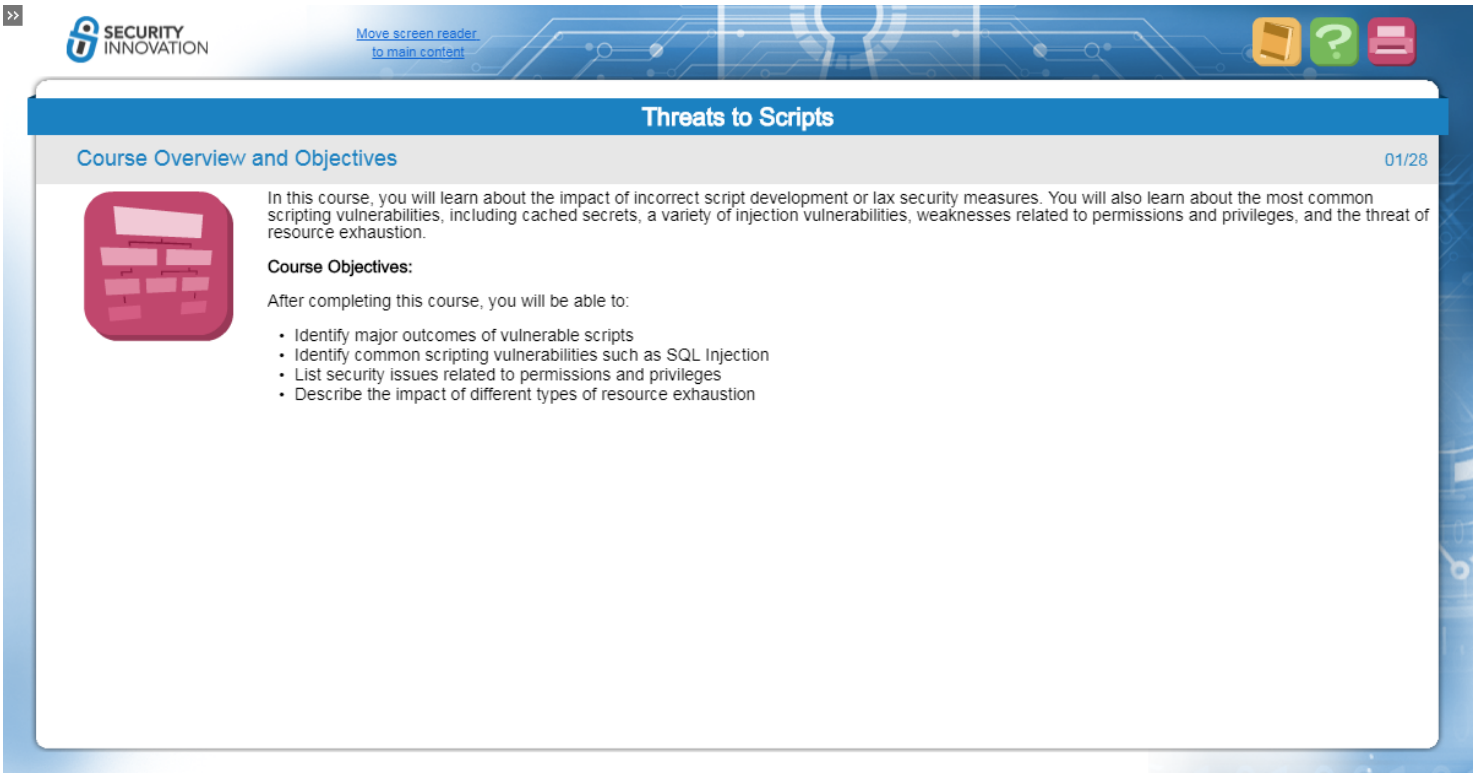
Course Overview and Objectives



Narration

In this course, you will learn about the impact of incorrect script development or lax security measures. You will also learn about the most common scripting vulnerabilities, including cached secrets, a variety of injection vulnerabilities, weaknesses related to permissions and privileges, and the threat of resource exhaustion.

After completing this course, you will be able to identify major outcomes of vulnerable scripts, identify common scripting vulnerabilities such as SQL Injection, list security issues related to permissions and privileges, and describe the impact of different types of resource exhaustion.

On Screen Text

Course Overview and Objectives

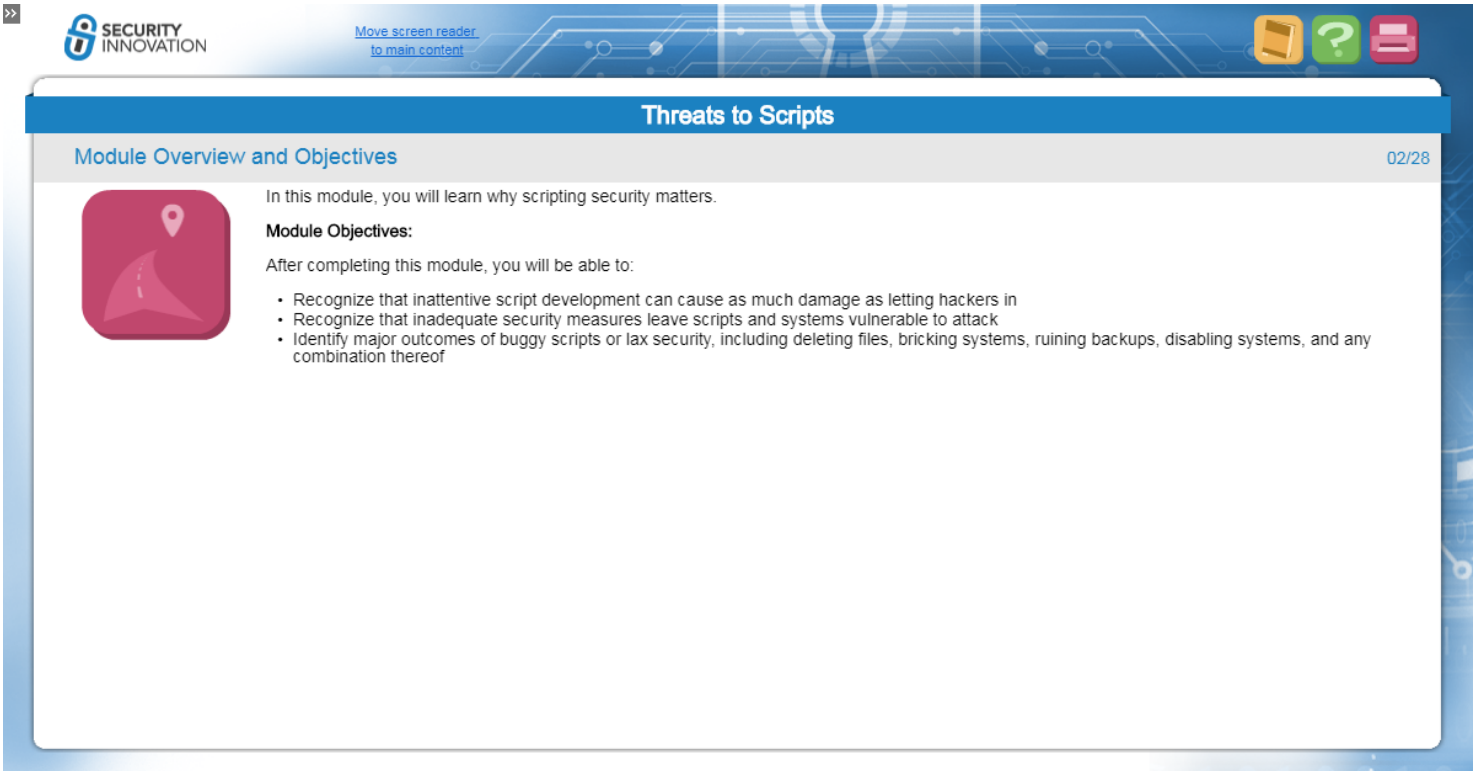
In this course, you will learn about the impact of incorrect script development or lax security measures. You will also learn about the most common scripting vulnerabilities, including cached secrets, a variety of injection vulnerabilities, weaknesses related to permissions and privileges, and the threat of resource exhaustion.

Course Objectives:

After completing this course, you will be able to:

- Identify major outcomes of vulnerable scripts
- Identify common scripting vulnerabilities such as SQL Injection
- List security issues related to permissions and privileges
- Describe the impact of different types of resource exhaustion

Module Overview and Objectives



Narration

In this module, you will learn why scripting security matters.

After completing this module, you will be able to recognize that inattentive script development can cause as much damage as letting hackers in, and recognize that inadequate security measures leave scripts and systems vulnerable to attack.

You will also be able to identify major outcomes of buggy scripts or lax security, including deleting files, bricking systems, ruining backups, disabling systems, and any combination thereof.

On Screen Text

Module Overview and Objectives

In this module, you will learn why scripting security matters.

Module Objectives:

After completing this module, you will be able to:

- Recognize that inattentive script development can cause as much damage as letting hackers in
- Recognize that inadequate security measures leave scripts and systems vulnerable to attack
- Identify major outcomes of buggy scripts or lax security, including deleting files, bricking systems, ruining backups, disabling systems, and any combination thereof

Why Scripting Security

Narration

Scripting security is an important topic because vulnerable scripts can and do cause tremendous damage and result in real compromises, whether they are short personal scripts or complex applications spanning thousands of hosts.

For example, if the attacker can manipulate the behavior of a script by sending malicious input, then the attacker is often able to run arbitrary commands or gain unauthorized access.

In addition to hacker attacks, buggy scripts can cause damage directly to the system or cause temporary Denial of Service conditions that affect multiple users and prevent them from working or cause them to lose data. Let's consider some scenarios that can be caused by poor scripting language security.

On Screen Text

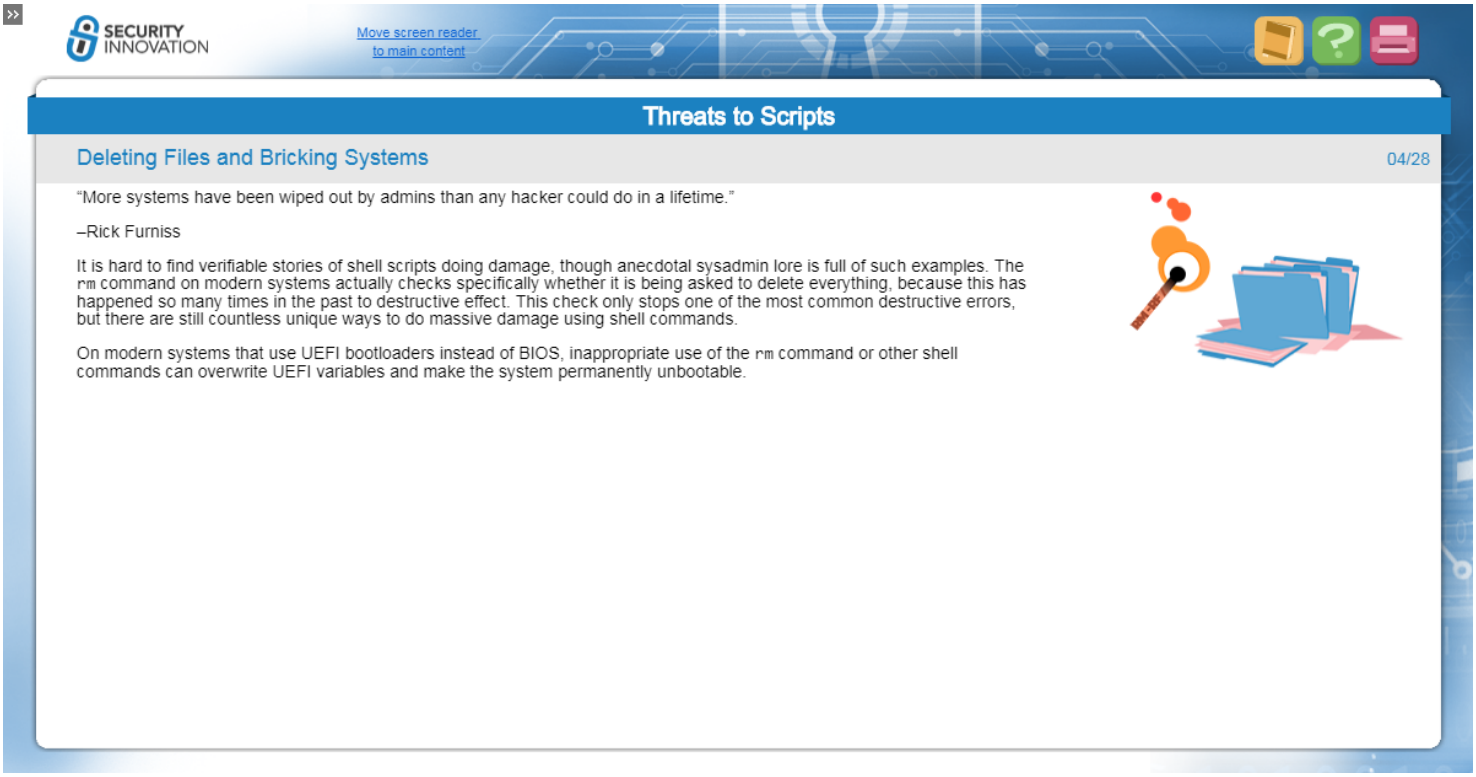
Why Scripting Security

Programs written in scripting languages can and do cause tremendous damage and result in real compromises.

