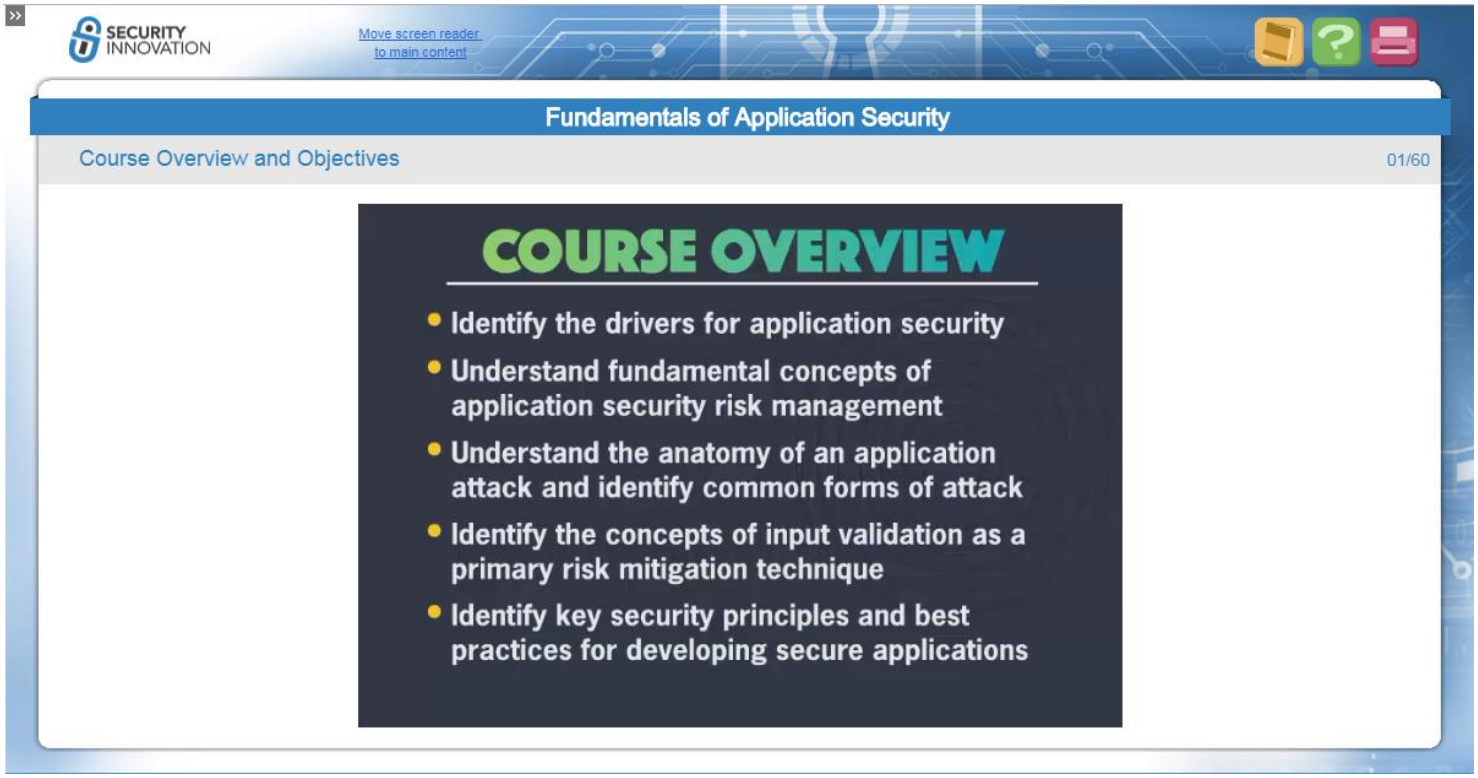
Course Overview and Objectives



Narration

In this course, you will be able to identify the drivers for application security, understand the fundamental concepts of application security risk management, and understand the anatomy of an application attack and identify common forms of attack.

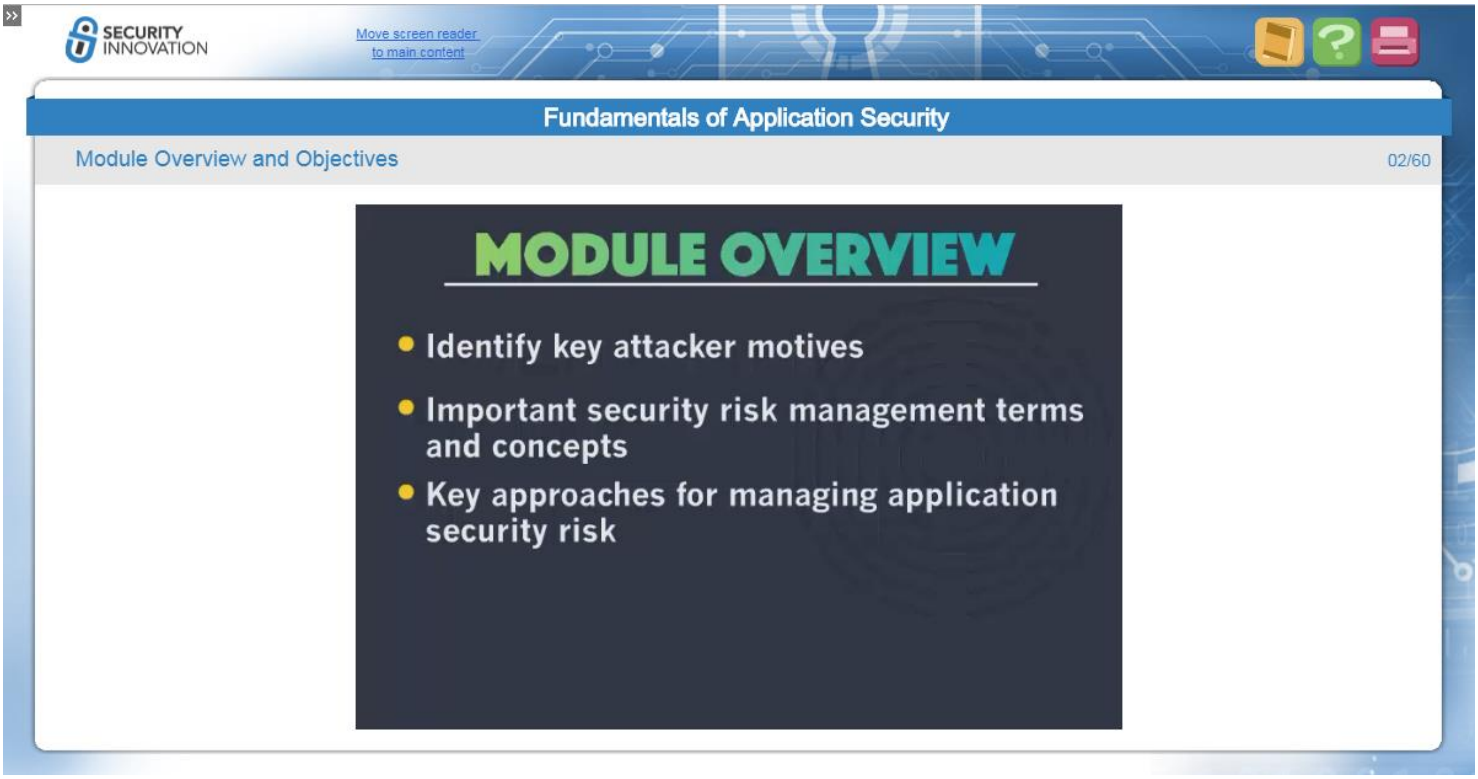
You will be also able to identify the concepts of input validation as a primary risk mitigation technique.

In addition, you will be able to identify the security principles and best practices for developing secure applications.

On Screen Text

Course Overview and Objectives

Module Overview and Objectives



Narration

This module will teach you application security fundamentals that will form the basic foundation for later modules and courses.

After completing this module you will be able to understand what application security is and understand the technical, business, and regulatory drivers for application security.

You will also be able to identify key attacker motives, important security risk management terms and concepts, and key approaches for managing application security risk.

On Screen Text

Module Overview and Objectives

What is Application Security?

Narration

Let’s begin by gaining an understanding of what application security is. Application security simply refers to an application’s ability to withstand malicious attack.

For example, how resilient is your application to unauthorized attempts to access customer credit card information? Or how well does your application resist attacks designed to shut it down?

In order to do this, applications are designed and developed against three key security goals: confidentiality, integrity and availability. These are sometimes referred to as the CIA triad.

The goal of confidentiality is to prevent unauthorized disclosure of or access to sensitive data managed by applications, such as personal and financial data.

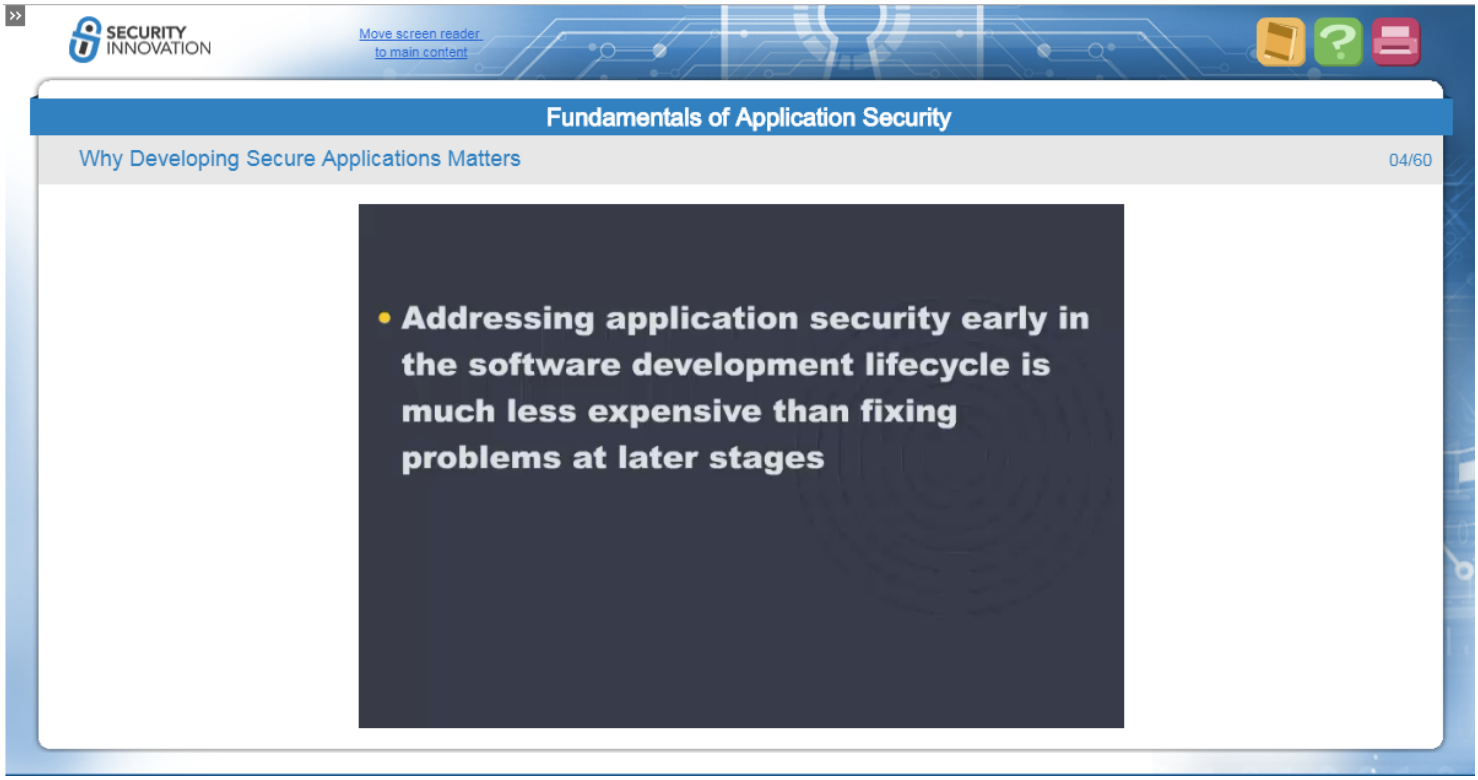
The goal of integrity is to protect data from unauthorized modification or deletion.

And, the goal of availability is to ensure that the application and its services remain accessible to users.

On Screen Text

What is Application Security?

Why Developing Secure Applications Matters



Narration

High-profile security breaches and the negative press that follows make application security an important business consideration.

Today, it is no longer a matter of if your application will get hacked, but rather when it will happen.

Lack of application security in today's threat landscape places your organization and its customers at risk for significant loss.

Here are some important reasons why you and your organization should be concerned about application security:

The first reason is loss of revenue. An application security breach can cause damage such as disruption of business operations,

theft of proprietary data, and, in some cases, regulatory or state fines – all of which can have direct financial impact to your organization.

In addition, your brand can be damaged if customers no longer trust your ability to keep their data safe.

If a security breach involves customers' personal data that gets misused by criminals, this could cause further ill will towards your organization.

In general, customers today are better educated about security, and are demanding that the organizations they do business with meet high application security standards.

This creates competitive pressure: Organizations that don't meet the standard take the risk of losing business to competitors who can use the security of their applications as a market advantage.

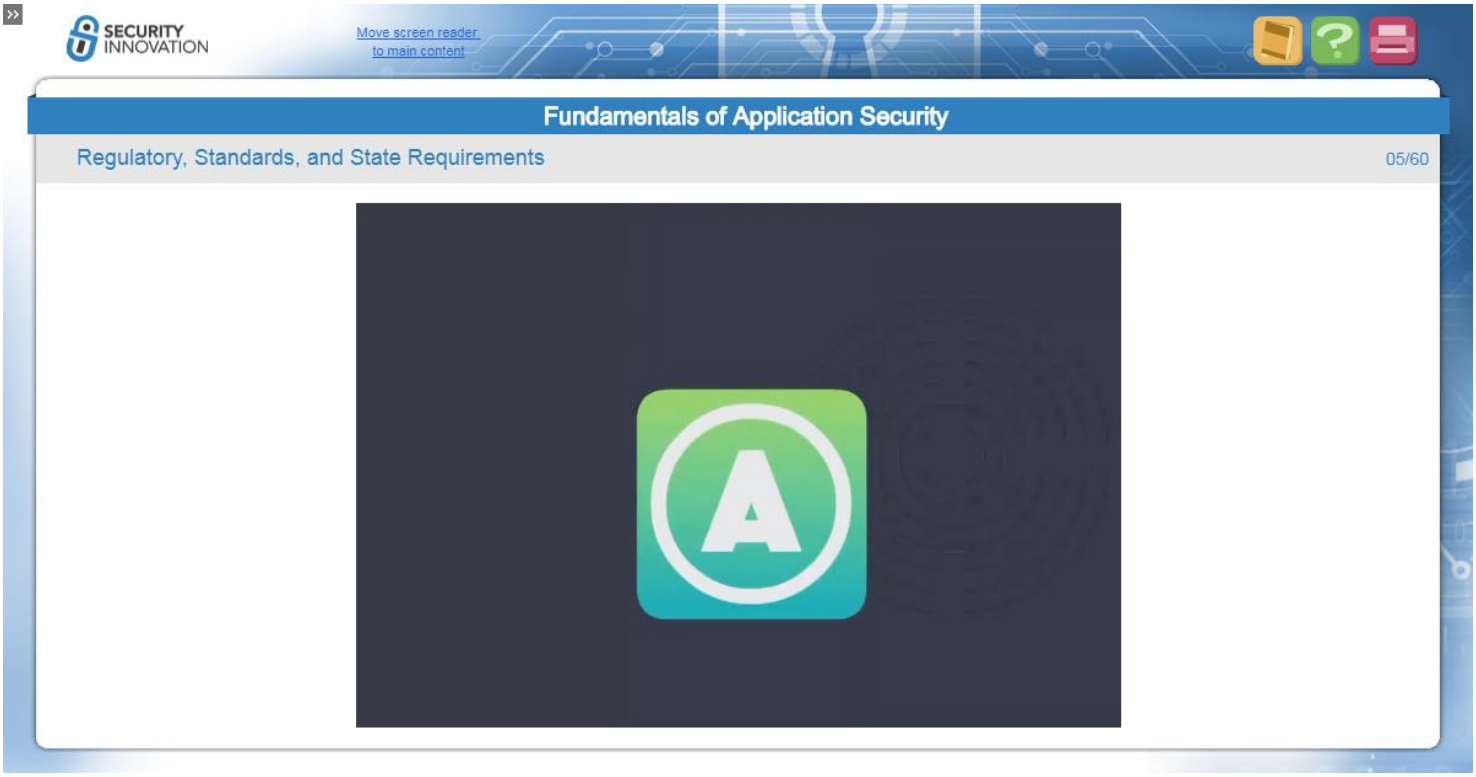
Finally, it's important to note that your organization will realize cost savings if you address application security early in the software development lifecycle.

This significantly less expensive than the cost of addressing security after you have been hacked.

On Screen Text

Why Developing Secure Applications Matters

Regulatory, Standards, and State Requirements



Narration

Depending on your industry, your organization may need to comply with strict regulations, standards, or state requirements regarding application security.

For example, the Payment Card Industry, or PCI, has many application security requirements that apply to all software that handles payment card data.

And, the healthcare and financial industries have their own regulations, standards, and guidelines that organizations must follow.

Failure to meet these requirements can lead to significant monetary fines and business penalties.

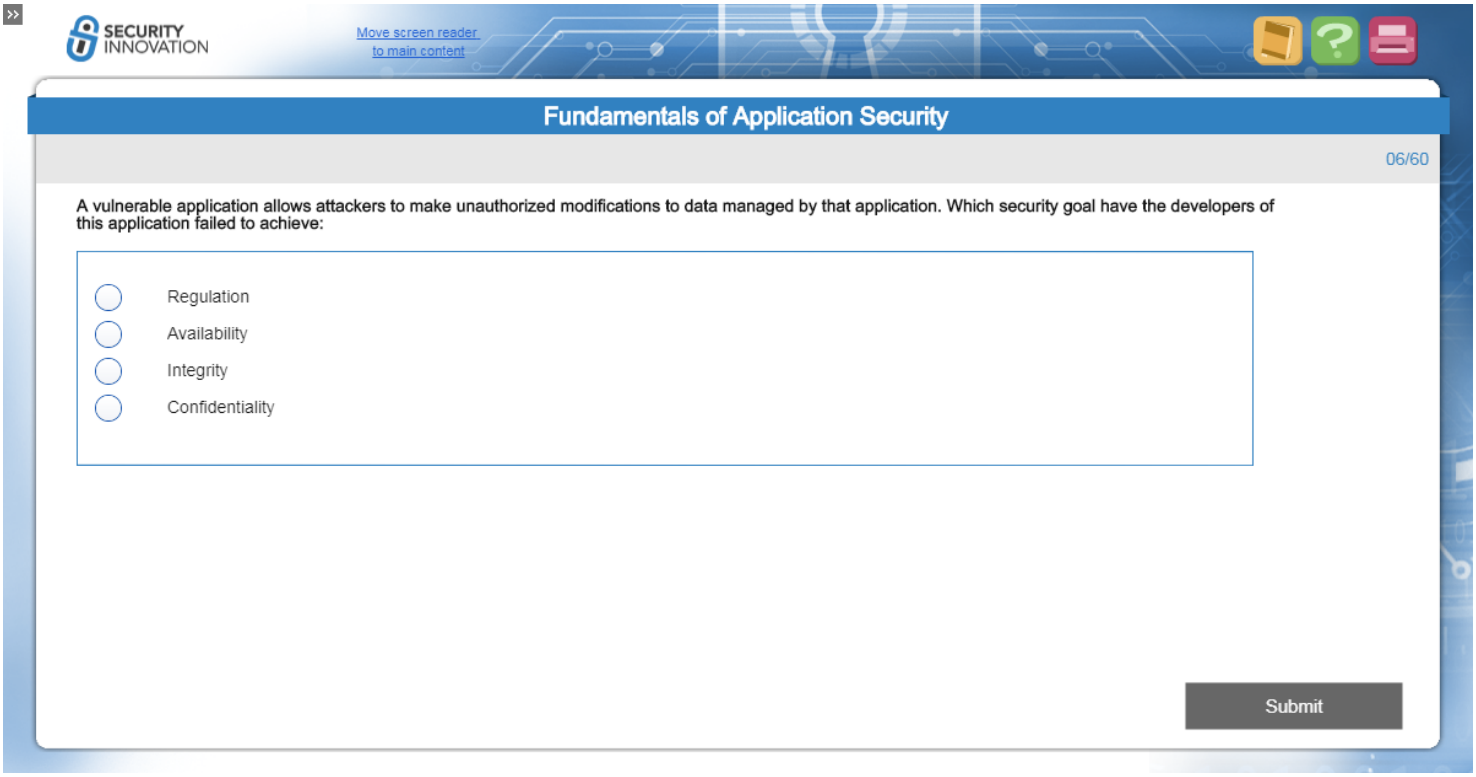
For very serious violations, companies could potentially be barred from conducting business altogether.

It is therefore important to become familiar with application security requirements relevant to your industry, in order to remain compliant.

On Screen Text

Regulatory, Standards, and State Requirements

Knowledge Check



Narration

On Screen Text

A vulnerable application allows attackers to make unauthorized modifications to data managed by that application. Which security goal have the developers of this application failed to achieve:

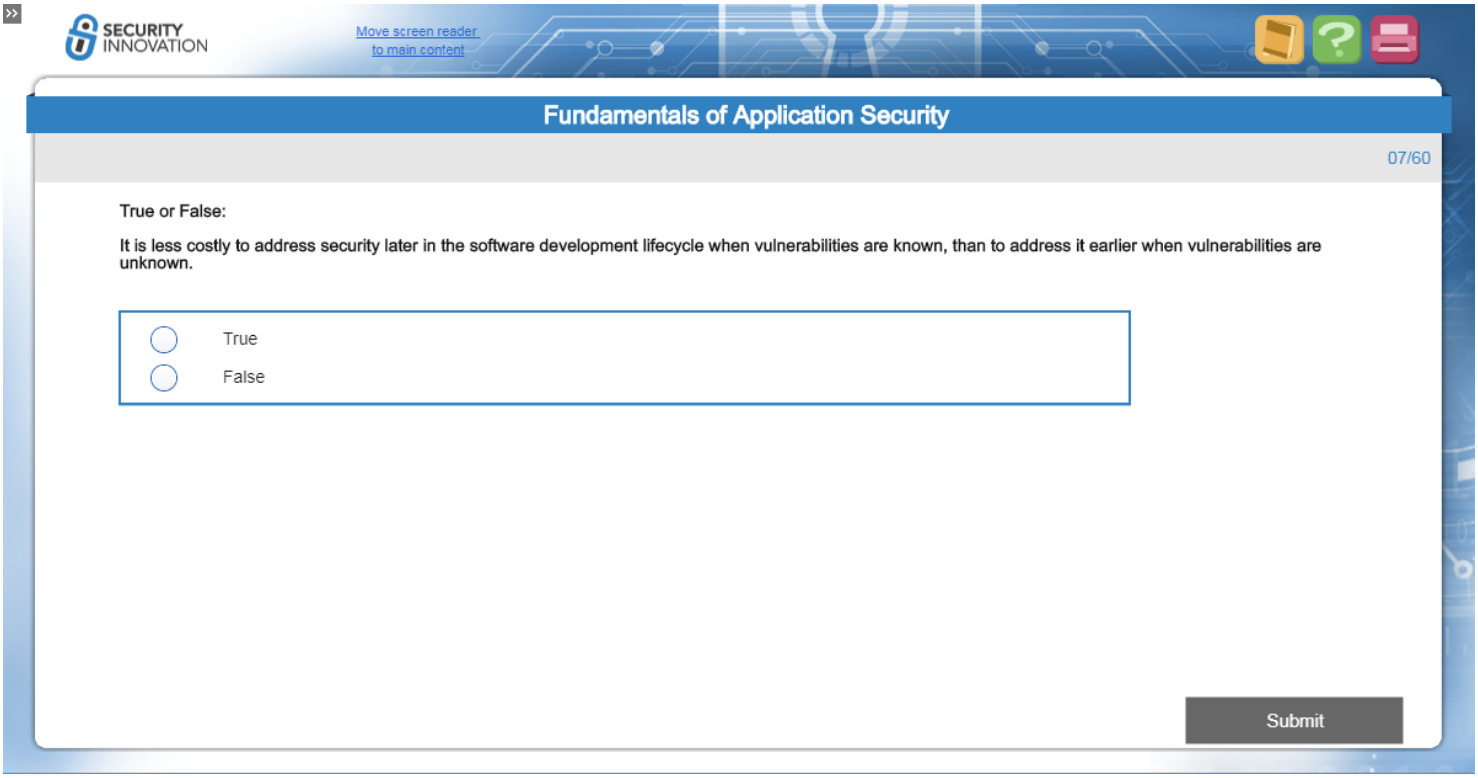
Regulation

Availability

Integrity

Confidentiality

Knowledge Check



Narration

On Screen Text

It is less costly to address security later in the software development lifecycle when vulnerabilities are known, than to address it earlier when vulnerabilities are unknown.

True

False

Understanding the Attacker

Narration

Why do attackers target your applications? It's impossible to know the motives of every individual attacker,

but here are some key reasons why attacks on applications are increasing:

Today's digital black market lets attackers convert stolen data into cash.

