

# SSTIC Challenge 2015

Erwan Hamon  
@r1hamon

May, 2015

## 1 Introduction

This is my solution to the SSTIC Challenge 2015: <http://communaute.sstic.org/ChallengeSSTIC2015>. If you want to have a try at the challenge and just need a little help you can refer to the “Clues” part of the beginning of each stage.

I suggest you Git your code as you develop your tools for solving the challenge. And commit as often as possible. This will help you find bugs you introduce as you get tired and drunk in the process. All the code I developed (quick and dirty of course!) will be available after the SSTIC at <https://github.com/r1git>. I will not include the original git folders that I used when solving the challenge. The commits comments are too offensive.

I also included a “Tracks of despair” section for each stage. That’s where I tell about how *not* to found the solution.

I apologize for my English, I thought that a lot of solutions would already be available in French.

## 2 Stage 1

After uncompressing the challenge.zip file, we find the file sdcard.img.

### 2.1 Clues

```
$ file sdcard.img
sdcard.img: x86 boot sector, mkdosfs boot message display, code offset 0x3c, OEM-ID "mkfs.fat",
sectors/cluster 4, root entries 512, Media descriptor 0xf8, sectors/FAT 244, heads 64,
sectors 250000 (volumes > 32 MB) , serial number 0xe50d883b, unlabeled, FAT (16 bit)
$ strings sdcard.img | tail -n1
java -jar encoder.jar -i /tmp/duckyscript.txt
```

Google search: “encoder.jar /tmp/duckyscript.txt” returns information about “USB Rubber Ducky”

```
$ mkdir mnt
$ sudo mount -o loop sdcard.img mnt/
$ ls mnt/
inject.bin
```

More search on Google gives the link to a java encoder source file: <https://github.com/midnitesnake/USB-Rubber-Ducky/blob/master/Encoder/src/Encoder.java>. Using a forensic tool (The Sleuth Kit for example) on the image shows a deleted file. It just contains the command `java -jar encoder.jar -i /tmp/duckyscript.txt` already found by strings. No new clue here.

## 2.2 Solution

As the clues show, we are dealing with a USB Rubber Ducky which is a USB key that acts as a keyboard. You plug it on a target computer and it starts emitting the key strokes programmed in its firmware. The `inject.bin` is such a firmware that is likely to have been produced by the `Encoder.java` found on github. Analysing the `Encoder.java` code, I developed a minimal decoder (`decoder.py`).

```
$ python decode.py > decoded
```

We see in the decoded file that the Rubber Ducky is programmed to send a `Windows+R` (which is a shortcut for executing a command on windows), then launch `cmd.exe` and then send many (3390) powershell commands. Each of those powershell command is encoded in base64 thanks to the `-enc` (equivalent to `-EncodedCommand`) of powershell. Those powershell commands are meant to decode 3389 base64 strings inside a `stage2.zip` file. Each time verifying that the user executing those commands is “challenge2015sstic” which is not relevant for us.

The last powershell command check the SHA1 checksum of the file giving us the opportunity to ensure that we will also decode it properly.

The task is therefore to:

1. parse each powershell command of the decoded file and decode the base64 scripts.
2. Decode the base64 included in each decoded script of step 1 and concatenate it to `stage2.zip` file

Both those steps are achieved by the `unpowershell.py` script.

```
$ python unpowershell.py > stage2.zip
$ sha1sum stage2.zip
ea9b8a6f5b527e72652019313c25b56ad27c7ec6 stage2.zip
```

The SHA1 checksum matches the one found in the last powershell command.

## 2.3 Tracks of despair

There was no real difficulty in that stage. The hardest part was actually to find the `Encoder.java` file with the clue “duckyscript”.

## 3 Stage 2

We are dealing with 3 files inside the stage2.zip.

### 3.1 Clues

The obvious memo.txt is self explanatory.

If you don't remember what a pk3 file is, Google it :).

```
$ file sstic.pk3
sstic.pk3: Zip archive data, at least v2.0 to extract
$ unzip sstic.pk3
$ nautilus textures/sstic

$ strings maps/sstic.bsp | grep key
"message" "Yes!\n You found my key !"
```

### 3.2 Solution

We are now dealing with an encrypted file. The memo.txt gives us everything needed to decrypt it, the algorithm, the IV and even the checksum of the decoded file so that we can ensure proper decryption. It is just missing the decryption key... It seems the emitter has hidden its key in a map of the Quake 3 FPS game. Is that plausible ? No. Do we care ? Nope... It's fun.

When looking in the textures/sstic directory of the map, we see a series of picture having hexadecimal colored parts on it as well as little symbols. There is too many of them so that a bruteforce seems unlikely. We need more information.

There are two possible ways to go here:

1. Installing the game and playing the map
2. Reversing the map

Let's try the first one. After installing Quake3 (you can find it easily on the net), getting the pak0.pak3 file (you can find it easily in your garage) installing the map, it's time to play. Of course you load the map with the \devmap sstic so that you can cheat during the game. Bring the Quake3 console and type the good old \noclip so that you can walk through walls. Wandering in the map, I quickly found 6 pictures containing hexadecimal colored value with a little black symbol. Some of the ones that we found in the textures/sstic directory of the pk3 file. At the same time, the noclip cheat code gives us direct access to a secret room where we see 8 black symbols associated with colors on a wall. Also, walking to that wall triggers the message "Yes! You found my key!".



It is not very hard to conclude that those are the symbols and colors of the hexadecimal pictures found before in the map. But I only found 6 out of 8. Anyway, we have enough information about the key and we can bruteforce the last two based on all the pictures found in the textures/sstic directory.

The dec.py script does that job, quickly finds the key and gives us the stage3.zip file.

### 3.3 Tracks of despair

I sadly have to admit that I again felt in a typical bug for this challenge: forgetting to remove the padding of the data after decryption. This of course gives a bad checksum and the bruteforce script tries the correct key without concluding to a solution. The good thing is that it forced me to try harder and therefore I tried to analyze the bsp file deeper. With the GtkRadiant map editor you can convert the bsp file to a map file and open it in the editor. In the editor and with static analysis of the bsp file, I found that the secret room is reachable by pressing buttons, rushing to a secret corridor in less than 30s, do a nice old-style rocket jump and land in the secret room to see the symbolic key... It's almost too bad that the \noclip trick gives you all of that for free. Yet, even with the editor I was not able to find the 2 missing parts of the key. The bruteforce was therefore still necessary.

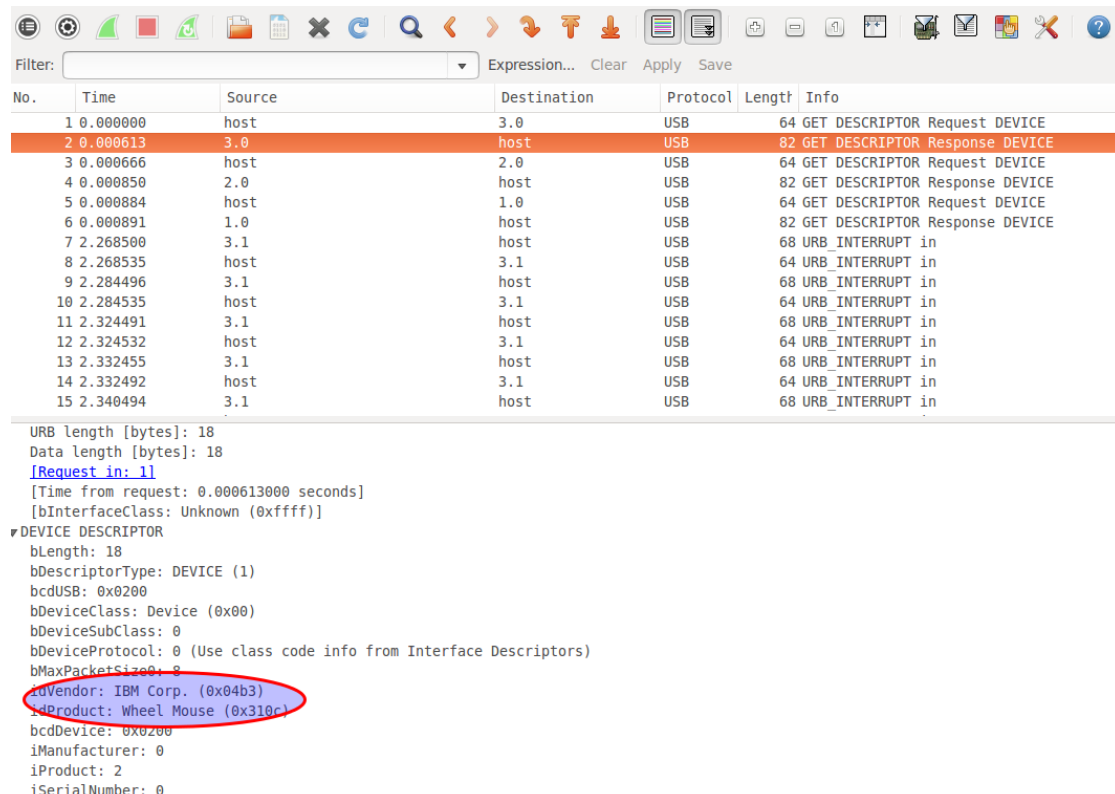
## 4 Stage 3

In this stage we have 3 files and again we need to find a key hidden in one of them in order to decrypt another one.

