java lsb隐写_LSB隐写工具对比（Stegsolve与zsteg）

原创

非专业de人士 于 2021-03-06 17:40:56 发布 1173 收藏 1
文章标签： java lsb隐写

本文链接：https://blog.csdn.net/weixin_42231085/article/details/115071185

起因

很久很久以前，有一道送分题没做出来，后来看writeup，只要zsteg就行了。

命令运行的结果

root@LAPTOP-GE0FGULA:/mnt/d# zsteg 瞅啥.bmp

[?] 2 bytes of extra data after image end (IEND), offset = 0x269b0e

extradata:0 .. ["x00" repeated 2 times]

imagedata .. text: ["r" repeated 18 times]

b1,lsb,bY ..

b1,msb,bY .. text: "qwxf{you_say_chick_beautiful?}"

b2,msb,bY .. text: "i2,C8&k0."

b2,r,lsb,xY .. text: "UUUUUU9VUUUUUUUUUUUUUUUUUUUUUU"

b2,g,msb,xY .. text: ["U" repeated 22 times]

b2,b,lsb,xY .. text: ["U" repeated 10 times]

b3,g,msb,xY .. text: "V9XDR\d@"

b4,r,lsb,xY .. file: TIM image, Pixel at (4353,4112) Size=12850x8754

b4,g,lsb,xY .. text: "3""""""3###33##3#UDUEEEEEDDUETEDEDDUEEDTEEEUT#!"

b4,g,msb,xY .. text: """""""""""""""""""""""""""""""DDDDDDDDDDDD""""DDDDDDDDDDDD*LD"

b4,b,lsb,xY .. text: "gffffvwgwfgwwfw"

b1,msb,bY读取到的flag，看的一脸懵逼，msb是啥？不是lsb隐写么？bY的b又是啥？我用stegsolve怎么没找到flag？

结论

两个工具的一些参数在理解上有点疑问，因此查看了源码。

Stegsolve的Data Extract功能，Bit Order选项MSBFirst和LSBFirst的区别，这个在扫描顺序中说明

zsteg不理解参数更多

-c：rgba的组合理解，r3g2b3则表示r通道的低3bit，g通道2bit,r通道3bit，如果设置为rbg不加数字的，则表示每个通道读取bit数相同，bit数有-b参数设置

-b：设置每个通道读取的bit数，从低位开始，如果不是顺序的低位开始，则可以使用掩码，比如取最低位和最高位，则可以-b 10000001或者-b 0x81

-o：设置行列的读取顺序，xy就是从上到下，从左到右，xy任意有大写的，表示倒序，不过栗子中有个bY令我费解，查看源码知道对于BMP的图片，可以不管通道，直接按字节读取，就是b的意思了，b再顺带表示x，也就是bY的顺序和xY是一样的，Yb和Yx的顺序是一样的，但是b这个的读取模式跟-c bgr -o xY好像是一样的(因为看BMP图片通道排列顺序是BGR)，不太理解专门弄个这个出来干嘛。

--msb和--lsb这个在组合顺序中说明

扫描顺序

行列顺序

先说下行列的扫描顺序

zsteg可以通过-o选项设置的8种组合(xy,xY,Xy,XY,yx,yX,Yx,YX),个人认为常用的就xy和xY吧

Stegsolve只有选项设置Extract By Row or Column，对应到zsteg的-o选项上就是xy和yx

字节顺序

然后是字节上的扫描顺序，因为是读取的bit再拼接数据的，那么一个字节有8bit数据，从高位开始读还是从低位开始读的顺序

Stegsolve：字节上的读取顺序与Bit Order选项有关，如果设置了MSBFirst，是从高位开始读取，LSBFirst是从低位开始读取

zsteg：只能从高位开始读，比如-b 0x81，在读取不同通道数据时，都是先读取一个字节的高位，再读取该字节的低位。对应到Stegsolve就是MSBFirst的选项。

组合顺序

对于Stegsolve和zsteg，先读取到bit数据都是先拿出来组合的，每8bit组合成一个字节，按照最先存放的Bit在低地址理解的话。

zsteg的--lsb和--msb决定了组合顺序

--lsb：大端存放

--msb：小端存放

源码片段,a内存储的是读取的Bit数据，所以msb是低地址的是低位，因此是小端存放。

if a.size >= 8

byte = 0

if params[:bit_order] == :msb

8.times{ |i| byte |= (a.shift<

else

8.times{ |i| byte |= (a.shift<

end

Stegsolve则是只有大端存放，即对应zsteg的—lsb，因为代码中有个extractBitPos变量，初始值是128，每组合1bit，就右移一次，到0后循环。

源码片段

```
private void addBit(int num)

{

if(num!=0)

{

extract[extractBytePos]+=extractBitPos;

}

extractBitPos>>=1;

if(extractBitPos>=1)

return;

extractBitPos=128;

extractBytePos++;

if(extractBytePos

extract[extractBytePos]=0;

}
```

