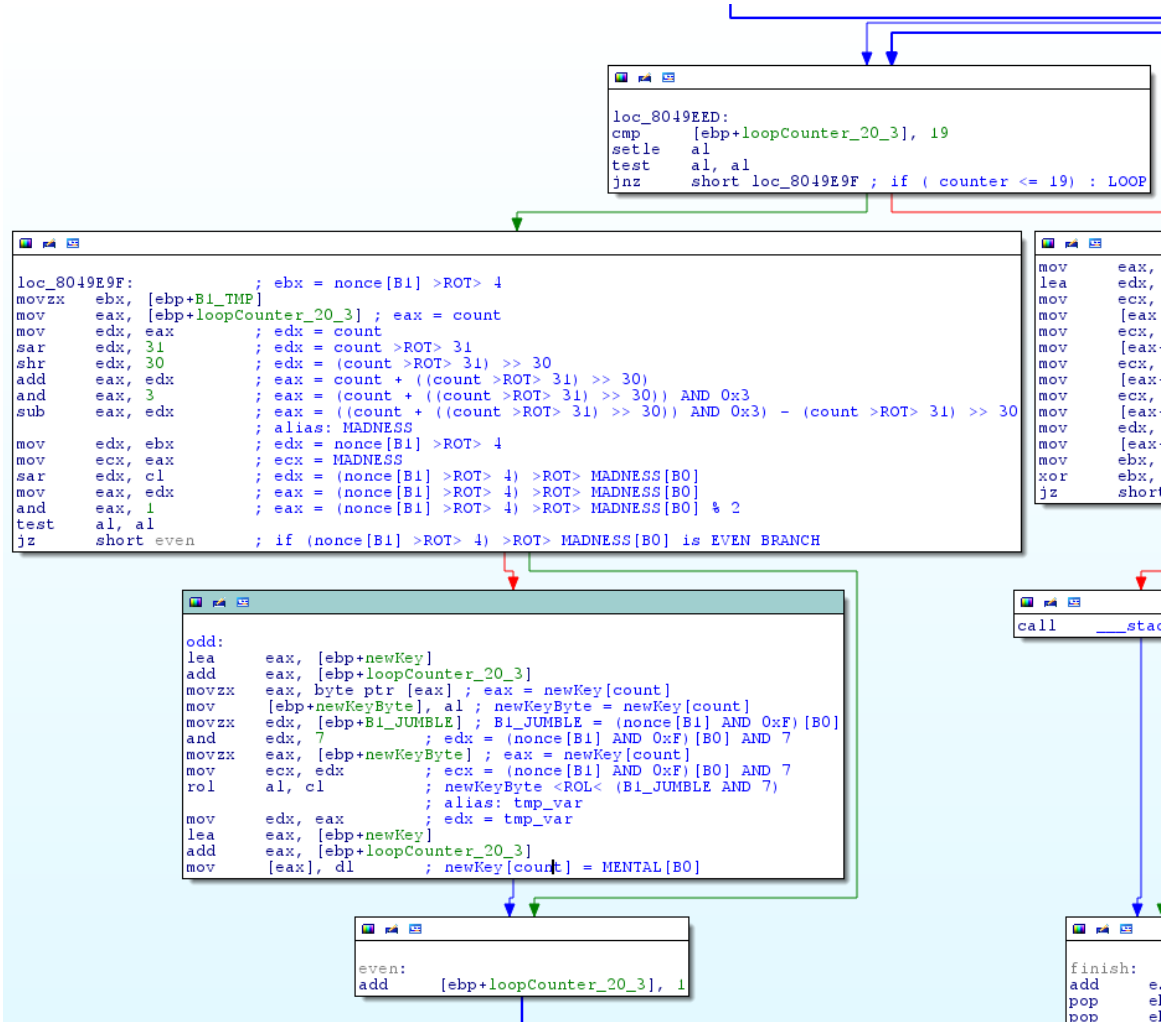
```
loc_8049E5D:
lea     eax, [ebp+newKey]
add     eax, [ebp+loopCounter_20_2]
movzx   eax, byte ptr [eax] ; eax = (ADDR(var_20) + count)[B0]
mov     edx, eax            ; edx = (ADDR(var_20) + count)[B0]
xor     dl, [ebp+nonce_B3] ; edx = nonce[B3] XOR ((ADDR(var_20) + count)[B0])
lea     eax, [ebp+newKey]
add     eax, [ebp+loopCounter_20_2]
mov     [eax], dl           ; store @ ADDR(var_20) + count) :
                            ; nonce[B3] XOR ((ADDR(var_20) + count)[B0])
add     [ebp+loopCounter_20_2], 1
```

Translates to:

```
# Loop 3
B3 = nonce[3]
for j in range(keyLen):
    newKey[j] = (newKey[j]^B3)&0xFF
```