### 4.1 Clues

The obvious memo.txt. Watch out for the name and mode of the cypher:

```
Cipher: Serpent-1-CBC-With-CTS
$ file paint.cap
paint.cap: tcpdump capture file (little-endian) - version 2.4, capture length 262144
$ wireshark paint.cap
```



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	host	3.0	USB	64	GET_DESCRIPTOR Request DEVICE
2	0.000613	3.0	host	USB	82	GET_DESCRIPTOR Response DEVICE
3	0.000666	host	2.0	USB	64	GET_DESCRIPTOR Request DEVICE
4	0.000850	2.0	host	USB	82	GET_DESCRIPTOR Response DEVICE
5	0.000884	host	1.0	USB	64	GET_DESCRIPTOR Request DEVICE
6	0.000891	1.0	host	USB	82	GET_DESCRIPTOR Response DEVICE
7	2.268500	3.1	host	USB	68	URB_INTERRUPT in
8	2.268535	host	3.1	USB	64	URB_INTERRUPT in
9	2.284496	3.1	host	USB	68	URB_INTERRUPT in
10	2.284535	host	3.1	USB	64	URB_INTERRUPT in
11	2.324491	3.1	host	USB	68	URB_INTERRUPT in
12	2.324532	host	3.1	USB	64	URB_INTERRUPT in
13	2.332455	3.1	host	USB	68	URB_INTERRUPT in
14	2.332492	host	3.1	USB	64	URB_INTERRUPT in
15	2.340494	3.1	host	USB	68	URB_INTERRUPT in

USB length [bytes]: 18  
Data length [bytes]: 18  
[Request in: 1]  
[Time from request: 0.000613000 seconds]  
[bInterfaceClass: Unknown (0xffff)]  
▼ DEVICE\_DESCRIPTOR  
bLength: 18  
bDescriptorType: DEVICE (1)  
bcdUSB: 0x0200  
bDeviceClass: Device (0x00)  
bDeviceSubClass: 0  
bDeviceProtocol: 0 (Use class code info from Interface Descriptors)  
bMaxPacketSize0: 8  
iVendor: IBM Corp. (0x04b3)  
iProduct: Wheel Mouse (0x310c)  
bcdDevice: 0x0200  
iManufacturer: 0  
iProduct: 2  
iSerialNumber: 0

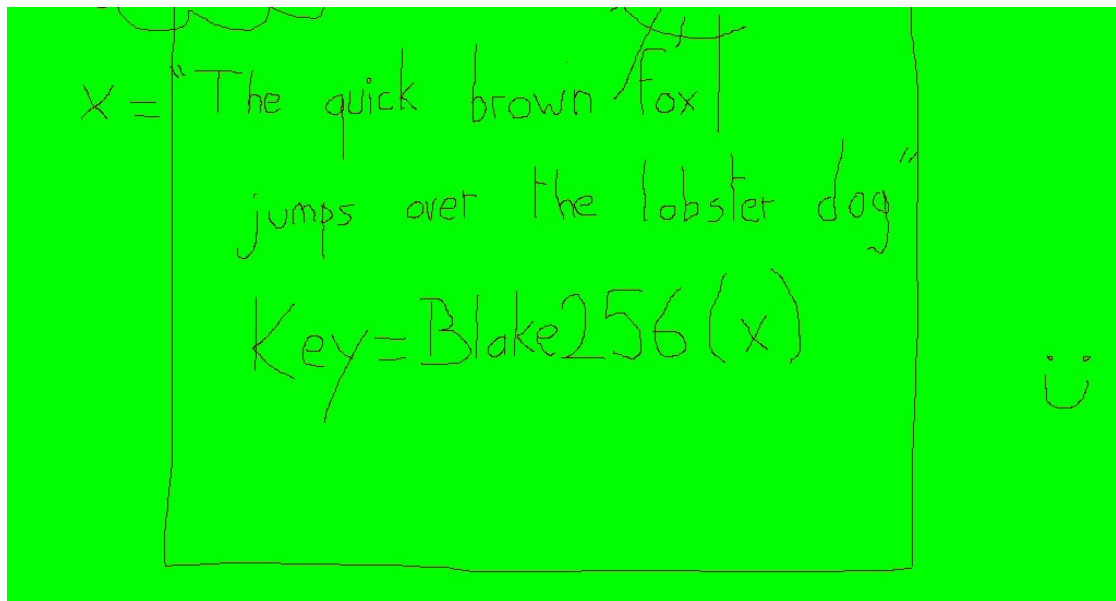
Google search on mode CBS-With-CTS: [http://en.wikipedia.org/wiki/Ciphertext\\_stealing#CBC\\_decryption\\_steps](http://en.wikipedia.org/wiki/Ciphertext_stealing#CBC_decryption_steps)

### 4.2 Solution

We now need to find a key hidden in the paint.cap file. The clues tell us that it is a capture of the usb communication from a wheel mouse. My first guess was that it was captured while the user was drawing the secret key manually inside paint. And it turned out that I spoiled myself because that is exactly what it is.

So we need to understand the usb mouse protocol and replay it to draw back the movement and clicks of the mouse. Not very difficult, but definitely fun. It appears that everything is encoded in the last 4 bytes of every “3.1” to “host” packets that you can read in wireshark. And you can find the description of the protocol at that url: [http://www.usbmadesimple.co.uk/ums\\_5.htm](http://www.usbmadesimple.co.uk/ums_5.htm). I developed the mouse.py script that parses the paint.cap, looks for the pattern corresponding to the mouse output, extracts the last 4 bytes and deduces the mouse movements and mouse clicks. If the mouse is clicked, it uses the Python Imaging Library to draw a line.

The output is then:



The key is therefore the hash of the string “The quick brown fox jumps over the lobster dog”. The hash is Blake256. A C implementation can be found here: <https://github.com/veorq/BLAKE/blob/master/blake256.c>

Now we have almost everything we need. We only need an implementation of Serpent-1 with the mode CBC-with-CTS enabled. I went for python-cryptoplus: <https://github.com/doegox/python-cryptoplus.git> The good thing is, it's in python. The bad thing is...it's in python. Not only is it slow but also the implementation of Serpent-1 is the one for educational purpose, easy to read but even slower. This would have been a no-go if brute force was needed. Fortunately it is not. The other bad thing is that library does not implement CBC-with-CTS mode. Yet it implements CBC and the wikipedia link found in the clues explains how to do CTS from a CBC mode. Almost too good to be true.

You can find the patch file for src/CryptoPlus/Cipher/blockcipher.py in the Annexe transforming the CBC in CBC-with-CTS. Then you:

```
export PYTHONPATH=/home/yourpath/python-cryptoplus/src
python dec.py.
```

6 minutes later on my laptop (!!!!! I told you...) the scripts decrypted the data.

### 4.3 Tracks of despair

I didn't loose track on this one. It was fun to make a program draw the paint image, it was very interesting to learn about CTS mode and it was very fun to (horribly) patch a library to make it do what I wanted.

## 5 Stage 4

This stage only consists of one file: stage4.html

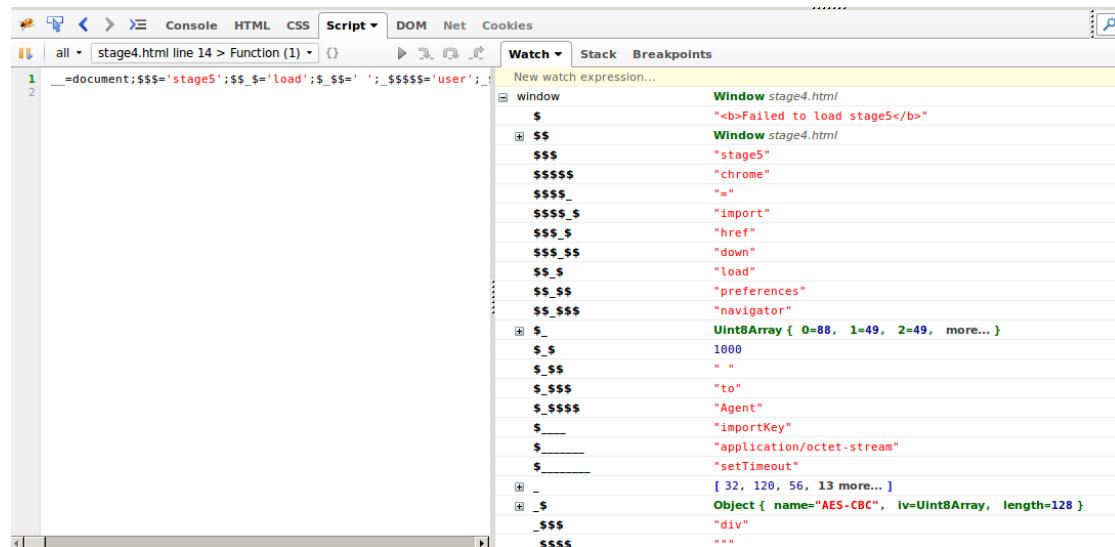
### 5.1 Clues

```
* { font-family: Lucida Grande, Lucida Sans Unicode,
  Lucida Sans, Geneva, Verdana, sans-serif; text-align: center; }
```

The Geneva fonts seems to have been developed for Apple Computers and is difficult to find on any other OS ([http://en.wikipedia.org/wiki/Geneva\\_%28typeface%29](http://en.wikipedia.org/wiki/Geneva_%28typeface%29)) Opening the stage4.html file in Firefox with Firebug and looking at the watch window of the javascript debugger gives a lot of variables with readable strings (notably: AES-CBC, the user agent of the browser, string manipulation functions...)

