The EntropyDeviationType Python Extension

`EntropyDeviationType` is an extension that is intended for finding data hidden within other data with no knowledge of the data itself. Specifically, the intended use case is to identify executable files (Portable Executables specifically) embedded in non-executable files. For example, malware hidden within a Microsoft Word or PDF document. This is a common occurrence within Advanced Persistent Threat (APT) style attacks which leverage client-side attacks in common business office file formats and often follow the generic pattern that within the exploit is a XOR encrypted executable that is dropped to the compromised system and then the host document is cleaned to remove the exploit.

The module contains two classes, `entropyDeviationType` and `xorTableSearchType`. Both classes are intended as proof of concepts and not immediately exportable to production. This package also contains an example utility, ``edfind.py``, which serves as both an immediately usable utility and as a rough primer on how to use the extension to quickly analyze and locate rogue data hidden within benign information streams.

1.0 DISCLAIMER

YOUR MILEAGE MAY VARY. AS WITH EVERYTHING TEST THOROUGHLY YOURSELF BEFORE UTILIZING IN PRODUCTION CODE. THIS MODULE HAS NOT RECEIVED EXTENSIVE TESTING AND MAY CONTAIN BUGS NO WARRANTY, EXPLICIT OR IMPLICIT IS PROVIDED. IT IS THE INTERNET. TRUST BUT VERIFY

Contents

1.0 DISCLAIMER.......... 1

2.0 Class Descriptions 3

2.1 Python Class Descriptions 4

2.1.1 The entropyDeviationType Python class.......... 4

2.1.2 The entDevReturnType Python Class.......... 10

2.1.3 The distributionType Python class.......... 12

2.1.4 The entDevType Python Class.......... 13

2.1.5 The xorTableSearchType Python Class 15

2.1.6 The xorTableReturnType Python Class 17

2.2 C++ Class Descriptions 19

2.2.1 The chunk_t C++ structure.......... 19

2.2.2 The dist_t C++ Structure 21

2.2.3 The entropy_retval_t C++ Structure 23

2.2.4 The entropy_wrapper_t C++ Class.......... 26

2.2.5 The entropy_t C++ class.......... 29

2.2.6 The xor_table_ret_t C++ structure 33

2.2.7 The xor_table_wrapper_t C++ Class 35

2.2.8 The xor_table_t C++ Class.......... 37

3.0 The epfind.py Utility.......... 39

3.0.1 epfind.py Command Line Options 39

3.1.0 epfind.py Example Usage.......... 41

3.1.1 Analysis of a PDF Document with an embedded executable encrypted with a one-byte key .. 41

3.1.2 Files with higher-entropy embedded data / One-Time Pad XOR 51

2.0 Class Descriptions

This extension is written in C++ with Python bindings provided by `boost::python`, in all earnest, if I was you, I would just use the C++ natively, but I am also biased towards C++. I have provided the Python bindings because that is a commonly requested interface. Layered on top of the C++ are the two Python classes that expose the functionality of the C++ module.

Experimental testing has shown that even with multiple iterations through the data set that this method is far quicker than brute-force XOR methods or even generating a table of pre-computed XOR encrypted values, id est:

```
for (auto x = 0; x < 255; x+=2)
        var[x]     = 'M'^x;
        var[x+1] = 'Z'^x;
 [...]
```

2.1 Python Class Descriptions

The following subsections detail the pure Python classes and related pseudo-classes that are generated by `boost::python` that are the interfaces or return types by the pure Python classes. These classes are:

▪ `entropyDeviationType`	The main pure Python class.
▪ `entDevType`	Pseudo-class that `entropyDeviationType` interfaces with.
▪ `entDevReturnType`	Pseudo-class that is the main return type used by `entDevType` and `entropyDeviationType`.
▪ `distributionType`	Pseudo-class that is the return type used by character frequency analysis related methods.
▪ `xorTableSearchType`	The main pure Python class.
▪ `xorTableType`	Pseudo-class that `xorTableSearchType` interfaces with.
▪ `xorTableReturnType`	Pseudo-class that is the only return type used by the `xorTableSearchType` and `xorTableType` classes.

2.1.1 The entropyDeviationType Python class

The entropyDeviationType class performs statistical entropy analysis to attempt to find significant deviations in an unknown blob of data such that you can find embedded PE files or otherwise abnormal data streams embedded into the host file. The functionality provided, theory, Python class interface documentation and C++ class interface documentation are detailed in-depth in this document.

The basic theory is that a PE file will have a different distribution than the file it is in embedded in, especially if it is encrypted whether it be XOR encrypted or otherwise. As such, you can split the data into chunks and analyze the distribution and look for deviation within the file. This class performs a Chi Square distribution test, Shannon entropy analysis and Monte Carlo Pi approximation upon a file and provides an interface to retrieving the scores of a given block of data and the whole file and also to calculating deviances of one block to another, the whole file or all blocks.

WARNING: IN A FUTURE RELEASE IT IS EXPECTED THAT THE MONTE CARLO PI APPROXIMATION WILL BE REMOVED AND REPLACED BY A MORE USEFUL METRIC

__init__(bs = 8192)	Takes a parameter, `bs` that represents the block size represented in byte to be used. The default value is 8192 or 8KB
openFile(name, whole = False)	Opens and reads the file specified by `name` performs whole file analysis if ``whole`` is ``True``
isValidBlockNumber(idx, base = 16)	Checks that a given index, `idx` is within the range of valid blocks; defaults to base 16 format, modified by the ``base`` parameter. Returns `True` if the index is valid.
isValidBlockRange(low, high, base = 16)	Checks that a given range of indices demarked by `low`` and `high` are valid; defaults to base 16 format and modified by the `base` parameter. Returns `True` if the range is valid.
getScore(idx, base = 16)	Gets the Chi, Shannon and Pi approximation score for a given block indicated by `idx`, which by default is specified in base 16 format but can be changed via the `base` parameter. Returns a `entDevReturnType` class; Throws an instance of `ValueError` if the `idx` is invalid.
getAllScores()	Retrieves the Chi, Shannon and Pi approximation scores for all blocks in the file. Returns a `list()` of `entDevReturnType` objects.
getWholeFileScore()	Retrieves the Chi, Shannon and Pi approximation scores for the entire file. Returns an `entDevReturnType` instance.
getXYDeviation(x, y, base = 16)	Retrieves the Chi, Shannon and Pi approximation deviation scores between two blocks indicated by the parameters `x` and `y`, which by default are specified in base 16 format but is changeable via the `base` parameter. Returns an `entDevReturnType` instance or throws an instance of `ValueError` if the specified range is invalid.
getBlockAllDeviation(x, base = 16)	Retrieves the Chi, Shannon and Pi approximation deviations between all

	blocks in a file against the block specified by the parameter `x`, which by default is specified in base 16 format, but is changeable via the `base` parameter. Returns a `list()` of `entDevReturnType` instances or throws an instance of `ValueError` if `x` is invalid.
getWholeFileDeviation(x, base = 16)	Returns an `entDevReturnType` instance containing the deviations for a block indicated by the parameter ``x`` relative to the entire file. The index is specified in base 16 by default, however that is customizable via `base` parameter. Throws an instance of `ValueError` if `x` is invalid.
getSequentialDeviation(x = 0, y = 0, base = 16)	Calculates the deviation for sequential blocks, both prior and following within a range of blocks that is specified by the `x` and `y` parameters or every block in the file by default. The `x` and `y` parameters by default are specified in base 16 format however this is customizable via the ``base`` parameter. Returns a `list()` of `dict()` objects with the keys `prior`, `next`, `index` and `dev` for the prior block number, next block number, the block the deviations are relative to and an instance of `entDevReturnType` respectively. Only one of the `prior` and `next` keys will be valid in any given list element. The other will have a value of `None`. Throws a `ValueError` if the index range specified by `x` through `y` is invalid.
findHighDeviation(c = 100, s = 20, e = 1)	**THIS METHOD IS AN ILLUSTRATED EXAMPLE ONLY.** Attempts to find blocks with high deviation values relative to the blocks around it. What constitutes high deviation is specified by the `c`, `s` and `e` parameters that denote the Chi Square, Shannon and Pi approximation Estimate respectively. Returns a `list()` of `entDevReturnType` instances for any blocks that match an empty `list()` if there were no matches.
getBlocksAverage(ilist)	Averages the Chi, Shannon and Pi approximation values in a `list()`

	specified by the `ilist` parameter. Returns an `entDevReturnType` instance or throws a `ValueError` instance if the method was passed an empty `list()` for in the `ilist` parameter.
isHighAverageChi(maxv, chi = 15)	**WARNING: METHOD IS AN ILLUSTRATED EXAMPLE ONLY.** Identifies blocks with uniform or near uniform Chi distributions for a range between the first block and the block specified by `maxv`. The blocks in that range have their scores averaged and then if the average exceeds a percentage specified by the `chi` parameter it returns `True`. Otherwise it returns `False`. The `maxv` parameter is specified in base 16 format and methods called by this method can throw a `ValueError` when an invalid `maxv` is specified.
priorHighAndNextLowShannon(idx, high = 20.0, low = 2.5)	**WARNING: METHOD IS AN ILLUSTRATED EXAMPLE ONLY.** Attempts to identify the beginning of a significant deviation by attempting to determine if the block denoted by the parameter `idx` has a high percentage of deviation in its Shannon score relative to the prior block and a low percentage of deviation in its Shannon score in the block that follows it. The high and low marks are denoted by the parameters `high` and `low` and default to 20% and 2.5% respectively. These values were chosen based on deviations in a very small sample and will result in high false negative and false positive results. Returns `True` if the prior blocks Shannon deviation exceeds `high` and the following blocks is less than `low`, otherwise it returns `False`. A `ValueError` is thrown if `idx`, `idx`-1 or `idx`+1 are invalid.
getSequentialLowShannon(idx, low = 1.7)	**WARNING: METHOD IS AN ILLUSTRATED EXAMPLE ONLY.** Attempts to identify sequential blocks of deviant data by looking for low Shannon score deviations in sequential blocks. The block to start at is specified by the `idx` parameter, which is specified in base 16

	format. What exactly constitutes a low percentage of deviation is specified by the parameter `low`, which defaults to 1.7%. This value was chosen based on analysis of a very small set of samples and is likely to result in high amounts of false positive and false negatives as a result. Returns the index of the highest block following `idx` that has a relative Shannon deviation less than `low`, or the index specified by `idx` if the following block does not match. Throws a `ValueError` if the index specified is invalid.
getSequentialCloseChi(lidx, hidx, dmax = 26.0)	**WARNING: METHOD IS AN ILLUSTRATED EXAMPLE ONLY.** Attempts to identify related deviant blocks between a range specified by the indices `lidx` and `hidx` respectively. Specifically this method attempts to identify blocks that have Chi Square scores that are within `dmax` percent of one another, which defaults to 26%. This value was chosen based on analysis of a very small sample set and is likely to result in high false positive and false negative rates if used as is. The theory is based on the observation that the distribution of shorter XOR keys varies relatively little. Returns the highest index of a block that follows `lidx` that deviates less than `dmax` percent, or `lidx` if the block immediately following `lidx` exceeds `dmax`%. Throws a `ValueError` if the index range specified is invalid.
coalesceSequential(lst, maxv = 2)	Takes a `list()` of `tuple()`'s in the format of `tuple((lowIndex, highIndex))` indicating a start and stop range of blocks and checks to see if sequential list elements have nearly overlapping ranges with a distance less than or equal to `maxv`. The concept behind this method is that once a sequence of suspicious blocks are identified it is not uncommon for a few outlier blocks to cause multiple ranges of suspect blocks that really is a single range of blocks. As such, this method checks to see if that is the case and coalesces the indices into a single range of blocks.

	Returns a `list()` of `tuple()`'s with high and low ranges.
calculateDistribution(x = 0, y = 0, base = 16)	Takes a range of block indices denoted by `x` and `y`, which are specified in base 16 format by default and calculates the frequency each character occurs in the block range. The idea is that shorter XOR keys across real data tend to encounter the value zero a lot, which leaks the key in question. Thus by analyzing the frequency of characters in a block range, we can easily spot abnormal sequential frequencies and quickly identify an XOR key as a result. Returns a `list()` sorted in descending order of `distributionType` instances of length `256`. Throws a `ValueError` if the range specified by `x` through `y` is invalid.

2.1.2 The entDevReturnType Python Class

The `entDevReturnType` class is a pseudo-class that is generated by `boost::python` that wraps a native C++ structure. It is used as the main return type in the `entropyDeviationType` Python class.

It contains the Chi-Square distribution test score, Shannon entropy analysis score and the Monte Carlo method Pi approximation score and error values. The exact interpretation of the data within the class varies depending on the context it is used in. It is either the exact score of a block or file, or how much each of the respective scores deviates from another block or the entire files score.

2.1.2A entDevReturnType Python Class Description

__init__(chi = 0.0, shannon = 0.0, estimate = 0.0, error = 0.0)	The constructor optional takes the four parameters, `chi`, `shannon`, `estimate` and `error` that initializes the respective member properties, or otherwise initializes them all to a value of 0.0.
ChiSquareValue	A read-only value that contains a Chi-Square distribution value.
chi_square_value	An alias for `ChiSquareValue`.
ChiSquare	A read-write property that contains a Chi-Square distribution value.
chi_square	An alias for `ChiSquare`.
ShannonValue	A read-only value that contains the Shannon Entropy distribution value.
shannon_value	An alias for `ShannonValue`.
Shannon	A read-write property that contains the Shannon Entropy distribution value.
shannon	An alias for `Shannon`
EstimateValue	A read-only value that contains the estimated Monte Carlo Pi Approximation value.
estimate_value	An alias for `EstimateValue`.
Estimate	A read-write property that contains the estimated Monte Carlo Pi Approximation value.
estimate	An alias for `Estimate`.
ErrorValue	A read-only value that contains the percentage of error from Pi that the `Estimate` is.
error_value	An alias for `ErrorValue`.

Error	A read-write property that contains the percentage of error from Pi the `Estimate` is.
error	An alias for `Error`.

2.1.3 The distributionType Python class

The `distributionType` class is a psuedo-class that is generated by `boost::python` that wraps a native C++ structure. It is used as a return type in methods dealing with character frequency in the `entropyDeviationType` Python class.

2.1.3A distributionType Class Description

__init__(value = 0, count = 0)	The constructor optionally takes two parameters that set the value of the `value` and `count` parameters, otherwise they are initialized to zero.
Value	A read-write property that represents the byte value in question, ranges from `0` to `255`.
value	An alias for `Value`.
Count	A read-write property that is the number of times the `Value` occurred.
count	An alias for `Count`.

2.1.4 The entDevType Python Class

The `distributionType` class is a psuedo-class that is generated by `boost::python` that wraps a native C++ class. It is used as the main interface to C++ by the `entropyDeviationType` pure Python class.

2.1.4A entDevType Class Description

__init__(blockSize = 8192)
__init__(data, blockSize = 8192, whole = False)
	Constructors that take the following parameters:
	
	`data` The contents of the file.
	`blocksize` The size of blocks to break data into.
	`whole` Indicates whether to perform whole file analysis or not.

setData(data)
setData(data, blockSize = 8192, whole = False)
	Sets the contents of a file to perform analysis upon. It takes the following parameters:
	
	`data` The contents of the file.
	`blocksize` The size of the blocks to break data into.
	`whole` Indicates whether to perform whole file analysis or not.

calculate()
calculate(x)
calculate(x,y)
	Methods that perform the actual entropy analysis that take the following parameters:
	
	`x` The block to calculate.
	`y` The ending block to calculate, forms a range of `x` through `y` blocks. Throws an instance of `ValueError` if any of the indices are invalid.

calculateDistribution(x,y)	Method that calculates the frequency distribution between a range of blocks starting with `x` and ending at `y`. Throws an instance of `ValueError` if the range `x` through `y` is invalid.
count()	Returns the size of the `list()` of blocks representing the file.
maxIndex()	The maximum index of the `list()` of blocks representing the file. Equivalent to `count()`-1 for values of `count()` that are greater than zero.
getDeviation(x,y)	Calculates the deviation between the blocks specified by `x` and `y`. Returns an `entDevReturnType` instance. Throws an

	instance of `ValueError` if the range specified by `x` through `y` is invalid.
getAllDeviations(x)	Calculates the deviation of *all* blocks in the file relative to the block at index **x**. Returns an **entDevReturnType** instance. Throws an instance of `ValueError` if the index specified by `x` is invalid.
getWholeFileDeviation(x)	Calculates the deviation of the entire file taken as a whole relative to the block at index `x`. Returns an `entDevReturnType` instance. Throws an instance of `ValueError` if the index specified by `x` is invalid.
getScore(x)	Retrieves the Chi Square, Shannon, and Monte Carlo Pi approximation scores for the block specified by `x`. Returns an instance of `entDevReturnType`. Throws an instance of `ValueError` if the index specified by `x` is invalid.
getWholeScore()	Retrieves the whole file score. Returns an instance of `entDevReturnType`.
getAllScores()	Retrieves the entropy scores for all blocks in the file. Returns an instance of `entDevReturnType`.
reset()	Reset the internal distribution/frequency statistics.
getDistribution()	Returns the `distributionType` calculated by the `calculateDistribution` method.

2.1.5 The xorTableSearchType Python Class

``xorTableSearchType`` is intended as a simple proof of concept that hopes to inspire incident response teams to rethink some of their approaches to identifying this sort of data. Specifically, the common method for identifying embedded executables is generally to attempt to brute force all possible XOR keys looking for combinations of the DOS magic header value, the common string contained within the DOS stub executable and/or the PE magic header. This is expensive both in terms of resources expended and time and is prone towards false positives and false negatives as the occurrence of these strings is not unique nor entirely uncommon, for instance all of them often occur on related mailing lists. Moreover, in order to reduce false positives many people have grown a dependence on strings commonly contained within the DOS stub header, which has no guarantee to actually exist.

The brute force itself is expensive, as teams often attempt to brute force all keys between 1 and 8 or more bytes across the entire file that generates $[2^8+2^{16}+2^{24}+2^{32+}2+2^{40}+2^{48}+2^{56}+2^{64}]$ keys and with roughly the complexity of:

$$(N*2^8)+((N/2^{16})*2^{16})+((N/2^{24})*2^{24})+((N/2^{32})*2^{32})+$$
$$((N/2^{40})*2^{40})+((N/2^{48})*2^{48})+((N/2^{56})*2^{56})+((N/^{264})*2^{64})$$

Where N is the length of the file in question. As such, this is a pretty intensive process and isn't realistic for tasks such as real-time scanning and similar. The approach demonstrated in this class pre-computes all possible one-byte XOR'd values for the strings MZ and PE\0\0. It then iterates across the blob of data looking first for a value that matches the MZ string and if found then scans within a certain limit (defaults to 512 bytes) for the string PE\0\0. If that value is found, then it uses the matching one-byte key at the offset 0x3C from the MZ string to extract the offset to the PE header, if the value extracted matches the location of the PE magic string previously found, then we classify it as having found an embedded PE. It is our suspicion that using this methodology will result in a lower false-positive and lower potential false-negative rate with a faster total overhead than methods iteratively looking for the various strings and/or looking for the generic DOS stub string (which may not even be present). This method is expected to result in lower false positives than the statistical approach at a much, much higher resource and time cost, but only for one-byte XOR keys. It is of course possible to utilize the same general algorithm for longer keys, largely bounded by the system resources. These longer key lengths were not implemented as this was only intended as a proof-of-concept that improves upon other public XOR brute force methods.

2.1.5A xorTableSearchType Python Class Description

__init__(maxoff = 512, base = 10)	Takes the parameter `maxoff` which indicates how far after an instance of the string MZ the class should look for the PE header magic value. Defaults to 512, specified in base 10 format but is changeable via the `base` parameter
openFile(name)	Opens and reads the file specified by `name`
findFirst()	Attempts to find the first instance of an embedded PE executable within another stream. Returns an instance of `xorTableReturnType` if any PE files were found, otherwise it throws an instance of `UserWarning`.
findAll()	Attempts to find ALL instances of embedded PE files within another stream. Returns a `list()` of `xorTableReturnType` instances, one for each PE file that was found. Otherwise it throws an instance of `UserWarning` if there were no embedded PE files located.

2.1.6 The xorTableReturnType Python Class

The `xorTableReturnType` class is a psuedo-class that is generated by `boost::python` that wraps a native C++ structure. It is used as a return type in *all* methods that return a value in the `xorTableSearchType` Python class.

