

Out-of-Band File Transfer on Closed Systems

An Insider's Options

Michael Rich
@miketofet
mike@tofet.net

Abstract

I present a method of transferring arbitrary binary files to and from an ostensibly closed system using standard office software and readily available office tools while avoiding the use of Internet access or magnetic media. These methods are available to any determined insider. Though the methods are not perfect, I used the techniques presented here in a successful proof-of-concept to deliver a selection of PowerSploit tools to an otherwise clean machine.

The Challenge

One day I was examining an internal information portal at work and discovered that the system was using client-side data validation to prevent XSS attacks. This sort of security method is very easy to circumvent by using standard penetration testing tools such as TamperData or Burp Suite. I really wanted to see what would happen if I could get the XSS test through to the server. But, I was on a closed system and not at liberty to install software even if I could download those tools.

At first I tried to find a way to forge the POST call on my work machine. This machine was your basic Windows box with a desktop full of standard office suites. I identified a couple of candidate options but I was soon distracted by the more generic question: "How can I install those tools that I want to use without alerting anyone?"

By "alerting anyone" I mean avoiding those methods that have been specifically secured and monitored on a high-value, closed network. Without going into detail, this generally precludes the use of any magnetic media or connecting to unapproved web or Internet resources. I consider any path that delivers arbitrary data without triggering an alert on established monitoring systems to be considered "out-of-band".

Disclaimer

Lest my network admins at work have a heart attack and revoke my access, let me be clear: I conceived of the idea at work but did all my development and testing on my home machine. I remain a loyal and faithful adherent to my user agreements.

Resources

Almost all general-purpose office networks, if they are to be of use to an organization, have to have a large variety of standard office software installed. They also are typically connected to a multi-function office machine of some kind. Folks simply have to have these tools available if they want to create, collaborate, publish, and share anything of value.

On my test machine I limited myself to access to a standard Microsoft Office suite, Adobe Acrobat, and a high quality multi-function office device.

Overall Goal and Method

The file transfer method I ended up pursuing was using the printer and scanner connected to my network. In any functioning office, it is perfectly normal for folks to print and scan, sometimes in mass quantities. Print jobs and scanned documents flowing to and from a networked printer are not likely to raise any curious security eyebrows.

I came up with a series of stages to leverage the printer and scanner as my vector for importing and exporting arbitrary binary data from a system. Each stage consists of printing data to a page from a source system, scanning this page on a scanner attached to closed network, and interpreting the scan on the target system to transfer the arbitrary binary code to the system. This process can also be reversed to remove arbitrary data from the target system as well.

For my proof-of-concept I started by printing and scanning raw text-based code snippets, moved on to interpreting a page of hex-encoded data, and finished by generating and using a page-sized bar code. Each stage increased the amount of data that could be packed onto single page of paper until I got to the point where I could use the method to deliver a set of PowerSploit tools to a closed workstation with only three printed and scanned pages

Stage 0: Get Microsoft Excel into attack mode

"Attack mode" in Microsoft Excel means enabling the Visual Basic for Applications (VBA) development environment. This is built into Excel, but typically not enabled by default. It is trivial to turn it on though. A user simply clicks on "Options" under the "File" Menu, selects "Customize Ribbon", checks the box next to "Developer", and clicks "Ok". This enables a "Developer" tab on the ribbon bar with a "Visual Basic" icon on the far left. Clicking that button opens the VBA development environment.

Why is this attack mode? For three main reasons: VBA can create and run arbitrary code, VBA can modify arbitrary files at the byte level, and VBA can execute arbitrary functions in arbitrary DLL files with inputs of the attacker's choosing. First, with access to VBA an attacker can now run arbitrary code. VBA is a fully fledged, if somewhat annoying, programming language. While doing this research I reinforced a possible law of security engineering: if a user can code, the system isn't secure. It's always been known that specialized users, such as programmers with access to code compilers, are a threat to a system's security. So, we watch them a little more carefully. But every user of a general-purpose office environment has access to a full Integrated Development Environment at all times. Odds are the organization lacks the manpower to watch every user to the same level of fidelity they lavish upon their specialized users. This can create a security seam a malicious insider can exploit.

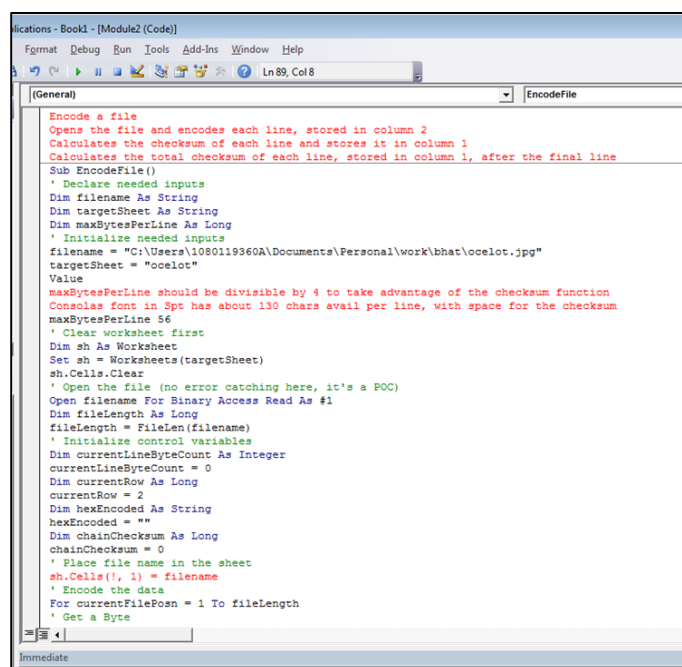
The second reason this puts Excel into attack mode is that VBA allows byte by byte binary reading and writing of arbitrary files. If the user has access to read a file they can basically do anything they want to it at that point. Of course even a minimally secured machine shouldn't allow the users to modify an important system file, but they can always copy it and modify their copy. This sort of functionality can be leveraged to circumvent security in ways that are only limited by the attacker's imagination and capability.

