CONTENT

ACRONYMS

API	Application programming interface
CSRF	Cross-Site Request Forgery
CSS	Cascading Style Sheets
DDoS	Distributed denial of service
DOM	Document Object Model
DoS	Denial of service
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
JSON	JavaScript Object Notation
OOP	Object-oriented programming
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
SPA	Single-page application
SSDL/SDLC	Secure software development life cycle
SSL	Secure Sockets Layer
XML	Extensible Markup Language
XSS	Cross-Site Scripting
XXE	XML External Entity

PREFACE

In this day and age, when security breaches in software is costing companies colossal fines and regulatory burdens, developing operationally hacker-resilient software that is also reliable in its functionality and recoverable when expected business operations are disrupted, is a must have. The assurance of confidentiality, integrity and availability is becoming an integral part of software development.

The main content in this book is structured into three major parts, with each part containing many individual chapters covering a wide array of topics. Ideally, you will venture through this book in a linear fashion, from page one all the way to the final page. Reading this book in that order will provide the greatest learning possible. This book can also be used as either a hacking reference or a security engineering reference by focusing on the first or second half, respectively.

The first part of this book is “Fundamental Concepts”.

The second part of this book is “Offense.” Here the focus of the book moves from recon and data gathering to analyzing code and network requests. Then with this knowledge we will attempt to take advantage of insecurely written or improperly configured web applications.

The third and final part of this book, “Defense,” is about securing your own code against hackers. In Part III, we go back and look at every type of exploit we covered in Part II and attempt to consider them again with a completely opposite viewpoint. This time, we will not be concentrating on breaking into software systems, but instead attempting to prevent or mitigate the probability that a hacker could break into our systems. Two programming language will be used in this part are: Java and ASP.Net core.

CHAPTER 1. WEB APPLICATION FUNDAMENTAL

A web application (or web app) is application software that runs on a web server, unlike computer-based software programs that are stored locally on the Operating System (OS) of the device. Web applications are accessed by the user through a web browser with an active internet connection. These applications are programmed using a client–server modeled structure—the user ("client") is provided services through an off-site server that is hosted by a third-party. Examples of commonly-used web applications include: web-mail, online retail sales, online banking, and online auctions.

1.1. HTTP

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World-Wide Web global information initiative since 1990. The first version of HTTP, referred to as HTTP/0.9, was a simple protocol for raw data transfer across the Internet. HTTP/1.0, as defined by RFC 1945, improved the protocol by allowing messages to be in the format of MIME-like messages, containing metainformation about the data transferred and modifiers on the request/response semantics. However, HTTP/1.0 does not sufficiently take into consideration the effects of hierarchical proxies, caching, the need for persistent connections, or virtual hosts. In addition, the proliferation of incompletely-implemented applications calling themselves "HTTP/1.0" has necessitated a protocol version change in order for two communicating applications to determine each other's true capabilities.

This specification defines the protocol referred to as "HTTP/1.1". This protocol includes more stringent requirements than HTTP/1.0 in order to ensure reliable implementation of its features.

Practical information systems require more functionality than simple retrieval, including search, front-end update, and annotation. HTTP allows an open-ended set of methods and headers that indicate the purpose of a request. It builds on the discipline of reference provided by the Uniform Resource Identifier (URI), as a location (URL) or name (URN), for indicating the resource to which a

method is to be applied. Messages are passed in a format similar to that used by Internet mail as defined by the Multipurpose Internet Mail Extensions (MIME).

The HTTP protocol is a request/response protocol. A client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a connection with a server. The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity metainformation, and possible entity-body content.

1.1.1. HTTP Request

A request message from a client to a server includes, within the first line of that message, the method to be applied to the resource, the identifier of the resource, and the protocol version in use.

```
GET / HTTP/1.1
Host: protonmail.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:80.0)
Gecko/20100101 Firefox/80.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp
,*/*;q=0.8
Accept-Language: vi-VN,vi;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
```

a) Request-Line

The Request-Line begins with a method token, followed by the Request-URI and the protocol version, and ending with CRLF. The elements are separated by SP characters. No CR or LF is allowed except in the final CRLF sequence.

b) Method

The Method token indicates the method to be performed on the resource identified by the Request-URI. The method is case-sensitive.

c) Request-URI

The Request-URI is a Uniform Resource Identifier and identifies the resource upon which to apply the request.

d) Request Header Fields

The request-header fields allow the client to pass additional information about the request, and about the client itself, to the server. These fields act as request modifiers, with semantics equivalent to the parameters on a programming language method invocation.

1.1.2. HTTP Response

After receiving and interpreting a request message, a server responds with an HTTP response message.

a) Status-Line

The first line of a Response message is the Status-Line, consisting of the protocol version followed by a numeric status code and its associated textual phrase, with each element separated by SP characters. No CR or LF is allowed except in the final CRLF sequence.

b) Status Code and Reason Phrase

The Status-Code element is a 3-digit integer result code of the attempt to understand and satisfy the request. These codes are fully defined in section 10. The Reason-Phrase is intended to give a short textual description of the Status-Code. The Status-Code is intended for use by automata and the Reason-Phrase is intended for the human user. The client is not required to examine or display the Reason-Phrase.

The first digit of the Status-Code defines the class of response. The last two digits do not have any categorization role. There are 5 values for the first digit:

- 1xx: Informational - Request received, continuing process
- 2xx: Success - The action was successfully received, understood, and accepted
- 3xx: Redirection - Further action must be taken in order to complete the request
- 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
- 5xx: Server Error - The server failed to fulfill an apparently valid request

c) Response Header Fields

The response-header fields allow the server to pass additional information about the response which cannot be placed in the Status-Line. These header fields give information about the server and about further access to the resource identified by the Request-URI.

Response-header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields MAY be given the semantics of response-header fields if all parties in the communication recognize them to be response-header fields. Unrecognized header fields are treated as entity-header fields.

1.1.3. HTTP Lab

a) HTTP lab 1

Download the following file, and open it up in Wireshark:

http://asecuritysite.com/log/webpage.zip

In this case a host connects to a Web server. Determine the following:

- Using the filter of `http.request.method=="GET"`, identify the files that the host gets from the Web server?
- Using the filter of `http.response`, determine the response codes. Which files have transferred and which have been unsuccessful?
- Which is the default file name on the server when the user accesses the top levels of the domain?
- Which type of image files does the client want to accept?
- Which language/character set is used by the client?
- Which Web browser is the client using?
- Which Web server technology is the server using?
- On which date were the pages accessed?

b) HTTP lab 2

Download the following file, and open it up in Wireshark:

http://asecuritysite.com/log/googleWeb.zip

In this case a host connects to the Google Web server. Determine the following:

- Using the filter of `http.request.method=="GET"`, identify the files that the host gets from the Web server?
- Using the filter of `http.response`, determine the response codes. Which files have transferred and which have been unsuccessful?
- Which is the default file name on the server when the user accesses the top levels of the domain?
- Which type of image files does the client want to accept?
- Which language/character set is used by the client?
- Which Web browser is the client using?
- Which Web server technology is the server using?
- On which date were the pages accessed?

1.2. Modern Versus Legacy Web Applications

Today's web applications are often built on top of technology that didn't exist 10 years ago. The tools available for building web applications have advanced so much in that time frame that sometimes it seems like an entirely different specialization today. A decade ago, most web applications were built using server-side frameworks that rendered an HTML/JS/CSS page that would then be sent to the client. Upon needing an update, the client would simply request another page from the server to be rendered and piped over HTTP. Shortly after that, web applications began making use of HTTP more frequently with the rise of Ajax (asynchronous JavaScript and XML), allowing network requests to be made from within a page session via JavaScript. Today, many applications actually are more properly represented as two or more applications communicating via a network protocol, versus a single monolithic application. This is one major architectural difference between the web applications of today and the web applications of a decade ago.

Oftentimes today's web applications are comprised of several applications connected with a Representational State Transfer (REST) API. These APIs are stateless and only exist to fulfill requests from one application to another. This means they don't actually store any information about the requester. Many of today's client (UI) applications run in the browser in ways more akin to a traditional desktop application. These client applications manage their own life cycle loops, request their own data, and do not require a page reload after the initial

boot- strap is complete. It is not uncommon for a standalone application deployed to a web browser to communicate with a multitude of servers. Consider an image hosting application that allows user login—it likely will have a specialized hosting/distribution server located at one URL, and a separate URL for managing the database and logins. It's safe to say that today's applications are often actually a combination of many separate but symbiotic applications working together in unison. This can be attributed to the development of more cleanly defined network protocols and API architecture patterns.

The average modern-day web application probably makes use of several of the following technologies:

- REST API
- JSON or XML
- JavaScript
- SPA framework (React, Vue, EmberJS, AngularJS)
- An authentication and authorization system
- One or more web servers (typically on a Linux server)
- One or more web server software packages (ExpressJS, Apache, NginX)
- One or more databases (MySQL, MongoDB, etc.)
- A local data store on the client (cookies, web storage, IndexDB)

Some of these technologies existed a decade ago, but it wouldn't be fair to say they have not changed in that time frame. Databases have been around for decades, but NoSQL databases and client-side databases are definitely a more recent development. The development of full stack JavaScript applications was also not possible until NodeJS and npm began to see rapid adoption. The landscape for web applications has been changing so rapidly in the last decade or so that many of these technologies have gone from unknown to nearly everywhere. There are even more technologies on the horizon: for example, the Cache API for storing requests locally, and Web Sockets as an alternative network protocol for client-to-server (or even client-to-client) communication. Eventually, browsers intend to fully support a variation of assembly code known as web assembly, which will allow non-JavaScript languages to be used for writing client-side code in the browser. Each of these new and upcoming technologies brings with it new security

holes to be found and exploited for good or for evil. It is an exciting time to be in the business of exploiting or securing web applications.

1.3. REST APIs

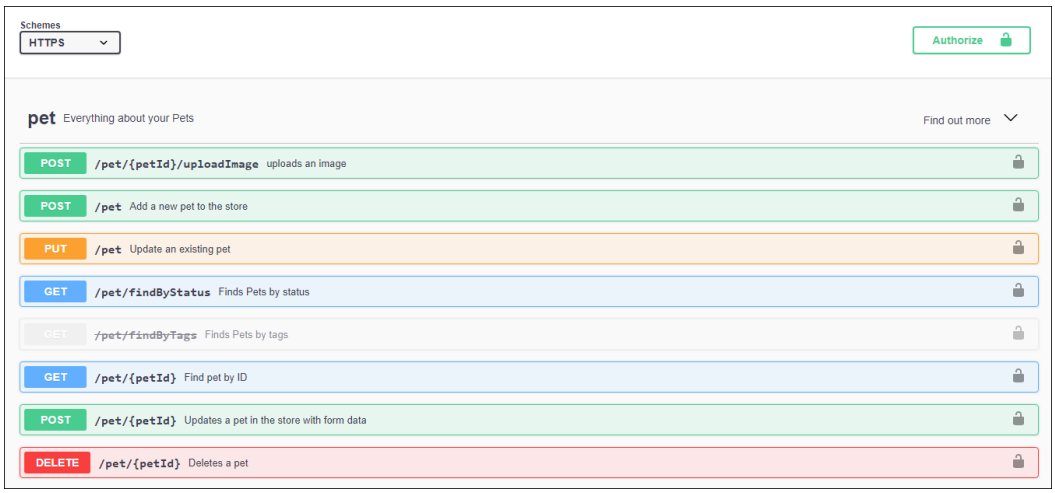
REST stands for Representational State Transfer, which is a fancy way of defining an API that has a few unique traits:

It must be separate from the client REST APIs are designed for building highly scalable, but simple, web applications. Separating the client from the API but following a strict API structure makes it easy for the client application to request resources from the API without being able to make calls to a database or perform server-side logic itself.

It must be stateless. By design, REST APIs only take inputs and provide outputs. The APIs must not store any state regarding the client's connection. This does not mean, however, that a REST API cannot perform authentication and authorization instead, authorization should be tokenized and sent on every request.

It must be easily cacheable To properly scale a web application delivered over the internet, a REST API must be able to easily mark its responses as cacheable or not. Because REST also includes very tight definitions on what data will be served from what endpoint, this is actually very easy to configure on a properly designed REST API. Ideally, the caches should be programmatically managed to not accidentally leak privileged information to another user.

Each endpoint should define a specific object or method Typically these are defined hierarchically; for example, `/moderators/joe/logs/12_21_2018`. In doing so, REST APIs can easily make use of HTTP verbs like GET, POST, PUT, and DELETE. As a result, one endpoint with multiple HTTP verbs becomes self-documenting.



In the past, most web applications used Simple Object Access Protocol (SOAP)- structured APIs. REST has several advantages over SOAP:

- Requests target data, not functions
- Easy caching of requests
- Highly scalable

Furthermore, while SOAP APIs must utilize XML as their in-transit data format, REST APIs can accept any data format, but typically JSON is used. JSON is much more lightweight (less verbose) and easier for humans to read than XML, which also gives REST an edge against the competition.

Here is an example payload written in XML:

```
<user>
<username>joe</username>
<password>correcthorsebatterystaple</password>
<email>joe@website.com</email>
<joined>12/21/2005</joined>
<client-data>
 <timezone>UTF</timezone>
 <operating-system>Windows 10</operating-system>
 <licenses>
 <videoEditor>abc123-2005</videoEditor>
 <imageEditor>123-456-789</imageEditor>
 </licenses>
</client-data>
</user>
```

And similarly, the same payload written in JSON:

```
{
"username": "joe",
"password": "correcthorsebatterystaple",
"email": "joe@website.com",
```

```
"joined": "12/21/2005",
"client_data": {
 "timezone": "UTF",
 "operating_system": "Windows 10",
 "licenses": {
 "videoEditor": "abc123-2005",
 "imageEditor": "123-456-789"
 }
}
}
```

Most modern web applications you will run into either make use of RESTful APIs, or a REST-like API that serves JSON. It is becoming increasingly rare to encounter SOAP APIs and XML outside of specific enterprise apps that maintain such rigid design for legacy compatibility. Understanding the structure of REST APIs is important as you attempt to reverse engineer a web application's API layer. Mastering the basic fundamentals of REST APIs will give you an advantage, as you will find that many APIs you wish to investigate follow REST architecture - but additionally, many tools you may wish to use or integrate your workflow with will be exposed via REST APIs.

1.4. JavaScript Object Notation

REST is an architecture specification that defines how HTTP verbs should map to resources (API endpoints and functionality) on a server. Most REST APIs today use JSON as their in-transit data format. Consider this: an application's API server must communicate with its client (usually some code in a browser or mobile app). Without a client/server relationship, we can- not have stored state across devices, and persist that state between accounts. All state would have to be stored locally. Because modern web applications require a lot of client/server communication (for the downstream exchange of data, and upstream requests in the form of HTTP verbs), it is not feasible to send data in ad hoc formats. The in-transit format of the data must be standardized.

JSON is one potential solution to this problem. JSON is an open-standard (not proprietary) file format that meets a number of interesting requirements:

- It is very lightweight (reduces network bandwidth).
- It requires very little parsing (reduces server/client hardware load).
- It is easily human readable.

- It is hierarchical (can represent complex relationships between data).
- JSON objects are represented very similarly to JavaScript objects, making consumption of JSON and building new JSON objects quite easy in the browser.

All major browsers today support the parsing of JSON natively (and fast!), which, in addition to the preceding bullet points, makes JSON a great format for transmitting data between a stateless server and a web browser.

The following JSON:

```
{
"first": "Sam",
"last": "Adams",
"email": "sam.adams@company.com",
"role": "Engineering Manager",
"company": "TechCo.",
"location": {
 "country": "USA",
 "state": "california",
 "address": "123 main st.",
 "zip": 98404
 }
}
```

Can be parsed easily into a JavaScript object in the browser:

```
const jsonString = `{
 "first": "Sam",
 "last": "Adams",
 "email" "sam.adams@company.com",
 "role": "Engineering Manager",
 "company": "TechCo.",
 "location": {
 "country": "USA",
 "state": "california",
 "address": "123 main st.",
 "zip": 98404
 }
}`;
// convert the string sent by the server to an object
const jsonObject = JSON.parse(jsonString);
```

JSON is flexible, lightweight, and easy to use. It is not without its drawbacks, as any lightweight format has trade-offs compared to heavyweight alternatives. These will be discussed later on in the book when we evaluate specific security differences between JSON and its competitors, but for now it’s important

to just grasp that a significant number of network requests between browsers and servers are sent as JSON today. Get familiar with reading through JSON strings, and consider installing a plug-in in your browser or code editor to format JSON strings. Being able to rapidly parse these and find specific keys will be very valuable when penetration testing a wide variety of APIs in a short time frame.

1.5. JavaScript

1.5.1. Fundamental

https://www.w3schools.com/js/

1.5.2. JavaScript Lab

Will provide later in Offline class

1.6. SPA Frameworks

Older websites were usually built on a combination of ad hoc script to manipulate the DOM, and a lot of reused HTML template code. This was not a scalable model, and while it worked for delivering static content to an end user, it did not work for delivering complex, logic-rich applications. Desktop application software at the time was robust in functionality, allowing for users to store and maintain application state. Websites in the old days did not provide this type of functionality, although many companies would have preferred to deliver their complex applications via the web as it provided many benefits from ease of use to piracy prevention.

Single-page application (SPA) frameworks were designed to bridge the functionality gap between websites and desktop applications. SPA frameworks allow for the development of complex JavaScript-based applications that store their own internal state, and are composed of reusable UI components, each of which has its own self-maintained life cycle, from rendering to logic execution.

SPA frameworks are rampant on the web today, backing the largest and most com- plex applications (such as Facebook, Twitter, and YouTube) where functionality is key and near-desktop-like application experiences are delivered.

1.7. Web Servers and Databases

1.7.1. Web Servers

A modern client-server web application relies on a number of technologies built on top of each other for the server-side component and client-side components to function as intended. In the case of the server, application logic runs on top of a software-based web server package so that application developers do not have to worry about handling requests and managing processes. The web server software, of course, runs on top of an operating system (usually some Linux distro like Ubuntu, CentOS, or RedHat), which runs on top of physical hardware in a data center somewhere.

But as far as web server software goes, there are a few big players in the modern web application world. Apache still serves nearly half of the websites in the world, so we can assume Apache serves the majority of web applications as well. Apache is open source, has been in development for around 25 years, and runs on almost every Linux distro, as well as some Windows servers.

1.7.2. Server-Side Databases

Once a client sends data to be processed to a server, the server must often persist this data so that it can be retrieved in a future session. Storing data in memory is not reliable in the long term, as restarts and crashes could cause data loss. Additionally, random-access memory is quite expensive when compared to disk. When storing data on disk, proper precautions need to be taken to ensure that the data can be reliably and quickly retrieved, stored, and queried. Almost all of today's web applications store their user-submitted data in some type of database - often varying the database used depending on the particular business logic and use case.

SQL databases are still the most popular general-purpose database on the market. SQL query language is strict, but reliably fast and easy to learn. SQL can be used for anything from storage of user credentials to managing JSON objects or small image blobs. The largest of these are PostgreSQL, Microsoft SQL Server, MySQL, and SQLite. When more flexible storage is needed, schema-less NoSQL databases can be employed. Databases like MongoDB, DocumentDB, and CouchDB store information as loosely structured "documents" that are flexible and can be modified at any time, but are not as easy or efficient at querying or

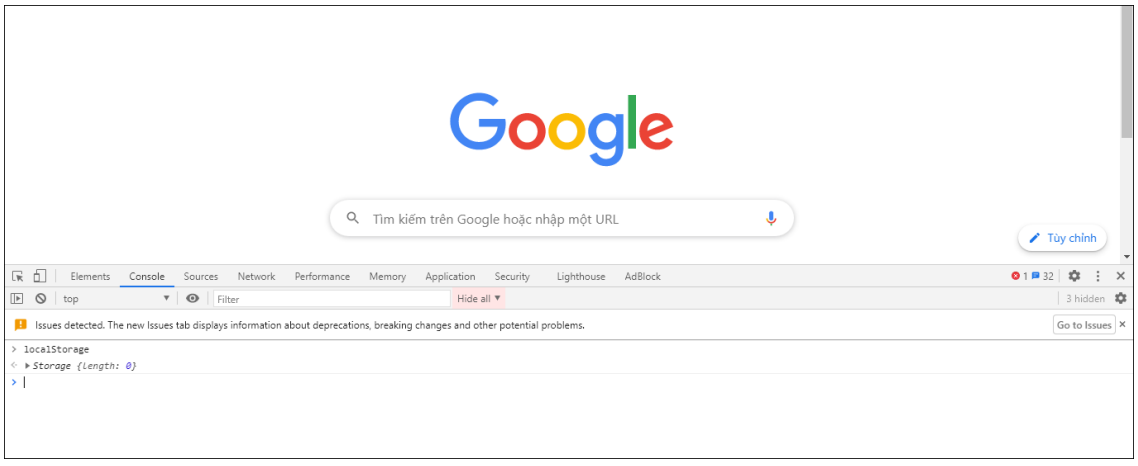
aggregating. In today's web application landscape, more advanced and particular databases also exist. Search engines often employ their own highly specialized databases that must be synchronized with the main database on a regular basis. An example of this is the widely popular Elasticsearch.

