Computer Security and the Internet: Tools and Jewels

Chapter 9: Malicious Software

P.C. van Oorschot

August 28, 2018

Comments, corrections, and suggestions for improvements are welcome and appreciated.
Please send by email to: paulv@scs.carleton.ca

PRIVATE COPY—NOT FOR PUBLIC DISTRIBUTION

Chapter 9

Malicious Software

This chapter discusses various types of *malicious software* or *malware*, including computer viruses and worms, botnets, rootkits, and other variations. There are many ways of naming and categorizing malware—we present terms and classifications useful to discuss and understand it. We consider the motives of those responsible for malware, why preventing malware is difficult, and why detecting and removing it can be hard.

Malware often exploits specific software vulnerabilities to gain a foothold on victim machines. Even when vulnerabilities are patched, or software updates eliminate entire classes of previous vulnerabilities, it remains worthwhile to understand past exploits—for general knowledge of recurring errors and common patterns that make exploits possible, to help us avoid repeating past mistakes in new designs. Thus exploit methods mentioned in this, and other chapters, are presented even in some cases where specific details exploited have been repaired in mainstream software releases. The lessons remain valuable, reinforcing good design principles for security.

9.1 Malware: unauthorized and malicious

Generally speaking, *malware* is any software which both

1. is unauthorized, in the sense of running without the informed consent of users; and
2. has intended functionality contrary to the best interests of users or the system.

Malware has some malicious goal. Almost always, a responsible user fully aware of its intended functionality would not run it. Our definition rules out users who run programs which they know will cause damage—we apologize for leaving such bad actors stranded, but that seems easier than figuring out what informed consent means for bad actors.

Exercise (Malware definition). If unauthorized software damages only machines other than the one it is installed on, is it malware by our definition? What about software inadvertently causing damage due to design or implementation errors?

QUESTIONS REGARDING MALWARE. We begin with some questions to foreshadow, and to organize a more detailed exploration later in this chapter.

1. *How does malware get onto computer devices?* One way is through web sites—both compromised legitimate sites, and malicious sites attracting traffic by links in *phishing* emails, search engine results, and web page ads.[1] Downloaded executables which users intentionally seek may be repackaged to include bundled malware; or users may be tricked to install executables which are either pure malware, or contain hidden functionality; or the site visit may result in software installation without a user's knowledge (through exploiting vulnerabilities in browsers or in applications they invoke to process content).[2] Worms spread malware by exploiting vulnerabilities in network communications services, enabled by modern devices with near-constant network connectivity. Computer viruses spread by various means including malicious email attachments. Malware may also be embedded in development repository source code—legitimate developers who write code may become inside attackers (*insiders*), or repositories may be compromised by outsiders. Computer firmware and hardware may even be malicious—depending on how firmware is provided and updated, and controls within the hardware supply chain.

2. *What makes malware hard to detect?* While easy in some cases, in general detection is hard for multiple reasons. What is malware depends on context, not functionality alone—e.g., SSH server software is not generally viewed as malware, but this changes if it is installed by an attacker for covert access to a system. Indeed, a short theoretical result (see Section 9.2) directly shows that it is undecidable what malware is. Personal viewpoints may also differ—is a useful program which also display ads to generate revenue malware? In this sense, some forms of malware are more aggressive than others. Malware is also specifically designed to be hard to detect and even to reverse engineer (e.g., see Section 9.3 for stealthy malware).

3. *How can installation of malware be prevented?* From the above, if we can't decide what malware is, it seems unreasonable to expect any program to prevent all forms of it. Restricting what software users can install on their machines reduces risks, but is both inconvenient and unpopular. Better user education is often suggested, and useful to some degree, but also difficult, costly, never-ending, and insufficient against many malware tactics including persuasive *social engineering*. Malware risks can be reduced by *code-signing* architectures which aim to verify, before installing or running it, that executable content originates from known sources. *Anti-virus/anti-malware* tools and *intrusion detection/prevention systems*[3] are industry-driven partial solutions. Some tools remove or filter out specific instances of malware, although in severe cases a host machine's entire software base may need to be re-installed with a clean base operating system and all applications—with data files not recoverable from backup storage lost. This does not make for a good day in the office!

[1] *Pharming* attacks also disrupt IP address resolution to misdirect browsers; see the Web Security chapter.
[2] Section 9.4 discusses *drive-by downloads* and malicious *active content* in web pages.
[3] These are discussed in the Monitoring and IDS chapter.

SOFTWARE CHURN, EASE OF INSTALLATION HELP MALWARE. In early days of computer systems, end-users were not directly involved in software installation or upgrades. Neither network-downloaded software nor wireless software update was available. Computers came with pre-installed software from the device manufacturer. Computer experts or information technology (IT) staff would update or install additional operating system or application software from master copies on local storage media (e.g., CD ROM or floppy disks, long before USB flash drives). Software upgrades were frustratingly slow. Today's ease of deploying and updating software on computing devices has greatly facilitated rapid evolution and progress in software systems—and the deployment of malware. Allowing end-users to easily authorize, install and update almost any software on their devices opened new avenues for malicious software to gain a foothold, including users tricked into "voluntarily" installing software which misrepresents its true functionality (e.g., *ransomware*), or has hidden functionality (e.g., *Trojan horse* software). Users also have few reliable signals from which to identify the web site a download arrives from, or whether even a properly identified site is trustworthy. Malware issues are increased by the high "churn rate" of software on network infrastructure including servers, and on end-user mobile devices, personal computers and workstations.

9.2 Viruses and Worms

In this section we discuss the first types of malware to gain notoriety: computer *viruses* and *worms*. Though differing in some aspects, they shared the distinguishing feature of *propagating*, i.e., employing clever means to cause their number of instances to grow, and spread across machines.

Closely following the early academic research of Fred Cohen, we define a computer *virus* as: *a program that can infect other programs or files by modifying them to include a possibly evolved copy of itself*. A typical virus replicates, spreading to further programs or files on the same machine; and also across machines aided by some form of human action, e.g., inserting into a device a USB flash drive (or floppy disk in the past), or clicking on an email attachment that turns out to be some form of executable file. A virus embeds itself into a *host* program or file that contains some form of executable content, and arranges affairs such that its own code runs when the host is processed or itself runs. Viruses typically check if a file is already infected; infecting only new files is more effective.

GENERIC STRUCTURE OF A VIRUS. The following pseudo-code gives a high-level overview of a computer virus.

```
loop
   remain_dormant_until_host_program_runs();
   propagate_with_user_help();   % strange type of process fork
   if trigger_condition_true() then
       run_payload();
endloop
```

The pseudo-code indicates the generic structure of a virus, with four stages:

1. dormancy;
2. propagation—how the malware spreads;
3. trigger condition—controls when to execute the payload; and
4. *payload*—a fancy name for the functionality malware delivers (other than propagating). Payload actions range from relatively benign (an image walking across a screen) to severe (erasing files, taking software actions which damage hardware).

GENERIC STRUCTURE OF A WORM. For comparison to viruses above, the following pseudo-code gives a high-level overview of a computer *worm*.

```
loop
   propagate_over_network();  % stranger type of process fork
   if trigger_condition_true() then
       run_payload();
endloop
```

WORMS DIFFER FROM VIRUSES. They differ in three main ways.

1. Worms propagate automatically and continuously, without user interaction.
2. Worms exploit software vulnerabilities, e.g., buffer overflows, while viruses tend to abuse software features or use social engineering.[4]
3. Worms spread across machines over networks, leveraging network protocols and network daemons rather than infecting host programs.

As a result of this last property, worms are also called *network worms* or sometimes *network viruses*. Note from these properties that worms, having no dormant stage, tend to spread more quickly, and are more likely to overload network communications channel capacity, causing a form of *denial-of-service*, even when that is not be their end-goal.

