Fight crime.
Unravel incidents... one byte at a time.

A Regular Expression Search Primer for Forensic Analysts

GIAC (GCFA) Gold Certification

Author: Timothy Cook, tcbcook@yahoo.com
Advisor: Robert VandenBrink

Accepted: March 29, 2011

Abstract

Often forensic texts and articles assume a level of experience and comfort with Linux command line string searching and text manipulation that a reader does not possess. This assumption tends to leave the reader to their own devices to puzzle out how to locate and extract specific string content from files. The focus of this paper is to introduce the reader to Linux string search and text manipulation commands and provide specific use cases and search patterns that will be of use to Forensic Analysts. The intent of this paper is to serve as an introduction to regular expressions and some Linux commands that can be used to locate and extract text for individuals who either do not have Linux command line experience or who use the Linux command line infrequently and can benefit from a refresher.

1. Introduction

This paper introduces some of the powerful ASCII pattern identification and manipulation tools that are available to Forensic Analysts from the command line of the Linux Operating System of the SANS Investigative Forensic Toolkit (SIFT) Workstation. It will discuss crafting a search to locate exactly the ASCII pattern that the reader seeks and will introduce two Linux commands that will allow the reader to isolate and extract exactly the data that is being looked for. Throughout this paper examples of specific commands and their syntax will be provided in an effort to clarify discussions and provide specific use examples. It is not the intent of this paper to transform the reader into a "Linux guru" but rather to provide the reader with enough familiarity of the Linux commands introduced to enable the reader to understand and be comfortable with the search patterns that will be built in this paper, or that the reader may need to craft on their own. This paper assumes that the reader is using the SANS Investigative Forensic Toolkit (SIFT) Workstation. The SIFT Workstation is a VMware appliance that is built on the Ubuntu Operating System and is used extensively in the SANS Forensic Curriculum and is available for download from http://computer-forensics.sans.org/community/downloads.

2. Text Searching Basics

2.1. What and How

Before a search can be crafted to locate specific information it is necessary to be able to define exactly what it is being looked for (most often as a pattern) and sometimes it is necessary to define what is not being looked for (what to exclude). As a simplistic example, let's say that the reader is tasked with looking in a forensic image for a deleted file containing a telephone contact list for a smuggling ring. Phone numbers can be recorded in many formats – in the United States they may be seven digits or ten digits and if it is an international number then it may be even longer. Additionally the digits may be separated by dashes, spaces, parentheses or not separated at all. While it might be possible to visually scan a small file and identify multiple phone number formats, for larger files this is not a reasonable approach. A better method would be to craft a

command or commands that will locate the phone numbers. As US area codes range from 201 to 999 one could start by attempting to locate "201-100-0000' and incrementally work to "999-999-9999", repeating for each alternative format or one could identify and define ASCII text patterns that can be used in an automated search.

After the "what" of the search has been defined it is necessary to address the "how." Expanding on the example above, suppose that it is determined that it is desired to identify all of the blocks of the image that contain telephone numbers from the United States and that it is safe to assume that a smuggling ring is not using toll free numbers. If a phone number is initially defined as ten digits separated by dashes ("###-###-####") that are not "800" or "88#" area code numbers then one way to approach this might be to first find all text in the "###-###-####" format and then exclude all of the hits that start with "800" or "88#" (there are additional U.S. toll free numbers however including them at this point would unnecessarily complicate this example). The "what" (10 digits in the format "###-###-####") and the "how" (identify all of the phone numbers and then exclude the numbers in the format of "800-###-###" or "88#-###-####") of the search have now been defined. This could be pseudo coded as:

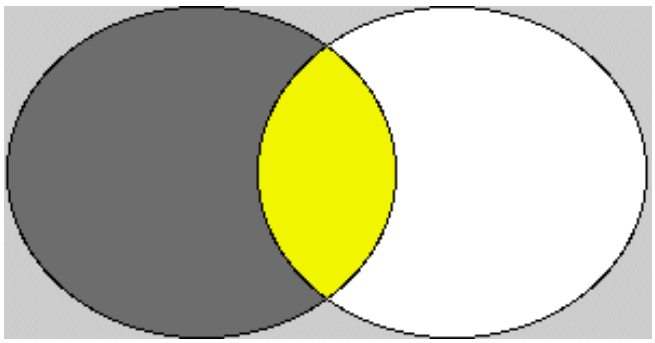
1. Locate all lines of text that contain "###-###-####" and place them in an output file.
2. Locate all of the lines of text in the output file that do not contain "800-###-###" and place them in a second output file.
3. Locate all of the lines of text in the second output file that do not contain "88#-###-###" and place them in a third output file

This would work as long as all of the phone numbers were in the specified format and no other text was in the specified format. If this isn't the case then it would be necessary to review and redesign the search. Perhaps it is discovered that phone numbers are present in the above defined format as well as in the format "(###) ###-####". Now the pseudo code might look like:

1. Locate all lines of text that contain "###-###-####" and place them in an output file.
2. Locate all lines of text that contain "(###) ###-####" and append them to the output file.

3. Locate all of the lines of text in the output file that do not contain "800-###-###" and place them in a second output file.
4. Locate all of the lines of text in the second output file that do not contain "88#-###-###" and place them in a third output file.
5. Locate all of the lines of text in the third output file that do not contain "(800) ###-###" and place them in a fourth output file.
6. Locate all of the lines of text in the fourth output file that do not contain "(88#) ###-###" and place them in a fifth output file.