Each type of database carries unique challenges and risks. SQL injection is a wellknown vulnerability archetype effective against major SQL databases when queries are not properly formed. However, injection-style attacks can occur against almost any database if a hacker is willing to learn the database's query model. It is wise to consider that many modern web applications can employ multiple databases at the same time, and often do. Applications with sufficiently secure SQL query generation may not have sufficiently secure MongoDB or Elasticsearch queries and permissions.

1.8. Client-Side Data Stores

Traditionally, minimal data is stored on the client because of technical limitations and cross-browser compatibility issues. This is rapidly changing. Many applications now store significant application state on the client, often in the form of configuration data or large scripts that would cause network congestion if they had to be downloaded on each visit.

In most cases, a browser-managed storage container called local storage is used for storing and accessing key/value data from the client. Local storage follows browser enforced Same Origin Policy (SOP), which prevents other domains (websites) from accessing each other's locally stored data. Web applications can maintain state even when the browser or tab is closed.



A subset of local storage called session storage operates identically, but persists data only until the tab is closed. This type of storage can be used when

data is more critical and should not be persisted if another user uses the same machine.

Finally, for more complex applications, browser support for IndexedDB is found in all major web browsers today. IndexedDB is a JavaScript-based object oriented programming (OOP) database capable of storing and querying asynchronously in the background of a web application.

CHAPTER 2. SECURE SOFTWARE CONCEPTS

Secure software development is intimately tied to the information security domain. For members of the software development team to develop secure software, a reasonable knowledge of security principles is required. The first knowledge domain area, Secure Software Concepts, comprises a collection of principles, tenets, and guidelines from the information security domain. Understanding these concepts as they apply to software development is a foundation of secure software development.

2.1. Security Basics

Security can be defined in many ways, depending upon the specific discipline that it is being viewed from. From an information and software development point of view, some specific attributes are commonly used to describe the actions associated with security: *confidentiality*, *integrity*, and *availability*. A second set of action-oriented elements, *authentication*, *authorization*, and *auditing*, provide a more complete description of the desired tasks associated with the information security activity. A final term, *non-repudiation*, describes an act that one can accomplish when using the previous elements. An early design decision is determining what aspects of protection are required for data elements and how they will be employed.

2.1.1. Confidentiality

Confidentiality is the concept of preventing the disclosure of information to unauthorized parties. Keeping secrets secret is the core concept of confidentiality. The identification of authorized parties makes the attainment of confidentiality dependent upon the concept of authorization, which is presented later in this chapter. There are numerous methods of keeping data confidential, including access controls and encryption. The technique employed to achieve confidentiality depends upon whether the data is at rest, in transit, or in use. Access controls are typically preferred for data in use and at rest, while encryption is common for data in transit and at rest.

2.1.2. Integrity

Integrity is similar to confidentiality, except rather than protecting the data from unauthorized access, integrity refers to protecting the data from unauthorized

alteration. Unauthorized alteration is a more fine-grained control than simply authorizing access. Users can be authorized to view information but not alter it, so integrity controls require an authorization scheme that controls update and delete operations. For some systems, protecting the data from observation by unauthorized parties is critical, whereas in other systems, it is important to protect the data from unauthorized alteration. Controlling alterations, including deletions, can be an essential element in a system's stability and reliability. Integrity can also play a role in the determination of authenticity.

2.1.3. Availability

Access to systems by authorized personnel can be expressed as the system's availability. Availability is an often-misunderstood attribute, but its value is determined by the criticality of the data and its purpose in the system. For systems such as email or web browsing, temporary availability issues may not be an issue at all. For IT systems that are controlling large industrial plants, such as refineries, availability may be the most important attribute. The criticality of data and its use in the system are critical factors in determining a system's availability. The challenge in system definition and design is to determine the correct level of availability for the data elements of the system.

The objective of security is to apply the appropriate measures to achieve a desired risk profile for a system. One of the challenges in defining the appropriate control objectives for a system is classifying and determining the appropriate balance of the levels of confidentiality, integrity, and availability in a system across the data elements. Although the attributes are different, they are not necessarily contradictory. They all require resources, and determining the correct balance between them is a key challenge early in the requirements and design process.

2.1.4. Authentication

Authentication is the process of determining the identity of a user. All processes in a computer system have an identity assigned to them so that a differentiation of security functionality can be employed. Authentication is a foundational element of security, as it provides the means to define the separation of users by allowing the differentiation between authorized and unauthorized users.

In systems where all users share a single account, they share the authentication and identity associated with that account.

It is the job of authorization mechanisms to ensure that only valid users are permitted to perform specific allowed actions. Authentication, on the other hand, deals with verifying the identity of a subject. To help understand the difference, consider the example of an individual attempting to log in to a computer system or network. Authentication is the process used to verify to the computer system or network that the individual is who they claim to be. Authorization refers to the rules that determine what that individual can do on the computer system or network after being authenticated. Three general methods are used in authentication. To verify their identity, a user can provide:

- Something you know
- Something you have
- Something about you (something that you are)

The most common authentication mechanism is to provide something that only you, the valid user, should know. The most common example of something you know is the use of a userid (or username) and password. In theory, since you are not supposed to share your password with anybody else, only you should know your password, and thus by providing it you are proving to the system that you are who you claim to be. Unfortunately, for a variety of reasons, such as the fact that people tend to choose very poor and easily guessed passwords or share them, this technique to provide authentication is not as reliable as it should be. Other, more secure, authentication mechanisms are consequently being developed and deployed.

Originally published by the U.S. government in one of the "rainbow series" of manuals on computer security, the categories of shared "secrets" are as follows:

- What users know (such as a password)
- What users have (such as tokens)
- What users are (static biometrics such as fingerprints or iris pattern)

Today, because of technological advances, new categories have emerged, patterned after subconscious behaviors and measurable attributes:

- What users do (dynamic biometrics such as typing patterns or gait)
- Where a user is (actual physical location)

Another common method to provide authentication involves the use of something that only valid users should have in their possession, commonly referred to as a token. A physical-world example of this is the simple lock and key. Only those individuals with the correct key will be able to open the lock and thus achieve admittance to your house, car, office, or whatever the lock was protecting. For computer systems, the token frequently holds a cryptographic element that identifies the user. The problem with tokens is that people can lose them, which means they can't log in to the system, and somebody else who finds the key may then be able to access the system, even though they are not authorized. To address the lost token problem, a combination of the something-you-know and something-you-have methods is used—requiring a password or PIN in addition to the token. The key is useless unless you know this code. An example of this is the ATM card most of us carry. The card is associated with a personal identification number (PIN), which only you should know. Knowing the PIN without having the card is useless, just as having the card without knowing the PIN will also not provide you access to your account. Properly configured tokens can provide high levels of security at an expense only slightly higher than passwords.

The third authentication method involves using something that is unique about the user. We are used to this concept from television police dramas, where a person's fingerprints or a sample of their DNA can be used to identify them. The field of authentication that uses something about you or something that you are is known as biometrics. A number of different mechanisms can be used to accomplish this form of authentication, such as a voice print, a retinal scan, or hand geometry. The downside to these methods is the requirement of additional hardware and the lack of specificity that can be achieved with other methods.

2.1.5. Authorization

After the authentication system identifies a user, the authorization system takes over and applies the predetermined access levels to the user. Authorization is the process of applying access control rules to a user process and determining if a particular user process can access an object. There are numerous forms of access control systems, and these are covered later in the chapter. Three elements are used in the discussion of authorization: a requestor (sometimes referred to as the subject), the object, and the type or level of access to be granted. The

authentication system identifies the subject to be one of a known set of subjects associated with a system. When a subject requests access to an object, be it a file, a program, an item of data, or any other resource, the authorization system makes the access determination as to grant or deny access. A third element is the type of access requested, with the common forms being read, write, create, delete, or the right to grant access rights to other subjects.

2.1.6. Accounting (Auditing)

Accounting is a means of measuring activity. In IT systems, this can be done by logging crucial elements of activity as they occur. With respect to data elements, accounting is needed when activity is determined to be crucial to the degree that it may be audited at a later date and time. Management has a responsibility for ensuring work processes are occurring as designed. Should there be a disconnect between planned and actual operational performance metrics, then it is management's responsibility to initiate and ensure corrective actions are taken and effective. Auditing is management's lens to observe the operation in a nonpartisan manner. Auditing is the verification of what actually happened on a system. Security-level auditing can be performed at several levels, from an analysis of the logging function that logs specific activities of a system, to the management verification of the existence and operation of specific controls on a system.

Auditing can be seen as a form of recording historical events in a system. Operating systems have the ability to create audit structures, typically in the form of logs that allow management to review activities at a later point in time. One of the key security decisions is the extent and depth of audit log creation. Auditing takes resources, so by default it is typically set to a minimal level. It is up to a system operator to determine the correct level of auditing required based on a system's criticality. The system criticality is defined by the information criticality associated with the information manipulated or stored within it. Determination and establishment of audit functionality must occur prior to an incident, as the recording of the system's actions cannot be accomplished after the fact.

Audit logs are a kind of balancing act. They require resources to create, store, and review. The audit logs in and of themselves do not create security; it is only through the active use of the information contained within them that security functionality can be enabled and enhanced. As a general rule, all critical

transactions should be logged, including when they occurred and which authorized user is associated with the event. Additional metadata that can support subsequent investigation of a problem is also frequently recorded.

2.1.7. Non-repudiation

Non-repudiation is the concept of preventing a subject from denying a previous action with an object in a system. When authentication, authorization, and auditing are properly configured, the ability to prevent repudiation by a specific subject with respect to an action and an object is ensured. In simple terms, there is a system in place to prevent a user from saying they did not do something, a system that can prove, in fact, whether an event took place or not. Non-repudiation is a very general concept, so security requirements must specify the subject, objects, and events for which non-repudiation is desired, as this will affect the level of audit logging required. If complete non-repudiation is desired, then every action by every subject on every object must be logged, and this could be a very large log dataset.

2.2. System Tenets

The creation of software systems involves the development of several foundational system elements within the overall system. Communication between components requires the management of a communication session, commonly called session management. When a program encounters an unexpected condition, an error can occur. Securely managing error conditions is referred to as exception management. Software systems require configuration in production, and configuration management is a key element in the creation of secure systems.

2.2.1. Session Management

Software systems frequently require communications between program elements, or between users and program elements. The control of the communication session between these elements is essential to prevent the hijacking of an authorized communication channel by an unauthorized party. Session management refers to the design and implementation of controls to ensure that communication channels are secured from unauthorized access and disruption of a communication. A common example is the Transmission Control Protocol (TCP) handshake that enables the sequential numbering of packets and allows for packet retransmission for missing packets; it also prevents the introduction of

unauthorized packets and the hijacking of the TCP session. Session management requires additional work and has a level of overhead, and hence may not be warranted in all communication channels. User Datagram Protocol (UDP), a connectionless/sessionless protocol, is an example of a communication channel that would not have session management and session-related overhead. An important decision early in the design process is determining when sessions need to be managed and when they do not. Understanding the use of the channel and its security needs should dictate the choice of whether session management is required or not.

2.2.2. Exception Management

There are times when a system encounters an unknown condition, or is given input that results in an error. The process of handling these conditions is referred to as exception management. A remote resource may not respond, or there may be a communication error—whatever the cause of the error, it is important for the system to respond in an appropriate fashion. Several criteria are necessary for secure exception management. First, all exceptions must be detected and handled. Second, the system should be designed so as not fail to an insecure state. Last, all communications associated with the exception should not leak information.

For example, assume a system is connecting to a database to verify user credentials. Should an error occur, as in the database is not available when the request is made, the system needs to properly handle the exception. The system should not inadvertently grant access in the event of an error. The system may need to log the error, along with information concerning what caused the error, but this information needs to be protected. Releasing the connection string to the database or passing the database credentials with the request would be a security failure.

2.2.3. Configuration Management

Dependable software in production requires the managed configuration of the functional connectivity associated with today's complex, integrated systems. Initialization parameters, connection strings, paths, keys, and other associated variables are typical examples of configuration items. As these elements can have significant effects upon the operation of a system, they are part of the system and need to be properly controlled for the system to remain secure. The identification

and management of these elements is part of the security process associated with a system.

Management has a responsibility to maintain production systems in a secure state, and this requires that configurations be protected from unauthorized changes. This has resulted in the concept of configuration management, change control boards, and a host of workflow systems designed to control the configuration of a system. One important technique frequently employed is the separation of duties between production personnel and development/test personnel. This separation is one method to prevent the contamination of approved configurations in production.

2.3. Authentication Labs

2.3.1. Wireshark Form Base Authentication Lab

Will provide later in Offline Class

2.3.2. Wireshark Basic Authentication Lab

Will provide later in Offline Class

2.3.3. SEToolkit Attack Authentication

Will provide later in Offline Class

CHAPTER 3. SOFTWARE DEVELOPMENT METHODOLOGIES

Software development methodologies have been in existence for decades, with new versions being developed to capitalize on advances in teamwork and group functionality. While security is not itself a development methodology, it has been shown by many groups and firms that security functionality can be added to a development lifecycle, creating a secure development lifecycle. While this does not guarantee a secure output, including security in the process used to develop software has been shown to dramatically reduce the defects that cause security bugs.

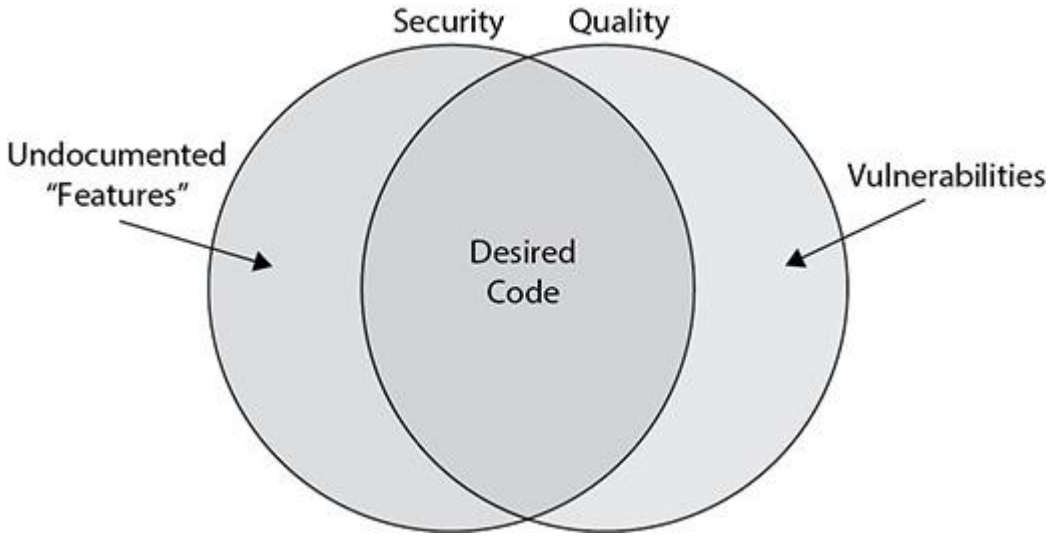
3.1. Secure Development Lifecycle

The term “secure development lifecycle” (SDL) comes from adding security to a software development lifecycle to reduce the number of security bugs in the software being produced. There are a wide range of different software development lifecycle models in use today across software development firms. To create a secure development lifecycle model, all one has to do is add a series of process checks to enable the development process to include the necessary security elements in the development process. The elements that are added are the same, although the location and methodology of adding the new elements to the process are dependent upon the original process.

3.1.1. Security vs. Quality

Quality has a long history in manufacturing and can be defined as fitness for use. Quality can also be seen as absence of defects. Although this may seem to be the same thing as security, it is not, but they are related. Software can be of high quality and free of defects and still not be secure. The converse is the important issue; if software is not of high quality and has defects, then it is not going to be secure. Because a significant percentage of software security issues in practice are due to basic mistakes where the designer or developer should have known better, software quality does play a role in the security of the final product. Ross Anderson, a renowned expert in security, has stated that investments in software quality will result in a reduction of security issues, whether the quality program targets security issues or not.

If the product has quality but lacks security, then the result is a set of vulnerabilities. If the software is secure but is lacking in quality, then undocumented features may exist that can result in improper or undesired behaviors. The objective is to have both quality and security in the final output of the SDL process.



3.1.2. Security Features != Secure Software

A common misconception is when someone confuses security features with secure software. Security features are elements of a program specifically designed to provide some aspect of security. Adding encryption, authentication, or other security features can improve the usability of software and thus are commonly sought after elements to a program. This is not what secure development is about. Secure software development is about ensuring that all elements of a software package are constructed in a fashion where they operate securely. Another way of looking at this is the idea that secure software does what it is designed to do and only what it is designed to do. Adding security features may make software more marketable from a features perspective, but this is not developing secure software. If the software is not developed to be secure, then even the security features cannot be relied upon to function as desired. An example of this was the case of the Debian Linux random-number bug, where a design flaw resulted in a flawed random-number function which, in turn, resulted in cryptographic failures.

3.2. Secure Development Lifecycle Components

SDLs contain a set of common components that enable operationalization of security design principles. The first of these components is a current team-

awareness and education program. Having a team that is qualified to do the task in an SDL environment includes security knowledge. The next component is the use of security gates as a point to check compliance with security requirements and objectives. These gates offer a chance to ensure that the security elements of the process are indeed being used and are functioning to achieve desired outcomes. Three sets of tools—bug tracking, threat modeling, and fuzzing—are used to perform security-specific tasks as part of the development process. The final element is a security review, where the results of the SDL process are reviewed to ensure that all of the required activities have been performed and completed to an appropriate level.

3.2.1. Software Team Awareness and Education

All team members should have appropriate training and refresher training throughout their careers. This training should be focused on the roles and responsibilities associated with each team member. As the issues and trends associated with both development and security are ever changing, it is important for team members to stay current in their specific knowledge so that they can appropriately apply it in their work.

Security training can come in two forms: basic knowledge and advanced topics. Basic security knowledge, including how it is specifically employed and supported as part of the SDL effort, is an all-hands issue and all team members need to have a functioning knowledge of this material. Advanced topics can range from new threats to tools, techniques, etc., and are typically aimed at a specific type of team member (i.e., designer, developer, tester, project manager). The key element of team awareness and education is to ensure that all members are properly equipped with the correct knowledge before they begin to engage in the development process.

3.2.2. Gates and Security Requirements

As part of the development process, periodic reviews are conducted. In an SDL, these are referred to as gates. The term “gate” is used, as it signifies a condition that one must pass through. To pass the security gate, a review of the appropriate security requirements is conducted. Missing or incomplete elements can prevent the project from advancing to the next development phase until these elements or issues are addressed. This form of discipline, if conducted in a firm

and uniform manner, results in eventual behavior by the development team where the gates are successfully negotiated as a part of normal business. This is the ultimate objective; the inclusion of security is a part of the business process.

3.2.3. Bug Tracking

Bug tracking is a basic part of software development. As code is developed, bugs are discovered. Bugs are elements of code that have issues that result in undesired behaviors. Sometimes, the behavior results in something that can be exploited, and this makes it a potential security bug. Bugs need to be fixed, and hence, they are tracked to determine their status. Some bugs may be obscure, impossible to exploit, and expensive to fix; thus, the best economic decision may be to leave them until the next major rewrite, saving cost now on something that is not a problem. Tracking all of the bugs and keeping a log so that things can be fixed at appropriate times are part of managing code development.

Security bug tracking is similar to regular bug tracking. Just because something is deemed a security bug does not mean that it will always be fixed right away. Just as other bugs have levels of severity and exploitability, so do security bugs. A security bug that is next to impossible to exploit and has a mitigating factor covering it may not get immediate attention, especially if it would necessitate a redesign. Under these circumstances, the security bug could be left until the next update of the code base.

There are many ways to score security bugs; a common method is based on the risk principle of impact (damage) times probability of occurrence. The DREAD model addresses this in a simple form:

Risk = Impact * Probability

Impact = (**D**RE**A**D)

- Damage. Note that Damage needs to be assessed in terms of Confidentiality, Integrity, and Availability
- Affected Users (How large is the user base affected?)

Probability = (D**RE**A**D**)

- Reproducibility (How difficult to reproduce? Is it scriptable?)
- Exploitability (How difficult to use the vulnerability to effect the attack?)
- Discoverability (How difficult to find?)

Measuring each of the DREAD items on a 1 to 10 scale, with 1 being the least damage or least likely to occur and 10 being the most damaging or likely to occur, provides a final measure that can be used to compare the risk associated with different bugs.

The acronym DREAD refers to a manner of classifying bugs: Damage potential, Reproducibility, Exploitability, Affected user base, and Discoverability.

One of the problems with scoring bugs has to do with point of view. A developer may see a particular bug as hard to exploit, whereas a tester viewing the same bug from a different context may score it as easy to exploit. Damage potential is also a highly context-sensitive issue. This makes detailed scoring of bugs subjective and unreliable. A simple triage method based on a defined set of severities—critical, important, moderate, and low—will facilitate a better response rate on clearing the important issues. A simple structure such as this is easy to implement, difficult for team members to game or bypass, and provides a means to address the more important issues first.

