Exfiltrating data over air gaps via hard drive LED modulation

Denver Cohen
Calvin Figuereo-Supraner

April 30, 2017

Abstract

This paper replicates and extends the recent LED-it-GO data exfiltration technique by Guri et al. We devise a more robust communication link that implements a true frequency shift keying method. An optical receiver system was designed that uses a hardware-based photodiode receiver which leverages GNU Radio. A Bash and Python script were designed to transmit the FSK data using the hard drive indicator LED, and a receiver recovery method was designed. The codebase for this project is documented in further detail at `https://github.com/clfs/led-airgap`.

1 Introduction

Data exfiltration is a technique that can be used to copy or transfer data from a storage device, usually performed illegally or with the intent to not be noticed. Air gapping is a common preventative measure against data theft, where the storage device or network of storage devices are physically disconnected from the public Internet or LANs. However, air gaps are insufficient on their own to prevent alternative forms of data leakage.

A powered-on computer creates a number of emissions that can be specifically crafted to leak data over non-Internet protocols. Electromagnetic signals generated by RAM operation [1], acoustic emissions from fans [2] or hard drives [3], thermal readings from varied workloads [4], and monitor backlight levels [5] have all been used to transmit data via specific and well-timed modulation techniques. This paper describes an exfiltration technique that uses the hard drive activity LED present on the front plate of many desktop computers. This attack is based on, in part, an earlier attack named LED-it-GO developed by Guri et al. [6]. They describe the advantages of their attack using five metrics:

Covertness: Hard drive LEDs are frequently active and do not draw attention.

Speed: Hard drive LEDs can toggled at many kHz, improving throughput.

Visibility: Optics are limited by line-of-sight, permitting attacks at distance.

Availability: Hard drive LEDs are present on most modern desktops or servers.

Privilege level: Hard drive LEDs can be toggled entirely from userspace.

Our attack intends to trade speed for visibility: by altering certain characteristics of the original attack's modulation scheme, some throughput can be sacrificed to create a method that is much more ideal for very long-range exfiltration. The following section on implementation details how this is performed.

2 Implementation

The exfiltration scheme created is composed of two parts: a transmitter (the compromised computer) and a receiver (a custom setup involving a telescope and additional electronics). Both portions of the overall implementation are discussed in their respective sections.

2.1 Transmitter

The transmitting computer was a Dell Optiplex 990 with a Western Digital Blue 7200 RPM hard drive, running 64-bit Ubuntu Linux 16.04. The front plate of the case was removed for ease of hard drive LED access during development. The LED status is directly controlled by the motherboard's BIOS instead of the operating system, so indirect means were developed to toggle the LED from userspace instead. Reads and writes to the hard drive activated the LED, which completed the necessary hardware setup for proper operation. The remaining transmitter design focused on software development, which properly modulated the LED for optical transmission.

The software created provides a means to read, encode, and transmit data of the user's choice out to the receiver in a simplified fashion. To summarize quickly, an example of the software invocation process is shown in Listing 1, which involves three steps.

Listing 1: Malware invocation process

```
cat secretfile.txt | ./encoder.py | ./readbfsk.sh
```

Reading

The first step of Listing 1 uses the `cat` command to read from a file and write the result to standard output. Note that this step, nor any other steps, require root account privileges: in the interest of preserving useful characteristics of the LED-it-GO attack, this advantage was kept unchanged. However, that means this attack alone cannot exfiltrate data it does not have access to; privilege escalation exploits are necessary to obtain additional data, but this was judged to be outside the scope of this project.

Encoding

The second step of Listing 1 invokes the user-written Python script `encoder.py` to read data from the previous step on standard input, encode it via packet scheme, and re-output the encoded string on standard output. The scheme used is pictured in Table 1. The packet encoding scheme uses a fixed-size header to more easily identify start-of-transmission points, and variable-size data packets to avoid zero-padding and reduce overhead.