This search method is similar to the manner in which gravel quarries sort rocks. First a bucket of earth is dumped onto a conveyor that carries it to a series of screens. These screens start out very large but subsequent screens decrease in size. As the earth reaches each screen the pieces smaller than the holes in the screen pass through and the pieces that are larger than the holes in the screen are retained. While both filtering methods accomplish what they are intended to, they are crude and resource expensive. As each line of pseudo code represents a separate command line sent to the processor the fewer the lines of pseudo code the fewer processor cycles it will take (not to mention less human involvement) to execute resulting in a faster, more efficient search. Fortunately for us we are working with computers rather than gravel. In order to improve the pseudo code the Boolean Operators "AND", "OR" and "NOT" are going to be introduced. One way to demonstrate these operators is through the use of a Venn diagram. Let's suppose that the Venn diagram below represents dinosaurs with the dark area representing those that eat only plant matter (herbivores), the light area representing those that eat only meat (carnivores) and the area where the circles overlap representing those that eat both plant matter and meat (omnivores).



The combined contents of both circles could be expressed as “dinosaurs that eat meat OR plants” while the overlap area could be expressed as “dinosaurs that eat meat AND plants”. Likewise the dark area could be expressed as “dinosaurs that eat plants NOT meat” and the light area could be expressed as “dinosaurs that eat meat NOT plants”.

With the introduction of the “OR” operator the pseudo code might now look like the following:

1. Locate all lines of text that contain the numerical patterns “###-###-####” OR “(###) ###-####” and place them in an output file.
2. Locate all of the lines of text in the first output file that do not contain the numerical patterns “800-###-###” OR “(800) ###-####” OR “88#-###-###” OR (88#) ###-###” and place them in a second output file.

The use of the “OR” logic operator has reduced the pseudo code from six lines to two and by including the “NOT” operator it can further be reduced to one line:

1. Locate all lines of text that contain the numerical patterns {“###-###-####” OR “(###) ###-####”} NOT {“800-###-###” OR “(800) ###-####” OR “88#-###-###” OR (88#) ###-###”} and place them in an output file.

After the search has been clearly defined and made as efficient as possible it is time to learn the Linux commands that will enable the reader to execute the search and manipulate the results.

3. The Linux Commands

Before beginning a discussion of the Linux commands that will assist the reader in locating and extracting information let's take a look at the typical syntax used by Linux commands:

command [OPTIONS] OBJECT

The OPTIONS, sometimes referred to as "command line options" or "switches", are identified by a leading hyphen ("-") and control how the Linux command is executed by the operating system. They can be used to modify the command's interpretation of the input or how it formats or displays the output, and can be either omitted or used in combinations.

The OBJECT is what the Linux command will use as input and is typically a file name. If the command follows a "pipe" (pipes will be discussed in the next section) then the OBJECT is omitted as the Linux Operating System understands that the object to perform the command on is the output of the command to the left of the pipe.

Linux users can always learn more about a command or refresh their memory of a command's syntax by accessing the manpage for the command via the "man" command (which, humorously, there is a manpage for…) using the following syntax:

man [-k] name

The optional "-k" switch tells the "man" command to use the name following as a pattern in a keyword search of a utilities summary database. This option can be useful if a user doesn't remember a specific Linux command but believe that they will recognize it if they see it.

3.1. Pipes and Redirects

Pipes and redirects are used to instruct Linux to do something with the output of a command other than print it to "STDOUT" (the screen) and can be very useful in combining the output of multiple searches, in sorting the output of searches into files or in refining the output of searches:

- "|" (the character that cohabitates a key with "\") is known as a pipe symbol and instructs the computer to execute the command to the right of the symbol using STDIN (the output of the command to the left of the symbol) as input. Or to express this in another manner it can be used to direct the output of one search/manipulation into another search/manipulation.

  "Linux command 1" | "Linux command 2"

- ">" can be placed after a command to instruct the system to redirect the output of the command to a destination specified to the right of the redirect. This destination can be any number of things but in this paper text files are going to be used if the file does not exist already the command will create it. If the designated file does already exist then it will be over written and the original contents of the file will be lost.

  "Linux command 1" > output_file.txt

- ">>" when placed after a command it instructs the system to redirect the output of the command and to append it to the destination specified. As with the redirect above, if the file does not exist it will be created however when something is appended to a file it is added to the end of the file and does not overwrite the existing file.

  "Linux command 1" >> output_file.txt

3.2. srch_strings – Getting the ASCII Out

The 'srch_strings" command is what is taught in the Forensic Curriculum at the SANS Institute for extracting the printable content from a forensic image. By default it extracts printable strings of at least 4 characters in length but this behavior can be changed through the use of options. The Ubuntu Linux manpage for the srch_strings command (Ubuntu, 2005) gives the syntax for the command as:

srch_strings [option(s)] [file(s)]

The options that the reader is likely to encounter or use are:

-a Scan the entire file rather than just the data section.

-n This option can be used to change the minimum length of printable characters to extract from the default value of 4.

-t The allowable values for this option are {o,x,d} and instruct the command to display in the first column the offset of the text in octal, hex or decimal respectively.

-h Display the command line help

The command, as taught in the SANs Forensic courses (Lee, 2011) and modified to accommodate the file names used in this paper, is:

srch_strings –a –t d forensic_image.img > string_file.txt

This command will extract all sequences of printable ASCII text of at least 4 characters in length from the file named "forensic_image.img" and output it to the text file "string_file.txt" preceding each line of printable ASCII text with the decimal offset of it's location in the "forensic_image.img" file.

