v1pad Manual

Author : Rick Lotero
Created: November 14, 2016
Updated: 6/9/2018 2:51:00 PM

This is the manual for v1pad.

Table of Contents

Dedication

v1pad is dedicated to Jehovah God and to my wife Bessy.

Jehovah God is the God of the Bible. He created me and gave me the ability to design and build computer programs. My wife Bessy believed I could create v1pad before I believed it myself. She put up with me spending literally years designing and building it when I could have been doing other things.

v1pad is being made available as open source software under version 2.0 of the Apache License for many reasons. One reason is to provide good online privacy for everyone who wants it. Everyone deserves privacy. Another reason is to make it easy for real encryption experts to examine the source code and determine whether v1pad really provides strong encryption and is free of back doors and other deliberate weaknesses. Not all reasons are listed here.

Introduction

v1pad

v1pad is a command-line tool that encrypts and decrypts files using one-time-pad encryption. v1pad uses a virtual pad, meaning that the one-time pad is generated dynamically based on user input instead of being stored in a file. The name v1pad is an abbreviation for Virtual One-Time Pad.

How to use this manual

The next section in this manual gives a simplified description of v1pad and how to use it. The rest of the manual describes v1pad in more detail. This includes a more complete description of v1pad features, how they work, and how to use them.

If you just want to use v1pad to encrypt and decrypt files and don't want to bother with any internal details or advanced features, all you really need to do is read the section on using v1pad.

If you want to really understand how v1pad works or take advantage of more advanced v1pad features, you should read the whole manual.

v1pad feature summary

Here is a table of the major v1pad features.

Name	Description	Status	Comments	See Also
One-time-pad encryption	Encryption using a virtual one-time pad generated based on user input	Optional, but enabled by default	Enabled with the "most secure" pad generation algorithm by default. Can be turned off. Other pad generation algorithms can be used	One-time-pad encryption A simplified version of v1pad encryption
Plain-text encryption	Encrypting plain text prior to applying one-time pad	Optional, but enabled by default	Enabled with the "most secure" plain-text encryption algorithm by default. Can be turned off. Other plain text encryption algorithms can be used	Plain Text Encryption
Password	Entering a password	Required	The v1pad password is really a passphrase, meaning a single line of text that can include spaces and tabs. Length can be between 8 and 255 characters, inclusive. Multibyte characters are allowed.	Passphrase

Name	Description	Status	Comments	See Also
Keyfiles	Extending the internal encryption key using data read from files	Optional. Cannot be disabled, but user can choose not to enter any keyfiles	Best practice is to specify one or more keyfile directories that v1pad reads recursively. Can also enter keyfile names directly. Keyfile entries must use single-byte characters. Maximum keyfile entry length is slightly less than 4k characters. v1pad fails if there is at least one keyfile entry, but no valid keyfiles were specified.	Keyfiles
Keyfiles Base Directory	Defining a base directory for keyfiles, which allows actual keyfile entries to be specified using pathnames relative to the keyfiles base directory (not relative to the current working directory)	Optional. Enabled through out-of-the box configuration file settings. Can be unset or set to other values in configuration file or on command line	Intended to make it easier to enter keyfiles. v1pad ships with default keyfiles and a keyfiles base directory setting that points to the default keyfiles directory. This allows the default keyfiles to be used by just entering a period (.) character.	How v1pad uses keyfiles
Random Block	Adding a block of "really-random" data to the internal encryption key so the generated pad will be different every time	Optional, but enabled by default. Can be disabled using a command-line argument	The Random Block is encrypted and written to a pseudo-random location in the output file.	The Random Block

Name	Description	Status	Comments	See Also
Input data padding	Padding the input data to an even multiple of the random block size	Optional, but enabled by default. Enabled or disabled along with the Random Block feature.	Input data padding makes handling the Random Block easier and ensures that plain text encryption will produce sufficiently-complex output	The Random Block
Additional password(s)	Making the way the keyfiles and random block are processed depend on both the passphrase and the additional password(s)	Optional. Cannot be disabled, but the user can choose not to enter any additional passwords	The additional passwords are really add-on keyfile entries. Syntax is <letter>::<string>. <string> may be empty. <string> may contain spaces and tabs. Since this is a keyfile entry, it must be composed of single-byte characters with a maximum total length slightly less than 4k characters long	Comments on the outline of setup steps How v1pad uses keyfiles Keyfile Examples
Dynamic keyfiles base directory modification	Modifying the keyfiles base directory in the middle of entering keyfiles, so that some keyfiles are found relative to one keyfiles base directory, and others relative to a different keyfiles base directory	Optional. Cannot be disabled, but the user can choose not to enter any add-on keyfiles entries that modify the keyfiles base directory	This is a special case add-on keyfile entry. Syntax is (b|B)::<path>. <path> must either be a null string (clears the keyfiles base directory) or a valid directory name readable by the OS user running v1pad	How v1pad uses keyfiles Keyfile Examples

Name	Description	Status	Comments	See Also
Key Indirection	Replacing user key input (passphrase and keyfile entries) with modified versions internally	Optional and disabled by default	Intended to provide a defense against key logger software. User must create Key Indirection data files before using Key Indirection. The modified version of the passphrase is very complex. Includes an optional sub-feature that provides protection against an attacker who knows the original key input and has a copy of the Key Indirection data files	Key Indirection

Using v1pad

This section of the manual provides a brief introduction to v1pad. The concepts and features described here are explained in more detail later in the manual. Most of the information here is over-simplified to give a general high-level understanding of v1pad without getting bogged down in details.

Overview

v1pad is a command-line tool that encrypts and decrypts files using one-time-pad encryption. v1pad uses a virtual pad, meaning that the one-time pad is generated dynamically based on user input instead of being stored in a file. The name v1pad is an abbreviation for Virtual One-Time Pad.

The core of v1pad is one-time-pad encryption. During encryption, the one-time-pad is constructed dynamically, combined with the plain text to produce encrypted text, and forgotten. During decryption, the one-time-pad is constructed dynamically, combined with the encrypted text to produce plain text, and forgotten.

To make v1pad resistant to "plain text" attacks, by default the plain text is encrypted using some method other than one-time-pad encryption before it is combined with the virtual pad. In a sense, it is doubly-encrypted. The first encryption is with a method other than one-time-pad encryption, and the second encryption is with the virtual pad. During decryption, the data has to be doubly-decrypted: first it is decrypted using the virtual pad, and then it is decrypted using the other method.

v1pad prompts for a password and zero or more keyfiles. It uses the password, data read from the keyfiles (if any), and a block of random bytes to construct a large internal key called a keyspace. v1pad uses data from the keyspace for encrypting and decrypting files.

The v1pad password is really a passphrase. Space and tab characters are allowed, the maximum length is 255 characters, and multibyte characters are allowed. The passphrase must be at least eight characters long, and must contain at least one lower-case letter, one upper-case letter, and one numeric or special character.

Keyfiles must be readable and at least 128 bytes long. To make it easier to use multiple keyfiles, v1pad allows the list of keyfiles to include directories which are read recursively. Keyfiles are an optional feature. v1pad works without keyfiles, but it provides much better security with keyfiles.

v1pad uses pseudo-random numbers and "really-random" numbers. Pseudo-random numbers are generated by a Cryptographically-Secure Pseudo-Random Number

Generator (CSPRNG) based on user input. Really-random numbers are also generated using a CSPRNG, but the seed value for the CSPRNG is generated automatically in a way intended to make it difficult to reproduce later.

By default, a block of really-random bytes is included in the keyspace so that the virtual one-time-pad produced when running v1pad is different every time, even if the same password and keyfiles are used to encrypt more than one file. The block of really-random bytes also adds complexity to the keyspace. To make decryption possible, the exact same block of really-random bytes is encrypted and saved to a pseudo-random location in the output file.

v1pad is deliberately complex, but it is about the same to use as any other encryption tool. It takes the input and output file names as command-line arguments. By default, it prompts for key information. It can also read key information from standard input.

Some sample command lines:

```
v1pad action=Encrypt inputfile=Input.txt outputfile=Encrypted.txt

v1pad Encrypt Input.txt Encrypted.txt

v1pad Action=Decrypt InputFile=Encrypted.txt OutputFile=Decrypted.txt

v1pad Decrypt Encrypted.txt Decrypted.txt
```

v1pad uses <name>=<value> syntax for most arguments and <name> or `no`<name> syntax for Boolean arguments. Mandatory arguments may optionally be specified in positional order and without argument names, as demonstrated in the examples above. Argument names are case-insensitive, but argument values are case-sensitive.

To get full v1pad command-line help information, use the `Help` Boolean argument. For example:

```
v1pad help
```

Installing v1pad

v1pad is delivered as a zip file. To install it, just unzip the zip file into an appropriate directory. This document calls the directory where v1pad was unzipped the v1pad installation directory.

On Windows, v1pad is ready to run once you have unzipped it.

On Linux, some additional setup is required. After unzipping v1pad on Linux, you must go to the `Misc` subdirectory under the v1pad installation directory and run the program `v1pad_setup.sh` to link the v1pad executables. `v1pad_setup.sh` also modifies the `get_sys_info_v1p` program to be setuid root so it can query system information as root.

You will be prompted for your `sudo` password when `v1pad_setup.sh` runs, unless you recently ran some other `sudo` command.

v1pad gets some information from the configuration file `v1pad.properties` in the `config_v1pad` subdirectory under the v1pad installation directory. It will read configuration information from any directory that contains a file called `v1pad.properties` if that directory is specified as the value for the optional `ConfigurationDirectory` command-line argument. By default, `ConfigurationDirectory` is set to the directory `config_v1pad` in the current working directory.

To change the v1pad configuration, edit `v1pad.properties` in a text editor. By default, `v1pad.properties` only specifies values for the `KeyfilesBaseDirectory` argument and (on Linux) the `SystemQueryProgram` argument, but you can specify any command-line argument in the v1pad configuration file except for the `ConfigurationDirectory` argument itself. Inside the configuration file, argument names are case-sensitive, and Boolean arguments use standard <name>=<value> syntax, where the value must be a case-insensitive match to one of the strings: TRUE, FALSE, Yes, No, t, f, Y, N.

Before using v1pad

As was mentioned earlier, v1pad uses both pseudo-random numbers and "really-random" numbers, both of which it generates using a Cryptographically-Secure Pseudo-Random Number Generator (CSPRNG). The algorithm for automatically generating the seed value to produce really-random numbers uses information about the hardware configuration of your computer and the current state of the computer. The program `get_sys_info_v1p.exe` (`get_sys_info_v1p` on Linux) collects this information from your computer. Before you use v1pad, you should run this program once to make sure it works properly. If it does not work properly, v1pad will either fail or will produce low-quality "really-random" numbers.

Prerequisite software

On Windows, get_sys_info_v1p uses `WMIC` to collect system hardware information and system state information. Since `WMIC` ships with Windows, by default this program should be available. Some anti-virus or security software may restrict what `WMIC` can do or prevent it from running. If `WMIC` is present on your Windows system but get_sys_info_v1p still does not appear to work properly, it is possible that your anti-virus or security software is preventing it from running properly.

On Linux, get_sys_info_v1p uses the optional software packages `dmidecode` and `lshw`, plus various standard Unix utilities to collect system hardware information and system state information. If `dmidecode` and `lshw` are not installed on your Linux computer, you must install them in order for get_sys_info_v1p (and v1pad) to work properly.

Running get_sys_info_v1p

get_sys_info_v1p is located in the top level of your v1pad installation directory. Type `get_sys_info_v1p Help` to see detailed command-line argument information.

The command `get_sys_info_v1p all .` should cause `get_sys_info_v1p` to produce three output files in the current working directory: `System_ID_Info_Long.txt`, `System_State_Info.txt`, and `System_ID_Info_Short.txt`.

The file contents should have a format similar to the example data given in the appendix. The exact data is expected to be different on different computers. The format and contents are different on different Operating Systems and may even be different with different versions of the same Operating System.

If get_sys_info_v1p produces three output files, doesn't display any obvious error messages when run, and the format of the output files looks more-or-less the same as the sample output files for your OS, you can consider get_sys_info_v1p to have run successfully.

v1pad command-line arguments

v1pad is a command-line tool you can run from an Operating System shell prompt. It can also be run as a step in a batch file or in some other executable program.

The first three arguments to v1pad are: `Action`, `InputFile`, and `OutputFile`. These three arguments are mandatory. Because they are mandatory arguments, you can optionally specify just the argument values in the exact order <Action> <InputFile> <OutputFile>. This is not a valid syntax for the v1pad optional arguments.

The only two valid values for the Action argument are "Encrypt" and "Decrypt".

v1pad supports many optional command line arguments that are explained in the v1pad help text. Many v1pad command line arguments are only useful for very specific situations. Only the optional command-line arguments most likely to be used are described here.

The `Interactive` Boolean argument toggles prompting for the v1pad passphrase and list of keyfile entries. By default, v1pad runs in interactive mode. To make v1pad read the passphrase and keyfile entries from standard input, specify `noInteractive`.

v1pad has a few arguments that tell it how to encrypt and decrypt files: `OneTimePadAlgorithm`, `PlainTextAlgorithm`, and `PlainTextReKeyFrequency`. They are described here briefly and elsewhere in more detail.

`OneTimePadAlgorithm` tells v1pad how to generate the virtual pad. The valid values are `PadNone`, `PadSimpleDirect`, `PadSimpleScrambled`, `PadUniform`, and `PadKeyspace`. `PadNone` tells v1pad not to perform one-time pad encryption (only perform plain text encryption). `PadSimpleDirect` tells v1pad to generate one-time pad by directly using the CSPRNG output. `PadSimpleScrambled` tells v1pad to generate one-time pad by using a scrambled version of the CSPRNG output. `PadUniform` tells v1pad to generate one-time pad consisting of uniformly-distributed but randomly ordered byte values. `PadKeyspace` tells v1pad to generate one-time pad by combining CSPRNG output with the current contents of the keyspace. The default value is `PadKeyspace`, which is probably the hardest to reverse-engineer.

`PlainTextAlgorithm` tells v1pad how to encrypt the plain text before combining it with the virtual pad. The valid values are `PlainNull`, `PlainBLS`, `PlainPVS`, and `PlainAES128`. `PlainNull` tells v1pad not to encrypt the plain text (only perform one-time-pad encryption). `PlainBLS` tells v1pad to encrypt the plain text using the Bit-Level Scramble block cypher (BLS). `PlainPVS` tells v1pad to encrypt the plain text using Positional Value Substitution (PVS). `PlainAES128` tells v1pad to encrypt the plain text using the Advanced Encryption Standard (AES) block cypher with a 128-bit key (16-byte key). The default value is `PlainBLS`, which is more secure than the other two cyphers.

`PlainTextReKeyFrequency` tells v1pad how often to generate new keys for each plain text encryption algorithm. The valid values are `ReKeyOuter`, `ReKeyMiddle`, and `ReKeyInner`. `ReKeyOuter` tells v1pad to change the plain text key in the outer loop (once for every 16 Meg of plain text). `ReKeyMiddle` tells v1pad to change the plain text key in the middle loop (once for every 128 K of plain text). `ReKeyInner` tells v1pad to change the plain text key in the inner loop (once for every 4k of plain text). The default value is `ReKeyInner`, which would seem to be the most secure option.

The `KeyfilesBaseDirectory` argument specifies the default directory to use when searching for keyfiles. If a given keyfile entry is a relative pathname and `KeyfilesBaseDirectory` is specified, the full pathname of the keyfile entry is calculated relative to the keyfiles base directory instead of relative to the current working directory.

`KeyfilesBaseDirectory` has no default value, but the v1pad configuration file that ships with v1pad sets `KeyfilesBaseDirectory` to the default keyfiles directory that ships as a subdirectory of the `config_v1pad` directory in the v1pad installation directory. This configuration allows specifying the default keyfiles directory by entering a single period (.) character.

The default keyfiles are binary files 4k bytes long containing random binary data. They are located in four subdirectories under the `keyfiles` directory, with 128 keyfiles in each subdirectory. The default keyfiles are set up to meet all of the criteria for ideal keyfiles, but there is nothing special about their contents or organization. They make it easier for people to use keyfiles. If you don't use your own keyfiles, we recommend that you use the default keyfiles. Using keyfiles provides much better security than not using keyfiles,

even if you use the same default keyfiles that every other v1pad user has. See the section on keyfiles for a detailed explanation of why this is true.

The `KeyIndirectionMode` and `KeyIndirectionBaseDirectory` command-line arguments are used by the Key Indirection feature. By default, `KeyIndirectionMode` is set to "Disabled" and `KeyIndirectionBaseDirectory` has no value. This turns off the Key Indirection feature. If you want to use Key Indirection, read the section on Key Indirection first. Like keyfiles, Key Indirection provides higher security, but causes v1pad to rely on additional data files. Just as a file encrypted using keyfiles cannot be decrypted without the proper keyfiles, a file encrypted using Key Indirection cannot be decrypted without the proper Key Indirection data files.

`LogDirectory` and `LogFile` specify the directory and file name for the v1pad log file. They default to v1pad.txt in the current user temporary directory (/tmp on Linux).

Simplified best practices

Easy, but not very safe

The easiest way to use v1pad is to only enter a passphrase. This will provide security which is about as good as your passphrase. A good passphrase will be hard for someone else to guess and so will provide good security. A bad passphrase (like "Password1") will be easy for someone else to guess and so will provide bad security.

To run v1pad with just a passphrase, enter the passphrase when prompted. Then when v1pad asks you for keyfiles, enter a blank line, which means "I don't want to use any keyfiles". v1pad will encrypt or decrypt your file using just your passphrase.

Recommended

The recommended way to use v1pad is to provide it with three pieces of information:

1. A passphrase
2. A keyfiles directory
3. An additional passphrase

For reasons described elsewhere, providing a keyfiles directory and an additional passphrase greatly increase the security that v1pad provides. The cost of this additional security is that you have to enter additional information when encrypting files and re-enter the exact same additional information in the exact same order when decrypting files.

The easiest way to specify a keyfiles directory is to use the default keyfiles directory shipped and configured with v1pad out-of-the-box. To use the default keyfiles directory, specify your first keyfiles entry by entering a single period (.) character followed by return or newline. The period character means current working directory. Since by default v1pad is configured to look for keyfiles in the default keyfiles directory shipped with v1pad, specifying the current working directory will cause v1pad to use the full set of default keyfiles for encryption and decryption.

You can also specify your own keyfiles. This normally provides even better security than using the default keyfiles directory. Make sure you have read and understood the section on keyfiles before specifying your own keyfiles.

You can think of the additional passphrase as a way to strengthen your original passphrase, or you can think of it as the second part of a two-part authentication method. v1pad uses the additional passphrase to ensure that the way the keyfiles are read and the way other internal data structures are set up does not depend only on your main passphrase, so it's good to specify something here that is not easy to guess.

The easiest way to specify an additional passphrase is to specify your second keyfiles entry by entering the exact string 'a::', followed by your additional passphrase, followed by return or newline. The additional passphrase can be almost any string of single-byte characters. It does not have a minimum length, and the maximum length is around 4k.

The additional passphrase is really an add-on keyfile entry. The full syntax is <letter>::<string>, where <letter> can be any upper-case or lower-case letter in the English alphabet except for the letter 'b', which has a special meaning. Feel free to use any letter other than 'b' to start your additional passphrase.

You may specify more than one keyfiles directory, more than one additional passphrase, and/or individual keyfiles instead of directories containing keyfiles. If you want to do any of these more complicated things, make sure you read the section on keyfiles first.

After entering a keyfiles directory and an additional passphrase as your first two keyfile entries, enter a blank line as your third keyfile entry. This blank line means "I don't want to specify any more keyfile entries".

As a simplified best practice, just use the default values for all of the optional v1pad command-line arguments. The default values are intended to provide strongest security at the cost of slowest performance. Unless you are encrypting and decrypting many files or very large files, this is a good trade-off.

Key Indirection provides protection against key logger software and greatly increases the effective complexity of passphrase input, but it has a setup cost and requires some additional work whenever you run v1pad. It is disabled by default because of the extra work required. See the section on Key Indirection for more information.

Concepts

This section of the manual describes various concepts related to v1pad in detail.

Some of the information here is general knowledge that you might learn elsewhere. Other information describes things specific to v1pad or new inventions created along with v1pad, so it may be helpful to read through this chapter before reading the other chapters.

Scientific Notation

This document uses scientific notation in some places. Scientific notation is a standard way to write numbers which is mostly only used for very large or very small numbers. The general format of scientific notation is <decimal_number>e<exponent>, where <decimal_number> is of the format <single_digit>.<at_least_one_more_digit> and <exponent> is an integer. Negative exponents are always written with a leading minus sign, and non-negative exponents are always written with a leading plus sign. Mathematically, a number in scientific notation is the product <decimal_number> multiplied by 10^<exponent> (ten raised to the <exponent> power).

Some examples of numbers in scientific notation are:

Number in standard notation	Number in scientific notation
-1	-1.0e+0
0	0.0e+0
0.000001	1.0e-6
0.02	2.0e-2
0.01	1.0e-2
0.1	1.0e-1
1	1.0e+0
10	1.0e+1
100	1.0e+2
200	2.0e+2
1,000,000	1.0e+6
1,048,576	1.048576e+6
1,000,000,000	1.0e+9

One-time-pad encryption

Background

One-time pad encryption has been used for a long time. The way it works is to generate a random pad with the same length as the plain text and use modular arithmetic to combine

the plain text with the pad to produce encrypted text. Decryption is the inverse: the encrypted text is combined with the pad using modular arithmetic to produce the original plain text.

One-time pad encryption has the property of "perfect secrecy", meaning that without the one-time pad there is no way at all to extract the plain text from the encrypted text. It is in theory possible to try all possible pads with the same length as the encrypted text one at a time, but this will just produce all possible plain texts with the same length as the encrypted text, with no real way to know which one matches the original.

The full set of conditions required for one-time pad encryption to meet the claim of perfect secrecy are actually very difficult to meet. The first condition is that the pad must be the same length as the plain text. The second condition is that the pad must be completely random. The third condition is that the pad can only be used once, ever. The fourth condition is that the pad must be kept completely secret. The requirements for the pad can be briefly summarized as: long, random, unique, and secret.

Making a pad that is the same length as the plain text is easy unless the plain text is extremely long.

Making a completely random pad is harder. To really get a completely random pad, it is necessary to have a source of completely random data. Since it's hard to be sure any given source of supposedly random data is completely random, implementations have in general settled for generating pads from data that is at least statistically random.

Making sure a pad is only used once is a challenge. It is possible to generate a set of 100 completely random pads, discard each pad after using it, and when done with those 100 generate a new set. What is hard is to be sure that the chances of anyone at all generating a pad identical to one that's already been used are vanishingly small.

Keeping the pad completely secret is also difficult. There must be two copies of the pad: one for the person encrypting and another one for the person decrypting, so already the problem is harder because there are two copies to protect. If the pads are files stored on a computer, they could possibly be accessed by a third party. If the third party finds the pads, that breaks the secrecy, so the pads themselves would need to be encrypted or stored on some kind of encrypted hardware. The secrecy of the pads themselves is limited by whatever technology is used to protect them, which in most cases means there is some industry-standard encryption protocol used, which takes a single password as its key.

Related to keeping the pads completely secret is the problem of making sure the person who encrypts and the person who decrypts have the same pads. If a physical meeting is not possible, this means that somehow the pads need to be distributed over a network, which means the pads could be read in transit.

The above constraints have made one-time pad encryption impractical for most use cases.

As strong as one-time pad encryption is, it has at least three weaknesses:

1) A plain text attack that is only useful if pads are re-used
2) A “man in the middle” plain text attack
3) A limited heuristic attack based on the message length

The plain text attack that is only useful if pads are re-used is a pretty simple attack. If the attacker knows the plain text for one message in a set of messages encrypted using the same pad, they can just XOR the plain text with the known message to produce the pad bytes, and then XOR the pad bytes with the other messages to get at least the first N bytes of the other messages, where N is the length of the known plain text message.

The “man-in-the-middle” plain text attack is also relatively simple. If an attacker somehow knows the plain text of a message encrypted with one-time pad encryption, and they can intercept the encrypted message before the recipient has a chance to read it, they can XOR the encrypted message with the plain text to get the one-time-pad bytes. Then they can alter the plain text in a way that does not change its length, re-encrypt it with the correct one-time-pad bytes, and send it on (or leave the altered version where they found it). The recipient will never know that the message was altered.

The limited heuristic attack based on the message length is probably less important, but it worth mentioning anyway. If an attacker is monitoring communications and knows something about the subject of the conversation, they may be able to get some information based just on the length of a message. For example, if they are expecting a “yes” or “no” answer to be given, and they see an encrypted message come across that only has two bytes in it, they can safely assume the answer was “no”.

In v1pad

As was mentioned earlier, v1pad implements one-time-pad encryption using a pad which is constructed from user input and some random values.

From above, the pad used in one-time-pad encryption must be long, random, unique, and secret.

The long and random requirements are met by using the user input and random values to seed a cryptographically-secure pseudo-random number generator (CSPRNG). In general, CSPRNGs have statistically-random output and a fairly long period, so unless the input file is very long, any good CSPRNG will produce enough statistically-random numbers to construct a pad.

The uniqueness requirement is met by including a block of 255 really-random bytes into the v1pad keyspace. 255 random bytes can produce up to 256^255 distinct values. According to the “birthday problem”, an attacker would have to generate about 256^127

blocks of 255 random bytes to have a better than 50% chance of producing the same block of random bytes. 256^127 is 7.0e+305, which is a very big number.

The secret requirement is met to the extent that the user input and the block of random values are kept secret. This means a little more than not telling someone what the input is. It must be very difficult for someone else to come up with the same input values in order for them to really be considered secret. Many of the features of v1pad were built specifically to help keep the user input and the block of random values secret.

Ideally, the user input and random values should be complex enough that it is computationally infeasible for an attacker to discover them by just trying all possible values (trying all possible values is known as a "brute force" attack).

Computational Infeasibility

Computational infeasibility is related to the time available for solving a problem and the resources available for solving it. When a problem is computationally infeasible to solve, the amount of work required to solve the problem is so large that it's not reasonable to believe it can be done with a specific set of resources within a specific time period. That does not at all mean that the problem is theoretically impossible to solve.