The third reason is VBA's ability to make direct calls to arbitrary functions in arbitrary DLLs with arbitrary inputs. VBA can also call an executable file with any arbitrary file name; the file does not have to end with a ".exe" or ".dll" extension. That's a whole lot of arbitrary to take advantage of. I use it here to eventually run my custom DLL but the bottom line is VBA exposes a lot of attack surface to take advantage of and that's why I call entering Developer mode "putting Excel into attack mode".

Stage 1: VBA script import

Now that VBA is active, we need it to run code we have carefully written and tested in our lab. This is actually quite straightforward. We simply print the script file in the lab, scan it on the target network, use Adobe Acrobat Optical Character Recognition to turn that scan back in to code, and cut and paste the code back into VBA.

This won't be perfect, of course. In the OCR process, Adobe will frequently drop the comment delimiter, making for bad lines of code. It also likes to drop the "=" sign, turning assignment operations into a statement that VBA may interpret as a kind of function call. Luckily for us, VBA is quite likely to highlight any egregious syntax errors in red as shown below:



The screenshot shows the VBA Editor window titled "Applications - Book1 - [Module2 (Code)]". The code is written in VBA and contains several syntax errors highlighted in red. The errors are as follows:

- Line 1: `Encode a file` (comment)
- Line 2: `Opens the file and encodes each line, stored in column 2` (comment)
- Line 3: `Calculates the checksum of each line and stores it in column 1` (comment)
- Line 4: `Calculates the total checksum of each line, stored in column 1, after the final line` (comment)
- Line 5: `Sub EncodeFile()`
- Line 6: `' Declare needed inputs`
- Line 7: `Dim filename As String`
- Line 8: `Dim targetSheet As String`
- Line 9: `Dim maxBytesPerLine As Long`
- Line 10: `' Initialize needed inputs`
- Line 11: `filename = "C:\Users\1080119360A\Documents\Personal\work\bhat\ocelot.jpg"`
- Line 12: `targetSheet = "ocelot"`
- Line 13: `Value`
- Line 14: `maxBytesPerLine should be divisible by 4 to take advantage of the checksum function` (comment)
- Line 15: `Consoles font in Spt has about 130 chars avail per line, with space for the checksum` (comment)
- Line 16: `maxBytesPerLine 56`
- Line 17: `' Clear worksheet first`
- Line 18: `Dim sh As Worksheet`
- Line 19: `Set sh = Worksheets(targetSheet)`
- Line 20: `sh.Cells.Clear`
- Line 21: `' Open the file (no error catching here, it's a POC)`
- Line 22: `Open filename For Binary Access Read As #1`
- Line 23: `Dim fileLength As Long`
- Line 24: `fileLength = FileLen(filename)`
- Line 25: `' Initialize control variables`
- Line 26: `Dim currentLineByteCount As Integer`
- Line 27: `currentLineByteCount = 0`
- Line 28: `Dim currentRow As Long`
- Line 29: `currentRow = 2`
- Line 30: `Dim hexEncoded As String`
- Line 31: `hexEncoded = ""`
- Line 32: `Dim chainChecksum As Long`
- Line 33: `chainChecksum = 0`
- Line 34: `' Place file name in the sheet`
- Line 35: `sh.Cells(1, 1) = filename`
- Line 36: `' Encode the data`
- Line 37: `For currentFilePosn = 1 To fileLength`
- Line 38: `' Get a Byte`

Figure 1: VBA syntax errors

Also, once you fix the obvious errors and try to run it for the first time VBA will highlight any run time errors:

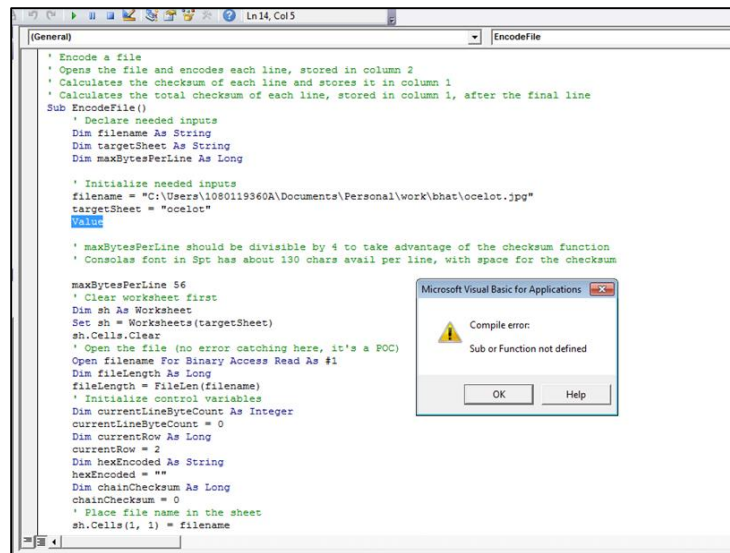


Figure 2: VBA run time errors

It is very possible the code won't work, even though it compiles and runs. In that case the attacker will need to examine what they've got and compare it to their original print out. One consistent mistake I saw was the movement of statements based upon an OCR misinterpretation of indentations or wrap-around lines. For example, consider the word "Value" below:

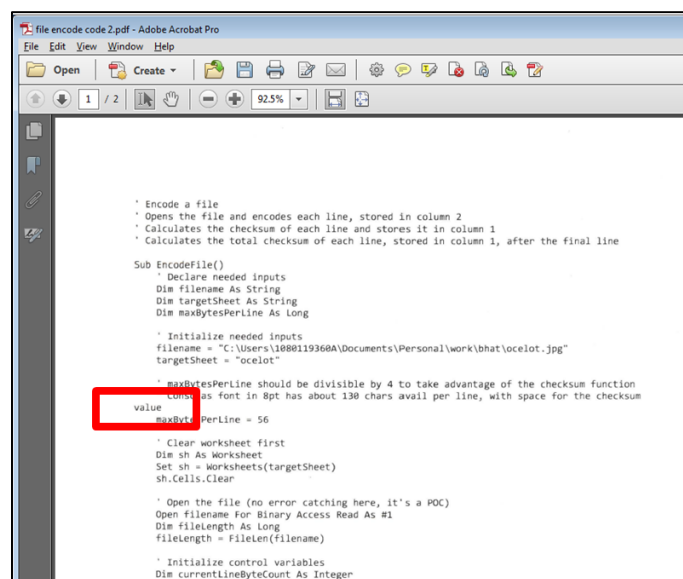


Figure 3: "Value" original location

After OCR Adobe moves its location to before all of the comment lines:

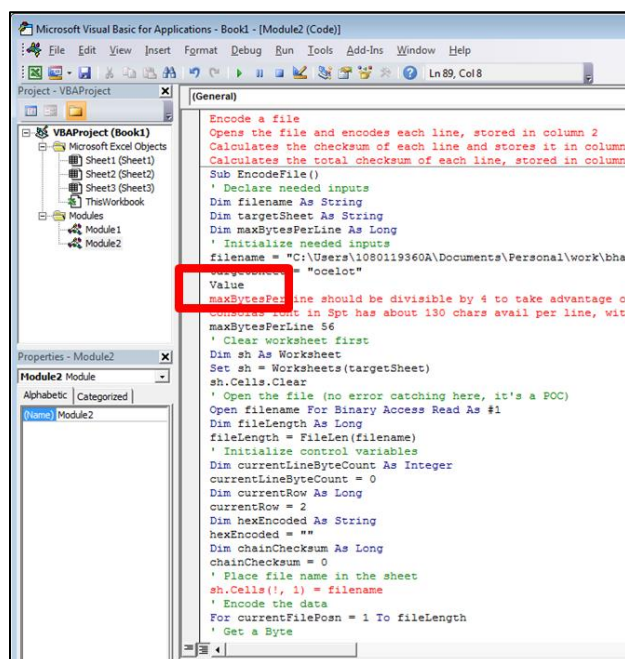


Figure 4: "Value" final location

This same error will be present in any indented code block. The trailing delimiter for that code block will typically get moved to just after the opening delimiter. For example, with a FOR...NEXT statement, Adobe will move the NEXT statement to just under the FOR statement, effectively skipping the entire code block.

Luckily VBA doesn't require indentation for its code blocks, they are text-delimited. This means you can remove all of the indents from the code and print and scan it that way. This makes for harder-to-read code but it will maintain the overall code structure.

Some amount of editing will almost certainly be needed to make the code work but overall it is quite straightforward to load a script of arbitrary complexity into VBA in this manner. If that script was compact enough the attacker could just type it in, of course.

Stage 2: Hex Magic

I wrote a VBA script that will take an arbitrary file and generate a printable hex encoded text file out of it. It can also reverse the process to turn hex code back in to the original file. That is a rather straightforward process. The magic comes in by making this hex code printable, scannable, and easily interpreted by the OCR algorithm to recreate the original file with as few errors as possible.