EMAIL-BASED MALWARE. Email-based malware combining virus and worm properties is called an *email virus*, *email worm*, or *mass-mailing worm-virus*. It spreads through mail-related file infection, email attachments, and/or features of email infrastructure and clients (often enabled-by-default). It typically requires a user action (e.g., opening an email client or reading a message), and possibly social engineering. A common tactic is to use the mail client's address book as a source of next-targets—and since email may have a long list of intended recipients, the spread is one-to-many in parallel.

MAGIC, MALWARE AND PRIVILEGES. There is nothing magical about viruses, worms, and other malware. They are simply software, with power and functionality as

[4] McIllroy states [19]: *"If you have a programmable computer with a file system inhabited by both programs and data, you can make viruses. It doesn't matter what hardware or operating system you are using."*

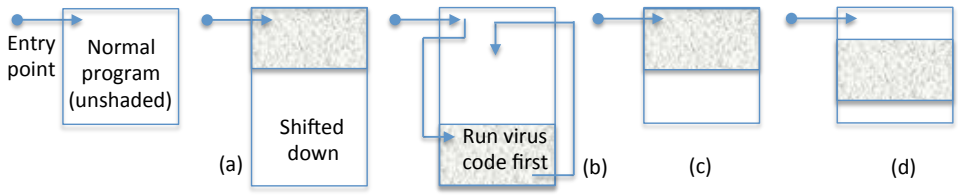

Figure 9.1: Virus strategies for code location. Virus code is shaded. (a) Shift and prepend. (b) Append. (c) Overwrite from top. (d) Overwrite at interior.

available to other software. On the other hand, the tremendous functionality of regular software may itself seem magical—and just as ordinary software can, malware can do extraordinarily complex things, especially if it runs with *root* or *admin* privileges.[5]

Exercise (Malware privileges). What determines the OS privilege level of malware?

PROGRAM FILE VIRUSES. Most viruses infect executable program files. How and where virus code is inserted (in the host file) varies. Strategies include (see Figure 9.1):

1. Shift and prepend. The virus code is inserted at the front after shifting the original file, which is arranged to execute after the virus code. This increases the file length.

2. Append virus code to end of host file. This is convenient in file formats where the program entry point `JUMP`s to a start-execution point within the file. This original jump target is changed to be the first line of the appended virus code. The virus code ends by jumping to the originally indicated start-execution point.

3. Overwrite the host file, starting from the top. The host program is destroyed (so it should not be critical to the OS's continuing operation). This increases the chances the virus is noticed, and complicates its removal (a removal tool will not have the original program file content available to restore).

4. Overwrite the host file, starting from some interior point (with luck, one that execution is expected to reach). As above, a negative side effect is damaging the original program. However an advantage is gained against virus detection tools which, as an optimization, take short-cuts such as scanning for viruses at the starts and ends of files—this strategy may evade such tools.

Other variations involve relocating parts of program files, copying into temporary files, and arranging control transfers. These have their own complications and advantages in different file formats, systems, and scenarios; the general ideas are similar. If the target program file is a binary executable, address adjustments may be required if code segments are shifted or relocated; these issues do not arise if the target is an OS *shell script*.

[5] Recall that in Unix-type systems, *root* or *superuser* is a user (i.e., account) having full permissions on all files and programs; *administrator* accounts in other systems have almost as many privileges.

Exercise (Shell script viruses). Aside from binary executables, programs with virus-like properties can be created using command shells and scripts. Explain, with examples, how Unix shell script viruses work. (Hint: see [19]. Using command shells and scripts, and environmental properties such as the search order for executable programs, virus-like programs can replicate without embedding themselves in other programs. An example is what are called *companion viruses*. Peter Szor's alternate definition for a computer virus is thus: *a program that recursively and explicitly copies a possibly evolved copy of itself.*)

BRAIN VIRUS (1986). The *Brain* virus, commonly cited as the first PC virus, is a *boot sector virus*.[6] Networks were less common; most viruses spread from an infected program on a floppy disk, to one or more programs on the PC in which the floppy was inserted, then to other PCs the floppy was later inserted into. On startup, an IBM PC would read, from read-only memory (ROM), code for its basic input/output system (BIOS). Next, early PCs started their loading process from a floppy if one was present. After the BIOS, the first code executed was read from a *boot sector*, which for a floppy was its first sector. Execution of boot sector code would result in further initialization and then loading of the OS into memory. Placing virus code in this boot sector resulted in its execution before the OS. Boot sector viruses over-write or replace-and-relocate the boot sector code, so that virus code runs first. The Brian virus occasionally destroyed the file allocation table (FAT) of infected floppies, causing loss of user files. It was not, however, particularly malicious, and though stealthy (by a hooked interrupt handler, attempted boot sector reads showed a saved original), the virus binary contained the note "Contact us for vaccination" and the phone number and Pakistani address of the two brothers who wrote it!

On later PCs, the boot sector was defined by code at a fixed location (the first sector on the hard disk) of the master boot record (MBR) or partition record. Code written into the MBR would be run—making that an attractive target to write virus code into.

CIH CHERNOBYL VIRUS (1998-2000). The *CIH* or *Chernobyl virus*, found first in Taiwan and affecting Windows 95/98/ME machines primarily in Asia, was very destructive (per-device) and costly (in numbers of devices damaged). It demonstrated that malware can cause hardware as well as software damage. It overwrites critical sectors of the hard disk including the partition map, crashing the OS; depending on the device's file allocation table (FAT) details, the drive must be reformatted with all data thereon lost. (I hope you have all been carefully backing up your data!) Worse yet, CIH attempts to write to the system BIOS firmware—and on some types of Flash ROM chip, the Flash write-enable sequence used by CIH succeeds. Victim machines then will not restart, needing their Flash BIOS chip reprogrammed or replaced. This is a malicious payload!

CIH was also called *Spacefiller*—unlike viruses which insert themselves at the top or tail of a host file (see above), it inserts into unused bytes within files (in file formats that pad up to block boundaries), and splits itself across such files as necessary—thus also defeating anti-virus programs which look for files whose length changes.

DATA FILE VIRUSES AND RELATED MALWARE. Simple text files (plain text without formatting) require no special processing to display. In contrast, modern data doc-

[6] Similar malware is now called a *bootkit*. As a lesson, malware that runs before the OS is hard to detect.

uments contain embedded scripts and mark-up instructions; "opening" them for display or viewing triggers associated applications to parse, interpret, template, and pre-process them with macros for desired formatting and rendering. In essence, the data document is "executed". Two types of problems follow. 1) Data documents may be used to exploit software vulnerabilities in the associated programs, resulting in malware on the host machine. 2) Such malware may spread to other files of the same file type through common templates and macro files; and to other machines by document sharing with other users.

Exercise (Macro viruses: Concept 1995, Melissa 1999). (a) Summarize the major technical details of *Concept virus*, the first "in-the-wild" *macro virus* infecting Microsoft Word documents. (b) Another macro virus that infected such documents was *Melissa*. Neither had malicious payload, but this second one gained widespread media attention as the first mass-mailing *email virus*. Spread by Outlook Express, it chose 50 email addresses from the host's address book as next-victim targets. Summarize its major technical details.

Exercise (Data file malware: PDF). Find two historical incidents involving malware in Adobe PDF (Portable Document Format) files, and summarize the technical details.

UNDECIDABILITY OF DETECTING A VIRUS. It turns out to be impossible for a single program to correctly detect all viruses. To prove this we assume the existence of such a program and show that this assumption results in a contradiction. Let V be a virus detector program which, given any program P, returns a {**true, false**} result $V(P)$ correctly answering "Is P a virus?". With V in hand, consider program instance P^*.