For example, if a person needed to compute the value 256^127 using only their mind and with no other tools available, that would be computationally infeasible for most people, even if given a few days to come up with a solution. If that same person was given a pen, paper, and a sufficient motive for doing the work, they could probably compute 256^127 in an hour or two using longhand multiplication. If that same person was given a calculator that can handle numbers as large as 256^127 (like the Windows calculator), they could probably compute 256^127 in a few seconds.

In the context of computer-based encryption, we have a situation similar to the person trying to compute 256^127. First of all, different computers have different capabilities. A problem that takes one hour to solve on a laptop computer may only take a few minutes or a few seconds on a more powerful computer. In addition, the time available for solving a problem has a big impact on whether it is computationally feasible or computationally infeasible. The problem that takes one hour to solve on a laptop computer is computationally infeasible to solve on that same computer if the requirement is to get an answer in one minute, but it's perfectly feasible to solve if the requirement is to get an answer some time in the next day.

Because the goal of v1pad is to really implement one-time-pad encryption using a virtual pad, we need to make sure that the amount of work required to find the exact set of random values and/or the exact user input values by with a "brute force" attack is large enough to be computationally infeasible. For that, we need to start by setting the resource and time constraints.

Ideally, we'd like it to be computationally infeasible for anyone to discover the random values or the user input by brute force computation, ever. If that was the goal, what would the resource and time constraints have to be in order to meet it?

In a paper on the computational capacity of the entire universe, Seth Lloyd of MIT concluded that over its estimated lifetime, the universe itself when treated as a single, giant computer can at the most have performed "no more than 10^120 ops on 10^90 bits". That would seem to mean that the computational capacity of the universe itself is 1.0e+120 operations, though maybe if the whole universe was treated as a single quantum computer focused on solving a single problem for 14 billion years you could get up around 1.0e+210 or so (10^120 * 10^90 is 10^210).

Assuming this paper is correct, anything above 1.0e+210 operations is computationally infeasible, period. Even if you used the whole universe as a single giant computer to figure out the value and were willing to wait 14 billion years to get the answer, you still wouldn't have it if the number of required computations is above 1.0e+210. It seems reasonable to assume that no human being and no human organization has the ability to use the whole universe as a single giant computer, and no one is willing to wait 14 billion years to get an answer, so there is no way at all to perform that many calculations. No one can do it: not with a quantum computer, and literally "not in a million years".

Based on the above, it would seem that to make breaking v1pad by brute force computationally infeasible, it is sufficient to make sure the number of operations required to break any one part of the algorithm is at least 1.0e+210.

Passphrase

A passphrase is essentially a password that can contain space and tab characters. Passphrases may have longer maximum lengths than passwords. This document uses the terms password and passphrase pretty much interchangeably.

Keyspace

A keyspace is basically a big encryption key. In v1pad, the keyspace always contains information from the main v1pad password. It ideally also contains information from the random block and the keyfiles.

Keyfiles

Keyfiles are files whose content is used somehow as part of the encryption key. If any keyfiles are specified when running v1pad, v1pad selects some number of bytes from the keyfiles and uses them to make the keyspace more complex.

Random Numbers

Random numbers can be broken into two groups: really-random numbers and pseudo-random numbers.

Really-random numbers can only be reliably generated by really-random physical events, like atmospheric turbulence and radioactive decay. They can also be approximated by sampling mostly-random physical events, like by looking at noise on a sound card.

Pseudo-random numbers are statistically-random numbers generated by some deterministic process. Given the same starting state, a pseudo-random number generator will always produce the exact same sequence of statistically-random numbers.

Truly random numbers are hard to generate on a computer, although there are several ways they can at least be approximated. Pseudo-random numbers are easy to generate. There are programs called Pseudo Random Number Generators (PRNGs) and Cryptographically-Secure Pseudo-Random Number Generator (CSPRNGs) which can generate a statistically-random sequence of numbers given some initial seed data.

Normally only pseudo-random numbers are used for the purposes of encrypting and decrypting data, since it is a requirement to reproduce the exact same sequence of pseudo-random numbers when encrypting and decrypting.

v1pad uses both pseudo-random numbers and an approximation of really-random numbers, but it uses them for different purposes.

Key Indirection

The general idea behind Key Indirection is for the actual input key to be intercepted internally and replaced with a modified key. This provides some protection against key logger software, since the key logger software at best can provide the attacker with the original key, not the internally-modified key.

v1pad implements Key Indirection as an optional feature. As a side effect of v1pad's implementation of Key Indirection, the internally-modified password is always complex enough to be computationally infeasible to break using brute force methods. This is a good reason to use Key Indirection even if you aren't worried about key logger software. To get this higher level of security, you have to do additional work to set up Key Indirection data files and manage their availability (so they are available for encryption and decryption, but not easy to steal).

BLS: Bit-Level Scramble

Bit-Level Scramble is a varying-length block cypher developed for use with v1pad. It's very simple, but at the same time powerful, scalable, and performant.

BLS encryption is implemented by performing a simple random scramble of each block of data at the bit level, using a set of pseudo-random numbers stored in an array. The scrambling algorithm is a variation of the standard Fisher-Yates / Durstenfeld algorithm. BLS decryption is implemented by filling an array with the exact same set of pseudo-random numbers used for encryption, and then running the encryption steps backwards.

The complexity of BLS output is equivalent to all possible combinations of all the one bits (or zero bits) in the set of input bytes. The output complexity is highest when the ratio of one bits to zero bits is 1:1, and lower when the ratio is skewed. The BLS encryption key is an array of 8*N random numbers, where N is the number of input bytes. BLS can be run with any non-zero block size. It produces higher output complexity with larger block sizes. There is no theoretical limit to the BLS block size, but there are practical limits.

See the description of the `PlainBLS` plain text transformation method in v1pad for more information on BLS.

PVS: Positional Value Substitution

Positional Value Substitution is a substitution cypher developed for use with v1pad. It is very fast but has known weaknesses.

Positional Value Substitution as implemented in the v1pad project uses an input block size of 256 bytes. The PVS key is made up of three two-dimensional arrays: Lookups, Scrambles, and Offsets. All three consist of 256 arrays of unsigned bytes, each of which contains the values from 0 to 255 in a pseudo-random order. The Lookups array holds mappings between input values and output values. The Offsets and Scrambles arrays cause the Lookups arrays to be mapped to the bytes in each input block in up to 256^2 different ways.

The difference between PVS encryption and PVS decryption is that for decryption the Lookups array is set up differently. For encryption, Lookups[*][I] contains the output value O corresponding to the input value I. For decryption, Lookups[*][O] contains the original input value I corresponding to the original output value O.

The PVS key can be re-engineered using a batch chosen plain text attack. Because of this, PVS should only be used if the PVS encryption key will not be re-used (chosen plain text attacks are only possible when the encryption key is re-used). v1pad normal mode is one example where it is safe to use PVS.

See the description of the `PlainPVS` plain text transformation method in v1pad for more information on PVS.

AES: Advanced Encryption Standard

Advanced Encryption Standard is a public domain encryption algorithm approved by the US Government for some applications. It is widely believed to be secure if implemented properly.

For more information about AES itself, see external sources like Wikipedia.

v1pad uses AES with a 128-bit (16-byte) key in Cypher Block Chaining mode.

See the description of the `PlainAES128` plain text transformation method in v1pad for more information on how v1pad uses AES.

Auth Files

Auth files contain two blocks of encrypted data of the same size, which are stored as base-64 encoded text. The Auth part stands for both Authentication and Authorization, which are the primary purposes of Auth files. The process of successfully validating an Auth file produces a single decrypted data block, which can either be discarded or used as input data for some other purpose.

An Auth block is two blocks of data in some more complex file which are created and validated the same way as the two data blocks in a standalone Auth file. Auth blocks are usually used to encrypt data in the rest of the file. They can also be used for other purposes.

The inputs for creating Auth files are a single block of really-random bytes of some specified length and a single encryption password. The data block is encrypted using the input password and written to the Auth file. Then the exact same data block is encrypted using some transformation of the input password and written to the Auth file.

Auth files are validated with the reverse process. An input password is given, the first data block is read from the Auth file and decrypted using the input password, and the decrypted first block is saved for future use. Then the second data block is read from the Auth file and decrypted using the same transformation of the input password which was used to create the Auth file, and the decrypted second block is saved for future use. Then the two decrypted data blocks are compared. If they are identical, the encryption password matches the validation password, and Authorization or Authentication is successful. If they are not identical, the encryption password does not match the validation password, and Authorization or Authentication is not successful.

Auth files are a possibly more secure implementation of the classic password validation approach of storing a hash of the password. Here instead of storing a hash of the password itself, we store the results of using the password to encrypt completely random data along with the results of using a transformation of the password to encrypt that same

random data. There is no rainbow table attack possible here, and on the surface of things, it would appear that password collisions would be extremely rare. Two different Auth files created using the same input password would not contain the same data, since the two different Auth files would use different starting blocks of random bytes.

Auth files and Auth blocks are used extensively by the Key Indirection feature. Outside of Key Indirection, v1pad only uses Auth files in the standalone `make_auth_file` utility. See the appendix for more information on the `make_auth_file` utility.

Table Hash

Table Hash is a scalable secure hashing algorithm designed along with v1pad, but not currently used by v1pad itself. It was designed to provide minimal hash collisions and to scale to any output block size from 1 byte to almost 64k bytes.

The reason Table Hash is not actually used in v1pad is simple. Table Hash is a very new secure hashing algorithm that has not been tested by fire. If some weakness were discovered in Table Hash, they could compromise v1pad itself. Instead, v1pad currently uses SHA2 and SHA3 hashes.

Assuming Table Hash proves itself to be a truly-secure secure hashing algorithm, v1pad may later be changed to use Table Hash in some places where it currently uses SHA2 or SHA3 hashes.

See the section on Table Hash for more information on Table Hash.

How v1pad sets up the keyspace

As was mentioned in the overview, v1pad prompts for a password and zero or more keyfiles. It uses the password, data read from the keyfiles (if any), and a block of random bytes to construct a large internal key called a keyspace. v1pad uses data from the keyspace for encrypting and decrypting files. This section explains all of this in more detail.

Overview

v1pad uses an internal keyspace of 4096 bytes, or 4k bytes. It uses a keyspace this large in part so that it will be able to fully-initialize the Cryptographically-Secure Pseudo-Random Number Generator (CSPRNG) from the keyspace, and in part so that the inherent complexity of the keyspace will be very large.

There are two main ways that the keyspace can be set up: with keyfiles and without keyfiles. The v1pad initialization logic is organized so that the steps to set up the keyspace without keyfiles are a subset of the steps to set up the keyspace with keyfiles.

Simple Mode

There is an optional feature in v1pad called simple mode in which the block of really-random bytes and the associated padding logic are absent. This is somewhat of a leftover from pre-release versions of v1pad.

Simple mode has the advantage of producing a smaller output file in less time because it skips initializing the "really-random" number generator and does not include the random block or any padding bytes in the output file. However, since it does not include the random block, any files encrypted using the same password, keyfile entries, and encryption parameters in simple mode will use the same set of virtual one-time-pad bytes, so anyone who uses simple mode must take great care not to ever re-use the same combination of password, keyfile entries, and encryption parameters. Also, because simple mode does not pad the encrypted text, the very last block of encrypted text may be small, and so may be easier to decrypt.

Simple mode is less secure than normal mode, so using it is not recommended. To enable simple mode, set the `EncryptionModeFlag` command-line argument to `Simple`. The default value for `EncryptionModeFlag` is `Normal`.

Outline of setup steps

1) Clear byte sources array.
2) Process the password, which includes
    a. Get password / passphrase from user
    b. Initialize the pseudo-random number generator from the password
    c. Set up the password byte source
3) Get the list of keyfile entries from the user
4) Even if no keyfile entries were specified
    a. Initialize Keyfile Entries Hash A and Keyfile Entries Hash B to some reasonable non-zero starting values.
    b. Get two 1k arrays of hashes based on the keyfile entries list: Keyfile Entries Hash A and Keyfile Entries Hash B. The hashes depend on the keyfile entries list contents and the keyfile entries list order. The two arrays of hashes differ only in the transformations performed on each keyfile entry just before merging each entry into the internal hashing data structures. If the list of keyfile entries is null, these arrays continue to have their original default values
    c. Mix the password byte source into Keyfile Entries Hash A, which basically XORs each original byte in Keyfile Entries Hash A with a randomly-chosen byte from the password byte source.
    d. Use the modified version of Keyfile Entries Hash A to re-initialize the random number generator so that the random numbers used below to read the keyfiles depend on both the password and on the set of keyfile entries
    e. Read the keyfiles to set up the keyfile byte sources. The keyfile byte sources are only changed if there is at least one valid keyfile specified.
    f. Re-initialize the keyspace using the current state of the random number generator and the current state of the byte sources data structure
    g. Re-initialize the random number generator from the keyspace array
5) If not running in simple mode
    a. Initialize the really-random number generator
    b. Get a set of 256 random bytes using the really-random number generator
    c. Determine how many padding bytes are required so that the input data can be rounded up to the next highest multiple of the block size (256 bytes)
    d. Create a “random block” which contains 255 of the random bytes plus one byte to track the number of padding bytes used
    e. Encrypt the random block
    f. Use pseudo-random numbers to determine where the random block will be written in the output file
    g. Set up random-bytes byte sources using the 255 random bytes
    h. The state of the pseudo-random number generator at this point may depend only on the passphrase or may depend on both the passphrase and the keyfile entries. Depending on both the passphrase and the keyfile entries provides better security, but that is only possible if least one keyfile entry was specified.

    i. Re-initialize the keyspace using the current state of the random number generator and the current state of the byte sources data structure
    j. Re-initialize the random number generator from the keyspace array
6) Even if no keyfile entries were specified
    a. Mix the current keyspace array into Keyfile Entries Hash B and use the resultant array to re-initialize the random number generator. Depending on whether simple mode is used, the random numbers used to process this second pass through the keyfiles will either depend on the password and the first traversal of the keyfiles or they will depend on the password, the first traversal of the keyfiles, and the random block
    b. Clear all keyfile byte sources
    c. Read the keyfiles to set up the keyfile byte sources again. The keyfile byte sources are only changed if at least one valid keyfile was specified
    d. If any valid keyfiles were specified, the keyspace produced by this final traversal includes the bytes from the password byte source and the random-byes byte sources (if any) set up earlier, plus a new set of keyfile bytes from the final traversal of the keyfiles. The set of keyfile bytes in the final keyspace depends indirectly on the set of keyfile bytes from the initial traversal of the keyfiles, but does not contain the actual keyfile data from the initial traversal
7) Re-initialize the keyspace using the current state of the random number generator and the current state of the byte sources data structure
8) The final re-initialization of the random number generator from the keyspace doesn't happen here, but it does happen later as part of the main encryption or decryption loop.

Comments on the outline of setup steps

The simplest path through the initialization steps above is to run in simple mode and not use keyfiles. The most complicated path is to run in normal mode and specify at least one keyfile entry. Not surprisingly, the simplest path provides the lowest level of security, and the most complicated path produces the highest level of security.

If simple mode is specified and no keyfiles are entered, only the initial setup based on the passphrase and the final re-creation of the keyspace run completely, so the contents of the keyspace really only depend on the passphrase. The two arrays of keyfile entry hashes are faithfully processed and used to reset the random number generator, but since there are no keyfile entries, these two arrays will always be set to their default values. When the passphrase is processed, it is expanded as needed so that the output fills the entire 4k bytes of the keyspace. Then the pseudo-random number generator and the byte sources array are initialized based on a combination of the initial keyspace derived from the passphrase and the passphrase itself. During the final re-creating of the keyspace, the keyspace is re-initialized from the byte sources data structure and padded to 4k bytes if needed. The keyspace is always padded for this use case, since the passphrase is smaller than 4k bytes, and v1pad requires that the keyspace be fully-populated.

In normal mode with no keyfiles entered, the steps related to the random block also run. This means the final contents of the keyspace depend on both the password and a block of 255 really-random bytes. The output bytes are padded to the next highest even multiple of 256 bytes. A "random block" containing the original 255 really-random bytes plus one byte tracking the number of padding bytes is constructed, and then encrypted. Later on, when actually encrypting the input file, the random block is written to the file at a pseudo-random location. The encrypted output bytes are written to the output file in blocks of 256 bytes each as well, so that it's easy to get the random block back when decrypting and hard to distinguish the random block from the rest of the output. When the final logic to recreate the keyspace runs, the resultant keyspace is padded, since the passphrase bytes and the random bytes combined together add up to less than 4k bytes.

If simple mode is specified and at least one keyfile entry is entered, all steps except the random block processing run fully. Bytes are chosen randomly from the keyfiles so that the byte sources array will contain exactly 4k bytes, and there will be no need to pad the resultant keyspace array. The full set of steps for processing keyfiles are described elsewhere. Here we just describe keyfile features as needed to explain other things.

We take two passes through the keyfiles and mix in hashes of the original keyfile entries to better protect against the passphrase or list of keyfiles being known or easy to guess. The first pass through the keyfiles depends on both the passphrase and the keyfiles. After the first pass has completed, the pseudo-random number generator state depends on the passphrase, the exact ordered list of keyfile entries, and the contents of the keyfiles. Without this first pass through the keyfiles, the sequence of random numbers that drives the second and final pass through the keyfiles would depend only on the passphrase.

What happens if both the passphrase and the set of keyfiles are known or easy to guess? Even if this happens, the hashes of the original keyfile entries provide some protection. Mixing hashes of the original list of keyfile entries with passphrase bytes and re-initializing the pseudo-random number generator prior to the first pass through the keyfiles makes this pass through the keyfiles depend on both the passphrase and the exact list of keyfile entries specified, which includes both the order of the entries and the set of entries. If this set of entries is small or there is only one entry, this does not provide very strong protection.

Add-on keyfile entries are keyfile entries that serve mainly to address this weakness. They are not treated as keyfiles. One way of thinking of them is that they are additional passwords used specifically to protect against the set of real keyfiles becoming known. For example, if your password is "Password1" and you use the default set of keyfiles by just typing a period character (.) followed by a blank line when prompted for keyfile entries, it would seem that an attacker could decrypt your file after a while by combining standard password guessing software with the assumption that you used the default set of keyfiles. But if before or after typing a line containing a period character (.), you also type a line containing 'x::peachy!', you just made the attacker's problem much harder. There is no obvious way to know that you included an add-on keyfile entry, to know

where you put it relative to the real keyfile entry, or to know the exact contents of the add-on keyfile entry.

One could argue that with so many possible (though optional) protections built in to the first pass through the keyfiles, there is no real need to mix an additional hash into the keyspace and re-initialize the pseudo-random number prior to the second pass through the keyfiles. Possibly that's true, but a little extra security doesn't hurt.

The last of the four main use cases is running v1pad in normal mode with at least one keyfile entry. This causes all steps in the outline to run.

There is an issue with protecting the random block. Without any keyfiles at all, the state of the pseudo-random number generator at the time the random block is processed depends only on the passphrase, which means a known or easy-to-guess passphrase completely breaks the security. With keyfiles, the first pass through the keyfiles makes the set of random numbers used to encrypt the random block and pick its location in the output file depend both on the passphrase and the set of keyfile entries. As was discussed above, even the set of keyfiles becoming known may not be a fatal flaw, though to protect against that, you need to include at least one add-on keyfile entry. Because the block of really-random numbers is included into the mix, the final pass through the keyfiles is even harder to predict, which is good, and the final keyspace should be different every time, even if the exact same password and set of keyfile entries (including their order and any add-on entries) is used for encrypting more than one file.

In general, we've assumed that entering at least one keyfile entry will make sure there is at least one valid keyfile. v1pad checks to make sure this is true and fails if there is at least one keyfile entry but no candidate keyfiles. There is some benefit to providing add-on keyfile entries without any real keyfiles, but there is much more benefit to doing both. The intent of this check is both to encourage you to use real keyfiles and to warn you if you think you are using real keyfiles but aren't really. For example, if you type the name of your keyfiles directory wrong, that provides perfectly good entropy to enhance your password and protect the random block, but probably what you really intended was to use the keyfiles in your keyfiles directory to add many times more entropy.

To summarize, v1pad constructs a 4k internal keyspace from user input, and later uses that keyspace to encrypt or decrypt data files. Using keyfiles is highly-recommended because it adds significant complexity to the internal keyspace and also provides some protection against weak or known passphrases. Including at least one add-on keyfile entry when specifying the set of keyfiles is also recommended because it provides defense-in-depth in case both the passphrase and the set of keyfiles are easy to guess or become known.

Passphrase

The v1pad password is really a passphrase, even though in this document the terms password and passphrase are often used interchangeably. Space and tab characters are allowed, the maximum length is 255 characters, and multibyte characters are allowed. The passphrase must be at least eight characters long, and must contain at least one lower-case letter, one upper-case letter, and one numeric or special character.

v1pad supports up to 255 passphrase characters, which may be more than 255 passphrase bytes. If an N-character passphrase is really composed of multibyte characters, that translates to 2**N* or 4**N* passphrase bytes, depending on the platform or character set (or both). If the passphrase characters are single-byte characters (ASCII or one of its extensions), then the *N* characters are really only *N* bytes.

v1pad has special-case logic that can distinguish single-byte characters from real multibyte characters and make sure each single-byte character produces exactly one byte of input, even though it was originally represented as a two-byte or a four-byte value. However, this only works for pure single-byte character input. Any single-byte characters in a passphrase containing a mix of single-byte characters and true multibyte characters will be treated as if they were true multibyte characters, which probably means each one will be represented by an ASCII (or other single-byte character set) byte along with one or three null bytes.

In case real multibyte characters are used and the multibyte character set in question doesn't have the concepts of lower-case, upper-case, numeric, and/or special characters, you may need to disable part of the password complexity checking logic by specifying the `noMandatoryPwCharComplexity` command-line argument, or by adding an entry `MandatoryPwCharComplexity=FALSE` in the v1pad configuration file. This turns off checking to ensure at least one lower-case, one upper-case, and one numeric or special character are included in the passphrase but does not turn off checking that requires the length of the passphrase to be between 8 and 255 characters.

Passphrase Complexity

The complexity of an *N*-character minimally-compliant password is given by the formula 43*26^(*N*-1).

You would need to enter a 149-character minimally-compliant ASCII passphrase to have a passphrase complexity higher than 1.0e+210 (43*26^148 is 1.1e+211). It should take fewer multibyte characters to hit a minimum complexity of 1.0e+210. Exactly how many multibyte characters would be required depends on the input character set and the size of the internal representation of a multibyte character on the current operating system.

Setting a lower bar:

1) An 8-character minimally-compliant passphrase has a complexity of 3.4e+11
2) A 14-character minimally-compliant passphrase has a complexity of 1.1e+20, which is slightly stronger than a 64-bit key in other symmetric encryption systems (ignoring how you would populate the 64 bits)
3) A 28-character minimally-compliant password has a complexity of 6.9e+39, which is slightly stronger than a 128-bit key
4) A 55-character minimally-compliant passphrase has a complexity of 1.1e+78, which is slightly stronger than a 256-bit key
5) A 109-character minimally-compliant passphrase has a complexity of 2.8e+154 which is slightly stronger than a 512-bit key in other symmetric encryption systems

Setting a higher bar, it would take a 218-character minimally-compliant passphrase to be slightly stronger than a 1024-bit symmetric encryption system, though there are not many 1024-bit symmetric encryption systems available (this is a 128-byte key).

See also Key Indirection for an easy way to get nice, complex passphrases (but not without some setup work on your part).

Passphrase Portability

Because different platforms represent multibyte characters using different numbers of bytes and may also use different character sets, it is unlikely that a file encrypted with a multibyte-character passphrase on Linux can be decrypted using the same multibyte-character passphrase on Windows (or some other platform). If it's important to be able to decrypt a given file on a platform other than the one on which it was encrypted, the safest strategy is to only use ASCII characters in the passphrase.

Keyfiles

Keyfiles must be readable and at least 128 bytes long. To make it easier to use multiple keyfiles, v1pad allows the list of keyfiles to include directories which are read recursively. Keyfiles are an optional feature. v1pad works without keyfiles, but it provides much better security with keyfiles.

How v1pad uses keyfiles

Keyfiles provide a way to add additional entropy / complexity to an encryption key. In general, it is more effective to add additional entropy using a file than it is to add additional entropy by entering a longer password. A ten-character file name is just as easy to type and at least as easy to remember as an additional 10 password characters. The difference is that the additional 10 password characters add about 10 bytes of entropy, whereas the contents of a file may add much more than 10 bytes of entropy.

Keyfiles are used in TrueCrypt (and VeraCrypt). TrueCrypt XORs the keyfile contents with the password, and the order in which keyfiles are specified is not important. TrueCrypt also supports a keyfiles directory feature, in which all readable files in the keyfiles directory beyond a certain length are used as keyfiles.

v1pad reads keyfiles to get additional entropy / complexity and supports reading keyfiles from a directory tree. v1pad uses keyfile bytes to make its internal key longer, while TrueCrypt alters its internal key using keyfile bytes without increasing the key length.

v1pad also introduces new methods for specifying and processing keyfiles. These new methods make keyfiles significantly more powerful and at the same time easier to use.

The new keyfile-related features in v1pad are:

1) Supporting a keyfiles base directory. If a keyfiles base directory is specified, relative pathnames in keyfile entries are resolved relative to the keyfiles base directory instead of relative to the current working directory
2) Using the list of keyfiles itself as a source of entropy
3) Allowing more than just keyfiles to be specified in the keyfiles list. This includes an optional command to dynamically change the keyfiles base directory, along with other optional “commands” whose only purpose is to add complexity to the list of keyfiles
4) Reading data from the keyfiles in a way that maximizes the computational complexity required to reproduce the exact same set of file locations read
5) Reading data from the keyfiles in a way that minimizes the information that an external observer can collect

The ability to read keyfile directories recursively makes it easier to specify multiple keyfiles. The cost of using this feature is that it makes the set of keyfiles depend on the exact contents of a directory tree instead of on an explicit list. Any changes to a keyfiles directory may change how keyfiles are processed.

v1pad processes any directories present in the list of keyfile entries using a breadth-first recursive traversal. At any given recursive level, first all readable files with a specific minimum size are added to the list of candidate keyfiles, then all subdirectories are processed recursively. If at any point in time the algorithm realizes that it already added the maximum number of files to the list of candidate keyfiles, it stops. For better stability, the list of keyfile entries is first converted from relative pathnames to full pathnames, and the list of full pathnames of candidate keyfiles is sorted in alphabetical order prior to any further processing.

v1pad keyfile processing depends on the exact ordered set of keyfile entries. It also uses an alphabetical sort of the full pathnames of candidate keyfiles for better stability. This is somewhat of a conflict, since the alphabetical sort wipes out the original ordering. The conflict is resolved by first taking hashes of two distinct transformations of the original unprocessed list of keyfile entries and setting that data aside for future use. Then v1pad performs later work like expanding keyfile entries to full pathnames, recursively reading directories, sorting the list of candidate keyfiles in alphabetical order, etc.