3.3. grep – The String Search Tool of Choice

New Linux users often wonder where some of the "funny commands" got their names. According to "Netizens: On the History and Impact of the Net" (Hauben, 1996) the UNIX grep command was created by Ken Thompson on March 3, 1973 and is a derivation of the editor command that it simulated:

"One afternoon I asked Ken Thompson if he could lift the regular expression recognizer out of the editor and make a one-pass program to do it. He said yes. The next morning I found a note in my mail announcing a program named grep. It worked like a charm. When asked what that funny name meant, Ken said it was obvious. It stood for the editor command that it simulated, g/re/p (global regular expression print)."

The SIFT workstation, built on the Ubuntu Operating System, includes GNU grep (GNU is a recursive acronym for GNU's Not Unix). The Ubuntu manpage for the grep (Ubuntu, 2005) command gives the syntax for the command as:

grep [OPTIONS] PATTERN [FILE…]

For the purpose of the examples below it is assumed that [FILE…] is the ASCII string content from the forensic image that was extracted to the "string_file.txt" file using the "srch_strings" command on the SIFT workstation.

The grep command is actually a combination of four separate commands (grep, egrep, fgrep and rgrep), each of which handles regular expressions differently. The two that are going to be discussed in this paper are grep, which interprets patterns as being Basic Regular Expressions (BRE), and egrep, which interprets patterns as being Extended Regular Expressions (ERE). The differences between the two are varied but center around how certain characters in patterns are interpreted. While discussing the differences with John Bambenek (coauthor of "*grep Pocket Reference*" which is published by O'Reilly Media) he stated that "you can make almost everything work in grep that works in egrep, but it is messy" and advised that "if I were doing it, I'd tell them to live in egrep."

The OPTIONS can modify both how the grep command interprets the patterns as well as how it displays the output, and can be either omitted or used in combinations. Some of the available options will be discussed here and examples provided as well.

Pattern Modifiers

-E Interpret the pattern provided as an Extended Regular Expression. This is equivalent to using the egrep command:

grep –E 'forensic' string_file.txt

-e PATTERN

This instructs grep to use the pattern that follows it as a literal pattern and is useful when a pattern includes characters that can be interpreted as either literals or meta-characters (meta-characters will be discussed later when patterns are discussed). The command below would look for the specified text in the text file, with preceding hyphen, and will not interpret the hyphen as a switch:

grep –e '-grade' string_file.txt

-f FILE Open the named file and use the contents as patterns to search for. This allows the use of a file that contains a dirty word list or set of patterns of interest to your investigation. It is useful to maintain a Master Pattern File or several subject specific pattern files (such as drug_terms.txt or hacking_terms.txt) that can be used to select tested patterns rather than recreating patterns every time the need arises. The command below would look in string_file.txt for every word or pattern listed in the text file dirty_words.txt:

grep –f dirty_words.txt string_file.txt

-i Ignore case, or do a "case insensitive" search on the pattern. As Linux is case sensitive this can be very useful. The command below would look in string_file.txt for every upper or lower case combination of the letters "forsenic":

grep –i 'forensic' string_file.txt

-v This option is described as "invert-match" and it instructs the grep command to output every line that does NOT contain the pattern. It is useful when trying to exclude specific patterns (such as the "88#" numbers in our example above). The command below would look in string_file.txt for every line of text that does NOT contain the lower case pattern "forensic":

grep –v 'forensic' string_file.txt

-w Match only if the pattern is matched as a word (white space before and after) rather than as a subset of a word. This would ensure that a search for "the" only returned hits for the word "the" and not as a subset of perhaps "rather" or "theory". The command below would look in string_file.txt for every line that contains the lower case word "the":

grep –w 'the' string_file.txt

Output Control

-c Instead of printing the pattern matches just provide a count of matching lines (note that if the pattern is matched 3 times in 1 line it will only increase the count by 1). The command below would return the number of lines in string_file.txt that contain the lower case pattern "forensic":

grep –c 'forensic' string_file.txt

-o Print only the matched pattern rather than the whole line and if a line contains multiple matches print each match on its own line:

grep –ow 'forensic' string_file.txt

Multiple switches can be specified by either listing them singly (each with their own hyphen) or by stringing them together behind one hyphen but users must always consider what affect switches will have on the pattern that that is specified.

"-i –w" or "-iw" Perform a case insensitive search for the pattern specified as a word.

"-i -w -c" or "-iwc" Return a count of the lines that contain the upper or lower case pattern specified as a word.

"-i –v" or "-iv" Exclude all lines that contain upper or lower case combination of the pattern.

"-i –v -c" or "-ivc" Return a count of the lines that don't contain the upper or lower case combination of the pattern.

"-i –w -f" or "-iwf" Perform a case insensitive search for the contents of the named file as whole words. This combination is very useful when we have a list of terms or patterns compiled into a dirty word list file.

The PATTERN represents the text that the reader wants to locate and is in the form of a regular expression (sometimes referred to as "regex' or "regexp"). A regular expression is nothing more than a character or set of characters that can be used to match a single character, multiple characters or combination of characters and properly crafted

regular expressions can enable the reader to efficiently locate any content desired. As an example, a regular expression can be crafted that will locate the lower case word "forensic" or the upper case word "FORENSIC" or any combination of upper and lower case letters as well as lines that start or end with combinations of "forensic" or even lines that don't start or end with combinations of "forensic" as well as "forensic" as a whole word or as a part of a word (such as "forensics" or "forensically"). Regular expression patterns are customarily enclosed in single quotes (the single quote shares a key with the double quote and is located just to the left of the "Enter" key) to clearly delineate the beginning and end of the pattern. The more one learns about grep and regular expressions the more accomplished one will be come at locating exactly the content, and only the content, that they seek.

