No cON Name Facebook CTF Quals 2013

Dragon Sector write-ups

Team information

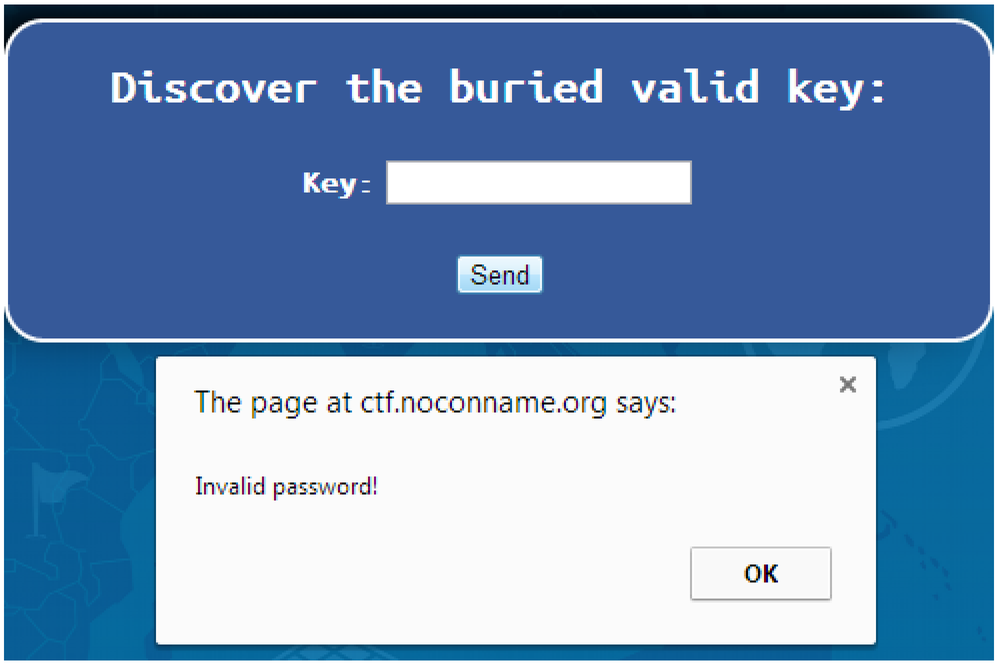
Please use the following e-mail address to contact our team's captain: gynvael@coldwind.pl

If qualified, we will provide a list of up to four people from our team to attend the finals, based on their availability.

Access Level 1 Solution

Flag: 23f8d1cea8d60c5816700892284809a94bd00fe7347645b96a99559749c7b7b8
Solved by: valis

For this challenge we are given a simple web page with "Key" text input. On submitting the form we get an "Invalid password!" Javascript popup.



The page at ctf.noconname.org says:

Invalid password!

The form is submitted to *login.php* script on the server, but it just redirects back to the first page.

We can assume that if we pass the JS validation we'll get something more (a flag perhaps?) from the server-side.

Looking at the page source a reference to the *crypto.js* file stands out. It contains obfuscated code, but it's really easy to deobfuscate - just replace 'eval' with 'alert' and whole code will pop up in the message box.

Here's that code:

```
function simpleHash(str)
{
	var i,hash=0;
	for(i=0;i<str.length;i++)
	{
		hash+=(str[i].charCodeAt()*(i+1))
	}
	return Math.abs(hash)%31337
}

function ascii_one(foo)
{
  foo=foo.charAt(0);
  var i;
  for(i=0;i<256;++i)
  {
        var hex_i=i.toString(16);

        if(hex_i.length==1)
        hex_i="0"+hex_i;

        hex_i="%"+hex_i;
        hex_i=unescape(hex_i);
        if(hex_i==foo)
        break
  }
  return i
}

function numerical_value(str)
{
  var i,a=0,b;
  for(i=0;i<str.length;++i)
  {
        b=ascii_one(str.charAt(i));
        a+=b*(i+1)
  }
  return a
}

function encrypt(form)
{
  var res;
  res=numerical_value(form.password.value);
  res=res*(3+1+3+3+7);
  res=res>>>6;
  res=res/4;
  res=res^4153;
  if(res!=0)
  {
        alert('Invalid password!')
  }else{
        alert('Correct password :)')
  }
  form.key.value=numerical_value(form.password.value);
  form.verification.value="yes"+simpleHash(form.password.value);
  return false;
}
```

As we can see, in the **encrypt()** function, the password is transformed into a number by adding its letters ASCII codes. The result, after some additional math operations, is compared against zero. So we have to find a string that will satisfy this condition (there are many such strings, but we just need one of them).

