Engineering Simple AntiMalware Tools

Tanveer Salim

February 2019

1 LICENSE Notification

2 Specifiction

IPv6 currently suffers from many security bugs, and in the future as applications requiring IPv6 continue to rise, the risk in using IPv6 will increase. The most influential security software that protects user data in IPv6 will be done by experts on secure coding in C. Undergraduate students that are proficient in C and how pointers are used in the language may contribute to open source projects that fix security patches in currentmost IPv6 stacks.

Implementing a Secure Sockets Layer/Transport Security Layer is an excellent method to prepare undergraduate college students for Linux kernel development. It forces them to master the C programming language, especially with how pointers are used in C. The security layers encrypt data as they are transported to machines and are the foundation for the HTTPS port in IPv4 and IPv6.

3 Instructions to Add Object Code to Library Archives

Instructions on how to add object code to ar archive files, so as to make C functions throughout your entire machine are found by clicking this link.

4 Deliverables

4.1 Deliverable for Week of April 7, 2019

This week's deliverable can be found by clicking this link for tscd and this link for sha256.

4.2 Deliverable for Week of March 24, 2019

This week's deliverable can be found by clicking this link.

4.3 Deliverable for Week of March 31, 2019

The deliverable will be found by clicking this link.

4.4 Deliverable for Week of April 14, 2019

This week's deliverable can be found by clicking this link.

5 HyperLinks In This Book

All hyperlinks in this work are highlighted in sky blue text like this one. The hyperlink in the previous sentence brings you to the author's GitHub Repository "SSL-TLS", which contains progress made so far in implementing the SSL Library.

6 Reading Source Code

If you wish to view the source code currently made for the SSL/TLS implementation, please visit this hyperlinked URL.

7 Deliverable: Week of March 10, 2019

I completely rewrote the program http.c from scratch and have named it "myhttp.c". The source code file "myhttp.c" is found in the ch1 directory on the GitHub page hyperlinked in the URL mentioned above. This simple program downloads the html source code of a website. The program accepts only one URL for an HTTP-based website only. While rewriting the program, I noticed several features of the code were outdated. The following structs, which are blocks of computer memory storing smaller variables, were outdated: hostent, sockaddr_in, and sockaddr_in6. The function sprintf is also outdated and should be replaced with snprintf_s.

7.1 Outdated Structs

All structs mentioned above have been replaced with the sole struct addrinfo, where the coder only needs to use the getaddrinfo and freeaddrinfo functions to retrieve network binary address data from the internet.

7.2 Security Flaws in Code

Several security flaws in the source code were found in the author's (Joshua Davies) source code for http.c.

For instance:

```
*proxy_port = atoi( colon_sep + 1 );
```

atoi in the line of code above is an insecure function. If the string pointer sent as input into atoi does not actually represent a number, atoi will simply return 0. Even worse, atoi can generate an integer outside of the range of integers for int variables. It is better to assign a literal value to an int variable from a string by performing:

```
*proxy_port = (int)strtol( colon_sep + 1 );
```

Moreover, sprintf is used. This function can lead to a buffer overflow. This is when more data is appended to an array than the maximum capacity permits. Buffer overflows are the fundamental strategy all malicious software use to exploit computer systems.

The following code illustrates how dangerous sprintf is:

```
sprintf( get_command, "GET http://%s/%s HTTP/1.1\r\n", host, path );
```

The problem with the above code has to do wit the two "%s". There is no limit to how long host and path can be. A far better function would be:

```
snprintf( get_command, GET_COMMAND_LEN, "GET http://%s/%s HTTP/1.1\r\n",
host, path );
```

In snprintf, only the first GET_COMMAND_LEN characters will be copied to get_command starting from the beginning of the get_command array.

Still, there are several security flaws even in this function.