This brings us to the topic of bug bars. A bug bar is a measurement level that, when exceeded, indicates that the bug must be fixed prior to delivering the product to customers. Bugs that come in below the bar can wait until a later date for fixing. Bug bars are an important tool for prioritizing fixes while maintaining a level of code security. Any bug that exceeds the bar must be addressed in the current release, thus making bug fixes an issue based on risk, not ease of closure.

https://www.google.com/about/appsecurity/reward-program/

https://bugcrowd.com/vulnerability-rating-taxonomy#methodology

3.2.4. Threat Modeling

Threat modeling is a design technique used to communicate information associated with a threat throughout the development team. The threat modeling effort begins at the start of the project and continues throughout the development effort. The purpose of threat modeling is to completely define and describe threats to the system under development. In addition, information as to how the threat will be mitigated can be recorded. Communicating this material among all members of the development team enables everyone to be on the same page with respect to understanding and responding to threats to the system.

Threat modeling begins as a team effort, where multiple team members with differing levels of experience and expertise examine a design with respect to where it can be attacked. Threat modeling is best performed during the application design phase, as it is easier to make application changes before coding. It is also less costly than adding mitigations and testing them after code has been implemented and onwards.

The threat modeling process is designed around the activities performed as part of the software development process. Beginning with examining how the data flows through the system provides insight into where threats can exist. Beginning at the highest level, and typically using data flow diagrams, the threat model effort is focused on how data moves through an application. The elements that require attention include:

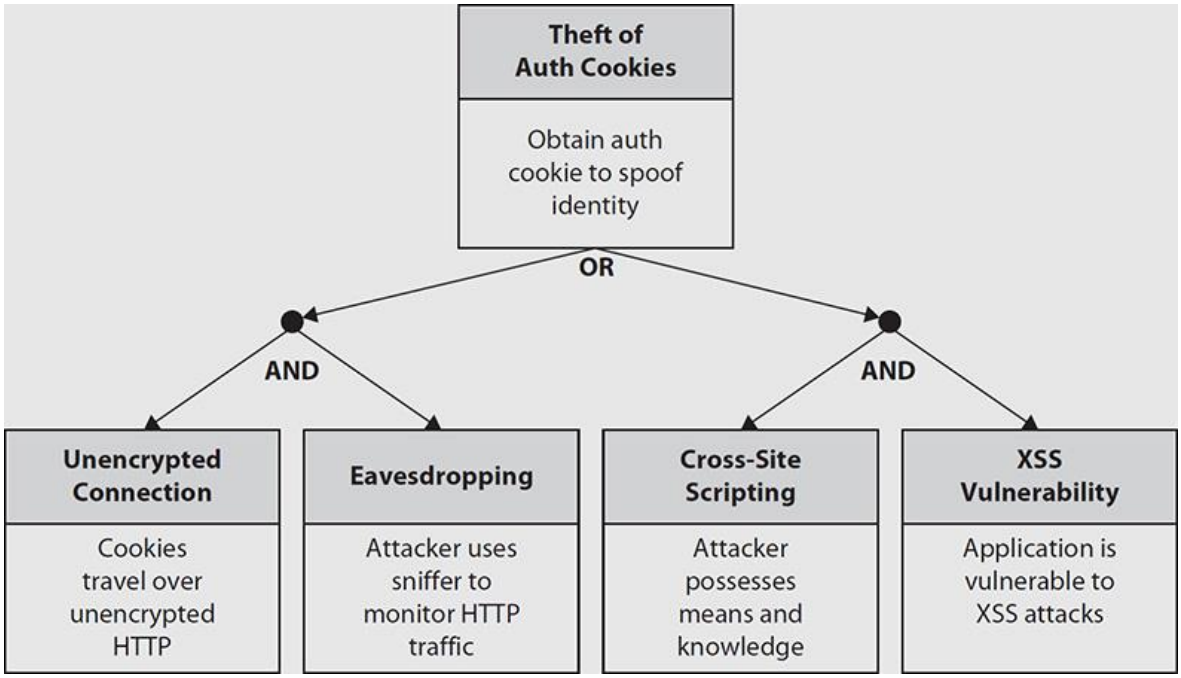
- The application as a whole
- Security and privacy features
- Features whose failures have security or privacy implications
- Features that cross trust boundaries

Close attention can be paid to the point where data crosses trust boundaries. At each location, a series of threats is examined. Microsoft uses the mnemonic STRIDE to denote the types of threats.

The term STRIDE refers to sources of threats: Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege.

As the threat model is constructed, the information about threats, their source, the risk, and mitigation methods are documented. This information can be used throughout the development process. Keeping the concept of threats, vulnerabilities, and risk up front in the developers' minds as they work results in software where the number and severity of vulnerabilities are reduced.

Threat Trees. Part of threat modeling is examining the sources of threats to software. Threat trees are a graphical representation of the elements that must exist for a threat to be realized. Threat trees present a threat in terms of a logical combination of elements, using ANDs and ORs to show the relationships between the required elements. This assists the development team, for if two elements are required, A AND B, then the blocking of either A or B will mitigate the threat.



3.2.5. Fuzzing

Fuzzing is a test technique where the tester applies a series of inputs to an interface in an automated fashion and examines the outputs for undesired behaviors. This technique is commonly used by hackers to discover unhandled input exceptions. The concept is fairly simple: For each and every input to a system, a test framework presents a variety of inputs and monitors the results of the system response. System crashes and unexpected behaviors are then examined for potential exploitation. Two different kinds of inputs can be presented: random and structured. Random inputs can find buffer overflows and other types of input vulnerabilities. Structured inputs can be used to test for injection vulnerabilities, cross-site scripting, and input-specific vulnerabilities such as arithmetic overflow conditions. The two can be used together to manipulate payloads via buffer overflows.

Fuzzing frameworks can present two different forms of inputs: mutation or generation based. Mutation-based input streams use samples of existing data, mutating them into different forms. Generation-based input streams create input streams based on models of the system. Fuzzing is frequently connected to other forms of testing, as in white box and black box testing. Fuzzing has also been responsible for finding a large number of vulnerabilities in software. The reasons for this are twofold: First, it is relatively easy to set up a large number of tests, and

second, it attacks the data input function, one of the more vulnerable points in software.

3.2.6. Security Reviews

Adding a series of security-related steps to the SDL can improve the software security. But as the old management axiom goes, you get what you measure, so it is important to have an audit function that verifies that the process is functioning as desired. Security reviews operate in this fashion. Sprinkled at key places, such as between stages in the SDL, the purpose of these reviews is not to test for security, but rather to examine the process and ensure that the security-related steps are being carried out and not being short-circuited in order to expedite the process. Security reviews act as moments where the process can be checked to ensure that the security-related elements of the process are functioning as designed. It may seem to be a paperwork drill, but it is more than that. Security reviews are mini-audit type events where the efficacy of the security elements of the SDL process is being checked.

3.2.7. Mitigations

Not all risks are created equal. Some bugs pose a higher risk to the software and need fixing more than lesser risk-related bugs. Use of a system such as DREAD in the gating process of bug tracking informs us of the risk level (Risk = Impact * Probability) associated with which bugs pose the greatest risk and therefore should be fixed first. The mitigation process itself defines how these bugs can be addressed.

When bugs are discovered, there is a natural tendency to want to just fix the problem. But as in most things in life, the devil is in the details, and most bugs are not easily fixed with a simple change of the code. Why? Well, these errors are caught much earlier and rooted out before they become a “security bug.” When a security bug is determined to be present, four standard mitigation techniques are available to the development team:

- Do nothing
- Warn the user
- Remove the problem
- Fix the problem

The first, do nothing, is a tacit acknowledgment that not all bugs can be removed if the software is to ship. This is where the bug bar comes into play; if the bug poses no significant threat (i.e., is below the critical level of the bug bar), then it can be noted and fixed in a later release. Warning the user is in effect a cop-out. This pushes the problem on to the user and forces them to place a compensating control to protect the software. This may be all that is available until a patch can be deployed for serious bugs or until the next release for minor bugs. This does indicate a communication with the user base, which has been shown to be good business, for most users hate surprises, as in when they find out about bugs that they feel the software manufacturer should have warned them of in advance.

Removing the problem is a business decision, for it typically involves disabling or removing a feature from a software release. This mitigation has been used a lot by software firms when the feature was determined not to be so important as to risk an entire release of the product. If a fix will take too long or involve too many resources, then this may be the best business recourse. The final mitigation, fixing the problem, is the most desired method and should be done whenever it is feasible given time and resource constraints. If possible, all security bugs should be fixed as soon as possible, for the cascading effect of multiple bugs cannot be ignored.

3.3. Software Development Models

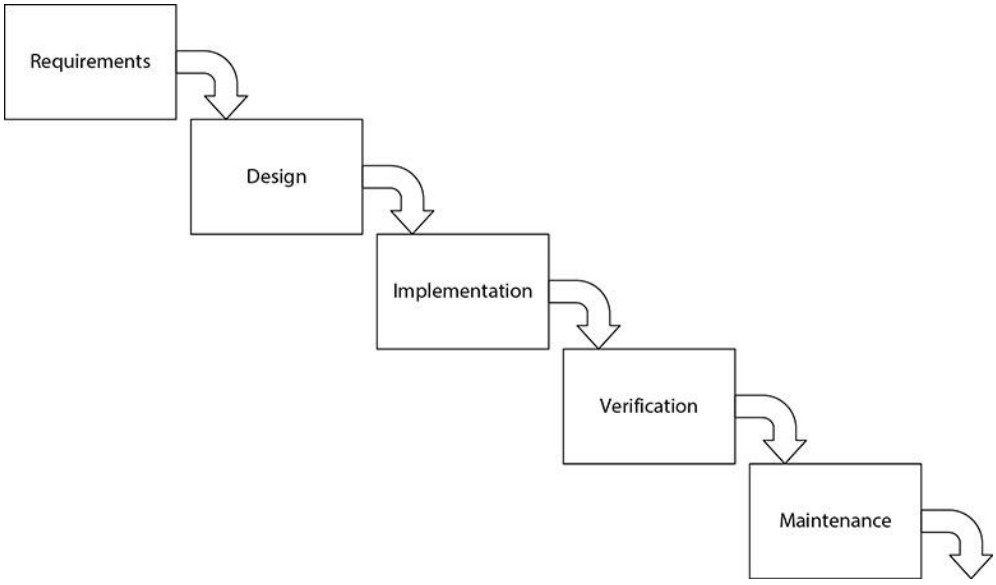
Professional software development is a methodical and managed process. It is a team effort that begins with requirements, and then through a process of orchestrated steps, designs, implements, and tests the software solution to the requirements. Many different methodologies are employed to achieve these objectives. The most common methodologies are variants of the following basic models:

- Waterfall model
- Spiral model
- Prototype model
- Agile model

In today's environment, most firms have adopted a variant of one or more of these models that are used to guide the development process. Each model has its strengths and weaknesses, as described in the following sections.

3.3.1. Waterfall

The waterfall model is a development model based on simple manufacturing design. The work process begins and progresses through a series of steps, with each step being completed before progressing to the next step. This is a linear, sequential process, without any backing up and repeating of earlier stages. A simple model where the stages of requirements precede design and design precedes coding, etc. Should a new requirement "be discovered" after the requirement phase is ended, it can be added, but the work does not go back to that stage. This makes the model very nonadaptive and difficult to use unless there is a method to make certain that each phase is truly completed before advancing the work. This can add to development time and cost. For these and other reasons, the waterfall model, although conceptually simple, is considered by most experts as nonworkable in practice.

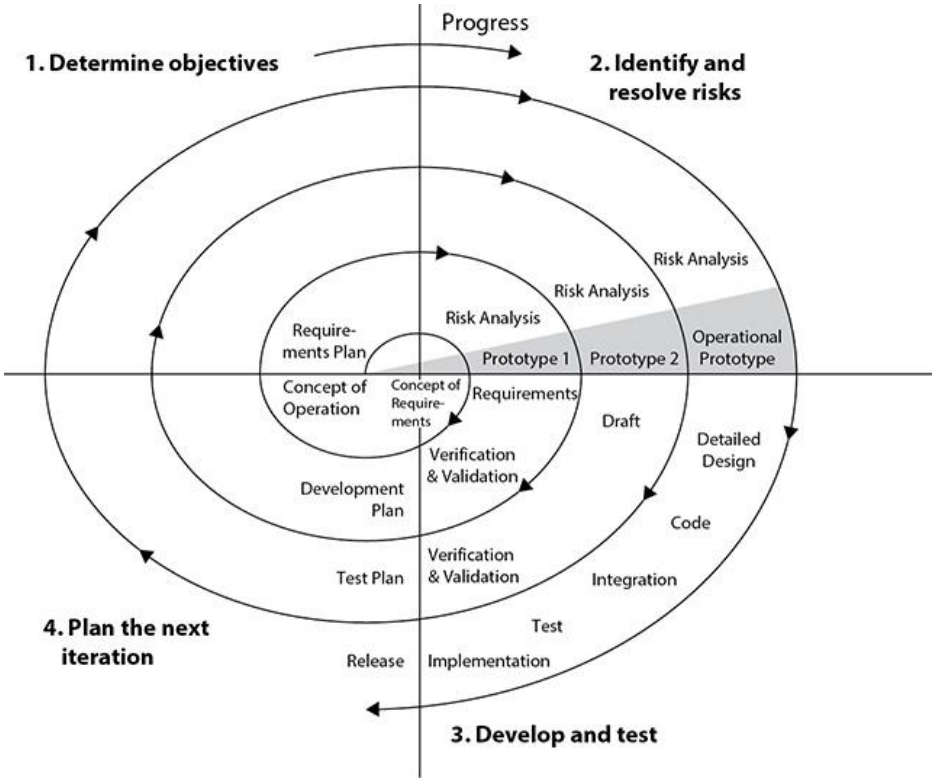


The waterfall methodology is particularly poorly suited for complex processes and systems where many of the requirements and design elements will be unclear until later stages of development. It is useful for small, bite-sized pieces, and in this manner is incorporated within other models such as the spiral and agile methods.

3.3.2. Spiral

The spiral model is an iterative model for software development. The process operates in a spiral fashion, where specific process steps can be repeated in an iterative fashion. This model stems from a prototyping mentality, where things are built in increments, adding functionality with each trip through the process

steps. The spiral model can include many different elements in each cycle, and the level of detail is associated with the aspect and context of application. Documentation is created as the process occurs, but only the documentation for the current work. Risk analysis can be applied at each cycle, both to the current and future iterations, thus assisting in planning.



The spiral model is suited for larger projects, although a large project may consist of several interconnected spirals, mainly to keep scope manageable in each group. A smaller-scale version of the spiral model can be seen in some of the agile methods, which will be presented in a later section.

3.3.3. Prototype

Just as the waterfall model can trace its roots to manufacturing, so can prototyping. A prototype is a representative model designed to demonstrate some set of functionality. In the software realm, this could be mockups of user input screens or output functions. Prototyping comes in two flavors: throwaway and evolutionary. In throwaway prototyping, a simulation is constructed to gain information on some subset of requirements. Mocking up user input screens is a form of prototyping, one that can gain valuable user feedback into the design process.

In evolutionary prototyping, a mockup that provides some form of functionality is created, allowing testing of the designed functionality. Should the design pass, then other elements can be added, building the system in a form of accretion. Prototyping can also be either horizontal or vertical in nature. A vertical prototype is a stovepipe-type element, such as a data access function to a database. This allows exploration of requirements and design elements in a limited working environment.

A horizontal prototype is more of a framework or infrastructure. This is also commonly used in software development, as most major software is modular in nature and can work well in a defined framework, one that can be tested and explored early in the development process via a prototype.

Prototyping can work well in conjunction with other incremental development methodologies. It can work inside the spiral model and as part of an agile development process. One of the primary reasons for prototyping is to reduce risk. By limiting early resource expenditure while obtaining a model that allows testing of major design elements and ideas, prototyping enables better risk management of complex development processes.

3.3.4. Agile Methods

Agile methods are not a single development methodology, but a whole group of related methods. Designed to increase innovation and efficiency of small programming teams, agile methods rely on quick turns involving small increases in functionality. The use of repetitive, small development cycles can enable different developer behaviors that can result in more efficient development. There are many different methods and variations, but some of the major forms of agile development are:

- Scrum
- XP (extreme programming)

XP is built around the people side of the process, while scrum is centered on the process perspective.

Additional agile methods include methods such as Lean and Kanban software development, Crystal methodologies (aka lightweight methodologies), dynamic systems development method (DSDM), and feature-driven development (FDD). The key to securing agile methods is the same as any other methodology.

By incorporating security steps into the methodology, even agile can be a secure method. Conversely, any methodology that ignores security issues will not produce secure code.

a) Scrum

The scrum programming methodology is built around a 30-day release cycle. Highly dependent upon a prioritized list of high-level requirements, program changes are managed on a 24-hour and 30-day basis. The concept is to keep the software virtually always ready for release. The master list of all tasks is called the product backlog. The 30-day work list is referred to as a sprint, and the daily accomplishment is called the burn-down chart.

From a security perspective, there is nothing in the scrum model that prevents the application of secure programming practices. To include security requirements into the development process, they must appear on the product and sprint backlogs. This can be accomplished during the design phase of the project. As additions to the backlogs can occur at any time, the security team can make a set of commonly used user stories that support required security elements. The second method of incorporating security functionality is through developer training. Developers should be trained on security-related elements of programming, such as validating user input, using only approved libraries, etc.

The advantage of scrum is quick turns of incremental changes to a software base. This makes change management easier. There is nothing about the scrum model that says developers do not have the same level of coding responsibility that is present in other models. There are limitations in the amount of planning, but in a mature agile environment, the security user stories can be already built and understood. The only challenge is ensuring that the security elements on the product and sprint backlogs get processed in a timely manner, but this is a simple management task. Security tasks tend to be less exciting than features, so keeping them in the process stack takes management effort.

b) XP

Extreme programming is a structured process that is built around user stories. These stories are used to architect requirements in an iterative process that uses acceptance testing to create incremental advances. The XP model is built around the people side of the software development process and works best in

smaller development efforts. Like other agile methods, the idea is to have many incremental small changes on a regular time schedule. XP stresses team-level communication, and as such, is highly amenable to the inclusion of security methods.

One of the hallmarks of XP methods is the use of feedback in a team environment to manage the three-step process of release planning/iteration/acceptance testing. New user stories can (and are) introduced through this feedback mechanism, as well as reports of bugs. Security can be included in the programming model in several ways. First, it can be done through a series of security requirements, proposed as user stories and run along with all of the other requirements. Second, programmers can use appropriate programming practices, such as the use of approved functions and libraries, approved design criteria, validate all inputs and outputs, etc. In this respect, XP programmers have the same level of responsibilities as do all other programmers in other development methodologies. Last, the concept of abuser stories has been proposed and is used by some teams to examine security-specific elements and ensure their inclusion in the development process. Abuser stories are similar to user stories, but rather than listing functionalities desired in the software, they represent situations that need to be avoided.

3.4. Questions

1. Creating a secure development lifecycle involves:

A. Adding security features to the software
B. Including threat modeling
C. Training coders to find and remove security errors
D. Modifying the development process, not the software product

2. A software product that has security but lacks quality can result in:

A. Exploitable vulnerabilities
B. Undocumented features that result in undesired behaviors
C. Poor maintainability
D. Missing security elements

3. Which of the following is not an attribute of an SDL process?

A. Fuzz testing

B. Bug bars

C. Authentication

D. Developer security awareness

4. Periodic reviews to ensure that security issues are addressed as part of the development process are called:

A. Security gates

B. Security checklist

C. Threat model

D. Attack surface area analysis

5. The term DREAD stands for:

A. Damage potential, Recoverability, Exploitability, Asset affected, and Discoverability

B. Damage potential, Reproducibility, Exploitability, Affected user base, and Discoverability

C. Damage potential, Reproducibility, External vulnerability, asset Affected, and Discoverability

D. Design issue, Reproducibility, Exploitability, Asset affected, and Discoverability

6. The term STRIDE stands for:

A. Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege

B. Spoofing, Tampering, Reproducibility, Information disclosure, Denial of service, and Elevation of privilege

C. Spoofing, Tampering, Reproducibility, Information disclosure, Discoverability, and Elevation of privilege

D. Spoofing, Tampering, Repudiation, Information disclosure, Discoverability, and Elevation of privilege

7. Which of the following describes the purpose of threat modeling?

A. Enumerate threats to the software

B. Define the correct and secure data flows in a program

C. Communicate testing requirements to the test team

D. Communicate threat and mitigation information across the development team

8. A tool to examine the vulnerability of input interfaces is:

A. Threat model

B. Bug bar

C. Attack surface analysis

D. Fuzz testing framework

9. A linear model for software development is the:

A. Scrum model

B. Spiral model

C. Waterfall model

D. Agile model

10. User stories convey high-level user requirements in the:

A. XP model

B. Prototyping model

C. Spiral model

D. Waterfall model

11. Bug bars are used to:

A. Track bugs

B. Score bugs

C. Manage bugs

D. Attribute bugs to developers

Answers

1. D. The creation of a secure development lifecycle process involves multiple changes to the development process itself, not the software.