I did a lot of tests to figure out how much data I could pack on to a single page of hex code. By using 8 pt Consolas font I was able to get about 3.6kB of data on a single page. But, no matter how finely I scan the hex code, there are always errors in the OCR. Some of these errors are simple transcription errors, such as "1" (one) being converted to "l" (lowercase L) or "5" being read as an "S". This type of error is easy to filter out since a lowercase L and an S are not valid hexadecimal characters and can simply be considered a 1 or a 5 respectively when reading the hex data. Other errors are a little more insidious, such as a "B" (the letter B) being read as an "8" (the number 8). Both B and 8 are valid hex numbers so it isn't clear which one it should be

at any given location in scanned hex text. This error is also quite common, so I needed to find an easy way to filter it out. In the end I replaced "B" with "#" and "D" with "?" in the printed text. This eliminated a large number of transcription errors. There are other errors such as added periods and added spaces that also needed to be addressed. I also investigated using other encoding such as Base-64, but that encoding uses almost the entire range of printable characters as valid symbols. Because of this it is impossible to identify a transcription error and the file cannot be recreated.

But, no matter how much automatic correction I put in, something is bound to go wrong. With the corrections discussed above--and some other ones visible in the Hex Magic code--I only had 1 error in 1210 lines of printed, scanned, and OCR'd hex text. That was pretty good, but it was an error nonetheless. To help detect these errors, Hex Magic generates a simple 2-byte XOR checksum for each line, and a 2-byte XOR checksum for the whole file. I chose this as my checksum because I needed something compact that I could include with each line that wouldn't take up much space. Also, the XOR function is built into VBA.

As Hex Magic decodes scanned and converted hex text, it will highlight all rows where the XOR checksum fails. For example:

	A	B	C	D	E	F	G	H
1	9E40	#031	4000000080013A?2748?6?3#1E#2E060100001E02001000000053696					
107	35EF	507E0669?EE085F61F#05#ECF2#64AA6A31635#84#?60CC0394483360C#F843E						
138	84E4	8AE38399A?51E95E441#9A890986686FF45894F599C43~87#4#8803C171A4755						
225	13F2	F391?##?46~6?E#2#83C2025A210#?9E99A0103F04A4F78C401AA0F726?48389						
405	7E77	72881#6587#457219150?75C351~400C3C593518C4E76355830?F8FC3E4C990F6						
551	55#E	E5566?C?693?EA?0709705?99A95711?2452#787697AE?#767622A09CE?#E4?7						
569	8E2E	329F2EC?C3971#281#E8940?74CA007A8C0174C5?#?6F#102628A37C?2C5?104						
747	FC9C	814F805455274AF622A0010CFE?5?A7836879C84#438F85786EE823808A63301						
859	F#SA	6F#8920C1#A4E??0314?76F22#076957E#70?EFF#23A4E13F7E2?E#8531C2AA?						
1184	A472	AC72363#7328A52E#?885E421I?14A#A826EA753918773?0E945EC6650F521F6						
1217	A#90							
1218								

Figure 5: Checksum errors in Hex Magic

At that point, the attacker has to locate the line of hex code in his original document, compare it to the converted hex text and correct the error manually. This may seem daunting but by using "Ctrl-F" to find the offending XOR checksum in Adobe it's actually a straightforward and efficient solution. When using this technique to deliver a 65K payload, I was able to get the file successfully decoded in under 10 minutes.

An example of a full hex-encoded page of data can be seen at attachment 1.

Hex Magic proved to be very reliable. In fact it could be used to deliver payloads without the need for a scanner. It may be very, very tedious to type in line after line of hex code, but it would work perfectly and leave almost no trace on the network. However, it doesn't put enough data on each page. 3.6K per page is not very efficient. A payload like PowerSploit would need 232 pages of closely printed and scanned text to be delivered. Mimikatz would need 150 pages. But my main goal was never to deliver a final payload with this method; I intended to

use HexMagic to deliver an executable capable of encoding and decoding printable data at the pixel level. I called this executable Sideload.

Stage 3: The Big Bar Code and the Sideload DLL

I wanted to pack as much data on to a single page of paper as possible and to do that I needed every scanned pixel to mean something useful. After considering this problem for a little bit it didn't take me long to realize this is exactly what QR codes, data matrices, and other well-known 2-D barcodes already do. I played around with these for a little while to see just how much data I could squeeze on a single page using traditional 2-D barcodes and ended up with a number around 25 kB. But, as I studied these bar codes, it was clear they were designed with a different problem set in mind.

QR codes and their kin are designed to be read by cell phone cameras, or machine vision systems. They are designed to be scanned at strange orientations, in poor lighting, with possible physical defacement. Because they are designed for these circumstances, there is a lot of data area lost to error correction and orientation features. I was in more control of how my bar codes would be read. I was going to carefully put my bar codes on a high-quality scanner. I could control the settings of this scanner to get the best possible image. In short, I could expect far less error than a free-range 2-D bar code would expect. So, I decided to design my own, full page bar code.