Starting from C11, a brand new function named snprintf_s provides extra protection against buffer overflows. snprintf_s protects against:

- the conversion specifier %n is present in format
- any of the arguments corresponding to %s is a null pointer
- format or buffer is a null pointer
- bufsz is zero or greater than RSIZE_MAX
- encoding errors occur in any of string and character conversion specifiers
- (for sprintf_s only), the string to be stored in buffer (including the trailing null) would exceeds bufsz

For now, I have only replaced sprintf with snprintf. There is a problem with using snprintf. C11 is not anywhere near as portable as the previous standards. Even GNU11, the standardization of the GNU C compiler that is supposed to add support for C11 features, does not have support for snprintf_s. I will have to implement all "_s" functions from the C11 standardization that I need from scratch and append them to the libc.a standard c library archive.

7.3 Modifying The Standard C Library

Professional modifiers of the Standard C Library should ideally have reverse engineering skills. A coder may be able to see what functions, macros, and global variables, are in a C source code library by reading ".h" header files. But, they cannot see the original source code implementation. Instead, when a person downloads the Standard C Library package, all the functions are preinstalled as object machine code instead (".o" files). So to officially determine how a standard function (e.g. printf from stdio.h) truly works on their machine, the coder must read the disassembled machine code in libc.a, which is the archive of Standard C Library Functions.

I became interested in how coders modify The Standard C Library when I realized that many compilers and their library functions did not support many of the secure functions that secure coders like the CERT C Secure Coding Committee recommends.

In Joshua Davies' work "Implementing SSL and TLS", Davies uses four standard library insecure functions in http.c: atoi, strlen, and sprintf. I have also added the use of strncat in my implementation of myhttp.c. It is better to use strnlen_s, snprintf_s, (int)strtol, and strncat_s as replacements. Yet, most industry standard compilers do not support these functions except for strtol.

By now, I have already added the object code of my own implementations of the strnlen_s and strncat_s to the libcygwin.a ar archive. But I did promise the source code I make will be portable at least on Windows (Cygwin), macOS, Linux, and BSD systems. Yet, the libcygwin ar archive can only exist on Windows systems.

To solve this problem, I am currently developing a script named "c11.sh" that will automatically detect which operating system is being used and accordingly modify the necessary header files and add the object code for the standard library functions I have implemented. The major benefit in appending object to standard library functions that the gcc compiler automatically searches at runtime gurantees that the brand new functions defined in the object code can be used by and C source file **literally anywhere** in the machine.

7.3.1 Instructions To Modify Header Files For Standard Library Functions

To begin, a coder must first determine what ISO C standard their GNU C library supports. The following terminal command will print the line that defines the C standard the GCC compiler supports:

```
gcc -dM -E - < /dev/null | grep -i STDC_VERSION
```

By now, all GCC implementations claim they support a standard that is greater than or equal to 2011. Bear in mind this implies the GNU C Standard is used and NOT the ISO C Standard. The GNU C standard incorporates libraries like glibc that are not native to the ISO C standard. This is why many of the secure functions that are available in the ISO C Standard are not present on GNU C based systems like Linux or Cygwin.

Next, the coder must determine if the function they are looking for is in libc.a archive. The following command will highlight which ".o" object files contain the function that the coder is looking for:

```
nm -o libc.a | grep -w function-being-looked-for-here
```

It is actually very easy for a coder to append functions to a Standard C Library Archive like libc.a.

The first improvements to The Standard C Library were simple edits to the stdio.h, stdint.h, and stddef.h libraries.

I had to define two standardized C11 variable types that are not native to the GNU: errno_t and rsize_t.

I first added the definitions to the /usr/include/stdio.h header file to make it very easy to access these data types. I then added these data type definitions to the /usr/include/stddef.h header file since C11 standardization demands this. I finally added the definition for errno_t to the /usr/include/errno.h file since the ISO C11 standardization requires this.

7.3.2 Deciding Which Standard ar Archive To Modify

To decide this, perform the following command in your command line:

```
gcc -v file_name.c -o file_name.o -Wl,--verbose
```

The above command will force the compiler to display every pathway and library it searched in its attempt to correctly link all standard library object code to the raw object code of your source C file.

