A Brief Introduction to Information Security*

Rainer Böhme Tyler Moore

August 26, 2013

The goal of this chapter is to give a brief introduction to the modern view of information security as a prerequisite to organizing an open, free, and democratic information society. It introduces design principles and, on a high level of abstraction, the technical terminology needed to discuss economic incentives around the provision of information security. It is targeted to people with a background in economics or social sciences. Readers trained in computer science or engineering disciplines may recognize most topics. They are invited to simply skip this chapter, though many computer scientists may find the presentation and emphasis different from what they have previously encountered.

1 Protection Goals: Confidentiality, Integrity, and Availability

Let us first reflect on the notion of security in general and then narrow it down to information security. *Protection* means to prevent undesirable events by having defenses in place which physically or logically rule out that these events may happen. *Security* refers to protection against intentional malice. This is different from *safety*, which refers to the protection against accidental threats, such as technical or human failure. In a broad sense, safety defends against nature whereas security defends against intelligent beings. Safety can be modeled with probability theory using measurements or historical data, for instance about the reaction of material under physical stress. For safety analyses, it is often sufficient to look at average failure rates. This is not adequate for security, because intentional action does not necessarily obey probabilistic rules.

Instead, those seeking to disrupt a system as well as those tasked with protecting a system can reasonably be expected to anticipate every simple probabilistic rule and adapt their actions strategically. This has several implications. First, because intentional malice is strategic, game theory is better suited to model security problems than probability theory. Second, the engineers and computer scientists responsible for security have traditionally adopted a simpler

*Please note that this is a working draft. Feedback is much appreciated. Please email rainer.boehme@uni-muenster.de (for Univ. of Münster students) or tylerm@smu.edu (for SMU students) with any comments.

approach to deal with strategic adversaries: prepare for the worst case. Security engineers often posit a hypothetical adversary and assign her extraordinary capability (e.g., the ability to observe *all* communications and surreptitiously modify messages at will). They then investigate whether such an attacker could defeat the proposed defense. If a weakness is found, then the system is regarded as absolutely insecure; if no attack can be found, then the system is deemed secure. Consequently, instead of looking at the average case, statements of security are only deemed reliable if they consider the worst case.

Digital *information* is information encoded in discrete numbers. There are conventions which define the mapping between pieces of information and their digital representation. Standard mappings exist for different types of information. For example, text is composed of letters which can be individually mapped according to the ASCII table; images are matrices of measures of light intensity expressed on a scale from 0 to 255; sound waves are sampled at constant intervals and stored as a time series of measurement points. Likewise, an electronic stock exchange encodes the instruction to buy some stock at a certain price in a precisely defined tuple of numbers.

The digital representation of information creates several profound implications. First, it is practically costless to create perfect copies of information. Second, the information can be transmitted anywhere immediately. These two implications have transformed information industries such as publishing and journalism by uprooting business models. Third, information can be remembered indefinitely. Combined with advances in computation, this has made it possible to maintain detailed records of economic transactions and use the past to better inform future behavior. Finally, digitally encoded information lacks provenance. Unlike physical documents, where changes can readily be seen by markings, any modifications to digital information cannot easily be detected by simply inspecting the encoded data.

Information security is the endeavor to achieve *protection goals* specific to information. While there is no globally consistent ontology of all conceivable protection goals in information security, a broad consensus has been reached for a triad of fundamental protection goals: confidentiality, integrity, and availability.

Definition 1. *Confidentiality* means that information is accessible only to authorized parties.

Consider a stock market with electronic transactions. Suppose that a stock broker wishes to issue the following transaction: (BUY, 200 shares, GOOG, \$600.25). The broker may wish to keep this information private, so that only the exchange knows of the transaction. Hence, to achieve the goal of confidentiality a system must ensure that only computers run by the exchange are authorized to view the details of the transaction.

Note that confidentiality does not cover prior knowledge. If the broker emails a colleague to express his intentions to purchase the stock, then the goal can still be met even if the colleague knew in advance the information to be transmitted.

Furthermore, any breach of confidentiality is virtually impossible to undo. Once information has been disclosed to unauthorized parties, there is no available mechanism to retrieve the lost information and ensure that the unauthorized party no longer has the information. This is precisely why confidentiality breaches can be so damaging to affected firms.

Definition 2. *Integrity* means that modification of information can be detected.

Continuing with the stock purchase example, a malicious party might wish to modify the transaction en route to the exchange in order to gain financially. For example, the malicious party could try to lower the transaction price to (BUY, 200 shares, GOOG, $550.25), for a profit of $20 000. To counter such attacks, the exchange must devise ways to automatically detect that the information on the transaction has been altered.

Integrity does not, however, imply that incomplete or adulterated information can be replaced or corrected. Instead, integrity simply ensures that the fact that something is missing or incorrect can be reliably determined. Furthermore, integrity must be understood in the sense of unaltered information, not validity in terms of absolute correctness. For example, an integrity mechanism for stock transactions need not verify that the stock price actually correspond to the offer price when executed. Instead the mechanism could simply check for modification of the original order. In other words, whether the encoded information is an accurate representation of a phenomenon existing in the real world is beyond the remit of a system ensuring information integrity. Thus, this more limited definition conveniently avoids having to deal with a fundamental epistemic puzzle.

Definition 3. *Availability* means that authorized parties can access information (and use resources) when and where it is needed.

Availability is a fundamental requirement for any electronic stock exchange. Its systems must be designed so that incoming transactions can be received and processed in a timely manner from all prospective traders. Otherwise unfair transactions may proceed. For instance, an adversary who can temporarily block purchase transactions from some traders can make her own purchases at the lower price before others drive up the asking price.

Unlike confidentiality and integrity, availability often requires guarantees for more than just the information itself. It may also require that a functionality be available, e.g., the stock exchange's ability to accept transactions. Hence, availability guarantees need more precise specification on when and where the functionality is needed. Also, any reasonable definition for availability of information implies integrity, because otherwise any random source emitting data is sufficient to fulfill a narrower defined protection goal.

In the definitions above we refer to *authorized parties*, which begs the questions who are the parties and how are they authorized. *Parties* can be thought of as human beings controlling computer systems, or programs acting on their

behalf. *Authorization* is the decision a principal, typically a computer system, must take in deciding whether the party is allowed to undertake the requested task. In fact, this authorization decision is the fundamental challenge of security engineering, since all resulting behaviors depend critically on getting the authorization decision correct.

Authorization is closely related, but subtly different, to the tasks of identification and authentication. Identification answers the question "Who are you"? Authentication answers the question "Is it really you?" Finally, authorization answers the follow-on question to authentication "Knowing who you are, are you allowed to do something?" It is a common mistake to conflate the three concepts: for example, engineers and policy makers often expect that deploying an authentication mechanism can automatically solve the authorization problem, when in fact the decision over whether to authorize authenticated parties is left unanswered.

Authorization decisions can be *explicit* or *implicit*. Explicit authorizations include access control, which restricts resources in a computer by design of the operating system (discussed in Section 3.2). Implicit authorization occurs in the use of cryptography, where not knowing a secret key precludes a principal from decrypting ciphertext (discussed in Section 3.3). Both are authorization decisions because they discriminate between authorized and unauthorized parties. See Box 1 for a detailed example of how identification and authentication mechanisms guide the authorization decision.

2 Computer Systems and Networks

Before we deepen the discussion of how protection goals for information, it is useful to recall selected design principles of the systems and networks used to store, process, and transmit digital information. These principles define what level of information security we can expect in the best case.

2.1 Computer architecture

It is always a long shot to summarize seven decades of research on computing architecture in a few paragraphs. Nevertheless, we do our best here by discussing four ideas that are fundamental to understand the limits of information security.

2.1.1 Code is data

Computers are machines that deterministically manipulate data by following the instructions laid down in a computer program. The mathematician John von Neumann influenced computer science as much as he did economics with his seminal works on game theory [16]. He invented a computing architecture that does not distinguish between instructions describing a computer program and the data it processes; in brief, *code is data*. Virtually all computing devices we interact with today follow this design principle. They have a fixed amount

of numbered memory cells to store digital information. The values stored in memory can be interpreted as program code, i.e., the instructions telling a microprocessor what to do. Examples instructions include adding the value of two memory cells, comparing them, or continuing the execution with the instruction at a specified memory address. Alternatively, the very same values can be interpreted as any other kind of encoded information that is processed as data by the program.

What are the economic implications of code being data? For one, the von Neumann architecture gives computers the flexibility of general-purpose machines. Hence, computers are cheap to repurpose, simply by loading the next program or dataset into memory. Using the language of economics, not only is the marginal cost of production close to zero, but so is the marginal cost of the *means* of production. While we might observe the first in the offline world at least approximately (e. g., printing an additional copy of a newspaper), it is much harder to come up with an example for the latter (an additional printing press costs a fortune). Finally, and extending the previous implication, programs can be designed so that they generate new programs. That is, the "data" output by one program is frequently another program. While such a capability might seem frivolous, in fact it is essential to making the interactive web we are accustomed to using. Furthermore, this characteristic is essential to modern development-tool chains, enabling automated reconfiguration and cheap reuse of ever more specialized software components. Taken together, these attributes unleash economies of scale and division of labor unprecedented in the offline world.

