OUT-OF-BAND FILE TRANSFER ON CLOSED NETWORKS

An Insider’s Options

Michael Rich
mike@tofet.net
Overview

- The Challenge
- The Tools Available
- Phase 0 – Set up
- Phase 1 – Hex Attack
- Phase 2 – Attack of the Big Barcode
- Bringing it all Together
- Future/Branch Research Paths
- Conclusion
The Challenge

There I was, hacking the collaboration portal..
How could I intercept the POST call to modify the inputs?
  - Tamper Data, Burp Suite, etc..

How could I forge the POST call?
  - Curl, wget, etc..

Eventually: “How could I load one of these tools on to my closed, secure network without getting caught?”
The Conditions

- Closed, secured network (sort of)
- USB ports secured & monitored
- CD use secured & monitored
- Host-based security system
- Data transfer entry points do exist (DOTS)
  - Not in control of attacker
  - Unknown scanning rules
  - Leaves logs
- Windows / MS-Office environment
Tools Available

- MS Office = Visual Basic for Applications
- Professional level printers & scanners
- Adobe Acrobat OCR
Phase 0 - Set up

- Put Excel into attack mode
Phase 0 - Set up

Sub EncodeFile()
    ' Declare needed inputs
    Dim filename As String
    Dim targetSheet As String
    Dim maxBytesPerLine As Long

    ' Initialize needed inputs
    filename = "C:\Users\1080119360\Documents\Personal\work\bhat\ocelot.jpg"
    targetSheet = "ocelot"

    maxBytesPerLine should be divisible by 4 to take advantage of the checksum function
    ' Console font in 8pt has about 130 chars available per line, with space for the checksum value
    maxBytesPerLine = 56

    ' Clear worksheet first
    Dim sh As Worksheet
    Set sh = Worksheets(targetSheet)
    sh.Cells.Clear

    ' Open the file (no error catching here, it's a POC)
    Open filename For Binary Access Read As #1
    Dim fileLength As Long
    fileLength = FileLen(filename)

    ' Initialize control variables
    Dim currentLineByteCount As Integer
    currentLineByteCount = 0
    Dim currentRow As Long
    Dim currentRow As Long

End Sub
Phase 0 - Set up

- Consolas Font – Down to 8 pt font
Phase 0 - Set up
Use Phase 0 methods to make Excel a binary file hex encoder/decoder

Why hex?
- Printable text
- Tests showed excellent OCR results

<table>
<thead>
<tr>
<th></th>
<th>Hex Encoding</th>
<th>Base64 Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoding</td>
<td>Consolas, 8 pt font</td>
<td>Consolas, 12 pt font</td>
</tr>
<tr>
<td>Word Length Errors</td>
<td>0 in 73 words</td>
<td>9 in 73 words</td>
</tr>
<tr>
<td>(80 char words)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transcription Errors</td>
<td>0 in 5840 symbols</td>
<td>216 in 5840 symbols</td>
</tr>
<tr>
<td>Error Types</td>
<td>1 to 1</td>
<td>Many to Many</td>
</tr>
</tbody>
</table>
Hex Encoding is good, but probably not perfect
Need compact error detection
2-byte XOR checksum
Phase I – Oops.. Real World

- Assumptions, assumptions, assumptions
- 1551 errors in 135,420 symbols (1.1 % error)
  - B to 8: 261; 1 to l: 359; 5 to S: 864
  - D to 0, O: 57; 6 to G,q,b: 3
- Alternative characters:
  - # for B
  - ? for D
- Auto-replace other major errors
  - “1”, “S”, “.”, “ “
- Add strong visual indicators
- 1 manual correction in 1210 lines of text

Demo Time!
Phase 1 - Hex Attack

- **Pros:**
  - Extremely reliable
  - Can be entered by hand if no scanner

- **Cons:**
  - Low data density: \(~3.6K\) per page best case
  - Common Tools:
    - PowerSploit: 835 kB = 232 pages
    - Mimikatz: 538 kB = 150 pages
  - No exfiltration “compression” advantage
Phase 2

Worked, ~ 49K/page

Fail (online)

Fail (online)

C40  Base256
Phase 2

- ~ 25K data per page
- 60% error correction
- Good features
  - Timing lines
  - Reed-Solomon FEC
- Different design problem
- I can make it better!
Phase 2 - Big Barcode

- Data grid where each pixel represents one bit state (white = 1, black = 0)
- Printed at 72 dpi, get about 88 bytes across
- ~ 85 kB data per page
Finding the timing marks