Stegsolve

了解一下Data Extract以及不同通道存储图片的隐写

Data Extract

功能简要说明

面板

配置选项后，是通过Preview按钮进行数据的读取，因此直接跟进该按钮事件。

Bit Planes：选取通道要读取的bit位。

Bit Plane Order：一个像素值包含多个通道，不同通道的读取数据，Alpha一直是最先读的，然后会根据该项的配置决定读取顺序。

Bit Order：读取数据时，每次仅读取1Bit，该项是控制读取一个通道字节数时，读取的方向，MSBFirst表示从高位读取到低位，LSBFirst表示从低位读取到高位。因此只有当通道勾选的Bit个数大于1时，该选项才会影响返回的结果。

代码分析

文件：Extract.java

按钮事件：

```
/**

* Generate the extract and generate the preview

* @param evt Event

*/

private void previewButtonActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_previewButtonActionPerformed

generateExtract();

generatePreview();

}//GEN-LAST:event_previewButtonActionPerformed
```

跟进generateExtract()，存在内部调用，先列举了另外两个方法。

```
/**

* Retrieves the mask from the bits selected on the form

*/

/*读取Bit Planes的配置，图片getRGB会返回一个整型，如果存在alpha，那么范围最大值就是0xffffffff，从高位至低位，每一个字节按顺序对应为 A R G B，所以getMask就是获取要获取对应Bit的掩码，存为this.mask，this.maskbits记录是全部要读取的Bit数。

*/

private void getMask()

{

mask = 0;

maskbits = 0;

if(ab7.isSelected()) { mask += 1<<31; maskbits++;}

if(ab6.isSelected()) { mask += 1<<30; maskbits++;}

if(ab5.isSelected()) { mask += 1<<29; maskbits++;}

if(ab4.isSelected()) { mask += 1<<28; maskbits++;}

if(ab3.isSelected()) { mask += 1<<27; maskbits++;}

if(ab2.isSelected()) { mask += 1<<26; maskbits++;}

if(ab1.isSelected()) { mask += 1<<25; maskbits++;}

if(ab0.isSelected()) { mask += 1<<24; maskbits++;}

if(rb7.isSelected()) { mask += 1<<23; maskbits++;}
```

```
if(rb6.isSelected()) { mask += 1<<22; maskbits++;}
if(rb5.isSelected()) { mask += 1<<21; maskbits++;}
if(rb4.isSelected()) { mask += 1<<20; maskbits++;}
if(rb3.isSelected()) { mask += 1<<19; maskbits++;}
if(rb2.isSelected()) { mask += 1<<18; maskbits++;}
if(rb1.isSelected()) { mask += 1<<17; maskbits++;}
if(rb0.isSelected()) { mask += 1<<16; maskbits++;}
if(gb7.isSelected()) { mask += 1<<15; maskbits++;}
if(gb6.isSelected()) { mask += 1<<14; maskbits++;}
if(gb5.isSelected()) { mask += 1<<13; maskbits++;}
if(gb4.isSelected()) { mask += 1<<12; maskbits++;}
if(gb3.isSelected()) { mask += 1<<11; maskbits++;}
if(gb2.isSelected()) { mask += 1<<10; maskbits++;}
if(gb1.isSelected()) { mask += 1<<9; maskbits++;}
if(gb0.isSelected()) { mask += 1<<8; maskbits++;}
if(bb7.isSelected()) { mask += 1<<7; maskbits++;}
if(bb6.isSelected()) { mask += 1<<6; maskbits++;}
if(bb5.isSelected()) { mask += 1<<5; maskbits++;}
if(bb4.isSelected()) { mask += 1<<4; maskbits++;}
if(bb3.isSelected()) { mask += 1<<3; maskbits++;}
if(bb2.isSelected()) { mask += 1<<2; maskbits++;}
if(bb1.isSelected()) { mask += 1<<1; maskbits++;}
if(bb0.isSelected()) { mask += 1; maskbits++;}
}
/**
* Retrieve the ordering options from the form
*/
/* 读取Order setting的配置，主要就是rgbOrder的不同值对应的顺序
*/
private void getBitOrderOptions()
{
```

```
if(byRowButton.isSelected()) rowFirst = true;
else rowFirst = false;
if(LSBButton.isSelected()) lsbFirst = true;
else lsbFirst = false;
if(RGBButton.isSelected()) rgbOrder = 1;
else if (RBGButton.isSelected()) rgbOrder = 2;
else if (GRBButton.isSelected()) rgbOrder = 3;
else if (GBRButton.isSelected()) rgbOrder = 4;
else if (BRGButton.isSelected()) rgbOrder = 5;
else rgbOrder = 6;
}
/**
* Generates the extract from the selected options
*/
private void generateExtract()
{
getMask();//获取掩码，每个像素值要获取的对应Bit的掩码，以及每个像素值获取Bit的个数。
getBitOrderOptions();//获取Order settings
int len = bi.getHeight() * bi.getWidth();//获取总的像素点
len = len * maskbits; // 总的像素点*每个像素点获取的Bit数=总的Bit数
len = (len +7)/8; // 总的Bit数转换到总的字节数，+7是没满一个字节的Bit数也对应到一个字节。(极端点比如总的Bit数就1~7Bit,也是要转为1字节，所以需要+7)
extract = new byte[len];//存储读取到的字节数据
extractBitPos = 128; // 每8个Bit组成一个字节数据，extractBitPos相当于权值，从128开始，因此读取的每8Bit，先读到的在高位。
extractBytePos = 0;
//System.out.println(bi.getHeight()+" "+bi.getWidth()+" "+len+" "+mask);
// 根据rowFirst参数来选择读取顺序，调用extractBits读取数据
if(rowFirst)
{
for(int j=0;j
for(int i=0;i
```

```
{
//System.out.println(i+" "+j+" "+extractBytePos);
extractBits(bi.getRGB(i, j));
}
}
else
{
for(int i=0;i
for(int j=0;j
extractBits(bi.getRGB(i, j));
}
}
```