Von Neumann's architecture design also has security implications. Obviously, it is possible to formulate protection goals for computer programs like for any other kind of digital information. (If it is wise to require *confidentiality* of code is a different question; see Section 3.1.) However, the advantage of flexibility can turn into a security weakness if a computer program does not remain in complete control of its *integrity*. In a typical scenario, a user may be authorized to modify the data for a program (e. g., edit text in a word processor, upload an image to a web server), but he would not be authorized to change the program itself. Because code is data, and data is code, a malicious user might try to infiltrate the system with new program code initially disguised as data. This way, the instructions are stored in the computer's memory, potentially ready for execution. In many practical instances, even minor programming mistakes in the system's original program can lead to the execution of such infiltrated code and thereby allow an attacker to take control over the system. Once in control, a malicious program can leverage the fact that programs can generate new programs and modify the instructions of programs stored persistently on the system. This is what makes computer viruses so noxious.

2.1.2 Layers of abstraction

One of the fundamental concepts of computer science is *abstraction*, that is, specifying the meaning and behavior of software while hiding the details of im-

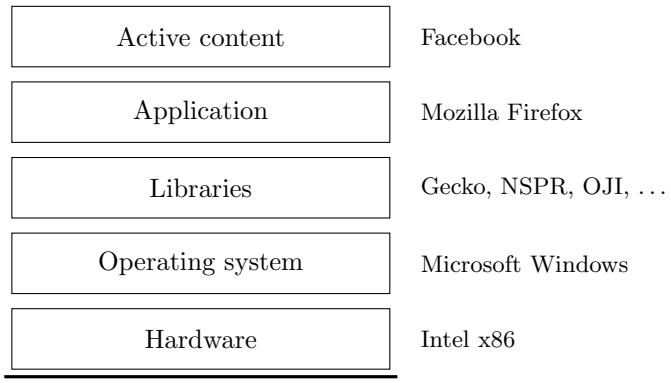

Figure 1: Layered architecture: simplified execution stack of a typical computer system with examples for using a web application. Higher layers cannot defend against security holes on lower layers.

plementation. Code that has been written to take advantage of abstraction can be easily composed and reused by others who remain completely ignorant to particular implementation details. Such so-called *modular* code makes it easy for developers to rapidly create sophisticated programs. As a result, the design complexity of the software running modern computer systems has skyrocketed. In this regard, software engineering is fundamentally different from mechanical engineering, where physical constraints limit the number of possible combinations in the design space.

To keep highly complex designs from becoming unwieldy, computer scientists often organize abstraction into a series of nested layers. This is reflected in Cambridge computer science legend David Wheeler's famous statement

> "All problems in computer science can be solved by another level of indirection."

Multiple layers of abstraction are typically arranged in a hierarchical execution stack, as depicted in Figure 1. In an ideal world, each layer offers a standardized interface to provide functionality for the next higher layer. In practice, the actual dependencies between layers can take more complicated forms.

Figure 1 shows that the operating systems abstract from the hardware. Therefore, higher layers sitting on top of the operating system do not need to know specifics of the hardware. This is why an application running on Microsoft Windows runs on many different types of hardware supported by Windows. Likewise, active content executed in a web browser runs in any browser regardless of the underlying operation system and hardware. Libraries and collections of libraries called frameworks are less visible to the end user, but they make a developer's life easier. For example, instead of implementing a Java virtual machine in every application that supports the Java programming lan-

guage, the Open Java Interface (OJI) is a reusable building block that can be included in many different applications. Similarly, libraries exist for frequent tasks like displaying images, reading and writing specific file formats, or doing cryptography. Remember that Figure 1 is but one example. Practical systems can have additional layers, such as virtualizations. Also the granularity of the execution stack may vary (e. g., one could include an explicit layer for the BIOS stored in firmware), but the principle remains always the same.

Abstraction brings the advantage of compatibility because higher layers only interact with the next lower layer and therefore are independent of all layers underneath. In other words, they trust that all lower layers behave as expected. But is this trust always justified? Here we have a security implication: higher layers have no chance of identifying malfunction at lower layers. Therefore, they cannot defend themselves against attacks originating at lower layers. This means that even the most secure encryption function, if implemented at the application layer, cannot keep secrets confidential from the underlying operating system and hardware! Whoever controls (i.e., programs) a lower layer, has full control over the higher layers.

In other words, if the operating system is programmed to learn (and leak) a user's Facebook password, it always can. There is no way to avoid this. If, however, the lower level can be made secure, then the higher level cannot compromise its security: a properly implemented execution stack prevents websites from learning the system password. (Except for user faults, like reusing the same password.)

Being completely at the lower layers' mercy becomes even more concerning if we increase the granularity of our view on the execution stack and recall that the operating system itself consists of many different layers, such as device drivers between the hardware and the actual OS. These layers are developed by many different parties. If only one of them has a security problem, it affects the security of the entire system. The same holds for software libraries, which are often developed without security in mind. Countless anecdotes recount how a single marginally important library has wrecked the security of large and widely used applications.

This one-out-of-many rule leads to another important notion in information security. Since it is sufficient to compromise one part of a system, a rational and resource-constrained adversary will target the weakest part; figuratively the *weakest link*, using the analogy of a chain.

Another way to look at lower layers determining the security of higher layers is to frame it as *vertical propagation* of errors and attacks. This picture is only complete if we assume that the software *development processes* of all involved layers are secure. In fact, modern software development tool chains are highly automated and use many specialized applications. First and foremost, editors enable users to write computer programs in source code. Then, compilers translate source code in human-readable programming languages to binary machine code instructions. Linkers merge machine code from libraries with the code of the actual software. Source-code-management systems keep track of different versions and enable collaborative development between hundreds of

programmers. Code generators produce template source code for commonly used functionality. And integrated development environments provide a graphical user interface to manage all these tasks. It is important to recall that all these development tools run on the application layer of a, possibly distributed, system. If any of these tools malfunctions in a way that the integrity of the resulting output is violated, the error propagates *horizontally* along the tool chain and into the software under development. This threat is not only theoretical; in 2003 it was discovered that the source code repository of a popular operating system has been compromised. The incident remained undetected for several months.

Therefore, to have full confidence in the security of a particular piece of software, we have to ensure that at least the same level of security is maintained in *all lower layers* of the system running the software and *all preceding stages* of the software-development process; and, by extension, all lower layers of all systems used for software development. It is easy to see that this level of confidence is very hard to reach in practice.

We summarize the above discussion by amending Wheeler's statement:

> "All problems in computer science can be solved by another level of indirection, *except security problems*."

This highlights that security is different from many other fields of computer science.

2.1.3 Moore's law

Intel co-founder Gordon E. Moore published a paper in 1965 where he observed that the the density of integrated circuits seemed to be doubling each year with striking regularity [12]. He then boldly predicted that the doubling could continue for ten years or longer. Higher density circuits are more powerful circuits in terms of computation.

Unlike most predictions about the future, Moore's proved mostly right. Computer performance has doubled roughly every 18 months, not just for the 10 years following the publication of Moore's paper, but through today. One explanation for why the prediction was correct is that the prophecy was self-fulfilling. In effect, Moore was setting a target for companies like Intel to aim for.

The implications of this exponential growth in computing power has been transformational. Computers have shrunk in size and cost, as well as grown in capability. Furthermore, the expansion of digital storage capacity has also grown exponentially. This has enabled the widespread digitization of information that would have otherwise been unthinkable even a generation ago. The advances in processing mean that analyzing the resulting data is now more feasible than before.

The security implications are that the success of the digital transformation has created liabilities that previously did not exist. Paper records cannot be

efficiently searched and accessed, but they also cannot be remotely manipulated or deleted by a malefactor.

Another security implication is that any system whose security relies upon computational intractability must now put an expiration date on the security guarantee. As time passes, it is likely that something which is too expensive to carry out will become affordable in a few years' time. For example, trying out all the possible keys to unlock an encrypted message might be infeasbible on today's computers. Suppose for argument's sake that guessing all the keys would take 4 years if begun today. If processor speed doubles every 18 months, then after only 6 years the computations would take just three months to finish. 12 years from now the computations should take less than a week to find the secret key, while after 24 years it should take around half an hour.

2.1.4 The halting problem

Given the exponential growth in computing power put forth in Moore's law, one might be forgiven for expecting that all computing problems can be solved by simply waiting a few years. In fact, some classes of computing problems are so fundamentally hard that even huge rises in computing power will not help.

One famous instance of an unsolvable problem is the decision whether a computer program terminates or not. More importantly, the proof by Alan Turing has substantial security implications: it is impossible to write a program that analyses the code of another program and, in general, make reliable predictions about its behavior when executed. Consequently, it is generally impossible to automatically distinguish between benign and malicious software from analyzing the code. Therefore, to have confidence in the behavior of software, the whole development process has to be known. As not all stages of this process are sufficiently formalized to enable automated verification (so-called *model checking*), gauging the security of moderately complex software involves tedious and fallible manual code reviews. This asymmetry between the effort needed to design complex systems (very little because of automated development-tool chains) and the effort needed to assure their security (tremendous) is an important obstacle to security in practical systems.

The halting problem should inspire some humility for any designer of technology that aims to automatically audit code for security vulnerabilities. Most security holes in code are much more subtle to identify than determining whether the code will finish executing. In fact, many such holes can also be shown to be impossible for a computer to decide. This does not mean that all code audits should be abandoned. Rather, it means that we must accept that no code audit is perfect, and none will ever find all the security vulnerabilities that may be present.