Using the list of keyfiles itself as a source of entropy takes maximum advantage of the input data. Entropy is collected from the keyfiles list by aggregating a block of possibly transformed keyfile entry list data, securely right-sizing it to a fixed size (currently 1k bytes), and finally XORing that result with the last intermediate result (if any). This continues until there is no more input to process. The result is an array of bytes at least as large as the internal state of the cryptographically-secure pseudo-random number generator (CSPRNG). This array is then combined with other data to produce a modified array. Which other data is used depends on whether this is the first or second pass through the keyfiles. The modified array in turn is used to re-seed the CSPRNG. This ensures that the CSPRNG output used to read the keyfiles depends on both the other data and the exact contents of the list of keyfile entries.

Allowing more than just keyfiles to be specified in the keyfiles list makes it easy to add additional complexity. The syntax <letter>::<string> is used to specify add-on information. <string> can be pretty much any sequence of printable single-byte characters, or a null string. Most of the add-on information is just merged into the keyfile entry list checksums.

The special-case add-on command `b::`<path> or `B::`<path> specifies or changes the base directory used to compute paths for keyfiles (the keyfiles base directory). This is primarily a usability feature, but it also adds additional entropy to the system.

It is valid for <path> to be an empty string. Specifying an empty string clears the keyfiles base directory so that relative pathnames entered after the `B::` add-on keyfile

entry in the initial set of keyfile entries are expanded to full pathnames using the current working directory instead of using the keyfiles base directory. If <path> is not empty, it must be a valid directory that can be read by the user running the program.

If <path> is a relative pathname, it is evaluated relative to the current keyfiles base directory setting unless the keyfiles base directory is not set. If the keyfiles base directory is not set and <path> is a relative pathname, it is evaluated relative to the current working directory.

The computational complexity required to duplicate the exact sequence of bytes read from the keyfiles is maximized as follows:

1. Start by sorting the list of candidate keyfiles alphabetically by full pathname
2. Randomly select up to 128 of the candidate keyfiles to be actual keyfiles
3. Randomly sort the list of actual keyfiles
4. Treat the randomly-sorted keyfiles list as a single virtual file. Select bytes at random locations in the virtual file until the desired number of keyfile bytes have been selected. Map each selected byte in the virtual file to the corresponding byte in an underlying keyfile, and track this information for future use

The keyfile processing information that an external observer can get is minimized as follows:

1) Scramble the list of non-keyfiles, treat them as a single virtual file, choose bytes at random locations in the virtual file, and map the virtual locations back to the underlying non-keyfiles
    a. The only real difference between keyfile and non-keyfile processing is that for non-keyfiles, the exact location selected inside the non-keyfile is not tracked: only the number of times a location inside each non-keyfile was selected is tracked
2) Make it even harder to distinguish real keyfiles from non-keyfiles by reading a proportional number of bytes from each set of files. To compute the proportional number for the non-keyfiles, compute the ratio (non-keyfiles virtual concatenated file size) / (keyfiles virtual concatenated file size), and multiply that by the number of bytes to read from keyfiles.
3) Make it harder for an external observer to know exactly what bytes are used to add entropy and in what order they are used by reading bytes from each keyfile and from each non-keyfile as follows:
    a. Read both keyfiles and non-keyfiles in the original alphabetical order as if they were a single group of files instead of two groups
    b. Read bytes from both the keyfiles and the non-keyfiles. The keyfile bytes are used, and the non-keyfile bytes are discarded
    c. Read individual bytes from each file at the specified set of locations within each file in order from the first location to the last location.

        i. This is a more efficient way to read the file bytes. It also optimizes for the use case where the exact same location within a given file is selected more than once. This use case should be relatively rare if the sum of the sizes of all keyfiles is greater than 4k bytes. It will happen more frequently if the sum of the sizes of all keyfiles is less than 4k bytes
    d. Internally re-map the bytes found at a given location in each file with the original ordered set of file locations so that the end result of reading each keyfile is the set of bytes found at the randomly-selected locations in the exact same order as they were originally selected
4) Store the bytes read from the keyfiles in the keyfiles byte sources.

Complexity introduced by keyfiles

If keyfiles are used, the number of byes read from the keyfiles is 4k minus (number of passphrase bytes) minus (number of random bytes). Assuming an 8-character ASCII password and simple mode, 4096 minus 8 bytes (4088) would be read from the keyfiles. Assuming a 255-character multibyte character password and normal mode, 4096 minus (255*2 bytes on Windows or 255*4 bytes on Linux) minus 255 bytes (a total of 3331 bytes on Windows or 2821 bytes on Linux) would be read from keyfiles. Everything else would fall somewhere in between these two extremes. The maximum complexity that can be stored in N bytes is 256^N, so the maximum complexity that can be added by keyfiles falls somewhere between 256^4088 (7.7e+9844) and 256^3331 (7.0e+8021) for Windows or 256^2821 (4.4e+6793) for Linux.

Whether a specific set of keyfiles provides the maximum theoretical entropy depends mostly on the contents and sizes of the keyfiles. If the keyfiles contain a random set of byte values, all 256 possible byte values should be present, so the full complexity should be possible. An exception would be if the total size of all keyfiles is small. For example, if there is only a single 128-byte keyfile, it's impossible for all 256 possible byte values to be represented, so the best-case complexity would be somewhere around 128^N instead of 256^N.

The same limitations apply to the contents of the keyfiles. If the keyfiles only contain ASCII text, the theoretical maximum complexity is reduced to 128^N. Taking more extreme use cases: if the keyfiles only contain numeric digits, the complexity is reduced to 10^N; if the keyfiles only contain two distinct values, the complexity is reduced to 2^N; and if the keyfiles only contain many copies of a single byte value, the complexity is reduced to 1.

The rest of this section assumes the keyfiles contain binary data with a relatively uniform distribution of the 256 possible byte values. In real life, the keyfiles will probably contain less ideal data.

The complexity of reading through the keyfiles may be smaller or larger than the complexity that can be stored in the number of bytes taken from the keyfiles. In the cases

where it is larger, the best-case effective complexity is limited to the maximum complexity that can be stored in the number of bytes taken from the keyfiles (256^N). In the cases where it is smaller, the best-case effective complexity is limited to the number of possible traversals through the set of keyfiles.

For example, in v1pad normal mode with a 16-character ASCII passphrase, 4096 minus 16 minus 255 (3825) of the keyspace bytes come from the keyfiles, with a maximum storage complexity of 256^3825 (3.3e+9211). If the total number of keyfile bytes is 4k, the complexity of randomly selecting 3825 bytes from the 4096 total byes is 4k!/(4k minus 3825)!. 4k! is too large for the Windows calculator to handle, but we can compute the answer using logarithms. 4k!/(4k minus 3825)! is 4096!/271!. This can be written as 4096 * 4095 * 4094 * … * 272 * 271! / 271!, which simplifies to 4096 * 4095 * 4094 * … * 272. Taking the log (base 10) of this gives log(4096) + log(4095) + log(4094) + … + log(272). I wrote a Perl program to compute the sum of the base 10 logarithms of a range of positive integers and used it to calculate the above expression. The answer is 12476.3048463865. Raising 10 to this power is antilog(0.3048463865) * 10^12476, which is 2.0e+12476. For this use case, the traversal complexity is larger than the storage complexity, and the best-case effective complexity is the storage complexity, which is 3.3e+9211.

Taking the same example above, but with only a single 128-byte keyfile, v1pad will over-sample the keyfile to get the required 3825 keyfile byes. The best-case storage complexity is 128^N instead of 256^N, since at most 128 distinct byte values can be contained in a 128-byte file. Here the maximum storage complexity is 128^3825, which is 1.2e+8060. Since on the average all of the bytes in the 128-byte keyfile will be selected randomly, the best-case complexity of traversing the same keyfile multiple times will be something like 128!*3825/128, which is 1.1e+217. In practice, it would be somewhat smaller, since not all of the 128 bytes will be selected the same number of times, and it seems likely that there are some duplicate values in a 128-byte keyfile. For this case, the traversal complexity is much smaller than the storage complexity, so the best-case effective complexity is at most 1.1e+217.

Taking the same example above, but with the default keyfiles, we know that the default keyfiles are all 4k each, and v1pad will automatically pick 128 of them to be actual keyfiles, so the total size of the virtual keyfile is 128 * 4k = 512k = 524288. From above, we'll get 3825 of the keyspace bytes from the keyfiles, with a maximum storage complexity of 256^3825 (3.3e+9211). The traversal complexity with 512k keyfile bytes is given by the formula 512k!/(512k minus 3825)!, which is 524288!/(520460)!, which is (524288 * 524287 * 524286 * … * 520459 * 520460!)/ 520460!, which is (524288 * 524287 * 524286 * … * 520459). Taking the log of each entry and running the logsum program gives the sum of the base-10 logs as 21899.8640292819, which translates back to 7.3e+21899, which we see is much larger than the maximum storage complexity of 3.3e+9211, and also quite a bit larger than the traversal complexity with only 4k of total keyfile bytes (2.0e+12476). In general, once we have at least 4k of total keyfile bytes available, the traversal complexity will be much higher than the storage complexity, so there is no major advantage to using very large keyfiles.

As one more example, assume we entered a 255-character multibyte character password on Linux and are running v1pad in normal mode. This would cause only 2821 bytes to be read from the keyfiles, which is the smallest possible number larger than zero. This gives a maximum storage complexity of 256^2821 (4.4e+6793). Assuming the sum of the sizes of all keyfiles is 4k bytes, the traversal complexity is 4k! / (4096 – 2821)!, which is 4096!/1275!. Using the strategy above to compute the answer, we get 4096 * 4095 * 4094 * … * 1276 * 1275! / 1275!, which is 4096 * 4095 * 4094 * … * 1276. Taking the base-10 log of this expression gives log(4096) + log(4095) + log(4094) + … + log(1276). Using a program to compute this expression gives 9611.80953317727. The actual number is then antilog(0.80953317727) * 10^9611, which is 6.4e+9611. This number is larger than 4.4e+6793, so as in the previous example with at least 4096 keyfile bytes, the traversal complexity is larger than the storage complexity, so the best-case complexity is limited to the storage complexity, which is about 4.4e+6793.

If there are multiple distinct keyfiles, the virtual keyfile constructed by concatenating each of the keyfiles together in a pseudo-random order can be constructed in N! different ways, where N is the number of keyfiles. The range of possibilities is from 1! (1) to 128! (3.9e+215).

Using keyfiles

If you use keyfiles, they form part of your encryption key. Keyfiles make the internal v1pad encryption key much longer, so they make v1pad encryption much stronger. But everything comes with a cost. The cost of using keyfiles is mostly remembering which keyfiles you used and in what order you entered them when encrypting a given file. However, it is also important to make sure you chose your keyfiles wisely and make sure that nothing happens to your keyfiles or to the directory in which they are located.

The contents of each keyfile, the full pathname of each keyfile, the exact set of keyfile entries, and the order of keyfile entries are all significant. Any change at all to any of these keyfile attributes will for all practical purposes make it impossible to decrypt any files encrypted using that set of keyfiles.

For example, if a keyfile moves from one directory to another or is renamed, it will be in a different location in the list of full pathnames of the actual keyfiles. When v1pad uses the same password, random block, and ordered set of keyfile entries to seed the random number generator, it will of course produce the same sequence of pseudo-random numbers. However, because the initial list of alphabetically-sorted keyfile full pathnames has a different order, the resultant virtual keyfile will be constructed in a different order, so the keyspace will contain different values, and you will not be able to decrypt any files which were encrypted using that set of keyfiles before renaming or moving that particular file. It’s possible with more than 128 candidate keyfiles that randomly the file you moved or renamed is not selected as a keyfile either in the original ordering or in the altered ordering, and that moving this non-keyfile around does not impact the ordering of

the actual keyfiles. Although this is possible, especially with 512 candidate keyfiles, it would be unwise to assume this will happen.

The prior section provides information about the complexity v1pad extracts from keyfiles in different use cases. Here are some general guidelines based on that information. First of all, using any keyfiles at all adds a huge amount of complexity to the keyspace, so using keyfiles is a good idea. Second, the total size of all keyfiles should be 4k or larger. Third, it's a good idea to use keyfiles that contain a full set of all 256 possible byte values. Fourth, using multiple keyfiles adds additional complexity, which is good.

The default keyfiles that ship with v1pad meet all of these criteria. There are 512 of them, each one is 4k in size, and they contain a random set of byte values. If you're not ready to select and use your own keyfiles, using the default keyfiles is a much better option than not using any keyfiles at all.

Ideally, you'd use your own set of keyfiles. If you use your own set of keyfiles, keep the following recommendations in mind.

First, use files that are already available on your computer or on removable media. The advantage of using your own keyfiles is that no one will know what keyfiles you used. If an attacker does not know what keyfiles you used, their chances of decrypting your file are pretty close to zero, even if they somehow learn or guess your exact password. Using files that are already available for some other reason / purpose allows your keyfiles to hide in plain sight.

Second, use files that do not change over time located in a directory structure that does not change over time. If even one keyfile changes or is removed, you will not be able to decrypt files encrypted using that set of keyfiles.

Third, avoid files with limited value domains, like files containing only ASCII text. Don't worry about finding files that contain completely random values. The idea is to use files that you already have on your computer, which means that they have some purpose other than being used as keyfiles, which in turn means their contents are probably not completely random.

Fourth, use multiple keyfiles with a total size of 4k or larger. The section on keyfile complexity demonstrates that the number of permutations for almost 4k bytes taken from a pool of 4k or more bytes is very large. More complexity means harder to break. Don't settle for weaker encryption!

v1pad supports keyfiles and data files larger than 2 Gig on both Windows and on Linux. Using very large keyfiles will slow down keyfile processing and not add much additional security. From the earlier section, once the total number of available keyfile bytes hits 4k or so, the keyfile traversal complexity is already higher than the complexity that can be stored in the keyfile byte sources, so there is not much practical benefit to having a gigantic pool of keyfile bytes. Although v1pad will at most read about 4k bytes from any

single file, it still takes time to skip through the bytes that it doesn't read. With smaller files, the time to skip unused bytes is not significant. With very large files, it can take up to several seconds per file. The main reason v1pad supports very large keyfiles is so that you can use them if you choose to. Just understand that there is a performance penalty for using very large keyfiles.

Some examples of good places to get your own keyfiles include music CDs or directories containing downloaded music files, directories containing pictures or videos, and backup directories. These are good places because they already exist on your computer (or on some removable media), they are unlikely to change over time, they should mostly contain real 8-bit values (though this might not be true for some backup directories), and they tend to contain multiple files with a total size quite a bit larger than 4k.

Some examples of bad places to get your own keyfiles include temporary internet files, directories containing installed software, and any directory structure where you regularly add or remove files or directories. Although temporary internet files tend to look pretty random, they will eventually be deleted. Directories containing installed software change when the software is configured, patched, or upgraded. Any directory where you regularly add or remove files or directories is a bad candidate unless you want to type out the names of individual keyfiles every time and never re-organize the directory structure.

Implicit in this advice is the idea that you want maximum security with minimum work. That usually means you'll pick a root directory containing multiple keyfiles (directly in the root directory, in child directories, or both), then specify that directory as a single keyfile entry. Updating the keyfiles base directory setting in the v1pad config file may be more convenient but is not as secure. The v1pad config file is a plain text file and it could possibly be read by others.

You can explicitly list each possible keyfile in a directory, but it's less work to just list the directory itself and let v1pad read the directory to get the list of keyfiles. However, if you want to live dangerously and use keyfiles located in directories that change over time, you would be wise to list each keyfile individually so that the presence / absence of other files in the directory cannot change the set of keyfiles processed.

Including at least one add-on keyfile entry in addition to at least one normal keyfile entry provides best security, but it also requires more typing and requires that you remember exactly what add-on entries you used and in exactly what order you entered them. This is true of real keyfile entries as well.

Keyfile Examples

Here are a few examples of specifying keyfiles.

The examples assume a standard v1pad installation on Windows with no changes to the keyfiles base directory setting in the v1pad configuration file, and also assume that there

is a `C:\Music\Michael Card\Joy in the Journey` directory on the computer which contains mp3 files corresponding to this album.

```
<keyfile entries below here>
.
<keyfile entries above here>
```

Specify the period (.) character, which means “current working directory”.

Combining this with the default keyfiles base directory setting points to the top-level directory for the default keyfiles shipped with v1pad. v1pad recursively reads the default keyfiles directory, then randomly selects 128 of the files as keyfiles and 384 of them as non-keyfile decoys.

There are 4.5e+123 possible ways to construct an unordered set of 128 keyfiles from a pool of 512 candidate keyfiles, and there are 3.9e+215 possible orderings for each of the possible sets of 128 keyfiles, which is a total of 1.7e+339 possible ordered sets of keyfiles.

```
<keyfile entries below here>
.
m::Mary had a little lamb
<keyfile entries above here>
```

Combine the full set of default keyfiles with the add-on entry `m::Mary had a little lamb`.

Any upper-case or lower-case letter can be used before the two colon characters for the purposes of adding complexity to the list of keyfile entries. Upper-case and lower-case ‘b’ is also used to change the keyfiles base directory.

```
<keyfile entries below here>
set02\keyfileS02K002.dat
set02\keyfileS02K086.dat
set02\keyfileS02K002.dat
<keyfile entries above here>
```

List specific keyfiles using their relative pathnames. This example assumes the keyfiles base directory is set to point to the default keyfiles, so v1pad looks for these keyfiles in the `set02` subdirectory under the default keyfiles directory.

Another way to do this would be to reset the keyfiles base directory to the `set02` subdirectory under the default keyfiles directory using `b::set02`, and then just list the file names without the leading `set02\` prefix.

For example:

```
<keyfile entries below here>
B::set02
```

```
keyfileS02K002.dat
keyfileS02K086.dat
keyfileS02K002.dat
<keyfile entries above here>
```

Note that `b::set02` is a relative pathname. The current keyfiles base directory setting is taken into account when resetting the keyfiles base directory, so `set02` in this context refers to the `set02` subdirectory under the default keyfiles directory. If the keyfiles base directory had not been set, `set02` in this context would have referred to a `set02` subdirectory in the current working directory.

The first and third keyfile entries are the same. This is allowed. v1pad will process the same file multiple times if it is listed multiple times. In general, having duplicate keyfile entries is not recommended because it reduces the complexity extracted from the keyfiles.

Specifying only three keyfiles greatly reduces the number of ways the set of keyfiles can be ordered (only three ways, since two of them have the same contents), but the total size of these three files is 12k, so still there will be many possible ways bytes can be chosen from the resultant virtual keyfile. Based on examples above, the number should be so large that the effective complexity will be around 256^N, where N is 4k minus the sum of the sizes of the passphrase and random block.

```
<keyfile entries below here>
C:\Music\Michael Card\Joy in the Journey
<keyfile entries above here>
```

Specify your own keyfiles directory by entering its full pathname. Using file and directory names that contain embedded spaces works fine on Windows. Not sure what would happen on Linux.

v1pad reads your keyfiles directory recursively, builds a candidate keyfile list containing up to 512 (or so) of the readable files 128 bytes or longer, randomly selects up to 128 of them as keyfiles, and uses the remaining candidate keyfiles (if any) as non-keyfile decoys.

```
<keyfile entries below here>
b::C:\Music\Michael Card\Joy in the Journey
.
<keyfile entries above here>
```

Specify your own keyfiles directory by resetting the keyfiles base directory using the `b::` add-on keyfile entry, then specify a period character (,) as a real keyfile entry. Since the period character (.) means “current working directory” and the keyfiles base directory is set, the period character will evaluate to the top level of your own keyfiles directory.

Changes to the keyfiles base directory take place immediately while processing the list of keyfile entries. If the first entry had been a period character (.), and the second entry had

been `b::C:\Music\Michael Card\Joy in the Journey`, the effect would have been to use the default set of keyfiles, since the period character (.) at that point would refer to the top level of the default keyfiles base directory. The second line would reset the keyfiles base directory, but that doesn't really matter because there would not be any real keyfile entries listed after it.

```
<keyfile entries below here>
C:\Music\Michael Card\Joy in the Journey
a::That would be inconceivable
<keyfile entries above here>
```

Specify your own keyfiles directory, and also include an add-on entry which just adds complexity to the list of keyfile entries.

```
<keyfile entries below here>
B::C:\Music\Michael Card\Joy in the Journey
El Shaddai.mp3
..\Joy in the Journey\Immanuel.mp3
..\..\Michael Card\Joy in the Journey\El Shaddai.mp3
<keyfile entries above here>
```

Resets the keyfiles base directory to a user-defined value, then lists three user-defined keyfiles using different ways of specifying their relative paths.

The two more complicated relative paths point to the same directory as the simpler relative path. The only real reason to specify them using the more complicated syntax is to add complexity to the list of keyfile entries.

Random Numbers

v1pad uses pseudo-random numbers and "really-random" numbers. Pseudo-random numbers are generated by a Cryptographically-Secure Pseudo-Random Number Generator (CSPRNG) based on user input. Really-random numbers are also generated using a CSPRNG, but the seed value for the CSPRNG is automatically generated in a way intended to make it difficult to reproduce later.

The ISAAC CSPRNG

v1pad uses the Isaac CSPRNG, which is a public-domain CSPRNG. It originally used the Mersenne Twister, but later switched to Isaac because the Mersenne Twister is not cryptographically secure, and Isaac seemed fast, robust, and secure.

ISAAC has an internal state of 256 unsigned four-byte integers and allows that entire internal state to be specified at initialization time. 256 unsigned four-byte integers form an effective key size of 1024 bytes, which is able to represent 256^1024 (1.1e+2466) unique initial states. According to the wiki where I got the ISAAC code (http:// http://burtleburtle.net/bob/rand/isaacafa.html), ISAAC has a guaranteed minimum cycle length of 2^40 (1.1e+12) and an average cycle length of 2^8295 (1.1e+2497).

The main v1pad encryption / decryption logic uses two CSPRNGs (actually two instances of the ISAAC CSPRNG): one that is intended to produce a sequence of pseudo-random numbers that depends 100% on user input, and a second one which is intended to produce a sequence of pseudo-random numbers that is for all practical purposes truly random.

Really-Random Number Generation

This section gives an explanation of how the "really-random" number generator is seeded, since the security of v1pad depends in part on how close the output of the "really-random" CSPRNG is to true random numbers. We'll drop the quotes around "really-random" starting here, and let the description below speak for itself.

The v1pad really-random number generator produces a statistically-random sequence of numbers that cannot be easily reproduced or compromised. The basic strategy is to take a decent cryptographically-secure pseudo-random number generator, seed it with a globally-unique value, and count on the combination of a good CSPRNG and a globally-unique seed value to produce a unique sequence of pseudo-random numbers. The difficult part is to generate a good globally-unique value. Probably any good CSPRNG would work fine if it is initialized with an appropriate seed value.

The globally-unique value is generated as follows: query the operating system for computer identification information and system state information, then add to this some data about the currently-running program. The purpose of the computer identification information is to uniquely identify the computer on which the random number generation software is running. The purpose of the system state information is to uniquely identify the current high-level state of the computer. The purpose of the data about the currently-running program is to handle the case where multiple programs could be generating random numbers at the same time on the same computer. Taken together, these three sets of information form a globally-unique identifier which is difficult to reproduce later.

Some examples of computer identification information are: 1) the motherboard manufacturer, model, and serial number; and 2) the model, serial number, and firmware version for all disk drives. Either one of these two values should be pretty close to globally unique all on its own. For that to not be true, motherboard manufacturers would have to issue duplicate serial numbers for a given motherboard model, and disk drive manufacturers would have to do the same thing. Neither of these is likely.

Some examples of system state information are: 1) Current system time, free physical memory, and free virtual memory; and 2) Process Name, Process ID, User CPU time, System CPU time, virtual memory in-use, and total number of page faults so far. The current system time is clearly unique on a single computer. Ignoring that for now (or assuming it can be easily reproduced at a later time), the other attributes are worth consideration. The free physical and virtual memory will cycle up and down over time, so it seems like specific values here will narrow things down, but not strongly. The state of the current set of running processes seems more unique. Although the same set of processes and maybe even the same set of processes and process IDs is likely to repeat over time for the processes that start automatically when the computer boots up, the memory and CPU stats seem more likely to be unique. Because most computers are connected to the internet and their startup processes do things like synchronize the system time, look for software updates, download new antivirus data files, etc., and because all processes on the computer contend for the same set of CPU and virtual memory resources, it's likely that a given process at time boot + X after one reboot will have slightly different CPU and memory stats than it had at time boot + X after an earlier or after a later reboot. Is this set of information guaranteed to uniquely identify a point in time on the current computer? Maybe not, but it could come close, and it does provide additional input data that would be much harder for an attacker to duplicate later on than just the current system time.

Some examples of information about the current running program include current timestamp, process ID, user and system CPU times, and process startup time. In the context of a program running at a specific point in time on a specific computer, the process ID alone is sufficient to uniquely identify the program. The other attributes just make it harder for someone else to later exactly reproduce the exact same state information.

The above information is used to initialize a CSPRNG as follows:

1) For each of the three sets of information above (computer ID info, computer state info, and program run info):
    a. Write a single concatenated string containing all of the information in this category separated by space characters
    b. Fill a 4k array with secure hashes based on the concatenated string. The routine that converts a string to a 4k array also computes an offset value
2) Compute merged 4k buffers and a merged offset by XORing the three 4k checksums buffers and XORing the three offset values
3) Fill an array to initialize the CSPRNG by taking N bytes from `merged_array[(index+offset) mod 4k]` and formatting them properly. N in this case is the standard size of the array used to initialize the CSPRNG, measured in bytes.
4) Initialize the CSPRNG using the array prepared above
5) Get random numbers from this CSPRNG

The CSPRNG output with this seed is for all practical purposes truly random, since the output is statistically random and it cannot easily be reproduced at a later time.