Attackers and organized crime gangs are highly incentivized to attack applications like yours that may contain valuable data.

For example, stolen credit card numbers could fetch as high as $100 dollars per number on the black market.

Medical records and social security numbers can fetch even more because users can't simply cancel their medical records or social security number as they could do with a credit card.

A single application could house personal data on thousands, hundreds of thousands, or even millions of people, and could yield an incredible pay day for criminals.

In addition, it no longer takes a lot of skill to attack an application. Attackers have many publicly available tools that they can use to do the work for them.

Also, the number of targets has dramatically increased, and so has their availability online.

If an attacker is unable to compromise one application, there are plenty more targets to try.

Now that you have some understanding of attackers, let's look at how you can manage the application security risk they create.

On Screen Text

Understanding the Attacker

Security Risk Management Concepts and Terms

Narration

Listed here are some important terms and concepts you should understand in order to better manage your organization's application security risks.

Vulnerability:

A vulnerability is a weakness in your application that an attacker could potentially exploit.

This can be weakness in the code itself or perhaps even an application design flaw.

Exploit:

An exploit is a malicious application or a set of commands used to take advantage of vulnerabilities in your application.

The purpose of an exploit is to produce some unexpected or unintended behavior in your application,

such as exposing sensitive information or gaining unauthorized access to certain components.

Countermeasure:

A countermeasure is a step taken to address or reduce the potential for damage caused by a vulnerability in your application code.

Threat / Threat Agent:

A threat is an event that could exploit a vulnerability in your application code and cause some undesirable behavior.

Malicious hackers, or attackers, are often called threat agents because they exploit vulnerabilities in your application code to meet their objectives.

Attack:

An attack is an action that leverages one or more vulnerabilities to realize a threat.

For example, a malicious hacker may exploit a vulnerability in your authentication system to gain unauthorized access to an area of your application.

Asset:

An asset is a resource of value. For example, an asset to your organization may be a database of the health or financial information of your customers.

Risk:

Finally, risk is the potential for loss or damage to an asset. Whenever a developer follows insecure coding practices,

they increase the risk that an attacker may be able to exploit the flawed code.

On Screen Text

Security Risk Management Concepts and Terms

Managing Application Security Risks

Narration

Now that you are familiar with key security risk management concepts and terms,

it is important to underscore the principle that no application can be completely secure or resilient to all attacks.

You can never completely eliminate the risk that your application may be hacked.

The key to great application security, then, is to correctly manage your risk.

There are four main approaches to managing risk: you can accept, avoid, transfer, or mitigate risk.

Accepting risk means that you take no proactive measures to address your risk and that you are willing to accept the full consequences of a threat to an asset.

This is sometimes a reasonable approach if the cost of addressing the risk greatly exceeds both the value of the asset you wish to protect, and the consequences should the risk be realized.

Usually, you should use this approach only as a last resort.

When accepting risk, you should always have a contingency plan, which is a set of actions you plan to take if the risk is realized.

Avoiding risk is the opposite of accepting risk. To avoid risk, you remove the exposure of an asset to the threat, or you remove the asset entirely.

For example, let's say your application manages customer credit card data, which certainly involves risk.

Your organization can choose to avoid this risk altogether by not handling credit card data at all.

This leads us to the next risk management technique called transference, which means you transfer the burden of the risk onto a third party.

For example, instead of handling card information yourself, you may elect to pay a third-party with expertise in the payment card industry to handle it for you,

thus transferring the risk to them. Insurance is a classic example of transferring risk.

By purchasing insurance against some undesirable event, you're transferring the risk to the insurance company for a fee,

to decrease the overall loss should the risk be realized.

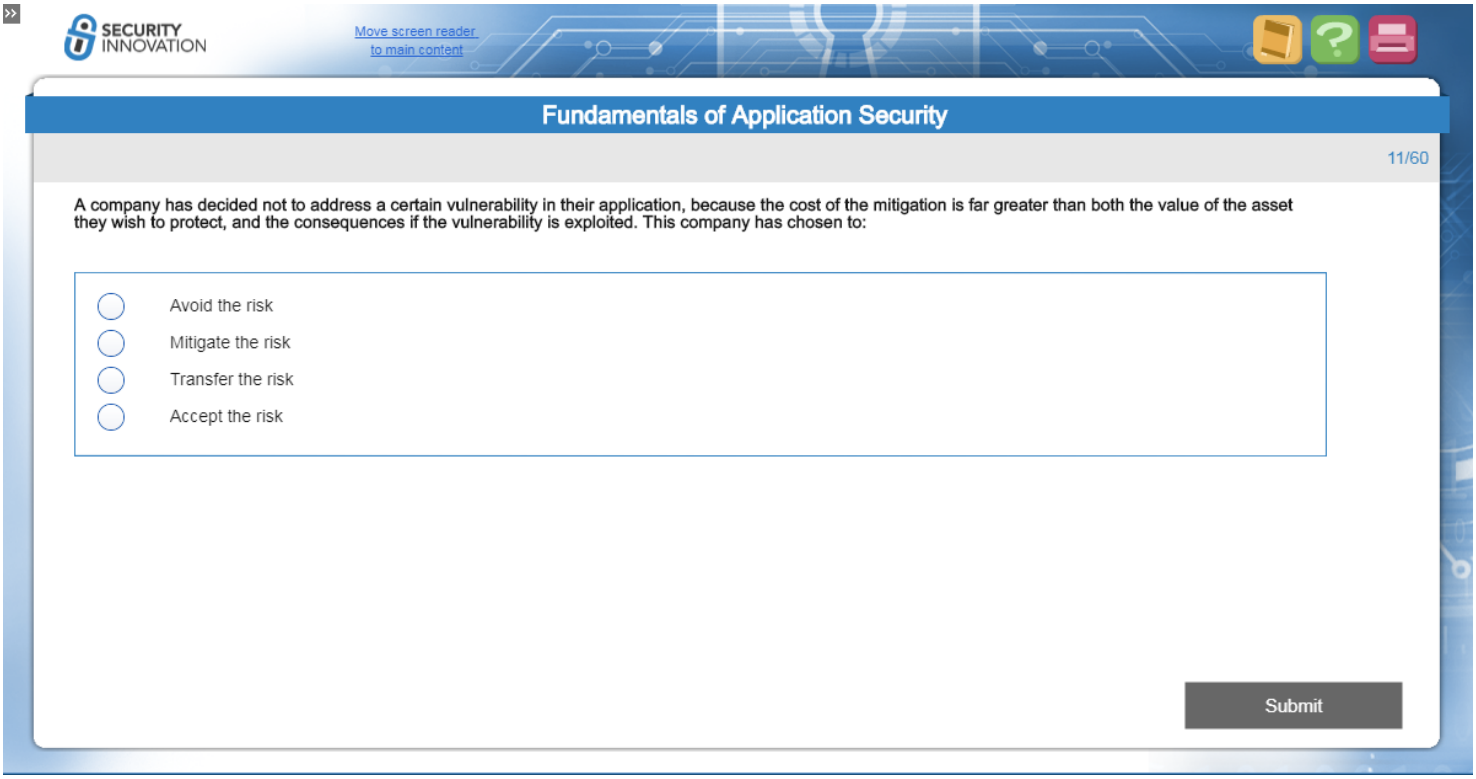
The final risk management approach is to mitigate risk, which means that you take proactive steps to reduce an asset's exposure to a risk.

For example, you can reduce the risk of an attacker guessing user account passwords by requiring all users to use strong passwords.
Or, to reduce the chances of an attacker intercepting sensitive data transmitted between your online application and users,
you could protect the data in transit by using Transport Layer Security (TLS).
Mitigation is the most common and recommended way to address application security risks.
It should be your primary approach to managing application security risk.

On Screen Text

Managing Application Security Risks

Knowledge Check



Narration

On Screen Text

A company has decided not to address a certain vulnerability in their application, because the cost of the mitigation is far greater than both the value of the asset they wish to protect, and the consequences if the vulnerability is exploited. This company has chosen to:

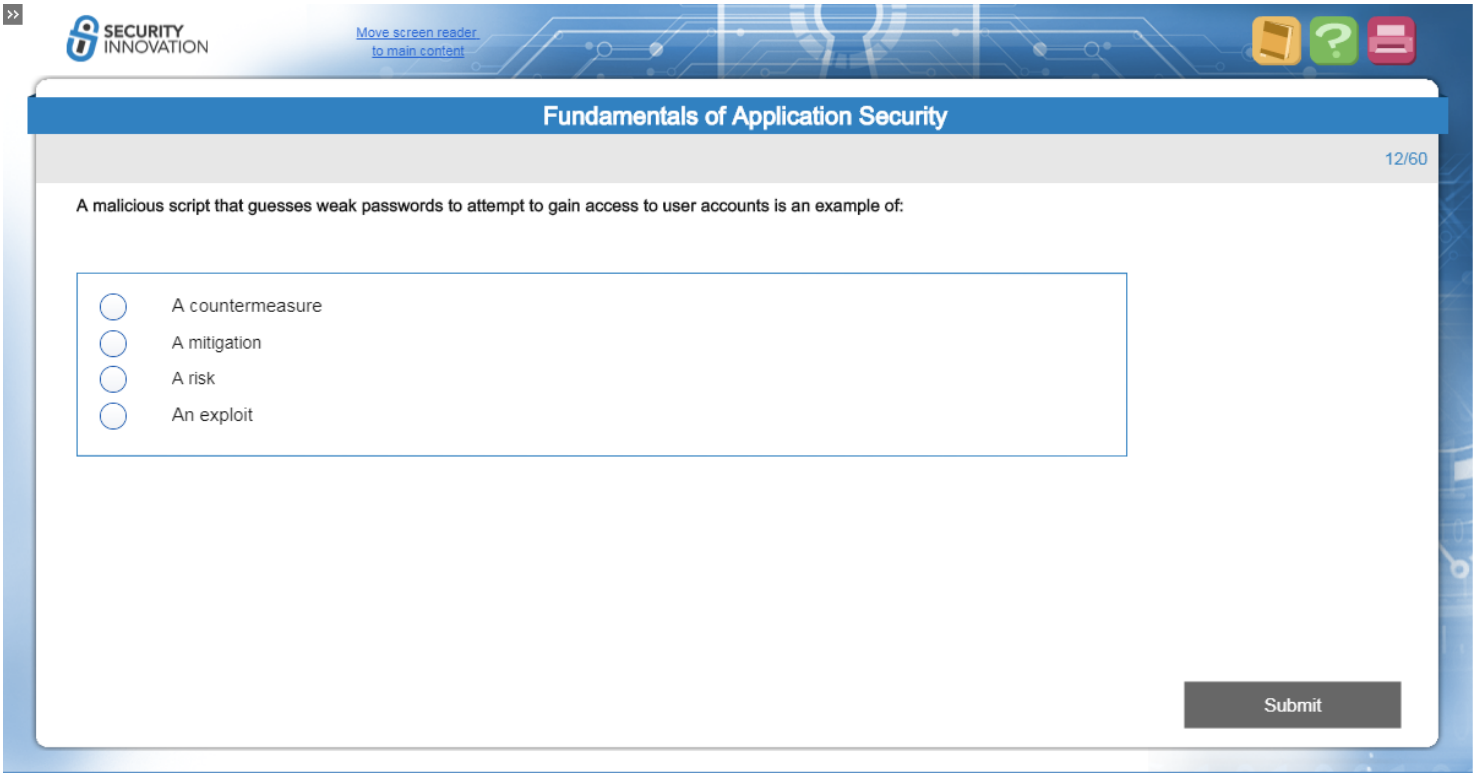
Avoid the risk

Mitigate the risk

Transfer the risk

Accept the risk

Knowledge Check



Narration

On Screen Text

A malicious script that guesses weak passwords to attempt to gain access to user accounts is an example:

A countermeasure

A mitigation

A risk

An exploit

Module Summary

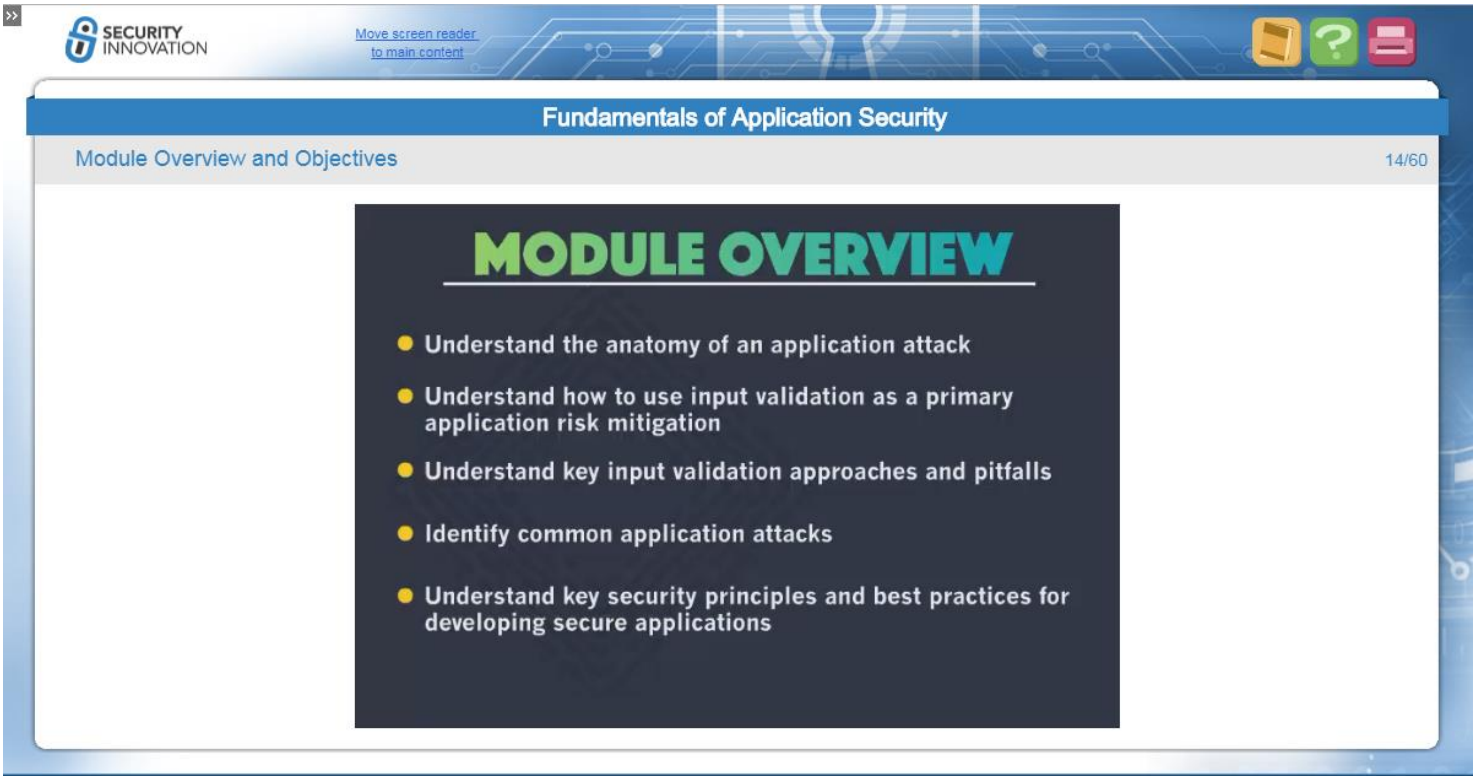
Narration

On Screen Text

Note – This slide does not contain audio. Please continue to the next section once you have finished revieweing this material

Module Summary

Module Overview and Objectives



Narration

This module will introduce you to secure application development principles.

You will learn about the anatomy of an application attack and primary countermeasures to mitigate the risk from those attacks.

You will get an introduction to the most common attacks and you will learn about key secure application development principles and best practices to simplify your secure development efforts.

After completing this module you will be able to: Understand the anatomy of an application attack,

understand how to use input validation as a primary application risk mitigation,

understand key input validation approaches and pitfalls,

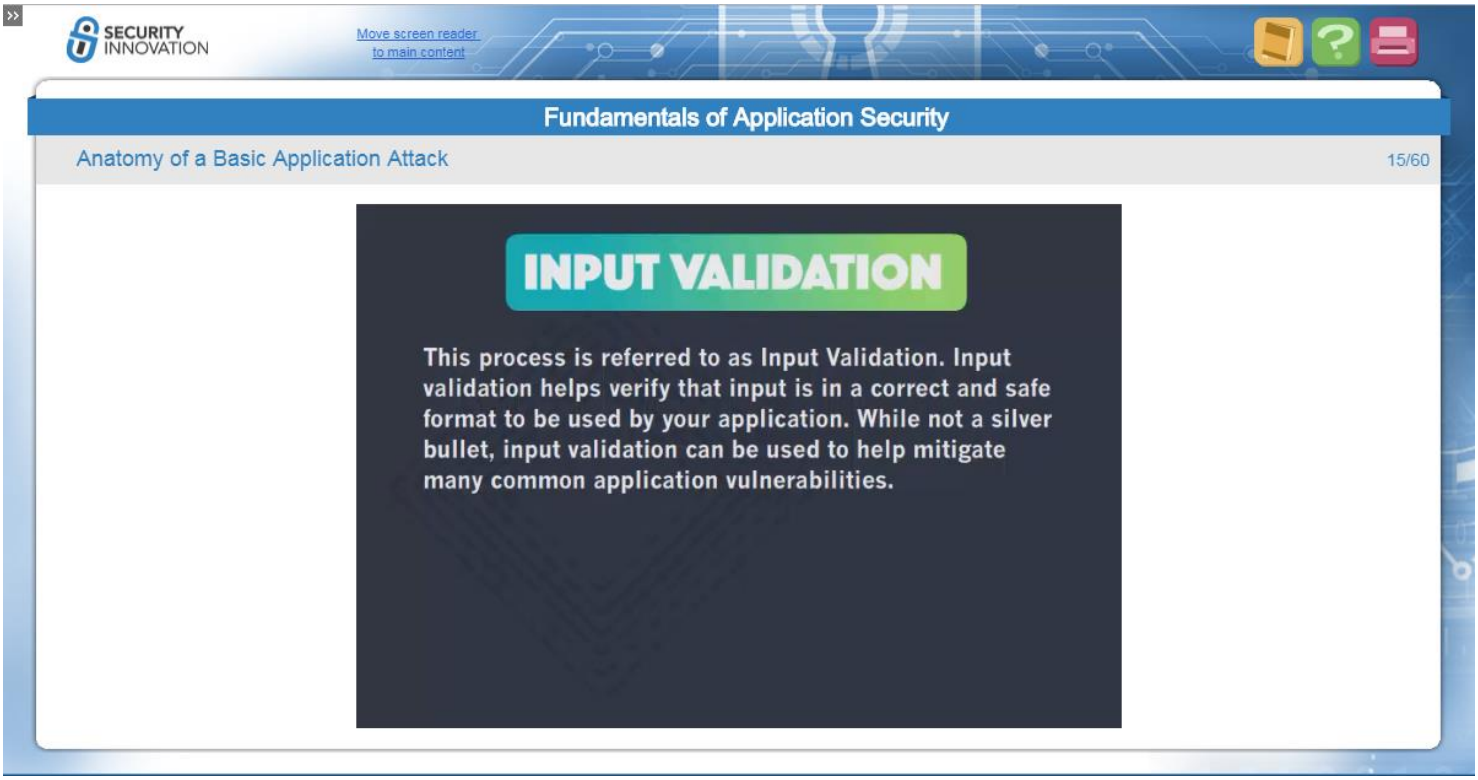
identify common application attacks,

and understand key security principles and best practices for developing secure applications.

On Screen Text

Module Overview and Objectives

Anatomy of a Basic Application Attack



Narration

In the previous module, you learned about security risk management,

the need to protect assets, application vulnerabilities, and threats to software security.

Let's now see how these all work together during an attack. By understanding the basic attack anatomy,

you will build the knowledge you'll need to understand most application attacks. Let's begin.

Any method that allows data to be input to your application is considered an interface.

Common types of user interfaces include web forms, command line parameters, and APIs.

An attacker interacts with your application in order to perform nefarious actions.

To do this, attackers identify a vulnerability they can exploit to submit malicious data to the application in order to execute an unintended command or action,

for example, an attacker could supply a malicious data that would cause your application to execute a command,

expose sensitive information, or even cause your application to crash making it unavailable to users.

As a developer, it is your job to implement counter measures to ensure your application accepts only valid data and blocks malicious data.

This process is referred to as input validation. Input validation helps verify that input is in a correct and safe format to be used by your application.

While not a silver bullet, input validation can be used to help mitigate many common application vulnerabilities.

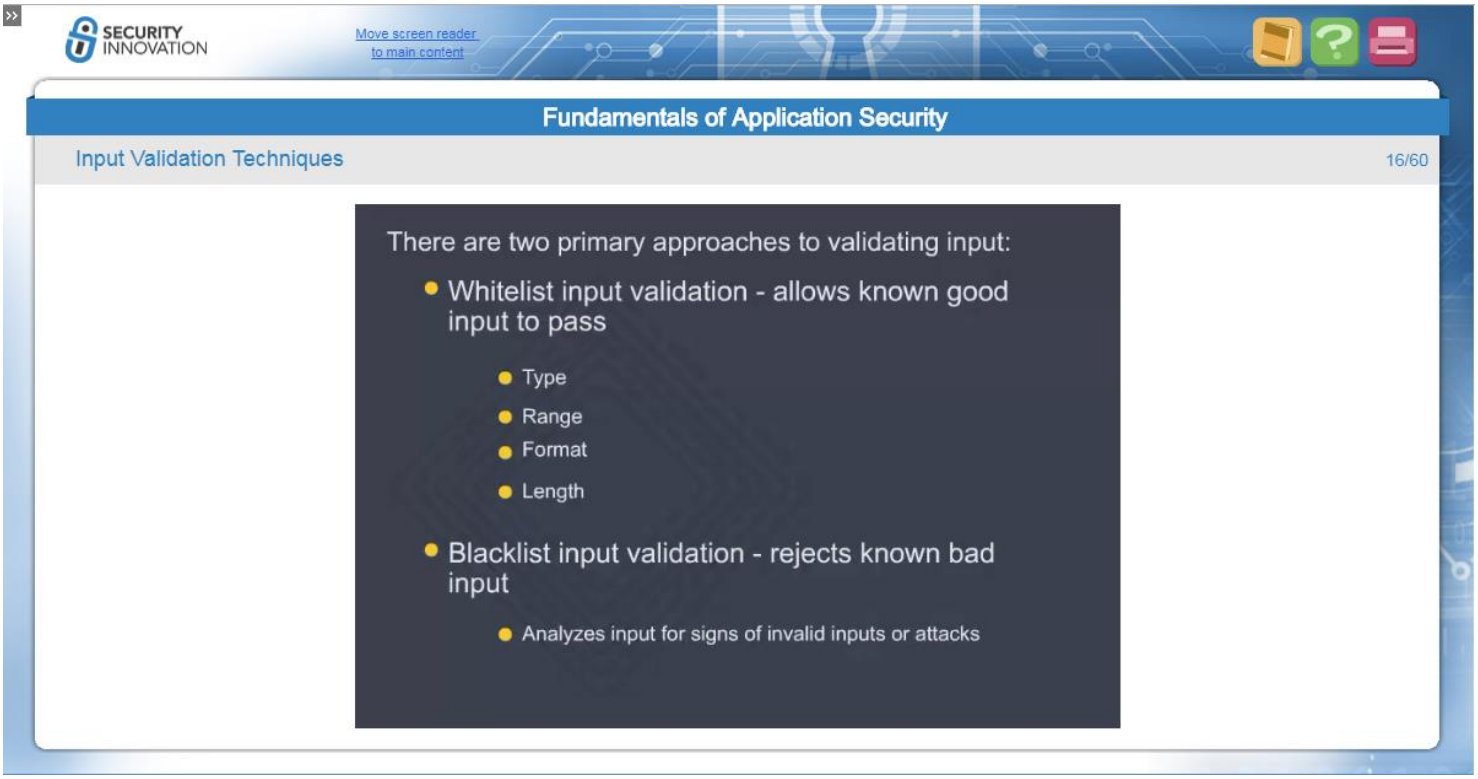
What are the primary techniques for validating data? How do you determine if data is valid or not?

This leads to the next topic in our discussion, input validation techniques.

On Screen Text

Anatomy of a Basic Application Attack

Input Validation Techniques



Narration

The two main approaches to input validation are whitelist and blacklist validation.

Whitelisting allows known good input to pass, and rejects everything else.

It robustly validates input for type, range, format, and length.

Blacklisting rejects only known bad input, and allows everything else.

