ATTACKING OPENSTACK

INFRASTRUCTRE SECURE DEVELOPMENT

WWW.DEVSECOPSGUIDES.COM

Attacking OpenStack

· Aug 5, 2024 · 18 min read

Table of contents

Apply Restrictive File Permissions

- Incorrect Example
- Writing Files with Python
- Correct Example
- Secure File Creation in Python
- Verify Ownership and Group

Avoid Dangerous File Parsing and Object Serialization Libraries

- Incorrect Usage
- Correct Usage

Python Pipes to Avoid Shells

- Incorrect
- Correct

Unvalidated URL redirect

- Incorrect
- Correct

Validate Certificates on HTTPS Connections to Avoid Man-in-the-Middle Attacks

- Incorrect
- Correct

Create, Use, and Remove Temporary Files Securely

- Python Temporary File Handling
- Incorrect
- Correct

Restrict Path Access to Prevent Path Traversal

- Incorrect

Correct

Use Subprocess Securely

- Incorrect
- Correct

Parameterize Database Queries

- SQLAlchemy
  - Incorrect
  - Correct
- MySQL
  - Incorrect
  - Better
  - Correct
- PostgreSQL (Psycopg2)
  - Incorrect
  - Correct

Protect Sensitive Data in Config Files from Disclosure

- Incorrect
- Correct
- Consequences
  - Example Log Entries

Use Secure Channels for Transmitting Data

- Clear Example
- Less Obvious Example
  - Incorrect

Escape User Input to Prevent XSS Attacks

- Incorrect
  - Correct

Resources

Show less

Attacking OpenStack, an open-source cloud computing platform, involves exploiting vulnerabilities in its components and configuration to gain unauthorized access or disrupt services. One common attack vector is through the OpenStack Dashboard (Horizon), which can be susceptible to cross-site scripting (XSS) and cross-site request forgery (CSRF) attacks if not properly secured. Attackers can also target the API endpoints, which are often exposed to the internet. By exploiting insecure API calls or leveraging weak authentication mechanisms, attackers can manipulate cloud resources, potentially leading to data breaches or service interruptions. Furthermore, insecure configurations, such as using default passwords or failing to apply security patches, provide additional opportunities for exploitation.

Another critical aspect of attacking OpenStack is leveraging vulnerabilities within its network and storage components. OpenStack's Neutron, responsible for networking, can be targeted through attacks on the underlying network infrastructure, such as VLAN hopping or manipulating network policies to intercept or reroute traffic. Similarly, the Cinder and Swift services, which handle block and object storage respectively, can be exploited if they are misconfigured, allowing attackers to access or corrupt stored data. Privilege escalation attacks are also a significant threat, where attackers can exploit vulnerabilities in the underlying hypervisor or the Keystone identity service to gain higher-level access, thereby compromising the entire cloud environment. Ensuring comprehensive security measures, regular updates, and stringent access controls are crucial to

Apply Restrictive File Permissions

To ensure the security of files containing sensitive information, such as configuration files with passwords, it is crucial to apply restrictive file permissions. This prevents unauthorized access and potential exploitation through information disclosure or code execution. Files should be created with permissions that limit access to only the owning user or service and group, avoiding any world/other access.

Incorrect Example

Consider a configuration file for a service named "secureserv" stored in `secureserv.conf`. If the file permissions are set as follows:

```
COPY
ls -l secureserv.conf
-rw-rw-rw-   1 secureserv  secureserv   6710 Feb 17 22:00
secureserv.conf
```

The file permissions are set to `666`, allowing read and write access to everyone on the system. This is insecure because it exposes the sensitive information contained within the configuration file to all users.

Writing Files with Python

When writing files in a Unix-like operating system using Python, the file permissions are influenced by the system's umask setting. By default, this may be too permissive for sensitive data:

```
COPY
with open('testfile.txt', 'w') as fout:
    fout.write("secrets!")
```

Checking the permissions of the created file:

```
ls -l testfile.txt
-rw-r--r--  1 user  staff  4 Feb 19 10:59 testfile.txt
```

The file is readable by others, which is not secure for sensitive information.