Table 1: Packet encoding scheme

Header (once, fixed-size)		Data (n times, variable-size)	
Field	Size (bits)	Field	Size (bits)
Preamble	13	Preamble	13
Content length	16	Content	0-128
FEC code	32	FEC code	128

The preamble chosen in both packets was a Barker code, which possesses the ideal autocorrelation property: all of the off-peak autocorrelation coefficients are as small as possible. This property is advantageous, since false positives are unlikely to occur if the preamble is shifted by a small amount.

The content length header field represents the length of the plaintext content in bits. In this implementation, only 16 bits are used, so the maximum plaintext transmission is 8 KiB. While that number represents a relatively small file size, it is still large enough to transmit important files like `/etc/shadow` or a 4096-bit RSA key.

The content data field is of variable length, which eliminates the need for zero-padding and slightly reduces overhead. However, this commonly introduces difficulty inferring timing on the receiver end.

Finally, the forward error correction field in both packets uses a Reed-Solomon code. The mathematical depth behind Reed-Solomon codes is too great to discuss fully here, but they operate by appending a fixed number of check symbols to the data that it protects [7]. If t symbols are appended to the data, the Reed-Solomon code can detect up to t erroneous symbols, and correct up to $\lfloor t/2 \rfloor$ erroneous symbols. For example, the forward error correction (FEC) field in data packets is 128 bits long, or 16 symbols (assuming one byte per symbol). This means any combination of at most eight erroneous symbols can be fully recovered, and these errors can be present in either the content field or the FEC field itself.

Modulation

The third and last step of Listing 1 is piping the encoded string to `readbfsk.sh`, a Bash script that performs the actual modulation of the LED. The Bash script performs three main tasks:

1. Read the encoded string from standard input
2. Cause the kernel to drop the page cache from memory
3. Read data to modulate the LED accordingly

The output of `encoder.py` is a single long string of ASCII zeros or ones. This is read in before the page cache is dropped, since the kernel will immediately start caching again. The `sync` and `/proc/sys/vm/drop_caches` commands are used to achieve this effect. The most important step is step three, where the actual modulation occurs.

The chosen modulation scheme was binary frequency-shift keying (BFSK), which uses a pair of distinct frequencies to represent binary data: a short burst of one frequency represents one bit state while a different frequency burst represents the other bit state. To actually toggle the LED at the chosen frequencies, the UNIX utility `dd` performed reads from the hard drive to ensure the LED was lit. Including the `direct` flag with `dd` avoided caching to improve timing consistency. While writes would theoretically work as well, they leave evidence on the filesystem when used and would have to be deleted for covert operation. Preliminary tests led to the decision to use approximately 4.5 kHz and 7 kHz for the modulation frequencies, and 8 ms for the burst time (equivalent to a data rate of 120 baud). A 4.5 kHz burst corresponds to a 20 KiB disk read performed 35 times, and a 7 kHz burst corresponds to a 60 KiB disk read performed 25 times. These values were selected from the start as the upper threshold of throughput for frequencies that could be detected at the time.

2.2 Receiver

The receiver was designed and built for the purpose of recovering the transmitted signal. The selected design was a single photodiode receiver with heavy analog front-end filtering. The image in Figure 1 depicts the finalized design of the hardware implementation. This was built in a die-cast metal housing and had a heavy power supply filtering section for maximum noise rejection and isolation. The receiver design was heavily influenced by KA7OEI's optical receiver [8]. The additions made including the following:

Analog filtering: The low-pass and discriminator circuits were redesigned for the chosen FSK frequencies between 5 and 7 kHz.

A 7809 voltage regulator: The circuit works best with a regulated voltage since the power source was a battery.

A heavier power input filter: A LC Pi filter was used on the input with a 1 mH ferrite-core inductor and 330 µF capacitors to filter out 60 Hz. The PSRR of the 7809 regulator is high, but this filter insures proper clean power into the regulator at 12 V.