Program P^*: **if** V(P^*) **then** exit, **else** infect-a-new-target

Now what happens when V is run on P^*? The result depends on whether P^* is a virus.

- Case 1: V(P^*) is **true**. This implies P^* is declared a virus by V.
  Running the code in P^* above results is no action, so P^* is not actually a virus.
- Case 2: V(P^*) is **false**. This implies P^* is declared a non-virus by V.
  Here, the code in P^* itself will infect a new target, so P^* is a virus.

So such a virus detector V cannot exist—because it existence would result in a logical contradiction. (This is called *proof by contradiction*.)

WHAT THIS MEANS IN PRACTICE. This proof sketch may seem like a trick, but it holds true—as a valid theoretical result. Should we then give up trying to detect viruses in practice? No. Even if no program that can detect *all* viruses, the next question is whether useful programs can detect many, or even some, viruses. That answer is yes—and thus the security industry's history of anti-virus products. But as detection techniques improve, the individuals creating viruses continue to develop new techniques making detection more and more difficult—and we have what is often called an *arms race*.

Exercise (Virus detection methods). If no program exists that can detect all malicious software, then how do anti-virus products work? What kinds of malware do they detect, how do they do it, and how is malware written to try to escape detection? (Hint: see [22].)

ANTI-DETECTION STRATEGIES. A virus making no attempt to evade detection consists of ordinary machine instructions, i.e., static cleartext code as in normal programs.

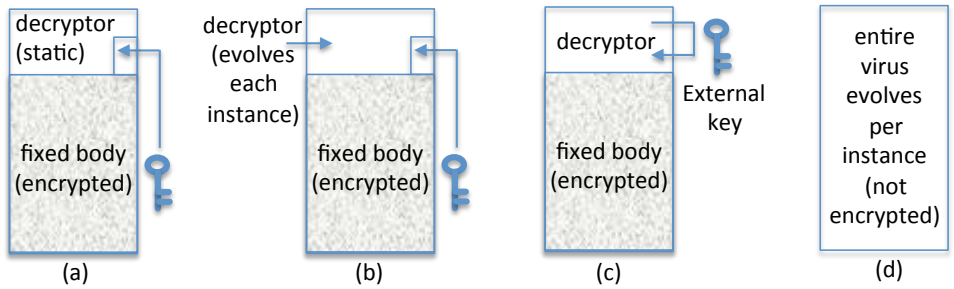

Figure 9.2: Virus anti-detection strategies. (a) Encrypted body. (b) Polymorphic. (c) External decryption key. (d) Metamorphic.

Advanced viruses and other malware attempt to conceal code content to complicate or avoid detection. The following techniques (Fig. 9.2) give one way to classify viruses.

1. *Encrypted body*. A simple form of hiding uses fixed mappings (e.g., *XOR* with a fixed string) or basic symmetric-key encryption using the same key across instances. Execution requires first decrypting the virus body, by a small *decryptor* portion which remains unmodified (and thus easily detected by a string-matching virus detector). To complicate detecting the modified body, the key, which is stored in the decryptor to allow decryption, can be changed on each new infection.

2. *Polymorphic virus*. These viruses have fixed bodies encrypted with per-instance keys as above, but change their decryptor portions across infections by using a *mutation engine*. A weak form stores a fixed pool of decryptors in the body, selecting one as the actual decryptor in a new infection. In strong forms, a mini-compiler creates new decryptor instances by combining functionally equivalent sets of machine instructions (yielding combinatorially large numbers of variations); for example, machine instructions to subtract a register from itself, and to xor with itself, produce the same result of a zero in the register. Techniques are related to those used for non-malicious code obfuscation and by optimizing compilers. After polymorphic virus decryption reveals its static body, that remains detectable by string-matching; virus detection tools thus pre-run executables in emulators to detect in this way.

3. *Encrypted with external decryption key*. To complicate manual analysis of an infected file that is captured, the decryption key is stored external to the virus itself. There are many possibilities, e.g., in another file on the same host machine or an external machine. The key could be generated on-the-fly from host-specific data. It could be retrieved from a networked device whose address is obtained through a level of indirection—such as a search engine query, or a domain name look-up with a frequently-changed name-address mapping.

4. *Metamorphic virus*. These use no encryption and thus have no decryptor portion. Instead, on a per-infection basis, the virus rewrites its own code, mutating both its body (infection and payload functionality) and the mutation engine itself. Elaborate metamorphic viruses have carried source code and enlisted compiler tools on host machines to aid their task.

The above means aim to hide virus code content. Other tactics aim to hide tell-tale signs of infection itself, such as changes to file system attributes (e.g., file bytelength and timestamp), the location or existence of code, and existence of running processes and the resources they consume. For hiding techniques used by rootkits, see Section 9.3.

THE 1988 INTERNET WORM. The *Morris worm* was the first widescale computer worm incident demonstrating the power of such attacks. It directly infected 10% of Internet devices of the day (only Sun 3 systems and VAX computers running some variants of BSD Unix), but worm-related traffic overloaded networks and caused system crashes through resource consumption—and thus widespread *denial-of-service*—despite having no malicious payload. Upon gaining a running process on a target machine, the initial base malware, like a "grappling hook", made network connections to download further components—not only binaries but also source code to be compiled on the target machine for local compatibility. It took steps to hide itself. Four software artifacts were exploited: a stack buffer overrun in fingerd (the Unix **finger** daemon which accepts network connections), a backdoor-like **debug** command in the sendmail program, a password-guessing attack on the file */etc/passwd* used along with **rexec**, and abuse of trusted remote logins through */etc/hosts.equiv*.

Exercise (Morris worm details). Summarize further technical details of the *Morris worm*, and the lessons learned. (Hint: see [28], [35] and [29, pp.19-23].)[7]

SPREADING TECHNIQUES USED BY WORMS. Worms differ from viruses by their means of spread. A worm's universe of possible next targets is the set of network devices reachable from it—traditionally the full IPv4 address space, and now IPv6. A simple spreading strategy is to select as next-target a random IPv4 address; a subset will be populated and vulnerable. The *Code Red II* worm (2001) used a *localized scanning* strategy, selecting a next-target IP address according to the following probabilities:

- 0.375: an address within its host machine's class B address space (/16 subnet);
- 0.5: an address within its host machine's class A network (/8 network);
- 0.125: an address chosen from the entire Internet.

The idea is that if topologically-nearby machines are similarly vulnerable, targeting local machines spreads malware faster once already inside a corporate network. This method, those used by the *Morris worm* (above), and other *topologically-aware scanning* strategies select next-target addresses by harvesting information on the current host machine,

[7] One recovery procedure was for all users on affected systems to change their passwords. That was 1988. When a 2016 Yahoo! compromise affected over a billion users, Yahoo! users were asked to do the same. Hmmm ... is this progress?

including: email address lists, peer-to-peer lists, URLs on disk, addresses in browser bookmark and favourite site lists. These are all expected to be populated addresses.

To following ideas have been suggested to improve the speed of worm spreading:

1. *hit-list scanning*. The time to infect all members of a vulnerable population is dominated by early stages before a critical mass is built. Thus to accelerate the initial spreading, lists are built of perhaps 10,000 hosts believed to be more vulnerable to infection than randomly selected addresses—generated by stealthy scans beforehand over a period of weeks or months. The first instance of a worm retains half the list, passing the other half onto the next victim, and so on.

2. *permutation scanning*. To reduce contacting machines already infected, next-victim scans are made according to a fixed ordering (permutation) of addresses. Each new worm instance starts at a random place in the ordering; if a given worm instance learns it has contacted a target already infected, the instance resets its own scanning to start at a random place in the original ordering. A machine infected in the hit-list stage is reset to start scanning after its own place in the ordering.