Instead of analyzing the math and trying to reverse it, it can be done much faster using brute force approach:

```
function check(res)
{
  res=res*(3+1+3+3+7);
  res=res>>>6;
  res=res/4;
  res=res^4153;
  if(res!=0)
  {
       return false;
       }
       return true;
}

for (i = 1; i < 99999999; i++)
{
       if (check(i)) {
              alert(i);
              break;
       }
}
```

The script quickly found **62540** as the number that satisfies conditions in **encrypt()**.
Now we just have to find a string that given to **numerical_value()** will produce the same number.

Again, we opted for a quick, low-tech, solution and just tried calling **numerical_value()** on different length strings composed of letter '**z**' to get a value close to **62540**, and then adjusted the string manually to get the exact number (it is easy as each letter adds to the number its ASCII code in decimal multiplied by it's position in the string).
Final result: **gzzzzzzzzzzzzzzzzzzzzzzzzzzzzwAz**

After entering this code into the challenge page form we got this:

Congrats! you passed the level! Here is the key:
23f8d1cea8d60c5816700892284809a94bd00fe7347645b96a99559749c7b7b8

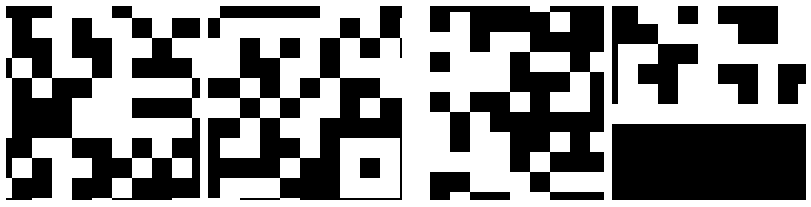
Access Level 2 Solution

Flag: 788f5ff85d370646d4caa9af0a103b338dbe4c4bb9ccbd816b585c69de96d9da
Solved by: valis

In this one we get a *level.apk* file - an Android app.

Extracting it (ordinary *unzip* will suffice) shows a number of interesting files:

- *classes.dex* - contains the code
- *res/raw/*.png* - 16 PNG files named *a.png*, *b.png* … *p.png* - each image is 97x97 pixels and contains a part of QR code - so it's a 4x4 puzzle. By that point we could probably just print out the images and do the puzzle by hand, but let's look at the code.



Using *dex2jar* utility on *classes.dex* produces a *.jar* file with java classes, and opening those in *jd-gui java decompiler* shows clean Java code.

