Introduction to Computer Security
MPCS 56511, Winter 2019
Final Project

Contents:

Group Members:

Alex Michael
Dipanshu Sehjal
Paul Chang

The final deliverable package is available alongside this document in the same repository.

The development code can be found at Github repo:

https://github.com/amichael7/python-stegosploit/tree/master/project-stegosploit

Introduction to Computer Security
MPCS 56511, Winter 2019
Final Project

Stegosploit

Exploit Delivery with Steganography and Polyglots

Introduction

Description:

Our project implements a technique used for image-based exploit delivery with two broad underlying techniques–***Steganography*** (practice of concealing a file, message, image, or video within another file, message, image, or video) and ***Polyglot*** (multiple data formats on a single container co-exist without breaking other's specs or syntax). Dubbed as Stegosploit, it creates a way to encode drive-by (malicious content may be able to exploit vulnerabilities in the browser to run malicious code without the user's knowledge) browser exploit and deliver them through image files. These payloads go undetectable into the browser. Drive-by browser exploits are steganographically encoded into JPG images. The resultant image file is fused with HTML and JavaScript decoder code, turning it into an HTML + Image polyglot (ImaJS). The polyglot looks and feels like an image, but is decoded and triggered in a victim's browser when loaded. Using Stegosploit, it is possible to transform virtually any Javascript based browser exploit into a JPG or PNG image.

We extended this utility with Python for creation of a polyglot. The Python script packs the **steganographically encoded exploit** and a **decoder** into a single container--polyglot. The polyglot can be used to attack the victim's browser that can be rendered as HTML or executed as JavaScript.

The offensive JavaScript code is encoded into image pixels, each character represented by an 8-bit grayscale pixel in a JPG file, thus exploit code is converted into

an innocent JPG file. The JPG image is then loaded in a browser and decoded using a HTML5 CANVAS. Decoding is performed via JavaScript. The decoder code itself is not detected as being offensive, since it only performs CANVAS pixel manipulation.

Implementation:

Stage 1: **Steganographically encoding the exploit code**

An image is essentially an array of pixels. Each pixel can have three colour channels--red, green and blue. Each channel is represented by an 8-bit value, which provides 256 discrete levels of colour. A black and white image uses the same values for R, G and B channels for each pixel. We can visualise 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.

Encoding the exploit bit stream on higher bit layers will result into significant visual distortion of the resultant image. The goal is to encode the exploit bit stream into lower bit layers, preferably bit layer 2 which comprises of the LSB of all the pixels. The resultant image with the bitstream encoded on layer 2 shows little or no visual aberration.

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.

1. Iterative Encoding for JPG Images

JPG encoders can use variable quality settings. A lower quality setting offers maximum compression. However, the maximum quality setting does not provide us with lossless compression. Certain pixels will still be approximated to their neighbours no matter what. To further minimise 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 3x3 performs much better compared to encoding on every consecutive pixel.

Iterative encoder tool, *iterative_encoding.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.

Stage 2: Creating an HTML + JPG polyglot

The final product of Stegosploit is a single JPG image that will trigger the exploit in the victim's browser.

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. The browser will render the HTML directly when loaded, and execute any embedded Javascript code along the way.

Basic JPG file structure follows the JPEG File Interchange Format (JFIF). JFIF files contain several *segments*, each identified by a 2-byte marker *FF xx* followed by the segment's data. We will add HTML decoder data to *APP0* segment of the JPG. The *APP0* segment immediately follows the *SOI* segment (start of image). Increasing the 2 byte length field of *APP0* gives us extra space at the end of the segment in which to place the HTML decoder data.

2.1 Decoding a Steganographically encoded exploit

A high level overview of the process of triggering the exploit is described below:

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 JavaScript 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.

The HTML decoder template with decoder JavaScript is packed as tightly as possible.

2.2 Decoding the exploit code from pixels

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

1. Bit Layer. It has to match the bit layer used for encoding the bitstream.
2. Encoding Channel.
3. Pixel Grid.

After the *pixel data* from the image is loaded into the ***Canvas***, the decoder JavaScript reconstructs the exploit code bitstream from the pixel values in the encoded bit layer, and the exploit code is reassembled into JavaScript code from the decoded bitstream. This exploit code is then executed as JavaScript.

2.3 Fusing decoder HTML and JavaScript into a JPG--A Polyglot

JPG decoders would have no problem in properly displaying the image contained in the HTML+JPG polyglot because JPG decoders will just ignore the HTML. Browsers, however, would encounter problems when trying to properly render HTML tags. The extra JPG data would end up "polluting" the DOM.

To prevent JPG data from interfering with HTML, we can use a few strategically placed HTML comment delimiters. The HTML decoder code is embedded between blocks of random data in the *APP0* segment starting from offset 0x0014. Anything that's not part of the *HTML decoder code* is put into HTML comments, i.e., comment out the rest of the JPG file data.