2.2 Network architecture

We present core principles of network architecture using the Internet as the primary example. We note that many concepts discussed here in the context of

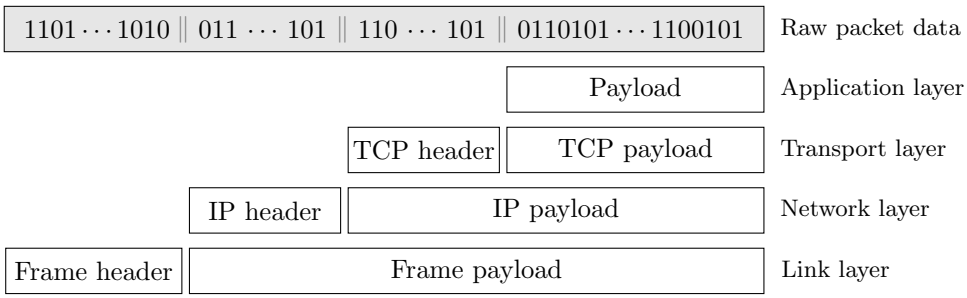

Figure 2: Layers of the network stack define the nested composition of a network packet (here for example: TCP/IP)

the Internet generalize to other kinds of networks.

Early computers were designed as standalone devices. They ran applications which performed a task on some data stored on the device, such as solving series of mathematical equations or presenting text in a word processor. Even when personal computers gained popularity in the 1980s, their primary function was to carry out localized tasks. Data and code were exclusively exchanged by carrying around physical storage devices such as floppy disks. Yet as we can see so readily now because of the Internet, there is enormous value in configuring computers to communicate with each other directly.

Beginning in the 1970s, researchers began developing a series of *protocols* that enabled computers to communicate. Communications protocols define rules for exchanging information between computers. What matters most here is that all computers use an agreed upon standard, so that each can readily interpret the messages sent to each other.

The protocols comprising the Internet adhere to layers of abstraction as explained in Section 2.1.2, and they are collectively referred to as the *network stack* (see Fig. 2). Implicit in the design of these protocols are several key decisions that affect the Internet's operation to this day. At the lowest level is the physical layer, which specifies how bits are transmitted over a communications channel. This is naturally different for communication over, say, copper wires, than it is for wireless radios. Regardless of how communication is established at the physical layer, it next interfaces with the data link layer, typically Ethernet. This layer enables to address machines connected in a local area network (LAN).

On top of the data link layer is the network layer, which specifies how computers can communicate across networks. The Internet Protocol (IP) specifies how to include a source and destination address. In IP version 4, addresses are globally unique 32 bits numbers. This is important for two reasons. First, IP is configured so that computers are globally addressable – one only has to include the correct address and the message should be delivered. Second, while 2^{32} addresses (approximately 4 billion) seemed like an inexhaustible resource in the 1970s, it is now recognized to be far too small given the Internet's popularity

today. Consequently, a new protocol called IP version 6 has been developed, which includes 128-bit addresses (capable of supporting around 10^{38} addresses). However, IP version 6 has experienced slow adoption, and IP version 4 is still most widely used.

Above the network layer lies the transport layer, which enables end-to-end communications between devices on a network. For example, the Transmission Control Protocol (TCP) establishes a connection between computers at different IP addresses so that a series of messages can be sent. TCP ensures that messages are received by the recipient by keeping track of which messages have been sent and their order. In the event of a failed transmission, the message can be re-sent. By contrast, User Datagram Protocol (UDP) does not bother with establishing a connection, which means that it cannot determine whether a message was actually received at its destination. UDP is useful in cases where dropped messages do not need to be retransmitted, which is more common than one might expect. For instance, video is often transmitted over UDP, since resending data about a missing frame from 2 minutes earlier is not particularly useful.

Finally, on top of the transport layer is the application layer. Here we see lots of different protocols, ranging from HTTP (the protocol of the web) to SMTP (the protocol for sending e-mail).

In some ways, Internet protocols are a great equalizer. Computers that can communicate using these protocols can interact with any other computer using the protocol. For example, the servers that host web pages are simply general-purpose computers running additional software that includes instructions on how to respond to HTTP requests originating from other computers. This can have negative security implications, since criminals often compromise the computers running web servers by exploiting vulnerabilities in the software running on the servers, just as they would target an end user's computer. Naturally, criminals might value a web server for a popular website much higher than they would a home user, since this gives them an opportunity to compromise the communications between the server and its many visitors. Nevertheless, the common software underscores how correlated vulnerabilities can be within the computing infrastructure.

Messages are sliced into *packets*, which are simply groupings of bits. The bits in packets can either be treated as "headers" (specifying protocol information) or "payloads" (specifying data). Packets are sent between end-user computers via special intermediate computers called routers. Routers claim responsibility for delivering packets to a range of IP addresses within its own purview or "network". They announce their ability to deliver packets to other networks via routers using the Border Gateway Protocol (BGP). Other routers receive the announcements and update their own routing table with the new information if they can reach the addresses by taking a smaller number of "hops" than before the announcement. These routers process incoming packets by inspecting the IP headers to locate the destination address. They then look up in a routing table for the "closest" router to the destination to pass along the packet. Consequently, the Internet's global addressability is achieved through a bottom-up,

decentralized process.

It is also worth noting that routers do not make any formal guarantees that they will deliver packets to destinations. Instead, routers are merely expected to make a "best-effort" to deliver incoming packets.

To arrive at standards for communication, one needs standards bodies to coordinate decisions. In the case of the Internet, design decisions were kept informal and created outside the existing state-dominated standards bodies. Internet protocols were first designed and proposed in documents called RFCs (Requests for Comments) organzied by the Internet Engineering Task Force (IETF). These documents reflected rough community consensus, where the community was the group of engineers who were first interested in designing the Internet. Proposed standards came into force when software implementing them proved useful and was widely deployed. Consequently, the Internet's design usually reflected the values of this sometimes eccentric group. It was designed to be inherently decentralized and respectful of other networks' autonomy.

The Internet's designers also assumed that all network participants are basically good and trustworthy. This greatly simplified protocol design. No means of authentication was required, apart from being physically connected into the network. The information contained in packets was assumed to be accurate, so constructing routing tables from announcements was straightforward.

As we can now plainly see, the Internet's design often frustrates those who take a more paranoid view of the world than the Internet's creators did. Most obviously, the lack of authentication has enabled "spoofing" attacks, where malefactors can pretend to be other computers by changing the source identifier. Combined with information's inherent lack of provenance, attributing misdeeds to perpetrators is very difficult.

But there are more subtle implications for security as well. Best-effort delivery of packets makes it hard to distinguish between an honest router that has been overloaded and a compromised one that is deliberately wreaking havoc. Global addressability has brought enormous benefits by connecting distant parts of the globe together, but it also makes it far easier to carry out criminal acts from remote jurisdictions. Every computer with an IP address is exposed to packets from all over the world – in other words, there are no safe neighborhoods on the Internet.

Finally, the decentralized nature of the Internet makes imposing security precautions and changes very difficult to carry out. Any changes to the protocols must be developed by consensus, and they often only work if all independent parties agree to adopt the changes. Coordination is often impossible, as evidenced by the extremely low adoption rates of IP version 6, many years after its introduction.

3 Engineered Defenses to Achieve Protection Goals

As computer systems and networks were developed, it was recognized that additional effort would be required to ensure the security of these systems. We now

describe how computer systems have been engineered to achieve the protection goals of confidentiality, integrity and availability.

3.1 Threat models

Security engineering begins with a recognition that all security is relative. In particular, security must be defined in terms of the goals, knowledge and capabilities bestowed upon adversaries. These assumptions about adversary behavior are codified in a *threat model*.

The *goal* of an attacker could simply be to disrupt the protection goals of defenders. Nonetheless, security engineers like to focus on a narrower set of adversary goals. For instance, a profit-motivated adversary's goal is to make money, regardless of target. The goal of a terrorist, on the other hand, is to cause disruption. A government-sponsored hacker's aim may be to gather intelligence by harvesting communications by other nation-states. In each case, the adversary has an implicit utility function that she attempts to maximize. Security engineers typically do not model the utility function explicitly; rather, they envision malicious behaviors that are consistent with stated goals.

When looking at many heterogenous potential targets of attack, the following distinction of adversary goals has been established in the security economics literature. We call an attacker *targeted* if she is only interested in attacking the specific system under consideration (i.e., her utility is zero for all other potential targets). By contrast, *opportunistic* attackers gain utility from successfully attacking any target and thus might choose to concentrate their efforts on the weaker targets.

A second aspect of the threat model is the extent of *knowledge* assigned to the adversary. While the specifications of a computer system or communications protocol may be kept secret as a precaution, most threat models assume that adversaries know as much about how a system works as do its designers. This deeply-ingrained assumption can be traced to the 19th-century *Kerckhoffs' Principle* [8], which asserts that the security of a system should not depend upon its design remaining secret. One motivation for cryptographers to adhere to this principle is so that their systems can accommodate a stronger threat model where adversaries possess extensive knowledge. See box 2 for a discussion of how Kerckhoffs' Principle has affected cryptographic design.

