Chapter 7

Malicious Software

The official version of this book is available at
https://www.springer.com/gp/book/9783030834104

ISBN: 978-3-030-83410-4 (hardcopy), 978-3-030-83411-1 (eBook)

version: 15 Oct 2021

Chapter 7

Malicious Software

This chapter discusses malicious software (*malware*) in categories: computer viruses and worms, rootkits, botnets and other families. Among the many possible ways to name and classify malware, we use groupings based on characteristics—including propagation tactics and malware motives—that aid discussion and understanding. We consider why it can be hard to stop malware from entering systems, to detect it, and to remove it.

Malware often takes advantage of specific software vulnerabilities to gain a foothold on victim machines. Even when vulnerabilities are patched, and software updates eliminate entire classes of previous vulnerabilities, it remains worthwhile to understand past failures, for awareness of recurring failure patterns. Thus in a number of cases here and in other chapters, we discuss some malware instances even if the specific details exploited are now well understood or repaired in software products of leading vendors. The lessons remain valuable to reinforce good security design principles, lest we repeat past mistakes.

7.1 Defining malware

We define *malicious software* (*malware*) as software intentionally designed or deployed to have effects contrary to the best interests of one or more users (or system owners or administrators), including potential damage related to resources, devices, or other systems. Damage might involve, e.g., data, software, hardware, or compromise of privacy.

In most cases, if users had full knowledge of the design intent or possible consequences of such malware, they would (if given a choice) not allow it to run. In this sense, malware runs without the explicit approval of an (all-knowing, benign) user. A broader class, *harmful software*, includes also software that *inadvertently* causes damage due to design or implementation errors. Harmful software is a concern in *dependable and secure computing*, but is not our primary focus here—although the same vulnerabilities may come into play. Indeed, any means by which benign software may end up causing harm can typically be harnessed maliciously. That said, this chapter is organized around the strategies and end-goals (design intent) of malware.

QUESTIONS REGARDING MALWARE. We begin with some questions that introduce

concepts to be discussed, and help organize our exploration in this chapter.

1. *How does malware get onto computer devices?* One way is via web sites—by links in *phishing* emails, search engine results, and web page ads directing traffic to both compromised legitimate sites and malicious sites. (Related *pharming* attacks, in Chapter 11, disrupt IP address resolution to misdirect browsers.) Downloaded executables that users intentionally seek may be repackaged to include bundled malware; users may be tricked to install executables that are either pure malware, or contain hidden functionality; or a site visit may result in software installation without user knowledge. (As Section 7.6 explains, this may happen by *drive-by downloads* and malicious *active content* in web pages exploiting browser vulnerabilities, or via applications that browsers invoke to process content.) *Computer worms* spread malware by exploiting vulnerabilities in network communications services. *Computer viruses* spread by various means including malicious email attachments. Malware may also be embedded in source code in development repositories; legitimate developers may play the role of *insiders* (Chapter 1), or repositories may be compromised by *outsiders*. Even computer firmware and hardware may be malicious—depending on how firmware is provided and updated, and controls within the hardware supply chain.
2. *What makes malware hard to detect?* Detection is easy in some cases, but hard in general, for multiple reasons. What malware is 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, an easy theoretical result (Section 7.2) shows that malware identification is an undecidable problem. Personal viewpoints may also differ—is a useful program that also displays ads to generate revenue malware? In this sense, some forms of malware are more aggressive than others. Often, malware is also specifically designed to be hard to detect, and hard to reverse-engineer (Sections 7.3–7.5 discuss anti-detection and hiding techniques.)
3. *How can installation of malware be prevented?* If we can't decide what malware is (above), it seems unreasonable to expect any program to prevent all forms of it. Restricting what software users are allowed to 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 that test, before installing or running, that executable content (including updates) is from known sources. *Anti-virus/malware* tools and *intrusion detection systems* (Chapter 11) are industry-driven partial solutions. Some tools remove or filter out specific instances of detected malware; in severe cases a host machine's entire software base may need to be re-installed with a clean base OS and all applications—with loss of any data files not recoverable from backup storage. (Losing files in this way can ruin a good day at the office!)

SOFTWARE CHURN, EASE OF INSTALLATION ENABLE MALWARE. In early computer systems, end-users were not directly involved in software installation or upgrades. Neither network-downloaded software nor wireless software updates existed. Computers

came with pre-installed software from device manufacturers. Expert information technology (IT) staff would update or install new operating system or application software from master copies on local storage media via CD ROM or floppy disks. 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—as well as deployment of malware. Allowing end-users to easily authorize, install and update almost any software on their devices opened new avenues for malware to gain a foothold, e.g., by tricking users to "voluntarily" install software that misrepresents its true functionality (e.g., *ransomware*) or has hidden functionality (*Trojan horse* software). Users also have few reliable signals (see Chapter 9) from which to identify the web site a download arrives from, or whether even a properly identified site is trustworthy (legitimate sites may become compromised). These issues are exacerbated by the high "churn rate" of software on network infrastructure (servers, routers) and end-user devices. Nonetheless, an evolving set of defenses allows us to (almost) keep up with attackers.

7.2 Viruses and worms

The first types of malware to gain notoriety were computer *viruses* and *worms*. They differ in some aspects, but share a distinguishing *propagation* feature—they employ clever means to cause their number of instances to grow, and spread across machines.

DEFINITION: VIRUS. Following pioneer 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 whether a file is already infected; infecting only new files is more effective.

Computer virus	Computer worm

`loop`
`  remain_dormant_until_host_runs();`
`  propagate_with_user_help();`
`  if trigger_condition_true() then`
`  run_payload();`
`endloop;`
	`loop`
	`  propagate_over_network();`
	`  if trigger_condition_true() then`
	`  run_payload();`
	`endloop`

Table 7.1: Comparison of viruses and worms (pseudo-code). The propagation steps may be viewed as a strange (malicious) variation of process forking. Viruses are possible due not to flaws, but to the nature of computers and their features: *"If you have a programmable computer with a filesystem inhabited by both programs and data, you can make viruses. It doesn't matter what hardware or operating system you are using"* [36].

GENERIC STRUCTURE. A high-level comparison of virus and worm structure is given by the pseudo-code in Table 7.1. It shows four generic parts or stages of a virus.

1. *Dormancy*. A virus is typically dormant until the host program runs.
2. *Propagation*. This is when (and how) the malware spreads.
3. *Trigger condition*. This controls when the payload is executed.
4. *Payload*. This is the functionality delivered by the malware (other than propagating). Payload actions range from relatively benign (an image walking across a screen) to severe (erasing files, or taking software actions that damage hardware).

HOW WORMS DIFFER. Worms differ from viruses in three main ways.

i) Worms propagate automatically and continuously, without user interaction.

ii) Worms spread across machines over networks, leveraging network protocols and network daemons (rather than infecting host programs beforehand as viruses do).

iii) Worms exploit software vulnerabilities, e.g., buffer overflows, while viruses tend to abuse software features or use social engineering.

As a result of how they spread, worms are also called *network worms* or *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* (Chapter 11), even when that is not 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 email-related file infection, attachments, and features of clients and infrastructure (often enabled by default). It typically requires a user action (e.g., opening an email client or reading a message), and may involve *social engineering* (tricking the user into taking some action). A common tactic is to extract next-targets from the mail client's address book. Since email allows long recipient lists, spreading is one-to-many.

MAGIC, MALWARE AND PRIVILEGES. There is nothing magical about viruses, worms, and other malware. They are simply software, with power and functionality as available to other software. On the other hand, the tremendous functionality of regular software may itself seem magical—and like "ordinary" software, malware can thus do extraordinarily complex things, especially if it runs with elevated privileges.