Whether they are personal scripts or complex applications spanning thousands of hosts, it is important to follow sound security practices whenever using scripting languages.

Overlooking these principles can and does result in nightmare scenarios. Let's take a look at some of them.

Deleting Files and Bricking Systems



Narration

A script allows you to do whatever you can normally do faster and more efficiently. Any damage that you can cause can therefore be accomplished with maximum speed and efficiency using a script. It is easy to understand why there are few reports of damage caused by shell scripts.

Consider an employee that breaks everything using a buggy script. That employee might try to cover it up and if successful, then nobody will ever notice the damage. If the damage is noticed, then the employee might get fired and not want to tell the story to anyone out of embarrassment.

Since the damage is internal, the company is then put in a similar situation: try to cover it up or lose business. If the cover up is successful, then the story will not be public, and if business is lost, there is no real reason to explain to anyone what exactly happened, since it won't bring the lost business back anyway.

Usually, fixing the damage means restoring from backup or rebooting the systems, so there is a strong chance that nobody outside the organization will notice. Other than a lot of dread, some downtime and some lost unsaved data, there will usually be little outward sign that a disaster has taken place, but not everybody recovers from disasters quickly and successfully.

On Screen Text

Deleting Files and Bricking Systems

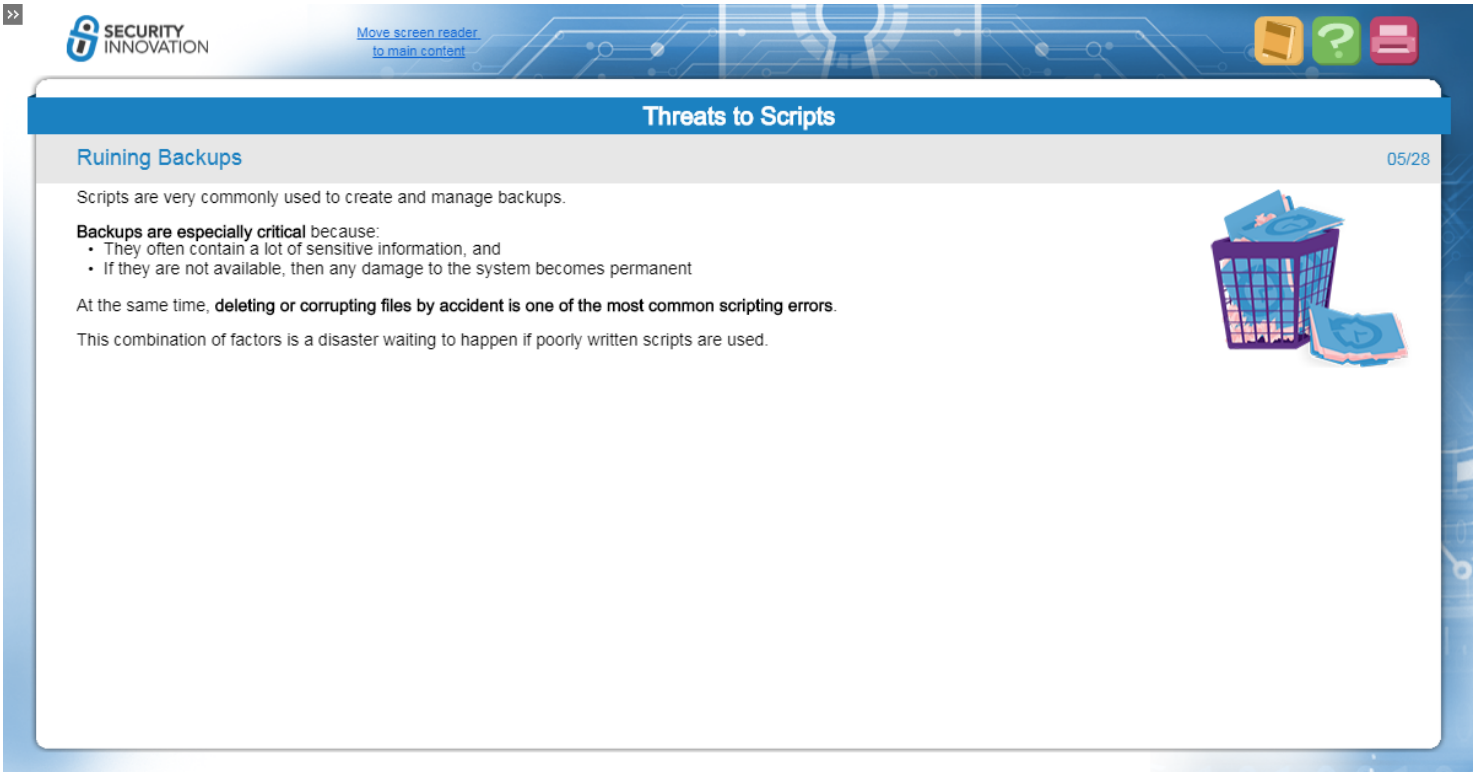
"More systems have been wiped out by admins than any hacker could do in a lifetime."

–Rick Furniss

It is hard to find verifiable stories of shell scripts doing damage, though anecdotal sysadmin lore is full of such examples. The `rm` command on modern systems actually checks specifically whether it is being asked to delete everything, because this has happened so many times in the past to destructive effect. This check only stops one of the most common destructive errors, but there are still countless unique ways to do massive damage using shell commands.

On modern systems that use UEFI bootloaders instead of BIOS, inappropriate use of the `rm` command or other shell commands can overwrite UEFI variables and make the system permanently unbootable.

Ruining Backups



Narration

A buggy script can destroy existing backups, fill up the backup storage media making future backups impossible, or create backups that are incomplete or unusable. The bugs might not be trivial and a script that works correctly in testing might still cause damage when the system environment changes or an attacker manipulates it.

The backups themselves might fall into the hands of attackers. Backups often contain credentials that can be leveraged to perform additional attacks. Database backup scripts might allow attackers to access or change data in the database.

Injecting malicious code into backups can help attackers maintain unauthorized access even after the system has been restored to what is presumed to be a trusted state.

On Screen Text

Ruining Backups

Scripts are very commonly used to create and manage backups.

Backups are especially critical because:
- They often contain a lot of sensitive information, and
- If they are not available, then any damage to the system becomes permanent

At the same time, **deleting or corrupting files by accident is one of the most common scripting errors**.

This combination of factors is a disaster waiting to happen if poorly written scripts are used.

Disabling Systems

Narration

Besides damaging data, a script that uses up too many resources can make a system unusable. Being rendered unusable is especially troublesome for systems that can only be accessed remotely and are shared by multiple users. In that case, nobody can use the system until someone physically goes and restarts it, or otherwise stops the script from running.

A script that uses multiple systems can cause all of them to become temporarily or permanently unusable.

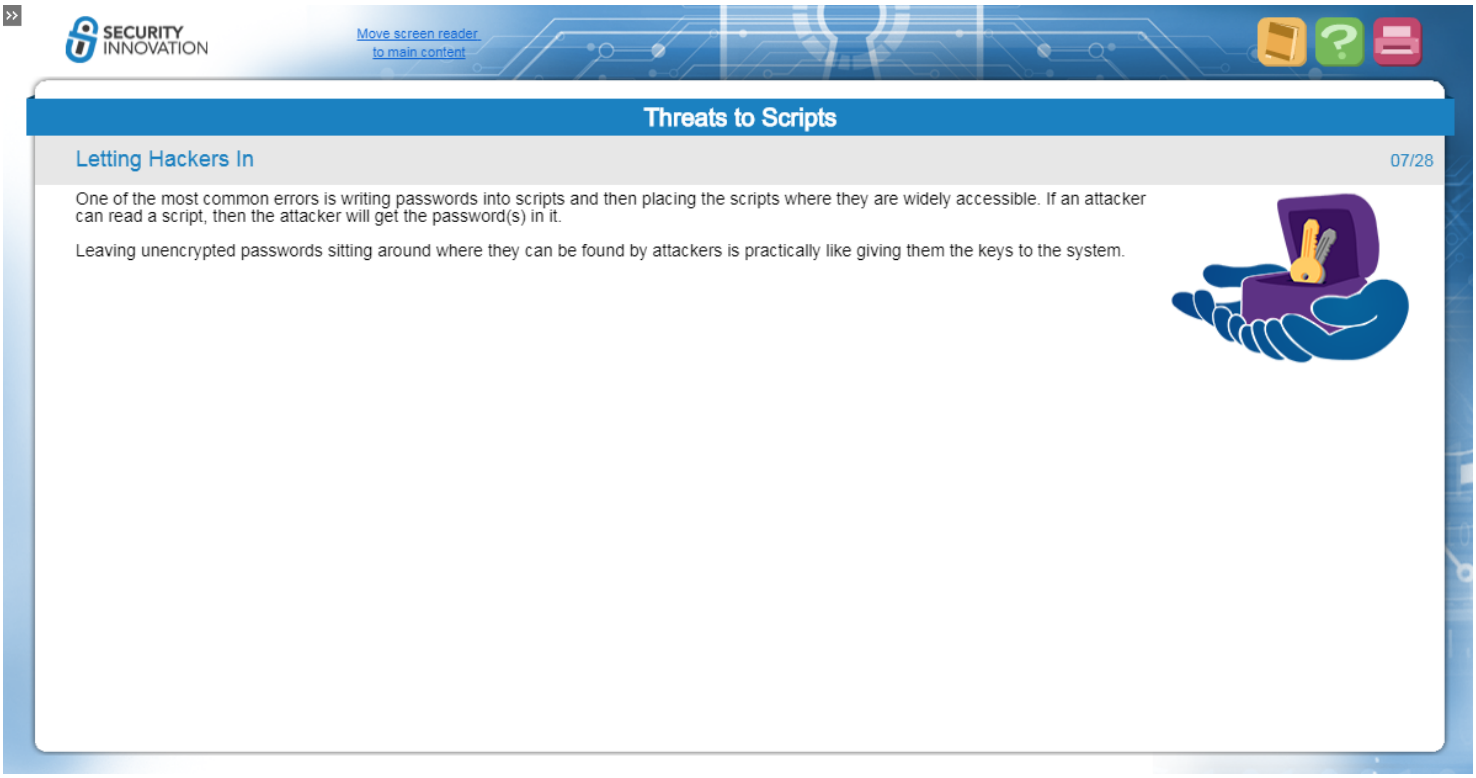
On Screen Text

Disabling Systems

A poorly written script can use up excessive resources to the point that the system becomes unusable. A script that uses multiple systems can render multiple systems unusable.

Being rendered unusable is particularly bad for systems that can only be accessed remotely and are shared by multiple users or perform some kind of infrastructure role. For example, an entire network can be brought down if routers are taken out of commission by a script.

Letting Hackers In



Narration

Besides doing damage directly, poorly written scripts can let hackers in. One of the most common errors is writing passwords into scripts and then placing the scripts where they are widely accessible.

Some people even manage to publish their scripts together with their passwords on the web for everyone in the world to see. Leaving passwords around basically hands the keys directly to the attacker and unfortunately happens in real life all too frequently.