The final aspect of a threat model is the set of *capabilities* ascribed to an adversary. These include assumptions about the extent of computational power available, ranging from a single PC to a supercomputer or a network of many thousands of machines. The time available to an adversary in targeting the defender is another consideration, ranging from a single point to all time. The adversary's distribution also factors in, particularly with respect to the ability to observe communications. Some attackers are assumed to be local, only capable of observing communications on a single channel. By contrast, the most capable adversaries are global; that is, they can observe all communications on a network. Finally, an important distinction of capabilities is whether the adversary is *active* or *passive*. Passive adversaries merely observe communications,

while active adversaries can intercept and modify communications en route.

We can conveniently summarize each of these capabilities by making an assumption about the financial resources of the adversary. We could then leave the adversary to devote resources to different capabilities based on her goal.

3.2 Access control for system security

One broad aspect of security engineering is to protect computer systems from misuse. Once the authorization decision has been made, systems security boils down to *access control*: ensuring that the authorized user can access and modify only those resources to which he is entitled.

The general idea is to enforce access control for any given layer of a system architecture on a lower layer. Web browsers enforce access control between web pages and their active content, the operating system (OS) enforces access control between users and their applications, and the hardware supports access control between parts of the operating system.

In particular, most modern operating systems separate the processes that run the operating system from the processes run by users. This is an artificial barrier in computer architectures, since code and data need not be differentiated at the physical layer. Consequently, the separation can be achieved with a superuser who controls system resources, can install programs and so on. User data, meanwhile, is kept separate, and at a lower privilege level. The data and processes for each user are also kept separate by the OS, which can typically only be accessed by the superuser.

Of course, there may be circumstances where users on a system wish to share resources. This can be achieved by setting access control policies. Under *mandatory access control*, decisions about which processes read and write to others is set by a system administrator, typically in accordance with a security policy. For example, in a system hosting data at different classification levels, the security policy might specify that classified processes can read data at the classified and unclassified level, but only write to the classified level in order to prevent information leakage to unclassified sources. Likewise, an unclassified process may be allowed only to read and write to unclassified data. The job of the operating system is to assign levels to processes and data, and enforce the specified policy. In other words, mandatory access control takes a "top-down" approach to access control policy: a consistent policy is applied throughout to all system resources.

Unsurprisingly, centralizing access control decisions can lead to inflexible systems, where cooperation among users is stymied. *Discretionary access control*, by contrast, lets users set access controls directly in a "bottom-up" manner. Consider the approach taken by Unix-style operating systems (including Linux and Mac OS). Files are assigned an owner and group (potentially consisting of multiple users). Users can be given permission to read, write and/or execute each file. The file owner can set these permissions for herself, and separately for the associated group and other users. For example, a text document might

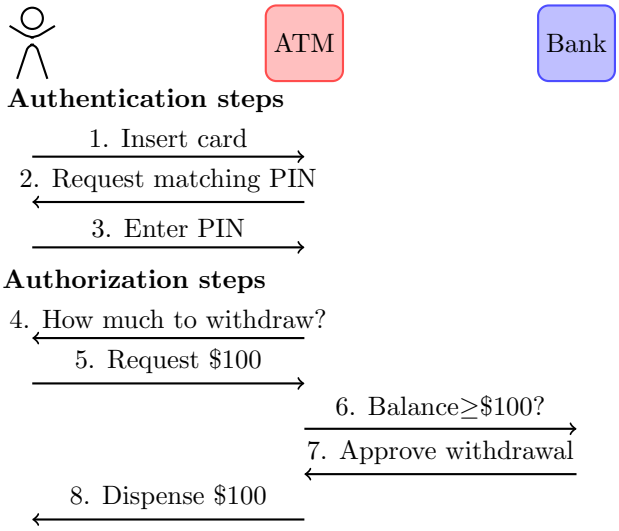

Figure 3: Authentication and authorization example using an ATM.

have read and write permissions for the file owner, but only read access to the group and no access at all for others.

The key point is that the file owner gets to decide what the permissions should be. This allows much greater flexibility in setting access control policies when compared to the mandatory approach. However, it also makes enforcing consistent policies much harder, and makes it essential that user accounts be protected from takeover by adversaries. While mandatory access control has been adopted in some specialized circumstances (e.g., military applications), most consumer- and business-focused operating systems have adopted some form of discretionary access control.

Regardless of the approach to access control taken by the operating system, secure systems should be engineered in accordance with the *principle of least privilege*. The principle states that any file or process should be assigned the minimum level of permissions needed in order to complete the required task. The principle naturally conflicts with the desire for systems that are easy-to-use and adaptable to changing circumstances.

Moreover, assigning up-to-date permissions is hard. As users' roles change, their need to access and modify system resources also changes. It is much easier to grant access to new resources than it is to remember to revoke access to resources that are no longer needed.

How computers identify human beings

Box 1. Computer systems authorize users to access protected resources. But before making any authorization decisions the system must first determine which user it is interacting with. To identify a user, an authentication mechanism is used. Computer systems associate specifically identifying information, so-called *attributes*, with a claimed identity, such as by storing the pair (ID, attribute value) in a database. Upon first encountering a user, the system will prompt the user for her associated identifier and attribute.

The identifier is assigned in a way that can be uniquely represented in the computer system (e.g., assigning a unique number rather than relying on the stated name). While the associated attribute can be anything known to the person, typically the attribute value is kept secret so that it can only be known to the person and the computer system (e.g., a password or PIN). If the attribute value is only known to the user and computer system, then the system can be confident that the person providing the attribute value and the associated identifier is the one who is expected.

In addition to associating a secret with an identifier, systems can store attribute values about the person that can be verified by a computer. Such "biometrics", ranging from fingerprints to iris scans, can usefully distinguish between individuals. Unfortunately, verifying these attribute values can be quite difficult in practice, since the sensor collecting the information may be compromised or fooled. Furthermore, even when the verification works well, it is important to remember that such biometric information is not sufficient on its own to identify people to computer systems. An associated secret is still required, since the biometric information can be easily copied and used by anyone possessing the data.

While those operating computer systems are primarily concerned with identifying people in order to make authorization decisions, they should not neglect the equally important task of people authenticating the computer system. Because authentication mechanisms exchange secret information between users and computers, adversaries may try to trick users into sharing the secret with impersonating devices.

To better understand the authentication and authorization phases, we can step through an example all readers should be familiar with: withdrawing cash from an ATM. Figure 3 presents the steps in sequence. First, the user arrives at the ATM and inserts his bank card into the machine. Reading the identifier off the card, the ATM then prompts the user for the PIN associated with the card. If the user supplies the correct PIN, then he has been successfully authenticated. This sets off an authorization phase. The ATM first asks the user for information to aid the authorization decision: how much money to withdraw and from what account. The ATM passes this information back to the bank, which confirms whether there is enough money in the account to cover the withdrawal request. If so, then the ATM dispenses the cash and the transaction is complete.

ATMs have long been targeted by financially-motivated criminals. Consequently, we can learn quite a bit about how adversaries circumvent the iden-

tification, authentication and authorization process by taking a closer look at ATM fraud.

Attackers can exploit the authentication or authorization phase. When attacking authentication, adversaries can either target how banks authenticate customers or how customers authenticate banks. The former approach is more common. Many banks allow customers to select their own PINs. Given the chance, people often choose PINs poorly. They select PINs that could be easily guessed, from 1234 to anniversary dates and birth years. Researchers estimate that a customer-chosen PIN could be correctly guessed after stealing just 11 cards [3]. Another common approach is for criminals to install "skimmers" on ATMs, which copy card details as the card is inserted to the ATM [13]. The PIN is retrieved by placing a pin-hole camera on the same ATM. In both cases, authentication of customers is broken by stealing the PIN associated with cards.

A less common tactic is for criminals to deploy fake ATMs designed to harvest credentials [18]. Such scams exploit how people "authenticate" ATMs. More precisely, this approach takes advantage of the fact that people do not authenticate ATMs apart from determining that they are physically present. Note that deploying fake ATMs is comparatively rare, due to the expense and risk of placing new ATMs compared to installing skimmers. However, attacks on how customers authenticate banks are far more common in the online space where bank storefronts are simply digital information that can be easily copied.

Attacks can also exploit weaknesses in authorization mechanisms. For instance, in one large ATM authorization failure, miscreants managed to alter cash withdrawal limits, extracting $13 million from just 21 pre-paid debit cards [10].

3.3 Cryptography for communication security

Protecting information stored on a computer system is a necessary, but not sufficient, step towards meeting the protection goals of confidentiality, integrity and availability. Security engineers must also devise mechanisms to protect the communications *between* systems over computer networks. Cryptography has been used to protect the confidentiality and integrity of communications for millennia, and it remains essential for protecting information systems today.

When discussing cryptography (and indeed all of communications security), it is customary to name the actors involved. The two parties communicating are called Alice and Bob. The adversaries get names too: Eve is trying to passively eavesdrop on the communication between Alice and Bob, while Mallory tries to actively interfere by intercepting, manipulating and blocking communications. We will learn more about Eve and Mallory's exploits in the next section, but for now we will focus on Alice and Bob.