Its inherent weakness is that the set of all bad inputs is potentially infinite and cannot be completely detected,

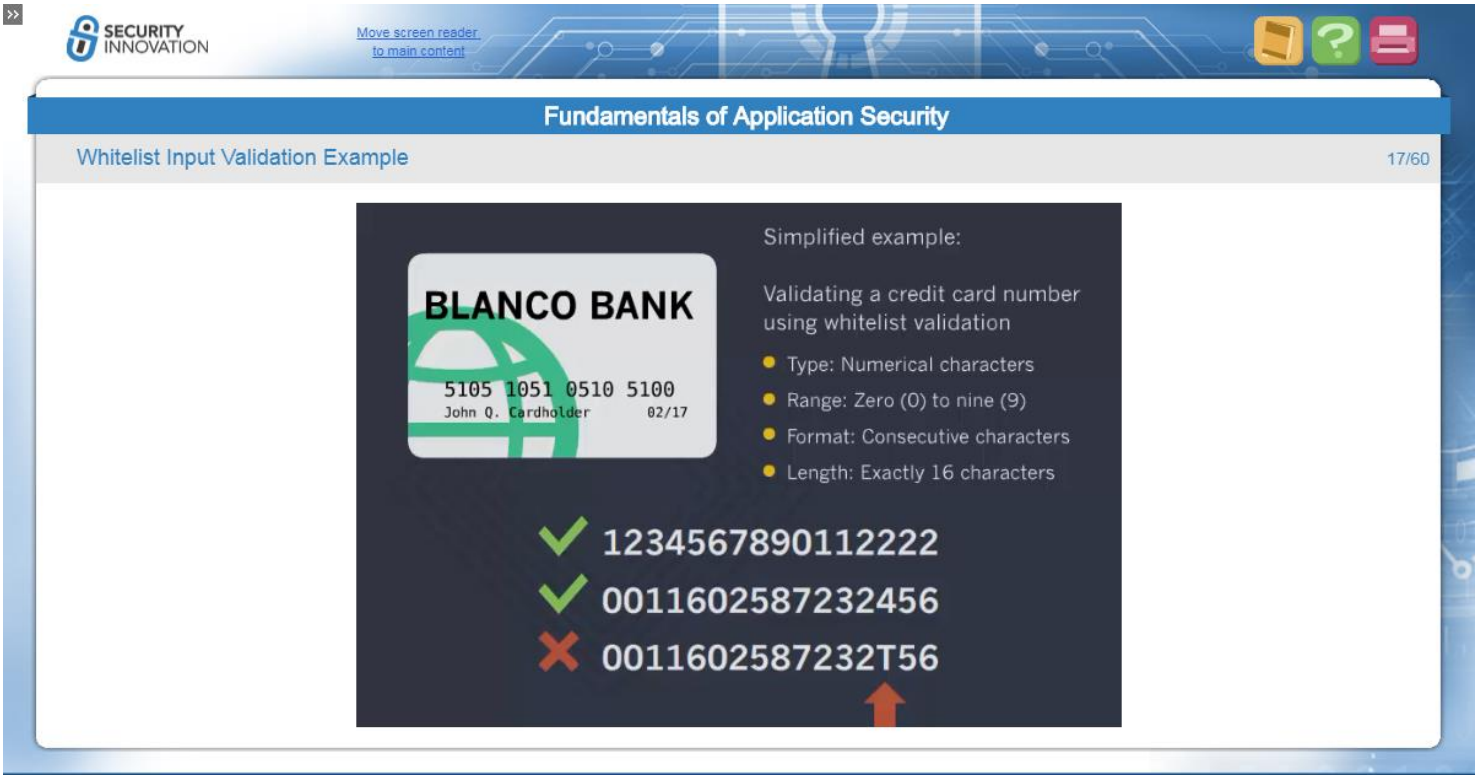
whereas the whitelist approach models a finite set and is easier to implement and less prone to error.

Therefore, always use whitelist validation, and use blacklisting only as a supplementary measure.

On Screen Text

Input Validation Techniques

Whitelist Input Validation Example



Narration

Let’s look at a simplified example of validating a credit card number using the whitelist validation approach.

As we learned previously, the whitelist approach first defines expected types, ranges, formats and lengths and validates input against that criteria.

In the example of a credit card, we would expect a type of all numerical characters.

The numerical characters would have a range from zero to nine.

The format and length of the input would be 16 consecutive characters.

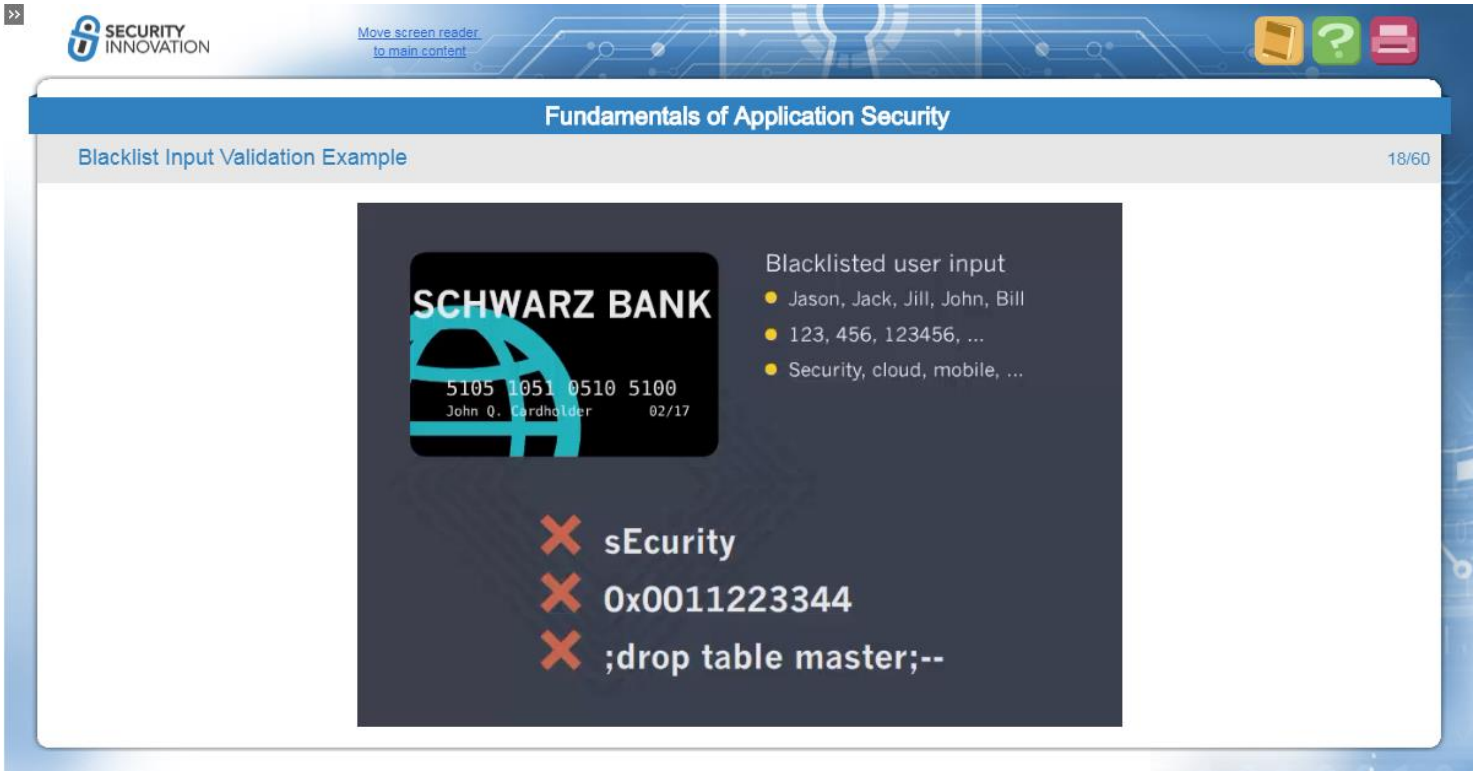
Here’s an example of a fictitious credit card number that matches our input validation type, range, format and length criteria.

Here’s another example. Note that it violates our range requirement that the range of each character be from zero to nine.

On Screen Text

Whitelist Input Validation Example

Blacklist Input Validation Example



Narration

To use the blacklist technique to validate a credit card, you must first define a list of inputs that are not valid credit card numbers.

In our example shown here, inputs like Jason, Jack, Jill, 123, or mobile are clearly not valid credit card numbers.

Any input that is not in our pre-defined blacklist is considered a valid input.

It is easy to see that an attacker could bypass this blacklist technique.

An attacker could submit invalid credit card input that is not on the blacklist,

and trick the input validation implementation into thinking a valid credit card input was submitted.

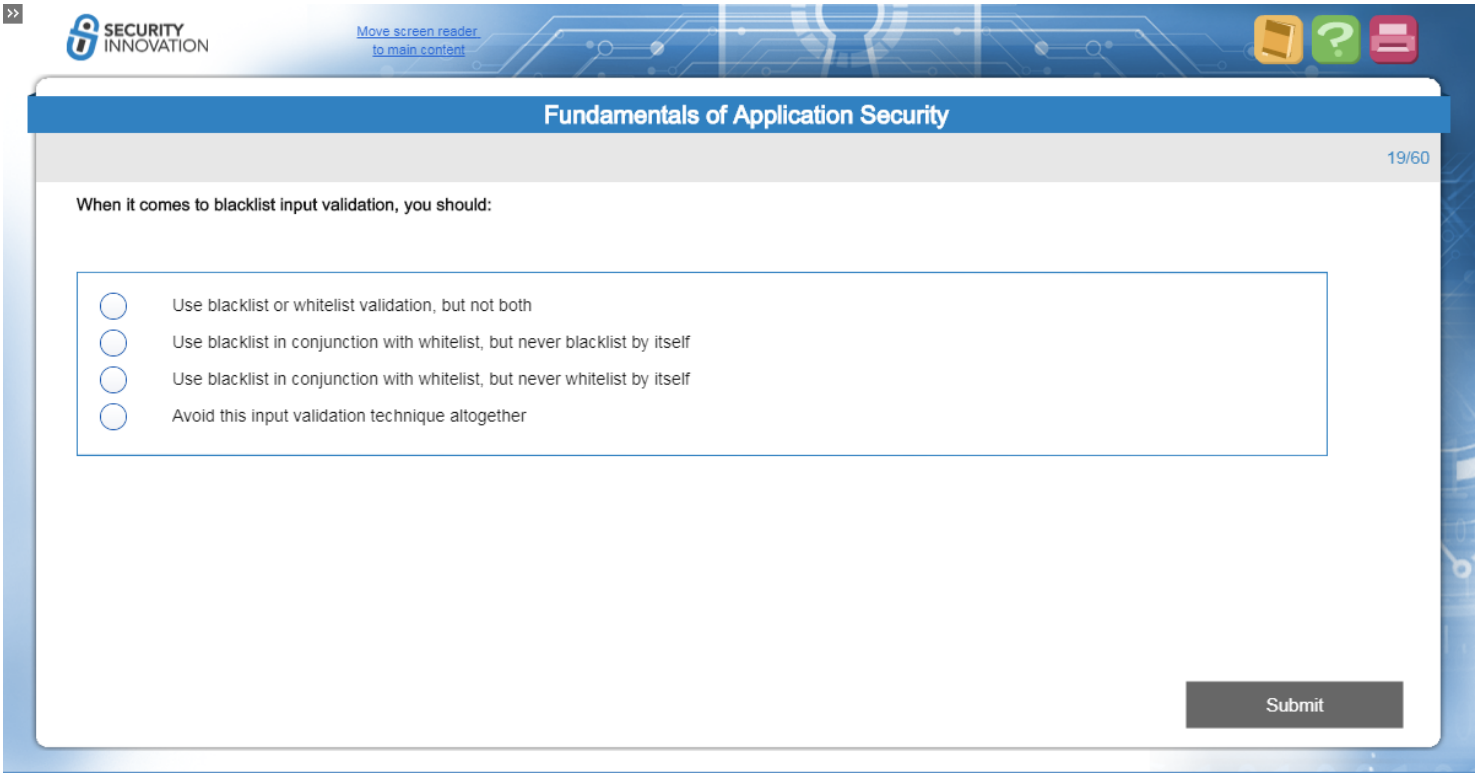
We could add these invalid inputs to our blacklist; however our blacklist set would still be incomplete.

In fact, it would be impossible to create a complete blacklist set because the set of invalid credit card numbers is infinite.

On Screen Text

Blacklist Input Validation Example

Knowledge Check



Narration

On Screen Text

When it comes to blacklist input validation, you should:

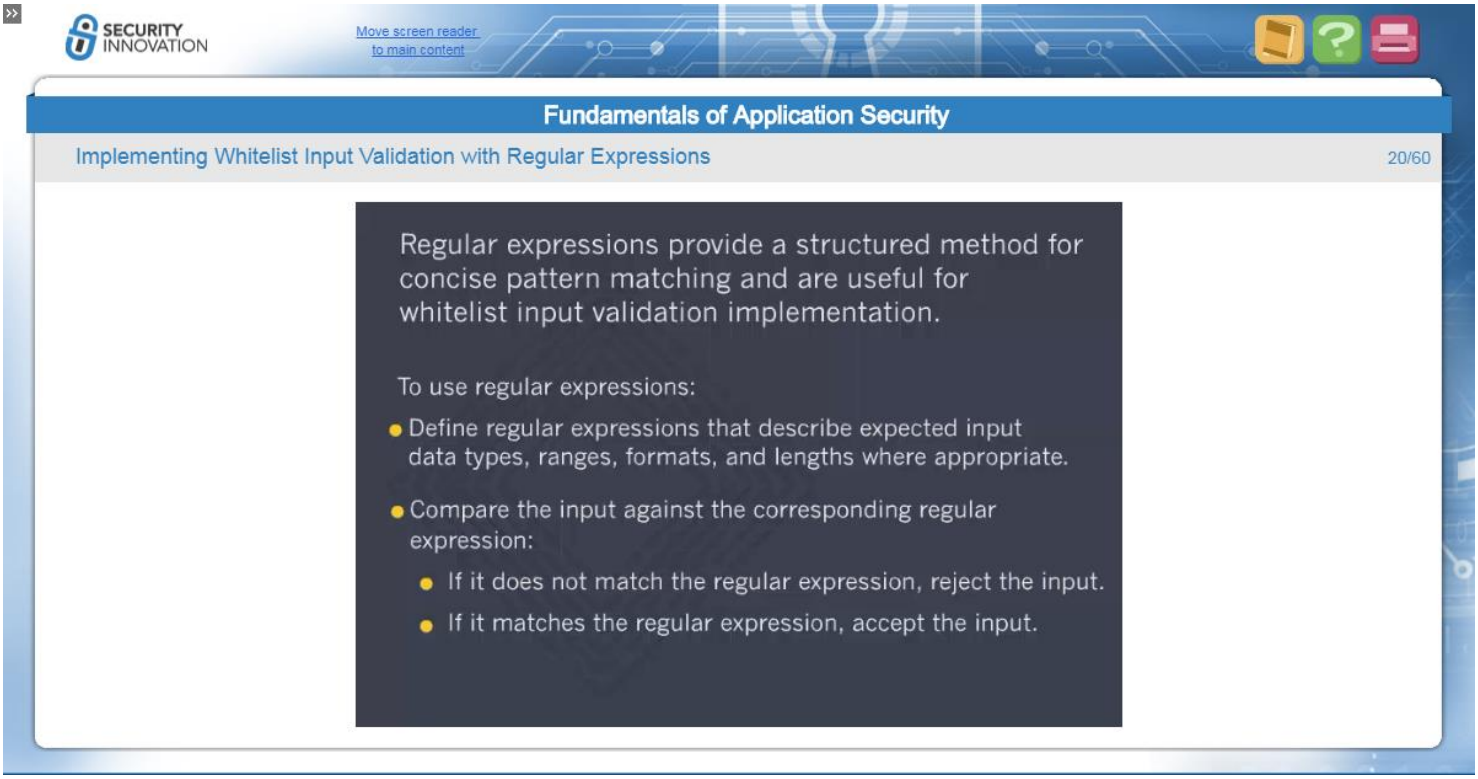
Use blacklist or whitelist validation, but not both

Use blacklist in conjunction with whitelist, but never blacklist by itself

Use blacklist in conjunction with whitelist, but never whitelist by itself

Avoid this input validation technique altogether

Implementing Whitelist Input Validation with Regular Expressions



Narration

Regular expressions provide a structured method for concise pattern matching that you can use to implement whitelist validation.

Several regular expression libraries are available, and many programming languages have built-in regular expression support.

For example, the .NET Framework and Java programming languages natively support regular expressions.

To use regular expressions, you first define regular expressions that describe expected input data types, ranges, formats, and lengths where appropriate.

You then compare the input received by your application against the corresponding regular expression.

If the input does not match the regular expression, it should be considered bad and rejected.

If the input matches the regular expression, then it is considered safe and can be used.

On Screen Text

Implementing Whitelist Input Validation with Regular Expressions

Whitelist Input Validation Example with Regular Expressions

Narration

Here's our credit card example using whitelist validation with regular expressions.

As indicated before, many modern programming languages have built-in support for regular expressions.

Here is a regular expression that implements our whitelist validation criteria.

The first criteria is that the type of the data must be all numerical characters.

A second criteria is that the characters must have a range from zero to nine.

This is implemented using the back-slash d regular expression fragment.

The last two criteria are that the format of the input must be consecutive characters and the length of the input must be exactly sixteen characters.

This is implemented using the highlighted regular expression fragments.

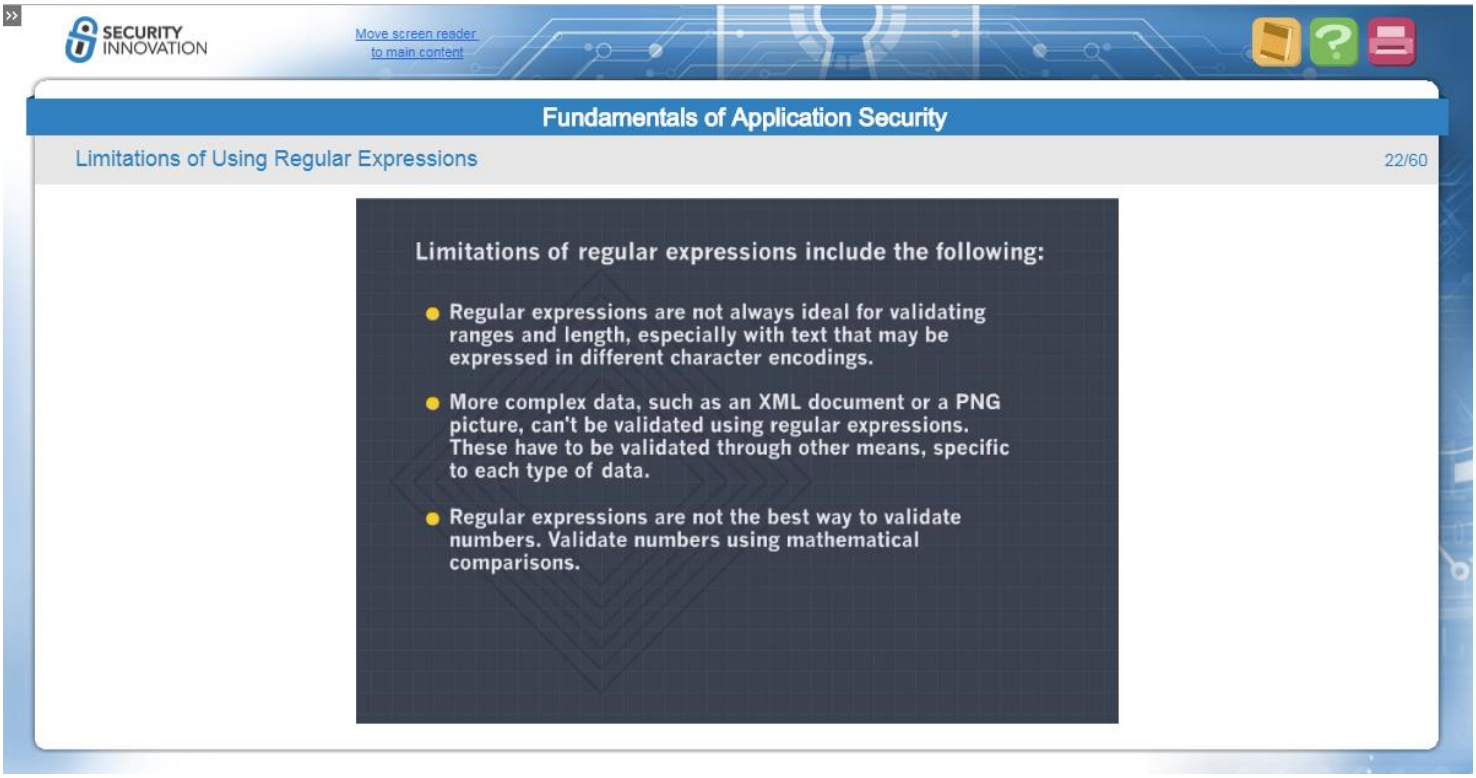
As you can see, it is fairly easy to implement whitelist validation with regular expressions.

However, use of regular expressions does have limitations, which will be discussed next.

On Screen Text

Whitelist Input Validation Example with Regular Expressions

Limitations of Using Regular Expressions



Narration

In practice, regular expressions are a useful way to implement white-list validation,

and can be used to validate the format for most text input. However, they are not always ideal to validate range and length,

especially with text that may be expressed in different character encodings.

Further, regular expressions are unable to easily validate more complex data like an XML document or an image file.

Finally, regular expressions are not the best way to validate numbers.

Numbers should be validated by using mathematical comparisons instead.

On Screen Text

Limitations of Using Regular Expressions

Common Sources of Untrusted Data

Narration

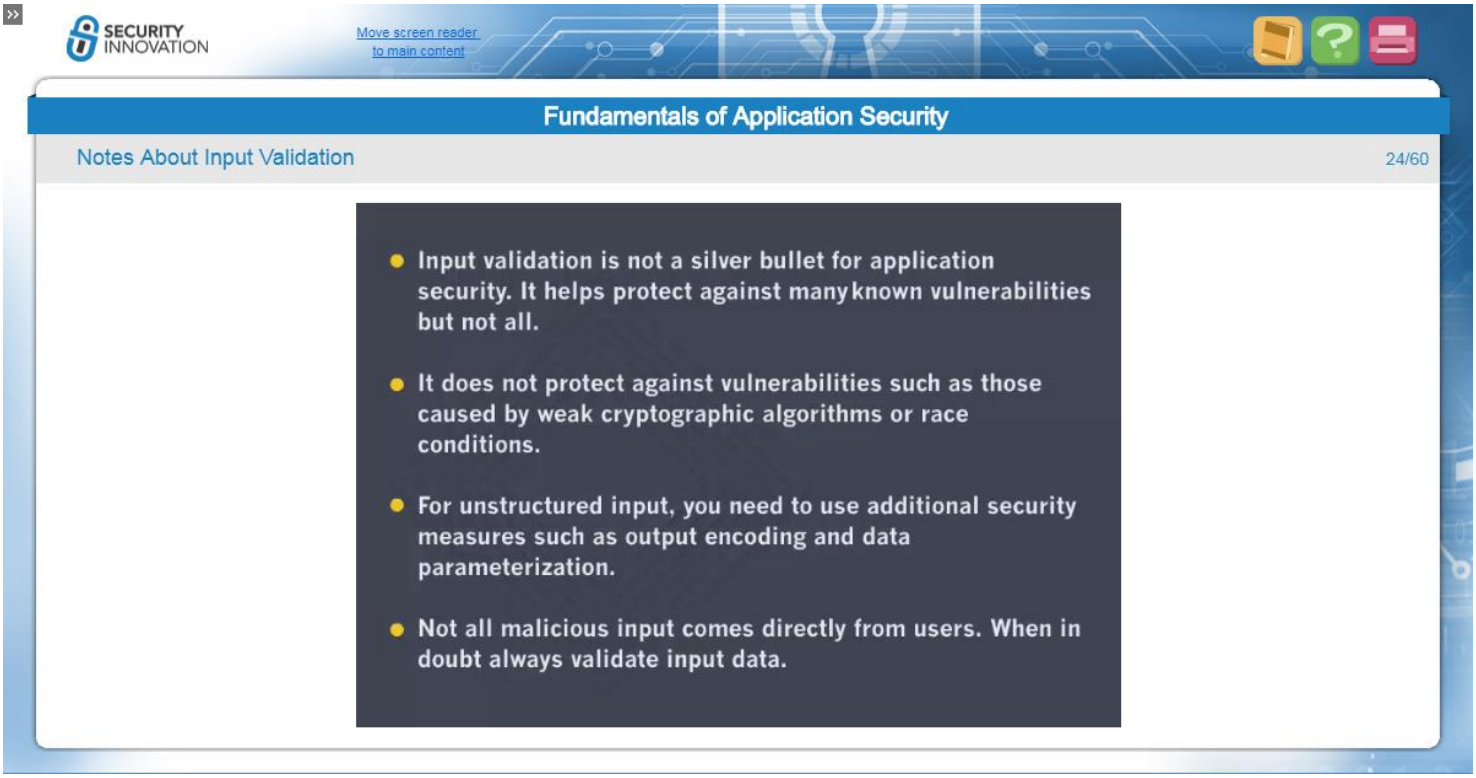
Untrusted data can enter your application through any interface, such as those listed on-screen.

When in doubt, consider all input to be untrusted and properly validate that input before using it in your application.