2.1.6A xorTableReturnType Class Description

Offset	A read-write property that contains the offset within the file that the first byte of the encrypted PE file was found.
offset	An alias for the `Offset` property.
Key	A read-write property that contains the value of the key that the encrypted PE file found was encrypted with.
key	An alias for the `Key` property.

2.1.7 The xorTableType Python Class

The `xorTableType` class is a psuedo-class that is generated by boost::python that wraps a native C++ class. It is used as the main interface by the `xorTableSearchType` Python class.

2.1.7A xorTableType Class Description

__init__(max_peoff = 512)
__init__(data, max_peoff = 512)
	The first constructor that takes an optional integer, `max_peoff` that specifies how far in bytes the class should search after locating the DOS magic string value. The second constructor takes a string, `data` that is the contents of the file to be searched. The optional parameter `max_peoff` indicates how far in bytes the class should search after locating the DOS magic string value.

setData(data)	Takes a string, `data` that is the contents of the file to be searched.
findFirst()	Searches the contents of a file for an encrypted PE file; returns only the first embedded PE file found.
findAll()	Searches the contents of a file for *all* encrypted PE files contained within it.

2.2 C++ Class Descriptions

This extension is primarily written in C++. The aforementioned Python classes are wrappers to their respective C++ classes and one can also elect to further extend them or otherwise extract them for their own project.

The classes are written in portable C++11 and are free and clear from any platform specific constructs and as such should compile without complication on any platform that has a mostly standards compliant Standard Template Library and C++ compiler that supports C++11. Furthermore, as the classes primarily utilize specific integer widths, the classes should have no complication on either 32-bit or 64-bit platforms.

The classes were written initially in a Microsoft Visual Studio 2013 environment and then later exported to a Linux/G++ environment for Python. The extension at present does not compile cleanly as a Python module under Windows primarily due to the requirement that native extensions utilize the same MSVC library that the Python library was compiled against, which is an extremely outdated version. Moreover, the wrappers that convert STL C++ containers to Python data types apparently use Python C API functionality that changed or was removed in Python 3, specifically functions relating to `PyString's`. Aside from that, the classes should compile and operate cleanly against Python 3, however this has not been tested.

2.2.1 The chunk_t C++ structure

The ``chunk_t`` structure is utilized in several places throughout the Python extension and represents the base data type for what is referenced as blocks throughout this documentation. It is a simple structure containing only two members which are accessible via direct access. This structure is not accessible directly from the Python class interface and is abstracted away from the user.

2.2.1A chunk_t Structure Description

```
typedef std::shared_ptr< uint8_t > byte_buf_t;

struct chunk_t {
    byte_buf_t  buf;
    std::size_t len;

    chunk_t(void);
```

```
                chunk_t(const std::vector< uint8_t >& v);
                ~chunk_t(void);
        };

        typedef std::vector< chunk_t > chunk_vec_t;
```

byte_buf_t buf	A `std::shared_ptr` to `uint8_t` that points to the actual data of the chunk.
std::size_t len	An unsigned integer of type `std::size_t` that refers to the length of the memory pointed to by the `buf` member.

chunk_t(void)
chunk_t(std::vector< uint8_t >&)
	The constructors with the first being the default constructor that initializes `buf` to `nullptr` and `len` to zero. The second constructor takes a reference to `std::vector< uint8_t >` and initializes `buf` to point to dynamically allocated memory that is a copy of the `std::vector< uint8_t >`. `len` is initialized to contain the length of the `buf` member.

chunk_vec_t	A `std::vector` containing an array of `chunk_t` objects.

2.2.2 The dist_t C++ Structure

The `dist_t` structure is utilized in several places throughout the Python extension and represents the base data type for what is referenced as blocks throughout this documentation. It is a simple structure containing only two members which are accessible via direct access or via getter and setter methods, which are in place primarily as to take advantage of `boost::python` property related functionality. This class is the underlying C++ structure for the Python `distributionType` class.

2.2.2A dist_t Class Description

```
struct dist_t {
    uint8_t       value;
    std::size_t   count;

    dist_t(void);
    dist_t(uint8_t, std::size_t);
    ~dist_t(void);
    uint8_t getValue(void);
    void setValue(uint8_t);
    std::size_t getCount(void);
    void setCount(std::size_t);
};

typedef std::vector< dist_t > dist_vec_t;
```

uint8_t value	The `uint8_t` member that refers to the specific byte value in question.
std::size_t count	A `std::size_t` width integer that refers to the number of occurrences of `value`.

dist_t(void)
dist_t(uint8_t, std::size_t)
	The constructors, with the first taking no parameters and initializes `value` and `count` to values of zero. The second constructor initializes the values of `value` to its first parameter and `count` to its second.

uint8_t getValue(void)	A getter method that is utilized by the Python wrapper class `distributionType`. This method returns the `value` member. This method is called whenever a user accesses the `value` or `Value` property in a `distributionType` object in Python.

void setValue(uint8_t)	A setter method that is utilized by the Python wrapper class `distributionType`. This method sets the `value` member to the value of its only parameter. This method is called whenever a user sets the `value` or `Value` property in a `distributionType` object in Python.
std::size_t getCount()	A getter method that is utilized by the Python wrapper class `distributionType`. This method returns the `count` member. This method is called whenever a user accesses the `count` or `Count` property in a `distributionType` object in Python.
void setCount(std::size_t)	The setter method that is utilized by the Python wrapper class `distributionType`. It sets the `count` member to the value of its only parameter. This method is called whenever a user accesses the `count` or `Count` property in a `distributionType` object in Python.
dist_vec_t	A `std::vector` of `dist_t` objects. This is represented by a `list()` of `distributionType` instances in Python.

2.2.3 The entropy_retval_t C++ Structure

The `entropy_retval_t` structure is utilized extensively throughout the `entropy_t` and `entropy_wrapper_t` classes and represents the main return value object type of the `entDevReturnType` Python class.

It is a simple structure containing only the base data types and getter and setter methods so as to add properties to the related Python object. This class is the underlying C++ structure for the `entDevReturnType` Python object.

2.2.3A entropy_retval_t Structure Description

```
struct entropy_retval_t {
        long double chisquare;
        long double shannon;
        long double estimate;
        long double error;

        entropy_retval_t(long double, long double, long double, long double);
        ~entropy_retval_t(void);

       void setChiSquare(long double);
       long double getChiSquare(void);
       void setShannon(long double);
       long double getShannon(void);
       void setEstimate(long double);
       long double getEstimate(void);
       void setError(long double);
       long double getError(void);
};

typedef std::vector< entropy_retval_t > entropy_retvec_t;
```

long double chisquare	A `long double` that contains the value of the Chi Square distribution test score value or the Chi Square deviation value.
long double shannon	A `long double` that contains the value of the Shannon entropy distribution score value or the Shannon entropy deviation value.
long double estimate	A `long double` that contains the value of the Monte Carlo Pi approximation value or the Monte Carlo Pi deviation value.

long double error	A `long double` that contains a value representing the percent difference of `estimate` relative to Pi or the deviation value depending on the exact context of the objects usage.
entropy_retval_t(void) **entropy_retval_t(long double, long double, long double, long double)**	The two constructors with the first initializing all of the related members to `0.0` and the second initializing the `chisquare` member to the first parameter, the `shannon` member to the second and `estimate` and `error` to the third and fourth parameters.
void setChiSquare(long double)	A method that takes a `long double` parameter and sets the `chisquare` member's value to its value. This method is called whenever a user changes the value of chi_`square` in an instance of an `entDevReturnType` object in Python.
long double getChiSquare(void)	A method that returns the `long double` value of the `chisquare` member. This method is called whenever a user accesses the `chi_square` member of an instance of an `entDevReturnType` object in Python.
void setShannon(long double)	A method that takes a `long double` parameter and sets the `shannon` member to its value. This method is called whenever a user changes the value of `shannon` in an instance of an `entDevReturnType` object in Python.
long double getShannon(void)	A method that returns a `long double` value of the `shannon` member. This method is called whenever a user accesses the `shannon` member of an instance of an `entDevReturnType` object in Python.
void setEstimate(long double)	A method that takes a `long double` parameter and sets the `estimate` member to its value. This method is called whenever a user changes the value of `estimate` in an instance of an `entDevReturnType` object in Python.
long double getEstimate(void)	A method that returns a `long double` value of the `estimate` member. This method is called whenever a user accesses the `estimate` member of an instance of an `entDevReturnType` object in Python.
void setError(long double)	A method that takes a ``` ``long double`` ``` parameter and sets the ``` ``error`` ``` member to its value. This method is called whenever a user changes the value of `error` in an instance of an `entDevReturnType` object in Python.

long double getError(void)	A method that returns a ``long double`` value of the ``error`` member. This method is called whenever a user accesses the ``estimate`` member of an instance of an ``entDevReturnType`` object in Python.

2.2.4 The entropy_wrapper_t C++ Class

The `entropy_wrapper_t` class is a native wrapper that encapsulates the functionality of the `entropy_t` class. This encapsulation encompasses both translating exceptional events and errors into formats geared towards Python but also tying multiple disjointed parts of the entropy_t API into a single method or similar. It contains and stores the entire file, in a `std::vector`, all relevant `entropy_retval_t`'s for the analyzed data, an `entropy_retval_t` containing the scores for the entire file, a `std::vector` of `dist_t` objects for each blocks frequency distribution and a `std::vector` containing instances of `chunk_t` that are the actual blocks of the file.

2.2.4A entropy_wrapper_t Class Description

```
class entropy_wrapper_t {
        private:
                chunk_vec_t              m_cvec;
                entropy_retvec_t         m_retvec;
                entropy_retval_t         m_whole;
                bool                     m_wholeDone;
                entropy_t                m_entdev;
                std::vector< uint8_t >   m_data;
                std::vector< dist_t >    m_dist;

        protected:
        public:

        entropy_wrapper_t(std::size_t bs = 8192);
        entropy_wrapper_t(const std::vector< uint8_t >&, std::size_t bs = 8192, bool whole = false);
        ~entropy_wrapper_t(void);

        void reset(void);

        void setDataOverload(const std::vector< uint8_t >&);
        void setData(const std::vector< uint8_t >&, std::size_t bs = 8192, bool whole = false);

        std::size_t getCount(void);
        std::size_t getMaxIndex(void);

        void calculate(void);
        void calculate(std::size_t);
        void calculate(std::size_t, std::size_t);
        entropy_retval_t getChunkScore(std::size_t);
        entropy_retval_t getWholeFileScore(void);
        entropy_retvec_t getAllChunkScores(void);

        entropy_retval_t getDeviation(std::size_t, std::size_t);
        entropy_retvec_t getAllDeviations(std::size_t);
        entropy_retval_t getWholeDeviation(std::size_t);

        void calculateDistribution(std::size_t, std::size_t);
        dist_vec_t getDistribution(void);
};
```

chunk_vec_t m_cvec	A `std::vector` of `chunk_t` structures that contain the contents of the file being analyzed.
entropy_retvec_t m_retvec	A `std::vector` of `entropy_retval_t` objects corresponding to the scores for each block of the file.
entropy_retval_t m_whole	A `std::vector` containing the scores for the entire file.
bool m_wholeDone	A Boolean value that is set to `true` when the entire file has had its scores calculated, otherwise it is `false`.
entropy_t m_entdev	An instance of an **entropy_t** class, the main class of the extension**.**
std::vector< uint8_t > m_data	A `std::vector` of the data containing in the file that is being analyzed.
std::vector< dist_t > m_dist	A `std::vector` of `dist_t` objects corresponding to the frequency distribution last calculated. It is empty if this has never occurred.
entropy_wrapper_t(std::size_t bs = 8192) **entropy_wrapper_t(const std::vector< uint8_t >&,std::size_t bs = 8192, bool whole = false)**	The constructors for the class. The first takes an optional parameter, `bs`, which indicates what size blocks the input file should be split into. The second takes a `std::vector` that contains the data to be analyzed, followed by an optional block size and finally an optional Boolean value that indicates whether the user would like to have whole file analysis performed on the data or not. The default block size is `8192` bytes and the default regarding whether to perform whole file analysis is false; indicating that it will not be performed.
void reset(void)	Reset the internal distribution/frequency statistics.
void setDataOverload(const std::vector< uint8_t >&) **void setData(const std::vector< uint8_t >&, std::size_t bs = 8192, bool whole = false)**	These methods will set the `m_data` `std::vector` to contain the contents of its first parameter. The second method will optionally have a `bs` and `whole` parameter that indicate what block size should be used and whether to perform whole file analysis or not. The default block size is `8192` bytes and whole file analysis is not performed. The first method is an overloaded version of the second, however due

	to restrictions in `boost::python`, it was given a different method name.
std::size_t getCount(void)	Returns a `std::size_t` containing the value of the number of elements in the `m_data` `std::vector`, this is how many bytes long the file or data stream was at the time that it was initialized via the constructor or `setData()` method of this class.
std::size_t getMaxIndex(void)	Returns a `std::size_t` containing the value of the maximum valid index that can be used on the m_data `std::vector`. It is equivalent to calling getCount() and subtracting 1 providing that is at least 1 element in the `std::vector`.
void calculate(void) **void calculate(std::size_t ce)** **void calculate(std::size_t cs, std::size_t ce)**	Methods that perform the actual entropy analysis calculations. The first takes no parameters and operates on all blocks in the file. The second specifies only an ending block and calculates the scores for all blocks before the index `ce`. The last takes both a starting and ending index between the rages of `cs` and `ce` and operates on blocks within that range. If any of the indices are invalid, it will silently return raising a `ValueError` in Python.
entropy_retval_t getChunkScore(std::size_t v1)	This method returns the scores for chunk number `v1`; if the index is invalid the function will silently return but raising a `ValueError` in Python
entropy_retval_t getWholeFileScore(void)	Retrieves the whole file score, first calculating it is necessary.
entropy_retvec_t getAllChunkScores(void)	Retrieves the scores for all chunks in the file and returns them in a `std::vector` containing instances of `entropy_retval_t`.
entropy_retval_t getDeviation(std::size_t v1, std::size_t v2)	Retrieves the deviation scores for all chunks within the range of `v1` through `v2`. If this range is invalid the method silently returns, however it will raise a `ValueError` inside of Python.
entropy_retvec_t getAllDeviations(std::size_t v1)	Retrieves the deviation score for all blocks relative to `v1`. If v1 specifies an invalid index, the method will silently return but will however raise a `ValueError` in Python.
entropy_retval_t getWholeDeviation(std::size_t v1)	Retrieves the deviation score for the entire file relative to `v1` and returns a `std::vector` containing instances of `entropy_retval_t`.
void calculateDistribution(std::size_t cs, std::size_t ce)	Calculates the distribution frequency for all `256` possible characters within the range of blocks specified by `cs` through `ce`. If the range specified is invalid, the method silently returns however it will raise a ValueError inside of Python.
dist_vec_t getDistribution(void)	Retrieves the `dist_vec_t` of the distribution frequency for all 256 possible characters as previously calculated via the `calculateDistribution()` method. This vector is empty if the calculations have not been performed.

2.2.5 The entropy_t C++ class

The entropy_t class provides the main functionality of the extension and is the 'lowest level' interface available. In terms of hierarchy, when accessed in Python, it is accessed via a C++ wrapper class, entropy_wrapper_t, which in turn is wrapped by a pseudo Python class, entDevType, which is wrapped and utilized by a pure Python class, entropyDeviationType. As this includes multiple layers of indirection and other functionality such as converting strings and lists to vectors, it is expected that utilizing this class directly will yield much better performance.

That said, while a stated goal of this extension was to substantially increase the performance of these tasks, it was not written with the goal of being super-fast production quality code.

2.2.5A entropy_t Class Description

```
#define IDEAL_SIZE 256

class entropy_t
{
    private:
        std::size_t                                  m_bsize;
        std::array< std::size_t, IDEAL_SIZE >        m_dist;
        long double                                  m_freedom;
        long double                                  m_ideal;
        long double                                  m_csquare;
        long double                                  m_shannon;
        std::size_t                                  m_count;

        std::size_t                                  m_inside;
        std::size_t                                  m_outside;
        long double                                  m_estimate;
        long double                                  m_error;

    protected:
        inline long double getEstimate(void);
        inline long double getError(long double);

    public:
        entropy_t(std::size_t bs = 8192);
        entropy_t(const std::vector< uint8_t >&, std::size_t bs = 8192);
        ~entropy_t(void);

        chunk_vec_t slice(const std::vector< uint8_t >&, std::size_t = 0);
        void update(const std::vector< uint8_t >&);
        void update(const std::vector< uint8_t >&, std::size_t);
        void update(const chunk_t&);
        void update(const uint8_t*, std::size_t);

        void reset(void);
        void calculateValues(void);
        entropy_retval_t getValues(bool calculate = true);
        entropy_retval_t getBlockDeviation(entropy_retval_t& v1, entropy_retval_t& v2);
        entropy_retvec_t getBlockDeviationFromAllBlocks(const entropy_retvec_t&, std::size_t);
        dist_vec_t getDistribution(void);
};
```

std::size_t m_bsize	A `std::size_t` member that indicates the size of chunks the class operates on; defaults to `8192`.
std::array< std::size_t, IDEAL_SIZE > m_dist	A `std::size_t` of `IDEAL_SIZE` elements (`256`) that contains a `std::size_t` of the frequency distribution of each possible byte encountered.
long double m_freedom	The Chi-Square distribution degrees of freedom; or `IDEAL_SIZE` – 1.
long double m_ideal	A member variable that is initialized to the value of `IDEAL_SIZE`, or `256`, representing the range of characters that can occur.
long double m_csquare	This member variable stores the calculate Chi-Square distribution score.
long double m_shannon	This member variable stores the Shannon entropy analysis score.
std::size_t m_count	This member variable is a count of the number of bytes analyzed.
std::size_t m_inside	This represents the number of bytes that fall inside of the 1x1 area in the Monte Carlo method Pi Approximation test.
std::size_t m_outside	This member variable represents the number of bytes that fell outside of the 1x1 area in the Monte Carlo method Pi Approximation test.
long double m_estimate	This member variable stores the estimate value in the Monte Carlo method Pi approximation test.
long double m_error	This member variable stores the percent difference between `m_estimate` and the value of Pi.

entropy_t(std::size_t bs = 8192) **entropy_t(const std::vector< uint8_t >& v, std::size_t bs = 8192)**	The constructors with the first taking only an optional `std::size_t` parameter that specifies the size of blocks that will be analyzed and defaulting to `8192` bytes if not otherwise provided. The second takes a `std::vector` of `uint8_t`'s that contains the actual data we will analyze, and the optional `bs` parameter specifying the length of blocks desired.
chunk_vec_t slice(const std::vector< uint8_t >&, std::size_t = 0)	This method will split the vector specified in the first parameter into `chunk_t`'s the size of its second parameter. If the optional second parameter is not specified or explicitly passed with a value of zero, then the block size that was specified in the constructor is used to determine the size of the blocks. This method returns a `std::vector` of `chunk_t` objects representing the file or `std::vector` that was passed as the first parameter.
void update(const std::vector< uint8_t >& v) **void update(const std::vector< uint8_t >& c, std::size_t bs)** **void update(const chunk_t& v)** **void update(const uint8_t* ptr, std::size_t len)**	These methods all take a block of data specified in various formats and perform the actual analysis upon them. The first takes a parameter of a `std::vector` of `uint8_t`'s, the second will first split the `std::vector` into a `chunk_vec_t` containing blocks corresponding to the size of the second parameter and operates on them. The third takes a reference to a `chunk_t` directly and the last method is the one that all of the others call, which takes a raw pointer to `uint8_t` that points to the data to operate on and is of a length specified in the second parameter. If by the time the last method which takes a raw pointer has a first parameter which is equal to `nullptr` or a second parameter that is zero, then an instance of `std::runtime_error` is thrown.
void reset(void)	Reset the internal distribution/frequency statistics variables.
void calculateValues(void)	This method will take the various members, such as the distribution metrics stored in `m_dist` and the count of hits inside and outside the 1x1 area and calculate the actual scores thereby setting the values of the related member variables.
entropy_retval_t getValues(bool calculate = true)	This method is intended to be called after a call to `update()` and depending on the value of the calculate parameter, after a call to `calculateValues()` has been made. If the parameter calculate is `true`, then this calculation is performed automagically,

	otherwise it is skipped when provided a parameter whose value is `false`. It returns an `entropy_retval_t` with the scores calculate for the blocks in question.
entropy_retval_t getBlockDeviation(entropy_retval_t& v1, entropy_retval_t& v2)	This method takes two `entropy_retval_t`'s as parameters, which are the scores returned for a block specified by `v1` and the scores of a block specified by `v2`. It then calculates the difference between the two blocks and stores these values in an `entropy_retval_t` structure which is returned to the user. If either of the parameter values are invalid, an instance of `std::runtime_error` is thrown.
entropy_retvec_t getBlockDeviationFromAllBlocks(const entropy_retvec_t& blocks, std::size_t bnum)	This method is akin to the `getBlockDeviation()` method, and indeed has its functionality provided by iterating across the `entropy_retvec_t` parameter `blocks` and calculates the difference between each blocks score values and the block specified in the `bnum` parameter. If the index specified in the second parameter is invalid, then an instance of `std::runtime_error` is thrown. This method returns a `std::vector` of `entropy_retval_t`'s that contain the percentage of difference between each block relative to the `bnum` parameter.
dist_vec_t getDistribution(void)	This method iterates across the `m_dist` member and records the occurrence counts for each of the `256` possible bytes and returns them in a `std::vector` of `dist_t`'s.