Generally, a library will be represented in the output of the above command in the following format:

-l{archive_name} where archive_name comes from archive_name.a

Multiple libraries represented as above will be displayed somewhere in the output. For instance, when the above command was used on the test.c file for testing the accuracy of my implementation of strnlen_s.c, the libraries searched was found in one of the lines as follows:

```
-lgcc_s -lgcc -lcygwin -ladvapi32 -lshell32 -luser32 -lkernel32 -lgcc_s -lgcc
```

7.3.3 Appending Object Code to Static Library Archives

When C source code files are compiled, they are normally translated into machine code, also known as object code. However, not all functions in a C source code file are necessarily defined directly in the source code file. For instance, the printf function is perhaps the most widely used function in the C programming language. But, of course, it is almost never directly defined in the C source code file. The compiler looks for the object code defining the printf function elsewhere. Wherever that object code is, it is packaged into an archive of object code known as an ar archive.

C source code files may be compiled without the compiler linking the object code of functions that are not defined in the source code themselves to the final object code of the source code. To do this:

```
gcc -c file_name.c
```

The -o flag cannot be used along with the -o flag since linking of external object code to the file_name.c is not supposed to happen. When the -c flag is used on file_name.c, the compiler will automatically translate whatever source code is in file_name.c to object code all by itself. By default, gcc will name the output file file_name.o.

Archive ar files are really just a collection of object code files. To create ar archive files:

```
ar -c archive_name.a
```

To append object files to an ar archive:

```
ar -r archive_name.a file1.o file2.o file3.o ...
```

The -r flag will not append the aforementioned files but will also replace preexisting object code files in the ar archive with the version in the specified in the filename arguments field.

To literally append an object code file, even if that means adding a duplicate version of the object code file in the ar archive:

```
ar -q archive_name.a file1.o file2.o file3.o ...
```

The -q and -r flags textbfcannot be used at the same time.

The C source code files you wish to append to an ar archive library should never have a main function in them. This will confuse the compiler when it is trying to process logical flow of execution of programs.

8 The Importance of Quines in Cybersecurity

To avoid detection by an antivirus system, malware should ideally utilize the least amount of memory from both the RAM and hard drive space or at least try to delete its memory footprint throughout its existence in the infected machine.

The most notable point in a malware's lifespan where this is critical is when malware copies itself, independently of any other program, and often injects its own source code to other programs in the infected host's machine.

But how does a program copy and then prints its own source code into another file without the assistance of any other program or user?

The answer is that the virus must store its own source code in an object in its own programming, often in a variable that stores a stream of ASCII letters (called a 'string' object). Programs that copy themselves without accepting any input whatsoever are known as quines.

The following hyperlink opens a page to a simple quine program that directly prints its own source into the console.

This honestly was the shortest quine the author could make using string literals(ASCII letters wrapped in "" marks).

However, bright hackers like Kenneth Lane Thompson, the man who implemented UNIX, invented a much simpler method to generate a quine program. The hacker simply makes a separate program that copies the FILE contents into a massive char array and prints the char array, complete with the hexadecimal representation of each char in the target program into the original program, turning the original program into a quine program.

A simple example of this program is found here demonstrate_ken_quine.c.

At first glance the program looks dauntingly tedious, but in reality a hacker can easily generate a program that automatically inserts the char array (q) at the beginning of the program for you. Originally, the quine program above looked like this: demonstrate_ken.c.

Then, the ken_quine_generator.c program accepted demonstrate_ken.c as an stdin argument and inserted the char array q at the beginning of the new file that was stored in demonstrate_ken_quine.c

demonstrate_ken_quine.c gives a proper example of how malware copy can easily copy themselves.

For example to test demonstrate_ken_quine.c, the following command line arguments were performed:

```
gcc demonstrate_ken_quine.c -o dkq.o
```

```
./dkq.o > quine2.c
```

```
gcc quine2.c -o quine2.o

./quine2.o > quine3.c

gcc quine3.c -o quine3.o

./quine3.o > quine4.c

gcc quine4.c -o quine4.o

./quine4.o > quine5.c

gcc quine5.c -o quine6.o

...
```

The tests can go on forever. This process demonstrates how a self-reproducing program like malware can copy itself multiple times. The only change the malware needs to make: insert the pathway before the target file and the quine program will generate the target file at that location.