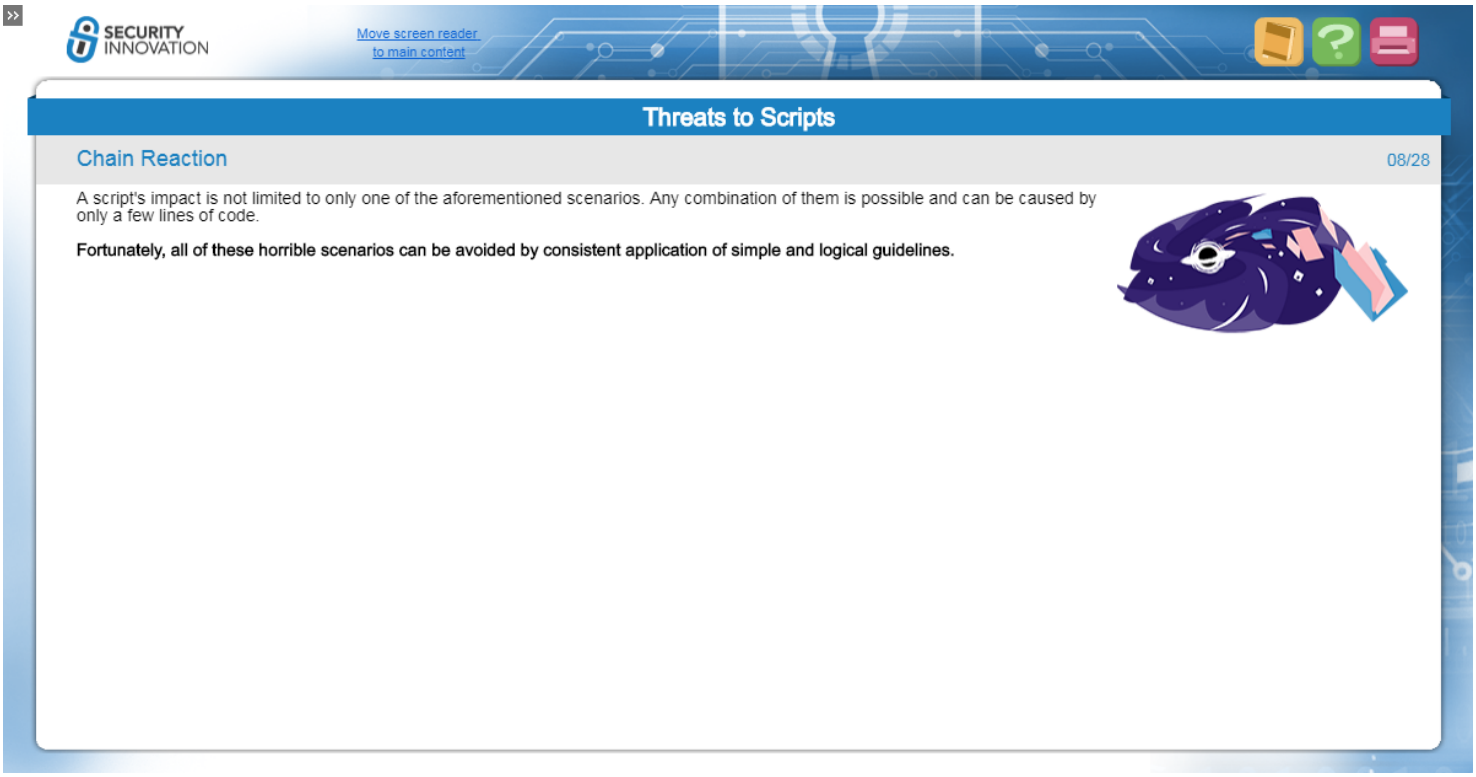
On Screen Text

Letting Hackers In

One of the most common errors is writing passwords into scripts and then placing the scripts where they are widely accessible. If an attacker can read a script, then the attacker will get the password(s) in it.

Leaving unencrypted passwords sitting around where they can be found by attackers is practically like giving them the keys to the system.

Chain Reaction



Narration

A script’s impact is not limited to only one of the aforementioned scenarios.

A particularly destructive script might corrupt backups, delete current data, and disable networked systems all at the same time, all completely unintentionally and caused by only a few innocent-looking lines of code.

Fortunately, all of these horrible scenarios can be avoided by consistent application of simple and logical guidelines.

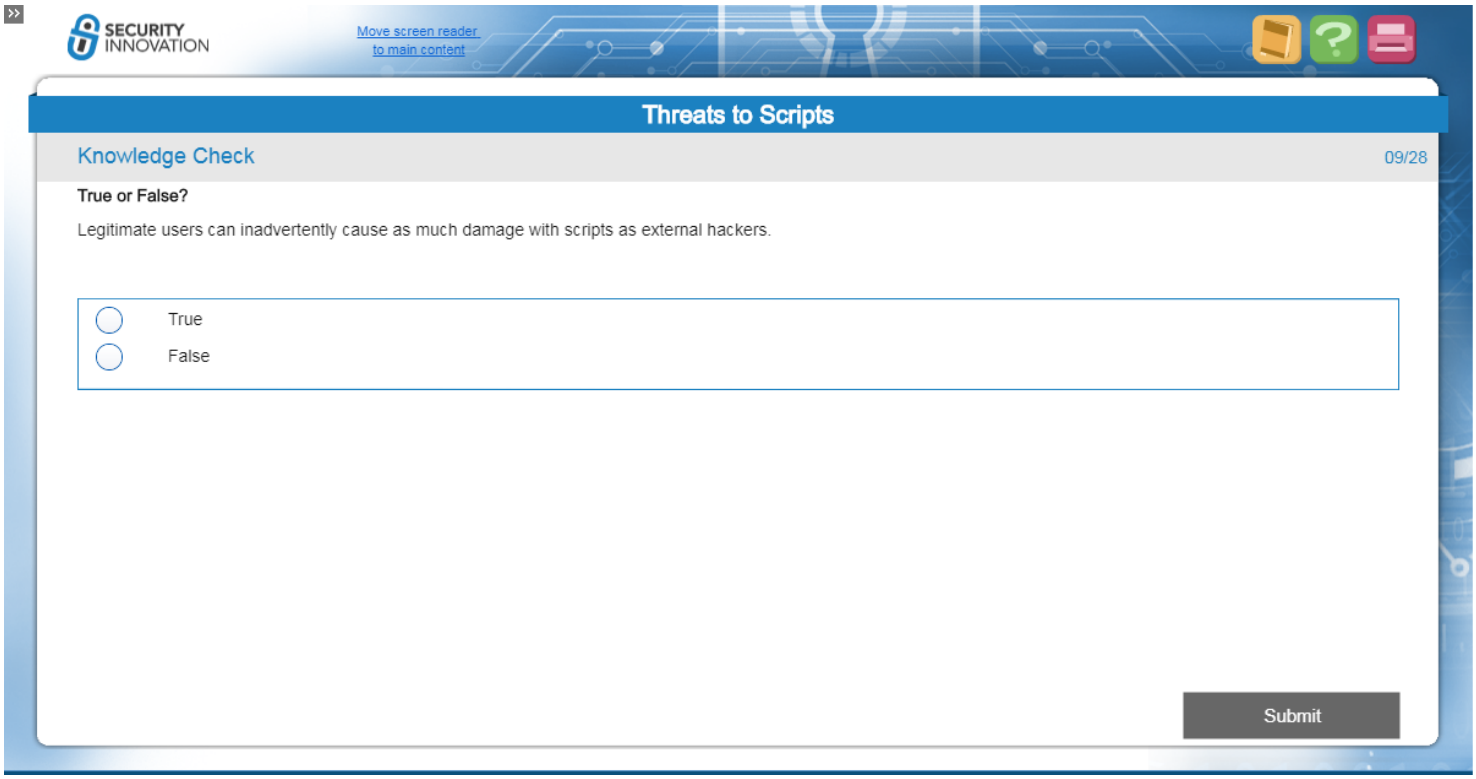
On Screen Text

Chain Reaction

A script's impact is not limited to only one of the aforementioned scenarios. Any combination of them is possible and can be caused by only a few lines of code.

Fortunately, all of these horrible scenarios can be avoided by consistent application of simple and logical guidelines.

Knowledge Check



On Screen Text

Knowledge Check

True or False?

Legitimate users can inadvertently cause as much damage with scripts as external hackers.

True

False

Module Summary

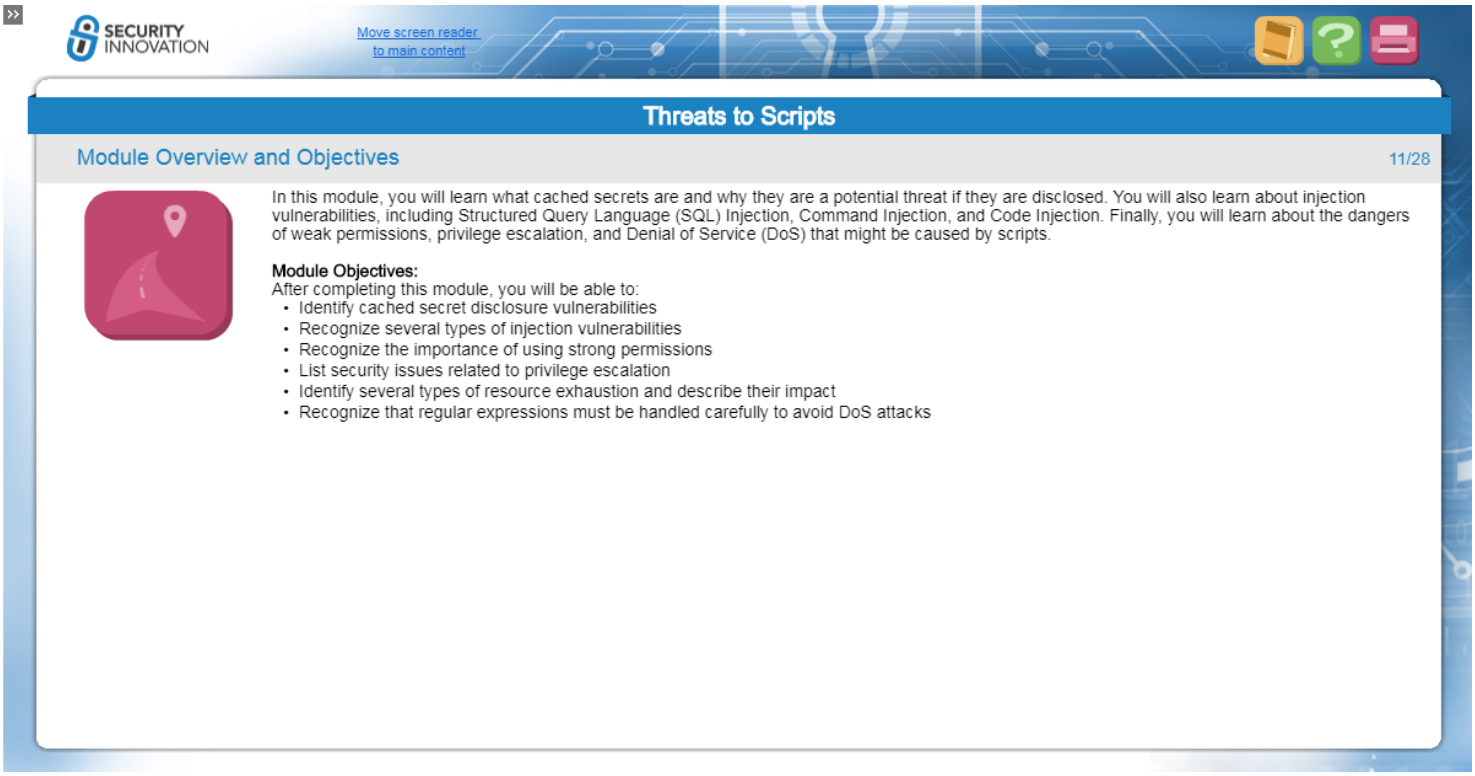
On Screen Text

Module Summary

In this module, you learned why scripting security matters. Scripting vulnerabilities can let hackers in, thus making various threats possible, including deleting files, bricking systems, ruining backups, disabling systems, and any combination thereof.

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

Module Overview and Objectives



Narration

In this module, you will learn what cached secrets are and why they are a potential threat if they are disclosed. You will also learn about injection vulnerabilities, including Structured Query Language (SQL) Injection, Command Injection, and Code Injection. Finally, you will learn about the dangers of weak permissions, privilege escalation, and Denial of Service (DoS).

After completing this module, you will be able to identify cached secret disclosure vulnerabilities, recognize several types of injection vulnerabilities, list security issues related to privilege escalation, identify several types of resource exhaustion and describe their impact, and recognize that regular expressions must be handled carefully to avoid DoS attacks.

