Basic Concepts and Models of Cybersecurity

Chapter · February 2020

DOI: 10.1007/978-3-030-29053-5_2

2 authors, including:

Dominik Herrmann
Otto-Friedrich-Universität Bamberg
79 PUBLICATIONS **887** CITATIONS

SEE PROFILE

Chapter 2
Basic Concepts and Models of Cybersecurity

Dominik Herrmann and Henning Pridöhl

Abstract This introductory chapter reviews the fundamental concepts of cybersecurity. It begins with common threats to information and systems to illustrate how matters of security can be addressed with methods from risk management. In the following, typical attack strategies and principles for defence are reviewed, followed by cryptographic techniques, malware and two common weaknesses in software: buffer overflows and SQL injections. Subsequently, selected topics from network security, namely reconnaissance, firewalls, Denial of Service attacks, and Network Intrusion Detection Systems, are analysed. Finally, the chapter reviews techniques for continuous testing, stressing the need for a free distribution of dual-use tools. Although introductory in nature, this chapter already addresses a number of ethical issues. For instance, well-intended security mechanisms may have undesired side effects such as leaking sensitive information to attackers. As asymmetries and externalities are at the core of many security problems, devising effective security solutions that are adopted in practice is a challenge.

Keywords Advanced persistent threat · Availability · Black hats · Certificates · Confidentiality · Cryptography · Integrity · Malware · Supply-chain attack · Vulnerabilities · White hats

2.1 Introduction

Honesty was never a given in human history. In the physical world, we can rely on decades of experience to defend against malicious actors. We have devised sophisticated laws that govern what is acceptable and what is illegal. In addition, we have a number of technical means at our disposal to secure our property and our secrets.

D. Herrmann (✉) · H. Pridöhl
Privacy and Security in Information Systems Group (PSI), University of Bamberg, Bamberg, Germany
e-mail: dominik.herrmann@uni-bamberg.de; henning.pridoehl@uni-bamberg.de

M. Christen et al. (eds.), *The Ethics of Cybersecurity*, The International Library of Ethics, Law and Technology 21,
https://doi.org/10.1007/978-3-030-29053-5_2

However, we are still in the process of learning how to secure cyberspace. Cyberspace has become the handle of choice to refer to the virtual world created by networked computer systems that affect large parts of our lives; securing it is challenging. According to Bruce Schneier "complexity is the enemy of security" (Chan 2012). There are not only more devices hooked up to the Internet, but also more manufacturers building them, which increases both the size and diversity of the systems forming the cyberspace and thus the probability of failures.

Moreover, cybersecurity is subject to significant asymmetries. Attackers can choose from a large variety of approaches, while defenders have to pay attention to every detail and be prepared for anything at any time. Therefore, successful attacks are not necessarily the result of negligence. Sometimes security controls are in place but are not used properly, for instance, because they conflict with the needs of users. Given these difficulties, there is now much interest in reactive security, which embraces the insight that we cannot prevent all attacks.

In this chapter, we introduce the basic concepts of cybersecurity. We start by defining common threats in Sect. 2.2 and reviewing typical attack and defence techniques in Sect. 2.3. Subsequently, we present security fundamentals in various domains, namely cryptography for data security in Sect. 2.4, malware in Sect. 2.5, software security in Sect. 2.6 and network security in Sect. 2.7. Finally, we stress the importance of continuous testing in Sect. 2.8 before we conclude the chapter in Sect. 2.9.

2.2 Threats

Before we can discuss attacks and defences in cyberspace, we must clarify what is at stake. In the following, we review the fundamental protection goals that help us gain a comprehensive picture of all aspects of security.

Before the term 'cybersecurity' became fashionable, discussions focused on computer security. The goal of computer security is to protect assets. Valuable assets can be hardware (e.g. computers and smartphones), software and data. These assets are subject to threats that may result in loss or harm.

Computer security consists of information security and systems security. It is instructive to consider the foundations of these two fields, which laid the ground for cybersecurity. Information security is concerned with the protection of data (potentially processed by computers) and any information derived from its interpretation. In systems security, we aim to ensure that (computer) systems operate as designed; i.e. attackers cannot tamper with them.

2.2.1 Information Security

We begin our discussion of threats with information security. There are three protection goals in information security: confidentiality, integrity and availability (Anderson 1972; Voydock and Kent 1983), commonly referred to as the 'CIA triad' (the origin of this abbreviation is unknown). Security measures have the purpose of addressing one or more of these objectives, as follows:

- Confidentiality: prevent unauthorised information gain.
- Integrity: prevent or detect unauthorised modification of data.
- Availability: prevent unauthorised deletion or disruption.

These protection goals apply both to data at rest, i.e. stored on a computer or on paper, and to data in transit, i.e. when data is sent over a network. The definitions refer to 'unauthorised' activities, which implies that there is an understanding about which actors are supposed to be allowed to interact with the data.

In some scenarios, there is only one authorised actor. An example in the context of the protection goal confidentiality is a smartphone or a computer with encrypted storage (sometimes called 'full-disk encryption'). In this case, only the owner of the device is authorised. An example for the goal availability is to backup data so that it remains accessible when a machine fails.

Most of the time, there are several authorised actors; often there are precisely two. For instance, the protection goal confidentiality may be relevant when a sender sends an e-mail to a particular recipient. Confidentiality is also essential during online banking. Here, we also want integrity protection for the exchanged messages to avoid transactions being modified.

The three fundamental protection goals of confidentiality, integrity and availability refer to the content. Besides content, we may also be concerned with the identity of other actors. For instance, we would like to know when the sender of an e-mail message has been forged. The protection goal *authenticity* prevents actors from impersonating someone else, usually by providing others with a means to verify a claimed identity. A related and even stronger protection goal is *non-repudiation*, which prevents actors from denying that they carried out a particular act, for instance, sending a message. Authenticity and non-repudiation are necessary to hold actors accountable (Gollmann 2011: 38).

2.2.2 Systems Security

How should we design systems so that they provide security for data stored on them? This question is at the centre of systems security. Consequently, the protection goals that are pursued in systems security are the same ones as in information security.

Often there are multiple ways to achieve the desired goal. For instance, confidentiality can be achieved by encrypting data or by a combination of authentication (e.g. by requiring users to enter a password) and access control (rules that govern which user is allowed to access which particular files). Designing systems that use a suitable combination of security measures is a non-trivial task.

However, systems security is not limited to achieving information security. Some systems hold no particularly interesting data at all. However, we rely on them and their functionality, i.e. the proper flow of a process. For instance, if an authentication system component of an operating system contains a bug, attackers may be able to shut it down (preventing authorised users from controlling the server) or bypass it (allowing unauthorised users to control the server). Integrity and availability are common protection goals in systems security. Keeping a particular procedure confidential may be a goal to secure intellectual property. However, it is considered bad practice to hide how a system works for reasons of security (cf. Sect. 2.3.2).

Of particular interest in systems security are so-called *cyber-physical systems* that affect the real world, such as traffic lights, autopilots, industrial robots, and control systems for chemical processes or power plants. Some of these systems are considered critical infrastructures; i.e. failures may have a significant impact on society. Policy makers are concerned that future wars might be fought by attacking critical infrastructures to cause chaos—without having to use physical force (Wheeler 2018). Well-known attacks on cyber-physical systems include the Stuxnet malware, which was used to sabotage an Iranian uranium enrichment facility at Natanz in 2010 (Langner 2013) and an attack on a Ukrainian power plant in 2015 (Zetter 2016).

2.2.3 Security Versus Safety

The cybersecurity community differentiates between security and safety (cf. Fig. 2.1). Harm can be caused by humans or by nonhuman events (Pfleeger et al. 2015). Examples of nonhuman events are natural disasters such as earthquakes, fires, floods, loss of electrical power, faults of hard disks and so on. Human threats are either benign or malicious. Benign threats are the result of accidents and inadvertent human errors such as mistyping a command, whereas malicious acts result from bad intentions.

Ensuring that a system remains operational during natural disasters and when faced with human errors (i.e. benign threats) is a matter of *safety*. Safety is crucial in cyber-physical systems, where the failure of a system may harm humans. Safety has a long tradition in engineering, for instance, in cars and airplanes that contain many critical systems designed for maximum dependability.

In contrast, matters of *security* focus on malicious acts of humans, which are called attacks. There are random attacks and directed attacks. In random attacks, attackers do not care who they attack as long as there is something to gain from the victim (cf. pickpockets in the physical world). In the electronic domain, phishing

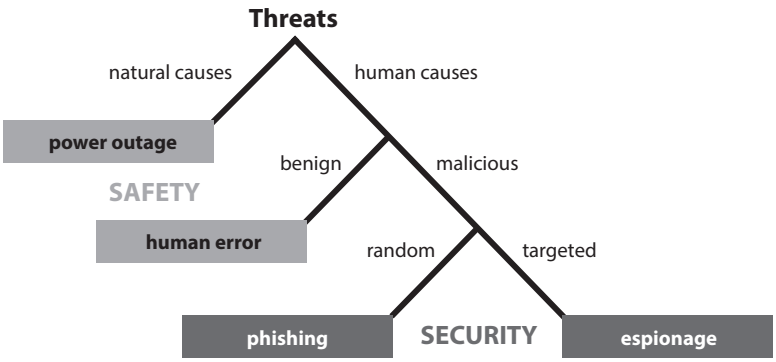

Fig. 2.1 Safety versus security

scams are a well-known example. In contrast, targeted attacks are directed at a particular victim. Targeted attacks are more difficult to defend against than random attacks because attackers act strategically, i.e. they may dynamically change their course of actions in response to security measures.

2.2.4 Security as Risk Management

Building software and hardware are complex and error-prone tasks. On average, every 1000 lines of code contain three to 20 bugs, and even a thorough code review reduces this number only by one order of magnitude (McConnell 2004). There are various ways in which these bugs can affect the security of a system. The 'Common Weakness Enumeration' (https://cwe.mitre.org) is a community-developed list of weaknesses. *Weaknesses* are generic types of mistakes that occur frequently. We discuss two common weaknesses in more detail later on, namely buffer overflows (see Sect. 2.6.1) and SQL injections (see Sect. 2.6.2).

A concrete realisation of a weakness in a particular product is called a *vulnerability*. A vulnerability is "a flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy" (Shirey 2007). Vulnerabilities in widely deployed products are assigned a unique identifier and archived in the 'Common Vulnerabilities and Exposures' (https://cve.mitre.org), which contained more than 115,000 entries in June 2019.

