52.1 Introduction

This chapter provides an overview on information systems security. It starts by discussing the key concepts and underlying principles of computer security, common types of threats and attacks, and the origins of software vulnerabilities.

The chapter then reviews UNIX-like security abstractions: the hierarchy of processes in a system, hardware support for separating processes, the UNIX file system hierarchy, and authentication. Then it is shown how these protections extend to the network, how a typical TCP/IP connection is protected on the Internet, and how web servers and browsers use such protections. Security flaws are also covered with a detailed discussion of the most dangerous software vulnerabilities: buffer overflows and memory corruption, SQL injection, cross-site scripting (XSS), and time-to-check-time-to-use race conditions. Following this, the chapter argues that all the vulnerabilities cross multiple layers of abstraction (application, compiler, operating systems (OSes), and architecture) and discusses research directions to mitigate this challenge and better protect information systems.

52.2 What Is Computer Security?

Computer security addresses policies and mechanisms to protect automated computer systems so that the confidentiality, integrity, and availability of their resources are protected. These resources can be software (the OS and application programs), firmware, hardware, information and data (e.g., a file
Managing and Securing the IT Infrastructure and Systems

system, and all networked communication between two end systems [34]. The three pillars upon which computer security is based (confidentiality, integrity, and availability) are commonly referred in the literature by the CIA acronym [10,51].

Confidentiality requires that private, confidential, or sensitive information and system resources are not disclosed to unauthorized parties. This can be achieved through access control or encryption. Access control policies manage the access to the system and its resources and have two main functions: authentication and authorization. Authentication is the process that determines the identity of someone attempting to access a system or a set of data [27]. This assessment of the individual or any other entity (e.g., another system) is usually done through credentials, such as a password, fingerprint, or a card. Authorization is the process that determines the set of resources or data a person or a third-party entity (once authenticated) are allowed to access. For example, once authenticated in a certain Linux machine, user bob has access to all files located in his home directory, but he may not have access to the files located in user alice’s home directory.

Integrity requires that data or a system itself is altered only in an authorized manner. It is usually described as a two-concept definition: data integrity and system integrity [34]. Data integrity requires that information and programs are changed only in a specified and authorized manner. System integrity requires that a system functions free from any unauthorized change or manipulation.

Availability requires that a system works in a timely manner and its resources and data are promptly available to authorized users.

52.2.1 Vulnerabilities, Threats, Attacks, and Countermeasures

A vulnerability is an error or a weakness in the design or implementation of a system that could be exploited to violate its security policy. Software systems cannot be guaranteed to be free from vulnerabilities because designers and programmers make mistakes, and current verification and testing techniques cannot assure that a significantly complex piece of software meets its specification in the presence of errors or bad inputs. The likelihood of a successful exploitation of a vulnerability depends on its degree and the intelligence of the exploitation. A threat is a potential for violation of security caused by the existence of vulnerabilities. A threat might never be realized, but the fact that it exists requires that measures should be taken to protect the system against this potential danger. An attack is a threat that was realized or a concrete exploitation of a system vulnerability. We call the agent that performed such malicious actions the attacker. An attack is active when it attempts to alter system resources and affect its operations, and it is passive when it attempts to learn or leverage system information without affecting its operation or assets [49].

A countermeasure is any procedure taken to reduce a threat, a vulnerability, or an attack by eliminating or preventing it, or minimizing the harm it can cause by discovering, stopping, or reporting it so that some recovery action is taken [49]. Countermeasures can be classified into three types: preventive, detective, and corrective. A preventive countermeasure prevents an attack from happening in the first place. For example, the use of a password prevents non-authorized users to access a system, and network filters prevent known malicious packets from being processed by the network stack at the OS. A detective countermeasure accepts that an attack may occur, but is able to determine when a system is under compromise and possibly stop the attack or at least warn a system administrator. This usually involves the monitoring of several system attributes. After an attack occurs, a corrective countermeasure (usually the hardest to implement) tries to bring the system back to a stable error-free state, for example, by restoring a file system with backup data.

52.2.2 Origins of Current Vulnerabilities

Even though information systems will never be immune to errors, it is important to understand how the interplay of different layers of abstraction exacerbates the problem. Consider the simplest security model: a gatekeeper. An analogy to the gatekeeper model is the bank teller. Suppose a bank customer
needs to use a bank service or access their account. They will walk up to the bank teller’s window and give them instructions about what they would like to be done (e.g., withdraw money, deposit, transfer, or obtain a cashier’s check). The bank teller is a trusted entity that actually carries out these actions on behalf of the customer, while at the same time, ensuring that the bank’s policies are being followed. For example, in the case of money withdrawal, the bank policy is to never give out money to a customer before subtracting the same amount from their account.

This abstraction represents the gatekeeper concept from an early OS called MULTICS [17,21]. In MULTICS, processes were separated into several rings, and the lower the ring number some code is executing in, the greater its privileges are. The system must be aware of ring crossing at the architecture layer. When code in ring $i$ attempts to transfer control to code in ring $j$, for the code in ring $j$ to do something on its behalf, a fault occurs and control is given to the OS. The gatekeeper is the software abstraction that handles this fault. Its name originates from the view of crossing rings as crossing walls separating them. Ideally the gatekeeper should be as simple as possible. Simplicity allows for ease of inspection, testing, and verification. The design is streamlined and the likelihood of an error is reduced. On the other hand, if the gatekeeper is complex, involving a set of abstractions where many high-level, more complex abstractions are built on top of simpler, lower-level ones in multiple tiers, the possibility of introduction of an avenue for deceiving the gatekeeper is greatly increased.