### 5.2 Solution

Here we are dealing with an obfuscated javascript script. When working on the clues, I realize the Firebug extension of Firefox did a good job de-obfuscating part of the script: if you go in the "Script/Function 1" menu of Firebug, you get the script on one line with all readable characters.



Javascript Beautifier (<http://jsbeautifier.org/>) helps cleaning that one-line code (file firebug.js) and now we need to basically do variables substitution to get something we could analyze. That exactly what the deob.py script does. The first part of the main() function initialize some variable names and the loop then substitutes and simplifies until there is no substitution left.

We now see a reference to <chrome://browser/content/preferences/preferences.xul> which is the link used by Firefox to access its preferences. The script is very clear now, it decrypts the data field with AES-CBC, initializing the IV with the 16 first characters inside the brackets of the user-agent and setting the key to the 16 last characters inside the brackets of the user-agent. We have clues indicating that Firefox under an Apple system is used. And we have the Firefox User Agent string reference for Mac [https://developer.mozilla.org/en-US/docs/Web/HTTP/Gecko\\_user\\_agent\\_string\\_reference](https://developer.mozilla.org/en-US/docs/Web/HTTP/Gecko_user_agent_string_reference).

Mac OS X version	Gecko user agent string
Mac OS X on Intel x86 or x86_64	Mozilla/5.0 (Macintosh; Intel Mac OS X x.y; rv:10.0) Gecko/20100101 Firefox/10.0
Mac OS X on PowerPC	Mozilla/5.0 (Macintosh; PPC Mac OS X x.y; rv:10.0) Gecko/20100101 Firefox/10.0

The script `brute.py` bruteforces the different user-agent (remember we are just interested by what is inside the brackets) until the good one is found and the data is decrypted.

### 5.3 Tracks of despair

Everything went fine on this stage until the bruteforce of the user-agent. I actually didn't found the official Mozilla user-agent reference until the very end. Therefore I was basing all my bruteforce on online database of user-agents (like <http://www.useragentstring.com/pages/Firefox/>). I used wget to get a lot of data from those sites and try different



combinations through a script. I always included a country inside the bracket of the user-agent when actually there is none in the solution. But almost all user-agents in the databases included a country or a language. Life is unfair.

## 6 Stage 5

We unzip a schematic.pdf and an inject.bin file.

### 6.1 Clues

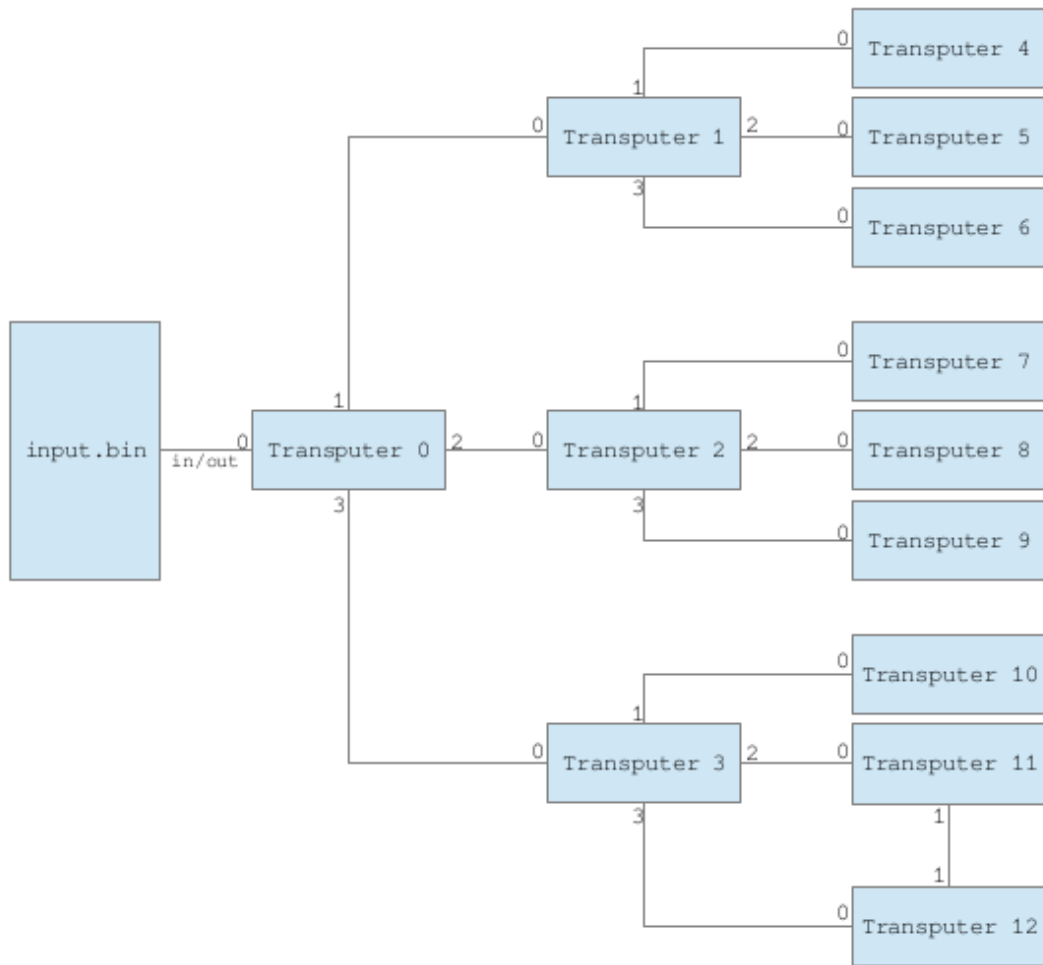
We see references to ST20 architecture and transputers in schematic.pdf.

A Google search brings us to a site with a lot of documentation on transputers and especially the Instruction Set Reference: <http://www.transputer.net/iset/iset.asp>

```
$ strings input.bin | grep tar
congratulations.tar.bz2
```

### 6.2 Solution

This stage was the hardest and of course the most interesting in my opinion. We had to deal with transputers. Transputers are basically processors working in parallel and exchanging synchronized messages via some sort of serial links. One processor can communicate with up to 4 other processors. The schematic.pdf gives us the architecture for this stage:



We see the input.bin file that we unzipped on the left. It is sent to Transputer 0 which is then responsible for transmitting the code and data to Transputer 1, 2 and 3. They themselves are responsible for feeding the rest of the transputers. Notice that transputers 11 and 12 are connected together. Typical communication between 2 transputers consists of two consecutive messages: the first one being the length of the next message to be received. Another fun fact about transputers: it mainly uses 3 registers (plus one for larger operands) for calculation and uses reverse polish calculus like the one we used on our old Texas Instrument calculator. Those 3 registers act like a First In Last Out stack, you can therefore push up to 3 values. The fourth push would dump the bottom of the stack and you would lose your first push. For example, Push 10, Push 5, Push 1, Add, would result in the stack containing 6 (5+1) and 10 from the top.