3. *Internet-scale hit lists*. A list of (most) servers on the Internet can be pre-generated by scanning tools. For a given worm which spreads by exploits that a particular web server platform is vulnerable to, the addresses of all such servers can be pre-identified by scanning (vs. a smaller hit-list above). In 2002, when this approach was first proposed, there were 12.6 million servers on the Internet; a full uncompressed list of their IPv4 addresses (32 bits each) requires only 50 megabytes.

Using initial hit-list scanning to quickly seed a population (along with topologically-aware scanning perhaps), then moving to permutation scanning to reduce re-contacting infected machines, and then Internet-scale hit lists to reach pre-filtered vulnerable hosts directly, it was estimated that a *flash worm* could spread to all vulnerable Internet hosts in just tens of seconds, "so fast that no human-mediated counter-response is possible".

Exercise (Email virus-worms). Summarize major technical details of the following malware incidents spread by email: *ExploreZip*, *ILOVEYOU*, *Sircam*, *Bagle*, *MyDoom*.

Exercise (Worm incidents). (a) Summarize major technical details of the following worm instances, noting any special lessons or "firsts" related to these specific incidents. *Code Red*, *Nimda*, *SoBig*, *Sapphire/Slammer*, *Blaster*, *Witty*, *Sasser*, *Koobface*. (b) Look up, and summarize, the defining technical characteristics of a *self-stopping worm*.

9.3 Stealth: trojan horses, backdoors, keyloggers, rootkits

Malware may use *stealthy* tactics to escape or delay detection. Stealthy malware of various forms is named based on goals and methods used. We discuss a few types here.

TROJANS. By legend, the *Trojan horse* was an enormous wooden horse offered as a gift to the city of Troy. Greek soldiers hid inside as it was rolled within the city gates,

Figure 9.3: Trojan Horse (courtesy C. Landwehr, original photo, Mt. Olympus Park, WI)

emerging at nightfall to mount an attack. Today, software delivering malicious functionality instead of, or in addition to, purported functionality—with the malicious part possibly staying hidden—is called a *Trojan horse* or trojan software. Some trojans are installed by trickery through fake updates—e.g., users are led to believe they are installing critical updates for Java or video players like Adobe Flash, or anti-virus software updates; other trojans accompany miscellaneous free applications such as screen savers re-packaged with the malware. Trojans may perform benign actions while doing their evil in the background; an executable greeting card delivered by email may play music and display fancy graphics, while deleting files. The malicious functionality may become apparent immediately after installation, or might remain undetected for some time. If malware is silently installed without end-user knowledge or actions, we tend not to call it a trojan—reserving that term for when the installation of software with other-than-promised functionality is "voluntarily" (through deception) accepted into our protected zone.

BACKDOOR PROGRAMS. A *backdoor* provides ongoing remote access to a compromised machine through network connections bypassing normal authentication, and possibly hidden by rootkit functionality (see below). This enables installation of updated malware payloads, and a stealthy analogue of legitimate *remote administration* or *remote desktop* tools. Backdoors may be in stand-alone programs, or embedded into legitimate programs—e.g., standard login interface code may be modified to grant login access to a special-cased `userid`, without requiring any password. A backdoor is perhaps best viewed as a feature that facilitates other malware; this feature is common in trojans.

ROOTKITS. A *rootkit* is a collection of executable software surrepticiously installed on a computer, which takes active measures to conceal its existence, in order to provide long-term control of selected aspects of the host OS. The means used to remain hidden, the controlling of parts of the OS, and any other malicious actions (e.g., surveillance or information theft), comprise its payload functionality. *Kernel-mode rootkits* seek to

gain *root* level access, and preserve this concealed state of compromise—thus literally, a "kit" that preserves "root" access, seeking a type of *hypervisor* status over the operating system itself. The target system is said to be *rooted*. While this terminology centres on root privileges, less powerful *user-mode rootkits* meeting the opening description run with only *user-mode* privileges as held by regular applications.

ROOTKIT OVERVIEW AND GOALS. In discussing rootkits, *attacker* refers to the party aiming to install the rootkit. Most rootkits serve malicious purposes, but not all—some are used by law enforcement to gather intelligence, and by systems administrators in *honeypots* to observe the action of intrusions or malware on host systems. Rootkits typically replace system code, modify system data structures that do not impact core operating system functions, and filter results reported back to processes—aiming to hide attacker processes, files, and network connections. Rootkit payloads may include:

- *backdoor* programs (see above) for ongoing remote access to a rooted machine. This may facilitate the machine being enlisted in a botnet (see Section 9.5).
- software *keylogger* programs, which record and send user keystrokes to an attacker. The recording is done by "hooking" appropriate system calls (see below). Information targets include credit card details, passwords for password-encrypted files, and username-password pairs for online banking, corporate VPNs or enterprise accounts.
- surveillance or session logging software. Surrepticious remote use of device microphones, webcams, and sensors (e.g., GPS for geolocation) allows eavesdropping even when users are not active on their rooted device. When a user is active, their local session (including mouse movements and keystrokes) can be reflected to a remote attacker's desktop, providing a continuous screen capture. Milder versions can record subsets of information (e.g., web sites visited, files accessed).

Rootkit success requires 1) installation, 2) remaining hidden, and 3) payload functionality. The payload determines the ultimate damage, like other malware, but what distinguishes rootkits are the techniques used to remain hidden. Rootkits are viewed as "post-intrusion" tools; there are (many possible) means of installation, a few of which we mention next, but these are usually considered apart from the rootkit itself.

INSTALLING ROOTKITS. Examples of means to install kernel rootkits follow; some are non-obvious even to technical people.

1. Exploiting a vulnerability in kernel code—e.g., a buffer overflow in a networking daemon, or parsing errors in user-space applications which alter kernel parameters.
2. Modifying the boot process mechanism. One possibility is for a rogue boot loader to alter the kernel after it is loaded, but before the kernel runs.
3. Modifying code or data swapped (paged) to disk. If kernel elements are swapped onto disk, and that memory is writable by user-space applications, kernel integrity may be modified on reloading the swapped page. The *Blue Pill* exploit did this.

4. Interfaces to physical address space. This includes Direct Memory Access (DMA) writes to alter kernel memory, through use of hardware devices with such access (e.g., video and sound cards, network cards, disk drives).

5. Normal kernel module installation. An administrator may install a supposedly valid kernel module or new device driver with trojan rootkit functionality. Similarly, an attacker may socially engineer a user or administrator, who has acquired privilege to dynamically load a kernel module (see below), to load a malicious such module.

LOADABLE KERNEL MODULES. One method to install rootkits is through existing tools that allow the introduction of OS kernel code. A *loadable kernel module* (LKM) is executable code packaged as a component that can be added or removed from a running kernel, to extend or retract kernel functionality, including to implement system calls and hardware drivers. Many kernel rootkits are also LKMs. Most commercial operating systems support some form of such kernel modules which are dynamically loadable, and facilities to load and unload them—e.g., by specifying the module name at a command line interface. An LKM includes routines to be called upon loading or unloading.[8]

IMPLICATION OF MONOLITHIC KERNELS. While the importance of modular OS kernels is now recognized, many older kernels have a monolithic design consisting of core kernel code plus loadable modules and device drivers, but no memory isolation between different kernel and OS components. In this case, a kernel compromise means that malware has complete control to read or write anything in kernel memory.