Most modern information systems are based on what can be called UNIX-like systems (including Windows-based systems). These systems combine memory separation of different processes with a hierarchical file system and process hierarchy to achieve security. These systems are designed with complex and cross-layered security abstractions. For example, for security reasons, a process can access memory only inside its address space. However, the implementation of the abstraction of functions uses the process address space (the stack region) at user level. The stack region should be under strict OS control, but processes have access to their address space. Further, sensitive low-level control information (a function return pointer) belonging to the architecture level (value of the program counter register) can be accessed by the OS and processes because it is stored on the stack.

Furthermore, the application of certain security principles can add to the complexity of the gatekeeper and even conflict with one another. In a classic and much-referenced paper from 1975, Saltzer and Schroeder [43] described several design principles, provided by experience, for the design of information systems and security mechanisms.

One of these principles is fail-safe defaults. It states that the design of a system or a security mechanism should be very conservative regarding access to objects and functionalities. The default situation should lack of access, and the security mechanism should identify which subjects should be allowed access and under which circumstances. The intuition behind this principle is that an error in the security mechanism design will fail by restricting access, and this situation would be easily identified by those subjects (a user, program, or another system) who should be granted access to the system. The alternative design, whose default situation is complete access with access control decisions based on exclusion of rights, will fail by allowing access to a subject that should not receive it. Examples of the application of this principle would be to enforce that only well-typed programs are loaded into memory as processes, or to only allow access to memory locations until they are explicitly the right type that the access is meant for. These examples conflict with the principle of economy of mechanism that states that the design of a secure system or security mechanism should be as simple and as small as possible. UNIX-like systems give each process its own address space and several ways to communicate with other processes. Each process is responsible for the accesses it makes to its own memory, which errs more on the side of the second principle.

52.2.3 Why Is Securing a System Hard?

There are many reasons that make computer security a challenge. First, as we have mentioned before, systems will never be vulnerability-free because they are becoming more and more complex and
diverse and are devised by humans, who are likely to introduce flaws into the system. Another challenge is that the problem of checking if a piece of code contains malicious logic is undecidable [10]. Malware detectors usually target a particular type of attack and may present false positives and false negatives. Also computer security is an arms race between attackers and security researchers and engineers. Unfortunately, the attackers have the edge because their goal is much simpler than those of a security researcher: an attacker needs to find a suitable vulnerability and exploit it; a security engineer must address all types of vulnerabilities a system may have and patch them while avoiding the introduction of new vulnerabilities in the process. Further, we have been witnessing an increase in the complexity of malware and attackers. The current generation of attackers is extremely creative, financially and politically motivated, and structured much like any well-operated criminal organization [1]. Traditional security models and solutions have difficulty keeping up with these attackers’ level of innovation and ingenuity. Finally, many systems are not (or were not) designed with security as a requirement. In many cases, security is an afterthought and sometimes is perceived as an inconvenience or a burden, for example, decreasing performance, increasing costs, and making systems harder to operate.

52.3 UNIX-Like Security Abstractions

In this section, we will build a UNIX-like system from the ground up conceptually, to see what the different protection mechanisms are and how they interact with each other through many layers of abstraction.

The protection mechanisms that form the foundation of a UNIX-like system can be divided into two types: reference-checking and reference-omitting. Reference-checking mechanisms use an access control list to check all accesses to an object to see if the subject trying to access the object has permission to perform that kind of access, where the access could be an operation such as read, write, or execute. Reference-checking mechanisms associate with each object a list of the accesses that various subjects can perform on that object. Reference-omitting mechanisms only give a reference that can be used to make an access to subjects that have permission to make that access, so that every access need not be checked. In other words, every subject is associated with a list of objects that it can access. Note that the references given to a subject to refer to a subject must be protected themselves for reference-omitting mechanisms, however. Reference-omitting mechanisms are often referred to as capabilities and reference-checking as access control lists, but in this section, we will avoid these names to avoid confusion and historical connotations.

An example of a reference-omitting mechanism is a page table. Page table mappings have bits to allow write access, for example, but there is no notion of different processes in these access controls. The mechanism that stops one process from accessing the physical memory of another process is that it has no virtual-to-physical mapping in its page table for the physical page frames of the other process. So it is not a matter of an access control check based on distinguishing one process from another, but rather the process that is not supposed to access the memory has no possible way to calculate or forge a virtual address that refers to the physical memory of the other process. The reference is omitted, rather than checked.

