Detecting Hydan: Statistical Methods For Classifying The Use Of Hydan Based Stegonagraphy In Executable Files

It is known that HYDAN changes the statistical distribution of Sub and Add calls in the assembly code to embed the "hidden data". Before this paper, there were no publicly released tools or methods available to detect HYDAN. The methods previously used to detect HYDAN have been inefficient and involved extensive manual processes that could not be easily automated. This paper presents a method to take the assembly code (using a disassembler) and to feed this into a statistical language, in order to det...

DETECTING HYDAN

STATISTICAL METHODS FOR CLASSIFYING THE USE OF HYDAN BASED STEGONOGRAPHY IN EXECUTABLE FILES

GIAC Certified Incident Handler (GCIH) Gold

Author: Craig S. Wright, CraigSWright@acm.org

Adviser: Carlos Frederico Cid

Accepted: 22 June 2008

Index

i. Abstract

It is known that HYDAN changes the statistical distribution of **Sub** and **Add** calls in the assembly code to embed the "*hidden data*". Before this paper, there were no publicly released tools or methods available to detect HYDAN. The methods previously used to detect HYDAN have been inefficient and involved extensive manual processes that could not be easily automated. This paper presents a method to take the assembly code (using a disassembler) and to feed this into R, a statistical language, in order to detect if the file has been altered steganographically.

The method uses a set of statistical tests to determine both the use of HYDAN and the extent of use in a file.

Executive Summary

Steganography is the art and science of hiding text messages in other data (Provos, 2003). This is commonly graphics files, audio files and video files. HYDAN is a steganographic tool that is designed to hide data inside of a binary executable file.

Steganography when used with multimedia based files can impair the quality of the output or display. This is usually managed such that the degradation remains unnoticed to the casual viewer/listener. Computers can display many more colors than the human visual system can process and hence a reduced color map in an image may be unnoticeable to simple visual or audit analysis. Changing data within an executable code segment is more problematic. The alteration of a single byte of binary code can result in an irreparable corruption of the code destroying the functionality of the code segment.

HYDAN is a method of encapsulating data steganographically within an executable code segment without either altering the function of the code or varying the file size of the executable.

This paper presents a method that can be used to detect HYDAN based steganography. This is simply an initial means and should be improved if it is to be used in any serious endeavor. To do this, the code could be compiled into a single program[1] that incorporates the disassembly and comparison in a

[1] A means of compiling R code into a standalone executable is presented in "Compiling R: A Preliminary Report" by Luke Tierney (2001). The paper is available from:

http://www.ci.tuwien.ac.at/Conferences/DSC-2001/Proceedings/Tierney.pdf

An R compiler called RCC that takes the R interpreted code and compiles it in C++/R format is available from: http://hipersoft.cs.rice.edu/rcc/index.html

single binary. This would then return a value for those segments that have embedded data.

What is Steganography?

Provos and Honeyman (2003) define steganography (aka stego) as "*the art and science of hiding communication; a steganographic system thus embeds hidden content in unremarkable cover media so as not to arouse an eavesdropper's suspicion*".

The majority of modern steganographic systems begin with discovering the redundant bits within the host media or data. The goal is to be able to modify the host data in a manner that does not obliterate the integrity of the source data. Another objective of steganography is to not be detected in the host file. It is in effect, a means of hiding data within other data.

Although contemporary steganography through the use of computers is a relatively recent field, both Richmond (1998) as well as Johnson & Jajodia (1998) make mention of an ancient example. In their paper they note the example of an early steganographic system. Richmond notes the practices of the ancient Athenians where the head of a messenger was shaved and subsequently tattooed with a message that would be covered with hair rendering it unseen if the messenger was captured. Johnson & Jajodia mention how this same system was adopted by a Roman general who shaved a slave's head and tattooed a message on it sending the messenger on the errant after the hair grew back.

The majority of steganographic methods (Provos, 2001) that have been developed in modern times have been centered on hiding a message within images and audio files (such as BMP, GIF, JPEG, WAV and MP3 file formats). A number of other methods include hiding messages within Word documents or even within embedded macros and Metadata (Provos & Honeyman, 2003).

The secret to effective steganography is that it needs to be difficult to detect (McGill, 2005).

HYDAN

Rakan El-Khalil designed a novel steganographic technique called HYDAN. The name selected, HYDAN [hI-dn] is erudite and holds a message in itself. The word actually means to *hide or conceal*. First developed in 2003, HYDAN hides data or messages in Binary Executables.

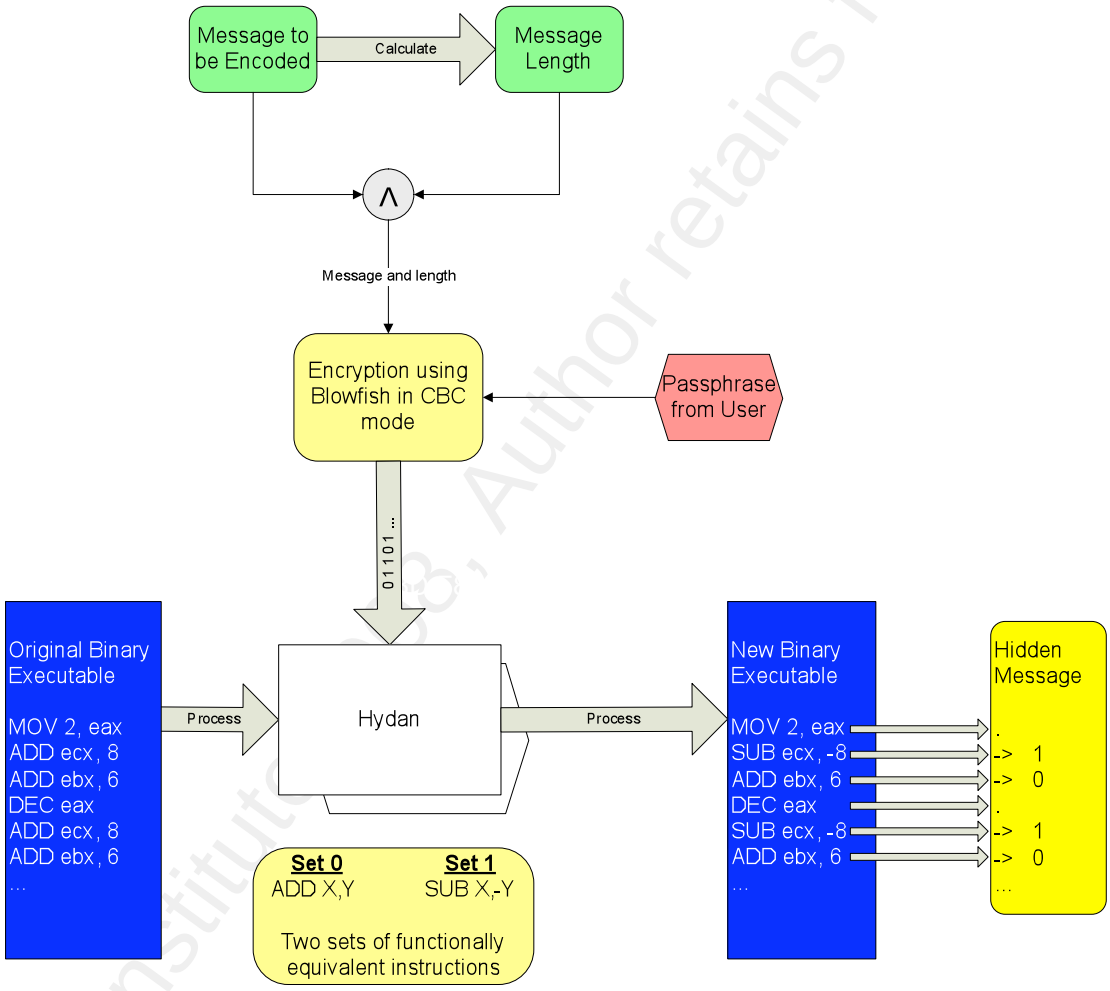
The main website for HYDAN (http://www.crazyboy.com/HYDAN/) offers a number of uses for HYDAN:

- ***Covert Communication:*** *embedding data into binaries creates a covert channel that can be used to exchange secret messages.*
- ***Signing:*** *a program's cryptographic signature can be embedded into itself. The recipient of the binary can then verify that it has not been tampered with (virus or trojan), and is really from who it claims to be from. This check can be built into the OS for user transparency.*
- ***Watermarking:*** *a watermark can be embedded to uniquely identify binaries for copyright purposes, or as part of a DRM scheme. Note: this usage is not recommended as HYDAN implements fragile watermarks.*

How HYDAN Functions

HYDAN steganographically secretes a message into an executable. It is a method of encapsulating data steganographically within an executable code segment without either altering the function of the code or varying the file size of the executable. HYDAN is designed to make use of a number of redundancies that exist within X86 binary instruction code or assembly language. The X86 assembly

language set has instances where two instructions are fundamentally the same. In addition, certain combinations of this code are rarely if ever used. HYDAN uses this anomaly to replace a standard format used commonly in code with an unusual code combination (i.e. replacing an "ADD 1" function with a "SUB -1").



These code anomalies are not extremely common. As such, HYDAN is limited to being able to embed only one byte of a hidden message for approximately each 110 bytes of executable code. This is far less efficient than other steganographic applications. Many of these can hide as much as one byte within 17 bytes of a .jpeg file. This gives HYDAN a far lower rate of embedding than other methods, but this does not remove its functionality.

HYDAN is open source and includes the capability to encrypt the messages with the blowfish algorithm. To do this a passphrase needs to be included. This makes it difficult to determine what the message stored within the executable is, but a method to attempt to brute force it could be developed from the detection method presented in this paper.

Steganography, and in particular HYDAN can also be used to embed an executable file with a watermark or a digital signature. This allows the file to be marked and possibly tracked.