On Screen Text

Module Overview and Objectives

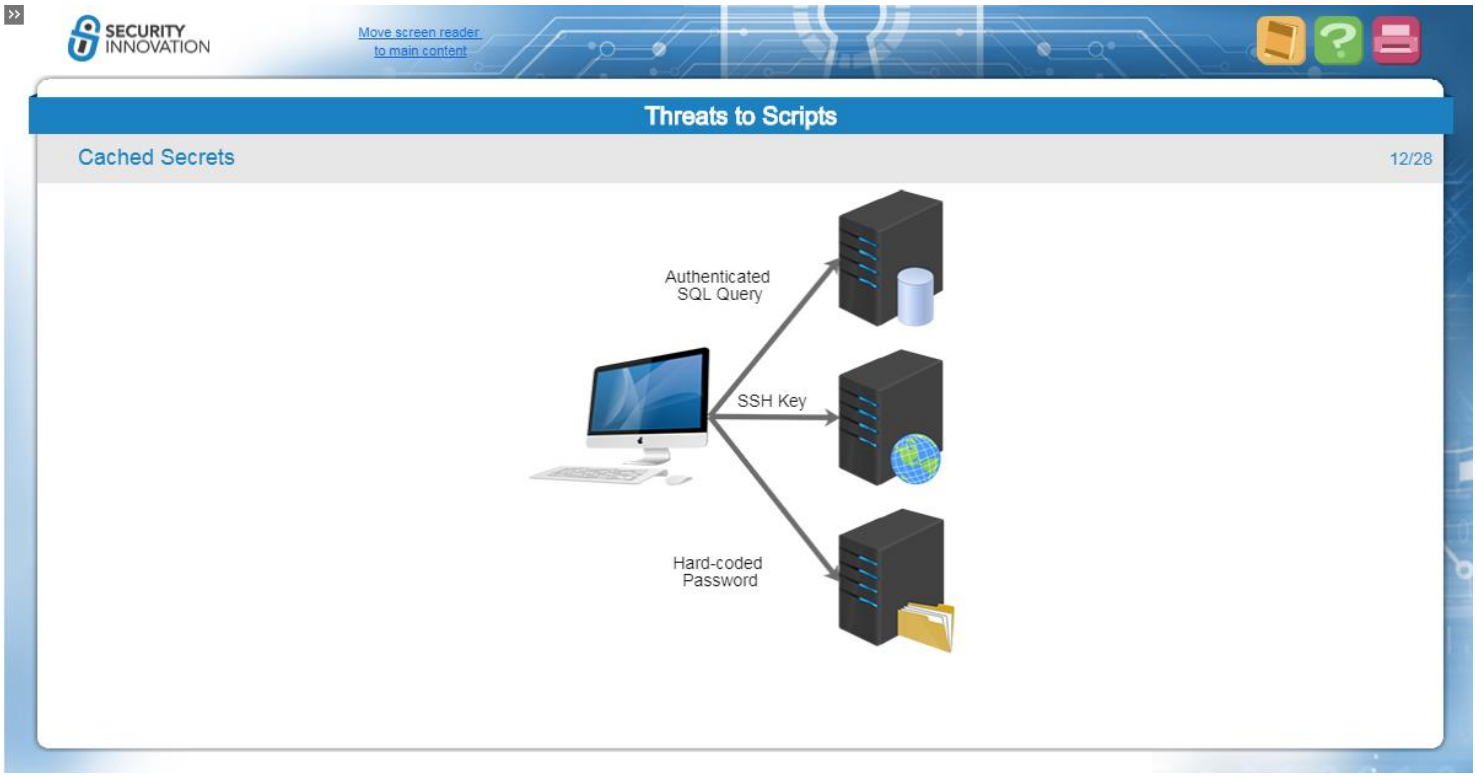
In this module, you will learn what cached secrets are and why they are a potential threat if they are disclosed. You will also learn about injection vulnerabilities, including Structured Query Language (SQL) Injection, Command Injection, and Code Injection. Finally, you will learn about the dangers of weak permissions, privilege escalation, and Denial of Service (DoS) that might be caused by scripts.

Module Objectives:
After completing this module, you will be able to:

- Identify cached secret disclosure vulnerabilities
- Recognize several types of injection vulnerabilities
- Recognize the importance of using strong permissions
- List security issues related to privilege escalation

- Identify several types of resource exhaustion and describe their impact
- Recognize that regular expressions must be handled carefully to avoid DoS attacks

Cached Secrets



Narration

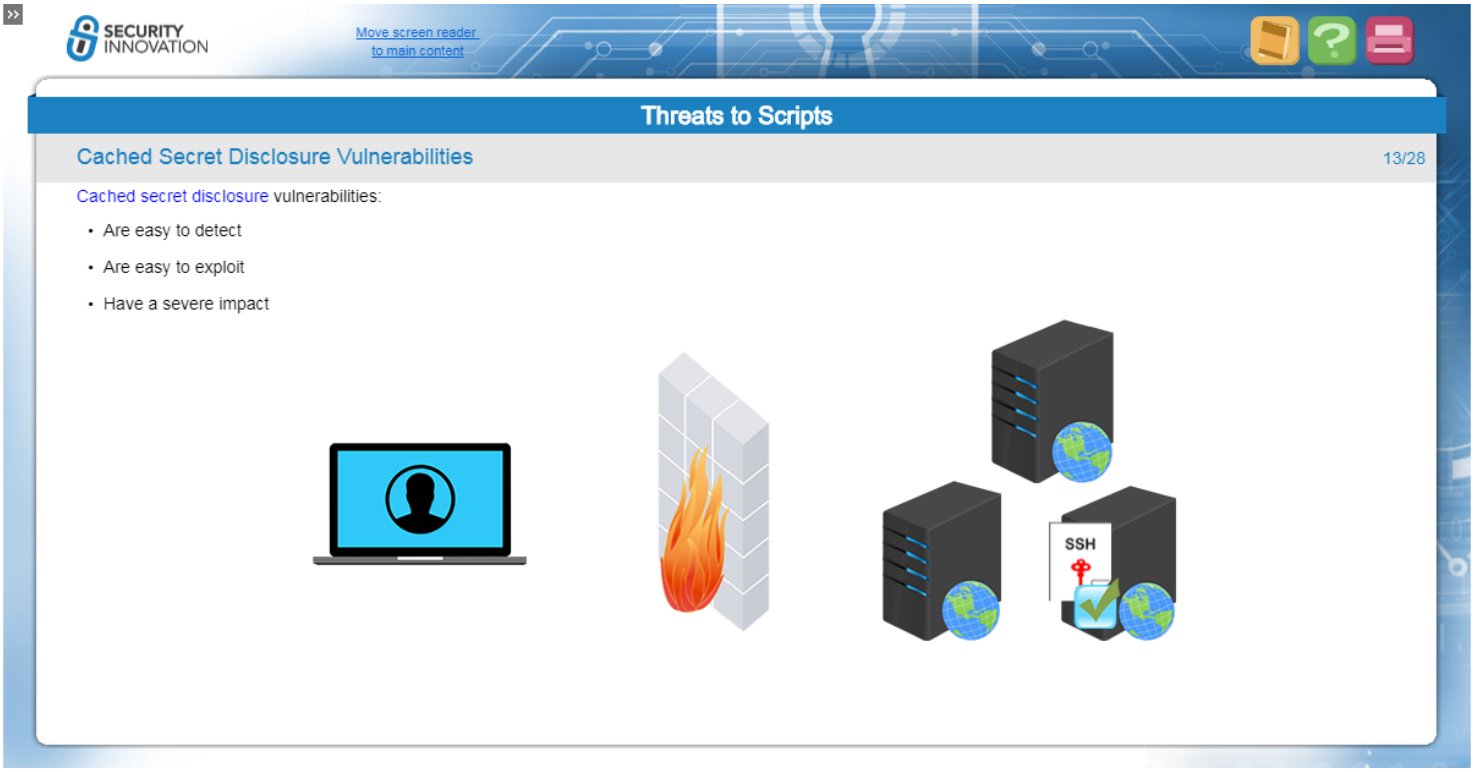
Cached secrets are authentication credentials or other secrets stored by an application or script. Examples include database connection strings such as authenticated Structured Query Language (SQL) queries, cryptographic keys such as Secure Shell (SSH) key files or private keys, and hard-coded passwords.

For web applications, these secrets are often present in connection strings used to connect to database servers. For shell scripts, authentication credentials might be hard-coded into scripts, or they might be stored on disk as SSH keys or some other type of key file.

On Screen Text

Cached Secrets

Cached Secret Disclosure Vulnerabilities



Narration

Disclosure of cached secrets is one of the most common and most serious vulnerability types that affects both shell and interpreted language scripts.

Some of the common ways that cached secrets are disclosed are when users upload their authentication credentials to public source code repositories for the entire world to see or otherwise place copies of secrets where they don't belong, such as on personal devices and portable storage media.

Exploiting cached secret disclosure vulnerabilities is extremely easy and has a severe impact. This vulnerability literally hands the keys to the attacker and all the attacker has to do is find what system the keys belong to.

If an attacker gets access to wherever they are stored, the attacker will be able to use the credentials to authenticate and gain unauthorized access to things like backup or file servers that often contain sensitive information.

Oftentimes the attacker will not have direct access to the system that the keys are for, which creates some hurdles, but this is still not an ideal situation as we would rather not give our authentication keys away to attackers.

On Screen Text

Cached Secret Disclosure Vulnerabilities

Cached secret disclosure vulnerabilities:

- Are easy to detect
- Are easy to exploit

- Have a severe impact

Injection Vulnerabilities

Narration

Injection vulnerabilities are very common in scripting languages because these languages make it very simple to concatenate strings and include user input in them. Scripting languages also make it very easy to invoke commands and execute database queries using these concatenated strings.

The result is a fertile field for injection vulnerabilities that allow attackers to hijack applications. There are many types of injection vulnerabilities and we are going to look at the most dangerous ones in the following slides. In addition to granting unauthorized access to attackers, privileged scripts also allow privilege escalation.

On Screen Text

Injection Vulnerabilities

Injection Vulnerabilities:

- Common in scripting languages
- Cause privilege escalation in privileged scripts
- Multiple types of injection are possible depending on context: SQL Injection, Command Injection, etc.

Preventing Injection Vulnerabilities

Narration

If a script does handle untrusted data, it should validate it and separate it from code. Validating all untrusted input is a good first line of defense against injection exploits. Validating data should even mitigate potential command injections in commands that are invoked by the script.

In general, injection vulnerabilities are prevented by separating data from command syntax. Bash automatically separates data in variables that are passed to commands as arguments from the name of the command.

For Perl and other scripting languages, it is important to use functions that separate command names and data passed to them into separate arguments when invoking commands and passing untrusted data to them. However, for scripting, especially shell scripting, such functions are often not available.

When these functions are not available, preventing injection vulnerabilities means encoding special characters in untrusted data for the correct context before concatenating it into command structures, or avoiding this concatenation altogether. It is much simpler to avoid concatenation than to encode consistently or correctly, unless trusted encoding (APIs) for the correct context are available.

For scripting, especially shell scripting, trusted encoding APIs are often not available, meaning that it is best to avoid concatenating data into sensitive contexts. That said, it is important to understand what encoding, also called escaping, of special characters for sensitive contexts means.

On Screen Text

Preventing Injection Vulnerabilities

To prevent injection vulnerabilities:

- Validate all input and data
- Use functions that prevent concatenation of untrusted data into command syntax
- Avoid concatenating data into syntax

SQL Injection

Narration

SQL Injection vulnerabilities are created whenever applications concatenate untrusted input with SQL syntax to form SQL queries.

A successful SQL Injection attack can have a very serious impact. It allows the attacker to extract or manipulate the data in the application database. In some cases, SQL Injection also results in command injection—this situation usually happens when stacked queries are enabled on the database server.

SQL Injection exploitation techniques are virtually identical for many different platforms, and are very simple to execute using widely available tools. Successful attacks provide a lot of leverage to attackers. High return for low effort is part of the reason why this vulnerability type is one of the greatest threats.

On Screen Text

SQL Injection

SQL injection (SQLi) vulnerabilities occur when an application concatenates untrusted data with SQL syntax to construct queries.
A successful SQL Injection attack allows the attacker to execute arbitrary queries on the application database with the full privileges of the application being exploited.

For example:

- Attackers might steal or modify database data, and bypass or change the authentication mechanisms and authorization rules of the database.
- In some configurations, SQL Injection results in command execution, granting virtually full control of the application server.

Command Injection

Narration

Applications and scripts often execute external commands as part of their functionality. If an attacker is able to manipulate the choice of external commands and/or their parameters, the attacker will be able to abuse this functionality to execute arbitrary commands.

In a small minority of cases, there will be limits on the scope of the available commands due to unintended peculiarities of the application's or the system's inner workings, but most of the time an attacker can take full control of the application using this vulnerability.