The resultant *polyglot* increases in size, successfully embedding the HTML data in the artificially created space at the end of the APP0 segment.

Stage 3: **Exploit Delivery**

The attacker can:

1. Host the image on an attacker controlled web server and send its URL link to the victim.
2. Upload the image on 3rd party websites and provide direct links.
3. Share the exploited image with other means, like, a thumb-drive, etc.

Demonstration

Project structure:

encoding contains tools for encoding an exploit onto an image with the browser. ***interative_encoding.html*** encodes the exploit onto a JPG image.

exploits contains the decoder HTML and the exploit.

images contains the original and encoded JPG images.

polyglots contains polyglots (decoder HTML + JPG).

scripts contains the python scripts to create a polyglot

Usage:

We have tested this exploit on a Windows 7 SP1 VM with IE9 and Firefox 21+
Following demonstration was run on Firefox 56. We will encode a *"JavaScript alert box"* exploit in the image. Victim's browser will pop-up an alert box when the polyglot is loaded in the browser.

Instructions for demonstration:

Step 1: Host a web server. We need a web server to encode images with a browser.

From ***project-stegosploit's*** root directory, execute:

python -m http.server 8000

Step 2: On Firefox 56, navigate to

http://localhost:8000/encoding/iterative_encoding.html

1. Load the image
2. Bit layer = 2
3. Grid = 3

Illustration 1: Load image into the browser. Provide a JS exploit in the text box.