First task is to disassemble the input.bin code thanks to the reference document found during the clues gathering phase. The script extract.py extracts all transputers codes and save it to individual files. The rev.py script is responsible for disassembling as well as following the values of the 3 stack registers (and the operand register) and output it when there is a store in memory. We could call that concolic execution but that would be very insulting to people working on that subject. It needs two files (pop.txt and sec2.txt) which are the primary opcodes and secondary opcodes found and copy-pasted from the documentation. Here is an example of the output of the script when launched with the proper arguments (offset, rebase...):

```
0x32 0x4c ldc 0xc load constant
0x33 0xd0 stl 0x0 store local
STACK[0] = 0xc
0x34 0x13 ldlp 0x3 load local pointer
0x35 0x24 pfix 0x4 prefix
0x36 0xf2 opr mint 0x2 operate minimum integer
0x37 0x54 ldnlp 0x4 load non-local pointer
0x38 0x77 ldl 0x7 load local
0x39 0x61 nfix 0x1 negative prefix
0x3a 0x95 fcall 0x5 function call
0x3b 0x40 ldc 0x0 load constant
0x3c 0x11 ldlp 0x1 load local pointer
0x3d 0x23 pfix 0x3 prefix
0x3e 0xfb opr sb 0xb operate store byte
STACK[1] = 0x0
0x3f 0x13 ldlp 0x3 load local pointer
0x40 0xf1 opr lb 0x1 operate load byte
0x41 0x13 ldlp 0x3 load local pointer
0x42 0x83 adc 0x3 add constant
0x43 0xf1 opr lb 0x1 operate load byte
0x44 0x23 pfix 0x3 prefix
0x45 0xf3 opr xor 0x3 operate
0x46 0x13 ldlp 0x3 load local pointer
0x47 0x87 adc 0x7 add constant
0x48 0xf1 opr lb 0x1 operate load byte
0x49 0x23 pfix 0x3 prefix
0x4a 0xf3 opr xor 0x3 operate
0x4b 0x2f pfix 0xf prefix
0x4c 0x4f ldc 0xf load constant
0x4d 0x24 pfix 0x4 prefix
0x4e 0xf6 opr and 0x6 operate
0x4f 0x11 ldlp 0x1 load local pointer
0x50 0x23 pfix 0x3 prefix
0x51 0xfb opr sb 0xb operate store byte
STACK[1] = (0xff & *((&STACK[3] + 0x7)) ^ *((&STACK[3] + 0x3)) ^ STACK[3]))
0x52 0x41 ldc 0x1 load constant
0x53 0xd0 stl 0x0 store local
STACK[0] = 0x1
```

From left to right: [Adress][Opcode][Operation][Opcode 0xf (argument)][meaning]  
STACK[x] is the xth slot of local data of a process.

Now I'll go straight on describing what the reverse engineering of input.bin gives (I'll use Tx for Transputer x). Transputer 0 receives the first byte of input.bin. It interprets

it as the length of the next message, receives it and runs it. It prepares its stack and memory for the process to run, then send “Boot Ok” to the listener on link 0 (the same serial link where it received the code). Then it forwards the next messages received on link0 (the next data available in input.bin) to T1, T2 and T3. T1, T2 and T3 runs the code received and get prepared to similarly forward the next messages to T4-T12. T4-T12 receive their code as well and runs it.

When T0 has finished spreading all those codes to the transputers it sends “Code Ok” to link0. T0 then waits to receive “KEY:” followed by 12 bytes of a key that are initialized in input.bin as 12 times 0xFF. Then it sends “Decrypt” to link0 and waits to read the encrypted data 1 byte at a time.

Each time T0 receives a byte of encrypted data, it sends the 12 bytes of its local key to T1, T2 and T3, XOR the results and uses it to decrypt the received byte. Then it modifies the nth byte (modulo 12) of its local key before sending the result on link0.

**First interesting thing to notice, during the 12 first steps, the calculus of T1-T12 are not involved.** Only T0 is doing a simple calculus between the encrypted bytes and the key. For each encrypted bytes, T1-T3 are forwarding the key sent by T0 to their respective transputers and XOR the answers before sending them back to T0. **T0 is the only transputer to receive the encrypted bytes. T1-T12 don’t.**

T4-T12 are the only transputer’s code to use the “fcall” or “call” operand (depending on the documentation). It does exactly what you would expect in assembly: it calls a subroutine until ret is reached to come back just after the calling instruction. Arguments are to be placed on the local stack (value you access with ldl – load local). But here, a little trick is played on us: one argument is useless and the subroutines have access to farther elements of the stack actually accessing the local stack of the caller routine... Just a simple obfuscating technique specific to the transputer design.

The reverse of all Transputer’s function has been implemented in trans.py. T0 is the main() function, and T1-T12 are the F1-F12 function. Just run the script to launch the test vector given in schematic.pdf.

We now need to discover the 12 bytes key. We also have the clue that the file we will get at the end of the decryption is a .tar.bz2 file and it turns out the first bytes of a bz2 file are well defined (<http://en.wikipedia.org/wiki/Bzip2>):

```
.magic:16          = 'BZ' signature/magic number
.version:8         = 'h' for Bzip2 ('H'uffman coding), '0' for Bzip1 (deprecated)
.hundred_k_blocksize:8 = '1'..'9' block-size 100 kB-900 kB (uncompressed)

.compressed_magic:48 = 0x314159265359 (BCD (pi))
.crc:32            = checksum for this block
```

So we might expect for the 12 first decrypted bytes: “Bzh[1-9]1AY&SY??” We don’t know which block-size ([1-9]), but the better compression is obtained with 9 and it’s the default. We’ll start with that but we must include the rest in our bruteforce. The 2 last bytes must be bruteforced, we can not anticipate that part of the crc32.

Remember how the 12 first bytes are directly derived from the 12 first bytes of the encrypted data and the key? For those bytes, here is what apply (main() function of trans.py):

```
decryptedbyte = (encryptedbyte xor (n + 2*key[n])) & 0xff
```

Where n is the position of the encrypted byte.

For the first 12 bytes, we know the encrypted data (tailing data in input.bin), we also know some of the decrypted data (bzip2 header), we should solve the equation above to find the key bytes. But beware, for every (decrypted, encrypted, n) tuple, there is 2 and sometimes 3 solutions for key[n] that are solutions of the equation because of the “&0xff”. For example,  $0x41 = 0x41 \& 0xff$  but also  $0x41 = 0x141 \& 0xff$  and  $0x41 = 0x241 \& 0xff$ .  $(n+2*key[n])$  can not reach the 0x300 value with  $key[n] \leq 0xff$  and even rarely reach the 0x200 value. We will therefore at first only bruteforce the 2 first cases.

So the bruteforce script dec.py has to try  $65536$  (2 crc bytes) \*  $9$  (block size) \*  $4096$  (2 solutions for each byte of the key) =  $2\,415\,919\,104$ . Ok... that's within reach of a computer but I reversed the transputer code in python... Python is slow. Using pypy (<http://pypy.org/>) gets it about 10 times faster. But still it's getting too long. Now there are multiple way to improve that:

- optimize the logic of the transputer (Hard. Keep it in mind if completely stuck).
- optimize the code of the transputer (Possible but don't expect big improvement).
- port the code to a faster langage (Possible but don't expect big improvement)
- find another weakness in the way the decryption is done (Medium. That's the spirit of the challenge though...)
- optimize the brute-force logic (certainly possible, bzip2 format must not have given everything it could).

I decided to try the two last options at first and see if more will be needed. The problem is we have to decrypt the whole the file, then SHA256 the result to compare it to the expected checksum. That's too many operations. One way of improvement would be to only decrypt the beginning of the file and check if it looks like a valid bzip2 file. Looking at what could be anticipated in the rest of the bzip2, we see that the bit following the crc is deprecated and always at 0, then there is the 3 first bits of a pointer which is likely to be at 0 as well. And actually looking at many bzip2 file, those 4 bits are always at 0. This would filter out some beginning of decryption and improve the brute force by a theoretical factor of  $2^4 = 16$ . Not so bad but not good enough.

Looking at how the key is mixed in T0, we should not expect a strong avalanche effect from this algorithm. Therefore, I decided to make statistics during the bruteforce (which is never very far from the good key) of the 36 first decrypted bytes. It appeared that the 32nd byte was often decrypted as a 0xff. Again deciding that this byte MUST be 0xff filters out a lot of decryption candidates improving the bruteforce by a factor of

$2^8 = 256$ .

Those conditions added to the bruteforce, the script ends with the solution in less than an hour on my laptop with no more optimization.

So to make it short. I had to bet that:

- no key byte will have the property  $n + 2key[n] \geq 0x200$
- there is 4 null bits in the decrypted text at the 0xe position
- there is the 0xff byte in the decrypted text at the 0x20 position

Those 3 bets were very likely to be true. And they actually were true, preventing me from having to dig somewhere else.

### 6.3 Tracks of despair

When you reverse a code you cannot execute and develop it in another language, you have many holes to step your foot in. You can: misinterpret the language, misinterpret the logic of the code, create a bug in the logic of the code you produce, create a direct bug in the code you produce... Having the python code properly work with the test vector (thank you so much for that... this stage would have been so hard without it) was not so easy. Even though the logic of the transputers is not that complicated. I had to develop test cases and sometimes develop in another language to ensure I properly mimic the code of the transputers. I'm not even sure that developing a full emulator of the transputers would have helped.

I also lost a couple of hours because of a very simple bug due to the fact that developing late at night is not good for your mind: in python, you need to specifically "clone" an array (using a slice for example) to have 2 different arrays, otherwise you just get a reference to the same array and modifying one modifies the other. I already told myself a while ago that I should not forget this. This time I won't. For sure.