Interesting code from MainActivity.java:

```
  public void yaaaay()
  {
       ArrayList localArrayList = new ArrayList();
       Bitmap localBitmap1 = BitmapFactory.decodeResource(getResources(), 2130968581);
       Bitmap localBitmap2 = BitmapFactory.decodeResource(getResources(), 2130968589);
       Bitmap localBitmap3 = BitmapFactory.decodeResource(getResources(), 2130968588);
       Bitmap localBitmap4 = BitmapFactory.decodeResource(getResources(), 2130968580);
       localArrayList.add(localBitmap1);
       localArrayList.add(localBitmap2);
       localArrayList.add(localBitmap3);
       localArrayList.add(localBitmap4);
       Bitmap localBitmap5 = BitmapFactory.decodeResource(getResources(), 2130968582);
       Bitmap localBitmap6 = BitmapFactory.decodeResource(getResources(), 2130968587);
       Bitmap localBitmap7 = BitmapFactory.decodeResource(getResources(), 2130968578);
       Bitmap localBitmap8 = BitmapFactory.decodeResource(getResources(),2130968591);
       localArrayList.add(localBitmap5);
       localArrayList.add(localBitmap6);
       localArrayList.add(localBitmap7);
       localArrayList.add(localBitmap8);
… up to localBitmap15 …

       int i = new Random().nextInt(localArrayList.size());
       makeMeHappyAgain(makeMeHappy(localBitmap1, localBitmap2, localBitmap3,
localBitmap4), makeMeHappy(localBitmap5, localBitmap6, localBitmap7, localBitmap8),
makeMeHappy(localBitmap9, localBitmap10, localBitmap11, localBitmap12),
makeMeHappy(localBitmap13, localBitmap14, localBitmap15, localBitmap16));
       Bitmap localBitmap17 = (Bitmap)localArrayList.get(i);
       this.secret.setImageBitmap(localBitmap17);
  }
  public Bitmap makeMeHappy(Bitmap paramBitmap1, Bitmap paramBitmap2, Bitmap
paramBitmap3, Bitmap paramBitmap4)
  {
       Bitmap localBitmap = Bitmap.createBitmap(paramBitmap1.getWidth() +
paramBitmap2.getWidth() + paramBitmap3.getWidth() + paramBitmap4.getWidth(),
paramBitmap1.getHeight(), Bitmap.Config.ARGB_8888);
       Canvas localCanvas = new Canvas(localBitmap);
       localCanvas.drawBitmap(paramBitmap1, 0.0F, 0.0F, null);
       localCanvas.drawBitmap(paramBitmap2, paramBitmap1.getWidth(), 0.0F, null);
       localCanvas.drawBitmap(paramBitmap3, paramBitmap1.getWidth() +
paramBitmap2.getWidth(), 0.0F, null);
       localCanvas.drawBitmap(paramBitmap4, paramBitmap1.getWidth() +
paramBitmap2.getWidth() + paramBitmap3.getWidth(), 0.0F, null);
       return localBitmap;
  }

  public Bitmap makeMeHappyAgain(Bitmap paramBitmap1, Bitmap paramBitmap2, Bitmap
paramBitmap3, Bitmap paramBitmap4)
  {
       Bitmap localBitmap = Bitmap.createBitmap(paramBitmap1.getWidth(),
paramBitmap1.getHeight() + paramBitmap2.getHeight() + paramBitmap3.getHeight() +
paramBitmap4.getHeight(), Bitmap.Config.ARGB_8888);
       Canvas localCanvas = new Canvas(localBitmap);
       localCanvas.drawBitmap(paramBitmap1, 0.0F, 0.0F, null);
       localCanvas.drawBitmap(paramBitmap2, 0.0F, paramBitmap1.getHeight(), null);
       localCanvas.drawBitmap(paramBitmap3, 0.0F, paramBitmap1.getHeight() +
paramBitmap2.getHeight(), null);
       localCanvas.drawBitmap(paramBitmap4, 0.0F, paramBitmap1.getHeight() +
paramBitmap2.getHeight() + paramBitmap3.getHeight(), null);
       return localBitmap;
  }
```

As we can see the **yaaay()** shows a random puzzle piece, but there are calls to methods **makeMeHappy()** and **makeMeHappyAgain()** that merge the pieces into one image - the only problem is that their result is not used.

There are two basic approaches to solve this. One is to recompile the code modified to do the following:

```
this.secret.setImageBitmap(makeMeHappyAgain(makeMeHappy(...)));
```

This would show the full image in app instead of just one random piece.

However we didn't have a working Android SDK setup with us, so we've opted for the second method - analyze the code and reproduce it externally.

It turned out to be quite simple - **makeMeHappy()** merges 4 arguments horizontally and **makeMeHappyAgain()** vertically. All we need is a mapping between resource numbers and filenames - those are listed in R.java:

```
  public static final class raw
  {
        public static final int a = 2130968576;
        public static final int b = 2130968577;
        public static final int c = 2130968578;
        public static final int d = 2130968579;
        ...
```