Loop 4

Recall that between loops 3 and 4 the scramble function transforms the second nonce byte for use in loop 4.



```
loc_8049EED:
cmp     [ebp+loopCounter_20_3], 19
setle   al
test    al, al
jnz     short loc_8049E9F ; if ( counter <= 19) : LOOP
```

```
loc_8049E9F:                ; ebx = nonce[B1] >ROT> 4
movzx   ebx, [ebp+B1_TMP]
mov     eax, [ebp+loopCounter_20_3] ; eax = count
mov     edx, eax            ; edx = count
sar     edx, 31             ; edx = count >ROT> 31
shr     edx, 30             ; edx = (count >ROT> 31) >> 30
add     eax, edx            ; eax = count + ((count >ROT> 31) >> 30)
and     eax, 3              ; eax = (count + ((count >ROT> 31) >> 30)) AND 0x3
sub     eax, edx            ; eax = ((count + ((count >ROT> 31) >> 30)) AND 0x3) - (count >ROT> 31) >> 30
                            ; alias: MADNESS
mov     edx, ebx            ; edx = nonce[B1] >ROT> 4
mov     ecx, eax            ; ecx = MADNESS
sar     edx, cl             ; edx = (nonce[B1] >ROT> 4) >ROT> MADNESS[B0]
mov     eax, edx            ; eax = (nonce[B1] >ROT> 4) >ROT> MADNESS[B0]
and     eax, 1              ; eax = (nonce[B1] >ROT> 4) >ROT> MADNESS[B0] % 2
test    al, al
jz      short even          ; if (nonce[B1] >ROT> 4) >ROT> MADNESS[B0] is EVEN BRANCH
```

```
mov     eax,
lea     edx,
mov     ecx,
mov     [eax
mov     ecx,
mov     [eax
mov     ecx,
mov     [eax
mov     ecx,
mov     [eax
mov     edx,
mov     [eax
mov     ebx,
xor     ebx,
jz      short
```

```
odd:
lea     eax, [ebp+newKey]
add     eax, [ebp+loopCounter_20_3]
movzx   eax, byte ptr [eax] ; eax = newKey[count]
mov     [ebp+newKeyByte], al ; newKeyByte = newKey[count]
movzx   edx, [ebp+B1_JUMBLE] ; B1_JUMBLE = (nonce[B1] AND 0xF)[B0]
and     edx, 7              ; edx = (nonce[B1] AND 0xF)[B0] AND 7
movzx   eax, [ebp+newKeyByte] ; eax = newKey[count]
mov     ecx, edx            ; ecx = (nonce[B1] AND 0xF)[B0] AND 7
rol     al, cl              ; newKeyByte <ROL< (B1_JUMBLE AND 7)
                            ; alias: tmp_var
mov     edx, eax            ; edx = tmp_var
lea     eax, [ebp+newKey]
add     eax, [ebp+loopCounter_20_3]
mov     [eax], dl           ; newKey[count] = MENTAL[B0]
```

```
call    ___stac
```

```
even:
add     [ebp+loopCounter_20_3], 1
```

```
finish:
add     e
pop     e
pop     e
```

```
Translates to:
```

```
    # Byte transform
    B1 = nonce[1]
    B1_JUMBLE = (B1&0xF)
    B1_TMP = (B1 >> 4)

    # Loop 4
    for k in range(keyLen):
        if(B1_TMP >> (k&0x3)) % 2 != 0:
            newKey[k] = rol1(newKey[k],(B1_JUMBLE&0x7))
```

Now we put that all together into a python function.

```
def shuffleKey(inKey, inNonce):
        # Build byte array from digest.
        seedKey = [ord(x) for x in inKey]
        newKey  = [0]*len(seedKey)
        keyLen  = len(seedKey)

        # Loop 1
        B0 = nonce[0]
        for i in range(keyLen):
                B0_JUMBLE = (i+B0)%20
                newKey[B0_JUMBLE]=(seedKey[i])

        # Loop 2
        B2 = nonce[2]
        for count in range(8):
                B2_JUMBLE = (B2>>count)%2
                if B2_JUMBLE == 0:
              newKey[count] = (~(newKey[count]))

        # Loop 3
        B3 = nonce[3]
        for j in range(keyLen):
                newKey[j] = (newKey[j]^B3)&0xFF

        # Byte transform
        B1 = nonce[1]
        B1_JUMBLE = (B1&0xF)
        B1_TMP = (B1 >> 4)

        # Loop 4
        for k in range(keyLen):
                if(B1_TMP >> (k&0x3)) % 2 != 0:
                        newKey[k] = rolll(newKey[k],(B1_JUMBLE&0x7))

        # Convert char array back to string.
        key = "".join(map(chr,newKey))
        return key
```

The only element that didn’t easily translate into Python was the left roll instruction at 0x8049EDD. However a quick internet search will find you a function to perform this operation in Python.