The most basic building block of regular expressions are called literals and are those that match one character, such as specific letters and numbers. These can be combined to find literal combinations of ASCII text. The command below would return all text where the lower case letter "t" was followed by lower case "h" and then lower case "e" and would return, if they were present, lines with such words as "the", "rather", "theory", etc.:

```
grep 'the' string_file.txt
```

This can be combined with grep options to be more or less specific. For instance, in order to find only the lowercase word "the" we would add the "-w" option:

```
grep –w 'the' string_file.txt
```

If it doesn't matter if the pattern is interpreted as upper or lower case the following could be used:

```
grep –i 'the' string_file.txt
```

If only the word "the" in upper or lower case letters is desired:

```
grep –iw 'the' string_file.txt
```

And to exclude any line that contained the upper or lower case word "the":

```
grep –viw 'the' string_file.txt
```

In addition to literals grep utilizes meta-characters, or characters that have a special meaning in regular expressions and, unless specified, are not used as literals. To define a single instance of a range of characters one can use what is termed a "character class" which is a list of characters enclosed by the meta-characters "[" and "]". If the first character in the list is a '^' then it will match any character that is NOT in the list. A range of characters can be defined by providing the first and last characters in the range separated by a hyphen (to test for a literal hyphen it must come first or last in the range, as in [-ad] which would locate a single instance of "-", "a" or "d"):

[abcdef] or [a-f]	Any one of the letters "a", "b", "c", "d", "e" or "f".
[23456] or [2-6]	Any one of the numbers "2", "3", "4", "5" or "6".
[^3456] or [^3-6]	NOT one of the numbers "3", "4", "5" or "6".

Thus one can locate either the letter combination "the" or "The" through the following command:

```
grep –E '[Tt]he' string_file.txt
```

And one could combine this with the "–w" option to further specify that only the words "The" or "the" are sought:

```
grep –Ew '[Tt]he' string_file.txt
```

Certain ranges or classes of characters are so useful that they have been predefined in "named character classes" and those that might be useful in a forensic search of text have listed below.

[:alpha:]	Any alphabetic character A-Z or a-z (same as [a-zA-Z]).
[:digit:]	The digits 0-9 (same as [0-9]).
[:alnum:]	Any alphanumeric character (same as [a-z0-9A-Z]).
[:upper:]	The upper case letters A-Z (same as [A-Z]).
[:lower:]	The lower case letters a-z (same as [a-z]).
[:punct:]	Punctuation symbols.
[:space:]	Whitespace comprised of space characters or tabs.

[:blank:]	Whitespace comprised of TAB, space or carriage return.
[:xdigit:]	Hex characters a-f, A-F or 0-9 (same as [a-f0-9A-F]).

Note that the brackets are part of the named character class, and when using named character classes in a pattern it is necessary to include them within brackets so that they actually appear to have double brackets around them ([[:xdigit:]]). Examples of this will be provided later in the paper when a regular expression is developed for a MAC address.

Another type of meta-character is a positional character, or anchor, which allows users to specify where in the line of text a string pattern should be located. While some of these may not be relevant to a search of a text dump of a forensic image (as the lines of text would not be in the same format that they would be if the text file were searched independently) they can be useful in searching individual text files and can significantly improve the efficiency of our search.

^ A carat ("^") outside of a character class has a different meaning than a carat inside a character class (recall that a "^" inside a character class signifies "NOT"). Outside of a character class, as in "^A" or "^[0-9]", it is a starting anchor and indicates that one is looking for lines that begin with the "A" or a digit. Likewise "^It" would match lines that start with the word "It". This allows grep to restrict its pattern comparison to the beginning of each line thus improving the performance of the search:

```
grep -E '^It' string_file.txt
```

$ A "$", as in "A$" indicates that one is looking for the lines that end with the "A" and "$night" would match lines that end with "night". This allows grep to just search the characters at the end of a line thus improving the performance of the search:

```
grep -E 'night$' string_file.txt
```

. A period (often referred to as a "dot") means "any one character can be here" and it is used where one wants to specify that the position must be occupied by a non-blank character. As an example, the pattern "mark." would match "marks" but not "mark" or "marker":

```
grep -E 'mark.' string_file.txt
```

Iteration or modifier meta-characters control the number of times that the preceding character is to be used in a search and are useful in looking for variations of patterns.