读取数据是extractBits，nextByte是读取到的一个像素点的值，如果是lsbFirst(也就是选了Bitorder为LSBFirst，默认是MSBFirst)，则是从低位从高位按顺序读取(每个通道选取2Bit以上才会有影响，如果只读取1Bit则无所谓了)。

栗子：读取alpha通道，lsbFirst，extract8Bits(nextByte,1<<24)，掩码是从24位开始，依次左移1位，左移8次；msbFirst，extract8Bits(nextByte,1<<31)，掩码是从31位开始，依次右移，右移8次。

```
/**
* Extract bits from the given byte taking account of
* the options selected
* @param nextByte the byte to extract bits from
*/
private void extractBits(int nextByte)
{
if(lsbFirst)
{
extract8Bits(nextByte,1<<24);
switch(rgbOrder)
{
case 1: //rgb
extract8Bits(nextByte,1<<16);
extract8Bits(nextByte,1<<8);
```

```
extract8Bits(nextByte,1);
break;
case 2: //rbg
extract8Bits(nextByte,1<<16);
extract8Bits(nextByte,1);
extract8Bits(nextByte,1<<8);
break;
case 3: //grb
extract8Bits(nextByte,1<<8);
extract8Bits(nextByte,1<<16);
extract8Bits(nextByte,1);
break;
case 4: //gbr
extract8Bits(nextByte,1<<8);
extract8Bits(nextByte,1);
extract8Bits(nextByte,1<<16);
break;
case 5: //brg
extract8Bits(nextByte,1);
extract8Bits(nextByte,1<<16);
extract8Bits(nextByte,1<<8);
break;
case 6: //bgr
extract8Bits(nextByte,1);
extract8Bits(nextByte,1<<8);
extract8Bits(nextByte,1<<16);
break;
}
}
else
{
```

```
extract8Bits(nextByte,1<<31);
switch(rgbOrder)
{
case 1: //rgb
extract8Bits(nextByte,1<<23);
extract8Bits(nextByte,1<<15);
extract8Bits(nextByte,1<<7);
break;
case 2: //rbg
extract8Bits(nextByte,1<<23);
extract8Bits(nextByte,1<<7);
extract8Bits(nextByte,1<<15);
break;
case 3: //grb
extract8Bits(nextByte,1<<15);
extract8Bits(nextByte,1<<23);
extract8Bits(nextByte,1<<7);
break;
case 4: //gbr
extract8Bits(nextByte,1<<15);
extract8Bits(nextByte,1<<7);
extract8Bits(nextByte,1<<23);
break;
case 5: //brg
extract8Bits(nextByte,1<<7);
extract8Bits(nextByte,1<<23);
extract8Bits(nextByte,1<<15);
break;
case 6: //bgr
extract8Bits(nextByte,1<<7);
extract8Bits(nextByte,1<<15);
```

extract8Bits(nextByte,1<<23);

break;

}

}

}

extract8Bits方法，针对每个通道是要单独调用一次的，nextByte是读取的一个像素点的数据，bitMask是对应通道的掩码(根据extractBits方法的说明可知，如果是lsbFirst则是对应通道掩码的最低位，msbFirst则是对应通道掩码的最高位)，在extract8Bits方法最后也有根据是lsbFirst的值选择是左移还是右移，循环8次。

bitMask循环，与this.mask与，如果不为0，说明是要读取的bit，此时就将nextByte与bitMask想与，把该bit的值存入extract

```
/**

* Examine 8 bits and check them against the mask to

* see if any should be extracted

* @param nextByte The byte to be examined

* @param bitMask The bitmask to be applied

*/

private void extract8Bits(int nextByte, int bitMask)

{

for(int i=0;i<8;i++)

{

if((mask&bitMask)!=0)

{

//System.out.println("call "+ mask+" "+bitMask+" "+nextByte);

addBit(nextByte & bitMask);

}

if(lsbFirst)

bitMask<<=1;

else

bitMask>>>=1;

}

}
```

addBit方法,num是读取的像素值与相应bit的掩码相与后的结果，如果不为0，表示那个Bit为1，否则为0，extractBitPos相当于权值，如果为1，就加extractBitPos，然后extractBitPos右移一位，如果为0就不需要加，但每次extractBitPos都是需要右移一位的，如果extractBitPos还是大于1的，说明还没循环过8次，所以就return了，如果不大于1，说明8次了，那么重置extractBitPos为128，extractBytePos+1，新的字节extract[extractBytePos]的初始值为0。

```
/**

* Adds another bit to the extract

* @param num Non-zero if adding a 1-bit

*/

private void addBit(int num)

{

if(num!=0)

{

extract[extractBytePos]+=extractBitPos;

}

extractBitPos>>=1;

if(extractBitPos>=1)

return;

extractBitPos=128;

extractBytePos++;

if(extractBytePos

extract[extractBytePos]=0;

}
```