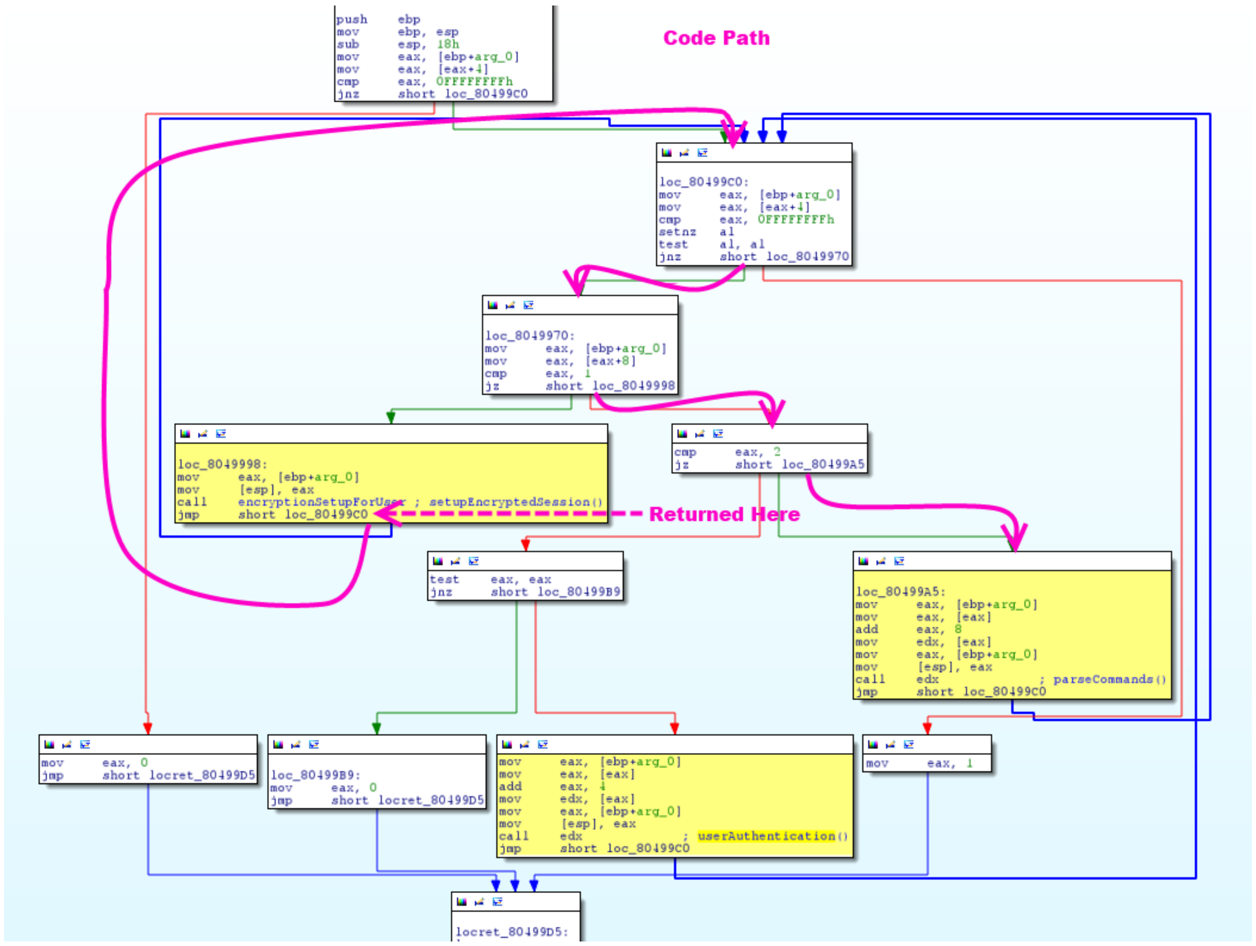
With a functional key scrambling algorithm in client.py we can calculate a seed key, scramble it and decrypt the challenge sent by the server. If we’re successful, our client will decrypt “Test12345678” then re-encrypt “Test12345678” and send it back to the server.

When run the next iteration of our client script and this time the local server prints the following output.

```
RE03: User 'LUK_MURPHY' Authenticated
RE03: Scrambling Key
RE03: User 'LUK_MURPHY' encryption setup
```

You'd think after the scramble function we'd be done… close, but these challenges were intended to be time sinks so there is a bit more to go.

Using GDB to step past the challenge/response comparison in encryption setup, we return to the loop which originally called the authentication and encryption setup functions. Thinking about this loop, it looks like this loop is a state machine with three states, unauthenticated, authenticated and encryption setup with each highlighted block below being the processing for each state.



```
push    ebp
mov     ebp, esp
sub     esp, 18h
mov     eax, [ebp+arg_0]
mov     eax, [eax+4]
cmp     eax, 0FFFFFFFFh
jnz     short loc_80499C0
```

```
loc_80499C0:
mov     eax, [ebp+arg_0]
mov     eax, [eax+4]
cmp     eax, 0FFFFFFFFh
setnz   al
test    al, al
jnz     short loc_8049970
```

```
loc_8049970:
mov     eax, [ebp+arg_0]
mov     eax, [eax+8]
cmp     eax, 1
jz      short loc_8049998
```

```
loc_8049998:
mov     eax, [ebp+arg_0]
mov     [esp], eax
call    encryptionSetupForUser ; setupEncryptedSession()
jmp     short loc_80499C0
```

```
cmp     eax, 2
jz      short loc_80499A5
```

```
test    eax, eax
jnz     short loc_80499B9
```

```
loc_80499A5:
mov     eax, [ebp+arg_0]
mov     eax, [eax]
add     eax, 8
mov     edx, [eax]
mov     eax, [ebp+arg_0]
mov     [esp], eax
call    edx                 ; parseCommands()
jmp     short loc_80499C0
```

```
mov     eax, 0
jmp     short locret_80499D5
```

```
loc_80499B9:
mov     eax, 0
jmp     short locret_80499D5
```

```
mov     eax, [ebp+arg_0]
mov     eax, [eax]
add     eax, 4
mov     edx, [eax]
mov     eax, [ebp+arg_0]
mov     [esp], eax
call    edx                 ; userAuthentication()
jmp     short loc_80499C0
```