HIJACKING SYSTEM CALLS. User-mode programs do not have access to kernel memory; kernel-mode allows access to all memory. Applications which require access to services involving kernel memory do so by *system calls* to operating system functions, services and library utilities. The called services execute with kernel-mode privileges. The service names are resolved to the addresses of the code implementing them through a service address table, essentially a list of code pointers. For example, on some Windows systems, this is the System Service Dispatch Table, SSDT. There are various methods of hijacking system calls. Some kernel rootkits change entries in such system call tables, redirecting the calls to rootkit code (which may, e.g., call the legitimate code and post-process results—see below). Intercepting calls in this way is known as *hooking*; the new handling code is called the *hook*. Such hooking also has many legitimate uses, including by anti-virus software. An alternative to hooking individual call table entries is to effectively replace an entire table by changing the address that calling routines expect to find it at, to that of a substitute table created elsewhere. A third hijacking method over-writes the code implementing targeted system calls.

[8] Recall the process to generate an executable suitable for loading itself involves several steps. A *compiler* turns *source code* into a collection of machine code instructions—a *binary* or *object* file. A *linker* combines one or more object files to generate an *executable* file or *program image*, which a *loader* moves from disk or secondary storage into the target machine's main memory, adjusting addresses (*address relocation*) if necessary. Linkers are compile-time tools, while loaders are run-time tools, typically included in the OS kernel. More complicated loaders such as *dynamic linkers*, can load and link shared libraries.

Exercise (Inline hooking). Another hooking method, *detour patching*, uses a *detour function* followed by a *trampoline function*. Diagram how this is used by malware.

KERNEL OBJECT MODIFICATION AND PRUNING OF REPORTED LISTS. Having the ability to read or write kernel memory, kernel rootkits may alter data structures that only core kernel software should ever modify. As one example, they can easily escalate the privilege of any process, e.g., in the OS's list of running processes, by simply setting the process id of the lucky process to `root` or `admin` (in Unix, setting process `uid` to 0). Other examples of objects useful to modify are: the list of files in a directory, the loaded module list, the scheduler list, and the process accounting list. A rootkit may easily hide a process or files by tampering with such lists and related system call results by:

1. *direct kernel object modification* (DKOM). Kernel data structures are directly altered, e.g., removing malicious processes from lists, so that they go unreported.

2. manipulating results returned by system calls. For example, when a call is made requesting a list, the resulting list can be post-processed to prune out rootkit-related processes before returning the result to the caller. This can be done by hooking the system call table (see above). More generally, a rootkit may execute some operation instead of the legitimate system call, or before or after invoking it.

Exercise (User-mode rootkits). On some systems, user-mode rootkits work by intercepting, in the address space of all user-mode processes, *resource enumeration APIs*, i.e., system calls which generate reports from secondary data structures the OS builds to efficiently answer resource-related queries. Malware-related items are filtered out before returning results. This is analogous to hooking system calls in kernel space, without needing kernel privileges. Detecting user-mode rootkits is easier than kernel rootkits, using effective tools. Give a high-level explanation, of how user-mode rootkits can be detect by a *cross-view difference approach* which compares the results returned by two API calls at different levels. (Hint: see [44]. It also makes the interesting observation that back in 2005, over 90% of rootkit incidents reported in industry were user-mode rootkits!)

Exercise (DLL injection and user-mode rootkits). *DLL injection* is a well-known programming technique. Look up and summarize how user-mode rootkits abuse it.[9]

Exercise (Case study: keylogger rootkit and Linux kernel backdoors). Summarize the technical details of the *lvtes* keylogger rootkit, which hides by modifying a module list; and give a technical overview of Linux kernel *backdoors*. (Hint: see [4].)

Exercise (Zeus malware). Summarize the technical details of *Zeus* malware, related to activity of keyloggers, ransomware, and botnets.

Exercise (Rootkits vs. trojans). Explain what distinguishes rootkits from trojans.

Exercise (Compiler trap door). An attacker puts a backdoor into the implementation of the OS `login` command, granting an unauthorized `userid` login access without entry of any password. This is done by modifying the (source code of the) compiler that compiles this system software, building this special-case logic into the `login` executable when

[9] ***Give a reference hint.

the compiler creates it. This leaves visible evidence to anyone examining the compiler source code. Explain how the evidence can be removed, by building functionality into the compiler executable to insert the backdoor into the `login` software, and to re-introduce this compiling capability into the compiler executable, even after this functionality is removed from the compiler source code. (Hint: see Thompson's classic paper [43].)

Exercise (Case study: Stuxnet worm-rootkit, 2010).[10] Summarize the technical details of *Stuxnet*, a worm with rootkit functionality said to have been the most elaborate malware development exercise to date. It targets only a specific industrial control system, and apparently severely damaged Iranian nuclear enrichment centrifuges. (Hint: see [11].)

Exercise (Case study: Sony copy protection rootkit, 2005). Summarize the technical details of the *Sony rootkit*, intended to provide copy protection on Sony CDs.

9.4 Drive-by downloads and droppers

MALWARE EXPLOITING USE OF BROWSERS. The rich functional design of Internet browsers is also exploited by malware. Web pages are documents written in HTML, a tag-based *markup language* indicating how pages on web servers should be displayed on user devices. To enable powerful *web applications* such as interactive maps, HTML supports many types of embedded content beyond static data and images. This includes sequences of instructions in *scripting languages* such as JavaScript.[11] Such *active content*—small program snippets which the browser executes as it displays a web page—runs on the user device. The browser's job is to (process and) display the web pages it receives, so in this sense, the execution of content embedded in the page is "authorized" simply by visiting a web page, even if the page includes malicious content embedded through actions of an attacker. Through a combination of a browser-side vulnerability, and a compromised or exploited server application by which a few lines of HTML code is injected into a web page, a simple web-page visit can result in binary executable malware being silently downloaded and started on the user device. This is called a *drive-by download*.

MEANS OF DRIVE-BY EXPLOITATION. Drive-by downloads use several technical means.[12] Questions to help our understanding are: how does a malicious script get embedded into a web page, how are malicious binaries downloaded, and why is this invisible to users? Malicious scripts originate from various sources, such as:

1. web-page ads (often provided through several levels of third parties);
2. web *widgets* (small third-party apps executed within a page, e.g., weather updates);
3. server vulnerabilities allowing attackers to directly inject scripts;
4. sites which solicit and reflect user-provided content (e.g., web forums);

[10] ***Stuxnet details***

[11] Older examples of active content include Flash, ActiveX controls, and Java applets.

[12] We mention a few briefly here. For more details, see the Web Security chapter.

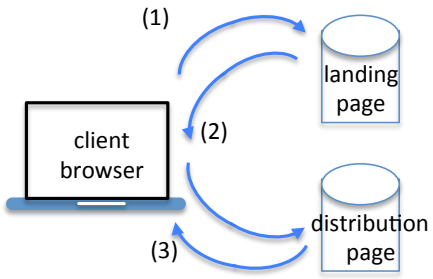

Figure 9.4: Drive-by download involving browser redirection. A browser visiting an original site (1) is often redirected (2) to a distribution site that causes silent download of malware (3), e.g., possible due to browser vulnerabilities. The redirection (2) may involve several redirect hops through intermediate sites.

5. links users click in received HTML email with malicious parameters/scripts.

A short script can redirect a browser to load a page from an attacker site, with that page redirecting to a site which downloads binaries to the user device. Silent downloading and running of a binary should not be possible, but is, through scripts which exploit browser vulnerabilities—most vulnerabilities eventually get fixed, but the pool is deep, attackers are creative, and software is always evolving. Note that the legitimate server initially visited is not involved in the latter steps. Being so frequent and rapid in legitimate browsing, redirects generally go unnoticed; and injected content is easy to visibly hide by invisible components like a zero-pixel `iframe`.

