```
/*
Step 0 would be the ability to identify minified code (i.e. no comments or
whitespaces). This should be very easy but will get them familiar with the
environment and tools.
*/

function hello() {
alert("cruel world");
}

// transformed into
function hello(){alert("cruel world")}

/*
Step 1 would be to identify obfuscated code. This is not entirely decidable
but as the trivial example below shows it is pretty obvious to now
*/
function hello() {
alert("cruel world");
}

// transformed into
eval(function(p,a,c,k,e,d){e=function(c){return
c};if(!''.replace(/^/,String)){while(c--){d[c]=k[c]||c}k=[function(e){return
d[e]}];e=function(){return'\\w+'};c=1};while(c--){if(k[c]){p=p.replace(new
RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('0 1(){2("3
4")}',5,5,'function|hello|alert|cruel|world'.split('|'),0,{}))

// and another example
var _0xb3c2=["\x63\x72\x75\x65\x6C\x20\x77\x6F\x72\x6C\x64"];function
hello(){alert(_0xb3c2[0]);} ;

// and this one is really sick. BTW, to me this would look like a malicious
code (which is a false positive of my approach)
var
Ol0='7kSKlBXYjNXZfhSZwF2YzVmb1hSZ0lmc35CduVWb1N2bktTKwwG
MfhCZslGaDRmblBHch5yTxkkC70FMblyJkFWZodCKl1WYOdWYUlnQzR
nbl1WZsVEdldmL05WZtV3YvRGI9AyTxkEIyFmdKsTKMJVVuQnbl1Wdj9
GZoQnbl52bw12bDlkUVVGZvNmbltyJ9wmc1ZyJrkiclJnclZWZy5CduVWb1
N2bkhCduVmbvBXbvNUSSVVZk92YuV2Kn0jZlJnJnsyJr9WPjJ3c0V2Z/8S
```

bvNmLy9GdhN2c1ZmYvxWb0hmLpBXYv8iOwRHdodCI9AyYyNnLwwGMf
pwOpcCdwlmcjN3JoQnbI1WZsVUZ0FWZyNmL05WZtV3YvRGI9ACMsBz
XgIXY2tzJFNTJ0BXayN2cvM0MlEEMlQ0NlEEMlI0MlkjMlIjMlQGby92dwIT
JsVWdyNmMyUCOyUCdyVGbhBjMlAjMlEEMlI0NlAjMlkjMlgjMl8GbsVGaw
ITJu9Wa0Nmb1ZWRzUCdwlmcjN3QzUyJ9UGchN2cl9FIyFmd';var
_0x84de=["ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuv
wxyz0123456789+/=","","charAt","indexOf","fromCharCode","length"];functi
on _1I1(data){var lOIlOI=_0x84de[0];var
o1,o2,o3,h1,h2,h3,h4,bits,i=0,enc=_0x84de[1];do{h1=lOIlOI[_0x84de[3]](d
ata[_0x84de[2]](i++));h2=lOIlOI[_0x84de[3]](data[_0x84de[2]](i++));h3=lOIl
OI[_0x84de[3]](data[_0x84de[2]](i++));h4=lOIlOI[_0x84de[3]](data[_0x84de
[2]](i++));bits=h1<<18|h2<<12|h3<<6|h4;o1=bits>>16&0xff;o2=bits>>8&0xf
f;o3=bits&0xff;if(h3==64){enc+=String[_0x84de[4]](o1);} else
{if(h4==64){enc+=String[_0x84de[4]](o1,o2);} else
{enc+=String[_0x84de[4]](o1,o2,o3);} ;} ;} while(i<data[_0x84de[5]]);;return
enc;} ;function lOI(string){var ret=_0x84de[1],i=0;for(i=string[_0x84de[5]]-
1;i>=0;i--){ret+=string[_0x84de[2]](i);} ;return ret;} ;eval(_1I1(lOI(Ol0)));

/*
Step 2 would be to provide some metric of the "maliciousness" of the code. I don't mind what the scale is, but it should classify the 'hello' function as benign and the example below as malicious. It would be interesting to check it with common JS libraries. I would provide the students 3-4 examples and keep an additional 2-3 to ourselves so we can check their implementation on unknown input files.

https://github.com/douglascrockford/JSON-js/blob/master/cycle.js
https://github.com/douglascrockford/JSON-js/blob/master/json2.js
https://github.com/evanvosberg/crypto-js/blob/master/src/md5.js

Option I - identify uses of very long strings or very long arrays. You should handle the option to disguise this via:
- string concatenation
- array concatenation or element addition

Option II - identify uses of strings or arrays of ints whose content "feels" like machine code. For example, you can search for sequences of bytes that translate into legitimate x86 instructions. This is easily bypassed and I'm pretty sure is not an easy computational problem (probably a hard one) but we can ignore it for now.

*/

/*
General tools:
UglifyJS - https://github.com/mishoo/UglifyJS
Esprima - http://esprima.org/
*/