Step 3: Process and encode the exploit in to the image

1. Click on **process** button to process the image.
2. Click on **iterate** button to encode the JPG image iteratively.

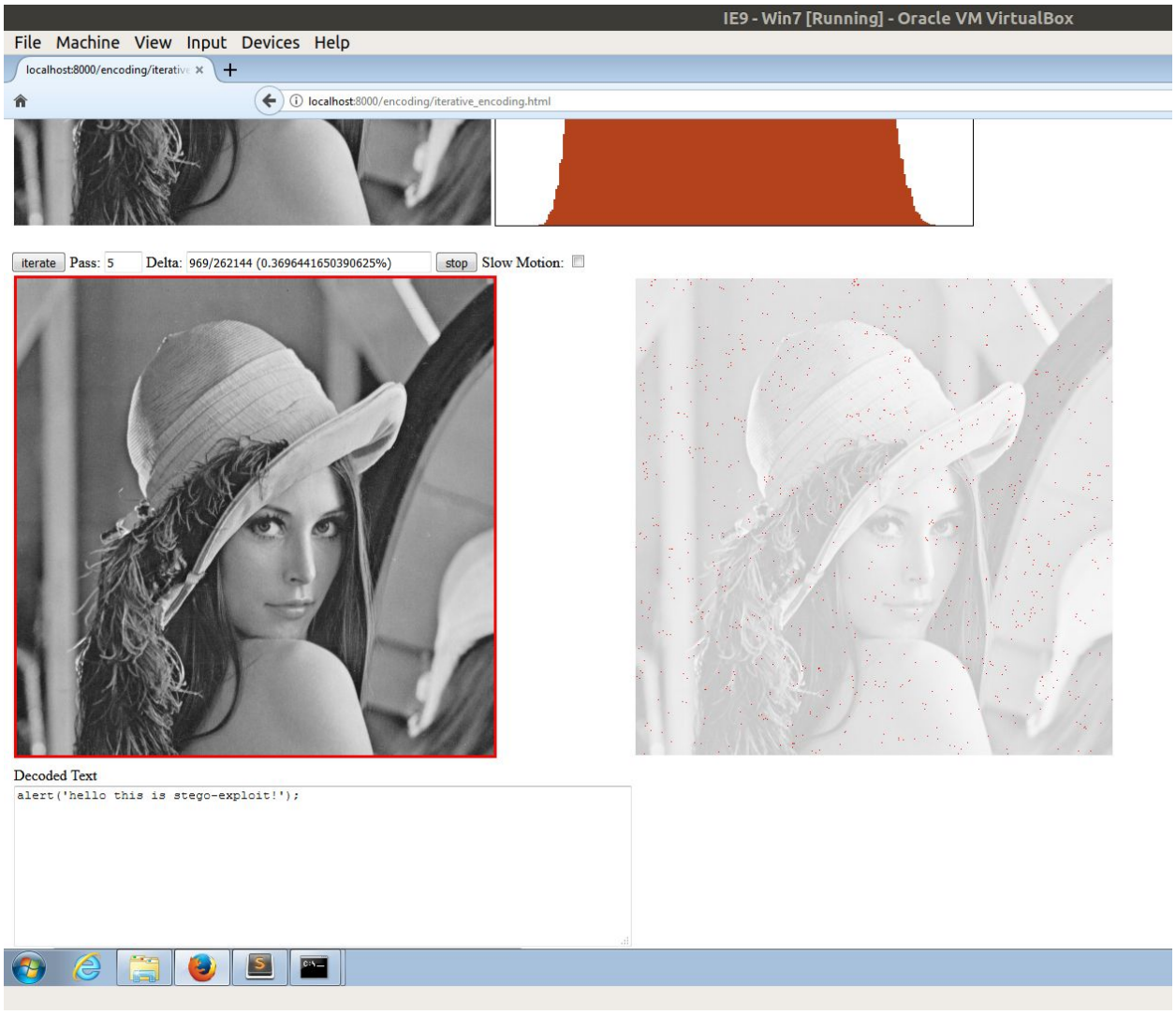

Illustration 2: There are a few pixels that still differ between the source and encoded image, but in this case, they do not contribute to errors in the decoded message.

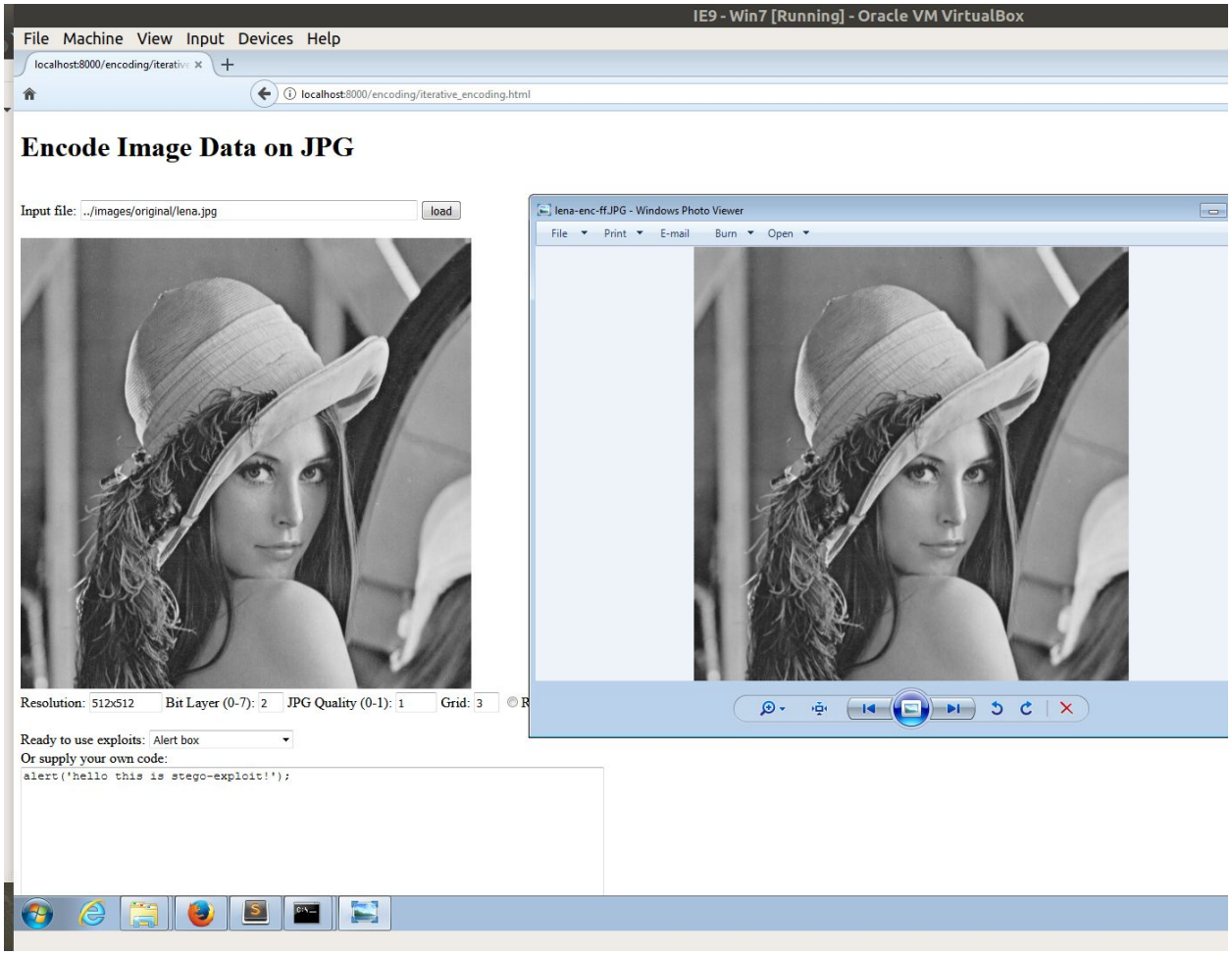

Illustration 3: Comparison b/w original (left) and encoded (right) images. The difference is indiscernible.

