sets the label text to the result. If we save out the resource that is being manipulated, we can play around with it. The Python code in the right side of Figure 1 is the decode button re-implement to help us figure out what we are dealing with. When this Python code is run, the following is printed out showing the solution to Challenge 1 as “Text 1” in the output.

```
Text 1: 3rmahg3rd.b0b.d0ge@flare-on.com
Text 2: r3amghr3.d0b.b0degf@alero-.noc m
Text 3: ¶Uძ☺ᒧ¶UH☻V◆H◆V☻♥☺ &
```

Figure 2: Challenge 1 result

Challenge 2: Javascrap

The next challenge (to the bane of some of our players) is not a Windows PE file. Instead we have a version of the website www.flare-on.com. There has to be something special about this version of the website. So we look at page source of *home.html*. If we compare this version with the live challenge website, one line in particular stands out:

<?php include "img/flare-on.png" ?>

Why would a PNG image be loaded as a PHP script? When we open this image with an image viewer, the banner comes up so it is definitely an image. Since we know that the image is being loaded as a PHP script, we search for *php* inside of the image file and find the following:

19C0 AE 42 60 82 3C 3F 70 68 70 20 24 74 65 72 6D 73 .B`.<?php $terms

19D0 3D 61 72 72 61 79 28 22 4D 22 2C 20 22 5A 22 2C =array("M", "Z",

19E0 20 22 5D 22 2C 20 22 70 22 2C 20 22 5C 5C 22 2C "]", "p", "\\",

19F0 20 22 77 22 2C 20 22 66 22 2C 20 22 31 22 2C 20 "w", "f", "1",

1A00 22 76 22 2C 20 22 3C 22 2C 20 22 61 22 2C 20 22 "v", "<", "a", "

1A10 5D 22 2C 20 22 7A 22 2C 20 22 20 22 2C 20 22 73 Q", "z", " ", "s

Pulling this out of the image leaves us with:

```
<?php
 $terms=array("M", "Z", "]", "p", "\\", "w", "f", "1", "v",...
 $order=array(59, 71, 73, 13, 35, 10, 20, 81, 76, 10, 28, 63, 12,...
 $do_me="";
 for($i=0;$i<count($order);$i++)
 {
 $do_me=$do_me.$terms[$order[$i]];
}
 eval($do_me);
 ?>
```

So we have an array of characters, and then an array of indexes into the array of characters being used to build a string that gets evaluated. If we replace that *eval* with a call to *echo*, we can see the following strings in the display:

```
$_= 'aWYoaXNzZXQoJF9QT1NUWyJcOTdcNDlcNDlcNjhceDRGXDg0XDExNlx4Njh...
$__='JGNvZGU9YmFzZTY0X2RlY29kZSgkXyk7ZXZhbCgkY29kZSk7';
$___="\x62\141\x73\145\x36\64\x5f\144\x65\143\x6f\144\x65";
eval($___($__));
```

This reveals more obfuscation! Applying the same trick again results in the following lines of code:

```
$code=base64_decode($_);
```

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
```