I kept three main features of 2-D bar codes: bit-level encoding, timing lines, and Reed-Solomon forward error correction. Bit-level encoding simply means that each pixel in the bar code represents a bit state. A white pixel represents an "on" bit and a black pixel represents an "off" bit. I laid my bytes out horizontally across the page, 8 pixels per byte, about 88 bytes per pixel line. These pixel lines represent the "meat" of the bar code, the information the bar code is trying to get across. Timing lines are extra lines in a known pattern built into a specific location of a 2-D bar code to help locate the "meat". After some experimentation, I used lines of alternating light and dark pixels on all four sides of the "meat" of the bar code. An example of what a completed, full-page Big Bar Code looks like can be seen at attachment 2. Using the timing marks, I was able to locate and decode the meat in a scanned bar code with a very high level of success. Typically I was able to decode the bar code with less than 1% error over about 84 kB of encoded data on a scanned page. You can see a heat map of a typical decode trial below.



Figure 6: Big Bar Code error heat map

Each black pixel represents a successfully decoded bit and the red pixels are missed bits. The pattern shown is pretty typical. My main takeaway from all of my tests was that there will be errors and I need to be able to handle them.

That's where Reed-Solomon Forward Error Correction (R-S FEC) comes in. Almost all of the popular 2-D bar codes out there use R-S FEC to handle this exact situation. Despite their widespread use, I struggled to find a working, open source R-S FEC C++ library. The vast majority of the open source R-S libraries out there actually do Forward Erasure Correction and not Forward Error Correction. Forward Erasure Correction is excellent if some of the data is lost during transmission but will not help if the data is present and mangled at the bit level, which is exactly what occurs when reading the Big Bar Code. In the end I wrote my own R-S FEC C++ library with significant help from an excellent python-based tutorial located at: https://en.wikiversity.org/wiki/Reed%E2%80%93Solomon_codes_for_coders

After I got all the pieces working, I put them together into a library called Sideload. The Sideload library will take a file, encode it with R-S FEC, split it into one-page Big Bar Code chunks, and generate a bitmap for each page. These bitmaps can be printed at 72 DPI to just about fill a 8.5 by 11 inch page from corner to corner. Along the way the Sideload library will provide the exact file length and calculate the MD5 sum of the original source file.

On the target system, the attacker scans the Big Bar Codes, points the Sideload library at the resulting image files, and provides the original file length and MD5 sum. Sideload will pull the

data from the images, decode the R-S FEC encoding, and test the resulting binary against the original MD5 sum. If all goes well, you will have then transferred that arbitrary binary data from your source machine to your target machine without using magnetic media. For my proof-of-concept, I bundled the Sideload library into a DLL I could execute from VBA.

The Out-of-Band Proof of Concept

For my POC I wanted to deliver a portion of the PowerSploit tool set to a system without using magnetic media or downloading it from the network. Here are the steps I took:

- 1: Create PowerSploit payload zip file
- 2: Encode payload with Sideload resulting in 3 Big Bar Code sheets to print
- 3: Print out Hex Magic VBA code
- 4: Create zip file from Sideload DLL and associated VBA use code
- 5: Encode the DLL zip file with Hex Magic and print (made about 16 pages)
- 6: Move to scanner on target system
- 7: Scan the Hex Magic VBA code
- 8: Scan the Hex Magic-encoded DLL zip file
- 9: Scan the 3 Big Bar Codes
- 10: Move to target computer
- 11: Put Excel into "attack" mode
- 12: Convert the Hex Magic VBA scan to text with Adobe OCR
- 13: Paste the VBA script into Excel, fix as necessary
- 14: Convert the encoded DLL zip file scan to text with Adobe OCR
- 15: Load into Excel and decode with Hex Magic script, repairing text as required
- 16: Unzip the DLL zip file, load the VBA use code into Excel
- 17: Convert each BBC scan into a PNG using Adobe
- 18: Set up Sideload VBA script with file names, original file length, and MD5 sum
- 19: Run script to decode the Big Bar Codes and recover the payload
- 20: Celebrate!

In practice I was able to complete all of the steps of this POC to deliver about 162 kB of PowerSploit tools to my target machine in about 15 minutes.

Preventing this Attack

This file transfer vector is a bit difficult to stop since it is using standard office tools in a fairly standard manner, but it isn't impossible. The most obvious indicator to look for is the Sideload DLL. As a set code file, it will definitely have a signature that can be detected by anti-virus and other tools, assuming the attacker doesn't attempt to obfuscate the code or use other AV defeat methods. You do have to be careful to not limit your scanning to just ".DLL" files. As I stated previously, VBA can be programmed to call any file as an executable. If you called the file "Sideload.txt" you could still run it from VBA. The next step has to be monitoring or control of the printing and scanning resources. If the data or system you are trying to protect is of high-enough value, then every print job and every scan job should be examined by a reviewer. This will obviously be manpower intensive, as well as something that slows business productivity, so it probably isn't a reasonable control for most circumstances. Lastly, somehow preventing VBA from being turned on in Excel would stop this vector completely. I have not spent any time figuring out how to do this, but it may be possible.