不同通道读取图片

功能简要说明

首先生成的图片仅是黑白图片，每个像素点的值根据读取的bit位的值，如果为1设置为白色，如果为0设置为黑色。

代码分析

打开图片后，程序主界面上的按钮可以获取不同通道的图片，这里仅讨论Alpha7~0，Red7~0，Green7~0，Blue7~0，也就是每个通道。

在StegSolve.java中定位到按钮方法

```
private void forwardButtonActionPerformed(ActionEvent evt) {

if(bi == null) return;
```

```
transform.forward();
updateImage();
}
private void fileOpenActionPerformed(ActionEvent evt) {
JFileChooser fileChooser = new JFileChooser(System.getProperty("user.dir"));
FileNameExtensionFilter filter = new FileNameExtensionFilter("Images", "jpg", "jpeg", "gif", "bmp", "png");
fileChooser.setFileFilter(filter);
int rVal = fileChooser.showOpenDialog(this);
System.setProperty("user.dir", fileChooser.getCurrentDirectory().getAbsolutePath());
if(rVal == JFileChooser.APPROVE_OPTION)
{
sfile = fileChooser.getSelectedFile();
try
{
bi = ImageIO.read(sfile);
transform = new Transform(bi);
newImage();
}
catch (Exception e)
{
JOptionPane.showMessageDialog(this, "Failed to load file: " +e.toString());
}
}
}
```

主要方法定位到了Transform类，打开文件时初始化，参数是图片的数据。

Transform.java

构造函数，originalImage记录原始图片数据，transform是转换后的数据，先初始化为原始图片数据，transNum的值对应不同的操作。

```
/*
* transforms
* 0 - none
```

```
* 1 - inversion
* 2-9 - alpha planes
* 10-17 - r planes
* 18-25 - g planes
* 26-33 - b planes
* 34 full alpha
* 35 full red
* 36 full green
* 37 full blue
* 38 random color1
* 39 random color2
* 40 random color3
* 41 gray bits
*/
Transform(BufferedImage bi)
{
originalImage = bi;
transform = originalImage;
transNum=0;
}
```

forward方法,，每次点击一次按钮，为加一次transNum,然后根据transNum的值去执行对应的操作。transNum值对应的操作除了注释中的说明，也可以从getText方法中获取，栗子：Alpha plane 0对应的transNum值为9

```
public void forward()
{
transNum++;
if(transNum>MAXTRANS) transNum=0;
calcTrans();
}
public String getText()
{
switch(transNum)
```

```
{
case 0:
return "Normal Image";
case 1:
return "Colour Inversion (Xor)";
case 2:
case 3:
case 4:
case 5:
case 6:
case 7:
case 8:
case 9:
return "Alpha plane " + (9 - transNum);
case 10:
case 11:
case 12:
case 13:
case 14:
case 15:
case 16:
case 17:
return "Red plane " + (17 - transNum);
case 18:
case 19:
case 20:
case 21:
case 22:
case 23:
case 24:
case 25:
```

```
return "Green plane " + (25 - transNum);
case 26:
case 27:
case 28:
case 29:
case 30:
case 31:
case 32:
case 33:
return "Blue plane " + (33 - transNum);
case 34:
return "Full alpha";
case 35:
return "Full red";
case 36:
return "Full green";
case 37:
return "Full blue";
case 38:
return "Random colour map 1";
case 39:
return "Random colour map 2";
case 40:
return "Random colour map 3";
case 41:
return "Gray bits";
default:
return "";
}
}
```

calcTrans方法，是一个switch方法，根据transNum的值调用方法，而我关心的不同通道获取的图片都是调用transfrombit方法，这里仅截取关心的

```
private void calcTrans()

{

switch(transNum)

{

case 2:

transfrombit(31);

return;

case 3:

transfrombit(30);

return;

case 4:

transfrombit(29);

return;

case 5:

transfrombit(28);

return;

case 6:

transfrombit(27);

return;

case 7:

transfrombit(26);

return;

case 8:

transfrombit(25);

return;

case 9:

transfrombit(24);

return;

case 10:
```

```
transfrombit(23);
return;
case 11:
transfrombit(22);
return;
case 12:
transfrombit(21);
return;
case 13:
transfrombit(20);
return;
case 14:
transfrombit(19);
return;
case 15:
transfrombit(18);
return;
case 16:
transfrombit(17);
return;
case 17:
transfrombit(16);
return;
case 18:
transfrombit(15);
return;
case 19:
transfrombit(14);
return;
case 20:
transfrombit(13);
```

```
return;
case 21:
transfrombit(12);
return;
case 22:
transfrombit(11);
return;
case 23:
transfrombit(10);
return;
case 24:
transfrombit(9);
return;
case 25:
transfrombit(8);
return;
case 26:
transfrombit(7);
return;
case 27:
transfrombit(6);
return;
case 28:
transfrombit(5);
return;
case 29:
transfrombit(4);
return;
case 30:
transfrombit(3);
return;
```

```
case 31:

transfrombit(2);

return;

case 32:

transfrombit(1);

return;

case 33:

transfrombit(0);

return;

default:

transform = originalImage;

return;

}

}
```