2.2.6 The xor_table_ret_t C++ structure

The `xor_table_ret_t` structure is the object type returned by all functionality relating to the XOR table search family of classes. In Python, the `xorTableReturnType` wraps it directly. It is a relatively simple structure, that contains only two member variables and getter and setter methods that are bound to properties in Python.

2.2.6A xor_table_ret_t Structure Description

```
struct xor_table_ret_t {
	std::size_t	offset;
	std::uint16_t	key;

	xor_table_ret_t(void) : offset(0), key(0) { return; }
	~xor_table_ret_t(void) { return; }

	std::size_t getOffset(void) { return offset; }
	void setOffset(std::size_t o) { offset = o; return; }

	std::uint16_t getKey(void) { return key; }
	void setKey(std::uint16_t k) { key = k; return; }
};

typedef std::vector< xor_table_ret_t > xor_table_retvec_t;
```

std::size_t offset	The offset at which the embedded PE file was found.
std::uint16_t key	The key at which the PE file located at `offset` was encrypted with. This will be an 8-bit value stored in a 16-bit integer.
xor_table_ret_t(void)	The basic constructor; it initializes both the `offset` and `key` variables to zero.
std::size_t getOffset(void)	This method returns the `offset` variable and exists to allow seamless property integration with Python.
void setOffset(std::size_t)	This method sets the `offset` variable to the value passed for its first parameter. This setter method exists to allow seamless property integration with Python.
std::uint16_t getKey(void)	This method returns the value of the `key` variable. This getter method exists to allow seamless property integration with Python.

void setKey(std::uint16_t)	This method sets the value of the `key` variable to the value of the parameter it is passed. This setter method exists to allow seamless property integration with Python.
typedef std::vector< xor_table_ret_t > xor_table_retvec_t	A data-type that is a `std::vector` containing instances of `xor_table_ret_t`. It is used for instance by functionality the searches for more than one instance of an embedded PE file. The `std::vector` is converted to a `list()` when it is exported into Python.

2.2.7 The xor_table_wrapper_t C++ Class

The `xor_table_wrapper_t` class is the class directly wrapped and exported to Python as the `xorTableSearchType`. It encapsulates all of the functionality of the native "lower level" `xor_table_t` class. As with all of the classes and structures in this family, its "proof-of-concept" nature is clearly shown by the simplicity of the interface.

2.2.7A xor_table_wrapper_t Class Description

```
class xor_table_wrapper_t {
    private:
        xor_table_t    m_xor;

    protected:
    public:
        xor_table_wrapper_t(std::size_t max_peoff = 512);
        xor_table_wrapper_t(const std::vector< uint8_t >&, std::size_t max_peoff = 512);
        ~xor_table_wrapper_t(void);

        void setData(const std::vector< uint8_t >&);
        xor_table_ret_t find_first(void);
        xor_table_retvec_t find_all(void);
};
```

xor_table_wrapper_t(std::size_t max_peoff = 512)
xor_table_wrapper_t(const std::vector< uint8_t >&, std::size_t max_peoff = 512)
	The first constructor takes only an optional `std::size_t` parameter which dictates how far after an occurrence of the 'MZ' string the class should search for the 'PE\0\0' string. By default, it is limited to 512 bytes which should adequately cover most executable files. The second constructor takes as its first parameter a reference to a `std::vector` of `uint8_t`'s that contain the data of the file to be analyze and again has an optional second parameter that dictates how far to search for the PE magic string.

void setData(const std::vector< uint8_t >&)	This method takes a reference to a `std::vector` of `uint8_t`'s that contain the data set to be analyzed for an embedded PE file. If the data was already set via a previous call to this method or one of the constructors, then the data is reset.

xor_table_ret_t find_first(void)	This method will search an executable looking for an encrypted or unencrypted instance of a PE file. It will search the entire file but stop after the first executable is found. If no executable is found, then in C++ it returns an empty `xor_table_ret_t` with default values all set to zero, however in Python it will raise a `UserWarning` indicating that it was unable to locate a PE file.
xor_table_retvec_t find_all(void)	This method performs the exact same functionality as the find_first() method, however instead of stopping its search after a single instance of a PE file is located, it will continue its search throughout the remainder of the data searching for more. Thus, this function will always search the entirety of the data set. It will return a `std::vector` of `xor_table_ret_t`'s—one for each PE file that is found. If no PE files were identified, then the std::vector will be empty and a `UserWarning` exception is thrown in Python.

2.2.8 The xor_table_t C++ Class

The xor_table_t class is the bottom layer C++ class for all related XOR table search functionality. All Python or C++ objects relating to this functionality eventually call into this class. It is only slightly more complex than the wrapper class, which was written to create an interface that better models the expected use of the class and as a natural work-around for constraints placed on the API by exporting an interface to Python.

2.2.8A xor_table_t Class Description

```
class xor_table_t
{
    private:
        std::size_t                         m_span;
        std::array< uint8_t, MZ_SIZE >      m_mzv;
        std::array< uint8_t, PE_SIZE >      m_pev;
        uint8_t*                            m_vec;
        std::size_t                         m_siz;

    protected:
        void init_tables(void);
        bool find_pe(std::size_t kidx, std::size_t off);
        uint32_t get_peoff(std::size_t, uint8_t);
        bool findAtOffset(xor_table_ret_t& out, std::size_t offset = 0);

    public:
        xor_table_t(std::size_t max_peoff = 512);
        xor_table_t(const std::vector< uint8_t >& v, std::size_t max_peoff = 512);
        ~xor_table_t(void);
        void set_file(const std::vector< uint8_t >&);
        bool find_first(xor_table_ret_t&);
        bool find_all(xor_table_retvec_t&);
};
```

xor_table_t(std::size_t max_peoff = 512)
xor_table_t(const std::vector< uint8_t >& v, std::size_t max_peoff = 512)

The first constructor takes only an optional `std::size_t` parameter which dictates how far after an occurrence of the 'MZ' string the class should search for the 'PE\0\0' string. By default, it is limited to 512 bytes which should adequately cover most executable files. The second constructor takes as its first parameter a

	`std::vector` of `uint8_t`'s that contain the data of the file to be analyze and again has an optional second parameter that dictates how far to search for the PE magic string.
void set_file(const std::vector< uint8_t >&)	This method takes a reference to a `std::vector` of `uint8_t`'s that contain the data set to be analyzed for an embedded PE file. If the data was already set via a previous call to this method or one of the constructors, then the data is reset.
bool find_first(xor_table_ret_t&)	This method will search an executable looking for an encrypted or unencrypted instance of a PE file. It will search the entire file but stop after the first executable is found. If no executable is found, then it will return false and the parameter passed to the function is returned with both members of the structure initialized to zero. Otherwise the members refer to the offset and key that the embedded PE file was found.
bool find_all(xor_table_retvec_t&)	This method performs the exact same functionality as the find_first() method, however instead of stopping its search after a single instance of a PE file is located, it will continue its search throughout the remainder of the data searching for more. Thus, this function will always search the entirety of the data set. It will return true if any PE files were identified and the offset and key for each still be stored in an instance of `xor_table_ret_`t which is stored in a `std::vector`. If no PE files were identified, then the std::vector will be empty and the method will return false

3.0 The epfind.py Utility

The edfind.py utility is a Swiss army knife of sorts for entropy analysis that demonstrates the various functionality of the extension module. It utilizes every aspect of the module and additionally builds example functionality on top of it. For instance, a purely proof-of-concept demonstration option attempts to automatically check for “suspect” blocks. It was written against a single file with a one-byte XOR encrypted executable contained within it and as such the default functionality is not expected as an out of the box solution to meet all circumstances, it is purely for demonstration. However, when run over approximately 1600 randomly selected Microsoft Office Word documents (doc and docx) and PDF documents it exhibited approximately a 20% false positive rate, which is relatively impressive given that it was written against a single file. Less impressive is that a much smaller sample set that contained XOR encrypted executables had a false negative rate of about 6 in 10 files.

With more thorough analysis and a better attempt at fine tuning the various functionality, it is expected that both the false positive and false negative rate could be significantly improved.

In terms of other functionality, the utility allows you to view Chi-Square distribution, Shannon entropy and Monte Carlo method Pi approximation scores for arbitrarily sized blocks of a file or the entire file itself. Moreover, it can be run in a deviation mode that retrieves the differences between arbitrarily sized blocks of a file, or the entire file itself against any one or more of the blocks.

While this document attempts to be fairly in-depth and cover all aspects of the module, the source code and edfind.py utility should be considered the premiere authoritative source for information on the subject.

3.0.1 epfind.py Command Line Options

Below we will discuss the various command line options for the utility and in the next section we will demonstrate the usage of the utility to identify hidden data streams.

```
usage: edfind.py [-h] [--blocksize BLOCKSIZE] [--blockscore] [--wholescore]
          [--blockdev] [--blocknumber BLOCKNUMBER] [--wholedev]
          [--xydev XYDEV XYDEV] [--seqdev] [--seqxy SEQXY SEQXY]
          [--suspect] [--frequency] [--freqcount FREQCOUNT]
          [--freqxy FREQXY FREQXY] [--xor] [--xorall]
          file
```

file	The input file to scan
-h, --help	show this help message and exit
--blocksize BLOCKSIZE, -b BLOCKSIZE	The size of the blocks to split the input file into; specified in bytes. If this option is not provided a default blocksize of 8192 bytes is used.
--blockscore, -s	This option will retrieve and print the Chi, Shannon and Pi approximation scores for every block in the file.
--wholescore, -w	The option is akin to `–blockscore`, however it calculates the scores against the entire file.
--blockdev, -d	The option retrieves the deviation for a given block of data relative to all other blocks in the file. It requires the additional option `–blocknumbe.`
--blocknumber BLOCKNUMBER, -n BLOCKNUMBER	The block number to perform `–blockdev` against, specified in hexadecimal format.
--wholedev, -o	Akin to `–blockdev`, however it retrieve the block specified by `–blocknumber`'s deviation relative to the entire file's scores.
--xydev XYDEV XYDEV, -y XYDEV XYDEV	Retrieves and prints a blocks deviation relative to another specified block.
--seqdev, -q	This option calculates and prints the deviation of a block relative to its neighbor blocks.
--seqxy SEQXY SEQXY, -e SEQXY SEQXY	Calculates and prints the sequential deviation of a range of blocks.
--suspect, -u	Experimental functionality intended as a proof of concept that couples together multiple aspects of the extension modules functionality and attempts to automatically identify suspect blocks.
--frequency, -f	This causes the frequency of each byte encountered for all blocks to be retrieved and printed; Especially useful for identify the beginning and end of XOR encrypted streams with short key lengths
--freqcount FREQCOUNT, -c FREQCOUNT	This option implies and subsequently modifies the `–frequency` option such that only FREQCOUNT of the most common bytes have their frequency printed out. For instance, specifying a FREQCOUNT of 4 will retrieve and print only the 4 most common byte values for each block.
--freqxy FREQXY FREQXY, -r FREQXY FREQXY	Akin to `–frequency`, however instead of retrieving the scores for every block it will only retrieve and print the scores between the two parameter indices specified.
--xor, -x	Attempts to locate any one-byte XOR encrypted PE files embedded within the file via pre-

	computer tables. Stops after the first instance is found.
--xorall, -a	Attempts to locate any one-byte XOR encrypted PE files embedded within the file via pre-computer tables. This option will search for all embedded executables and not just the first.

3.1.0 epfind.py Example Usage

Generally, it’s easier to understand a given feature set for a tool through demonstration. As such, I have created a series of Word and PDF documents that I embedded a XOR encrypted executable into. The executable in question is not malware and thus is going to deviate from the typical data that one could expect to find in the wild, however it felt like a happy medium rather than dealing with live malware for the sake of demonstration. The executables in question are standard Windows system binaries, particularly cmd.exe and wowreg32.exe from a standard Windows 8.1 installation. In a couple instances these binaries were packed as that more properly models live data as something approaching over 90% of malware is distributed in a packed format. Interestingly, this makes the analysts job easier as the compression and/or encryption functionality provided by most packers increases the data entropy and makes it easier to spot.

When embedded into the files, no regard was given to making sure that it was properly inserted into the OLE streams or similar and a random offset was chosen and the executable encrypted with an unknown value but known length key and inserted into the host file. We will go through each of the features looking at this data now to give the potential user a better feel for what they would be looking for in a live data set and how to use the tool properly.

3.1.1 Analysis of a PDF Document with an embedded executable encrypted with a one-byte key

We will start with a file that has an embedded PE file with a one-byte key used to XOR encrypt it. The file in question is a copy of Bruce Dang’s 2008 Blackhat Japan talk pertaining to parsing Microsoft Office formats as part of analyzing and countering these exact threats. If you are not familiar with the talk already, it’s a decent read and is recommended. The file itself is a PDF containing, presumably, the PowerPoint exported slide deck he used for the talk. As such, its contents are primarily images, which will exhibit compression and other patterns not entirely dissimilar from encryption. As the information entropy for a single byte key models the same distribution as when it is not encrypted, this is in some ways a harder case to identify than encrypted executables with longer key lengths. In other ways, its actually incredibly easy to identify and in this example we demonstrate the difficulty and catch-22 nature that the attacker encounters when utilizing this methodology—longer keys are more secure and harder to identify via frequency analysis, but they deviate significantly from the host data in most circumstances. Shorter keys are easy to identify via frequency analysis, however their distribution is closer to what we would expect from the general host file.

Starting at the most obvious place, the beginning, we first examine the block scores for the file. We use the default of 8192 bytes as that again appears to be a happy medium. Several anti-virus industry white papers on the average size of malware show that they are (a) growing in size over time; and (b) tend to be in the several hundred kilobyte range. Smaller blocks result in more overhead and worse performance, whereas larger blocks can result in the data from a hidden data stream essentially being lost in the noise of the host file it's embedded into. The default block size was chosen largely at random and while it has shown itself to be fairly suitable, your own personal mileage may vary and you are encouraged to experiment and determine if a different block size better fits your particular use case.

At any rate, we first examine the block scores of the file:

```
$ bin/edfind.py -s BlackHat-Japan-08-Dang-Office-Attacks-ONE.pdf | less
FILE: BlackHat-Japan-08-Dang-Office-Attacks-ONE.pdf BLOCK COUNT: 286 BLOCK SIZE: 8192

    ALL SCORES
        BN:   0         C: 13022.5000    S: 7.4409     ES: 3.4414     ER: 8.7120
        BN:   1         C: 573.6250      S: 7.9530     ES: 3.1445     ER: 0.0935
        BN:   2         C: 808.3750      S: 7.9414     ES: 3.2266     ER: 2.6334
        BN:   3         C: 679.3750      S: 7.9487     ES: 3.2031     ER: 1.9210
        BN:   4         C: 4766.4375     S: 7.7627     ES: 3.1719     ER: 0.9547
        BN:   5         C: 664.5625      S: 7.9483     ES: 3.1328     ER: 0.2803
        BN:   6         C: 573.0000      S: 7.9540     ES: 3.2969     ER: 4.7100
        BN:   7         C: 817.1250      S: 7.9420     ES: 3.1875     ER: 1.4402
        BN:   8         C: 564.5625      S: 7.9519     ES: 3.3125     ER: 5.1595
        BN:   9         C: 5219.1250     S: 7.7273     ES: 3.1562     ER: 0.4644
        BN:   A         C: 696.5625      S: 7.9474     ES: 3.2383     ER: 2.9858
        BN:   B         C: 839.5625      S: 7.9375     ES: 3.2852     ER: 4.3701
        BN:   C         C: 3281.8750     S: 7.7451     ES: 2.9766     ER: 5.5443
        BN:   D         C: 4168.2500     S: 7.6538     ES: 2.9219     ER: 7.5197
        BN:   E         C: 4341.1875     S: 7.6421     ES: 2.8867     ER: 8.8292
        BN:   F         C: 4242.0000     S: 7.6451     ES: 2.8867     ER: 8.8292
        BN:  10         C: 4847.0000     S: 7.6063     ES: 2.5977     ER: 20.9395

 […]
```

Here, first we have an output line that seems self-evident but contains the file name being analyze, the number of blocks the file was split into and the size of the blocks in question. For any given block number, we can determine the offset into the file by multiplying the block number by the block size. We passed the `–s` option, which indicates that we would like to calculate and print the Chi, Shannon and Monte Carlo Pi approximation scores for each block. As noted throughout the documentation, the Pi approximation has shown itself to be less useful than hoped and will likely be removed at a future point in time. Essentially the idea is that as the data becomes more random, that the number of points that intersect our 1x1 area will increase and the estimated value will get closer and closer to approaching the value of Pi. What the result thus far has been is that the block sizes are potentially too small for this method to be overly useful and the numbers produced are essentially sort of random in most instances and thus not overly useful. This will likely be replaced by another metric, potentially standard deviation, which is more useful for this task.

At any rate, the lines are in the format of:

```
BN: <Block Number> C: <Chi-Square Score> S: <Shannon Score> ES: <Pi
Approximation Estimate> E: <Pi approximation error>
```

The Estimate is the actual value we calculated whereas the Error value is the percent of difference between the Estimate and the actual value of Pi. What testing has thus far revealed is that as the data becomes more random, the Chi value decreases with XOR encrypted data that has a key as long as the data (effectively a One-Time Pad) having a value generally in the 200-300 range. The Shannon entropy score seems to peg almost precisely at 7.97 in those instances and as noted, in theory the Pi approximation should approach Pi but often does not.

Moreover, we can see from those first 16 blocks that there is a bit of variance between each block. What encrypted data, as we will see shortly tends to do providing the block size is small enough is result in a far greater uniformity across blocks than native document data tends to. This is to say that blocks 0x00 through 0x0C have a fair amount of variation in them, whereas the remaining blocks displayed start to look more like a typical cipher-text in that the Chi scores are far more uniform. However, the scores are not quite high or low enough for our purposes. Moreover, as you use the tool more, you will begin to note that there is a natural ebb and flow of the data with punctuations by outlier blocks. For instance, blocks 0x0A and 0x0B are outliers that deviate significantly from the neighboring blocks in terms of Chi and we can see from the Shannon score that they have a slightly higher density of information entropy. Whereas the blocks that follow show a growing lack of entropy before finally reaching a uniformity in the 4000 range for the Chi value. The Shannon scores also begin to mostly become uniform and while the Pi scores show a lot of deviation in terms of error, we would take care to note that the numbers to the left of the decimal and to a lesser degree in the 10s place to the right of it also show some level of uniformity.