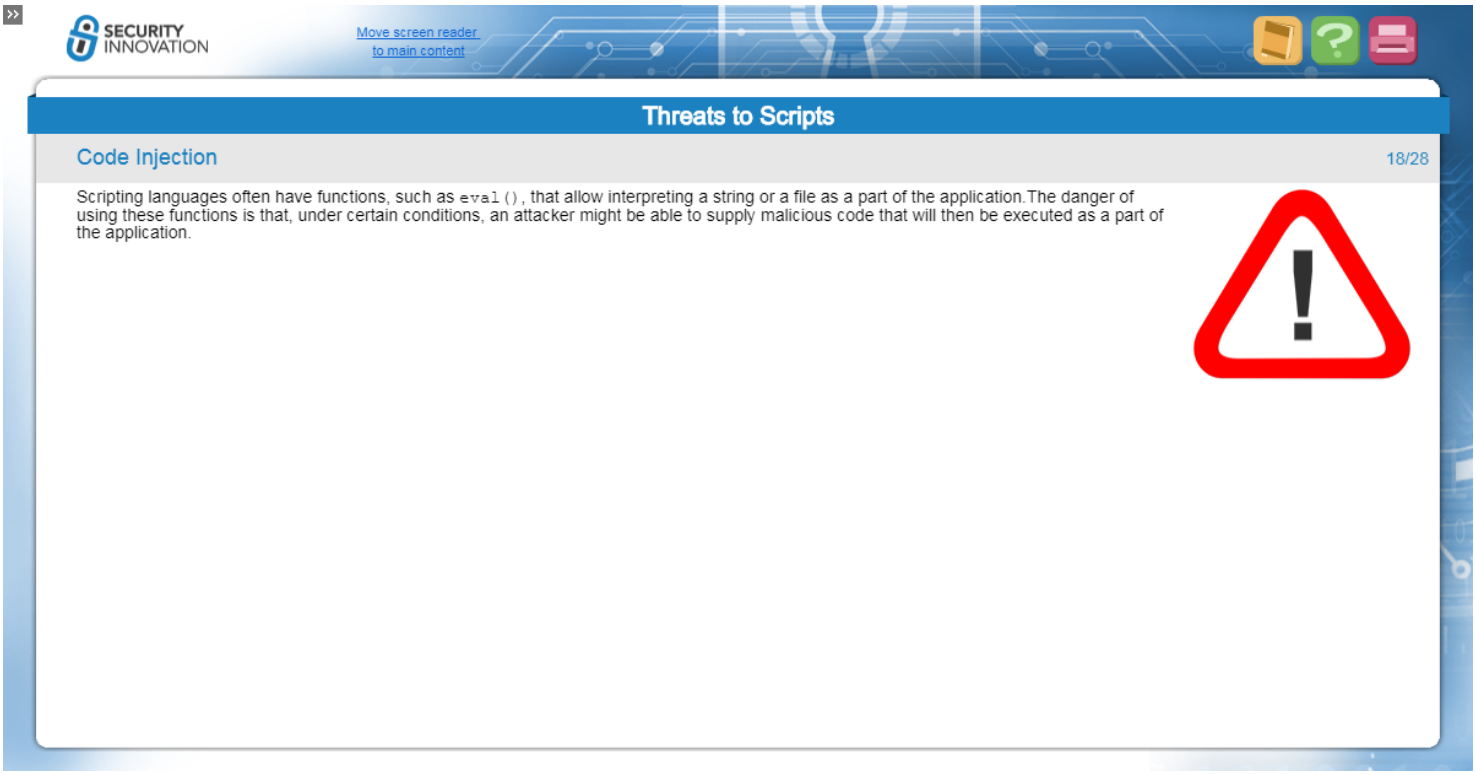
If the attacker is able to leverage additional vulnerabilities, or the system is not configured properly, the attacker will be able to take full control of the system.

A Command Injection vulnerability acts as a virtual backdoor for an attacker to use the application and the system for their purposes. Any data that is stored by or accessed by the application can also be compromised as a result of Command Injection.

On Screen Text

Command Injection
Command injection is one of the most dangerous and common vulnerability types. It is easy to find and exploit; its severity cannot be overstated.
Command injection occurs in applications that invoke external commands via the underlying operating system. It allows attackers to manipulate the commands that are being invoked, typically injecting data that will cause arbitrary commands to be executed in a sequence that will grant unauthorized access.
Command Injection is assigned Common Weakness Enumeration / CWE-78 and is included in the A1: Injection category as the top threat in every version of the Open Web Application Security Project (OWASP) Top 10 project.

Code Injection



Narration

Scripting languages often have functions, such as eval(), that allow interpreting a string or a file as a part of the application. The danger of using these functions is that, under certain conditions, an attacker might be able to supply malicious code that will then be executed as a part of the application.

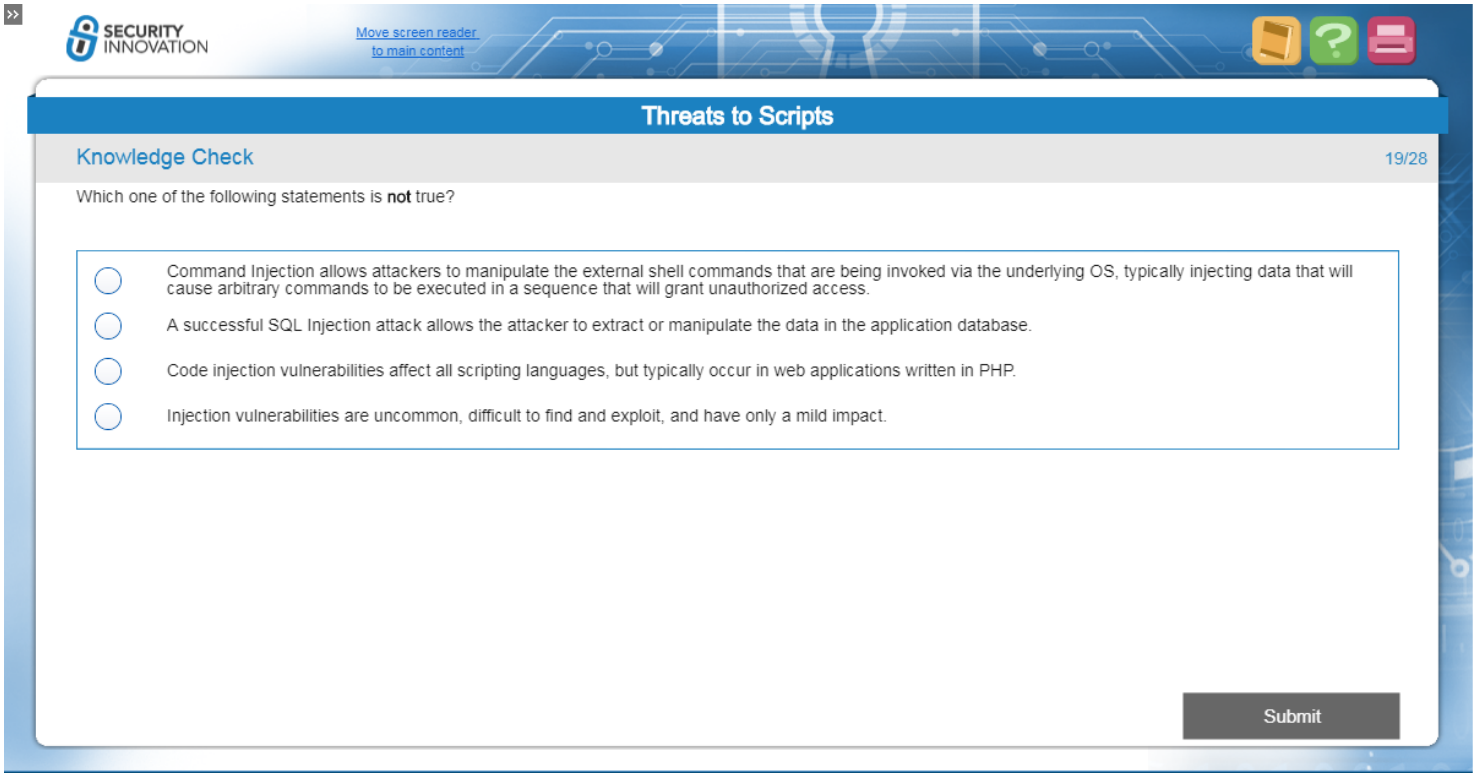
Code Injection gives the attacker full control of the application. Code Injection vulnerabilities affect all scripting languages, but typically occur in web applications written in PHP. To prevent Code Injection vulnerabilities, validate all input and avoid using dangerous functions, such as eval().

On Screen Text

Code Injection

Scripting languages often have functions, such as `eval()`, that allow interpreting a string or a file as a part of the application.The danger of using these functions is that, under certain conditions, an attacker might be able to supply malicious code that will then be executed as a part of the application.

Knowledge Check



On Screen Text

Knowledge Check

Which one of the following statements is **not** true?

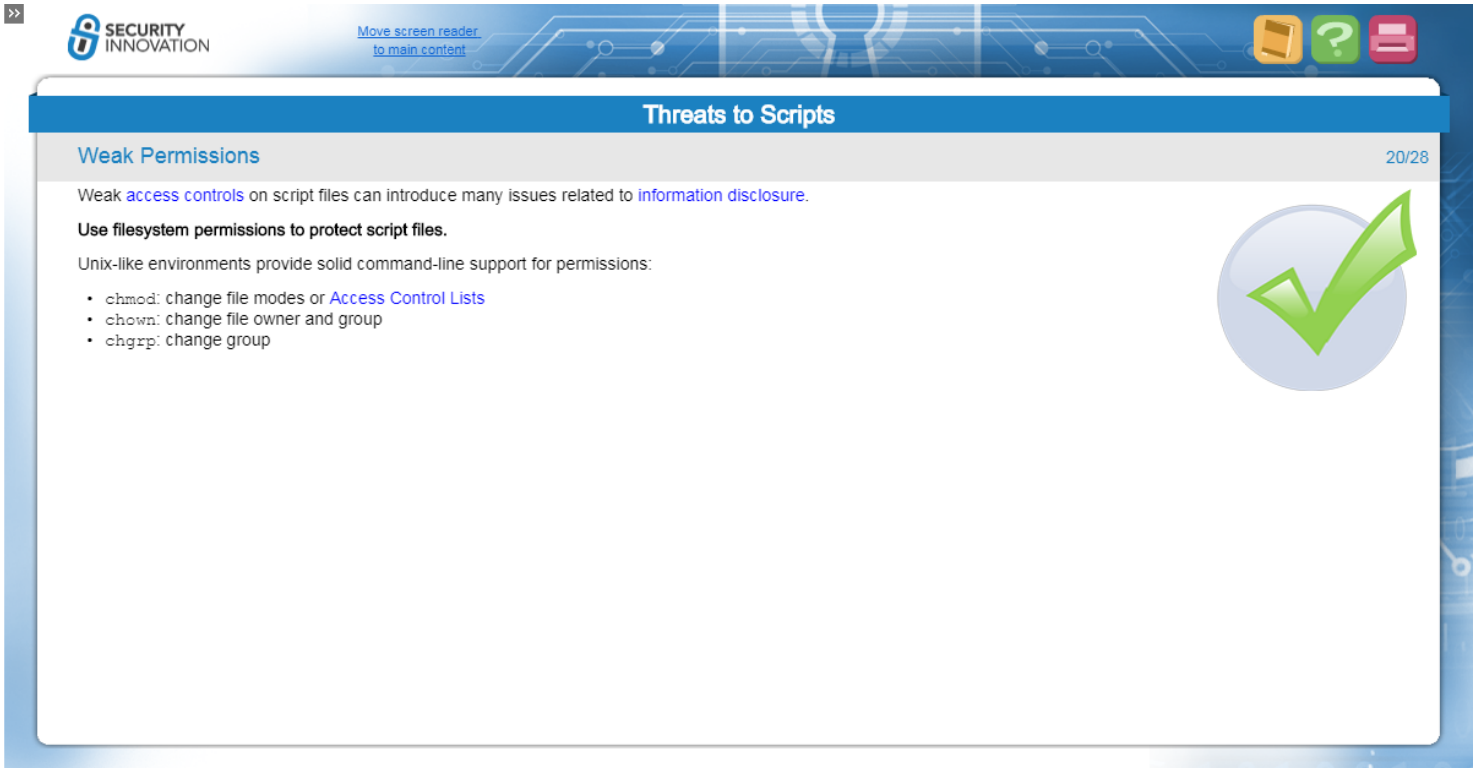
Command Injection allows attackers to manipulate the external shell commands that are being invoked via the underlying OS, typically injecting data that will cause arbitrary commands to be executed in a sequence that will grant unauthorized access.

A successful SQL Injection attack allows the attacker to extract or manipulate the data in the application database.

Code injection vulnerabilities affect all scripting languages, but typically occur in web applications written in PHP.

Injection vulnerabilities are uncommon, difficult to find and exploit, and have only a mild impact.

Weak Permissions



Narration

Weak access controls on script files can introduce a number of issues related to information disclosure. Mature modern operating systems provide access controls for filesystems in the form of permissions.

If script files are not protected with these controls, unauthorized users might view a script's contents and gather information from it, which might be useful for carrying out additional attacks.

If there are cached secrets stored by a script and these secrets are also not protected by filesystem permissions, then attackers will be able to recover the secrets from the script with incredible ease.

Unix-like operating systems provide a mature and consistent permissions model that is easily integrated into scripts. The chmod, chown, and chgrp commands can be used to change the permissions of files and directories from scripts. Dynamic programming languages also provide effective support for Unix-style filesystem permissions.