52.3.1 Hardware Support

First, what do we mean by a “UNIX-like” system? We mean a system that combines memory separation of different processes with a hierarchical file system and process hierarchy to enable security. Although we will not discuss Microsoft Windows in this section, under this definition, Windows is a UNIX-like system. Windows has a rich set of security features [42], but the foundations of security in Windows are still based on the abstractions of processes and files that UNIX pioneered. UNIX owes
many of its security ideas to MULTICS [44,17]. In MULTICS, processes were separated by segmenta-
tions and organized into 64 rings, with inner-more rings having more privileges. A process could
only request for higher-privileged processes to do tasks on their behalf by going through a gatekeeper
that initiated code defined at a higher privilege, which had the effect of restricting the interaction
between the two processes based on this code that resided in an inner-more ring. The code itself
would reason about what it should or should not do on behalf of the lower-privileged process with
respect to security. UNIX reduced this idea down to two rings, with the kernel in the inner ring and
user space processes in the outer ring, and with the processes themselves being kept separate by a
different mechanism.

To support the abstractions of UNIX-like security mechanisms, all that is required of the hardware is
a mechanism to separate these two rings and then a different mechanism to separate the processes. For
modern UNIX-like systems, a supervisor bit that distinguishes kernel mode from user mode is enough
for a ring abstraction where there are only two rings. Most modern CPUs that have support for OSes
have a bit that indicates when the CPU is in supervisor mode vs. user mode. The exact mechanism var-
ies, but the effect is the same. In the MIPS instruction set architecture, for example, this bit is implied
by the virtual address. The x86 architecture provides four rings, but typical UNIX-like system use only
two of them (the inner-most and the outer-most) [13].

Not every system is based on UNIX-like abstractions, but UNIX-like abstractions have proven
to be very portable. Eros [48] is an OS that is radically different from UNIX in terms of security
abstractions and uses all four rings of the x86. Eros is considered a capability-based system. Levy [29]
describes historical systems that were based on the notion of capabilities and required much more
elaborate hardware for OS security support than is required by UNIX. In fact, for many secure system
designs that are not UNIX-like, the hardware and the OS are codesigned for each other, which can
limit portability and requires careful thought about the security abstractions whenever two different
systems interact.

UNIX-like security mechanisms also need some way for processes to be kept separate. Specifically,
one process should not be able to read from or write to other process’s memory without such permission
being explicitly granted. Originally, UNIX used the PDP-7’s segmentation mechanism to separate the
memory address spaces of processes. Segmentation divides physical memory into contiguous segments
that can be referenced with a base and offset. Because processes could not change their base and the
bounds of their offsets are checked, processes cannot access the other segments in memory. Because
of external fragmentation, however, paging is now much preferred to segmentation as a way to sepa-
rate processes’ address spaces in physical memory. As discussed earlier, paging is a reference-omitting
mechanism, in which processes are given only virtual address mappings in their page tables for those
physical memory page frames that they are allowed to access. In this way, there is no way for the process
to calculate a virtual address pointer that references the memory of another process unless such a refer-
ence is placed in the page table of that process by the OS kernel.

52.3.2 Security Abstractions Implemented in Software

Once the hardware has provided the OS with the two important protection mechanisms that we have
now discussed (one to separate the kernel from user space and another to separate processes within user
space), kernel data structures can be built to support security at higher levels of abstraction. Individual
processes can interact with other processes or with the kernel only through explicit requests to the ker-
nel called system calls (similar to MULTICS’s gatekeeper mechanism). Only the kernel has the privileges
necessary to directly access system resources such as the network, hard drive, keyboard, and monitor. In
this way, the kernel can separate these resources into different files for different users, different network
connections, etc., and enforce access controls.

A UNIX-like system uses two key data structures as the basis for security: file system access controls
and the process hierarchy. A typical UNIX-like system has different users that must authenticate to the
system and then a special user called root that can override most access controls. Users own processes, and they also can own file system objects. File system objects can be actual files, network sockets, other interprocess communications mechanisms, special character devices, or anything that processes can interact with through a file abstraction.

The most basic access controls are those on actual files on the file system, for example,

```
-rwxr-xr-x 1 johndoe johndoe 253 2011-11-21 10:37 myprogram.pl
```

In this example, the user johndoe can read from or write to myfile.txt, and any user in the faculty group can read from this same file, but other users cannot. Anyone is allowed to enter the Mydirectory directory and list the files there because of the global read and execute permissions (assuming they are already able to reach this point in the directory structure), and anybody can read and execute the executable program myprogram.pl. UNIX-style file permissions are a reference-checking mechanism; consequently, all attempts to add a file to a process's file descriptor table must be checked.

It is important to note that these access controls are checked for processes owned by users, and they are typically checked only at the time the file is opened. Processes can have any number of files opened for reading or writing at one time, and these are placed in a file descriptor table so that the process can refer to files in system calls to the kernel by the number of the file descriptor entry. The file descriptor table is a reference-omitting mechanism; once a file is placed in the file descriptor table for a certain type of access, the process can perform that access on the file without any additional checks.

In UNIX-like systems, there is a special process called a shell that prompts a user for commands and allows the user to cause the system to do a wide range of actions. A typical user shell process in a UNIX-like system has access to tools such as compilers that allow it to basically do arbitrary computations and any sequence of system calls desired. Thus, the security of a UNIX-like system is based not on what a user process cannot do within its own address space, but on restricting how it interacts with other processes.

This leads to the hierarchy of processes. This hierarchy is not necessarily a privilege hierarchy where parents always have more privileges than their children. This is a very important point, because prevention of privilege escalation and interacting with remote users from around the world over the network in a secure way are both based on limiting the influence that one process can have over another.