Worse, the quine can instead append itself to a program that is frequently ran daily on the system's machine, propagating the spread of the virus further.

But not only viruses can use this tactic. So can anti-malware, by injecting segments of destructive self-reproducing code into the virus's own body, effectively using their own strategies against them.

9 How Antivirus Software Detects Simple Viruses

The simplest of viruses simply copy themselves into other directories or inject themselves into other files. They make no attempt to modify their own programming each time they mutate. Such viruses are very easy for antivirus software to spot.

9.1 Case Study: The Creeper Virus

The Creeper virus was the first computer virus in recorded computer history. It spread through the ARPANET. Every single time Creeper was executed, it would first print a message: "I'm the Creeper, catch me if you can!". It would then open a connection, copy itself to another machine, and then delete itself in the current machine. The virus was designed by Bob Thomas in 1971 to demonstrate that self replicating programs are indeed a possible–and they are.

9.2 Case Study: The First Computer Nematode: Reaper

Reaper is the first computer nematode. It was designed by security researchers to delete Creeper in infected machines.

Although Reaper was successful, antivirus researchers today are STRONGLY against the idea of using another computer virus to take down a preexisting virus. Even with the best intentions, nematodes have caused just as much damage as the preexisting virus. There are also concerns about privacy. The nematode assumes the user is fine with automatically manipulating file and RAM contents without asking first. A major attempt to remove an already famous internet worm outbreak, Blaster, was the Welchia nematode.

Welchia contaminated 75% of the intranet of US Navy Marine Corps intranet and left it useless for some time. It then infected the State Department network.

Although Welchia was a nematode designed to destroy Blaster, it often slowed down performance of the infected machine, even if the Blaster worm had already been removed from the computer. And once again, the Welchia worm did not ask the user's permission before executing. No one even knows who the writer of Welchia is thus classified as malware itself since it historically did far more harm than the good the writer intended.

9.3 AntiMalware Technique: Virus Signature Detection

Today's Antivirus companies advocate identifying a virus before it has a chance to do damage the machine–then delete it permanently.

The virus signature method is used for a virus that has the same hexadecimal byte pattern for every one of its copies–a weakness in its design against malware.

Antivirus companies must be careful about how long the hexadecimal byte pattern of each virus definition. Make the pattern too long, and the typical virus scan will take too long. Make it too short and the scan will accidentially declare an innocent file as a virus (known as a false positive amongst computer virus researchers).

A good size would be an 8 byte hexadecimal pattern. A single byte must be represented by no more than a two hexadecimal digit pattern. So there will be 16 hexadecimal digits total in a virus signature model that identifies viruses. The byte pattern should ideally be taken from the end of the file, since the beginnings of files may contain information on the file type. So a good idea would be to record the last 64 bits of the virus.

Viruses are rarely written in binary or hexadecimal code. They are now usually written in a C-like language or Intel x86-64 assembly language. To find the last 16 hexadecimal digits of a virus source code, a hex dump program must be used. A famous, simple program for this is xxd.

I have made my own simplified implementation of this called myxxd.

Hex dump programs simply translate every character in a file into its hexadecimal two-digit representation.

As a simple example of how this is done with myxxd, consider the file letter_T.txt. All that is contained in this file is the letter T(Hexadecimal value: 0x54) immediately followed by a newline character (has the hexadecimal value 0x0a):

```
myxxd letter_T.txt
```

```
00000000:      540a
```

myxxd uses a zero-based index system to count how many hexadecimal digits. To demonstrate a zero-based index system. Let us pretend we have a list of the numbers 1-5, inclusive. Below these numbers is their index in a zero-based index system.

```
1   2   3   4   5
0   1   2   3   4
```

The number 1 in the example above has an index of 0, not 1. That is how all zero-based index systems start counting with. Notice the last digit in the sequence thus has an index of 4, not 5.