DEPLOYMENT MEANS VS. MALWARE CATEGORY. Drive-by downloads can install various types of malware—including keyloggers, backdoors, and rootkits. They can result in zombies being recruited into botnets. Rather than a separate category of malware, one might view drive-by downloads as a deployment means for spreading malware—but then again, a major defining characteristic of both viruses and worms is the way in which they each spread. A major difference from other malware, however, is that drive-by downloads involve user devices going to a web site in a "pull" model, whereas traditional viruses and worms spread in a "push" model.

DROPPERS. A *dropper*, narrowly defined, is malware whose sole function and goal is installing other malware. Thus such a dropper does not have a malicious payload itself, but rather installs malware that does. Droppers often install backdoors (discussed earlier), which allow installation of further malware and updates. A step up from a virus or worm that spreads without malicious payload is one without malicious payload but that drops in backdoors, allowing future malware to be installed; this can also be done by drive-by downloads. While the term *dropper* suggests depositing at the victim machine, and *backdoor* suggests a door to be returned to, the code that a dropper leaves behind may itself initiate a later network communication back to a central malware source. One of the first widely spread malware programs with such functionality was the virus *Babylonia* (1999),

which after installation, downloaded additional files to execute; it was non-malicious, and otherwise would have received greater notoriety. This functionality moved the world a step closer to botnets (see Section 9.5, next).[13]

9.5 Ransomware, botnets and other beasts

Ransomware is malware with a specific motive: to extort users. This typically involves compromise of a device, and communication between the compromised device and a remote computer controlled by attackers. Attackers often communicate with and control large numbers of compromised devices, in which case the collection is called a *botnet*. We discuss these and a few other forms of malware in this section.

RANSOMWARE THAT ENCRYPTS. A particularly powerful type is *encrypting ransomware*, which encrypts user data files using public-key cryptography. In this way, the malware executable need not contain the (decrypting) private key, nor expose it to risk of recovery by the legitimate user. Users are asked to pay a sum of money, before a given deadline, in return for the decryption key (from a remote site) that allows recovery of their files. Payment is commonly demanded in hard-to-trace, non-reversible forms such as pre-paid cash vouchers or *bitcoin*. Note that removal of the malware itself, which in some cases is easy, does not solve the problem: encrypted files remain unavailable.

NON-ENCRYPTING RANSOMWARE. Other variations of ransomware make files unavailable not through encryption but rather by standard access control means, or threaten to erase user files or reformat disks, or (falsely) claim to have encrypted files. Other non-encrypting ransomware may deny a user access to operating system functionality until ransom is paid, and disable operating system debug modes with reduced functionality (sometimes called *safe mode* or safe boot). In this case, ransomware may involve rootkit functionality to make removal difficult.

DEPLOYMENT OF RANSOMWARE. Ransomware may be deployed by the means as any other malware, including as trojan software (with users tricked into installing it), or by exploiting software vulnerabilities, including through drive-by downloads.

Exercise (Software download attacks). Socially-engineered software download attacks trick users to install malicious software. In such web-based attacks, summarize tactics for (a) gaining user attention, and (b) deception and persuasion. (Hint: see [23]. On a different note: the non-malicious 1999 email virus-worm *Happy99* socially-engineered users to run the attached executable.)

Exercise (Clicking to execute email attachments). Some operating systems, aiming for user-friendly interfaces, are set by default to hide file extensions. This hides an email attachment's extension, e.g., including `.exe` in the case of an executable file. If an attacker then sends an email attachment `prettyPicture.jpg.exe`, what filename will the user see? When a user double-clicks on a file in graphical UI, and the file is an executable,

[13] Malware resulting from drive-by download sites, and droppers, should be compared to *autorooters* which scan and remotely infect network services. See the Monitoring and IDS chapter. Exericise: discuss the 2000 MafiaBoy DoS attack.

what default action does an OS typically take? When an email attachment with extension `.zip` is clicked or double-clicked, what happens—are scripts run or files executed?

Exercise (Ransomware incidents). Summarize major technical details of the following ransomware instances: *Gpcode*, *CryptoLocker*, *CryptoWall*, *Locky*.

SHELLCODE, ZOMBIES AND BOTNETS. A common goal of malware is to provide an OS command shell interface that runs at *root* level, and can receive instructions from an external source. A payload that includes delivering this functionality is called *shellcode*. A computer that has been compromised by malware and can be remotely controlled, or that reports back into a controlling network (e.g., with collected information), is called a *bot* (robot) or *zombie*, the latter deriving from bad movies. A coordinated network of such machines is called a *botnet*, and the individual controlling it is the *botnet herder*. Owners of the machines on which zombie malware runs are often unaware of this state of compromise (so perhaps it is the owners that are really the zombies!).

Botnets play a big role in cybercrime. They provide critical mass and economy of scale to attackers. Compromised machines take directions—perhaps to spread further malware to increase the botnet size, to carry out distributed denial of service attacks (*DDoS*), or carry out spam-related campaigns. Spam can generate revenue through sales (e.g., of pharmaceuticals), or drive users to malicious web sites, or help spread ransomware, or keyloggers seeking bank-related and credit card details. Botnets are rented to other attackers for similar purposes. In the early 2000s, when the compromise situation was particularly bad on certain commodity operating systems, it was only half-jokingly said that all PCs were expected to serve two years of military duty in a botnet.[14]

Exercise (Motivation for botnets). Research and discuss early motivations for botnets. (Hint: see [3].)

BOTNET COMMUNICATION STRUCTURES AND TACTICS. A simple botnet command and control architecture involves a central administrative server in a client-server model, with control communications over IRC channels, allowing the herder to send one-to-many commands. Centralized systems suffer from a single point of failure—if the central node is easily found and shut down, or the centralized communications channel. The channel is obvious if zombies are coded to contact a fixed IRC server, port and channel; using a set of such fixed channels brings only marginal improvement. More advanced botnets use peer-to-peer communications, coordinating over any suitable network protocol (including HTTPS); or replace centralized control by a multi-tiered communication hierarchy in which the bot herder at the top is insulated from the zombies at the bottom by one or more tiers of intermediate or proxy communications nodes. A creative tactic that has been used is zombie machines getting their control information or malware updates by connecting to a URL, but due to arrangements made by the bot herder, the normal DNS resolution process results in the URL resolving to different IP addresses even over short periods. Such tactics complicate the reverse engineering and shutting down of botnets.

Exercise (Case study: Torpig botnet). One botnet explored in detail by security re-

[14] ***Look up statistics from research papers on the percentage of machines compromised. Add end citation to underground economy studies of UCSD.

searchers is *Torpig*. Summarize what is known about it. (Hint: see [40] and [39].)

Exercise (Botnet incidents). Summarize major technical details of the following botnet instances: *Storm*, *Conficker*, *BredoLab*, *ZeroAccess*, *Mirai*.[15]

ZERO-DAY EXPLOITS. A *zero-day exploit* (or 0-day) is an attack taking advantage of a software vulnerability that is unknown to developers of the target software, the users, and the informed public. The terminology derives from an implied timeline—the day a vulnerability becomes known is the first day, and the attack precedes that. Zero-days thus have free reign for a period of time; to stop them requires that they be detected, understood, countermeasures be made available, and then widely deployed. In many non-zero-day attacks, software vulnerabilities are known, exploits have been seen "in the wild" (in the real world, beyond research labs), software fixes and updates are available, and yet for various reasons the fixes remain undeployed. Things are worse with zero-days—the fixes are still a few steps from even being available. The big deal about zero-days is the element of surprise and extra time that buys attackers.