UNIX processes can use the `fork()` system call to create new children below them in the hierarchy. These children typically inherit all of the parent’s open file descriptors as well as the parent’s real user ID and effective user ID. The parent can also pass arguments to the child. The real user ID is what the ID of the actual user who initiated the process is, while the effective user ID is the one that actually is checked for controlling access to files and other system resources. Typically, these IDs are the same, but they can be different when the child uses the `execve()` system call to mount a binary with the setuid bit set, for example,

```
-rwxr-xr-x 1 root root 36864 2011-02-14 15:11/bin/su
```

The “s” in the permissions indicates that the setuid bit is set for this binary program file. This bit means that any process that mounts this binary (mounting means that the old executable code along with the rest of the address space of the process is discarded in memory and the process is now running the binary code in /bin/su) has their effective user ID set to the owner of the file, in this case root, rather than the process that executed it. For example, `passwd` is a UNIX utility for changing passwords that has the setuid bit set, for example,

```

The `execve()` system call is used for mounting new binaries. Most binaries do not have the setuid bit set, so that the child still has the same effective user ID of the parent. Binaries such as /bin/su, which is used to change to another user and possibly the root user by entering that user’s password,
need the effective permissions of root to read the password file and authenticate a new user and then create a shell process for the new user.

Now we have built a conceptual picture that will help to understand the exploits in Section 52.4. The hierarchy of processes comes from parents who fork() children and then those children sometimes execve() to run other binaries. The parents can pass arguments and open file descriptors to the children. The children may sometimes have higher privileges than the parent by mounting setuid binaries. The root of the tree is typically the init process, which forks login children that run as root but can read users’ usernames and passwords and fork children that become shell processes after dropping their permissions to the appropriate level for that user. So we have a hierarchy of processes, where lower privileged processes can have ancestors, siblings, and descendants with a higher privilege level than themselves and where there are many ways for processes to communicate through files, sockets, pipes, and other mechanisms.

The major thing missing from this conceptual picture is the network. It is possible for a process on one system to open a network socket to a remote system that is treated by both processes the same as any other entry in their file descriptor table. For example, a web browser on one system can request a remote socket with a web server on another system, and if the kernels of these two systems allow this access and successfully do a three-way TCP handshake, the processes can now communicate over the network.

What we will see in Section 52.4 is that when processes can communicate with one another one process can influence the actions of the other through software vulnerabilities and then subvert security mechanisms.

52.4 Vulnerabilities and Countermeasures

In this section, we discuss several common and dangerous vulnerabilities or security flaws that allow attackers to target user-level programs and OSes. The context here is that each of these vulnerabilities allows attackers to violate the assumptions upon which the security abstractions of a UNIX-like system are built.

52.4.1 Buffer Overflows

This vulnerability is one of the most common and dangerous security flaws and was first widely publicized by the Morris Internet Worm in 1988 [2], which compromised thousands of machines and caused millions of dollars in losses.

A buffer is a memory area in which contiguous data (an array or a string) can be stored. A buffer overflow is the condition at an interface under which more data are placed into a particular finite size buffer than the capacity allocated [35]. This location could hold other program or system variables and data structures, function pointers, or important program control information such as a function return address.

To understand buffer overflows, it is important to review how a program is laid out in memory by the OS for execution. The process address space or the memory region that can be accessed by the process is divided into four main regions. The code region is read-only, fixed-sized, and contains the program machine code. The data area is also fixed-sized and contains global and static variables and data structures used by the program. The heap is used for dynamic allocation of program variables and data structures. The stack is used to implement the abstraction of function calls. It grows toward the lower memory addresses of a process and stores function’s local variables, parameters, and also control information such as a function return address.

Consider the code snippet illustrated in Figure 52.1. Upon calling b, a pushes the parameters for b on the stack in reverse order of declaration. The return address of a (the next instruction that should continue to be executed when b returns) is also saved on the stack. Then the address of a’s stack frame pointer is saved on the stack. This is necessary to recover it after function b returns. Notice that b’s local
variables (including the vulnerable buffer) are stored below the saved value of the frame pointer and the return address. Also when an input is stored in a buffer, the bytes are placed toward higher addresses in the process address space, i.e., toward the saved frame pointer and return address.