On Screen Text

Common Sources of Untrusted Data

Notes About Input Validation



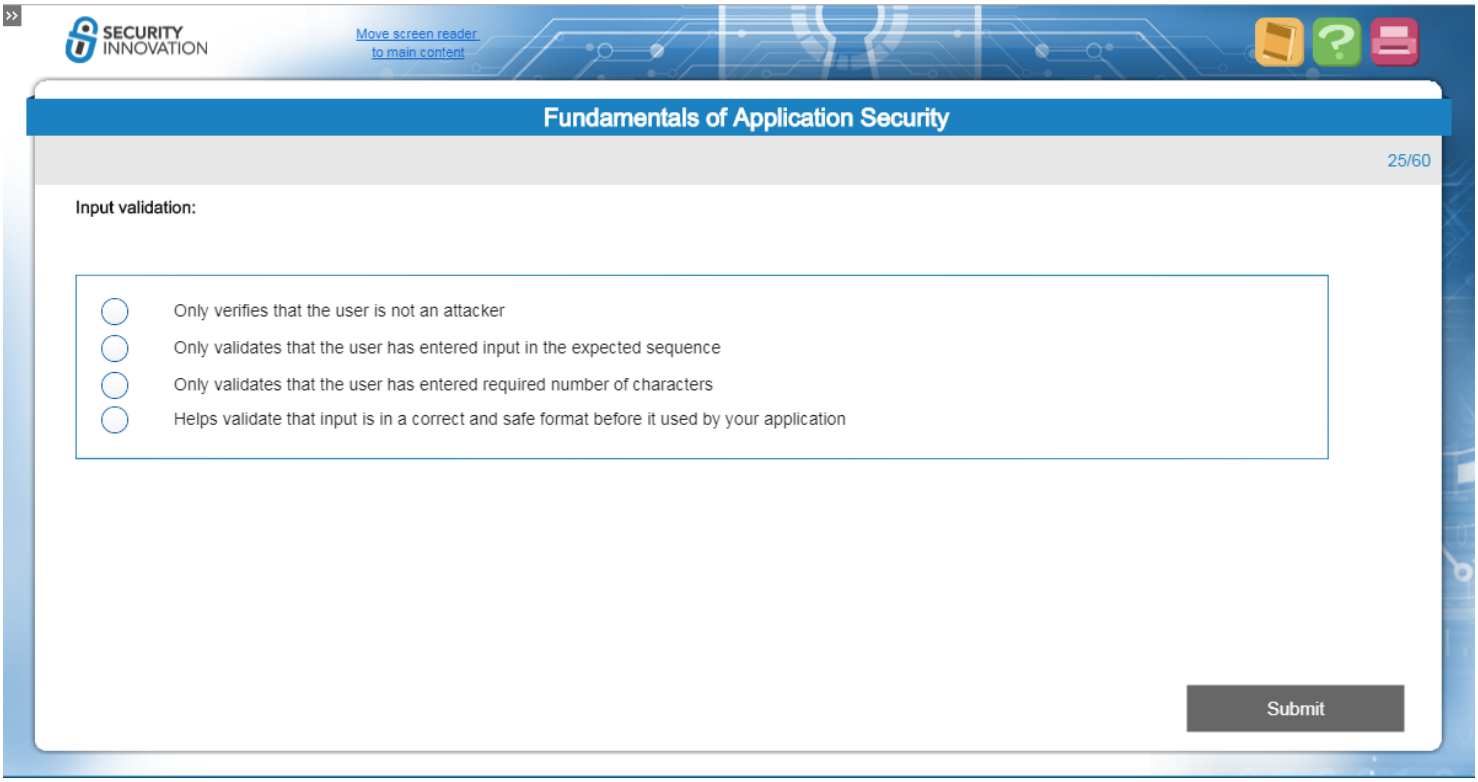
Narration

While input validation is a very effective control for mitigating application security risk,
it should not be considered a silver bullet. It helps protect against many known vulnerabilities, but not all.
For example, some application vulnerabilities are not triggered due to malicious input,
such as use of weak encryption algorithms, or timing-based vulnerabilities that allow an attacker to act upon a resource ahead of your application.
In addition, using input validation is only effective in cases where the input has a defined structure,
such as phone numbers, dates and credit card numbers. For unstructured input,
such as a user comment, you need to use additional security measures to protect your application,
such as output encoding and data parameterization, which will be discussed later in this course.
Finally, note that not all malicious input comes from users.
Malicious data can come from compromised libraries or outside services that your application uses,
or even dependencies, such as a third-party data repository.
When in doubt as to whether data is trusted or not,
always err on the side of caution and validate input data.

On Screen Text

Notes About Input Validation

Knowledge Check



Narration

On Screen Text

Input validation:

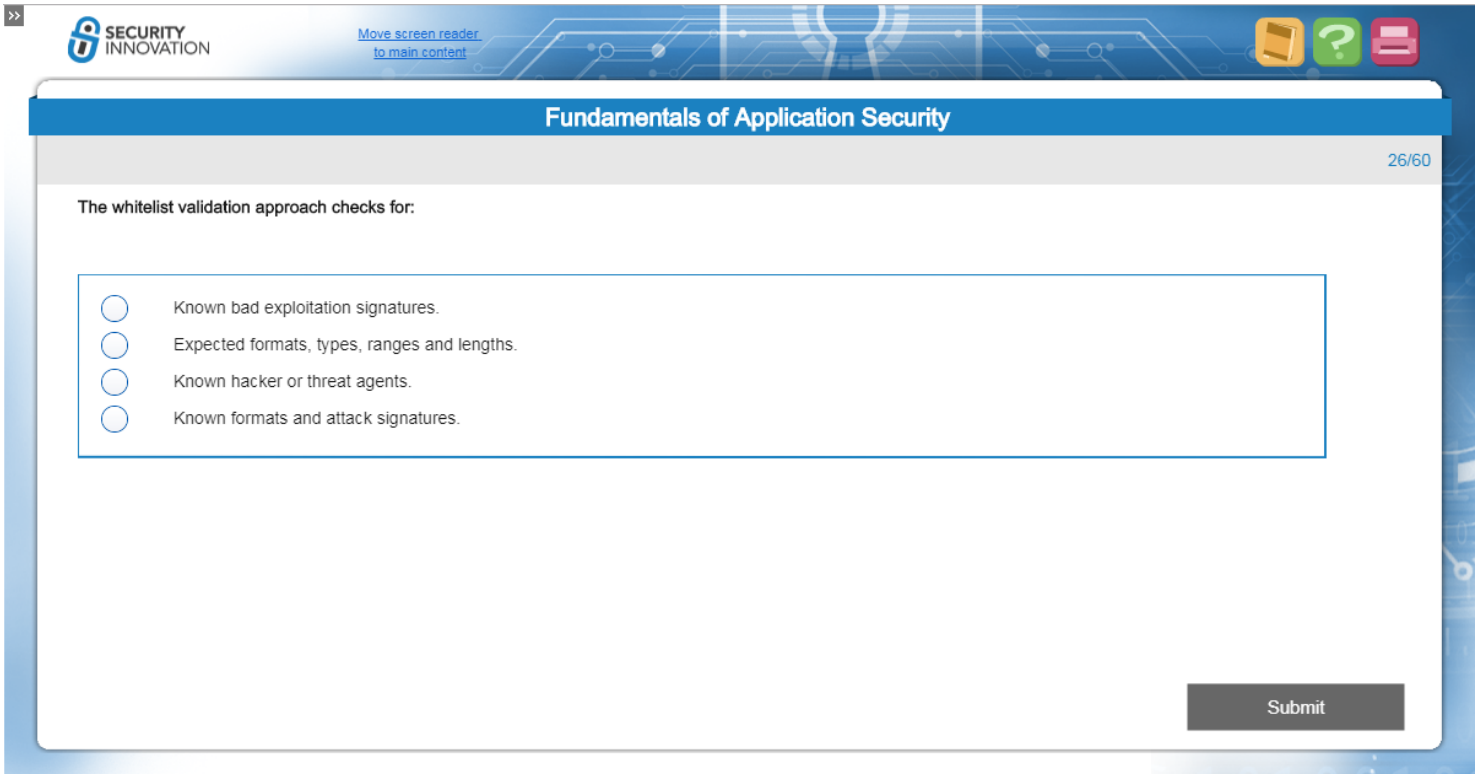
Only verifies that the user is not an attacker

Only validates that the user has entered input in the expected sequence

Only validates that the user has entered required number of the characters

Helps validate that input is in a correct and safe format before it used by your application

Knowledge Check



Narration

On Screen Text

The whitelist validation approach checks for:

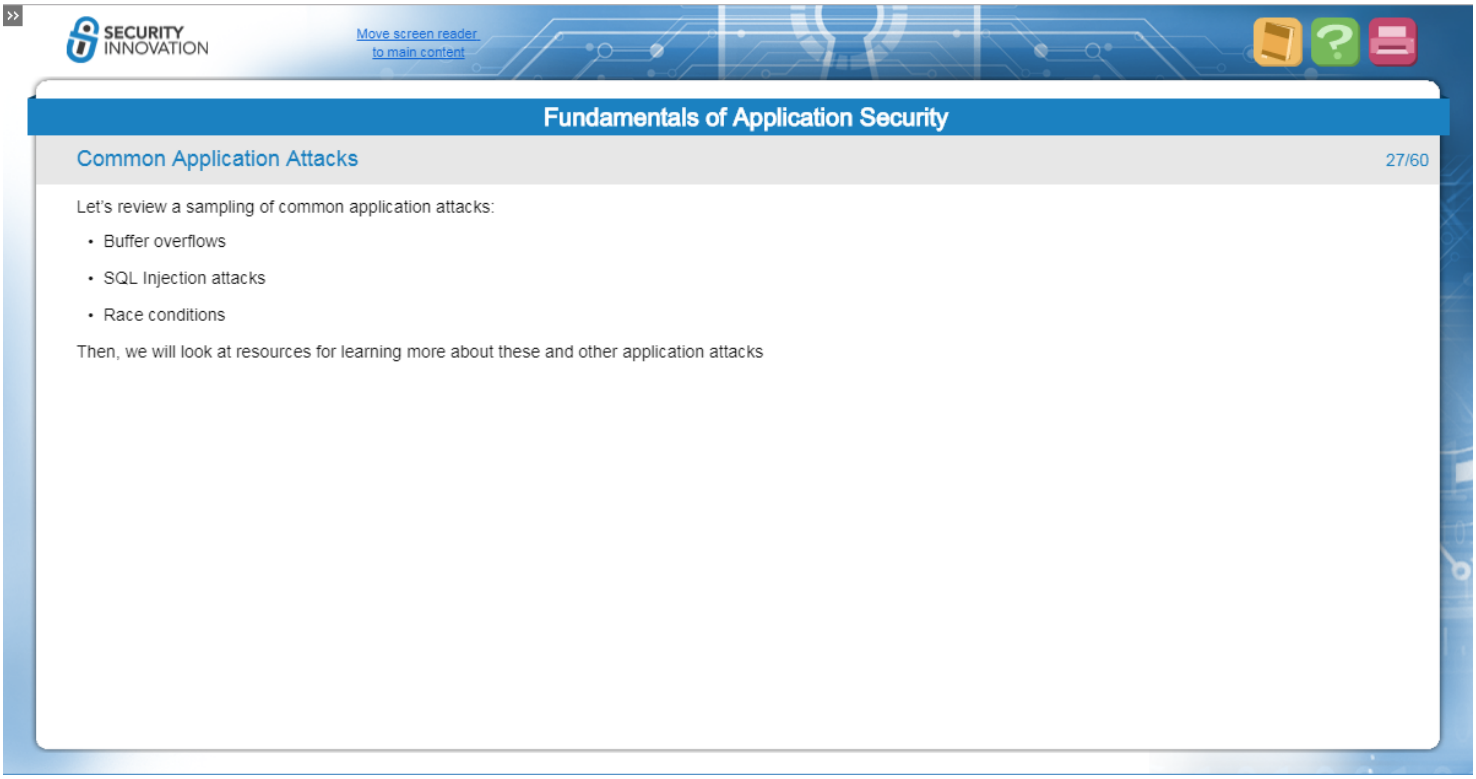
Known formats and attack signatures.

Known bad exploitation signatures.

Known hacker or threat agents.

Expected formats, types, ranges and lengths

Common Application Attacks



Narration

Now that you're familiar with the basic anatomy of an application attack
and how input validation plays a role in significantly reducing the risk to your application from attack,
let's turn our attention to looking at the common real-world attacks listed on-screen.
This list is a very small subset of the total application attacks in the wild.
The goal here is to simply become familiar with common application attacks and how attackers use them.
We'll then discuss resources that are available for learning more about these and other application attacks.

On Screen Text

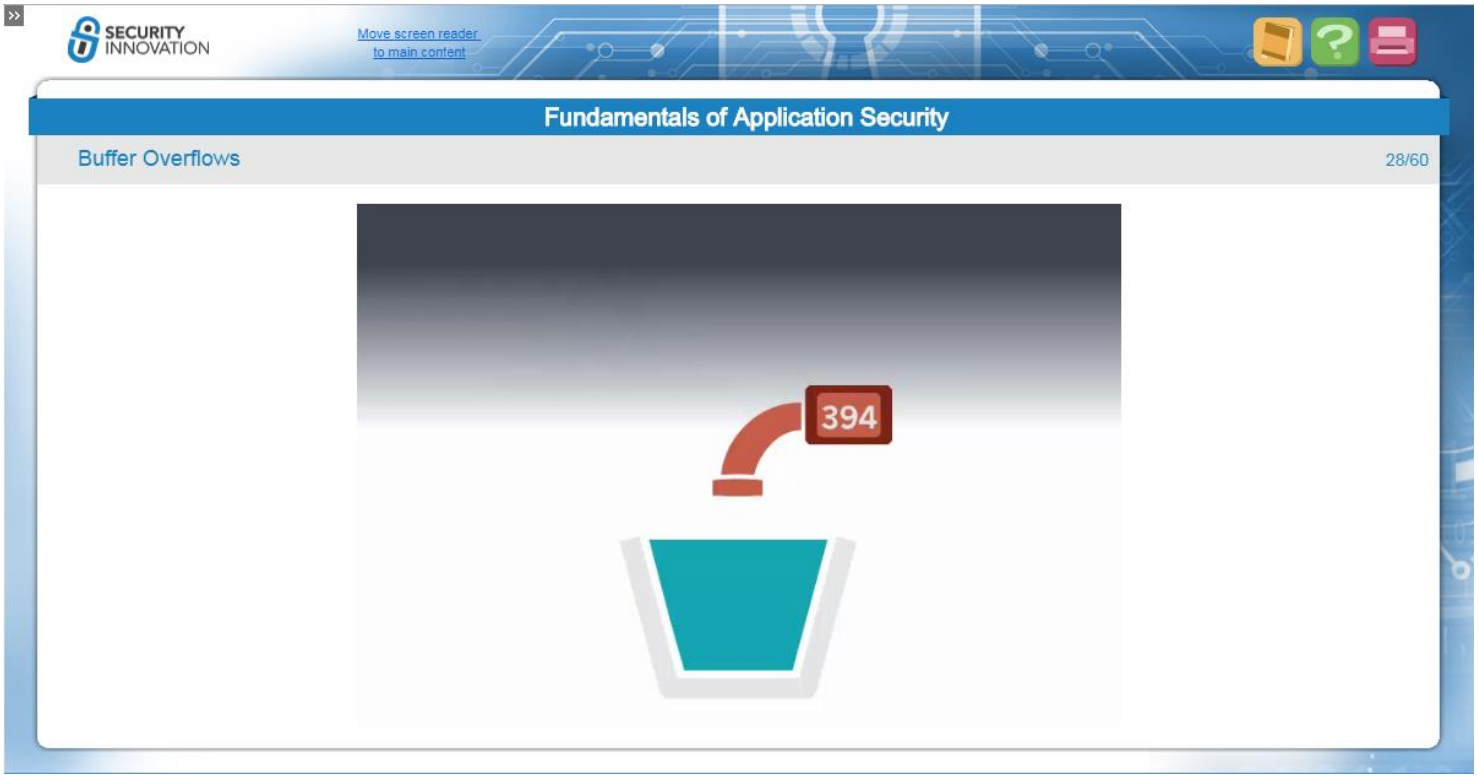
Common Application Attacks

Let's review a sampling of common application attacks:

- Buffer overflows
- SQL Injection attacks
- Race conditions

Then, we will look at resources for learning more about these and other application attacks

Buffer Overflows



Narration

Buffer overflows occur in applications where more data is written into a buffer than that buffer has been allocated to handle.

The result is that memory adjacent to the buffer gets overwritten, which typically causes the application to behave incorrectly or crash.

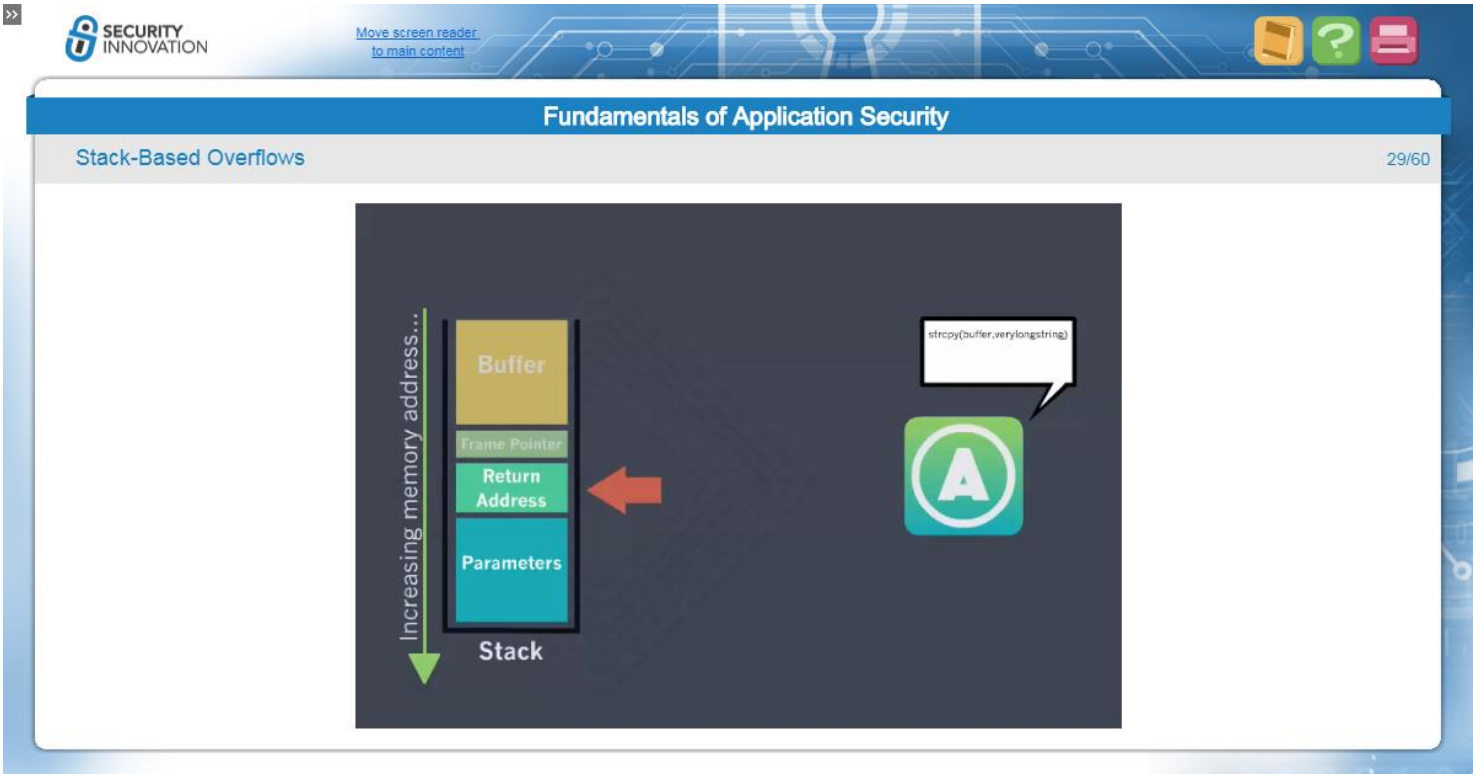
Buffer overflows can be exploited whenever an application attempts to write untrusted data into a fixed-length buffer

without first validating the input or accurately calculating the buffer size.

On Screen Text

Buffer Overflows

Stack-Based Overflows



Narration

Stack-based buffer overflows are caused by the fact that data is written into a local buffer without proper bounds checking.

When a function is called, a stack frame is pushed onto the program stack.

The program stack consists of the function's parameters, the return address, the frame pointer, and the function's local variables.

When a local buffer is overflowed, following memory locations on the stack including the return address of the function end up being overwritten by user data.

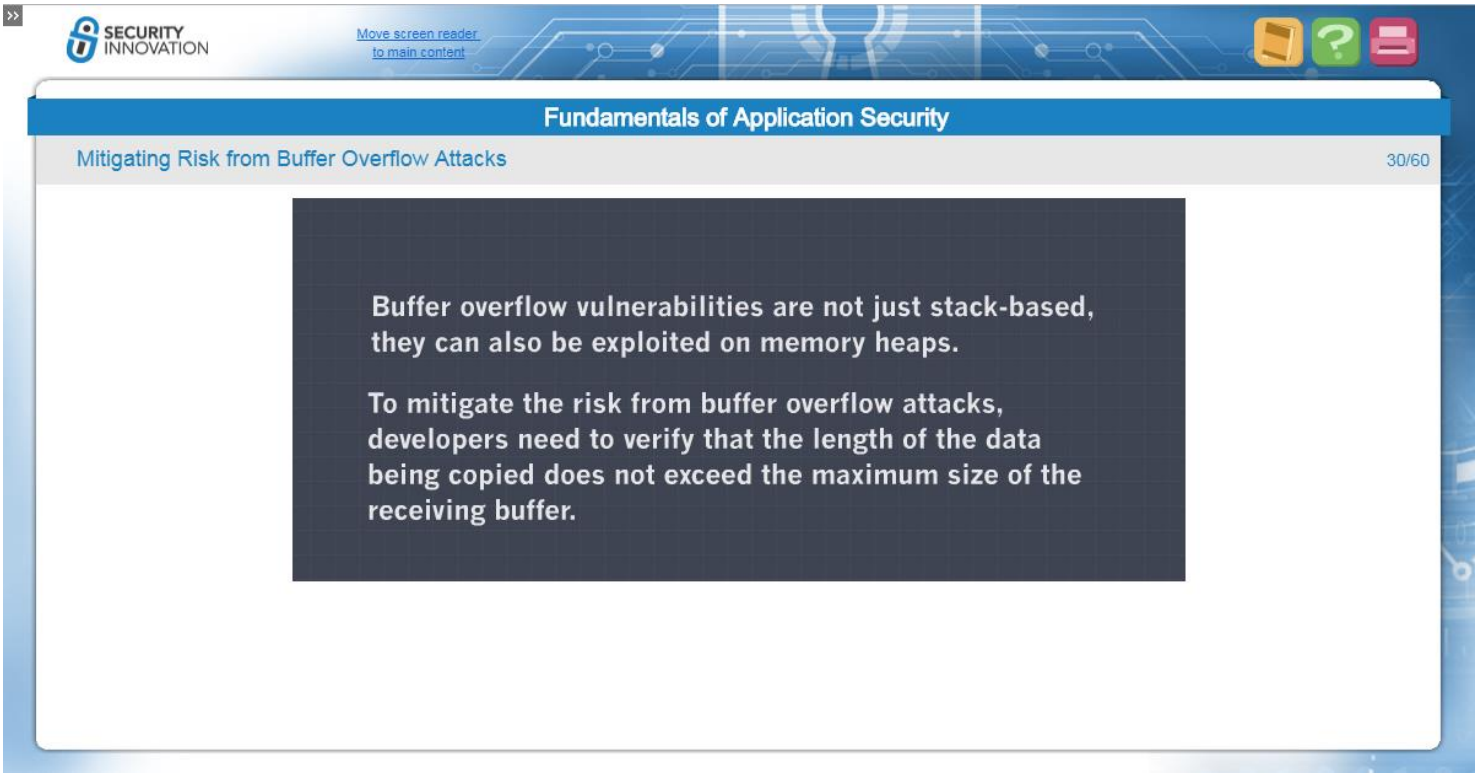
This poses a security issue because the return address determines the location of the next instruction to be executed after the function returns.

This means that if an attacker can control the value of a function's return address he can potentially control execution of the vulnerable program.