The Embedding Function

HYDAN processes the byte code of an executable application sequentially. In this process it is searching for instructions with functional equivalents (as noted above). Each time that such an instruction is found, it is replaced (substituted) with the alternate (and equivalent) instruction that corresponds to the data being embedded by HYDAN. This process is repeated bit by bit for the data.

For instance, the table below documents the changes made to embed the binary 1001101:

	Original code	**HYDAN Embedded Message**
1	**ADD %eax, 10**	**SUB %eax, -10**
0	**ADD %eax, 25**	**ADD %eax, 25**
0	**ADD %eax, 09**	**ADD %eax, 09**
1	**ADD %eax, 20**	**SUB %eax, -20**
1	**ADD %eax, 50**	**SUB %eax, -50**
0	**ADD %eax, 50**	**ADD %eax, 50**
1	**ADD %eax, 35**	**SUB %eax, -35**

The substitution set is: **{"ADD %reg, $imm", "SUB %reg, $imm"}.**

Each time an instruction is passed by HYDAN that has the form **"ADD %register, $imm"** a single bit of hidden message can be encoded. Where the bit value in the message equals "**0**"; the code is unchanged. Where the bit value in the message equals "**1**", HYDAN substitutes the assembly "**ADD**" function with a "**SUB**" function of the form: **"SUB % register, -$imm"**. To do this, it simply subtracts the negative of the value that originally was to be added.

For a detailed explanation of Assembly coding, see Hyde (2004); Irvine (2007); Duntemann (2000). For a specific focus on reversing see Eilam (2005).

Other equivalent instructions do exist. HYDAN does not make use of these; however, a variant on the theme could be created using these additional functions. El-Khalil and Keromytis (2003) provide a detailed list of equivalent functions in the appendix to their paper for Open BSD(see figure below).

Set of Instructions	Instruction Name	Bits	Dist w/in Set	Global Dist
add32	add r/m32, r32	31909	100.00%	1.06%
	add r32 , r/m32	0	0.00%	
addsub32-1	add eax, imm32	4576	84.66%	0.18%
	negative form	547	10.12%	
	sub eax, imm32	282	5.22%	
	negative form	0	0.00%	
addsub32-2	add r/m32, imm32	7356	44.89%	0.55%
	negative form	1470	8.97%	
	sub r/m32, imm32	7560	46.14%	
	negative form	0	0.00%	
addsub32-3	add r/m32, imm8	960798	60.86%	52.60%
	negative form	509372	32.27%	
	sub r/m32, imm8	108266	6.86%	
	negative form	174	0.01%	

The inclusion of a greater number of equivalent functions would make a greater subset of data to test. This would make detection more difficult due to a great cost in computer cycles. HYDAN does not use these other functions.

Prior to embedding the encrypted message and header HYDAN follows a random walk that skips a random number of useable byte code instructions. This is designed to amplify the amount of exertion required by a detection procedure in the hope that it will make it infeasible. The random walk is seeded using the user-supplied passphrase.

The random walk is seeded by the user-supplied passphrase to increase the detection workload. Supposedly the method proposed by Neils Provos (Provos, 2001) is utilized. This technique requires that the embedded data is distributed homogeneously throughout the original file (the cover-text). In HYDAN (El-Khalil & Keromytis, 2003), the number of bits skipped is stated to lie in a distribution of range [0, $2 \notin \frac{T_c}{T_m}$]. This is defined with the value T_c being the number of bits remaining in the original file, and T_m being defined as the remaining length of the message to be encoded. It is stated that the aforementioned interval is recalculated for every 8 bits of message that are embedded by HYDAN.

In the version of HYDAN tested (0.11) the initial embedding of the data jumped a number of instructions on the Windows XP host and did not follow the prior stated distribution. This is displayed in the section of the paper "Finding Where the Data Encoding Starts".

Version 0.13 of HYDAN was compiled on Linux and the random jump was not tested for this version as it is not essential for the detection the HYDAN to do this test. Even if the random walk function did correctly, it would not change

the results of the test as these are based on an analysis of the entire code set and not an extraction.

The Decode Function

To reverse the function and extract a hidden message with HYDAN, the embedded data in the byte code is read. When either an "**ADD**" function or "**SUB**" function is read, the message is reconstructed bit by bit as follows:

ADD % register, $imm Is read as a binary digit 0.

SUB % register, -$imm Is read as a binary digit 1.

When the message entire byte code has been read, the message length has been extracted. If the message was encrypted first, it may now pass to be decrypted using the passphrase that was originally used to encrypt the message. To extract the hidden message, HYDAN employs the user-supplied passphrase to again seed the random-walk algorithm. With this, it first extracts the length of the embedded data.

When the message has been extracted to the required length, the remaining binary stream is decrypted.

Encrypting and Decrypting the Message

HYDAN requests that a passphrase is entered to both encrypt and decrypt the message it is to embed or decode within an executable.

The process to create an encrypted message with HYDAN occurs using the following process:

1. HYDAN calculates the length of the message to be encoded.
2. HYDAN appends the message length as a header at the start of the message to be encoded.
3. The message and length header are encrypted using the CBC mode of the Blowfish algorithm. The message

length is encrypted with the data. The passphrase supplied from the user is used to secure the encrypted message and acts as a key.

4. The data is embedded into the application using the embedding method listed previously.

The process to decrypt the message works in the following manner:

1. The decode function (as defined above) is run to extract all of the bits that could be a part of a message from the data.
2. As the message length is unknown prior to the data being extracted, the entire stream needs to be processed.
3. The message is decrypted using the key that was supplied to encrypt the message. As the Blowfish algorithm is run in CBC mode (a stream cipher mode), it does not need to know the total length of the message and header before starting to decrypt it.
4. When the header is decrypted, HYDAN can then use this information to truncate the message size and only return the original message.

The process used by HYDAN is not really efficient. It does make the guessing of the message body more difficult.

Instructions for using HYDAN

The file, "*hdn_insns.c*" that was distributed in earlier versions of HYDAN has a complete list of instructions. The operation will be covered in detail later in the paper, but the main functions are embedding or decoding a hidden message. The commands to do this are:

Embedding the hidden message

To hide the message, **<message>** with the file "**/bin/ls**" where the output file is "**ls.steg.HYDAN**" in the local drive, the following command is issued:

./HYDAN /bin/ls <message> ls.steg.HYDAN

Decoding the hidden message

To reveal the message that was hidden within the file "**ls.steg.HYDAN**" which will display to STD_OUT (usually the screen), the following command is issued:

./HYDAN-decode ls.steg.HYDAN

Attacking HYDAN

The process used by HYDAN can be attacked and broken. The ease to which the steganographic function can be subverted limits HYDAN's effectiveness as a watermarking tool. There are a number of primary attack vectors that render HYDAN ineffective that range from overwriting the data (such as using HYDAN again with an alternate message) to detecting its use. It is not necessary to detect that HYDAN has been used on a file to render it ineffective using the overwrite method.

The detection of HYDAN is discussed below. Once the use of HYDAN has been detected, the message could be extracted. A brute force attack could be run against the data that has been extracted, but this is unlikely to prove effective. For HYDAN to be truly effective as a steganographic tool, an analyst should not be able to detect if a message has been incorporated into an executable (let alone be able to decrypt it). The method discussed later in the paper limits the effectiveness of HYDAN for steganographic purposes.

Overwriting

HYDAN has no defense against overwriting other than volume. Until now there has been no simple method to detect if HYDAN has been used and overwriting all executables on a system is problematic to say the least.

By simply running HYDAN over an executable with another message, the original message will be irretrievably lost. The intended recipient would than not be able to retrieve the message (assuming that this was the only version of the executable and message).

It has been noted (Slashdot, 2004) that the addition of an error correcting code to the encoding of the message coupled with an addition of an algorithm that distributes the message in a seemingly random manner throughout the binary based on the specific passphrase could correct this flaw. More effectively would be to also reduce the amount of the message and insert it multiple times.

Parts of the original message could be overwritten while still enabling the original message to be reconstructed. This of course also has flaws. By ensuring that the length of a new message is greater than the encoding fraction will allow, the message can be still be overwritten rendering this new method ineffective as well.

As noted above, the sheer volume of binary files that exist makes this problematic. To be effective, this method also needs to be used when a steganographic message is detected and not on all files. Again, detection becomes the critical component.

Detection

The main focus of this paper is on detecting the presence of HYDAN. The goal of HYDAN (like all steganographic tools) was to not exhibit any obvious patterns that could be easily detected. Additionally, the header (length) information is encrypted and embedded in a manner that does not produce an easily recognizable marker or token. Encrypting the data helps to make the distribution of the data more random.

HYDAN fails at the assembly code level. HYDAN is vulnerable to the simple statistical analysis techniques presented later in this paper, as it does not imitate the distribution of instructions normally found in a binary executable. The instructions that HYDAN replaces are functionally equivalent, but they form a pattern that seldom occurs naturally in assembly code.

Both global statistical distributions from the patterns of assembly code across programs as a whole as well as local statistical distributions as they occur in hosts or even individual software packages need to have a level of conformity after the application of HYDAN if it is to be successful.

When modifying byte code instructions it is necessary to create a method that embeds data consistently across procedures. HYDAN has not achieved this. Hence the statistical methods presented below manage to detect even the smallest HYDAN derived steganography.

Decryption

HYDAN embeds the data using a method that is not efficient, but that makes the retrieval of the data difficult. HYDAN accomplishes this through the use of the Blowfish algorithm in CBC mode. The data (which consists of the message length plus the message) is encrypted using a user-specified passphrase that acts as the key. HYDAN encrypts the entirety of the information including the message length.

This could make decryption far more difficult, except that it is possible to guess the message length. HYDAN inserts the message sequentially into the executable beginning from a random position at an arbitrary function within the executable and works to the end of the message.