transfrombit方法，参数d基本就是读取第dbit的数据，根据之前的说明 Alpha 7是getRGB的数据的最高位，第31bit，根据getText方法可以知道Aplpha 7对应的transNum值为2，再看calcTrans的case2就是调用transfrombit(31)。

```
private void transfrombit(int d)

{

transform = new BufferedImage(originalImage.getWidth(), originalImage.getHeight(), BufferedImage.TYPE_INT_RGB);

for(int i=0;i

for(int j=0;j

{

int col=0;

int fcol = originalImage.getRGB(i,j);

if(((fcol>>>d)&1)>0)//右移d个bit位，再取最低位，如果大于0表示对应Bit位为1，那么就设置对应像素值为0xffffff，也就是(255,255,255)，对应白色，如果Bit位为0，则是设置为(0,0,0)，对应为黑色

col=0xffffff;

transform.setRGB(i, j, col);

}

}
```

zsteg

跟进一下代码执行流程，了解各个参数的意义。

入口

程序执行流程的文件

/bin/zsteg

/lib/zsteg.rb run方法

/lib/zsteg/cli/cli.rb run方法，这里会对参数解析，这里截取一些之后需要用到的参数，完整的自行看源码吧，解析完参数后，主要是最后的动态方法调用，@actions=['check']，因此动态调用check方法

```
def run
@actions = []
@options = {
:verbose => 0,
:limit => Checker::DEFAULT_LIMIT,
:order => Checker::DEFAULT_ORDER
}
optparser = OptionParser.new do |opts|
opts.banner = "Usage: zsteg [options] filename.png [param_string]"
opts.separator ""
opts.on("-c", "--channels X", /[rgba,1-8]+/,
"channels (R/G/B/A) or any combination, comma separated",
"valid values: r,g,b,a,rg,bgr,rgba,r3g2b3,..."
) do |x|
@options[:channels] = x.split(',')
# specifying channels on command line disables extra checks
@options[:extra_checks] = false
end
opts.on("-b", "--bits N", "number of bits, single int value or '1,3,5' or range '1-8'",
"advanced: specify individual bits like '00001110' or '0x88'"
) do |x|
a = []
x = '1-8' if x == 'all'
x.split(',').each do |x1|
```

```
if x1['-']
t = x1.split('-')
a << Range.new(parse_bits(t[0]), parse_bits(t[1])).to_a
else
a << parse_bits(x1)
end
end
@options[:bits] = a.flatten.uniq
# specifying bits on command line disables extra checks
@options[:extra_checks] = false
end
opts.on "--lsb", "least significant BIT comes first" do
@options[:bit_order] = :lsb
end
opts.on "--msb", "most significant BIT comes first" do
@options[:bit_order] = :msb
end
opts.on("-o", "--order X", /all|auto|[bxy,]+/i,
"pixel iteration order (default: '#{@options[:order]}')",
"valid values: ALL,xy,yx,XY,YX,xY,Xy,bY,...",
){ |x| @options[:order] = x.split(',') }
if (argv = optparser.parse(@argv)).empty?
puts optparser.help
return
end
@actions = DEFAULT_ACTIONS if @actions.empty?
argv.each do |arg|
if arg[','] && !File.exist?(arg)
@options.merge!(decode_param_string(arg))
argv.delete arg
end
```

```
end
argv.each_with_index do |fname,idx|
if argv.size > 1 && @options[:verbose] >= 0
puts if idx > 0
puts "[.] #{fname}".green
end
next unless @img=load_image(@fname=fname)
@actions.each do |action|
if action.is_a?(Array)
self.send(*action) if self.respond_to?(action.first)
else
self.send(action) if self.respond_to?(action)
end
end
end
rescue Errno::EPIPE
# output interrupt, f.ex. when piping output to a 'head' command
# prevents a 'Broken pipe - (Errno::EPIPE)' message
end
```

/lib/zsteg/cli/cli.rb check方法

```
def check Checker.new(@img, @options).check end
```

/lib/zsteg/checker.rb initialize方法，初始化一些成员变量，@extractor也是传入了图像数据的，通道判断了图片属性是否有alpha通道。

```
def initialize image, params = {}
@params = params
@cache = {}; @wastitles = Set.new
@image = image.is_a?(ZPNG::Image) ? image : ZPNG::Image.load(image)
@extractor = Extractor.new(@image, params)
@channels = params[:channels] ||
if @image.alpha_used?
%w'r g b a rgb bgr rgba abgr'
```

else

%w'r g b rgb bgr'

end

@verbose = params[:verbose] || -2

@file_cmd = FileCmd.new

@results = []

@params[:bits] ||= DEFAULT_BITS

@params[:order] ||= DEFAULT_ORDER

@params[:limit] ||= DEFAULT_LIMIT

if @params[:min_str_len]