2. B. Poor quality can result in undocumented features that result in exploitable output conditions.

3. C. Authentication is a security feature, not an attribute of secure development.

4. A. Security gates are the points in the development cycle where proper security processes are checked for completion.

5. B. The term DREAD refers to a manner of classifying bugs: Damage potential, Reproducibility, Exploitability, Affected user base, and Discoverability.

6. A. The term STRIDE refers to sources of threats: Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege.

7. D. Threat modeling is a tool used to communicate information about threats and the mitigation procedures to all members of the development team.

8. D. Fuzz testing is the application of a series of random inputs to an input interface to test for exploitable failures.

9. C. The waterfall model for software development consists of linear steps progressing in just one way.

10. A. User stories are associated with agile methods, typically XP.

11. B. A bug bar is a measurement level that, when exceeded, indicates that the bug must be fixed prior to delivering the product to customers.

CHAPTER 4. SECURE SOFTWARE REQUIREMENTS

Requirements are the blueprint by which software is designed, built, and tested. As one of the important foundational elements, it is important to manage this portion of the software development lifecycle (SDLC) process properly. Requirements set the expectations for what is being built and how it is expected to operate. Developing and understanding the requirements early in the SDLC process are important, for if one has to go back and add new requirements later in the process, it can cause significant issues, including rework.

4.1. Functional Requirements

Functional requirements describe how the software is expected to function. They begin as business requirements and can come from several different places. The line of business that is going to use the software has some business functionality it wishes to achieve with the new software. These business requirements are translated into functional requirements. The IT operations group may have standard requirements, such as deployment platform requirements, database requirements, Disaster Recovery/Business Continuity Planning (DR/BCP) requirements, infrastructure requirements, and more. The organization may have its own coding requirements in terms of good programming and maintainability standards. Security may have its own set of requirements. In the end, all of these business requirements must be translated into functional requirements that can be followed by designers, coders, testers, and more to ensure they are met as part of the SDLC process.

4.1.1. Role and User Definitions

Role and user definitions are the statements of who will be using what functionality of the software. At a high level, these will be in generic form, such as which groups of users are allowed to use the system. Subsequent refinements will detail specifics, such as which users are allowed which functionality as part of their job. The detailed listing of what users are involved in a system form part of the use-case definition. In computer science terms, users are referred to as subjects. This term is important to understand the subject-object-activity matrix presented later in this section.

4.1.2. Objects

Objects are items that users (subjects) interact with in the operation of a system. An object can be a file, a database record, a system, or program element. Anything that can be accessed is an object. One method of controlling access is through the use of access control lists assigned to objects. As with subjects, objects form an important part of the subject-object-activity matrix. Specifically defining the objects and their function in a system is an important part of the SDLC. This ensures all members of the development team can properly use a common set of objects and control the interactions appropriately.

4.1.3. Activities/Actions

Activities or actions are the permitted events that a subject can perform on an associated object. The specific set of activities is defined by the object. A database record can be created, read, updated, or deleted. A file can be accessed, modified, deleted, etc. For each object in the system, all possible activities/actions should be defined and documented. Undocumented functionality has been the downfall of many a system when a user found an activity that was not considered during design and construction, but still occurred, allowing functionality outside of the design parameters.

4.1.4. Subject-Object-Activity Matrix

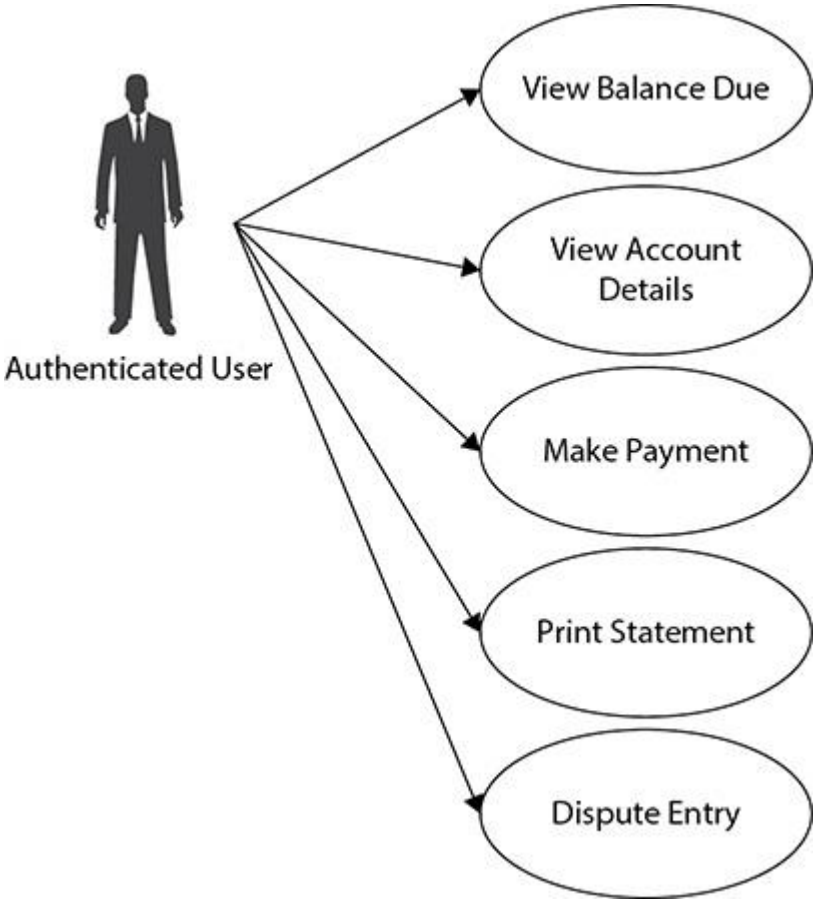
Subjects represent who, objects represent what, and activities or actions represent the how of the subject-object-activity relationship. Understanding the activities that are permitted or denied in each subject-object combination is an important requirements exercise. To assist designers and developers in correctly defining these relationships, a matrix referred to as the subject-object-activity matrix is employed. For each subject, all of the objects are listed, along with the activities for each object. For each combination, the security requirement of the state is then defined. This results in a master list of allowable actions and another master list of denied actions. These lists are useful in creating appropriate use and misuse cases, respectively. The subject-object-activity matrix is a tool that permits concise communication about allowed system interactions.

4.1.5. Use Cases

Use cases are a powerful technique for determining functional requirements in developer-friendly terms. A use case is a specific example of an intended

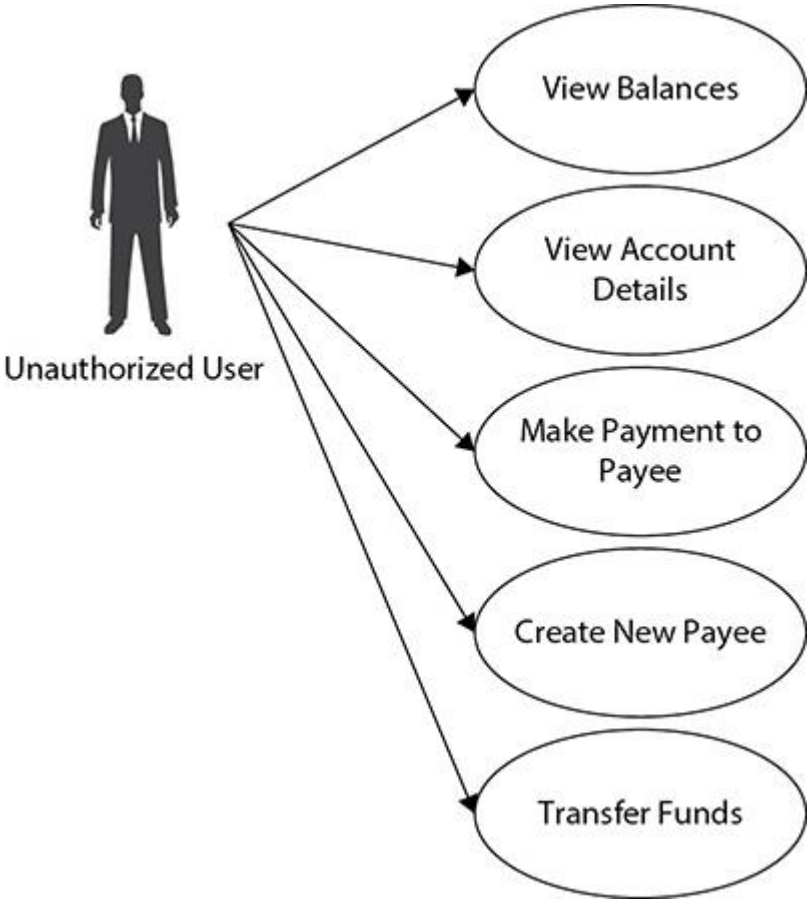
behavior of the system. Defining use cases allows a mechanism by which the intended behavior (functional requirement) of a system can be defined for both developers and testers. Use cases are not intended for all subject-object interactions, as the documentation requirement would exceed the utility. Use cases are not a substitute for documenting the specific requirements. Where use cases are helpful is in the description of complex or confusing or ambiguous situations associated with user interactions with the system. This facilitates the correct design of both the software and the test apparatus to cover what would otherwise be incomplete due to poorly articulated requirements.

Use-case modeling shows the intended system behavior (activity) for actors (users). This combination is referred to as a use case, and is typically presented in a graphical format. Users are depicted as stick figures, and the intended system functions as ellipses. Use-case modeling requires the identification of the appropriate actors, whether person, role, or process (nonhuman system), as well as the desired system functions. The graphical nature enables the construction of complex business processes in a simple-to-understand form. When sequences of actions are important, another diagram can be added to explain this.



4.1.6. Abuse Cases

Misuse or abuse cases can be considered a form of use case illustrating specifically prohibited actions. Although one could consider the situation that anything not specifically allowed should be denied, making this redundant, misuse cases still serve a valuable role in communicating requirements to developers and testers.



Misuse cases can present commonly known attack scenarios, and are designed to facilitate communication among designers, developers, and testers to ensure that potential security holes are managed in a proactive manner. Misuse cases can examine a system from an attacker's point of view, whether the attacker is an inside threat or an outside one. Properly constructed misuse cases can trigger specific test scenarios to ensure known weaknesses have been recognized and dealt with appropriately before deployment.

4.1.7. Secure Coding Standards

Secure coding standards are language-specific rules and recommended practices that provide for secure programming. It is one thing to describe sources of vulnerabilities and errors in programs; it is another matter to prescribe forms

that, when implemented, will preclude the specific sets of vulnerabilities and exploitable conditions found in typical code.

Application programming can be considered a form of manufacturing. Requirements are turned into value-added product at the end of a series of business processes. Controlling these processes and making them repeatable is one of the objectives of a secure development lifecycle. One of the tools an organization can use to achieve this objective is the adoption of an enterprise-specific set of secure coding standards.

Organizations should adopt the use of a secure application development framework as part of their secure development lifecycle process. Because secure coding guidelines have been published for most common languages, adoption of these practices is an important part of secure coding standards in an enterprise. Adapting and adopting industry best practices are also important elements in the secure development lifecycle.

One common problem in many programs results from poor error trapping and handling. This is a problem that can benefit from an enterprise rule where all exceptions and errors are trapped by the generating function and then handled in such a manner so as not to divulge internal information to external users.

4.2. Operational Requirements

Software is deployed in an enterprise environment where it is rarely completely on its own. Enterprises will have standards as to deployment platforms, Linux, Microsoft Windows, specific types and versions of database servers, web servers, and other infrastructure components.

Software in the enterprise rarely works all by itself without connections to other pieces of software. A new system may provide new functionality, but would do so touching existing systems, such as connections to users, parts databases, customer records, etc. One set of operational requirements is built around the idea that a new or expanded system must interact with the existing systems over existing channels and protocols. At a high level, this can be easily defined, but it is not until detailed specifications are published that much utility is derived from the effort.

One of the elements of secure software development is that it is secure in deployment. Ensuring that systems are secure by design is commonly seen as the

focus of an SDLC, but it is also important to ensure systems are secure when deployed. This includes elements such as secure by default and secure when deployed. Ensuring the default configuration maintains the security of an application if the system defaults are chosen, and since this is a common configuration and should be a functioning configuration, it should be secure.

Software will be deployed in the environment as best suits its maintainability, data access, and access to needed services. Ultimately, at the finest level of detail, the functional requirements that relate to system deployment will be detailed for use. An example is the use of a database and web server. Corporate standards, dictated by personnel and infrastructure services, will drive many of the selections. Although there are many different database servers and web servers in the marketplace, most enterprises have already selected an enterprise standard, sometimes by type of data or usage. Understanding and conforming to all the requisite infrastructure requirements are necessary to allow seamless interconnectivity between different systems.

CHAPTER 5. SECURE SOFTWARE DESIGN

Designing an application is the beginning of implementing security into the final application. Using the information uncovered in the requirements phase, designers create the blueprint developers use to arrive at the final product. It is during this phase that the foundational elements to build the proper security functionality into the application are initiated. To determine which security elements are needed in the application, designers can use the information from the attack surface analysis and the threat model to determine the “what” and “where” elements. Knowledge of secure design principles can provide the “how” elements. Using this information in a comprehensive plan can provide developers with a targeted foundation that will greatly assist in creating a secure application.

5.1. Secure Design Principles

5.1.1. Good Enough Security

When designing in security aspects of an application, care should be exercised to ensure that the security elements are in response to the actual risk associated with the potential vulnerability. Designing excess security elements is a waste of resources. Underprotecting the application increases the risk. The challenge is in determining the correct level of security functionality. Elements of the threat model and attack surface analysis can provide guidance as to the level of risk. Documenting the level and purpose of security functions will assist the development team in creating an appropriate and desired level of security.

5.1.2. Least Privilege

One of the most fundamental approaches to security is least privilege. Least privilege should be utilized whenever possible as a preventative measure. Embracing designs with least privilege creates a natural set of defenses should unexpected errors happen. Least privilege, by definition, is sufficient privilege, and this should be a design standard. Excess privilege represents potentially excess risk when vulnerabilities are exploited.

5.1.3. Separation of Duties

Separation of duties must be designed into a system. Software components can be designed to enforce separation of duties when they require multiple conditions to be met before a task is considered complete. These multiple

conditions can then be managed separately to enforce the checks and balances required by the system. In designing the system, designers also impact the method of operation of the system.

As with all other design choices, the details are recorded as part of the threat model. This acts as the communication method between all members of the development effort. Designing in operational elements such as separation of duties still requires additional work to happen through the development process, and the threat model can communicate the expectations of later development activities in this regard.

5.1.4. Defense in Depth

Defense in depth is one of the oldest security principles. If one defense is good, multiple overlapping defenses are better. The threat model document will contain information where overlapping defenses should be implemented. Designing in layers as part of the security architecture can work to mitigate a wider range of threats in a more efficient manner.

Because every piece of software can be compromised or bypassed in some way, it is incumbent on the design team to recognize this and create defenses that can mitigate specific threats. Although all software, including the mitigating defenses, can fail, the end result is to raise the difficulty level and limit the risk associated with individual failures.

Designing a series of layered defense elements across an application provides for an efficient defense. For layers to be effective, they should be dissimilar in nature so that if an adversary makes it past one layer, a separate layer may still be effective in maintaining the system in a secure state. An example is the coupling of encryption and access control methods to provide multiple layers that are diverse in their protection nature and yet both can provide confidentiality.

5.1.5. Fail Safe

As mentioned in the exception management section, all systems will experience failures. The fail-safe design principle refers to the fact that when a system experiences a failure, it should fail to a safe state. When designing elements of an application, one should consider what happens when a particular element fails. When a system enters a failure state, the attributes associated with security, confidentiality, integrity, and availability need to be appropriately maintained.

Failure is something that every system will experience. One of the design elements that should be considered is how the individual failures affect overall operations. Ensuring that the design includes elements to degrade gracefully and return to normal operation through the shortest path assists in maintaining the resilience of the system. For example, if a system cannot complete a connection to a database, when the attempt times out, how does the system react? Does it automatically retry, and if so, is there a mechanism to prevent a lockup when the failure continually repeats?

5.1.6. Economy of Mechanism

The terms security and complexity are often at odds with each other. This is because the more complex something is, the harder it is to understand, and you cannot truly secure something if you do not understand it. During the design phase of the project, it is important to emphasize simplicity. Smaller and simpler is easier to secure. Designs that are easy to understand, easy to implement, and well documented will lead to more secure applications.

If an application is going to do a specific type of function—for example, gather standard information from a database—and this function repeats throughout the system, then designing a standard method and reusing it improves system stability. As systems tend to grow and evolve over time, it is important to design in extensibility. Applications should be designed to be simple to troubleshoot, simple to expand, simple to use, and simple to administer.

5.1.7. Complete Mediation

Systems should be designed so that if the authorization system is ever circumvented, the damage is limited to immediate requests. Whenever sensitive operations are to be performed, it is important to perform authorization checks. Assuming that permissions are appropriate is just that—an assumption—and failures can occur. For instance, if a routine is to permit managers to change a key value in a database, then assuming that the party calling the routine is a manager can result in failures. Verifying that the party has the requisite authorization as part of the function is an example of complete mediation. Designing this level of security into a system should be a standard design practice for all applications.

5.1.8. Open Design

Part of the software development process includes multiple parties doing different tasks that in the end create a functioning piece of software. Over time, the software will be updated and improved, which is another round of the development process. For all of this work to be properly coordinated, open communication needs to occur. Having designs that are open and understandable will prevent activities later in the development process that will weaken or bypass security efforts.

5.1.9. Least Common Mechanism

The concept of least common mechanism is constructed to prevent inadvertent security failures. Designing a system where multiple processes share a common mechanism can lead to a potential information pathway between users or processes. The concepts of least common mechanism and leverage existing components can place a designer at a conflicting crossroad. One concept advocates reuse and the other separation. The choice is a case of determining the correct balance associated with the risk from each.

Take a system where users can access or modify database records based on their user credentials. Having a single interface that handles all requests can lead to inadvertent weaknesses. If reading is considered one level of security and modification of records a more privileged activity, then combining them into a single routine exposes the high-privilege action to a potential low-privilege account. Thus, separating mechanisms based on security levels can be an important design tool.

5.1.10. Psychological Acceptability

Users are a key part of a system and its security. To include a user in the security of a system requires that the security aspects be designed so that they are psychologically acceptable to the user. When a user is presented with a security system that appears to obstruct the user, the result will be the user working around these security aspects. Applications communicate with users all the time. Care and effort are given to ensuring that an application is useable in normal operation. This same idea of usability needs to be extended to security functions. Designers should understand how the application will be used in the enterprise and what users will expect of it.

When users are presented with cryptic messages, such as the ubiquitous "Contact your system administrator" error message, expecting them to do anything that will help resolve an issue is unrealistic. Just as care and consideration are taken to ensure normal operations are comprehensible to the user, so, too, should a similar effort be taken for abnormal circumstances.

5.1.11. Weakest Link

Every system, by definition, has a "weakest" link. Adversaries do not seek out the strongest defense to attempt a breach; they seek out any weakness they can exploit. Overall, a system can only be considered as strong as its weakest link. When designing an application, it is important to consider both the local and system views with respect to weaknesses. Designing a system using the security tenets described in this section will go a long way in preventing local failures from becoming system failures. This limits the effect of the weakest link to local effects.

5.1.12. Leverage Existing Components

Modern software development includes extensive reuse of components. From component libraries to common functions across multiple components, there is significant opportunity to reduce development costs through reuse. This can also simplify a system through the reuse of known elements. The downside of massive reuse is associated with a monoculture environment, which is where a failure has a larger footprint because of all the places where it is involved.

During the design phase, decisions should be made as to the appropriate level of reuse. For some complex functions, such as in cryptography, reuse is the preferred path. In other cases, where the lineage of a component cannot be established, then the risk of use may outweigh the benefit. In addition, the inclusion of previous code, sometimes referred to as legacy code, can reduce development efforts and risk.

5.1.13. Single Point of Failure

The design of a software system should be such that all points of failure are analyzed and that a single failure does not result in system failure. Examining designs and implementations for single points of failure is important to prevent this form of catastrophic failure from being released in a product or system. Single points of failure can exist for any attribute, confidentiality, integrity, availability, etc., and may well be different for each attribute. During the design phase, failure

scenarios should be examined with an eye for single points of failure that could cascade into entire system failure.

5.2. Technologies

Technologies are one of the driving forces behind software. New technology is rare without a software element. And software uses specific technology to achieve its security objectives. This chapter will examine some of the basic technologies used to enable security functionality in software.