HYDAN uses the Blowfish algorithm in a stream-cipher mode to encrypt and decrypt data using the passphrase supplied by the user as the key. If no passphrase is supplied, a null key is used to encrypt and hence decrypt the data.

HYDAN uses the Blowfish block cipher in cipher-block-chaining (CBC) mode. In CBC mode (Goldreich, 2004, Pp 404-408) the existing plaintext block is combined with the preceding cipher-text block using the XOR function previous to running the encryption function.

This allows the use of a block cipher (which is designed to operate on small fixed-size blocks of plaintext or ciphertext that are generally in the order of 64 or 128 bits in length) to encrypt longer messages (Goldreich, Pp 408-416, 2004).

The functions used by HYDAN to encrypt and decrypt data are included in the following table:

Encryption	***Decryption***
$C_i = E_K(P_i \zeta C_{i-1})$	$P_i = D_K(C_i) \zeta C_{i-1}$

Here, the elements of the equation are (Goldreich, Pp 404-418, 2004):

- C_i C_i is the ciphertext block at position "i" (from $C_1,..., C_n$) that is obtained after applying the block-cipher to each block of the plaintext.
- C_{i-1} C_{i-1} is the ciphertext block at position "i-1".
- E_K E_K is the function that takes block B of size b as an input and returns the encrypted block (which will also happen to be of size b).
- P_i The plaintext is partitioned into n blocks $P_1,..., P_n$ of size b. P_i is the i^{th} block.

- D_K is the function that describes the decryption operation. It is in effect the reverse of E_K.

Mathematically, a block-cipher can be seen as pair of two functions E_K and D_K that depend on a key K ().

A detailed process to capture the encrypted header length and use this as both a means to Brute force the data and also to simply determine the message length will be expounded in a follow-up paper to this one.

What is HYDAN and how is it used?

The following section acts as a tutorial on how HYDAN is installed and used.

Installing HYDAN

The source code for HYDAN may be downloaded from: http://www.crazyboy.com/HYDAN.

To install HYDAN on Linux/Unix, simply extract the source code and then compile it. The following example demonstrates this process:

```
$ cd /usr/local/bin

$ tar -xvfz HYDAN-0.13.tar.gz

$ cd /usr/local/bin/HYDAN

$ make
```

The system will compile and install the HYDAN binary. The process is more complex on Windows and will require a different methodology based on the compiler used.

I have used Turbo C++ Explorer to compile HYDAN in Windows. This can be downloaded free from: http://cc.codegear.com/free/turbo.

Running HYDAN

Once HYDAN is installed, running it is simple. The command is:

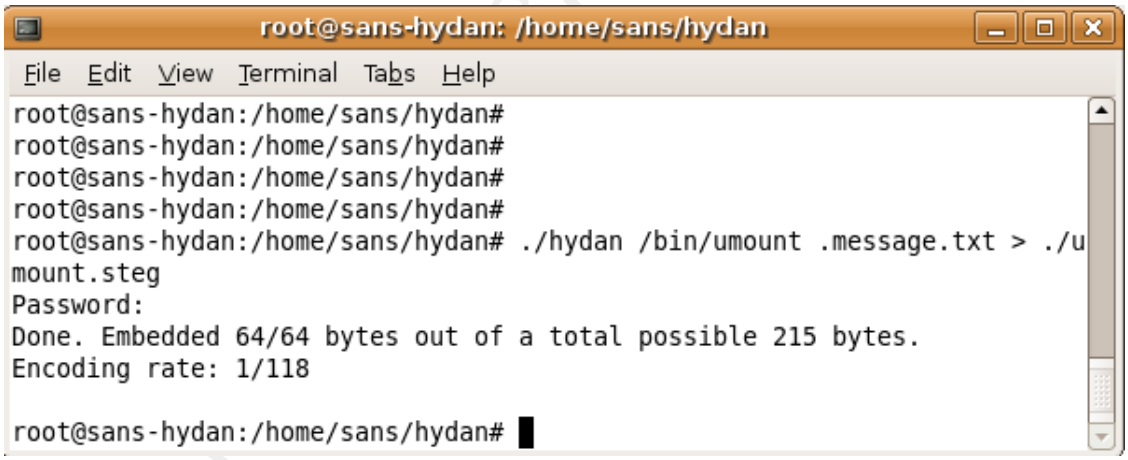
/usr/local/bin/HYDAN/HYDAN /path/binary /path/message_to_add > /path/updated_binary

In this instance, HYDAN has been installed in the directory, **/usr/local/bin/HYDAN.** The file, **/path/binary** is the binary to be encoded. The message is contained in **message_to_add**. The output binary with the encoded message is now **/path/updated_binary**. The program will prompt the user for a password and then proceed to encode the message.

The newly created binary will not have the same permissions and the new binary with the encoded message needs to have the execute permission set.

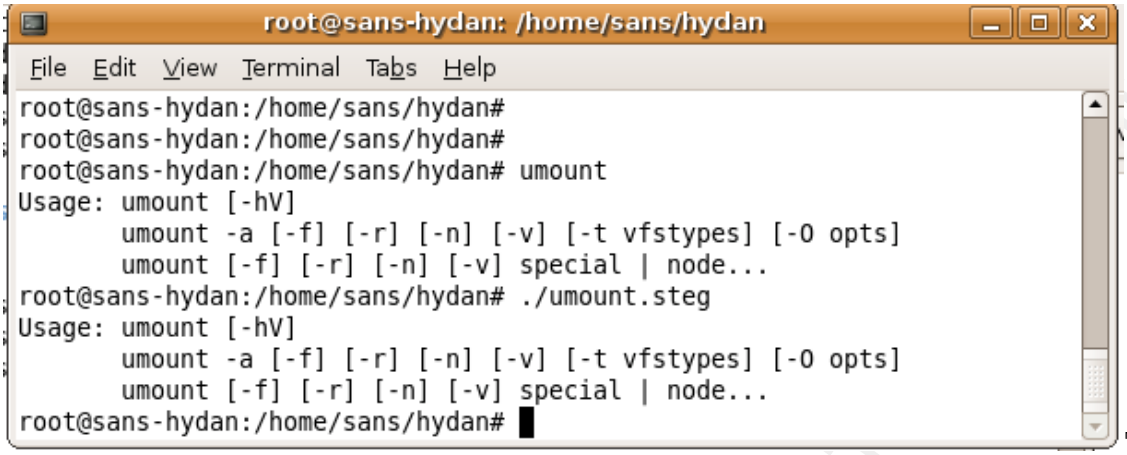
```
chmod u+x /path/updated_binary
```

The timestamp of the file will have changed. With sufficient privileges, a skilled UNIX or Windows used can change the timestamp to match that of the former file.

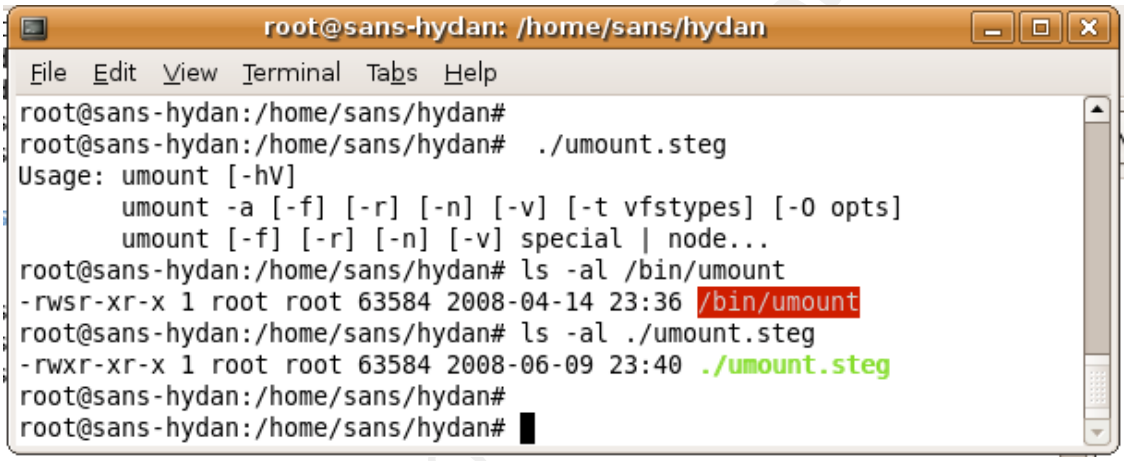


Comparing the newly created binary

The new binary will perform precisely as the previous command without the encoding. It will fail to meet an integrity check if a hashing program (such as AIDES or Tripwire) has been used. For some selected binaries on platforms such as Redhat Linux, the hash is saved for use with the **pkgadd** command. This will also detect the change.



The file has not changed in size.