Correct Example

To correct this, set restrictive permissions to ensure only the owning user can read and write to the file. Update the file permissions as follows:

```
chmod 0600 secureserv.conf
ls -l secureserv.conf
-rw-------   1 secureserv  secureserv   6710 Feb 17 22:00
secureserv.conf
```

Secure File Creation in Python

You can securely create a file in Python with appropriate permissions using `os.open` with specific flags:

```
import os

flags = os.O_WRONLY | os.O_CREAT | os.O_EXCL
with os.fdopen(os.open('testfile.txt', flags, 0o600), 'w') as fout:
    fout.write("secrets!")
```

Verify Ownership and Group

It is essential to verify the ownership and group of the file, ensuring only the necessary users have access. Follow the principle of least privilege: if group access is not needed, do not grant it. This approach minimizes the risk of unauthorized access and enhances the overall security of sensitive data files.

Avoid Dangerous File Parsing and Object Serialization Libraries

Many common libraries used for reading configuration files and deserializing objects can be dangerous because they may allow execution of arbitrary code. Libraries such as PyYAML and pickle do not provide strong separation of data and code by default, enabling code to be embedded inside the input.

Often, inputs to these libraries are untrusted or only partially trusted, coming from configuration files or provided via REST APIs. For instance, YAML is frequently used for configuration files but can also contain embedded Python code, which could provide an attacker with a method to execute code.

Many, but not all, of these libraries offer safe interfaces that disable features enabling code execution. It is crucial to use the safe functions to load input. The most obvious function to use is often not the safe one, so always check the documentation for libraries not covered here.

Common libraries used to load data in Python programs include YAML, pickle, and eval. While PyYAML offers a safe way to load data, pickle and eval do not.

Module	Problem	Use	Avoid
PyYAML	Allows creating arbitrary Python objects.	`yaml.safe_load`	`yaml.load`
pickle	Allows creating arbitrary Python objects.	Do not use	`pickle.load`, `pickle.loads`

Module	Problem	Use	Avoid
cPickle	Allows creating arbitrary Python objects.	Do not use	`cPickle.load`, `cPickle.loads`
eval	Runs all input as Python code	Do not use	`eval`
exec	Runs all input as Python code (Python 3.x)	Do not use	`exec`

Incorrect Usage

`yaml.load` is the obvious function to use but it is dangerous:

```
import yaml
import pickle

conf_str = '''
!!python/object:__main__.AttackerObj
key: 'value'
'''
conf = yaml.load(conf_str)
```

Using `pickle` or `cPickle` with untrusted input can result in arbitrary code execution:

```
import pickle
import cPickle

user_input = "cos\nsystem\n(S'cat /etc/passwd'\ntR.'\ntR."
cPickle.loads(user_input)  # results in code execution
pickle.loads(user_input)   # results in code execution
```

Similarly, `eval` and `exec` are difficult to use safely with input from an untrusted source:

COPY

```
user_input = "os.system('cat /etc/passwd')"
eval(user_input)  # execute python expressions

user_input = "import os; os.system('cat /etc/passwd')"
exec(user_input)  # execute _any_ python code
```

Correct Usage

Here we use PyYAML's safe YAML loading function:

COPY

```
import yaml

conf_str = '''
- key: 'value'
- key: 'value'
'''
conf = yaml.safe_load(conf_str)
```

There is no safe alternative for `pickle.load`. It is generally advisable to avoid using `pickle` for serialization of data objects altogether, especially when dealing with untrusted sources.

Python Pipes to Avoid Shells

You should take a look at the shell injection document before this one.

A lot of the time, our codebase uses `shell=True` because it’s convenient. The shell provides the ability to pipe things around without buffering them in memory and allows a malicious user to chain additional commands after a legitimate command is run.