If we look at the sequential blocks of this segment of the file in its entirety, we find the following:

```
BN:  B     C: 839.5625      S: 7.9375     ES: 3.2852     ER: 4.3701
BN:  C     C: 3281.8750     S: 7.7451     ES: 2.9766      ER: 5.5443
BN:  D     C: 4168.2500     S: 7.6538     ES: 2.9219     ER: 7.5197
BN:  E     C: 4341.1875     S: 7.6421     ES: 2.8867     ER: 8.8292
BN:  F     C: 4242.0000     S: 7.6451     ES: 2.8867     ER: 8.8292
BN: 10     C: 4847.0000     S: 7.6063     ES: 2.5977     ER: 20.9395
BN: 11     C: 4757.6250     S: 7.6122     ES: 2.8008     ER: 12.1684
BN: 12     C: 4507.0000     S: 7.6192     ES: 2.7812     ER: 12.9561
BN: 13     C: 4528.1250     S: 7.6293     ES: 2.6484     ER: 18.6206
```

BN: 14	C: 4418.6250	S: 7.6349	ES: 2.8086	ER: 11.8564
BN: 15	C: 4794.3125	S: 7.6181	ES: 2.8047	ER: 12.0122
BN: 16	C: 4382.9375	S: 7.6404	ES: 2.8125	ER: 11.7011
BN: 17	C: 4591.1875	S: 7.6272	ES: 2.7031	ER: 16.2208
BN: 18	C: 4767.5625	S: 7.6310	ES: 2.7695	ER: 13.4341
BN: 19	C: 4695.2500	S: 7.6263	ES: 2.9102	ER: 7.9527
BN: 1A	C: 4832.8750	S: 7.6253	ES: 2.8359	ER: 10.7779
BN: 1B	C: 4919.8750	S: 7.6122	ES: 2.9375	ER: 6.9478
BN: 1C	C: 4692.7500	S: 7.6290	ES: 2.9336	ER: 7.0902
BN: 1D	C: 4621.6875	S: 7.6288	ES: 2.9414	ER: 6.8058
BN: 1E	C: 3098.8125	S: 7.7362	ES: 2.8555	ER: 10.0202
BN: 1F	C: 2472.5000	S: 7.7872	ES: 2.9414	ER: 6.8058
BN: 20	C: 2060.5625	S: 7.8203	ES: 2.6875	ER: 16.8965
BN: 21	C: 2042.5000	S: 7.8233	ES: 2.9727	ER: 5.6830
BN: 22	C: 1744.5625	S: 7.8512	ES: 2.9102	ER: 7.9527

The blocks 0x0D through 0x1D match the sort of pattern we might expect when looking for hidden data streams. There is a fair amount of uniformity in the Chi and Shannon scores, and the Pi approximation values are semi-uniform and deviate a fair amount from the surrounding blocks. If we look, very specifically we see some "kinda sorta" uniformity in the Error percentage. Moreover, the block immediately preceding this series of blocks shows a high rate of deviation from the block immediately preceding it and conforms more closely to the blocks that follow it. This generic pattern is generally the result of multiple types of data in one block and marks a shift in the underlying data. That is to say, if it is an embedded executable it is relatively unlikely it will start immediately at the beginning of one of our blocks and as such we will end up with both the underlying host file data and the embedded data mixed into a block. This exhibits itself as a large deviation followed by a smaller deviation. That said, we will add the block range 0x0D through 0x1D to our suspicious list of blocks and continue analyzing with the intention of returning to those blocks and examining them in more detail later.

Moving further through the file, we eventually come to the following section of blocks:

BN: 28	C: 624.0000	S: 7.9501	ES: 3.2539	ER: 3.4517
BN: 29	C: 5181.0000	S: 7.7273	ES: 3.3711	ER: 6.8079
BN: 2A	C: 17580.6875	S: 7.4266	ES: 3.3125	ER: 5.1595
BN: 2B	C: 6910.9375	S: 7.6884	ES: 3.2773	ER: 4.1421
BN: 2C	C: 10664.3125	S: 7.4915	ES: 3.4141	ER: 7.9808
BN: 2D	C: 15791.9375	S: 7.4749	ES: 3.2734	ER: 4.0277
BN: 2E	C: 770.8125	S: 7.9378	ES: 3.1602	ER: 0.5874
BN: 2F	C: 17423.9375	S: 7.3913	ES: 3.2578	ER: 3.5674
BN: 30	C: 21339.8750	S: 7.1488	ES: 3.5898	ER: 12.4866
BN: 31	C: 655.1250	S: 7.9472	ES: 3.1719	ER: 0.9547
BN: 32	C: 666.6875	S: 7.9456	ES: 3.2070	ER: 2.0405
BN: 33	C: 87727.1250	S: 6.1935	ES: 2.2109	ER: 42.0932
BN: 34	C: 50754.6250	S: 6.3530	ES: 2.2695	ER: 38.4247

```
BN:  35     C: 56675.8750     S: 6.3096     ES: 2.3789     ER: 32.0604
BN:  36     C: 51845.6250     S: 6.3474     ES: 2.3047     ER: 36.3132
BN:  37     C: 57290.8125     S: 6.3268     ES: 2.4375     ER: 28.8859
BN:  38     C: 58930.5625     S: 6.3281     ES: 2.3633     ER: 32.9335
BN:  39     C: 50657.1875     S: 6.3146     ES: 2.5352     ER: 23.9211
BN:  3A     C: 52939.2500     S: 6.3377     ES: 2.5195     ER: 24.6896
```

Here, we see a series of blocks that exhibit initially a non-uniform distribution and then suddenly at block 0x33 we see a large spike in deviation relative to block at 0x32 followed again by a series of mostly uniform blocks. The Chi scores all converge in the 50000-59000 range and there is about a 1.0 shift in the Shannon score. Moreover if we again look at the Pi approximation scores, specifically the Error percentage, we see a large shift in the error rate and those values also converge. This sort of occurrence is precisely why this functionality hasn't been entirely removed yet, as it is not totally useless and in some cases is the metric of choice for identifying hidden data.

Continuing on with the blocks following 0x3A we find:

```
BN:  3A   C: 52939.2500    S: 6.3377  ES: 2.5195    ER: 24.6896
BN:  3B   C: 57933.0625    S: 6.2719  ES: 2.4258    ER: 29.5085
BN:  3C   C: 55347.5625    S: 6.3213  ES: 2.4531    ER: 28.0649
BN:  3D   C: 59412.5625    S: 6.2294  ES: 2.4336    ER: 29.0927
BN:  3E   C: 294981.5000   S: 5.1146  ES: 1.7617    ER: 78.3254
BN:  3F   C: 65713.5625    S: 6.2969  ES: 2.9336    ER: 7.0902
BN:  40   C: 51774.5000    S: 6.3365  ES: 2.5469    ER: 23.3509
BN:  41   C: 63068.0000    S: 6.3222  ES: 2.5859    ER: 21.4876
BN:  42   C: 57024.0625    S: 6.3928  ES: 2.7109    ER: 15.8858
BN:  43   C: 55179.7500    S: 6.3580  ES: 2.5117    ER: 25.0774
BN:  44   C: 50985.3125    S: 6.4162  ES: 2.6680    ER: 17.7522
BN:  45   C: 56001.9375    S: 6.3606  ES: 2.4414    ER: 28.6796
BN:  46   C: 55784.5000    S: 6.3404  ES: 2.5938    ER: 21.1216
BN:  47   C: 95045.7500    S: 6.2751  ES: 1.9727    ER: 59.2570
BN:  48   C: 350111.1875   S: 4.8677  ES: 0.1211    ER: 2494.3475
BN:  49   C: 1591737.3125  S: 1.3023  ES: 0.0625    ER: 4926.5482
BN:  4A   C: 2088960.0000  S: 0.0000  ES: 0.0000    ER:   inf
BN:  4B   C: 2088960.0000  S: 0.0000  ES: 0.0000    ER:   inf
BN:  4C   C: 2088960.0000  S: 0.0000  ES: 0.0000    ER:   inf
BN:  4D   C: 2088960.0000  S: 0.0000  ES: 0.0000    ER:   inf
BN:  4E   C: 2088960.0000  S: 0.0000  ES: 0.0000    ER:   inf
BN:  4F   C: 2088960.0000  S: 0.0000  ES: 0.0000    ER:   inf
BN:  50   C: 2088960.0000  S: 0.0000  ES: 0.0000    ER:   inf
BN:  51   C: 2088448.0625  S: 0.0018  ES: 0.0000    ER:   inf
BN:  52   C: 2088960.0000  S: 0.0000  ES: 0.0000    ER:   inf
BN:  53   C: 2088960.0000  S: 0.0000  ES: 0.0000    ER:   inf
BN:  54   C: 2088960.0000  S: 0.0000  ES: 0.0000    ER:   inf
BN:  55   C: 2088960.0000  S: 0.0000  ES: 0.0000    ER:   inf
BN:  56   C: 2088960.0000  S: 0.0000  ES: 0.0000    ER:   inf
BN:  57   C: 626987.8750   S: 4.2029  ES: 0.0117    ER: 26708.2573
BN:  58   C: 207007.7500   S: 5.4928  ES: 0.0000    ER:   inf
BN:  59   C: 370263.8125   S: 4.4937  ES: 1.1367    ER: 176.3738
```

Here we see that the pattern holds through block 0x3D, then deviates significantly, then converges back on the same generally pattern at about block 0x40. This is another good sign that we're looking at the embedded data, but not necessarily authoritatively so. However, as soon as we get to block 0x4A we encounter an almost entirely uniform series of blocks running through block 0x56. The Shannon and Pi scores are 0 The Chi distribution scores sky-rocket and so on. So what is going on there? This is a decent sign that we have encountered a series of blocks that are entirely or almost entirely one value, for instance a block that is entirely filled with zero's—or another value in the case of a one-byte XOR key. This coupled with the pattern prior to these blocks should leave us pretty confident that we've identified the sequence of blocks that contain the embedded executable, let's continue through the file and see if we can spot when exactly the pattern ends.

```
BN:  56        C: 2088960.0000   S: 0.0000     ES: 0.0000     ER:   inf
BN:  57        C: 626987.8750    S: 4.2029     ES: 0.0117     ER: 26708.2573
BN:  58        C: 207007.7500    S: 5.4928     ES: 0.0000     ER:   inf
BN:  59        C: 370263.8125    S: 4.4937     ES: 1.1367     ER: 176.3738
BN:  5A        C: 584377.4375    S: 3.8567     ES: 1.2031     ER: 161.1194
BN:  5B        C: 556420.9375    S: 4.5722     ES: 1.1367     ER: 176.3738
BN:  5C        C: 415127.0000    S: 4.8768     ES: 1.2227     ER: 156.9482
BN:  5D        C: 515672.6250    S: 3.5989     ES: 0.9062     ER: 246.6585
BN:  5E        C: 194348.7500    S: 6.2318     ES: 2.2930     ER: 37.0098
BN:  5F        C: 12978.0000     S: 7.4986     ES: 3.3828     ER: 7.1307
BN:  60        C: 2241.5000      S: 7.8798     ES: 3.1562     ER: 0.4644
BN:  61        C: 33056.5000     S: 7.1508     ES: 3.4727     ER: 9.5334
BN:  62        C: 617.3125       S: 7.9482     ES: 3.0703     ER: 2.3216
BN:  63        C: 549.1250       S: 7.9539     ES: 3.1406     ER: 0.0308
BN:  64        C: 877.3125       S: 7.9322     ES: 3.1133     ER: 0.9094
BN:  65        C: 7188.7500      S: 7.6370     ES: 3.3125     ER: 5.1595
BN:  66        C: 17839.1250     S: 7.2973     ES: 3.4453     ER: 8.8155
BN:  67        C: 580.8125       S: 7.9517     ES: 3.2734     ER: 4.0277
BN:  68        C: 536.2500       S: 7.9555     ES: 3.1602     ER: 0.5874
```

Sure enough, when we get to block 0x5D there is another marked spike in all of the values and then we go back to a fairly non-uniform distribution. If we look at the first block range we identified, we will notice that the shift between the distributions was far more gradual and gradient like—it increased gradually until it was uniform and then decreased gradually until it was no longer so. Here, in the blocks 0x32 through 0x5D, the shifts in values are far more sudden and dramatic. That's not even considering the blocks we encountered that had precisely uniform distributions across the board—a tell-tale sign that we've encountered a long string of exactly the same value, which is not something we'd generally expect to find in a PDF document; it generally will not have several kilobytes of zero's or similar; executable files however often exhibit that trait. Going back to the data, we see that blocks 0x5D and 0x5E deviate significantly from all of the other blocks, the Shannon score in 0x5D drops about a full point and then almost doubles in 0x5E. This again is going to be indicative of multiple data types colliding into the same block. Thus, we can add blocks 0x32 or 0x33 through blocks 0x5D or 0x5E to our list for further analysis. We can be pretty sure that we've found it given the values of the blocks 0x4A through 0x56, but that wouldn't make a very good tutorial if we stopped there.

Going through the rest of the file, the general pattern where each block shows a decent amount of deviation from those around it and no real uniformity mostly holds. At block 0xF0 through about 0xFC we have another string of blocks that show a fair amount of uniformity, however the shift between them and their neighbors is more gradual. Finally, at the end of the file, the last two blocks are significantly different from anything else in the file. In PDFs, this is actually to be expected as the data contained at the end of a PDF *is* significantly different than most of the file—there is an cross-reference table that holds a list of objects is just a sequence of ASCII block offsets followed by revision counts, which will generally speaking cause the contents to deviate significantly from the rest of the file.

At any rate, let's take a look at the actual frequency of the values stored in the blocks we identified as suspect and see if we can find a pattern. For now, we will skip over the functionality that calculates the deviation of blocks relative to one another and save that for a file that is a little harder for us to analyze.

We tell the tool this is what we want with the following options:

```
$ bin/edfind.py --frequency --freqcount 3 BlackHat-Japan-08-Dang-Office-Attacks-ONE.pdf | less
```

Specifically, the `–frequency` option indicates we want to retrieve the raw frequency of values in each block and the `–freqcount` indicates that we only want to view the top 4 values for each block. If we do not specify this, we will be bombarded with data showing the frequency count for every value between zero and 256 in every block of data in the file. When executed we're presented with the following output:

```
BYTE FREQUENCY FOR BLOCKS [  0  : 11D ]
  BN:   0    [V: 01 C:  121 P:    4.24]    [V: 30 C:  115 P:    3.30]    [V: C0 C:  10C P:    7.35]
  BN:   1    [V: 00 C:   5F P:   45.16]    [V: 01 C:   3C P:    8.70]    [V: 80 C:   37 P:    9.52]
  BN:   2    [V: 00 C:   7E P:   62.50]    [V: 0F C:   42 P:    7.87]    [V: 1E C:   3D P:    1.65]
  BN:   3    [V: 00 C:   6D P:   56.47]    [V: 01 C:   3D P:    1.65]    [V: 03 C:   3C P:    3.39]
  BN:   4    [V: 80 C:   BC P:    8.31]    [V: 02 C:   AD P:    1.75]    [V: 08 C:   AA P:    1.78]
  BN:   5    [V: 00 C:   64 P:   32.56]    [V: 01 C:   48 P:   13.33]    [V: 1E C:   3F P:   13.56]
  BN:   6    [V: 00 C:   52 P:   27.78]    [V: 01 C:   3E P:    3.28]    [V: 40 C:   3C P:    0.00]
  BN:   7    [V: 00 C:   82 P:   60.00]    [V: 01 C:   46 P:    8.96]    [V: 03 C:   40 P:    3.17]
  BN:   8    [V: 00 C:   53 P:   42.34]    [V: 40 C:   36 P:    1.87]    [V: AA C:   35 P:    0.00]
  BN:   9    [V: 01 C:   BD P:    9.42]    [V: 04 C:   AC P:    4.76]    [V: 47 C:   A4 P:    1.23]
  BN:   A    [V: 00 C:   63 P:   14.05]    [V: 80 C:   56 P:   34.01]    [V: 01 C:   3D P:    3.33]
  BN:   B    [V: 00 C:   68 P:   20.11]    [V: 80 C:   55 P:   13.84]    [V: 01 C:   4A P:    6.99]
  BN:   C    [V: 40 C:   7A P:   24.88]    [V: 1D C:   5F P:    0.00]    [V: 07 C:   5F P:    1.06]
  BN:   D    [V: 3A C:   74 P:    8.07]    [V: E2 C:   6B P:    1.89]    [V: E9 C:   69 P:    6.90]
  BN:   E    [V: 5C C:   73 P:    3.54]    [V: E2 C:   6F P:    3.67]    [V: 55 C:   6B P:    2.84]
  BN:   F    [V: 5C C:   70 P:    1.80]    [V: AE C:   6E P:    9.52]    [V: E2 C:   64 P:    3.05]
```

```
BN:  10      [V: AF C:   78 P:    9.61]       [V: D7 C:   6D P:   12.68]        [V: 5C C:   80 P:    6.45]
```

Here, going through first how the output is formatted are the first 16 blocks of the file. Again, the first field is the block number, then we're given three sets of fields delimited by square brackets. This format was chosen due to the fact that without the brackets it was often hard to visually identify what section of data we were looking at. At any rate, within the brackets we have a value denoted by V, one denoted by C and finally one denoted by P. This refer to the 'value', 'count' and 'percentage' respectively. The value is the actual byte value in question—if the byte in the file is 0x41, then this field would be 0x41. The count is the number of occurrences this value had within the given block and the percentage is how much more frequently this value occurred than the next highest value. For instance, in the first block the values with the highest frequencies were 0x01, 0x30 and 0xC0. 0x01 occurred 4.24% more often than 0x30, which occurred 3.30% more often than 0xC0, which in itself occurred 7.35% more frequently than whatever was the 4th most common value.

Now that we understand the format, let's take a look at the actual data. The specific rationale of this functionality is that a byte with a value of Z that is exclusive-or'd with a value of zero will result in a value of Z. Taking advantage of this, we can specifically look for places where we would expect to encounter a lot of zero's and actually recover the plain-text key. So, looking at the first blocks, we can immediately discount blocks 0x01 through 0x08 and blocks 0x0A and 0x0B from being XOR encrypted. The value most commonly encountered there is zero and any value exclusive or'd with zero is the value itself. Unless the key was 0x00, then these blocks could not be encrypted and if it was encrypted with a key whose value was zero, then the data was not encrypted but rather embedded into the document plain-text.

Arriving at blocks 0x0D through 0x10, which are part of a sequence we decided earlier to analyze a bit more in-depth, we see that the frequency doesn't match what we would expect with a single byte XOR key. However, in a couple of the blocks we find that there are some values that occur in the top 3 repeatedly, and as we will see later this is often indicative of a multi-byte XOR key where the values of the key shift across the top frequency due again to the differences between where the data itself starts and the segmenting of the file into blocks. That said, the values in question—0x5C and 0xE2 only really occur in even distributions with values that do not repeat in other blocks. As such, we can probably discount them as being XOR keys, although we would generally want to look at a wider set of distributions both in terms of the values in each block (say 5-8 or so to start) and more blocks in the series to see if a pattern with those values emerges. As the blocks 0x4A through 0x56 exhibited extremely abnormal behavior, let's skip ahead to block 0x32 and look at the frequency through at least 0x4A or 0x4B.

```
BN:  32        [V: 0B C:   4B P:    9.79]     [V: 2C C:   44 P:    9.23]    [V: 02 C:   3E P:    4.96]
BN:  33        [V: F9 C:  5B9 P:   96.86]     [V: B1 C:  1FD P:   22.27]    [V: 69 C:  197 P:   31.58]
BN:  34        [V: F9 C:  352 P:   50.18]     [V: B1 C:  1FD P:   14.77]    [V: 06 C:  1B7 P:   33.20]
BN:  35        [V: F9 C:  3AA P:   53.00]     [V: B1 C:  221 P:   19.98]    [V: 69 C:  1BE P:    6.00]
BN:  36        [V: F9 C:  35E P:   42.99]     [V: B1 C:  22D P:   36.77]    [V: 06 C:  180 P:    6.45]
BN:  37        [V: F9 C:  3A6 P:   48.07]     [V: 06 C:  23C P:    4.65]    [V: B1 C:  222 P:   44.57]
BN:  38        [V: F9 C:  406 P:   65.29]     [V: B1 C:  20B P:   21.61]    [V: 06 C:  1A5 P:   11.56]
```

```
  BN:  39              [V: F9 C:  2D9 P:   11.61]      [V: B1 C:  289 P:   34.08]    [V: 06 C:  1CC P:    3.77]
  BN:  3A              [V: F9 C:  35A P:   39.50]      [V: B1 C:  23F P:   22.87]    [V: 06 C:  1C9 P:   22.38]
  BN:  3B              [V: F9 C:  3DD P:   61.12]      [V: B1 C:  20E P:   18.71]    [V: 06 C:  1B4 P:   20.48]
  BN:  3C              [V: F9 C:  3B6 P:   53.16]      [V: B1 C:  227 P:   26.05]    [V: 72 C:  1A8 P:   26.44]
  BN:  3D              [V: F9 C:  3AB P:   40.15]      [V: B1 C:  271 P:   26.24]    [V: 72 C:  1E0 P:   27.49]
  BN:  3E              [V: F9 C:  BB8 P:  132.59]      [V: 69 C:  260 P:   84.44]    [V: F8 C:   F7 P:    6.26]
  BN:  3F              [V: F9 C:  392 P:    1.99]      [V: 06 C:  380 P:   71.31]    [V: 10 C:  1A9 P:   31.29]
  BN:  40              [V: F9 C:  2E7 P:   12.13]      [V: 06 C:  292 P:   39.49]    [V: B1 C:  1B9 P:    7.53]
  BN:  41              [V: F9 C:  3FC P:   28.70]      [V: 06 C:  2FC P:   67.60]    [V: B1 C:  17A P:   27.03]
  BN:  42              [V: F9 C:  3C9 P:   41.34]      [V: 06 C:  27D P:   45.24]    [V: B1 C:  192 P:   14.69]
  BN:  43              [V: F9 C:  3AE P:   36.39]      [V: 06 C:  28C P:   62.11]    [V: B1 C:  157 P:   13.37]
  BN:  44              [V: F9 C:  366 P:   49.64]      [V: 06 C:  20C P:    2.12]    [V: B1 C:  201 P:   32.13]
  BN:  45              [V: F9 C:  3E4 P:   55.71]      [V: B1 C:  232 P:   39.66]    [V: 72 C:  178 P:    6.31]
  BN:  46              [V: F9 C:  39C P:   55.96]      [V: B1 C:  208 P:   12.46]    [V: 72 C:  1CB P:    4.00]
  BN:  47              [V: F9 C:  66B P:  128.93]      [V: B1 C:  163 P:   33.17]    [V: 69 C:   FE P:    4.02]
  BN:  48              [V: F9 C:  CED P:  156.96]      [V: F8 C:  18F P:   24.79]    [V: FB C:  137 P:   22.54]
  BN:  49              [V: F9 C: 1BF1 P:  193.45]      [V: F8 C:   77 P:   72.00]    [V: 89 C:   38 P:   15.38]
  BN:  4A              [V: F9 C: 2000 P:    0.00]      [V: FF C:    0 P:    0.00]    [V: FE C:    0 P:    0.00]
  BN:  4B              [V: F9 C: 2000 P:    0.00]      [V: FF C:    0 P:    0.00]    [V: FE C:    0 P:    0.00]
```

Here, as soon as we get to block 0x33, the distribution pegs at the value 0xF9. It generally occurs far significantly more often than any other value—often at least 50% more and as high at 193% more. Before we even get to blocks 0x4A and 0x4B, we can be pretty positive these blocks are XOR encrypted with the value 0xF9. The values 0xB1 and 0x06 occur fairly frequently as well, and so in a more general sense we might have encountered a multi-byte XOR key, although in those instances we would not expect one value to be so far above and beyond the top value relative to others. Then we arrive at blocks 0x4A and 0x4B and we can more or less confirm our suspicions—the value 0xF9 occurs 0x2000 (8192) times in a block of size 8192. This is as we expected, a tell-tale sign that we found the XOR key due to the trait that key XOR zero = key. Just to confirm that and demonstrate the XOR Table Search functionality, we are going to double check that, but we can be pretty sure we are done here. If we were an anti-virus or similar scanner that was processing a lot of attachments with the purpose of flagging files for further analysis or human intervention, we would definitely flag this one based off of the distribution scores we discussed previously and the almost certain frequency distribution that indicates to us that not only is part of the file XOR encrypted, but that those sections were XOR encrypted with a key value of 0xF9.