5.2.1. Authentication and Identity Management

Authentication is an identity verification process that attempts to determine whether users are who they say they are. Identity management is the comprehensive set of services related to managing the use of identities as part of an access control solution. Strictly speaking, the identity process is one where a user establishes their identity. Authentication is the act of verifying the supplied credentials against the set established during the identity process. The term identity management (IDM) refers to the set of policies, processes, and technologies for managing digital identity information. Identity and access management (IAM) is another term associated with the comprehensive set of policies, processes, and technologies for managing digital identity information.

a) Identity Management

Identity management is a set of processes associated with the identity lifecycle, including the provisioning, management, and deprovisioning of identities. The provisioning step involves the creation of a digital identity from an actual identity. The source of the actual identity can be a person, a process, an entity, or virtually anything. The identity process binds some form of secret to the digital identity so that at future times, the identity can be verified. The secret that is used to verify identity is an item deserving specific attention as part of the development process. Protecting the secret, yet making it usable, are foundational elements associated with the activity.In a scalable system, management of identities and the associated activities needs to be automated. A large number of identity functions need to be handled in a comprehensive fashion. Changes to identities, the addition and removal of roles, changes to rights and privileges

associated with roles or identities—all of these items need to be done securely and logged appropriately. The complexity of the requirements makes the use of existing enterprise systems an attractive option when appropriate.

Identity management can be accomplished through third-party programs that enhance the operating system offerings in this area. The standard operating system implementation leaves much to be desired in management of user-selected provisioning or changes to identity metadata. Many enterprises have third-party enterprise-class IDM systems that provide services such as password resets, password synchronization, single sign-on, and multiple identity methods.

Having automated password resets can free up significant help desk time and provide faster service to users who have forgotten their password. Automated password reset systems require a reasonable set of challenges to verify that the person requesting the reset is authorized to do so. Then, the reset must occur in a way that does not expose the old password. E-mail resets via a uniform resource locator (URL) are one common method employed for the reset operation. Users can have multiple passwords on different systems, complicating the user activity. Password synchronization systems allow a user to synchronize a set of passwords across connected, but different, identity systems, making it easier for users to access the systems. Single sign-on is an industrial-strength version of synchronization. Users enter their credentials into one system, and it connects to the other systems, authenticating based on the entered credentials.

b) Authentication

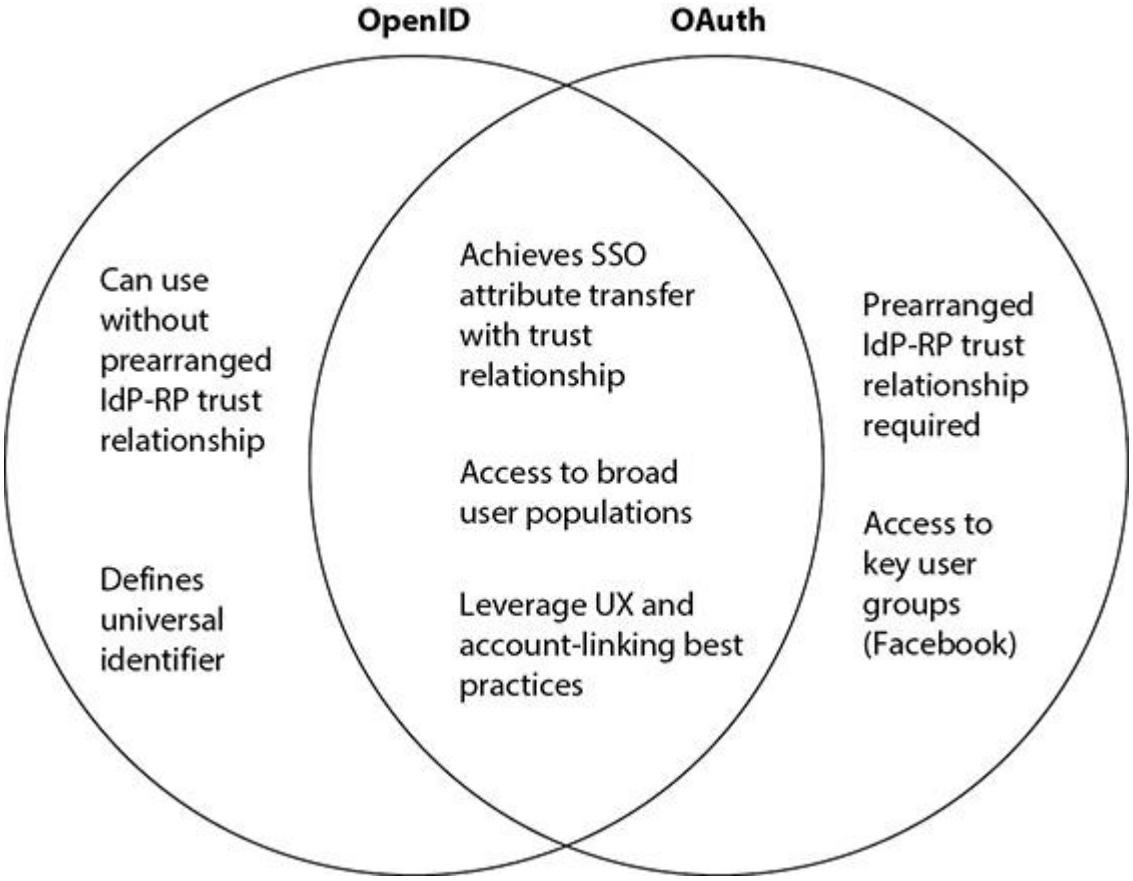
Authentication is the process of verifying that a user is who they claim to be and applying the correct values in the access control system. The level of required integration is high, from the storage systems that store the credentials and the access control information to the transparent handling of the information establishing or denying the validity of a credential match. When referring to authentication, one is referring to the process of verification of an identity. When one refers to the authentication system, one is typically referring to the underlying operating system aspect, not the third-party application that sits on top.

Authentication systems come in a variety of sizes and types. Several different elements can be used as secrets as part of the authentication process. Passwords, tokens, biometrics, smart cards—the list can be long. The types can be

categorized as something you know, something you have, or something you are. The application of one or more of these factors simultaneously for identity verification is a standard process in virtually all computing systems.

The underlying mechanism has some best-practice safeguards that should be included in a system. Mechanisms such as an escalating time lock-out after a given number of successive failures, logging of all attempts (both successful and failed), and integration with the authorization system once authentication is successful are common protections. Password/token reset, account recovery, periodic changes, and password strength issues are just some of the myriad of functionalities that need to be encapsulated in an authentication system.

Two of the more prevalent systems are OAuth and OpenID. OpenID was created for federated authentication, specifically to allow a third party to authenticate your users for you by using accounts that users already have. The OpenID protocol enables websites or applications (consumers) to grant access to their own applications by using another service or application (provider) for authentication. This can be done without requiring users to maintain a separate account/profile with the consumers.OAuth was created to eliminate the need for users to share their passwords with third-party applications. The OAuth protocol enables websites or applications (consumers) to access protected resources from a web service (service provider) via an application programming interface (API), without requiring users to disclose their service provider credentials to the consumers.



Both OpenID (for authentication) and OAuth (for authorization) accomplish many of the same things. Each protocol provides a different set of features, which are required by their primary objective, but essentially, they are interchangeable. At their core, both protocols have an assertion verification method. They differ in that OpenID is limited to the "this is who I am" assertion, while OAuth provides an "access token" that can be exchanged for any supported assertion via an API.

5.2.2. Credential Management

There are numerous methods of authentication, and each has its own set of credentials that require management. The identifying information that is provided by a user as part of their claim to be an authorized user is sensitive data and requires significant protection. The identifying information is frequently referred to as credentials. These credentials can be in the form of a passed secret, typically a password. Other common forms include digital strings that are held by hardware tokens or devices, biometrics, and certificates. Each of these forms has advantages and disadvantages.

Each set of credentials, regardless of the source, requires safekeeping on the part of the receiving entity. Managing of these credentials includes tasks such as credential generation, storage, synchronization, reset, and revocation. Because of

the sensitive nature of manipulating credentials, all of these activities should be logged.

a) X.509 Credentials

X.509 refers to a series of standards associated with the manipulation of certificates used to transfer asymmetric keys between parties in a verifiable manner. A digital certificate binds an individual's identity to a public key, and it contains all the information a receiver needs to be assured of the identity of the public key owner. After a registration authority (RA) verifies an individual's identity, the certificate authority (CA) generates the digital certificate. The digital certificate can contain the information necessary to facilitate authentication.

The following fields are included within an X.509 digital certificate:

- Version number: Identifies the version of the X.509 standard that was followed to create the certificate; indicates the format and fields that can be used.
- Serial number: Provides a unique number identifying this one specific certificate issued by a particular CA.
- Signature algorithm: Specifies the hashing and digital signature algorithms used to digitally sign the certificate.
- Issuer: Identifies the CA that generated and digitally signed the certificate.
- Validity: Specifies the dates through which the certificate is valid for use.
- Subject: Specifies the owner of the certificate.
- Public key: Identifies the public key being bound to the certified subject; also identifies the algorithm used to create the private/public key pair.
- Certificate usage: Specifies the approved use of the certificate, which dictates intended use of this public key.
- Extensions: Allow additional data to be encoded into the certificate to expand its functionality. Companies can customize the use of certificates within their environments by using these extensions. X.509 version 3 has expanded the extension possibilities.

Certificates are created and formatted based on the X.509 standard, which outlines the necessary fields of a certificate and the possible values that can be inserted into the fields. As of this writing, X.509 version 3 is the most current version of the standard. X.509 is a standard of the International Telecommunication Union (www.itu.int). The IETF's Public-Key Infrastructure (X.509), or PKIX, working group has adapted the X.509 standard to the more flexible organization of the Internet, as specified in RFC 3280, and is commonly referred to as PKIX for Public Key Infrastructure X.509.

The public key infrastructure (PKI) associated with certificates enables the passing and verification of these digital elements between firms. Because certificates are cryptographically signed, elements within them are protected from unauthorized alteration and can have their source verified. Building out a complete PKI infrastructure is a complex endeavor, requiring many different levels of protection to ensure that only authorized entities are permitted to make changes to the certification.

Setting up a functioning and secure PKI solution involves many parts, including certificate authorities, registration authorities, and certificate revocation mechanisms, either Certificate Revocation Lists (CRLs) or Online Certificate Status Protocol (OCSP).

X.509 certificates provide a wide range of benefits to any application that needs to work with public key cryptography. Certificates provide a standard means of passing keys, a standard that is accepted by virtually every provider and consumer of public keys. This makes X.509 a widely used and proven technology.

b) Single Sign-On

Single sign-on (SSO) makes it possible for a user, after authentication, to have his credentials reused on other applications without the user re-entering the secret. To achieve this, it is necessary to store the credentials outside of the application and then reuse the credentials against another system. There are a number of ways that this can be accomplished, but two of the most popular and accepted methods for sharing authentication information are Kerberos and Security Assertion Markup Language (SAML). The OpenID protocol has proven to be a well-vetted and secure protocol for SSO. However, as with all technologies,

security vulnerabilities can still occur due to misuse or misunderstanding of the technology.

The key concept with respect to SSO is federation. In a federated authentication system, users can log in to one site and access another or affiliated site without re-entering credentials. The primary objective of federation is user convenience. Authentication is all about trust, and federated trust is difficult to establish. SSO can be challenging to implement and because of trust issues, it is not an authentication panacea. As in all risk-based transactions, a balance must be achieved between the objectives and the risks. SSO-based systems can create single-point-of-failure scenarios, so for certain high-risk implementations, their use is not recommended.

5.2.3. Flow Control

In information processing systems, information flows between nodes, between processes, and between applications. The movement of information across a system or series of systems has security consequences. Sensitive information must be protected, with access provided to authorized parties and protected from unauthorized ones. The movement of information must be channeled correctly and protected along the way. There are technologies, firewalls, proxies, and queues that can be utilized to facilitate proper information transfer.

a) Firewalls

Firewalls act as policy enforcement devices, determining whether to pass or block communications based on a variety of factors. Network-level firewalls operate using the information associated with networking to determine who can communicate with whom. Next-generation firewalls provide significantly greater granularity in communication decisions. Firewalls operate on a packet level, and can be either stateless or stateful. Basic network firewalls operate on a packet-by-packet basis and use addressing information to make decisions. In doing so, they are stateless, not carrying information from packet to packet as part of the decision process. Advanced firewalls can analyze multiple packets and utilize information from the protocols being carried to make more granular decisions. Did the packet come in response to a request from inside the network? Is the packet carrying information across web channels, port 80, using authorized or unauthorized

applications? This level of stateful packet inspection, although difficult to scale, can be useful in providing significant levels of communication protection.

Firewalls are basically devices that, at the end of the day, are supposed to allow the desired communications and block undesired communications. Malicious attempts to manipulate a system via communication channels can be detected and blocked using a firewall. Firewalls can work with intrusion detection systems, acting as the enforcer in response to another system's inputs. One of the limitations of firewalls is governed by network architecture. When numerous paths exist for traffic to flow between points, determining where to place devices such as firewalls becomes increasingly difficult and at times nearly impossible. Again, as with all things security, balance becomes a guiding principle.

b) Proxies

Proxies are similar to firewalls in that they can mediate traffic flows. They differ in that they act as middlemen, somewhat like a post office box. Traffic from untrusted sources is terminated at a proxy, where the traffic is received and to some degree processed. If the traffic meets the correct rules, it can then be forwarded on to the intended system. Proxies come in a wide range of capabilities, from simple to very complex, both in their rule-processing capabilities and additional functionalities. One of these functionalities is caching—a temporary local storage of web information that is frequently used and seldom changed, like images. In this role, a proxy acts as a security device and a performance-enhancing device.

c) Application Firewalls

Application firewalls are becoming more popular, acting as application-specific gateways between users, and potential users, and web-based applications. Acting as a firewall proxy, web application firewalls can monitor traffic in both directions, client to server and server to client, watching for anomalies. Web application firewalls act as guards against both malicious intruders and misbehaving applications. Should an outsider attempt to perform actions that are not authorized to an application, the web application firewall can block the requests from getting to the application. Should the application experience some failure, resulting in, say, large-scale data transfers when only small data transfers are the norm, again, the web application firewall can block the data from leaving the enterprise.

d) Queuing Technology

Message transport from sender to receiver can be done either synchronously or asynchronously, and either have guaranteed transport or best effort. Internet protocols can manage the guarantee/best effort part, but a separate mechanism is needed if asynchronous travel is permissible. Asynchronous transport can alleviate network congestion during periods where traffic flows are high and can assist in the prevention of losing traffic due to bandwidth restrictions. Queuing technologies in the form of message queues can provide a guaranteed mechanism of asynchronous transport, solving many short-term network congestion issues. There are numerous vendors in the message queue space, including Microsoft, Oracle, and IBM.

5.2.4. Logging

An important element in any security system is the presence of security logs. Logs enable personnel to examine information from a wide variety of sources after the fact, providing information about what actions transpired, with which accounts, on which servers, and with what specific outcomes. Many compliance programs require some form of logging and log management. The challenges in designing log programs are what to log and where to store it.

What needs to be logged is a function of several criteria. First, numerous compliance programs—HIPAA, SOX, PCI DSS, EOC, and others—have logging requirements, and these need to be met. The next criterion is one associated with incident response. What information would investigators want or need to know to research failures and issues? This is a question for the development team—what is available that can be logged that would provide useful information for investigators, either to the cause of the issue or impact?

The "where to log it" question also has several options, each with advantages and disadvantages. Local logging can be simple and quick for the development team. But it has the disadvantage of being yet another log to secure and integrate into the enterprise log management system. Logs by themselves are not terribly useful. What makes individual logs useful is the combination of events across other logs, detailing the activities of a particular user at a given point in time. This requires a coordination function, one that is supported by many third-party software vendors through their security information and event management

(SIEM) tool offerings. These tools provide a rich analytical environment to sift through and find correlations in large datasets of security information.

a) To Log or Not to Log

One of the challenging questions is to what extent items should be logged. If all goes right and things never fail, there is still a need for some logging, if for no other reason than to administratively track what has occurred on the system. Logs provide a historical record of transactions through a system.Logs can also provide information that can be critical in debugging problems encountered by the software. Logs generated as part of error processing routines can provide great insight not only into individual program issues but also into overall system state at the time of the error.When deciding what to log, there are two central questions. First, when would we need the information, and what information would be needed in that use case? This defines the core foundation of a log entry. As each log entry takes time and space, both to record and later to sift through when using, an economic decision on "does this deserve logging?" must occur.The second foundational element to consider when logging is the security of the information being logged. Details, such as system state, user IDs, critical variable values, including things such as passwords–all of these are subject to misuse or abuse in the wrong hands, and thus must be protected. Logs require security. The level of security is a key element to determine in the creation of logging requirements.

b) Syslog

Syslog is an Internet Engineering Task Force (IETF)–approved protocol for log messaging. It was designed and built around UNIX and provides a UNIX-centric format for sending log information across an IP network. Although in its native form, it uses User Datagram Protocol (UDP) and transmits information in the clear, wrappers are available that provide Transport Layer Security (TLS)–based security and TCP-based communication guarantees. While syslog is the de facto standard for logging management in Linux and UNIX environments, there is no equivalent in the Microsoft sphere of influence. Microsoft systems log locally, and there are some Microsoft solutions for aggregating logs to a central server, but these solutions are not as mature as syslog. Part of the reason for this is the myriad of third-party logging and log management solutions that provide superior business-level analytical packages that are focused on log data.

5.2.5. Database Security

Databases are technologies used to store and manipulate data. Relational databases store data in tables and have a wide array of tools that can be used to access, manipulate, and store data. Databases have a variety of security mechanisms to assist in creating the appropriate level of security. This includes elements for confidentiality, integrity, and availability. The details of designing a database environment for security are beyond the scope of the CSSLP practitioner, but it is still important to understand the capabilities.

Encryption can be employed to provide a level of confidentiality protection for the data being stored in a database. Data structures, such as views, can be created, giving different parties different levels of access to the data stored in the database. Programmatic structures called stored procedures can be created to limit access to only specific elements based on predefined rules. Backup and replication strategies can be employed to provide near-perfect availability and redundancy for critical systems. Taken together, the protections afforded the data in a modern database can be comprehensive and valuable. The key is in defining the types and levels of protection required based on risk.

a) Encryption

Data stored in a database is a lucrative target for attackers. Just like the vault in a bank, it is where the valuable material is stored, so gaining the correct level of access, for instance, administrative rights, can be an attacker's dream and a defender's nightmare. Encrypting data at rest is a preventative control mechanism that can be employed virtually anywhere the data is at rest, including databases. The encryption can be managed via native database management system functions, or it can be done using cryptographic resources external to the database.

b) Triggers

Triggers are specific database activities that are automatically executed in response to specific database events. Triggers are a useful tool, as they can automate a lot of interesting items. Changes to a record can trigger a script; adding a record can trigger a script; define any database task and assign a script—this allows a lot of flexibility. Need to log something and include business logic? Triggers can provide the flexibility to automate anything in a database.

c) Views

Views are programmatically designed extracts of data in a series of tables. Tables can contain all the data, and a view can provide a subset of the information based on some set of business rules. A table could contain a record that provides all the details about a customer: addresses, names, credit card information, etc. Some of this information should be protected—PII and credit card information, for instance. A view can provide a shipping routine only the ship-to columns and not the protected information, and in this way, when using the view, it is not possible to disclose what isn't there.

d) Privilege Management

Databases have their own internal access control mechanism, which are similar to ACL-based controls to file systems. Designing the security system for data records, users, and roles requires the same types of processes as designing file system access control mechanisms. The two access control mechanisms can be interconnected, with the database system responding to the enterprise authorization systems, typically through roles defined in the database system.

5.2.6. Programming Language Environment

Software developers use a programming language to encode the specific set of operations in what is referred to as source code. The programming language used for development is seldom the language used in the actual instantiation of the code on the target computer. The source code is converted to the operational code through compilers, interpreters, or a combination of both. The choice of the development language is typically based on a number of criteria, the specific requirements of the application, the skills of the development team, and a host of other issues.

Compilers offer one set of advantages, and interpreters others. Systems built in a hybrid mode use elements of both. Compiled languages involve two subprocesses: compiling and linking. The compiling process converts the source code into a set of processor-specific codes. Linking involves the connecting of various program elements, including libraries, dependency files, and resources. Linking comes in two forms: static and dynamic. Static linking copies all the requirements into the final executable, offering faster execution and ease of distribution. Static linking can lead to bloated file sizes.

Dynamic linking involves placing the names and relative locations of dependencies in the code, with these being resolved at runtime when all elements are loaded into memory. Dynamic linking can create a smaller file, but does create risk from hijacked dependent programs.

Interpreters use an intermediary program to result in the execution of the source code on a target machine. Interpreters provide slower execution, but faster change between revisions, as there is no need for recompiling and relinking. The source code is actually converted by the interpreter into an executable form in a line-by-line fashion at runtime.