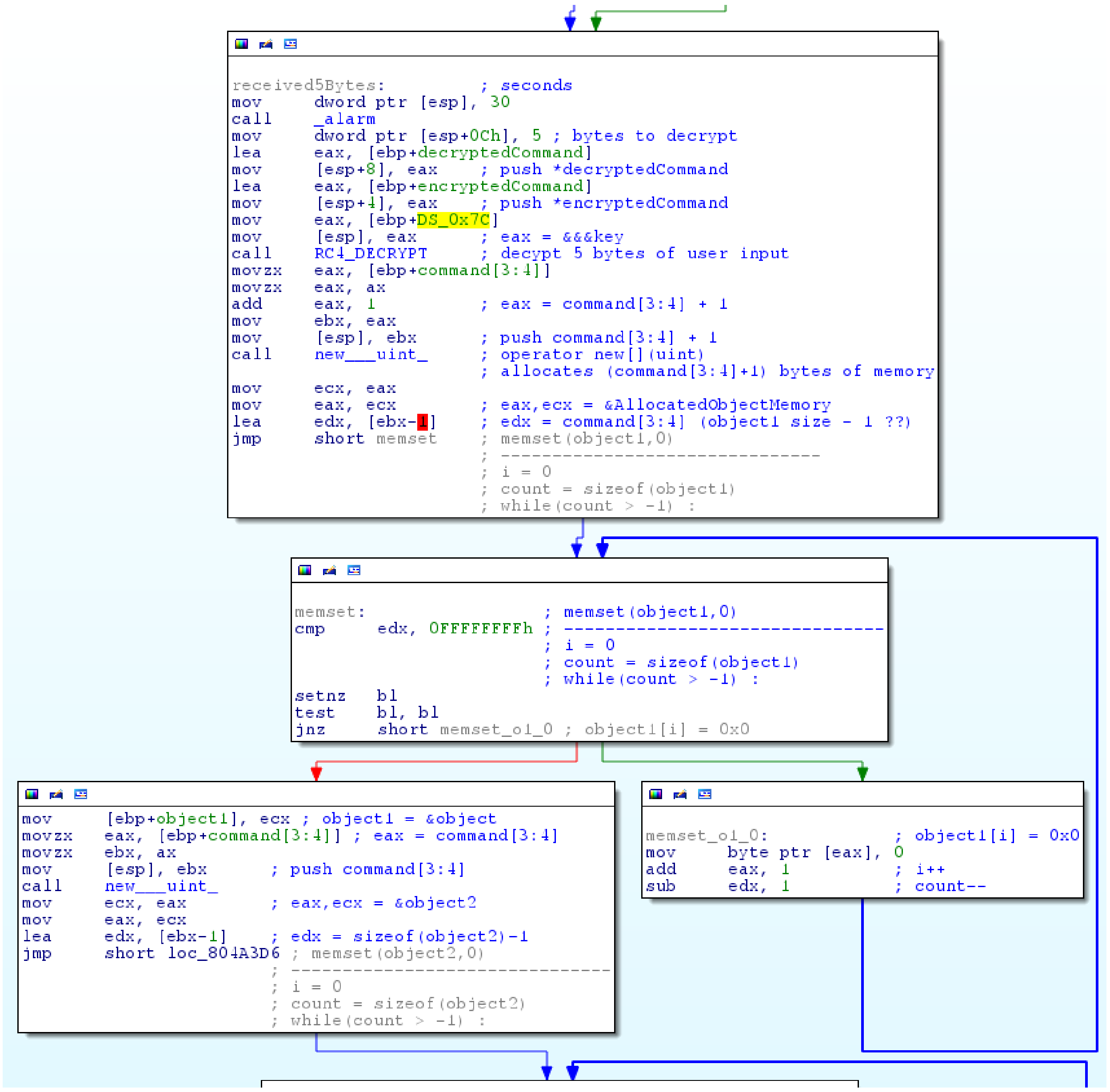
```
mov     eax, 1
```

```
locret_80499D5:
```

Now that we are in the encryption setup state, the server will now proceed to "call edx" at 0x80499B5 (as highlighted above). From the static listing in IDA we don't really know which function this is calling, so we use GDB to break at 0x80499B5 and show the edx register to determine the function being called.

```
(gdb) break *0x80499B5; c
(gdb) info reg
**** SNIP ****
edx            0x804a2f6     134521590
**** SNIP ****
```

The function at 0x804a2f6 begins by receiving 5 bytes of encrypted data from the client and decrypting it. The 4th and 5th bytes are used to provide the size parameter for a C++ object array constructor (operator new[](uint) @ 0x804A391). Once created, the memory allocated is zeroed.