On Screen Text

Stack-Based Overflows

Mitigating Risk from Buffer Overflow Attacks



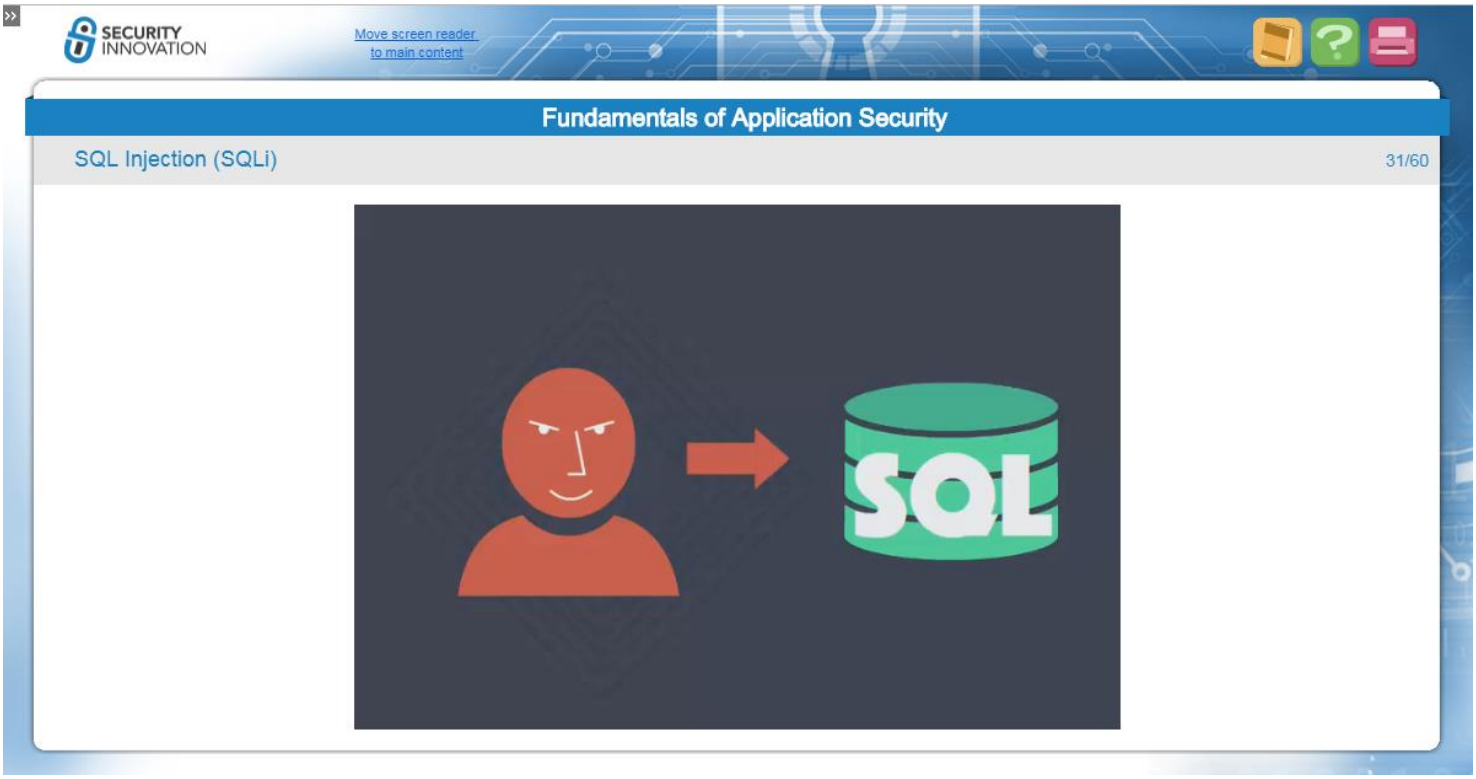
Narration

Keep in mind that buffer overflow vulnerabilities are not just stack-based,
they can also be exploited on memory heaps.
To mitigate the risk from buffer overflow attacks,
developers need to implement proper input validation in their applications.
Specifically, they need to verify that the length of data being copied into a static buffer
does not exceed the maximum size of that static buffer.
If it does, then there may be an exploitable buffer overflow vulnerability in their application code.

On Screen Text

Mitigating Risk from Buffer Overflow Attacks

SQL Injection (SQLi)



Narration

The next attack we will discuss is SQL injection, or SQLi, which is a technique that allows attackers to inject SQL commands into dynamic SQL statements.

These dynamic SQL statements are then executed by backend databases, enabling attackers to potentially execute commands in the context of the database.

SQL injection vulnerabilities are yet another example of attacks that are possible due to the failure of applications to validate un-trusted data.

SQL Injections are one of the most common attacks today. One reason is the ubiquity of SQL Data Stores, which increases the SQL injection attack surface.

Another reason may be that many applications are moving toward the Web,

which makes them more accessible to attack than traditional applications that reside within restricted internal networks.

Finally, nearly all modern day database management systems support SQL.

This means that malicious users need only learn the SQL language once

and then they are able to apply those attack skills to virtually any database management system,

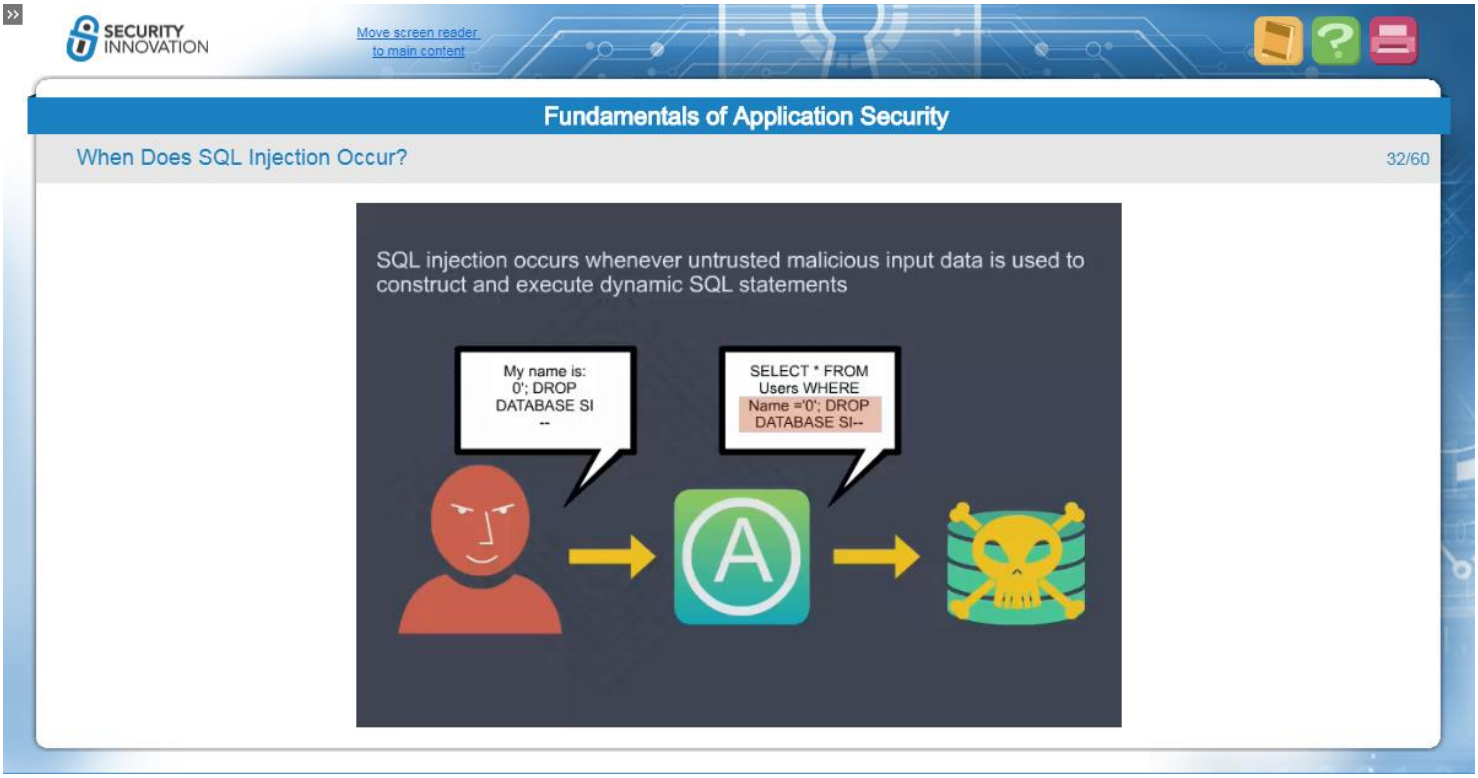
regardless of the vendor platform (Microsoft, Oracle, IBM, MySQL, and so on).

Let’s take a closer look at this attack pattern to review our understanding of this type of attack.

On Screen Text

SQL Injection (SQLi)

When Does SQL Injection Occur?



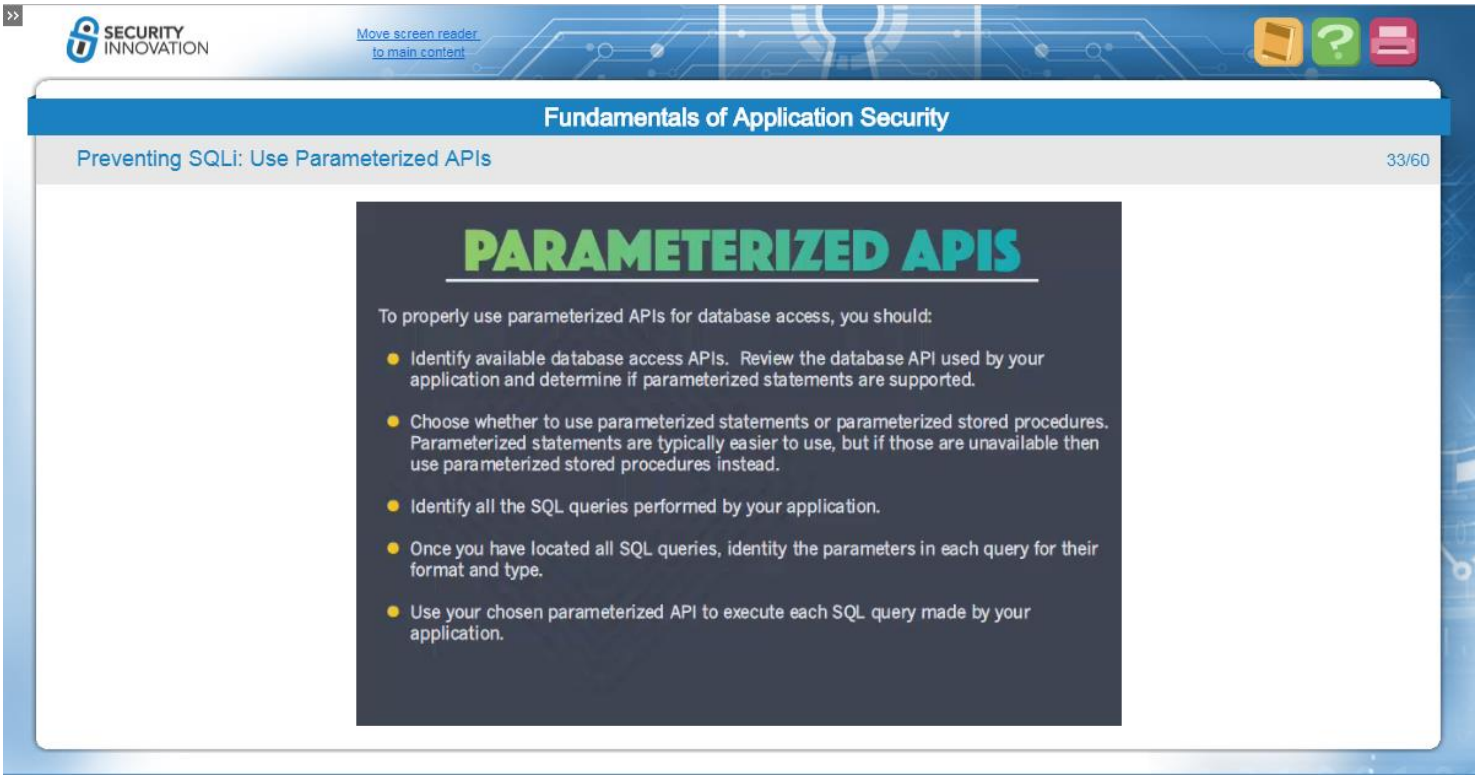
Narration

SQL injection vulnerabilities in code are created whenever untrusted input data is used to construct and execute dynamic SQL statements.
Let's examine a common scenario where an application reads data from a user,
constructs a dynamic SQL statement based on that data, and then forwards that dynamic SQL statement to a backend database for execution.
First, a malicious user locates an input into the application that is used to create a dynamic SQL statement.
For example, an input field where the user is supposed to enter their user name.
The malicious user instead enters "0;' DROP DATABASE SI",
which contains malicious SQL commands to drop the database called SI.
The application reads that input, fails to validate it and constructs a dynamic SQL statement.
In actuality, two dynamic SQL statements were created.
The first is to select everything from the Users table where the Name column equals 0,
and the second is a malicious command which instructs the executing database engine to drop or delete the database called SI.
The application forwards the SQL statements to the database,
which executes both commands. As a result, the database called SI is dropped.

On Screen Text

When Does SQL Injection Occur?

Preventing SQLi: Use Parameterized APIs



Narration

While input validation can be used to mitigate the risks from many attacks,

there may be times where input validation alone may not be sufficient.

In the case of SQL injection, it may be hard to validate input where the structure of the input is unknown or undefined, such as in a product review.

As a good security practice, type-safe SQL command parameters should be used in all situations where any variables are included in a SQL statement.

Type-safe SQL command parameters indicate to the database engine that the data defined in certain parameters are values only and should never be executed.

Marking input data as non-executable neutralizes any SQL commands that a malicious user may try to inject

and is extremely effective for mitigating risk from SQL injection attacks.

To properly use parameterized APIs for database access, you should:

Identify available database access APIs.

Review the database API used by your application and determine if parameterized statements are supported.

Choose whether to use parameterized statements or parameterized stored procedures.

Parameterized statements are typically easier to use,

but if those are unavailable then use parameterized stored procedures instead.

Identify all the SQL queries performed by your application.

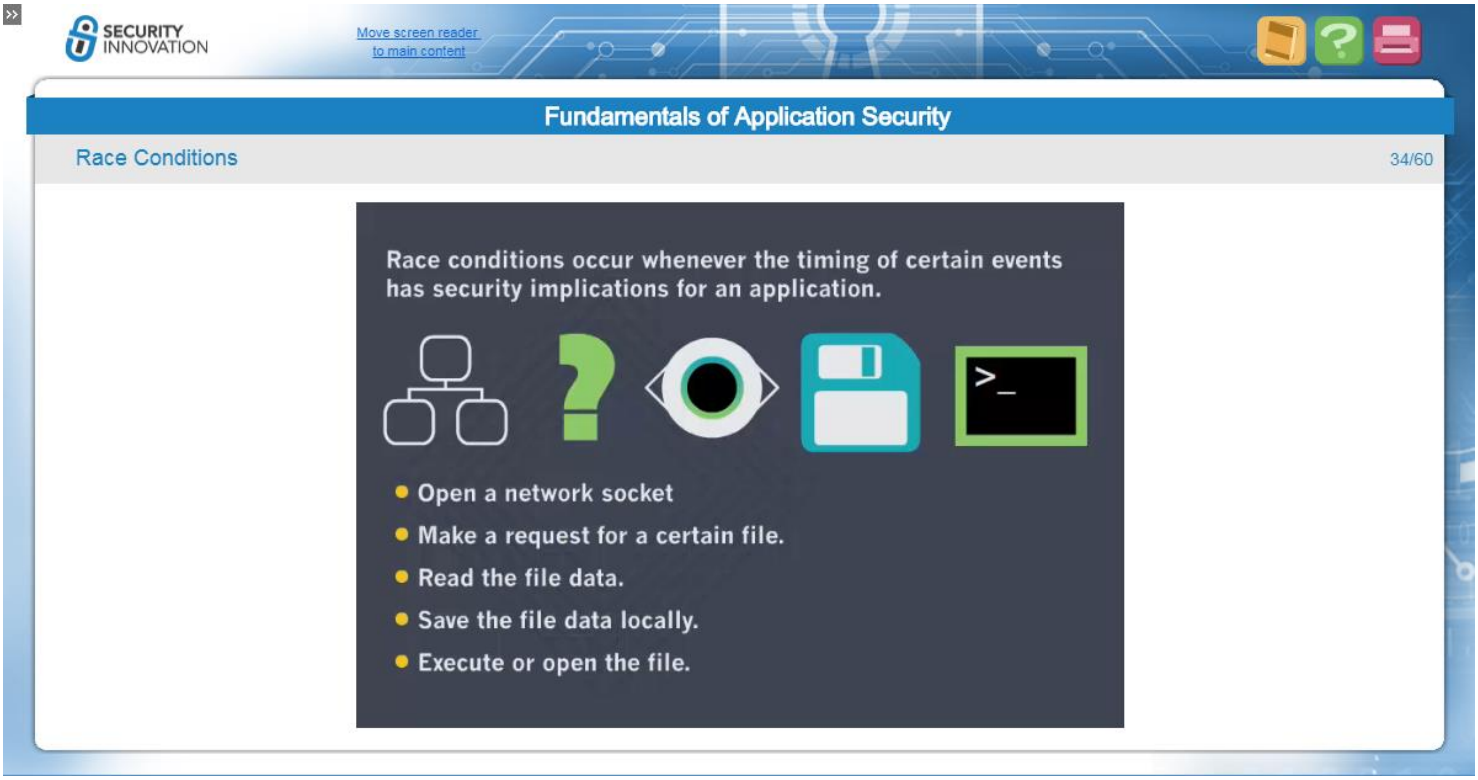
Once you have located all SQL queries, identity the parameters in each query for their format and type.

Use your chosen parameterized API to execute each SQL query made by your application.

On Screen Text

Preventing SQLi: Use Parameterized APIs

Race Conditions



Narration

The next vulnerability we'll discuss is race conditions.

All applications can be decomposed into a series of steps that need to be executed in some order to perform a given function.

For instance, an application may be programmed to download and execute a file from the Internet,

in which case the steps to fulfill that function could be decomposed into the following steps:

Open a network socket.

Make a request for a certain file.

Read the file data.

Save the file data locally.

Execute or open the file.

When the timing of certain application events has security implications,

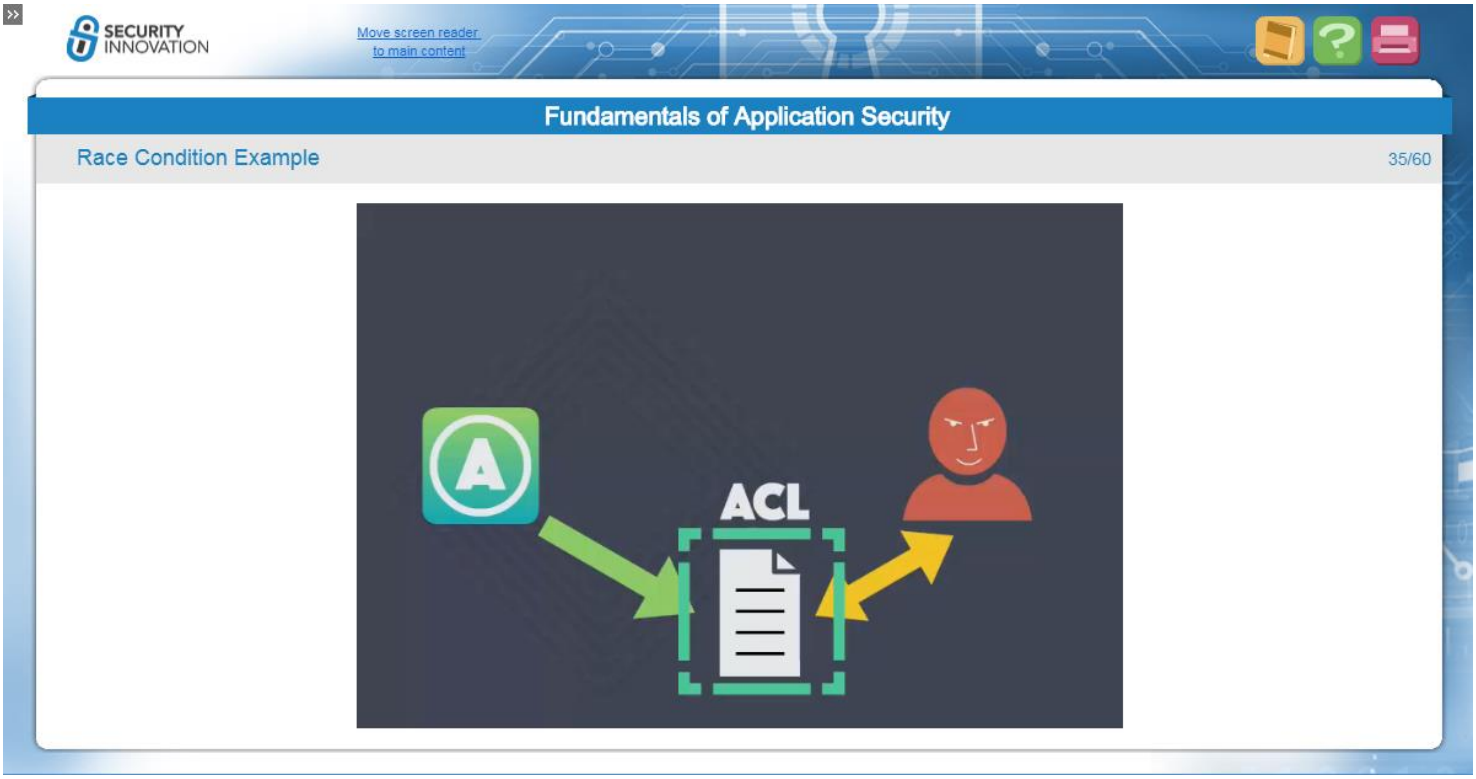
the possibility for an application security vulnerability known as "race conditions" is created.

Note that race conditions are an example of a security vulnerability that is not caused by failure to validate untrusted input data.

On Screen Text

Race Condition

Race Condition Example



Narration

Here is an example of a race condition. In this example, the application is storing potentially sensitive data in a temporary file.

To do this, the application performs the following simple steps:

It creates and saves sensitive information to a temporary file.

It applies a security access control list (ACL) so that only the application can access the temporary file.

Let's see how a malicious user can exploit this application behavior.

First the application creates and saves sensitive information to a temporary file.

Before the application has a chance to apply an ACL that would allow access to the file only by the application, a malicious user copies the file.

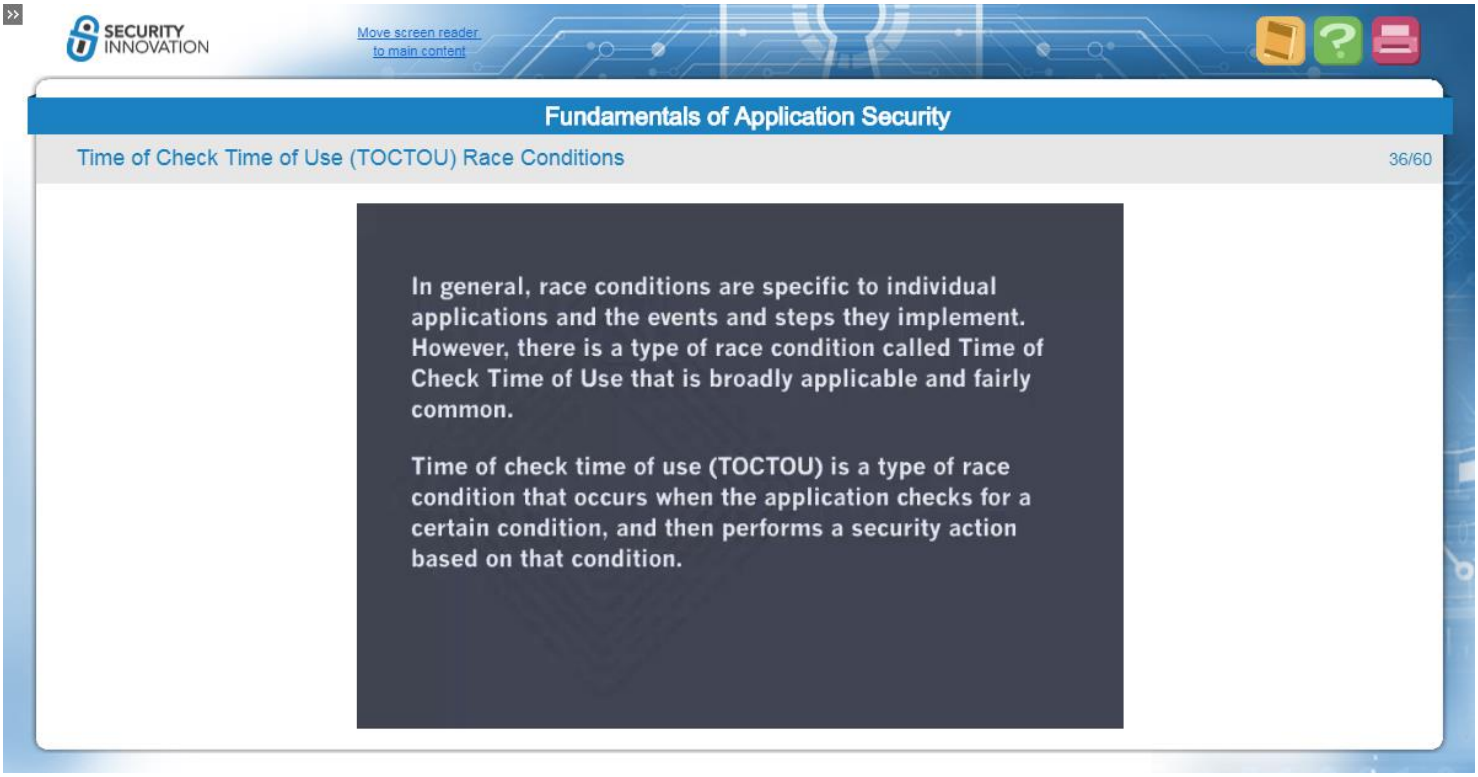
The application applies the ACL to the file,

not knowing that a malicious user has already attained a copy of the temporary file containing sensitive information.