To run the XOR table search functionality, you pass either the `–xor` or `–xorall` flags to the utility, these are by far the slowest parts of the application, but it will confirm or deny our suspicions to some value approaching 100%.

```
$ bin/edfind.py --xorall BlackHat-Japan-08-Dang-Office-Attacks-
ONE.pdf
FILE: BlackHat-Japan-08-Dang-Office-Attacks-ONE.pdf BLOCK COUNT: 286
BLOCK SIZE: 8192
```

```
     XOR TABLE SEARCH ALL
          OFFSET: 6622E ( 33  ) KEY: F9
```

Here, we confirm our results—the pre-computed XOR table search has identified a PE executable header at offset 0x6622E that is encrypted with the key 0xF9. If we divide that offset by the block size, 8192, we confirm that indeed the embedded executable starts in block 0x33.

To review what we've covered and learned, let's reiterate out points:

- We review the distribution of blocks, both in terms of uniformity and deviation to attempt to identify changes of note in the underlying data.
- Blocks with Chi-Square values in the 200-300 range and Shannon entropy values at or near 7.97 are sure fire signs of high entropy data—when looking at XOR encrypted data, we are typically talking about data that was encrypted with a longer length key.
- To initially identify deviant data, we want to look for sudden spikes, whether they be towards higher or lower values that deviate from the blocks around it. We then want to see some level of uniformity in the distribution.
- The Pi approximation values are not entirely useless, in the case above the percentage of Error was nearly constant across the embedded data and significantly higher than any other sequential series of blocks in the file.
- We can use frequency analysis to help identify not only if data is XOR encrypted, but also we can often recover the key in this manner. The underlying theory takes advantage that XOR encryption leaks the key whenever it attempts to encrypt a value of zero.
- A file encrypted with a single byte XOR key exhibits the same distribution as the plain-text, which in some ways makes it harder to spot. However analyzing the frequency will generally make this data immediately stand-out if the other statistically anomaly metrics based on entropy analysis do not do so themselves.
- A sequence of blocks with uniformity are not for sure signs of hidden data, this occurs frequently and is natural—the spikes surrounding them however often do not.
- The first and last blocks of embedded data tend to get lost into the host file and are sometimes harder to identify due to their not being aligned to our block size.

3.1.2 Files with higher-entropy embedded data / One-Time Pad XOR

So now that we've taken a look at a file with an embedded executable with a lower amount of entropy, let's take a look at the other end of the spectrum and review the same file with the same embedded executable, but this time instead of being encrypted with a single byte key, let's review it with a longer key—one as long as the embedded file simulating a one-time pad. This will give us a better idea of what true encryption will look like and also give us a better understanding of what the entropy analytics are actually analyzing. We will skip over the parts we already have and take a look almost strictly at the embedded data. It's pretty easy to spot just by reviewing the distribution scores so we are not skipping over anything overly important. We again pass the –blockscore (or –s) option and look at the file:

```
BN:  E4        C: 22454.6875    S: 7.1427     ES: 3.4766     ER: 9.6351
BN:  E5        C: 2943.5625     S: 7.8130     ES: 3.0078     ER: 4.4478
BN:  E6        C: 19197.6875    S: 7.2480     ES: 3.4609     ER: 9.2271
BN:  E7        C: 23342.6250    S: 7.1397     ES: 3.5508     ER: 11.5239
BN:  E8        C: 32748.1875    S: 6.9164     ES: 3.6055     ER: 12.8659
BN:  E9        C: 2094.6250     S: 7.8732     ES: 3.2383     ER: 2.9858
BN:  EA        C: 335.0000      S: 7.9705     ES: 3.0938     ER: 1.5464
BN:  EB        C: 294.3750      S: 7.9739     ES: 3.1914     ER: 1.5609
BN:  EC        C: 238.1875      S: 7.9790     ES: 3.2344     ER: 2.8686
BN:  ED        C: 284.3125      S: 7.9745     ES: 3.1289     ER: 0.4055
BN:  EE        C: 281.9375      S: 7.9752     ES: 3.1172     ER: 0.7829
BN:  EF        C: 261.0625      S: 7.9766     ES: 3.1445     ER: 0.0935
BN:  F0        C: 253.3125      S: 7.9777     ES: 3.0820     ER: 1.9325
BN:  F1        C: 228.4375      S: 7.9798     ES: 3.1602     ER: 0.5874
BN:  F2        C: 257.6875      S: 7.9775     ES: 3.2773     ER: 4.1421
BN:  F3        C: 281.8125      S: 7.9750     ES: 3.0859     ER: 1.8035
BN:  F4        C: 287.5625      S: 7.9745     ES: 3.1250     ER: 0.5310
BN:  F5        C: 206.7500      S: 7.9815     ES: 3.1328     ER: 0.2803
BN:  F6        C: 285.8125      S: 7.9748     ES: 3.0977     ER: 1.4184
BN:  F7        C: 255.7500      S: 7.9772     ES: 3.2227     ER: 2.5154
BN:  F8        C: 264.2500      S: 7.9766     ES: 3.2227     ER: 2.5154
BN:  F9        C: 248.8125      S: 7.9778     ES: 3.2188     ER: 2.3971
BN:  FA        C: 235.0625      S: 7.9794     ES: 3.2070     ER: 2.0405
BN:  FB        C: 217.9375      S: 7.9810     ES: 3.1445     ER: 0.0935
BN:  FC        C: 270.0625      S: 7.9760     ES: 3.0781     ER: 2.0619
BN:  FD        C: 279.8750      S: 7.9751     ES: 3.1016     ER: 1.2906
BN:  FE        C: 276.9375      S: 7.9756     ES: 3.1016     ER: 1.2906
BN:  FF        C: 254.5625      S: 7.9773     ES: 3.1836     ER: 1.3193
BN:  100       C: 263.2500      S: 7.9763     ES: 3.1133     ER: 0.9094
BN:  101       C: 256.5000      S: 7.9771     ES: 3.1133     ER: 0.9094
BN:  102       C: 250.1875      S: 7.9776     ES: 3.0820     ER: 1.9325
```

So here, with blocks 0xE4 through 0xE9, we see a decent amount of deviation per block with the natural flow of the data starting to decrease in entropy as we get closer to 0xE8, the increases slightly and when we reach 0xEA the entropy spikes and the Chi score for each block starts to become uniform at around 200-300 or so and the Shannon score becomes fairly uniform at about 7.97 or so. The Pi approximation values all begin to approach a much closer approximation of Pi as well. As I said, it is actually much easier to spot this sort of XOR encryption than ones with lower length keys, or at least is easier to spot just by looking at the scores. If we look at the actual frequency distribution alone, it becomes less of an easy task which is precisely the inverse of what we saw when dealing with a single byte length key.

```
BN:  E7    [V: 20 C: 1C5 P: 36.55]    [V: 01 C: 139 P:  4.91]    [V: 80 C: 12A P:  4.46]    [V: 10 C: 11D P:3.57]
BN:  E8    [V: 20 C: 265 P: 61.41]    [V: 40 C: 145 P:  2.18]    [V: 10 C: 13E P:  0.31]    [V: 04 C: 13D P:0.32]
BN:  E9    [V: 20 C: BE P: 52.49]     [V: 65 C: 6F P: 25.38]     [V: 74 C: 56 P:  2.35]     [V: 0A C: 54 P:  2.41]
BN:  EA    [V: EA C:  36 P:  7.69]    [V: E0 C: 32 P:  4.08]     [V: 01 C: 30 P:  2.11]     [V: EC C: 2F P:  0.00]
BN:  EB    [V: 96 C: 34 P:  8.00]     [V: 26 C: 30 P:  2.11]     [V: D3 C: 2F P:  0.00]     [V: 22 C: 2F P:  2.15]
BN:  EC    [V: 24 C: 2E P:  2.20]     [V: 1E C: 2D P:  2.25]     [V: E3 C: 2C P:  0.00]     [V: 33 C: 2C P:  2.30]
BN:  ED    [V: E8 C: 2F P:  0.00]     [V: 61 C: 2F P:  4.35]     [V: 5B C: 2D P:  0.00]     [V: 50 C: 2D P:  0.00]
BN:  EE    [V: 1C C: 35 P:  1.90]     [V: 98 C: 34 P:  8.00]     [V: 16 C: 30 P:  2.11]     [V: D8 C: 2F P:  4.35]
BN:  EF    [V: CD C: 32 P:  8.33]     [V: 38 C: 2E P:  2.20]     [V: DB C: 2D P:  0.00]     [V: 65 C: 2D P:  2.25]
BN:  F0    [V: C4 C: 34 P: 10.10]     [V: B8 C: 2F P:  2.15]     [V: 55 C: 2E P:  4.44]     [V: D6 C: 2C P:  0.00]
BN:  F1    [V: BA C: 35 P: 18.56]     [V: FB C: 2C P:  0.00]     [V: 31 C: 2C P:  0.00]     [V: 0E C: 2C P:  2.30]
BN:  F2    [V: 4C C: 2E P:  2.20]     [V: CE C: 2D P:  0.00]     [V: 61 C: 2D P: 2.25]      [V: A7 C: 2C P:  0.00]
BN:  F3    [V: 57 C: 31 P:  4.17]     [V: 4B C: 2F P: 2.15]      [V: 96 C: 2E P:  2.20]     [V: C1 C: 2D P:  0.00]
BN:  F4    [V: 01 C: 30 P:  2.11]     [V: 49 C: 2F P:  2.15]     [V: A3 C: 2E P:  2.20]     [V: A9 C: 2D P: 0.00]
```

Here we see in the first few blocks that the distribution doesn't really have any uniformity or spikes in the occurrence of any character that is out of the usual. The value 0x20 occurs a lot, but that is to be expected in a PDF as 0x20 correlates to an ASCII space character and there will be a lot of spaces in the PDF, which has a plain-text ASCII format. Then we reach the sections that contain our cipher text and the distribution evens out mostly. There is no single occurrence of any value that dominates, but at least in the top four values which a few significant outliers, we see that the distribution evens out between the variables for the most part—each value is between 0.00% to 2.5% or so as common as the next.

This is specifically what we are measuring with our Chi-Square distribution test that the distribution of byte values evens out and each value has a frequency about the same as every other value. To demonstrate this more clearly, let's look at the entire range of values for a given block that is encrypted with the one-time pad.

```
BYTE FREQUENCY FOR BLOCKS [ EC  : ED  ]
  BN:EC    [V: 24 C: 2E P: 2.20]    [V: 1E C: 2D P:  2.25]    [V: E3 C: 2C P:  0.00]    [V: 33 C: 2C P:  2.30]
  BN:EC    [V: B4 C: 2B P: 2.35]    [V: CA C: 2A P:  0.00]    [V: 59 C: 2A P:  0.00]    [V: 58 C: 2A P:  0.00]
  BN:EC    [V: 1B C: 2A P: 2.41]    [V: FF C: 29 P:  0.00]    [V: E7 C: 29 P:  0.00]    [V: 89 C: 29 P:  0.00]
  BN:EC    [V: E1 C: 28 P: 0.00]    [V: DF C: 28 P:  0.00]    [V: D0 C: 28 P:  0.00]    [V: CB C: 28 P:  0.00]
  BN:EC    [V: 73 C: 28 P: 0.00]    [V: 5A C: 28 P:  0.00]    [V: 48 C: 28 P:  0.00]    [V: 28 C: 28 P:  0.00]
  BN:EC    [V: 0C C: 28 P:0.00]     [V: 0B C: 28 P:  2.53]    [V: FD C: 27 P:  0.00]    [V: F1 C: 27 P:  0.00]
  BN:EC    [V: C4 C: 27 P:0.00]     [V: 8D C: 27 P:  0.00]    [V: 87 C: 27 P:  0.00]    [V: 81 C: 27 P:  0.00]
  BN:EC    [V: 52 C: 27 P: 0.00]    [V: 47 C: 27 P:  0.00]    [V: 32 C: 27 P:  0.00]    [V: 09 C: 27 P:  0.00]
  BN:EC    [V: BA C: 26 P: 0.00]    [V: 88 C: 26 P:  0.00]    [V: 57 C: 26 P:  0.00]    [V: 54 C: 26 P:  0.00]
  BN:EC    [V: DC C: 25 P: 0.00]    [V: DA C: 25 P: 0.00]     [V: BB C: 25 P:  0.00]    [V: A0 C: 25 P:  0.00]
  BN:EC    [V: 7E C: 25 P: 0.00]    [V: 49 C: 25 P:  0.00]    [V: 40 C: 25 P:  0.00]    [V: 39 C: 25 P:  0.00]
  BN:EC    [V: 0A C: 25 P: 0.00]    [V: 07 C: 25 P:  0.00]    [V: 00 C: 25 P:  2.74]    [V: DE C: 24 P:  0.00]
  BN:EC    [V: A4 C: 24 P: 0.00]    [V: 7D C: 24 P:  0.00]    [V: 78 C: 24 P:  0.00]    [V: 3C C: 24 P:  0.00]
  BN:EC    [V: 15 C: 24 P:2.82]     [V: FE C: 23 P:  0.00]    [V: F5 C: 23 P:  0.00]    [V: E4 C: 23 P:  0.00]
  BN:EC    [V: D6 C: 23 P:0.00]     [V: CC C: 23 P:  0.00]    [V: BF C: 23 P:  0.00]    [V: AD C: 23 P:  0.00]
  BN:EC    [V: A3 C: 23 P: 0.00]    [V: 8E C: 23 P:  0.00]    [V: 70 C: 23 P:  0.00]    [V: 64 C: 23 P:  0.00]
  BN:EC    [V: 37 C: 23 P:0.00]     [V: 30 C: 23 P:  0.00]    [V: 17 C: 23 P:  0.00]    [V: 14 C: 23 P:  0.00]
  BN:EC    [V: 05 C: 23 P: 2.90]    [V: F2 C: 22 P:  0.00]    [V: DD C: 22 P:  0.00]    [V: CD C: 22 P:  0.00]
  BN:EC    [V: BC C: 22 P:0.00]     [V: B6 C: 22 P:  0.00]    [V: B1 C: 22 P:  0.00]    [V: 76 C: 22 P:  0.00]
  BN:EC    [V: 55 C: 22 P: 0.00]    [V: 38 C: 22 P:  0.00]    [V: 2D C: 22 P:  2.99]    [V: FB C: 21 P:  0.00]
  BN:EC    [V: E2 C: 21 P: 0.00]    [V: C0 C: 21 P:  0.00]    [V: BD C: 21 P:  0.00]    [V: A7 C: 21 P:  0.00]
  BN:EC    [V: 9C C: 21 P: 0.00]    [V: 7A C: 21 P:  0.00]    [V: 79 C: 21 P:  0.00]    [V: 6D C: 21 P:  0.00]
  BN:EC    [V: 61 C: 21 P: 0.00]    [V: 56 C: 21 P:  0.00]    [V: 3B C: 21 P:  3.08]    [V: F3 C: 20 P:  0.00]
  BN:EC    [V: D8 C: 20 P: 0.00]    [V: D3 C: 20 P:  0.00]    [V: C2 C: 20 P:  0.00]    [V: C1 C: 20 P:  0.00]
  BN:EC    [V: 94 C: 20 P: 0.00]    [V: 8C C: 20 P:  0.00]    [V: 7B C: 20 P:  0.00]    [V: 72 C: 20 P:  0.00]
  BN:EC    [V: 5B C: 20 P: 0.00]    [V: 53 C: 20 P:  0.00]    [V: 2B C: 20 P:  0.00]    [V: 26 C: 20 P:  0.00]
  BN:EC    [V: 1A C: 20 P: 0.00]    [V: 06 C: 20 P:  3.17]    [V: FC C: 1F P:  0.00]    [V: F8 C: 1F P:  0.00]
  BN:EC    [V: E6 C: 1F P: 0.00]    [V: D5 C: 1F P:  0.00]    [V: B9 C: 1F P:  0.00]    [V: AE C: 1F P:  0.00]
  BN:EC    [V: 9F C: 1F P: 0.00]    [V: 9E C: 1F P:  0.00]    [V: 96 C: 1F P:  0.00]    [V: 8A C: 1F P:  0.00]
  BN:EC    [V: 68 C: 1F P: 0.00]    [V: 4A C: 1F P:  0.00]    [V: 3E C: 1F P:  0.00]    [V: 25 C: 1F P:  0.00]
  BN:EC    [V: EB C: 1E P: 0.00]    [V: AF C: 1E P:  0.00]    [V: AC C: 1E P:  0.00]    [V: A8 C: 1E P:  0.00]
  BN:EC    [V: 8F C: 1E P: 0.00]    [V: 84 C: 1E P:  0.00]    [V: 7F C: 1E P:  0.00]    [V: 6A C: 1E P:  0.00]
  BN:EC    [V: 5D C: 1E P: 0.00]    [V: 3A C: 1E P:  0.00]    [V: 20 C: 1E P:  0.00]    [V: 1F C: 1E P:  0.00]
  BN:EC    [V: 18 C: 1E P: 0.00]    [V: 03 C: 1E P:  3.39]    [V: E8 C: 1D P:  0.00]    [V: D1 C: 1D P:  0.00]
  BN:EC    [V: BE C: 1D P: 0.00]    [V: B0 C: 1D P:  0.00]    [V: AB C: 1D P:  0.00]    [V: 9B C: 1D P:  0.00]
  BN:EC    [V: 95 C: 1D P: 0.00]    [V: 80 C: 1D P:  0.00]    [V: 77 C: 1D P:  0.00]    [V: 74 C: 1D P:  0.00]
  BN:EC    [V: 66 C: 1D P: 0.00]    [V: 65 C: 1D P:  0.00]    [V: 50 C: 1D P:  0.00]    [V: 31 C: 1D P:  0.00]
  BN:EC    [V: 29 C: 1D P: 0.00]    [V: 04 C: 1D P:  3.51]    [V: F7 C: 1C P:  0.00]    [V: EA C: 1C P:  0.00]
  BN:EC    [V: 62 C: 1C P: 0.00]    [V: 5E C: 1C P:  0.00]    [V: 4F C: 1C P:  0.00]    [V: 45 C: 1C P:  0.00]
  BN:EC    [V: FA C: 1B P: 0.00]    [V: CF C: 1B P:  0.00]    [V: C9 C: 1B P:  0.00]    [V: B3 C: 1B P:  0.00]
  BN:EC    [V: 93 C: 1B P: 0.00]    [V: 90 C: 1B P:  0.00]    [V: 8B C: 1B P:  0.00]    [V: 67 C: 1B P:  0.00]
  BN:EC    [V: 43 C: 1B P: 0.00]    [V: 34 C: 1B P:  0.00]    [V: 2E C: 1B P:  0.00]    [V: 27 C: 1B P:  0.00]
  BN:EC    [V: 0D C: 1B P: 0.00]    [V: 08 C: 1B P:  3.77]    [V: ED C: 1A P:  0.00]    [V: 86 C: 1A P:  0.00]
  BN:EC    [V: 83 C: 1A P: 0.00]    [V: 60 C: 1A P:  0.00]    [V: 4B C: 1A P:  0.00]    [V: 3D C: 1A P:  0.00]
  BN:EC    [V: 10 C: 1A P: 3.92]    [V: F6 C: 19 P:  0.00]    [V: F0 C: 19 P:  0.00]    [V: D2 C: 19 P:  0.00]
  BN:EC    [V: C6 C: 19 P: 0.00]    [V: B8 C: 19 P:  0.00]    [V: B7 C: 19 P:  0.00]    [V: B5 C: 19 P:  0.00]
```

```
BN:EC    [V: 91 C: 19 P: 0.00]    [V: 7C C: 19 P:  0.00]    [V: 6C C: 19 P:  0.00]    [V: 5C C: 19 P:  0.00]
BN:EC    [V: 2C C: 19 P: 0.00]    [V: 22 C: 19 P:  0.00]    [V: 21 C: 19 P:  4.08]    [V: F9 C: 18 P:  0.00]
BN:EC    [V: D7 C: 18 P: 0.00]    [V: 98 C: 18 P:  0.00]    [V: 6B C: 18 P:  0.00]    [V: 4E C: 18 P:  0.00]
BN:EC    [V: 02 C: 18 P: 4.26]    [V: E0 C: 17 P:  0.00]    [V: 92 C: 17 P:  4.44]    [V: 3F C: 16 P:  0.00]
BN:EC    [V: E9 C: 15 P: 0.00]    [V: A5 C: 15 P:  0.00]    [V: 6E C: 15 P:  0.00]    [V: 4D C: 15 P:  4.88]
```