A potentiometer for photodiode reverse bias: The capacitance of the photodiode goes down as the reverse bias is increased. This lower junction capacitance allows for a faster response time and thus a higher frequency response.

A front-end filter for 120Hz rejection: This filter was a simple single pole RC low-pass filter between the first op-amp follower and the differential pair.

A half-rail generator: A TLE2426 half rail splitter was used to generate the midway voltage. This chip allowed for a lower half-rail impedance and provided a factory laser-trimmed resistor pair. This increased the accuracy of the midpoint.

A Q1 source degeneration capacitor: To combat the later 120 Hz optical spectrum noise issue, reducing this source degeneration capacitor decreased low frequency response. This was moved to a 10 µF capacitor with good results of 120 Hz rejection.

The hardware implementation also utilizes a telescope as an optical amplifier for reception of the hard drive LED. The system was run off an external 12 V lithium iron phosphate (LiFePO) battery.

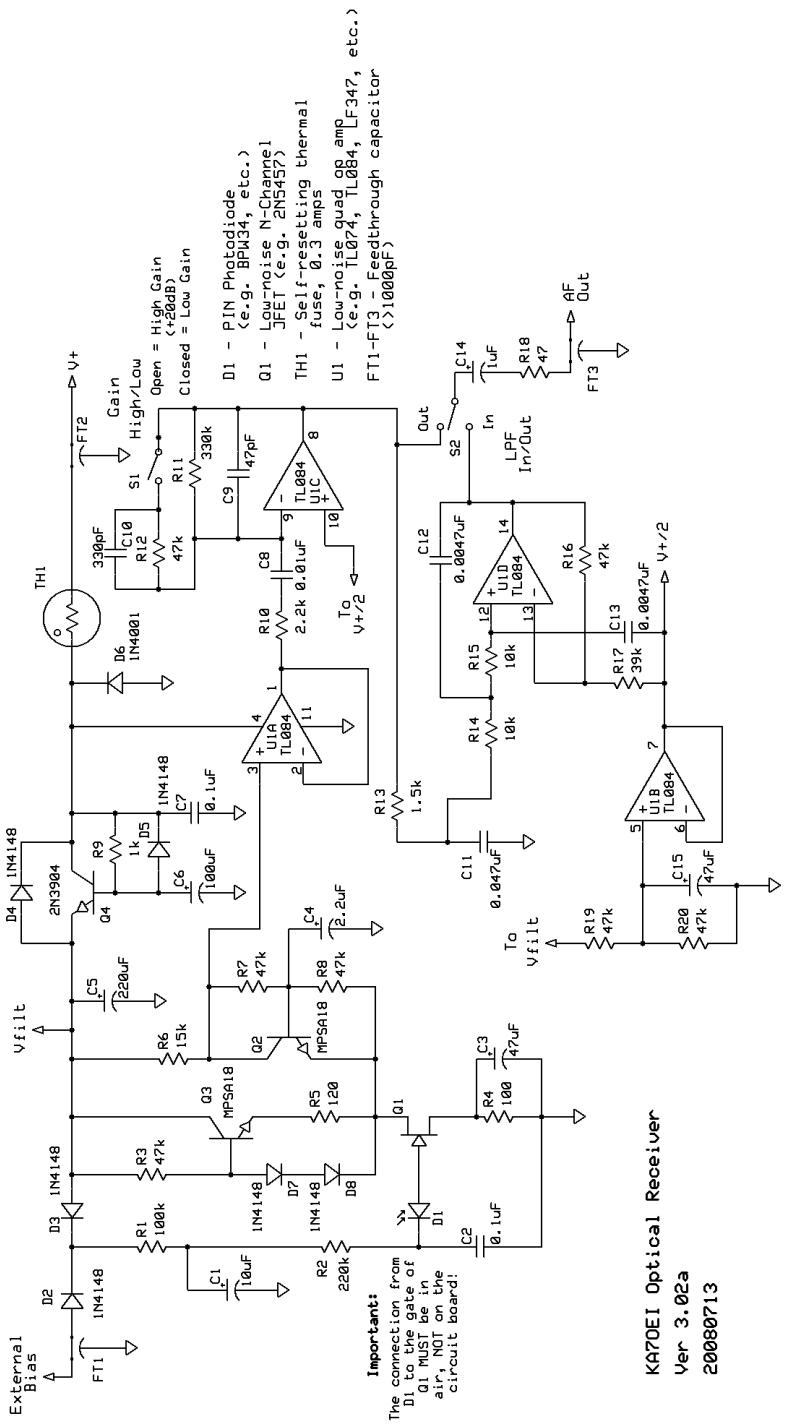

Figure 1: Initial receiver schematic

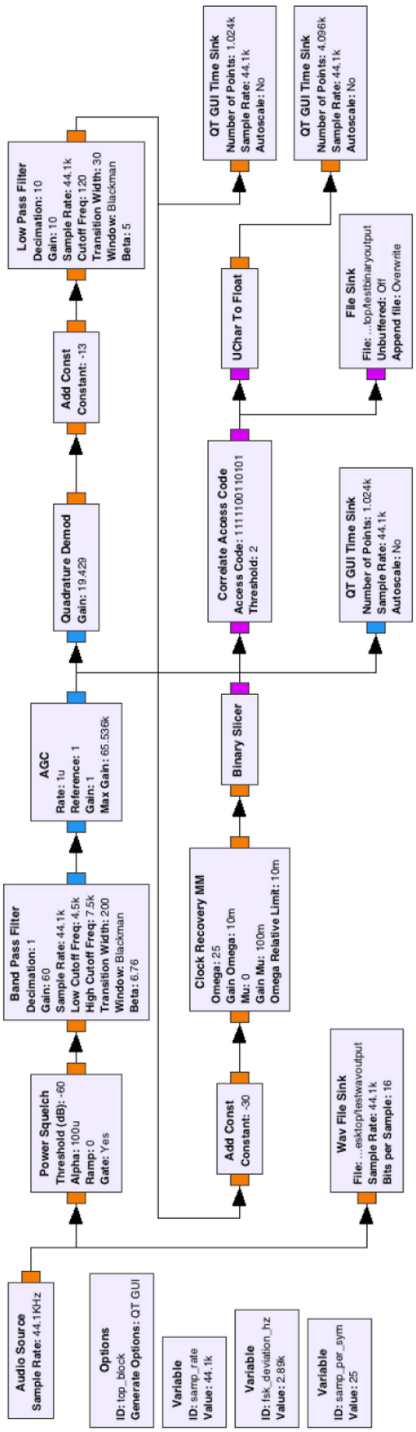

Figure 2: GNU Radio Flowgraph

To process the analog signal, a USB-based sound card was used. The Behringer UCA202, based on the Texas Instruments PCM2902 [8], was chosen due to a USB codec that had a 16-bit, 48 kHz ADC for additional leverage. This USB-based sound card is a good, cheap digitizer that only cost around 30 USD on Amazon. The digitized data then was used in GNU Radio, where the flowgraph in Figure 2 was realized.

Each block is listed below, in chronological order, with a description of why it was used.

Power squelch: The power squelch helped to remove spurious noise when there was no signal being detected by the receiver. This helped clean the output data.

Bandpass filter: This bandpass filter is sharply tuned to the two FSK modulation frequencies, in order to extract them from the overall input spectrum. The receiver does bandpass filtering in the analog domain, but nowhere near as sharply or tightly as this filter.

Automatic gain control (AGC): The AGC was used to level the input signal to a constant value. This is helpful in the next stage which will extract this signal from the surrounding spectrum.

Quadrature demodulator: This block converts from frequencies to numerical values. The dominant frequency of the input is mapped to a single value, which converts FSK frequencies into usable output.