A stack-based buffer overflow can cause several types of harm in a system or program. An unintentional buffer overflow may cause a program or an OS to terminate or crash, due to the corruption of data structures, stack frame pointer, or return address. For example, if the return address is overwritten with a random value, this value will be loaded into the PC register (called the program counter, which contains the address of the next instruction to be executed by the CPU), and the CPU might identify this value as an illegal memory access (possibly outside the process address space). A malicious buffer overflow will occur when the attacker carefully crafts an input where the return address is overwritten with an address chosen by the attacker. This address could be within the malicious input string and points to malicious instructions that open a shell (a command-line interface to the OS), giving the attacker administrator privileges in the system if the compromised program is privileged. The malicious address could also correspond to a library function (e.g., libc) allowing the attacker to perform malicious actions by combining existing functions (also called return-to-libc attack [31]).

52.4.1.1 Defenses

There are several different proposed solutions currently in use against buffer overflow attacks, and they can be broadly classified into two types: compile-time and run-time [30].

A compile-time solution involves adding code to programs with the goal of preventing and detecting buffer overflows. Type-safe languages such as Java and Python have the compiler generate code to check array bounds, thus preventing such attacks. The StackGuard gcc extension [15] protects the stack by inserting additional instructions to a function entry and exit code. The goal is to protect the stack by checking if it is corrupted. A function entry code writes a canary value (named after the miner's canary used to detect poisonous air in a mine and warn the miners) below the saved frame pointer on the stack. Upon exiting, the function exit code checks that the canary was not corrupted, as the attacker will need to corrupt the canary to tamper with the return address. This canary value must be randomly generated to avoid the possibility of an attacker guessing the value and overwriting it with the proper value during the attack, thus evading detection.

Run-time solutions protect vulnerable programs by detecting and stopping buffer overflow attacks during execution. One common approach is to protect the address space of a process against improper execution. A buffer overflow attack usually causes a control flow change that leads to the execution of malicious instructions that are part of the malicious input string in the stack. If the stack is marked
as nonexecutable, many instances of buffer overflow attacks can be stopped. The assumption is that executable code is supposed to be located in the code portion of a process address space and not on the stack, heap, or data areas. The Solar Designer’s StackPatch [18] modifies the address space of a process to make the stack non-executable. The $W \otimes X$ defense [55] ensures that no memory location in a process address space can be marked both as writable ($W$) and executable ($X$) at the same time. Most OSes now offer such protection. While this type of defense is considered an effective protection mechanism that is relatively simple to implement, there are situations where a program must place executable code in the stack [51]: just in time compilers (Java), implementation of nested functions in C, and signal handlers.

Another approach is address space randomization. In order to exploit a buffer overflow, an attacker needs to predict the location of a return address or know the exact location of a library function. Randomly arranging the memory positions of key areas such as the heap, stack, and libraries makes this prediction harder for the attacker. This idea is implemented in PAX ASLR [39] and is supported by many OSes.

52.4.2 Vulnerabilities and UNIX-Like Abstractions

Buffer overflows demonstrate an important part of UNIX-like security abstractions, which is that the code that a process runs and the semantics of the program are a key part of the security of a system. In UNIX-like systems, a setuid root binary such as `su` must carefully process all of its inputs to make sure that the terminal it inherited from its parent must provide a valid username and password before a shell for entering commands as a new user is created as its child. Similarly, a web server must check the inputs it receives on a network socket from a remote client to make sure it is a valid request within the HyperText Transfer Protocol (HTTP) protocol and then must only carry out the actions necessary to provide the HTTP service as per that protocol. Since “change the root password on the system” is not part of the HTTP protocol, the remote client cannot directly subvert security within the confines of the protocol. Rather, within the confines of UNIX-like security abstractions, attackers exploit vulnerabilities to have more influence on other processes than the security model for the system allows.

This notion of processes acting on other processes’ behalf (perhaps remotely) leads to three kinds of exploits for vulnerabilities: those that directly corrupt a process and take it over through communication, those that confuse other processes at higher levels of abstraction, and those that attack the communication channel between two processes to exploit a trust relationship between them.

52.4.2.1 Directly Corrupting a Process and Taking It Over

Control flow hijacking attacks communicate with other processes in a malicious way that causes that process to effectively be taken over so that its computations and system calls are now controlled by the attacker.