@min_str_len = @min_wholetext_len = @params[:min_str_len]

else

@min_str_len = DEFAULT_MIN_STR_LEN

@min_wholetext_len = @min_str_len - 2

end

@strings_re = /[x20-x7ernt]{#@min_str_len,}/

@extra_checks = params.fetch(:extra_checks, DEFAULT_EXTRA_CHECKS)

end

/lib/zsteg/checker.rb check方法,截取部分，会判断图片是否是bmp的，只有bmp的-o选项内才有b，如果设置为all也只是多了bY的选项，但是通过之后代码分析是可以by yb Yb的。判断order中是否有b用的是正则，因此大小写一样。接着数据读取就到check_channels方法了。

def check

@found_anything = false

@file_cmd.start!

if @image.format == :bmp

case params[:order].to_s.downcase

when /all/

params[:order] = %w'bY xY xy yx XY YX Xy yX Yx'

when /auto/

params[:order] = %w'bY xY'

end

else

```
case params[:order].to_s.downcase
when /all/
params[:order] = %w'xy yx XY YX Xy yX xY Yx'
when /auto/
params[:order] = 'xy'
end
end
Array(params[:order]).uniq.each do |order|
(params[:prime] == :all ? [false,true] : [params[:prime]]).each do |prime|
Array(params[:bits]).uniq.each do |bits|
p1 = @params.merge :bits => bits, :order => order, :prime => prime
if order[/b/i]
# byte iterator does not need channels
check_channels nil, p1
else
channels.each{ |c| check_channels c, p1 }
end
end
end
end
if @found_anything
print "r" + " "*20 + "r" if @need_cr
else
puts "r[=] nothing :(" + " "*20 # line cleanup
end
if @extra_checks
Analyzer.new(@image).analyze!
end
# return everything found if this method was called from some code
@results
ensure
```

@file_cmd.stop!

end

/lib/zsteg/checker.rb check_channels方法，首先判断是否设置了bit_order，没设置则两个都测试，之后就是区分两种模式了，channels有值的，最后是去的color_extractor.rb，没有值的去的byte_extractor.rb。

color_extractor模式，还要判断channels指定的模式，是就rgb还是会单独指定每个通道读取多少Bit的。确定过每个像素读取多少bit，然后乘以总的像素点除以8确认读取字节数。

byte)extractor模式，nbits是-b参数指定的读取bit数，乘以一行的字节数，再乘以高/8。

show_title title输出当前模式

data = @extractor.extract p1读取数据

def check_channels channels, params

unless params[:bit_order]

check_channels(channels, params.merge(:bit_order => :lsb))

check_channels(channels, params.merge(:bit_order => :msb))

return

end

p1 = params.clone

 # number of bits

 # equals to params[:bits] if in range 1..8

 # otherwise equals to number of 1's, like 0b1000_0001

nbits = p1[:bits] <= 8 ? p1[:bits] : (p1[:bits]&0xff).to_s(2).count("1")

show_bits = true

 # channels is a String

if channels

p1[:channels] =

if channels[1] && channels[1] =~ /AdZ/

 # 'r3g2b3'

a=[]

cbits = 0

(channels.size/2).times do |i|

*a << (t=channels[i*2,2])*

cbits += t[1].to_i

end

```
show_bits = false
@max_hidden_size = cbits * @image.width
a
else
# 'rgb'
a = channels.chars.to_a
@max_hidden_size = a.size * @image.width * nbits
a
end
# p1[:channels] is an Array
elsif params[:order] =~ /b/i
# byte extractor
@max_hidden_size = @image.scanlines[0].decoded_bytes.size * nbits
else
raise "invalid params #{params.inspect}"
end
@max_hidden_size *= @image.height/8
bits_tag =
if show_bits
if params[:bits] > 0x100
if params[:bits].to_s(2) =~ /(1{1,8})$/
# mask => number of bits
"b#{$1.size}"
else
# mask
"b#{(params[:bits]&0xff).to_s(2)}"
end
else
# number of bits
"b#{params[:bits]}"
end
```

end

title = [

bits_tag,

channels,

params[:bit_order],

params[:order],

params[:prime] ? 'prime' : nil

].compact.join(',')

return if @wastitles.include?(title)

@wastitles << title

show_title title

p1[:title] = title

data = @extractor.extract p1

if p1[:invert]

data.size.times{ |i| data.setbyte(i, data.getbyte(i)^0xff) }

end

@need_cr = !process_result(data, p1) # carriage return needed?