Overwriting the old binary with the new

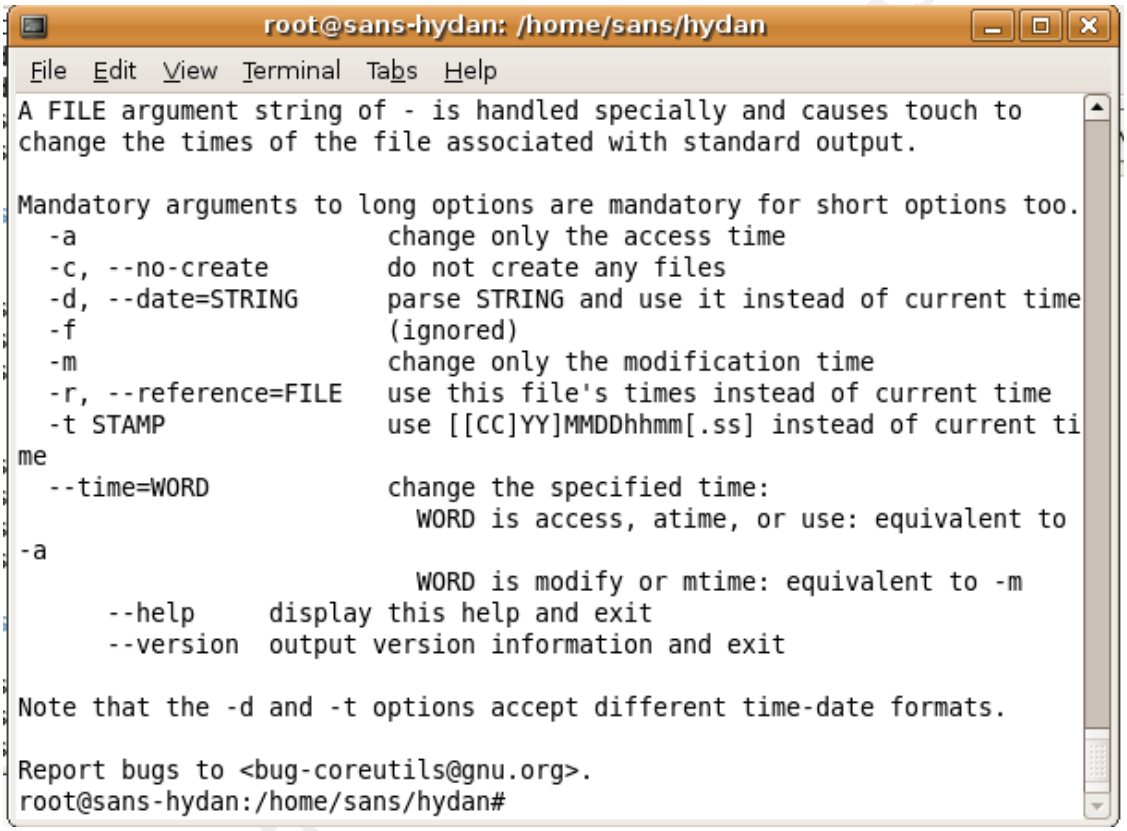
In many cases it will be necessary to have root privileges to be able to overwrite the old binary. In the case above, the shell was already running as root, but this may not be the situation for an attacker or user in general.

In the example given in the previous section, the following commands also need to be used to move the binary into its original directory and to set the permissions to match that of the original.

```
mv ./umount.steg /bin/umount

chown root:root /bin/umount

chmod 755 /bin/umount
```

The timestamp of the resultant binary will not be the same as the original. In UNIX/Linux, the **touch** command can be used to change the timestamp of the newly created binary so that it matches that of the former binary without encoding.

The command: `ls -al /path/binary` can be used to obtain the timestamp of the original binary before encoding.



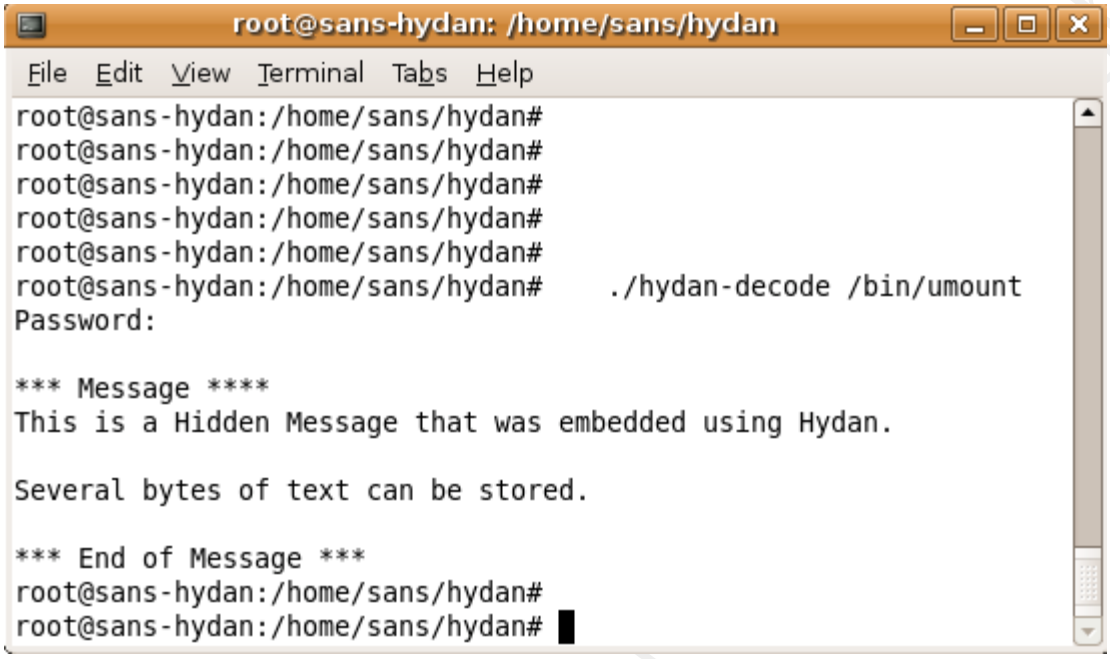
The command, **touch**, can then be used to update the timestamp so that it matches the original binary as closely as possible.

Decoding the message

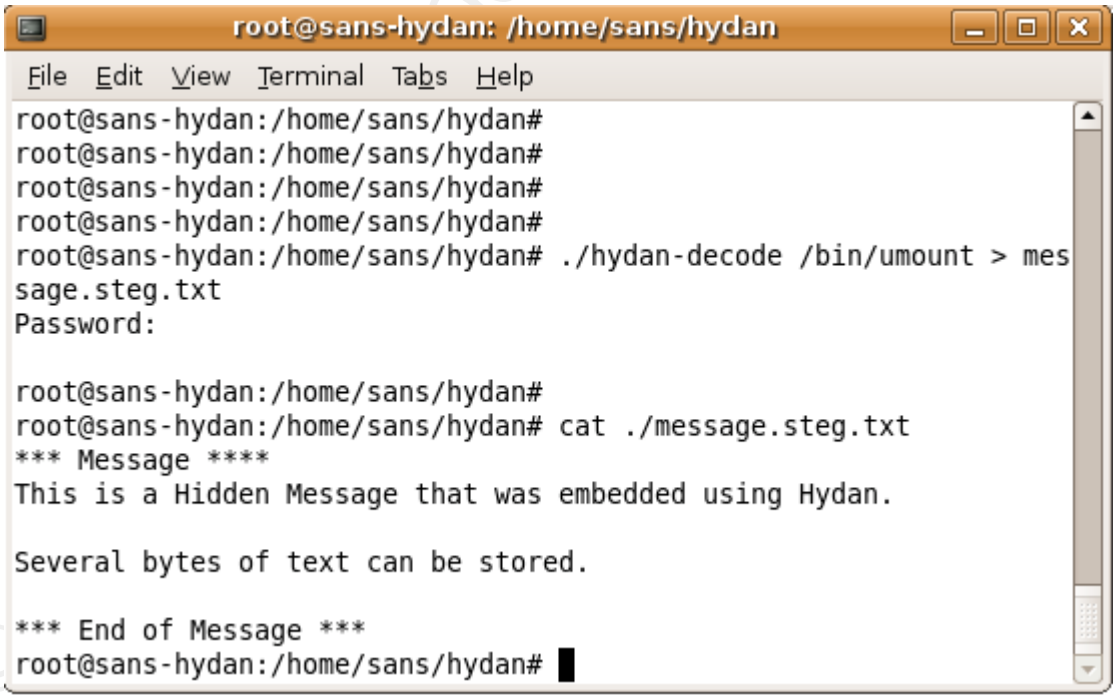
As long as you know which file the message is in and have the password, decoding it is simple. The command below and the image demonstrate this process.

`/usr/local/bin/HYDAN/HYDAN-decode /path/binary_with_message`

HYDAN will prompt for the password that was used to encode the message.



The hidden message may also be extracted to a file rather than being sent to STD-OUT (as stated – usually the screen). This is demonstrated in the image below:



The hidden message when recovered is identical to the original.

HYDAN Detection

There are a few means to discovering HYDAN. The simplest involves having either the original file or a checksum of the file. The issue comes in discovering steganographic information in files that have not been imaged or otherwise where the investigator has no recourse to checking the original file.

Method 1 - Checksums

The checksum of the original binary does not match with the resultant HYDAN produced binary. This allows the investigator to determine that the file has been altered, but not that it was altered using HYDAN. As such this provides no evidence that HYDAN has been used on the file.

An attempt can be made to attempt to extract text from these altered files using HYDAN and a guessed pass phrase. It is unlikely that this method would result in the detection of HYDAN.

Method 2 – Statistics

The primary focus of this research was to create a method to detect steganographically encoded messages that have been created using HYDAN. This was achieved using existing disassembly tools and the powerful "**R**" statistical language.

R has the functionality to call external programs. Using this capability, R can call an external disassembler, capture the byte code instructions (the de-compiled assembly language of the executable) that the disassembler outputs and feed these values into an array.

This information is then sorted to select a subset that contains only the instructions of interest. In the case of HYDAN, these are limited to:

ADD % register, $imm and

SUB % register, -$imm

Each ADD command is recorded into a separate variable as a value "0" and the SUB command where the value being subtracted is negative is encoded in the variable as a "1". Any SUB commands that have a positive value that is to be subtracted are discarded.

A baseline of the local and global distributions was estimated. To do this, a random selection of 500 binary executable files was selected using the rnorm() random normal function from R and a list of files from the systems tested was saved into an array.

The following systems where tested:

- Windows XP SP2
- Linux.

The details of the resulting distributions are included in the following sections.

A random selection of 10 files was encoded on both systems under test. These files were encoded using HYDAN to embed a message into each of the binary files. The processes listed above where used to embed the message and to extract it. Each message was successfully extracted from the steganographically encoded file.