The Random Block

By default, a block of really-random bytes is included in the keyspace so that the virtual one-time-pad produced when running v1pad is different every time, even if the same password and keyfiles are used to encrypt more than one file. The block of really-random bytes also adds complexity to the keyspace. To make decryption possible, the exact same block of really-random bytes is encrypted and saved to a pseudo-random location in the output file.

The v1pad random block can be considered an enhanced version of the classic password salting strategy. To understand that, first we'll review password salting, then we'll come back to the v1pad random block.

Passwords are often used to authenticate a user. For this use case, usually the software stores the output of a secure hash of the password instead of the password itself. Attackers have learned that an easy way to break this system is to pre-compute a set of secure hashes for common passwords and check to see if the actual output value matches the pre-computed value.

To avoid this attack, defenders have taken to adding a random value known as a salt to the original input password before running the secure hash algorithm. This makes it harder to attack by pre-computing output values because for N bytes of random data, any given password mixed with the N random bytes produces 256^N output values (ignoring hash collisions), not one of which is likely to match the output of running the algorithm on the original password by itself. The salt value is normally stored in unencrypted format along with the hash of the password.

The v1pad random block is similar to the classic salting strategy, but not identical. The differences are:

1) Mixing the additional random data with the original password to form a longer and stronger composite key for the purposes of encrypting plain text input data, not for the purposes of making the original password harder to reverse-engineer
2) Encapsulating the additional random data bytes in a single fixed-size block called the random block
3) Hiding the random block in the encrypted output instead of putting it in a known location
4) Encrypting the random block so that it's difficult for an attacker to recognize and recover the random block
5) Padding the last block of encrypted output to an even multiple of the random block size to simplify input/output
6) Performing various operations in chunks that are even multiples of the random block size
    a. The input/output block size must be identical to the random block size. Otherwise input/output would be very complicated
    b. The encryption block size must be an integral multiple of the input/output block size, so that encrypted blocks can be written and read back in chunks of the input/output block size

Logically, v1pad uses several different block sizes: the random block size, the input/output block size, the preferred encryption block size, the minimum encryption block size, and the size of the final block of encrypted output. For simplicity, all block sizes are powers of two. The random block size, the I/O block size, and the minimum encryption block size are all identical. The preferred encryption block size is 4k. The size of the final block of encrypted output is equal to the size of the final block of padded input, which is some multiple of the random block size. Mathematically, 4k >= final block size >= random block size.

If the padded input length is less than 4k, it is encrypted as a single block, then written to the output in chunks equal to the random block size. If the padded input length is greater than or equal to 4k, first some number of 4k blocks of input are encrypted and written to the output in chunks equal to the random block size, then the remaining input bytes are encrypted as a single block and written to the output in chunks equal to the random block size. The random block itself is written to the output somewhere among the other output blocks.

Random Block handling during encryption

This is implemented as follows. Here we assume the random block processing happens in the context described above in the overview section, so the pseudo-random number generator and the really-random number generator have already been initialized.

1) Pick a size for the random block that gives good collision resistance for the set of random bytes and provides strong encryption
2) Determine the total output length, which for this example will be the length of the input plain text padded to the next-highest block. Track this in terms of the number of output blocks, and add one to the total to account for the random block
3) Get a full block of random bytes from the really-random number generator
4) Use a pseudo-random number to determine where in the random block the padding length will be stored and store the padding length in the random block along with the random bytes. The random byte that is displaced by the padding length is used elsewhere
5) Use a pseudo-random number mod number of output blocks to determine where in the output the random block will be written
6) Encrypt a copy of the random block
7) When writing the encrypted plain text to the output file as part of the actual encryption process, write it chunks equal to the random block size. Once the chosen location for the random block comes up, first write the encrypted copy of the random block to the output, and then write the block of encrypted plain text
8) After successfully encrypting all of the plain text, if the randomly-chosen location for the random block happens to be at the very end of the output, write the random block to the very end of the output

Random Block handling during decryption

1) Use the same encryption algorithm and block sizes as was used for encryption
2) Initialize the pseudo-random number generator the exact same way as was done for encryption
3) Get a pseudo-random number mod number of blocks in the input (to decrypt). This is the location of the random block in the input
4) Read only the random block from the encrypted file and decrypt it
5) Get the random bytes and number of padding bytes from the decrypted random block
6) Mix the random bytes into the keyspace the exact same way as implemented for encryption
7) Read encrypted data from the input file one block at a time. When the chosen location for the random block comes up, skip over the random block.
8) If the random block happens to be at the very end of the input, just ignore the random block.

The Random Block Size

As was mentioned earlier, we want a random block size that gives good collision resistance for the set of random bytes and provides strong encryption. v1pad uses a 256-byte random block size because it is the minimum block size that meets these requirements.

A 256-byte random block size implies a 256-byte input/output block size. Somewhere between zero and 255 additional bytes will be needed to pad the last input block up to an even multiple of 256 bytes. This value fits easily into a single byte, so we can reserve just one byte in the random block to track the number of padding bytes and use the other 255 to contain the random values that are mixed into the keyspace.

A 256-byte random block gives good collision resistance. According to the "birthday problem", a block of 255 random bytes will require at least 256^(255/2) attempts in order to have better than a 50% chance of generating a collision. 255/2 is 127.5. Rounding down to 127, we get 256^127, which is 7.0e+305. 7.0e+305 is large enough to be computationally-infeasible to reverse-engineer, so this looks like a safe block size. If v1pad were to use a 128-byte block size, the formula for checking collision resistance would give 256^(127/2), which is 256^63, which is 5.2e+151. That doesn't quite meet our standards for computationally infeasible.

A 256-byte random block size adds significant complexity to the keyspace. Since the bytes in the random block may contain a full range of the possible values from zero to 255, and 255 of the bytes in the random block are added to the keyspace, the complexity that the random block adds to the keyspace is 256^255, which is 1.3e+614.

A 256-byte encryption block size is large enough to provide strong encryption with BLS. The best-case complexity for BLS with a 256-byte block size is 5.7e+614. Even with a very skewed distribution of zero and one bits so that on the average there is only a single one-bit per byte, BLS with a 256-byte block size will produce 3.5e+333 possible combinations, so this is a safe block size for BLS. BLS with a 128-byte block size has a best-case complexity of 4.5e+306. With only a single one-bit per byte, BLS with a 128-byte block size has an output complexity of 1.4e+166. We see that for a 128-byte block size, the best-case complexity meets our goal for computational infeasibility (complexity >= 1.0e+210), but the complexity with a very skewed distribution of zero and one bits does not.

See the section on how v1pad encrypts / decrypts files for a full description of BLS.

A 256-byte encryption block size is also large enough to provide a "strong" set of one-time-pad bytes. In a sense, this doesn't matter, since if we get everything else correct, the one-time-pad encryption provides "perfect secrecy". Even if someone could generate all possible pads with a specified length, all they'll get for their trouble when XORing each pad with the encrypted text is all possible plain texts with that same length. Having said that, the best-case complexity of a 256-byte block of one-time-pad bytes is 256^256, which is 3.2e+616. Not all pad generation schemes produce best-case complexity, but all of the currently supported pad generation schemes provide a complexity of at least 256!, which is 8.6e+506.

How v1pad encrypts files

The core of v1pad is one-time-pad encryption. During encryption, the one-time-pad is constructed dynamically, combined with the plain text to produce encrypted text, and forgotten. During decryption, the one-time-pad is constructed dynamically, combined with the encrypted text to produce plain text, and forgotten.

To make v1pad resistant to "plain text" attacks, by default the plain text is encrypted using some method other than one-time-pad encryption before it is combined with the virtual pad. In a sense, it is doubly-encrypted. The first encryption is with a method other than one-time-pad encryption, and the second encryption is with the virtual pad. During decryption, the data has to be doubly-decrypted: first it is decrypted using the virtual pad, and then it is decrypted using the other method.
.
Because v1pad encryption is complex, the explanation of how it works is broken into parts. The first part explains the basics of one-time-pad encryption using a virtual pad along with the currently-implemented pad generation algorithms. The second part explains the full algorithm, including how the plain text is encrypted. Before looking at either of these, we'll explain the admin bytes.

The Admin Bytes

By default, v1pad includes four "admin bytes" at the very start of all encrypted files. These admin bytes track the mode (simple or normal), pad generation algorithm, plain text encryption algorithm, plain text re-key frequency, and the format of the admin bytes themselves. Four bytes is more than big enough to track all of this information. Using four byes instead of some smaller number allows for new pad generation algorithms and plain text encryption methods to be added without changing the format of the admin bytes, plus includes room for possible additional unrelated information.

It is possible to tell v1pad to not store the admin bytes at the beginning of each encrypted file by setting the `SaveAdminBytes` command-line argument to FALSE (`noSaveAdminBytes`). This makes the encrypted files four bytes smaller and is arguably more secure, since then the encrypted file is just a blob of binary data with no information at all about how it was produced. The default is for v1pad to write the admin bytes. This is for convenience and because people have a limited ability to remember things. It's hard enough to remember the passphrase and keyfile entries. Remembering all of the encryption parameters and the `SaveAdminBytes` command-line argument setting along with the passphrase and keyfile entries is probably too much for most people, especially if months or years have passed since the file was encrypted.

If the `SaveAdminBytes` command-line argument is set to its default value of TRUE (`SaveAdminBytes`), the admin bytes are written to the start of each encrypted file. When

decrypting, the admin bytes are read to determine which encryption parameters were used to encrypt the data. Then decryption proceeds using the encryption parameters recorded in the admin bytes. The corresponding encryption parameters on the v1pad command line (or in the v1pad configuration file) are ignored.

If the `SaveAdminBytes` command-line argument is set to FALSE (`noSaveAdminBytes`), the admin bytes are not written to the start of each encrypted file. When decrypting, the admin bytes are not read, and the encryption parameters specified on the v1pad command line (or in the v1pad configuration file) are used for decryption.

Decryption won't work properly if the `SaveAdminBytes` command-line argument has different values during encryption and decryption. If decryption expects admin bytes but they are missing, it will either fail or produce incorrect results (normally incorrect results look like random binary data). If decryption does not expect admin bytes but they exist, it again will either fail or produce incorrect results. Whether decryption with a mismatched `SaveAdminBytes` setting fails or just produces incorrect results depends on many factors.

A simplified version of v1pad encryption

It is difficult in classic one-time pad implementations to endure that the pad is long, random, unique, and secret. Implementing one-time pad encryption using a virtual pad generated from the output of a Cryptographically-Secure Pseudo-Random Number Generator (CSPRNG) compresses all of this down to making sure that the input value used to seed the CSPRNG is unique and secret.

Unless the plain text is incredibly long, the output sequence of a CSPRNG is easily long enough. The CSPRNG output is already statistically random if the chosen CSPRNG is any good.

The input value used for most encryption algorithms is just a password. If that password is used to initialize the CSPRNG, and the password is sufficiently complex and never reused, the pad generated from the CSPRNG will be both unique and secret.

Ideally the input value would be at least long enough to fully-seed the CSPRNG, but this is difficult if the input value is just a password or a passphrase. To work around this, the password or passphrase is right-sized to 4k bytes using secure hashes derived from the input password. It is deliberately right-sized to a value more than long enough to fully-seed the CSPRNG. This right-sized value is called the keyspace. The process of right-sizing the user input to a keyspace does not add or remove entropy from the user input, but it does convert it to a format that's more convenient to work with.

As described in other parts of this manual, the keyspace normally also contains a block of really-random bytes and may also contain many bytes loaded from keyfiles. Keep in mind that this is the simplified explanation. Many details are ignored to focus on the basics of one-time-pad encryption using a virtual pad.

Once the user input has been converted into a keyspace, the CSPRNG is seeded from the keyspace. Then an algorithm is chosen for converting the CSPRNG output to padding bytes.

Seeding the CSPRNG from the keyspace is relatively simple. If the CSPRNG initialization array is an array of N unsigned four-byte integers, 4*N bytes are required to completely initialize the array. The right-sizing algorithm produces a 4k array and a four-byte unsigned offset. The 4*N bytes to fill in the CSPRNG initialization array are generated by reading 4*N bytes from `keyspace[(index+offset) mod 4k]`, converting every group of 4 bytes to a single 4-byte unsigned integer, and writing that integer to the CSPRNG initialization array. Then the CSPRNG initialization routine is called with the CSPRNG initialization array passed in as input.

If the CSPRNG initialization array is longer than 4k, the index for collecting the bytes which will fill up the initialization array can have values beyond 4k. The reference into the keyspace (`keyspace[(index + offset) mod 4k]`) is modular anyway, so the worst thing that happens is the resultant CSPRNG initialization array contains less entropy than it is capable of containing.

The one-time pad generation algorithms are all different, but each generates pad bytes using some internal block size. If the number of requested one-time pad bytes is not an even multiple of the internal block size, each algorithm generates additional one-time pad bytes by rounding the number of requested one-time pad bytes up to the next even multiple of the internal block size, but only returns the requested number of one-time pad bytes. This actually happens in simple mode, but not in normal mode. In normal mode, the input is padded to always be a multiple of 256 bytes. All of the internal block sizes used by the pad generation routines can be multiplied by some integer to create a 256-byte block of one-time-pad, so none of the routines has to generate additional one-time pad bytes in normal mode.

There are many possible ways to generate the virtual one-time pad. v1pad currently only implements four of them, plus the null method. The command-line argument used to specify the one-time-pad generation algorithm is `OneTimePadAlgorithm`. The valid values for `OneTimePadAlgorithm` are `PadNone`, `PadSimpleDirect`, `PadSimpleScrambled`, `PadUniform`, and `PadKeyspace`.

`PadNone` tells v1pad not to perform one-time pad encryption (only perform plain text encryption). `PadSimpleDirect` tells v1pad to generate one-time pad by directly using the CSPRNG output. `PadSimpleScrambled` tells v1pad to generate one-time pad by using a scrambled version of the CSPRNG output. `PadUniform` tells v1pad to generate one-time pad consisting of uniformly-distributed but randomly ordered byte values. `PadKeyspace` tells v1pad to generate one-time pad by combining CSPRNG output with the current contents of the keyspace. The default value is `PadKeyspace`, which is probably the hardest to reverse-engineer.

Now we'll expand on the details of each of these pad generation methods.

PadNone: Omit one-time Pad

`PadNone` skips generating one-time pad and so does not implement one-time-pad encryption. It is useful for just encrypting files using a plain text transformation.

Files encrypted with `PadNone` will by default include leading admin bytes, include a (fully-encrypted) random block, and be padded to an even multiple of 256 bytes. To just obtain output encrypted using a plain text transformation with none of the other information included, you would have to use simple mode and tell v1pad to omit the admin bytes.

It is valid to specify both `OneTimePadAlgorithm=PadNone` and `PlainTextAlgorithm=PlainNull`. What this does is output the original plain text, possibly with leading admin bytes, a (fully-encrypted) random block inserted somewhere, and the final block of plain text padded to an even multiple of 256 bytes. This is probably only useful for performance testing.

PadSimpleDirect: Pad with direct CSPRNG output

Here is how `PadSimpleDirect` pad generation works.

1) Figure out how many unsigned four-byte random integers are required to generate N bytes of pseudo-random output. It is N/4 if N is an even multiple of 4. It is N/4 + 1 if N is not an even multiple of 4. Call this Random Count
2) Loop Random Count times
    a. Get a random number
    b. Copy the four bytes in the random number to the next four positions in the output array of unsigned bytes, using logical operations to select the bytes

The fastest way to implement this would be to fill an array of unsigned four-byte numbers with Random Count pseudo-random numbers, then use memcpy() to write it directly to the corresponding array of unsigned bytes. v1pad uses the above algorithm instead so that the resultant array of one-time-pad bytes is the same regardless of whether the computer v1pad is running on is big-endian or little-endian.

`PadSimpleDirect` pad generation produces a block of pad bytes with a theoretical maximum complexity of 265^N, where N is the number of output bytes. For our current minimum block size of 256 bytes, that is 256^256, or 3.2e+616.

`PadSimpleDirect` pad generation uses N/4 or (N/4) + 1 four-byte random numbers to produce N bytes of one-time pad.

PadSimpleScrambled: Pad with scrambled CSPRING output

Here is how `PadSimpleScrambled` pad generation works.

1) Figure out how many unsigned four-byte random integers are required to generate N bytes of pseudo-random output. It is N/4 if N is an even multiple of 4. It is N/4 + 1 if N is not an even multiple of 4. Call this Random Count
2) Loop Random Count times
    a. Get a random number
    b. Use the four bytes in the random number to produce four new output bytes. At the very highest level, this is:
        i. Loop through the four bytes in the random number
        ii. For each byte, use the other three bytes to transform it
        iii. Output the four transformed bytes in a pseudo-random order

`PadSimpleScrambled` pad generation produces a block of pad bytes with a theoretical maximum complexity of 265^N, where N is the number of output bytes. For our current minimum block size of 256 bytes, that is 256^256, or 3.2e+616.

`PadSimpleScrambled` pad generation uses N/4 or (N/4) +1 four-byte random numbers to produce N bytes of one-time pad.

PadUniform:Pad with Uniformly-distributed bytes

Here is how `PadUniform` pad generation works.

1) Let the number of desired padding bytes be B. Round B up to the next highest multiple of 256. Don't change if B is already an integral multiple of 256. Call the rounded-up number R
2) Fill an array of R bytes with values from 0 to 255, repeating the sequence as many times as required to fill up the array. Call the array Data
3) Scramble Data at the byte level using a variation of the standard Fisher-Yates / Durstenfeld algorithm
4) Return the first B bytes of Data

`PadUniform` pad generation produces a block of pad bytes with a theoretical maximum complexity of N!, where N is the number of output bytes and N <= 256. After N = 256, it would be N! minus some amount for the duplicate permutations (I was not able to come up with a good formula for this). For our current minimum block size of 256 bytes, N! is 256!, which is 8.6e+506.

`PadUniform` pad generation uses N - 1 four-byte random numbers to produce N bytes of one-time pad if N is an even multiple of 256. It uses up to N + 254 four-byte random numbers to produce N byes of one-time pad if N is not an even multiple of 256.

Because the one-time pad generated by `PadUniform` contains at most N/256 copies of any given byte value, but the one-time pad generated by `PadSimpleDirect`, `PadSimpleScrambled`, and `PadKeyspace` encryption may contain any number of copies of each given byte value, the output of `PadUniform` is always less complex than the output of the other three pad generation algorithms. However, even with the minimum block size of 256 bytes, all four pad generation algorithms produce output that is complex enough so that it cannot be duplicated by brute force computation.

PadKeyspace: Pad with Keyspace and CSPRNG

Here is how `PadKeyspace` pad generation works.

1) Determine the required number of padding bytes by rounding up to the next highest multiple of 128 bytes. Call this value Required Size
2) Loop Required Size times
    a. Get an unsigned four-byte random number
    b. XOR the first and last bytes of the random number. Call this value Random Byte
    c. Take the 12 lowest-order bits from the middle two bytes of the random number as an offset into the keyspace. Call this value Random Offset
    d. Take the highest-order bit from the middle two bytes of the random number and call this value Offset Shift
    e. Take the remaining three unused bits of the random number, treat them as a 3-bit number, and call this value Byte Shift
    f. Circular-left-shift Random Offset by Offset Shift
    g. Circular-left-shift Random Byte by Byte Shift
    h. Set this padding byte to Random Byte XOR `keyspace[RandomOffset]`

`PadKeyspace` pad generation produces a block of pad bytes with a theoretical maximum complexity of 265^N, where N is the number of output bytes. For our current minimum block size of 256 bytes, that is 256^256, or 3.2e+616.

`PadKeyspace` pad generation uses N four-byte random numbers to produce N bytes of one-time pad if N is an even multiple of 128. It uses up to N+127 random numbers to produce N bytes of one-time pad if N is not an even multiple of 128.

The full v1pad encryption algorithm

The full v1pad encryption algorithm combines the simplified one-time pad encryption algorithm described above with some form of encryption of the plain text. Why should we do this, given that one-time-pad encryption provides “perfect secrecy”?

As strong as one-time pad encryption is, it has at least three weaknesses which were described in the concepts section.

1) A plain text attack that is only useful if pads are re-used
2) A "man in the middle" plain text attack
3) A limited heuristic attack based on the message length

v1pad addresses the plain text attack for re-used pads in three ways:

1) It makes sure the chances of a pad being re-used are minimal by including a block of random bytes in the composite encryption key
2) It removes the plain text from the encrypted output by replacing it with encrypted plain text, so that even if the pad is re-used, an attacker would still have to break the plain text encryption before they could recover the pad bytes
    a. For BLS plain text encryption, there is a special case in which an entire block of plain text is composed only of zero bits or only of one bits. For this special case, BLS is no help at all, and the attacker would be able to recover the pad bytes. This use case should not occur frequently, but it is a weakness, so it's a very good idea to always run v1pad in normal mode. This ensures that random bytes are included in the composite encryption key, which for all practical purposes ensures that the pad is never reused
    b. If the pad is reused, that means the v1pad internal key is reused, which implies that PVS encryption can possibly be reverse-engineered using a batch chosen plain text attack. However, in this context we're talking about output that is first encrypted with PVS and then XORed with one-time-pad bytes. The batch chosen plain text attack on PVS can be run against this output. However, since the PVS output is "polluted" with the one-time-pad bytes, it does not seem like the reverse-engineered PVS state would be useful for encrypting or decrypting anything. At best, maybe the reverse-engineered PVS state can exactly reproduce the same set of output bytes (PVS output + one-time-pad) given the same known plaintext. It's unlikely it can produce valid v1pad output for any other plaintext or decrypt any other real plaintext from valid v1pad output (all of this assuming the whole time we're using the exact same v1pad internal key).
3) Depending on the location of the random block in the output, v1pad may also displace the known plain text by 256 bytes. This makes it harder to know where the plain text even is in the encrypted output. Possibly this means a plain text attack would make it easier to find the random block, but it certainly would not make it any easier to decrypt the random block

v1pad addresses the "man in the middle" plain text attack with the same features mentioned above: the inclusion of random bytes in the encryption key, encrypting the plain text, and the randomly-located random block.

For the special case of BLS encryption where a block of the original plain text consists of only zero bits or only one bits, the attacker can surely recover the corresponding block of

pad bytes, if he/she can figure out where they are. However, there is no obvious way they can take their replacement plain text and insert it into the middle of the encrypted message. To do that, they would need to be able to first encrypt their replacement plain text in exactly the same way BLS would have encrypted it if it were present in the original message. To encrypt it properly, they would need to be able to reproduce the internal state of the encryption algorithm at the time BLS was run on the block of only zero bits or only one bits (resulting effectively in a no-op, but not without any work being done). There is no obvious way for the attacker to do this. The best they could reasonably do is insert random junk where the original block of all zero bits or all one bits was. But they can already insert random junk without knowing the pad bytes.

PVS is a weaker encryption algorithm than BLS in general, but it doesn't have BLS's weakness with blocks of all zero and all one bits. Combining PVS output with one-time-pad bytes would appear to make even the known batch chosen plaintext attack on PVS useless. This means there is no obvious way for an attacker to distinguish the PVS output from the one-time-pad bytes. Knowing the plain text doesn't seem to make this problem any easier to solve. Given this, it appears that PVS provides very good protection against the "man in the middle" plain text attack.

v1pad partially addresses the limited heuristic attack based on the message length by padding the message to the next highest 256-byte boundary. This is required in order to support adding the block of random bytes. For BLS, it also makes the plain text encryption stronger, since it ensures that no block of output is encrypted with a BLS block size smaller than 256 bytes.

Encrypting using the full v1pad encryption algorithm

v1pad encrypts files as follows.

1) Collect user input and generate a large internal key called a keyspace as described elsewhere in this document
2) Use the keyspace to seed a cryptographically-secure pseudo-random number generator (CSPRNG)
3) By default, write the admin bytes to the output
4) Process the file data using three nested loops, with an outer loop that iterates in chunks of up to 16 Meg, a middle loop that iterates in chunks of up to 128 k, and an inner loop that iterates in chunks of up to 4k
    a. At the top of each outer loop iteration
        i. Re-scramble the keyspace using BLS driven by CSPRNG output
        ii. Re-initialize the CSPRNG from the newly-scrambled keyspace
        iii. Reset the plain text encryption key if `PlainTextReKeyFrequency` is set to `ReKeyOuter`
    b. At the top of each middle loop iteration
        i. Reset the plain text encryption key if `PlainTextReKeyFrequency` is set to `ReKeyMiddle`

c. For each iteration of the inner loop
    i. If there is less than 4k of plain text to process, and the size of the plain text is not an even multiple of 256 bytes, pad the plain text to the next-highest 256-byte boundary
    ii. Encrypt up to 4k of plain text as a single block, which means:
        1. Reset the plain text encryption key if `PlainTextReKeyFrequency` is set to `ReKeyInner`
        2. Encrypt the block of plain text using the selected plain text encryption algorithm
        3. Generate one-time pad bytes using the selected pad generation method
        4. XOR the encrypted plaintext with the pad bytes to produce encrypted output
    iii. Write the encrypted output to the output file in 256-byte chunks, inserting the random block in its randomly-chosen location before writing the corresponding block of encrypted output
d. End inner loop
e. End middle loop
f. End outer loop

5) After encrypting all blocks of plain text and writing them to the output file, if the random block was randomly chosen to be written at the end of the file, write it at the end of the file
6) Done

The reasons we scramble the keyspace and re-initialize the CSPRNG from the keyspace for every 16 megabytes of input are to guard against short CSPRNG cycles, to protect against attacks where somehow the current CSPRNG internal state becomes known, and just to introduce added complexity into the algorithm.

Resetting the CSPRNG periodically is intended to avoid issues with the CSPRNG cycle length. ISAAC has a minimum cycle length of 2^40, which is 1.2 e+12. 16 megabytes of input is around 1.7e+7 bytes. The least-efficient supported pad generation algorithm requires somewhere around this number of four-byte pseudo-random numbers to generate 16 megabytes of padding bytes.

Plain text encryption uses the most random numbers if the plain text key is reset in the inner loop, fewer if the plain text key is reset in the middle loop, and the least amount if the plain text key is reset in the outer loop. Different plain text encryption algorithms require different amounts of random numbers to reset their encryption keys. BLS requires up to 32k random numbers each time its key is reset. PVS requires about 192k random numbers each time its key is reset. AES128 only requires eight random numbers each time its key is reset. That gives a range of eight extra random numbers for AES128 with its key reset in the outer loop to 768 Meg random numbers for PVS with its key reset in the inner loop, so worst case we require about 768 Meg random numbers to encrypt or decrypt 16 Meg of input, which is about 8.1e+8 random numbers. This is less than the minimum guaranteed ISAAC cycle length of 1.2e+12. There is no real danger of

exceeding the ISAAC cycle length in a single outer loop iteration in the current implementation.