## 7 Stage 6

We uncompress the congratulations.jpg picture!



But no obvious email address. The author of the challenge is also teasing us with a "one little last step?" at the bottom.

### 7.1 Clues

```
$ strings congratulations.jpg | grep BZh  
BZh91AY&SY
```

### 7.2 Solution

Now we can recognize a bzip2 header easily! The `bz.py` extracts data from there to the end. We don't need to care about the extra data because I learnt in the previous steps that bzip2 manages its own blocks size and will trash the rest.

### 7.3 Tracks of despair

That was the easiest stage! Just a little joke to end the challenge I guess...

## 8 Stage 7

We uncompress yet another picture (congratulations.png). The joke is not over:



The teasing line now says: "two little last steps ?".

### 8.1 Clues

```
$ strings congratulations.png | grep sTic | wc -l
28
```

The PNG format and its "chunks": [http://en.wikipedia.org/wiki/Portable\\_Network\\_Graphics#.22Chunks.22\\_within\\_the\\_file](http://en.wikipedia.org/wiki/Portable_Network_Graphics#.22Chunks.22_within_the_file)

The compression methods in PNG files: <http://www.w3.org/TR/PNG/#10Compression>

Opening the file with hachoir (<http://forensicswiki.org/wiki/Hachoir>):



address	name	type	size	data
00000000.0	id	Bytes	00000008.0	"\x89PNG\r\n\x1a\n"
00000008.0	header/	Chunk	00000025.0	
00000021.0	background/	Chunk	00000018.0	
00000033.0	physical/	Chunk	00000021.0	
00000048.0	time/	Chunk	00000019.0	2015-02-27 13:40:19
0000005b.0	chunk[0]/	Chunk	00004931.0	
0000139e.0	chunk[1]/	Chunk	00004931.0	
000026e1.0	chunk[2]/	Chunk	00004931.0	
00003a24.0	chunk[3]/	Chunk	00004931.0	
00004d67.0	chunk[4]/	Chunk	00004931.0	
000060aa.0	chunk[5]/	Chunk	00004931.0	
000073ed.0	chunk[6]/	Chunk	00004931.0	
00008730.0	chunk[7]/	Chunk	00004931.0	
00009a73.0	chunk[8]/	Chunk	00004931.0	
0000adb6.0	chunk[9]/	Chunk	00004931.0	
0000c0f9.0	chunk[10]/	Chunk	00004931.0	
0000d43c.0	chunk[11]/	Chunk	00004931.0	
0000e77f.0	chunk[12]/	Chunk	00004931.0	
0000fac2.0	chunk[13]/	Chunk	00004931.0	
00010e05.0	chunk[14]/	Chunk	00004931.0	
00012148.0	chunk[15]/	Chunk	00004931.0	
0001348b.0	chunk[16]/	Chunk	00004931.0	
000147ce.0	chunk[17]/	Chunk	00004931.0	
00015b11.0	chunk[18]/	Chunk	00004931.0	
00016e54.0	chunk[19]/	Chunk	00004931.0	
00018197.0	chunk[20]/	Chunk	00004931.0	
000194da.0	chunk[21]/	Chunk	00004931.0	
0001a81d.0	chunk[22]/	Chunk	00004931.0	
0001bb60.0	chunk[23]/	Chunk	00004931.0	
0001cea3.0	chunk[24]/	Chunk	00004931.0	
0001e1e6.0	chunk[25]/	Chunk	00004931.0	
0001f529.0	chunk[26]/	Chunk	00004931.0	
0002086c.0	chunk[27]/	Chunk	00000050.0	
0002089e.0	data[0]/	Chunk	00000004.0	

## 8.2 Solution

The clues show us 28 chunk tagged with non standard tag "sTic" and which contained data.

The script extract.py extracts all the data in the chunk and uncompress it using the python zlib library with a window buffer of 32 as specified for the PNG format.

## 8.3 Tracks of despair

My first guess was that the chunks were not ordered and that I needed to try some combinations to obtain the final file. I was even misled by the fact that sending a file command on the isolated chunks data gave false positive.

But the high entropy of the data inside the chunks quickly pushed me back on the right track.

## 9 Stage 8

You guessed it... another picture. congratulations.tiff:



"...three little last steps ? "

### 9.1 Clues

The Tiff format: [http://en.wikipedia.org/wiki/Tagged\\_Image\\_File\\_Format](http://en.wikipedia.org/wiki/Tagged_Image_File_Format)

```
$hexedit congratulations.tiff
```

```

00000000 49 49 2A 00 08 00 00 00 09 00 00 01 03 00 01 00 00 00 7C 02 00 00 01 01 03 00 01 00 00 00 DA 01 00 00 02 01
00000024 03 00 03 00 00 00 7A 00 00 00 03 01 03 00 01 00 00 00 01 00 00 00 06 01 03 00 01 00 00 00 02 00 00 00 11 01
00000048 04 00 01 00 00 00 80 00 00 00 15 01 03 00 01 00 00 00 03 00 00 00 16 01 03 00 01 00 00 00 DA 01 00 00 17 01
0000006C 04 00 01 00 00 00 C8 CC 0D 00 00 00 00 00 08 00 08 00 08 00 00 01 00 00 00 00 00 00 00 00 01 00 00 00 01 00 00
00000090 01 00 01 00 00 01 00 00 00 01 00 01 00 00 01 00 00 00 00 00 00 00 01 01 00 01 00 00 00 01 00 00 00 00 01
000000B4 01 00 00 00 00 00 01 00 00 00 01 00 00 00 00 00 00 00 00 00 01 00 00 00 01 00 01 00 00 00 01 00 00 00 01
000000D8 00 00 00 01 00 01 00 00 00 01 00 00 01 00 00 00 00 01 01 00 00 01 00 00 01 00 00 00 01 00 00 01 00 00
000000FC 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 01 00 00 01 00 00 01 01 00 00
00000120 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 01 00 00 00 01 00 00 00 00
00000144 00 00 00 00 00 00 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 00 01 01 00 01 01 00 01
00000168 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 00 01 01 00 01 01 00 01
0000018C 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 00 01 01 00 01 01 00 01
000001B0 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 00 01 01 00 01 01 00 01
000001D4 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 00 01 01 00 01 01 00 01
000001F8 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 00 01 01 00 01 01 00 01
0000021C 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 00 01 01 00 01 01 00 01
00000240 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 00 01 01 00 01 01 00 01
00000264 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 00 01 01 00 01 01 00 01
00000288 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 00 01 01 00 01 01 00 01
000002AC 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 00 01 01 00 01 01 00 01
000002D0 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 01 01 00 00 00 00 00 00 00 01 00 00
000002F4 00 00 01 00 00 00 00 00 00 00 00 00 01 01 00 01 01 00 01 01 00 00 00 01 01 00 01 01 00 01 00 00 00 00 00
00000318 00 00 00 00 00 00 00 00 01 00 00 00 01 00 00 00 01 00 01 01 00 00 01 01 01 00 00 01 00 01 01 00 00 00 00 01

```

## 9.2 Solution

In the hexedit view, we are supposed to view the raw 3 color bytes of each pixel. The picture looks like it is surrounded with black pixels (00 00 00), but it's actually only very close to black (00 01 00 for example). So it seems that some information is hidden in the Least Significant Bits of the image's pixels. Watching how those LSB are altered in the file, we notice that only 2 of the 3 RGB bytes are actually modified to store a bit of information. The last one is not modified (would have it been altered randomly, the challenge would have been a "bit" harder).

The script grab.py extracts the relevant bits of the picture.

## 9.3 Tracks of despair

I spent some time decoding all 3 bytes of the color encoding. Taking them in order (on the x axis, line by line), as well as in reverse or along the y axis before checking if all bytes were relevant. I also got diverted by the fact the height and width of the image are multiple of 6... Don't ask.

## 10 Stage 9

Here is the expected gif file:



”...four little last steps ? ”

## 10.1 Clues

Hachoir view of the color map element of the gif:

address	name	type	size	data	description
0000000d.0	../				
0000000d.0	color[0]/	RGB	00000003.0		RGB color: Black
00000010.0	color[1]/	RGB	00000003.0		RGB color: Black
00000013.0	color[2]/	RGB	00000003.0		RGB color: Black
00000016.0	color[3]/	RGB	00000003.0		RGB color: #0B0402
00000019.0	color[4]/	RGB	00000003.0		RGB color: #070A06
0000001c.0	color[5]/	RGB	00000003.0		RGB color: #0E0806
0000001f.0	color[6]/	RGB	00000003.0		RGB color: #160A0B
00000022.0	color[7]/	RGB	00000003.0		RGB color: #290807
00000025.0	color[8]/	RGB	00000003.0		RGB color: #2D0706
00000028.0	color[9]/	RGB	00000003.0		RGB color: #0E100C
0000002b.0	color[10]/	RGB	00000003.0		RGB color: #1D0C0C
0000002e.0	color[11]/	RGB	00000003.0		RGB color: #140F0E

## 10.2 Solution

We observe that this gif file has a color map that allows to map a color encoding with a color for the rendering. In this color map, Hachoir nicely shows us that many colors have been mapped to black. One or many of those mappings to black are certainly used to hide information on the image.