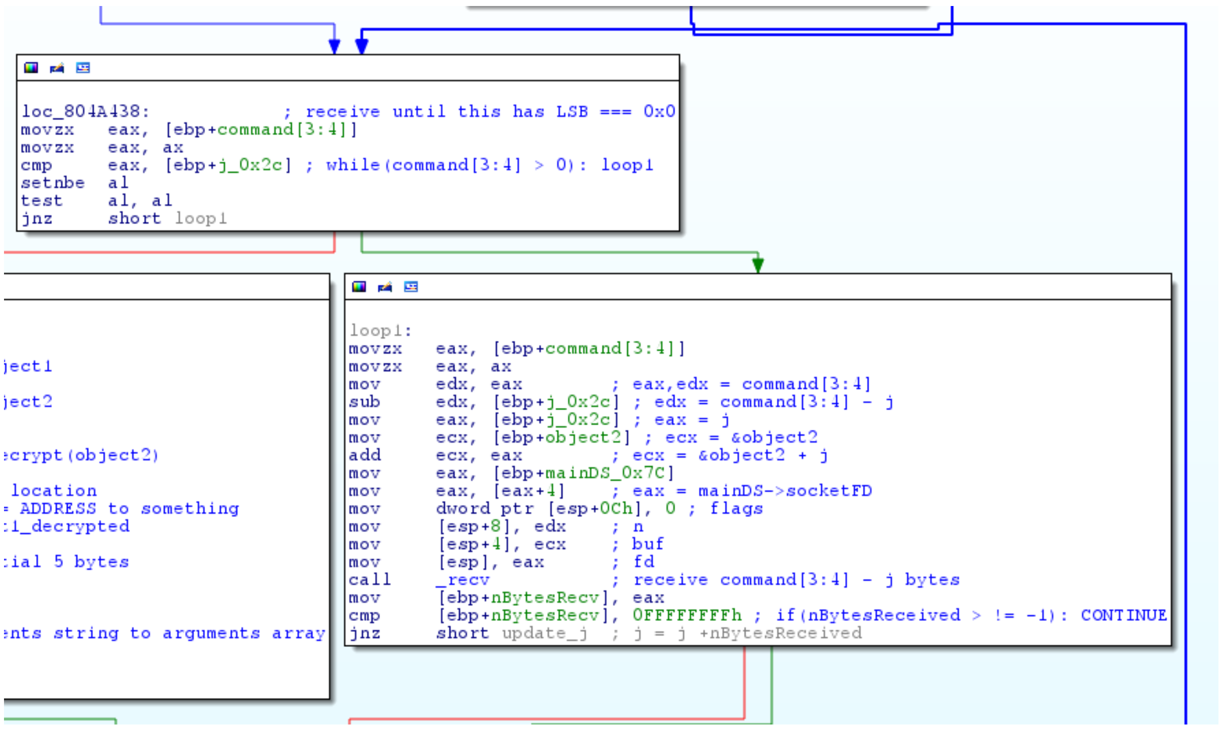
```
received5Bytes:              ; seconds
mov     dword ptr [esp], 30
call    _alarm
mov     dword ptr [esp+0Ch], 5 ; bytes to decrypt
lea     eax, [ebp+decryptedCommand]
mov     [esp+8], eax     ; push *decryptedCommand
lea     eax, [ebp+encryptedCommand]
mov     [esp+4], eax     ; push *encryptedCommand
mov     eax, [ebp+DS_0x7C]
mov     [esp], eax       ; eax = &&&key
call    RC4_DECRYPT      ; decypt 5 bytes of user input
movzx   eax, [ebp+command[3:4]]
movzx   eax, ax
add     eax, 1           ; eax = command[3:4] + 1
mov     ebx, eax
mov     [esp], ebx       ; push command[3:4] + 1
call    new___uint_      ; operator new[](uint)
                         ; allocates (command[3:4]+1) bytes of memory
mov     ecx, eax
mov     eax, ecx         ; eax,ecx = &AllocatedObjectMemory
lea     edx, [ebx-1]     ; edx = command[3:4] (object1 size - 1 ??)
jmp     short memset     ; memset(object1,0)
                         ; ------------------------------
                         ; i = 0
                         ; count = sizeof(object1)
                         ; while(count > -1) :
```

```
memset:                  ; memset(object1,0)
cmp     edx, 0FFFFFFFFh ; ------------------------------
                         ; i = 0
                         ; count = sizeof(object1)
                         ; while(count > -1) :
setnz   bl
test    bl, bl
jnz     short memset_o1_0 ; object1[i] = 0x0
```

```
mov     [ebp+object1], ecx ; object1 = &object
movzx   eax, [ebp+command[3:4]] ; eax = command[3:4]
movzx   ebx, ax
mov     [esp], ebx       ; push command[3:4]
call    new___uint_
mov     ecx, eax         ; eax,ecx = &object2
mov     eax, ecx
lea     edx, [ebx-1]     ; edx = sizeof(object2)-1
jmp     short loc_804A3D6 ; memset(object2,0)
                         ; ------------------------------
                         ; i = 0
                         ; count = sizeof(object2)
                         ; while(count > -1) :
```

```
memset_o1_0:             ; object1[i] = 0x0
mov     byte ptr [eax], 0
add     eax, 1           ; i++
sub     edx, 1           ; count--
```

With array 1 zeroed, the server creates and zeroes a second object array. Array 2 is then filled with encrypted bytes received from the client. When writing the client we need to ensure that the number of bytes sent to fill array 2 is the number as the value in the last 2 bytes of the initial 5 byte chunk we sent to the server (we don't yet know the function of the other 3 bytes). If we send too few bytes, the server will hang while waiting for more input.



```
loc_804A438:                ; receive until this has LSB === 0x0
movzx   eax, [ebp+command[3:4]]
movzx   eax, ax
cmp     eax, [ebp+j_0x2c] ; while(command[3:4] > 0): loop1
setnbe  al
test    al, al
jnz     short loop1
```

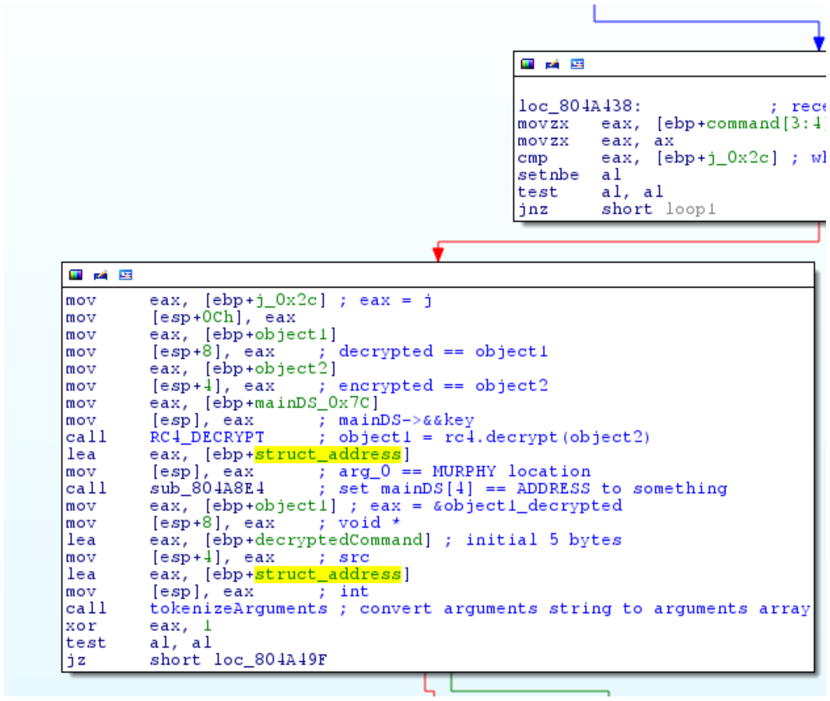
```
ject1

ject2


ecrypt(object2)

 location
= ADDRESS to something
:1_decrypted

:ial 5 bytes



ents string to arguments array
```

```
loop1:
movzx   eax, [ebp+command[3:4]]
movzx   eax, ax
mov     edx, eax          ; eax,edx = command[3:4]
sub     edx, [ebp+j_0x2c] ; edx = command[3:4] - j
mov     eax, [ebp+j_0x2c] ; eax = j
mov     ecx, [ebp+object2] ; ecx = &object2
add     ecx, eax          ; ecx = &object2 + j
mov     eax, [ebp+mainDS_0x7C]
mov     eax, [eax+4]      ; eax = mainDS->socketFD
mov     dword ptr [esp+0Ch], 0 ; flags
mov     [esp+8], edx      ; n
mov     [esp+4], ecx      ; buf
mov     [esp], eax        ; fd
call    _recv             ; receive command[3:4] - j bytes
mov     [ebp+nBytesRecv], eax
cmp     [ebp+nBytesRecv], 0FFFFFFFFh ; if(nBytesReceived > != -1): CONTINUE
jnz     short update_j  ; j = j +nBytesReceived
```