An attack on a system is possible if a system is exposed to an attacker and if it contains weaknesses that can be exploited. Unreachable systems cannot be attacked, and the mere presence of, e.g. a buffer overflow in a program, does not necessarily mean that it is exploitable. Furthermore, the fact that a system exposes an exploitable vulnerability does not mean that an attack is inevitable. The notion of *risk* captures this uncertainty. The severity of a risk is the product of the impact of an attack on an asset (typically concerning monetary loss) and the likelihood that the attack takes place. The likelihood of an attack depends on exposure and exploitability but also on

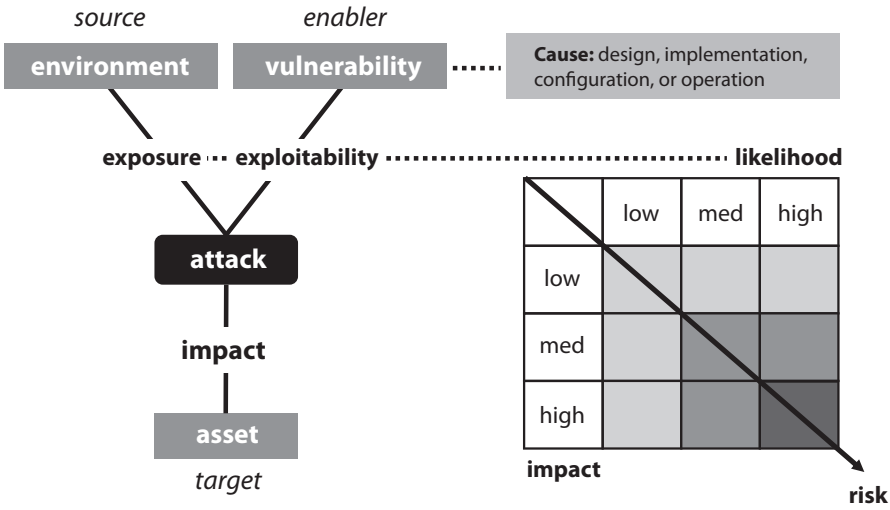

Fig. 2.2 Relationship between vulnerability and risk

the question of whether the attack has the desired impact to reach the goal of an adversary. In practice, it is difficult to predict impact and likelihood accurately. Figure 2.2 illustrates the relationship between risk and vulnerabilities.

There are various ways to handle risks (Shostack 2014). Firstly, risks can be avoided, e.g. by refraining from implementing a feature. Secondly, risks can also be mitigated, e.g. by implementing security controls (also called countermeasures) that decrease the likelihood and impact of a risk. Thirdly, risks can be transferred, e.g. by buying insurance that covers potential losses. Fourthly, risks can be accepted, i.e. by deciding to cover the costs of an attack. Acceptance may make sense for risks that are very unlikely.

In practice, system designers often try to transfer risks to the users of a system, creating a so-called *negative externality*. Transferring risks is feasible because of an asymmetric power ratio between system designers and users. This situation is problematic because operators of a system may have less incentive to take security seriously when the impact of attacks does not affect them but someone else.

2.3 Approaches for Attack and Defence

For an attack to succeed, an attacker needs a working method, an opportunity to attack and a motive (Pfleeger et al. 2015). It is therefore instructive to survey different types of attackers and attack techniques.

2.3.1 Attackers and Their Motives

What kind of attackers exist and what are their motives? In most cases, the same as in the physical world. For instance, corporate spies carry out cyber-attacks on organisations to obtain trade secrets. There are also cyber criminals, individuals or groups that seek financial gain. One of their methods of operation is holding their victims to ransom, either by installing ransomware on their machines, by threatening to release sensitive information, or by threatening to carry out a Denial of Service attack (cf. Sect. 2.7.3). The most advanced attackers are nation states that, for example, aim to influence politics in a counterpart or extend their power. Nation states can conduct very sophisticated attacks that require many financial resources. Many attacks by nation states reach the level of an *advanced persistent threat (APT)*, i.e. an attack that involves advanced techniques that allow an attacker to covertly compromise and potentially even control the systems of a victim for long periods of time.

Besides these 'professional' attackers, there are also hobbyists. The term 'script kiddies' refers to unskilled attackers that are only able to use ready-to-run tools for their attacks (see also Chap. 9). Moreover, there are hacktivists that perform attacks to further a cause and create publicity, e.g. free speech and anti-surveillance. Finally, there are rogue hackers that mostly attack systems out of curiosity. There are also hackers that attack for personal gain. They make fun of their victims by defacing their websites, brag about their abilities in their community and may even sell off sensitive data on the black market.

The term 'black hats' is used for attackers with malicious motives. In contrast, 'white-hat hackers' are interested in improving overall security. They report all discovered vulnerabilities to the respective system operators.

Many efforts aim to keep attackers 'out'. This practice neglects insiders that have much better opportunities to attack than outsiders do. Insiders may be disloyal employees (users or operators) in a particular organisation. A comprehensive view of insiders should also include all employees that work at vendors, i.e. suppliers that provide tools used within an organisation. There have been several attempts to attack high-profile targets by infecting their vendors with malware. This approach, which is called a *supply-chain attack*, is quite powerful and difficult to detect (Korolov 2018).

2.3.2 Defences

Most defences focus on proactive security. However, this is not sufficient because it is impossible to prevent all attacks with absolute certainty. Proactive techniques are therefore combined with reactive techniques to handle the residual risk. In total, there are six approaches to secure a system (Pfleeger et al. 2015: Section 2.1.5). We begin by describing the three proactive approaches.

– *Preventive controls* ensure that an attack against a target is not possible or not successful, e.g. by controlling exposure (e.g. by a firewall) or exploitability (e.g. by fixing a buffer overflow vulnerability).
– *Deterrence* merely increases the effort for an adversary, aiming to make the target unattractive. An example of a deterrence control is two-factor authentication, which requires additional proofs of identity (e.g. possession of a particular smartphone) besides knowing the correct password. Determined adversaries may still succeed if they can get access to the second authentication factor.
– In *deflection*, the goal of the defender is to redirect the efforts of an adversary to another target. Deflection can be achieved, for instance, by deploying *honeypots* within an organisation (Spitzner 2002). A honeypot is a non-production system that is intentionally set up to fool attackers. Adversaries cannot easily distinguish honeypots from production systems, and they are configured to look like attractive targets.

The next three approaches provide reactive security:

– *Detection controls* can focus either on real-time notifications or on documentation. Intrusion Detection Systems such as Snort (https://www.snort.org) can alert operators about suspicious network traffic in real time so that system administrators can thwart an ongoing attack. In contrast, logging solutions collect evidence that may become useful during a so-called 'post-mortem analysis' of a security incident. Logs may contain network traffic (often stored in the so-called NetFlow format that includes metadata but not the content of communication), user interactions, executed programs, modified files, and any other pieces of information that may be useful to track down the perpetrators ('attribution'). Post-mortem analysis may also be capable of figuring out the extent of the attack, i.e. what files and systems have been compromised.
– *Mitigation controls* reduce the impact of an attack. A frequently deployed mitigation control is network segmentation, which prevents machines located in different parts of a corporate network from communicating with each other. Thus, an adversary who has compromised the workstation of an employee in the human resources department cannot steal blueprints that are only accessible by members of the research department.
– *Recovery controls* help to revert the effects of an attack as fast as possible and to resume normal operation. Recovery measures include off-site backups as well as emergency playbooks that offer guidance during a crisis.

Typically, organisations will combine various techniques from the six categories. Ideally, they prevent the majority of the attempted attacks. The remaining attacks will then hopefully be detected and handled with reactive security techniques.

Saltzer and Schroeder (1975) have devised generic *Security Design Principles* for building secure systems. Over time, the principles have been refined (Smith 2012). We summarise them in the following:

– *Continuous improvement.* Security is a process and operators have to make changes to keep it secure on a continuous basis.

- *Least privilege.* Users and components should not have more access rights than necessary to carry out their tasks.
- *Defence in depth.* A single security mechanism should not be relied upon. Instead, multiple mechanisms should be used simultaneously, increasing the effort for adversaries.
- *Open design.* Mechanisms should not rely on the fact that adversaries do not know their design (no 'security by obscurity').
- *Chain of control.* Only trustworthy software should be executed whenever possible and non-trustworthy components should be restricted.
- *Deny by default.* Unless explicitly specified, no access should be granted.
- *Transitive trust.* If A trusts B and B trusts C, then A may also trust C.
- *Trust but verify.* Even if a component is trustworthy, its identity must be verified.
- *Separation of duty.* Critical tasks should be split up and delegated to separate components or individuals.
- *The principle of least astonishment.* Good usability of security mechanisms is essential; mechanisms should be comprehensible and consequences should be intuitive.

2.3.3 Stages of an Intrusion

We now consider a typical workflow during an attack by discussing the *Cyber Kill Chain*, a popular framework proposed by Lockheed Martin (Hutchins et al. 2011). It separates the actions of attackers that attempt to 'hack' into a secured network:

1. *Reconnaissance*: Research, identification and selection of targets, e.g. by crawling websites for e-mail addresses, social relationships, or information on specific technologies in use by the target.
2. *Weaponisation*: Coupling a remote access Trojan with an exploit into a deliverable payload. Typically, client application data files such as the Portable Document Format (PDF) or Microsoft Office documents serve as the weaponised deliverable.
3. *Delivery*: Transmission of the weapon to the targeted environment. Prevalent delivery vectors for weaponised payloads are e-mail attachments, websites and removable media such as USB sticks.
4. *Exploitation*: After the weapon is delivered to the target host, the malicious code of the attacker is triggered, either by exploiting an application or operating system vulnerability (such as a buffer overflow), by convincing users to click on an e-mail attachment or by leveraging operating system features that execute code automatically (e.g. 'autorun.inf' in Windows).
5. *Installation*: Installation of a remote access tool on the target system, which allows the adversary to maintain persistence inside the environment.

6. *Command and Control (C&C)*: Typically, compromised hosts connect outbound to a controller server on the Internet. Once the C&C channel is established, intruders have 'hands on the keyboard' access inside the target environment.
7. *Actions*: After progressing through the first six phases, intruders can take actions to achieve their original objectives, e.g. data exfiltration, which involves collecting and extracting information from the victim environment.