On Screen Text

Race Condition Example

Time of Check Time of Use (TOCTOU) Race Conditions



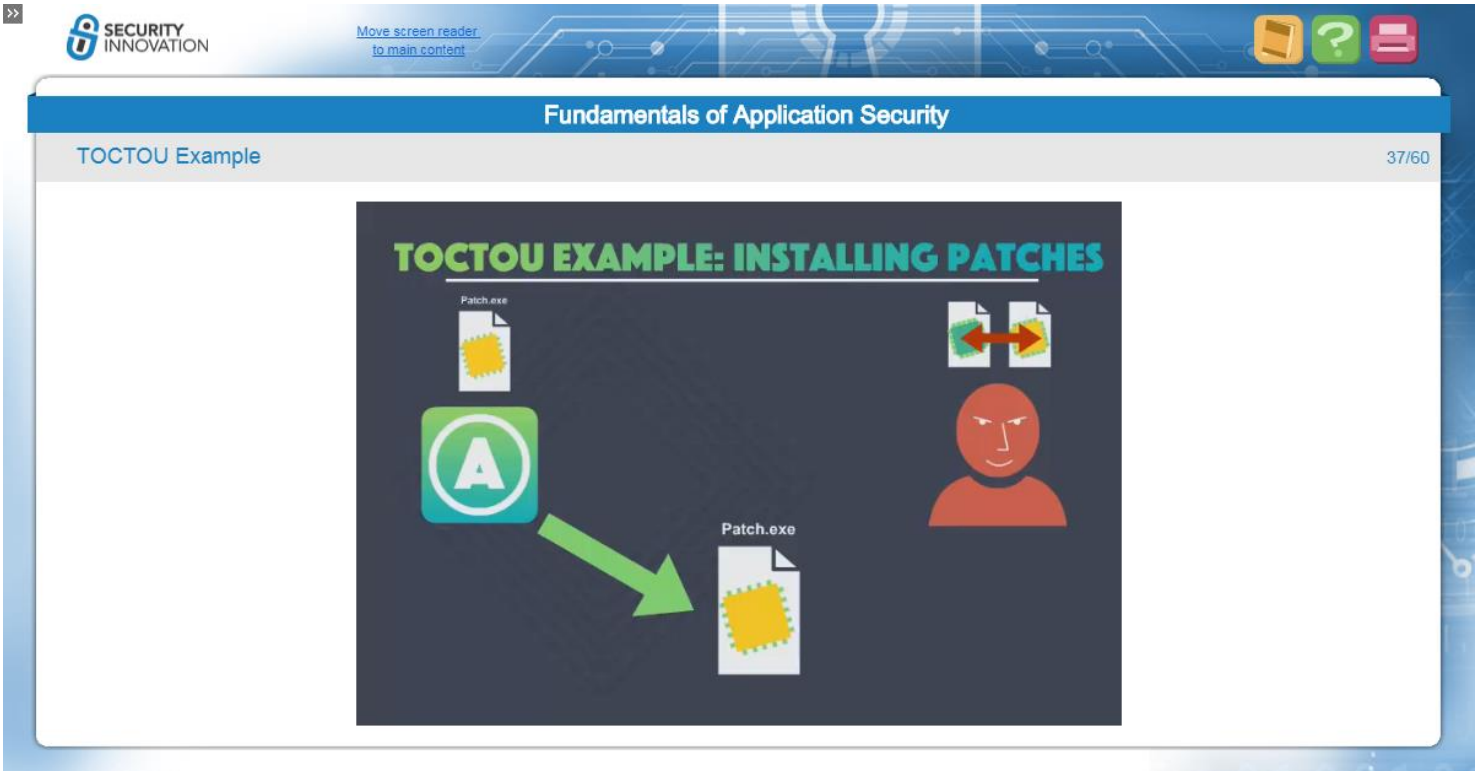
Narration

In general, race conditions are specific to individual applications and the events and steps they implement.
However, there is a type of race condition called Time of Check Time of Use that is broadly applicable and fairly common.
TOCTOU race conditions are a category of race conditions that occurs when an application checks for a certain condition,
and, based on the result of that condition check, performs a corresponding security action.
Let’s take a look at an example to gain a better understanding of TOCTOU issues.

On Screen Text

Time of Check Time of Use (TOCTOU) Race Conditions

TOCTOU Example



Narration

In this example, the application has been designed to automatically check for a valid digital signature when installing patches.

So the series of steps or events that the application implements to fulfill this scenario include:

Download and save the patch file.

Verify the digital signature of the patch file.

If the digital signature is invalid or missing, delete the patch file.

Let’s see how a malicious user can use a race condition to exploit this scenario:

The application downloads and saves the patch file.

The application reads and verifies the digital signature of the file.

The digital signature is valid. Before the application can execute the patch file,

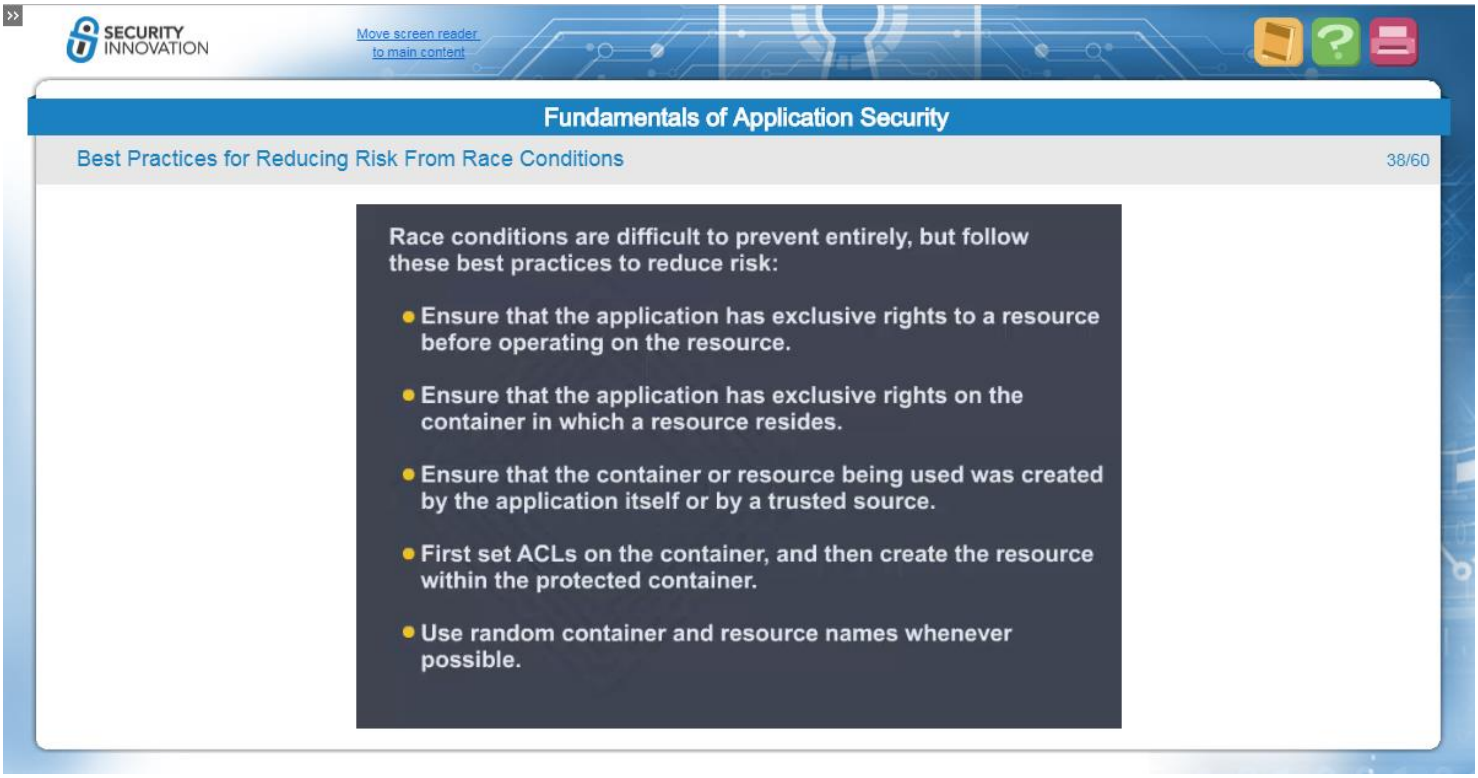
a malicious user swaps the patch file with an infected version.

The application executes the file that it believes to be un-tampered with and is subsequently compromised.

On Screen Text

TOCTOU Example

Best Practices for Reducing Risk From Race Conditions



Narration

Since race conditions are highly specific to individual applications,
there is no single technology or library that developers can use to reduce the risk from race conditions,
as with the case of other vulnerabilities, such as SQL injection.
There are however best practices that developers can follow that can help reduce the risk from race condition attacks. They are:
If an application relies on a resource, make sure that the application has attained exclusive rights to that resource prior to use,
and that the application has attained exclusive rights to the container that holds the resource.
For example, in the case of a file-based resource, ensure that the application has exclusive rights to the directory that contains the file.
If an application relies on a container or resource, ensure that the container or resource was created by the application itself or by a trusted source.
Going back to the example of a directory and file, ensure that it was the application or some other trusted source that created the directory and/or file.
When creating a container to hold a resource, such as a file,
set the ACLs on the container first and then create the file within the protected container.
Use random names when creating a container or resource.
There is no one fix-all solution for race conditions;
however if you follow the tips provided above when implementing your applications
the overall risk from race condition attacks should be significantly reduced.

On Screen Text

Best Practices for Reducing Risk From Race Conditions

The STRIDE Model

Narration

We've looked at buffer overflows, SQL injection, and race conditions,

which are just a few of the many security threats that you will confront as a secure application developer.

A valuable tool to help you understand, classify, and mitigate the broad spectrum of application security threats is known as the STRIDE model.

The STRIDE acronym stands for six categories of threats: spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege.

Spoofing threats involve an adversary impersonating someone or something to fool and exploit people or systems.

Tampering threats involve an adversary modifying data, usually as it flows across a network, in memory, or on disk.

Repudiation threats involve an adversary who performs an action and then plausibly denies having performed the action.

Information disclosure threats involve data that can be accessed by someone who should not have access.

Denial-of-service threats involve denying legitimate users access to a system or component.

Elevation of privilege threats involve a user or component being able to access data or programs for which they are not authorized.

You can use the STRIDE model to anticipate the types of threats that can affect your application.

To apply STRIDE, it is useful to gather program managers, developers, and testers together to discuss and identify potential threats and vulnerabilities.

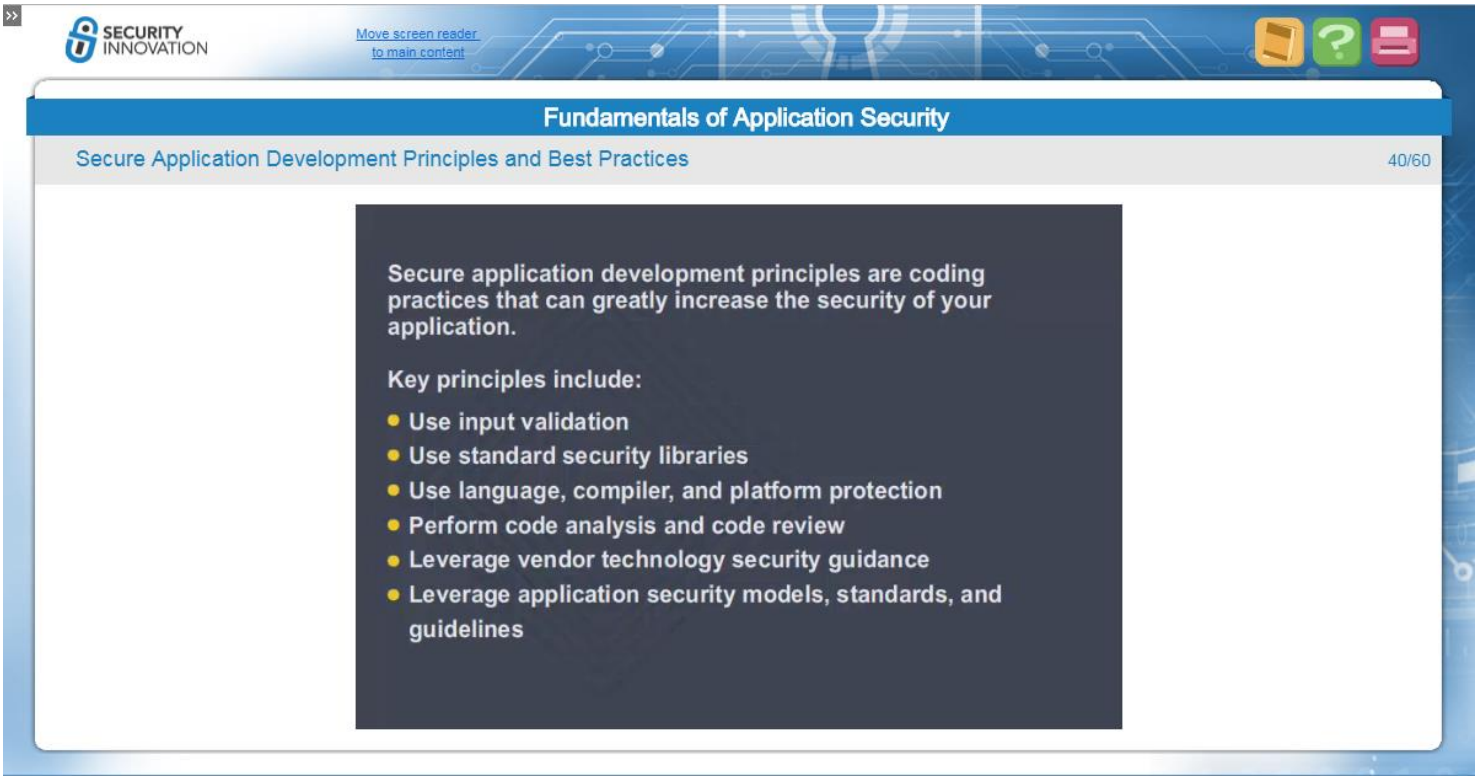
Testers, with their focus on how things go wrong, are often best suited to drive this meeting.

After the group has finished enumerating the threats, a program manager should create a list of all the elements from the diagram and list the threats that apply to each element.

On Screen Text

The STRIDE Model

Secure Application Development Principles and Best Practices



Narration

Secure coding principles are coding practices which, when followed, can greatly increase the security of an application.

There is no one principle that will render your application free from security vulnerabilities,

together these key principles can help you greatly improve your overall security posture:

Use input validation

Use standard security libraries

Use language, compiler, and platform protection

Perform code analysis and code review

Leverage vendor technology security guidance

Leverage application security models, standards, and guidelines

On Screen Text

Secure Application Development Principles and Best Practices

Input Validation Best Practices

Narration

Recall that input validation verifies that input is in a correct and safe format before it is used by your application.

It is a critical component of any application security strategy and when properly implemented,

it is considered the most effective method for mitigating risk from nearly every known application vulnerability.

As a general rule, any untrusted input should be validated. When in doubt, perform input validation anyways.

Be sure to implement input validation that is well tailored to the needs of your application.

Input validation is complex, because the structure of input, such as names, dates, and addresses, can vary widely.

So, use routines that cover the types of input that your application will need to handle.

Always perform validation on the server side. This is because the client side is too vulnerable to be relied upon for security—it may be bypassed or manipulated by an attacker.

You can still validate on the client for performance, and as a supplementary security measure.

If input passes both client and server validation checks, it is likely safe.

If it passes the client check but fails on the server side, it's a likely sign of an attack.

Note that validating on the client-side is acceptable, as long as it is used as a supplement to robust server side validation.

On Screen Text

Input Validation Best Practices

Use Standard Security Libraries

Narration

When implementing security controls in your application, there is often a standard library already written for you.

For example, Web applications are often susceptible to an attack called a cross-site scripting attack.

To mitigate the risk from these attacks, developers should use encoding libraries.

Cryptographic algorithms are the primary way with which applications protect the sensitive data they manage.

Always be sure to use a standard cryptographic library and avoid developing your own at all costs; and use secure communications libraries.

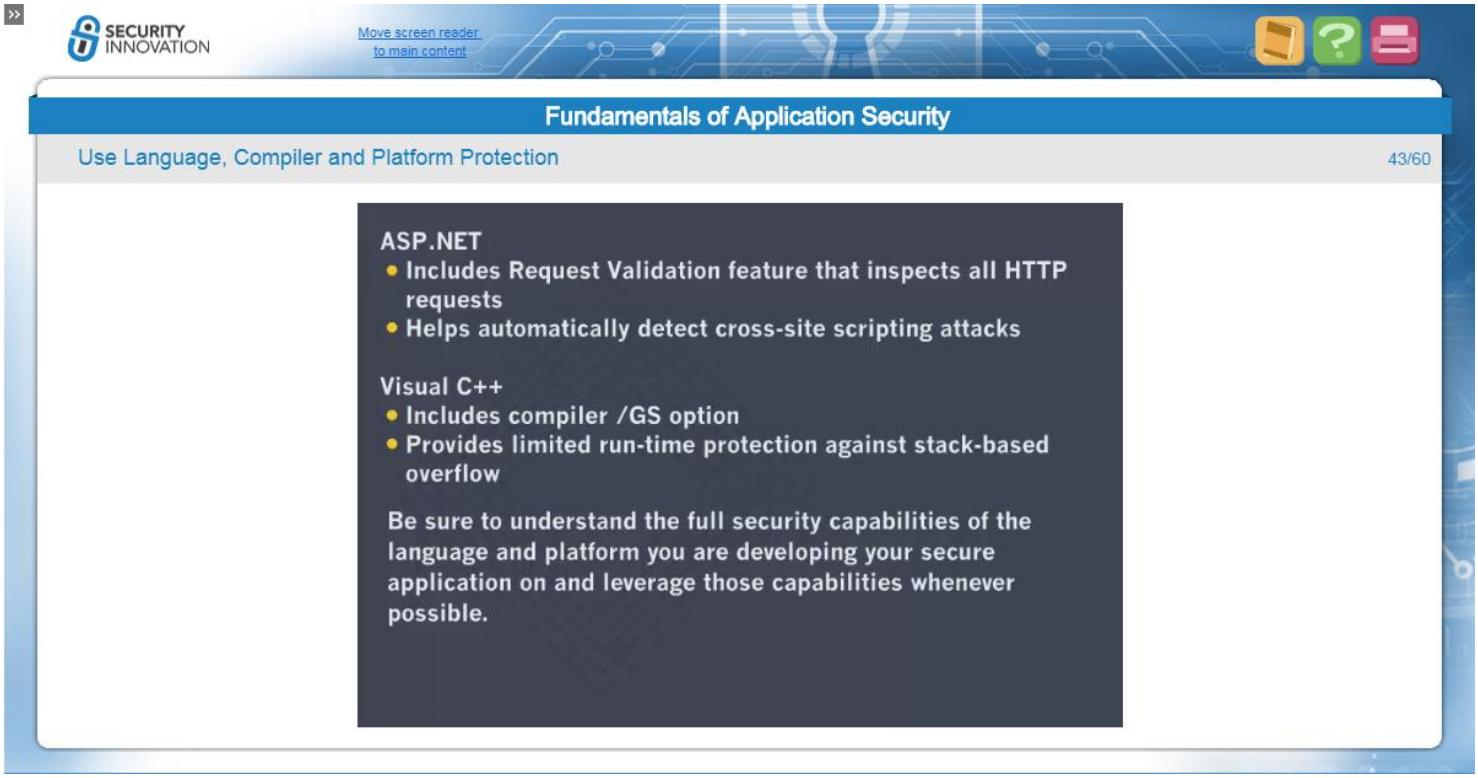
Not only does using a standardized library help ensure correctness of security mitigations and controls,

it also has the benefit of reducing your overall workload because you benefit from the work already done for you by security experts.

On Screen Text

Use Standard Security Libraries

Use Language, Compiler and Platform Protection



Narration

Programming languages and technology platforms often have built-in security features that you can use to reduce your overall workload in writing secure applications.

For example, Microsoft's ASP.NET has a built-in mechanism called Request Validation that detects certain cross-site scripting attacks, a very common Web attack.

Or if you are writing an application in C/C++ for Windows,

you can use the compiler's /GS option to add limited run-time protection against stack-based buffer overflows,

a very serious vulnerability that affects applications written in native or unmanaged languages.

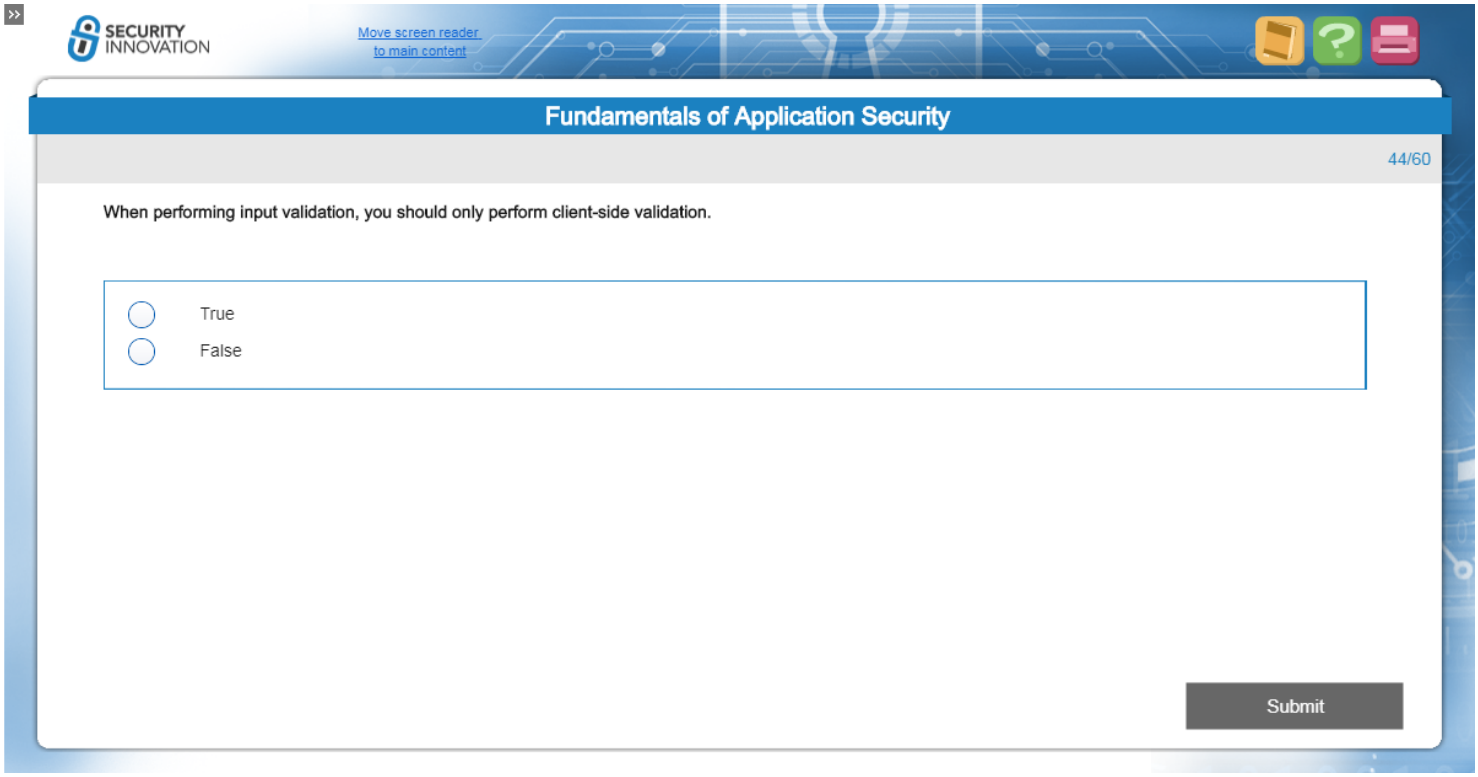
Be sure to understand the full security capabilities of the language and platform you are developing your secure application on

and leverage those capabilities whenever possible.