The most famous example of this kind of attack is the buffer overflow, which was explained earlier in this section. These are a special case of a broader problem known as memory corruption attacks and that includes not only buffer overflows [6], but also format string attacks [46] and double free()'s [7]. By corrupting the memory of another process through malicious inputs that it cannot handle properly, it is possible to inject the attacker’s own instructions to be executed by that process. This can be used to elevate privileges on the current system by corrupting another process with higher privileges, which might be a child with the setuid bit set. This can also be used to take control of a remote machine by corrupting one of its processes. This can be achieved by, for example, connecting to a web server that has a memory corruption vulnerability in its implementation. It can also be achieved by, for example, sending a user a malicious file through e-mail such as a PDF that exploits a software bug in their PDF reader. Mitigations for buffer overflow attacks were discussed earlier in this section.

In these previous examples, a process owned by the attacker sent malicious inputs to another process, either on the same system or on a remote system over the network, where the other process has privileges that the attacker needs in order to subvert security. Another example of directly taking over
a process is SQL injection. A typical SQL injection attack involves at least three processes that communicate remotely over the network. The attacker's remote client process sends form inputs to a web server process, which embeds these inputs into an SQL query to interact with a database. The attacker uses SQL delimiters in their form inputs to change the boundaries between the SQL code and their inputs. In this way, they can control the SQL code that the SQL server executes.

For example, web server code might read \$name and \$password from a web client and then build an SQL query for a database server using the following substitution:

\[
\text{SELECT } * \text{ FROM users WHERE name} = \text{'}\text{$name$} \text{'} \text{ and password} = \text{'}\text{$password$} \text{'}
\]

The single quote (’) is a special delimiter in the SQL language. The attacker can leave the password blank and enter this as their username:

\[
\text{'}; \text{DROP TABLE users-- comment ...}
\]

This causes the SQL server to execute the following SQL code that it receives from the web server, with the effect of the entire table of users in the database being deleted:

\[
\text{SELECT}^* \text{ FROM users WHERE name} = \text{'}; \text{DROP TABLE users-- comment ...}
\]

Fundamentally, SQL injection vulnerabilities are a problem where the inputs from the attacker span across more than a leaf in the parse tree of the resulting query [52]. For example, the name “bob” would be a single leaf in a parse tree of the SQL language for the resulting query, but the name “’; DROP TABLE users -- comment ...” will be a tree itself with “’”, “DROP”, “TABLE”, “users”, and “--comment” all being separate leaves in the tree. Su and Wassermann [52] describe a dynamic approach to detect SQL injection exploits.

XSS is another type of vulnerability where information being passed between more than two processes leads to the ability for the attacker to control another process via malicious inputs. In the case of XSS, a typical XSS attack allows one web client to pass scripted code to another client through the server. For example, suppose a forum on a web server allows users to add HyperText Markup Language (HTML) to their forum comments so that other users can see formatted text with tags, such as bold text, for example, \text{This is a bold forum comment}. Suppose also that the forum server does a poor job of filtering out HTML code that could harm other clients, such as Javascript:

\[
\text{body onload} = \text{‘alert(‘xss’)}
\]

This is a real example from a college campus course content management system [26] that allows students and faculty to post HTML-formatted messages in forums or as e-mail. An HTML blacklist is applied to not allow HTML tags that lead to the execution of Javascript, but the blacklist only filters out well-formed HTML (in this case well-formed HTML would have a closing angle bracket, “>”) while most browsers will execute the Javascript code \text{alert(‘xss’)} even without well-formed HTML. In this example, if a student posts this string or sends it as an e-mail to the instructor, the message recipient will receive a pop-up window that simply says, “xss”. The ability to execute arbitrary Javascript could, however, give students the ability to cause the instructor’s web browser to update the gradebook or do any action that the instructor’s browser session has the privileges to perform. For a good explanation of XSS vulnerabilities and why blacklist approaches fail, as well as a dynamic approach that is sound and precise so that XSS vulnerabilities can be removed from source code at development time, see Wassermann and Su [59].

52.4.2.2 Confusing Other Processes at a Higher Level of Abstraction

It is also possible to confuse other processes at a higher level of abstraction, so that the process violates security on the attacker’s process behalf of even if the vulnerable process is not corrupted to the point where the attacker has complete control.
An example of this would be a Time of Check to Time of Use (TOCTTOU) race condition vulnerability. This kind of vulnerability occurs when privileged processes are provided with some mechanism to check whether a lower-privileged process should be allowed to access an object before the privileged process does so on the lower-privileged process’s behalf. If the object or its attribute can change between this check and the actual access that the privileged process makes, attackers can exploit this fact to cause privileged processes to make accesses on their behalf that subvert security.

The classic example of TOCTTOU is the sequence of system calls of `access()` followed by `open()`. The `access()` system call was introduced to UNIX systems as a way for privileged processes (particularly those with an effective user ID of root) to check if the user who owned the process that invoked them (the real user ID) has permissions on a file before the privileged process accesses the file on the real user ID's behalf. For example,