The original, untransformed message Alice wishes to send to Bob is called *plaintext*, while the encrypted message is called *ciphertext*. *Encryption* turns plaintext into ciphertext, while *decryption* does the reverse, turning ciphertext

Plaintext:	`THISISIMPORTANT`

Caesar	
Secret key:	`DDDDDDDDDDDDDDD`
Ciphertext:	`WKLVLVLPSRUWDQW`

Vigenère	
Secret key:	`DABDABDABDABDAB`
Ciphertext:	`WHJVITLMQRRUDNU`

One-time pad	
Secret key:	`DABHJIZXEBTULQP`
Ciphertext:	`WHJZRAHJTPKNLDI`

Figure 4: Symmetric encryption of the message 'THISISIMPORTANT' using Caesar, Vigenère, and one-time pad ciphers.

back into plaintext. *Cryptanalysis* is the study of breaking cryptography, where one attempts to reconstruct plaintext from ciphertext without access to any secrets shared between Alice and Bob. Meanwhile, cryptography covers both the design of encryption schemes and cryptanalysis.

There are two broad categories of cryptographic mechanisms: symmetric and asymmetric cryptography. We first discuss symmetric encryption, which has been in use far longer. The simplest encryption mechanism is the Caesar cipher, named after Julius Caesar, who used this technique to encipher messages sent from Rome. In the Caesar cipher, plaintext is encrypted by incrementing each letter by the same amount. Julius Caesar preferred to increment letters by 3. For example, to encipher the plaintext 'ET TU BRUTE', one would transform the 'E' to 'H', the 'T' to 'W' and so on, so that the ciphertext becomes 'HWWXEUXWH'. Upon receiving the encrypted message, Brutus can decrypt the message by decrementing by three letters: 'E' is three letters before 'H' and so on. (The cipher does nothing to spaces, which is why they are removed before transmission.)

From this simple example, we can already observe a key property of symmetric encryption schemes: the sender and receiver must have shared a secret in advance. In Caesar's case, the secret is rather simple: to advance the letters by three, rather than four or twelve.

In the 16th century, the so-called Vigenère cipher was developed, where a more sophisticated secret was shared between sender and recipient. Instead of transforming each letter by the same amount, the sender uses multiple Caesar ciphers in sequence, transforming letters by a different amount at each point.

To remember the sequence, a word could be used, indicating how many letters to transform each time. For example, to transform a letter by 3, as in a Caesar cipher, this corresponds to the letter 'D'. If the secret is 'DAB', then the first letter is shifted by three, the second letter shifts by zero, and the third by one, and then the process repeats (fourth letter by two, etc.).

Figure 4 demonstrates how the plaintext 'THISISIMPORTANT' can be encrypted using both the Caesar and Vigenère ciphers. Figure 4 also demonstrates the one-time pad, which takes Vigenère ciphers to the logical extreme. With a one-time pad, the secret is now as long as the message itself. One-time pads offer perfect secrecy, but it is easy to see at what cost: the secret shared between Alice and Bob is now as big as the plaintext they wish to share. Russian spies during the Cold War were known to use one-time pads with keys written on tiny books. As we will see when we discuss cryptanalysis in the next section, one-time pads only offer perfect secrecy when the key is only used to encrypt plaintext once.

Symmetric encryption schemes remain in widespread use today. The most widely used mechanism is called AES, the Advanced Encryption Standard. While the mechanism for using a small secret to encipher plaintext has gotten a lot more complicated since the days of Caesar, the same principle applies. Instead of the shared secret being a word, it is now a string of random characters measured in bits. For example, AES-256 uses secret keys that are 256 bits long, which literally means that the secret is a sequence of 256 0's and 1's when represented in binary form. As with all symmetric cryptographic mechanisms, the limiting factor remains distributing a shared secret between sender and recipient so that both parties can encrypt and decrypt messages.

In the 1970s, a huge breakthrough was achieved with the development of *asymmetric cryptography*. Unlike in symmetric cryptography, where a shared secret key must be exchanged before communicating, with asymmetric cryptography the communicating parties no longer need to share a secret in advance. It is not hyperbole to claim that the development of asymmetric cryptography was as essential as the emergence of the Internet in enabling e-commerce. Why? Asymmetric cryptography is what enables customers to send their credit-card details to Amazon in encrypted form without ever having physically gone to Seattle.

How does it work? The critical insight is to use one-way functions that are easy to compute but difficult to reverse without access to a secret. Rather than use a common key shared between sender and recipient, the recipient creates a *key pair*: a public key for encrypting messages and a private key for decrypting messages that have been encrypted with the public key. The public key can be broadcast for the world (including Eve) to see. So long as the private key is kept secret, only the person holding the private key can decrypt messages.

For the mathematically inclined, here is a simplified explanation of how RSA, the most popular asymmetric encryption function, works. There are several subtleties that we omit for the sake of brevity. The clever insight of RSA (and indeed all asymmetric cryptographic primitives) is how the key pair is created. RSA exploits the fact that it is very difficult to factor the product of two large

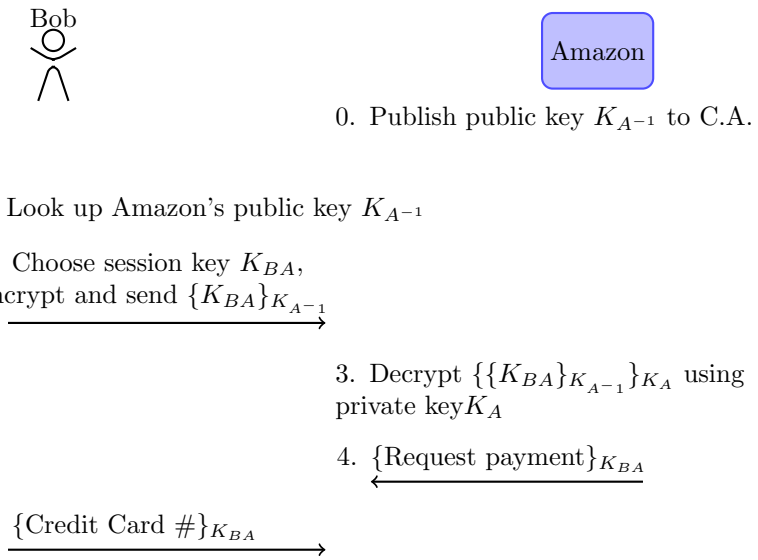

Figure 5: Using asymmetric cryptography to exchange a secret key to use in subsequent symmetric encryption.

prime numbers, where large means several hundreds of decimal digits. To create a key pair, one must first choose two large prime numbers p and q, and keep them secret as one's private key. The public key is given by the product $n = p \cdot q$ and another randomly selected number e that does not share a factor with p or q. To encrypt plaintext M, one has to compute $C = M^e \mod n$. To decrypt ciphertext C, one performs the reverse operation on the ciphertext $M = \sqrt[e]{C} \mod n$. The trick is that finding the e-th root modulo n is easy to carry out *only* when p and q, the factors of n, are known (i.e., by possessing the private key).

More important than understanding the underlying mathematics is to see how asymmetric cryptography works in practice. Figure 5 shows what happens at a high level when Bob tries to buy something from an e-commerce site, say Amazon. Well before Bob has arrived at Amazon, Amazon has created a key pair and published the public key to a Certificate Authority such as Verisign. When Bob arrives at Amazon, he looks up Amazon's certificate to learn Amazon's public key $K_{A^{-1}}$. While in principle Bob could encrypt all his communications with Amazon's public key, in practice he does not do that. Instead, Bob chooses a secret key K_{BA} to be used in a symmetric encryption session with Amazon. He then encrypts the key using Amazon's public key (represented by $K_{BAK_{A^{-1}}}$ in the figure).

Upon receiving the message, in step 3 Amazon uses its private key to decrypt the message and recover the secret key. Finally, in steps 4 and 5 Bob and Amazon exchange encrypted communications using their newly shared secret

key. This is roughly what modern web browsers do behind the scenes using the SSL protocol any time the user visits a URL beginning with 'https'.

While the cryptography examples discussed thus far have concentrated on ways to protect the confidentiality of communications, asymmetric cryptography can also be used to protect the integrity of communications. All one needs to do is swap the encryption and decryption roles of the keys in the key pair. In the example above, the message recipient (say Alice) distributed her public key so that anyone can encrypt communications but only Alice can decrypt the messages. If instead, Alice wrote a message and encrypted it with her private key, then anyone (including Bob) can decrypt the message using her widely disseminated public key. However, Bob can be assured that only Alice could have encrypted the message, as she is the only one possessing her private key. Encryption used this way is referred to as digital signatures. In fact, digital signatures are better than their traditional counterparts, since each signature is unique and unforgeable, provided that the private key is not compromised.

Kerckhoffs' Principle and the design of cryptosystems

Box 2. While this often comes as a surprise to those new to information security, most encryption schemes are in fact publicly disclosed. There are a few reasons for this. The foremost reason comes from *Kerckhoffs' Principle* [8], which asserts that the security of a system should not depend upon its design remaining secret. Cryptographers typically assume that capable adversaries could learn how their encryption schemes work even if they tried to hide the design. Consequently, most encryption mechanisms are set up so that only the key must be kept secret. This is advantageous because it restricts the amount of information that must be protected from an adversary. By contrast, in order to remain secure, a cryptosystem with a secret design would require that every implementation be kept hidden from an adversary. This can be very hard to ensure, particularly if many different hardware and software designers implement the system. Given the choice between an encryption algorithm that must remain secret and an equivalent one without such a requirement, the public algorithm is naturally preferred.

Second, public disclosure invites outside scrutiny. A publicly-disclosed encryption algorithm that has not been shown to be insecure inspires confidence in its use. Third, there are substantial network effects, since senders and recipients must agree to use the same encryption algorithm. Public disclosure makes encryption algorithms more widely available and therefore more attractive to prospective users.