Areas for Improvement

Improve Big Bar Code decoding: Though I was seeing less than 1% of error in my bit reads during decoding, the pattern of those errors requires me to use quite a bit of space for R-S FEC parity bytes to ensure the message goes through. In my POC about 35% of each Big Bar Code was dedicated to parity meaning I only got about 55 kB of actual data per page (out of a possible 85 kB). R-S works in 255-byte chunks and I have to provide enough parity to handle the likely worst case error rate per chunk. If I can drop the number of bit errors per chunk down, I can dedicate more space to data per bar code. I already did some work in this direction by adding additional timing lines but I ended up with extra errors I didn't expect. I'm sure, with more experimentation, I could improve the results.

Use 2^{16} R-S FEC: I already mentioned that I had problems finding a working C++ R-S FEC library. Part of that was because I was being picky. I really wanted a library that worked at the 2^{16} level whereas those I did find only worked at 2^8 . As I stated above, R-S FEC works in 255 byte chunks, unless you use an underlying field math capable of 2^{16} . Then you will be able to work in chunks of 65535 words of 2 bytes each, for a total of 131 kB-sized chunks. This would be ideal for my application. Currently I'm running at ~35% parity bytes because I have to handle the worst possible error rate in a 255-byte chunk but I've already stated I can decode a bar code with < 1% bit error across the whole page. Since the page only holds 85 kB of data, I would only need 2-3% parity if I was working in 2^{16} correction space. This would be a tremendous improvement.

There's a catch of course. I actually built a 2^{16} R-S FEC algorithm and discovered two problems. First, it is very slow. R-S FEC complexity grows quickly with the size of the code word, so the increase in code word size slowed my tests down tremendously, especially when I started using kilobytes of parity. That wouldn't be a huge problem, except for the fact that the 2^{16} algorithm simply wasn't working. It would happily error correct in certain locations and not in others. To help debug this problem I needed to run tests. Every test was taking far too long to run. In the interests of getting a working POC running, I went back to 2^8 . I think I can solve this problem eventually and greatly improve the data per page ratio.

Use colored pixels: The Big Bar Code only uses 2 colors, so each bit needs its own pixel. If I used 4 colors I could show 2 bits per pixel and double the data per page ratio. If I could use 16 and get 4 bits per pixel, I could manage ~340 kB of data per page. My experiments in this direction showed that it might be possible, but it also introduces further decoding errors in to the process. Printers, even high-quality office laser printers, simply don't cleanly print and clearly scan a stream of essentially random colors at the pixel level. My 4-color attempt did show some promise, but the 16-color trial looked fairly hopeless. I also stopped pursuing this path in the interest of completing a successful POC.

Conclusion

Users that can write code on their machines represent a possible threat to security, it is as simple as that. What isn't so simple is that every user with access to a standard desktop productivity suite can write code. A security architect needs to examine the system they are trying to secure by taking a step away from firewalls and cable separation and fully explore all of the resources available to an insider.

The Hex Magic, R-S FEC library, and Sideload DLL code will be available on github at: github.com/richmr

Attachments:

- 1: Hex encoded file example
- 2: Full page Big Bar Code example

Attachment 1: File.zip

9##1,504#0304140000000800?67342483463#?3EEA0C0000?41F000009000000526F61726#2E747874A5594#AE?C36169?07C81E#880CA5#4026
8#?3,8?#411201E243?31?0E8214#6249F4934885AAAAACECA23?79?#E?73EE255AA#2F3696462#F92F8#99F73CFF?E8C318?76C439F4?#C98C?
5093,?9948?1?E2C994?1998#4F#998?906?3F#?CC5A#4#AE3763#C9912F1F45516#817F39333379#C?92E2?99EA7C?9C?714#0?016392417AEC4
E29C,1FA3E?4?#1E#301633FA6CCE29E235AEC36193C7?317F3CF55#624672FC52533##209#780#57#?3A#7F0F?8CE#?27CE2A32EC6573F2C95C
C68A,62EC4FF#61F56167AF2E8#843C687297A207583378F?7#933721169C153AE7AF943CF44?E009C#F0A4#7E935#84C235?442#18AA8C2FE69?
F7E1,0#654?#E65335988FC6C361FAE5880F?7C7C1#9?9#E#456892998186449#2?#E#9F98EA#CFAA97F314F#63A89#1#A#84#?898A0F464160#
C4#E,216C807E31F9?F62F83#56888#0#558#14A?6245548A09C27CF#C?873145481#5526?3E?561109C?8?FF50C0?#18A1035CA626CAC52?9F4
##15,F1164C70#77A345C8#A#6#E#8C9467F46180CEC?9F5C7AF5?9539?0FF2648876C219FE72213A4F6?E?66EC3A91?69414FE3CC730C?FE3F7
A4AC,0E4CC089AE10614E##7694457FE94?660?E79852#CF1781A4516#A#CC4909?#E959C?AEC0?67430A91C?0CF00?ECC97F63CA6F?C45755EE1
594?,483?1E10F2#9?41F#9F3#0?AF?#742871#332C066F4C3160C#607?C82A3F0?AA7E80198A9F?#1E40E?#55?F0A6CA0?671A3#3AA#326E881A
9CF#,A2ECA46AE455A?#8#4?C0022?FEFC76EF07780#77F894?A#86E47921C8CF30#AE95A?#A#A07FC9E705?7A4474721F5?#45?F2149?088
?9C1,FFE3831830A2#CC0C925C5A?AEC?134E#8AC135C#6161F?C?#1493?798C73525A?801185E4A53120?CE658F05E5F134?#244F?3A?64740F5
A8F2,2#15#0?E81?54F085F8?9F?#AE?C5FC60F2362F630C1##69A3320C?238C0#0344E1A365F453CC71193779617?5A264#2913?733801EC37
C8C7,51AE894724EF72807720E42F91868F?949E4C06?091084E9F12F42328172887837514CFC75991045#8C187?19E7?#3AAC6508?64715C43A8
0E14,C986193A4C08381F7A7FF5F?6A1526#88C8612E9?FE560979#9A5002C183C#A#C72#99#?989F21C084200#?5#024?C20E6#?#45?E?3720F
#F7C,8291A81581#398A2E19?8F016A808F7E53309F64333001A#?385FC05064C76801476762A57735#5E7CA28C?F7EF36F061714F0AA15E0?#AF
C602,5?#9390?F616E2C54F?153EE354F9#2A97E3A9E4418#0#533300FE82A940372E8#E#?712E778F693?3F7F7?F0FA#2A4034?8?40AC2A94?C6
2878,8E?9E72678718?6F54C3EA8?0670AE77305?#5E#4011241F5856C598C395#0806#3AC7#?65F71?CF#5?#3A95A?FA90099C0#7C?E0F056467
E18F,263C6FA713419319247418720?4200AFF4C4E03E29C?F352FAE7F986CE06C2E#CCE0#F92#5?24EA62793474#1AC0#3C6632FE63?17830#07
?035,C986193A4C08381F7A7FF5F?6A1526#88C8612E9?FE560979#9A5002C183C#A#C72#99#?989F21C084200#?5#024?C20E6#?#45?E?3720F
5?57,F6FE?48?360C4242C?2E8C5E2902A0A228?7CE?23E?623#5F98FE?5312FE63FAE28?#E?EF78C59A2A6F391#04A3794?577FC5F924628?6?
0EC?,928758AFE88788#E5A9FFC889A6?#09203887043#5A2?241AF469?51F24?7334#633#7CE39492CE36F#0E6A43A3207EF#?E88ECA14AC1A0
E479,50F16265E05#2441E6#7#31A067E71A81?7AA61?A155AC6911397#C10##104#E#409771500431309377341?EE3F0484#?875079A3E?F0F50
6163,A83CA6?5695#07C4?4E53E9?3AE87052?F4A48283EE2848#C?9#143AF#999F#3?644A04#0#9F187E5?91A428#29E4F78C91FEF570A3#0A2?
?1?,896C4F99349FF0A6B431C4233F?#24#1E96748A979841#3C1?8F#?7376?3E3495F3?3#838042F1?#0?#1F8E4E6FE97128##29419E#15AE
4080,A550155621C28?4FE164A4EC6AEFEFAA2?F02E47891F717F0?7F81C68A75492A56C917380?F72EE#A4?5?EF#72E08A6E4?672108?#257FF1
80?#?,9?92445EE1F#5E40?14#FE6C51?02FF#9EA95273922?3922592639#077#3743E25169AEC202C717?594327#C86#F8?#2A870F917CAF4A9E
2795,8413929#E47C#5#9545F20AE6A?#3#70#E34?5C1FE4E0C266553470CA4626409#?1FC9504#5CA7C17#71EA33A40?42F72E8394#3F2AA77?#
A#41,0441#2?20A43?#7?6285F970949669C?E73C10AA7523?#284CCECF35CC5E77E9621#6148782AF366E9A?E0F498825600?7939CE3F84F76E
79?0,47#1EA2C99A5#?AC2998E58E?8265?5#70EFC8?7E69AC951F?A83792?24FF0EC1#F2EA52186109C9798#33F4F4E9#003877A9993A381A07
#453,#4#EE048C#44176A?070#4C2060A9?99#?A1A314F067#A#819C274#68#C#FA0F0?#E06588228F9290787202E32#1215##E779C55?734CEE