Here, we see the entire range of all possible 256 values within block 0xEC.As we look through the data, what we see is each value is more or less as probable as every other character—with a large volume of them exhibiting exactly the same volume as others. This is what we would expect from random data, as there will be either no or very little bias towards any specific value if it's random and each value has a probability of occurrence as every other value. Generally speaking, data that is encrypted will have a more even distribution—in the actual encryption algorithms like AES and similar, this is a desired quality in part to help prevent known-plaintext attacks. This is complicated by the fact that compressed files and encrypted files tend to look pretty similar to our analytics, for instance, consider the following distributions for file compressed with different algorithms:

```
ZIP file (.zip):
BN:30   [V: D7 C: 33 P: 0.00]    [V: 05 C: 33 P: 10.31]    [V: F2 C: 2E P: 2.20]    [V: 0D C: 2D P: 2.25]
BN:31   [V: 8F C: 34 P: 8.00]    [V: D3 C: 30 P: 0.00]     [V: B3 C: 30 P: 0.00]    [V: 14 C: 30 P: 2.11]
BN:32   [V: D5 C: 35 P: 5.83]    [V: FB C: 32 P: 2.02]     [V: E9 C: 31 P: 2.06]    [V: 13 C: 30 P: 2.11]
BN:33   [V: 67 C: 34 P: 1.94]    [V: DF C: 33 P: 1.98]     [V: 33 C: 32 P: 6.19]    [V: 6D C: 2F P: 2.15]
BN:34   [V: 75 C: 35 P: 1.90]    [V: F5 C: 34 P: 3.92]     [V: AD C: 32 P:0.00]     [V: 76 C: 32 P: 8.33]
BN:35   [V: 7E C: 31 P: 2.06]    [V: F4 C: 30 P: 4.26]     [V: FC C: 2E P: 0.00]    [V: 77 C: 2E P: 0.00]
BN:36   [V: F9 C: 30 P: 0.00]    [V: 99 C: 30 P: 2.11]     [V: FD C: 2F P: 0.00]    [V: B2 C: 2F P: 0.00]

BZip2 (.tar.bz2):
BN:30   [V: 7F C: 31 P: 6.32]      [V: A9 C: 2E P: 0.00]    [V: 7B C: 2E P: 2.20]    [V: 33 C: 2D P: 0.00]
BN:31   [V: 00 C: 3F P: 23.01]     [V: 4C C: 32 P: 2.02]    [V: 94 C: 31 P: 4.17]    [V: 3D C: 2F P: 2.15]
BN:32   [V: 00 C: 48 P: 32.26]     [V: 27 C: 34 P: 0.00]    [V: 02 C: 34 P: 1.94]    [V: 6F C: 33 P: 1.98]
BN:33   [V: 00 C: 37 P: 9.52]      [V: 6C C: 32 P: 6.19]    [V: E6 C: 2F P: 0.00]    [V: A7 C: 2F P: 2.15]
BN:34   [V: 00 C: 49 P: 35.48]     [V: 18 C: 33 P: 4.00]    [V: FB C: 31 P: 0.00]    [V: DB C: 31 P: 6.32]
BN:35   [V: 00 C: 35 P: 3.85]      [V: 08 C: 33 P: 4.00]    [V: BD C: 31 P: 8.51]    [V: 81 C: 2D P: 2.25]
BN:36   [V: C0 C: 31 P: 4.17]      [V: 29 C: 2F P: 2.15]    [V: 00 C: 2E P: 2.20]    [V: A0 C: 2D P: 0.00]

Gzip (.tar.gz):
BN:30   [V: BF C: 34 P: 14.43]    [V: D7 C: 2D P 0.00]     [V: AF C: 2D P: 2.25]    [V: ED C: 2C P: 0.00]
BN:31   [V: 7C C: 37 P: 17.82]    [V: 1C C: 2E P: 2.20]    [V: DF C: 2D P: 0.00]    [V: 4F C: 2D P: 2.25]
BN:32   [V: 3B C: 33 P: 1.98]     [V: A5 C: 32 P:4.08]     [V: 4B C: 30 P: 6.45]    [V: EE C: 2D P: 0.00]
BN:33   [V: DE C: 41 P: 26.09]    [V: CA C: 32 P:8.33]     [V: E9 C: 2E P: 2.20]    [V: AF C: 2D P: 0.00]
BN:34   [V: EA C: 34 P: 8.00]     [V: D0 C: 30 P:0.00]     [V: AD C:  30 P: 6.45]   [V: B5 C: 2D P: 0.00]
BN:35   [V: F5 C: 30 P: 0.00]     [V: 1E C: 30 P: 2.11]    [V: BE C: 2F P: 2.15]    [V: 53 C: 2E P: 2.20]
BN:36   [V: CB C: 2F P: 0.00]     [V: 8F C: 2F P: 2.15]    [V: 9B C: 2E P: 2.20]    [V: 81 C: 2D P: 4.55]

LZMA (.tar.xz):
```

```
BN:  30    [V: A0 C: 34 P:  8.00]    [V: F2 C: 30 P: 0.00]    [V: 3A C: 30 P: 6.45]    [V: 6C C: 2D P: 0.00]
BN:  31    [V: 3C C: 2F P:  0.00]    [V: 05 C: 2F P: 2.15]    [V: 8E C: 2E P: 2.20]    [V: 73 C: 2D P: 4.55]
BN:  32    [V: F5 C: 33 P:  1.98]    [V: 19 C: 32 P: 6.19]    [V: FC C: 2F P: 2.15]    [V: B6 C: 2E P: 2.20]
BN:  33    [V: C7 C: 32 P:  2.02]    [V: 26 C: 31 P: 4.17]    [V: 1D C: 2F P: 0.00]    [V: 03 C: 2F P: 2.15]
BN:  34    [V: 31 C: 31 P:  0.00]    [V: 19 C: 31 P: 2.06]    [V: 3C C: 30 P: 2.11]    [V: F5 C: 2F P: 2.15]
BN:  35    [V: 44 C: 30 P:  0.00]    [V: 2D C: 30 P: 4.26]    [V: FA C: 2E P: 0.00]    [V: C4 C: 2E P: 2.20]
BN:  36    [V: 0A C: 34 P: 10.10]    [V: AA C: 2F P: 4.35]    [V: A6 C: 2D P: 0.00]    [V: 6B C: 2D P: 2.25]

AES-256 Encrypted ZIP (.zip):
BN:30   [V: D1 C: 2F P: 0.00]   [V: 96 C: 2F P: 0.00]     [V: 42 C: 2F P: 0.00]     [V: 27 C:   2F P: 4.35]
BN:31   [V: 6F C: 31 P: 0.00]   [V: 6D C: 31 P:  8.51]    [V: DD C: 2D P:  .25]     [V: FF C:   2C P: 0.00]
BN:32   [V: C7 C: 30 P: 2.11]   [V: 62 C: 2F P:  2.15]    [V: 74 C: 2E P:  0.00]    [V: 15 C:   2E P: 2.20]
BN:33   [V: DC C: 37 P: 9.52]   [V: 6C C: 32 P:  8.33]    [V: 93 C: 2E P:  0.00]    [V: 2F C:   2E P: 0.00]
BN:34   [V: FF C:  30 P: 4.26]  [V: BE C: 2E P:  0.00]    [V: 59 C: 2E P:  0.00]    [V: 51 C:   2E P: 0.00]
BN:35   [V: 63 C: 32 P: 8.33]   [V: 4E C: 2E P:  2.20]    [V: FC C: 2D P:  2.25]    [V: E1 C:   2C P: 0.00]
BN:36   [V: 04 C: 32 P: 6.19]   [V: 8A C: 2F P:  6.59]    [V: A9 C: 2C P:  2.30]    [V: B5 C:   2B P: 0.00]
```

The blocks from the file were chosen more or less at random with the only real criteria being that they didn't come from the beginning or end of the file and that they more or less modeled the distribution and generally didn't have too many outliers relative to the rest of the file. As we can see, the distribution of the values looks a lot like the encrypted data with the only really significant difference being that this even distribution tends to be punctuated by higher deviations than with the cipher-text. When we get to the last file, which is a ZIP file that was subsequently also AES-256 encrypted with a password, these punctuations decrease quite a bit and the distribution evens out more.

As such, we obviously cannot use entropy analysis to do all of the work for identifying hidden data—there are just too many edge cases where it becomes implausible to discern whether deviant data we're looking at is normal or not. Thus, this is mostly a library that is intended to serve as a starting point for things like host intrusion prevention systems (HIDS), Anti-virus (AV), and so on—to help better identify what data should be given the more resource intensive validation checks such as attempting to parse the actual file structure and similar. In fact, using entropy analysis in a similar manner already is used extensively throughout the industry—just in a slightly different context. It's used fairly extensively to determine when an executable file is packed and to aid in the automated unpacking of the executable. So, by taking the same concepts and applying it to data that mixes multiple data streams with one embedded and hidden in another is not really that large of a leap.

So to reiterate our points in this section:

- Data encrypted with longer XOR keys have much more entropy. They're easier to spot as deviant from other data that itself is not high entropy, but harder to identify by the raw frequency counts alone.
- Unlike shorter XOR keys, we will not be able to utilize the 'value zero trick' in these circumstances because One-Time Pads utilize a key that is as long as the data being encrypted.

- If the entropy of the host file data is closer to the embedded data, we may not be able to overly discern which is which through these methods.
- Compressed data tends to look a lot like high-entropy encrypted data, just with intermittent spikes in the frequency that deviate relatively significantly from cipher-text.
- Entropy analysis and statistical analysis should not be thought of as a magic bullet and really should only improve the performance of existing more resource intensive methods by more properly targeting them towards suspect data.

3.1.3 The effect of different sizes of block on the entropy analysis scores

Earlier we said that as the data gets more random the Chi-Square distribution test scores approach the 200-300 range, the Shannon entropy score approaches 7.97 and the Pi approximation scores approach Pi, albeit only "kinda sorta". This isn't entirely true, all of these are dependent on the size of the block in question. As the blocks get bigger, how close the Pi approximation approaches Pi decreases the more random the data is. The Chi-Square score is modified, albeit not as dramatically as other scores and the Shannon score reaches higher values—for instance 7.99 or higher.

To demonstrate this, let's examine a block of data generated by a PRNG in different block sizes. We generated the data by pointing the `dd` program to `/dev/random` and as such the data should be as random as one can expect from the Linux PRNG. One significant caveat however is that this was done inside of a virtual machine. To help improve this, we seeded the Linux PRNG repeatedly with data from the hardware generators from www.random.org. At any rate, let's take a look at the file with the default block size:

```
FILE: random-34k.data BLOCK COUNT: 5 BLOCK SIZE: 8192

ALL SCORES
BN:  0        C: 266.3750     S: 7.9762     ES: 3.1836     ER: 1.3193
BN:  1        C: 250.0000     S: 7.9780     ES: 3.1562     ER: 0.4644
BN:  2        C: 235.6875     S: 7.9792     ES: 3.1992     ER: 1.8013
BN:  3        C: 236.9375     S: 7.9790     ES: 3.1367     ER: 0.1554
BN:  4        C: 249.1514     S: 7.8458     ES: 3.1258     ER: 0.5043
```

As we can see, at the default block size of 8192 bytes or 8KB, the scores fall into the range previously stated—the Chi score hovers around 250 and the Shannon score is mostly at 7.97. The Pi approximation scores approach Pi, but there is really no discernable pattern to them as the error ranges between 0.4 and 1.8. If we resize the blocks so that they're twice as large, or 16384 bytes/16KB, we get the following values:

```
FILE: random-34k.data BLOCK COUNT: 3 BLOCK SIZE: 16384

ALL SCORES
BN:   0        C: 238.1875     S: 7.9895     ES: 3.1699     ER: 0.8937
BN:   1        C: 236.1250     S: 7.9896     ES: 3.1680     ER: 0.8326
BN:   2        C: 249.1514     S: 7.8458     ES: 3.1258     ER: 0.5043
```

Here we can see that the Chi-Square scores stayed mostly the same but decreased slightly and the Shannon scores increased into the 7.98 range for the most part. The last block decreased slightly, but that conforms to what we saw in the default block size. The Monte Carlo method Pi approximation scores indeed got closer to the correct value of Pi, which is what we expect for random data that increases in size. So let's take a look at even larger blocks and see how it modifies the scores, this time let's look at blocks three times the default size, or 24KB:

```
FILE: random-34k.data BLOCK COUNT: 2 BLOCK SIZE: 24576

ALL SCORES
BN:   0        C: 231.2708     S: 7.9932     ES: 3.1797     ER: 1.1981
BN:   1        C: 230.2965     S: 7.9821     ES: 3.1353     ER: 0.2001
```

Here again, we same relative deviation is encountered, the Chi score decreases slightly and hovers around 230 and the Shannon scores increase into the 7.99/7.98 range. The Pi score however deviates from what we expected somewhat and in one block the error is larger and in the other it's smaller. This sort of less deterministic behavior is why we're probably going to replace the Pi approximation in a later release, it is useful, just not as much as we had hoped.

Okay, so we have seen above that as the block size increases, the same high-entropy data generates scores that change—with the Chi score decreasing, the Shannon score increasing and the Pi approximation scores further approaching Pi, albeit in a somewhat random fashion. So the obvious next question is what happens when the block size of high-entropy data decreases? Does the pattern still hold in that the Chi score will increase, the Shannon score decrease and the Pi approximation score start to deviate further? To determine that, let's take a look at the same file this time with smaller blocks. Let's first start out with a block half the size of the default, or 4096 bytes in length:

```
FILE: random-34k.data BLOCK COUNT: 9 BLOCK SIZE: 4096

ALL SCORES
BN:   0        C: 272.2500     S: 7.9518     ES: 3.1797     ER: 1.1981
BN:   1        C: 256.6250     S: 7.9536     ES: 3.1875     ER: 1.4402
BN:   2        C: 230.7500     S: 7.9592     ES: 3.1484     ER: 0.2174
```

```
BN:   3             C: 241.8750         S: 7.9566          ES: 3.1641          ER: 0.7102
BN:   4             C: 223.2500         S: 7.9595          ES: 3.2422          ER: 3.1027
BN:   5             C: 279.7500         S: 7.9505          ES: 3.1562          ER: 0.4644
BN:   6             C: 257.7500         S: 7.9533          ES: 3.1406          ER: 0.0308
BN:   7             C: 241.7500         S: 7.9565          ES: 3.1328          ER: 0.2803
BN:   8             C: 249.1514         S: 7.8458          ES: 3.1258          ER: 0.5043
```

As we review the data, we see that indeed the scores deviate mostly as expected. The Chi scores increase, but remain within the mid-200s range albeit slightly smaller than the scores retrieved by the default block size of 8192. The Shannon score decreases to mostly the 7.95 range, and the Pi approximation scores start to become more distanced from Pi, against, in a less deterministic fashion than hoped. If this pattern holds, then when we again decrease the size, we should see the Chi scores increase into even higher values, the Shannon decrease slightly and the Pi approximation scores deviate even further. As such, let's look at the file in 2KB blocks:

```
FILE: random-34k.data BLOCK COUNT: 17 BLOCK SIZE: 2048

ALL SCORES
BN:   0             C: 265.2500         S: 7.9035          ES: 3.2969          ER: 4.7100
BN:   1             C: 265.7500         S: 7.9060          ES: 3.0625          ER: 2.5826
BN:   2             C: 306.2500         S: 7.8924          ES: 3.1562          ER: 0.4644
BN:   3             C: 222.7500         S: 7.9181          ES: 3.2188          ER: 2.3971
BN:   4             C: 236.0000         S: 7.9158          ES: 3.2344          ER: 2.8686
BN:   5             C: 267.5000         S: 7.9044          ES: 3.0625          ER: 2.5826
BN:   6             C: 245.2500         S: 7.9119          ES: 3.1250          ER: 0.5310
BN:   7             C: 240.5000         S: 7.9124          ES: 3.2031          ER: 1.9210
BN:   8             C: 193.7500         S: 7.9301          ES: 3.2344          ER: 2.8686
BN:   9             C: 234.0000         S: 7.9142          ES: 3.2500          ER: 3.3356
BN:   A             C: 264.2500         S: 7.9070          ES: 3.0938          ER: 1.5464
BN:   B             C: 267.5000         S: 7.9035          ES: 3.2188          ER: 2.3971
BN:   C             C: 243.5000         S: 7.9117          ES: 3.0938          ER: 1.5464
BN:   D             C: 247.2500         S: 7.9073          ES: 3.1875          ER: 1.4402
BN:   E             C: 213.0000         S: 7.9218          ES: 3.1406          ER: 0.0308
BN:   F             C: 237.0000         S: 7.9133          ES: 3.1250          ER: 0.5310
BN:  10             C: 249.1514         S: 7.8458          ES: 3.1258          ER: 0.5043
```

Indeed, the pattern holds. The Chi scores mostly gravitate towards the mid-to-high end of the 200s with a single block reaching the low 300 range, the Shannon scores decrease into the 7.90-7.98 range and we begin to see even more deviation from the Pi scores. So, if we were to view the file as a whole, what we would expect is a Chi score that sits somewhere in the mid-200 range, a higher Shannon score and a Pi score that further approaches Pi:

```
FILE: random-34k.data BLOCK COUNT: 5 BLOCK SIZE: 8192

WHOLE FILE SCORE
C: 242.8282           S: 7.9948           ES: 3.1674           ER: 0.8152
```

And indeed, the pattern again holds. So, to review what we've discussed we can make the following observations:

- When dealing with high-entropy data such as random data or cipher-texts, the bigger the block the lower the Chi-Square scores are, although they average out somewhat. Furthermore, the Shannon score slightly increases and the Pi approximation scores become more accurate and approach Pi a bit more although it does so somewhat randomly.
- Inversely, when we decrease the block size, the scores show a similar inverse deviation.
- Pi approximation scores work better with larger blocks of data.