The distribution of byte code instructions in the HYDAN encoded files was compared to both the local (the system it was created on) and global (the concatenation of both systems) distributions. The results of this experiment are included below with the R function that was used and a number of statistical visualizations.

R (a Statistical Programming Language)

The analysis of the data was conducted using the R statistical language. The functions have been left in an

interpretive mode for this paper, but there are compilers that support the R language. In addition, RServe is a precompiled engine that can be used to run the interpreted language as a batch script. It is available from http://www.rforge.net/Rserve/faq.html.

What is R and where do I get it?

As is noted on the primary website (http://www.r-project.org/), "*R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS*".

R is available from of the many mirrors listed at http://cran.r-project.org/mirrors.html. It may be downloaded and installed freely. A number of graphical front-ends (such as Rattle from http://rattle.togaware.com/) exist to simplify the process of using and deploying R.

Reading in the data

R has the capability to make remote system calls. This can be used to call other programs from within R. An example would be using R system calls to open a URL with Mozilla:

```
system(paste('"c:/Program Files/Mozilla Firefox/firefox.exe"',
             '-url www.sans.org/rr), wait = FALSE)
```

In this case, the command to be run should be loaded as a batch file in Windows or a Shell script in Unix. The windows script below is designed for a single file.

```
"c:\data\dis.exe c:\windows\system32\cacls.exe >
c:\data\calcs.asm"
```

Using variable input this can be increased to test multiple files or to take the output of a file listing as the input feed. The command lists the disassembled program code. The output format seems difficult to understand and interpret, but the secret lies in understanding that only a limited set of data from this output is needed for the statistical tests.

All that is needed is a small component of the code.

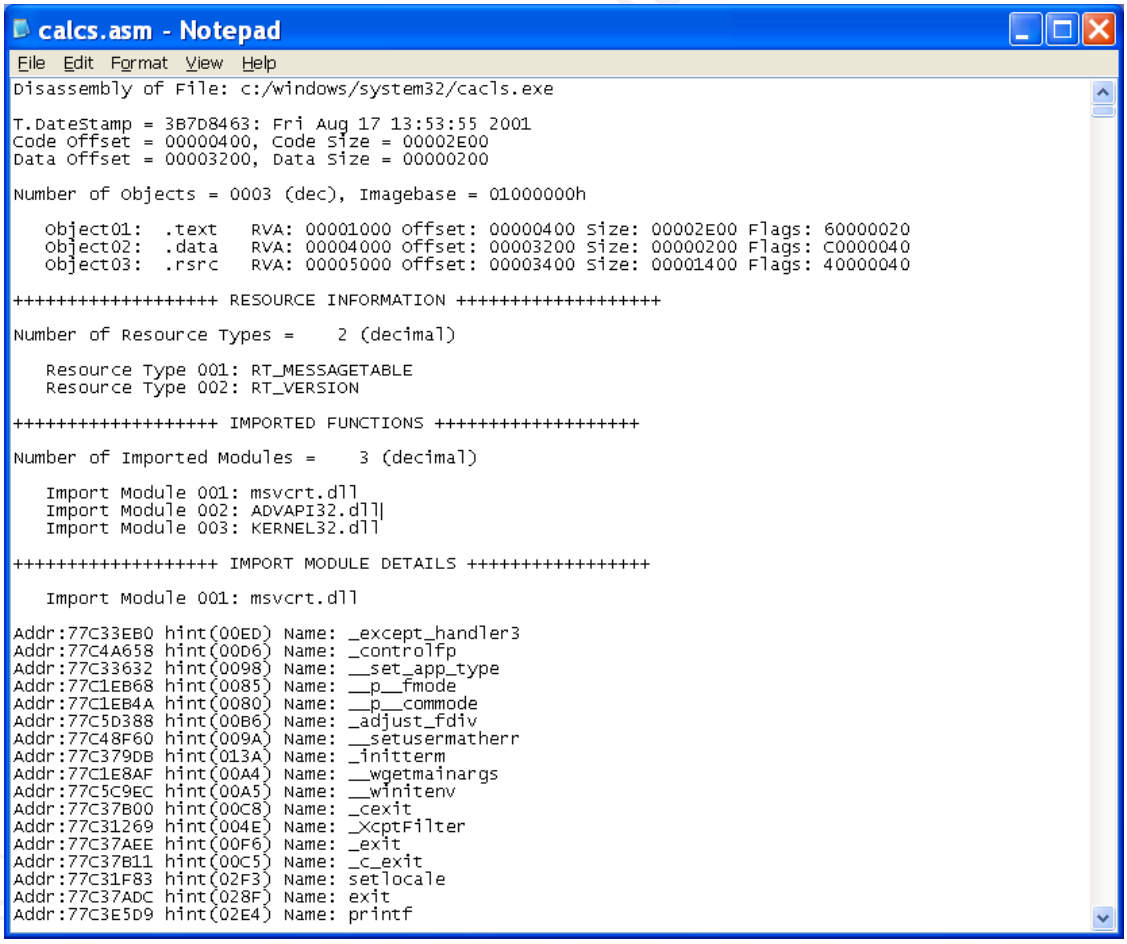
ADD % register, $imm becomes ADD,$imm

SUB % register, -$imm becomes SUB, -$imm

In fact, all that is finally needed is the "-" sign. Actually, a simple command such as the one below can be used to read the data into R. This results in a lower detection rate as non-tested commands are also loaded:

```
Test.HYDAN <- read.csv("c:/data/calcs.asm", header=FALSE)
```

The loss of information is minimal as the function is actually simple to detect (as will be demonstrated below). The better option would be to use a call to Perl to run a REGEX call (regular expressions).



Example of the disassembled output from cacls.exe

This reduces the output in the data file to be tested and adds a table such as the one below:

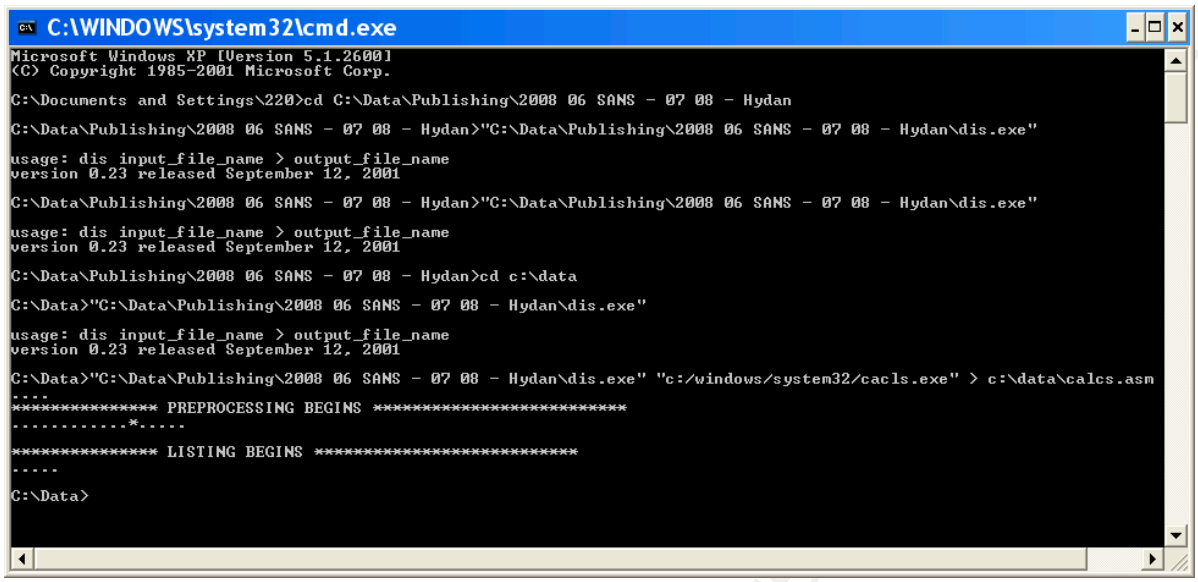
```
> Test.HYDAN <- read.csv("c:/data/example.csv", header=FALSE)
> Test.HYDAN
     V1       V2
1   sub       10
2   add      eax
3   add      eax
4   sub       10
5   add      eax
6   add        8
7   sub       14
8   add      esi
…
```

Disassembling the binary

The Windows XP version of the code is displayed. For this exercise the following disassemblers were tested:

- **Windows XP:** Win32 Program disassembler http://www.geocities.com/~sangcho/disasm.html
- **Linux:** Perl x86 Disassembler description http://linux.softpedia.com/get/Programming/Disassemblers/Perl-x86-Disassembler-1155.shtml

The Linux option is the simplest to port as it runs in Perl. It is simple to change the R code below to run this function. The R code to call an application is included below. Alternatively, the **Win32 Disassembler** is portable to Linux and the source file has details on this process. This disassembler had periodic problems with Windows XP running with SP2 +.