A hybrid solution takes advantage of both compiled and interpreted languages. The source code is compiled into an intermediate stage that can be interpreted at runtime. The two major hybrid systems are Java and Microsoft .NET. In Java, the intermediate system is known as Java Virtual Machine (JVM), and in the .NET environment, the intermediate system is the common language runtime (CLR).

a) CLR

Microsoft's .NET language system has a wide range of languages in the portfolio. Each of these languages is compiled into what is known as common intermediate language (CIL), also known as Microsoft Intermediate Language (MSIL). One of the advantages of the .NET system is that a given application can be constructed using multiple languages that are compiled into CIL code that is executed using the just-in-time compiler. This compiler, the common language runtime (CLR), executes the CIL on the target machine. The .NET system operates what is known as managed code, an environment that can make certain guarantees about what the code can do. The CLR can insert traps, garbage collection, type safety, index checking, sandboxing, and more. This provides a highly functional and stable execution environment.

b) JVM

In Java environments, the Java language source code is compiled to an intermediate stage known as byte code. This byte code is similar to processor instruction codes, but is not executable directly. The target machine has a Java Virtual Machine (JVM) that executes the byte code. The Java architecture is

referred to as the Java Runtime Environment (JRE), which is composed of the JVM and a set of standard class libraries, the Java Class Library. Together, these elements provide for the managed execution of Java on the target machine.

c) Compiler Switches

Compiler switches enable the development team to control how the compiler handles certain aspects of program construction. A wide range of options are available, manipulating elements such as memory, stack protection, and exception handling. These flags enable the development team to force certain specific behaviors using the compiler. The /GS flag enables a security cookie on the stack to prevent stack-based overflow attacks. The /SAFEH switch enables a safe exception handling table option that can be checked at runtime. The designation of the compiler switch options to be used in a development effort should be one of the elements defined by the security team and published as security requirements for use in the SDL process.

d) Sandboxing

Sandboxing is a term for the execution of computer code in an environment designed to isolate the code from direct contact with the target system. Sandboxes are used to execute untrusted code, code from guests, and unverified programs. They work as a form of virtual machine and can mediate a wide range of system interactions, from memory access to network access, access to other programs, the file system, and devices. The level of protection offered by a sandbox depends upon the level of isolation and mediation offered.

e) Managed vs. Unmanaged Code

Managed code is executed in an intermediate system that can provide a wide range of controls. .NET and Java are examples of managed code, a system with a whole host of protection mechanisms. Sandboxing, garbage collection, index checking, type safe, memory management, and multiplatform capability—these elements provide a lot of benefit to managed code-based systems. Unmanaged code is executed directly on the target operating system. Unmanaged code is always compiled to a specific target system. Unmanaged code can have significant performance advantages. In unmanaged code, memory allocation, type safety, garbage collection, etc., need to be taken care of by the developer. This makes

unmanaged code prone to memory leaks like buffer overruns and pointer overrides and increases the risk.

CHAPTER 6. COMMON SOFTWARE VULNERABILITIES AND COUNTERMEASURES

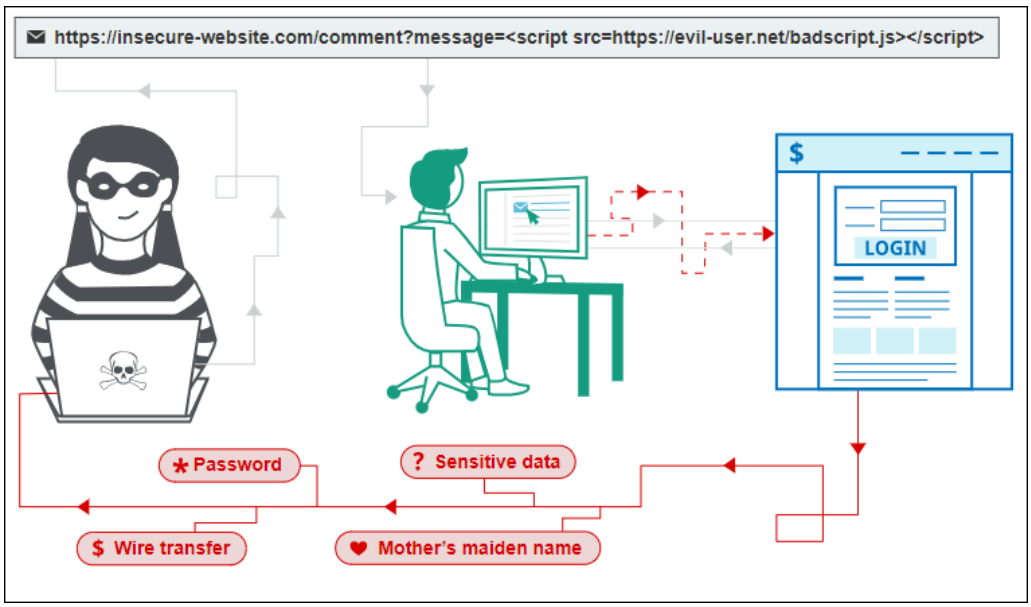
The errors associated with software fall into a series of categories. Understanding the common categories of vulnerabilities and learning how to avoid these known vulnerabilities have been proven to be among the more powerful tools a development team can use in developing more secure code. While attacking the common causes will not remove all vulnerabilities, it will go a long way toward improving the code base. This chapter will examine the most common enumerations associated with vulnerabilities and programming errors.

6.1. Cross-site scripting

6.1.1. What is cross-site scripting (XSS)?

Cross-site scripting (also known as XSS) is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable application. It allows an attacker to circumvent the same origin policy, which is designed to segregate different websites from each other. Cross-site scripting vulnerabilities normally allow an attacker to masquerade as a victim user, to carry out any actions that the user is able to perform, and to access any of the user's data. If the victim user has privileged access within the application, then the attacker might be able to gain full control over all of the application's functionality and data.

Cross-site scripting works by manipulating a vulnerable web site so that it returns malicious JavaScript to users. When the malicious code executes inside a victim's browser, the attacker can fully compromise their interaction with the application.



6.1.2. What are the types of XSS attacks?

There are three main types of XSS attacks. These are:

- Reflected XSS, where the malicious script comes from the current HTTP request.
- Stored XSS, where the malicious script comes from the website's database.
- DOM-based XSS, where the vulnerability exists in client-side code rather than server-side code.

a) Reflected cross-site scripting

Reflected XSS is the simplest variety of cross-site scripting. It arises when an application receives data in an HTTP request and includes that data within the immediate response in an unsafe way.

Here is a simple example of a reflected XSS vulnerability:

```
https://insecure-website.com/status?message=All+is+well.
<p>Status: All is well.</p>
```

The application doesn't perform any other processing of the data, so an attacker can easily construct an attack like this:

```
https://insecure-
website.com/status?message=<script>/*+Bad+stuff+here...+*/</scri
pt>
<p>Status: <script>/* Bad stuff here... */</script></p>
```

If the user visits the URL constructed by the attacker, then the attacker's script executes in the user's browser, in the context of that user's session with the application. At that point, the script can carry out any action, and retrieve any data, to which the user has access.

b) Stored cross-site scripting

Stored XSS (also known as persistent or second-order XSS) arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.

The data in question might be submitted to the application via HTTP requests; for example, comments on a blog post, user nicknames in a chat room, or contact details on a customer order. In other cases, the data might arrive from other untrusted sources; for example, a webmail application displaying messages received over SMTP, a marketing application displaying social media posts, or a network monitoring application displaying packet data from network traffic.

Here is a simple example of a stored XSS vulnerability. A message board application lets users submit messages, which are displayed to other users:

```
<p>Hello, this is my message!</p>
```

The application doesn't perform any other processing of the data, so an attacker can easily send a message that attacks other users:

```
<p><script>/* Bad stuff here... */</script></p>
```

c) DOM-based cross-site scripting

DOM-based XSS (also known as DOM XSS) arises when an application contains some client-side JavaScript that processes data from an untrusted source in an unsafe way, usually by writing the data back to the DOM.

In the following example, an application uses some JavaScript to read the value from an input field and write that value to an element within the HTML:

```
var search = document.getElementById('search').value;
var results = document.getElementById('results');
results.innerHTML = 'You searched for: ' + search;
```

If the attacker can control the value of the input field, they can easily construct a malicious value that causes their own script to execute:

```
You searched for: <img src=1 onerror='/* Bad stuff here... */'>
```

In a typical case, the input field would be populated from part of the HTTP request, such as a URL query string parameter, allowing the attacker to deliver an attack using a malicious URL, in the same manner as reflected XSS.

6.1.3. What can XSS be used for?

An attacker who exploits a cross-site scripting vulnerability is typically able to:

- Impersonate or masquerade as the victim user.
- Carry out any action that the user is able to perform.
- Read any data that the user is able to access.
- Capture the user's login credentials.
- Perform virtual defacement of the web site.
- Inject trojan functionality into the web site.

6.1.4. How to prevent XSS attacks

Preventing cross-site scripting is trivial in some cases but can be much harder depending on the complexity of the application and the ways it handles user-controllable data.

In general, effectively preventing XSS vulnerabilities is likely to involve a combination of the following measures:

- Filter input on arrival. At the point where user input is received, filter as strictly as possible based on what is expected or valid input.
- Encode data on output. At the point where user-controllable data is output in HTTP responses, encode the output to prevent it from being interpreted as active content. Depending on the output context, this might require applying combinations of HTML, URL, JavaScript, and CSS encoding.
- Use appropriate response headers. To prevent XSS in HTTP responses that aren't intended to contain any HTML or JavaScript, you can use the Content-Type and X-Content-Type-Options headers to ensure that browsers interpret the responses in the way you intend.

- Content Security Policy. As a last line of defense, you can use Content Security Policy (CSP) to reduce the severity of any XSS vulnerabilities that still occur.

6.1.5. Common questions about cross-site scripting

How common are XSS vulnerabilities? XSS vulnerabilities are very common, and XSS is probably the most frequently occurring web security vulnerability.

How common are XSS attacks? It is difficult to get reliable data about real-world XSS attacks, but it is probably less frequently exploited than other vulnerabilities.

What is the difference between XSS and CSRF? XSS involves causing a web site to return malicious JavaScript, while CSRF involves inducing a victim user to perform actions they do not intend to do.

What is the difference between XSS and SQL injection? XSS is a client-side vulnerability that targets other application users, while SQL injection is a server-side vulnerability that targets the application's database.

How do I prevent XSS in PHP? Filter your inputs with a whitelist of allowed characters and use type hints or type casting. Escape your outputs with `htmlentities` and `ENT_QUOTES` for HTML contexts, or JavaScript Unicode escapes for JavaScript contexts.

How do I prevent XSS in Java? Filter your inputs with a whitelist of allowed characters and use a library such as Google Guava to HTML-encode your output for HTML contexts, or use JavaScript Unicode escapes for JavaScript contexts.

6.1.6. XSS Labs

Low Reflected XSS Source

```
<?php

header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL
 ) {
    // Feedback for end user
    echo '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
}
```

```
?>
```

Medium Reflected XSS Source

```
<?php

header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL
 ) {
    // Get input
    $name = str_replace( '<script>', '', $_GET[ 'name' ] );

    // Feedback for end user
    echo "<pre>Hello ${name}</pre>";
}

?>
```

High Reflected XSS Source

```
<?php

header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL
 ) {
    // Get input
    $name = preg_replace( '/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i',
'', $_GET[ 'name' ] );

    // Feedback for end user
    echo "<pre>Hello ${name}</pre>";
}

?>
```

Impossible Reflected XSS Source

```
<?php

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL
 ) {
    // Check Anti-CSRF token
    checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_t
oken' ], 'index.php' );
```

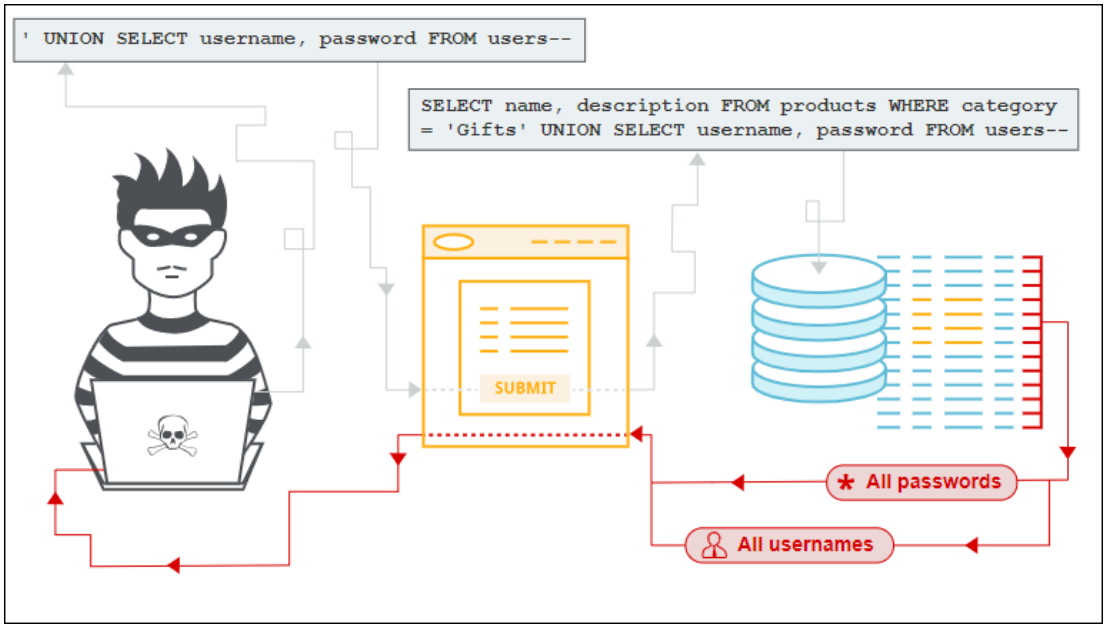
```
    // Get input
    $name = htmlspecialchars( $_GET[ 'name' ] );

    // Feedback for end user
    echo "<pre>Hello ${name}</pre>";
}

// Generate Anti-CSRF token
generateSessionToken();

?>
```

Attack XSS with BEEF

In Wild:

- https://phuclong.com.vn/
- https://fado.vn/

6.2. SQL injection



6.2.1. What is SQL Injection?

SQL injection is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It generally allows an attacker to view data that they are not normally able to retrieve. This might include data belonging to other users, or any other data that the application itself is able to access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behavior.

In some situations, an attacker can escalate an SQL injection attack to compromise the underlying server or other back-end infrastructure, or perform a denial-of-service attack.

6.2.2. SQL injection examples

There are a wide variety of SQL injection vulnerabilities, attacks, and techniques, which arise in different situations. Some common SQL injection examples include:

- Retrieving hidden data, where you can modify an SQL query to return additional results.
- Subverting application logic, where you can change a query to interfere with the application's logic.
- UNION attacks, where you can retrieve data from different database tables.
- Examining the database, where you can extract information about the version and structure of the database.
- Blind SQL injection, where the results of a query you control are not returned in the application's responses.

a) Retrieving hidden data

Consider a shopping application that displays products in different categories. When the user clicks on the Gifts category, their browser requests the URL:

```
https://insecure-website.com/products?category=Gifts
```

This causes the application to make an SQL query to retrieve details of the relevant products from the database:

```
SELECT * FROM products WHERE category = 'Gifts' AND released = 1
```

This SQL query asks the database to return:

- all details (*)
- from the products table
- where the category is Gifts
- and released is 1.

The restriction released = 1 is being used to hide products that are not released. For unreleased products, presumably released = 0.

The application doesn't implement any defenses against SQL injection attacks, so an attacker can construct an attack like:

```
https://insecure-website.com/products?category=Gifts'--
```

This results in the SQL query:

```
SELECT * FROM products WHERE category = 'Gifts'--' AND released
= 1
```

The key thing here is that the double-dash sequence -- is a comment indicator in SQL, and means that the rest of the query is interpreted as a comment. This effectively removes the remainder of the query, so it no longer includes AND released = 1. This means that all products are displayed, including unreleased products.

Going further, an attacker can cause the application to display all the products in any category, including categories that they don't know about:

```
https://insecure-website.com/products?category=Gifts'+OR+1=1--
```

This results in the SQL query:

```
SELECT * FROM products WHERE category = 'Gifts' OR 1=1--' AND
released = 1
```

The modified query will return all items where either the category is Gifts, or 1 is equal to 1. Since 1=1 is always true, the query will return all items.

b) Subverting application logic

Consider an application that lets users log in with a username and password. If a user submits the username wiener and the password bluecheese, the application checks the credentials by performing the following SQL query:

```
SELECT * FROM users WHERE username = 'wiener' AND password =
'bluecheese'
```

If the query returns the details of a user, then the login is successful. Otherwise, it is rejected.

Here, an attacker can log in as any user without a password simply by using the SQL comment sequence -- to remove the password check from the WHERE clause of the query. For example, submitting the username administrator'-- and a blank password results in the following query:

```
SELECT * FROM users WHERE username = 'administrator'--' AND
password = ''
```

This query returns the user whose username is administrator and successfully logs the attacker in as that user.

c) Retrieving data from other database tables

In cases where the results of an SQL query are returned within the application's responses, an attacker can leverage an SQL injection vulnerability to retrieve data from other tables within the database. This is done using the UNION keyword, which lets you execute an additional SELECT query and append the results to the original query.

For example, if an application executes the following query containing the user input "Gifts":

```
SELECT name, description FROM products WHERE category = 'Gifts'
```

then an attacker can submit the input:

```
' UNION SELECT username, password FROM users--
```

This will cause the application to return all usernames and passwords along with the names and descriptions of products.

d) Examining the database

Following initial identification of an SQL injection vulnerability, it is generally useful to obtain some information about the database itself. This information can often pave the way for further exploitation.

You can query the version details for the database. The way that this is done depends on the database type, so you can infer the database type from whichever technique works. For example, on Oracle you can execute:

```
SELECT * FROM $version
```

You can also determine what database tables exist, and which columns they contain. For example, on most databases you can execute the following query to list the tables:

```
SELECT * FROM information_schema.tables
```

e) Blind SQL injection vulnerabilities

Many instances of SQL injection are blind vulnerabilities. This means that the application does not return the results of the SQL query or the details of any database errors within its responses. Blind vulnerabilities can still be exploited to access unauthorized data, but the techniques involved are generally more complicated and difficult to perform.

Depending on the nature of the vulnerability and the database involved, the following techniques can be used to exploit blind SQL injection vulnerabilities:

You can change the logic of the query to trigger a detectable difference in the application's response depending on the truth of a single condition. This might involve injecting a new condition into some Boolean logic, or conditionally triggering an error such as a divide-by-zero.

You can conditionally trigger a time delay in the processing of the query, allowing you to infer the truth of the condition based on the time that the application takes to respond.

You can trigger an out-of-band network interaction, using OAST techniques. This technique is extremely powerful and works in situations where the other techniques do not. Often, you can directly exfiltrate data via the out-of-band channel, for example by placing the data into a DNS lookup for a domain that you control.

6.2.3. How to prevent SQL injection

Most instances of SQL injection can be prevented by using parameterized queries (also known as prepared statements) instead of string concatenation within the query.

The following code is vulnerable to SQL injection because the user input is concatenated directly into the query:

```
String query = "SELECT * FROM products WHERE category = '"+
input + "'";
Statement statement = connection.createStatement();
```

```
ResultSet resultSet = statement.executeQuery(query);
```

This code can be easily rewritten in a way that prevents the user input from interfering with the query structure:

```
PreparedStatement statement =
connection.prepareStatement("SELECT * FROM products WHERE
category = ?");
statement.setString(1, input);
ResultSet resultSet = statement.executeQuery();
```

Parameterized queries can be used for any situation where untrusted input appears as data within the query, including the WHERE clause and values in an INSERT or UPDATE statement. They can't be used to handle untrusted input in other parts of the query, such as table or column names, or the ORDER BY clause. Application functionality that places untrusted data into those parts of the query will need to take a different approach, such as white-listing permitted input values, or using different logic to deliver the required behavior.

For a parameterized query to be effective in preventing SQL injection, the string that is used in the query must always be a hard-coded constant, and must never contain any variable data from any origin. Do not be tempted to decide case-by-case whether an item of data is trusted, and continue using string concatenation within the query for cases that are considered safe. It is all too easy to make mistakes about the possible origin of data, or for changes in other code to violate assumptions about what data is tainted.

6.2.4. SQL Injection Labs

Low SQL Injection Source

```
<?php

if( isset( $_REQUEST[ 'Submit' ] ) ) {
    // Get input
    $id = $_REQUEST[ 'id' ];

    // Check database
    $query  = "SELECT first_name, last_name FROM users WHERE use
r_id = '$id';";
    $result = mysqli_query($GLOBALS["___mysqli_ston"],  $query )
 or die( '<pre>' . ((is_object($GLOBALS["___mysqli_ston"])) ? my
sqli_error($GLOBALS["___mysqli_ston"]) : (($___mysqli_res = mysq
li_connect_error()) ? $___mysqli_res : false)) . '</pre>' );
```

```
    // Get results
    while( $row = mysqli_fetch_assoc( $result ) ) {
        // Get values
        $first = $row["first_name"];
        $last  = $row["last_name"];

        // Feedback for end user
        echo "<pre>ID: {$id}<br />First name: {$first}<br />Surn
ame: {$last}</pre>";
    }

    mysqli_close($GLOBALS["___mysqli_ston"]);
}

?>
```

Impossible SQL Injection Source

```
<?php

if( isset( $_GET[ 'Submit' ] ) ) {
    // Check Anti-CSRF token
    checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_t
oken' ], 'index.php' );

    // Get input
    $id = $_GET[ 'id' ];

    // Was a number entered?
    if(is_numeric( $id )) {
        // Check the database
        $data = $db-
>prepare( 'SELECT first_name, last_name FROM users WHERE user_id
 = (:id) LIMIT 1;' );
        $data->bindParam( ':id', $id, PDO::PARAM_INT );
        $data->execute();
        $row = $data->fetch();

        // Make sure only 1 result is returned
        if( $data->rowCount() == 1 ) {
            // Get values
            $first = $row[ 'first_name' ];
            $last  = $row[ 'last_name' ];

            // Feedback for end user
            echo "<pre>ID: {$id}<br />First name: {$first}<br />
Surname: {$last}</pre>";
        }
    }
}
```

```
// Generate Anti-CSRF token
generateSessionToken();

?>
```

SQLmap

Real website:

- http://www.keenlight.com.hk/
- http://hk-waitat.com/web/

6.3. Insecure direct object references (IDOR)

6.3.1. What are insecure direct object references (IDOR)?

Insecure direct object references (IDOR) are a type of access control vulnerability that arises when an application uses user-supplied input to access objects directly. The term IDOR was popularized by its appearance in the OWASP 2007 Top Ten. However, it is just one example of many access control implementation mistakes that can lead to access controls being circumvented. IDOR vulnerabilities are most commonly associated with horizontal privilege escalation, but they can also arise in relation to vertical privilege escalation.