In fact, governments often attempt to coordinate the selection of encryption algorithms. The US National Institute of Standards and Technology (NIST) has organized international competitions for selecting standards for different forms of encryption. For example, the Advanced Encryption Standard (AES) competition selected a symmetric encryption mechanism called Rijndael, developed by two Belgian cryptographers.

A consequence of this tendency towards public disclosure is that encryption is not normally a point of competitive differentiation among firms offering security services. There are some notable exceptions, however. For instance, the MIFARE contactless smart cards widely deployed in public transportation systems such as the London Underground rely on proprietary ciphers. Following deployment, the underlying encryption schemes have been shown to be easily broken [9].

While substantial mathematical insights were required to advance the state of cryptography in the 20th century, many of the challenges that remain are decidedly not mathematical in nature. Why does cryptography remain "hard" in practice?

First, key management is difficult to get right. For symmetric key encryption, distributing keys to all communicating parties can be burdensome. While asymmetric encryption appears to solve the key distribution problem on its face, it is not quite so simple. In particular, establishing a *public-key infrastructure* with authoritative key records has proven elusive in the decades since the development of asymmetric cryptography. Furthermore, ensuring that the right identity is associated with the presented key is difficult.

Second, configuring systems can be difficult. Often, parties must explicitly coordinate to share information, such as public keys, before secure communication may commence. Unfortunately, encryption mechanisms have long been notoriously difficult to use for all but the most technically savvy [17].

A third challenge is that cryptographic mechanisms are often brittle and do not fail gracefully. When there is a problem with a key, then all communications are unreadable. While this provides maximum protection to the confidentiality of communications, it can certainly hinder availability. Many firms and individuals who value confidentiality nonetheless prioritize availability of communications, and so may be reluctant to use encryption. For example, many of the routing protocols used to organize Internet traffic do not use encryption, even though enhanced versions of the protocols have been widely available for years. One reason network operators are reluctant to adopt the more secure versions is a fear that legitimate traffic would be disrupted if and when the decryption fails.

A final reason why cryptography is hard to get right is that the threat models considered by cryptographers often prove inadequate. Attackers often violate the assumptions made by cryptographers on how they would behave. We will discuss this in greater detail in Section 4.

4 Security Threats

Thus far we have described how computer scientists have designed computer systems and networks. We then outlined the technical mechanisms created to

defend computer systems and communications against attacks that undermine protection goals of confidentiality, integrity and availability. We now examine how things go wrong – how adversaries overcome defenses and exploit weaknesses in the protection mechanisms.

In each case, adversaries exploit failed assumptions. We first review what happens when attackers violate assumptions made by defenses. Attackers can also violate the assumptions threat models make about their own behavior and capabilities. Finally, we consider how violations of assumptions of how people and firms behave can help attackers.

4.1 System vulnerabilities: violating engineering assumptions

As explained in Section 2, the main way to defend computer systems is to restrict access to files and processes to only authorized users. Most operating systems distinguish between regular users and superusers, who can access and modify files of ordinary users. System vulnerabilities overcome the access control restrictions put in place, often by violating assumptions about how the system operates. The adversary's goal is frequently *privilege escalation*: taking on the role of superuser to carry out unauthorized actions such as installing malicious software or reading sensitive files.

Modern software is complex and inherently ridden with programming mistakes, called "bugs". Adversaries can exploit some of these bugs in order to carry out attacks. The most common mode of attack is to overwrite memory and execute unauthorized commands. Recall that under the Von Neumann computer architecture, there is no distinction between code and data in memory. Most computer programs accept user input, such as numbers or strings of characters, in order to complete their task. When executed, the computer allocates a limited amount of memory to store the input. When normal users interact with the program, they usually provide inputs that fit well within the space allocated. However, an adversary can provide a much longer input that exceeds the space allotted by the compiler. In this extra space, the adversary can include instructions to execute functionality with the privilege level of the program accepting input.

A similar principle applies to *code-injection attacks*. Here, attackers supply malicious code as input to web forms, which executes unauthorized commands on the web server even though they do not have proper authorization. For a more detailed explanation of how code-injection attacks work, see Box 3.

The ease with which adversaries can write exploits to execute programs at the same privilege level as another running program need not be a big problem on its own. However, the reason it is a problem is that software developers needlessly design programs to require execution as the superuser. Often it is marginally less convenient to write programs that do not require superuser privileges, but because the cost of increased vulnerability is borne by the user executing the program, many developers do not make the effort.

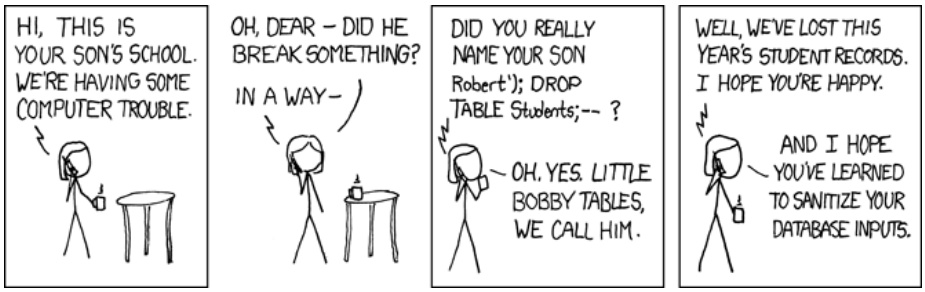

Figure 6: Humorous example of code-injection attack, courtesy `http://xkcd.com/327/`.

Finally, it is worth mentioning that not all vulnerabilities require privilege escalation.

Vulnerabilities are not merely a problem for computer systems in isolation. In fact, they often affect both end users and web servers, and spread between them. A common technique to compromise computers en masse is the so-called *drive-by-download*. Here, a web server is compromised so that the adversary can modify the HTML code presented to web visitors. The modified code surreptitiously links to an exploit that is run on a user's computer when she browses to the compromised web server. The unauthorized code usually exploits a vulnerability present in the browser software or a browser plug-in (e.g., Adobe Flash).

Note that in drive-by-download attacks (and indeed each of the attacks described in this section), encryption does not offer any added protection. This is because encryption solves a more limited problem: protecting the confidentiality and integrity of communications. Encryption is of no help when the code being executed is malicious, apart from hiding knowledge of the infection from eavesdroppers!

Input Validation Example

Box 3. See `http://www.thegeekstuff.com/2012/02/xss-attack-examples/` for a description of how cross-site scripting attacks work. (Note that reading this article is optional.)

4.2 Cryptanalysis: violating physical or mathematical assumptions

The primary goal of cryptanalysis is to descramble ciphertext without knowing the decryption key. How does cryptanalysis work? A straightforward approach

is to use *brute force*, trying all possible keys and checking if any return something intelligible. For a key of length ℓ bits, brute force takes on average $2^{\ell-1}$ attempts before being successfully deciphered; provided that the plaintext has enough redundancy (i.e., structure) to distinguish correct guesses from the random garbage produced by wrong keys. Consequently, adversaries try to identify shortcuts that enable smarter guessing so that the key can be recovered using fewer attempts. Formally, adversaries devise guessing algorithms so that only 2^k guesses are required, where $k < \ell$.

What form might shortcuts take? Consider the Caesar cipher described earlier that shifted letters by 3. Note that in English, the frequency of letters is distributed unevenly, with 'E' appearing a lot more often than 'Z'. If the adversary knew that the messages were encrypted using a Caesar cipher, then using brute force she would have to try all 26 shifts to see what works. By looking at the most frequently occurring letter in a long piece of ciphertext, the adversary can infer the amount of shifting (i.e., if she sees lots of H's in the ciphertext, then a shift of 3 would be a good guess).

Now that we know a bit about how cryptanalysis works, it can be helpful to think of the goal of ciphers in a new way: to make ciphertext appear as random as possible. It is impossible to create purely random ciphertext from a small key, but the best ciphers can do a good job approximating randomness for many different sources of input. Cryptanalysis, then, uses techniques to identify and exploit sources of non-randomness to make educated guesses about likely keys. This often requires observing lots of encrypted communications, perhaps by encrypting plaintext known to the adversary to identify patterns.

The strengths of attacks on cryptosystems are expressed in terms of the number of bits required to guess the key – k bits for 2^k guesses. So long as $k < \ell$, then an attack is deemed successful. This concise metric is perceived to be a good measure of the robustness of an encryption scheme. For example, AES-128 uses a key length of 128 bits. So by brute force a computer would have to try on average $2^{127} \approx 10^{38}$ attempts (roughly 100 times 1 trillion times 1 trillion times 1 trillion) to guess the key. Even if the security were shown to be far weaker (e.g., $k = 64$), this would still require approximately ten million times one trillion guesses, beyond reach from all but perhaps those in possession of supercomputers. Of course, the exponential growth in computing power means that previously unthinkable computations may eventually be possible. Consequently, recommended key lengths do slowly increase over time.

The preceding discussion included implicit assumptions about physical characteristics of computer systems. For instance, when calculating the time required to carry out brute-force cryptanalysis, we typically assume exponential growth in computing power consistent with Moore's Law. In fact, this growth could speed up or slow down. If it sped up considerably, due to an unforeseen technological breakthrough, then the security of symmetric encryption algorithms could be called into question. Similarly, theoretical cryptanalysis techniques for factoring large prime numbers that leverage quantum computing have already been proposed. If quantum computers ever materialize, public-key encryption algorithms based on the difficulty of factoring large prime number

are doomed.