We don't know which ones are used to hide information. The script `col.py` replaces all black mappings in the color map with a random color.

The result is:



Hurray !

## 10.3 Tracks of despair

Thanks to Hachoir, this stage was straight forward.

## 11 Conclusion

I believe a good challenge is about learning stuff, getting stuck sometimes, having fun and making you think about it in your shower.

Those goals were achieved in my case and I enjoyed it.

It was not exceedingly difficult and therefore we'll certainly hear that it was easy. That's definitely a very relative statement. Anyway, thinking back about it, it felt like the kind of marathon you are always proud to finish whatever time it took you.

Many thanks to the author(s).

*Merci Vanessa pour tes encouragements et ta patience...*

# Annexe

## Stage 1

decode.py

```
import sys
import struct

f = open("../inject.bin", "rb")

def p(sttr, el=True):
    sys.stdout.write(sttr)
    if el:
        sys.stdout.write("\n")

def bytetochar(h, s):
    v = struct.unpack('B',h)[0]
    if (v+0x5d>=97 and v+0x5d<=122):
        v = v+0x5d
        if s:
            v -= 0x20
        res = chr(v)
    elif (v+0x13>=49 and v+0x13<=57):
        res = chr(v+0x13)
    elif v == 0x2c:
        res = ','
    elif v == 0x2d:
        res = '-'
    elif v == 0x2e:
        res = '.'
    elif v == 0x27:
        res = "'"
    else:
        res = ' '

    return res

def decode():
    byte = f.read(1)
    while byte != "":
        if byte == '\x00':
            #p("\nDELAY "+str(int(f.read(1).encode("hex"),16)))
            f.read(1)
        elif byte == '\x29':
            p("\nCTRL ESC")
        elif byte == '\x48':
            p("\nPAUSE")
        elif byte == '\x28':
            p("\nENTER")
            byte = f.read(1)
            if (byte != '\x00'):
                p("Enter not followed by 0")
        else:
            s = f.read(1)
            shift = False
            if (s == '\x02'):
                shift = True
            elif (s != '\x00'):
                p("???",False)
            b = bytetochar(byte, shift)
```

```

        if b=='' :
            p("UNDECODED "+byte.encode("hex")+"("+byte+")")
        else:
            p(b, False)
    byte = f.read(1)

decode()

```

## Stage 2

### dec.py

```

from Crypto.Cipher import AES
from Crypto.Hash import SHA256
import sys

unpad = lambda s : s[0:-ord(s[-1])]

encoded = open("../encrypted", "r").read()
ha = SHA256.new()
ha.update(encoded)
check = ha.digest().encode("hex")

if(check != "91d0a6f55cce427132fc638b6beecf105c2cb0c817a4b7846ddb04e3132ea945"):
    print "BAD INPUT"
    sys.exit()

IV = '5353544943323031352d537461676532'.decode('hex')
key = ["9e2f31f7", "8153296b", "3d9b0ba6", "", "b0daf152", "b54cdc34", "ffe0d355", ""]

p1 = ["7695dc7c", "f61a3560", "36c2e6fc", "3c66fa3b", "8154c63a",
      "8ca39515", "e8c67d28", "7c16f3e9", "a5cb854f", "fbfac1eb"]

p2 = ["eda879c3", "26609fac", "c2e15ca0", "93fa1122", "db12fe60",
      "42404ba0", "c70a5383", "9dfc72db", "43210a41", "5a689be0"]

i1 = -1
i2 = 0

while True:
    i1 += 1
    if i1 == len(p1):
        i1 = 0
        i2 += 1
        if i2 == len(p2):
            print "Tested all"
            break
    key[3] = p1[i1]
    key[7] = p2[i2]
    obj = AES.new("".join(key).decode('hex'), AES.MODE_OFB, IV)
    decoded = unpad(obj.decrypt(encoded))
    hashed = SHA256.new()
    hashed.update(decoded)
    res = hashed.digest().encode("hex")

    if(res == "845f8b000f70597cf55720350454f6f3af3420d8d038bb14ce74d6f4ac5b9187"):
        print "Found correct key:", "".join(key)
        f = open("decrypted", "w")
        f.write(decoded)
        f.close()
        break

```



```

else:
    print "Failed ",i1,i2,"".join(key)

```

## Stage 3

mouse.py

```

import sys
import struct
from PIL import Image, ImageDraw

mse = open("../paint.cap").read()
inf = "\xc0\x8d\xe7\xf6"+"x00"*4+"x43"

i=-1
startx = 0
starty = 0

im = Image.new('RGBA', (1024, 1024), (0, 255, 0, 0))
draw = ImageDraw.Draw(im)

while(True):
    i = mse.find(inf, i+1)
    if i == -1:
        break
    mv = mse[i+0x40:i+0x44]
    #sys.stdout.write(mv)

    if mv[0]=='\x01':
        but = True
    elif mv[0]=='\x00':
        but = False
    else:
        print "mouse != 1 !!!"
        but = False

    mvx = struct.unpack('b',mv[1])[0]
    mvy = struct.unpack('b',mv[2])[0]
    if mv[3]!='\x00':
        print "wheel !!!"

    if but:
        draw.line(((startx,starty), (startx+mvx,starty+mvy)), fill=128)
    startx += mvx
    starty += mvy
im.show()

```

git diff blockcipher.py

```

diff --git a/src/CryptoPlus/Cipher/blockcipher.py b/src/CryptoPlus/Cipher/blockcipher.py
index 78d2669..3cd952d 100644
--- a/src/CryptoPlus/Cipher/blockcipher.py
+++ b/src/CryptoPlus/Cipher/blockcipher.py
@@ -298,23 +298,20 @@ class CBC:
     self.cache += data
     if len(self.cache) < self.blocksize:
         return ''
-    needed = len(self.cache)%self.blocksize
+    needed = self.blocksize - (len(self.cache)%self.blocksize)
+    myend = (len(self.cache)/self.blocksize * self.blocksize) - self.blocksize

```

```

        print "Size cache:", len(self.cache)
        print "Needed:", needed
        print "myend:", myend
-       for i in xrange(0, myend+1, self.blocksize):
-           plaintext = util.xorstring(self.IV, self.codebook.decrypt(self.cache[i:i + self.blocksize]))
-           if(i<myend):
-               self.IV = self.cache[i:i + self.blocksize]
-               decrypted_blocks+=plaintext
-               self.cache = self.cache + plaintext[-needed:]
+       decr = self.codebook.decrypt(self.cache[myend:myend+self.blocksize])
+       self.cache = self.cache + decr[-needed:]
+       self.cache = self.cache[:myend] + self.cache[myend+self.blocksize:] + self.cache[myend:myend+self.blocksize]
-       for i in xrange(myend, myend+self.blocksize+1, self.blocksize):
-           print "i:", i
+       print "Cache padded:", len(self.cache)
+       for i in xrange(0, myend+self.blocksize+1, self.blocksize):
+           plaintext = util.xorstring(self.IV, self.codebook.decrypt(self.cache[i:i + self.blocksize]))
+           self.IV = self.cache[i:i + self.blocksize]
+           decrypted_blocks+=plaintext
+       print "Last i:", i

        self.cache = self.cache[i+self.blocksize:]
        return decrypted_blocks

```

## dec.py

```

from CryptoPlus.Cipher import python_Serpent
from Crypto.Hash import SHA256
import sys
import serpent

#key = Blake256("The quick brown fox jumps over the lobster dog")
key = "66c1ba5e8ca29a8ab6c105a9be9e75fe0ba07997a839ffae9700b00b7269c8d"

unpad = lambda s : s[0:-ord(s[-1])]

encoded = open("../encrypted", "r").read()
ha = SHA256.new()
ha.update(encoded)
check = ha.digest().encode("hex")

if(check != "6b39ac2220e703a48b3de1e8365d9075297c0750e9e4302fc3492f98bdf3a0b0"):
    print "BAD INPUT"
    sys.exit()

IV = '5353544943323031352d537461676533'.decode('hex')
print IV

obj = python_Serpent.new(key.decode('hex'), python_Serpent.MODE_CBC, IV)
print "Trying"

decoded = obj.decrypt(encoded)[-2:]
print "Done"
hashed = SHA256.new()
hashed.update(decoded)
res = hashed.digest().encode("hex")

if(res == "7beabe40888fbbf3f8ff8f4ee826bb371c596dd0cebe0796d2dae9f9868dd2d2"):
    print "Yesss !"
else:
    print "Failed "