The Cyber Kill Chain has been adopted by many practitioners to reason about security architectures. However, this framework is also subject to criticism (Engel 2014; Sheridan 2018). The Cyber Kill Chain has been proposed at a time when security focused on prevention. Reactive security measures were mostly non-existent at that time. Once attackers had breached the firewall, they could often move around the network without much restriction. Nowadays, many networks implement the principles of least privilege, separation of duties, and defence in depth. As a result, lateral movement becomes noisier, which gives defenders more chances to detect attackers.

Moreover, the Cyber Kill Chain focuses on attacks that involve running malware (cf. Sect. 2.5) on the machines of users that work inside the infrastructure of a victim. Not all attacks require all the steps mentioned above. For instance, sensitive data stored on an improperly secured web server may be exfiltrated with a single request exploiting an SQL injection vulnerability (cf. Sect. 2.6.2).

2.4 Threats and Solutions in Data Security

Storing and transmitting data is at the core of many computing tasks. Adversaries may interfere either with 'data at rest' (stored on a system) or 'data in transit'. In this section, we review common attacks on data and introduce the concepts of cryptographic countermeasures. Our discussion focuses on data in transit, using a simplistic model that consists of a sender and a recipient of messages.

2.4.1 Unauthorised Disclosure of Information

We begin with attacks on confidentiality, which means we consider adversaries that are interested in learning secrets. Obtaining data at rest, e.g. on the system of the sender or the receiver, will generally require attackers to intrude into a system (cf. Sect. 2.3.3). In contrast, data in transit can be obtained more stealthily by eavesdropping on the transmission. Eavesdropping is possible in many distributed systems that consist of multiple components, which communicate over public networks. Attackers that control intermediary systems (such as routers or Wi-Fi access points) that are used to forward traffic between sender and receiver have access to all exchanged messages. Eavesdropping is also possible in case of wireless

communication if the attacker is close enough to the communicating parties. Eavesdroppers are said to be passive attackers because they do not interfere with transmissions.

The standard countermeasure to prevent attacks on confidentiality is to encrypt data. A prerequisite for encrypted communication is for the sender and recipient to establish a *cryptographic key*, often just a sufficiently large number of random bits. In the case of *symmetric* cryptography, sender and receiver use the same key. The key has to be exchanged 'out of band', i.e. over a channel that is not under the control of the considered adversaries.

The sender feeds a message together with the key to an encryption function, obtaining the encrypted text (ciphertext) of the message. The recipient decrypts the ciphertext by supplying it along with the same key to the decryption function. An eavesdropper would have to guess the key by attempting all possible combinations. For a popular key size such as 256 bits, this would require $2^{256} \approx 10^{77}$ trials. Equipped with one million computers, each of which being capable of trying out one billion keys per second, an adversary would still need more than 1054 years on average to complete such a task.

Note that encryption is typically only applied to the content of messages, i.e. the identities of sender and recipient are transmitted in the clear. Routers need these addresses to forward a message towards its destination. This fact allows eavesdroppers to perform traffic analysis attacks: Adversaries still learn who communicates with whom, at what time, and how often. Traffic analysis attempts can be made more difficult by using multiple layers of encryption and forwarding messages over additional nodes to obfuscate their route. The Tor network (http://torproject.org) is a practical system that uses these techniques.

2.4.2 Unauthorised Modification and Fabrication

In the following, we discuss attacks on integrity by active attackers. Common objectives include modifying messages exchanged between the sender and recipient or sending faked messages to the recipient.

For technical reasons not elaborated here, merely using encryption is not sufficient to prevent modification of the underlying plaintext. Therefore, even encrypted messages need additional integrity protection. A basic integrity protection technique works as follows: the sender supplies the message (its content and possibly also the sender and receiver addresses) along with a cryptographic key (which has to be exchanged out of band, like before) into a function that generates a *message authentication code (MAC)*. The MAC is sent to the recipient together with the message. The recipient feeds the message, the key, and the MAC to a verification function that checks whether the MAC fits the message. As adversaries do not have access to the key, they cannot generate a correct MAC after they have modified a message. This technique cannot prevent modifications; however, it allows the recipient to detect whether any tampering has taken place.

If there is an agreement between sender and recipient that all messages are going to contain a MAC, attackers cannot create fake messages on their own. However, attackers can intercept a message of another user and send them to the designated recipient once again. Such a *replay attack* is useful for messages that instruct the recipient to perform a particular action, for instance, to unlock a door or to reset the password of an account. Replay attacks can be detected by the recipient as follows: sender and receiver agree that the sender adds a counter value to each message, which is supposed to be incremented with every message. Replays can then be detected because their counter value is smaller than a previously seen value or equal to the last seen one. The attacker cannot manipulate the counter value as it is also protected by the MAC.

Nevertheless, even replay detection is not sufficient in all cases. Consider the example of modern cars with a 'smart' entry system. Whenever the key is close to the car, the doors will automatically unlock if you attempt to open them. Car thieves have found a cheap technique to exploit this comfortable feature by working in teams (Greenberg 2017). The first perpetrator either gets close to the victim (in a coffee shop queue) or to the key (which may sit on a cupboard right behind the front door at home), carrying an antenna working on the same frequency as the smart key. The antenna is connected to a wireless transmitter with an extended range. The second perpetrator walks up to the car with the same equipment. This setup allows the thieves to carry out a *relay attack*, which makes the car believe that its key is close. Many modern cars have been shown to be vulnerable to relay attacks (Francillon et al. 2011). In principle, cars could be programmed to detect relay attacks, for instance, by measuring the delays between messages. Until manufacturers have upgraded security, consumers have to take care of themselves, e.g. by shielding the key or removing its battery.

2.4.3 The Benefits of Asymmetric Cryptography

Up to now, we have discussed what is called symmetric encryption and symmetric authentication—an approach that has several weaknesses. Firstly, these approaches require that each pair of senders and receivers that wants to communicate with each other have exchanged a secret key out of band. For n participants $0.5 \cdot n\,(n–1)$ keys have to be exchanged, i.e. in a system comprised of 20 components there would be 190 different keys. Thus, the symmetric approach scales poorly.

Secondly, sender and receiver have to store identical keys on their devices. This design increases the risk of key compromise because the adversary can obtain the keys either from the sender's or the receiver's device.

Thirdly, there are applications where symmetric message authentication is not sufficient. Consider a message containing the statement "I, Bob Miller, owe 100 Euros to Laura Fisher." Assume that Laura receives a letter with this statement in her

mailbox, however without any further indication of the sender. If the letter also contains a MAC and Laura can verify the MAC with the key she has exchanged with Bob, then Laura can be confident that it was indeed Bob who sent the letter. She has confirmed the authenticity of Bob's identity. However, let us assume that Bob later denies that he wrote the message. In that case Laura will not be able to convince a court that the MAC proves that Bob Miller wrote this message—after all, the key used for the MAC is not only known to Bob but also to her, i.e. she could have forged that message herself.

Asymmetric cryptography (also called public-key cryptography) allows us to overcome these limitations. In contrast to the symmetric approach, every entity (user or component) creates a *key pair*, which consists of a public key and a private key. The public key is shared with everyone else, and the private key is kept a secret.

Senders have to obtain the public key of the recipients with whom they want to communicate. As with symmetric cryptography, the key exchange is a sensitive matter. In particular, integrity protection is required, i.e. all parties must be certain that they obtained the authentic public keys. Without integrity protection, an adversary could interfere with the initial transmission of the public key. This would allow the adversary to forward the public key of a self-generated key pair to other parties. As a result, the adversary would become a so-called *man in the middle (MitM)*. MitM attackers can impersonate communication parties and decrypt messages designated for them. After decryption with the adversarial key, a MitM can encrypt the message with the public key of the designated recipient and forward the message towards the recipient, which makes it impossible for the recipient to detect that any kind of eavesdropping or manipulation has taken place. Although the concept of MitM attacks is considered basic knowledge, MitM attacks keep taking place in practice (cf., e.g. Cimpanu 2018; Seals 2018; Walker 2018).

Once senders have obtained a public key of their communication partner, they can create an encrypted message by feeding the message and the public key into an encryption function to obtain the ciphertext. The recipient can then retrieve the plaintext of the message by feeding the ciphertext and the corresponding private key to a decryption function.

Message authentication works similarly. A sender signs a message by feeding it together with the sender's private key into a signing function. Everyone who is in possession of the public key of the sender can then verify the message. The verification function consumes a message, the public key of the purported sender, and the signature. If verification succeeds, this means that the message has not been tampered with (integrity) and that the signature was genuinely generated by the private key that belongs to the public key used during verification (non-repudiation).

Asymmetric cryptography is in widespread use today. Most prominently, it is used to secure e-mails with the S/MIME and OpenPGP message formats. It also plays a vital role in securing the World Wide Web, which we discuss in the next section.

2.4.4 Case Study: Secure HTTP

Browsers typically communicate with web servers via HTTP (Hypertext Transfer Protocol), which is specified (among others) in RFC 7230 (Fielding and Reschke 2014). Today, many web servers respond by redirecting the browser to an HTTPS URL, which ensures that the connection between browser and server is protected against eavesdropping and tampering. Furthermore, HTTPS prevents adversaries on the network from impersonating a web server (which would allow adversaries, among others, to steal log-in credentials that are entered on web sites hosted there).

The security mechanisms of HTTPS are implemented with the Transport Layer Security (TLS) protocol. The most recent version, TLS 1.3, is specified in RFC 8446 (Rescorla 2018). This means that web servers are equipped with key pairs, which are associated with one or more domain names (e.g. www.uni-bamberg.de). The asymmetric key pair of a web server is *not* used to encrypt the actual data. The reason for this design is to provide a property known as *forward secrecy*: Even attackers that obtain the private key of a web server in the future shall not be able to learn the contents of a communication that has been observed (and stored) in the past. Therefore, the asymmetric key pair is only used to establish ephemeral symmetric session keys, which are then used to encrypt and authenticate the requests of the browser and the responses of the server.