Step 4: Create a polyglot

scripts\polyglot_with_jpg.py creates a polyglot.
It takes (1) decoder HTML, (2) encoded JPG, and (3) output polyglot name as arguments.

Execute:

python scripts\polyglot_with_jpg.py exploits\decoder.html images\encoded\lena-enc-ff.JPG polyglots\lena_poly_demo.html

The resultant polyglot will be saved under polyglots\ directory. We add an HTML extension to the polyglot because the browser has to sniff the content as HTML and render it as an HTML. We couldn't find a vulnerable browser that would automatically interpret the content as HTML.

Step 5:

In the same browser, navigate to ***http://localhost:8000/polyglots/***

Click on the polyglot. In the above demonstration it is ***lena_poly_demo.html***

Click on the image and the offensive JavaScript is triggered. Alert box appears!

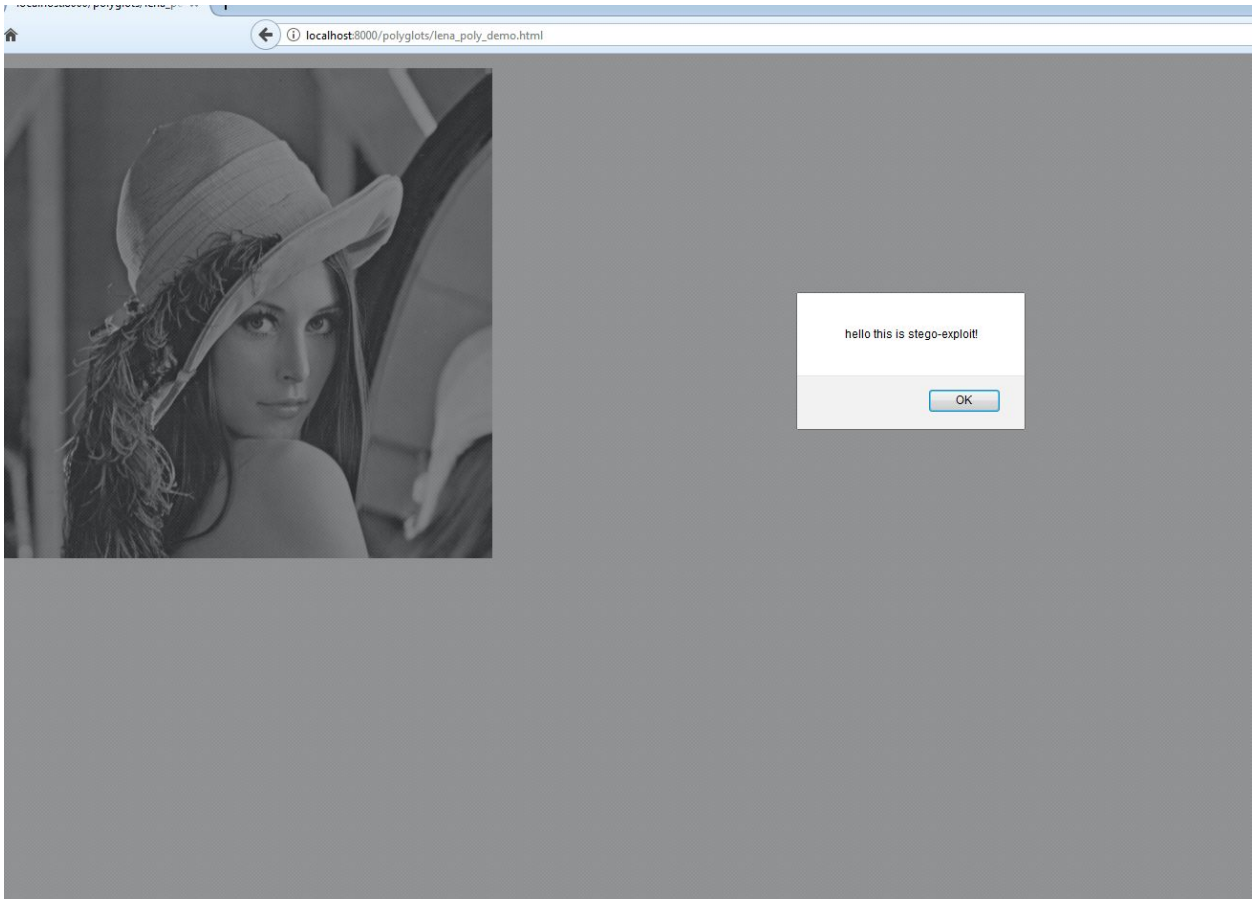

Illustration 4: Exploit triggered upon clicking the image.