```

```
f = open("decrypted", "w")
f.write(decoded)
f.close()
```

## Stage 4

deob.py

```
import re
import sys

def repl(myfind, myto, mystr):
    fr = myfind.replace('$', '\\$')
    reg = r"(?![_\$\s])(" + fr + r")"?[_\$\s]"
    pattern = re.compile(reg)
    return re.sub(pattern, myto, mystr)

def main():
    f = open("firebug.js", "r").read()
    f = repl('_____', 'FUN1', f)
    f = repl('_____', 'localVar', f)
    f = repl('_____', 'arg', f)
    f = repl('__$', 'localVar2', f)
    f = repl('_____', 'FUN2', f)
    f = repl('_____', 'FUN3', f)
    f = repl('_____', 'FUN4', f)
    f = repl('_____', 'index', f)
    f = repl('_', 'localArray', f)
    f = repl('$_', 'localVar3', f)
    f = repl('__$', 'localVar4', f)
    f = repl('_', 'localVar5', f)
    f = repl('_____$', 'localArray2', f)
    f = repl('__$', 'FUN5', f)
    f = repl('__$', 'FUN6', f)
    f = repl('____$$', 'FUN7', f)
    f = repl('__$', 'FUN8', f)
    f = repl('__$', 'FUN9', f)

    for i in range(0,40):
        f2 = open("firebug.js", "r")
        l = f2.readline()
        while(l):
            #print l
            regex = r"[\t]*([_\$\s]*) = (.*);"
            pa = re.compile(regex)
            res = re.match(pa, l)
            if res:
                #print res.group(1), res.group(2)
                f = repl(res.group(1), res.group(2), f)
            l = f2.readline()
        f = f.replace('\\' + '\\', '\\')
        f2.close()
    print f

main()
```

brute.py

```
import re
import sys
import itertools
import time

from Crypto.Hash import SHA
from Crypto.Cipher import AES
from os import listdir, rename
from os.path import isfile, join, basename

unpad = lambda s : s[0:-ord(s[-1])]

data = open("data").read()[:-1].decode("hex")
ch = "08c3be636f7dffd91971f65be4cec3c6d162cb1c"

ptfm = ["Macintosh;", "Macintosh; U;"]
arch = ["Intel", "PPC"]
cy = ["", " fr;", " en-US;", " en-UK;", " en-GB;"]

version = [""]
for i in range(0,11):
    version += [" 10."+str(i)]

# Find possible firefox versions in online web page
rv = []
mydata = open("ffver.html").read()
reg = r"href=\"([0-9\\.]+)/"
pattern = re.compile(reg)
res = re.findall(pattern, mydata)
rv = sorted(set(res))
# For each .0 version add the version without the .0
for el in rv:
    if el[-2:] == ".0":
        rv += [el[:-2]]

all = [ptfm, arch, version, cy, rv]
total = len(ptfm)*len(arch)*len(version)*len(cy)*len(rv)
print "Trying a total of:", total
ct = 0
prog = 0
timer = 10000
stt = time.time()
for t in itertools.product(*all):
    timer -= 1
    if timer == 0:
        dt = time.time() - stt
        eta = (total / 10000) * dt
        print "ETA:", eta, "s"
    if ct == total/10:
        prog+=1
        print str(prog*10)+"% done"
        ct = 0
    ua = t[0]+" "+t[1]+" Mac OS X"+t[2]+";"+t[3]+" rv:"+t[4]
    #print "<"+ua+">"

    iv = ua[:16]
    key = ua[-16:]
    if(len(key)<16):
        continue
```

```

obj = AES.new(key, AES.MODE_CBC, iv)
decoded = unpad(obj.decrypt(data))
hashed = SHA.new()
hashed.update(decoded)
res = hashed.digest().encode("hex")

if(res == ch):
    print "User agent found !"
    print ua
    f = open("decrypted", "w")
    f.write(decoded)
    f.close()
    sys.exit()

ct += 1

```

## Stage 5

### extract.py

```

import struct

data = open("../input.bin", "r").read()

i = 0xf8+1
n = 1
size = 1
while(size!=0):
    F1 = data[i:i+0xc]
    #fi = open("F"+str(n), "w")
    #fi.write(F1)
    #fi.close()
    size = struct.unpack("<I", F1[0:4])[0]
    dest = struct.unpack("<I", F1[4:8])[0]
    print "T"+str(n)+":", "Index", hex(i), "Size", hex(size), "Dest", hex(dest)
    i += 0xc
    if size != 0:
        D1 = data[i:i+size]
        fi = open("T"+str(n), "w")
        fi.write(D1)
        fi.close()
        i += size
    n+=1

```

### rev.py

```

import struct
import sys

A = ""
B = ""
C = ""
oreg = 0

def ps():
    global A,B,C
    print "A = ",A
    print "B = ",B
    print "C = ",C

def push(i):

```

```

    global A,B,C
    C=B
    B=A
    A=i

def fpop():
    global A,B,C
    res = A
    A=B
    B=C
    return res

popf = open("pop.txt", "r")

pop = []
popi = []
for i in range(0,16):
    ops = popf.readline().rstrip().split(' ')
    pop.append(ops[0])

for i in range(0,16):
    ops = popf.readline().rstrip()
    popi.append(ops)

popf = open("sec2.txt", "r")

sec = []
seci = []
for line in popf:
    ops = line.rstrip().split(' ')
    sec.append(ops[0])
    if len(ops)>1:
        seci.append(" ".join(ops[1:]))
    else:
        seci.append("")

if(len(sys.argv) >= 3):
    start = int(sys.argv[2])
else:
    start = 1

if(len(sys.argv) >= 4):
    ad = int(sys.argv[3])
else:
    ad = 0

data = open(sys.argv[1], "r").read()[start:]
prev = -1
for op in data:
    m = struct.unpack("B", op)[0]
    o = (m & 0xf0) >> 4
    n = (m & 0x0f)
    if o == 0xf:
        ind = 0
        if prev>=0x21 and prev<=0x29:
            ind = prev-0x20
        do = sec[n+ind*16]
        doi = seci[n+ind*16]
    else:
        do = ""
        doi = ""
    print '{0:5s} {1:5s} {2:5s} {3:10s} {4:5s} {5:24s} {6:16s}'.format(hex(ad), hex(m), pop[o], do, hex(n), popi[o],

```

```

if(prev<0x21 or prev>0x29):
    if(o==1):
        push("&STACK["+str(n)+"]")
    elif(o==2):
        oreg |= n
        oreg <= 4
    elif(o==4):
        push(hex(n|oreg))
    elif(o==6):
        oreg |= ~n
        oreg <= 4
    elif(o==7):
        push("STACK["+str(n)+"]")
    elif(o==8):
        push("(" + fpop() + " + " + hex(n|oreg) + ")")
    elif(o==0xd):
        print("STACK["+str(n)+"] = " + fpop())
    elif(o==0xf):
        if(n==1):
            if(A[0] == '&'):
                A = A[1:]
            else:
                A = "*" + A + ""
        elif(n==2):
            arr = fpop()
            index = fpop()
            push(arr + "[" + index + "]")
        elif(n==5):
            push("(" + fpop() + " + " + fpop() + ")")
        elif(n==8):
            push("(" + fpop() + " * " + fpop() + ")")
        elif(n==9):
            print("if " + fpop() + " > " + fpop())
        elif(n==0xa):
            index = fpop()
            arr = fpop()
            push(index + "[4*" + arr + "]")
    if(o!=2 and o!=6):
        oreg = 0
else:
    if(o==0xf):
        oreg = 0
    if(prev==0x23):
        if(n==3):
            push("(" + fpop() + " ^ " + fpop() + ")")
        elif(n==0xb):
            dest = fpop()
            if(dest[0]=='&'):
                dest = dest[1:]
            val = fpop()
            print(dest + " = " + val)
    elif(prev==0x24):
        if(n==0):
            val1 = fpop()
            val2 = fpop()
            push("(" + val2 + " >> " + val1 + ")")
        elif(n==1):
            val1 = fpop()
            val2 = fpop()
            push("(" + val2 + " << " + val1 + ")")
        if(n==6):
            push("(" + fpop() + " & " + fpop() + ")")

```

```

        elif(prev==0x21):
            if(n==0xf):
                val1 = fpop()
                val2 = fpop()
                push("(" + val2 + " % " + val1 + ")")
            elif(prev==0x25):
                if(n==0xa):
                    C=B
                    B=A
        elif(o==2):
            oreg |= n
            oreg <<= 4

    #ps()

    prev = m
    ad += 1