Low-pass filter: The output of the quadrature demodulator is filtered to remove any spurious pulses in the waveform, giving a clearer view of the binary data for processing.

Clock recovery, Mueller-Muller: This block takes in a number of samples and converts the that sampling grouping into a bitstream of 0s and 1s. It does this by finding the data peak and comparing it to the zero-crossing.

Binary slicer: Unfortunately, GNU Radio uses complex numbers for its clock recovery and the design needs precisely 0s and 1s in the next stage. This block does that conversion.

Correlate access code: As mentioned previously, a 13-bit Barker code denotes the packet preamble. This block looks for the preamble within a certain Hamming distance and flags it in the bitstream with a two, three, or four. This can then be used later for finding the start of data payloads.

File output: The next few blocks record the binary output stream to a file.

3 Results

3.1 Testing

The first test was outdoors, with the test subject computer in the window of the second floor of Atwater Kent. Due to the lighting on campus, and especially the sodium vapour street lamps, the detector was swamped with 120 Hz optical noise. To prevent this, a stronger 120 Hz low-pass filter was added in the front end of the optical detector. Another, second test under the same conditions was conducted with better success, but unfortunately no usable data was output. Subsequently, the setup was moved indoors to the third floor of Atwater Kent, where there was a lower optical noise environment. In this improved setting, the receiver end was able to recover data with over 120 ft between the telescope and the computer. The final test used this setup to transmit the plaintext phrase "Hello this is the hard drive LED" 13 times, and the resulting file was saved.

3.2 Data

The output of the final test was recorded as a binary file, as specified in the GNU Radio flowgraph. In order to parse the data and decode the transmitted message, a variety of techniques were used to attempt message recovery. There were a number of obstacles that prevented a perfect, 1:1 decoder structure during the post-processing stage:

Non-contiguous transmissions: While the recording repeats the transmission thirteen times, each time-instance is separated by meaningless data that must be separated from actual content.

Bit errors: There was a non-zero probability that any given bit might be incorrectly interpreted as the opposite value.

Bit erasures: There was also a non-zero probability that a bit might not appear in the recording at all – this can occur when the receiver wrongly estimates the length of a long string of ones or zeros.

Due to the limited time-frame available to decode the given recording, much of the file analysis was done manually, using Python and various text-processing utilities, such as `grep` and `awk`. Two of the techniques used for manual data recovery are described below. Both techniques focused on using Python to output strong candidates for valid data, and UNIX utilities to parse the files that the Python scripts would create.

The first technique used was best described as preamble averaging. The Correlate Access Code block from the GNU Radio flowgraph labeled preambles in the binary output file, which meant repeated packets could be identified throughout the file. When the repetitions were identified, each packet was "aligned" on the preamble, and each bit after the preamble was determined via majority vote. The parsed output for this process is shown in Listing 2. The first 13 bits are the Barker code, and the following eight bits correspond to the letter 'H' in ASCII. This correctly matches the first letter of first data packet's plaintext content, "Hello this is th", which was mentioned in the earlier section.

Listing 2: Preamble averager output, parsed and truncated

```
pkt1      1111100110101:01001000[...]
```

The second technique used was best described as ASCII windowing. One major difficulty in creating a robust decoder structure was the task of determining where byte boundaries were. In ASCII plaintext, one character will occur every eight bits. However, if there is unwanted noise between each transmission, or a bit erasure, or any shift in location of where the decoder algorithm points, the ASCII plaintext may be incorrectly interpreted due to a bit misalignment. This was approached using a sliding window technique: by starting at every possible location in the received data, reading forward a fixed number of bits, and outputting only lines with a high likelihood of ASCII plaintext, decoder alignment can be ensured. The parsed output for this process is shown in Listing 3. The number 55157 represents the location in the recording (at the 55157^{th} bit), and the following bit string represents data with a high likelihood of validity. This nearly correctly matches content from the second data packet, "e hard drive LED", again mentioned earlier.