? The question mark "?" is used to match the preceding character 0 or 1 times (in other words the character is optional). The pattern "honou?r" would match either the British spelling of "honour" or the American spelling "honor":

```
grep -E 'honou?r' string_file.txt
```

* The asterisk is used to match the preceding character 0 or more times. The pattern "bo*k" would locate both "book" and the abbreviation "bk" as well as a fat fingered "bok" or "boook". The asterisk is often used in conjunction with a period to indicate "any text 0 or more times" or more generally, ".*" is a "wildcard" equivalent to "any text":

```
grep -E 'bo*k' string_file.txt

grep -E 'beginning.*end' string_file.txt
```

+ The plus sign is used to match the preceding character 1 or more times. The pattern "bo+k" would locate both "book" and the fat fingered "bok" or "boook" but would not locate the abbreviation "bk":

```
grep -E 'bo+k' string_file.txt
```

{n} This is used to indicate that the preceding character or sub pattern should match exactly "n" times. If the reader were looking for a zip

code they would be looking for a 5 digit number and the pattern might look like:

grep -Ew '[0-9]{5}' string_file.txt

{n, } This is used to indicate that the preceding character or sub pattern should match "n or more" times. A search pattern for a word that is at least 10 characters long might look like the following:

grep -E '[[:alpha:]]{10,}' string_file.txt

{n,m} This is used to indicate that the preceding character or sub pattern should match at least "n" times and no more than "m" times. A search pattern for a stock number that was made of anywhere between 9 and 13 numbers or letters may look like the following:

grep -E '[[:alnum:]]{9,13}' string_file.txt

There are four additional meta-characters that are of importance because they enable the user to define how patterns should be interpreted. These are the parentheses, backslash, the vertical bar or pipe symbol and the word boundary symbol.

\ The backslash character (sometimes referred to as the escape character) is used when it is desired to include in a pattern one of the meta-characters listed above as a literal character. Unlike the "-e" switch the escape character just affects the character that follows it and not the whole pattern. As an example, to search in a text file for the file named "drugs.xls" one would need to escape the period and the pattern would look like 'drugs\.xls'. Similarly, to search for a backslash one would need to escape it with a backslash ('\\') and to define a pattern that begins with a hyphen one would need to type '\-' so that grep does not interpret it the hyphen as another switch:

grep -E '\$[0-9]' string_file.txt

grep -E '\-grade' string_file.txt

grep -E 'drugs\.xls' string_file.txt

() Parentheses enable characters to be grouped in a pattern to create sub-patterns. When parenthesis are followed by an iteration meta-character it indicates that the pattern within is to be treated by repetition symbols as a single character. To clarify this, in order to find either Tim or Timothy, but not Timothyothy the pattern would look like:

grep –E 'Tim(othy)?' string_file.txt

The sub pattern in parentheses followed by the "?" indicate that one is looking for 0 to 1 instances of the sub pattern "othy".

| When utilized in a regular expression pattern the pipe represents a logical OR. It is often combined with the parentheses to group multiple characters that are allowable in a specific position of the pattern. The example below would look for the word "the" with upper or lower case "T" and with lower case "h" and "e":

grep –E '(T|t)he' string_file.txt

Astute readers might note that this pattern is similar to a pattern previously used in the discussion of brackets. Both patterns would have the same result however using the character class is considered more efficient than using alternation. Where the use of alternation makes sense is when looking for alternate sub patterns:

grep –E '(Jan|Feb|Mar)' string_file.txt

\b This indicates a word boundary and can be used to indicate that the pattern is to be found at the beginning of a word, at the end of a word or is to be located as a whole word by placing the \b at the beginning, end or both, respectively:

grep –E '\bthe' string_file.txt

grep –E 'the\b' string_file.txt

grep –E '\bthe\b' string_file.txt

Users need to be careful when combining meta-characters and fully consider how grep will interpret the patterns that are defined. As an example, preceding a single character with "^" and following it with "*" would combine to mean "any line that starts with 0 or more of this character", which would match every line in the file. Users also need to be aware of the affect that the optional command line switches that are used will have on the grep command's interpretation of a pattern.

3.3.1. Pattern Construction and Maintenance

Now that the reader has a grasp of the basics of grep and regular expressions it is time to craft some simple grep searches. Before we do though let's revisit the "-f" option and the concept of the dirty word list. Dirty word lists can be very useful but Master Dirty Word lists can also quickly become burdensome. As an example, a Master Dirty Word list for all possible drug terms could easily contain several thousand patterns, many in foreign languages, and most of which a forensic examiner might never encounter in a lifetime of searches. Consider how long it would take grep to compare every one of the several thousand patterns in such a file to the entire contents of a 5 GB srch_strings output file. Instead it is better to develop and maintain one's own subject specific dirty word lists that can be used to collect patterns that are relevant to searches that the user has conducted in the past and from which the user can cut and paste case specific dirty word lists for each search based on what is known about a case and terms of interest that are found through the examination of the forensic image being reviewed. This process often involves building new case specific dirty word lists for subsequent searches as one learns more about the contents of the image from the results of previous searches.

One of the things that it makes sense to keep in a master file is extended regular expression patterns for things that one may wind up searching for in different cases, such as the examples below. In each case the pattern (some of which come from "*grep Pocket Reference*" by Bambenek and Klus and are included with the permission of O'Reilly Media) will be presented and then the pattern will be broken down so that the reader can better understand what has been presented in this paper. Each of the patterns provided has been tested on the SIFT workstation.

Search patterns list:

Note: Many of the following patterns contain a leading and following \b which indicates that the pattern is to be searched for as a word or a standalone pattern.

File Extensions: grep –Ei '\.(txt|exe|xls|doc|docx|jpg|bmp)\b'

The trailing \b with no leading \b defines that the pattern is to be located at the end of a word but need not comprise the entire word. The beginning of the pattern is an escaped period indicating that it is to be interpreted literally as a "dot" and the sub pattern that follows contains a selection of common file extensions. The grep command utilized the "-i" switch to make the search case insensitive.