In principle, this key establishment takes place for every new connection. This design, however, opens up a possibility for MitM attacks that aim to impersonate the destination web server. To prevent any tampering with the messages in the key establishment phase, the web server signs some of the messages with its private key. The browser can verify their integrity and authenticity with the public key of the web server. However, typically the client will not know the public key of the web server. This problem is tackled by making web servers send their public key to the client during the key establishment. However, without additional safeguards, this approach would allow MitM attackers to impersonate a web server by injecting their own key into the communication. This problem is overcome by introducing so-called *certificates*. Instead of sending the raw public key, a web server sends a certificate, which contains its public key, the domain names for which this certificate is valid, and a digital signature of a so-called Certification Authority (CA). CAs are organisations that issue certificates. A certificate is only issued to web server operators that can prove ownership of the domains to be included in the certificate. This approach prevents MitM attackers from forging certificates on the fly.

To verify the certificate presented by a web server, the browser needs the public key of the CA that issued that certificate. Browsers are equipped with the public keys of a number of large CAs by default (root certificates). If a web site uses a certificate from a different CA, the web server will include the certified public key of one or more intermediate CAs so that the browser can follow the chain of trust until one of the trusted root certificates is reached.

It is insightful to review different attacks against HTTPS. The objective of the adversary is either to eavesdrop on data in transit or to impersonate a particular web

server while users attempt to connect to it, with the ultimate goal of learning sensitive pieces of information such as the passwords of users. We review two well-known attacks in the following.

The first attack, *sslstrip*, was presented by Marlinspike (2011). This attack can be conducted by adversaries that control routers, for instance, a Wi-Fi access point that is being used by a victim to connect to the Internet. Whenever the victim visits a website via HTTP, sslstrip watches the (unencrypted) HTTP response for attempts by the web server to redirect the user's browser to the secure HTTPS version. In this case, sslstrip removes the redirection from the HTTP response. As a result, the user's browser will never learn that the web server intended to serve a secure version. Many users will not notice the mishap and enter sensitive data. The adversary can trivially eavesdrop on all communication before sslstrip forwards the traffic to the web server (of course, encrypted with HTTPS, as requested by the server).

Universally preventing sslstrip attacks is not trivial because of the conservative architecture of the World Wide Web: It relies on HTTP for initial contact. As a first step, the Electronic Frontier Foundation (EFF) has released the browser extension HTTPS Everywhere that replaces all HTTP connection attempts with HTTPS for a list of websites known to support HTTPS (Electronic Frontier Foundation 2018). A more generic approach envisions that web servers indicate that they support HTTPS by adding a 'Strict Transport Security' header to their responses (Hodges et al. 2012). The information that a web server supports HTTPS is then cached by browsers for a defined amount of time, which prevents sslstrip attempts after the initial connection. The initial connection remains vulnerable as it still relies on HTTP.

The second attack on HTTPS connections exploits the fact that every CA in the root certificate store can be used to issue a certificate for any domain name and that all major browsers will trust those certificates. Given that browsers trust several hundreds of CAs, there is a substantial risk that one of them will be compromised. Several CAs have been hacked in the past, resulting in the issuance of rogue certificates. Well-known cases are the CAs TürkTrust, Comodo, and DigiNotar (Laurie 2014).

In the past, users could not make out rogue certificates and there was no affordable way for most site owners to detect that another CA has issued a certificate for their domain.

A promising approach to detect rogue certificates is the *Certificate Transparency* initiative, which requires all CAs to add every issued certificate into one of several publicly verifiable append-only log files (Laurie et al. 2013). These log files are implemented in a tamper-proof fashion so that CAs cannot retroactively lie about a certificate they have issued. Browsers will only accept certificates from CAs that participate in this programme, which serves as a strong incentive for CAs to participate. Site owners can run monitors that continuously check whether certificates for their domains have been issued by rogue CAs, which minimises the amount of time such certificates can be used for malicious purposes. However, the deployment of Certificate Transparency comes with a catch, as we discuss in Sect. 2.7.1.

2.5 Malware Threats and Solutions

Malicious software, malware for short, is a significant threat to information and systems security. Malware is "a program that is inserted into a system, usually covertly, with the intent of compromising the confidentiality, integrity, or availability of a victim's data, applications, or operating system or otherwise annoying or disrupting the victim" (Souppaya and Scarfone 2013). Following the approach of Stallings and Brown (2014), we discuss propagation methods and payloads. After that, we consider countermeasures.

2.5.1 Propagation and Delivery

Some malware is designed to spread on its own. A well-known example is the SQL Slammer worm that infected more than 75,000 hosts over the Internet in 2003 (Moore et al. 2003). SQL Slammer exploited a buffer overflow vulnerability (cf. Sect. 2.6.1) in Microsoft's SQL Server. The vulnerable systems were reachable because the servers were not protected by a firewall (cf. Sect. 2.7.2).

Since then, the prevalence of firewalls has increased significantly. Therefore, malware authors have to rely on the help of humans for delivery. There are still some viruses around that infect files or file systems in the hope that users will exchange these with others, e.g. via USB sticks. However, most malware is now delivered via the Internet.

In the absence of vulnerabilities, the only way to infect a system consists in convincing a user to execute the malware. A typical approach consists in attaching malware to e-mails and tricking victims to execute it, exploiting their curiosity and insufficient technical expertise. Such attacks employ the same techniques that are also used for phishing. Sophisticated attackers use social engineering techniques to improve their chances, in quite the same way as so-called *spear-phishing* attacks target a particular person.

Another technique is called *drive-by download*. Here, users are tricked into visiting a website that is controlled by an attacker. The website is crafted to exploit a vulnerability (e.g. a buffer overflow, cf. Sect. 2.6.1) in the browser, ultimately forcing the browser to execute the malicious payload of the attacker. A more sophisticated variant of drive-by downloads are *malvertising* attacks (Nichols 2015). Here, attackers insert their malicious code into ads that they place on popular websites, which results in the infection of all visitors that have not patched their browser.

Adversaries that have researched their targets very well may be able to carry out a *waterholing* attack. A waterholing attack is possible if an adversary finds a way to either compromise a website that is typically visited by a victim or a server that hosts updates for software that is used by the victim. The attacker can then place the malware on this website, waiting for the victim to download it. In 2017, a state-sponsored waterholing attack was conducted by releasing a maliciously infected update for the CCcleaner tool (Amir 2017).

2.5.2 Payloads

Once a piece of malware is run on a target, it will execute its payload. The malware will typically deceive the user about its purpose, for instance, by exposing some benign functionality or an error. This kind of malware is called a *Trojan horse*.

In the past, the primary objective of malware was system corruption, by either deleting all files on a machine or preventing it from booting its operating system. Later on, malware authors discovered that they could exploit the fact that many users do not have backups: so-called *ransomware* encrypts the files on a system and demands the payment of a ransom in exchange for the decryption key and a tool that recovers the data.

Other payloads include key loggers to exfiltrate account credentials as well as remote control tools. Attackers that control a large number of systems can build up botnets that perform orchestrated activities such as sending out large amounts of spam e-mails or Distributed Denial of Service attacks (cf. Sect. 2.7.3).

2.5.3 Countermeasures

Baseline countermeasures against malware are the timely installation of security patches and user awareness training. These countermeasures try to avoid automated exploitation of known vulnerabilities and unintended execution of malware by naïve users. A typical—and if consequently followed also sensible—recommendation is to scrutinise e-mails with attachments, refraining from opening suspicious ones. However, it is difficult to spot a professionally executed spear-phishing attack.

Automated prevention of malware infections is the purpose of the so-called ‘anti-virus’ (AV) solutions. AV solutions monitor a system for suspicious activities that are indicative of malware. In principle, there are two approaches to decide whether a particular executable is malicious or not. The traditional method relies on malware signatures that are continually updated by the vendor. The effectiveness of signature-based AV tools is limited because they fail to detect slightly modified malware samples. In addition, AV tools increasingly rely on static and dynamic code analysis (heuristics). However, even this behaviour-based approach is not able to detect malware with 100% accuracy. Moreover, it may result in many false alerts (cf. Sect. 2.7.4).

Although widely deployed in organisations, some security practitioners are sceptical of AV tools. Firstly, professional attackers test their malware with a large number of AV tools, tweaking until it is not detected anymore. Secondly, some AV tools have been shown to introduce additional vulnerabilities (Anthony 2017a). A particularly interesting case is Windows Defender, the default AV engine of Windows, which scans all incoming e-mails for malware. Due to a vulnerability in Windows Defender, attackers could send specially crafted e-mails to victims that contained code that was automatically executed upon reception—even if the user never opened the e-mail (Anthony 2017b).

An alternative approach to AV solutions consists in executing suspicious files within a *sandbox*. Sandboxes are isolated machines instrumented with extensive monitoring capabilities. In contrast to behaviour-based AV tools, sandboxes do not have to make a real-time decision. While promising, sandboxes are no silver bullet. Malware authors have adapted to this new countermeasure; for instance, by delaying the execution of the payload until the timeout of the sandbox analysis has expired.

In some cases, it may be tempting to use active defence in order to defeat malware, for instance, by attempting to shut down its command and control infrastructure (cf. Sect. 2.3.3). Whether 'hacking back' is legal and ethically justifiable is an ongoing debate (Dittrich 2012; Schmidle 2018; see also Chap. 16). There have been incidents where interference with good intentions has caused harm. A noteworthy example is the case of the German e-mail provider Posteo that has deleted a mailbox used by the authors of the Petya ransomware (Cimpanu 2017). As a result, users who were willing to pay (or had already paid) the ransom could not get in touch with the authors any more to obtain the decryption key for their data. Initially, Posteo's decision was received critically. However, later on it was discovered that the particular variant of Petya used in the attack had been programmed to *delete* files (rather than encrypting them). Therefore, Posteo's act could be justified in the end, because no one would have gotten back their files anyway (Spring 2017).

2.6 Threats and Solutions in Software Security

Software security is concerned with weaknesses that result from programming errors. In the following, we present two common weaknesses, namely, buffer overflows and SQL injections. Subsequently, we discuss how vulnerabilities are found and reported to the vendors.