On Screen Text

Use Language, Compiler and Platform Protection

Knowledge Check



Narration

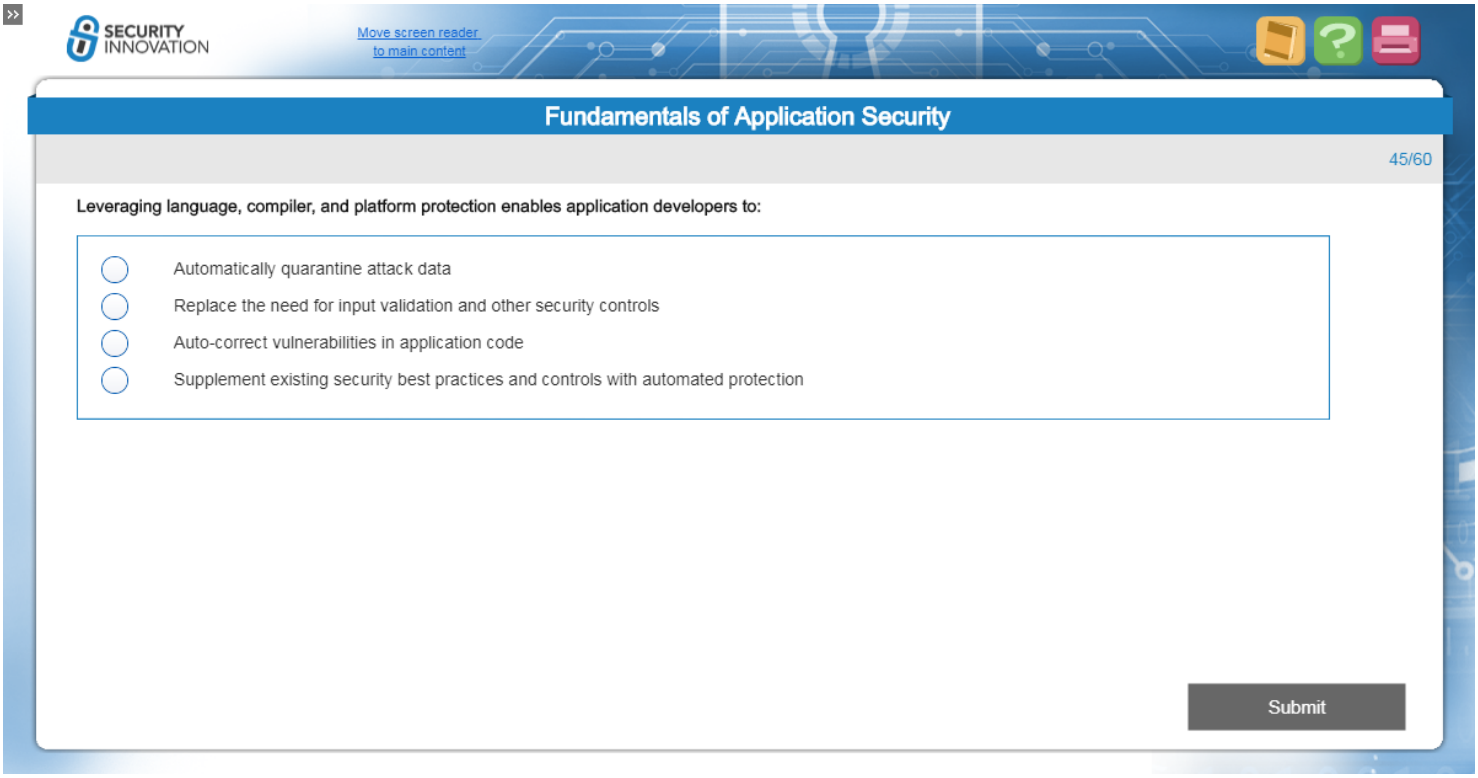
On Screen Text

When performing input validation, you should only perform client-side validation.

True

False

Knowledge Check



Narration

On Screen Text

Leveraging language, compiler, and platform protection enables application developers to:

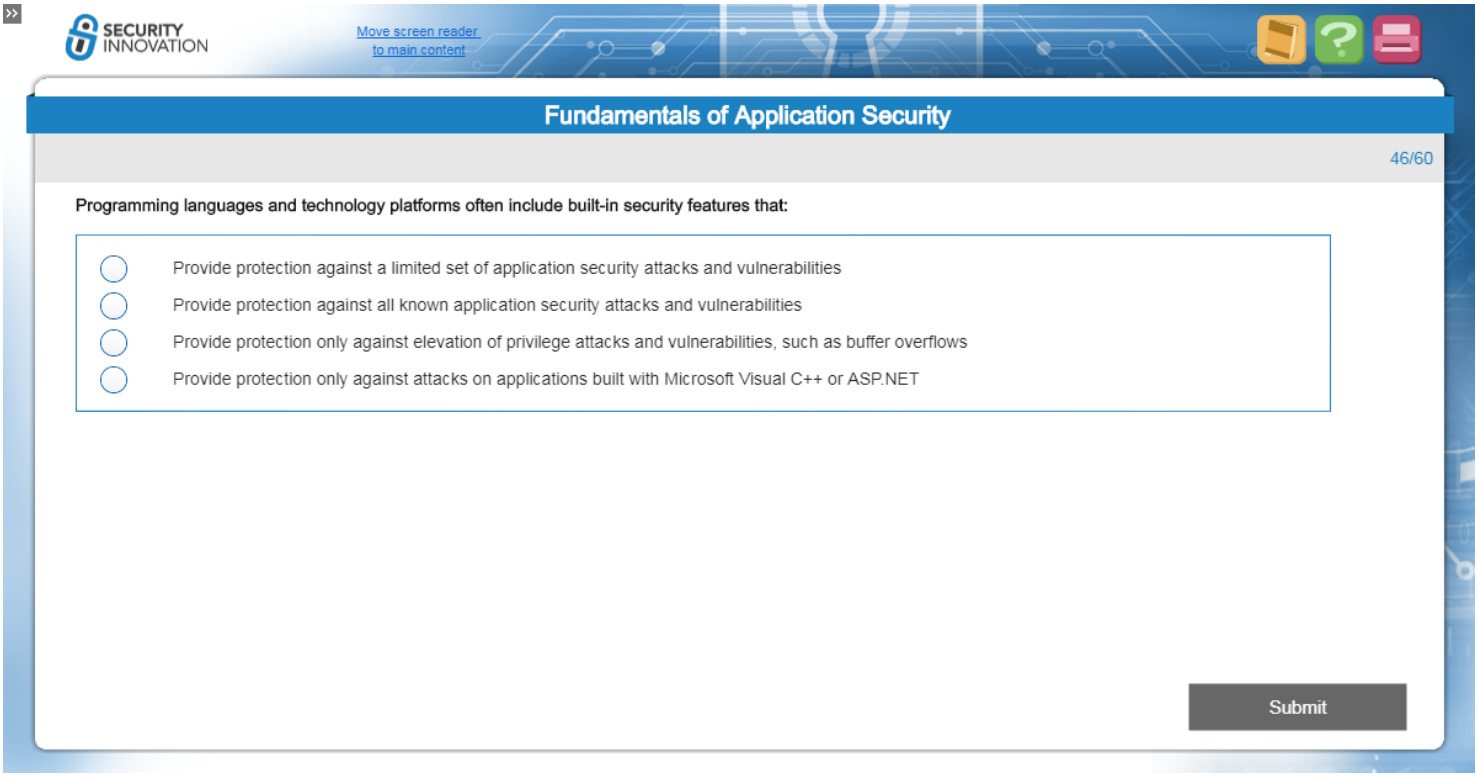
Automatically quarantine attack data

Replace the need for input validation and other security controls

Auto-correct vulnerabilities in application code

Supplement existing security best practices and controls with automated protection

Knowledge Check



Narration

On Screen Text

Programming languages and technology platforms often include built-in security features that:

Provide protection against limited set of application security attacks and vulnerabilities

Provide protection against all known application security attacks and vulnerabilities

Provide protection only against elevation of privilege attacks and vulnerabilities, such as bugger overflows

Provide protection only against attacks on applications built with Microsoft Visual C++ or ASP.NET

Use Code Analysis and Code Review

Narration

The next secure development principle is to use code analysis tools and code review.

Code analysis tools are software-based tools that inspect application source code in either compiled or non-compiled forms for known defects.

These tools can be categorized into static analysis and binary analysis.

The details of these two tool categories will be discussed momentarily.

Code review is the manual inspection of application source code typically by a security analyst.

Both code analysis and code review are an important component of any secure application development effort.

They allow you to detect and correct vulnerabilities in legacy applications,

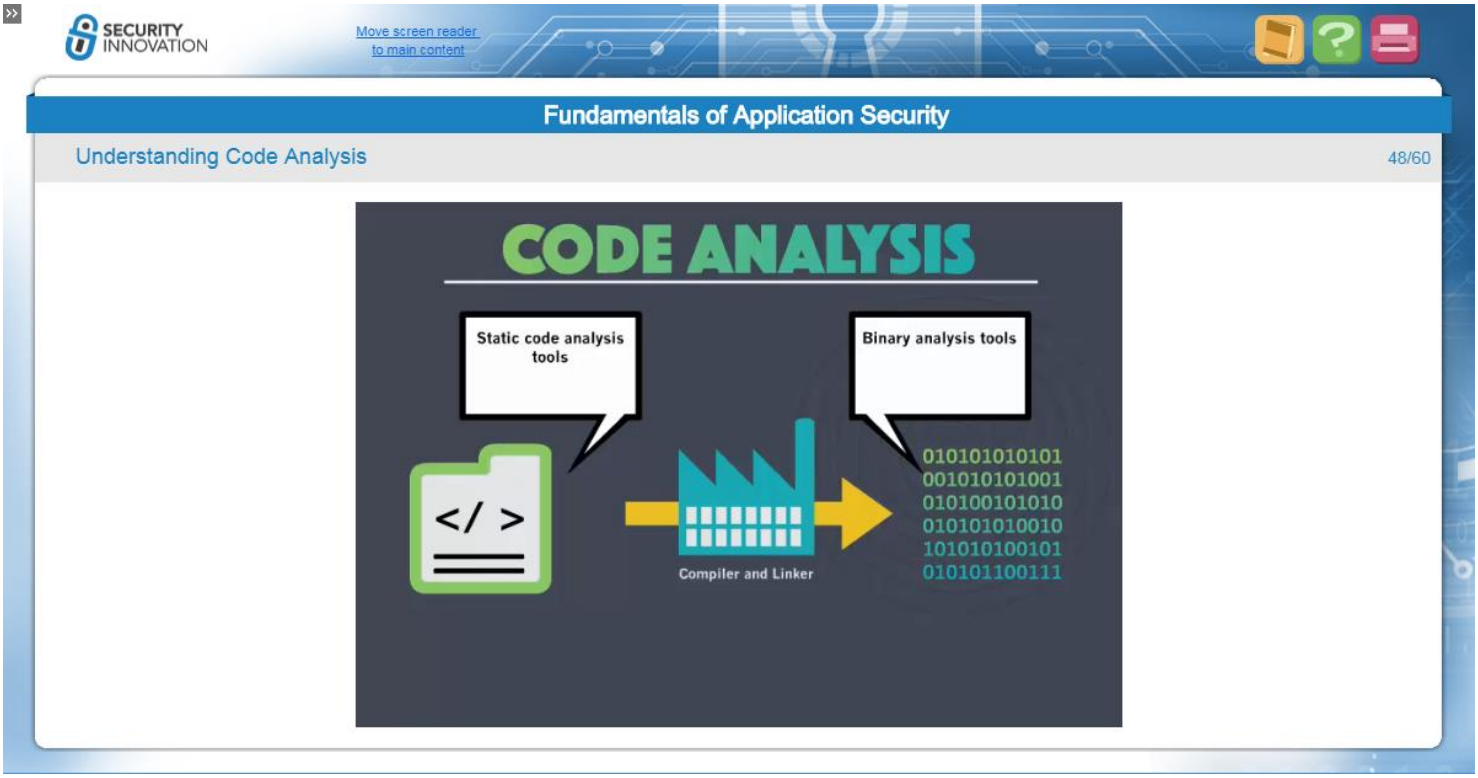
as well as detect and correct vulnerabilities in current applications fairly early in the development lifecycle,

where the cost of fixing vulnerabilities is low.

On Screen Text

Use Code Analysis and Code Review

Understanding Code Analysis



Narration

Let’s first focus our attention on code analysis.

As mentioned earlier, there are two categories of code analysis tools: static analysis and binary analysis.

In order to better understand the difference between the two, consider the application compilation lifecycle.

All applications are first expressed as source code.

These are the human readable instructions created by developers that implement software, such as Java code.

A compiler and linker takes that source code and turns it into machine code, often called an application binary.

The key difference between static analysis tools and binary analysis tools is the input into these tools.

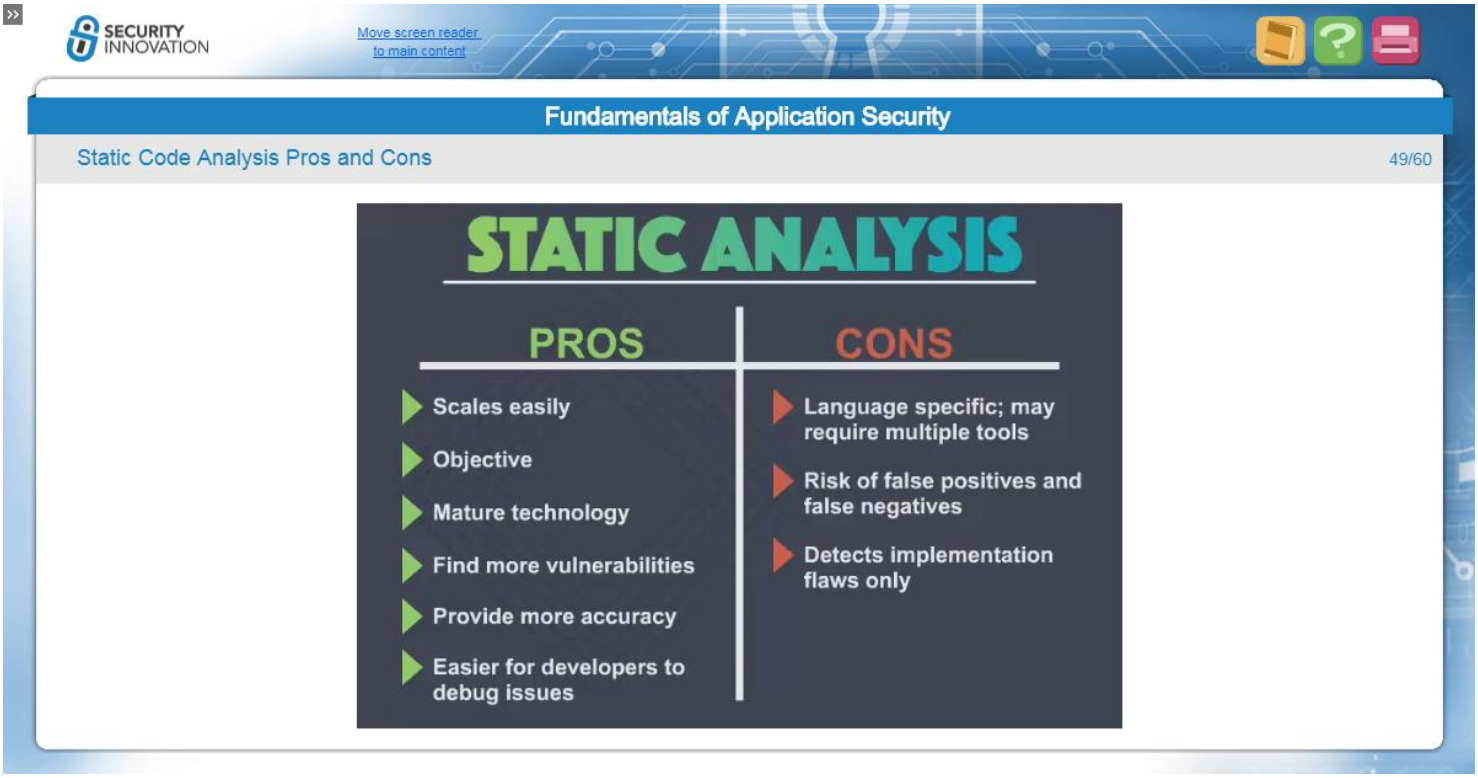
Static analysis tools analyze source code, while binary analysis tools analyze application binaries.

Each have specific benefits and limitations which we will discuss next.

On Screen Text

Understanding Code Analysis

Static Code Analysis Pros and Cons



Narration

We'll first take a look at static analysis tools to understand some of their benefits and limitations.

Let's begin on the "pros" side: Compared to manual code reviewers, static analysis software tools can scale fairly easily to analyze large amounts of code, and they are objective and unbiased.

When compared to binary analysis, static analysis technology is generally more mature.

Static analysis tools tend to find more vulnerabilities and provide more accuracy than binary analysis tools.

Finally the issues identified by static analysis tools are easier to debug, since they work with human readable source code rather than compiled code.

Static analysis tools also have certain limitations. First, they read source code and are language specific, so organizations that use many programming languages may require many static analysis tools.

In addition, they may produce false positives by flagging items that are not vulnerabilities, and produce false negatives by missing actual vulnerabilities.

This often creates more work for development teams, and can eventually degrade confidence in these tools.

Finally, static analysis tools can only find implementation flaws, that is, specific coding patterns that lead to vulnerabilities.

They cannot necessarily detect flaws related to the business logic of the application.

For example, if an application incorrectly assigns excessive rights to a user contrary to the intended design,

or if an application contains a bug where certain operations are performed out of order, a static analysis tool might not be able to detect it.

On Screen Text

Static Code Analysis Pros and Cons

Binary Analysis Pros and Cons

Narration

Now let's take a look at some pros and cons of binary analysis tools.

Like static analysis tools, binary analysis tools can scale to review large amounts of code, and are more objective and unbiased than human reviewers.

And, binary analysis tools are more accurate than static analysis tools.

This is because binary analysis tools operate on the actual code that gets executed,

whereas static analysis tools operate on source code, which may get optimized and re-arranged by compilers.

However, binary analysis tools do have limitations.

They are far less mature when compared to static analysis tools, so the breadth of vulnerabilities that they can detect is limited for now.

Also, the bugs identified by binary analysis tools are difficult to debug,

since developers have to be able to interpret compiled code.

Finally, binary analysis tools suffer from the same limitations as static analysis tools:

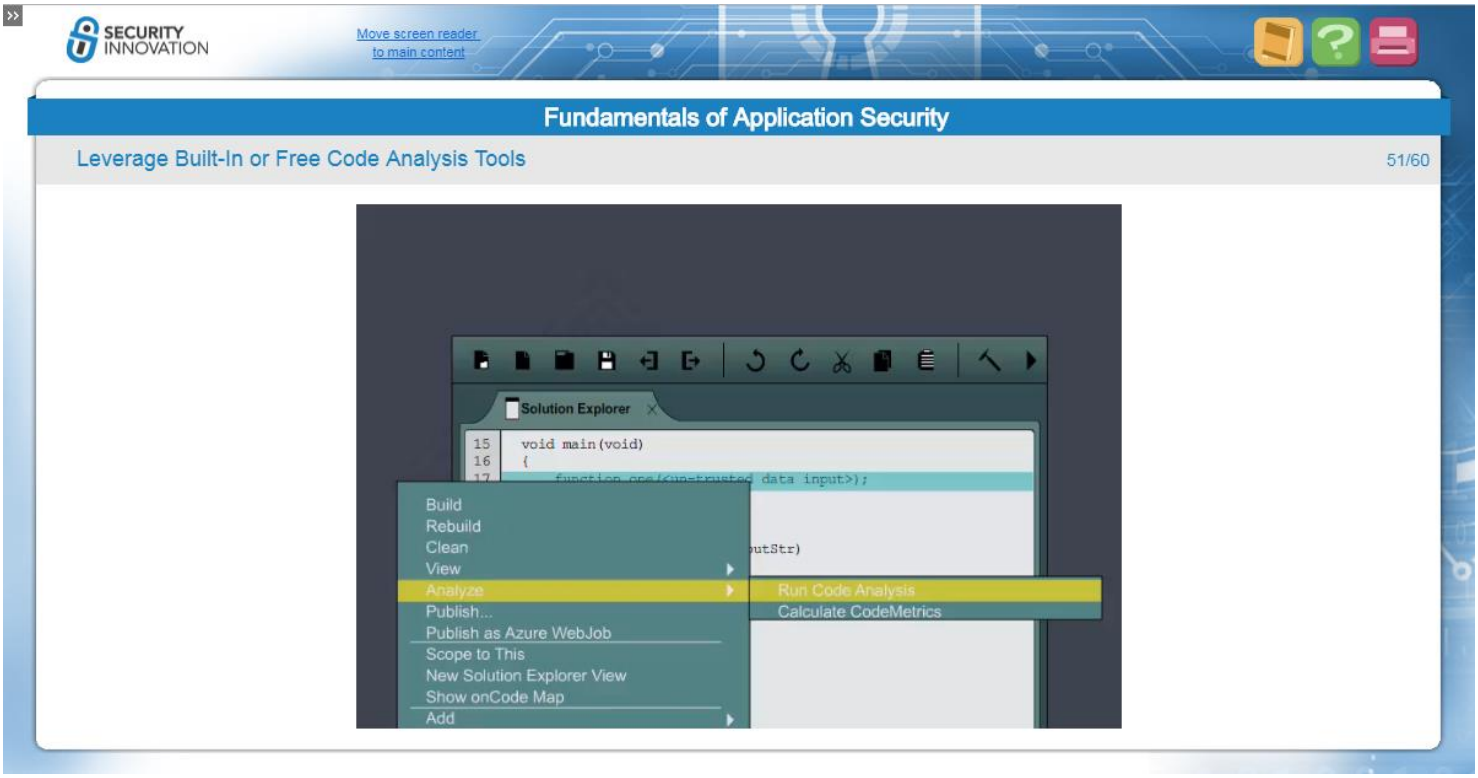
They are language-specific, so organizations may require many tools;

they produce false positives and false negatives; and they are unable to detect issues other than implementation flaws.

On Screen Text

Binary Analysis Pros and Cons

Leverage Built-In or Free Code Analysis Tools



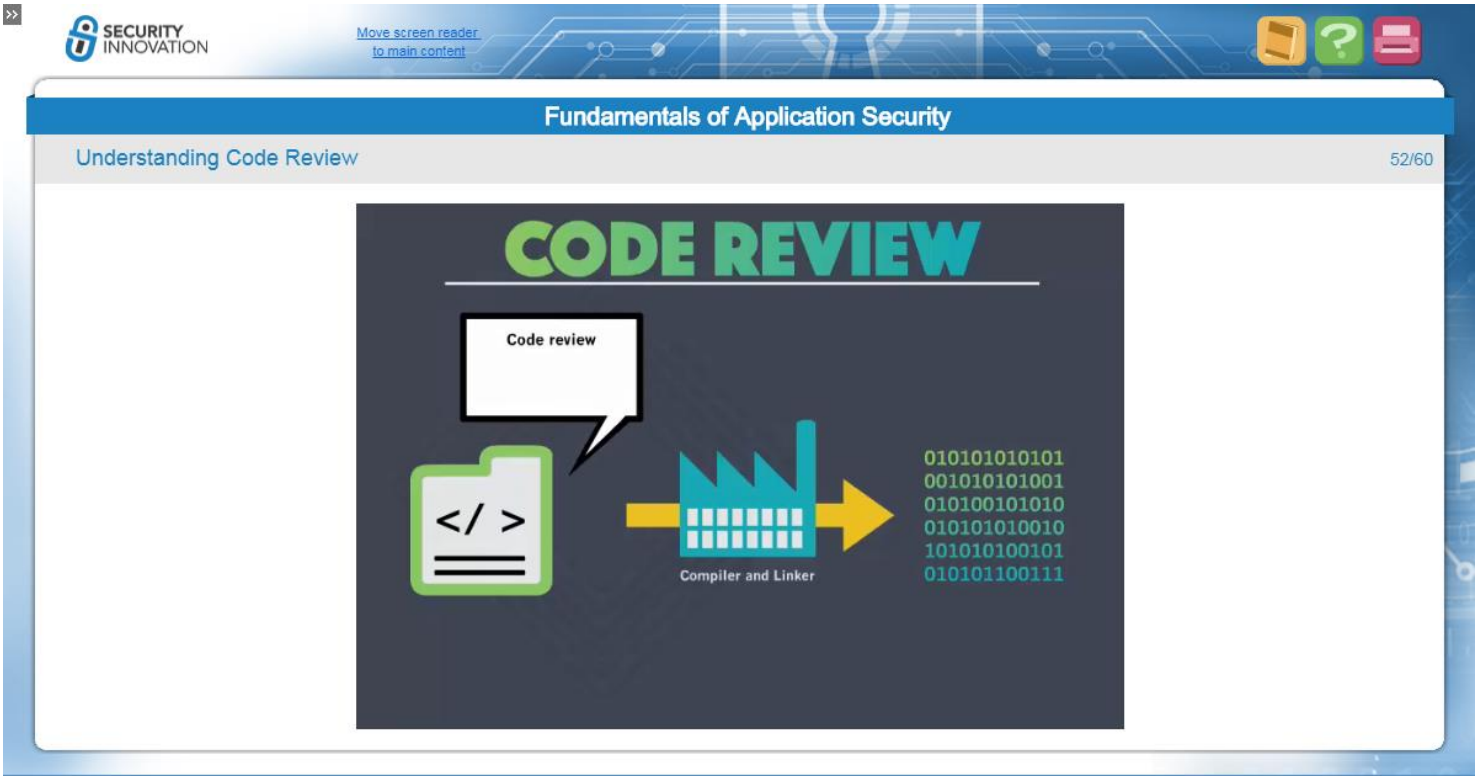
Narration

It is important to note that many Integrated Development Environments (IDEs) now include built-in code analysis tools. While these built-in tools may not find quite the comprehensive set of vulnerabilities as their commercial counterparts, they can often help you identify common vulnerabilities in code at no extra cost to you.

On Screen Text

Leverage Built-In or Free Code Analysis Tools

Understanding Code Review



Narration

Let’s now turn our attention back to the application compilation lifecycle, in order to understand code review.

Recall that applications are first expressed as source code.

The source code is processed by a compiler and linker to produce binary code.

Code review, like static analysis, uses human-readable source code to detect implementation flaws.

However, the analysis is performed by a human and not a software tool.

On Screen Text

Understanding Code Review

Code Review Pros and Cons

Narration

Manual code review has benefits and limitations of its own.