@found_anything ||= !@need_cr

end

/lib/zsteg/extractor.rb 根据-o选项中是否包含b选择不同模式

def extract params = {}

@limit = params[:limit].to_i

*@limit = 2**32 if @limit <= 0*

if params[:order] =~ /b/i

byte_extract params

else

color_extract params

end

end

在分类说明两个模式的时候，先将一个方法拿出来做个说明，bit_indexes

bit_indexes

通过代码可以知道，在扫描一个字节的时候，zsteg是固定的从高位扫描至低位的

```
def bit_indexes bits
if (1..8).include?(bits)
# number of bits
# 1 => [0]
# ...
# 8 => [7,6,5,4,3,2,1,0]
bits.times.to_a.reverse
else
# mask
mask = bits & 0xff
r = []
8.times do |i|
r << i if mask[i] == 1
end
r.reverse
end
end
```

byte_extract

/lib/zsteg/extractor/byte_extractor.rb data列表是用于存储字节数据，a是用于存储bit数据。

通过byte_iterator方法遍历每个字节，会根据order参数是否有小写b，决定x方向的正序还是倒序，是否有小写y决定y方向的正序还是倒序。

根据x,y的值读取到对应字节，然后根据bit_indexes获取的bidx(注定只能高位至低位)去读取对应Bit值

当a.size为8时，就会组成一个字节，根据bit_order的值决定a中的8bit数据是大端还是小端

msb是小端，lsb是大端。

```
module ZSteg
class Extractor
# ByteExtractor extracts bits from each scanline bytes
# actual for BMP+wbStego combination
module ByteExtractor
def byte_extract params = {}
```

```
bidxs = bit_indexes params[:bits]

if params[:prime]

pregenerate_primes(

:max => @image.scanlines[0].size * @image.height,

:count => (@limit*8.0/bidxs.size).ceil

)

end

data = ''.force_encoding('binary')

a = [0]*params[:shift].to_i # prepend :shift zero bits

byte_iterator(params) do |x,y|

sl = @image.scanlines[y]

value = sl.decoded_bytes.getbyte(x)

bidxs.each do |bidx|

a << value[bidx]

end

if a.size >= 8

byte = 0

if params[:bit_order] == :msb

8.times{ |i| byte |= (a.shift<

else

8.times{ |i| byte |= (a.shift<

end

#printf "[d] %02x %08bn", byte, byte

data << byte.chr

if data.size >= @limit

print "[limit #@limit]".gray if @verbose > 1

break

end

end

end

if params[:strip_tail_zeroes] != false && data[-1,1] == "x00"
```

```
oldsz = data.size
data.sub!(/x00+Z/,'')
print "[zerotail #{oldsz-data.size}]".gray if @verbose > 1
end
data
end
# 'xy': b=0,y=0; b=1,y=0; b=2,y=0; ...
# 'yx': b=0,y=0; b=0,y=1; b=0,y=2; ...
# ...
# 'xY': b=0, y=MAX; b=1, y=MAX; b=2, y=MAX; ...
# 'XY': b=MAX,y=MAX; b=MAX-1,y=MAX; b=MAX-2,y=MAX; ...
def byte_iterator params
type = params[:order]
if type.nil? || type == 'auto'
type = @image.format == :bmp ? 'bY' : 'by'
end
raise "invalid iterator type #{type}" unless type =~ /A(by|yb)Z/i
sl0 = @image.scanlines.first
# XXX don't try to run it on interlaced PNGs!
x0,x1,xstep =
if type.index('b')
[0, sl0.decoded_bytes.size-1, 1]
else
[sl0.decoded_bytes.size-1, 0, -1]
end
y0,y1,ystep =
if type.index('y')
[0, @image.height-1, 1]
else
[@image.height-1, 0, -1]
end
```

```
# cannot join these lines from ByteExtractor and ColorExtractor into
# one method for performance reason:
# it will require additional yield() for EACH BYTE iterated
if type[0,1].downcase == 'b'
# ROW iterator
if params[:prime]
idx = 0
y0.step(y1,ystep){ |y| x0.step(x1,xstep){ |x|
yield(x,y) if @primes.include?(idx)
idx += 1
}}
else
y0.step(y1,ystep){ |y| x0.step(x1,xstep){ |x| yield(x,y) }}
end
else
# COLUMN iterator
if params[:prime]
idx = 0
x0.step(x1,xstep){ |x| y0.step(y1,ystep){ |y|
yield(x,y) if @primes.include?(idx)
idx += 1
}}
else
x0.step(x1,xstep){ |x| y0.step(y1,ystep){ |y| yield(x,y) }}
end
end
end
end
end
end
color_extractor
```

/lib/zsteg/extractor/color_extractor.rb data列表是用于存储字节数据，a是用于存储bit数据。

通过coord_iterator方法遍历每个字节，会根据order参数是否有小写x，决定x方向的正序还是倒序，是否有小写y决定y方向的正序还是倒序。

根据x,y的值读取到对应字节，然后根据bit_indexes获取的ch_masks(注定只能高位至低位)去读取对应Bit值，只是还要根据channel的值，如果是单个字符，表示读取的bit数是通过-b设置的，因此传入params[:bits]，否则就是2个字符，读取第2个字符表示读取的bit数。