Dividing the cycle length of 1.2e+12 by the worst-case of set of required random numbers 8.1e+8 gives around 1,365. Multiplying this number by 16 Meg gives about 21.8 Gig. If we were to skip resetting the ISAAC CSPRNG state in the outer loop, we would only be safe for files up to about 21 Gig long, so the additional protection introduced by re-initializing the CSPRNG after every 16 Megabytes of input seems like a good insurance policy.

Resetting the CSPRNG also ensures that a known CSPRNG state can only compromise part of the output. This was important when v1pad used the Mersenne Twister. The Mersenne Twister is just a pseudo-random number generator (PRNG), not a cryptographically-secure pseudo-random number generator (CSPRNG), so this additional protection seemed prudent. It is probably not strictly required for ISAAC or any other true CSPRNG.

Decrypting using the full v1pad encryption algorithm

v1pad decrypts files as follows.

1) Collect user input and generate a large internal key called a keyspace as described elsewhere in this document
    a. For decryption, this includes fetching the random block from its location in the encrypted file, decrypting it, and reading the useful data from it. The data in the random block includes the block of random data to mix into the keyspace and the number of padding bytes added (if any)
2) Use the keyspace to seed a cryptographically-secure pseudo-random number generator (CSPRNG)
3) By default, read the admin bytes from the encrypted file
4) Process the file data using three nested loops, with an outer loop that iterates in chunks of up to 16 Meg, a middle loop that iterates in chunks of up to 128 k, and an inner loop that iterates in chunks of up to 4k
    a. At the top of each outer loop iteration
        i. Re-scramble the keyspace using BLS driven by CSPRNG output
        ii. Re-initialize the CSPRNG from the newly-scrambled keyspace
        iii. Reset the plain text encryption key if `PlainTextReKeyFrequency` is set to `ReKeyOuter`
    b. At the top of each middle loop iteration
        i. Reset the plain text encryption key if `PlainTextReKeyFrequency` is set to `ReKeyMiddle`
    c. For each iteration of the inner loop
        i. Decrypt up to 4k of plain text as a single block, which means:

1. Read up to 4k of encrypted text from the encrypted file in 256-byte chunks, skipping the random block if it's present between any of the blocks of real encrypted output
2. Reset the plain text encryption key if `PlainTextReKeyFrequency` is set to `ReKeyInner`
3. Generate one-time pad bytes using the selected pad generation method
4. XOR the encrypted text with the pad bytes to produce a block of encrypted plain text
5. Decrypt the block of encrypted plain text using the selected plain text encryption algorithm to produce possibly-padded plain text
    a. If the plain text was padded, discard/ignore the padding bytes after first decrypting the entire block of encrypted plain text

ii. Write the decrypted output to the output file in 256-byte chunks

d. End inner loop
e. End middle loop
f. End outer loop

5) After decrypting all blocks of encrypted text and writing them to the output file, if the random block was randomly chosen to be written at the end of the input file, just ignore it
6) Done

By default, v1pad decryption reads the admin bytes to get the original encryption parameters and ignores any command-line options that tell it to do anything different. If the admin bytes are omitted, you must specify the exact same encryption parameters when decrypting as you specified when encrypting.

For files encrypted using v1pad normal mode, v1pad decryption is currently faster than v1pad encryption. The reason is simple. v1pad decryption gets the block of really-random bytes from the encrypted output file, but v1pad encryption has to create the block of really-random bytes itself. In order to get any really-random bytes at all, the really-random number generator must be initialized by querying the Operating System, and this can take up to several seconds.

Plain Text Encryption

To make v1pad resistant to "plain text" attacks, by default the plain text is encrypted using some method other than one-time-pad encryption before it is combined with the virtual pad. In a sense, it is doubly-encrypted. The first encryption is with a method other than one-time-pad encryption, and the second encryption is with the virtual pad. During decryption, the data has to be doubly-decrypted: first it is decrypted using the virtual pad, and then it is decrypted using the other method.

v1pad currently supports three plain text encryption methods, plus the null method. The plain text encryption methods are Bit-Level Scramble (BLS), Positional Value Substitution (PVS), and Advanced Encryption Standard (AES). AES is a public domain encryption method selected by the US Government through an open competition. BLS and PVS are both novel encryption methods developed for use with v1pad. For each real plain text encryption method, v1pad allows resetting the plain text encryption key in the outer loop, in the middle loop, or in the inner loop.

The command line argument used to specify the plain text encryption method is `PlainTextAlgorithm`. The valid values for `PlainTextAlgorithm` are `PlainNull`, `PlainBLS`, `PlainPVS`, and `PlainAES128`.

`PlainNull` tells v1pad not to perform plain text encryption (only perform one-time-pad encryption). `PlainBLS` tells v1pad to encrypt plain text using Bit-Level Scramble (BLS). `PlainPVS` tells v1pad to encrypt plain text using Positional Value Substitution (PVS). `PlainAES128` tells v1pad to encrypt plain text using Advanced Encryption Standard (AES) with a 128-bit (16-byte) key. The default value is `PlainBLS` because it provides the strongest encryption for a 256-byte block size.

Now we'll describe each of the supported plain text encryption methods in more detail

PlainNull: No Plain-Text Transformation

`PlainNull` skips encrypting the plain text, so the only protection provided with `PlainNull` is the one-time pad. The one-time pad provides very strong encryption, but it does have a few weaknesses.

It is valid to specify both `OneTimePadAlgorithm=PadNone` and `PlainTextAlgorithm=PlainNull`. What this does is output the original plain text, possibly with leading admin bytes, a (fully-encrypted) random block inserted somewhere, and the final block of plain text padded to an even multiple of 256 bytes. This is probably only useful for performance testing.

PlainBLS: Transform Plain Text with BLS

Bit-Level Scramble is a varying-length block cypher. It's very simple, but at the same time powerful, scalable, and performant.

BLS encryption is implemented by performing a simple random scramble of each block of data at the bit level, using a set of pseudo-random numbers stored in an array. The scrambling algorithm is a variation of the standard Fisher-Yates / Durstenfeld algorithm. BLS decryption is implemented by filling an array with the exact same set of pseudo-random numbers used for encryption, and then running the encryption steps backwards.

Security of BLS

For a given block size, BLS is less secure than traditional block cyphers. Traditional block cyphers have an output complexity of 256^N, where N is the number of bytes in the block. For the same number of bytes, BLS has a best-case output complexity of ((8N)!)/((4N)!)^2, and will produce less complex output if the proportion of zero bits to one bits is not close to 1:1. For example, a 64-byte traditional block cypher has an output complexity of 256^64, which is 1.3e+154. BLS with a 64-byte block size has a best-case output complexity of ((8*64)!)/((4*64)!)^2, which is 4.7e+152. The best-case complexity for BLS is close, but still around 100 times less complex.

One advantage of BLS over traditional block cyphers is that it can be used with much larger block sizes than most traditional block cyphers with no significant performance penalty.

v1pad currently uses BLS with a minimum block size of 256 bytes and a maximum block size of 4096 bytes. BLS with a 128-byte block size has a best-case output complexity of (8*128)!/((4*128)!)^2, which is 4.5e+306. BLS with a 256-byte block size has a best-case output complexity of (8*256)!/((4*256)!)^2, which is 5.7e+614.

BLS with a 4096-byte block size has a best-case output complexity of (8*4096)!/((4*4096)!)^2, which is 32768!/(16384!)^2. These numbers are much too large to compute with the Windows calculator, but we can calculate this result using logarithms. 32768!/(16384!)^2 is (32768 * 32767 * 32766 * … * 16385 * 16384!) / (16384! * 16384 * 16383 * 16382 * … * 1), which simplifies to (32768 * 32767 * 32766 * … * 16385) / (16384 * 16383 * 16382 * … * 1). Taking the base-ten logarithm of this expression gives (log(32768) + log(32767) + log(32766) + … + log(16385)) - (log(16384) + log(16383) + log(16382) + … + log(1)). Computing the two series of sums with a program gives 71797.8769057286 - 61936.0817960312, which is 9861.7951096974. The actual number is antilog(0.7951096974) * 10^9861, which is 6.2e+9861.

BLS does not require a perfect 1:1 ratio of zero bits to one bits. It works correctly with a skewed distribution, but the output will be less complex. The best-case complexity for BLS occurs when there is an exact 1:1 ratio of zero bits to one bits. This can be calculated using the simplified formula (<total_bits>!) / ((<total_bits>/2)!)^2. The general formula is (<total_bits>!) / ((<total_bits>-<total_one_bits>)!*(<total_one_bits>!)). The exact same formula would apply if we looked at the total number of zero bits instead of the total number of one bits.

Taking a somewhat extreme case, suppose on the average there is only a single one bit per byte. For a 128-byte block size, that would be 128 one bits out of 128*8 = 1024 total bits. Applying the general formula, we get 1024!/(896!)*(128!). This is 1.4e+166, which is less than the desired minimum complexity of 1.0e+210. For a 256-byte block size, that would be 256 one bits out of 256*8 = 2048 total bits. Applying the general formula, we

get 2048!/(1792!)*(256!). This is 3.5e+333, which is well into the computationally-infeasible range.

For a 4096-byte block size, that would be 4096 one bits out of 4096*8 = 32768 total bits. Applying the general formula, we get 32768!/(28762!)*(4096!). We'll use logarithms to compute this. 32768!/(28762!)*(4096!) is (32768 * 32767 * 32766 * … * 28763 * 28762!) / (28762! * 4096 * 4095 * 4094 * … * 1), which simplifies to (32768 * 32767 * 32766 * … * 28763) / (4096 * 4095 * 4094 * … * 1). Talking the base ten logarithm of this expression gives (log(32768) + log(32767) + log(32766) + … + log(28763)) - (log(4096) + log(4095) + log(4094) + … + log(1)). Computing the two sums using a program gives 17977.9536545966 - 13019.5614277442, which is 4958.3922268524. The actual number is antilog(0.3922268524) * 10^4958, which is 2.5e+4958.

The key for BLS is an array of pseudo-random numbers, and the overall v1pad encryption algorithm is also based on pseudo-random numbers, so BLS fits well with the rest of the v1pad encryption logic. Encrypting or decrypting N bytes of plain text using BLS requires (8*N) - 1 pseudo-random numbers. Normally some number for N is chosen so that all array elements can easily fit into computer memory. Block processing runs faster in memory, and very large values of N may not fit into memory.

Essentially, this means that BLS uses a longer key size for larger blocks, which is another reason to run BLS using large block sizes. The pseudo-random numbers used by BLS come from a CSPRNG, so the effective key size for BLS also depends on the number of possible starting states for the CSPRNG and the CSPRNG's cycle length.

If the input to the CSPRNG is derived from an 8-byte password, then the effective key size for BLS is at most 8 bytes (probably less), even if the internal state of the CSPRNG is larger than 8 bytes and the cycle length of the CSPRNG is longer than 63 ((8*8) - 1). To avoid this weakness, v1pad does two things:1) it builds a large internal key (the keyspace) at least big enough to fully-initialize the CSPRNG; and 2) it includes as much real complexity into the keyspace as possible so that when the CSPRNG is initialized from the keyspace, the actual complexity of the CSPRNG initialization array is close to the maximum complexity that the initialization array can represent.

How BLS works with different re-key options

As mentioned above, the key for BLS is an array of pseudo-random numbers. For a block size of N bytes, BLS requires a key size of (8*N) - 1.

v1pad's main file encryption / decryption routine tracks how many times the inner loop has executed within a given outer loop iteration in the variable Inner Loop Count. The inner loop executes 32*128 = 4096 times within a given outer loop iteration. Each iteration of the inner loop will process at most 4k of input.

When BLS is run with `PlainTextReKeyFrequency` set to `ReKeyInner`, every time a block of text is encrypted, v1pad refreshes the BLS key with (8*N) - 1 new random numbers, and the random numbers are read from the key array from index zero to index (8*N) - 2. Any offset passed to BLS is ignored.

When BLS is run with `PlainTextReKeyFrequency` set to `ReKeyMiddle`, the BLS key array is refreshed with 8*N new random numbers at the top of the middle v1pad encryption loop, and the random numbers are read from the key array using modular arithmetic. An offset is combined with the index that loops through all of the bits in the plain text so that BLS reads the random numbers at location (index + offset) mod 8*N. The offset used is 1024 * (<inner loop count> mod 32), so for each of the 32 possible inner loop iterations within a single middle loop iteration, the offset advances by 32k/32, which uniformly splits up the 32k random numbers in the BLS key array.

When BLS is run with `PlainTextReKeyFrequency` set to `ReKeyOuter`, the BLS key array is refreshed with 8*N new random numbers at the top of the outer v1pad encryption loop, and the random numbers are read from the key array using modular arithmetic. An offset is combined with the index that loops through all of the bits in the plain text so that BLS reads the random numbers at location (index + offset) mod 8*N. The offset used is 8 * <inner loop count>, so for each of the 4096 possible inner loop iterations within a single outer loop iteration, the offset advances by 32k/4k, which uniformly splits up the 32k random numbers in the BLS key array.

When `PlainTextReKeyFrequency` is set to `ReKeyMiddle` or `ReKeyOuter`, the BLS key array is filled with 8*N random numbers, but BLS still only uses (8*N) - 1 of them at a time. The "extra" random number is included so that the modular access of the random numbers will always be able to read a valid random number. If this was not done, the uninitialized value at the very end of the key array would cause trouble later.

Standalone BLS

BLS as implemented in the v1pad project is a generic encryption algorithm that will provide strong encryption in the context of v1pad itself or when run standalone. When the BLS encrypt/decrypt routines are called with the option to reset the BLS key array internally, BLS will work fine for an N-byte block size as long as the BLS key array is able to contain at least 8*N - 1 unsigned four-byte random numbers. When the BLS routines are called with a pre-populated key array, BLS will work fine as long as the BLS key array is pre-populated with exactly 8*N unsigned four-byte random numbers and the offset is set to some reasonable value.

It would be unwise to re-use the same set of random numbers in the BLS key array for more than 8*N calls to the base BLS routines. Even with the offset set to <call number>, the same sequence of random numbers would be re-used starting with call number ((8*N) + 1). This will still encrypt/decrypt the input data, but it is unsafe because with the

proper inputs, an attacker may be able to reverse-engineer the internal state of the BLS key array.

PlainPVS: Transform Plain Text with PVS

Positional Value Substitution (PVS) is a substitution cypher. Classic substitution cyphers have a single substitution mapping table that determines the output byte value for each input byte. For example, the mapping could be set up so that the letter E is always replaced with the letter Z.

Positional Value Substitution implements a more complex mapping. PVS has a fixed block size. For each position in the block, there is a separate substitution mapping table. Taking the example above, the letter E may be replaced with the letter Z when E is found in the first position in the block, but it's unlikely to be replaced with the letter Z when found in the second position in the block, since the first and second positions in the block each have their own pseudo-randomly-scrambled substitution mapping table. This is intended to make it much harder to implement a heuristic attack on PVS based on the frequency in which each output value occurs.

Traditionally the weakness of substitution cyphers is the fact that there is a one-to-one mapping between input byte values and output byte values. Since the distribution of input values usually follows specific patterns, it's likely that the distribution of output values also follows the same patterns. For example, E is the most-used letter in English, so with a single substitution mapping table, it's a good bet that if the input was English text, the most-frequently-occurring output value corresponds to the letter E. PVS should not have that weakness, since for PVS E is just an index into one of many randomly-scrambled lookup tables, and in any given lookup table, it would seem to be equally probable that any one of the possible 256 single-byte values is present at index E.

To take full advantage of the complexity stored in the lookup tables and to make it (a little) harder to reverse-engineer the PVS key with a plain text attack, two things are done. First, the mapping between the block positions and the substitution mapping tables is not fixed. Instead, there is a separate set of scrambles tables used so that each time a block is processed, the contents of the scrambles table corresponding to that block are used to map the block positions to the lookup tables. Second, each time a block is processed, the corresponding scrambles table is traversed using a different offset from an offsets table corresponding to each scrambles table.

PVS as implemented in the v1pad project uses a block size of 256 bytes. The PVS key is made up of three two-dimensional arrays: Lookups, Scrambles, and Offsets. All three consist of 256 arrays of unsigned bytes, each of which contains the values from 0 to 255 in a random order.

PVS takes as input an array of unsigned bytes, the size of the array, and the logical starting byte address of the first byte in the array. If at any point when processing the

array, the working logical byte address is higher than the maximum data the current implementation of PVS can safely process, the address wraps around modularly.

Because the PVS block size is 256 bytes and because of the way the PVS key is used to encrypt or decrypt input data, the most data that can be encrypted without refreshing the PVS key is 256^3 bytes, which is 16 Meg. When converting the current logical byte address into array accesses, we take 8 bits for the offsets array, 8 bits for the scrambles array, and 8 bits for the data index, which is a total of 24 bits (16 Megabytes).

All of this looks like:

```
Setup:

 - Lookups[256][256],   sub-arrays have [0-255] in a random order
 - Scrambles[256][256], sub-arrays have [0-255] in a random order
 - Offsets[256][256],   sub-arrays have [0-255] in a random order

Crypt:

For Loopvar from 0 to InputBlockSize-1
 - LogicalAddress = (StartingLogicalAddress + Loopvar) mod 16 Meg
 - OFarray = highest-order eight bits of LogicalAddress
 - SCarray = middle eight bits of LogicalAddress
 - DataIndex = lowest-order 8 bits of LogicalAddress
 - SCindex = (DataIndex + Offsets[SCarray][OFarray]) mod 256
 - OutputValue =
     Lookups[Scrambles[SCarray][SCindex]][InputData[Loopvar]]
```

StartingLogicalAddress needs to start at zero and increment by 256, so that it eventually reaches (16 Meg - 256) on the final iteration.

For the first and second block of 256 bytes processed, we have:

```
Byte 0:
     Lookups[Scrambles[0][0  +Of[0][0]]][InputData[Loopvar]]
Byte 1:
     Lookups[Scrambles[0][1  +Of[0][0]]][InputData[Loopvar]]
Byte 255:
     Lookups[Scrambles[0][255+Of[0][0]]][InputData[Loopvar]]

Byte 256:
     Lookups[Scrambles[1][0  +Of[1][0]]][InputData[Loopvar]]
Byte 257:
     Lookups[Scrambles[1][1  +Of[1][0]]][InputData[Loopvar]]
Byte 511:
     Lookups[Scrambles[1][255+Of[1][0]]][InputData[Loopvar]]
```

For the next 255 blocks of 64k each, we'll reference a new offset for each scrambles array. For each scrambles array, we'll get the same sequence of lookup arrays referenced

across the 64k blocks when looked at as a circular list of numbers, but with a different starting number each time. This is a different traversal of the lookups arrays for each scrambles array across all 256 64k blocks of input.

Next 64k block:

```
Byte 64k + 0:
     Lookups[Scrambles[0][0  +Of[0][1]]][InputData[Loopvar]]
Byte 64k + 1:
     Lookups[Scrambles[0][1  +Of[0][1]]][InputData[Loopvar]]
Byte 64k + 255:
     Lookups[Scrambles[0][255+Of[0][1]]][InputData[Loopvar]]

Byte 64k + 256:
     Lookups[Scrambles[1][0  +Of[1][1]]][InputData[Loopvar]]
Byte 64k + 257:
     Lookups[Scrambles[1][1  +Of[1][1]]][InputData[Loopvar]]
Byte 64k + 511:
     Lookups[Scrambles[1][255+Of[1][1]]][InputData[Loopvar]]
```

It is not possible for us to get the same starting number for a given scrambles array across different 64k blocks because we set up the arrays and references so that each scrambles array has its own unique set of offsets that run from 0 to 255.

It is possible for Scrambles[a] to have the same contents as Scrambles[b] when each one is viewed as a circular list of numbers. As a special case of this, the two circular lists of numbers could be exactly aligned so that Scrambles[a] and Scrambles[b] have exactly the same contents.

I tried to calculate the chances of this using the same math used for the "birthday problem". By my calculations, the probability that at least one of the 256 scrambles arrays is circularly-identical to another array is very close to 0, meaning that there is very little chance of this happening.

With this implementation, the PVS key is 256*256*3 bytes long, or 192k. It can be used to encrypt up to 16 Meg of data before the key needs to be refreshed. Key refresh takes 256*3*255 random numbers to set up, which is slightly fewer than 192k four-byte random numbers.

In the context of v1pad, each inner loop iteration processes up to 4k of data, the middle loop processes up to 128k of data, and the outer loop processes up to 16 Meg of data. All three loops are valid rekey points for PVS. For all use cases, we can either pass the total size of the data processed so far to PVS (that's an 8-bytes integer because of long files support) or have a separate data processed counter for PVS and zero it out every time the PVS key is initialized.

The exact same algorithm is used to decrypt using PVS. The difference between PVS encryption and PVS decryption is that for decryption the Lookups array is set up differently.

To set up a given values mapping table for decryption:

1. Load a temporary array Temp with the values from 0 to 255
2. Pseudo-randomly scramble Temp using exactly the same set of pseudo-random numbers used to scramble the corresponding values mapping table when it was set up for encryption
3. Set the actual values mapping array so that Lookups[*][Y] is the index in Temp of Value Y.

Because the PVS key is large and contains a lot of scrambled data, it is expensive to set up. Once the PVS key has been set up, PVS is actually a very fast encryption algorithm, since it just performs table lookups for both encryption and decryption.

Security of PVS

I wrote an algorithm to reverse-engineer PVS in an attempt to convince myself that it cannot be done. I'm sad to say that an attacker can pretty easily reverse-engineer PVS using a batch chosen plain text attack.

Furthermore, I'm convinced that this general strategy of encrypting using lookup tables is fundamentally flawed in the sense that there does not appear to be ANY variant that resists a chosen plain text attack. There are variants that require only a single chosen plain text and variants that require 256 or more chosen plain texts, but there don't appear to be any variants that really resist chosen plain text attacks.

That does not mean PVS has no value at all. It just means that PVS is only safe to use for specific use cases. In the context of v1pad, PVS is only safe to use in normal mode. In normal mode, the user inputs are combined with really-random data to produce a different internal key for each v1pad run. It's not possible to mount a chosen plain text attack under these conditions (that attack requires re-use of the same encryption key). In simple mode, the really-random data is omitted, so the same user inputs will always produce the same internal key, and so it is possible to mount a chosen plain text attack.

I can't endorse my own PVS algorithm for standalone use. Most encryption systems only take a single key as input. Using different keys each time you encrypt something is painful, especially if you use encryption a lot. Therefore, it seems reasonable that someone using PVS outside the context of v1pad would re-use the same encryption key, and PVS is vulnerable to a batch chosen plain text attack under these conditions.

Let me state it a little more strongly, just in case the above was not clear. **Don't use PVS outside the context of v1pad normal mode.** It is fast and produces nice complex

output, but it is vulnerable to a batch chosen plain text attack, and someone may figure out how to mount this attack against your implementation.

How PVS works with different re-key options

As mentioned above, the key for PVS is three two-dimensional arrays of unsigned bytes, the total size of the PVS key is 192k, and it happens to require slightly fewer than 192k unsigned four-byte random numbers to set up the PVS key. The maximum amount of data that can safely be encrypted without refreshing the PVS key is 16 Megabytes.

v1pad's main file encryption / decryption routine tracks how many bytes of data have been encrypted / decrypted with PVS since the last time the PVS key was refreshed and supplies this value as the starting logical byte address to PVS each time a set of data is encrypted or decrypted using a previously-set-up PVS key.

When PVS is selected as the plain text encryption algorithm and a one-time-pad encryption routine is called with `PlainTextReKeyFrequency` set to `ReKeyInner`, the PVS key is reset inside the PVS encryption routine itself for every inner loop iteration. This requires a little less than 192k new pseudo-random numbers every time. For this use case, the starting logical byte address is hardcoded to zero.

When PVS is run with `PlainTextReKeyFrequency` set to `ReKeyMiddle`, the PVS key array is refreshed at the top of the middle v1pad encryption loop. The starting logical byte address passed to PVS is the one described above.

When PVS is run with `PlainTextReKeyFrequency` set to `ReKeyOuter`, the PVS key array is refreshed at the top of the outer v1pad encryption loop. As with `ReKeyMiddle`, the starting logical byte address passed to PVS for `ReKeyOuter` is the one specifically computed for this purpose.

Standalone PVS

Per the comments in the section on the security of PVS: **Don't use PVS outside the context of v1pad normal mode.** See that section for more details.

PlainAES128: Transform Plain Text with AES128

Advanced Encryption Standard (AES) is a public domain encryption algorithm approved by the US Government for some applications. It is widely believed to be secure if implemented properly.

v1pad uses AES with a 128-bit (16-byte) key in Cypher Block Chaining mode. The AES Key and Initialization Vector are generated from CSPRNG output and refreshed according to the `PlainTextReKeyFrequency` command-line argument.

v1pad uses AES128 instead of AES256 for two reasons: 1) AES128 is faster; and 2) AES256 appears to have some potential weaknesses because of issues with its key schedule, whereas AES128 does not have these same weaknesses.

Keep in mind the primary functional reason v1pad uses AES is to prevent plain text attacks on data that is also encrypted using a virtual one-time pad, so even if AES256 really is more secure than AES128, the difference in security may not be worth the performance hit.

AES is also supported in v1pad for a few non-functional reasons:

1) To provide a "safe" option for those who are concerned about using BLS or PVS, which are new encryption methods not yet evaluated by the security community
2) To show that v1pad can be used as a framework for various plain text encryption methods
3) To provide some kind of performance comparison between BLS, PVS, and a more well-understood encryption method

AES128 is a block cypher with a 16-byte block size, and it doesn't like to work with block sizes smaller than 16 bytes. In v1pad normal mode, this is irrelevant, since v1pad normal mode does everything with a minimum block size of 256 bytes, and 256 is an even multiple of 16.

In v1pad simple mode, this becomes an issue. v1pad will happily encrypt a file whose size is not an even multiple of 16 bytes using AES128 in v1pad simple mode, but the encrypted file will include some padding bytes to round it up to an even 16-byte boundary. When that same file is decrypted, the decrypted version will have a rounded-up size, and the final few bytes in the file beyond the original file contents will all be zero bytes. To avoid this "quirk", use v1pad in normal mode.