Mathematical assumptions can also be violated to carry out cryptanalysis. For example, in RSA, it is assumed that the product of two large primes cannot easily be factored. In fact, it has been shown that some large numbers are easier to be factored than others. In theory, this could enable an attacker to decrypt encrypted communications.

Nonetheless, this is not where most realized attacks occur. In fact, this key-counting exercise is a good demonstration of how woefully inadequate traditional cryptographic measures of security from cryptography are in representing the true nature of threats. As Adi Shamir recounted [14] in a speech accepting the Turing award (roughly the Nobel Prize for computer science) recognizing his contributions to cryptography:

> Cryptography is usually bypassed. I am not aware of any major world-class security system employing cryptography in which the hackers penetrated the system by actually going through the crypt-analysis. [...] Usually there are much simpler ways of penetrating the security system.

We next review some of these "simpler ways" of defeating security mechanisms.

4.3 Violating assumptions about attacker behavior

Threat models help security engineers clarify the requirements for what their systems should protect against. Consequently, well-designed systems are usually protected against the attacks that have been accounted for in the threat model. An adversary, then, will naturally look for easier ways to disrupt the protection goals. Weaknesses are often found when the adversary behaves differently than has been assumed in the threat model.

Consider a castle designed to withstand a ground-based assault. The architect might include a moat and entry through a single, well-protected drawbridge. But what happens if the enemy can launch an aerial assault? The castle has not been designed to account for that expanded threat model, and so remains vulnerable to attack. Similarly, many attacks on information systems succeed when the adversary disrespects the assumptions made about his behavior by the system designers.

Threat modeling is a static estimation of adversary behavior that does not account for strategic behavior or dynamic interaction between attacker and defender. This approximation can go wrong in two ways. On the one hand, threat models can ascribe too much capability to an attacker or focus too much on a single method of attack, leading to "over-engineering" and over-investment against certain threats. On the other hand, threat models can miss attacks by not accounting for behaviors and capabilities, such as the castle designers not accounting for aerial assaults.

The threat model traditionally adopted by cryptographers suffers from both defects: overestimation of attacker capability in some areas while ignoring plausible strategies in other respects. The traditional threat model is for a nation-state adversary capable of recording large amounts of encrypted communications, and then using supercomputer resources to crack a key. In some cases this is a viable threat model, but often times this is unrealistic.

A single 1024-bit RSA key can be factored to obtain the private key from the public key with about 30 years of computational effort. Yet NIST guidelines recommend moving to 2048-bit keys [2]. In addition to protecting against adversaries with the capabilities of nation-states, another goal is to keep encryption unbreakable for many years to come. This is to protect against an adversary capable of observing all communications, storing them, and waiting until Moore's law makes the operations computationally cheaper.

While this may be good practice for very sensitive communications, it is likely far too cautious for most applications. Moreover, using longer keys imposes substantial costs on communications infrastructures. Web servers regularly transmit encrypted communications, and using 2048-bit keys will introduce delays to millions of communications. Due to these delays and added computational costs, many web services may decide not to encrypt more routine traffic, even if doing so would enhance consumer privacy and protect against certain types of attacks [6]. These opportunity costs are unfortunately not part of the calculation that goes into recommending suitable key lengths.

But perhaps the biggest limitation of cryptanalysis is its singular focus on decrypting ciphertext without access to the encryption key. This is a very costly task, even if cryptanalysis can reduce the number of guesses required to a manageable number. An attacker whose goal is to decrypt the ciphertext is free to do this by any means. Any rational attacker will see the long key lengths and seek out an easier way to decrypt communications. Consequently, adversaries frequently try to obtain the keys directly from the target, since protecting keys from accidental disclosure is hard.

Alas, stealing the key is outside the scope of the cryptanalyst's threat model, even though it is by far the most common way encrypted communications are decrypted. Why ignore this more realistic threat model? Because it is not mathematically "interesting". Instead, protecting the key requires an emphasis on operational and systems' security. It may also require understanding more deeply how people are duped into turning over their keys, which has social and behavioral explanations that lack the precision of mathematics favored by cryptographers.

Furthermore, a number of practical, assumption-busting cryptanalysis techniques have proliferated in recent years. For instance, many attacks exploit side channels – measuring side effects of the hardware carrying out cryptographic operations in order to make inferences about the composition of keys. Some attacks focus on variations in the time it takes CPUs to carry out computations, while others exploit differences in the power consumed. In some cases, a long period of observation is required in order to identify statistical patterns that correlate with certain operations. Other times, no fancy statistics are re-

quired. Encryption keys are stored in volatile memory (i.e., RAM), which has the property that when a computer is powered off the memory is cleared. However, security researchers have demonstrated that memory does not immediately clear after losing power, and in some cases keys can be recovered [15]. The time that the keys remain recoverable can be extended by exposing the memory to extremely cold temperatures [7].

4.4 Violating assumptions about defender behavior

Adversaries are not the only ones who behave differently than system designers expect. Regular computer users also make decisions that deviate from expectations and facilitate attacks. Instead of identifying sophisticated privilege-escalation attacks described in Section 4.1 to circumvent access control, adversaries can trick users into sharing their passwords. Fake dialog boxes and web pages can be set up to prompt users for their credentials. Because distinguishing genuine interactions from fakes can be hard in a digital environment, many such schemes succeed. One reason for their success is that system designers often only plan for their systems to authenticate users, but they fail to consider how users should authenticate systems.

System security is predicated on users being entrusted to only take actions in their own interest. While this might seem to be a reasonable expectation, in fact it can be difficult for users to know what a program actually does before it is executed. Deceptive programs could appear to do something useful, such as display a screensaver, when they in fact also do harm hidden in the background.

Unfortunately, one popular industry response to the presence of malicious software is to issue excessive security warnings, such as the User Account Control mechanisms first implemented in Windows Vista by Microsoft. The proliferation of security warnings habituates many users to ignore all warnings, since it would be impossible to get any work done if all warnings were heeded.

Finally, criminals employ the well-worn tactics of con artists to dupe people into doing what they want. We will discuss these in greater detail in Chapter **??**.

5 Countering Security Threats

Thus far we have described how computer systems and networks work, how they are protected against attack, and how these protections are in turn attacked. When weaknesses in defenses are uncovered and exploited by attackers, defenders have two choices. They can go back and make fundamental improvements to the defenses (e.g., develop provable compilers using model checking), or they can attempt to counter the attacks themselves. The latter approach is more reactive, and it is prone to trigger an expensive and potentially never-ending cycle of counter-attack and defense. Nonetheless, most of the information security industry has taken this reactionary approach, and so we now briefly review many of the countermeasures that have been developed.

We can group the countermeasures according to how reactive they are to security threats. *Ex post* countermeasures attempt to fix vulnerabilities as they are discovered and stop attacks as they are carried out. *Ex ante* countermeasures attempt to counter threats before they are carried out.

5.1 Ex post countermeasures

The first collection of technologies is most reactionary, in that they counter flaws or attacks as they are encountered. When a software vulnerability is discovered, the developer creates and disseminates a patch that plugs the hole. Most operating systems have developed capabilities to automatically distribute patches as they become available. Patch timing is not always straightforward, particularly for firms that must ensure that the patch does not interfere with other software. Chapter **??** reviews these trade-offs in much greater detail.

Another pervasive security countermeasure is antivirus (AV) software, which protects computers by looking for "signatures" of malicious code running locally on a system. Signatures match the binary code of known malware. There is an ongoing arms race between attackers and AV operators – attackers repackage the malicious code so that it appears different even when it has not changed operationally. This task can be easily automated. The AV operators gather and test newly observed binaries for malicious behavior, and then dynamically update the database of signatures. Unfortunately, theoretical results favor the attackers: Cohen showed that detecting viruses in general reduces to the halting problem [4]. Consequently, AV software is likely to always remain a step behind the malware writers.

In addition to monitoring computers for malicious behavior, one can use intrusion detection systems (IDSes) to look for attacks at the network level. Rule-based IDSes check for known attack patterns. Since attacks originate over the Internet, IDSes can look for dubious communication between devices. For example, an IDS could monitor for attempts to automatically log in to a remote computer using a password dictionary by flagging an excessive number of login attempts over a short period of time from the same remote IP address. A complementary type of intrusion detection system, called anomaly-based detection, looks for strange patterns in network traffic without relying on pre-defined rules of malicious behavior.

There are several impediments to the success of intrusion detection systems. First, the Internet is "noisy" in that there are lots of spurious packets flowing around, which makes deciding on what constitutes an attack hard. Second, whenever real attacks are comparatively rare, false positives can undermine the usefulness of detection. For example, suppose there are ten real attacks per million sessions. A false detection rate of 1% would imply that there are 1 000 false alarms for every real one. Finally, it is worth noting that IDS alarms are often counted as distinct attacks, thereby contributing to inflated reports claiming that organizations are subject to thousands or millions of daily attacks. Thus, imperfections in security technology compounds the difficulty of accurately measuring security incidents, which is a theme we return to throughout parts two

and four of the book.