†**RABBITS.** If a new category of malware was defined for each unique combination of features, the list would be long, with a zoo of strange animals as in the Dr. Seuss children's book *I Wish That I Had Duck Feet*. While generally unimportant in practice, some remain useful to mention, to give an idea of the wide spectrum of possibilities, or simply to be aware of terminology. For example, the term *rabbit* is sometimes used to describe a type of virus that rapidly replicates to consume memory and/or CPU resources to reduce system performance on a host; others have used the same term to refer to a type of worm that "hops" between machines, removing itself from the earlier machine after having found a new host—so it replicates, but without population growth.

†**EASTER EGGS.** While not malware, but of related interest, an *Easter egg* is a harmless special feature, credit, or bonus content, hidden in a typically large program, accessed through some non-standard means, special keystroke sequence or codeword.

†**Exercise** (Easter eggs: history). Look up and summarize the origin of the term *easter egg* in the context of computer programs. Give three additional historical examples.

LOGIC BOMBS. A *logic bomb* is a sequence of instructions, often hosted in a larger program, which takes malicious action under a specific (set of) condition(s), e.g., when a particular user logs in, or a specific account is deactivated (such as when an employee is fired). If the condition is a specific date, it may be called a *time bomb*. In pseudo-code:

```
if trigger_condition_true() then run_payload()
```

This same construct was in our pseudo-code descriptions of viruses and worms. The term *logic bomb* simply emphasizes that a malicious payload is conditional, putting the bad outcome under programmable control. From this viewpoint, essentially all malware is a form of logic bomb (often with default condition `TRUE`). Thus logic bombs are also spread by any means that spreads malware (e.g., viruses, worms, drive-by downloads).

[15] DDoS (distributed denial of service) attacks are discussed in the chapter on Monitoring and IDS.

Category name	Property			
	BREEDS?	HOSTED?	STEALTHY?	NOTES
virus	yes	yes	no	VU
worm	yes	no	no	VN
trojan	no	yes	yes	SE,T,S
backdoor	no	may be	yes	SE,T,S
rootkit, keylogger	no	no	yes	SE,T,S
drive-by download	no	no	often	browser "pulls"
botnet	–	–	often	organizes zombies

Table 9.1: Malware categories and properties. Codes for "propagation means" in the NOTES column: VU (vulnerability exploited with user-enabling), VN (vulnerability exploited in network services), SE (social engineering), T (intruder; includes malware on an already-compromised system), S (insider; includes developers, and web sites hosting malware knowingly or not).

9.6 Categorizing malware

CHARACTERISTICS USEFUL TO CATEGORIZE MALWARE. The properties and context from earlier sections can help us classify malware, based on the following questions.

- Does it breed, in the sense of self-replicating? Note that a drive-by download web site causes malware to spread, but the site itself does not self-replicate. Similarly, trojans and rootkits may spread by various means, but such means are typically independent of the core functionality which characterizes them.

- By what means is it spread? Automatically over networks or with user help? If the latter, does it involve social engineering to persuade users to take an action triggering installation (even if as simple as a mouseclick on some user interfaces)?

- Does it require the aid of an *insider*? (i.e., someone with some privilege, authorization or access beyond that of ordinary users, or non-registered users of a system).

- Is it covert by way of trying to conceal its functionality (stealthy), or take measures to hide itself from detection (rootkit-like)?

- Does it require a host program, as a parasite does?

- Is it transient (e.g., active content in HTML pages) or persistent (installed on disk, restarting later)?

MALWARE OBJECTIVES. Another way of categorizing malware is to use not its technical characteristics, but rather its underlying goals. Here are a few.

1. *Damage to host machine and its data.* The goal may be intentional destruction of data, or disrupting the host machine. Examples include crashing the operating system, and deletion, corruption, or modification of files or entire disks.

2. *Information theft.* Documents stolen may be corporate strategy files, intellectual property, credit card details, and personal data. Electronic credentials stolen, including account details, passwords and crypto keys, may allow fraudulent account login, including for online banking and enterprise accounts; or sold en masse, to others on underground or non-public networks (e.g., *darknets*). Stolen information is sent to attacker-controlled computers.

3. *Direct financial gain.* Direct credit cards risks include deceiving users to purchase unneeded online goods such as fake anti-virus software. Users may also be extorted, as in the case of *ransomware*. Malware may generate revenue by being rented out, e.g., on darknets (see above).

4. *Ongoing surveillance.* User voice, video, and actions may be recorded surreptitiously, captured by microphones and web cameras on mobile and desktop devices, and software which records web sites visited, keystrokes and mouse movements.

5. *Spread malware.* Compromised machines may be used to further spread malware.

6. *Control of computing resources.* Once a machine is compromised, code may be installed for later execution or to allow later access through *backdoor* interfaces which bypass normal access controls. Computing cycles and communications resources of compromised devices may be remotely controlled for arbitrary purposes, and organized into botnets. Compromised computers may be used as host servers for *phishing* attacks, or *stepping stones* for further attacks, in both cases reducing the risk of such attacks being traced back to the individuals behind them. Zombie machines that are part of a spam-sending botnet are sometimes called *spambots*, while those in a DDoS botnet are *DDoS zombies*.

PROTECTING SECRETS AND LOCAL DATA. The risk of client-side malware motivates encrypting locally stored data. *Encrypted file systems* automatically encrypt data stored to the file system, and decrypt data upon retrieval. To encrypt all data written to disk storage, either software or hardware-supported *disk encryption* can be used.

Exercise (Encrypting data in RAM). Secrets such as passwords and cryptographic keys which are in cleartext form in main memory (RAM) open the possibility of compromise. For example, upon system crashes, RAM memory is often written to disk for forensic purposes. (a) What can be done to address this concern? (b) If client-side malware scans RAM memory to find crypto keys, are they easily found? (Hint: see [32]).

Exercise (Hardware storage for secrets). Being concerned about malware access to secret keys, you decide to store secrets in a *hardware security module* (HSM), which prevents operating system and application software from directly accessing secret keys. Does this fully address your concern, or could malware running on a (now) untrusted host misuse the HSM? (Hint: look up the *confused deputy problem*.)

9.7 ‡Endnotes and further reading

SUPPOSE RUNNING OF UNAUTHORIZED CODE WAS FULLY STOPPED. Many computer security problems would disappear if all unauthorized code was prevented from running. We have no easy way to do this in today's Internet ecosystem—and *social engineering* methods that trick users into running code make the term *authorized* itself hard to define. But suppose we had a way. What classes of attack would remain? Here are a few. (1) Impersonation attacks using fraudulent credentials over legitimate authentication interfaces. (2) Attacks that somehow bypass standard authentication interfaces. (3) Attacks which abuse input interfaces, through specially-crafted inputs which exploit software vulnerabilities—but many of these result in running unauthorized code, which our premise excludes. It's interesting to think about what other major classes this leaves.

IMPORTANCE OF REVERSE ENGINEERING AS A SKILL. As noted, those who create malware use various means to make its detection, prevention, and removal difficult. Security analysts whose job it is to provide tools addressing malware, including anti-virus or anti-malware tools, must make use of various *reverse engineering* tools and skills.