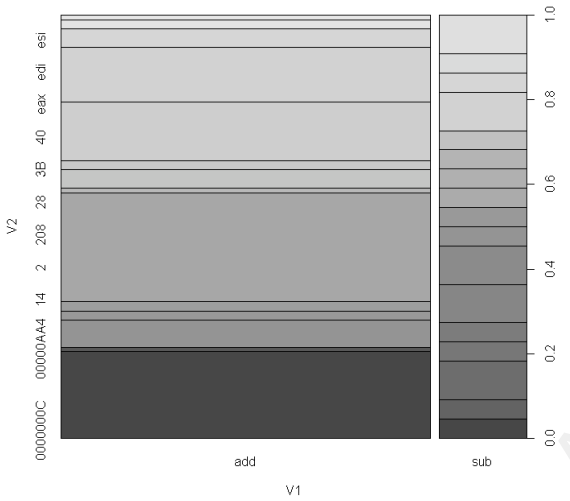
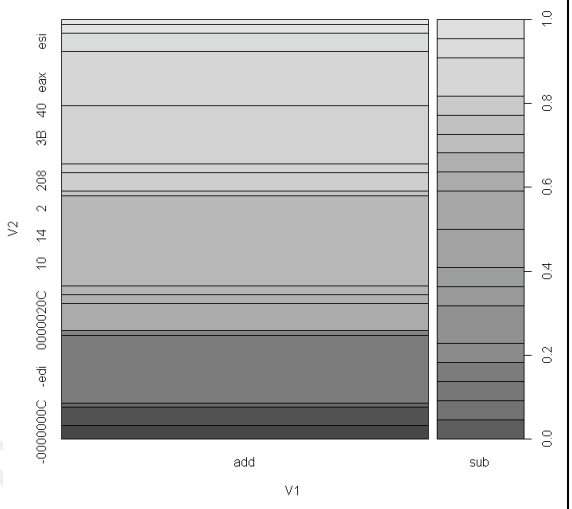
Detecting HYDAN

HYDAN is surprisingly simple to detect when the extra data is removed. All that is sought in the data is the negative sign. In all cases where the assembly language equivalent function; "**SUB, -value**" or "**Add, -value**"; is found in the code, the encoded binary is likely to be an indicator of the use of HYDAN. The proof of this assertion is included below. As such, any significant increase in the equivalent function is an indicator of the use of HYDAN.

By simply examining the table (Test.HYDAN), any negative value that is found in column V2 demonstrates significant evidence at the alpha=0.1 level that HYDAN has been used. The ordinary distribution of negative values, or for that matter, equivalent assembly language functions is so low as to produce statistically significant results for even the smallest message.

(*alpha=0.1% is far in excess of normal scientific or even legal requirements of proof that generally look at alpha* = 5%. For further details on statistical significance, see http://www.statsoft.com/textbook/stathome.html. A number of other statistical references have been included at the end of the paper).

A simple plot of the data that demonstrates any significant level of difference is all that is required to determine the use of HYDAN visually and a simple t-test can be coded to automatically determine the use of HYDAN.

Before HYDAN	**After HYDAN**
	

These plots are made as simply as with the commands:

```
> plot(Test.HYDAN2)
> plot(Test.HYDAN)
```

In this instance, the dataset, "Test.HYDAN" is the set of data from the file where HYDAN has not been used. The dataset, "Test.HYDAN2" is the output from the file where HYDAN was used to hide a small (1kb) message.

Even at the low encoding rate of $1/110$ (or 1 in every 110 bits) stated by El-Khalil and Keromytis (2003), HYDAN is easily detected. Further testing did demonstrate that HYDAN was extremely difficult to conclusively detect at the alpha =5 level for an encoding rate of around $1/15{,}000$. The problem with this is that this makes the use of HYDAN ineffective as a means of encoding steganographic data as a file of about 15Mb is needed to encode a 1Kb hidden message.

The Distribution

To create a distribution, 500 files on both Windows and Linux where tested. These where combined to form a global dataset.

The summary results of the Global datasets

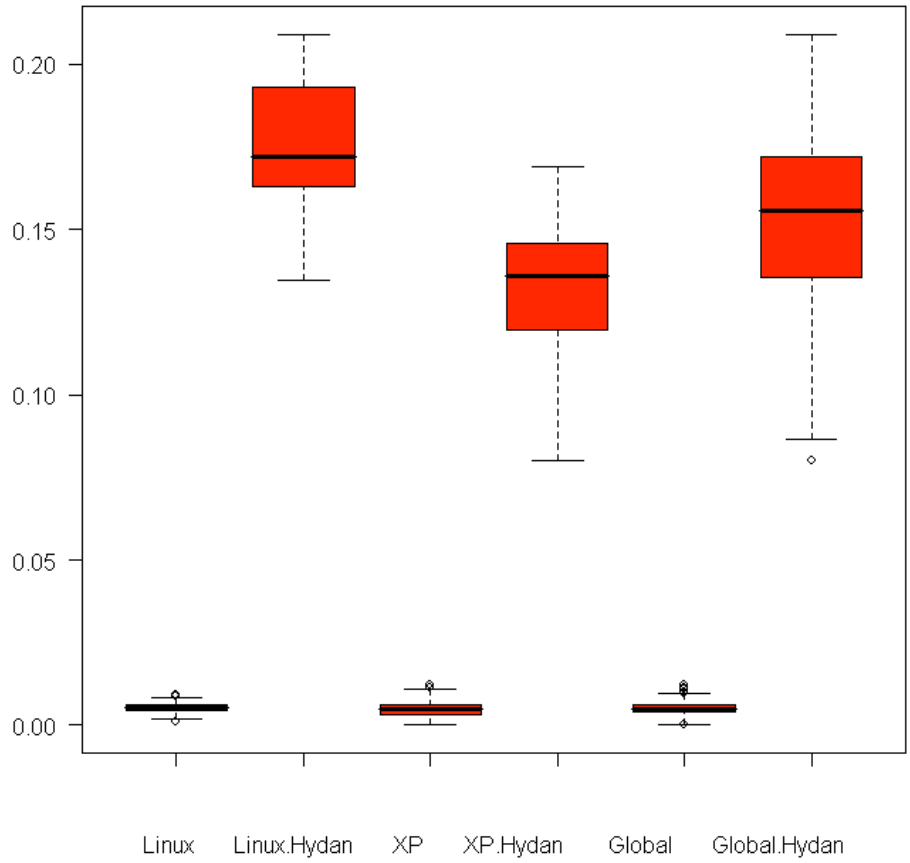
```
> summary(Global) # The files that have not had HYDAN used on them
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
0.0001041 0.0038660 0.0050950 0.0050350 0.0061710 0.0123800
> summary(Global.test)# HYDAN used on these files
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.08021 0.13560 0.15580 0.15140 0.17120 0.20910
>
```

A simple z or t-test will statistically determine even the simplest datasets. To run a simple t-test in R we use the following command:

```
> t.test(Global.test-Global)
        One Sample t-test

data:  Global.test - Global
t = 140.8821, df = 999, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 0.1443012 0.1483779
sample estimates:
mean of x
0.1463395
>
```

In the above example, we have tested the Global set against the test data. "Global.test" is the set of files where HYDAN was used. "Global" is the set of files that occurred naturally on the system. The bloxplot below demonstrates the

distribution of data with HYDAN encoding and naturally occurring binary distributions.



Finding the starting point of HYDAN (with a measure) for error is simple. By converting all negatives (those register values that have been switched by HYDAN) to be represented by a "1" and letting the other values be a "0" we can plot the dataset ("plot(HYDAN.data)") . From this on a single program we can visually see the presence of HYDAN in the code as well as having a good starting point to determine where the random walk function started adding the data.

Once the data is loaded, a simple test is required to see if the mean value of the distribution is zero. The testing process is simple. We take the extracted dataset and test whether this equals the natural distribution. Alternatively,

we could test against a means of "0", but for the exercise we shall test the natural distribution.

```
> t.test(linux.test.1, mu=mean(linux1), conf.level=0.001)

        One Sample t-test

data:  linux.test.1
t = 25.0016, df = 9, p-value = 1.258e-09
alternative hypothesis: true mean is not equal to 0.0052126
0.1 percent confidence interval:
 0.1741266 0.1741441
sample estimates:
mean of x
0.1741354

> t.test(XP.test.1, mu=mean(XP1), conf.level=0.001)

        One Sample t-test

data:  XP.test.1
t = 14.1112, df = 9, p-value = 1.915e-07
alternative hypothesis: true mean is not equal to 0.004856961
0.1 percent confidence interval:
 0.1286020 0.1286246
sample estimates:
mean of x
0.1286133

>
```

As can be seen in either case (p=1.258e-09 and p=1.915e-07 respectively for Linux and XP) the chances of finding HYDAN encoding are overwhelming.

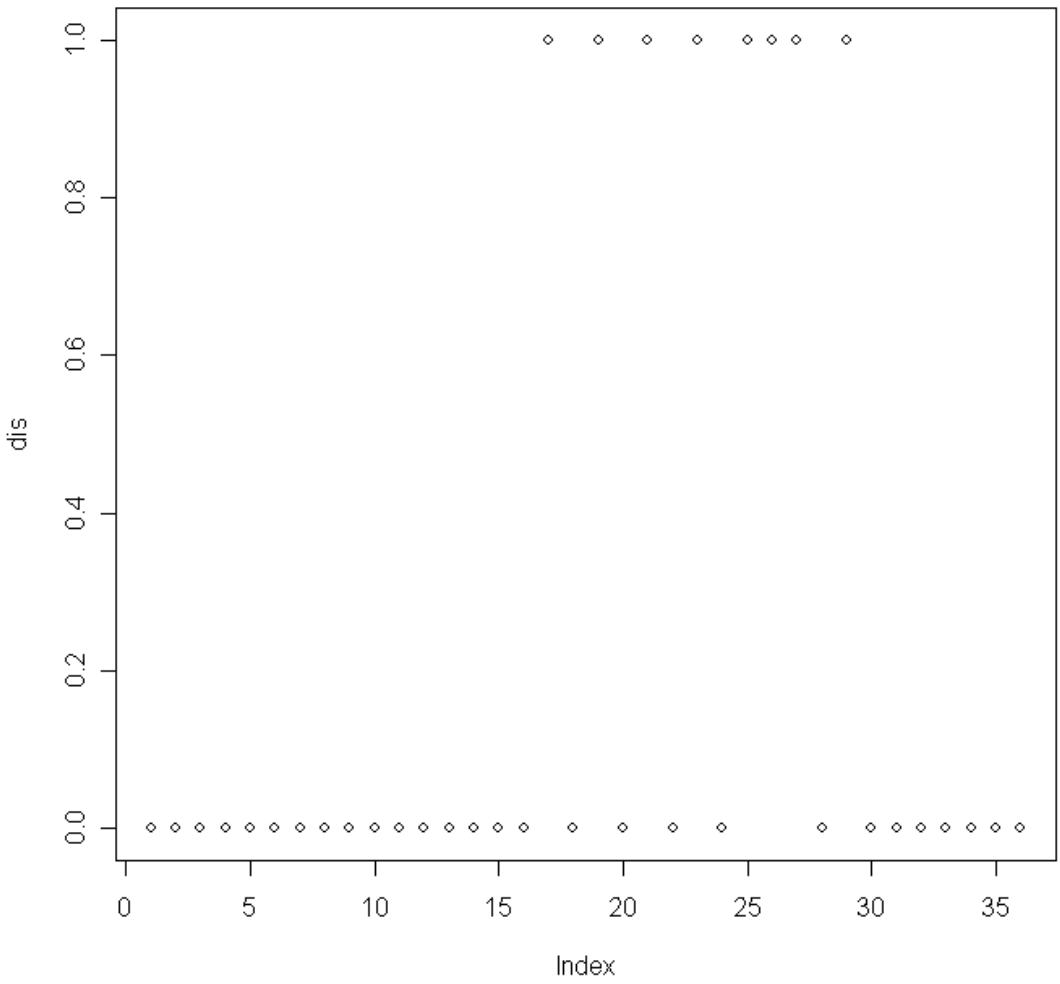
All tests demonstrate that the distributions of values that represent "negatives" or those values that have been reversed by HYDAN are significantly different than those that occur naturally.