Listing 3: ASCII window output, parsed and truncated

```
55157    b'ard drivE LED\x8c\x01[...]
```

4 Conclusion

Considering the short timeframe in which this project was created, the ability to recover data under our original model was a relative success. The transmitted content, "Hello this is the hard drive LED", was fully manually recoverable at the binary file output, albeit present in a discontinuous, error-prone form. Some of the obstacles faced during the design and testing process can hopefully be used

as content to inform future work. The largest of these concerns and potential improvements are listed below.

Signal-to-noise ratio (SNR): The data can be recovered more accurately and at a higher throughput if the SNR is increased. Using a receiving photodiode optimized for the actual frequency of the LED could improve SNR by 5 or 6 dB.

Encoding overhead: The ratio between overhead and plaintext content was kept high, at nearly 1:1 for this design. There may exist headroom to reduce this number while still ensuring accuracy of received data.

Automated decoder: The process of decoding the recorded data was done in an error-prone and manually-extensive manner. Research could be done into erasure-resilient codes, such as parity checks [9] or Foundtain codes [10].

Testing environment: The location for testing the transmitter and receiver setup was less than ideal, since WPI campus is well-lit, and the transmission distance indoors was limited. Testing in a variety of environments could give better insight into the utility of the attack.

With or without these improvements, it is hoped that this work can contribute to a general awareness of these types of attacks, or inspire another group to create additional optical attacks for future use.

References

[1] Mordechai Guri et al. "GSMem: Data Exfiltration from Air-Gapped Computers over GSM Frequencies". In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 849–864. ISBN: 978-1-931971-232. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/guri.

[2] Mordechai Guri et al. "Fansmitter: Acoustic Data Exfiltration from (Speakerless) Air-Gapped Computers". In: *CoRR* abs/1606.05915 (2016). URL: http://arxiv.org/abs/1606.05915.

[3] Mordechai Guri et al. "DiskFiltration: Data Exfiltration from Speakerless Air-Gapped Computers via Covert Hard Drive Noise". In: *CoRR* abs/1608.03431 (2016). URL: http://arxiv.org/abs/1608.03431.

[4] Mordechai Guri et al. "BitWhisper: Covert Signaling Channel between Air-Gapped Computers using Thermal Manipulations". In: *CoRR* abs/1503.07919 (2015). URL: http://arxiv.org/abs/1503.07919.

[5] V. Sepetnitsky, M. Guri, and Y. Elovici. "Exfiltration of information from air-gapped machines using monitor's LED indicator". In: *2014 IEEE Joint Intelligence and Security Informatics Conference*. Sept. 2014, pp. 264–267. DOI: 10.1109/JISIC.2014.51.

[6] Mordechai Guri et al. "LED-it-GO: Leaking (a lot of) Data from Air-Gapped Computers via the (small) Hard Drive LED". In: *CoRR* abs/1702.06715 (2017). URL: http://arxiv.org/abs/1702.06715.

[7] I. S. Reed and G. Solomon. "Polynomial Codes Over Certain Finite Fields". In: *Journal of the Society for Industrial and Applied Mathematics* 8.2 (1960), pp. 300–304. DOI: 10.1137/0108018. eprint: http://dx.doi.org/10.1137/0108018. URL: http://dx.doi.org/10.1137/0108018.

[8] KA7OEI. *Receivers for low-bandwidth optical (through-the-air) communications*. http://modulatedlight.org/optical_comms/optical_rx1.html. [Online; accessed 04-April-2017]. 2015.

[9] Sean Eron Anderson. *Bit Twiddling Hacks*. https://graphics.stanford.edu/~seander/bithacks.html. [Online; accessed 01-May-2017]. 2005.

[10] Luby et al. *Raptor Forward Error Correction Scheme for Object Delivery*. RFC 5053. RFC Editor, Oct. 2007, pp. 1–46. URL: https://tools.ietf.org/html/rfc5053.