URLs: grep – E '\bhttps?://.+\.(com|net|org|uk|mil|gov|edu)'

The leading \b with no following \b defines that the pattern is to be located at the beginning of a word but that it need not comprise the entire word. The "?" following the "s" indicates that it is optional so that the pattern will locate both "http" and "https". Note the use of the dot before the "+" to indicate 1 or more characters followed by an escaped "dot" and then a selection of valid domain names.

SSN: grep –E '\b[0-9]{3}(|-)[0-9]{2}(|-)[0-9]{4}\b'

[0-9]({3}(|-) defines 3 digits that are followed by either a space or a dash. This is repeated for a pattern of 2 more digits and then 4 more digits. One can also add an alternate format of 9 continuous digits by changing the "(|-)" to "(|-|)".

MAC Addresses: grep –E '\b([[:xdigit:]]{2}:){5}[[:xdigit:]]{2}\b'
grep –Ei '\b([0-9a-f]{2}:){5}[0-9a-f]{2}\b'

Two versions are presented here in order to show both the usage of a named character class as well as the fact that sometimes it makes the patterns longer and harder to read. In the top pattern the ([[:xdigit:]]{2}:){5} indicates a sub pattern of 2 hexadecimal characters followed by a ":" repeated 5 times and then followed by a final 2 hexadecimal characters. This will return any combination of digits from 00:00:00:00:00:00" to "FF:FF:FF:FF:FF:FF" in upper or lower case characters. The second version is the exact same pattern but utilizes [0-9a-f] instead of the named character class and relies on the grep switch "-i" to cover upper and lower alphabetic characters. Finally, the ":" could be replaced with a sub pattern of (|-|:) to account for addresses that are alternately space or dash separated.

IP Addresses: grep –E '\b[0-9]{1,3}(\.[0-9]{1,3}){3}\b'

[0-9]{1,3} defines a pattern of anywhere from 1 to 3 digits. This is followed by a sub pattern of a literal period followed by 1 to 3 digits which is repeated exactly 3 times. This will return any combination of digits from "0.0.0.0' to "999.999.999.999." Since IP addresses only range from "0.0.0.0" to "255.255.255.255" depending upon the search results it may be necessary to create a pattern to exclude higher number groups from the results or one could build in a range of 0 – 255 in each place at the expense of making the pattern more complex.

Credit Card: grep –E '\b[0-9]{4}((|-)[0-9]{4}){3}\b'

[0-9]{4} defines a pattern of 4 digits followed by a sub pattern of an optional space or dash followed by a sub pattern of 4 digits which is repeated exactly 3 times. This will return any combination of digits from "0000-0000-0000-0000" to "9999-9999-9999-9999" with the dash being optional or replaced by a space.

American Express: grep –E ‘‘\b[0-9]{4}(|-|)[0-9]{6}(|-|)[0-9]{5}\b’

Apparently American Express is not just like the other guys and for that reason the American Express pattern is a little more straight forward. Here the pattern defines 4 digits followed by 6 digits followed by 5 digits, all of it optionally separated by spaces or dashes.

Email Addresses: grep –Ei ‘\b.+@.+\.(com|net|org|uk|mil|gov|edu)\b’

Two things that all email addresses have in common are the “@” in the middle and the “dot domain name” at the end. This pattern looks for 1 or more characters (defined by the “.+”) followed by “@” and then 1 or more additional alphanumeric characters and ending in a literal dot followed by one of the top level domain names listed. This is just an example and if a specific domain name or a domain name not in the sub pattern is desired it can be substituted or added.

US Phone: grep –E ‘\b(\(|)[0-9]{3}(\) |-| |\)-|)[0-9]{3}(-| |)[0-9]{4}\b

The sub patterns on this one deserve close attention. The first sub pattern is “(\(|)” and can be read as “optionally starting with a literal “(“. The second sub patterns is “(\) |-| |\)-|)“ which defines optionally a literal “)” followed by a space or a dash or a space or a literal “)” followed by a dash or no separator at all. The third sub pattern is simpler and indicates a dash or a space or no separator at all. This would locate a phone number in any of the following formats:

(800)-555-1212
(800) 555-1212
800-555-1212
8005551212
800 555 1212

3.4. Text Extraction Made Easy

If, instead of returning whole lines of text, a user wants to extract only the pattern that they are looking for, and don't mind having each instance of the pattern returned on its own line, then the user can use the "-o" switch with grep to easily accomplish this. But what if the user wants alternate line content in addition to, or instead of, the pattern? Perhaps one is looking for the offset number at the beginning of the line of text in the srch_strings command output file or a pattern is only a part of the information that the user seeks. Linux provides commands that can facilitate that as well and two of these commands will be discussed below.

3.4.1. cut

The Linux "cut" command is one of more normal sounding Linux commands as well as one of the simpler commands available to extract text from a file or a line of output based on position. While there are more powerful extraction tools available, sometimes all the user needs is a simple text extraction command and cut is easy to use and fast. The format of the command is listed on the Ubuntu manpage (Ubuntu, 2005) as:

cut OPTION…[FILE]…

The [FILE…] parameter can be either the name of an input file or, if the cut command follows a pipe, it can be omitted. Some of the options available for the cut command are:

-c Lists a character or multiple character column positions in the line to extract. The character positions can be a single character position number, multiple character position numbers separated by a comma or a dash or a combination of these. The option "-c5" would extract only the 5th character while "–c5,6" would extract the 5th and 6th characters from the text and "-c5-25" would extract all of the characters from the 5th to the 25th character. The option "-c5- " would extract from the 5th

character to the end of the line of text. The position of the desired data in the line of text needs to be very consistent for this option to be useful:

cut –c5,6 Pattern1_Search.txt

grep – Ei 'Pattern1' string_file.txt | cut –c5-25

grep – Ei 'Pattern1' string_file.txt | cut –c5,6,10-20

-f This option tells cut to select the specified field or fields which are by default tab or space separated. As with the "-c" option above we can select a specific field or use commas and dashes to specify multiple fields. Assuming that one used the "–t" option with the srch_strings command to have each line of text preceded by the offset at which it is to be found in the forensic image the option "–f1" would return a list of all the file offsets that contain the pattern defined in the grep command:

cut –f1 Pattern1_Search.txt

grep – Ei 'Pattern1' string_file.txt | cut –f5,8,10-13

-d This option allows users to specify a field delimiter for the "-f" option. If one is dealing with a database export or a Comma Separated Value (.CSV) file then it may be needed to specify a non-default field delimiter such as a comma or colon. It is sometimes necessary to enclose the specified delimiter in quotes:

cut –d: –f5,6 Pattern1_Search.txt

grep – Ei 'Pattern1' string_file.txt | cut –d"," –f5-

3.4.2. awk

The Linux awk command is actually a command line utility used to execute programs written in the awk programming language. The awk programming language, whose name is an acronym for the last names of its three authors (Aho, Weinberger and

Kernighan), is specialized for textual data manipulation. The awk language is capable of performing many actions including regular expression pattern matching, mathematical calculations and elaborate output formatting. This paper is going to restrict it's coverage of awk to rearranging and formatting fields and adding simple text to output.

Because it is a command that invokes a shell that executes a program the syntax of the awk command appears different than other Linux commands introduced thus far. The Ubuntu manpage for awk (Ubuntu, 2005) lists the syntax as:

pattern { action }

The manpage further states that in the absence of a pattern that the action shall be performed on any input record (as when following a pipe). Similar to the cut command in awk there is a default field separator of white space or the user can define a field separator. The first field is designated as $1, the second $2 and so forth and if one wants to print the entire record or line $0 is used. Note that if multiple fields are printed they should be separated by a comma to ensure that the output is separated by a space:

-F Field Separator

```
awk –F: '{print $0}' Pattern1_Search.txt
grep –Ei 'Pattern1' string_file.txt | awk –F: '{print $0}'
```

As was mentioned above, one of the things that awk can do that cut can't is to insert text into the output, allowing users to organize the output. Note that text must be enclosed with a double quote:

```
awk –F: '{print "name: ", $3, " Phone: " $1}' Pattern1_Search.txt
grep –E 'Pattern1' string_file.txt | awk –F: '{print $3, " Phone:" $1}'
```

In addition to the print function awk is also capable of utilizing the printf function to add formatting that can create uniform columns of output. The printf function allows the user to specify different formats and modifiers for the output, some of which are:

%	Format control character – precedes each of the characters below when used in a printf statement.
-	The minus sign when used in conjunction with one of the formats below indicates that the value being printed should be left justified. If the minus sign is omitted the value will be right justified by default.
c	ASCII text.
s	String.
i	Integer.
\n	This instructs printf to insert a newline character. If it is not included then the lines of output will print all together on one line.

The printf formats are declared immediately after the printf statement and any added text is embedded in the formatting section. As an example of this the statements below will print the third field left justified and then the first field right justified, each in a column of a minimum of 10 characters in width. If the value of the field is longer than the minimum length it will not be truncated:

```
awk '{printf "%-10s %10s\n",$3,$1}' Pattern2_Search.txt

grep –E 'Pattern2' string_file.txt | awk '{printf "%-10s %10s\n",$3,$1}'

awk '{printf "Name: %-10s Phone: %10s\n",$3,$1}' Pattern2_Search.txt
```

If a user desires to generate a report with uniform columns, column headings or explanatory text then the printf function is the way to go.

4. Conclusion

The Linux Operating System that the SIFT workstation is built on provides powerful command line tools that enable Forensic Analysts to locate and extract ASCII text from forensic images or text files as required. The ability to handle both regular expressions and extended regular expressions makes grep a very useful command line search tool that can locate and extract text using patterns that are crafted based on the specifics of an investigation.

Timothy Cook, tcbcook@yahoo.com

Pipes allow users to combine Linux commands to refine searches and to carve out and manipulate the data that is being looked for. Additional grep commands used in conjunction with redirects, as well as the awk and cut commands, enable users to accumulate the information extracted in output files.

Forensic Analysts who take the time to master these tools can craft regular expression patterns to quickly and easily locate and extract information of specific interest to their investigations. The ability to leverage these tools and techniques to their full advantage, quickly, accurately, and effectively, is a key weapon in the Forensic Analyst's arsenal.

Timothy Cook, tcbcook@yahoo.com

5. Appendix – Phone Number Search Solution

Given what been covered in this paper, let's briefly revisit the phone number search that was used as an example earlier. In order to locate all of the non-toll free phone numbers in a srch_string generated text file, and output the file offsets to a new file, the reader could use the following command:

```
grep –E '\b(\(|)[0-9]{3}(\) |-| |\)-|)[0-9]{3}(-| |)[0-9]{4}\b |
    grep –Ev '\b(\(|)(800|(88)[0-9])(\) |-| |\)-|)[0-9]{3}(-| |)[0-9]{4}\b |
    cut –f1 > offsets.txt
```

6. References

Bambenek, J., & Klus, A. (2009). Grep pocket reference. Sebastopol, CA: O'Reilly Media.