Therefore "00000000:" represents the 0-indexed hexadecimal digit: 54(the hexadecimal representation of 'T'). The hexadecimal digit "0a" at the end of the file represents the character for a newline.

myxxd will display hexadecimals for all characters in a file in a neat table format until it reads the hexadecimal for EOF (0xff).

For instance, the following shows the last two lines outputted by myxxd is of a hex dump of the program virus_signature.o (from virus_signature.c), a program that will be discussed next:

```
myxxd virus_signature.o
[Lines omitted]
```

```
...
00027500:     656e   645f   5f00   5f5f   696d   705f   6673   6565
00027510:     6b00
```

9.4 Major Upgrade To myxxd: tscd

A major upgrade has been done on myxxd, now named tscd. The program now color codes the output when printed to stdout. But when the output is written to a separate output file, no color codes are displayed in the file, as this is impossible.

A man-page style documentation will later be provided on the program.

For now, the following describes tscd and how it is to be used:

NAME

tscd

SYNOPSIS

tscd [options] [infile [outfile]]

DESCRIPTION

tscd can create a hexadecimal, decimal, binary, or octal dump of any file. The numerical values will be displayed in a table.To the rightmost of each row will be the ASCII characters corresponding to the ASCII codes in the table.

OPTIONS

-b Binary dump specified. Each character in file will be translated into its binary number form and displayed in the table.

-c Specify number of columns per row in ASCII code table.

-d Decimal Dump specified. Each character in file will be translated into its binary number form and displayed in the table.

-o Octal Dump specified. Each character in file will be translated into its octal number form and displayed in the table.

EXAMPLES

Print simple hexdump of a file to stdout

```
% tscd swiss.txt
```

```
00000000:  49 20 6c 69 6b 65 20 63 68 65 65 73 65 73 2e 0a I like cheeses.·
```

Print simple hexdump of a file where 20 ASCII hexadecimal codes are printed per row to stdout

```
% tscd -c 20 swiss_cheese.c
```

```
00000000:  49 20 6c 69 6b 65 20 63 6f 6f 6b 69 65 73 2e 0a 0a 49 20 6c I like cookies
00000014:  69 6b 65 20 63 6f 6f 6b 69 65 73 2e 0a 0a 49 20 6c 69 6b 65 ike cookies.··
```

```
00000028:  20 63 6f 6f 6b 69 65 73 2e 0a 0a 49 20 6c 69 6b 65 20 63 6f  cookies.··I l
0000003c:  6f 6b 69 65 73 2e 0a 0a 49 20 6c 69 6b 65 20 63 6f 6f 6b 69 okies.··I like
00000050:  65 73 2e 0a 0a 49 20 6c 69 6b 65 20 63 6f 6f 6b 69 65 73 2e es.··I like co
00000064:  0a
```

Print binary dump of a file named tscd4.c to a separate file named target.c

```
% tscd -b swiss.txt swiss_binary_dump.txt
```

The last sixteen hexadecimal digits in these two lines is displayed using the virus_signature.o program:

```
./virus_signature.o virus_signature.o
705f667365656b00
```

virus_signature.o simply displays the last sixteen hexadecimal digits in a file. In this case, its own. Today, modern antivirus software stores hexadecimal strings like these in a database file on the user's machine and searches for the presence of each hexadecimal string in the files the antivirus scans. If there is a pattern match, the antivirus declares the associated virus has been found in that file.

9.5 Modern Virus Signature Database Strategies

Previously, it was mentioned that the last bytes of a program, which does not mutate when it self-replicates, can be used to quickly identify the program and then remove it. But antivirus researchers have learned from experience no matter how or where the bytes of such a file are extracted from, the virus signature program may still accidentially declare an innocent file to be infectious simply because it has the same signature as the real, corresponding malware. This has become such a big problem in the past that antivirus researchers adopted a checksum technique to ensure that false positives do not happen when investigating a suspicious file.