Here’s a python script we used to assemble the puzzle:

```
#!/usr/bin/python
import Image

out = Image.new("RGB", (4*97,4*97))

out.paste(Image.open("f.png"), (0,0))
out.paste(Image.open("m.png"), (97,0))
out.paste(Image.open("l.png"), (2*97,0))
out.paste(Image.open("e.png"), (3*97,0))

out.paste(Image.open("c.png"), (0,97))
out.paste(Image.open("k.png"), (97,97))
out.paste(Image.open("g.png"), (2*97,97))
out.paste(Image.open("o.png"), (3*97,97))

out.paste(Image.open("p.png"), (0,2*97))
out.paste(Image.open("a.png"), (97,2*97))
out.paste(Image.open("b.png"), (2*97,2*97))
out.paste(Image.open("n.png"), (3*97,2*97))

out.paste(Image.open("h.png"), (0,3*97))
out.paste(Image.open("d.png"), (97,3*97))
out.paste(Image.open("j.png"), (2*97,3*97))
out.paste(Image.open("i.png"), (3*97,3*97))

out.save("qr.png")
```

And the final QR itself:

Scanning *qr.png* with *zbarimg* gives us the flag:

```
$ zbarimg qr.png
QR-Code:788f5ff85d370646d4caa9af0a103b338dbe4c4bb9ccbd816b585c69de96d9da

scanned 1 barcode symbols from 1 images in 0.01 seconds
```

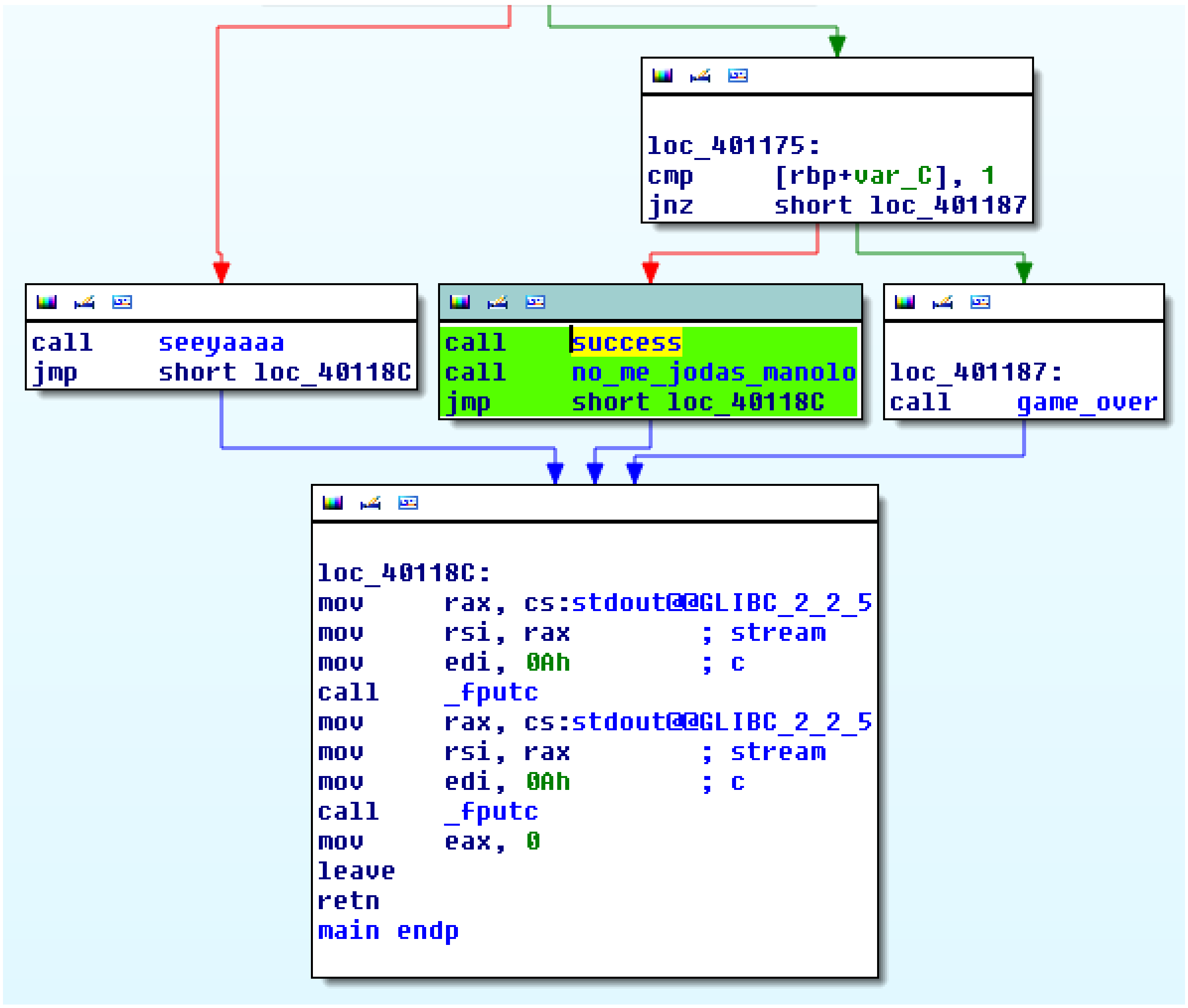
One funny detail: puzzles assembled exactly as in the app code didn't get us a valid code **-** two pieces in second row had to be switched. Not sure if it was intended by authors.