On Screen Text

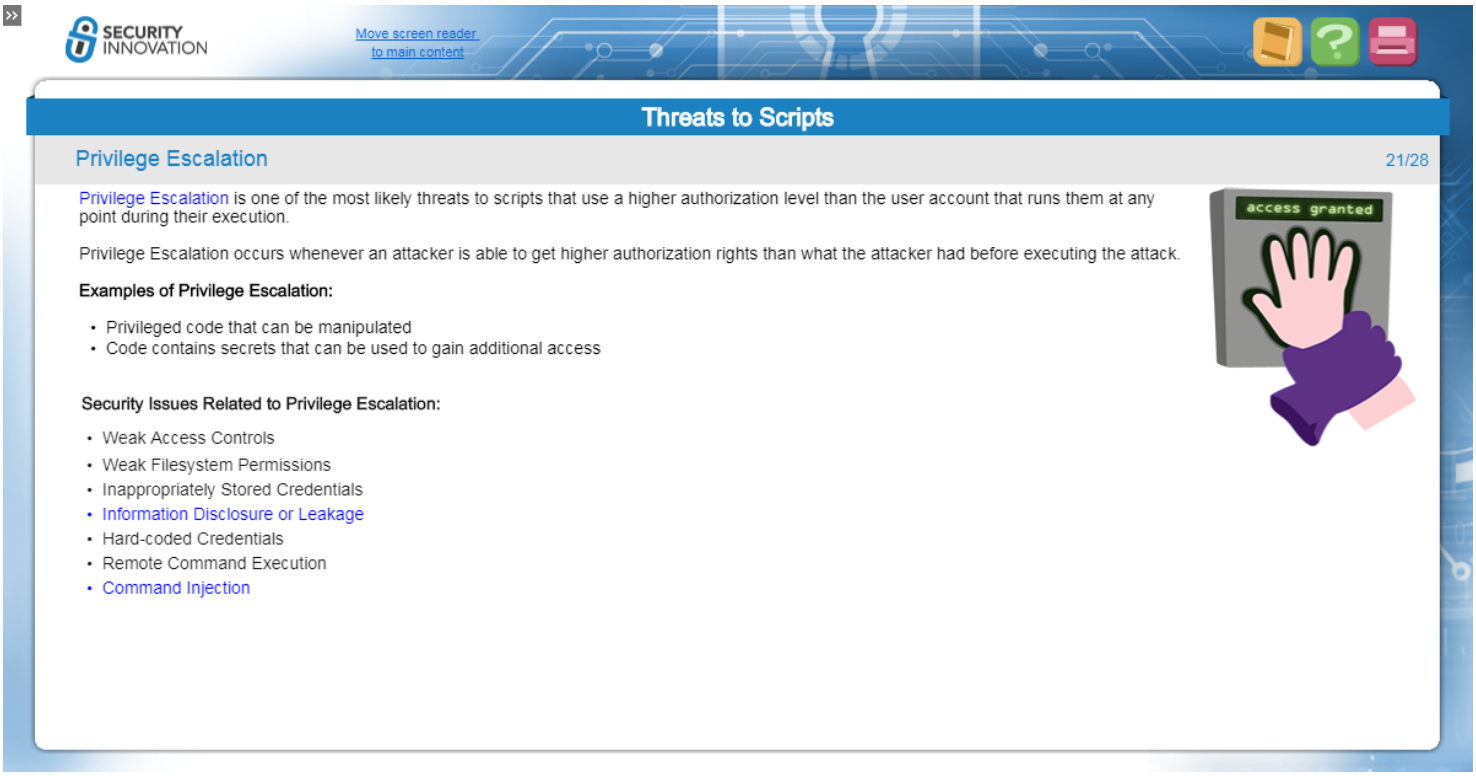
Weak Permissions

Weak access controls on script files can introduce many issues related to information disclosure.

Use filesystem permissions to protect script files.
Unix-like environments provide solid command-line support for permissions:

- `chmod`: change file modes or Access Control Lists
- `chown`: change file owner and group
- `chgrp`: change group

Privilege Escalation



Narration

Privilege Escalation is one of the most likely threats to scripts that use a higher authorization level than the user that invokes them or provides input to them at any point during their execution.

Privilege Escalation is really more of an impact than a vulnerability type, because there are many different and distinct types of bugs that can result in it, and these types of programming errors usually have their own names.

Privilege Escalation occurs whenever an attacker is able to get higher authorization rights than what the attacker had before executing the attack.

For example, if a portion of a script runs as root and the attacker is able to manipulate that section to execute arbitrary commands, then that would be Privilege Escalation and also a Command Injection vulnerability.

Another example is if the script includes the authentication key for a privileged remote Secure Shell (SSH) account and the attacker is able to view this key because of incorrect filesystem permissions set on the script files; that would also be a type of Privilege Escalation and a whole slew of other security issues that have multiple names: Weak Access Controls/Weak Filesystem Permissions, Inappropriately Stored Credentials/Information Disclosure/Information Leakage/Hard-coded Credentials, and Remote Command Execution/Command Injection.

On Screen Text

Privilege Escalation

Privilege Escalation is one of the most likely threats to scripts that use a higher authorization level than the user account that runs them at any point during their execution.

Privilege Escalation occurs whenever an attacker is able to get higher authorization rights than what the attacker had before executing the attack.

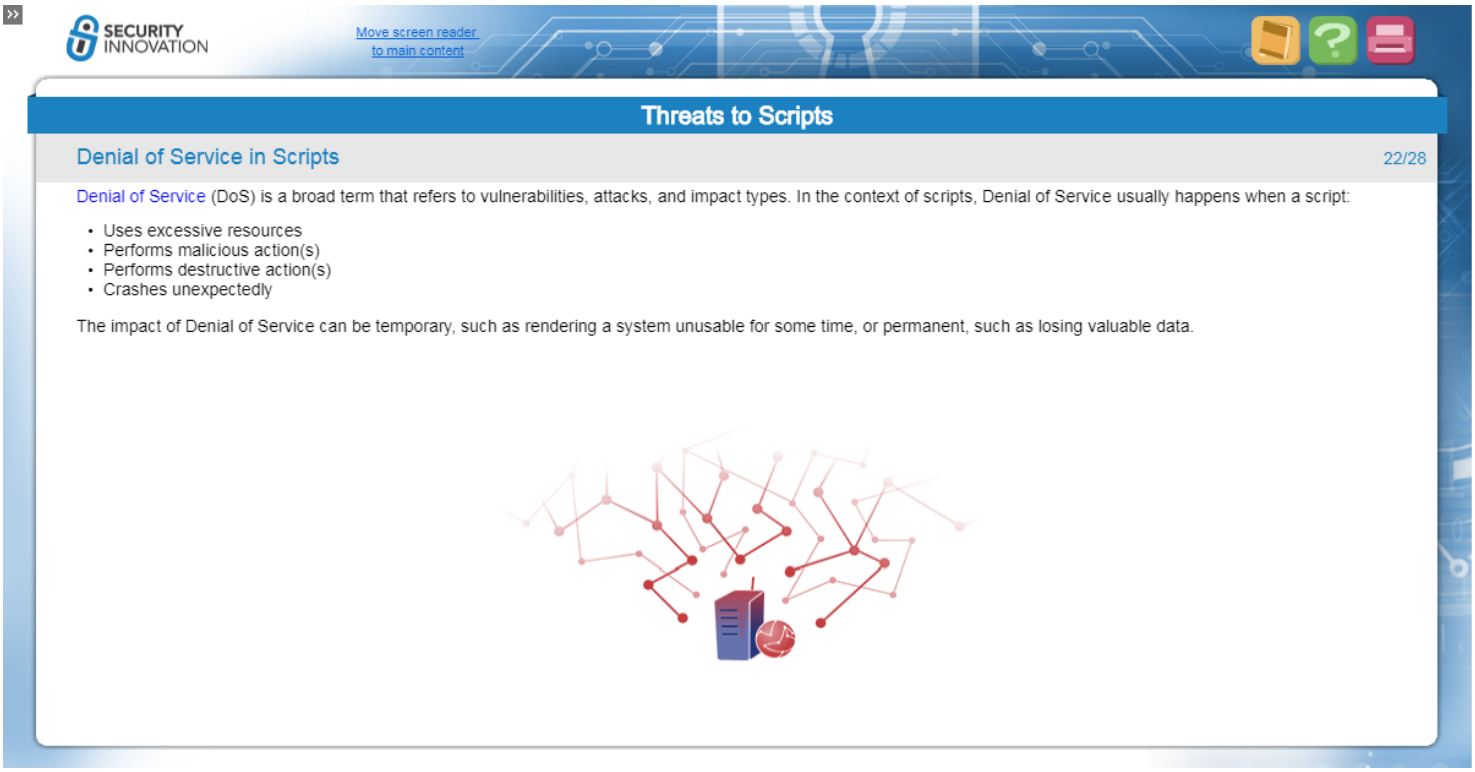
Examples of Privilege Escalation:

- Privileged code that can be manipulated
- Code contains secrets that can be used to gain additional access

Security Issues Related to Privilege Escalation:

- Weak Access Controls
- Weak Filesystem Permissions
- Inappropriately Stored Credentials
- Information Disclosure or Leakage
- Hard-coded Credentials
- Remote Command Execution
- Command Injection

Denial of Service in Scripts



Narration

In general, Denial of Service (DoS) refers to any vulnerability and attack set that results in the loss of availability of any information system component.

There are many different types of Denial of Service vulnerabilities and attacks.

Some DoS vulnerabilities are often present in the underlying platform itself.

Denial of service vulnerabilities in the context of scripting usually occur when the script can be manipulated to use an excessive amount of the system's computational resources, manipulated to crash, or manipulated to perform some kind of malicious and/or destructive action, such as deleting files.

Poorly written scripts might contain DoS vulnerabilities in themselves, or they might cause DoS conditions for the systems they are running on.

For example, a script might be written in a way that causes sensitive data to be erased by mistake, which is a DoS condition caused by the script. Another script might be written in a way that allows an attacker to erase sensitive files; this is an example of a script that contains a DoS vulnerability. When a script crashes, it might leave a mess of files behind and possibly some devices mounted or unmounted.

If the volume of this data increases with every script execution, eventually a Denial of Service condition will result due to the filesystem being filled with junk.

Clearly, the script needs to clean up during error handling, but be careful to only delete the right files.

It is worth noting that most DoS attacks are temporary in nature and effects, but the most serious attacks can have permanent impact, such as loss of valuable data.

On Screen Text

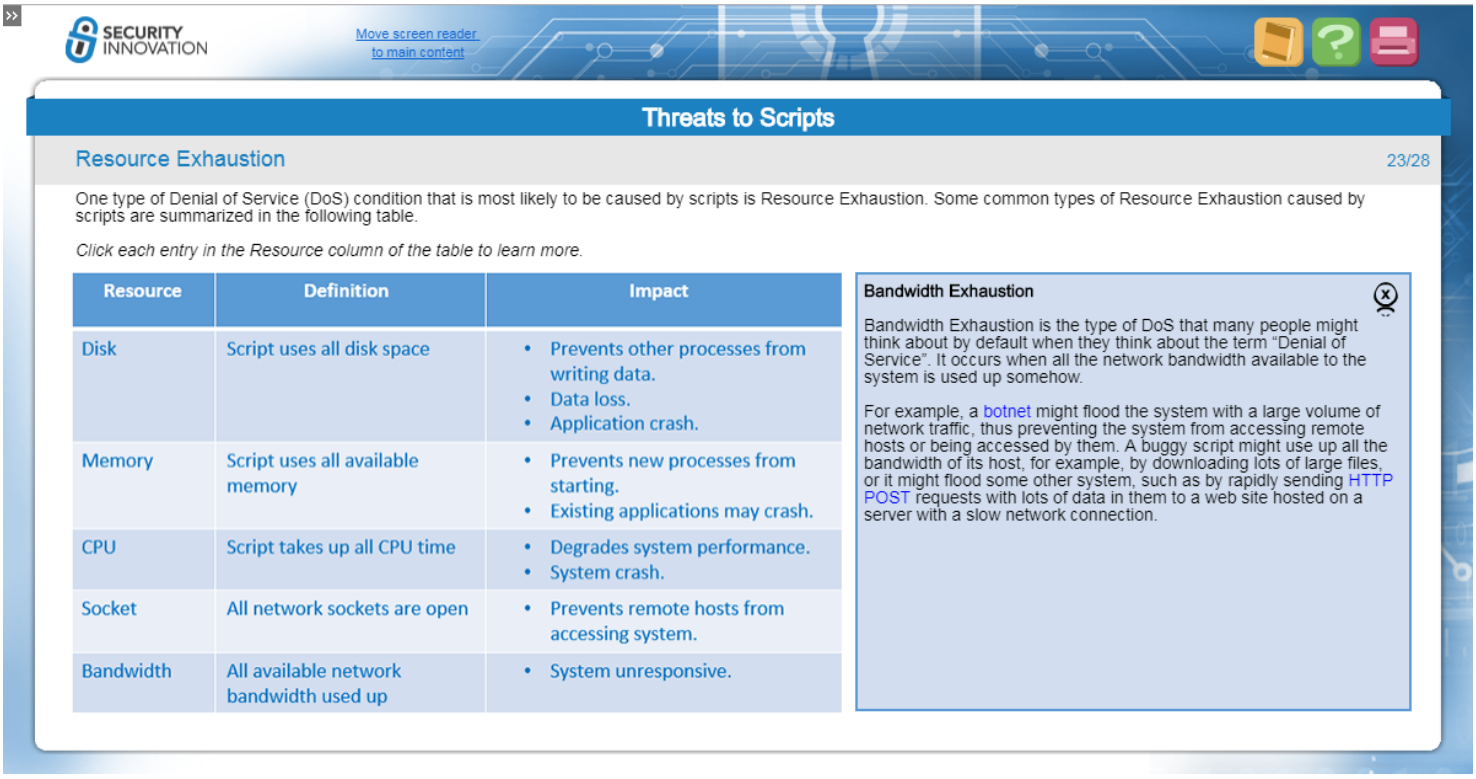
Denial of Service in Scripts

Denial of Service (DoS) is a broad term that refers to vulnerabilities, attacks, and impact types. In the context of scripts, Denial of Service usually happens when a script:

- Uses excessive resources
- Performs malicious action(s)
- Performs destructive action(s)
- Crashes unexpectedly

The impact of Denial of Service can be temporary, such as rendering a system unusable for some time, or permanent, such as losing valuable data.