9.5.1 The SHA-256 Algorithm Secured with HMAC

At the time of this writing, the best hashing algorithm to use for such a purpose is SHA-256. This hashing algorithm provides the best balance between being resistant to brute-force attacks (not counting rainbow table attacks) and being fast enough to satisfy consumer demands. All other hashing options are either too easy to crack or are too slow for the marketing world to adopt.

Hash Algorithms are NOT the same thing as encryption algorithms. Encryption algorithms allow us to reverse the encrypted message back into its original form with a decryption key. But strings generated from hash algorithms like SHA-256 are impossible to reverse back to the original input. Regardless of the size and characters of the input, the hash algorithm will always return a fixed-size irreversible string of random ASCII characters called a "hash" string. For the sake of The Virus Database, the SHA-256 will be used exactly as it was intended to be use: to verify that the data contents of a source file indeed match those of a known target file.

There are two types of target files that would be of interest to an antivirus developer:

1. Files whose contents are expected to remain constant. Using the SHA-256 hash on such files is a clever way of ensuring that the data contents of these files indeed remain constant. And if they do not, then the antivirus program then performs scans and heuristic analysis to determine if the file contents have been tampered by malware. This is why security experts recommend that each file or program we download is first verified using an advanced hash like these before using them.

2. Malware whose program contents always remain constant or whose certain program contents remain constant. There are only few types of malware whose soure code contents do not completely change throughout self-replications. One of these is the Trojan Dropper. Once Trojan Droppers infect a computer, they merely release the malware payload that they either carry in their own source code or download from a malware-hosting website. But Trojan Droppers, although very dangerous, do not themselves make copies of themselves. So a virus signature can be used to identify them and prevent the user from accessing the website that hosts them. In general, using an SHA-256 file integrity checksum method is only effective on a file where all bytes of data stored in the file remain the same, no byte is expected to be removed, and no byte is expected to be added.

9.6 How To Securely Update Virus Database Signatures

Ensuring a good checksum is used to identify virus files and to keep track of the integrity of user files is only half the battle.

Often, the virus signature database on a person's computer must be updated to ensure the machine is as safe as possible. However, as the The Antivirus Hacker's Handbook points out, hackers can ironically hack into a the server that controls virus databases– and then install malware directly from the server. There are three mechanisms Antivirus companies are suppose to use, but Antivirus companies do not always follow them:

1. Use HTTPS exclusively to download signatures.

2. Catalog files: Ideally, a complete list of downloadable files and remote relative URIs to download from are located in a set of catalog files. These catalog files may contain information about supported platforms and different product versions.

3. This is the part where most Antivirus companies fall short on: The Antivirus software verifies that each and every file downloaded is the true version the Antivirus company intended to send. At the time of this writing, the best method to verify that the virus database update is legitimate is to use an SHA-256 hash checksum against each file followed by checking the downloaded file against an expected RSA-2048 digital signature.

Why use the two methods hand-in-hand? The SHA-256 hash is an excellent method to determine the file contents were downloaded exactly as expected. The SHA-256 in this case is known as a digital certificate. But today, hackers are becoming notoriously more capable of brute-force generating hashes that match the expected hash. To prevent this, the resulting SHA-256 hash string of the input file is then checked against an RSA-2048 digital signature to ensure the source is authentic.

9.7 Introduction to xMorphic Viruses

In response to the virus signature method, virus writers began designing viruses that modify their own source code for each and every copy they made of themselves.

The first generation of these viruses were Oligomorphic viruses.

9.8 Oligomorphic Viruses

Oligomorphic viruses are viruses that modify their own decryption algorithm in a fixed number of ways. Each new copy of the virus randomly chose one of these decryption algorithms. However, since there was a fixed number of different decryption algorithms (hereafter called "decryptors"), virus signatures could still be made for them.

9.9 Polymorphic Viruses

In response, virus writers began designing viruses that changed every byte in each of its copies' decryptor programming. These viruses were called polymorphic viruses. Since polymorphic viruses would generate an infinite number of different decryptors if left to run forever, the virus signature method was insufficient. There was no way to store an infinite number of virus signatures–and that was just for one polymorphic virus!