2.6.1 Case Study: Buffer Overflows

The most dominant security weakness in applications written in C and C++ are buffer overflows (Erickson 2008). To understand how buffer overflows work and what risks they impose, we have to introduce the basic ideas of memory management in C/C++ applications. Computations usually require some storage space in the computer's main memory, namely a buffer. A buffer has a specific location in the main memory and a given size. In C/C++, software developers are responsible for ensuring that buffers are large enough for the input they should hold. Programming languages that put this burden on the software developer are said to miss a security feature called 'memory safety'.

```
1 #include <stdio.h>
2 void main(void) {
3     int privilege_level = 1;
4     char buf[124];
5     fgets(buf, 1024, stdin);
6     if(privilege_level > 10) {
7         printf("You have admin rights. Level: %d\n",
8                privilege_level);
9     }
10    printf("Your input was: %s \n", buf);
11 }
```

Fig. 2.3 Example of a C program with a buffer overflow vulnerability

If software developers fail to allocate enough space for a buffer, they introduce a weakness into the code, a buffer overflow. An adversary can turn this weakness into a vulnerability by writing or reading outside the buffer, affecting other buffers located in the main memory, either before or after the original buffer. Modifying the content of this other buffer can influence the behaviour of the application; in particular, it may allow the adversary to execute arbitrary commands. Thus, a buffer overflow can result in the loss of confidentiality, integrity and availability.

To get the picture, consider the source code in Fig. 2.3, which reads input from the user and outputs it again. It contains an administration function that can only be activated by exploiting a buffer overflow.

Line 1 includes a common software library that makes it easier for the developer to read in user input and generate output shown to the user. In line 2, we define the main function of the program. Everything from line 3 to line 10 is part of this function and is executed in sequential order when the main function is executed; this happens at the start of the application. Line 3 defines a variable called 'privilege_level', which can store integer values. Initially, the privilege_level variable has a value of '1'. Variables allocate space in the main memory, in this case 4 bytes. Line 4 allocates 124 bytes for a buffer called 'buf', also in the main memory, next to the privilege_level variable. In line 5, the program reads a user's input from the keyboard by invoking the function 'fgets' (which is defined in the file 'stdio.h'). fgets is instructed to read up to 1024 bytes and write them into the buffer buf. However, buf can only store 124 bytes, thus introducing a buffer overflow. Line 6 checks for a condition: lines 7 and 8 only get executed if privilege_level is above 10. These lines print the user's current privilege level.

An adversary can exploit the buffer overflow to gain administrative privileges and execute lines 7 and 8. He starts the application and provides a specially crafted input. This input consists of arbitrary 124 bytes to fill buf, followed by 4 bytes with the value he wants privilege_level to have. So if he enters 'AAA...AAABBBB' (124 times 'A' followed by four 'B's), the application will print: 'You have admin rights. Level: 1111638594'. Internally, the application calls fgets on the adversary's input.

Consequently, fgets writes 128 bytes to buf. Since buf has only a size of 124 bytes, fgets continues writing to the memory location 'behind' buf, which in our example holds the value for privilege_level (the concrete locations of variables in memory depend on various factors; in our example we assume them to be as explained). Thus, privilege_level gets overwritten with four 'B's, which are consecutively interpreted as 1111638594.

We have discussed a simple example where we can spot the buffer overflow in the source code easily. However, in the source code of real-world applications, buffer overflows are more subtle, often hidden in calculations of buffer sizes. In addition, user input is not restricted to direct input on the keyboard, as in our example above. In real-world applications, buffer overflows may show up when reading image, audio, and video files, during the execution of JavaScript on web pages, and while processing network communication data.

Buffer overflows result from human mistakes. Thus, they cannot be prevented in all circumstances. Several techniques have been developed to make the exploitation of buffer overflows more difficult (e.g. Larsen and Sadeghi 2018). These techniques include data execution prevention (DEP), address space layout randomization (ASLR), stack canaries, and control-flow integrity (CFI). The diversity of defenses is the result of a cat-and-mouse game between defenders and attackers. Attackers consistently discover new ways to circumvent protections, for instance, return-oriented programming (ROP) against DEP (Buchanan et al. 2008).

2.6.2 Case Study: SQL Injections

Web applications commonly store their data in SQL (Structured Query Language) databases. However, this requires careful handling of users' input to avoid so-called SQL injections (Stuttard and Pinto 2011). To understand SQL injections, we first introduce the basics of SQL-based database systems.

SQL databases store data in tables. Each table has a name and several columns. Every row holds an individual record; just as we would expect it for a table. We will consider a table named 'users' with the columns 'id', 'email', 'password' and 'last_active' (cf. Table 2.1). To query the database, we use a domain-specific language, the SQL. An SQL statement describes which data to fetch from the database.

The idea behind an SQL injection is to maliciously modify the statement, either to extract additional information from the database or to modify the behaviour of an application, e.g. to bypass a login screen. We elaborate on the latter.

Table 2.1 A table in an SQL database that is used by an application vulnerable to SQL injections

id	email	password	last_active
1	john@example.com	3858f62230ac3c915f300c664312c63f	2018-09-01
2	jane@example.com	96948aad3fcae80c08a35c9b5958cd89	2018-10-14

```
1 <?php
2 $email = $_POST['email'];
3 $pw = hash_password($_POST['password']);
4 $query = "SELECT id, last_active
5           FROM users
6           WHERE email = '$email' AND password = '$pw'";
7 $resource = $db->query($query);
8 if($resource->numRows() == 1) {
9     $user = $resource->fetchRow();
10    echo "User logged in. ID: ", $user['id'];
11 }
12 ?>
```

Fig. 2.4 Login source code fragment of a PHP program that is vulnerable to SQL injections

We consider the program shown in Fig. 2.4, which implements a login form in PHP, a programming language often used for web applications.

Line 2 reads the e-mail address from the user input in the browser, while line 3 reads the user's password. Additionally, line 3 applies a hash function to the entered password to compare it against the value stored in the database later on. This avoids storing the password in clear text, which is considered bad practice. Lines 4–7 construct an SQL statement. There are different types of SQL statements; the most common ones are SELECT, UPDATE, INSERT and DELETE. Our statement selects certain data from the database; hence it starts with a SELECT followed by the columns we are interested in, namely 'id' and 'last_active'. However, the database system still needs to know which table we want to query, since multiple tables might use the same column names, e.g. 'id' to store a unique identifier for every record. Therefore, we use the FROM keyword to specify the table we are interested; in our case: 'users'. Now that the database system is aware of the table and its columns we wish to receive, we can apply a filter to fetch only a subset of all rows in the table. The WHERE condition on line 6 performs filtering: we state that we are only interested in rows which match the entered e-mail address and the provided password. Since e-mail addresses are unique, a correct input on the login form (consisting of e-mail address and password) will fetch exactly one row from the database, e.g. issuing the following SQL statement: SELECT id, last_active FROM users WHERE email = 'john@example.com' AND password = '38...3f'. The SQL statement is sent to the database system on line 7, while line 8 checks if exactly one row is returned. If that is the case, we execute lines 9 and 10 to read the result from the database and display the value stored in the id column of the row that matches the user's e-mail and password.

While this implementation of the login form works well for non-malicious inputs, it is prone to SQL injections and allows an adversary to bypass the login. In line 6, the application passes user input to an SQL statement without sanitising it first. We assume an adversary would enter some arbitrary password 'abc' into the password field and write the following into the e-mail address field in the login form:

$$'\text{OR } 1 = 1 \text{ LIMIT } 1\text{--}$$

```
1 <?php
2 $stmt = $db->prepare(
3     "SELECT id, last_active FROM users
4      WHERE email = ? AND password = ?")
5 $stmt->bind_param("ss", $email, $password);
6 $result = $stmt->execute();
7 ?>
```

Fig. 2.5 PHP code with a prepared statement to protect against SQL injection attacks

This will result in a valid SQL statement, which the application sends to the database system: SELECT id, last_active FROM users WHERE email = '' OR 1 = 1 LIMIT 1 -- ' AND password = 'abc'. We briefly discuss why this SQL statement results in a successful login without knowing a password. Compared to the benign SQL statement, the adversary alters the WHERE condition and adds an additional LIMIT keyword. In SQL, two dashes followed by a space (--) start a comment which will be ignored by the database system. Hence, our condition only reads WHERE email ='' OR 1 = 1 and is followed by LIMIT 1. The condition is true if the e-mail is empty (which is never the case) or if 1 is equal to 1 (which is always the case). Consequently, the condition matches all rows. However, the code checks in line 8 whether the database has returned exactly one row. Hence, the adversary adds a LIMIT 1 clause to ask the database system to return only the first row matching the condition. Thus, the check on line 8 passes and line 9 receives a valid row from the 'users' table. The adversary has successfully bypassed the login without knowing a password or e-mail address. More critically, an adversary could use the same SQL injection vulnerability to steal the whole database content, using a UNION SELECT statement.

SQL injections can be prevented by using prepared statements, which address the underlying problem of SQL injections: confusion of data and code. In our example above, the e-mail address field should have been treated as data. Prepared statements explicitly separate data from code, making SQL injections impossible. To this end, the SQL statements contain placeholders rather than the actual data. The pieces of data that are inserted instead of the placeholders are sent separately to the database. The source code in Fig. 2.5 illustrates prepared statements.

Line 2–4 create a prepared statement 'stmt' using questions marks ('?') as placeholders for data. On line 5, actual values are assigned to the question marks (declaring them as two strings). After that the query is executed on line 6. The data will be used by the database system in the places marked with the placeholders.

In real-world applications, SQL injections appear especially when SQL statements are constructed dynamically, e.g. when conditions are added and removed based on the users' input. Since SQL injections are easily avoidable, their occurrence is an indicator for the lacking security education of developers.

2.6.3 Finding and Handling Vulnerabilities

Vulnerabilities can be found in applications using different methods; they may be kept secret or reported to vendors, either publicly or privately. Vendors respond in different ways to those reports and differ in their approaches to addressing the issue. Besides fixing known vulnerabilities, vendors can take preventive measures to avoid vulnerabilities in the first place or apply defence-in-depth techniques for mitigation. We elaborate on these aspects, beginning with how to find vulnerabilities and concluding on techniques for prevention and mitigation.

As seen in the previous case studies, vulnerabilities can be found by carefully reading the source code. This method is called a code audit and is typically performed by trained security auditors. Security auditors may use tools for assistance. Those tools highlight source code locations that potentially contain a vulnerability. However, false positives are quite common. These locations are reported to contain a vulnerability, although they are fine.