Exercise (Malware privileges). Does malware control its own privileges? Explain. (Hint: Chapter 5, and Section 7.4 for the relationship of root privileges to kernel mode.)

PROGRAM FILE VIRUSES. Most viruses infect executable program files. How and where virus code is inserted (in the host file) varies. Strategies include (Figure 7.1):

(a) 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.

(b) 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. The 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.

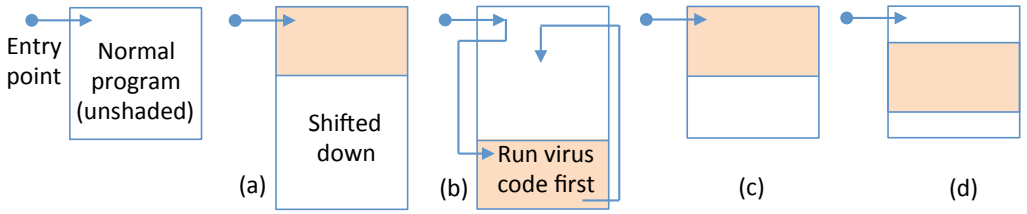

Figure 7.1: Virus strategies for code location. Virus code is shaded. (a) Shift and prepend. (b) Append. (c) Overwrite from top. (d) Overwrite at interior.

(c) 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 that the virus is noticed, and complicates its removal (a removal tool will not have the original program file content available to restore).

(d) Overwrite the host file, starting from some interior point (with luck, a point 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 that, as an optimization, take shortcuts such as scanning for viruses only at the start and end 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*.

‡**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: [36]).

‡**VIRUSES: ALTERNATE DEFINITION.** Using command shells and scripts, and environment variable settings 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*. Szor's alternative 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*.[1] 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 overwrite or replace-and-relocate the boot sector code, so

[1] Similar malware is called a *bootkit* (Section 7.4); malware that runs before the OS is hard to detect.

that virus code runs first. The Brain virus occasionally destroyed the *file allocation table* (FAT) of infected floppies, causing loss of user files. It was not, however, particularly malicious—and although stealthy,[2] the virus binary contained the note "Contact us for vaccination" and the correct 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 always carefully back 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 truly malicious payload!) *CIH* was also called *Spacefiller*—unlike viruses that insert themselves at the top or tail of a host file (Figure 7.1), 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 that 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 documents contain embedded scripts and markup instructions; "opening" them for display or viewing triggers associated applications to parse, interpret, template, and preprocess 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 a virus 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 technical details of *Concept virus*, the first "in-the-wild" *macro virus* infecting Microsoft Word documents. (b) Summarize the technical details of another macro virus that infected such documents: *Melissa*. (Aside: neither had a malicious payload, but Melissa gained 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.)

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

VIRUS DETECTION: UNDECIDABLE PROBLEM. It turns out to be impossible for a single program to correctly detect all viruses. To prove this we assume the existence

[2] Brain was the first malware known to use rootkit-like deception. Through a hooked interrupt handler (Section 7.5), a user trying to read the boot sector would be shown a saved copy of the original boot sector.

of such a program and show that this assumption results in a logical contradiction (thus, *proof by contradiction*). Suppose you claim to hold a virus detector program V that, given any program P, can return a {TRUE, FALSE} result $V(P)$ correctly answering: "Is P a virus?" Using your program V, I build the following program instance P^*:

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

Now let's see what happens if we run V on P^*. Note that P^* is a fixed program (does not change). Exactly one of two cases can occur, depending on whether V declares P^* a virus:

> CASE 1: V(P^*) is TRUE. That is, V declares that P^* is a virus.
> In this case, running P^*, it simply exits. So P^* is actually not a virus.
>
> CASE 2: V(P^*) is FALSE. That is, V declares that P^* is not a virus.
> In this case running P^* will infect a new target. So P^* is, in truth, a virus.

In both cases, your detector V fails to deliver on the claim of correctly identifying a virus. Note this argument is independent of the details of V. Thus no such virus detector V can exist—because its existence would result in this contradiction.

WHAT THIS MEANS. This proof sketch may seem like trickery, but it is indeed a valid proof. Should we then give up trying to detect viruses in practice? No. Even if no program 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 agents creating viruses continue to develop new techniques, making detection increasingly difficult. This results in an attacker-defender *move-countermove game* of increasing complexity.

VIRUS DETECTION IN PRACTICE. A basic method to detect malware is to obtain its object code, and then find *malware signatures*—relatively short byte-sequences that uniquely identify it. Candidate signatures are regression-tested against extensive program databases, to ensure uniqueness (to avoid mistakenly flagging a valid program as a virus). Then, signatures for malware active in the field are stored in a dataset, and before any executable is run by a user, an AV (anti-virus) program intervenes to test it against the dataset using highly efficient pattern-matching algorithms. This denylist-type mechanism protects against known malware, but not new malware (Section 7.7 discusses such "zero-days" and using system call hooking to intervene). Alternatively, an allowlist-type mechanism to detect malware uses *integrity-checker* or change-detection programs (e.g., Tripwire, Chapter 2), using lists of known-good hashes of valid programs. An AV program may forego byte-matching on a to-be-run executable by use of such allowlists, or if the executable has a valid digital signature of a trusted party. An extension of byte-match signatures is the use of *behavioral signatures*; these aim to identify malware by detecting sequences of actions (behaviors) pre-identified as suspicious (e.g., system calls, attempts to disable certain programs, massive file deletions). Briefly pre-running target executables in an emulated environment may be done to facilitate behavioral detection, and so that malware self-decrypts (below), which then allows byte-pattern matching.

‡**Exercise** (Self-stopping worm). Look up, and summarize, the defining technical characteristics of a *self-stopping worm* (hint: [33]).

‡**Exercise** (Email worm-viruses). Summarize the technical details of these malware incidents spread by email: a) *ExploreZip*, b) *ILOVEYOU*, c) *Sircam*, d) *Bagle*, e) *MyDoom*.

‡**Exercise** (Worm incidents). Summarize the technical details of these incidents, noting any special lessons or "firsts" related to individual instances: i) *Code Red* and *Code Red II*, ii) *Nimda*, iii) *SoBig*, iv) *Sapphire/Slammer*, v) *Blaster*, vi) *Witty*, vii) *Sasser*.

7.3 Virus anti-detection and worm-spreading techniques

This section discusses basic methods used by viruses to attempt to avoid being detected, and some tactics used by network worms seeking to spread more rapidly.

ANTI-DETECTION STRATEGIES. A virus making no attempt to evade detection consists of static cleartext code as in normal programs. Advanced viruses may use encryption or self-variation (as explained next) in an attempt to evade being identified and reverse-engineered. This gives one way to classify viruses, as follows (Fig. 7.2).

(a) *Virus with 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 that remains unmodified (which is 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.

(b) *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.

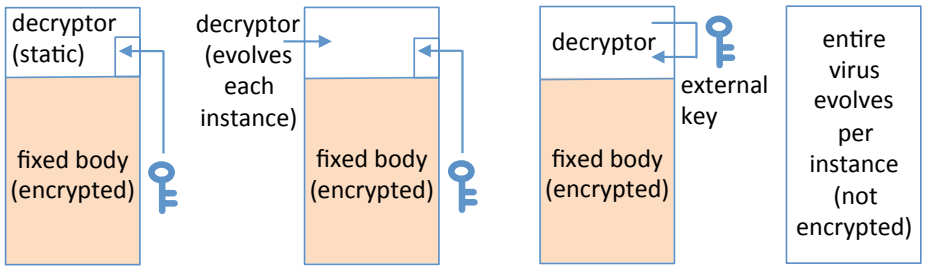

Figure 7.2: Virus anti-detection strategies. (a) Encrypted body. (b) Polymorphic virus, including self-mutation of decryptor. (c) External decryption key. (d) Metamorphic virus.