The AES code used by v1pad comes from the "Tiny AES" project on GitHub (https://github.com/kokke/tiny-AES-c). I had to make two bug fixes in order to get it to work properly with v1pad (they are documented in the source code).

Security of AES

See external literature, like Wikipedia.

How AES works with different re-key options

The AES key for AES128 in Cypher Block Chaining mode consists of a 16-byte pseudo-random key and a 16-byte pseudo-random initialization vector. v1pad handles both of these together as if they were a single key value.

The AES literature seems to indicate that AES128 in CBC mode has possible collisions (per the "birthday problem") after about 2^64 blocks have been encrypted. That's about 1.8e+19 16-byte blocks, or about 3.0e+20 bytes. Based on that number, it seems perfectly safe to use AES128 in v1pad, as the minimum key refresh frequency in v1pad is once every 16 Megabytes (about 1.7e+7 bytes).

v1pad's main file encryption / decryption routine tracks how many bytes of data have been encrypted / decrypted with AES since the last time the AES key was refreshed and supplies this value as additional information to AES each time a set of data is encrypted or decrypted using a previously-set-up AES key.

Since this is the same counter behavior as for PVS, it re-uses the same counter variable internally. The main difference between how this counter variable is used for AES and PVS is that for AES, we only care if the counter is zero or non-zero. If the counter is zero, we tell the base AES routines to perform key setup and to take the initialization vector from the AES key structure. If the counter is non-zero, we tell the base AES routines to skip the key setup and use the last saved output block as its initialization vector.

When AES is selected as the plain text encryption algorithm and a one-time-pad encryption routine is called with `PlainTextReKeyFrequency` set to `ReKeyInner`, the AES key is reset inside the AES encryption wrapper routine itself for every inner loop iteration. This requires eight new pseudo-random numbers every time. For this use case, the additional information is hardcoded to zero.

When AES is run with `PlainTextReKeyFrequency` set to `ReKeyMiddle`, the AES key is refreshed at the top of the middle v1pad encryption loop. The additional information passed to AES is the value of the counter variable described above.

When AES is run with `PlainTextReKeyFrequency` set to `ReKeyOuter`, the AES key is refreshed at the top of the outer v1pad encryption loop. As with `ReKeyMiddle`, the additional information passed to AES for `ReKeyOuter` is the one specifically computed for this purpose.

Standalone AES

Per the comments in the section on the security of AES, AES is a public domain encryption standard widely believed to be secure if implemented properly.

Key Indirection

Introduction

At a high level, the Key Indirection feature was designed as a defense against key logger software. The design goal was to make sure that knowing the exact key information entered by the user would not be sufficient for decrypting an encrypted file. As a side effect, Key Indirection greatly strengthens the v1pad passphrase (unless it's already extremely strong).

Key Indirection replaces the passphrase and keyfile entries entered by the user with different data. Then v1pad runs using the new key information instead of the original key information entered by the user. The exact replacement mapping depends on the type of input. Password input is replaced with what looks like a completely random set of ASCII characters. File and directory name input is conditionally replaced with different file and directory name input. The user determines the mapping between the original file or directory input values and the replacement file or directory values when they create the Key Indirection data files.

Here is a table giving a high-level overview of what Key Indirection replaces.

Use Case	**Syntax**	**Preplaces**	**Replacement value**
Primary v1pad passphrase	A line of 8 to 255 characters (multibyte characters allowed)	The entire line	Random-looking ASCII text
Generic add-on keyfile entries	<letter>::<string>, where <letter> is not 'b' or 'B'	Just the <string> part	Random-looking ASCII text
Real keyfile entries	A line of up to about 4k single-byte characters	The entire line	Either not replaced, or the whole line is replaced by some other pre-defined string
Keyfile base directory add-on keyfile entries	<letter>::<path>, where <letter> is 'b' or 'B'	Just the <path> part	Either not replaced, or the whole <path> is replaced by some other pre-defined string

Key Indirection works by reading data files set up in advance by the user. The primary v1pad passphrase is used as a key to determine which Key Indirection data set to use for a particular v1pad run. If there is no Key Indirection data set that corresponds to the primary v1pad password entered for a v1pad run, the Key Indirection feature is not used.

Key Indirection provides better security in two ways:

1. It automatically generates longer and stronger passphrases than most people could ever type or remember
2. It partially protects against key logger software

Key Indirection only completely protects against key logger software if the attacker does not have access to the Key Indirection data set corresponding to the primary v1pad passphrase recorded by the key logger software. If the attacker has the exact set of key information entered by the data owner and also has the corresponding Key Indirection data set, the attacker will be able to decrypt the file (unless the data owner uses a more advanced feature explained later).

To make it harder for an attacker to obtain the Key Indirection data set used for a specific v1pad run, it's a good idea to keep all Key Indirection data sets on a removable drive and point the Key Indirection base directory to the removable drive when running v1pad.

Key Indirection command-line arguments

There are two optional v1pad command-line arguments related to Key Indirection: `KeyIndirectionMode` and `KeyIndirectionBaseDirectory`.

`KeyIndirectionMode` defaults to "Disabled". Valid values are "Disabled", "Enabled", and "Required". When `KeyIndirectionMode` is set to "Disabled", the Key Indirection feature is turned off. When `KeyIndirectionMode` is set to "Enabled", the Key Indirection feature is turned on if possible. If v1pad is not able to turn on the Key Indirection feature, it continues running with the Key Indirection feature turned off. When `KeyIndirectionMode` is set to "Required", the Key Indirection feature must be turned on. If v1pad is not able to turn on the Key Indirection feature, v1pad fails.

`KeyIndirectionBaseDirectory` has no default value. Valid values are any directory that can be both read and written by the user running v1pad. `KeyIndirectionBaseDirectory` must be specified either on the command line or in the configuration file in order for the Key Indirection feature to be successfully turned on.

Key Indirection runtime setup overview

Because Key Indirection uses the v1pad master passphrase to find the correct data files, Key Indirection runtime setup happens after the user enters the v1pad passphrase, but before the passphrase data is really processed. That way if Key Indirection is enabled, the passphrase entered by the user can be immediately replaced with a different passphrase.

Because Key Indirection only impacts how key information is processed, the internal Key Indirection state is cleared after the user enters the keyfile information and before the

keyfile information is really processed by v1pad. If Key Indirection is enabled, the keyfile entries are modified before the internal Key Indirection state is cleared, and the modified keyfile entries are processed by v1pad.

When `KeyIndirectionMode` is set to "Disabled", v1pad just prints a message that Key Indirection is turned off and continues.

When `KeyIndirectionMode` is set to "Enabled" or "Required", v1pad makes an effort to locate and read the correct Key Indirection data set. If it is able to do this, it sets up some internal data structures and prints a message that Key Indirection is enabled. If it is not able to do this, the Key Indirection setup logic fails. If Key Indirection setup fails when `KeyIndirectionMode` is set to "Enabled", v1pad just prints a message that Key Indirection is disabled, clears any internal data structures that were partially set up, and continues. If Key Indirection setup fails when `KeyIndirectionMode` is set to "Required", v1pad clears all Key Indirection internal data structures and fails.

At a high level, Key Indirection runtime setup works as follows:

1. Make sure the Key Indirection base directory is valid
2. Check to see if there is a Key Indirection data set in the Key Indirection base directory corresponding to the main v1pad passphrase
3. Set up internal Key Indirection data structures using the appropriate Key Indirection data set
4. Mark Key Indirection as turned on

If any of the above steps fails, Key Indirection setup fails.

The self-destruct data file

In the overview section, we mentioned that there is a feature that can protect against an attacker who not only has the exact key information but also has the Key Indirection data set corresponding to the v1pad primary passphrase. The self-destruct data file was designed to provide protection even against an attacker who has both the key information and the correct Key Indirection data set.

The self-destruct data file contains a block of data that is XORed with the data from the two primary data files so that the final data used by v1pad depends on both the self-destruct data file and the two primary data files.

At a high-level, the self-destruct data file introduces a single point of failure into the Key Indirection setup process, and the Key Indirection setup process is modified to remove the self-destruct data file after reading it. The end result is a broken setup.

To recover from this broken state, Key Indirection offers two options: automatic recovery and manual recovery. In automatic recovery, the self-destruct recovery file in the corresponding Key Indirection data set is encrypted with the corresponding v1pad

passphrase, and v1pad can automatically re-create the self-destruct data file as part of the Key Indirection setup process. In manual recovery, the self-destruct recovery file in the corresponding v1pad data set is encrypted with some passphrase other than the corresponding v1pad passphrase, and the data owner must run a standalone program to create a new self-destruct data file from the self-destruct recovery data file before v1pad can be used to encrypt or decrypt the target data file.

If the relevant Key Indirection data set is configured for manual recovery, an attacker who gets all of the key information and the correct Key Indirection data set will not be able to decrypt the target data file. If the relevant Key Indirection data set is configured for automatic recovery, an attacker who gets all of the key information and the correct Key Indirection data set will be able to easily decrypt the target data file.

Whether the user wants to configure a given Key Indirection data set for automatic recovery or manual recovery is up to the user. Automatic recovery protects against attackers who can get all of the key information but cannot get the Key Indirection data set, and it's easy to use. Manual recovery also protects against attackers who can get the Key Indirection data set, but it's harder to use because of the additional manual step required every time encryption or decryption is performed using that data set.

How Key Indirection substitution works

As was mentioned above, Key Indirection replaces the user-supplied key information with different key information. Password entries are replaced by what looks like random ASCII text. File and directory entries may or may not be replaced by different file or directory names, depending on how the relevant Key Indirection data set was configured.

Let's start with the simple one. The Key Indirection local properties file contains a set of <name>=<value> pairs. If a given file or directory exactly matches <name>, Key Indirection replaces it with <value> before any keyfile entries are processed.

The password substitution is more complicated. The password entries are the main v1pad passphrase and the <string> portion of any add-on keyfile entries (except for add-on keyfile entries that start with 'b::' or 'B::', which are treated as directory entries)

For each password entry, Key Indirection first determines the desired length of the replacement entry. If the original entry is shorter than 255 bytes, Key Indirection will pick a random target length somewhere between 149 and 255, inclusive. If the original entry is 255 bytes long or longer (possible for longer multibyte passphrases and all add-on keyfile entries), Key Indirection sets the target length to the original length. If the randomly-selected target length for an original entry is less than the original entry length, it is reset to the original entry length. All of this is done to ensure that the target password length is at least 149 bytes, to allow the target length to vary, and to make sure that the target length is at least as long as the original entry length.

Once the target length has been established, Key Indirection fills an array with the desired number of bytes taken from a cryptographically-secure pseudo-random number generator (CSPRNG). Then the bytes from the original password entry are modularly XORed into the array. Then the values in the array are used to select the same number of ASCII characters from a randomly-scrambled array that contains one copy of each printable ASCII character (96 characters total, including space and tab). The candidate replacement password is validated to make sure it meets minimum complexity rules and is modified to comply if for some reason it does not already comply. The end result of this process is used as the replacement password.

Note that <letter>::<null_string> is a valid add-on keyfile entry. For this use case, there is nothing to XOR into the array of pseudo-random bytes. The algorithm above handles this as a special case and still produces a random-looking replacement password between 149 and 255 bytes long. The only difference is that the replacement password for <null_string> only depends on the initial state of the CSPRNG when it was generated. For non-null original passwords, the replacement password depends on both the initial state of the CSPRNG and on the original password.

For an N-byte replacement password, the minimum guaranteed complexity is 43*(26^(N-1)), but a more likely complexity is 96^N. A replacement password with the minimum possible length of 149 characters will have a guaranteed minimum complexity of 43*(26^128) (which is 1.1e+211) and a more likely complexity of 96^149 (which is 2.3e+295). Longer replacement passwords are more complex.

Key Indirection data sets

A Key Indirection data set consists of the following three files:

1) The transformations data file
2) The local properties data file
3) The self-destruct recovery data file

The transformation data file and the local properties data file are the primary data files. The self-destruct recovery data file is used to re-create the self-destruct data file.

All three files are text files that contain the base-64-encoded representation of binary data. All three files start with Auth blocks with a random block size of 1k (see the discussion on Auth files and Auth blocks in the concepts section).

The transformations data file just contains an Auth block encrypted with a passphrase that will later be used as the primary v1pad passphrase. During v1pad setup, the Auth block in the transformations data file is decrypted using the current v1pad passphrase. If decryption fails, this data set does not correspond to the current v1pad passphrase. If decryption succeeds, the decrypted 1k block of random data is XORed with the corresponding 1k block of decrypted random data from the self-destruct data file, and the resultant 1k data block is used to initialize the cryptographically-secure pseudo-random

number generator (CSPRNG) that the Key Indirection feature uses to generate the random-looking ASCII text that replaces password information provided by the user.

The local properties data file contains an initial Auth block encrypted with the same passphrase as the transformations data file. During v1pad setup, the Auth block in the local properties data file is decrypted using the current v1pad passphrase. If decryption fails, this data set does not correspond to the current v1pad passphrase. If decryption succeeds, the decrypted 1k block of random data is XORed with the corresponding decrypted 1k block of random data from the self-destruct data file, and the resultant 1k data block is used to initialize the CSPRNG that the Key Indirection feature uses to decrypt the other entries in the local properties data file.

The other entries in the local properties data file are <name>=<value> pairs padded to about 4k bytes and encrypted. Each <name>=<value> pair is implemented as two separate padded and encrypted data blocks. The first block stores the value of <name>, and the second block stores the value of <value>. During the Key Indirection setup process, the <name>=<value> mapping information in the local properties data file is loaded into a hash table, and the hash table is later used to replace keyfile entries or the <path> portion of keyfiles base directory add-on keyfile entries.

As was mentioned above, only file or directory entries whose value exactly matches one of the <name> entries in the hash table are replaced with <value> instead. Any file or directory entries that do not exactly match one of the <name> entries in the hash table are left unchanged.

The self-destruct recovery data file also contains an initial Auth block. If the Auth block in the self-destruct recovery data file is encrypted using the same passphrase as the transformations data file and the local properties data file, the data set is configured for automatic recovery. We also call these data sets homogeneous because the Auth block in all three data files is encrypted using the same passphrase. If the Auth block in the self-destruct recovery data file is encrypted with some passphrase other than the one used to encrypt the Auth blocks in the transformations data file and the local properties data file, the data set is configured for manual recovery. We also call these data sets “consistent but not homogeneous” because the two primary data files have their Auth blocks encrypted with the same passphrase, but the self-destruct recovery file does not.

When the Auth block in the self-destruct recovery data file is read using the passphrase which was originally used to create it, the result is a decrypted 1k data block which is used to initialize the CSPRNG that the Key Indirection feature uses to decrypt the other entry in the self-destruct recovery data file. The other entry in the self-destruct data file is an encrypted version of the 1k block of random data which will be used to create the self-destruct file itself.

Whether the self-destruct data file is created automatically by v1pad or by running a separate standalone program, the high-level steps are the same. The 1k data block stored as the payload in the self-destruct recovery data file is used as input along with the

specified v1pad master passphrase to create the self-destruct data file. The self-destruct data file only contains an Auth block encrypted using the v1pad master passphrase. It doesn't contain any other data.

Key Indirection data file names

All Key Indirection data files mentioned here must be located in the Key Indirection base directory. Otherwise v1pad will not find them.

The self-destruct data file is always named SD.txt. It has a simple name because it's not permanently part of any given data set. Instead, it's generated from some specific data set, used to process that exact data set, and then deleted. It's deliberately not a permanent file.

The three data files which form each data set are all named <prefix>_<data_set_name>.txt. The prefix for the transformations data file is TD. The prefix for the local properties data file is LP. The prefix for the self-destruct recovery data file is SR. <data_set_name> is normally a ten-character string composed of characters more-or-less randomly chosen from the set of all upper-case ASCII letters and the digits zero through nine. In the rare case where there are multiple data sets in a Key Indirection base directory and the randomly-chosen <data_set_name> for a new data set conflicts with the randomly-chosen <data_set_name> for an existing data set, the <data_set_name> for the new data set will be something like <10_chars>_<4_digits>, where the <4_digits> starts at 0001 and increases by one with each conflict.

For example, if there is already an existing data set named D8VKNWX7CR, it will consist of the transformations data file TD_D8VKNWX7CR.txt, the local properties data file LP_D8VKNWX7CR.txt, and the self-destruct recovery data file SR_D8VKNWX7CR.txt. If for some reason the randomly-generated data set name for a new data set also is D8VKNWX7CR, the new data set will be automatically renamed to D8VKNWX7CR_0001 to avoid a naming conflict. The new data set would then consist of the transformations data file TD_D8VKNWX7CR_0001.txt, the local properties data file LP_D8VKNWX7CR_0001.txt, and the self-destruct recovery data file SR_D8VKNWX7CR_00001.txt

Creating Key Indirection data sets

Master and Derived Data Sets

There are actually two kinds of Key Indirection data sets: master data sets and derived data sets. Derived data sets are the ones mentioned elsewhere in this section. From a v1pad runtime point of view, only derived data sets are significant.

Master data sets only contain a base properties data file named BP_<data_set_name>.txt. They are essentially templates used to create one or more derived data sets, where each derived data set has a local properties file which logically has the same contents as the base properties file in the master data set.

Master data sets may be stored in the same directory as the corresponding derived data sets or in some other directory. As a security best practice, it's better to keep them in a separate directory, since the master data sets will only be used to create derived data sets, so there is no reason to make them available on the computer where v1pad is running.

Master data set names are not randomly-generated. Instead, the same algorithm that generates the derived data set names is run using the Key Indirection master password as input (instead of using random bytes as input). For a given master password, the corresponding master data set name will always be the same, even if you delete it and re-create it. In contrast, for a given data encryption password, the corresponding derived data set name is generated using really-random data, so if you delete the derived data set corresponding to a given data encryption password and then re-create it, the re-created derived data set will have a different name.

The reason for this difference is that master data sets are intended to be private, whereas derived data sets are intended to be public. No one but the data owner should ever know the master password for a master data set, and all operations on a master data set are intended to be run on a protected computer (ideally one not connected to the internet). In contrast, derived data sets are intended to be public in the sense that they are referenced in an environment that may be hostile, and the data encryption password for a derived data set may become known for various reasons (key loggers being one possibility). Because derived data sets are used in a potentially hostile environment, it seemed prudent to make sure that there is no easy way to tell which derived data set corresponds to a given v1pad passphrase.

Of course, if there's only one derived data set in the Key Indirection base directory, and v1pad was able to successfully enable Key Indirection, it's obvious which derived data set was used. However, if there are two or more derived data sets in the Key Indirection base directory, there is really no way to tell which one v1pad used to enable Key Indirection. To support this "defense-in-depth" feature, v1pad's logic that enables Key Indirection on purpose reads all data sets in the Key Indirection base directory, even though at most one of them will actually be used to enable Key Indirection.

Data Set Management Utilities

v1pad provides two standalone utilities for managing data sets: `ki_dataset_util` and `ki_destruct_util`. Their features and what they are used for are described here. For detailed information about their command line arguments, type `<utility> help` at the OS prompt.

`ki_dataset_util` is used for creating master data sets and derived data sets. If a master data set corresponding to a specified Key Indirection master password already exists, it will not be re-created. Instead, it will just be validated. Similarly, if a derived data set corresponding to a given Key Indirection data encryption password (future v1pad main passphrase) already exists, it will not be re-created (it will just be validated).

The reason existing data sets are not re-created is that they all use Auth blocks, and Auth blocks are based on blocks of really-random data. Re-creating a data file that uses Auth blocks will produce a physically different file every time because the block of really-random data will be different every time (ignoring the possibility of collisions, which should be very rare).

For master data sets, this doesn't matter much. They are really just composed of a single file which is never read by v1pad, so as long as the file is self-consistent it doesn't matter what random data is used to create its Auth block.

For derived data sets, this is a major issue. As an example, suppose we create a derived data set corresponding to data encryption passphrase X, and we use this derived data set to encrypt a file <data_file>. When v1pad transforms the passphrase X to X', it uses random numbers from a cryptographically-secure pseudo random number generator (CSPRNG) which was seeded using a combination of the block of random data from the Auth block in the transformations data file and the block of random data from Auth block in the self-destruct data file. If we delete and re-create the derived data set corresponding to the data encryption password X, we'll get logically the same contents, but physically all three of the data files (the transformations data file, the local properties data file, and the self-destruct recovery data file) will have Auth blocks with different contents because all three are based on new 1k blocks of really-random data. The 1k block of random data in the self-destruct recovery file which will later be used to create the self-destruct file will also be different. This means that when v1pad tries to decrypt <data_file> using the new derived data set corresponding to the passphrase X, it will get a different replacement passphrase X'' instead of the original replacement passphrase X', so it will not be able to successfully decrypt <data_file>.

Because of the above issue, neither v1pad nor `ki_dataset_util` nor `ki_destruct_util` ever deletes any Key Indirection data files, except for the self-destruct data file, which is intended to be deleted and re-created as part of normal Key Indirection processing. You also should be careful not to ever delete any Key Indirection data files, unless you have never used them to encrypt or decrypt any target data files with v1pad.

When creating a new master data set with `ki_dataset_util`, you will need to enter the Key Indirection master password and at least one <name>=<value> pair.

When creating a new derived data set with `ki_dataset_util`, you will need to enter the Key Indirection master password, the Key Indirection data encryption password (future v1pad passphrase), and the Key Indirection self-destruct recovery password. To

create a homogeneous data set (automatic recovery), re-enter the data encryption password when prompted for the self-destruct recovery password. To create a non-homogeneous data set (manual recovery), enter a value different from the data encryption password when prompted for the self-destruct recovery password.

`ki_destruct_util` is used to create the self-destruct data file from the self-destruct recovery data file. It takes as input the Key Indirection data encryption password (future v1pad passphrase) and the Key Indirection self-destruct recovery password. If it finds a derived data set in the Key Indirection base directory which corresponds to the specified data encryption password, it tries to decrypt only the self-destruct recovery data file in that data set using the self-destruct recovery password. If it is able to decrypt the self-destruct recovery data file, it uses the information in the self-destruct recovery data file to create the self-destruct file in the Key Indirection base directory.

`ki_destruct_util` doesn't really care if the data set in question is set up for automatic recovery (data encryption password same as self-destruct recovery password) or for manual recovery (data encryption password not same as self-destruct recovery password). It always treats the data encryption password and the self-destruct recovery password as different values. Whether they are really different depends on how the relevant data set was configured.

`ki_destruct_util` is your only option for creating the self-destruct data file corresponding to a data set which is configured for manual recovery. You may use `ki_destruct_util` to create the self-destruct data file corresponding to a data set which is configured for automatic recovery if you want. For either use case, if you create the self-destruct data file for a data set corresponding to the data encryption password X and then run v1pad with X as the master passphrase (and Key Indirection is not disabled), v1pad will recognize that there is already a self-destruct data file corresponding to the passphrase X, and it will not try to re-create it. v1pad will still delete the self-destruct data file after it completes setting up the internal Key Indirection data structures.

As a security best practice, only run `ki_dataset_util` and `ki_destruct_util` on a standalone computer that is not connected to the internet. There is no defense against an attacker who can get your self-destruct recovery password along with your v1pad key information and the appropriate Key Indirection data set.

`ki_dataset_util` and `ki_destruct_util` are able to use their own configuration directory with a corresponding configuration file. We'll call this directory the dataset utilities configuration directory. The default name and location for the dataset utilities configuration directory is `config_ki_dataset_util` in the current working directory. The configuration file must be named `ki_dataset_util.properties` and must be located in the configuration directory.

By default, Key Indirection is not enabled and the dataset utilities configuration directory does not exist. If you use Key Indirection frequently, you may want to create a dataset utilities configuration directory, create a configuration file

`ki_dataset_util.properties` in that directory, and specify values for the `KiMasterDirectory` and `KiDerivedDirectory` parameters in the configuration file. Creating a dataset utilities configuration directory with a corresponding configuration file will make it easier to run `ki_dataset_util` and `ki_destruct_util`.

It's probably easiest to create the dataset utilities configuration directory in the v1pad installation directory, since this is where `ki_dataset_util` and `ki_destruct_util` exist by default. You can also put the dataset utilities configuration directory in some other location, and reference it using the `ConfigurationDirectory` command-line argument when running either of the two data set utility programs.

Using Key Indirection

At a high level, automatic passphrase enhancement is probably the strongest reason for the average person to use Key Indirection. The single greatest weakness in most encryption systems is the user input. It's hard for people to create, remember, and type passphrases long and complex enough to be truly computationally infeasible to break. Key Indirection overcomes this by automatically generating a new pseudo-random passphrase based in part on the original passphrase and ensures that the new passphrase is computationally infeasible to break.

Key Indirection was designed to provide protection against key logger software, and to the extent that the Key Indirection data sets are not available to an attacker, it succeeds in that goal. Possibly the average person is not that concerned about their computer being infected with key logger software, but it's nice to know that there is a way to keep data private even if your computer is infected with key logger software.

Security best practices

As with most v1pad features, Key Indirection provides higher security at a cost. Like keyfiles, Key Indirection provides higher security, but causes v1pad to rely on additional data files. Just as a file encrypted using keyfiles cannot be decrypted without the proper keyfiles, a file encrypted using Key Indirection cannot be decrypted without the proper Key Indirection data files.

Operationally, the main cost of Key Indirection is needing to make the Key Indirection data files available to v1pad in a way that makes it hard for an attacker to get a copy. This means they should not be stored in the same place as the file encrypted using v1pad. If they are stored on the same computer as the encrypted file, an attacker who can get on your computer (either directly or over the internet) can just copy your Key Indirection data files along with the file(s) encrypted using v1pad and their key logger data.

The most practical solution to this is to store them on removable storage media (like a USB thumb drive), configure the Key Indirection base directory to point somewhere on

the removable media, have removable media mounted on your computer when running v1pad, and dismount the removable media as soon as you are finished running v1pad. That's a bunch of additional manual steps. Whether the extra security provided by Key Indirection is worth running additional manual steps is a personal decision.