At 0x804A464 array 2 is decrypted, with the result being stored in object array 1. Following the call at 0x804A488 we see the server tokenize the decrypted array 1 (based on semi-colon delimiters).

Also note that this function initializes a third object and stores a pointer in a C++ data structure.

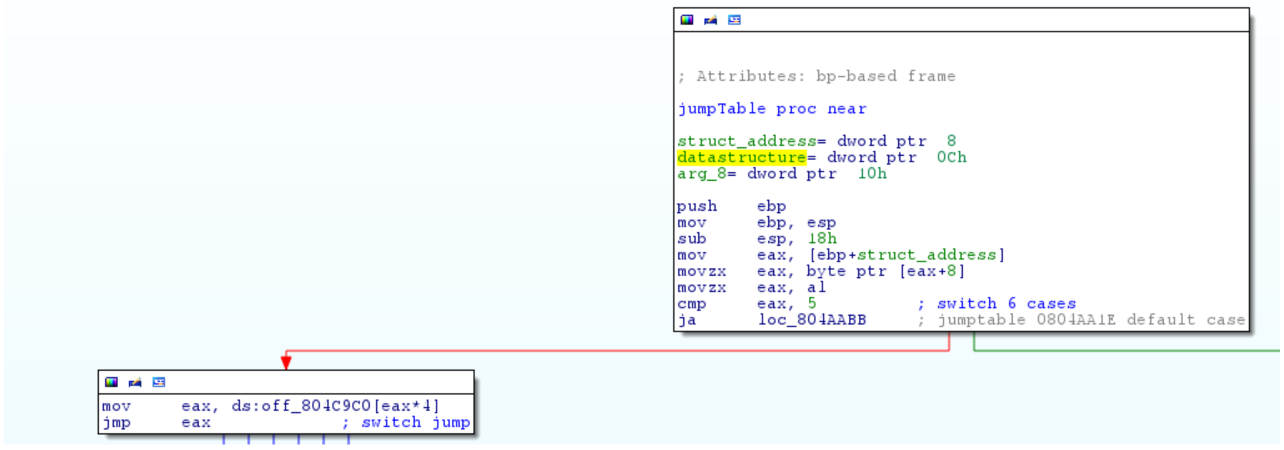
It's important to note the creation of object 3 because starting at 0x804A4A5 the server deletes array 1 and 2, and we haven't done anything useful with the data we sent to the server yet!



```
loc_804A438:                ; rece
movzx   eax, [ebp+command[3:4
movzx   eax, ax
cmp     eax, [ebp+j_0x2c] ; wl
setnbe  al
test    al, al
jnz     short loop1
```

```
mov     eax, [ebp+j_0x2c] ; eax = j
mov     [esp+0Ch], eax
mov     eax, [ebp+object1]
mov     [esp+8], eax     ; decrypted == object1
mov     eax, [ebp+object2]
mov     [esp+4], eax     ; encrypted == object2
mov     eax, [ebp+mainDS_0x7C]
mov     [esp], eax       ; mainDS->&&key
call    RC4_DECRYPT      ; object1 = rc4.decrypt(object2)
lea     eax, [ebp+struct_address]
mov     [esp], eax       ; arg_0 == MURPHY location
call    sub_804A8E4      ; set mainDS[4] == ADDRESS to something
mov     eax, [ebp+object1] ; eax = &object1_decrypted
mov     [esp+8], eax     ; void *
lea     eax, [ebp+decryptedCommand] ; initial 5 bytes
mov     [esp+4], eax     ; src
lea     eax, [ebp+struct_address]
mov     [esp], eax       ; int
call    tokenizeArguments ; convert arguments string to arguments array
xor     eax, 1
test    al, al
jz      short loc_804A49F
```

Soon after creating object 3 and tokenizing the client input, the server calls into a function containing a switch statement (call at 0x804A4E0 to function at 0x804A9FE).



```
; Attributes: bp-based frame

jumpTable proc near

struct_address= dword ptr  8
datastructure= dword ptr  0Ch
arg_8= dword ptr  10h

push    ebp
mov     ebp, esp
sub     esp, 18h
mov     eax, [ebp+struct_address]
movzx   eax, byte ptr [eax+8]
movzx   eax, al
cmp     eax, 5          ; switch 6 cases
ja      loc_804AABB     ; jumptable 0804AA1E default case
```

```
mov     eax, ds:off_804C9C0[eax*4]
jmp     eax             ; switch jump
```

Before reversing any particular branch in depth, lets obtain a cursory understanding of each case in the switch statement.