```c
if (access("/home/bob/symlink", R_OK | W_OK) != -1)
{
    // Symbolic link can change here
    f = fopen("/home/bob/symlink", "rw");
    ...
}
```

What makes this is a vulnerability is the fact that the invoker of the privileged process can cause a race condition where something about the file system changes in between the call to `access()` and the call to `open()`. For example, the file `/home/bob/symlink` can be a symbolic link that points to a file the attacker is allowed to access during the `access()` check but is changed to point to a different file that needs elevated privileges for access. The attacker can usually win this race by causing the `open()` system call to block for a hard drive access to read one of the directory entries (for more details see Borisov et al. [12]). The only effective way to mitigate TOCTTOU vulnerabilities is to identify sequences of system calls that are unsafe with respect to concurrency and not use them in privileged code. For example, secure code should never use `access()` followed by `open()`, which means that the `access()` system call has fallen into disuse since predicating file opens was its purpose.

### 52.4.2.3 Attacking the Communication Channel between Two Processes

Directly corrupting another process or confusing it at a higher level of abstraction typically entails one process sending malicious inputs to another that the receiving process cannot handle correctly with respect to security due to a bug. It is also possible for malicious agents on the network to corrupt the communication channel between two processes. When two processes communicate over the Internet, there is typically some trust relationship implied, such as that a web browser trusts that it is communicating with the web server a user intended to contact when it passes the user’s username and password along on the network. Network attacks seek to exploit this trust relationship.

Network attacks and the security mechanisms to thwart them are too numerous to discuss thoroughly here. For discussion of the most salient attacks on the most common protocols, see Tews et al. [56] regarding Address Resolution Protocol (ARP) injection and Wired Equivalent Privacy (WEP) attacks, Qian and Mao [41] regarding TCP hijacking, Savage et al. [45] regarding a large-scale attack on TCP congestion control, Karlin et al. [25] regarding attacks on Border Gateway Protocol (BGP) routing, and Wright [61] regarding an attack on Domain Name System (DNS).

We will use a typical HTTP connection with a typical network stack to illustrate the different possible points of attack. If a user wants to go to http://www.example.com, the first step is to find an IP address for this URL. The user’s system is configured to use a specific DNS server for this lookup, which is typically the DNS server for their organization or Internet Service Provider. But before the DNS server can
be contacted by IP address, the user’s system must find an ARP address for either the DNS server or a
gateway router that can reach the DNS server. ARP is a typical layer 2 protocol of the OSI model [40],
and layer 1 is the physical layer.

Above ARP is the IP layer, layer 3. This allows packets to be routed, for example, to the DNS server
and back. The user’s system makes a DNS request and the DNS server returns the IP address for http://
www.example.com. Now the user uses the TCP protocol (layer 4) on top of IP routing to connect to
the remote machine where the process for the web server that hosts http://www.example.com is run-
ing. The web browser and web server can now communicate through an interprocess communication
mechanism called a socket using a layer 7 (application layer) protocol that they have agreed on, in this
case HTTP.

Disclosure, deception, disruption, and usurpation [49] are possible at all of these layers. For ARP, any
machine on the local network can answer for any IP address, even the gateway. At the IP layer, any of
the routers that any packets for the connection go through have arbitrary control over that packet, and
attacks on the routing system such as BGP attacks [25] mean that attackers can ensure that certain con-
nections will be routed through their routers. Higher layers can also be attacked, for example, responses
to DNS requests can be forged. Because of all of these possible attacks, an important protocol for securing
interprocess communication over the Internet in an end-to-end fashion with public key cryptography is
Secure Sockets Layer (SSL) at layer 6 (the presentation layer). Using a public key infrastructure (PKI),
typically the Certificate Authority (CA) system for the Internet, it is possible for two remote processes to
set up an encrypted channel between them. Because SSL is a layer 6 protocol that is implemented in librar-
ies, applications can still communicate with a socket Inter-Process Communication (IPC) mechanism,
with the SSL library performing the encryption and decryption relatively transparently. The encryption is
end-to-end, meaning that it provides a level of privacy, integrity, and authentication that mitigates many
network-based attacks.

In summary, UNIX-like systems start from very simple hardware protections that prevent processes
from directly accessing blocks of data on the hard drive or gaining raw access to the network card to
interfere with others’ network connections. They then build many layers of abstraction on top of this.
Process hierarchies, file system hierarchies, network protocols such as ARP, TCP/IP, DNS, and BGP, and
many other abstractions build a bridge between the hardware’s simple notion of security and the security
properties that human users expect for higher-level concepts like URLs (e.g., http://www.example.com).
Software bugs called vulnerabilities make it possible for attackers to violate the assumptions that these
layers of abstraction are built upon.

52.5 Vulnerabilities and Layers of Abstraction

In this section, we discuss how security vulnerabilities in computer systems cross multiple layers of
abstraction (application, compiler, OS, and architecture). This cross-layer existence not only makes the
problem of classifying vulnerabilities hard but also hinders the development of strong and flexible solu-
tions to remove or mitigate them.

52.5.1 Challenge of Classifying Vulnerabilities

Bishop and Bailey [11] have analyzed many vulnerability taxonomies and showed that they are imper-
fect because, depending on the layer of abstraction a vulnerability is being considered, it can be classi-
fied into multiple ways.

A taxonomy is a classification system that allows a concept (e.g., a vulnerability) to be uniquely clas-
sified. The RISOS [3], the Protection Analysis [24], Landwehr et al. [28], and Aslam [8] taxonomies rep-
resented important studies to help systems designers and administrators to understand vulnerabilities
and better protect computer systems. The Protection Analysis study [24], for instance, sought to under-
stand OS security vulnerabilities and also identify automatable techniques for detecting them. Bishop and
Bailey [11] also summarized Peter Neumann's presentation of this study [32] with 10 classes of security flaws (which are further summarized here).

1. Improper protection domain initialization and enforcement: vulnerabilities related to the initialization of a system and its programs and the enforcement of the security requirements.
   a. Improper choice of initial protection domain: vulnerabilities related to an initial incorrect assignment of privileges.
   b. Improper isolation of implementation detail: vulnerabilities that allow users to bypass a layer of abstraction (e.g., the OS or the architecture) and write directly into protected data structures or memory areas, for instance, I/O memory and CPU registers.
   c. Improper change: vulnerabilities that allow an unprivileged subject to change the binding of a name or a pointer to a sensitive object so that it can bypass system permissions.
   d. Improper naming: vulnerabilities that allow two objects to have the same name, causing a user to possibly execute or access the wrong object. For example, if two hosts have the same IP address, messages destined to one host might arrive at another one.