E893,FE24008944F781E?7784A#183452A987FA082F9591AE32#0#0892#E368A7#5F33A2FFA55A579FCA2A274#01#7?7#3#5478A923334E?90#?
#??9,6#7449A6A7F4C96CA151C706#3558FEF5A95A705CC?95?A84E07F0258F?3F1E62?4AA263F7E71122006C3176E?E17E27E11E0624629A
0A25,2A90#9F5A8F782248E80A6C07C2875#031C85A?#AAC982?5#?572F2E8C0F9F64A0373584815#724E##393424C4#26E473187#9397994C
6#48,E#C071C6F?7535E54C90FF9137C51FC575F5031#4854E83#AC55CE#?A0EC479941#?ACAA6AC05CA5F#?28?FFCACF7C8?96#4A12917F72?9
2740,1019F8872E73?FF87F?#5?A4758#832#7#F1F40797CAF25CF372C?8163F4?#0E5E26#0EF76#227?7F77126?1#741C061F?038?73E55306EA
1C24,62EF9E5355AC45?9E1?503EA6AF55AC5561EF739AF4E58862482?669A9439C43#9C4C3?8C9A91C9615?9EF89C06A0#6?6169A3A09A8A75?1
9A57,53?6?4C844E6419?2346A4C9#520F#072822F72CC?7AF#17A#40FAC3EC996C752?3FC200?C94E0A?28F70A3?4C990E0F3?1EECS#8214A511A
0477,63FF97F61E3C26669474978?0301020FEF8364592?#1F72411#30094CCC012017267F#29A960152EF11C07#A15EA42592#A03?5E605E175
73F4,#AE710504#1016AA4F6C7F#5A1#0AF44?F17EA5EC8029F48?#007F0#0434#6F7E1C91901#ECF43C67#9720794256F451E05?#7F836872115
17??,F2#9346752CA003C93063373#3A93?C90719?A38F78A5A25#547779CA4A61HE67#F56543?E4?2#32#3786#3F#157A66F601#55#7053AE9A##
29#6,A?174030A769979165EA08E05E3?AA2469EF?#27942E5C0A411E2A?937#8?63#195A401CFE41E37131FF769F9629725AF126?32936F6684E
8#F7,395788E966938E81?055274F5AE7546F056A9818491803914CA51?AE1?36C5FF23E6F177C?36F?3109?144A?#1866#9876F9080503CA7?32
332A,7#E#21#09?F99429A6#570C1CEEE7#5?7EEF3C1F68334E93E#0E75F27F75FE2831F15903EE#EE6C?72F42#8323546C39F30F9062716F1C0E
3#44,C97993859169898F6#2E44F9EE14276546?91212F826#F0CC4450CF#3C#2?4#C#626419A41#8EE233#0A7649A9#66370?1266328CAE80#E?
73#5,E42#3?708E22?1?4AAC102EA?#2#61727884C63?67521CC6C1C89?148232F22?E?3?#A74#43?AE22#60?AC?A2500C30576303976?E154E53
2468,597F3288EE139001A4354519F4#5112CEA5C7A3203#A3?9#E?73EC3723E?1421?8#7304?#E5614F32A15AF88A261A66369795C47214F7A2E
25C7,6#225E85E4691E?#1125AE0E2549#C339CC1?AE6?088EC?5#01AF?#?#6F?E79?671#FA9#FFC3E82005764?6930AF15?79181#5?#6?5760?CA
8F94,F60AE2F9CC?94F9C9C16F0?AA1A08459E1E4EE64?23A?#?#E6C9?E726#3097#88FF3#1274FA4103?1528AE416F6FE64306566A7#3?E?#49
1000,?A8515C61980?01?#4680CEACC7C9?7E27?9?053F549#A7F1E#14146?#4046501261609?537CE5F425C6AC1E66?9#C06C9791605F3A#38A1
A1?9,AE94?263?679873593E3E70E08402216E1?6?AC?7530?48614272141866338EC#3917A5E431#5C80#25?34A9C141#EE5278?851329E?67EA
#728,9E25FA2C18?3?863E6?C?FE59#2466#C14E8E3#81F5766AE?690#830AC?3857A?1A65?5C414EEC10657#5F9###2C7377FA5F3E#188#381?2
?9A8,2C?C25EEE198077CA7FE3#54471CF0E130082F?15FA2?1EE9?58177F?8C?ECA?022?36A4?A63?89A8CF7199?64472F10916F192C1F?04?25
0#24,546#12420C16#6?8522A4#C?C0246EAF51F2101?899?5#5C#58A?554274?7E2?0F8950?#6842#F0E7?6617A1737E61E9660051096E8843F0
8365,#F39197E00FA43E?786#10C?9AE0A57F0AAE7289F69482?CA0504771F9546990038#AC5510#27#6C0#8A?F9A?86A274FAF3?65C74326A?4
474?,?639897008C5FA?1548#9F7?AE?#03?117AF#683?A078391C?35A70C?CC#EC259388E3A6C20F73EC92?#4C56AE#38781038C#2F6?A6CC#18
F055,F870A4848573?AF3#6760#EF6##332C50?A406056672E#65C?088F?6##C75A04791?8EE870?0321439E897?9021#9CE7#324?#0#F6F72F41
#?34,A33E91633567?F644A?C6#47209F961A47E9?4#97E436#?589#C7907784?#85C60957E43FAF1FA#5A25?87A#3858E367E835A94001E984?F
6A31,664F86650607?297045A4EF5E#853?AF3A50781087#67E6#381859?397323969256F?2537C45#F5CF32160F#1AE22?#4F?5F1A62AAF36#64
C?66,926198AA393480F1630?3#92F#?63A4#C9C0AF#7#5#3FC#A04A2C3F?3#189C26498FE?C3F10#C7FF00504#01021400140000000800?67342
2F8A,483463#?3EEA0C0000?41F000090000000000000010020000000000000526F61726#2E747874504#050600000000100010037000000
110?,110?00000000
#C73,

Attachment 2: image.jpg

Md5: 86e96ddc7c9d2c7ee468b1b4db8b234c

Encoded file length: 72384; n = 255, k = 140