当a.size为8时，就会组成一个字节，根据bit_order的值决定a中的8bit数据是大端还是小端 msb是小端，lsb是大端。

```
module ZSteg

class Extractor

# ColorExtractor extracts bits from each pixel's color

module ColorExtractor

def color_extract params = {}

channels = Array(params[:channels])

#pixel_align = params[:pixel_align]

ch_masks = []

case channels.first.size

when 1

# ['r', 'g', 'b']

channels.each{ |c| ch_masks << [c[0], bit_indexes(params[:bits])] }

when 2

# ['r3', 'g2', 'b3']

channels.each{ |c| ch_masks << [c[0], bit_indexes(c[1].to_i)] }

else

raise "invalid channels: #{channels.inspect}" if channels.size != 1

t = channels.first

if t =~ /A[rgba]+Z/

return color_extract(params.merge(:channels => t.split('')))

end

raise "invalid channels: #{channels.inspect}"

end

# total number of bits = sum of all channels bits

nbits = ch_masks.map{ |x| x[1].size }.inject(&:+)
```

```
if params[:prime]
pregenerate_primes(
:max => @image.width * @image.height,
:count => (@limit*8.0/nbits/channels.size).ceil
)
end
data = ''.force_encoding('binary')
a = [0]*params[:shift].to_i # prepend :shift zero bits
catch :limit do
coord_iterator(params) do |x,y|
color = @image[x,y]
ch_masks.each do |c,bidxs|
value = color.send(c)
bidxs.each do |bidx|
a << value[bidx]
end
end
#p [x,y,a.size,a]
while a.size >= 8
byte = 0
#puts a.join
if params[:bit_order] == :msb
8.times{ |i| byte |= (a.shift<
else
8.times{ |i| byte |= (a.shift<
end
#printf "[d] %02x %08bn", byte, byte
data << byte.chr
if data.size >= @limit
print "[limit #@limit]".gray if @verbose > 1
throw :limit
```

```
end
#a.clear if pixel_align
end
end
end
if params[:strip_tail_zeroes] != false && data[-1,1] == "x00"
oldsz = data.size
data.sub!(/x00+Z/,'')
print "[zerotail #{oldsz-data.size}]".gray if @verbose > 1
end
data
end
# 'xy': x=0,y=0; x=1,y=0; x=2,y=0; ...
# 'yx': x=0,y=0; x=0,y=1; x=0,y=2; ...
# ...
# 'xY': x=0, y=MAX; x=1, y=MAX; x=2, y=MAX; ...
# 'XY': x=MAX,y=MAX; x=MAX-1,y=MAX; x=MAX-2,y=MAX; ...
def coord_iterator params
type = params[:order]
if type.nil? || type == 'auto'
type = @image.format == :bmp ? 'xY' : 'xy'
end
raise "invalid iterator type #{type}" unless type =~ /A(xy|yx)Z/i
x0,x1,xstep =
if type.index('x')
[0, @image.width-1, 1]
else
[@image.width-1, 0, -1]
end
y0,y1,ystep =
if type.index('y')
```

```
[0, @image.height-1, 1]
else
[@image.height-1, 0, -1]
end
# cannot join these lines from ByteExtractor and ColorExtractor into
# one method for performance reason:
# it will require additional yield() for EACH BYTE iterated
if type[0,1].downcase == 'x'
# ROW iterator
if params[:prime]
idx = 0
y0.step(y1,ystep){ |y| x0.step(x1,xstep){ |x|
yield(x,y) if @primes.include?(idx)
idx += 1
}}
else
y0.step(y1,ystep){ |y| x0.step(x1,xstep){ |x| yield(x,y) }}
end
else
# COLUMN iterator
if params[:prime]
idx = 0
x0.step(x1,xstep){ |x| y0.step(y1,ystep){ |y|
yield(x,y) if @primes.include?(idx)
idx += 1
}}
else
x0.step(x1,xstep){ |x| y0.step(y1,ystep){ |y| yield(x,y) }}
end
end
end
```

end

end

end

结果

起因里的zsteg的参数现在都解释过了，而用stegsolve没有看到flag是因为8bit数据是按照大端模式组成的字节，而flag是需要以小端模式组成，所以当我选择stegsolve来做题时，注定是拿不到flag了，都是时辰的错。

然后bY其实和xY的结果是一样的，只是要确定通道的排列方式，bmp按顺序存的通道顺序是bgr。

```
root@LAPTOP-GE0FGULA:/mnt/d# zsteg -c bgr -o xY --msb -b1 瞅啥.bmp

[?] 2 bytes of extra data after image end (IEND), offset = 0x269b0e

b1,bgr,msb,xY .. text: "qwxf{you_say_chick_beautiful?}"
```

再来看下stegsolve，首先知道-o是Y，因此图片需要倒一下，所以手动修改bmp的高度为原值的负值，图片就倒过来了。

选中的序列和flag的值，生成二进制序列对比一下，应是每8个bit都是倒序的。

```
#encoding:utf-8

from binascii import b2a_hex,a2b_hex

flag = "qwxf{you_say_chick_beautiful?}"

stegsolve = "8eee1e66de9ef6aeface869efac61696c6d6fa46a686ae2e9666ae36fcbe"

flag = bin(int(b2a_hex(flag),16))[2:]

stegsolve = bin(int(stegsolve,16))[2:]

def show(a,b):

if len(a) % 2 != 0:

a = '0'+a

if len(b) % 2 != 0:

b = '0'+b

for i in xrange(0,len(a),8):

print a[i:i+8]+" "+b[i:i+8]

show(flag,stegsolve)
```

自己看下结果吧。