Even if the testing was to be set with a mean larger than that of the expected range, the embedding of even small amounts of data is detectable.

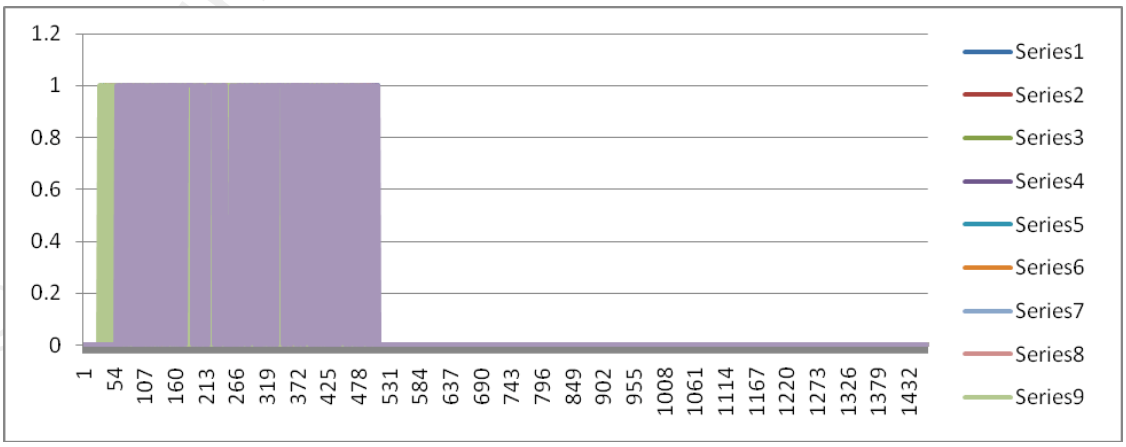
```
> t.test(linux.test.1, mu=mean(0.01), conf.level=0.001)

        One Sample t-test

data:  linux.test.1
t = 24.2931, df = 9, p-value = 1.624e-09
alternative hypothesis: true mean is not equal to 0.01
0.1 percent confidence interval:
 0.1741266 0.1741441
sample estimates:
mean of x
0.1741354

>
```

Finding Where the Data Encoding Starts

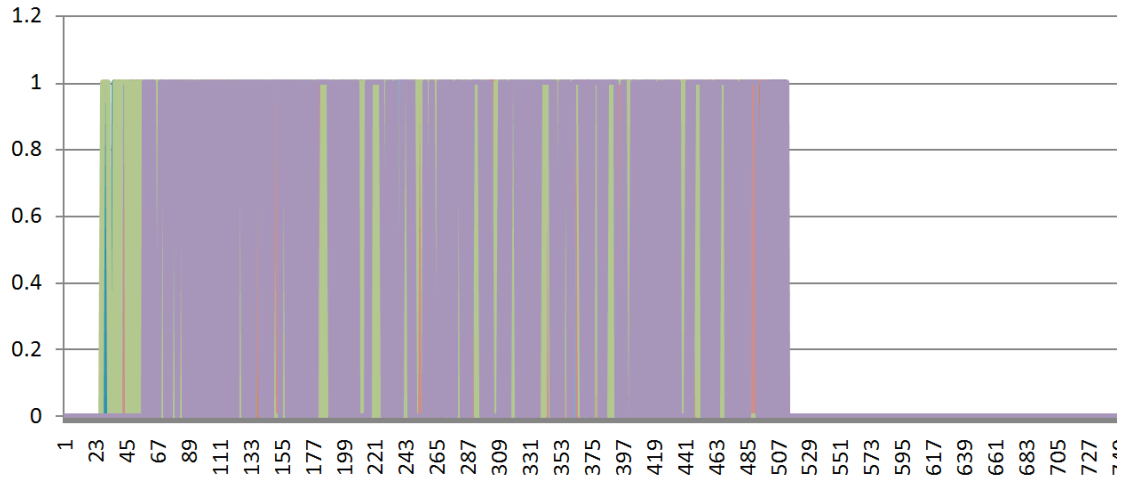
Even finding the start of the data is simple. From the plots below, we can see the probable position of where the data has started to be encoded.



As can be seen from the plot, we can visually deduce that HYDAN has started adding data between command 15 and 17 with a reasonable level of confidence.



A close-up demonstrates this better. The sample of 10 files was again plotted as before. Again the start of the HYDAN encoding is visually available.



The start location of the HYDAN data does vary, but it is simple to determine the likely start location visually. The random walk algorithm described by El-Khalil and Keromytis (2003) does not describe the distribution actually found. It is uncertain if this would be a consequence of the platform used for this section of testing (Windows XP) or if there was some other flaw.

What this means for HYDAN (or Future Lessons)

El-Khalil & Keromytis (2003) state that the methodology used in HYDAN is based on the work of Provos (2001) stating that the method proposed by Provos is used to increase the entropy of the embedding process.

HYDAN creates a noticeable distortion in the natural distribution of instructions. If HYDAN or another tool was to change the instruction sets used or to even use multiple equivalent instruction sets simultaneously, this would do little to significantly reduce the ease to which HYDAN can be discovered.

Provos (2001) however stated that:

"Detectability is also used as a bias in the selection process. The selector does not try to reduce only the number of changed bits but also the overall detectability.

Whenever a bit has to be modified, its detectability will be added to a global bias. A higher accumulated bias reduces the likelihood that this specific embedding will be used".

Zollner et al. (1998) advocate that there are two essential stipulations that are required to produce a secure steganographic function:

- *The secret key used to embed the hidden message is unknown to the adversary.*
- *The adversary does not know the cover medium.*

The issue with HYDAN is the latter of these requirements. The distributions of alternative assembly instructions are low enough in the wild to ensure that any use of HYDAN will be easily detected. This makes the use of HYDAN for any operational purpose infeasible.

Even the proposed enhancements to HYDAN (El-Khalil & Keromytis, 2003, Pp. 7-8) fail to offer any improvement. The set of possible alternative and equivalent instructions is too unevenly distributed. The methods presented in this paper can be simply changed to incorporate the testing of all equivalent instructions with little if any overhead.

In creating a steganographic process, the developer needs to find a model where the distribution of the original medium is unknown to the adversary. HYDAN has not achieved this. Unlike image based steganography, any preprocessing step designed to introduce randomness into a cover medium based on binary code significantly alters the natural distribution of the instruction sets.

This makes the process of deducing the existence of an embedded message trivial as it is not possible to embed a message of any significant size while preserving the natural distribution of the cover medium.

Plausible Deniability

The methods of "Plausible Deniability" proposed by Provos (2001, p. 7) could be incorporated into HYDAN to make the detection of a "true" message more difficult. The addition of capability to embed additional messages would allow the person who created a message to hand over the pass-phrase of an innocuous message and claim that only a single message was embedded into the covertext.

However, even this is problematic. If the passphrase for the innocuous message is handed over, the process of comparing the sample with a single message and the captured message is simple. To do this, the analyst could simply recreate the message as follows:

1. Using an original copy of the binary, the passphrase that has been handed over and the message that has been recovered, re-run the embedding process with the captured message.
2. Compare the binary that was originally captured with the newly created one.
3. If there is any difference, the existence of a second (or further) message would be determined.

Being that HYDAN uses the user supplied pass-phrase as the seed for its random walk, the repeated use of HYDAN with the same message, passphrase and binary will always result in an identical output. That is, there is no randomness between use of the program.

As Zöllner et al. (1998) state, "*An advanced solution to this problem is to have an indeterministic embedding*

operation. An indeterministic operation or process gives different results (within a certain range) every time it is computed. In other words, it contains randomness".

This could be incorporated to create a detectable program that still allowed for plausible denial.

Conclusion and Future Research

HYDAN is not particularly difficult to detect statistically. This paper presented a preliminary method that could be further refined into a production level tool if the need to detect HYDAN or a future variant was required. The R detection function could be compiled using an R code compiler rather than leaving it running in an interpreted mode as was done in this paper.

Statistical tools such as R provide an excellent tool for the analysis of data from computer systems and networks. These statistical tests could be expanded to uncover other forms of steganography. The methods in this paper have demonstrated that it is not necessary to analyze the entire binary executable as was supposed by the author of HYDAN. The distribution of functionally equivalent but uncommon byte code instructions becomes statistically significant well before the entirety of these functions have been analyzed.