Furthermore, there are plenty of false negatives because it is not possible to detect all vulnerabilities automatically. Firstly, code audit tools apply heuristics, i.e. approximations of how the source code may behave; they are only as good as their heuristics are. Secondly, complete reasoning about the source code would be equivalent to deciding the halting problem,[1] which is known to be impossible (Chess and McGraw 2004). Therefore, complete reasoning is not possible. Thirdly, identifying security-related logic bugs—i.e. bugs that are highly specific to the concrete behaviour of an application—require a machine-readable specification of the application's behaviour, which in most cases does not exist. Moreover, a specification does not necessarily cover the human intent, thus being erroneous itself. Consequently, tools can never replace a security auditor in a code audit.

Performing a code audit requires access to the source code of an application. Unless an application is open source software, the source code is typically not available to external auditors, who analyse an application without being instructed by the vendor. In this case, auditors have to perform reverse engineering, i.e. understand the application's machine code, which is intended to be run by a computer and not easily understandable for humans. Even with tool support, it is impossible to recover the source code completely. Despite these hurdles, many vulnerabilities are found with reverse-engineering techniques.

[1] The halting problem asks an abstract machine model, the Turing machine, to decide whether a computer program terminates (halts) on a given input or runs forever. It is undecidable, i.e. it cannot be answered for all computer programs and inputs, despite the fact that there is a 'yes' or 'no' answer for every program and input. The Church-Turing thesis states that what humans and Turing machines can compute is equivalent. Given this thesis, humankind cannot answer all questions for which there are answers; even with unlimited computational resources (Sipser 2012).

A third technique is fuzzing. Fuzzing feeds millions of different random inputs to an application and checks for unintended behaviour such as crashes. A crash is a good indicator of the existence of a vulnerability. Inputs that lead to crashes are then stored for later analysis. To generate those inputs, a fuzzer modifies existing inputs and observes which parts of an application are executed given the modified input. To increase the likelihood of a crash, the fuzzer tries to execute all parts of an application. The motivation behind this method is to find parts that are usually not executed on expected user inputs and are therefore untested for (security) bugs. Fuzzing has proven surprisingly effective: for instance, a fuzzer found several vulnerabilities in the popular OpenVPN software even after two code audits had already been performed (Vranken 2017).

After a vulnerability is found, the security auditor may decide to keep it secret or to report it. Motivations for keeping a vulnerability secret include planned criminal actions, espionage by secret services, and accessing a suspect's device by law enforcement. In all those cases, it is likely that an exploit is developed to make use of the vulnerability. A vendor cannot fix a vulnerability as long as he is not aware of it. Thus, unreported vulnerabilities often stay unfixed for a long time. Vulnerabilities without a fix are called 'zero days' or '0-days'.

There are two approaches to the publication of vulnerabilities: full disclosure and responsible disclosure. In full disclosure, the vulnerability is disclosed in public, without notifying the vendor in advance. Advocates of full disclosure argue that all users of a vulnerable software should have the same information regarding the vulnerability to be able to assess their risks and take appropriate countermeasures until a fix is released. They accept the risk that adversaries may use the information to develop an exploit and target the users of the vulnerable software. Furthermore, proponents of full disclosure argue that full disclosure puts more pressure on the vendor to faster create and ship a fix and to care more about security in the first place.

In contrast to full disclosure, responsible disclosure (sometimes also called coordinated disclosure) mandates informing the vendor first, usually granting it a specific timeframe to release a fix before going public. The length of this embargo is a trade-off between putting pressure on the vendor and giving the vendor the opportunity to investigate the issue thoroughly, including extensive testing of the fix. A typical value is 90 days. Vendors may ask for an extension of the embargo. However, it is at the discretion of the finder to grant it. For instance, there has been a high-profile case in which security researchers working at Google have not granted Microsoft an extension (Tung 2018).

Responsible disclosure is not without flaws. Some software is distributed by different organisations that may release a fix at different times. This is the case for Linux distributions that contain thousands of different software packages. A fix released by one Linux distribution can provide information about the vulnerability, which can then be used by adversaries to attack users of other Linux distributions that have not released a fix yet. Furthermore, the more people are involved with developing and distributing a fix, the more likely it is that information about the vulnerability leaks before a fix is shipped.

Vendors should follow established best practices for adequate handling of vulnerabilities (see also Chap. 15). Firstly, they should provide a dedicated security contact on their website to ensure that vulnerability reports reach the right group within an organisation. Otherwise, support staff who are not educated in reading technical security reports might ignore those reports due to misunderstandings. In addition, it is recommended to provide a public key (cf. Sect. 2.4.3) for exchanging encrypted mails with the security contact, e.g. using OpenPGP. Secondly, the vendor should acknowledge the receipt of a vulnerability report and after investigating the issue, confirm the problem (if it is valid). Thirdly, the vendor is expected to suggest a schedule for a coordinated release of a fix and the report. Guidelines and detailed recommendations have been published by Householder et al. (2017).

Vulnerability finders invest their time to make users of the vendor's software more secure. Legal threats as a response to a report are considered immoral and may result in a Streisand effect, i.e. trying to hide or censor some information has the effect of unintentionally distributing the information more widely. Today, this often occurs through social media and results in negative publicity for the software vendor.

Instead of legal threats, the security community encourages vendors to be transparent about security problems in their products. Moreover, vendors should provide as much information as possible to allow their users to accurately assess any risks they may be exposed to. Quickly providing a fix is considered best practice. Besides, some vendors offer a bug bounty program, which provides vulnerability reporters with monetary compensation.

2.7 Threats and Solutions in Network Security

Many systems are interconnected over networks. This increases their exposure. In the following, we consider selected threats to networked systems.

2.7.1 Case Study: Reconnaissance

Reconnaissance of the target is an essential part of sophisticated attacks. Networked systems provide a significant amount of information that can be used to launch attacks that are tailored to the environment of the victim and thus more likely to succeed.

Attackers benefit from the fact that the Internet has been designed to be an open network. For instance, information about network operators is publicly available so that system administrators of different parts of the world can communicate with each other in case of problems. This kind of information can be looked up with the so-called 'whois service', a distributed database that holds contact information about anyone who has leased IP addresses or domain names. Given some seed

information such as an IP address (e.g. 141.13.240.24) or a domain name (e.g. www.uni-bamberg.de) of a target, the whois service helps attackers finding other and related systems run by the same organisation. Moreover, whois 'leaks' names and contact information of employees, which can be useful for social engineering.

Some of the information shared via whois is considered personal data and therefore protected under the General Data Protection Regulation of the European Union. As a result, the German registrar DENIC stopped unrestricted access to contact information for all '.de' domains in 2018 (DENIC eG 2018). This move considerably increases the effort for system administrators that want to contact domain owners to resolve problems (Winterfeldt 2018).

Whois is not the only system that leaks information. For instance, attackers can use the Domain Name System, which is a distributed database that maps domain names such as example.com to IP addresses. Many administrators assign telling names to their servers that help attackers understand the purpose of a system. Reverse DNS lookups allow attackers to look up these hostnames (e.g. webmail05.example.net) given the IP address of a system of interest.

Moreover, attackers can exploit two relatively new systems that aim to increase transparency, but come with an inherent security trade-off, namely Certificate Transparency (cf. Sect. 2.4.4) and Passive DNS. These systems have been created to mitigate particular security problems. However, they have the side effect of leaking sensitive information to attackers. Certificate Transparency creates transparency about all TLS certificates that are registered. Passive DNS services make available all domain names that are looked up by a group of DNS clients. Both services leak the hostnames of internal systems, helping attackers find potential targets.

Finally, attackers use port scanners to enumerate all publicly reachable hosts and services. With tools such as nmap, attackers can obtain a list of open ports and additional information such as the software that might be offering the ports as well as the operating system. If system administrators of a target have been careless or negligent, they might have forgotten to set up strict firewall rules (cf. Sect. 2.7.2) that prohibit unauthorised connection attempts to sensitive services from the outside. Although port scans are not harmful on their own, they certainly help to increase the effectiveness and efficiency of attacks.

A relatively new development is that attackers do not necessarily have to use a port scanner themselves. For an initial sweep of a target, attackers can also rely on the information provided by services such as shodan.io and censys.io. These two services continuously scan (a large part of) the Internet and make the results available on their website via a convenient full-text search engine. Their actual purpose is to help system administrators secure their networks by simplifying continuous monitoring. However, they also help attackers find improperly secured systems without having to send a single packet to a target. This dilemma makes shodan.io an interesting tool for creating awareness about vulnerable industrial control systems that are insufficiently protected (cf., e.g. Gallagher 2018).

2.7.2 Case Study: Perimeter Security Via Firewalls

Firewalls are systems that are deployed to restrict the access to services on the network layer. These services are either internal services that should not be available from outside an organisation's network or services on the Internet that should not be accessed by the employees of an organisation.

On the network, information is sent in packets. Each packet consists of a header and a payload. The payload contains the data that is being sent. The header contains information about the sender, the receiver, and the so-called ports being used. Services listen on particular ports (identified by a number between 1 and 65535). A packet is delivered to a service, if the port number stated in the packet corresponds to the port number of the service.

Most firewalls filter packets solely based on their header. To allow only access to specific services, a system administrator can configure a firewall to drop all packets that do not match a list of specific port numbers. The underlying assumption behind such firewall rules is that particular services listen on specific ports, e.g. web servers listen on port 443 (the default port for HTTPS, cf. Sect. 2.4.4) for encrypted communication. However, this assumption does not hold necessarily, since services can be reconfigured to listen on arbitrary ports.

Thus, firewalls can be bypassed using ports that are commonly allowed in the firewall's configuration, such as port 443. If users inside a corporate network want to access the Internet without any restrictions, they can run a tunnel service on a publicly reachable Internet server on port 443 and send their communication through this tunnel, which forwards it to the Internet, bypassing the firewall.

As a response to tunnel services, some firewalls check if the packets contain data for a specific service, e.g. they check if packets for port 443 actually contain HTTPS data. This technique is called Deep Packet Inspection (DPI). It is an open debate whether DPI is an acceptable practice. Opponents of DPI argue by comparing packets to postal mail: the packet's header is like the address data on the envelope and must be read by the postal service for delivery, while the packet's payload is like the letter inside the envelope. DPI looks at the payload; therefore, it is like opening the envelope of every letter, thus violating postal privacy. It is noteworthy that even DPI cannot entirely prevent users bypassing a firewall. Thus, data exfiltration prevention is another cat-and-mouse game between attackers and defenders. For example, there are sophisticated tunnelling techniques, e.g. DNS tunnels such as *iodine* (Nussbaum et al. 2009), that trick DPI solutions by hiding the exchanged data within DNS messages (which are typically not restricted by firewalls).