Resource Exhaustion



Narration

One type of Denial of Service (DoS) condition that is most likely to be caused by scripts is Resource Exhaustion.

Resource Exhaustion itself has many subtypes, but what they all have in common is that they use up some limited resource on the system, thus preventing other processes from using that resource and often crashing the other processes or preventing new processes from running.

For example, a script that runs many CPU-intensive commands can use up all of the available CPU time and reduce performance for the other processes on the system.

While this situation is always inconvenient, it is particularly troublesome in a multi-user environment where performance might degrade to the point of the system becoming unusable for everyone, especially if multiple users are running highly inefficient scripts at the same time.

Some common types of Resource Exhaustion are: Disk Exhaustion, Memory Exhaustion, CPU Exhaustion, Socket Exhaustion, and Bandwidth Exhaustion.

On Screen Text

Resource Exhaustion

One type of Denial of Service (DoS) condition that is most likely to be caused by scripts is Resource Exhaustion. Some common types of Resource Exhaustion caused by scripts are summarized in the following table.

Click each entry in the Resource column of the table to learn more.

Disk Exhaustion

Disk Exhaustion occurs when a script uses up all the disk space, preventing other processes from writing data. This causes loss of unsaved data and often causes applications to crash, especially when all the space on the system partition is used up.

Memory Exhaustion

Memory Exhaustion occurs when a script uses up all the available memory; this prevents new processes from starting and often causes the existing processes to crash.

CPU Exhaustion

CPU Exhaustion occurs when a script takes up all the CPU time, which usually significantly degrades performance for the entire system. In some cases, the entire system might freeze or crash.

Socket Exhaustion

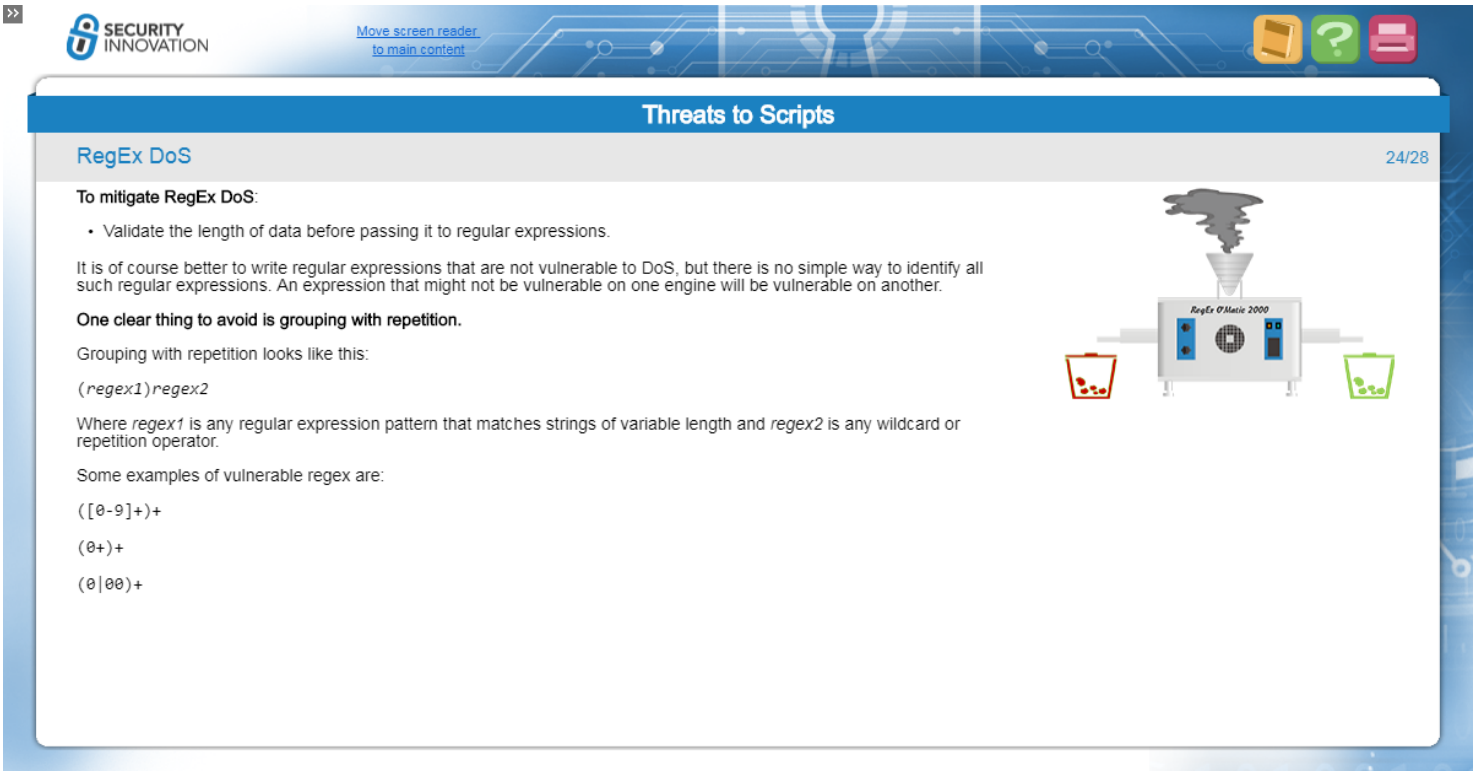
Socket Exhaustion occurs if all the network sockets are opened. Although this is not very likely in scripts, it can happen. Socket Exhaustion prevents remote hosts from accessing the system over the network.

Bandwidth Exhaustion

Bandwidth Exhaustion is the type of DoS that many people might think about by default when they think about the term “Denial of Service”. It occurs when all the network bandwidth available to the system is used up somehow.

For example, a botnet might flood the system with a large volume of network traffic, thus preventing the system from accessing remote hosts or being accessed by them. A buggy script might use up all the bandwidth of its host, for example, by downloading lots of large files, or it might flood some other system, such as by rapidly sending HTTP POST requests with lots of data in them to a web site hosted on a server with a slow network connection.

RegEx DoS



Narration

Regular expression denial of service vulnerabilities occur when a script applies regular expressions to data in a manner that requires prohibitive computational resources to complete.

Regular expression denial of service attacks result in the vulnerable application taking up a lot of CPU time and potentially causing the system to become unresponsive.

To prevent regular expression denial of service attacks, check the length of data before passing it to regular expressions.

It is of course better to write regular expressions that are not vulnerable to DoS, but there is no simple way to identify all such regular expressions.

One clear thing to avoid is grouping with repetition.

The computational complexity of vulnerable regular expressions increases exponentially with the size of the input.

If input length is restricted to a very short size, then there will be no noticeable impact.

For example, it is very hard to cause regex denial of service with input length of three characters.

The exact length before denial of service occurs is different for different expressions and systems.

Even though checking length might not completely prevent regular expression denial of service vulnerabilities, it is useful to mitigate their impact even in cases where vulnerable expressions have not been identified in advance.

On Screen Text

RegEx DoS

Regular Expression (regex) Denial of Service vulnerabilities:

- Are caused by malformed data passed to RegEx
- Result in CPU or Memory Exhaustion DoS

To mitigate RegEx DoS:

- Validate the length of data before passing it to regular expressions.

It is of course better to write regular expressions that are not vulnerable to DoS, but there is no simple way to identify all such regular expressions. An expression that might not be vulnerable on one engine will be vulnerable on another.

One clear thing to avoid is grouping with repetition.

Grouping with repetition looks like this:

`(`*`regex1`*`)`*`regex2`*

Where *regex1* is any regular expression pattern that matches strings of variable length and *regex2* is any wildcard or repetition operator.

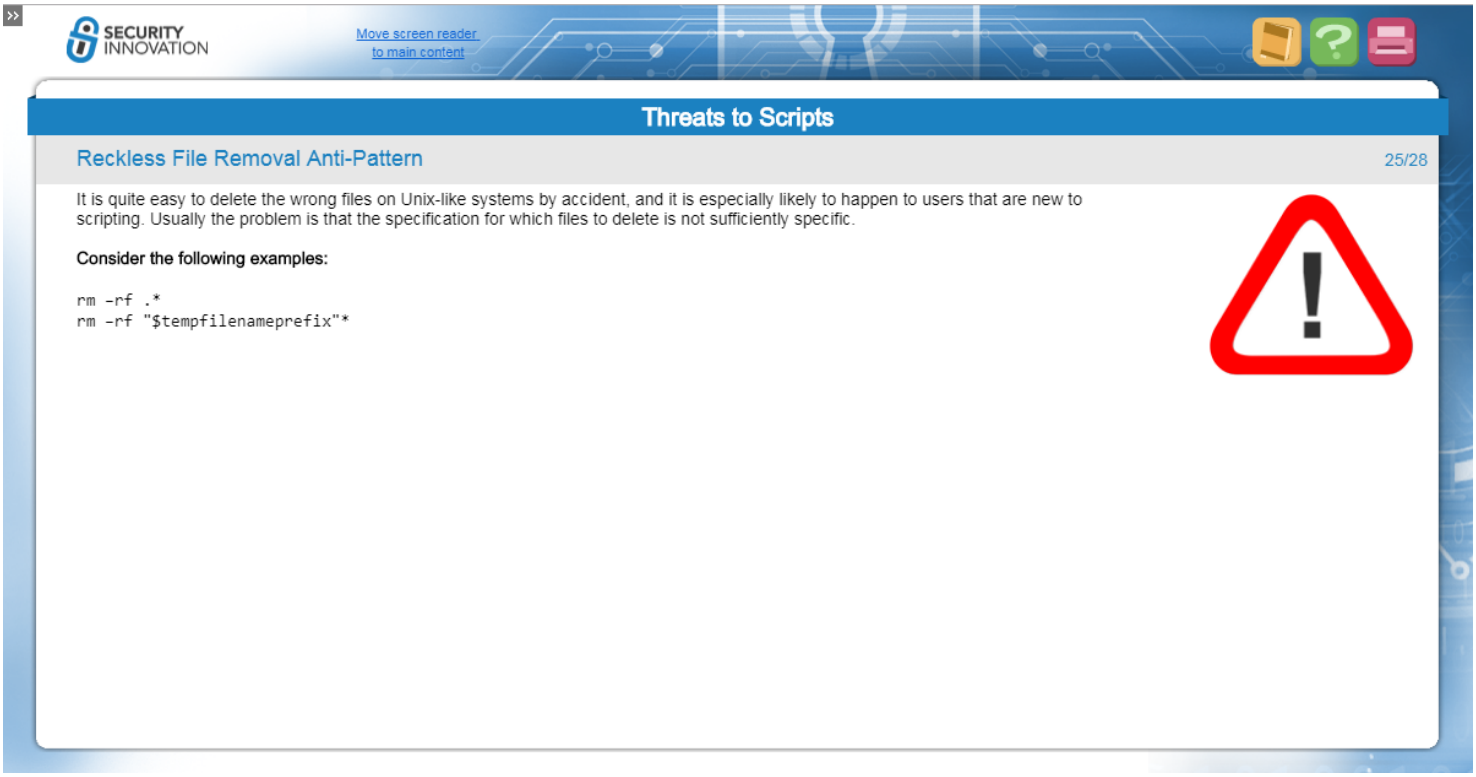
Some examples of vulnerable regex are:

```
([0-9]+)+

(0+)+

(0|00)+
```

Reckless File Removal Anti-Pattern



Narration

It is quite easy to delete the wrong files on Unix-like systems by accident.

This mistake is especially likely to happen to users that are new to scripting because they often don't realize the full power of the rm command combined with shell globbing and wildcards.

The usual problem is that the user is trying to delete multiple files with one command and doesn't made the pattern sufficiently specific. Consider these examples.

In the first example, the user is trying to delete the hidden files in the directory.

The reality is that the rm command will merrily go on to recursively delete all the files on the system when the command is run.

In the second example, the user is trying to delete some temporary files that all start with the same prefix.

If the variable containing the prefix is somehow empty, then all the files and subdirectories in the current directory will be erased, most likely including the script itself.

If an attacker is able to manipulate the value of this variable, then the script can be manipulated into deleting arbitrary files that the user running the script has the filesystem permissions to delete.

If the prefix does have a value but is not specific enough, then other files that happen to have names that start with the same pattern as the temporary files will also be deleted.