Filters, such as firewalls and spam filters, offer a simpler network-level tool for blocking content deemed harmful. Filters operate at different levels of the protocol stack, depending on the sophistication of the filter. In general, filters operating at lower levels are simpler to configure but blunter as a tool for deciding what gets blocked. Filters operating at higher levels block more of the bad with less collateral damage, though at increased complexity and privacy risk.

Packet filtering operates at the IP layer (layer 3), and so it can selectively block by IP address and port number. For example, a firewall can deal with IP spoofing by allowing through only local IP addresses for outgoing messages, and only non-local IP addresses for incoming messages. Similarly, firewalls can choose to only allow through traffic on particular port numbers (e.g., allow incoming HTTP traffic on port 80). Using packet filtering, firewalls can reduce the harm emanating from infected machines. For instance, many firms block outgoing traffic on port 25 (SMTP, for sending email) on employee computers. If all company email is sent out via a separate mail server, then any email servers on employee machines are likely sending spam. While this certainly helps to fight spam, blanket port-level filtering can stymie legitimate uses as well.

Application relays can be used to selectively filter only the traffic deemed harmful. For instance, spam filters can examine email messages to block spam, while firewalls can be configured to remove executables from incoming email attachments. To do this, all traffic passes through the relay so that its content can be inspected before being passed onwards. One challenge with application filters is that what is included in the application layer may change quickly, as the pace of innovation at the application layer moves much faster than at lower layers.

Firewalls impose a particular view of computer networks that is not always realistic. They typically view outside networks as dangerous and internal networks as safe. Consequently, they block incoming traffic but do nothing about network traffic "behind" the firewall. Attackers can get around this by infecting machines through services allowed to pass through the firewall (such as web traffic), and then spreading infections from inside the network.

Viewed in isolation, each of these ex post countermeasures is imperfect for countering security threats. In principle, they should not be viewed as a substitute for secure system design. In practice, nevertheless, firms often sell a combination of countermeasures as a solution. Security engineers refer to the practice as *defense in depth*: adopting multiple countermeasures simultaneously. Borrowed from military strategy, the idea behind defense in depth is to use multiple strategies for stopping attacks in the hope that one approach will succeed. While clearly a second-best strategy, this hodgepodge of countermeasures is the primary defense offered in the marketplace. We investigate the reasons for arriving at such an outcome in later chapters.

5.2 Ex ante countermeasures

The second group of countermeasures is more proactive, in that the defenses are applied prior to an attack taking place. Some countermeasures do not require making changes to the underlying infrastructure. These include compliance mechanisms, where audits check to ensure that a firm's operational security is consistent with best practices. Compliance regimes check that ex post mechanisms are properly configured in the event of attack. One limitation of compliance regimes is that they often only check for what can be easily verified, such as increasing encryption key length or setting password policies that require frequent changes and adhere to minimum length. Unfortunately, mechanisms that can be easily verified may not actually increase operational security.

Penetration testing can be thought of as auditing by simulated attack. Here, rather than wait for attacks to be realized, penetration testers attempt to find vulnerabilities in deployed systems and plug the holes before an attacker can find them. As with compliance regimes, accurate threat modeling is essential: the simulated penetrations should correspond closely to real incidents, otherwise the exercise will not prevent realistic attacks.

Some ex ante countermeasures make fundamental changes to the security of the infrastructure, but these can only be used as part of a longer-term strategy. For example, Microsoft made a concerted effort to improve the security of its operating system code when developing Windows Vista. They adopted secure software engineering techniques to reduce the number of bugs appearing in code. This includes requiring software developers to explicitly consider a threat model when writing code. Once the code is written, it goes through several phases of testing, which looks for common vulnerabilities such as memory overwriting or code injection. Some of these checks have been automated, while others are performed manually. The upshot for Microsoft has been a substantial reduction in the number of vulnerabilities found in its software: Windows Vista experienced 45% fewer vulnerabilities in its first year of release than Windows XP did [5].

Other more fundamental changes include changing the architecture of computer systems and networks in order to improve security. The challenge here is that security goals often directly conflict with traditional benefits of information technology. For example, the principle of abstraction makes it possible to cheaply reuse software in different settings. In fact, the layered architecture described in Section 2.1.2 is explicitly designed to make reuse beneficial. The scaling benefits of abstraction, reuse and the resulting economies of scale are the winning principle that boosted the software industry and spurred productivity growth. They also help explain explain why dominant markets in software frequently emerge. However, easy scaling has negative security implications, since vulnerabilities also propagate and can trigger widespread failures. To counter this threat, some advocate increasing the diversity present in computer systems and networks, from the operating systems software used to the routers that are deployed. Yet increasing diversity necessarily leads to a corresponding reduction in efficiency. It is not clear what the right balance is to strike between diversity and efficiency.

Design complexity also naturally emerges in computer systems. Why? One reason is that code reuse increases dependencies and design complexity. Another is the pressure to design systems that maintain "legacy" compatibility (i.e., compatibility with old versions). This is attractive in order to keep existing customers happy, but it also means that systems tend only to grow and never shrink in size. Perhaps the strongest driver of complexity is the temptation to always add new features, particularly if the new feature could attract additional customers.

Yet it is recognized that software complexity greatly increases the likelihood that there will be bugs and vulnerabilities in the underlying code. As security guru Bruce Schneier puts it: "complexity is the enemy of security". But why is complexity a security problem? Well, more features requires more code which tends to lead to more bugs. Yet increasing complexity can sometimes lead to more than just an additive risk. Often the interfaces between different system components are the critical links along which attacks propagate, and failure at a single point in the chain could disrupt the entire system. Another challenge of complex systems is that they tend to have more and richer interfaces, making it hard to do input validation properly. Finally, vulnerabilities can propagate from legacy systems, particularly design flaws. Again, we point to an insightful remark from David Wheeler: "Compatibility means deliberately repeating other people's mistakes."

Given complexity's downsides, some organizations are trying to reduce the design complexity of their systems and networks. Firms can do this by explicitly striving for more simplified systems, such as by making clean breaks with former versions of software. For instance, Microsoft decided to re-write much of its code base when developing Windows Vista in order to increase the security compared to its previous version, XP. Nonetheless, there are significant reasons why fighting complexity is the exception rather than the norm. Again, Microsoft's development of Vista is telling: it was delivered behind schedule and over budget. This was a luxury that Microsoft could perhaps afford, given its dominant position in operating systems at the time. Other firms are often not so fortunate.

6 Summary

Building secure systems is difficult. In principle, we know how to do it. In practice, few of the security mechanisms we have just presented are adopted. Why not? The behavior of stakeholders is different from that anticipated by system designers. And how might we explain the behavior of human beings and organizations facing constraints and competing interests? With economics, of course.

7 Further Reading

For an introduction to security engineering, we refer interested readers to Anderson's tome [1]. For a comprehensive book on cryptography, see Menezes, van Oorschot and Vanstone [11].

References

[1] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, second edition, 2008.

[2] Elaine Barker and Allen Roginsky. Nist special publication 800-131a, transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths, January 2011. `csrc.nist.gov/publications/nistpubs/800-131A/sp800-131A.pdf`.

[3] Joseph Bonneau, Sören Preibusch, and Ross Anderson. A birthday present every eleven wallets? The security of customer-chosen banking PINs. In *FC '12: Proceedings of the the 16th International Conference on Financial Cryptography*, March 2012.

[4] Fred Cohen. Computer viruses, theory and experiments. In *Proceedings of the 7th National Computer Security Conference*, pages 240–263, Gaitherburg, MD, 1984. National Bureau of Standards.

[5] Microsoft Corporation. Sdl helps build more secure software. `http://www.microsoft.com/security/sdl/learn/measurable.aspx` (last accessed May 24, 2012).

[6] Ian Grigg and Peter Gutmann. The curse of cryptographic numerology. *IEEE Security and Privacy*, 9:70–72, 2011.

[7] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In Paul C. van Oorschot, editor, *USENIX Security Symposium*, pages 45–60. USENIX Association, 2008.

[8] Auguste Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, pages 5–38, January 1883.

[9] Gerhard Koning Gans, Jaap-Henk Hoepman, and Flavio D. Garcia. A practical attack on the mifare classic. In *Proceedings of the 8th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications*, CARDIS '08, pages 267–282, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] Brian Krebs. Coordinated atm heist nets thieves $13m. *Krebs on Security*, April 2011. http://krebsonsecurity.com/2011/08/coordinated-atm-heist-nets-thieves-13m/.

[11] Alfred Menezes, Paul van Oorschot, and Scott Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[12] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, Apr 19 1965.

[13] Federal Bureau of Investigation. Taking a trip to the atm? beware of 'skimmers', July 2011. http://www.fbi.gov/news/stories/2011/july/atm_071411.

[14] Adi Shamir. Cryptography: State of the science, 2002. http://amturing.acm.org/vp/shamir_0028491.cfm.

[15] Sergei P. Skorobogatov. Data remanence in flash memory devices. In Josyula R. Rao and Berk Sunar, editors, *CHES*, volume 3659 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2005.

[16] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[17] Alma Whitten and J. D. Tygar. Why johnny can't encrypt: a usability evaluation of pgp 5.0. In *Proceedings of the 8th conference on USENIX Security Symposium - Volume 8*, SSYM'99, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.

[18] Kim Zetter. Malicious atm catches hackers. *Wired*, August 2009. http://www.wired.com/threatlevel/2009/08/malicious-atm-catches-hackers/.