Incorrect

Here is a simple function that uses `curl` to grab a page from a website and pipe it directly to the `wordcount` program to tell us how many lines there are in the HTML source code.

COPY

```
import subprocess

def count_lines(website):
    return subprocess.check_output('curl %s | wc -l' % website,
shell=True)

# >>> count_lines('www.google.com')
# '7\n'
```

(That output is correct, by the way - the Google HTML source does have 7 lines.)

The function is insecure because it uses `shell=True`, which allows shell injection. A user who instructs your code to fetch the website `; rm -rf /` can do terrible things to what used to be your machine.

If we convert the function to use `shell=False`, it doesn’t work.

COPY

```
def count_lines(website):
    args = ['curl', website, '|', 'wc', '-l']
    return subprocess.check_output(args, shell=False)

# >>> count_lines('www.google.com')
# curl: (6) Could not resolve host: |
# curl: (6) Could not resolve host: wc
# Traceback (most recent call last):
#   File "<stdin>", line 3, in <module>
#   File "/usr/lib/python2.7/subprocess.py", line 573, in check_output
#     raise CalledProcessError(retcode, cmd, output=output)
# subprocess.CalledProcessError: Command
# '['curl', 'www.google.com', '|', 'wc', '-l']' returned non-zero exit
status 6
```

The pipe doesn't mean anything special when `shell=False`, so `curl` tries to download the website called `|`. This does not fix the issue; it causes it to be more broken than before.

If we can't rely on pipes with `shell=False`, how should we do this?

Correct

```
def count_lines(website):
    args = ['curl', website]
    args2 = ['wc', '-l']
    process_curl = subprocess.Popen(args, stdout=subprocess.PIPE,
shell=False)
    process_wc = subprocess.Popen(args2, stdin=process_curl.stdout,
stdout=subprocess.PIPE, shell=False)

    # Allow process_curl to receive a SIGPIPE if process_wc exits.
    process_curl.stdout.close()
    return process_wc.communicate()[0]

# >>> count_lines('www.google.com')
# '7\n'
```

Rather than calling a single shell process that runs each of our programs, we run them separately and connect `stdout` from `curl` to `stdin` for `wc`. We specify `stdout=subprocess.PIPE`, which tells subprocess to send that output to the respective file handler.

Treat pipes like file descriptors (you can actually use FDs if you want); they may block on reading and writing if nothing is connected to the other end. That's why we use `communicate()`, which reads until EOF on the output and then waits for the process to terminate. You should generally avoid reading and writing to pipes directly unless you really know what you're doing - it's easy to work yourself into a situation that can deadlock.

Note that `communicate()` buffers the result in memory - if that's not what you want, use a file descriptor for `stdout` to pipe that output into a file.

Unvalidated URL redirect

It is common for web forms to redirect to a different page upon successful submission of the form data. This is often done using a `next` or `return` parameter in the HTTP request. Any HTTP parameter can be controlled by the user and could be abused by attackers to redirect a user to a malicious site.

This is commonly used in phishing attacks. For example, an attacker could redirect a user from a legitimate login form to a fake, attacker-controlled login form. If the page looks enough like the target site and tricks the user into believing they mistyped their password, the attacker can convince the user to re-enter their credentials and send them to the attacker.

Here is an example of a malicious redirect URL:

COPY

```
https://good.com/login.php?next=http://bad.com/phonylogin.php
```

To counter this type of attack, all URLs must be validated before being used to redirect the user. This should ensure the redirect will take the user to a page within your site.