On Screen Text

Reckless File Removal Anti-Pattern

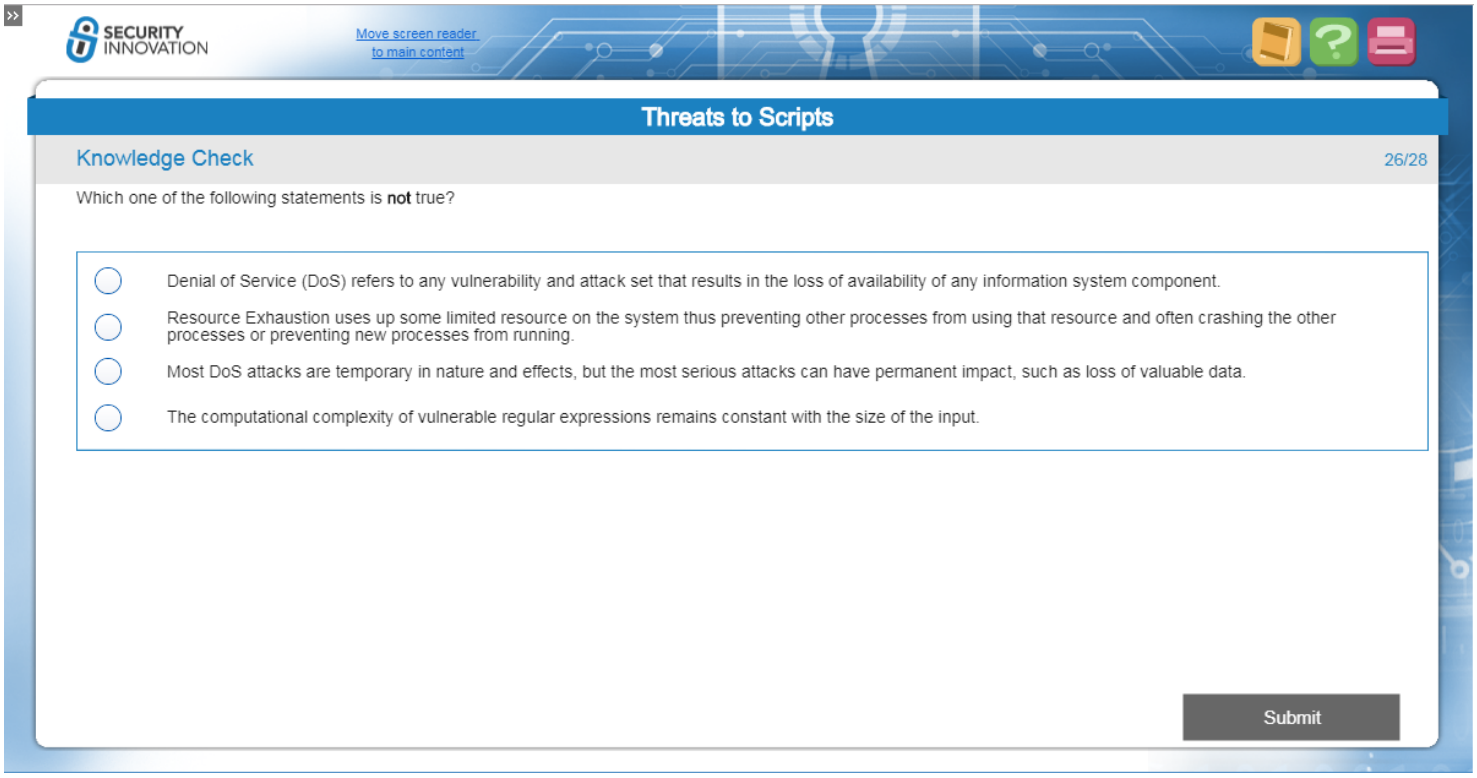
It is quite easy to delete the wrong files on Unix-like systems by accident, and it is especially likely to happen to users that are new to scripting. Usually the problem is that the specification for which files to delete is not sufficiently specific.

Consider the following examples:

```
rm –rf .*

rm –rf "$tempfilenameprefix"*
```

Knowledge Check



On Screen Text

Knowledge Check

Which one of the following statements is **not** true?

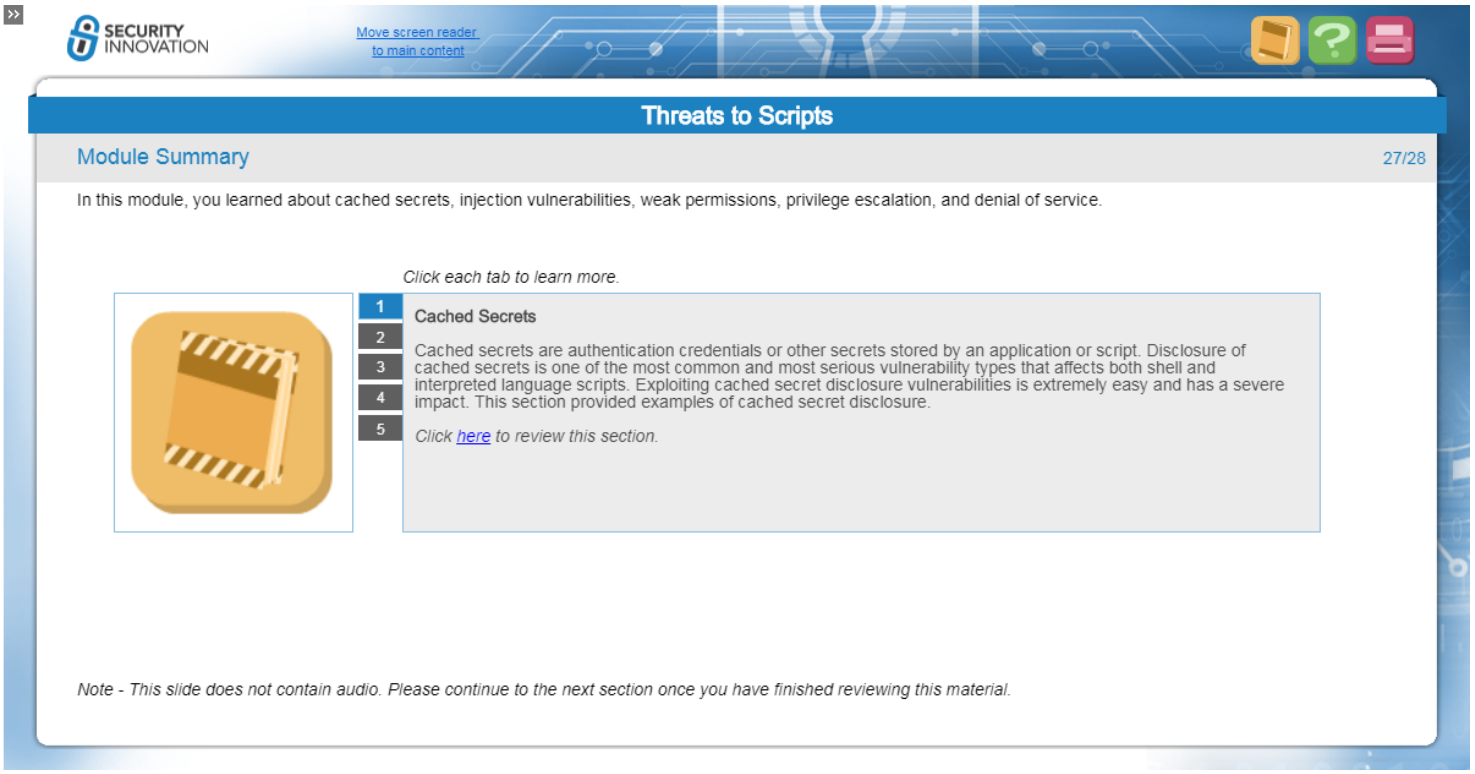
Denial of Service (DoS) refers to any vulnerability and attack set that results in the loss of availability of any information system component.

Resource Exhaustion uses up some limited resource on the system thus preventing other processes from using that resource and often crashing the other processes or preventing new processes from running.

Most DoS attacks are temporary in nature and effects, but the most serious attacks can have permanent impact, such as loss of valuable data.

The computational complexity of vulnerable regular expressions remains constant with the size of the input.

Module Summary



Narration

On Screen Text

Module Summary

In this module, you learned about cached secrets, injection vulnerabilities, weak permissions, privilege escalation, and denial of service. *Click each tab to learn more.*

Cached Secrets

Cached secrets are authentication credentials or other secrets stored by an application or script. Disclosure of cached secrets is one of the most common and most serious vulnerability types that affects both shell and interpreted language scripts. Exploiting cached secret disclosure vulnerabilities is extremely easy and has a severe impact. This section provided examples of cached secret disclosure.

Click here to review this section.

Injection Vulnerabilities

Injection vulnerabilities are very common in scripting languages because these languages make it very simple to concatenate strings and include user input in them. Scripting languages also make it easy to invoke commands and execute

database queries using these concatenated strings. The result is a fertile field for injection vulnerabilities that allow attackers to hijack applications. This section introduced several types of injection vulnerability.

Click here to review this section.

Weak Permissions

Weak access controls on script files can introduce many issues related to information disclosure. This section described access controls and permissions provided by mature modern operating systems to mitigate this issue.

Click here to review this section.

Privilege Escalation

Privilege Escalation is one of the most likely threats to scripts that use a higher authorization level than the user that invokes them or provides input to them at any point during their execution. This section enumerated security issues related to Privilege Escalation.

Click here to review this section.

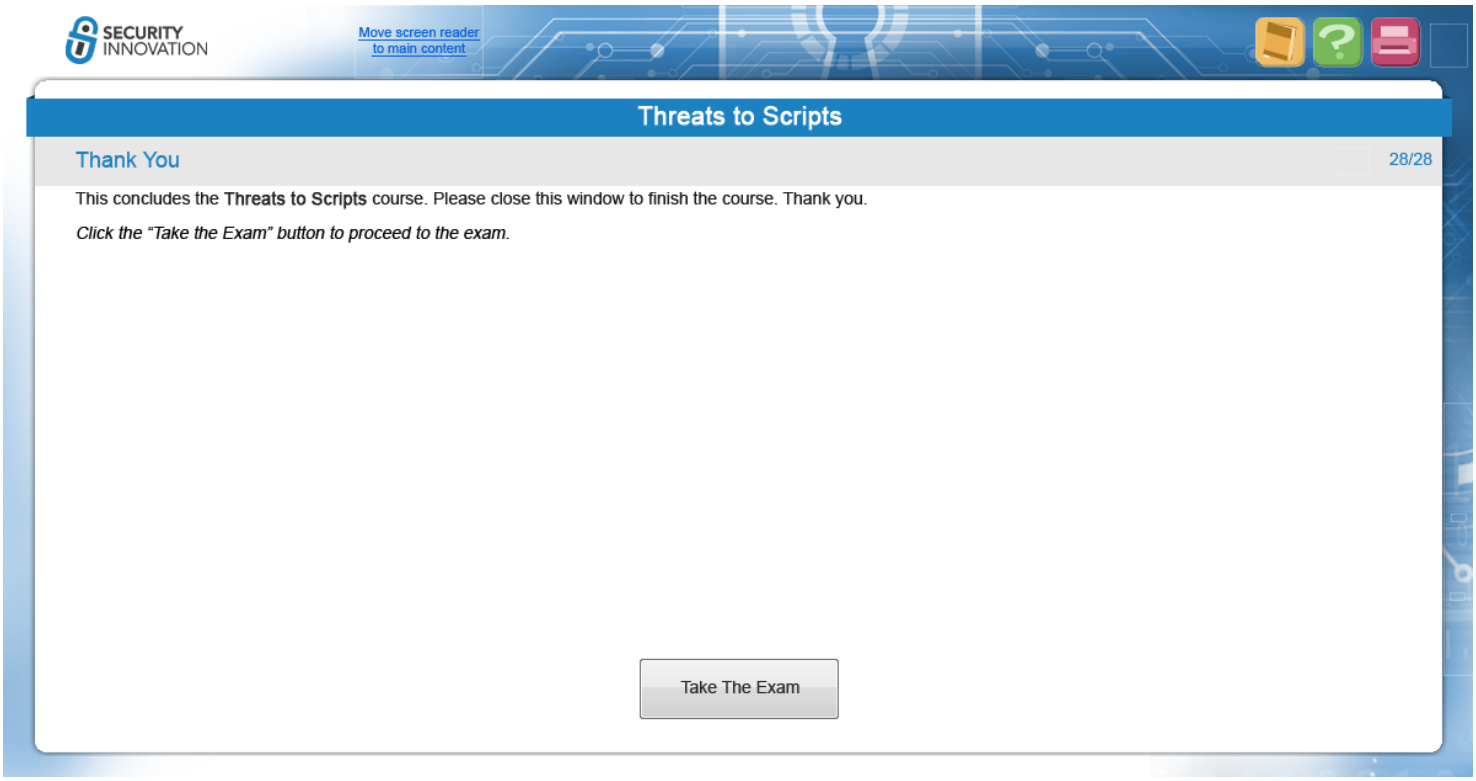
Denial of Service

Poorly written scripts might contain DoS vulnerabilities in themselves, or they might cause DoS conditions for the systems they are running on. This section discussed DoS due to resource exhaustion, Regular Expression DoS, and reckless removal of files.

Click here to review this section.

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

Thank You



On Screen Text

Thank You

This concludes the **Threats to Scripts** course. Please close this window to finish the course. Thank you.

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