6.3.2. IDOR examples

There are many examples of access control vulnerabilities where user-controlled parameter values are used to access resources or functions directly.

a) IDOR vulnerability with direct reference to database objects

Consider a website that uses the following URL to access the customer account page, by retrieving information from the back-end database:

```
https://insecure-
website.com/customer_account?customer_number=132355
```

Here, the customer number is used directly as a record index in queries that are performed on the back-end database. If no other controls are in place, an attacker can simply modify the customer_number value, bypassing access controls to view the records of other customers. This is an example of an IDOR vulnerability leading to horizontal privilege escalation.

An attacker might be able to perform horizontal and vertical privilege escalation by altering the user to one with additional privileges while bypassing

access controls. Other possibilities include exploiting password leakage or modifying parameters once the attacker has landed in the user's accounts page, for example.

b) IDOR vulnerability with direct reference to static files

IDOR vulnerabilities often arise when sensitive resources are located in static files on the server-side filesystem. For example, a website might save chat message transcripts to disk using an incrementing filename, and allow users to retrieve these by visiting a URL like the following:

```
https://insecure-website.com/static/12144.txt
```

In this situation, an attacker can simply modify the filename to retrieve a transcript created by another user and potentially obtain user credentials and other sensitive data.

6.3.3. IDOR Labs

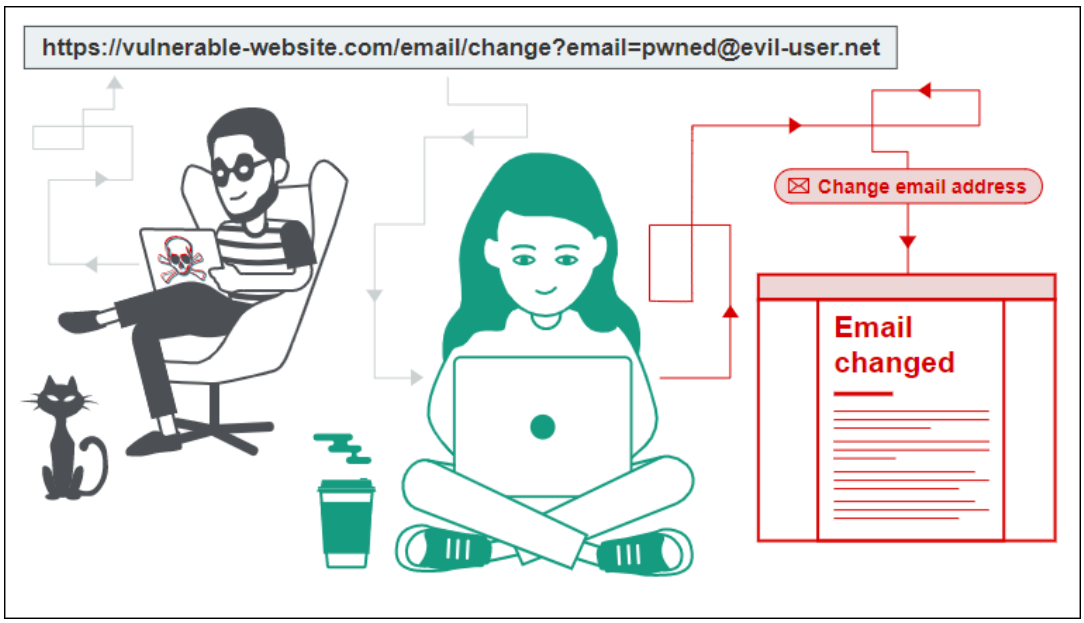
Virtual Lab: https://juice-shop.herokuapp.com/

In wild: https://fado.vn/

6.4. Cross-site request forgery (CSRF)

6.4.1. What is CSRF?

Cross-site request forgery (also known as CSRF) is a web security vulnerability that allows an attacker to induce users to perform actions that they do not intend to perform. It allows an attacker to partly circumvent the same origin policy, which is designed to prevent different websites from interfering with each other.



6.4.2. How does CSRF work?

For a CSRF attack to be possible, three key conditions must be in place:

- A relevant action. There is an action within the application that the attacker has a reason to induce. This might be a privileged action (such as modifying permissions for other users) or any action on user-specific data (such as changing the user's own password).
- Cookie-based session handling. Performing the action involves issuing one or more HTTP requests, and the application relies solely on session cookies to identify the user who has made the requests. There is no other mechanism in place for tracking sessions or validating user requests.
- No unpredictable request parameters. The requests that perform the action do not contain any parameters whose values the attacker cannot determine or guess. For example, when causing a user to change their password, the function is not vulnerable if an attacker needs to know the value of the existing password.

For example, suppose an application contains a function that lets the user change the email address on their account. When a user performs this action, they make an HTTP request like the following:

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 30
```

```
Cookie: session=yvthwsztyeQkAPzeQ5gHgTvlyxHfsAfE

email=wiener@normal-user.com
```

This meets the conditions required for CSRF:

The action of changing the email address on a user's account is of interest to an attacker. Following this action, the attacker will typically be able to trigger a password reset and take full control of the user's account.

The application uses a session cookie to identify which user issued the request. There are no other tokens or mechanisms in place to track user sessions.

The attacker can easily determine the values of the request parameters that are needed to perform the action.

With these conditions in place, the attacker can construct a web page containing the following HTML:

```
<html>
  <body>
    <form action="https://vulnerable-website.com/email/change"
method="POST">
      <input type="hidden" name="email" value="pwned@evil-
user.net" />
    </form>
    <script>
      document.forms[0].submit();
    </script>
  </body>
</html>
```

If a victim user visits the attacker's web page, the following will happen:

- The attacker's page will trigger an HTTP request to the vulnerable web site.
- If the user is logged in to the vulnerable web site, their browser will automatically include their session cookie in the request (assuming SameSite cookies are not being used).
- The vulnerable web site will process the request in the normal way, treat it as having been made by the victim user, and change their email address.

6.4.3. Preventing CSRF attacks

The most robust way to defend against CSRF attacks is to include a CSRF token within relevant requests. The token should be:

- Unpredictable with high entropy, as for session tokens in general.
- Tied to the user's session.
- Strictly validated in every case before the relevant action is executed.

6.4.4. CSRF Labs

Low CSRF Source

```
<?php

if( isset( $_GET[ 'Change' ] ) ) {
    // Get input
    $pass_new  = $_GET[ 'password_new' ];
    $pass_conf = $_GET[ 'password_conf' ];

    // Do the passwords match?
    if( $pass_new == $pass_conf ) {
        // They do!
        $pass_new = ((isset($GLOBALS["___mysqli_ston"]) && is_ob
ject($GLOBALS["___mysqli_ston"])) ? mysqli_real_escape_string($G
LOBALS["___mysqli_ston"],  $pass_new ) : ((trigger_error("[MySQL
ConverterToo] Fix the mysql_escape_string() call! This code does
 not work.", E_USER_ERROR)) ? "" : ""));
        $pass_new = md5( $pass_new );

        // Update the database
        $insert = "UPDATE `users` SET password = '$pass_new' WHE
RE user = '" . dvwaCurrentUser() . "';";
        $result = mysqli_query($GLOBALS["___mysqli_ston"],  $ins
ert ) or die( '<pre>' . ((is_object($GLOBALS["___mysqli_ston"]))
 ? mysqli_error($GLOBALS["___mysqli_ston"]) : (($___mysqli_res =
 mysqli_connect_error()) ? $___mysqli_res : false)) . '</pre>' )
;

        // Feedback for the user
        echo "<pre>Password Changed.</pre>";
    }
    else {
        // Issue with passwords matching
        echo "<pre>Passwords did not match.</pre>";
    }

    ((is_null($___mysqli_res = mysqli_close($GLOBALS["___mysqli_
ston"]))) ? false : $___mysqli_res);
}

?>
```

Impossible CSRF Source

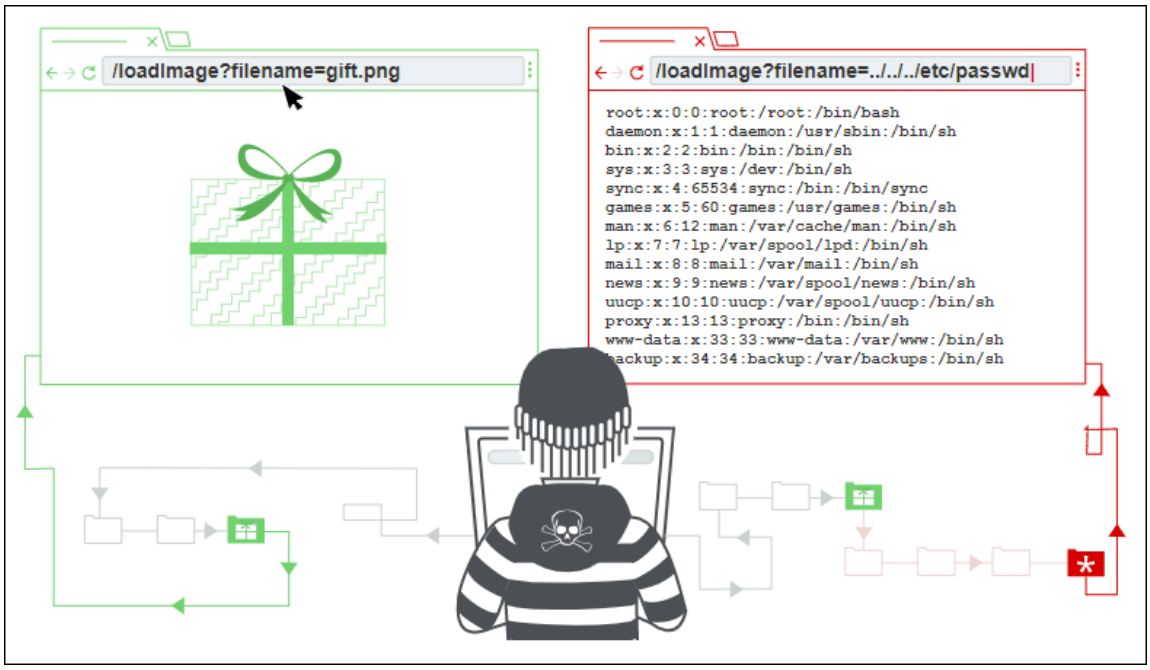
```
<?php

if( isset( $_GET[ 'Change' ] ) ) {
    // Check Anti-CSRF token
    checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_t
oken' ], 'index.php' );

    // Get input
    $pass_curr = $_GET[ 'password_current' ];
    $pass_new  = $_GET[ 'password_new' ];
    $pass_conf = $_GET[ 'password_conf' ];

    // Sanitise current password input
    $pass_curr = stripslashes( $pass_curr );
    $pass_curr = ((isset($GLOBALS["___mysqli_ston"]) && is_objec
t($GLOBALS["___mysqli_ston"])) ? mysqli_real_escape_string($GLOB
ALS["___mysqli_ston"],  $pass_curr ) : ((trigger_error("[MySQLCo
nverterToo] Fix the mysql_escape_string() call! This code does n
ot work.", E_USER_ERROR)) ? "" : ""));
    $pass_curr = md5( $pass_curr );

    // Check that the current password is correct
    $data = $db-
>prepare( 'SELECT password FROM users WHERE user = (:user) AND p
assword = (:password) LIMIT 1;' );
    $data-
>bindParam( ':user', dvwaCurrentUser(), PDO::PARAM_STR );
    $data->bindParam( ':password', $pass_curr, PDO::PARAM_STR );
    $data->execute();

    // Do both new passwords match and does the current password
 match the user?
    if( ( $pass_new == $pass_conf ) && ( $data-
>rowCount() == 1 ) ) {
        // It does!
        $pass_new = stripslashes( $pass_new );
        $pass_new = ((isset($GLOBALS["___mysqli_ston"]) && is_ob
ject($GLOBALS["___mysqli_ston"])) ? mysqli_real_escape_string($G
LOBALS["___mysqli_ston"],  $pass_new ) : ((trigger_error("[MySQL
ConverterToo] Fix the mysql_escape_string() call! This code does
 not work.", E_USER_ERROR)) ? "" : ""));
        $pass_new = md5( $pass_new );

        // Update database with new password
        $data = $db-
>prepare( 'UPDATE users SET password = (:password) WHERE user =
(:user);' );
        $data-
>bindParam( ':password', $pass_new, PDO::PARAM_STR );
        $data-
>bindParam( ':user', dvwaCurrentUser(), PDO::PARAM_STR );
        $data->execute();
```

```
        // Feedback for the user
        echo "<pre>Password Changed.</pre>";
    }
    else {
        // Issue with passwords matching
        echo "<pre>Passwords did not match or current password i
ncorrect.</pre>";
    }
}

// Generate Anti-CSRF token
generateSessionToken();

?>
```

In Wild:

- https://tiki.vn/
- https://www.sendo.vn/

6.5. Directory traversal



6.5.1. What is directory traversal?

Directory traversal (also known as file path traversal) is a web security vulnerability that allows an attacker to read arbitrary files on the server that is running an application. This might include application code and data, credentials for back-end systems, and sensitive operating system files. In some cases, an attacker might be able to write to arbitrary files on the server, allowing them to modify application data or behavior, and ultimately take full control of the server.

6.5.2. Reading arbitrary files via directory traversal

Consider a shopping application that displays images of items for sale. Images are loaded via some HTML like the following:

```
<img src="/loadImage?filename=218.png">
```

The loadImage URL takes a filename parameter and returns the contents of the specified file. The image files themselves are stored on disk in the location /var/www/images/. To return an image, the application appends the requested filename to this base directory and uses a filesystem API to read the contents of the file. In the above case, the application reads from the following file path:

```
/var/www/images/218.png
```

The application implements no defenses against directory traversal attacks, so an attacker can request the following URL to retrieve an arbitrary file from the server's filesystem:

```
https://insecure-
website.com/loadImage?filename=../../../etc/passwd
```

This causes the application to read from the following file path:

```
/var/www/images/../../../etc/passwd
```

The sequence ../ is valid within a file path, and means to step up one level in the directory structure. The three consecutive ../ sequences step up from /var/www/images/ to the filesystem root, and so the file that is actually read is:

```
/etc/passwd
```

On Unix-based operating systems, this is a standard file containing details of the users that are registered on the server.

On Windows, both ../ and ..\ are valid directory traversal sequences, and an equivalent attack to retrieve a standard operating system file would be:

```
https://insecure-
website.com/loadImage?filename=..\..\..\windows\win.ini
```

6.5.3. How to prevent a directory traversal attack

The most effective way to prevent file path traversal vulnerabilities is to avoid passing user-supplied input to filesystem APIs altogether. Many application functions that do this can be rewritten to deliver the same behavior in a safer way.

If it is considered unavoidable to pass user-supplied input to filesystem APIs, then two layers of defense should be used together to prevent attacks:

The application should validate the user input before processing it. Ideally, the validation should compare against a whitelist of permitted values. If that isn't possible for the required functionality, then the validation should verify that the input contains only permitted content, such as purely alphanumeric characters.

After validating the supplied input, the application should append the input to the base directory and use a platform filesystem API to canonicalize the path. It should verify that the canonicalized path starts with the expected base directory.

Below is an example of some simple Java code to validate the canonical path of a file based on user input:

```
File file = new File(BASE_DIRECTORY, userInput);
if (file.getCanonicalPath().startsWith(BASE_DIRECTORY)) {
    // process file
}
```

6.5.4. Directory Traversal Labs

Low Directory Traversal Source

```
<?php

// The page we wish to display
$file = $_GET[ 'page' ];

?>
```

Medium Directory Traversal Source

```
<?php

// The page we wish to display
$file = $_GET[ 'page' ];

// Input validation
$file = str_replace( array( "http://", "https://" ), "", $file )
;
$file = str_replace( array( "../", "..\"" ), "", $file );
```

```
?>
```

High Directory Traversal Source

```
<?php

// The page we wish to display
$file = $_GET[ 'page' ];

// Input validation
if( !fnmatch( "file*", $file ) && $file != "include.php" ) {
    // This isn't the page we want!
    echo "ERROR: File not found!";
    exit;
}

?>
```

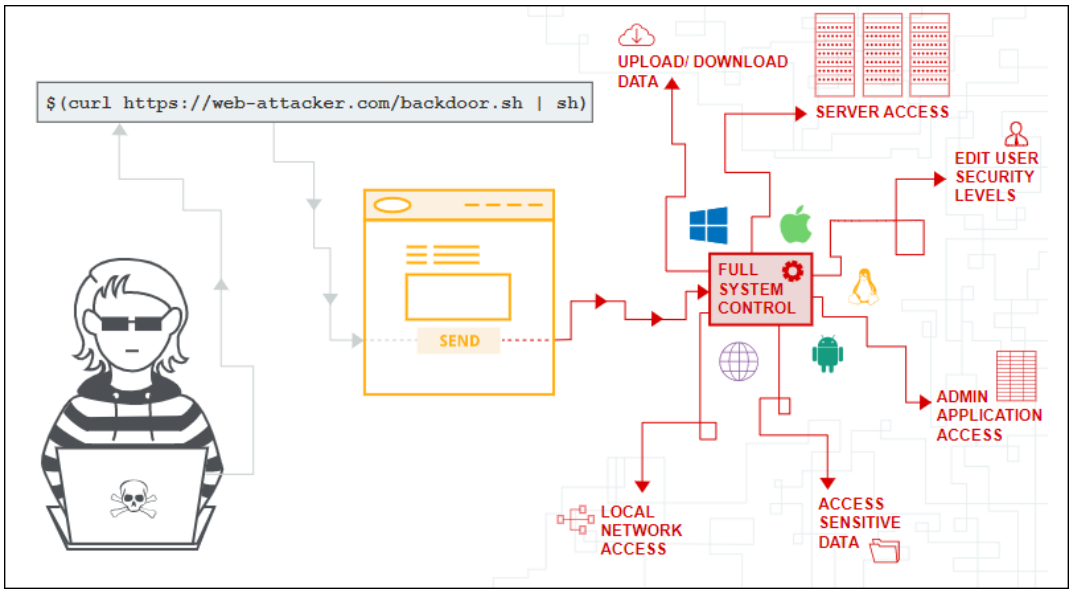
Impossible Directory Traversal Source

```
<?php

// The page we wish to display
$file = $_GET[ 'page' ];

// Only allow include.php or file{1..3}.php
if( $file != "include.php" && $file != "file1.php" && $file != "
file2.php" && $file != "file3.php" ) {
    // This isn't the page we want!
    echo "ERROR: File not found!";
    exit;
}

?>
```

In wild:

- https://www.vivelab12.fr/wp-content/plugins/jsmol2wp/php/jsmol.php?isform=true&call=getRawDataFromDatabase&query=php://filter/resource=../../../../../../../etc/passwd

6.6. OS command injection



6.6.1. What is OS command injection?

OS command injection (also known as shell injection) is a web security vulnerability that allows an attacker to execute arbitrary operating system (OS) commands on the server that is running an application, and typically fully compromise the application and all its data. Very often, an attacker can leverage an OS command injection vulnerability to compromise other parts of the hosting infrastructure, exploiting trust relationships to pivot the attack to other systems within the organization.

6.6.2. Executing arbitrary commands

Consider a shopping application that lets the user view whether an item is in stock in a particular store. This information is accessed via a URL like:

```
https://insecure-
website.com/stockStatus?productID=381&storeID=29
```

To provide the stock information, the application must query various legacy systems. For historical reasons, the functionality is implemented by calling out to a shell command with the product and store IDs as arguments:

```
stockreport.pl 381 29
```

This command outputs the stock status for the specified item, which is returned to the user.

Since the application implements no defenses against OS command injection, an attacker can submit the following input to execute an arbitrary command:

```
& echo aiwefwlguh &
```

If this input is submitted in the productID parameter, then the command executed by the application is:

```
stockreport.pl & echo aiwefwlguh & 29
```

The echo command simply causes the supplied string to be echoed in the output, and is a useful way to test for some types of OS command injection. The & character is a shell command separator, and so what gets executed is actually three separate commands one after another. As a result, the output returned to the user is:

```
Error - productID was not provided
aiwefwlguh
29: command not found
```

The three lines of output demonstrate that:

- The original stockreport.pl command was executed without its expected arguments, and so returned an error message.
- The injected echo command was executed, and the supplied string was echoed in the output.
- The original argument 29 was executed as a command, which caused an error.

Placing the additional command separator & after the injected command is generally useful because it separates the injected command from whatever follows the injection point. This reduces the likelihood that what follows will prevent the injected command from executing.

6.6.3. How to prevent OS command injection attacks

By far the most effective way to prevent OS command injection vulnerabilities is to never call out to OS commands from application-layer code. In virtually every case, there are alternate ways of implementing the required functionality using safer platform APIs.

If it is considered unavoidable to call out to OS commands with user-supplied input, then strong input validation must be performed. Some examples of effective validation include:

- Validating against a whitelist of permitted values.
- Validating that the input is a number.
- Validating that the input contains only alphanumeric characters, no other syntax or whitespace.

Never attempt to sanitize input by escaping shell metacharacters. In practice, this is just too error-prone and vulnerable to being bypassed by a skilled attacker.

6.6.4. OS Command Injection Labs

Low Command Injection Source

```
<?php

if( isset( $_POST[ 'Submit' ]  ) ) {
    // Get input
    $target = $_REQUEST[ 'ip' ];

    // Determine OS and execute the ping command.
    if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping  ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping  -c 4 ' . $target );
    }

    // Feedback for the end user
    echo "<pre>{$cmd}</pre>";
}

?>
```

Medium Command Injection Source

```
<?php

if( isset( $_POST[ 'Submit' ]  ) ) {
    // Get input
    $target = $_REQUEST[ 'ip' ];

    // Set blacklist
    $substitutions = array(
        '&&' => '',
```

```
        ';'  => '',
    );

    // Remove any of the charactars in the array (blacklist).
    $target = str_replace( array_keys( $substitutions ), $substi
tutions, $target );

    // Determine OS and execute the ping command.
    if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping  ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping  -c 4 ' . $target );
    }

    // Feedback for the end user
    echo "<pre>{$cmd}</pre>";
}

?>
```

High Command Injection Source

```
<?php

if( isset( $_POST[ 'Submit' ]  ) ) {
    // Get input
    $target = trim($_REQUEST[ 'ip' ]);

    // Set blacklist
    $substitutions = array(
        '&'  => '',
        ';'  => '',
        '| ' => '',
        '-'  => '',
        '$'  => '',
        '('  => '',
        ')'  => '',
        '`'  => '',
        '||' => '',
    );

    // Remove any of the charactars in the array (blacklist).
    $target = str_replace( array_keys( $substitutions ), $substi
tutions, $target );

    // Determine OS and execute the ping command.
    if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
```

```
            $cmd = shell_exec( 'ping  ' . $target );
        }
        else {
            // *nix
            $cmd = shell_exec( 'ping  -c 4 ' . $target );
        }

        // Feedback for the end user
        echo "<pre>{$cmd}</pre>";
}

?>
```

Impossible Command Injection Source

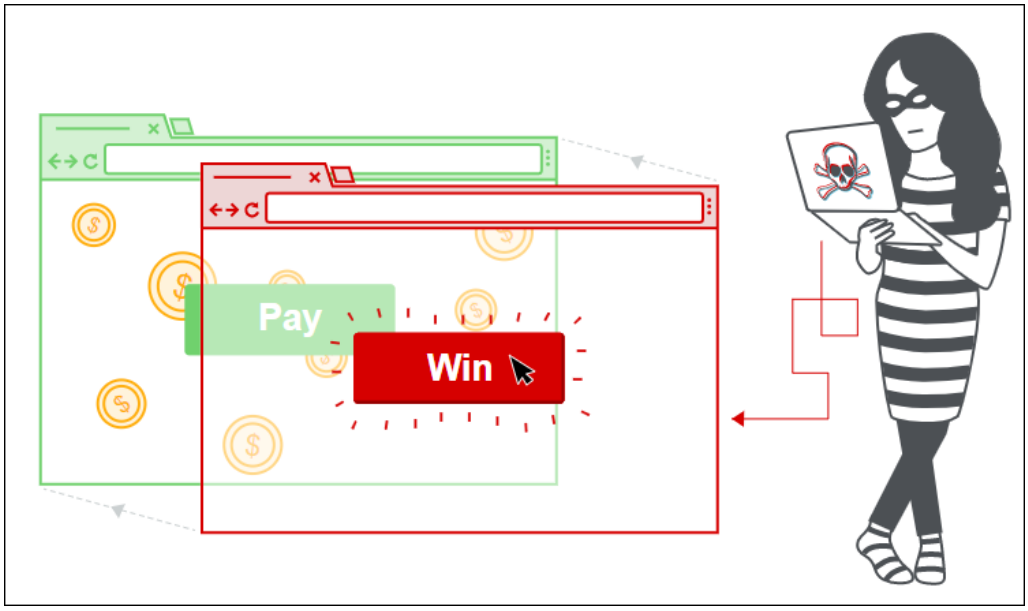
```
<?php

if( isset( $_POST[ 'Submit' ]  ) ) {
    // Check Anti-CSRF token
    checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_t
oken' ], 'index.php' );

    // Get input
    $target = $_REQUEST[ 'ip' ];
    $target = stripslashes( $target );

    // Split the IP into 4 octects
    $octet = explode( ".", $target );

    // Check IF each octet is an integer
    if( ( is_numeric( $octet[0] ) ) && ( is_numeric( $octet[1] )
 ) && ( is_numeric( $octet[2] ) ) && ( is_numeric( $octet[3] ) )
 && ( sizeof( $octet ) == 4 ) ) {
        // If all 4 octets are int's put the IP back together.
        $target = $octet[0] . '.' . $octet[1] . '.' . $octet[2]
. '.' . $octet[3];

        // Determine OS and execute the ping command.
        if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
            // Windows
            $cmd = shell_exec( 'ping  ' . $target );
        }
        else {
            // *nix
            $cmd = shell_exec( 'ping  -c 4 ' . $target );
        }

        // Feedback for the end user
        echo "<pre>{$cmd}</pre>";
    }
    else {
```

```
        // Ops. Let the user name theres a mistake
        echo '<pre>ERROR: You have entered an invalid IP.</pre>'
;
    }
}

// Generate Anti-CSRF token
generateSessionToken();

?>
```

6.7. Clickjacking (UI redressing)

6.7.1. What is clickjacking?

Clickjacking is an interface-based attack in which a user is tricked into clicking on actionable content on a hidden website by clicking on some other content in a decoy website.



6.7.2. How to prevent clickjacking attacks

We have discussed a commonly encountered browser-side prevention mechanism, namely frame busting scripts. However, we have seen that it is often straightforward for an attacker to circumvent these protections. Consequently, server driven protocols have been devised that constrain browser iframe usage and mitigate against clickjacking.

Clickjacking is a browser-side behavior and its success or otherwise depends upon browser functionality and conformity to prevailing web standards and best practice. Server-side protection against clickjacking is provided by defining and

communicating constraints over the use of components such as iframes. However, implementation of protection depends upon browser compliance and enforcement of these constraints. Two mechanisms for server-side clickjacking protection are X-Frame-Options and Content Security Policy.

6.7.3. X-Frame-Options

X-Frame-Options was originally introduced as an unofficial response header in Internet Explorer 8 and it was rapidly adopted within other browsers. The header provides the website owner with control over the use of iframes or objects so that inclusion of a web page within a frame can be prohibited with the deny directive:

```
X-Frame-Options: deny
```

Alternatively, framing can be restricted to the same origin as the website using the sameorigin directive

```
X-Frame-Options: sameorigin
```

or to a named website using the allow-from directive:

```
X-Frame-Options: allow-from https://normal-website.com
```

X-Frame-Options is not implemented consistently across browsers (the allow-from directive is not supported in Chrome version 76 or Safari 12 for example). However, when properly applied in conjunction with Content Security Policy as part of a multi-layer defense strategy it can provide effective protection against clickjacking attacks.

6.7.4. Content Security Policy (CSP)

Content Security Policy (CSP) is a detection and prevention mechanism that provides mitigation against attacks such as XSS and clickjacking. CSP is usually implemented in the web server as a return header of the form:

```
Content-Security-Policy: policy
```

where policy is a string of policy directives separated by semicolons. The CSP provides the client browser with information about permitted sources of web resources that the browser can apply to the detection and interception of malicious behaviors.

The recommended clickjacking protection is to incorporate the frame-ancestors directive in the application's Content Security Policy. The frame-ancestors 'none' directive is similar in behavior to the X-Frame-Options deny directive. The frame-ancestors 'self' directive is broadly equivalent to the X-Frame-Options sameorigin directive. The following CSP whitelists frames to the same domain only:

```
Content-Security-Policy: frame-ancestors 'self';
```

Alternatively, framing can be restricted to named sites:

```
Content-Security-Policy: frame-ancestors normal-website.com;
```

To be effective against clickjacking and XSS, CSPs need careful development, implementation and testing and should be used as part of a multi-layer defense strategy.

6.7.5. Clickjacking Labs

Virtual Labs: http://lab.awh.zdresearch.com

In Wild: https://sendo.vn

CHAPTER 7. SECURE CODING PRACTICES

7.1. Data Validation

Verification that the properties of all input and output data match what is expected by the application and that any potentially harmful data is made safe through the use of data removal, replacement, encoding, or escaping.

Conduct all data validation on the server side.

Encode data to a common character set before validating (Canonicalize).

Determine if the system supports UTF-8 extended character sets and if so, validate after UTF-8 decoding is completed.

Validate all client provided data before processing, including all form fields, URLs and HTTP header values. Be sure to include automated post backs from JavaScript, Flash or other embedded code.

Identify system trust boundaries and validate all data from external connections (e.g., Databases, file streams, etc.).

Validate all input against a "white" list of allowed characters.

Sanitize any potentially hazardous characters that must be allowed, like: <, >, ", ', %, (,), &, +, \, \', \"

Validate for expected data types.

Validate data range.

Validate data length.

Utilize a master validation routine to HTML entity encode all output to the client. If this is not possible, at least encode all dynamic content that originated outside the application's trust boundary.

7.2. Authentication and Password Management

Utilize standard security services when available, as they are designed to meet company requirements. (e.g., WSSO for intranet web based authentication)

Change all vendor-supplied default passwords and user IDs or disable the associated accounts.

Utilize re-authentication for critical operations.

Use two factor authentication for highly sensitive or high value transactional accounts.

Log out functionality should be available from all pages.

Log out functionality must fully terminate the associated session or connection.

If non-standard authentication is used, it must address the following:

- Validate the authentication data only on completion of all data input, especially for sequential authentication implementations.
- Error conditions must not indicate which part of the authentication data was incorrect. Error responses must be truly identical in both display and source code.
- Use only POST requests to transmit authentication credentials.
- Only send passwords over an encrypted connection.
- Enforce password complexity requirements. Passwords must include alphabetic as well as numeric and/or special characters.
- Enforce password length requirements. Passwords must be at least eight cha
- Obscured password on the user's screen. (e.g., On web forms use input type "password")
- Enforce account disabling after no more than five invalid logon attempts. The account must be disabled for a period of time sufficient to discourage brute force guessing of credentials.
- Password recovery and changing operations require the same level of controls as account creation and authentication.
- Password recovery should only send an email to a pre-registered email address with a temporary link/password which let’s the user reset the password.
- Temporary passwords and links should have a short expiration time.
- Password recovery questions should support sufficiently random answers.
- Enforce the changing of temporary passwords on the next use.
- Prevent password re-use.
- Passwords must be at least one day old before they can be changed, to prevent attacks on password re-use.
- Enforce password changes at least every 180 days. Critical systems may require more frequent changes.

- Disable "remember me" functionality for password fields.
- Log all authentication failures.
- Report unsuccessful logon attempts to a User ID, at its next successful logon.
- The last use of a User ID should be reported to the User ID at its next successful logon.
- Segregate authentication logic and use redirection during login.
- If your application manages credential storage, it should ensure that only the 1-way salted hashes of passwords are stored in the database, and that the table/file that stores the passwords and keys are “write”-able only to the application.
- Implement monitoring to identify attacks against multiple user accounts, utilizing the same password. This attack pattern is used to bypass standard lockouts.

7.3. Authorization and Access Management

Use only server side session objects for making authorization decisions.

Enforce authorization controls on every request, including server side scripts and includes.

Ensure that all directories, files or other resources outside the application's direct control have appropriate access controls in place.

If state data must be stored on the client, use encryption and integrity checking on the server side to catch state tampering. The application should log all apparent tampering events.

Enforce application logic flows to comply with business rules.

Use a single site wide component to check access authorization.

Segregate privileged logic from other application code.

Limit the number of transactions a single user or device can perform in a given period of time. The transactions/time should be above the actual business requirement, but low enough to deter automated attacks.

Use the referer header as a supplemental check only, it should never be the sole authorization check, as it is can be spoofed.

Create an Access Control Policy (ACP) to documents an asset's business rules and access authorization criteria and/or processes so that access can be

properly provisioned and controlled. This includes identifying access requirements for both the data and system resources.

If long authenticated sessions are allowed, periodically revalidate a user's authorization to ensure that their privileges have not changed.

Implement account auditing and enforce the disabling of unused accounts (After no more than 30 days from the expiration of an account's password.).

The application must support disabling of accounts and terminating of sessions when authorization ceases (Causes include changes to role, employment status, business process, etc.).

Isolate development environments from the production network and provide access only to authorized development and test groups. Development environments are often configured less securely than production environments and attackers may use this difference to discover shared weaknesses or as an avenue for exploitation.

7.4. Session Management

Establish a session inactivity timeout that is as short as possible. It should be no more than several hours.

If a session was established before login, close that session and establish a new session after a successful login.

Do not allow concurrent logins with the same user ID.

Use well vetted algorithms that ensure sufficiently random session identifiers.

Session ID creation must always be done on the server side.

Do not pass session identifiers as GET parameters.

Server side session data should have appropriate access controls in place.

Generate a new session ID and deactivate the old one frequently.

Generate a new session token if a user's privileges or role changes.

Generate a new session token if the connection security changes from HTTP to HTTPS.

Only utilize the system generated session IDs for client side session (state) management. Avoid using parameters or other client data for state management.

Utilize per-session random tokens or parameters within web forms or URLs associated with sensitive server-side operations, like account management, to prevent Cross Site Request Forgery attacks.

Utilize per-page random tokens or parameters to supplement the main session token for critical operations.

Ensure cookies transmitted over an encrypted connection have the "secure" attribute set.

Set cookies with the HttpOnly attribute, unless you specifically require client-side scripts within your application to read or set a cookie's value.

The application or system should log attempts to connect with invalid or expired session tokens.

Disallow persistent logins and enforce periodic session terminations, even when the session is active.

Especially for applications supporting rich network connections or connecting to critical systems.

Termination times should support business requirements and the user should receive sufficient notification to mitigation.

7.5. Sensitive Information Storage or Transmission

Implement approved encryption for the transmission of all sensitive information.

Encrypt highly sensitive stored information, like authentication verification data, even on the serverside.

Protect server side code from being downloaded.

Do not store passwords, connection strings or other sensitive information in clear text or in any noncryptographically secure manner on the client side. This includes embedding in insecure formats like: MS viewstate, Adobe flash or compiled code

Do not store sensitive information in logs

Implement least privilege, restrict users to just the functionality, data and system information that is required to perform their tasks.

Remove developer comments.

Remove unnecessary application and system documentation.

Turn off stack traces and other verbose system messages.

The application should handle application errors and not rely on the server configuration.

Filter GET parameters from the referer, when linking to external sites.

7.6. System Configuration Management

Ensure servers, frameworks and system components are patched.

Ensure servers, frameworks and system components are running the latest approved version.

Use safe exception handlers.

Disable any unnecessary Extended HTTP methods. If an Extended HTTP method that supports file handling is required, utilize an approved authentication mechanism. (e.g., WebDav)

Turn off directory listing.

Ensure SSL certificates have the correct domain name and are not expired.

Restrict the web server, process and service accounts to the least privileges possible.

Implement generic error messages and custom error pages that do not disclose system information.

When exceptions occur, fail secure.

Remove all unnecessary functionality and files.

Remove any test code.

Remove unnecessary information from HTTP response headers related to the OS, webserver version and application frameworks.

Prevent disclosure of your directory structure and stop robots from indexing sensitive files and directories by moving them into an isolated parent directory and then "Disallow" that entire parent directory in the robots.txt file.

Log all exceptions.

Restrict access to logs.

Log all administrative functions.

Use hashing technology to validate log integrity.

7.7. General Coding Practices

Utilize task specific built-in APIs to conduct operating system tasks. Do not allow the application to issue commands directly to the Operating System, especially through the use of application initiated command shells.

Use tested and approved managed code rather than creating new unmanaged code for common tasks.

Utilize locking to prevent multiple simultaneous requests to the application or use a synchronization mechanism to prevent race conditions.

Explicitly initialize all your variables and other data stores, either during declaration or just before the first usage.

Properly free allocated memory upon the completion of functions and at all exit points including error conditions.

In cases where the application must run with elevated privileges, raise privileges as late as possible, and drop them as soon as possible.

Avoid calculation errors by understanding your programming language's underlying representation and how it interacts with numeric calculation. Pay close attention to byte size discrepancies, precision, signed/unsigned distinctions, truncation, conversion and casting between types, "not-a-number" calculations, and how your language handles numbers that are too large or too small for its underlying representation.

Do not pass user supplied data to any dynamic execution function.

Restrict users from generating new code or altering existing code.

Review all secondary applications, third party code and libraries to determine business necessity and validate safe functionality, as these can introduce new vulnerabilities.

Implement safe updating. If the application will utilize automatic updates, then use cryptographic signatures for your code and ensure your download clients verify those signatures. Use encrypted channels to transfer the code from the host server.

7.8. Database Security

Use strongly typed parameterized queries. Parameterized queries keep the query and data separate through the use of placeholders. The query structure is defined with place holders and then the application specifies the contents of each placeholder.

Utilize input validation and if validation fails, do not run the database command.

Ensure that the input does not include unintended SQL keywords.

Ensure that variables are strongly typed.

Escape meta characters in SQL statements.

The application should use the lowest possible level of privilege when accessing the database.

Use secure credentials for database access.

Do not provide connection strings or credentials directly to the client. If this is unavoidable, credentials must be encrypted.

Use stored procedures to abstract data access.

Turn off any database functionality (e.g., unnecessary stored procedures or services).

Eliminate default content.

Disable any default accounts that are not required to support business requirements.

Close the connection as soon as possible.

The application should connect to the database with different credentials for every trust distinction (e.g., user, read-only user, guest, administrators).

7.9. File Management

Do not pass user supplied data directly to any dynamic include function.

Limit the type of files that can be uploaded to only those types that are needed for business purposes.

Validate uploaded files are the expected type.

Do not save files in the web space. If this must be allowed, prevent or restrict the uploading of any file that can be interpreted by the web server.

Turn off execution privileges on file upload directories.

Implement safe uploading in UNIX by mounting the targeted file directory as a logical drive using the associated drive letter or the chrooted environment.

When referencing existing files, use a hard coded list of allowed file names and types. Validate the value of the parameter being passed and if it does not match one of the expected values, use a hard coded default file value for the content instead.

Do not pass user supplied data into a dynamic redirect. If this must be allowed, then the redirect should accept only validated, relative path URLs.

Do not pass directory or file paths, use index values mapped to hard coded paths.

Never send the absolute file path to the user.

Ensure application files and resources are read-only.

Implement access controls for temporary files.

Remove temporary files as soon as possible.

7.10.Memory Management

Utilize input validation.

Double check that the buffer is as large as specified.

When using functions that accept a number of bytes to copy, such as `strncpy()`, be aware that if the destination buffer size is equal to the source buffer size, it may not NULL-terminate the string.

Check buffer boundaries if calling the function in a loop and make sure there is no danger of writing past the allocated space.

Truncate all input strings to a reasonable length before passing them to the copy and concatenation functions.

Specifically close resources, don't rely on garbage collection. (e.g., connection objects, file handles, etc.)

Use Non-executable stacks when available.

Avoid the use of known vulnerable functions (e.g., printf, strcat, strcpy, etc).

7.11. Lab Secure Webgoat - Java Spring Boot

Requirement:

- Java
- IntelliJ
- Github
- Sublime

Will be provided at Offline class

7.12. Lab ASP .Net Core Authentication

Requirement:

- ASP .Net core
- Visual Studio

Will be provided at Offline class

7.13. Lab Implement WAF Modsecurity

Requirement:

- VMWare

- Virtual Machine

Will be provided at Offline class

7.14. Lab Auditing Web Server and Web Service

Requirement:

- VMWare
- Virtual Machine

Will be provided at Offline class

7.15. Lab Source Code Analysis Tools

Will be provided at Offline class

7.16. Lab Dynamic Scan Vulnerabilities Tools

Requirement:

- ZAP
- Nessus

Will be provided at Offline class