(c) *Virus 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 on 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 lookup with a frequently changed name-address mapping.

(d) *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 strategies aim to hide the virus code itself. Other tactics aim to hide telltale signs of infection, such as changes to filesystem attributes (e.g., file bytelength, timestamp), the location or existence of code, and the existence of running processes and the resources they consume. Section 7.4 notes hiding techniques (associated with rootkits).

‡**IMPORTANCE OF REVERSE ENGINEERING AS A SKILL.** As malware authors use various means of obfuscation and encryption to make it difficult to detect and remove malware, *reverse engineering* is an important skill for those in the anti-virus (anti-malware) industry whose job it is to understand malware, identify it and provide tools that remove it. Defensive experts use extensive knowledge of machine language, interactive debuggers, disassemblers, decompilers and emulation tools.

AUTO-ROOTERS. An *auto-rooter* is a malicious program that scans (Chapter 11) for vulnerable targets, then immediately executes a remote exploit on a network service (per network worms) to obtain a root shell and/or install a rootkit, often with backdoor and associated botnet enrolment. Such tools have fully automated "point-and-click" interfaces (requiring no technical expertise), and may accept as input a target address range. Vulnerable targets are automatically found based on platform and software (network services hosted, and version) for which exploits are in hand. Defenses include: disabling unused network services, updating software to patch the latest known vulnerabilities, use of firewalls (Chapter 10) and intrusion detection systems (Chapter 11) to block and/or detect scans at gateways, and on-host anti-virus software to stop or detect intrusions.

LOCALIZED AND CONTEXT-AWARE SCANNING. Worms spread by a different means than viruses. A worm's universe of possible next-targets is the set of network devices reachable from it—traditionally the full IPv4 address space, perhaps parts of 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 randomly from the entire IPv4 address space.

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* (below), and other *context-aware scanning* strategies select next-target addresses by harvesting information related to the current host machine, including: email address lists, peer-to-peer lists, URLs on disk, and addresses in browser bookmark and favorite site lists. These are all expected to be populated addresses.

FASTER WORM SPREADING. The following ideas have been brought to the community's attention as means that improve the speed at which worms may spread:

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* (Chapter 11) beforehand over a period of weeks or months. The first instance of a worm retains half the list, passing the other half on to the next victim, and each proceeds likewise.
2. *permutation scanning*. To reduce re-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 that 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 hit-list scanning to quickly seed a population (along with context-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".

THE 1988 INTERNET WORM. The *Morris worm* was the first widescale incident demonstrating the power of network worms. It directly infected 10% of Internet devices (only Sun 3 systems and VAX computers running some variants of BSD Unix) then in use, but worm-related traffic overloaded networks and caused system crashes through resource consumption—and thus widespread *denial of service*. This was despite 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 local target (for compatibility). It took steps to hide itself. Four software artifacts exploited were:

1) a stack buffer overrun in `fingerd` (the Unix `finger` daemon, which accepts network connections resulting from the **finger** command);

2) a backdoor-like **debug** command (that remained enabled) in the `sendmail` program;
3) a password-guessing attack using */etc/passwd*, with discovered passwords then supplied with commands sent to remote computers using `rexec`;[3] and
4) abuse of trusted remote logins through */etc/hosts.equiv* using `rsh`.[4]

This early "wake-up call" foreshadowed a wave of malicious worms in the early 2000s.

‡**Exercise** (Morris worm details). Explain the technical details of the exploits used by the *Morris worm*, and the lessons learned (hint: [46], [56], [47, pages 19-23]). (Aside: one resulting recovery procedure was for all users on affected systems to change their passwords; this was in 1988. When a 2016 Yahoo! compromise affected over a billion users, Yahoo! users were asked to do the same. Hmmm ... is this progress?)

7.4 Stealth: Trojan horses, backdoors, keyloggers, rootkits

Malware may use *stealthy* tactics to escape or delay detection. Stealthy malware is named (e.g., Figure 7.3) based on goals and methods used. We discuss a few types in this section.

Figure 7.3: Trojan horse (courtesy C. Landwehr, original photo, Mt. Olympus Park, WI)

TROJAN HORSE. 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, 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, video players such as Adobe Flash, or anti-virus software; other Trojans accompany miscellaneous free applications such as screen savers repackaged with accompanying 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

[3] `rexec` allows execution of shell commands on a remote computer, if a username-password is also sent.
[4] Another *Berkeley r-command*, `rsh` sends shell commands for execution by a shell on a remote computer.

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 this term for when the installation of software with extra functionality is "voluntarily" accepted into a protected zone (albeit without knowledge of its full functionality).

BACKDOORS. A *backdoor* is a way to access a device bypassing normal entry points and access control. It allows ongoing stealthy remote access to a machine, often by enabling a network service. A backdoor program contacted via a backdoor may be used for malware installation and updates—including a RAT (*Remote Access Trojan*), a malicious analogue of legitimate *remote administration* or *remote desktop* tools. Backdoors may be stand-alone or embedded into legitimate programs—e.g., standard login interface code may be modified to grant login access to a special-cased username without requiring a password. A backdoor is often included in (provided by) Trojan software and rootkits.

ROOTKITS. A *rootkit* on a computing device is a set of software components that:

1) is surreptitiously installed and takes active measures to conceal its ongoing presence;
2) seeks to control or manipulate selected applications and/or host OS functions; and
3) facilitates some long-term additional malicious activity or functionality.

The techniques used to remain hidden and control other software functionality distinguish rootkits from other malware. The end-goal, however, is facilitating malicious payload functionality (e.g., surveillance, data theft, theft of CPU cycles). The main categories are user mode and kernel mode rootkits (*hypervisor rootkits* are noted on page 208).

USER MODE VS. KERNEL MODE. The term *rootkit* originates from Unix systems, where a *superuser* (user whose processes have UID 0) is often associated with username `root`; a system hosting a malicious user process running with UID 0 is said to be *rooted*. Recall that while a superuser process has highest privileges among user processes, it is still a *user process*, with memory allocated in *user space*, i.e., non-kernel memory; user processes do not have access to kernel memory, hardware, or privileged instructions. When malware was (later) created that compromised kernel software, the same term *rootkit* was reused—creating ambiguity. To be clear: the *mode bit* is a hardware setting, which changes from user mode to super*visor* (kernel) mode, e.g., on executing the opcode that invokes *system calls*; in contrast, super*user* implies UID 0, a data value recognized by OS software. A root (UID 0) process does not itself have kernel privileges; it can access kernel resources only by a syscall invoking a kernel function. Thus a *user mode rootkit* is a rootkit (per our opening definition) that runs in user space, typically with superuser privileges (UID 0); a *kernel mode rootkit* runs in kernel space (i.e., kernel software was compromised), with access to kernel resources and memory, and all processor instructions. Kernel mode rootkits are more powerful, and harder to detect and remove. Thus the single-word term *rootkit* is an incomplete description in general, and if interpreted literally as providing root-level privileges, understates the power of kernel rootkits.