3.1.4 Block score deviation demonstrated via 8-byte XOR

Okay, so we have reviewed just the generic block scores so far and have done so at two ends of the spectrum: with a 1-byte XOR key and an XOR key that is as long as the embedded file. Both were pretty easy to spot albeit with different methods. The 1-byte XOR data more closely conformed to the host file sans the blocks full of zeroes and looking at the raw frequency data showed us pretty clearly that the data was XOR encrypted. When we looked at the longer XOR key, the data had a pretty uniform distribution and looking at the raw frequency data was less useful, but the standard scoring metrics showed us that such data is pretty easy to spot. Moreover, we haven't even discussed other aspects of the extension, in particular functionality that calculates differences between the scores for blocks and block ranges. So let's take a look briefly at the same PDF file with the same embedded executable, but this time let's use a longer XOR key, say 8 bytes. In this instance, we can again easily spot the blocks full of zeroes and the general pattern of a large deviation followed by relative uniformity holds as well, so we will skip straight to the relevant sections of the file:

```
BN:  AA         C: 2342.4375     S: 7.8711     ES: 3.1758      ER: 1.0765
BN:  AB         C: 33199.0000    S: 7.1964     ES: 3.4023      ER: 7.6639
BN:  AC         C: 603.3750      S: 7.9501     ES: 3.1289      ER: 0.4055
BN:  AD         C: 486.8125      S: 7.9587      ES: 3.1719     ER: 0.9547
BN:  AE         C: 23303.5000    S: 7.2774      ES: 3.4180     ER: 8.0860
BN:  AF         C: 6687.7500     S: 7.5337     ES: 3.3438      ER: 6.0458
BN:  B0         C: 6607.6250     S: 7.5587     ES: 3.3047      ER: 4.9353
```

```
BN:  B1        C: 6717.0625     S: 7.5527     ES: 3.3633     ER: 6.5914
BN:  B2        C: 6416.8125     S: 7.5628     ES: 3.3711     ER: 6.8079
BN:  B3        C: 6785.5000     S: 7.5451     ES: 3.4102     ER: 7.8754
BN:  B4        C: 6624.5000     S: 7.5521     ES: 3.4219     ER: 8.1909
BN:  B5        C: 6518.8125     S: 7.5538     ES: 3.4258     ER: 8.2956
BN:  B6        C: 6759.1875     S: 7.5448     ES: 3.4062     ER: 7.7698
BN:  B7        C: 6721.2500     S: 7.5509     ES: 3.4062     ER: 7.7698
BN:  B8        C: 7210.3750     S: 7.5255     ES: 3.3438     ER: 6.0458
BN:  B9        C: 14706.0000    S: 7.2823     ES: 3.4258     ER: 8.2956
BN:  BA        C: 24410.5000    S: 7.0843     ES: 3.6016     ER: 12.7714
BN:  BB        C: 6947.8750     S: 7.5286     ES: 3.3672     ER: 6.6998
BN:  BC        C: 6047.8750     S: 7.5769     ES: 3.4062     ER: 7.7698
BN:  BD        C: 7235.6250     S: 7.5357     ES: 3.3945      ER: 7.4514
BN:  BE        C: 6651.3125     S: 7.5610     ES: 3.4922     ER: 10.0394
BN:  BF        C: 6332.2500     S: 7.5693     ES: 3.4258     ER: 8.2956
BN:  C0        C: 5886.3125     S: 7.5881     ES: 3.3867     ER: 7.2379
BN:  C1        C: 6651.4375     S: 7.5643     ES: 3.3008     ER: 4.8228
BN:  C2        C: 5274.0000     S: 7.6159     ES: 3.3984     ER: 7.5577
BN:  C3        C: 32201.8750    S: 7.0112     ES: 3.5820     ER: 12.2958
BN:  C3        C: 32201.8750    S: 7.0112     ES: 3.5820     ER: 12.2958
BN:  C4        C: 81919.3125    S: 6.0033     ES: 3.5352     ER: 11.1328
BN:  C5        C: 253952.0000   S: 3.0000     ES: 4.0000     ER: 21.4602
BN:  C6        C: 253952.0000   S: 3.0000     ES: 4.0000     ER: 21.4602
BN:  C7        C: 253952.0000   S: 3.0000     ES: 4.0000     ER: 21.4602
BN:  C8        C: 253952.0000   S: 3.0000     ES: 4.0000     ER: 21.4602
BN:  C9        C: 253952.0000   S: 3.0000     ES: 4.0000     ER: 21.4602
```

For the same of demonstration, let's assume the blocks full of zero's don't exist. This is fairly appropriate given that malware in the wild is less likely to have that trait due to packing and encryption and so on. In order to "fix" the zero problem, the data would have to be encrypted with an XOR key that is at least as long as the entire run of zeroes or else we will encounter the same distributions for those sections. Given that, we can still see the fairly uniform distribution in the Shannon and Chi scores when we encounter the file. The Chi scores are mostly in the 6000-7000 range, which differs by a fair amount from neighbor blocks around it. However, the scores themselves begin to approach a value more consistent with the host data, for instance the range we identified as potentially suspect when we looked at the file with a one-byte XOR key has scores in the 4000 range. Sometimes however, especially when dealing with high-entropy host data with high-entropy embedded data, the differences are not so easy to spot by eye and the uniformity of the scores gets lost. In those instances, it's often easier to look at the same data in terms of percent of differences than the actual scores.

We have multiple modes of operation that handle this case, for instance the `–blockdev` mode calculates a given blocks percent of difference from all of the other blocks, `--devxy` is similar, except that it calculates the differences between two specific blocks and finally we have `–seqdev` and `–seqxy.` The first.will calculate the percentage of deviation between sequential blocks, which is to say we calculate the difference for block 1 from block 2, block 2 from block 3 and so on. The second option performs the exact same task, but instead of doing it against all blocks in the file it takes two parameters that denote the start and stop indices for a range of blocks we want to calculate the differences for.

Let’s start by taking a look at the deviation of regular blocks of host data so we can get a feel for that, we do so by passing the `–blockdev` option and a specified `--blocknumber`:

```
$ bin/edfind.py -d -n 5 BlackHat-Japan-08-Dang-Office-Attacks-EIGHT.pdf | less
```

Which produces, in part, the following:

```
FILE: BlackHat-Japan-08-Dang-Office-Attacks-EIGHT.pdf BLOCK COUNT: 286 BLOCK SIZE: 8192

BLOCK   5   DEVIATION RELATIVE ALL BLOCKS
BN:   0         C: 180.5784     S: 6.5931     ES: 9.3880      ER: 187.5330
BN:   1         C: 14.6888      S: 0.0601     ES: 0.3734      ER: 99.9762
BN:   2         C: 19.5273      S: 0.0870     ES: 2.9484      ER: 161.5248
BN:   3         C: 2.2043       S: 0.0052     ES: 2.2195      ER: 149.0724
BN:   4         C: 151.0541     S: 2.3622     ES: 1.2392      ER: 109.2246
BN:   5         C: 14.7972      S: 0.0720     ES: 5.1033      ER: 177.5351
BN:   6         C: 20.5931      S: 0.0792     ES: 1.7305      ER: 134.8410
BN:   7         C: 16.2717      S: 0.0451     ES: 5.5758      ER: 179.3913
BN:   8         C: 154.8200     S: 2.8190     ES: 0.7453      ER: 49.4529
BN:   9         C: 4.7020       S: 0.0111     ES: 3.3109      ER: 165.6756
BN:   A         C: 23.2693      S: 0.1359     ES: 4.7474      ER: 175.8930
BN:   B         C: 132.6418     S: 2.5889     ES: 5.1151      ER: 180.7530
BN:   C         C: 144.9958     S: 3.7745     ES: 6.9677      ER: 185.6275
BN:   D         C: 146.8961     S: 3.9277     ES: 8.1765      ER: 187.6935
BN:   E         C: 145.8226     S: 3.8883     ES: 8.1765      ER: 187.6935
BN:   F         C: 151.7696     S: 4.3970     ES: 18.6776     ER: 194.7169
BN:  10         C: 150.9746     S: 4.3200     ES: 11.1916     ER: 190.9946
BN:  11         C: 148.5987     S: 4.2271     ES: 11.8890     ER: 191.5305
BN:  12         C: 148.8078     S: 4.0953     ES: 16.7568     ER: 194.0688
BN:  13         C: 147.7051     S: 4.0217     ES: 10.9139     ER: 190.7631
BN:  14         C: 151.3041     S: 4.2417     ES: 11.0526     ER: 190.8801
BN:  15         C: 147.3353     S: 3.9494     ES: 10.7753     ER: 190.6433
```

We chose block number 5 somewhat at random with really only a criteria that it wasn’t at the beginning or end of the file. As we look at the data, we see the same sort of spikes and valleys that the scores themselves have however the Shannon scores show that at least for the nine blocks after the first block (file headers and footers tend to deviate significantly from the rest of their content), that the information entropy is more or less the same. Here the Pi scores don’t serve us very well as almost all of them are in ranges well over 100% difference.

However, when we get to blocks 0x0B through 0x15, we see that the amount of difference between block 0x05 and them starts to become more uniform, as does the Shannon and Pi approximation scores. This is more or less to be expected because we’re really looking at the same data as when we were viewing the scores earlier, we’re just viewing it in terms of differences than as raw scores. For the same reasons as when we first looked at the file when it had a one-byte XOR encrypted executable embedded in it, we would likely take a closer look at all ranges that are relatively uniform. Let’s take a look at that now specifically looking at a block that falls within that mostly uniform range, say 0x0D:

```
BN:   C          C: 23.7949      S: 1.1859      ES: 1.8543      ER: 30.2422
BN:   D          C: 4.0646       S: 0.1532      ES: 1.2105      ER: 16.0187
BN:   F          C: 1.7538       S: 0.1138      ES: 1.2105      ER: 16.0187
BN:  10          C: 15.0578      S: 0.6228      ES: 11.7481     ER: 94.3086
BN:  11          C: 13.2060      S: 0.5457      ES: 4.2321      ER: 47.2232
BN:  12          C: 7.8096       S: 0.4528      ES: 4.9315      ER: 53.1004
BN:  13          C: 8.2764       S: 0.3210      ES: 9.8177      ER: 84.9327
BN:  14          C: 5.8316       S: 0.2473      ES: 3.9536      ER: 44.7630
BN:  15          C: 13.9706      S: 0.4674      ES: 4.0928      ER: 46.0012
BN:  16          C: 5.0212       S: 0.1750      ES: 3.8147      ER: 43.5083
BN:  17          C: 9.6567       S: 0.3483      ES: 7.7778      ER: 73.3010
BN:  18          C: 13.4137      S: 0.2990      ES: 5.3535      ER: 56.4512
BN:  19          C: 11.8915      S: 0.3605      ES: 0.4019      ER: 5.5966
BN:  1A          C: 14.7676      S: 0.3728      ES: 2.9851      ER: 35.6130
BN:  1B          C: 16.5408      S: 0.5457      ES: 0.5333      ER: 7.9061
BN:  1C          C: 11.8384      S: 0.3242      ES: 0.4003      ER: 5.8796
BN:  1D          C: 10.3172      S: 0.3273      ES: 0.6662      ER: 9.9674
BN:  1E          C: 29.4325      S: 1.0712      ES: 2.2989      ER: 28.5116
BN:  1F          C: 51.0710      S: 1.7271      ES: 0.6662      ER: 9.9674
BN:  20          C: 67.6754      S: 2.1520      ES: 8.3565      ER: 76.8073
BN:  21          C: 68.4539      S: 2.1903      ES: 1.7230      ER: 27.8235
BN:  22          C: 81.9809      S: 2.5465      ES: 0.4019      ER: 5.5966
BN:  23          C: 68.4350      S: 2.2414      ES: 2.9851      ER: 35.6130
BN:  24          C: 74.1537      S: 2.3939      ES: 0.2677      ER: 3.7612
BN:  25          C: 51.3289      S: 1.8181      ES: 1.6173      ER: 20.8788
BN:  26          C: 94.6866      S: 3.0066      ES: 0.9402      ER: 12.6527
BN:  27          C: 136.2424     S: 3.6503      ES: 8.2051      ER: 154.9369
BN:  28          C: 147.9159     S: 3.7974      ES: 10.7527     ER: 74.1581
```

As we look at the data sequentially around and following block 0x0D, we see that the data is relatively uniform relative to the other blocks, but we should notice a loose trend—which is that as we process the blocks that follow, they most increase in the volume of difference in terms of the Chi score. The

Shannon scores also fluctuate and grow more, but the difference is not as noticeable as it is in the Chi scores. If we go back and look at the scores themselves, we notice that this pattern loosely holds there as well, however it didn't jump out at us in the same way that looking at the differences did. If we contrast these differences with a block of embedded data relative to the other embedded data, we will see even more differences between the host file data that is somewhat uniform and the hidden data stream:

BN: AE	C: 110.5005	S: 3.7129	ES: 1.6129	ER: 20.3651
BN: AF	C: 0.4373	S: 0.2518	ES: 0.5824	ER: 8.6349
BN: B0	C: 1.6426	S: 0.0791	ES: 1.7575	ER: 28.7364
BN: B2	C: 4.5721	S: 0.1344	ES: 0.2320	ER: 3.2311
BN: B3	C: 1.0137	S: 0.1007	ES: 1.3841	ER: 17.7505
BN: B4	C: 1.3876	S: 0.0075	ES: 1.7271	ER: 21.6402
BN: B5	C: 2.9956	S: 0.0146	ES: 1.8412	ER: 22.8944
BN: B6	C: 0.6252	S: 0.1050	ES: 1.2695	ER: 16.4098
BN: B7	C: 0.0623	S: 0.0231	ES: 1.2695	ER: 16.4098
BN: B8	C: 7.0840	S: 0.3611	ES: 0.5824	ER: 8.6349
BN: B9	C: 74.5826	S: 3.6457	ES: 1.8412	ER: 22.8944
BN: BA	C: 113.6834	S: 6.3999	ES: 6.8424	ER: 63.8332
BN: BB	C: 3.3782	S: 0.3193	ES: 0.1161	ER: 1.6306
BN: BC	C: 10.4848	S: 0.3199	ES: 1.2695	ER: 16.4098
BN: BD	C: 7.4332	S: 0.2253	ES: 0.9249	ER: 12.2471
BN: BE	C: 0.9837	S: 0.1105	ES: 3.7607	ER: 41.4647
BN: BF	C: 5.8978	S: 0.2193	ES: 1.8412	ER: 22.8944
BN: C0	C: 13.1830	S: 0.4676	ES: 0.6944	ER: 9.3486
BN: C1	C: 0.9818	S: 0.1535	ES: 1.8757	ER: 30.9909
BN: C2	C: 24.0690	S: 0.8330	ES: 1.0399	ER: 13.6587
BN: C3	C: 130.9636	S: 7.4364	ES: 6.2992	ER: 60.4042
BN: C4	C: 169.6871	S: 22.8591	ES: 4.9830	ER: 51.2451
BN: C5	C: 189.6926	S: 86.2849	ES: 17.2944	ER: 106.0099
BN: C6	C: 189.6926	S: 86.2849	ES: 17.2944	ER: 106.0099

As we review this data corresponding to the differences between block 0xB1 and the other embedded data, we should immediately see the differences between this data and the aforementioned data that showed some level of uniformity. For instance, the level of uniformity in the Chi scores is greater with the difference between the blocks ranging in single digit percentages and in some cases less than 1%. This contrasts with the other blocks which entirely showed double digit percentage differences in the Chi score. Secondly, the Chi scores do not show an upward trend like the other blocks did. There is actually no real discernable pattern here, sometimes it's a few points more, sometimes a bit less. Next, the level of information entropy is somewhat constant. There is the outlier block at index 0xBA, however by and large the difference is less than 1%, often in the 0.3% range. Finally, there is a higher level of uniformity in the Pi approximation errors as well. Thus we can say with some degree of certainty that the distributions here are more uniform than the earlier blocks, there are no upward or downward trends and as such they are more indicative of cipher-text as the distribution *should* conform to this model when dealing with cipher-texts.

Okay, let’s recap what we’ve discussed in this section and move onto something a bit closer to the actual threat data we’d be looking at and while we’re at it demonstrate the sequential deviation feature.

- Sometimes the scores themselves make patterns harder to discern, we can elect to view the scores as percentage of difference relative to other blocks which will sometimes make them easier to spot.
- When view in this manner, we can often differentiate blocks that have somewhat uniform scores from blocks that really are uniform.
- Blocks of related data will often have a certain structure to them, but they will also often follow a discernable pattern—for instance in this example the percent of differences gradually increased for the most part showing; moreover, when we viewed them this way we noted that the differences between the blocks were higher than we really thought.
- When we viewed the actual encrypted executable in this manner, we noted that this ascending pattern vanished and that the data was far more uniform in its distribution.

3.1.5 Packed Executables in a File with Higher Entropy

Okay, thus far we’ve been using a sorta unrealistic use-case, which is that the executable in question was really only hidden by the XOR and due to this sections that were filled entirely with zeroes made the executable pretty easy to spot. This is due to the executables not being packed or otherwise obfuscated precisely because they were standard executable programs and not actually malware. Many academic and industry reports have shown that over 90% of malware is packed and so we can expect that this will be the case in most instances where we are viewing live data. Moreover, the file we used was relatively low in entropy and so when a long XOR key was used, it was equally easy to identify.

So, let’s get closer to reality and look at a different file with some packed instances of cmd.exe. As it turns out, US President Barack Obama’s long-form birth certificate fits this model very well. Let’s take a look at the file without anything embedded in it:

```
FILE: birth-certificate-long-form-CLEAN.pdf BLOCK COUNT: 47 BLOCK SIZE: 8192

ALL SCORES
BN:   0         C: 2015.8750    S: 7.8790     ES: 3.2383     ER: 2.9858
BN:   1         C: 814.5625     S: 7.9356     ES: 3.2188     ER: 2.3971
BN:   2         C: 623.9375     S: 7.9503     ES: 3.2266     ER: 2.6334
BN:   3         C: 675.6250     S: 7.9454     ES: 3.0938     ER: 1.5464
BN:   4         C: 681.8750     S: 7.9456     ES: 3.1250     ER: 0.5310
BN:   5         C: 772.6250     S: 7.9394     ES: 3.1328     ER: 0.2803
BN:   6         C: 582.6875     S: 7.9514     ES: 3.1445     ER: 0.0935
BN:   7         C: 593.7500     S: 7.9532     ES: 3.1914     ER: 1.5609
BN:   8         C: 1954.5625    S: 7.8853     ES: 3.0938     ER: 1.5464
```

```
BN:  9     C: 802.8750     S: 7.9308     ES: 3.0195     ER: 4.0424
BN:  A     C: 635.4375     S: 7.9456     ES: 3.1055     ER: 1.1632
BN:  B     C: 652.0000     S: 7.9417     ES: 3.1523     ER: 0.3411
BN:  C     C: 737.0625     S: 7.9361     ES: 3.1719     ER: 0.9547
BN:  D     C: 689.4375     S: 7.9405     ES: 3.1602     ER: 0.5874
BN:  E     C: 662.6875     S: 7.9445     ES: 3.1328     ER: 0.2803
BN:  F     C: 747.0625     S: 7.9374     ES: 3.1406     ER: 0.0308
BN: 10     C: 717.8750     S: 7.9417     ES: 3.2305     ER: 2.7512
BN: 11     C: 608.0625     S: 7.9481     ES: 3.1836     ER: 1.3193
BN: 12     C: 737.4375     S: 7.9357     ES: 3.2305     ER: 2.7512
BN: 13     C: 550.0000     S: 7.9539     ES: 3.0664     ER: 2.4519
BN: 14     C: 733.7500     S: 7.9402     ES: 3.1328     ER: 0.2803
BN: 15     C: 832.3750     S: 7.9336     ES: 3.0938     ER: 1.5464
BN: 16     C: 816.5625     S: 7.9349     ES: 3.1328     ER: 0.2803
BN: 17     C: 602.0625     S: 7.9484     ES: 3.1016     ER: 1.2906
BN: 18     C: 831.0000     S: 7.9340     ES: 3.0977     ER: 1.4184
BN: 19     C: 645.1250     S: 7.9464     ES: 3.0625     ER: 2.5826
BN: 1A     C: 696.0625     S: 7.9394     ES: 3.0273     ER: 3.7739
BN: 1B     C: 706.2500     S: 7.9401     ES: 3.0547     ER: 2.8450
BN: 1C     C: 590.0625     S: 7.9495     ES: 3.2539     ER: 3.4517
BN: 1D     C: 683.2500     S: 7.9425     ES: 3.1055     ER: 1.1632
BN: 1E     C: 678.3750     S: 7.9418     ES: 3.1250     ER: 0.5310
BN: 1F     C: 709.5000     S: 7.9405     ES: 3.0586     ER: 2.7136
BN: 20     C: 692.5625     S: 7.9415     ES: 3.0117     ER: 4.3123
BN: 21     C: 669.6250     S: 7.9436     ES: 3.1367     ER: 0.1554
BN: 22     C: 689.4375     S: 7.9432     ES: 3.1328     ER: 0.2803
BN: 23     C: 695.0625     S: 7.9405     ES: 3.2305     ER: 2.7512
BN: 24     C: 851.0625     S: 7.9285     ES: 3.1602     ER: 0.5874
BN: 25     C: 834.0000     S: 7.9343     ES: 3.1875     ER: 1.4402
BN: 26     C: 772.1875     S: 7.9327     ES: 3.0469     ER: 3.1087
BN: 27     C: 657.9375     S: 7.9426     ES: 3.0547     ER: 2.8450
BN: 28     C: 554.0000     S: 7.9511     ES: 3.0469     ER: 3.1087
BN: 29     C: 821.1875     S: 7.9343     ES: 3.0273     ER: 3.7739
BN: 2A     C: 946.8125     S: 7.9167     ES: 3.1172     ER: 0.7829
BN: 2B     C: 908.6875     S: 7.9203     ES: 3.1250     ER: 0.5310
BN: 2C     C: 1061.7500    S: 7.9084     ES: 3.0586     ER: 2.7136
BN: 2D     C: 2115.1250    S: 7.8768     ES: 3.1289     ER: 0.4055
BN: 2E     C: 28234.6827   S: 7.0330     ES: 3.5515     ER: 11.5425
```

Here when we review the data, we see that the file contains almost entirely sections that fall within the 600-800 range for its Chi-Square score. The values of the Shannon score are also mostly uniform and the Pi approximation scores almost all approach Pi significantly. This file has been analyzed up and down by a good cross-section of the world and basically we know that it's (a) Real or at least wasn't modified significantly after the digital copy was created (Research suggests that we can determine that the only real modification was that someone switched the orientation of the scanned document so that it faced right side up), (b) that it was created on a fairly high-end scanner (Xerox WorkCenter) and (c) That the contents of the file in its entirety almost contain compressed images.