2.7.3 Case Study: Denial of Service Attacks

In a Denial of Service (DoS) attack, an adversary tries to occupy a massive amount of the victim's resources. The goal is to deny these resources to legitimate users. Typical DoS attacks either create large amounts of traffic to fill up the victim's communication lines or exhaust the victim's computational resources.

Adversaries can also instruct many machines to participate in an attack. This results in a Distributed Denial of Service (DDoS) attack. To perform a DDoS attack, an adversary compromises thousands of machines. These machines then form a so-called botnet. One of the largest botnets for DDoS attacks, called Mirai, was built using insecure Internet of Things (IoT) devices, such as routers and IP cameras. Users often employ these devices without knowing their security ramifications. Once deployed, IoT devices are often poorly maintained and seldomly receive any security updates. The Mirai botnet attacked KrebsOnSecurity, a blog maintained by the security journalist Brian Krebs, with a bandwidth of 620 GBit/s (Krebs 2016). For comparison, many commercial websites are only connected to the Internet with a bandwidth of 1 GBit/s.

One particularly intriguing type of DoS attacks are amplification attacks. In an amplification attack, an adversary uses a third party, e.g. a DNS server, to perform the attack. The DNS server responds to a small request sent by the adversary with a large answer. To attack a victim, the adversary spoofs his sender address, setting it to the address of the victim. Consequently, the DNS server receives the small request from the adversary and sends a large response to the victim. Thus, the adversary's DoS traffic is amplified by the DNS server. Spoofing the sender address is possible because Internet Service Providers do not filter the traffic of their customers properly.

DoS attacks are made possible because of externality effects. Firstly, vendors of cheap IoT devices have no incentive to provide security updates for the whole lifetime of a product. Secondly, there is virtually no reason for Internet service providers to check for address spoofing. In both cases, there is a party that is passively responsible but does not bear the costs of attacks. To improve the state of affairs, vendors and service providers have to be externally incentivised, for instance, through legal regimes.

Often, attackers use DoS attacks to force victims into paying ransoms. Online shops lose money when they are not reachable for their customers. Therefore, they will do almost anything to stop an ongoing DoS as quickly as possible. Defending against DoS attacks is difficult for server operators in practice. After all, the defender must provision more resources than the attacker can consume, which is quite costly. Therefore, there is now a market for DoS protection. Companies in this market provide large amounts of resources and filter their customer's traffic for DoS attacks. Legitimate traffic is forwarded to the customer, while DoS traffic is discarded.

2.7.4 Case Study: Network Intrusion Detection Systems

Network Intrusion Detection Systems (NIDS) such as Snort try to detect attacks on the network layer. They look into the packets that arrive over the network and decide whether the communication associated with the packets might be an attack. There are two different types of NIDS: signature-based and anomaly-based.

Signature-based NIDS can only detect attacks that are already known. They rely on a database of signatures to identify attacks. A signature describes the content of network packets that can be observed during a specific attack; for instance, there is one signature for the Heartbleed attack (http://heartbleed.com) as well as one signature for the Shellshock attack (Seltzer 2014). Thus, to detect current threats, the database of a NIDS must be updated on a regular basis.

In contrast, an *anomaly-based NIDS* analyses network communication patterns within a network. After the NIDS has learned what 'normal' communication patterns look like, the NIDS tries to detect deviations. Those anomalies are then considered to be attacks or at least unwanted behaviour. Whereas anomaly-based NIDS have the advantage that there is no database to maintain, they rely on the questionable assumption that there was no malicious activity during training. Moreover, whenever the communication patterns on the network change, e.g. because new software is introduced, the NIDS has to be retrained.

Neither signature-based nor anomaly-based NIDS can detect all threats. Their information is limited to network communication. They have no information about the inner workings of the software used on the network. For instance, communication exploiting logic bugs can be difficult or impossible to distinguish from benign communication.

Furthermore, network communication is increasingly encrypted. Encrypted traffic cannot be analysed by NIDS. This limitation can be overcome by allowing the NIDS to intercept all encrypted traffic by adding its certificate to the root certificate store on all clients. This approach, which is called TLS interception, is a very intrusive form of Deep Packet Inspection (cf. Sect. 2.7.2). TLS interception has been called into question, because it allows the administrators of the NIDS to eavesdrop on all encrypted communication. Moreover, TLS interception often decreases the actual security of encrypted communications (Waked et al. 2018).

The evaluation of the accuracy of a NIDS is not straightforward. We have to consider four metrics: the true positive rate (attacks that are detected), the false negative rate (attacks that are not detected), the false positive rate (benign communication wrongly flagged as an attack) and the true negative rate (benign communication not flagged as attack).

Even very accurate NIDS generate many false positives (false alarms) because malicious traffic is much more seldom than 'normal' traffic. This is known as the *base rate fallacy* (Axelsson 1999). Assume that 1 out of every 100,000 packets has

a malicious payload (this is the base rate). Further assume that a hypothetical NIDS has an accuracy of 99.9%, which refers to the true positive rate and to the true negative rate, which are equal here. Most of the very few malicious packets will be classified correctly. However, during the reception of the 99,999 benign packets, the NIDS will generate about 100 false alarms. In other words: the operators of this hypothetical NIDS have to handle 100 times more false alarms than malicious payloads. The imbalance between false alarms and real alarms is not a theoretical problem, it is one of the most pressing issues in practical NIDS.

2.8 Continuous Testing

Properly securing a system means that defenders have to perform regular checks. After all, every change to the infrastructure, every update for a software package and every change in operational procedures may introduce vulnerabilities.

As described in Sect. 2.6.3, code audits can be used to detect vulnerabilities in software. Finding vulnerabilities in distributed systems is more involving. Common practices consist in running security scanners and performing penetration tests.

Security scanners such as Nessus and OpenVAS allow system operators to check their infrastructure for a wide array of known vulnerabilities by probing all devices within a defined address range. The specifics that determine how a scanner checks for a particular vulnerability are provided by the vendors of such scanners.

Whereas security scanners are typically set up by the operators of a system, penetration tests are usually conducted by specialised firms. Penetration tests are useful because they simulate a real attack. Among other things, they allow organisations to understand whether previously launched awareness campaigns on social engineering were effective and whether operators react sensibly when under pressure.

Many penetration testers use a toolkit called Metasploit (metasploit.com), which makes it possible to validate whether a particular vulnerability can be exploited—by actually exploiting it and launching a selectable payload. From an ethical perspective, Metasploit is interesting because it encapsulates exploits in ready-to-run packages, which eases the job of security analysts. Sharing exploit code is considered essential to improve security. However, in former times when exploits were shared on mailing lists, it was regarded as good practice to intentionally modify the code so that script kiddies would not be able to execute it. Metasploit has broken with this tradition, lowering the bar considerably.

Given its potential for damage, it is not surprising that there have been attempts to regulate the distribution of dual-use tools such as Metasploit (Schneier 2007; Hulme 2012). However, such a policy is mostly ineffective. Attackers will always find ways to get access to such tools. Moreover, restrictions make it difficult to use offensive tools for educational purposes, which would decrease the competence of the defenders in the long run.

2.9 Conclusion

In this chapter, we introduced the basic concepts and models of cybersecurity. Given the complexity of this field, there are many directions for further exploration. Nonetheless, even the basics presented in this chapter raise several ethical questions.

First, ethical issues are relevant for cybersecurity professionals, i.e. on the level of individuals. Security analysts may have to decide how they should deal with a newly discovered vulnerability. Should they only disclose it to the responsible vendor or also inform the public? If they decide to publish it, which details should be made available before the vulnerability is fixed? On the one hand, publishing too much or too early might cause significant harm. On the other hand, keeping the vulnerability secret prevents users of the vulnerable product from taking action on their own, and it decreases the vendor's incentive to actually ship a fix in a timely manner. This is only one example where the actual outcomes of various alternatives are difficult to predict, which is why there is no consensus about vulnerability disclosure in the community (more in Chaps. 3 and 4).

Ethical issues are also encountered on an organisational level. Most organisations struggle with finding a justifiable balance between investing in security and accepting the remaining risks. Security cannot be bought from the shelf because organisations have different needs. Moreover, effective security relies on humans—and humans tend to act (or fail) in surprising ways. Organisations may also be inclined to exploit power asymmetries that allow them to externalise their costs by transferring risks to users or other unrelated parties.

Finally, ethical issues also arise on an architectural level. It is challenging to predict how a new system or security mechanism will be used. This is particularly an issue for dual-use tools whose impact on security depends on the intentions of the actor. Another example is Certificate Transparency, which has been designed to solve a particular security issue. However, it can also be misused for reconnaissance. Of course, this kind of exploitation was foreseeable for experts, but it still startles system administrators whose internal hosts are now exposed in a public database. Building useful systems with limited misuse potential is a challenging problem for which we do not yet have readily available solutions.

Tackling ethical questions in the field of cybersecurity is difficult due to its very nature: We usually have to make decisions based on insufficient information. We often do not fully understand the consequences of turning a particular lever and systems exhibit surprising (emergent) behaviour once users (and creative adversaries) lay their hands on them. In rare cases, we may be able to collect some facts (e.g. by studying past events); however, it is questionable whether these are still applicable. After all, cybersecurity is an endless cat-and-mouse game with constantly changing rules.

Acknowledgements The chapter was created with funding from the European Commission (H2020-700540 CANVAS). The authors are grateful to Stephanie Loreck and Oleg Geier for comments on a draft of this chapter.