REFERENCES AND FURTHER READING. Aycock [2] is an excellent short book on malware. Szor [42] is comprehensive and authoritative. Curry motivates his classic 1992 book on Unix security [7] with an opening chapter giving brief summaries of early malware incidents. Denning's collection of papers [8] on malware appeared soon after the Internet worm and early viruses, including the virus primer by Spafford et al. [36] and overview by Cohen [5]. The proof by contradiction (Section 9.2) is from Cohen [6, p.64], a book based on one-day short courses. Ludwig's earlier book [17] includes assembler code, with a free online electronic edition. Related to early Unix viruses and also shell scripts (and aside from the earlier exercise citing McIlroy [19]) see Duff [9]. Kong [16] gives details on developing Unix kernel rootkits, with focus on maintaining (rather than developing exploits to gain) root access; for Windows kernel rootkits, see Hoglund and Butler [12] and Kasslin [14]. Jaeger et al. [13] discuss hardening kernels from rootkit-related malware abusing standard means to modify kernel code. For rootkits installing hypervisors or virtual machine monitors, see *Blue Pill* [30] and *SubVirt* [15]. *The Shellcoder's Handbook* [1] details techniques for running attacker-chosen code on victim machines, with justification "The bad guy's already know this stuff; the network-auditing, software-writing, and network-managing public should know it too". Similarly-positioned literature includes Stuttard and Pinto [41], McClure et al. [18], Skoudis and Zeltser [34], Skoudis and Liston [33], and (emphasizing reverse engineering) Peikara and Chuvakin [25]. Many attacks exploit lack of differentiation between code and data (e.g., altering control flow by over-writing code pointer data); manipulating data can thus manipulate program execution. Tracking a hacker differs from addressing malware—see Stoll [38].

Staniford et al. [37] analyze the spread of worms (e.g., *Code Red*, *Nimda*) and ideas for *flash worms*. Sources for information about malware include the U.S. national vulnerability database [24], related CVE list of Common Vulnerabilities and Exposures [20], the CWE dictionary of software weakness types [21], and the SecurityFocus (Symantec) vulnerability database [31]. For drive-by downloads see Provos et al. [26], [27].

References

[1] C. Anley, J. Heasman, F. Lindner, and G. Richarte. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes (2/e)*. Wiley, 2007. 718 pages. First edition (2004, J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, R. Hassell), 620 pages.

[2] J. Aycock. *Computer Viruses and Malware*. Springer Science+Business Media, New York, NY, 2006.

[3] D. Bradbury. The metamorphosis of malware writers. *Computers & Security*, 25(2):89–90, 2006.

[4] A. Chakrabarti. An introduction to Linux kernel backdoors. The Hitchhiker's World, Issue #9, 2004. `http://www.infosecwriters.com/hhworld/hh9/lvtes.txt`.

[5] F. Cohen. Implications of computer viruses and current methods of defense. In *[8] as Article 22*, pages 381–406, 1990. Updates an earlier version in *Computers and Security, 1988*.

[6] F. B. Cohen. *A Short Course on Computer Viruses (2nd edition)*. John Wiley, 1994.

[7] D. A. Curry. *UNIX System Security: A Guide for Users and System Administrators*. Addison-Wesley, 1992.

[8] P. J. Denning, editor. *Computers Under Attack: Intruders, Worms, and Viruses*. Addison-Wesley, 1990. Edited collection (classic papers, articles of historic or tutorial value.

[9] T. Duff. Experience with viruses on UNIX systems. *Computing Systems*, 2(2):155–171, 1989.

[10] M. W. Eichin and J. A. Rochlis. With microscope and tweezers: an analysis of the Internet virus of November 1988. In *IEEE Symp. Security and Privacy*, pages 326–343, 1989.

[11] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet dossier. Report, ver. 1.4, 69 pages, Symantec Security Response, Cupertino, CA, February 2005.

[12] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, 2005.

[13] T. Jaeger, P. van Oorschot, and G. Wurster. Countering unauthorized code execution on commodity kernels: A survey of common interfaces allowing kernel code modification. *Computers & Security*, 30(8):571–579, 2011.

[14] K. Kasslin, M. Ståhlberg, S. Larvala, and A. Tikkanen. Hide'n seek revisited – full stealth is back. In *Virus Bulletin Conference (VB)*, pages 147–154, 2005.

[15] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *IEEE Symp. Security and Privacy*, pages 314–327. IEEE Computer Society, 2006.

[16] J. Kong. *Designing BSD Rootkits: An Introduction to Kernel Hacking*. No Starch Press, 2007.

[17] M. Ludwig. *The Little Black Book of Computer Viruses*. American Eagle Publications, 1990. An exposition on programming computer viruses with complete virus code; electronic edition (1996) free online: `http://vx.netlux.org/lib/vml00.html`.

[18] S. McClure, J. Scambray, and G. Kurtz. *Hacking Exposed (6th edition)*. McGraw-Hill, 2009.

[19] M. D. McIlroy. Virology 101. *Computing Systems*, 2(2):173–181, 1989.

[20] Mitre Corp. CVE–Common Vulnerabilities and Exposures. `http://cve.mitre.org/cve/index.html`.

[21] Mitre Corp. CWE–Common Weakness Enumeration: A Community-Developed Dictionary of Software Weakness Types. `http://cwe.mitre.org`.

[22] C. Nachenberg. Computer virus-antivirus coevolution. *Commun. ACM*, 40(1):46–51, 1997.

[23] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad. Towards measuring and mitigating social engineering software download attacks. In *USENIX Security Symp.*, 2016.

[24] NIST. National Vulnerability Database. National Institute of Standards and Technology, U.S. Dept. of Commerce. `https://nvd.nist.gov/`.

[25] C. Peikari and A. Chuvakin. *Security Warrior*. O'Reilly Media, 2004.

[26] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your `iFRAME`s point to us. In *USENIX Security Symp.*, 2008.

[27] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser: Analysis of web-based malware. In *USENIX HotBots*, 2007.

[28] J. A. Rochlis and M. W. Eichin. With microscope and tweezers: the Worm from MIT's perspective. *Commun. ACM*, 32(6):689–698, 1989. Reprinted in [8] as Article 11; a more technical related paper by these authors appeared as [10].

[29] A. D. Rubin. *White-Hat Security Arsenal*. Addison-Wesley, 2001.

[30] J. Rutkowska. Subverting Vista kernel for fun and profit. Blackhat talk, 2006. `http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf`.

[31] SecurityFocus. Vulnerability Database. Symantec. `http://www.securityfocus.com/vulnerabilities`.

[32] A. Shamir and N. van Someren. Playing "hide and seek" with stored keys. In *Financial Cryptography and Data Security (FC)*, volume 1648 of *Lecture Notes in Computer Science*, pages 118–124. Springer, 1999.

[33] E. Skoudis and T. Liston. *Counter Hack Reloaded: A Step-by-Step Guide to Computer Attacks and Effective Defenses*. Prentice Hall, 2006. (with Tom Liston) 784 pages; see `http://www.counterhack.net/Counter_Hack`.

[34] E. Skoudis and L. Zeltser. *Malware: Fighting Malicious Code*. Prentice Hall, 2003. (with Lenny Zeltser) 672 pages; intended for systems administrators.

[35] E. H. Spafford. Crisis and aftermath. *Commun. ACM*, 32(6):678–687, 1989. Reprinted in [8] as Article 12.

[36] E. H. Spafford, K. A. Heaphy, and D. J. Ferbrache. A computer virus primer. In *[8] as Article 20*, pages 316–355, 1990.

[37] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in your spare time. In *USENIX Security Symp.*, 2002.

[38] C. Stoll. *The Cuckoo's Egg*. Simon and Schuster, 1989.

[39] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. A. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: Analysis of a botnet takeover. In *ACM Conf. Computer and Comm. Security (CCS)*, pages 635–647. ACM, 2009.

[40] B. Stone-Gross, M. Cova, B. Gilbert, R. A. Kemmerer, C. Kruegel, and G. Vigna. Analysis of a botnet takeover. *IEEE Security & Privacy Magazine*, 9(1):64–72, 2011.

[41] D. Stuttard and M. Pinto. *The Web Application Hacker's Handbook*. Wiley Publishing, 2008.

[42] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley and Symantec Press, 2005.

[43] K. Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, 1984.

[44] Y.-M. Wang and D. Beck. Fast user-mode rootkit scanner for the enterprise. In *Large Installation System Administration Conference (LISA)*, pages 23–30. USENIX, 2005.