The ideal use case for Key Indirection is to have a separate computer with no internet access dedicated to creating and maintaining the Key Indirection data files. If you have a separate computer that has no internet access, only a very determined attacker will be able to get any files from it, since they would need physical access to the computer first. Ideally this computer would be completely free of virus and spyware software and would have never been connected to the internet (even once). Practically speaking, this is difficult. You need to connect your computer to the internet in order to get software updates, and it may not be possible to be 100% sure a given computer is completely free of virus and spyware software. In spite of this, having a separate computer that is at least normally not connected to the internet and you have tried to make sure is clean is a huge step in the right direction. If this is not practical for you, then it's not practical. You have to live within your limitations.

If you have a separate computer that's not connected to the internet, you can keep copies of your Key Indirection data files on that computer and copy them to removable media from that computer. If you don't have a separate computer that's not connected to the internet, you should always write your Key Indirection data files directly to removable media, and you should only have the removable media mounted when performing Key Indirection data set operations or actually running v1pad. This greatly reduces the window of opportunity for an attacker to copy your Key Indirection data files.

Regarding manual -vs- automatic recovery of the self-destruct data file, manual recovery is much more secure and also much harder to use. Automatic recovery is easy to use and very secure if you protect your Key Indirection data files. Whether you want to go through the trouble of using manual recovery depends on how important it is that your data remain secure and how concerned you are that someone wants that data badly enough to go through a lot of trouble to get it. Possibly for many use cases the extra security provided by manual recovery is not worth the trouble, but the feature is there if needed.

Creating and managing data sets

In order to use Key Indirection at all, you need to plan ahead. Key Indirection works using data sets that you create in advance.

Whenever any program or web site requires a password, you have to provide one during setup and be able to produce the same password later on. Many people carefully choose their passwords ahead of time and either memorize them or store them in an encrypted repository and only memorize the password for that encrypted repository. These are reasonable and relatively secure approaches to managing passwords. Some people

always use the same password for every account or write their passwords down. Although these are easy ways to manage passwords, they are insecure and very much discouraged.

Key Indirection forces you to choose a passphrase ahead of time. You must already know your future v1pad passphrase (Key Indirection setup calls this your data encryption passphrase) in order to create a Key Indirection data set. If you want to configure your data set for manual recovery, you also need to know your (different) self-destruct recovery passphrase before creating your data set. If you want to configure your data set for automatic recovery, you just need to know your future v1pad passphrase and enter that same passphrase as both the data encryption passphrase and the self-destruct recovery passphrase.

If you use keyfiles and want to hide your keyfiles directory from key logger software, you need to have a plan for what your real and decoy keyfiles directories will be. You need to figure this out before creating the master data set which you will use to create the derived data set(s) that v1pad will actually read at runtime. You'll also need to determine the Key Indirection master password for this master data set.

When creating the master data set, enter your decoy keyfiles directory as <name> and your real keyfiles directory as <value>. If you will (or may) use more than one pair of real and decoy keyfile entries (either files or directories), you need to enter each one as a <name>=<value> pair when creating your master data set.

Even if you are not using keyfiles, you still need to enter at least one <name>=<value> pair when creating each master data set. If you're really not using keyfiles, it doesn't matter what <name>=<value> pair you enter.

Key Indirection on purpose allows you to choose not to remap all file and directory entries. You can remap all of them, remap some of them, or remap none of them. This is intended to make an attacker's life a little harder. They don't know (just from key logger data) whether v1pad is using the actual keyfile entries you provided or some other keyfile entries. If you want to play the game this way, probably you should provide more than one keyfile entry when running v1pad and remap at least one keyfile entry. Only providing one keyfile entry and not remapping it is a risky bluff.

Best Practices

This section of the manual describes best practices for using v1pad securely and with minimal effort.

General Comments

v1pad is designed to be computationally infeasible to break with brute force computation, but it can only provide this level of protection with the proper inputs. v1pad tries to provide very high security with minimal work, but it can't provide very high security with no work at all.

There is a balance between convenience and security. Know your own limitations and take advantage of v1pad features to the extent that you can remember what you did. There is no key recovery or key escrow feature in v1pad, so if you forget your passphrase or forget the exact set of keyfile entries you used and the exact order you entered them, you can also forget about ever decrypting your file.

The passphrase and the random block do not provide enough information to fully populate the keyspace. The keyfiles feature reads enough bytes from the keyfiles to completely populate the keyspace. Add-on keyfile entries make up for weak passphrases and known keyfiles by permuting the way the keyfiles are read, how the random block is encrypted, and the location of the random block so that none of these depend only on the passphrase and set of keyfiles. If you use real keyfiles and add-on keyfile entries, you will get the additional security they provide. If you don't use them, you won't get the additional security.

For ideal security, you should use a long and strong passphrase, specify at least one keyfile, make sure the total length of all keyfiles is at least 4k, and specify at least one add-on keyfile entry. You can do all of that very easily with three lines of input: one line for your passphrase, one line for your keyfiles directory, and one line for your add-on keyfiles entry.

Use your own keyfiles directory for best security. If you specify your own keyfiles directory, follow the recommendations in the section on keyfiles to select your keyfiles.

If you don't use your own keyfiles directory, use the default v1pad keyfiles directory. You can use the default keyfiles by typing a line containing a single period (.) character. By using the default keyfiles, you're not exactly revealing which keyfiles you're using, but it will be reasonable for an attacker to guess that you used the default keyfiles and try to decrypt your file by combining the default keyfiles with a set of commonly-used passwords. You should ideally include at least one add-on keyfiles entry to protect against this attack.

If three lines of input is too much to remember, skip the add-on keyfile entry and just specify your own keyfiles directory. Alternatively, use the default keyfiles plus an add-on keyfile entry. This is really three lines of input, but specifying a single line containing a period (.) character should be easy to remember. If even that is too much to remember, just enter a passphrase.

If your passphrase contains only ASCII characters, you would need to enter a 149-character passphrase to be sure that the passphrase complexity is higher than 1.0e+210. That is beyond what most people can remember, so don't worry about doing it. Just enter some reasonably-strong passphrase that you will be able to remember. Keep in mind that by not specifying any keyfile entries, you're greatly reducing the security provided by v1pad, but if a passphrase is all you can remember, just use a passphrase.

Especially if you only plan to use a passphrase, consider using Key Indirection. It's very highly recommended for people who only use passphrases because Key Indirection replaces the user-supplied passphrase with an internal passphrase a minimum of 149 characters long. The new internal passphrase should be strong enough to protect against any attacks. Even with the enhanced internal passphrase generated by Key Indirection, the v1pad keyspace won't even be close to fully-populated. Using keyfiles and add-on keyfile entries is better than just entering a passphrase. Using keyfiles and add-on keyfile entries with Key Indirection is even better than using them without Key Indirection.

Key Indirection best practices require keeping the Key Indirection data sets on removable media (like a USB thumb drive). If needing to insert and eject removable media prior to and after encrypting/decrypting files is not too much trouble for you, set up and use Key Indirection. This means create at least one Key Indirection data set, copy it to a removable media device, and modify your v1pad configuration file so that `KeyIndirectionMode` is set to "Enabled" or "Required" and `KeyIndirectionBaseDirectory` points to the appropriate directory on your removable media. Then make sure your removable media is mounted whenever you run v1pad, and you'll be using Key Indirection almost without even knowing it.

Choose wisely to balance what you can handle with the level of security you want. If you have to compromise one way or the other, stick with what you can handle. You don't want anyone else to read your files, but it does you no good to have a file that no one else can read if you can't read it either.

Encryption Algorithms

v1pad currently provides multiple encryption parameters and a few other encryption-related arguments. They all have default values. If you use the default values, you will get the maximum security and convenience v1pad has to offer, at the cost of also getting the highest overhead in terms of file sizes and encryption/decryption speed. Anything else is a trade-off between security and overhead.

This section assumes you read the rest of the manual, which gives detailed explanations of what all of these options do. Here we only discuss the high-level impact of different options.

The command-line argument that specifies the one-time-pad generation algorithm is `OneTimePadAlgorithm`. The default one-time pad generation algorithm is `PadKeyspace`. This probably the most secure way v1pad can generate one-time pad, but it's also the most expensive. `PadSimpleScrambled` is less complex and therefore faster. `PadSimpleDirect` is the least complex and therefore fastest. `PadUniform` has performance similar to `PadSimpleScrambled`, but its output is less complex than the output of the other three pad generation methods (it's still expected to be computationally infeasible to break).

You can also turn off one-time pad generation with `PadNone`. That's clearly faster than generating one-time pad, but it's also a lot less secure. This option allows encrypting with one of the plain text encryption algorithms and no one-time pad. This is probably only useful for performance testing or standalone evaluation of the plain text encryption algorithms.

The command-line argument that specifies the plain text encryption algorithm is `PlainTextAlgorithm`. The default plain text encryption algorithm is `PlainBLS`. Currently, the only other real plain text encryption algorithms are `PlainPVS` and `PlainAES128`.

All three are secure once combined with one-time pad bytes. BLS and AES are expected to be secure as standalone encryption methods, but PVS is not. However, PVS is significantly faster than both BLS and AES, and should be perfectly safe when combined with one-time pad encryption. AES appears to be a little less than twice as fast as BLS, but it's a little hard to say for sure, given that AES is running with a 16-byte block size in v1pad, whereas BLS is normally running with a 4096-byte block size. In theory, BLS as used in v1pad is many orders of magnitude more secure than AES because BLS uses much larger block sizes than AES does.

You can also turn off plain text encryption with `PlainNull`. That's clearly faster than encrypting the plain text, but it leaves the one-time-pad-encrypted file vulnerable to possible plain-text attacks. You can also turn off both one-time pad generation and plain text encryption. The result of this will be sort of like the original plain text (depending on other settings). The only obvious reason to do this is to test v1pad's I/O performance.

The command-line argument that specifies the plain text key refresh frequency is `PlainTextReKeyFrequency`. The default key refresh schedule for the plain text algorithms is `ReKeyInner`. This is clearly the most secure option, since it uses a new plain text key every time a block of input is encrypted. It's also the most expensive. `ReKeyMiddle` refreshes the plain text key 32 times less frequently than `ReKeyInner`, and still produces secure output by re-using the same plain text key but providing additional input so that the key is processed differently. `ReKeyOuter` refreshes the plain text key

128 times less frequently than `ReKeyMiddle`. `ReKeyOuter` uses the same strategy as `ReKeyMiddle` for safely re-using the plain text key: it passes different additional information each time the plain text encryption routine is called so that the plain text encryption routine processes the plain text differently every time.

For all of the above encryption parameters, the default values are the safest and also the slowest option.

You can get best performance by using `OneTimePadAlgorithm=PadSimpleDirect` to generate the one-time pad, using `PlainTextAlgorithm=PlainPVS` to encrypt the plain text, and using `PlainTextReKeyFrequency=ReKeyOuter` to refresh the plain text key as infrequently as is safe to do. This is expected to produce secure output more quickly, but it won't be quite as secure as using the default values. It's not currently clear whether the difference in security between these two extremes has any significant impact.

You can also select other values for these encryption parameters, with corresponding intermediate differences in security and performance.

You should normally use the default values for the encryption-related arguments `EncryptionModeFlag` and `SaveAdminBytes`, which are `EncryptionModeFlag=Normal` and `SaveAdminBytes` (as opposed to `noSaveAdminBytes`). It might make sense to set both `EncryptionModeFlag=Simple` and `noSaveAdminBytes` along with `OneTimePadAlgorithm=PadNone` to generate output files that are only encrypted with a plain text encryption algorithm (BLS, PVS, or AES128) and don't include any extra information, but the only obvious reason to do this would be for testing or evaluation.

Using simple mode omits both the random block and the padding feature, which speeds things up and makes the output smaller, but provides lower security and pushes the responsibility for making sure the one-time pad is never reused onto the user. Omitting the admin bytes at the start of the encrypted file makes the output file smaller but pushes the responsibility for remembering exactly what encryption parameters were used to encrypt the file onto the user. In general, neither one of these is a good idea.

Table Hash

This section describes Table Hash.

Introduction

Table Hash is a scalable secure hashing algorithm designed along with v1pad, but not currently used by v1pad itself. It was designed to provide minimal hash collisions and to scale to any output block size from 1 byte to almost 64k bytes.

Table hash was born out of frustration. I was frustrated that none of the SHA2 or SHA3 hashes has output sizes larger than 512 bits (64 bytes), and I was also frustrated by the fact that I had no way at all to know just how many hash conflicts the SHA2 or SHA3 hashes really produce.

One goal for v1pad is for it to be impractical for anyone to ever break it using brute force calculations, which I've interpreted to mean that the number of operations required to break any part of it should be higher than 1.0e+210. With this in mind, it's not surprising that I want a secure hashing algorithm that requires more than 1.0e+210 operations in order to have a better than 50% chance of finding a hash collision. None of the SHA2 or SHA3 hashes has this property, and I didn't really consider any other hashing algorithms. The most secure hashes in the SHA2 and SHA3 families are SHA2-512 and SHA3-512, each of which have 256 security bits, which translates to 1.2e+77 operations to have a better than 50% chance of finding a collision. This is a big number, but nowhere close to 1.0e+210.

Concatenating multiple, distinct hashes may not produce any more security than the weakest of the concatenated hashes. I think for the SHA2 family of hashes it's been proven that hash concatenation does not provide higher security. I don't believe that's been proven for the SHA3 family, but I don't think it's been proven that concatenating distinct SHA3 hashes provides higher security either. That left me with concatenating a SHA2-512 hash with a SHA3-512 hash, which may provide up to 512 security bits (1.3e+154) or may only provide 256 security bits.

My frustrations with the SHA2 and SHA3 hashes led me to spend many hours trying to come up with my own secure hashing algorithm that meets my requirements. Table hash is the result of all this work.

The Table Hash Algorithm

High-level

Table hash is very simple in concept. Instead of running a set of transformations on the input data to produce hash output, use randomly-scrambled lookup tables to produce hash output. This has an initial setup cost and requires a lot more memory to compute hash output for any given output block size, but it has the advantage of simplicity, and in theory can scale to very large output block sizes.

Initializing the CSPRNG

Starting with that idea, the next question is how to get the random numbers required to randomly-scramble the lookup tables. The solution in table hash is to use the input data itself to seed a Cryptographically-Secure Pseudo-Random Number Generator (CSPRNG), and then use the CSPRNG output to scramble the lookup tables.

For compatibility with other hashing algorithms, it is a requirement that table hash produce the same output if the "Collect" operation is called one time with a single string of X bytes or if the "Collect" operation is called multiple times with the same set of X bytes broken up into multiple strings. That means that somehow table hash needs to be able to accumulate input across multiple "Collect" calls, process the accumulated input whenever it fills up the memory used to accumulate it, and take care of any leftover data before computing the final hash value in the "Finalize" call. This requires a raw input buffer for accumulating the input data and a separate block input buffer to process the raw input in chunks of the block size.

Because table hash also initializes the CSPRNG from the raw input, there is a separate CSPRNG initialization buffer which is filled in when the raw input buffer fills up and used to initialize the CSPRNG and lookup tables before processing any data in the block input buffer. If the length of the raw input is longer than the raw input buffer, the 2^{nd} through Nth batches of raw input would seem to re-use the same CSPRNG state as the first batch. This in general is not good, as it lends itself to length extension attacks.

To address this issue, table hash has both a secure mode and a non-secure mode. In non-secure mode, table hash initializes the CSPRNG and lookup tables exactly once when processing the first raw input buffer. In secure mode, table hash re-initializes the CSPRNG and lookup tables every time it processes a new raw input buffer. Clearly it takes more work to re-initialize every time a raw input buffer is processed, and not all use cases for hashing require secure output, so secure mode is an optional feature. Note that secure mode and non-secure mode produce the same output for input strings with length less than or equal to the raw input buffer length, since for input strings of this length only one raw input buffer is ever processed.

To support both secure and non-secure modes while at the same time guarding against flaws similar to running a block cypher in "Electronic Codebook" mode, table hash uses one algorithm the first time it initializes the CSPRNG and lookup tables, but it uses a

different algorithm when it re-initializes the CSPRNG and lookup tables. The first-time initialization algorithm is: fill the CSPRNG initialization array based on the raw input data, reset the CSPRNG state using the initialization array, write an ordered set of values to the lookup tables, and finally scramble the ordered set of values in the lookup tables using the CSPRNG output. The re-initialization algorithm for the CSPRNG is: fill the CSPRNG initialization array based on the raw input data, modify the contents of the initialization array by XORing them with the existing CSPRNG state, and finally reset the CSPRNG state using the modified initialization array. This makes the re-initialized CSPRNG state depend not only on the contents of the new initialization array but also on the prior CSPRNG state. Similarly, the re-initialization algorithm for the lookup tables is: do not write an ordered set of values to the lookup tables – instead, just re-scramble the lookup tables starting from their current already-scrambled state. This makes the re-scrambled lookup tables depend not only on the new CSPRNG state but also on the earlier CSPRNG state.

Above we say that table hash fills in the CSPRNG initialization array based on a batch of raw input data. At the detail level, the size of the raw input data and the size of the raw input data array are variable, but the size of the CSPRNG initialization array is fixed. Table hash uses the ISAAC CSPRNG (the same one used by v1pad), which has an initialization array size of 1k bytes. The size of the raw input data buffer is always at least as big as the size of the CSPRNG initialization array. It must also always be at least as big as the input block size. For smaller block sizes, the CSPRNG initialization array is larger than the input block size, so the raw input buffer has the same size as the CSPRNG initialization array. For larger block sizes, the input block size is larger than the CSPRNG initialization array size, so the raw input buffer has the same size as the input block size.

The raw input buffer is always completely filled in prior to filling in the CSPRNG initialization array, unless there is not enough raw input data to fill the raw input data buffer. For use cases where the raw input data is smaller than the size of the CSPRNG initialization array, table hash repeats the raw input data as many times as is required to fully populate the CSPRING initialization array. For use cases where the length of a chunk of raw input data exactly matches the length of the CSPRNG initialization array, table hash exactly fills the initialization array with the raw input data. This can only happen for smaller input block sizes, when the raw input buffer size and the CSPRNG initialization array size are identical. For use cases where the length of a chunk of raw input data is longer than the length of the CSPRNG initialization array, table hash first fills in the CSPRNG initialization array fully, then XORs the remaining raw input data bytes with the fully-populated CSPRNG initialization array (using modular arithmetic to access the CSPRNG initialization array) until it runs out of raw input data.

Input block padding

The next question that comes up is how to pad the last input block before mixing it into the hash output accumulator. The solution in table hash is to add a set of special-case

lookup tables specifically to handle padding, so that each missing/null input byte in the final input block is replaced by a random padding value, and then process the random values as if they were part of the original input.

The fact that there is any padding at all implies that there must be input value collisions between full input blocks and padded input blocks, which in turn will produce intermediate output value collisions when processing the final input block, which in turn will produce full hash output value collisions if all of the prior input blocks were identical. Specifically, a full input block of size X+Y original bytes and a padded input block with the same X original bytes plus Y padding bytes, will produce intermediate value collisions when they both have the exact same final set of Y bytes.

This would seem to provide an easy way to generate hash conflicts for almost-full final input blocks. The closer the final input block is to a full block, the easier it is to find a conflict. For a block size of N bytes, if the final input block contains N-1 bytes, there are only 256 possible values that could produce a conflict with a full block. With a less-full partial block, the problem becomes exponentially harder, but may still be tractable. If the final output block contains N-4 bytes, there are now 256^4 values to look at, which is about 4 billion. It's computationally infeasible for a person to try 4 billion values, but it's not at all computationally infeasible for a computer to look at 4 billion values. At what point it becomes computationally infeasible for a computer to successfully generate hash collisions for partial input blocks depends on the size of the complete block, the power of the computer, and how much time you have.

All of this is mostly addressed in table hash by the fact that the input string is used to seed the CSPRNG, so adding or subtracting a byte of input at least in theory completely changes the set of pseudo-random numbers that the CSPRNG produces, which would seem to erase the possibility of easily finding a conflict just based on padding bytes. However, there are some corner cases where adding additional input characters does not change the CSPRNG output (because of the algorithms used to fully-populate a 1k CSPRNG initialization buffer from smaller or larger input strings), so it seemed like a good idea to eliminate the conflicts caused by padding, if possible.

The solution for this in table hash is to include an additional byte or two in the table hash output which makes the table hash output unique even when an input block is padded. A simple way to do this is to just add a null count to the end of the table hash output. If the block wasn't padded, the null count is set to zero. If the block was padded, the null count is set to the number of padding bytes. That gives a range of values between zero and N for an N-byte input block size. The range is not zero to N-1, since for the corner case of a null input string, the null count would actually be N. That allows table hash to use only one byte to track the null count for an input block size of up to 255 bytes. For input block sizes from 256 up to the currently-supported maximum, table hash needs a two-byte integer to track the null count.

This scheme completely eliminates hash output conflicts caused by padding at the cost of a slightly higher output block size, but it's a little too simple. Just looking at a single

uniqueness byte for now, if it's always at the end of the input, we can tell how long the last block was by just knowing the original block size and the value of the last output byte. That can be fixed somewhat by scrambling the mapping between the final intermediate hash value and the output hash value, so that the null count occurs in a random location in the output. Even that leaves a byte in the output that will tend toward lower or zero values, which is not good. The next step is to add yet another table of randomly-scrambled values, use the null count as an index into that table, and add this randomly-selected value to the output instead of the actual null count. This solution scales to null counts of 256 and higher by using two tables of randomly-scrambled values, one for the high-order byte of the null count and another for the low-order byte.

Because this whole concept of adding a uniqueness byte to the hash output is new, because it may not be strictly required for security purposes, and because for some applications it may not be important, having uniqueness byte(s) in the table hash output is an optional table hash feature.

The table hash algorithm takes as input parameters the output block size and a flag toggling the use of uniqueness bytes. To keep the same output block size and still allow for one or two uniqueness bytes, it sets its internal block size to the output block size minus one or two bytes. This makes things a little strange around the 256-byte boundary. With 256 output bytes, there is one uniqueness byte and a 255-byte internal block size. With 257 output bytes, there are two uniqueness bytes and still a 255-byte internal block size. Not until the output block size reaches 258 output bytes does the internal block size start increasing again to 256 bytes. Table hash with no uniqueness bytes has an input block size equal to the output block size.

Separate Initialization Array

For use cases where it is not important for the table hash output to be cryptographically secure (like for just loading a bunch of data into a hash table), it may be useful to take advantage of the speed of the hashing operations in table hash while avoiding the slowness of having to re-initialize the CSPRNG and look up tables for every input string. To cater to this use case, table hash allows the CSPRNG and lookup tables to be initialized from a separate initialization array in non-secure mode. That allows the set of input values to be hashed more quickly by re-using the same lookup table state for each input string. It also allows the exact hash values produced to hold stable for a given set of input data, regardless of which order the data is hashed in. A different stable set of hash values can be produced by just changing the contents of the initialization array.

Naming Convention

Coming back to the beginning, table hash was designed to be just another secure hashing algorithm, with the only difference being that it scales easily to larger and smaller block

sizes than most other secure hashing algorithms. Keeping compatibility with other secure hashing algorithms in mind, I came up with the following naming conventions:

- TBHnnnn (or TBH-nnnn) stands for table hash in secure mode, with uniqueness byte(s), and output size of nnnn bits
    - We use the most obvious name for (arguably) the most secure mode
    - Including space for the uniqueness byte(s) makes table hash a little weaker than other secure hashes with the same output size. For example, TBH512 really only uses 63 bytes for actually computing the hash, with one byte reserved for later, so it's weaker than SHA3-512, which uses 64 bytes for computing the hash. The fact that table hash can scale to much higher block sizes makes up for this once we get to output block sizes of 65 bytes and higher
- TBHXnnnn (or TBHX-nnnn) stands for table hash in secure mode, excluding uniqueness byte(s), and output size of nnnn bits
    - Excluding space for the uniqueness byte(s) makes table hash in theory the same strength as other secure hashes with the same output size, if in fact table hash in this mode is really secure. Taking the example above, TBHX512 uses the full 64 bytes for computing the hash, so it is in theory equivalent in security to SHA3-512.

The non-secure mode of table hash and the non-secure mode of table hash with a separate initialization vector are not covered here. Keep in mind I wanted to come up with names which indicate more-or-less what other secure hashing algorithms table hash with a given output bit size is drop-in compatible with, and other secure hashing algorithms don't have non-secure modes.

Caveats

I'm not at all satisfied with the quality of table hash output for small block sizes with pathological inputs. I see patterns in the intermediate output.

I tried fixing this by adding an additional pair of scrambles and offsets table so that the block level hash output is merged into the hash output accumulator differently for large enough block sizes and by transforming the byte values prior to merging them (with up to 37 possible transformations). That seemed to help down to about 4-byte blocks. For three-byte-blocks and lower, I still see repeating patterns every 37*2*<block_size> intermediate outputs. Maybe that could be fixed with another layer of scrambles and offsets tables, and/or with bumping the byte-level transformations up to some number higher than 37.

For now, I'm not going to try it. I already tried swapping out the current 37 byte-level transformations with 37 different/re-ordered ones, and that seemed to actually make things worse. The table hash output for larger block sizes seems fine, that was my main goal anyway, and there's always the possibility of investing more effort in this later.

Using Table Hash

The Table Hash code is completely contained in `crypt/crypt_tbh.c` and in `crypt/crypt_tbh.h`. That doesn't mean it compiles and builds standalone, but it is at least all located in one place. I think it only depends on v1pad utility routines from the `fwk` directory, on the code for the ISAAC CSPRNG in the `thirdparty` directory, on a few macro values from `crypt/crypt_macros.h`, and on two routines from the `v1pad` directory.

The utility routine `checksum_util` by default computes TBH and TBHX hashes along with various SHA2 and SHA3 hashes. The number of output bytes for the table hash hashes in `checksum_util` is currently the same as the number of output bytes for the two SHA3 shakes. `checksum_util` is described in more detail in the appendix.

There is also a standalone utility called `tbhsum` which can be used to run table hash. At a high level, `tbhsum` was designed to be more or less drop-in compatible with `sha256sum`. The UI is not the same, but the default functionality is similar, and it can be made to produce compatible output. The rest of this section describes how to use `tbhsum`.

The tbhsum utility

`tbhsum` by default computes the TBH256 hash of the input data and writes the result to standard output as hex digits.

Just typing `tbhsum` at the OS prompt with no arguments will cause `tbhsum` to list a summary of its command-line arguments. To get more complete help information, type `tbhsum help` at the OS prompt.