- Start with raster scan across the image
Reading a Big Barcode

- Finding centers of timing marks
- “Wiggle Fit” from “root” pixel
  - Best mask fit
Timing mark location is very successful:

Once all timing marks found, simply compute a grid of intersections to locate data:
Results:
- 20K of binary data: 189 bytes missed (0.953% error)
- 65K of binary data: 491 bytes missed (0.76% error)
- 72K of ASCII data: 972 bytes missed (1.35% error)
Reed Solomon FEC - Two Types

- **Forward ERASURE Correction**
  - Block 1
  - Block 2
  - Block 3
  - Block 4
  - P1: Block 1
  - P2: Block 2
  - P3: Block 3
  - P4: Block 4

- **Forward ERROR Correction**
  - B1
  - B3
  - B4
  - B6
  - B7

- C1: P1
- C2: P2
- C3: P3
- C4: P4
Reed Solomon encoding

- Codewords can be $2^s$ symbols long, each symbol $s$-bits wide
  - $S = 8$, codeword is 255 symbols; each symbol 8 bits wide
  - $S = 16$, codeword is 65535 symbols; each symbol 16 bits wide
  - Codewords can be less than $n$ symbols long
- Can correct up to “$t$” symbol errors (2 parity symbols required for find and correct each error)

From http://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html
Few open-source error correction libraries
- Those that do are $2^8$ only
Coding an R-S FEC

```python
def gf_poly_eval(p, x):
    y = p[0]
    for i in range(1, len(p)):
        y = gf_mul(y, x) * p[i]
    return y

def rs_generator_poly(nsym):
    g = [1]
    for i in range(0, nsym):
        g = gf_poly_mul(g, [1, g_exp[i]])
    return g

def gf_poly_div(dividend, divisor):
    # Fast polynomial division by using Extended Synthetic Division and optimization
doesn't work with standard polynomials outside of this galois field, see
    msg_out = bytearray(dividend) # Copy the dividend list and pad with 0 when
    for i in range(len(dividend)-(len(divisor)-1)):
        if coef > 0 # 
            msg_out[i+j] = g_mul(divisor[j], coef) # equivalent to the 
            # (but zoring is fast

    # The resulting msg_out contains both the quotient and the remainder, the
    # (the remainder has necessarily the same degree as the divisor -- not le
    # what we couldn't divide from the dividend), so we compute the index whe
    separator = -(len(divisor)-1)
    return msg_out[:separator], msg_out[separator:] # return quotient, remain

def rs_encode_msg(msg_in, nsym):
    # Reed-Solomon main encoding function, using polynomial division (algorithm
    gen = rs_generator_poly(nsym)
    # Init msg_out with the values inside msg_in and pad with len(gen)-1 bytes
    msg_out = [0] * (len(msg_in) + len(gen)-1)
    # Initializing the Synthetic Division with the dividend (= input message p
    msg_out[:len(msg_in)] = msg_in
```

https://en.wikiversity.org/wiki/Reed%E2%80%93Solomon_codes_for_coders
With $s=8$, $k=140$ to work reliably

~47 kB per page of data (~38 kB of parity)

- PowerSpoit: 18 pages (vs. 232 pages in hex)
- Mimikatz: 12 pages (vs. 150 pages in hex)

Demo Time!
Goal: Using techniques described here, install PowerSploit on a machine

<table>
<thead>
<tr>
<th>Step</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpret a page-sized bar code</td>
<td>✔️</td>
</tr>
<tr>
<td>Reed-Solomon Encoder/Decoder</td>
<td>✔️</td>
</tr>
<tr>
<td>Build Sideload Library</td>
<td>✔️</td>
</tr>
<tr>
<td>Encode, Print, Scan, Decode payload with library</td>
<td>✔️</td>
</tr>
<tr>
<td>Print, Scan, and load hex encoder/decoder into Excel</td>
<td>✔️</td>
</tr>
<tr>
<td>Emplace library using hex OCR method</td>
<td>✔️</td>
</tr>
<tr>
<td>Encode/decode using DLL called from Excel</td>
<td>✔️</td>
</tr>
</tbody>
</table>
Future/Branch Research

- Big Bar Code
  - Reduce size of BBC DLL
  - Improve error rates
  - Get $2^{16}$ Reed Solomon FEC working
  - Add color to BBC

- Excel-a-sploit
  - Hex Editor
  - Steganographic encoder/decoder
  - Restore command prompt
  - Direct DLL injection?
Big Bar Code POC was a success
Standard office tools provide a lot of power
If a user can code, a system is not secure
Innocuous input/output systems can be used for creative purposes
QUESTIONS?