To begin with the benefits: Manual code reviews have the distinct advantage over code analysis of generally producing fewer false negatives and false positives.

And, human code reviewers can find not only the implementation flaws, but also flaws in design and in business logic.

While manual code review provides some great benefits, it does come with certain limitations.

First, manual code review is very time consuming and does not scale very well.

Another limitation is that reviewers can become fatigued from reviewing large amounts of code,

which diminishes their ability to accurately identify vulnerabilities.

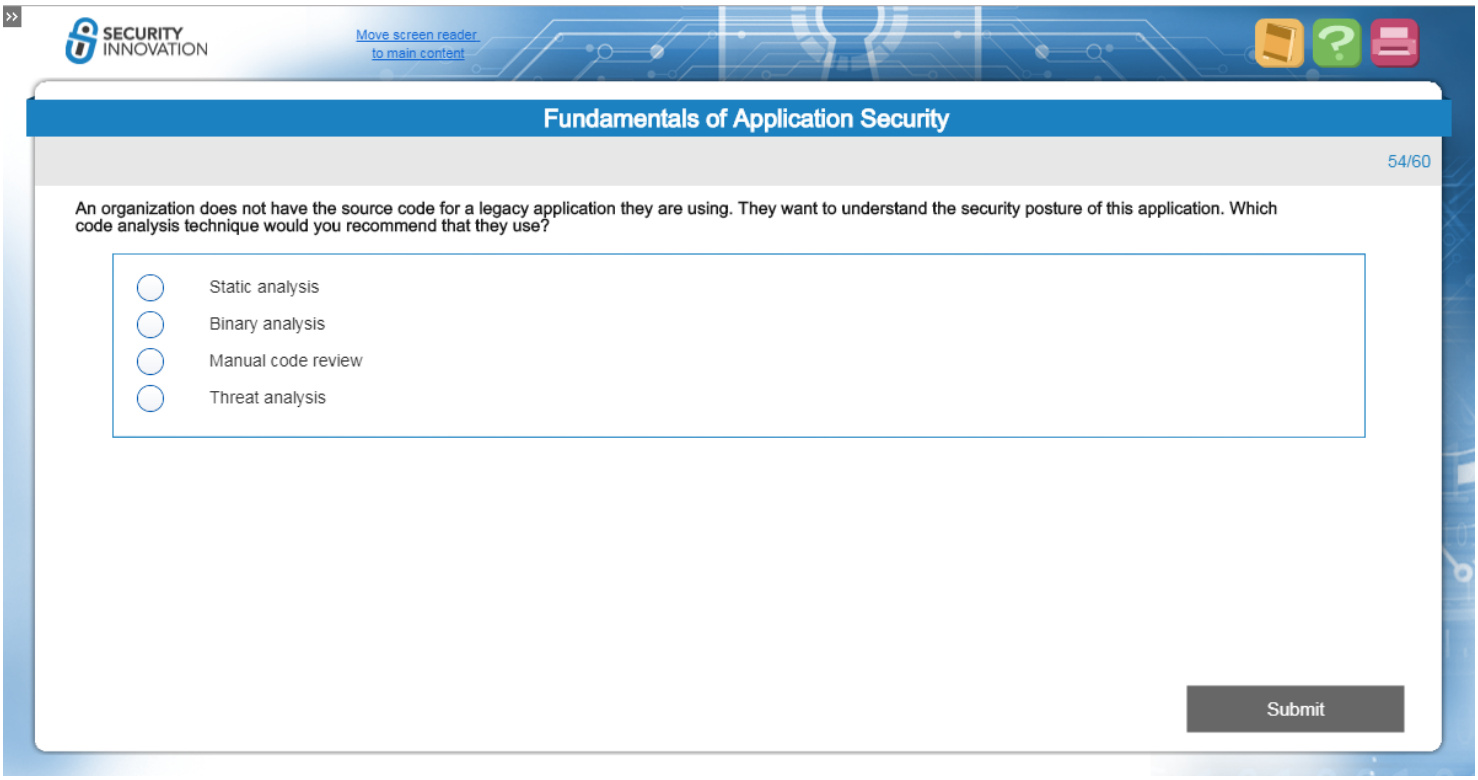
In addition, different reviewers have different areas of expertise,

so different vulnerabilities may get identified depending on who reviews the code.

On Screen Text

Code Review Pros and Cons

Knowledge Check



Narration

On Screen Text

An organization does not have the source code for a legacy application they are using. They want to understand the security posture of this application. Which code analysis technique would you recommend that they use?

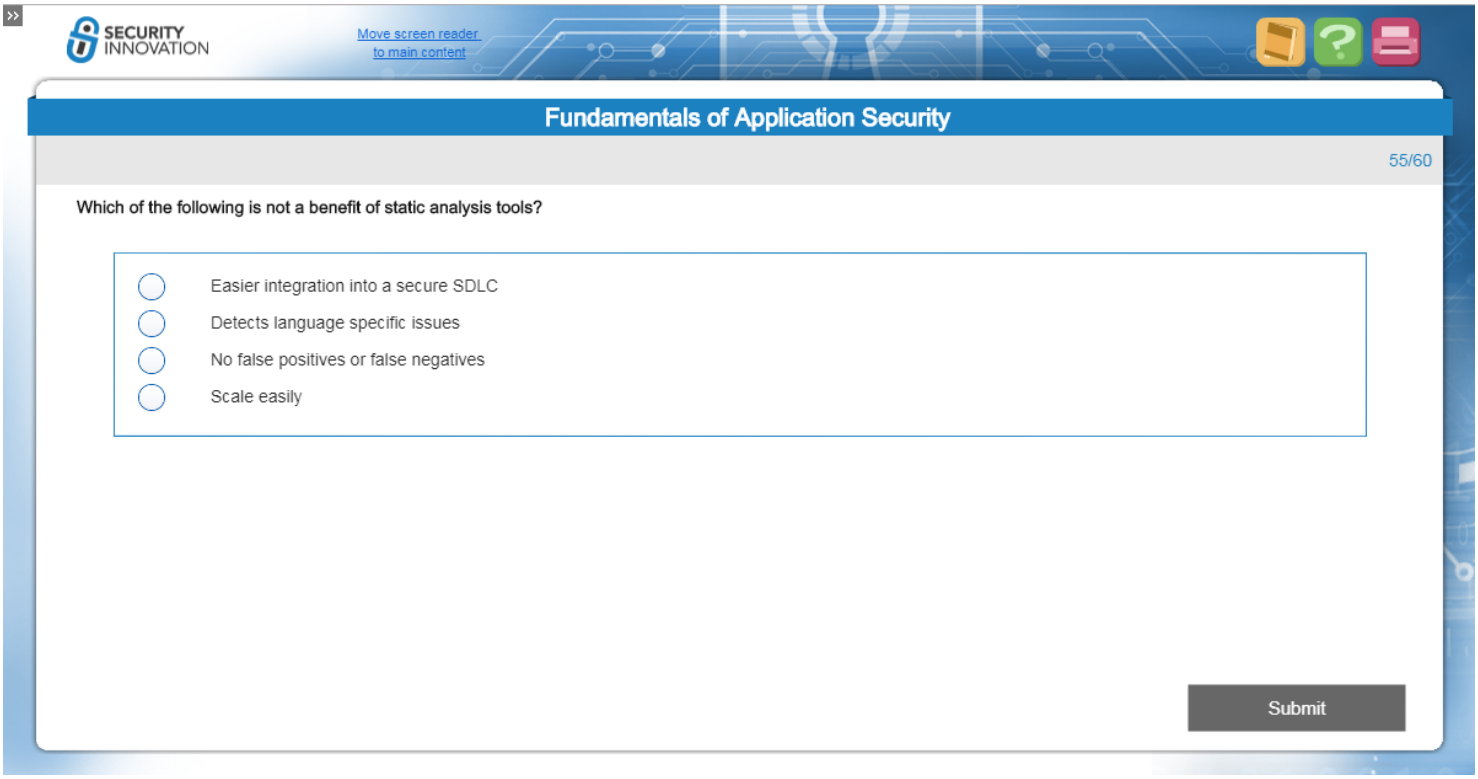
Static analysis

Binary analysis

Manual code review

Threat analysis

Knowledge Check



Narration

On Screen Text

Which of the following is not a benefit of static analysis tools?

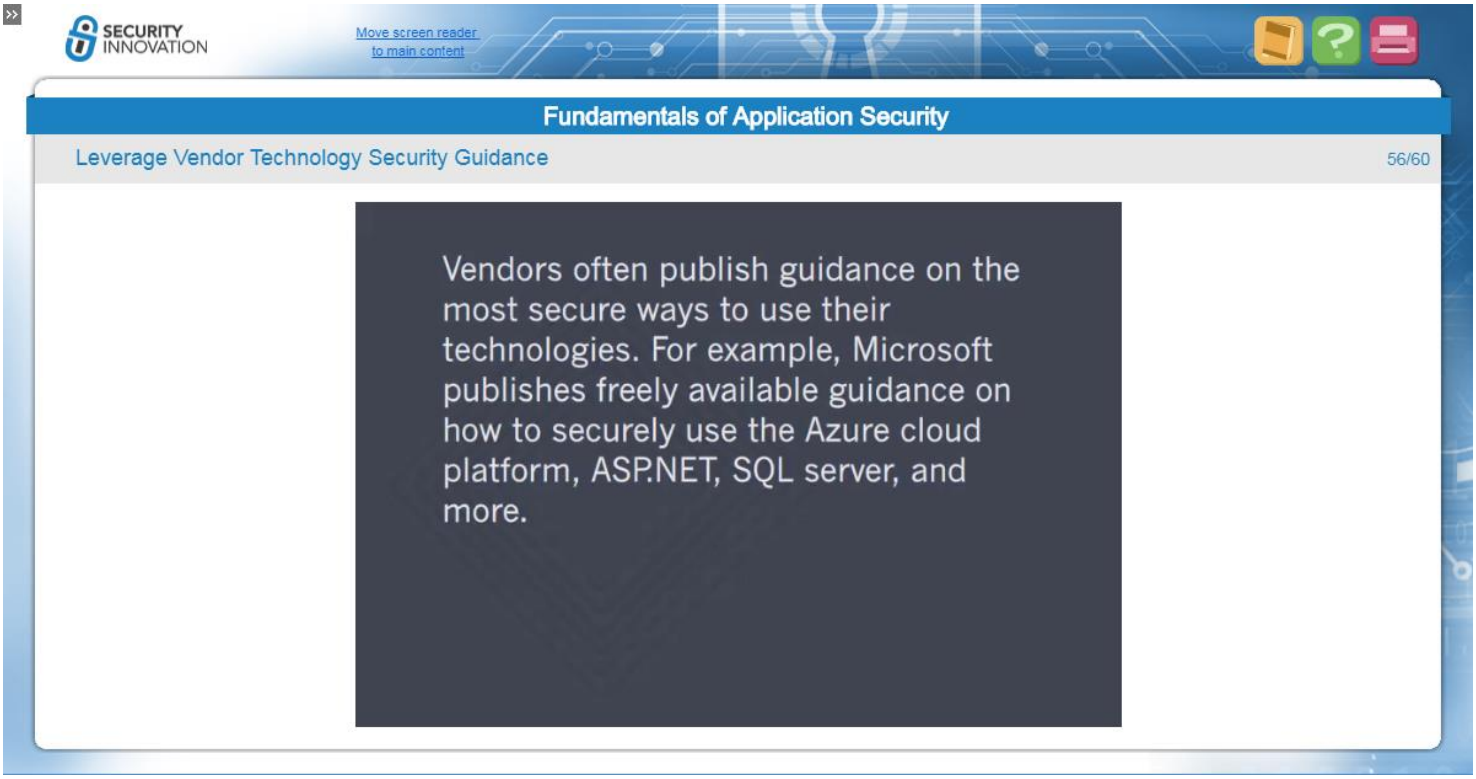
Easier integration into a secure SDLC

Detects language specific issues

No false positives or false negatives

Scale easily

Leverage Vendor Technology Security Guidance



Narration

Applications are often composed of several technologies working together to create value for organizations.
It's in the vendor's best interest to help customers successfully implement their technologies,
so vendors often publish freely available guidance on the most secure ways to use their technologies.
For example, Microsoft publishes guidance on how to securely use the Azure cloud platform,
ASP.NET, SQL Server, and more
You can save yourself significant effort in researching best practices for a given technology
by simply leveraging this work that vendors have already done for you.

On Screen Text

Leverage Vendor Technology Security Guidance

Leverage Secure Development Models, Standards and Guidelines

Narration

As a developer, you may not have the time, resources, or interest to become a security expert.

However, you still have a responsibility to your organization and your users to develop secure code.

Following secure development models, standards, and guidelines can help you understand key mitigations to reduce the risk from application security vulnerabilities.

This can help alleviate the need for you to become a security expert,

while still providing an efficient way for you to ensure a good level of security for your application and its users.

When you follow secure development models and guidelines,

keep these important points in mind: The first is that attacks and hacker techniques are continuously evolving.

Naturally, models and guidelines are regularly updated to reflect new threat landscapes.

It is therefore important that you periodically review the latest secure development models and guidelines

to help ensure that your application is resilient to current risks and attacks.

Second, following a particular secure development model or guideline will not render your application “hacker-proof” or free from exploitable vulnerabilities.

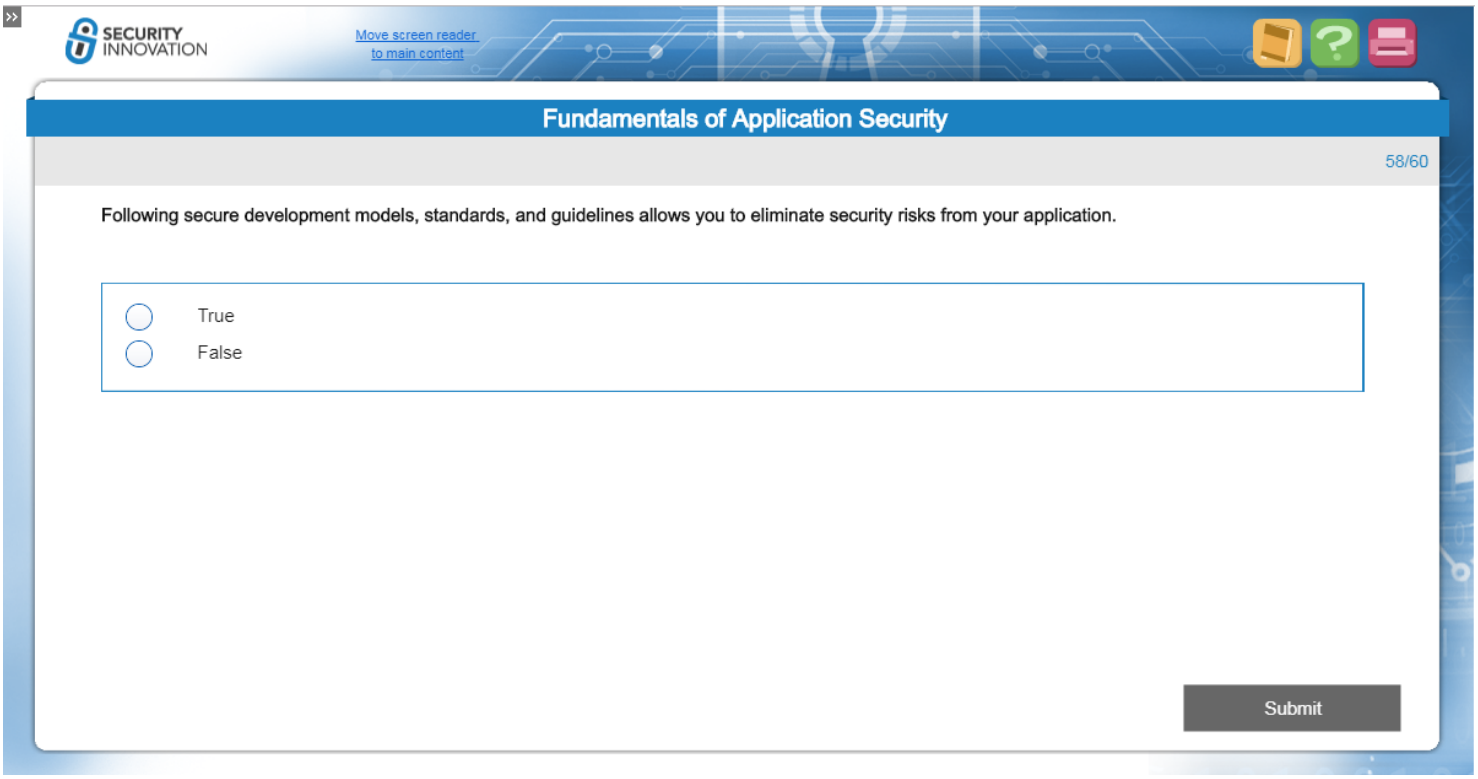
Models and guidelines highlight key risks and threats to your application at the time of writing,

but they do not cover all possible threats. There will undoubtedly be other risks that your application is exposed to that are not addressed by the model or guideline.

On Screen Text

Leverage Secure Development Models, Standards and Guidelines

Knowledge Check



Narration

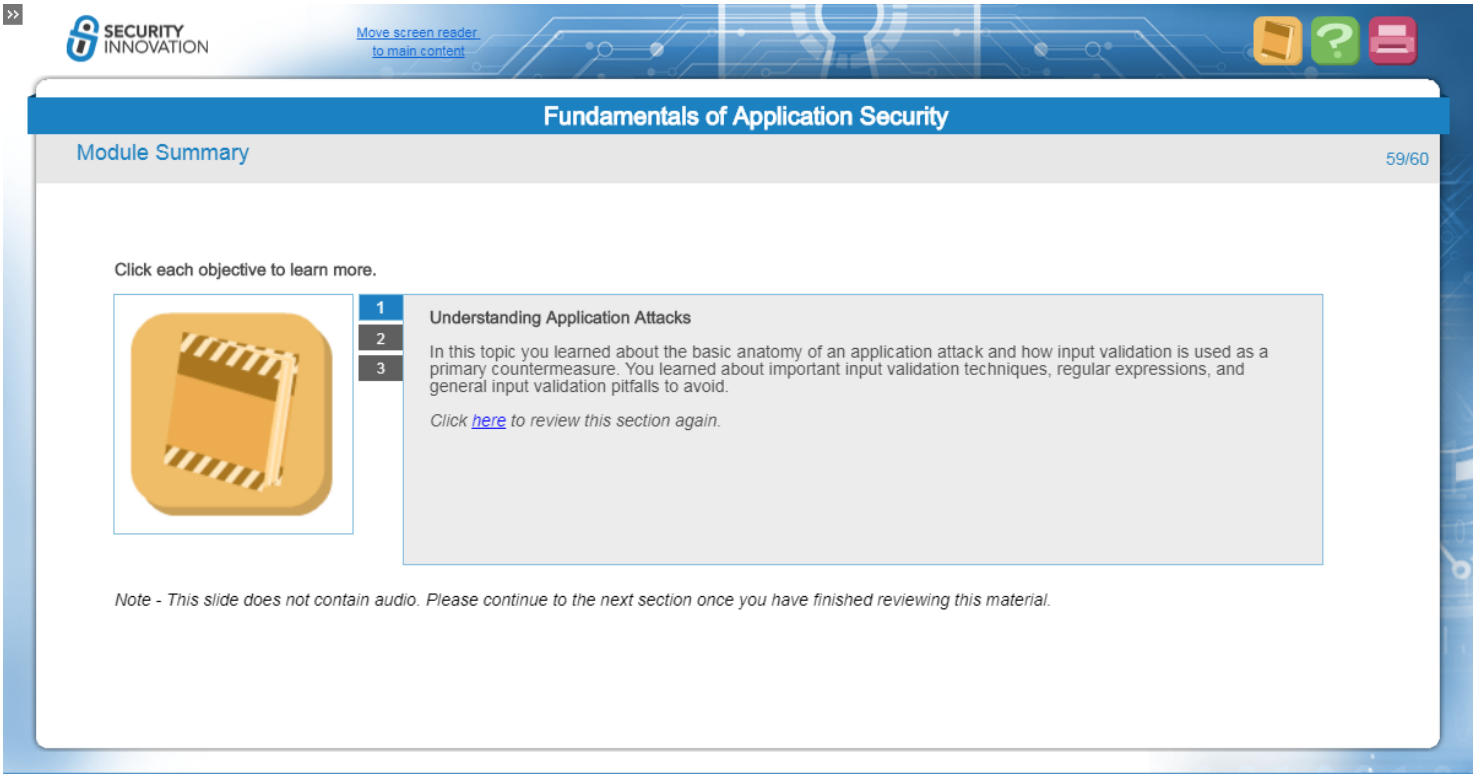
On Screen Text

Following secure development models, standards, and guidelines allows you to eliminate security risks from your application.

True

False

Module Summary



Narration

On Screen Text

Module Summary

Understanding Application Attacks

In this topic you learned about the basic anatomy of an application attack and how input validation is used as a primary countermeasure. You learned about important input validation techniques, regular expressions, and general input validation pitfalls to avoid.

Click here to review this section again.

Common Application Attacks

In this topic you were introduced to some common application attacks, such as buffer overflows, SQL injection, and race conditions. You also learned about additional resources to get more information about these and many other attacks.

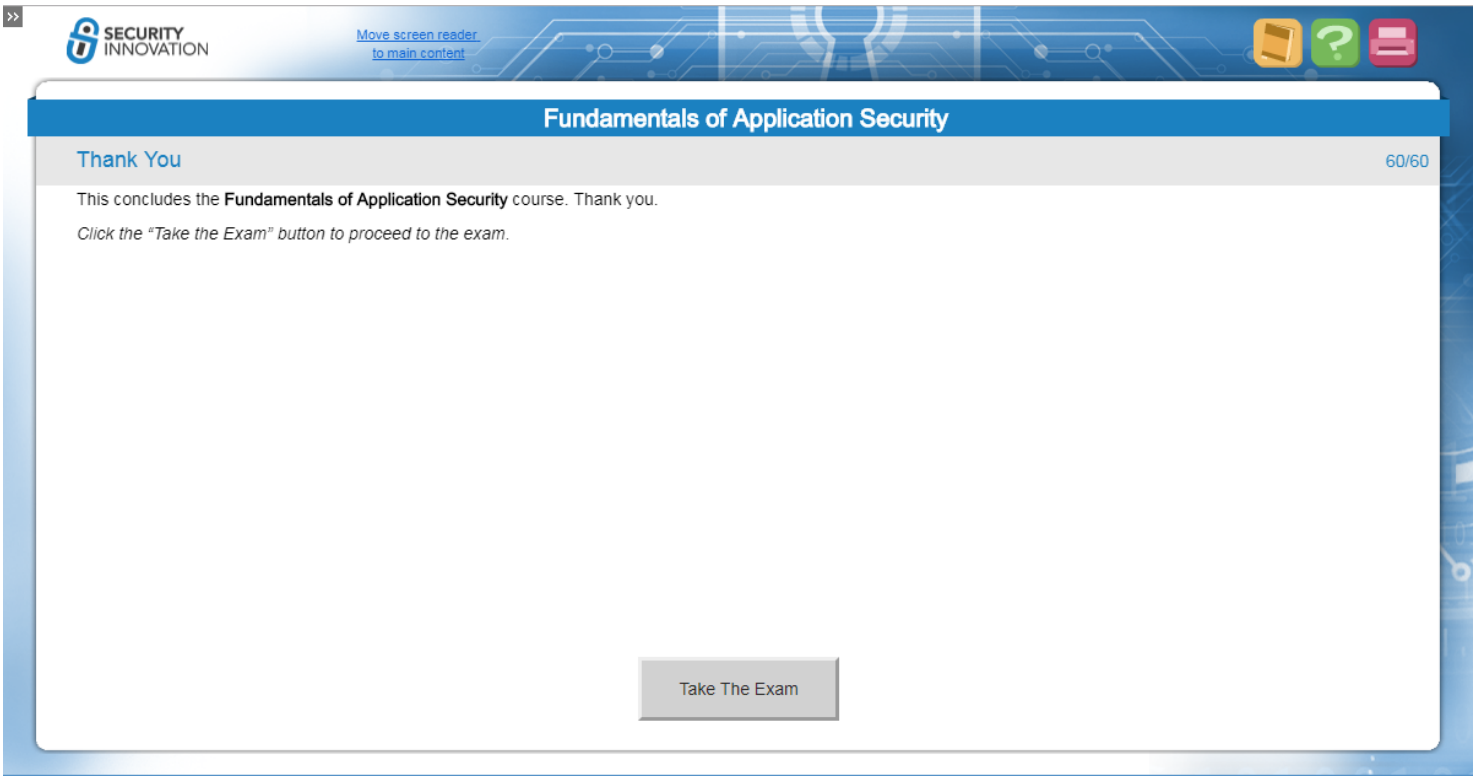
Click here to review this section again.

Key Secure Application Development Principles

In this topic you learned about some secure development principles and best practices, such as those for input validation, using standard libraries, and leveraging language and platform protection. You also learned about code analysis and code review, and the differences between the two. In addition, you learned about leveraging vendor guidance and secure development models, standards and guidelines as a means to help you in your secure development efforts.

Click here to review this section again.

Thank You



Narration

On Screen Text

Thank You

This concludes the **Fundamentals of Application Security** course. Thank you.

Click the "Take the Exam" button to proceed to the exam.