Unlike `sha256sum`:

- `tbhsum` has a mandatory `InputMode` argument which must be set to one of: "File", "Stdin", or "Prompt" (exactly as written here, case-sensitive)
    - To make `tbhsum` compute the checksum of a named file, specify `tbhsum File InputFile=<filepath>`
    - To make `tbhsum` compute the checksum of whatever you've piped to it on standard input, specify `tbhsum Stdin`
    - To make `tbhsum` prompt you for input strings, specify `tbhsum Prompt`
- `tbhsum` does not support multiple input filenames
- By default, `tbhsum` does not list the input file name after the hex digits for the hash output
    - To make `tbhsum` list the input file name like `sha256sum` does, specify `ShowInputFile` on the `tbhsum` command line after the mandatory arguments

    - The input file name is set to a hardcoded string when `InputMode=Stdin` or `InputMode=Prompt`. There is a different hardcoded string for each of these two input modes
- By default, `tbhsum` includes two comment lines giving the table hash parameters and the length of the input data before the start of the hex digits for the hash output
    - To remove these two comment lines from the `tbhsum` output, specify `noShowParameters` on the `tbhsum` command line after the mandatory arguments.
    - The format of the comment giving the table hash parameters is something like:

```
# Table Hash 256 bits u1 secure mode
```

        - The number of bits, the `u0|u1|u2` value, and secure -vs- non-secure mode can all change depending on optional `tbhsum` input values
    - The format of the comment giving the input data length is something like:

```
# 3613 input bytes
```

        - The only thing that can change in this output line is the number of input bytes
- `tbhsum` has many optional arguments which allow you to use most of the table hash features

tbhsum optional arguments

To make `tbhsum` run with larger or smaller output block sizes than 32 bytes (256 bits), specify `OutputBytes=<positive_integer>` on the `tbhsum` command line after the mandatory arguments. The minimum table hash output block size with uniqueness bytes is 2 bytes (one byte to really run the hash algorithm and one uniqueness byte). The minimum table hash output block size without uniqueness bytes is 1 byte. The maximum table hash output block size is 65282 with uniqueness bytes and 65280 without uniqueness bytes. This translates to an internal block size of 65280 (64k - 256 bytes) in both cases.

A table hash output block size of 256 bytes (2048 bits) should be plenty secure. 256 output bytes provides at least 127 security bytes, requiring 7.0e+305 operations to have at least a 50% chance of finding a hash collision. Higher block sizes are supported because it was not that hard to do, in case they are useful for some reason, and because at one point I thought I might use table hash with a 4k output block size internal to v1pad.

`tbhsum` with maximum and minimum output block sizes works fine. The unit tests for the base table hash routines include running table hash with various output block sizes between 1 byte and 4k bytes.

To make `tbhsum` run in non-secure mode, specify `noSecureMode` on the `tbhsum` command line after the mandatory arguments.

To make `tbhsum` run without any uniqueness bytes, specify `noIncludeUniquenessBytes` on the `tbhsum` command line after the mandatory arguments.

The `u0|u1|u2` value in the table hash parameters output comment indicates the number of uniqueness bytes in the table hash output. When `tbhsum` runs with no uniqueness bytes, the `u0|u1|u2` value in the table hash parameters output comment will be set to `u0`. When table hash runs with uniqueness bytes (the default), the `u0|u1|u2` value in the table hash parameters output comment will be set to `u1` for output block sizes up to 256 bytes. It will be set to `u2` for output block sizes of 257 bytes and higher.

To specify a separate CSPRNG initialization data file for `tbhsum` in non-secure mode, specify both `noSecureMode` and `InitFile=<filename>` on the `tbhsum` command line after the mandatory arguments. The `InitFile=<filename>` argument is ignored when running `tbhsum` in secure mode.

tbhsum examples

Running `tbhsum` on a file with the same output format as `sha256sum`:

```
sha256sum Input.txt
effba7ee6203d678f4de22c0feb4925e97ecf52acf5dda21b8a8fe295d827ba6
Input.txt

tbhsum File InputFile=Input.txt noShowParameters ShowInputFile
cc8646b355e74bdad3b2925187e2127cc1a8173ca38a1a4594714f72bbbdabb8
Input.txt
```

The output format is the same, but the actual output of `sha256sum` and `tbhsum` is different. This is expected because SHA2-256 and TBH-256 are different hashing algorithms.

Running `tbhsum` on a file with default output format:

```
tbhsum File InputFile=Input.txt
# Table Hash 256 bits u1 secure mode
# 3613 input bytes
cc8646b355e74bdad3b2925187e2127cc1a8173ca38a1a4594714f72bbbdabb8
```

Running `tbhsum` on user-entered text:

```
tbhsum Prompt

Enter one or more lines of text to hash.
Enter an empty line after last line of text.

Enter String: Hi Rick
```

```
Enter String: How are you?

Enter String:

# Table Hash 256 bits u1 secure mode
# 19 input bytes
4f8dd6628572c205a282ea1f04ee44575e66edc609262e226b22414841468928
```

Some blank output lines were skipped to make this easier to read.

Running `tbhsum` on standard input:

```
echo "ha ha" | tbhsum Stdin
# Table Hash 256 bits u1 secure mode
# 9 input bytes
dfe5c794c4e1661f7248f434176bd2795f2fa9c25228c995c287a4bad2a21c61
```

Running `tbhsum` on user-entered text over an input pipe:

```
echo "ha ha" | tbhsum Prompt

Enter one or more lines of text to hash.
Enter an empty line after last line of text.

Enter String:

Enter String:
# Table Hash 256 bits u1 secure mode
# 8 input bytes
0caf7971eeb3eda014e7c6d30197fa3ea39252909dfc7a8f452cd315388dcaaf
```

This is not the same result as running `tbhsum Prompt` with the same input values because effectively it's not the same input (notice that the input sizes are different). That's because when `InputMode=Stdin`, `tbhsum` treats standard input as a file, whereas when `InputMode=Prompt` and the user-entered text comes from standard input, `tbhsum` treats standard input as a set of strings on individual input lines terminated by a single empty input line.

Appendix

This appendix contains reference material.

Sample get_sys_info_v1p output

Here is some sample get_sys_info_v1p output.

Windows Sample

Sample output files on a Windows box

System_ID_Info_Long.txt

```
==Motherboard


Manufacturer=Dell Inc.
Product=052K07
SerialNumber=/CSY1FC2/CN129636500139/
Version=A00



==BIOS


Manufacturer=Dell Inc.
Name=1.2.3
SerialNumber=CSY1FC2



==CPUs


DeviceID=CPU0
Name=Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz
ProcessorId=BFEBFBFF000406E3



==Memory


BankLabel=BANK 0
Capacity=8589934592
Manufacturer=Micron
PartNumber=16KTF1G64HZ-1G6P1
```

```
SerialNumber=12151215


BankLabel=BANK 2
Capacity=8589934592
Manufacturer=Micron
PartNumber=16KTF1G64HZ-1G6P1
SerialNumber=12121212



==DiskDrives


FirmwareRevision=AX0P3D
InterfaceType=IDE
MediaType=Fixed hard disk media
Model=TOSHIBA MQ01ABD100
SerialNumber=           46ETTZTZT



==NetworkInfo



==TheComputer


IdentifyingNumber=CSY1FC2
Name=Inspiron 5559
UUID=4C4C4544-0053-5910-8031-C3C04F464332
Vendor=Dell Inc.
Version=



==OSinfo


Caption=Microsoft Windows 10 Home
CSName=DESKTOP-S0HC011
CurrentTimeZone=-480
InstallDate=20161125130451.000000-480
MUILanguages={"en-US"}
OSArchitecture=64-bit
RegisteredUser=Rick
SerialNumber=00342-20300-16768-AAOEM
TotalVirtualMemorySize=19156896
```

System_State_Info.txt

```
==CPUinfo
```

```
LoadPercentage=7


==OSinfo


FreePhysicalMemory=14261148
FreeSpaceInPagingFiles=2490368
FreeVirtualMemory=16540892
LastBootUpTime=20170228045543.493610-480
LocalDateTime=20170228051814.919000-480
SizeStoredInPagingFiles=2490368


==ProcessInfo


Handle=0
KernelModeTime=49403750000
Name=System Idle Process
PageFaults=2
PageFileUsage=0
UserModeTime=0


Handle=4
KernelModeTime=234375000
Name=System
PageFaults=14006
PageFileUsage=128
UserModeTime=0


<Skipped many similar entries>


Handle=7340
KernelModeTime=156250
Name=WMIC.exe
PageFaults=3075
PageFileUsage=3364
UserModeTime=156250
```

System_ID_Info_Short.txt

```
==Motherboard


Manufacturer=Dell Inc.
Product=052K07
SerialNumber=/CSY1FC2/CN129636500139/
Version=A00
```

```
==BIOS


Manufacturer=Dell Inc.
Name=1.2.3
SerialNumber=CSY1FC2



==CPUs


Name=Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz
ProcessorId=BFEBFBFF000406E3



==Memory


Capacity=8589934592
Manufacturer=Micron
PartNumber=16KTF1G64HZ-1G6P1
SerialNumber=12151215


Capacity=8589934592
Manufacturer=Micron
PartNumber=16KTF1G64HZ-1G6P1
SerialNumber=12121212
```

Linux Sample

Sample output files on a Linux box

System_ID_Info_Long.txt

```
==Motherboard
      Manufacturer: Dell Inc.
      Product Name: 052K07
      Serial Number: /CSY1FC2/CN129636500139/
      Version: A00
==BIOS
      Vendor: Dell Inc.
      Version: 1.2.3
      Release Date: 08/25/2016
      BIOS Revision: 1.2
==CPUs
      Socket Designation: U3E1
      Family: Core i7
      Manufacturer: Intel(R) Corporation
      ID: E3 06 04 00 FF FB EB BF
```

```
      Signature: Type 0, Family 6, Model 78, Stepping 3
      Version: Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz
==Memory
      Size: 8192 MB
      Locator: DIMM A
      Bank Locator: BANK 0
      Type: DDR3
      Type Detail: Synchronous
      Manufacturer: Micron
      Serial Number: 12151215
      Asset Tag: 9876543210
      Part Number: 16KTF1G64HZ-1G6P1
      Size: 8192 MB
      Locator: DIMM B
      Bank Locator: BANK 2
      Type: DDR3
      Type Detail: Synchronous
      Manufacturer: Micron
      Serial Number: 12121212
      Asset Tag: 9876543210
      Part Number: 16KTF1G64HZ-1G6P1
==DiskDrives
             description: ATA Disk
             product: TOSHIBA MQ01ABD1
             vendor: Toshiba
             version: 3D
             serial: 46ETTZTZT
==NetworkInfo
                description: Wireless interface
                product: Wireless 3160
                vendor: Intel Corporation
                version: 83
                serial: 2c:6e:85:f3:80:fe
                description: Ethernet interface
                product: RTL8101/2/6E PCI Express Fast/Gigabit Ethernet
controller
                vendor: Realtek Semiconductor Co., Ltd.
                version: 07
                serial: 84:7b:eb:44:f0:0c
==TheComputer
    name: rick-inspiron-5559
    description: Laptop
    product: Inspiron 5559 (06B2)
    vendor: Dell Inc.
    serial: CSY1FC2
    uuid: 44454C4C-5300-1059-8031-C3C04F464332
       product: DELL 78V9D63
       vendor: Panasonic
       version: 03/24/2016
       serial: 0EA1
       capacity: 37290mWh
==OSinfo
OS Info: Linux rick-Inspiron-5559 4.4.0-47-generic #68-Ubuntu SMP Wed
Oct 26 19:39:52 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
User Info: uid=0(root) gid=1000(rick)
groups=1000(rick),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpa
dmin),128(sambashare)
```

```
Timezone: -08:00
Date/time in UTC: Wed Mar  1 05:49:10 UTC 2017
```

System_State_Info.txt

```
==CPUinfo
 load average: 0.33 0.24 0.19
==OSinfo
Mem: 16322412 824856 13616296 347232 1881260 14810320
Swap: 16666620 0 16666620
Date/Time: Tue Feb 28 21:49:10 PST 2017
Uptime: 5:39
==ProcessInfo
A Process: 1 0 root dd2a3153 968c7e20 1 46293 185172 5876 Ss /sbin/init
splash
A Process: 2 0 root 00000000 00000000 0 0 0 0 S [kthreadd]
A Process: 3 2 root 00000000 00000000 0 0 0 0 S [ksoftirqd/0]
A Process: 4 2 root 00000000 00000000 0 0 0 0 S [kworker/0:0]

<skipped many similar entries>

A Process: 6195 6194 root 0f9676fa 9ae507c8 3 1127 4508 1688 S+ /bin/sh
./GetSystemState.sh ./dmidecode_info.txt ./lshw_info.txt
./System_State_Info.txt
A Process: 6207 6195 root d578d9b0 91b3cb98 2 11108 44432 3264 R+ ps --
no-headers -eo pid,ppid,user,eip,esp,psr,sz,vsz,rss,stat,args
A Process: 6208 6195 root c1f3f9b0 cb6440e8 3 7956 31824 3512 S+ awk
{print "A Process:", $0}
A Process: 6209 6195 root dd5bc9b0 a48c6cb8 0 3596 14384 700 S+ tr -s
[:blank:]
```

System_ID_Info_Short.txt

```
==Motherboard
      Manufacturer: Dell Inc.
      Product Name: 052K07
      Serial Number: /CSY1FC2/CN129636500139/
      Version: A00
==BIOS
      Vendor: Dell Inc.
      Version: 1.2.3
      Release Date: 08/25/2016
      BIOS Revision: 1.2
==CPUs
      Socket Designation: U3E1
      Family: Core i7
      Manufacturer: Intel(R) Corporation
      ID: E3 06 04 00 FF FB EB BF
      Signature: Type 0, Family 6, Model 78, Stepping 3
      Version: Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz
==Memory
      Size: 8192 MB
      Manufacturer: Micron
```

```
Serial Number: 12151215
Asset Tag: 9876543210
Part Number: 16KTF1G64HZ-1G6P1
Size: 8192 MB
Manufacturer: Micron
Serial Number: 12121212
Asset Tag: 9876543210
Part Number: 16KTF1G64HZ-1G6P1
```

End of sample get_sys_info_v1p output files.

v1pad utility programs

v1pad ships with a few other utility programs. These were created for use with v1pad development and packaging. They are shipped along with v1pad in case they are useful.

checksum_util

The Checksum Utility program computes one or more checksums on an input file and saves the results in a properties file. By default, it computes all of the SHA-2 and SHA-3 hashes, two variants of Table Hash hashes, and both SHA-3 shakes on the input file. It is also able to compute MD5 and SHA-1 hashes on the input file.

The number of bytes output by the SHA-3 shakes and the number of output bytes for Table Hash are both controlled by the optional command-line argument `ShakeLength`. The default value for `ShakeLength` is 256 bytes, but it can be set to any value from zero to 645. Zero re-maps to 256 internally, so zero really just means "use the default value".

The Checksum Utility can also be used to compare all of the hash values stored in a properties file with the results of computing the same hash values on an input file. This provides an easy way to verify that the current input file matches or does not match the saved hash values.

Sample Checksum Utility run

For this example, I copied some other PDF file to Junk.pdf in the current working directory and ran Checksums Utility on it. Here are the command lines, program output, and sample output file.

Generate checksums properties file:

```
./checksum_util Generate Junk.pdf Junk_checksums.properties

Date and Time are: #I# Tue, May 22, 2018  02:30:33 PM  (380212) #i#
```

```
Date and Time are: #I# Tue, May 22, 2018  02:30:33 PM  (395764) #i#


Checksum Utility
  Version 1.0
  Copyright 2018 Richard Joseph Lotero


Computed checksums and wrote to properties file
"Junk_checksums.properties".

Date and Time are: #I# Tue, May 22, 2018  02:30:39 PM  (193273) #i#
```

Compare current file with saved checksums properties file:

```
./checksum_util Compare Junk.pdf Junk_checksums.properties

Date and Time are: #I# Tue, May 22, 2018  02:32:00 PM  (795733) #i#

Date and Time are: #I# Tue, May 22, 2018  02:32:00 PM  (795733) #i#


Checksum Utility
  Version 1.0
  Copyright 2018 Richard Joseph Lotero



All hashes match (12 of 12 match).


Date and Time are: #I# Tue, May 22, 2018  02:32:06 PM  (608851) #i#
```

Compare some other file with saved properties file:

```
echo "Ha Ha" > x.txt

type x.txt
Ha Ha

./checksum_util Compare x.txt Junk_checksums.properties

Date and Time are: #I# Tue, May 22, 2018  02:36:07 PM  (212744) #i#

Date and Time are: #I# Tue, May 22, 2018  02:36:07 PM  (212744) #i#


Checksum Utility
  Version 1.0
  Copyright 2018 Richard Joseph Lotero



At least one hash does not match (0 of 12 match).
```

Contents of sample checksums properties file:

```
type .\Junk_checksums.properties
#
# Properties file generated on: 20180522-14:30:39.193273
#

#
# This properties file lists checksums for an input file
#
# The checksums here correspond to the input file: Junk.pdf
#

SHA2_224=24C1BB28E6EB7FFCFE74AED1F2D52C96FE70509B6B126274C422826B

SHA2_256=E4FF68D4A66FD886CC37C48E902F764F51A63F89F86F4F4750117189B4E42A
0E

SHA2_384=C8F27CA163C0024B723F5DAE96D4F2EC17C2C052D56EED9EAB8D1737F810EF
835515E8EF272AE15AB0620432C5EDF39B

SHA2_512=596F464EF3AE6B445198D4854B2AD2DAEBFA4A875A6469D8BFF5A6F8070F0D
B504254335F1D0F500AA39C1B03478C578C064AAD7FD3888F64754206C02EE6B63

SHA3_224=FAF64DCB00E0327CC37FC85C8B29E8492C024950FCA7B2960D9DFE5E

SHA3_256=BECAFC6467CF583EDD8E7D2AEC2206EF2D712196D96784ADF9C590EF66246A
F0

SHA3_384=2F1A7BA8F390C5D7C8683F5424EE0A8DFC751F067F0FB75341D21AB0B5A6EB
43B4083FCE5A56DE1BCF19A6941EC8F00D

SHA3_512=A5E5616FE5C85E719C19611506BE17EAE3723325777FA98495939EF91AEAC2
3D5F449D9BBC3894D13A2471297A727124CA4071469F3D5CAB3523D6E43BD20DE8

TBH=5AC9B8D846D43AB146503E2B97C16989691B037B4FED7C32E96FD4118C1E4E53466
582E3BD71272B247BD1D3552FCCDF8F8177D1CACAE58B1C8AE8912633E306F9B2250D18
6D4251E70DA696D142430CA40C62362B3EF6B8567FD5D95677AB51D89F16CE03F9CFF2E
1D9C5924A3C53D872D966C583D3F2AB5E46A9E7467303BB1D4F2CBF179C8C51C81E6571
DA0A280A731C3C35BDC60AD31B0EF9138418D88F06EA67CC79C72B925006158D9FD3498
FB32A513B19F826D42E1D399A648F4317ABB35A7508DBF1F96A695D431DE149AC9AE81E
B6052BE0957B9BF7910EAC01EB6B16125460AAB66A5A8C6A3F6BA616B9171CDC91F24CA
8C275841962FA861A8A

TBHX=B661616DB900720C7E8085AF69D6282617C4E6DB130A5F9C7487BD8822E16BB916
271B935B47FE4892EFEB4B93107EBA87247B460D6F62A7C40F4A0258D9A779AE1598D99
64AE12D66284DDB6D6696312ABB4B5F7A938175411B19E76D4300CB1814193E13332EA6
43BC0D0B1188FA599EF73A53B3FD6362CAC4ED4D9A1F59203A03C360F60E8F3D622A474
EFDE46D9246AED285D89EE6D1CDABC34A68C2483B4755E14278BCFE37EE97D90DD04E43
DA937A92AFA5EFC0515A7C7D46AC1CC8A99949AABEA0AF999C5FF0C071A39F65EB52919
0A3CC39E07E3D6A4A2FAB24BCB35E8667575519650EC68A3C83A2947B8CB428206A3935
E6F1DF053831C78849A8

SHAKE128=1FA1CCAD18058C0AA7944E7B2E658B6315390B4858EA76075831B59379C836
97F443015D328E444D8109CAE15F34DBE9D5B2D7C6A9BE5FD94C414A6B64B388E1F0236
A11376831DAFFAB574748CD80EB82CABF68EECEB8C9064F74C18DB35331C06DC7F72710
```

```
C20072FC852C45120FA6343F3A8B479B30D8356A57651ABD9ABD0A42D227445CB8796AE
250E92B55BF69151B69CB7743B25744E30C5D98EC93ED7EA023D7FC90D6EE505854895C
0CB1690BE97D492E0D92A9C5E6FEBD784CC63024FE7F87BBCAC7DD42C4CAD6C23815B4F
0A5FFADC5A34AF33CCF99EA620A8E315B8423F2AFE5175CF877E0F327DDA295D273A29B
9FC28015DE11F79C12D0F0C2

SHAKE256=6AFD94B9CAA1FD3911BCD8E99127ED8157110038573356CD0AABEC3BA76C9C
61B5DE71E082D31E9CC60355A66F1D62B3C4A1B7CD143D3916A293DFB352268E98F3467
1B0F07F7562FFBD5677DD61BED8C2C56B0C99700B7A2C9142DA20B79283C47239A9CBDE
A7B493BE1942B5AECA16903FFFFA1E55586CF9EFBCC92EFEE98F8A1D2C376E89DE6D4E0
23299642666EE78117CC12C02BABE443DA9053C7F5BE82741468A202D55C504BB0D0F0B
15D9B99B8A17749EA48CCCDACF4625A1407D623E88C49411C34F479D6023792D87C3684
D3ACAF423B756233D4BA1872BC792ECB2CE5E051A3A3638406DBF5DD6B900D959432FFC
A4063B3621FA2ABAE6C25935

#
# End of properties file
#
```

make_auth_file

make_auth_file is a standalone utility program that creates and validates Auth files. It was created earlier when one of the v1pad features required an Auth file. The feature didn't come over with the move to open source, but the utility still seems somewhat useful, so I kept it.

Sample Make Auth File run

For this example, I just ran make_auth_file from the v1pad top directory. Here are the command lines, program output, and sample output file.

Creating an Auth file

```
C:\> make_auth_file.exe . TestAuth.txt

Date and Time are: #I# Sat, May 20, 2017  12:02:52 PM  (955152) #i#

Date and Time are: #I# Sat, May 20, 2017  12:02:52 PM  (955152) #i#


Authorization / Authentication File Creation Utility
  Version 1.0
  Copyright 2017 Richard Joseph Lotero


Generating Authentication / Authorization File...


Enter Authorization / Authentication Passphrase: *********
```

```
Successfully generated authorization / authentication file
"TestAuth.txt".


Validating Authentication / Authorization File...


Enter Authorization / Authentication Passphrase: *********


Authorization / Authentication file "TestAuth.txt" validated
successfully!


Validation succeeded.

Date and Time are: #I# Sat, May 20, 2017  12:03:01 PM  (831079) #i#

C:\>echo %errorlevel%
0
```

Validating an Auth file (entering the correct password):

```
C:\>make_auth_file.exe . TestAuth.txt JustValidate

Date and Time are: #I# Sat, May 20, 2017  12:08:05 PM  (942030) #i#

Date and Time are: #I# Sat, May 20, 2017  12:08:05 PM  (942030) #i#


Authorization / Authentication File Creation Utility
  Version 1.0
  Copyright 2017 Richard Joseph Lotero


Validating Authentication / Authorization File...


Enter Authorization / Authentication Passphrase: *********


Authorization / Authentication file "TestAuth.txt" validated
successfully!


Validation succeeded.

Date and Time are: #I# Sat, May 20, 2017  12:08:10 PM  (942646) #i#

C:\>echo %errorlevel%
0
```

Validating an Auth file (entering an incorrect password):

```
C:\>make_auth_file.exe . TestAuth.txt JustValidate

Date and Time are: #I# Sat, May 20, 2017  12:08:24 PM  (272218) #i#

Date and Time are: #I# Sat, May 20, 2017  12:08:24 PM  (272218) #i#


Authorization / Authentication File Creation Utility
  Version 1.0
  Copyright 2017 Richard Joseph Lotero


Validating Authentication / Authorization File...


Enter Authorization / Authentication Passphrase: **************

validate_auth_file(): Decrypted data blocks do not match.
  Authorization / Authentication file "TestAuth.txt" failed validation.

make_auth_file(): Authentication / Authorization data file failed
validation

C:\>echo %errorlevel%
1
```

Contents of the sample Auth file

```
C:\>type TestAuth.txt
#
# TestAuth.txt
#
# Data file used to confirm authorization / authentication passphrase
#
# Generation Timestamp: 20170520-12:07:54.128281
#
# Do not change this file.
#

gedHX3cZ7YGmj6WoZlOEy5xfmBSMqE0ffg6BTObuS6Yt2IkrZ87qjrgkZJSMxznsU7OZxY+
PMYhE
A/RQGTwUPyaHP+7V+OKzjAYnmMBmZf6mjnxnGYr5KuKN/UbFLMwNmiJBCiEXMMuE7RKTk4V
i4L3E
k/rOmlfmkPcqRq+s87rlSWQuVRWsBmFRrOlvm2sk1OlmYHdYJnA3DfpS1lCD3g6pk530FTB
/t68O
1f7hwXQ9atluvb2Z8+h/dmxBVco5arOjmuUkwpuQMYX+f6VQcrtURFXhhiE+bZrBD+oInc6
d9fTl
9iLy/npWv1sa9JeCK00x4Qsu79Ti3GTtxoSenA==


sid5KsB4o9yo82XBizgwcu8g+qBunX1eOq9M8CPOQn94yvQ2oz5x5/ndzdjXbp8fu061T9i
YFt3c
+QuJz2v0DEr22FV+2O3TlINYybsHcuSROcGysMFdHCDg+Afcq0BFAJPgjydIjJvnQVd15v2
sxs+4
KzTedwC98akZw4xMBvlI84FHQYahmtId5PW1w0Al+4EvKVgqhRK7Cu2CGynLq1Z4ZgegiEp
5fJR5
```

V0dSDG/4P96I8yTshKM2RQtyeOW1zFTYvBqW2xRF53pD6HOoEvyVFHu5fJphCSND9tqndgD
SyDta
p/+4sA/buzjjHAAkCbq8udDNg0l8L0VwWYPStQ==

++++++++++++++++++

End of appendix.