The best way to detect polymorphic viruses is the following:

The virus is placed into a program that emulates the CPU and/or the machine's operating system. The emulator waits until the polymorphic virus finishes decrypting itself. When encrypted, there is no feasible way to distinguish the virus from meaningless garbage. But the source code of the virus remains identical to that of other copies after decryption completes. It is often at this point that the antivirus captures the virus source code and scans the naked source for a virus signature.

9.10 Metamorphic Viruses

Attempting to analyze the decrypted source code of a virus is least effective against Metamorphic viruses. Metamorphic viruses. Unlike polymorphic viruses, metamorphic viruses literally change their entire source code with each new copy it makes byte per byte. There is no way practical way to detect a pattern between any two virus copies.

The first Metamorphic virus was the Win95/Regswap virus, discovered in 1998.

The most influential Metamorphic Virus goes by many names. The famous nicknames for it are Simile and MetaPHOR. What is most alarming about this virus is that it came complete with a Metamorphic Engine: (Metamorphic Permutating High-Obfuscating Reassembler). The virus could grow or shrink in size with each new copy it made. Although the virus itself did not do anything harmful once it infected the machine, the true danger was the metamorphic engine.

How to Detect Metamorphic Viruses:

It is a waste of time to try to identify a Metamorphic virus by trying to find common patterns in its source code. After all, every byte of the source code changes with each new copy of the virus made. The true universal pattern shared by all metamorphic viruses is not in its source code. It is in its behavior.

9.11 The Birth of Heuristic Analysis

All types of viruses discussed so far have one thing in common, no matter how many copies the virus makes from itself, they all do the same exact thing. Today, heuristic analysis of malware is still in its infancy. But it follows the classical strategy antivirus software uses against polymorphic viruses.

According to a whitepaper published by ESET, one method to detect such malware is to use a rule system. The antivirus first begins analyzing a program after a trigger is alarm. One such trigger is when a computer program scans hexadecimal or binary code directly. This is exactly what Avast Antivirus Free does.

I installed the Avast Free Antivirus since it was rated as the best Linux Free Antivirus freeware.

While designing the ken_quine_generator.c program, copies of the program had to compiled and themselves executed successively to ensure ken_quine_generator.c actually was producing quines. But for each execution, Avast Free Antivirus killed the execution of the program. It reported the program was conducting "suspicious behavior" and the computer seemed to freeze. Shortly after, Avast said everything was fine and left the user to execute the program. Each time after the newest copy of the quine was compiled, Avast would repeat this process.

Here is what Avast Antivirus was doing. It detected ken_quine_generator.c was scanning its own hexadecimal source code. This was necessary to allow ken_quine_generator.c to replicate itself. Avast then transferred a copy of the program into an x86-64 CPU emulator of the Windows Operating System, and observed its behavior. Since ken_quine_generator.c merely made only one copy of a program, the Avast Antivirus program scored the program as a false alarm for a virus, and admitted everything was ok. The problem was Avast did this everytime the program was compiled, or translated to binary code.

Although Avast is supposed to boast the most advanced heuristic analysis engine today, there is obviously much work that needs to be done to avoid false positives (antivirus accidentally thinks innocent file is a virus) and/or false negatives (antivirus accidentally thinks a virus program is not a virus).

9.12 Future Problems with Heuristic Analysis

Right now, a virus writer is trying to develop a virus that not only is metamorphic, but that its behavior completely changes for each copy it makes. From hereon, such viruses are coined metaheuristical viruses. When such a virus hits the internet world, it will wreak havoc. Not only is it resistant to all known techniques where the source code of the virus is analyzed for a signature(called "static analysis" from hereon) but it is also resistant to all forms of behavioral and heuristical analysis. Both the source code and behavior of the virus change each time the virus replicates. Even if the antivirus is perfect enough to detect the intrusive behavior of each copy of the virus, the virus finishes replicating into a new form in time . A cheap prototype of such a virus already exists. They are called "Trojan Droppers" by antivirus researchers. The main problem with droppers is that they themselves are not a virus, so the antivirus will not detect them unless it monitors the internet traffic of the user's machine and recognizes that the computer is downloading malware from websites that are known to host malware. This requires the antivirus software to keep a database of the IPv4 addresses of malware-hosting websites and determining if such traffic is taking place. Once the antivirus realizes the Trojan Dropper is doing this, the antivirus kills execution of the program, deletes it, and if it has not already, delete any viruses that have infected the computer.