References

Amir W (2017) CCleaner backdoor attack: a state-sponsored espionage campaign. https://www.hackread.com/ccleaner-backdoor-attack-a-state-sponsored-espionage-campaign/. Last access 7 July 2019

Anderson JP (1972) Information security in a multi-user computer environment. Adv Comput 12:1–36

Anthony S (2017a) It might be time to stop using antivirus. https://arstechnica.com/information-technology/2017/01/antivirus-is-bad/. Last access 7 July 2019

Anthony S (2017b) Massive vulnerability in Windows Defender leaves most Windows PCs vulnerable. https://arstechnica.com/information-technology/2017/05/windows-defender-nscript-remote-vulnerability/. Last access 7 July 2019

Axelsson S (1999) The base-rate fallacy and its implications for the difficulty of intrusion detection. In: Proceedings of the 6th ACM conference on computer and communications security, CCS '99. ACM, New York, pp 1–7

Buchanan E, Roemer R, Shacham H et al (2008) When good instructions go bad: generalizing return-oriented programming to RISC. In: Ning P, Syverson PF, Jha S (eds) Proceedings of the 2008 ACM conference on computer and communications security, CCS 2008, Alexandria, Virginia, USA, October 27–31, 2008, ACM New York, pp 27–38

Chan CS (2012) Complexity the worst enemy of security. https://www.schneier.com/news/archives/2012/12/complexity_the_worst.html. Last access 7 July 2019

Chess B, McGraw G (2004) Static analysis for security. Secur Priv 2(6):76–79

Cimpanu C (2017) E-mail provider shuts down Petya Inbox Preventing Victims from re-covering files. https://www.bleepingcomputer.com/news/security/email-provider-shuts-down-petya-inbox-preventing-victims-from-recovering-files/. Last access 7 July 2019

Cimpanu C (2018) Popular Android Apps vulnerable to man-in-the-disk attacks. https://www.bleepingcomputer.com/news/security/popular-android-apps-vulnerable-to-man-in-the-disk-attacks/. Last access 7 July 2019

DENIC eG (2018) Extensive changes planned for DENIC Whois domain query: proactive approach for data economy and data protection. https://www.denic.de/en/whats-new/press-releases/article/extensive-innovations-planned-for-denic-whois-domain-query-proactive-approach-for-data-economy-and/. Last access 7 July 2019

Dittrich D (2012) So You Want to Take Over a Botnet… In: Kirda E (ed) 5th USENIX workshop on large-scale exploits and emergent threats, LEET '12, San Jose, CA, USA, April 24, 2012 Berkeley USENIX Association. https://www.usenix.org/conference/leet12. Last access 7 July 2019

Electronic Frontier Foundation (2018) HTTPS everywhere. https://www.eff.org/https-everywhere. Last access 7 July 2019

Engel G (2014) Deconstructing The Cyber Kill Chain. https://www.darkreading.com/attacks-breaches/deconstructing-the-cyber-kill-chain/a/d-id/1317542. Last access 7 July 2019

Erickson J (2008) Hacking: the art of exploitation, 2nd edn. No Starch Press, San Francisco

Fielding R, Reschke J (2014) Hypertext Transfer Protocol (HTTP/1.1): message syntax and routing. Request for comments, RFC 7230. https://tools.ietf.org/html/rfc7230. Last access 7 July 2019

Francillon A, Danev B, Capkun S (2011) Relay attacks on passive keyless entry and start systems in modern cars. In: Network distributed system security. The Internet Society, NDSS, Reston

Gallagher S (2018) Vulnerable industrial controls directly connected to Internet? Why not?. https://arstechnica.com/information-technology/2018/01/the-internet-of-omg-vulnerable-factory-and-power-grid-controls-on-internet/. Last access 7 July 2019

Gollmann D (2011) Computer security, 3rd edn. Wiley, Chichester

Greenberg A (2017) Just a pair of these $11 radio gadgets can steal a car. https://www.wired.com/2017/04/just-pair-11-radio-gadgets-can-steal-car/. Last access 7 July 2019

Hodges J, Jackson C, Barth A (2012) HTTP Strict Transport Security (HSTS). Request for comments, RFC 6797. https://tools.ietf.org/html/rfc6797. Last access 7 July 2019

Householder AD, Wassermann G, Manion A et al (2017) The CERT® guide to coordinated vulnerability disclosure. Special report CMU/SEI-2017-SR-022, Carnegie Mellon University, CERT Division

Hulme GV (2012) Metasploit review: ten years later, are we any more secure? https://searchsecurity.techtarget.com/feature/Metasploit-Review-Ten-Years-Later-Are-We-Any-More-Secure. Last access 7 July 2019

Hutchins EM, Cloppert MJ, Amin RM (2011) Intelligence-driven computer network defence informed by analysis of adversary campaigns and intrusion kill chains. Lead Issue Inf Warf Secur Res 1:80

Korolov M (2018) What is a supply chain attack? Why you should be wary of third-party providers. https://www.csoonline.com/article/3191947/data-breach/what-is-a-supply-chain-attack-why-you-should-be-wary-of-third-party-providers.html. Last access 7 July 2019

Krebs B (2016) KrebsOnSecurity hit with record DDoS. https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/. Last access 7 July 2019

Langner R (2013) To kill a centrifuge: a technical analysis of what Stuxnet's creators tried to achieve. https://www.langner.com/wp-content/uploads/2017/03/to-kill-a-centrifuge.pdf. Last access 7 July 2019

Larsen P, Sadeghi AR (eds) (2018) The continuing arms race: code-reuse attacks and defences. Association for Computing Machinery and Morgan & Claypool, New York

Laurie B (2014) Certificate transparency. Queue 12(8):10:10–10:19

Laurie B, Langley A, Kasper E (2013) Certificate transparency. Request for comments, RFC 6962. https://tools.ietf.org/html/rfc6962. Last access 7 July 7 2019

Marlinspike M (2011) sslstrip. https://moxie.org/software/sslstrip/. Last access 7 July 2019

McConnell S (2004) Code complete: a practical handbook of software construction, 2nd edn. Microsoft Press, Redmond

Moore D, Paxson V, Savage S et al (2003) Inside the Slammer Worm. IEEE Secur Priv 1(4):33–39

Nichols S (2015) You've been Drudged! Malware-squirting ads appear on websites with 100+ million visitors. https://www.theregister.co.uk/2015/08/14/malvertising_expands_drudge/. Last access 7 July 2019

Nussbaum L, Neyron P, Richard O (2009) On robust covert channels inside DNS. In: Gritzalis D, López J (eds) Emerging challenges for security, privacy and trust, 24th IFIP TC 11 international information security conference, SEC 2009, Pafos, Cyprus, May 18–20, 2009. Proceedings, IFIP Advances in Information and Communication Technology, vol 297. Springer, Berlin/New York, pp 51–62

Pfleeger CP, Pfleeger SL, Margulies J (2015) Security in computing, 5th edn. Prentice Hall Press, Upper Saddle River

Rescorla E (2018) The Transport Layer Security (TLS) Protocol Version 1.3. Request for comments, RFC 8446. https://tools.ietf.org/html/rfc8446. Last access 7 July 2019

Saltzer JH, Schroeder MD (1975) The protection of information in computer systems. Proc IEEE 63(9):1278–1308

Schmidle N (2018) The digital vigilantes who hack back. https://www.newyorker.com/magazine/2018/05/07/the-digital-vigilantes-who-hack-back. Last access 7 July 2019

Schneier B (2007) New German hacking law. https://www.schneier.com/blog/archives/2007/08/new_german_hack.html. Last access 7 July 2019

Seals T (2018) Bluetooth bug allows man-in-the-middle attacks on phones, laptops. https://threatpost.com/bluetooth-bug-allows-man-in-the-middle-attacks-on-phones-laptops/134332/. Last access 7 July 2019

Seltzer L (2014) Shellshock makes Heartbleed look insignificant. https://www.zdnet.com/article/shellshock-makes-heartbleed-look-insignificant/. Last access 7 July 2019

Sheridan K (2018) The cyber kill chain gets a makeover. https://www.darkreading.com/threat-intelligence/the-cyber-kill-chain-gets-a-makeover/d/d-id/1332892. Last access 7 July 2019

Shirey R (2007) Internet security glossary, Version 2. Request for comments, RFC 4949. https://tools.ietf.org/html/rfc4949. Last access 7 July 2019

Shostack A (2014) Threat modeling: designing for security, 1st edn. Wiley, Indianapolis

Sipser M (2012) Introduction to the theory of computation, 3rd. Cengage Learning, Boston

Smith R (2012) A contemporary look at Saltzer and Schroeder's 1975 design principles. IEEE Secur Priv 10(6):20–25
Souppaya M, Scarfone K (2013) Guide to malware incident prevention and handling for desktops and laptops. NIST Special Publication SP Gaithersburg 800–883
Spitzner L (2002) Honeypots: tracking hackers. Addison-Wesley Longman Publishing Co., Inc., Boston
Spring T (2017) ExPetr called a Wiper Attack, not Ransomware. https://threatpost.com/expetr-called-a-wiper-attack-not-ransomware/126614/. Last access 7 July 2019
Stallings W, Brown L (2014) Computer security: principles and practice, 3rd edn. Prentice Hall Press, Upper Saddle River
Stuttard D, Pinto M (2011) The web application Hacker's handbook: finding and exploiting security flaws, 2nd edn. Wiley, New York
Tung L (2018) Google's Project Zero exposes unpatched Windows 10 lockdown bypass. https://www.zdnet.com/article/googles-project-zero-reveals-windows-10-lockdown-bypass/. Last access 7 July 2019
Voydock VL, Kent ST (1983) Security mechanisms in high-level network protocols. ACM Comput Surv 15(2):135–171
Vranken G (2017) The OpenVPN post-audit bug bonanza. https://guidovranken.com/2017/06/21/the-openvpn-post-audit-bug-bonanza/. Last access 7 July 2019
Waked L, Mannan M, Youssef A (2018) To intercept or not to intercept: analyzing TLS interception in network appliances. In: Proceedings of the 2018 on Asia conference on computer and communications security, ASIACCS, vol 18. ACM, New York, pp 399–412
Walker J (2018) Cybersecurity company hit by man-in-the-middle attack. http://www.digitaljournal.com/tech-and-science/technology/cybersecurity-company-hit-by-man-in-the-middle-attack/article/510402. Last access 7 July 2019
Wheeler T (2018) In cyberwar, there are no rules. https://foreignpolicy.com/2018/09/12/in-cyber-war-there-are-no-rules-cybersecurity-war-defence/. Last access 7 July 2019
Winterfeldt B (2018) The fight is on to save access to WHOIS: a call to action for brand owners. http://www.circleid.com/posts/20180419_fight_is_on_to_save_access_to_whois_call_to_action_brand_owners/. Last access 7 July 2019
Zetter K (2016) Inside the cunning, unprecedented hack of Ukraine's power grid. https://www.wired.com/2016/03/inside-cunning-unprecedented-hack-ukraines-power-grid/. Last access 7 July 2019