Access Level 3 Solution

Flag: 9e0d399e83e7c50c615361506a294eca22dc49bfddd90eb7a831e90e9e1bf2fb
Solved by: Gynvael Coldwind

You are provided with a *level.elf* file, which is a 64-bit GNU/Linux, unstripped, dynamically linked executable.
Loading the file into IDA Pro you see an unobfuscated main function, which is quite short and easy to read.

Looking for an easy solution we've immediately spotted the following block of code:



```
loc_401175:
cmp     [rbp+var_C], 1
jnz     short loc_401187
```

```
call    seeyaaaa
jmp     short loc_40118C
```

```
call    success
call    no_me_jodas_manolo
jmp     short loc_40118C
```

```
loc_401187:
call    game_over
```

```
loc_40118C:
mov     rax, cs:stdout@@GLIBC_2_2_5
mov     rsi, rax        ; stream
mov     edi, 0Ah        ; c
call    _fputc
mov     rax, cs:stdout@@GLIBC_2_2_5
mov     rsi, rax        ; stream
mov     edi, 0Ah        ; c
call    _fputc
mov     eax, 0
leave
retn
main endp
```

Looking into the *success* functions shows that it prints, single character at a time, the message:

```
|  -> Congratulations! The key is:
```

This leads to an obvious conclusion that *no_me_jodas_manolo* function will print the final flag.

Since this function takes no arguments, it seemed to be enough to call if from the context of the main function - this can be done e.g. using GDB:

```
> gdb -q ./level.elf
Reading symbols from level.elf...(no debugging symbols found)...done.
(gdb) break *0x40118c ← address of the beginning of the last block in main
Breakpoint 1 at 0x40118c
(gdb) r
Starting program: level.elf
```

```
|  >  Type to win, only what I want to read...
|  >
|
|  -> I DON'T THINK SO
Breakpoint 1, 0x000000000040118c in main ()
(gdb) set $rip=0x040117B ← address of the "call success" instruction
(gdb) disable 1
(gdb) c
Continuing.
|
|  -> Congratulations! The key is:
|  9e0d399e83e7c50c615361506a294eca22dc49bfddd90eb7a831e90e9e1bf2fb

[Inferior 1 (process 2977) exited normally]
```

BONUS: Finding the proper password is simple as well. The main loop contains the following code block:

```
loc_4010F3:
call    getch
movsx   eax, al
mov     [rbp+var_4], eax
mov     eax, [rbp+var_8]
cdqe
mov     eax, dword ptr facebookctf_rocks[rax*4]
cmp     eax, [rbp+var_4]
jnz     short loc_40111E
```

Looking at facebookctf_rocks array in hex view shows:

```
00000000006033A0  20 00 00 00 53 00 00 00  55 00 00 00 52 00 00 00   ...S...U...R...
00000000006033B0  50 00 00 00 52 00 00 00  49 00 00 00 53 00 00 00  P...R...I...S...
00000000006033C0  45 00 00 00 21 00 00 00                           E...!...
```

The proper password is (excluding the quotes, including the initial space): " `SURPRISE!`".