ROOTKIT OVERVIEW, GOALS. In discussing rootkits, *attacker* refers to the deploying agent. While rootkits are malicious from the target machine's viewpoint, some have, from the deploying agent's viewpoint, intent that is noble or serves public good, e.g.,

gathering intelligence by law enforcement, or observing intrusions on domain hosts, by systems administrators using *honeypots*. Rootkits typically replace system code, modify system data structures that do not impact core OS functions, alter/erase log files, and filter results reported back to processes—to hide attacker processes, executables, and network connections. Rootkit payloads may be any (typically stealthy) malware, including:

i) *backdoor* functionality (above) for ongoing remote access to a compromised machine. This may facilitate the machine being enlisted in a botnet (Section 7.7).

ii) software *keylogger* programs, which record and send user keystrokes to an attacker. This involves *hooking* (Section 7.5) appropriate system calls. Information targets include credit card details, username-password pairs for online banking, corporate VPNs or enterprise accounts, and passwords for password-encrypted files.

iii) *surveillance* or session-logging software. Surreptitious 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 variations record subsets of information (e.g., web sites visited, files accessed).

Rootkit success depends on: 1) installation, 2) remaining hidden, and 3) payload functionality. As usual, the payload determines the ultimate damage. Malicious payloads with functionality visible to end-users betray stealth, but rootkit-related features (including stealthy installation, and difficult removal) may also be used by malware whose presence becomes clear, e.g., ransomware (Section 7.7). The means by which rootkits are installed (e.g., a buffer overflow in user-space or kernel software), are often considered separate from the techniques used to remain hidden (the latter being defining characteristics). In this sense, rootkits may be viewed as "post-intrusion" tools.

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

‡**Exercise** (`lvtes`, Linux kernel backdoors). a) Summarize the technical details of the *lvtes* keylogger rootkit, which hides by modifying a module list. b) Give a technical overview of Linux kernel backdoors. (Hint: [8].)

‡**Exercise** (Stuxnet 2010). *Stuxnet*, a worm with rootkit functionality, has been described as the most elaborate malware developed as of the date it appeared. It targets only a specific industrial control system, and severely damaged Iranian nuclear enrichment centrifuges. Summarize the technical details of this worm-rootkit (hint: [17]).

‡**Exercise** (Sony rootkit 2005). Summarize the technical details and controversy of the *Sony rootkit*, related to copy protection on Sony CDs (hint: [19]).

‡**Exercise** (Compiler trap door). An attacker backdoor in the implementation of the OS **login** command grants an unauthorized username login access without entry of any password. This can be done by modifying the (source code of the) compiler that compiles this system software, building special-case logic into the `login` executable when the compiler creates it. However, this leaves visible evidence to anyone examining 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: [63]. This is Thompson's classic paper.)

‡**Exercise** (Memory isolation meltdown). *Memory isolation* is a basic protection. Ideally, the memory of each user process is isolated from others, and kernel memory is isolated from user memory. a) Explain how memory isolation is achieved on modern commodity processors (hint: [31, Sect. 2.2]). b) Summarize how the *Meltdown* attack defeats memory isolation by combining a side-channel attack with out-of-order (i.e., early) instruction execution (hint: [31]; this exploits hardware optimization, not software vulnerabilities). c) Discuss how memory isolation between user processes, and between user and kernel space, relate to principle P5 (ISOLATED-COMPARTMENTS).

7.5 Rootkit detail: installation, object modification, hijacking

This section provides additional technical details on rootkits.

HIJACKING SYSTEM CALLS. User mode programs may not directly access kernel memory. Applications access kernel resources by *system calls* (page 195 and Chapter 6), most often through OS services and shared library utilities (running in user space) that invoke the kernel services (running with kernel mode privileges). The service names are resolved to function addresses through a service address table, e.g., a table of function pointers (sometimes called a *dispatch table*). Various methods enable *system call hijacking*. A kernel rootkit may alter entries in dispatch tables, to redirect calls to rootkit code, which might call the legitimate code and then postprocess results (Fig. 7.4). 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. A second hijacking method overwrites the code implementing targeted system calls. A third alternative does not hook individual call table entries, but instead replaces an entire table by changing the address used by calling routines to find it, to that of a substitute table elsewhere.

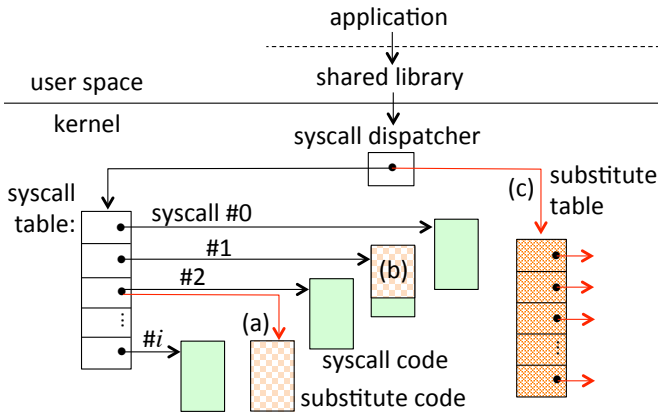

Figure 7.4: System call hijacking. (a) Hooking an individual system call; the substitute code (hook function) may do preprocessing, call the original syscall code (which returns to the substitute), and finish with postprocessing. (b) Overwriting individual system call. (c) Hooking the entire syscall table by using a substitute table.

‡**WINDOWS FUNCTION HOOKING.** On Windows systems, function pointer tables commonly hooked in kernel and user space, respectively, are: SSDT (System Service Dispatch Table) and IAT (Import Address Table). The addresses of functions in Windows shared libraries (*DLLs* or dynamically linked libraries) are made available through the IAT, the principal user-space dispatch table and thus a common target for user-space rootkits. Aside: while syscall interfaces are well documented in Unix (and typically accessed via C library wrapper functions), they are not openly documented on Windows—there, syscalls are accessed via the NTAPI/Native API provided by wrappers in the `ntdll.dll` library, and dispatched via SSDT.

INLINE HOOKING. An alternative to hijacking dispatch tables is *inline hooking*. It involves *detour patching*, using *detour* and *trampoline* functions (Fig. 7.5). This allows arbitrary-length preprocessing and postprocessing code around a target function. Dispatch table hooking can be detected by a simple cross-check of table addresses; inline hooking is not detected by that, but is visible by an integrity cross-check of the target function (hashing the executable code, and comparing to a known-good hash).

KERNEL OBJECT MODIFICATION, PRUNING REPORTS. A rootkit may hide a process, files or open network connections in two main ways:

a) *direct kernel object modification* (DKOM). Kernel data structures are directly altered, e.g., removing rootkit-related objects from a list, to go unreported. A kernel rootkit, having the ability to read and write kernel memory, may alter data structures meant for exclusive use by the kernel. For example, the privilege of any process in the kernel's list of running processes may be escalated by setting its UID to 0. Other examples of objects to modify include: the list of files in a directory, the loaded module list, the scheduler list, and the process accounting list (e.g., showing CPU usage).

b) *postprocessing results* of system calls. When a call is made requesting a report, the result can be pruned to remove rootkit-related processes and objects before it is returned. This can be done by hooking, preprocessing and postprocessing results of the legitimate system call (Figures 7.4, 7.5).

INSTALLING. Various means exist to install kernel rootkits or alter kernels.

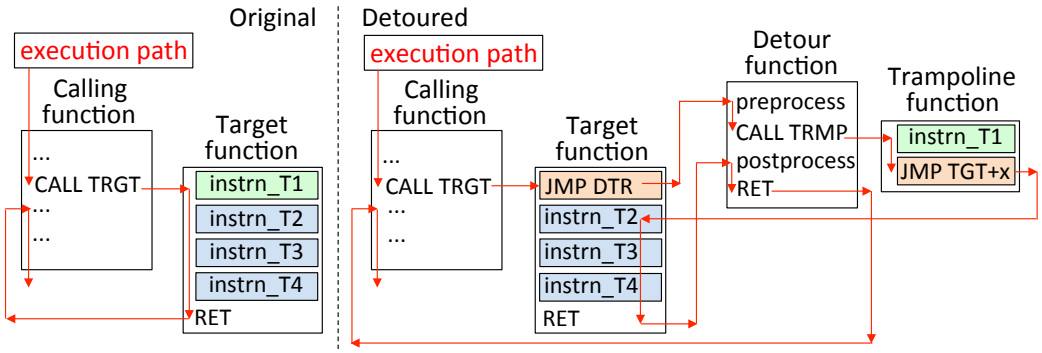

Figure 7.5: Inline hooking, detour and trampoline. A trampoline replaces the overwritten instruction, and enables the target function's return to the detour for postprocessing.