```

trans.py

```

import sys

var4 = 0
def F4(key):
    global var4
    for e in key:
        var4 = (var4 + e) & 0xff
    return var4

var5 = 0
def F5(key):
    global var5
    for e in key:
        var5 = (var5 ^ e) & 0xff
    return var5

var6_3 = 0
var6_1 = 0
def F6(key):
    global var6_3, var6_1
    if var6_3 == 0:
        for e in key:
            var6_1 = (var6_1 + e) & 0xffff
            var6_3 = 1
    aux1 = (var6_1 << 1) & 0xffff
    aux2 = (var6_1 & 0x4000) >> 0xe
    aux3 = (var6_1 & 0x8000) >> 0xf

    var6_1 = (aux1 ^ ((aux2 ^ aux3) & 0xffff)) & 0xffff
    return (var6_1 & 0xff)

def F7(key):
    res1 = 0
    res2 = 0
    for i in range(0,6):
        res1 = (key[i]+res1) & 0xff
        res2 = (key[i+6]+res2) & 0xff
    return (res1 ^ res2) & 0xff

```



```

var8_5 = []
var8_5.append([0]*12)
var8_5.append([0]*12)
var8_5.append([0]*12)
var8_5.append([0]*12)
var8_4 = 0
def F8(key):
    global var8_5, var8_4
    var8_5[var8_4] = key
    var8_4+=1
    if var8_4 == 4:
        var8_4 = 0
    res = 0
    for i2 in range(0,4):
        acc = 0
        for i0 in range(0,12):
            acc = (acc + var8_5[i2][i0]) & 0xff
        res = (res ^ acc) & 0xff

    return res

def F9(key):
    res = 0
    for i in range(0,12):
        res = ((key[i] << (0x7 & i)) ^ res) & 0xff

    return res

var10_4 = []
var10_4.append([0]*12)
var10_4.append([0]*12)
var10_4.append([0]*12)
var10_4.append([0]*12)
var10_2 = 0
def F10(key):
    global var10_4, var10_2
    var10_4[var10_2] = key
    var10_2+=1
    if(var10_2==4):
        var10_2=0
    res = 0
    for i in range(0,4):
        res = (res + var10_4[i][0]) & 0xff

    index = res & 0x3
    chrn = ((res >> 4) % 0xc) & 0xff

    return var10_4[index][chrn]

var12_3 = [0]*12
def F12(key, read):
    global var12_3
    tosend = (var12_3[1] ^ var12_3[5] ^ var12_3[9]) & 0xff
    var12_3 = key
    index = (read % 0xc) & 0xff
    res2 = var12_3[index]

    return tosend, res2

def F11(key):
    tosend = (key[0] ^ key[3] ^ key[7]) & 0xff

```

```

    read, res2 = F12(key, tosend)
    index = (read % 0xc) & 0xff
    res1 = key[index]

    return res1, res2

def F1(key):
    res1 = F4(key)
    res2 = F5(key)
    res3 = F6(key)
    return res1 ^ res2 ^ res3

def F2(key):
    res1 = F7(key)
    res2 = F8(key)
    res3 = F9(key)
    return res1 ^ res2 ^ res3

def F3(key):
    res1 = F10(key)
    res2, res3 = F11(key)
    #res3 = F12(key)
    return res1 ^ res2 ^ res3

LKey = []
L4 = 0
def main(ch):
    global LKey, L4
    newkey = (F1(LKey) ^ F2(LKey) ^ F3(LKey)) & 0xff
    res = (ch ^ (L4 + 2*LKey[L4])) & 0xff
    LKey = LKey[:]
    LKey[L4] = newkey
    L4+=1
    if(L4 == 0xc):
        L4 = 0
    return res

def initi():
    global var4, var5, var6_3, var6_1, var8_5, var8_4, var10_4, var10_2, var12_3, L4
    var4 = 0
    var5 = 0
    var6_3 = 0
    var6_1 = 0
    var8_5 = []
    var8_5.append([0]*12)
    var8_5.append([0]*12)
    var8_5.append([0]*12)
    var8_5.append([0]*12)
    var8_4 = 0
    var10_4 = []
    var10_4.append([0]*12)
    var10_4.append([0]*12)
    var10_4.append([0]*12)
    var10_4.append([0]*12)
    var10_2 = 0
    var12_3 = [0]*12
    #LKey = []
    L4 = 0

if __name__ == '__main__':
    data = "1d87c4c4e0ee40383c59447f23798d9fefe74fb82480766e".decode("hex")

```

```

LKey = map(ord, '*SSTIC-2015*')
for e in data:
    sys.stdout.write(chr(main(ord(e))))
print ""

```

dec.py

```

import trans
import time
import sys
import struct
import hashlib

```

```

data = open("../input.bin").read()[0x9ad:]
good = "a5790b4427bc13e4f4e9f524c684809ce96cd2f724e29d94dc999ec25e166a81"
hur = "9128135129d2be652809f5a1d337211affad91ed5827474bf9bd7e285ecf321"

```

```

ha = hashlib.sha256()
ha.update(data)
check = ha.digest().encode("hex")

```

```

if(check != good):
    print "BAD INPUT"
    sys.exit()
else:
    print "Input Ok"

```

```

data = map(ord, data)
ctr = 0
timestart = time.time()
step = 500000
tried = 0

```

```

# bs is the Block Size unknown value of bzip2 header (probably '9' == 0x39)
for bs in range(0x39, 0x30, -1):
    header = ("425a68"+str(hex(bs)[2:])+ "314159265359").decode("hex")
    # crc1 and crc2 are the unknown 2 bytes of the crc32
    for crc1 in xrange(0,256):
        for crc2 in xrange(0,256):
            #ti is the possible decrypted header tested
            ti = header+chr(crc1)+chr(crc2)

```

```

# sh nth bit at 1 will take the 2nd solution for n+2*key[n]
for sh in xrange(0, pow(2,12)):
    skip = False
    ctr+=1
    if(ctr%step==0 or tried>50):
        now = time.time()
        print "bs:", hex(bs), "index:",crc1,crc2, "time for last ",step,":", now-timestart, "s",
        tried = 0
        timestart = time.time()
    trans.LKey = [0]*12

```

```

for i in range(0,12):
    # The 2 possible solutions
    if(sh & pow(2,i)):
        thissh = 0x100
    else:
        thissh = 0
    calc = ((ord(ti[i]) ^ data[i]) | thissh) - i
    if calc%2 != 0:

```

```

        #print "Skip modulo"
        skip = True
        break
    if calc<0:
        skip = True
        break
    trans.LKey[i] = calc / 2
    if(trans.LKey[i]>255):
        skip = True
        break

if(skip):
    continue

trans.initi()

decr = [ trans.main(e) for e in data[0:0xf] ]
# Expecting the 4 null bits
if decr[0xe] & 0xf0 != 0:
    continue
middle = [ trans.main(e) for e in data[0xf:0x21] ]
# Expecting the 0xff value
if middle[17] != 255:
    continue
#If we are here we are going to try the full decryption and SHA256. This is expensive.
tried += 1

end = [ trans.main(e) for e in data[0x21:] ]
decr = decr + middle + end
decr = "".join(map(chr, decr))

ha = hashlib.sha256()
ha.update(decr)
check = ha.digest().encode("hex")

if(check == hur):
    print "We have a match !"
    f = open("decrypted", "w")
    f.write(decr)
    f.close()
    f = open("res", "w")
    f.write(str(map(chr, trans.LKey))+ " bs"+hex(bs))
    f.close()
    sys.exit()
else:
    pass

print "CTR:", ctr

```

## Stage 6

bz.py

```

f = open("../congratulations.jpg", "r").read()[0xd7d0:]
o = open("stage7.tar.bz2", "w")
o.write(f)
o.close()

```

## Stage 7

extract.py

```
import struct
import zlib

f = open("../congratulations.png", "r").read()
res = ""
start = 0
ct = 0

while start != -1:
    start = f.find("sTic", start+1)
    if (start != -1):
        size = struct.unpack(">I", f[start-4:start])[0]
        print "Found - Num:", ct, "Size:", size, "@", hex(start)
        if size <= 4919:
            res = res + f[start+4:start+size+4]
        ct+=1

print "Total len:", len(res)

unco = zlib.decompress(res, 32)

f = open("stage8.tar.bz2", "w")
f.write(unco)
f.close()
```

## Stage 8

grab.py

```
import sys

data = open("../congratulations.tiff").read()[0x80:]
data = map(ord, data)

byte = ""
res = ""

length = 474
width = 636

for y in xrange(0, length):
    for x in xrange(0, width*3, 12):
        byte = ""
        for j in range(0,12):
            # Skip 1 byte every 3 bytes
            if ((j+1)% 3 !=0):
                byte = byte + str(data[(x+y*width*3)+j] & 1)
        res += chr(int(byte,2))

sys.stdout.write(res)
```

## Stage 9

col.py

```
import random
import sys
data = open("../congratulations.gif", "r").read()

start = 0x10
end = 0x30a

while start != -1:
    start = data.find("\x00\x00\x00", start+1, end)
    if start != -1:
        a = int(random.random()*255)
        b = int(random.random()*255)
        c = int(random.random()*255)
        data = data[0:start]+chr(a)+chr(b)+chr(c)+data[start+3:]

sys.stdout.write(data)
```