Based on IDA’s graph and the comparison at 0x804AA0E we have 6 cases to investigate, each case contains a call to some other function (cases 1-6 at 0x804AA34, 0x804AA52, 0x804AA7A, 0x804AA9C, 0x804AAB4, 0x804AA9D respectively). By some trial and error we learn that the switch case is controlled by the 5 byte sequence we previously sent to the server (we already used bytes 4 and 5 to determine the size of array 1 and 2).

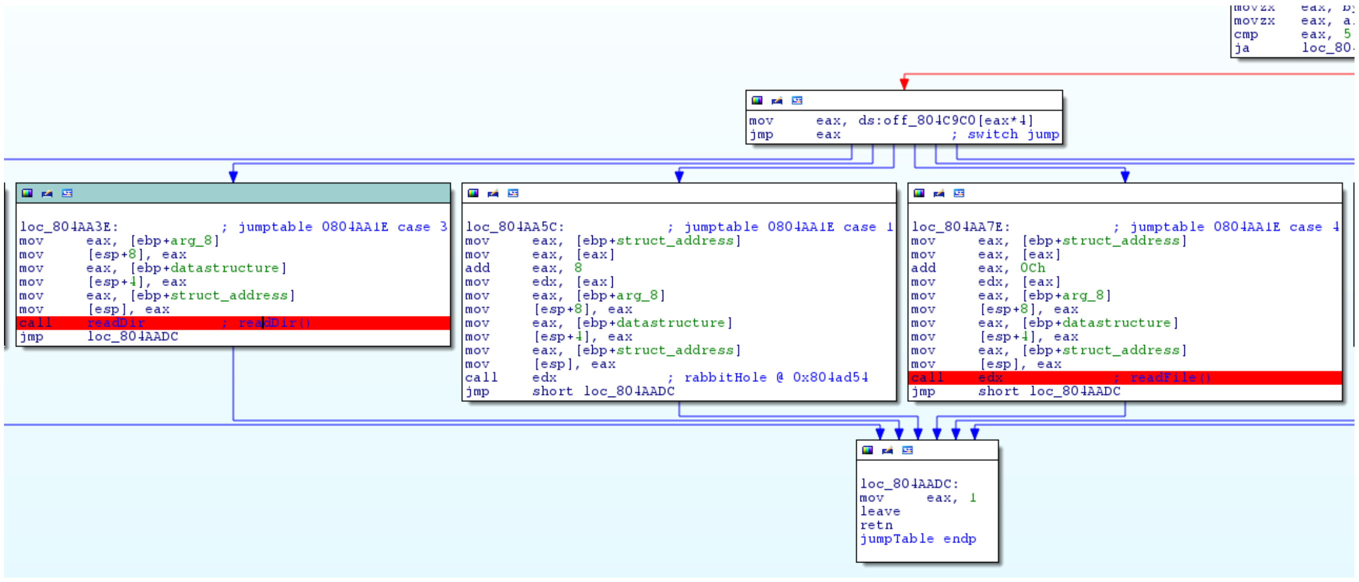
Inspecting registers in GDB shows that the first byte we sent corresponds to the switch case in this function.

We now know the purpose of 3 of the 5 bytes we sent to the server:

Switch case	????	????	arg size B0	arg size B1
B0	B1	B2	B3	B4

Some quick stepping through GDB reveals that the default case (0) exits the program.

Case 1 leads us down an enormous, multi-branch rabbit hole with no easily determinable function. It would be appropriate at this point to shake your fist and decry the use of C++ in this challenge.
Case 2 jumps back to the initial authentication method.
Case 3 contains some promising calls to string constructors, “opendir” and “readdir” functions.
Case 4 contains some further promising calls to string constructors and file-stream functions.
Case 5 … further rabbit holes.



```
movzx   eax, b
movzx   eax, a
cmp     eax, 5
ja      loc_80
```

```
mov     eax, ds:off_804C9C0[eax*4]
jmp     eax                     ; switch jump
```

```
loc_804AA3E:                    ; jumptable 0804AA1E case 3
mov     eax, [ebp+arg_8]
mov     [esp+8], eax
mov     eax, [ebp+datastructure]
mov     [esp+4], eax
mov     eax, [ebp+struct_address]
mov     [esp], eax
call    readDir                 ; readDir()
jmp     loc_804AADC
```