1. Standard kernel module installation. A superuser may install a supposedly valid kernel module (e.g., device driver) with Trojan rootkit functionality. Similarly, an attacker may socially engineer a superuser to load a malicious kernel module (LKM, below). A superuser cannot directly access kernel memory, but can load kernel modules.
2. Exploiting a vulnerability to kernel code—e.g., a buffer overflow in a kernel network daemon, or parsing errors in code that alters kernel parameters.
3. Modifying the boot process mechanism. For example, a rogue boot loader might alter the kernel after it is loaded, but before the kernel runs.
4. Modifying code or data swapped (paged) to disk. If kernel memory is swapped to disk, and that memory is writable by user processes, kernel integrity may be modified on reloading the swapped page. (This was done by *Blue Pill*, Section 7.9.)
5. Using interfaces to physical address space. For example, Direct Memory Access (DMA) writes may be used to alter kernel memory, through hardware devices with such access (e.g., video and sound cards, network cards, disk drives).

LOADABLE KERNEL MODULES. One method to install rootkits is through standard 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 (system calls and hardware drivers). Many kernel rootkits are LKMs. Most commercial operating systems support some form of dynamically loadable such kernel modules, 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.

‡**REVIEW: LINKING AND LOADING.** Generating an executable suitable for loading involves several steps. A *compiler* turns *source code* into *machine code*, resulting in a *binary* (i.e., *object*) file. A *linker* combines one or more object files to create an *executable* file (*program image*). A *loader* moves this from disk into the target machine's main memory, relocating addresses if necessary. *Static linkers* are compile-time tools; loaders are run-time tools, typically included in an OS kernel. Loaders that include *dynamic linkers* can load executables and link in shared libraries (DLLs on Windows).

‡**Exercise** (Modular vs. monolithic root, kernel). Modularity provided by a core kernel with loadable modules and device drivers does not provide memory isolation between distinct kernel software components, nor partition access to kernel resources. Kernel compromise still grants malware control to read or write anything in kernel memory, if the entire kernel operates in one privilege mode (vs. different hardware rings, Chapter 5). Linux *capabilities* (Chapter 6) partition superuser privileges into finer-grained privileges, albeit all in user space. Discuss how these issues relate to design principles P5 (ISOLATED-COMPARTMENTS), P6 (LEAST-PRIVILEGE), and P7 (MODULAR-DESIGN).

USER MODE ROOTKITS. On some systems including Windows, user mode rootkits operate by intercepting, in the address space of user processes, *resource enumeration APIs*. These are supported by OS functions that generate reports from secondary data structures the OS builds to efficiently answer resource-related queries. Such a rootkit filters out malware-related items before returning results. This is analogous to hooking

system calls in kernel space (without needing kernel privileges), but user mode rootkit changes made to one application do not impact other user processes (alterations to shared libraries will impact all user-space processes that use those libraries).

‡**Exercise** (User mode rootkit detection). It is easier to detect user mode rootkits than kernel rootkits. Give a high-level explanation of how user mode rootkits can be detected by a *cross-view difference* approach that compares the results returned by two API calls at different levels. (Hint: [64], which also reports that by their measurements, back in 2005 over 90% of rootkit incidents reported in industry were user mode rootkits.)

‡**Exercise** (Keyjacking). *DLL injection* and *API hooking* are software techniques with non-security uses, as well as uses in security defenses (e.g., anti-virus software) and attacks (e.g., rootkit software middle-person attacks). Explain how DLL injection is a threat to end-user private keys in client-side public-key infrastructure (hint: [34]).

‡**PROTECTING SECRETS AND LOCAL DATA.** The risk of client-side malware motivates encrypting locally stored data. *Encrypted filesystems* automatically encrypt data stored to the filesystem, 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 that are in cleartext form in main memory (RAM) are subject to compromise. For example, upon system crashes, RAM memory is often written to disk for recovery or 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: [51]).

‡**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 host misuse the HSM? (Hint: look up the *confused deputy problem*. Cf. Section 9.5.)

7.6 ‡Drive-by downloads and droppers

MALWARE EXPLOITING BROWSER USE. Malware also exploits the rich functional design of browser-server interaction. 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.[5] Such *active content*—small program snippets that 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,[6] and injection of a few

[5] Older examples of active content (Chapter 9) include Flash, ActiveX controls, and Java applets.

[6] For example, using heap spraying (Chapter 6) combined with script injection attacks (Chapter 9).

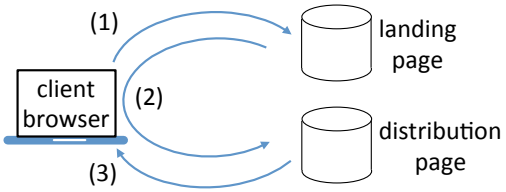

Figure 7.6: Drive-by download involving browser redirection. A browser visiting an original site (1) may be redirected (2) to a distribution site that causes silent download of malware (3), e.g., possible due to browser vulnerabilities. The redirection (2) may involve several redirect hops through intermediate sites. One defense involves detection by automated analysis (using *downloader graphs*, Section 7.9) comparing multi-step downloads that result from an initial download, to typical benign download patterns.

lines of script into a web page via a compromised or exploited server application, simply visiting a web page can result in binary executable malware being silently downloaded and run on the user device. This is called a *drive-by download* (Fig. 7.6).

MEANS OF DRIVE-BY EXPLOITATION. Drive-by downloads use several technical means as now discussed. Questions to help our understanding are: how do malicious scripts get embedded into web pages, how are malicious binaries downloaded, and why is this invisible to users? Malicious scripts are embedded 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. user-provided content reflected to others via sites (e.g., web forums) soliciting input;
4. malicious parameters as part of links (URLs) received in HTML email.

A short script can redirect a browser to load a page from an attacker site, with that page redirecting to a site that downloads binaries to the user device. Silent downloading and running of a binary should not be possible, but often is, through scripts that 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 (Fig. 7.6). Redirects generally go unnoticed, being frequent and rapid in legitimate browsing, and injected content is easy to hide, e.g., using 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—and may result in *zombies* being recruited into botnets. Rather than a separate malware category, one may view drive-by downloads as a deployment means or spreading method that exploits features of browser-server ecosystems. As a distinguishing spreading characteristic here, the victim devices visit a compromised web site in a *pull model*. (Traditional worms spread in a *push model*, with a compromised source initiating contact with next-victims.)

DROPPERS (DOWNLOADERS). A *dropper* is malware that installs (on a victim host) other malware that contains a malicious payload. If this involves downloading additional malware pieces, the dropper may be called a *downloader*. Droppers may install backdoors

(page 195) to aid installation and update. The payload may initiate network communications to a malware source or control center, or await contact. The initial malware installed, or a software package including both the dropper and its payload, may be called the *egg*. The dropper itself may arrive by any means including virus, worm, drive-by download, or user-installed Trojan horse software.

Example *(Babylonia dropper).* One of the first widely spread malware programs with dropper functionality was the *Babylonia* (1999) virus. After installation, it downloaded additional files to execute. Being non-malicious, it gained little notoriety, but its functionality moved the world a step closer to botnets (Section 7.7).