9.13 Interview with Malware Analysts

One of the best ways to ensure a career will be the one you are most happy with is by asking experienced professionals important career questions before committing to the role.

This week, I have asked Vinh T. Nguyen for career advice. Vinh T. Nguyen works with Dr. Akbar Namin at Texas Tech, who is leading a team of malware researchers. After the interview, Vinh T. Nguyen reminded me that I am free to talk with any member of his team for further information.

The following is a transcript of the advice he gave me:

Malware Analyst Interview

Vinh T Nguyen

If you had to live your life over, you become a malware researcher (working in an academic setting) again?

[Forgot to ask] But, wait, he is still overseeing a team of malware researchers (Recently published paper: MalViz: An Interactive Visualization Tool for Tracing Malware Akbar Namin, Tommy.Dang) –That actually leans towards yes.

Is the malware analyst profession overcrowded?

NO Undercrowded

If I worked on antimalware and related reverse engineering projects on my free time for two-four years, would it be difficult for me to get a job? What kind of job would I have to take first.

NO, again undercrowded and as long as you prove you have first-hand experience with malware analysis, you are fine. (That was how he recruited). What kind of job would I have to take first (None, necessarily). But a developer job close to or at C would not hurt. By the way, he recommended to learn machine learning so the antivirus software would recognized the behavior of certain classes of malware. This would be useful against malware that is hard to detect against a signature based system (like MetaPHOR).

What are the advantages and disadvantages of malware analysis as a profession.

No disadvantage, only positives it is a rapidly changing and exciting field, where there is an ongoing battle between antivirus designers and malware designers.

What courses at Texas Tech would benefit me?

Akbar Namin's course – Digital Forensics (NOT TAKING)

Would it be a good idea to engineer my own antivirus software?

Not exactly, better to focus on making a virus signature detection system instead. Once virus found, it is recursively removed from all directories.

What certifications may help (if any)?

CISSP

What major do you recommend for a malware analyst?

Computer Science

9.14 SHA-256

So far a complete rough draft of the implementation of SHA-256 has been completed.

9.15 Final Presentation

The final presentation will demonstrate a simple virus signature detection system that accurately identifies the EICAR Test FIle and removes the threat. The virus signature system stores the SHA-256 of the EICAR Test File in its database. When asked to scan the computer, it will recursively scan all files in all directories, starting from root, until it finds the EICAR file. Generally, the scanner accepts the entire text file as input and outputs a hash, which it compares to the SHA-256 hash of the EICAR test file.

Before testing with the EICAR test file, the virus scanner will simply be programmed to detect a simple textfile.

The textfile will start the same string the real EICAR test file is supposed to start with:

```
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

The EICAR organization provides the following official rules on how the test file may be formatted:

The first 68 characters is the known string. It may be optionally appended by any combination of whitespace characters with the total file length not exceeding 128 characters.

The fake EICAR textfile will simply be placed in prespecified directories and it will be the scanner's duty to recursively remove them.

The implementation of the rm command will be used as a reference in designing the scanner's capabilities.

Due to the simplicity of the scanner, it is only designed to replicate the process of deleting a virus that merely self-replicates, without any oligomorphic, polymorphic, or metamorphic properties

9.16 Detecting UTF-8 Support in Files

Some viruses hide by hiding in UTF-8. Currently, a program is in development that will detect whether or not a program is written in UTF-8. Currently, there is an issue that the program fails to print foreign characters such as Arabic characters.