Hauben, M., & Hauben, R. (1996). Netizens: On the history and impact of the net Retrieved September 19, 2011 from: http://www.columbia.edu/~hauben/netbook/

Lee, R. (2011). Forensics 508: File system forensic analysis. Bethesda, MD: SANS Institute.

Ubuntu Manpage Repository (2005). Ubuntu awk manpage Retrieved September 19, 2011 from: http://manpages.ubuntu.com/manpages/precise/en/man1/awk.1posix.html

Ubuntu Manpage Repository (2005). Ubuntu cut manpage Retrieved January 3, 2012 from: http://manpages.ubuntu.com/manpages/hardy/man1/cut.1.html

Ubuntu Manpage Repository (2005). Ubuntu grep manpage Retrieved September 19, 2011 from: http://manpages.ubuntu.com/manpages/hardy/man1/grep.1.html

Ubuntu Manpage Repository (2005). Ubuntu man manpage Retrieved January 13, 2012 from: http://manpages.ubuntu.com/manpages/hardy/en/man1/man.1posix.html

Ubuntu Manpage Repository (2005). Ubuntu srch_strings manpage Retrieved January 12, 2012 from: http://manpages.ubuntu.com/manpages/hardy/en/man1/srch_strings.1.html

Last Updated: February 25th, 2020

Upcoming SANS Forensics Training

SANS Northern VA - Reston Spring 2020	Reston, VA	Mar 02, 2020 - Mar 07, 2020	Live Event
SANS Secure Japan 2020	Tokyo, Japan	Mar 02, 2020 - Mar 14, 2020	Live Event
SANS Munich March 2020	Munich, Germany	Mar 02, 2020 - Mar 07, 2020	Live Event
SANS St. Louis 2020	St. Louis, MO	Mar 08, 2020 - Mar 13, 2020	Live Event
SANS Dallas 2020	Dallas, TX	Mar 09, 2020 - Mar 14, 2020	Live Event
Dallas 2020 - FOR500: Windows Forensic Analysis	Dallas, TX	Mar 09, 2020 - Mar 14, 2020	vLive
SANS vLive - FOR610: Reverse-Engineering Malware: Malware Analysis Tools and Techniques	FOR610 - 202003,	Mar 09, 2020 - Apr 22, 2020	vLive
SANS Paris March 2020	Paris, France	Mar 09, 2020 - Mar 14, 2020	Live Event
SANS Secure Singapore 2020	Singapore, Singapore	Mar 16, 2020 - Mar 28, 2020	Live Event
SANS London March 2020	London, United Kingdom	Mar 16, 2020 - Mar 21, 2020	Live Event
SANS Norfolk 2020	Norfolk, VA	Mar 16, 2020 - Mar 21, 2020	Live Event
SANS San Francisco Spring 2020	San Francisco, CA	Mar 16, 2020 - Mar 27, 2020	Live Event
SANS Oslo March 2020	Oslo, Norway	Mar 23, 2020 - Mar 28, 2020	Live Event
SANS Seattle Spring 2020	Seattle, WA	Mar 23, 2020 - Mar 28, 2020	Live Event
SANS Madrid March 2020	Madrid, Spain	Mar 23, 2020 - Mar 28, 2020	Live Event
SANS Secure Canberra 2020	Canberra, Australia	Mar 23, 2020 - Mar 28, 2020	Live Event
Mentor Session - FOR508	Sao Paulo, Brazil	Mar 25, 2020 - Mar 28, 2020	Mentor
SANS Abu Dhabi March 2020	Abu Dhabi, United Arab Emirates	Mar 28, 2020 - Apr 02, 2020	Live Event
SANS FOR585 Rome March 2020 (In Italian)	Rome, Italy	Mar 30, 2020 - Apr 04, 2020	Live Event
SANS Frankfurt March 2020	Frankfurt, Germany	Mar 30, 2020 - Apr 04, 2020	Live Event
SANS vLive - FOR508: Advanced Incident Response, Threat Hunting, and Digital Forensics	FOR508 - 202003,	Mar 31, 2020 - May 07, 2020	vLive
SANS 2020	Orlando, FL	Apr 03, 2020 - Apr 10, 2020	Live Event
SANS Riyadh April 2020	Riyadh, Kingdom Of Saudi Arabia	Apr 04, 2020 - Apr 16, 2020	Live Event
SANS Bethesda 2020	Bethesda, MD	Apr 14, 2020 - Apr 19, 2020	Live Event
SANS Minneapolis 2020	Minneapolis, MN	Apr 14, 2020 - Apr 19, 2020	Live Event
SANS London April 2020	London, United Kingdom	Apr 20, 2020 - Apr 25, 2020	Live Event
SANS Brussels April 2020	Brussels, Belgium	Apr 20, 2020 - Apr 25, 2020	Live Event
SANS Baltimore Spring 2020	Baltimore, MD	Apr 27, 2020 - May 02, 2020	Live Event
SANS Bucharest May 2020	Bucharest, Romania	May 04, 2020 - May 09, 2020	Live Event
SANS Security West 2020	San Diego, CA	May 06, 2020 - May 13, 2020	Live Event
Security West 2020 - FOR498: Battlefield Forensics & Data Acquisition	San Diego, CA	May 08, 2020 - May 13, 2020	vLive