7.7 Ransomware, botnets and other beasts

Ransomware is malware with a specific motive: to extort users. This typically involves compromise of a host, 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 FILES. A powerful type of malware is that which prevents access to files (*file lockers*) by encryption. It encrypts user data files, then asks users to pay a sum of money in return for a decryption key (e.g., from a remote site) that allows file recovery. Payment is demanded in hard-to-trace, non-reversible forms such as pre-paid cash vouchers or digital currencies; the dramatic increase in ransomware in parallel with *Bitcoin* is notable. Removal of the malware itself does not solve the problem: encrypted files remain unavailable. Ransomware may be deployed by any means used for other malware, including Trojan software installed by users unwittingly or via social engineering. Best practice defenses include regular backup of all data files.

DETAILS: ASYMMETRIC FILE LOCKING. If such ransomware uses public-key cryptography, the malware need not retain a decryption key discoverable by the victim. Consider a fixed ransomware (public, private) key pair (e_r, d_r). The malware, hard-coded with public e_r, creates a per-victim random symmetric key k; uses k to symmetrically encrypt victim files; uses e_r to public-key encrypt k as ciphertext $C = E_{e_r}(k)$; then deletes from memory k and the plaintext files. A payment demand message is displayed, showing C and contact details. If C is returned with payment, the malware agent uses its externally held d_r to decrypt C recovering k, which is then made available to the victim.

RANSOMWARE (NON-ENCRYPTING). Other variations of file lockers 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 user access to OS functionality until a ransom is paid (again, e.g., in bitcoin), and disable OS debug modes (e.g., *safe mode* or safe boot). In general, ransomware may involve rootkit functionality to make removal difficult.

‡**Example** *(WannaCrypt 2017).* *WannaCry* ransomware (a *cryptoworm*) reportedly infected over 200,000 Windows computers across 150 countries. It generated a 2048-bit

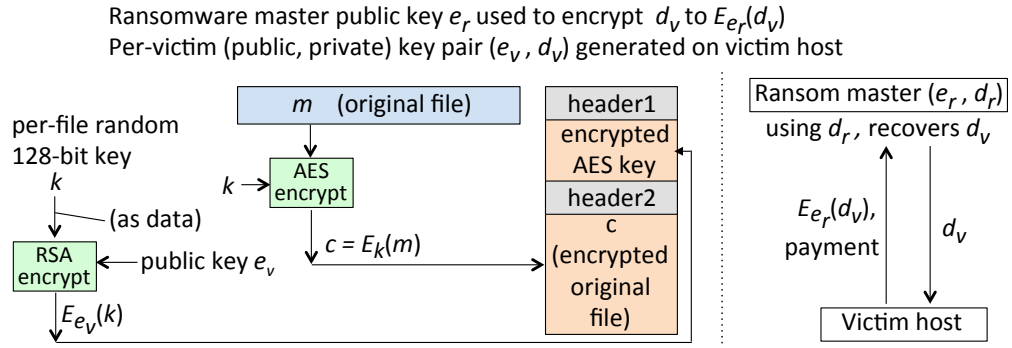

Figure 7.7: WannaCry file-locking ransomware: file structure. Each data file is AES-encrypted under a different key k. An encrypted file's `header1` includes an identifying ASCII string `WANACRY!` and AES keylength; `header2` provides a file type and byte-length of the original file. Compare to hybrid encryption, Chapter 2.

RSA key pair (e_v, d_v) for each victim (Fig. 7.7). For each file to be encrypted, a random 128-bit AES key k was generated and its public-key encryption $E_{e_v}(k)$ put in a file header followed by the ciphertext content. A hard-coded 2048-bit ransomware master public key e_r was used to encrypt one copy of the victim private key as $C_v = E_{e_r}(d_v)$, in place of C (above). This facilitates independent keys k for each file, whereas using a common key k across all files exposes k to recovery if file locking is detected before all files are locked.

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

BOTNETS AND ZOMBIES. A common goal of malware is to obtain an OS command shell interface and then arrange instructions sent to/from an external source. A payload delivering this functionality is called *shellcode* (Chapter 6). A computer that has been compromised by malware and can be remotely controlled, or that reports back in to 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*. The individual controlling it is the *botnet herder*. Botnets exceeding 100,000 machines have been observed. Owners of machines on which zombie malware runs are often unaware of this state of compromise (so perhaps the owners are the real zombies).

BOTNETS AND CRIME. Botnets play a big role in cybercrime. They provide critical mass and economy of scale to attackers. Zombies are instructed to spread further malware (increasing botnet size), carry out *distributed denial of service* attacks (Chapter 11), execute spam campaigns, and install keyloggers to collect data for credit cards and access to online bank accounts. Spam may generate revenue through sales (e.g., of pharmaceuticals), drive users to malicious web sites, and spread ransomware. 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.

BOTNET COMMUNICATION STRUCTURES AND TACTICS. A simple botnet command and control architecture involves a central administrative server in a client-server

model. Initially, control communications were over (Internet Relay Chat) *IRC channels*, allowing the herder to send one-to-many commands. Such centralized systems have a single point of failure—the central node (or a centralized communication channel), if found, can be shut down. 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 use a multi-tiered communication hierarchy in which the bot herder at the top is insulated from the zombies at the bottom by layers of intermediate or proxy communication nodes. For zombie machines that receive control information (or malware updates) by connecting to a fixed URL, one creative tactic used by bot herders is to arrange the normal DNS resolution process to resolve the URL to different IP addresses after relatively short periods of time. Such tactics complicate the reverse engineering and shutdown of botnets.

‡**Exercise** (Torpig 2006). The *Torpig* botnet involves use of a rootkit (*Mebroot*) to replace master boot records. In 2009, it was studied in detail by a research team that seized control of it for 10 days. Summarize technical details of this botnet (hint: [60]).

‡**Exercise** (Zeus 2007). Summarize technical details of *Zeus* bank Trojan/credential-stealing malware. (Hint: [4], [1]. Its control structure has evolved along with deployment related to keylogging, ransomware and botnets; source code became available in 2011.)

‡**Exercise** (Other botnets). Summarize technical details of these botnets (others are in Chapter 11): a) *Storm*, b) *Conficker*, c) *Koobface*, d) *BredoLab*, e) *ZeroAccess*.

‡**Exercise** (Botnet motivation). Discuss early motivations for botnets (hint: [5]).

ZERO-DAY EXPLOITS. A *zero-day exploit* (zero-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. The situation is 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 this buys attackers.

LOGIC BOMBS. A *logic bomb* is a sequence of instructions, often hosted in a larger program, that 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 programmatic control. From this viewpoint, essentially all malware is a form of logic bomb (often with default trigger condition `TRUE`). Thus logic bombs are

spread by any means that spreads malware (e.g., viruses, worms, drive-by downloads).

‡**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* (1965). 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 Trojan—a 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 historical examples.

7.8 Social engineering and categorizing malware

This section summarizes properties that distinguish malware categories. First, we revisit use of social engineering to trick users into installing malware, and related email abuses.

SOCIAL ENGINEERING AND USER-ENABLED INSTALLS. In contrast to silently installed malware, *social engineering* attacks may trick users into one-step download, installation and execution of malware. As an early instance, the (non-malicious) email worm-virus *Happy99* (1999) convinced users to run an attached executable. Some operating systems hide filename extensions (for user-friendly interfaces), but this aids such attacks by removing one of the few visible indicators to users, e.g., extensions such as `.exe` (executable). Note also: on a typical OS, double-clicking a file results in program execution—either running the file if it is itself executable, or for a data file, running the executable set by default as the associated program to open it—in both cases, transferring the user process' execution privileges. This is a form of user-enabled execution.