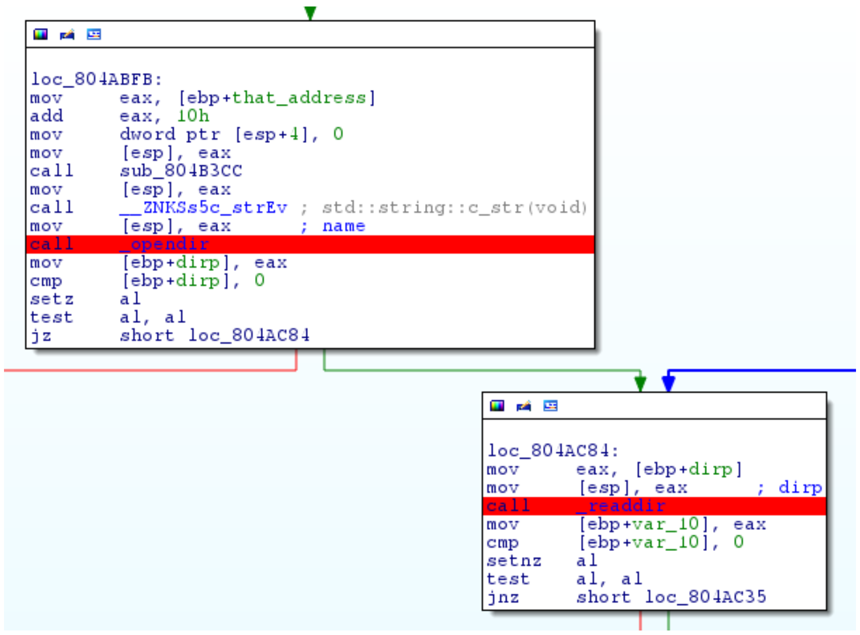
```
loc_804AA5C:                    ; jumptable 0804AA1E case 1
mov     eax, [ebp+struct_address]
mov     eax, [eax]
add     eax, 8
mov     edx, [eax]
mov     eax, [ebp+arg_8]
mov     [esp+8], eax
mov     eax, [ebp+datastructure]
mov     [esp+4], eax
mov     eax, [ebp+struct_address]
mov     [esp], eax
call    edx                     ; rabbitHole @ 0x804ad54
jmp     short loc_804AADC
```

```
loc_804AA7E:                    ; jumptable 0804AA1E case 4
mov     eax, [ebp+struct_address]
mov     eax, [eax]
add     eax, 0Ch
mov     edx, [eax]
mov     eax, [ebp+arg_8]
mov     [esp+8], eax
mov     eax, [ebp+datastructure]
mov     [esp+4], eax
mov     eax, [ebp+struct_address]
mov     [esp], eax
call    edx                     ; readFile()
jmp     short loc_804AADC
```

```
loc_804AADC:
mov     eax, 1
leave
retn
jumpTable endp
```

Lets focus on the function that appears to read from directories.

Running “man opendir” in a terminal reveals that opendir takes a single string as a parameter (the name of the dir to open). In GDB, We’ll break on the call to _opendir (0x804AC1C) to check what the string is. If we’re lucky it’ll just be the encrypted arguments we sent to the server.



```
loc_804ABFB:
mov     eax, [ebp+that_address]
add     eax, 10h
mov     dword ptr [esp+4], 0
mov     [esp], eax
call    sub_804B3CC
mov     [esp], eax
call    __ZNKSs5c_strEv ; std::string::c_str(void)
mov     [esp], eax       ; name
call    _opendir
mov     [ebp+dirp], eax
cmp     [ebp+dirp], 0
setz    al
test    al, al
jz      short loc_804AC84
```

```
loc_804AC84:
mov     eax, [ebp+dirp]
mov     [esp], eax       ; dirp
call    _readdir
mov     [ebp+var_10], eax
cmp     [ebp+var_10], 0
setnz   al
test    al, al
jnz     short loc_804AC35
```

Because we're running the server locally in a chroot we'll first create a test folder in the chroot directory.

```
#> mkdir /chroots/2013/testdir && touch /chroots/2013/testdir/test.txt
```

Now we'll need to set up an appropriate 5 byte packet.

0x03	0xFF	0xFF	0x08	0x00
B0	B1	B2	B3	B4

With this packet encrypted and sent to the server, we encrypt and send "`testdir/`" as the argument. With a breakpoint in place at 0x804AC1C we drop into GDB and check the argument to _opendir.

```
(gdb) x/i $pc
=> 0x804ac1c: call   0x80495b0 <opendir@plt>
(gdb) x/wx $esp
0xffacef50:   0x08ec806c
(gdb) x/s 0x08ec806c
0x8ec806c:    "testdir"
```

We now control the input to _opendir, but what happens once the dir is opened. Lets leap ahead here and assume that if this function opens and reads a dir then something will be sent back to the client (there's one final call to send at 0x804A593 that hasn't run yet). After sending the encrypted argument "`testdir/`" we attempt to receive data from the server, we receive a null. We try a second time and receive a blob of data! We assume that the null was a response code or perhaps a trailing/leading byte that we didn't deal with properly somewhere. Decrypting the second response in our client yields:

```
#> client.py
test.txt;..;.��
```

Great success, we're close. Assuming the challenge server contains the flag in a directory we would have just found the flag. The question now is how to read it. Note that the server is producing semi-colon delimited output and earlier we saw it tokenizing input with semi-colon delimiters. We could clean up the server output to only display printable characters but we don't need to bother just now.

If the server's protocol is consistent for reading directories and files we should be able to send the file path as an argument to switch case 4 and simply decrypt the server output. We send it to case number 4 because we recall that it has functions relating to file creation and reading.

This time we send the following 5 byte packet:

0x04	0xFF	0xFF	0x10	0x00
B0	B1	B2	B3	B4

Followed by the encrypted argument "`testdir/flag.txt`"

```
#> echo "Test Flag!!!" > /chroots/2013/testdir/flag.txt
#> client.py
��Test Flag!!!
```

Finally, we have confirmed we have the commands we need to read the flag from the server. We modify our client script to target the game server.

We run the script to get a directory listing of / which shows us the file thisistheflag.txt and we then run the script again to display the content of thisistheflag.txt giving us the flag for the final reverse engineering challenge. **HandleLoganNepenthes415**

```
#> client.py ls /
lib;.;dev;bin;..;etc;thisistheflag.txt��

#> client.py cat /thisistheflag.txt
��HandleLoganNepenthes415
```