So we can explain why these values look almost like a cipher-text, although they lack the appropriate level of overall uniformity—every block or two has a healthy amount of deviation. This is going to be due to the compression that was used, which if you recall often results in entropy scores that look fairly similar to encryption. Again, because the file is a PDF, we note that the last few sections are significantly different than the rest of the file and this is easily explained by the type of data that comes towards the end of a PDF (and indeed, PDF files are generally parsed from bottom up as a result).

At any rate, let's take a look at the file with some packed executables embedded and then look at the file with one of the same packed executables that has also been encrypted with a one-byte XOR key.

```
Packed with: Enigma64 (DEMO)
BN:  10     C: 717.8750         S: 7.9417     ES: 3.2305     ER: 2.7512
BN:  11     C: 608.0625         S: 7.9481     ES: 3.1836     ER: 1.3193
BN:  12     C: 737.4375         S: 7.9357     ES: 3.2305     ER: 2.7512
BN:  13     C: 550.0000         S: 7.9539     ES: 3.0664     ER: 2.4519
BN:  14     C: 733.7500         S: 7.9402     ES: 3.1328     ER: 0.2803
BN:  15     C: 832.3750         S: 7.9336     ES: 3.0938      ER: 1.5464
BN:  16     C: 816.5625         S: 7.9349     ES: 3.1328     ER: 0.2803
BN:  17     C: 602.0625         S: 7.9484     ES: 3.1016     ER: 1.2906
BN:  18     C: 831.0000         S: 7.9340     ES: 3.0977     ER: 1.4184
BN:  19     C: 645.1250         S: 7.9464     ES: 3.0625     ER: 2.5826
BN:  1A     C: 696.0625         S: 7.9394     ES: 3.0273     ER: 3.7739
BN:  1B     C: 17884.1250       S: 7.6515     ES: 3.1719     ER: 0.9547
BN:  1C     C: 249.7500         S: 7.9778     ES: 3.1797     ER: 1.1981
BN:  1D     C: 261.5625         S: 7.9767     ES: 3.2109     ER: 2.1596
BN:  1E     C: 242.7500         S: 7.9783     ES: 3.1172     ER: 0.7829
BN:  1F     C: 302.8750         S: 7.9730     ES: 3.1016     ER: 1.2906
BN:  20     C: 242.2500         S: 7.9787     ES: 3.2578     ER: 3.5674
BN:  21     C: 228.2500         S: 7.9801     ES: 3.1523     ER: 0.3411
BN:  22     C: 260.0625         S: 7.9767     ES: 3.1719     ER: 0.9547
BN:  23     C: 260.9375         S: 7.9767     ES: 3.1523     ER: 0.3411
BN:  24     C: 262.0000         S: 7.9769     ES: 3.1172     ER: 0.7829
BN:  25     C: 250.3750         S: 7.9782     ES: 3.1875     ER: 1.4402
BN:  26     C: 217.6875         S: 7.9806     ES: 3.1641     ER: 0.7102
BN:  27     C: 8627.2500        S: 7.7907     ES: 3.1680     ER: 0.8326
BN:  28     C: 108569.0625      S: 6.8816     ES: 3.4375     ER: 8.6082
BN:  29     C: 1090000.4375     S: 2.3658     ES: 3.7617     ER: 16.4852
BN:  2A     C: 153039.6250      S: 6.5503     ES: 3.3555     ER: 6.3740
BN:  2B     C: 627246.3750      S: 3.0964     ES: 3.5625     ER: 11.8149
BN:  2C     C: 494859.5000      S: 4.1338     ES: 3.7227     ER: 15.6088
```

As we should really expect, the embedded executable was trivial to identify. This is not surprising because the executable is embedded in plain-text, and so will have a significant spike in locations associated with the executable files headers even though it’s compressed and encrypted. We see exactly that, where the Chi distribution score spikes far above average and then drops well below average into the ranges we’d associate with a cipher-text. As we get to more plain-text executable data structures, we see that the Chi score again grows and the Shannon entropy score decreases. This ease is to be expected and is one of the reasons the embedded executables are often encrypted.

Let’s take a look at another that is packed with a different packer:

```
Packed with: Obsidium64 (DEMO)
BN:   7        C: 593.7500       S: 7.9532     ES: 3.1914     ER: 1.5609
BN:   8        C: 458745.6250    S: 5.0382     ES: 3.5977     ER: 12.6767
BN:   9        C: 723316.1250    S: 3.8925     ES: 3.7305     ER: 15.7856
BN:   A        C: 313847.8750    S: 5.4424     ES: 2.4297     ER: 29.3003
BN:   B        C: 615406.6875    S: 3.2006     ES: 1.1445     ER: 174.4873
BN:   C        C: 182168.5000    S: 5.9883     ES: 2.7656     ER: 13.5943
BN:   D        C: 227.3750       S: 7.9797     ES: 3.1523     ER: 0.3411
BN:   E        C: 258.5625       S: 7.9771     ES: 3.1523     ER: 0.3411
BN:   F        C: 228.3750       S: 7.9797     ES: 3.1367     ER: 0.1554
BN:  10        C: 261.1875       S: 7.9769     ES: 3.1602     ER: 0.5874
BN:  11        C: 278.9375       S: 7.9765     ES: 3.1172     ER: 0.7829
BN:  12        C: 277.3125       S: 7.9753     ES: 3.1680     ER: 0.8326
BN:  13        C: 273.5000       S: 7.9759     ES: 3.1641     ER: 0.7102
BN:  14        C: 261.8125       S: 7.9772     ES: 3.1328     ER: 0.2803
BN:  15        C: 242.1875       S: 7.9785     ES: 3.0312     ER: 3.6402
BN:  16        C: 246.3125       S: 7.9781     ES: 3.1758     ER: 1.0765
BN:  17        C: 239.9375       S: 7.9789     ES: 3.1523     ER: 0.3411
BN:  18        C: 244.0625       S: 7.9781     ES: 3.1055     ER: 1.1632
BN:  19        C: 288.5000       S: 7.9746     ES: 3.1641     ER: 0.7102
BN:  1A        C: 241.8750       S: 7.9788     ES: 3.1797     ER: 1.1981
```

We again see the same pattern, even if the Chi and Shannon scores didn’t deviate so much from the plain-text portions of the executable, we’d notice the large deviance values when we got to the cipher-text portions. So let’s take a look at something that is a packer but isn’t really meant for protection purposes and just uses compression, UPX for instance:

```
Packed with: UPX
BN:  1F      C: 19556.5000    S: 7.6185    ES: 3.1172    ER: 0.7829
BN:  20      C: 1051.3125     S: 7.9135    ES: 3.2461    ER: 3.2193
BN:  21      C: 932.1250      S: 7.9234    ES: 3.3516    ER: 6.2648
BN:  22      C: 987.8125      S: 7.9185    ES: 3.2070    ER: 2.0405
BN:  23      C: 1112.6250     S: 7.9087    ES: 3.3242    ER: 5.4938
```

```
BN:  24          C: 901.3125        S: 7.9243    ES: 3.2461    ER: 3.2193
BN:  25          C: 882.8125        S: 7.9252    ES: 3.3750    ER: 6.9158
BN:  26          C: 1076.2500       S: 7.9129    ES: 3.3203    ER: 5.3826
BN:  27          C: 804.1875        S: 7.9321    ES: 3.3594     ER: 6.4828
BN:  28          C: 879.6875        S: 7.9282    ES: 3.3086    ER:5.0475
BN:  29          C: 909.3750        S: 7.9233    ES: 3.3984    ER:7.5577
BN:  2A          C: 1694.2500       S: 7.8676    ES: 3.4258    ER: 8.2956
BN:  2B          C: 13921.1875      S: 7.5441    ES: 3.2344    ER: 2.8686
BN:  2C          C: 201092.0000     S: 6.0933    ES: 3.5625    ER: 11.8149
BN:  2D          C: 1008145.8750    S: 2.8663    ES: 3.7617    ER: 16.4852
BN:  2E          C: 191183.8750     S: 6.1814    ES: 3.4258    ER: 8.2956
BN:  2F          C: 657594.1250     S: 3.0702    ES: 3.6445    ER: 13.7998
BN:  30          C: 343743.0625     S: 4.5307    ES: 3.7578    ER: 16.3984
BN:  31          C: 4400.4375                    S: 7.8389     ES: 3.1680
```

Here again, we see the large deviations when it encounters the plain-text data structures, however the executable data itself conforms pretty closely to the data in the host file—this is because it is not encrypted but rather compressed, just like the data stored in the file. So that was trivial, but at least in part because we could easily see the plain-text portions due to the large deviations and/or the cipher-text where the level of information entropy grew significantly. How about we take a look at the UPX data again, but this time let's use a one-byte XOR key:

```
BN:  17          C: 602.0625         S: 7.9484       ES: 3.1016       ER: 1.2906
BN:  18          C: 831.0000         S: 7.9340       ES: 3.0977       ER: 1.4184
BN:  19          C: 704.5000         S: 7.9427       ES: 3.0938       ER: 1.5464
BN:  1A          C: 16275.0000       S: 7.6213       ES: 3.2734       ER: 4.0277
BN:  1B          C: 927.0625         S: 7.9235       ES: 3.3047       ER: 4.9353
BN:  1C          C: 1001.1250        S: 7.9185       ES: 3.3789       ER: 7.0234
BN:  1D          C: 1043.6875        S: 7.9123       ES: 3.2773       ER: 4.1421
BN:  1E          C: 988.3125         S: 7.9188       ES: 3.2656       ER: 3.7981
BN:  1F          C: 898.3750         S: 7.9250       ES: 3.3203       ER: 5.3826
BN:  20          C: 987.8125         S: 7.9191       ES: 3.4297       ER: 8.4000
BN:  21          C: 864.8125         S: 7.9262       ES: 3.3242       ER: 5.4938
BN:  22          C: 814.2500         S: 7.9335       ES: 3.2891       ER: 4.4836
BN:  23          C: 979.1875         S: 7.9175       ES: 3.3750       ER: 6.9158
BN:  24          C: 1247.6875        S: 7.8982       ES: 3.4414       ER: 8.7120
BN:  25          C: 10055.5000       S: 7.6308       ES: 3.3594       ER: 6.4828
BN:  26          C: 71387.8750       S: 6.9820       ES: 3.4180       ER: 8.0860
BN:  27          C: 1076078.0000     S: 2.4082       ES: 3.7930       ER: 17.1733
BN:  28          C: 159641.3125      S: 6.5189       ES: 3.4180       ER: 8.0860
BN:  29          C: 616041.1250      S: 3.1197       ES: 3.4688       ER: 9.4316
BN:  2A          C: 547725.6250      S: 3.8207       ES: 3.7188       ER: 15.5202
BN:  2B          C: 18376.4375       S: 7.4728       ES: 3.3359       ER: 5.8258
```

Here we have the same executable still embedded in the long-form birth certificate. It has been packed with UPX and then the resulting executable was encrypted with a single byte XOR key. As with the plain-text executable, it's fairly trivial to spot where it starts. If you said block 0x1A, you'd be almost correct. It actually starts in block 0x19 however it starts almost at the very end of that block and thus doesn't significantly impact the distribution as a result. So we can see that even with the executable being packed and then encrypted, the differences between it and even compressed data is significant enough to easily spot it relative to the host file. To demonstrate one last feature, let's take a look at this file when the utility is told to generate the data for sequential deviations—which is to say that it will print the differences between scores for block 1 to block 2, and block 2 to block 3 and block 3 to block 4 and so on. We're not going to spend a whole lot of time on it, as it doesn't actually show us a whole lot new in terms of analyzing this file, but it's actually a pretty handy feature for spotting abnormal spikes and valley's within the data relative to one block to another.

We specify that by passing the `–seqdev` option, or we can pass one of the other related options such as `–seqxy` which operates the same but you can specify only a subset of the blocks that you want to view:

```
$ bin/edfind.py --seqdev birth-certificate-long-form-ONE-UPX.pdf | less
```

Which produces the output:

```
BN [ 17  : 18  ]     C: 31.9508      S: 0.1822      ES: 0.1260      ER: 9.4300
BN [ 18  : 19  ]     C: 16.4767      S: 0.1103      ES: 0.1262      ER: 8.6382
BN [ 19  : 1A  ]     C: 183.4035     S: 4.1303      ES: 5.6442      ER: 89.0285
BN [ 1A  : 1B  ]     C: 178.4430     S: 3.8880      ES: 0.9501      ER: 20.2508
BN [ 1B  : 1C  ]     C: 7.6821       S: 0.0636      ES: 2.2209      ER: 34.9225
BN [ 1C  : 1D  ]     C: 4.1630       S: 0.0785      ES: 3.0516      ER: 51.6104
BN [ 1D  : 1E  ]     C: 5.4503       S: 0.0829      ES: 0.3582      ER: 8.6644
BN [ 1E  : 1F  ]     C: 9.5339       S: 0.0783      ES: 1.6607      ER: 34.5179
BN [ 1F  : 20  ]     C: 9.4834       S: 0.0749      ES: 3.2407      ER: 43.7856
BN [ 20  : 21  ]     C: 13.2785      S: 0.0897      ES: 3.1232      ER: 41.8348
BN [ 21  : 22  ]     C: 6.0227       S: 0.0928      ES: 1.0632      ER: 20.2489
BN [ 22  : 23  ]     C: 18.3934      S: 0.2023      ES: 2.5791      ER: 42.6711
BN [ 23  : 24  ]     C: 24.1145      S: 0.2447      ES: 1.9484      ER: 22.9871
BN [ 24  : 25  ]     C: 155.8465     S: 3.4438      ES: 2.4124      ER: 29.3407
BN [ 25  : 26  ]     C: 150.6135     S: 8.8797      ES: 1.7291      ER: 22.0080
BN [ 26  : 27  ]     C: 175.1146     S: 97.4170     ES: 10.4009     ER: 71.9521
```

Here the format is slightly different, where the block number is in other output is actually a range of block numbers, specifying that we're comparing one block to another—for instance the first line "`BN [ 17 : 18 ]`" says that the data being produced is the difference in scores for block 0x17 to block 0x18.

The rest of the line has the standard format. As we can see, once we get out of the XOR encrypted executable header data and into the compressed body, the difference between sequential blocks is relatively low. Moreover the variance between the Shannon scores is extremely low, mostly ranging less than 0.09%. While we're not demonstrating any new analysis in this particular example, this feature is useful once a suspect range has been identified and you want to say essentially "and show me all of the blocks that come after the suspect block that only vary by X%".

So to review what we've covered:

- Packed executables do not overly present a problem, especially the ones focused more on providing software security versus ones that just exist to make the executable smaller. This is largely due to the fact that the cipher-text varies significantly and really won't occur at the ratios examined unless the host file data is also encrypted.
- Even when the packed file is subsequently XOR encrypted, we can easily spot the differences from the host file data.
- A packer or similar scheme that compressed the data instead of encrypting it is more likely to bypass any entropy-based anomaly checks.
- The sequential deviation option can be useful in more automated analysis styles to tell the tool to also show you blocks which only deviate a certain amount after an initial suspect block is identified.

3.2 Putting it all together: automated analysis

While much of this manual has been focused on using the included example utility, edfind.py, the actual intention of this module is to provide a basic starting point for analyzing potentially suspicious files without performing expensive XOR brute force searches or similar. We can attempt to identify differences in the entropy levels and when we encounter a circumstance that matches then we can "drill down" and perform more expensive analysis.

Included in the edfind.py utility is a *pure example* function that attempts to identify suspicious block ranges in the spirit of performing such analysis in an automated manner. As stated elsewhere in the manual, these tests and in particular the ratios used are not expected to be overly successful when operating on real data sets and is not intended to do so. Rather, it is intended to provide a starting pointing for a person who seeks to utilize entropy analysis to identify hidden data as to the types of tests that can be performed and serve as a mental Launchpad of sorts for extending and modifying the base ideas presented in this module and manual. What follows is a brief description of the type of tests it performs.

The steps performed, in order are:

1. Attempt to find blocks with a high amount of deviation related to blocks around them. This calls into the Python extension and retrieves the sequential deviation for each block. It then compares this deviation to values passed as parameters to the method.
2. From this list of blocks, it attempts to identify sequential blocks that have:
    a. Relative uniformity in their Chi-Square score

    b. Deviate significantly from their neighbor blocks
3. If the suspicious blocks we've identified match the prior criteria, we then check to see if the Shannon score for the block immediately preceding it to see if it differs significantly in the amount of information entropy it has, and if so it checks the block that follows to see if the two blocks have relative uniformity in the amount of information entropy they both have.
4. If the results of (3) are positive, then we attempt to identify the entire range of blocks by looking for sequential blocks that have relative low Shannon scores when compared to one another.
5. If the results of (4) are positive, then we check the range to see if there is relative uniformity in the Chi-Squared scores when compared to one another.
6. We take all of the block ranges that conform to these standards and compile them into a list. We then check that list for ranges that neighbor each other but are within a short distance from one another. We do this due to the fact that there is often significant outlier blocks in the data and this will cause blocks that are part of the same executable to be identified as two different suspicious ranges of blocks. By coalescing this, we can better identify them as a contiguous region.
7. We then print out the blocks that have matched all of these criteria.

The tests themselves are written very loosely, as stated towards the beginning of this document it had a relatively high false positive rate and a higher than wanted false negative rate, but that this functionality was written against a single file with an embedded executable and not overly tested against a more authoritative dataset. This functionality is largely intended as an illustrated example of the types of tests that can be put together to help weed through large data-sets and thus save the more resource intensive tests for files that more adequately fit a criteria rather than attempting them on every single file.

3.3 Conclusion

In this manual, we have outlined the Python and C++ classes that comprise the extension, we have further illustrated example usages of a demonstration utility included with the distribution. We have analyzed files and demonstrated that the method of utilizing entropy analysis to help identify rogue data from benign is indeed useful. We have further tied together various parts of the utility into a very simple series of tests that demonstrate the type of tests one would want to perform to weed out benign data from the more interesting data in order to optimize the over-arching process. We have shown that when viewed from this perspective, it is actually not trivial to hide the embedded file without taking significant steps to match the embedded data to its host file and that utilizing encryption often makes it easier to identify, not harder. As such the attacker is forced into a trade-off, where in one sense it is harder to identify the hidden data because it is not embedded plain-text, but is easier to identify as being deviant relative to the host file precisely because it is not plain-text and deviates significantly from the data it is embedded in.

It is expected as time progresses the techniques used to embed this data will evolve and better model its host, but for the time being: this is not the case and entropy analysis is a cheap useful tool to deploy.