2. Improper validation: vulnerabilities related to improper checking of operands or function parameters.

3. Improper synchronization: vulnerabilities arising when a process fails to coordinate concurrent activities that might access a shared resource.
   a. Improper indivisibility: interruption of a sequence of instructions that should execute atomically.
   b. Improper sequencing: failure to properly order concurrent read and write operations on a shared resource.

4. Improper choice of operand and operation: vulnerabilities caused by a wrong choice of the function needed to be called from a process. For example, a cryptographic key generator invoking a weak pseudorandom number generator.

Bishop and Bailey [11] showed how this study and others [3,8,24,28] fail to uniquely identify vulnerabilities using two very common and dangerous security flaws as examples: buffer overflows (described here as an example) and TOCTTOU.

52.5.1.1 Nonunique Classification of Buffer Overflows

At the application layer, this vulnerability can be classified as type 2 (improper validation) because the flaw was caused by the lack of array bounds checking by the programming language compiler and also by the program itself. From the perspective of the subject (in this case the attacker process) who used the vulnerable program, the flaw can be classified as type 4 (improper choice of operand) as the string passed as a parameter was too long (overflowed the buffer at the vulnerable process).

At the OS layer, the vulnerability can be classified as 1b (improper isolation of implementation detail). In this case, the vulnerable process was able to bypass OS controls and overwrite an area of the process address space corresponding to a function return address.

We can also argue that at the architecture layer, this vulnerability can be classified as 1b (improper isolation of implementation detail) as the compromised process was also able to directly write into the PC register. When a function returns, the return address (which was maliciously overwritten) is written into the PC register and the program control flow jumps to that location. It can also be classified as 1c (improper change) as the value of the PC register was maliciously changed.

52.6 Research Issues

Protecting an OS kernel against vulnerabilities is a difficult problem given the complexity and variety of its code. Many proposed solutions employ virtualization. In the traditional virtual machine (VM) usage model [14], it is assumed that the VM is trustworthy and the OS running on top of it can be easily
compromised by malware. This traditional usage model comes with a cost: the semantic gap problem. There is a significant difference between the abstractions observed by the guest OS (high-level semantic information) and by the VM (lower-level semantic information). These solutions use introspection to extract meaningful information from the system they monitor/protect [20]. With introspection, the physical memory of the current VM instance is inspected and high-level information is obtained by using detailed knowledge of the OS algorithms and data structures. Another line of research [36] employs active collaboration between a VM and a guest OS to bridge the semantic gap. In this approach, a guest OS running on top of a VM layer is aware of virtualization and exchanges information with the VM through a protected interface to achieve better security. A sizable amount of research is being pursued toward building secure web browsers [9,22,54,58]. We have been witnessing a shift to a web-based paradigm where a user accomplishes most of their computing needs using just a web browser [57]. Original browsers were designed to render static web pages, but as web applications and dynamic content became common, many attackers explored weaknesses in the design of these browsers (e.g., XSS and memory exploitation) to launch attacks. Researchers now view browsers as OSes and are investigating new models where browsers are designed by having their components isolated using the process abstraction, which can contain or prevent web-based vulnerabilities.

Return-oriented programming is a relatively new type of attack [47] inspired by the return-to-lib-c approach of exploiting the stack. The attacker does not inject foreign malicious code but instead combines a large number of short instruction sequences (called gadgets) that return and transfer control to another gadget. Several gadgets crafted from legitimate libraries, OS code, or program instructions can be used to perform malicious computations. This type of attack is dangerous because it does not involve foreign code and can evade important defense approaches such as $W \otimes X$. Current research focuses on randomizing or rewriting code so as to prevent an attacker from finding useful gadgets in them [23,38]. Randomization as a way to thwart attacks is a promising idea that dates back to 1997 [19], but there still is no widely accepted methodology for determining how much more difficult randomization makes it for the attacker to develop exploits for any given vulnerability.

The mitigations discussed for each vulnerability in Section 52.2.2 are specific to one type of vulnerability. Another approach to secure systems is to provide processes with security mechanisms that allow for finer-grained separation. These mechanisms are largely based on Saltzer and Schroeder’s Principle of Least Privilege, and the security comes from the fact that when processes are corrupted by an attacker, the attacker is more limited if those processes have very limited privileges. SELinux [4] and Linux capabilities [13] are examples that have already been fully integrated into a modern OS. Capsicum [60] is an example of an effort that is still in the research phase.

Many attempts have been made to secure commodity code without making major changes to it (such as separating it into different privilege domains). One notable effort is dynamic information flow tracking (DIFT) [16,33,53]. DIFT systems have proven effective at detecting existing attacks in full systems without compromising on compatibility, but cannot make any guarantee about attacks designed for DIFT systems. This is because tracking information flow dynamically is difficult to do precisely enough for programs that make heavy use of memory indirections such as pointers and control structures such as for loops [5,50].

52.7 Summary

This chapter discussed the foundations for security in information systems, which are based on the triad of confidentiality, integrity, and availability. We can find the origins of many software security flaws in the intertwine ment of different layers of abstraction (application, compiler, OS, and architecture), which hinders the development of simple but strong protection mechanisms for information systems and their resources. Nearly all information systems run on UNIX-like OSes (where Windows-based systems are also included). Knowledge about how such OSes are designed allows us to understand their different protection mechanisms, how the hardware facilitates their security, and also their most common types
of vulnerabilities and associated countermeasures. The challenge of countering and even classifying vulnerabilities can also be traced to the fact that they cross multiple layers of abstraction.

Given that security vulnerabilities involve more than one layer of abstraction, defending against them should involve some form of collaboration between layers of abstraction to bridge their semantic gap. Current security solutions operate at one particular level of abstraction. For example, they operate at the application level as a user-level process [37], at the compiler level so that safer binary code is generated [15], at the system level as an OS security extension [4], and also at the architecture level, usually involving a VM layer [20]. Multiple-tiered and collaborative security approaches seem a promising research direction to mitigate these vulnerabilities.

References