Future research efforts have started to detail the process required to capture the encrypted header length and use this as both a means to Brute force the data and determine the message length.

Bibliography

Duntemann, Jeff (2000) "*Assembly Language Step-by-Step*" Wiley Press USA

Eilam, Eldad (2005) "*Reversing, the Secrets of Reverse Engineering*" Wiley Press USA

El-Khalil, Rakan & Keromytis, Angelos D. (2003) "*HYDAN: Hiding Information in Program Binaries*" Department of Computer Science, Columbia University in the City of New York, http://www1.cs.columbia.edu/~angelos/Papers/HYDAN.pdf

Hyde, Randall (2004) "*Write Great Code. Volume 1: Understanding the Machine*", No Starch Press.

Goldreich, Oded (2001) "Foundations of Cryptography I", Cambridge University Press, UK

Goldreich, Oded (2004) "Foundations of Cryptography II", Cambridge University Press, UK

Irvine, Kip R. (2007) "*Assembly Language for Intel-Based Computers*" 5th Ed. Pearson, Prentice Hall USA

Johnson, N.F. & Jajodia, S. (1998) "*Exploring Steganography: Seeing the Unseen*" Computer, vol. 31, no. 2, 1998, pp. 26-34.

Kuhnert, Petra & Venables, Bill (2005) "*An Introduction to R: Software for Statistical Modelling & Computing*". Cleveland, Australia. http://www.csiro.au/resources/Rcoursenotes.html

Maindonald, J. H. (2004) "*Using R for Data Analysis and Graphics: Introduction, Code and Commentary*". Centre for Bioinformation Science, Australian National University. http://wwwmaths.anu.edu.au/~johnm/ & http://www.maths.anu.edu.au/~johnm/r/usingR.pdf

McGill, L. (2005) "*Steganography: The Right Way*" SANS Reading Room, https://www2.sans.org/reading_room/whitepapers/stenganography/1584.php

Paradis, Emmanuel, (2004) "R for Beginners", http://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf.

Provos, Neils, & Honeyman, Peter. (2003), "*Hide and Seek: An Introduction to Steganography*" IEEE Security and Privacy, May/June 2003; IEEE Computer Society.

Provos, Neils (2001) "*Defending Against Statistical Steganalysis*". In: Proceedings of the 10th USENIX Security Symposium.

Richmond, J. A. (1998) "*Spies in Ancient Greece*" Greece & Rome, Second Series, Vol. 45, No. 1 (Apr., 1998), pp. 1-18, Cambridge University Press on behalf of The Classical Association.

SANS (2007) "*SEC 504*" SANS USA Courseware

Slashdot, (2004) "*HYDAN: Steganography in Executables*" Thu. Aug 12, 2004 http://slashdot.org/article.pl?sid=04/08/12/2051219

Wand, Matt., (2004) "*Fundamentals of R. A "Hands-On" Tutorial*", Department of Statistics, University of New South Wales http://web.maths.unsw.edu.au/~wand/web232/r-tut.txt & http://web.maths.unsw.edu.au/~wand/binf3001.html

Zollner, J., Federrath, H., Klimant, H., Ptzmann, A., Piotraschke, R., Westfeld, A., Wicke, G. & Wolf. G. (1998) "*Modelling the Security of Steganographic Systems*". In Proceedings of Information Hiding - Second International Workshop. Springer-Verlag, April 1998.

Websites

HYDAN - http://www.crazyboy.com/HYDAN/

Matt Wand's Bioinformatics course web page - http://web.maths.unsw.edu.au/~wand/binf3001.html

Null Hypothesis (Wikipedia) http://en.wikipedia.org/wiki/Null_hypothesis

R Windows release download - http://cran.r-project.org/bin/windows/base/release.htm

R graph library - http://addictedtor.free.fr/graphiques/allgraph.php

The R Project- http://www.r-project.org/

Significance Tests and The Null and the Alternative Hypothesis

http://www.bized.co.uk/timeweb/crunching/crunch_experiment_expl.htm

Statistical hypothesis testing (Wikipedia)

http://en.wikipedia.org/wiki/Statistical_hypothesis_testing

Wikipedia, article on box plot - http://en.wikipedia.org/wiki/Box_plot

Statistical References

Carlin, B.P. & Louis T.A. (2000) "*Bayes and Empirical Bayes Methods for Data Analysis*", Chapman and Hall.

Casella, George & Berger, Roger L (2002) "*Statistical Inference*" Duxbury Advanced Series

Congdon, P (2001). "*Bayesian Statistical Modelling*", Wiley

Dobson, Annette J. (2002) "An Introduction to Generalized Linear Models" 2nd Ed. CHAPMAN & HALL/CRC

Gelman, Andrew et al., (2003) "*Bayesian Data Analysis*", 2nd Edition; Chapman & Hall/CRC, London

Gilks, W.G., Richardson, S. & Spiegelhalter, D.J. (1995) "*Markov Chain Monte Carlo in Practice*", CRC Press.

Givens, Geof H. & Hoeting, Jennifer A. (2005) "Computational Statistics" Wiley

Madigan, David (2006) {Course notes to - BAYESIAN DATA ANALYSIS} http://www.stat.rutgers.edu/~madigan/bayes06/

Maindonald, John & Braun, John (2004) "Data Analysis and Graphics Using R, An example based approach" Cambridge University Press

Rice, John A. (1999) "*Mathematical Statistics and Data Analysis*" Duxbury Press

© SANS Institute 2008, Author retains full rights.

Last Updated: July 17th, 2018

Upcoming SANS Training

Click here to view a list of all SANS Courses

SANS Riyadh July 2018	Riyadh, SA	Jul 28, 2018 - Aug 02, 2018	Live Event
SANS Pittsburgh 2018	Pittsburgh, PAUS	Jul 30, 2018 - Aug 04, 2018	Live Event
Security Operations Summit & Training 2018	New Orleans, LAUS	Jul 30, 2018 - Aug 06, 2018	Live Event
SANS Hyderabad 2018	Hyderabad, IN	Aug 06, 2018 - Aug 11, 2018	Live Event
Security Awareness Summit & Training 2018	Charleston, SCUS	Aug 06, 2018 - Aug 15, 2018	Live Event
SANS Boston Summer 2018	Boston, MAUS	Aug 06, 2018 - Aug 11, 2018	Live Event
SANS San Antonio 2018	San Antonio, TXUS	Aug 06, 2018 - Aug 11, 2018	Live Event
SANS August Sydney 2018	Sydney, AU	Aug 06, 2018 - Aug 25, 2018	Live Event
SANS New York City Summer 2018	New York City, NYUS	Aug 13, 2018 - Aug 18, 2018	Live Event
SANS Northern Virginia- Alexandria 2018	Alexandria, VAUS	Aug 13, 2018 - Aug 18, 2018	Live Event
SANS Krakow 2018	Krakow, PL	Aug 20, 2018 - Aug 25, 2018	Live Event
Data Breach Summit & Training 2018	New York City, NYUS	Aug 20, 2018 - Aug 27, 2018	Live Event
SANS Chicago 2018	Chicago, ILUS	Aug 20, 2018 - Aug 25, 2018	Live Event
SANS Prague 2018	Prague, CZ	Aug 20, 2018 - Aug 25, 2018	Live Event
SANS Virginia Beach 2018	Virginia Beach, VAUS	Aug 20, 2018 - Aug 31, 2018	Live Event
SANS San Francisco Summer 2018	San Francisco, CAUS	Aug 26, 2018 - Aug 31, 2018	Live Event
SANS Copenhagen August 2018	Copenhagen, DK	Aug 27, 2018 - Sep 01, 2018	Live Event
SANS SEC504 @ Bangalore 2018	Bangalore, IN	Aug 27, 2018 - Sep 01, 2018	Live Event
SANS Wellington 2018	Wellington, NZ	Sep 03, 2018 - Sep 08, 2018	Live Event
SANS Amsterdam September 2018	Amsterdam, NL	Sep 03, 2018 - Sep 08, 2018	Live Event
SANS Tokyo Autumn 2018	Tokyo, JP	Sep 03, 2018 - Sep 15, 2018	Live Event
SANS Tampa-Clearwater 2018	Tampa, FLUS	Sep 04, 2018 - Sep 09, 2018	Live Event
SANS MGT516 Beta One 2018	Arlington, VAUS	Sep 04, 2018 - Sep 08, 2018	Live Event
Threat Hunting & Incident Response Summit & Training 2018	New Orleans, LAUS	Sep 06, 2018 - Sep 13, 2018	Live Event
SANS Baltimore Fall 2018	Baltimore, MDUS	Sep 08, 2018 - Sep 15, 2018	Live Event
SANS Alaska Summit & Training 2018	Anchorage, AKUS	Sep 10, 2018 - Sep 15, 2018	Live Event
SANS Munich September 2018	Munich, DE	Sep 16, 2018 - Sep 22, 2018	Live Event
SANS London September 2018	London, GB	Sep 17, 2018 - Sep 22, 2018	Live Event
SANS Network Security 2018	Las Vegas, NVUS	Sep 23, 2018 - Sep 30, 2018	Live Event
SANS DFIR Prague Summit & Training 2018	Prague, CZ	Oct 01, 2018 - Oct 07, 2018	Live Event
Oil & Gas Cybersecurity Summit & Training 2018	Houston, TXUS	Oct 01, 2018 - Oct 06, 2018	Live Event
SANS Brussels October 2018	Brussels, BE	Oct 08, 2018 - Oct 13, 2018	Live Event
SANS Pen Test Berlin 2018	OnlineDE	Jul 23, 2018 - Jul 28, 2018	Live Event
SANS OnDemand	Books & MP3s OnlyUS	Anytime	Self Paced