Example *(HTML email and preview panes).* Email began as text only. Over time, email clients came to support embedded HTML, allowing rich formatting and embedded images. Modern HTML supports embedded scripting, which then enabled (upon displaying email) execution of embedded scripts (e.g., JavaScript), including malware execution. The error was failure to foresee this rich functionality being exploited when combined with email's open design, which allows anyone to send email to a recipient with a known email address. Among other email clients that initially ran embedded JavaScript within HTML email was Microsoft Outlook. Simply viewing ("opening") an email message would run any embedded script, which might exploit available means to load additional resources to execute malicious programs. This was exacerbated by email preview panes (auto-preview), which display a portion of an email without a user explicitly "opening" the email; rendering this preview content would also run embedded scripts.

Today, essentially all email clients disable running of embedded scripts; embedded images (which commonly retrieve resources from external sites) are also no longer loaded by default, instead requiring an explicit click of a "load external images" button.

Example *(.*zip *files).* Filename extensions such as `.zip`, `.rar` and `.sfx` indicate a package of one or more compressed files. These may be *self-extracting executables*, containing within them scripts to uncompress, unpack, save files to disk, and begin an execution, without use of external utilities. This process may be supported depending on OS and host conventions, triggered by a double-click. If the package contains malware, on-host anti-virus (anti-malware) tools may provide protection if the unpacked software is recognizable as malicious. Few users appreciate what their double-click authorizes, and little reliable information is easily available on the scripts and executables to be executed.

Exercise (Clicking to execute). If a user interface hides filename extensions, and there is an email attachment `prettyPicture.jpg.exe`, what filename will the user see?

‡**Exercise** (Socially engineering malware installation). Consider web-based malware installation through social engineering. Summarize tactics for: (a) gaining user attention; and (b) deception and persuasion. (Hint: [41].)

‡**Exercise** (Design principles). Consider security design principles P1 (SIMPLICITY-AND-NECESSITY), P2 (SAFE-DEFAULTS), P5 (ISOLATED-COMPARTMENTS), P6 (LEAST-PRIVILEGE), P10 (LEAST-SURPRISE). Discuss how they relate to malware in the above examples of: (a) HTML email and auto-preview; and (b) self-extracting executables.

MALWARE CLASSIFICATION BY OBJECTIVES. One way to categorize malware is to consider its underlying goals. These may include the following.

1. *Damage to host 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. *Data theft.* Documents stolen may be corporate strategy files, intellectual property, credit card details, or personal data. Credentials stolen, e.g., account passwords or crypto keys, may allow fraudulent account login, including to online banking or enterprise accounts; or be 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 card risks include deceiving users into purchasing 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 (above).
4. *Ongoing surveillance.* User voice, camera video, and screen actions may be recorded surreptitiously, by microphones and web cameras on mobile and desktop devices, or by software that records web sites visited, keystrokes and mouse movements.
5. *Spread of malware.* Compromised machines may be used to further spread malware.
6. *Control of resources.* Once a machine is compromised, code may be installed for later execution or backdoor access. Remote use is made of computing cycles and communication resources for purposes including botnet service, *bitcoin mining*, as a host server for *phishing*, or as a *stepping stone* for further attacks (reducing risk that

Category name	Property (blank denotes: no)			
	BREEDS†	HOSTED	STEALTHY	VECTOR
virus	✓	✓		U
worm	✓			N
Trojan horse		✓	✓	E or S
backdoor		maybe	✓	T or S
rootkit, keylogger			✓	T or S
ransomware				T
drive-by download	★		✓	S

Table 7.2: Malware categories and properties. Botnets (unlisted), rather than a separate malware category, control other (zombie) malware. Codes for infection vector: U (user-enabled), N (network service vulnerability), E (social engineering), T (intruder, including when already resident malware or dropper installs further malware), S (insider, e.g., developer, administrator, compromised web site hosting malware). †Any category that breeds may spread a dropper (Section 7.6) for other types, e.g., rootkits, ransomware. ★The number of download sites does not increase, but site visits propagate malware.

an attack is traced back to the originating agent). Zombies enlisted to send spam are called *spambots*; those in a DDoS botnet are *DDoS zombies*.

MALWARE CLASSIFICATION BY TECHNICAL PROPERTIES. Another way to categorize malware is by technical characteristics. The following questions guide us.

a) Does it breed (self-replicate)? 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 that characterizes them.

b) Does it require a host program, as a parasite does?

c) Is it covert (stealthy), taking measures to evade detection and hide its functionality?

d) By what vector does infection occur? 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)?

e) Does it enlist the aid of an *insider* (with privileges beyond that of an external party)?

f) Is it transient (e.g., active content in HTML pages) or persistent (e.g., on startup)?

Table 7.2 summarizes many of these issues, to close the chapter.

7.9 ‡End notes and further reading

For malware, Szor [62] is recommended for in-depth discussion, and Aycock [3] for a concise introduction. Nachenberg [40] gives an overview of virus detection in practice. Curry's Unix security book [12] opens with a summary of early malware incidents. Denning's early collection [13] on malware includes a virus primer by Spafford [57] and an

overview by Cohen [9]. Our non-existence proof (Section 7.2) is from Cohen's book [10] based on one-day short courses. Ludwig's earlier book [32] includes assembler, with a free online electronic edition. See Duff [15] for early Unix viruses (cf. McIlroy [36]). Other sources of information about malware include the U.S. *NVD* (National Vulnerability Database) [42], the related *CVE* list (Common Vulnerabilities and Exposures) [38], the Common Weakness Enumeration (*CWE*) dictionary of software weakness types [39], and the SecurityFocus vulnerability database [50]. The industry-led Common Vulnerability Scoring System (*CVSS*) rates the severity of security vulnerabilities.

Kong [28] 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 [20] and Kasslin [24]. *The Shellcoder's Handbook* [2] details techniques for running attacker-chosen code on victim machines, noting "The bad guys already know this stuff; the network-auditing, software-writing, and network-managing public should know it too"; similarly see Stuttard [61] and McClure [35]. Many of these attacks exploit the mixing of code and data, including to manipulate code indirectly (vs. overwriting code pointers to alter control flow directly). For greater focus on the defender, see Skoudis and Zeltser [55], Skoudis and Liston [54], and (emphasizing reverse engineering) Peikari [43]. Tracking an intruder differs from addressing malware—see Stoll [59].

Staniford [58] analyzes the spread of worms (e.g., *Code Red*, *Nimda*) and ideas for *flash worms*. See Hunt [22] for the Detour tool, *DLL interception* (benign) and *trampolines* to instrument and functionally extend Windows binaries. For a gentle introduction to user mode and kernel rootkit techniques and detection, see Garcia [6]. To detect *user mode rootkits*, see Wang [64]. Jaeger [23] discusses hardening kernels against rootkit-related malware that abuses standard means to modify kernel code. For hardware-based virtual machines (HVMs), virtual machine monitors and *hypervisor rootkits* (including discussion of *Blue Pill* [48]), see *SubVirt* [27] and Desnos [14]. For *drive-by downloads* see Provos [44, 45]. For related studies of *droppers* and detecting them via analysis of downloader graphs, see Kwon [30]. For the underground economy business model of distributing malware on a *pay-per-install* basis and the resulting distribution structure, see Caballero [7]. In 1996, Young [65, 66] explained how public-key cryptography strengthened a reversible denial of service attack called *cryptovirology* (now *ransomware*). For defenses against file-encrypting ransomware, see Scaife [49] and UNVEIL [25]; for static analysis of *WannaCry*, see Hsiao [21]. On botnets aside from the exercises in Section 7.7, see Cooke [11] for an introduction, Shin [53] for *Conficker*, and BotMiner [18] for detection.

