Author: Kirils Solovjovs
Team: possiblesecurity.com
Challenge: Reversing / Easy / "JohnnyBoy"

JOHNNY BOY

This looks retro but somewhat futuristic! I always lose though, can you win? My friends solved this without IDA! https://youtu.be/Dem8Fq6hkAw

Files: chall.ino.hex

It is usually best to use systematic approach to be sure to cover all your bases whatever the task.

Let's try to enumerate everything we have from the description:
A) "I always lose though, can you win?" hints that **winning the game should display the flag**;
B) YouTube video provides us with valuable information about the challange. The game is apparently compiled for **ARDUBOY platform**[1]; information available online indicates that ARDUBOY uses **1-bit (black and white only) display**; video also allows us to peek at some ingame strings, e.g. "Press a button to start!";
C) Description saying "without IDA" (IDA is Interactive Disassembler[2]) hints that no sophisticated static or dynamic analysis tools are needed to solve the challenge;
D) The file *chall.ino.hex* is ASCII text, with CRLF line terminators. The lines start with ":", most of them followed by exactly 42 alphanumeric characters representing **hexadecimal data** (since each character is either a digit 0-9 or a letter A-F; and also indicated by the file extension, duh!).

There are two ways to proceed form here.

If ARDUBOY hardware or emulator is available, one could load the binary into ARDUBOY and try to win the game. This might turn impossible, so static analysis could be used to identify and patch winning criteria in the code. This however would require a disassembler, therefore we will be using another approach.

1a) Let's use a text editor to take a look at *chall.ino.hex* provided. There are some lines with mostly zeros:
```
:10010000000000000000000000000000000000EF
:10011000000000000000000000000000000000DF
:10012000000000000000000000000000000000CF
```

We can deduce that characters 2,3,4,5,6 of each line probably indicate memory address, line number, or some other sequential index. Quick look at the whole file assures us that lines are mostly in sequence, so we can safely ignore characters 1,2,3,4,5,6 (with character 1 being ":"). Additionally all lines contain zeros in positions 7,8,9, so we also ignore those.

Last two printable characters appear to be a checksum, since they are always non-zero even in a line of mostly zeros (see above). Let's remove those too. This can be nicely done in a text editor by replacing ^…..… with nothing and ..$ with nothing. (Those are regular expressions matching any 6 characters at the start of the line and any 2 characters at the end of the line respectively.) Afterwards we can remove any newlines (if there still are any) and copy the result into a hex editor, e.g. bless[3]. Saving this to a file (e.g. *chall.ino.bin*) will gives us a binary file that we can search for strings.

---

1   https://arduboy.com/
2   https://en.wikipedia.org/wiki/Interactive_Disassembler
3   https://github.com/bwrsandman/Bless

1b) Alternatively we can also perform this step using command line tools in Linux: `cat chall.ino.hex | sed 's/^.........//;s/...$//' | xxd -r -p > chall.ino.bin` (note the use of ...$ instead of ..$; this is because command line tools on Linux work with LF line terminators rather than CR LF as in this file, so we need our expression to match CR too).
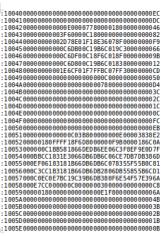
2) Remember that we are looking for **the flag displayed upon winning**. Therefore we look for strings in file *chall.ino.bin* that we just saved. File is small enough to do that from the (dis)comfort of a hex editor, or we can use command line: `strings -n8 chall.ino.bin`
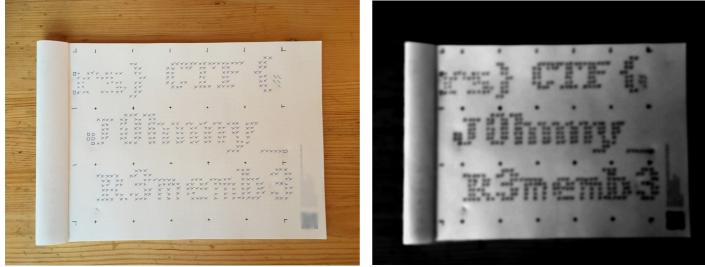
We get a handful of readable strings like: *to start!, Press a button, Game Over! Try again!, Arduino LLC, Arduino Leonardo.*

Our flag is not there.

3a) Glossing over *chall.ino.hex*, we notice some distinct patterns in the beginning of the file. It's bunch of zeros with some non-zero values. If you squint your eyes you can almost make out some images. Hey, but the **screen is black and white**, remember?

What happens if we try to **expand each nibble** (i.e. each character in the hexdump which equals to half a byte) **to it's binary representation**? Taking the last pattern at the begging of the file and writing it out on a piece of paper we end up with something that vaguely resembles some kind of text (As my paper is not long enough, note that I used symbols \ and □ to represent pixels of a neighboring nibble that I displaced to a different line for readability.) And

finally squinting our eyes yields a flag: `CTF{J0hnny_R3memb3rs}`. It seems a bit wrapped due to encoding, but that's not a challenge.

3b) But there's another way! Can you actually code? You're in luck, because you can save yourself quite some time extending data to the binary form on paper. It also helps, in case you did not choose the right pattern from the hexdump to begin with and did not end up with a flag. Knowing that we need to expand the hexdump to 1-bit color (binary) scheme, we can write this neat little gadget *solvejohnny.php*:

```php
<?php

$buffer=file_get_contents("php://stdin"); // read everything to buffer from standard input

$byte='';
for($i=0;$i<strlen($buffer);$i++){ // read one character from buffer
    if(!preg_match('/[a-z0-9]/i',$buffer[$i]))continue; // ignore non-hex characters
    $byte.=$buffer[$i];
    if(strlen($byte)==2){ // as soon as we have two hex characters (nibbles) in buffer
        // convert to binary, pad to full byte (8 bits), replace 0s with ' ' and 1s with '▓'
        echo str_replace('1','▓',str_replace('0',' ',
          str_pad(base_convert($byte,16,2),8,'0',STR_PAD_LEFT)));
        $byte='';
    }
}
```

Using our gadget we are able to decode all images inside the hexdump in one go:

```
cat chall.ino.hex | sed 's/^.........//;s/...$//' | php solvejohnny.php | less
```

All that is left to do is resize our terminal window and automatic line-wrapping will do all the magic for us. As soon we have our display aligned, we get what we're looking for.