Code signing of applications and OS code, using dedicated *code signing certificates*, is a defense against running unauthorized programs. For an overview of Windows Authenticode, requirements for signing user and kernel mode drivers, and abuses, see Kotzias [29] and Kim [26]. For a history of Linux kernel module signing, see Shapiro [52]. Meijer [37] explains severe vulnerabilities in commodity *hardware disk encryption*.

References (Chapter 7)

[1] D. Andriesse, C. Rossow, B. Stone-Gross, D. Plohmann, and H. Bos. Highly resilient peer-to-peer botnets are here: An analysis of Gameover Zeus. In *Malicious and Unwanted Software (MALWARE)*, pages 116–123, 2013.

[2] C. Anley, J. Heasman, F. Lindner, and G. Richarte. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes (2nd edition)*. Wiley, 2007.

[3] J. Aycock. *Computer Viruses and Malware*. Springer Science+Business Media, 2006.

[4] H. Binsalleeh, T. Ormerod, A. Boukhtouta, P. Sinha, A. M. Youssef, M. Debbabi, and L. Wang. On the analysis of the Zeus botnet crimeware toolkit. In *Privacy, Security and Trust (PST)*, pages 31–38, 2010.

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

[6] P. Bravo and D. F. Garcia. Rootkits Survey: A concealment story. Manuscript, 2009, `https://yandroskaos.github.io/files/survey.pdf`.

[7] J. Caballero, C. Grier, C. Kreibich, and V. Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *USENIX Security*, 2011. See also K. Thomas et al., *USENIX Security*, 2016.

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

[9] F. Cohen. Implications of computer viruses and current methods of defense. Article 22, pages 381–406, in [13], 1990. Updates earlier version in *Computers and Security*, 1988.

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

[11] E. Cooke and F. Jahanian. The zombie roundup: Understanding, detecting, and disrupting botnets. In *Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, 2005.

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

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

[14] A. Desnos, E. Filiol, and I. Lefou. Detecting (and creating!) an HVM rootkit (aka BluePill-like). *J. Computer Virology*, 7(1):23–49, 2011.

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

[16] 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.

[17] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier. Report, ver. 1.4, 69 pages, Symantec Security Response, Cupertino, CA, February 2011.

[18] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *USENIX Security*, pages 139–154, 2008.

[19] J. A. Halderman and E. W. Felten. Lessons from the Sony CD DRM episode. In *USENIX Security*, 2006.

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

[21] S.-C. Hsiao and D.-Y. Kao. The static analysis of WannaCry ransomware. In *Int'l Conf. Adv. Comm. Technology (ICACT)*, pages 153–158, 2018.

[22] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *3rd USENIX Windows NT Symp.*, 1999.

[23] 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.

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

[25] A. Kharraz, S. Arshad, C. Mulliner, W. K. Robertson, and E. Kirda. UNVEIL: A large-scale, automated approach to detecting ransomware. In *USENIX Security*, pages 757–772, 2016.

[26] D. Kim, B. J. Kwon, and T. Dumitras. Certified malware: Measuring breaches of trust in the Windows code-signing PKI. In *ACM Comp. & Comm. Security (CCS)*, pages 1435–1448, 2017.

[27] 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, 2006.

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

[29] P. Kotzias, S. Matic, R. Rivera, and J. Caballero. Certified PUP: Abuse in Authenticode code signing. In *ACM Comp. & Comm. Security (CCS)*, pages 465–478, 2015.

[30] B. J. Kwon, J. Mondal, J. Jang, L. Bilge, and T. Dumitras. The dropper effect: Insights into malware distribution with downloader graph analytics. In *ACM Comp. & Comm. Security (CCS)*, 2015.

[31] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, pages 973–990, 2018. See also "Spectre Attacks", Kocher et al., *IEEE Symp.* 2019.

[32] M. Ludwig. *The Little Black Book of Computer Viruses*. American Eagle Publications, 1990. A relatively early exposition on programming computer viruses, with complete virus code; the 1996 electronic edition was made openly available online.

[33] J. Ma, G. M. Voelker, and S. Savage. Self-stopping worms. In *ACM Workshop on Rapid Malcode (WORM)*, pages 12–21, 2005.

[34] J. Marchesini, S. W. Smith, and M. Zhao. Keyjacking: The surprising insecurity of client-side SSL. *Computers & Security*, 24(2):109–123, 2005.

[35] S. McClure, J. Scambray, and G. Kurtz. *Hacking Exposed 6: Network Security Secrets and Solutions (6th edition)*. McGraw-Hill, 2009.

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

[37] C. Meijer and B. van Gastel. Self-encrypting deception: Weaknesses in the encryption of solid state drives. In *IEEE Symp. Security and Privacy*, 2019.

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

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

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

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

[42] NIST. National Vulnerability Database. U.S. Dept. of Commerce. `https://nvd.nist.gov/`.

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

[44] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iFRAMEs point to us. In *USENIX Security*, 2008.

[45] 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.

[46] J. A. Rochlis and M. W. Eichin. With microscope and tweezers: The Worm from MIT's perspective. *Comm. ACM*, 32(6):689–698, 1989. Reprinted as [13, Article 11]; see also more technical paper [16].

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

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

[49] N. Scaife, H. Carter, P. Traynor, and K. R. B. Butler. CryptoLock (and Drop It): Stopping ransomware attacks on user data. In *IEEE Int'l Conf. Distributed Computing Systems*, pages 303–312, 2016.

[50] SecurityFocus. Vulnerability Database. http://www.securityfocus.com/vulnerabilities, Symantec.

[51] A. Shamir and N. van Someren. Playing "hide and seek" with stored keys. In *Financial Crypto*, pages 118–124, 1999. Springer LNCS 1648.

[52] R. Shapiro. A History of Linux Kernel Module Signing. https://cs.dartmouth.edu/~bx/blog/2015/10/02/a-history-of-linux-kernel-module-signing.html, 2015 (Shmoocon 2014 talk).

[53] S. Shin and G. Gu. Conficker and beyond: A large-scale empirical study. In *Annual Computer Security Applications Conf. (ACSAC)*, pages 151–160, 2010. Journal version: *IEEE TIFS*, 2012.

[54] E. Skoudis and T. Liston. *Counter Hack Reloaded: A Step-by-Step Guide to Computer Attacks and Effective Defenses (2nd edition)*. Prentice Hall, 2006 (first edition: 2001).

[55] E. Skoudis and L. Zeltser. *Malware: Fighting Malicious Code*. Prentice Hall, 2003.

[56] E. H. Spafford. Crisis and aftermath. *Comm. ACM*, 32(6):678–687, 1989. Reprinted: [13, Article 12].

[57] E. H. Spafford, K. A. Heaphy, and D. J. Ferbrache. A computer virus primer. Article 20, pages 316–355, in [13], 1990.

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

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

[60] 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 Comp. & Comm. Security (CCS)*, pages 635–647, 2009. Shorter version: *IEEE Security & Privacy* 9(1):64–72, 2011.

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

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

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

[64] Y. Wang and D. Beck. Fast user-mode rootkit scanner for the enterprise. In *Large Installation Sys. Admin. Conf. (LISA)*, pages 23–30. USENIX, 2005.

[65] A. L. Young and M. Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *IEEE Symp. Security and Privacy*, pages 129–140, 1996.

[66] A. L. Young and M. Yung. On ransomware and envisioning the enemy of tomorrow. *IEEE Computer*, 50(11):82–85, 2017. See also same authors: "Cryptovirology", *Comm. ACM* 60(7):24–26, 2017.