Incorrect

This example just processes the `next` argument with no validation:

```
import os
from flask import Flask, redirect, request

app = Flask(__name__)

@app.route('/')
def example_redirect():
    return redirect(request.args.get('next'))
```

Correct

The following is an example using the Flask web framework. It checks that the URL the user is being redirected to originates from the same host as the host serving the content.

```
from flask import request, redirect
from urllib.parse import urlparse, urljoin

def is_safe_redirect_url(target):
    host_url = urlparse(request.host_url)
    redirect_url = urlparse(urljoin(request.host_url, target))
    return redirect_url.scheme in ('http', 'https') and
host_url.netloc == redirect_url.netloc

def get_safe_redirect():
    url = request.args.get('next')
    if url and is_safe_redirect_url(url):
        return url
    url = request.referrer
    if url and is_safe_redirect_url(url):
        return url

    return '/'

@app.route('/')
def example_redirect():
    return redirect(get_safe_redirect())
```

The `is_safe_redirect_url` function checks that the scheme of the redirect URL is either `http` or `https` and that the netloc (network location part) of the redirect URL matches the host URL. This ensures that the user is only redirected to URLs within the same site.

The Django framework contains a `django.utils.http.is_safe_url` function that can be used to validate redirects without implementing a custom version. This built-in function provides a secure and efficient way to handle URL redirects, ensuring that users are only redirected to safe and trusted URLs within the same domain.

Validate Certificates on HTTPS Connections to Avoid Man-in-the-Middle Attacks

When developing a module that makes secure HTTPS connections, it is crucial to use a library that verifies certificates. Many such libraries provide an option to ignore certificate verification failures, but these options should be exposed to the deployer to choose their level of risk.

Although this guideline specifically mentions HTTPS, verifying the identity of the hosts you are connecting to applies to most protocols (SSH, LDAPS, etc.).

Incorrect

COPY

```
import requests
requests.get('https://www.openstack.org/', verify=False)
```

The example above uses `verify=False` to bypass the check of the certificate received against those in the CA trust store.

It is important to note that modules such as `httplib` within the Python standard library did not verify certificate chains until it was fixed in the 2.7.9 release. For more specifics about the modules affected, refer to CVE-2014-9365.

Correct

COPY

```
import requests
requests.get('https://www.openstack.org/',
verify='/path/to/ca_cert.pem')
```

The example above uses the CA certificate file to verify that the certificate received is from a trusted authority.

Create, Use, and Remove Temporary Files Securely

Creating temporary files safely is more complicated than it seems due to potential security vulnerabilities. It's crucial to use the correct library functions to avoid issues like TOCTOU (Time of Check to Time of Use) attacks.

Python Temporary File Handling

Use	Avoid
`tempfile.TemporaryFile`	`tempfile.mktemp`

Use	Avoid
`tempfile.NamedTemporaryFile`	`open`
`tempfile.SpooledTemporaryFile`	`tempfile.mkstemp`
`tempfile.mkdtemp`	

`tempfile.TemporaryFile` should be used whenever possible as it hides the file and cleans up the file automatically.

Incorrect

Creating temporary files with predictable paths leaves them open to TOCTOU attacks:

COPY

```
import os
import tempfile

# This will most certainly put you at risk
tmp = os.path.join(tempfile.gettempdir(), "filename")
if not os.path.exists(tmp):
    with open(tmp, "w") as file:
        file.write("defaults")
```

Using `tempfile.mktemp` is also insecure:

COPY

```
import tempfile

open(tempfile.mktemp(), "w")
```

Correct

Use the secure methods provided by the `tempfile` module:

```
import tempfile

# Use the TemporaryFile context manager for easy clean-up
with tempfile.TemporaryFile() as tmp:
    tmp.write(b'stuff')

# Clean up a NamedTemporaryFile on your own
tmp = tempfile.NamedTemporaryFile(delete=True)
try:
    tmp.write(b'stuff')
finally:
    tmp.close()

# Handle opening the file yourself
fd, path = tempfile.mkstemp()
try:
    with os.fdopen(fd, 'w') as tmp:
        tmp.write('stuff')
finally:
    os.remove(path)
```

Restrict Path Access to Prevent Path Traversal

A path traversal attack occurs when an attacker supplies input used to construct a file path, allowing access to unintended files. Mitigating these attacks involves restricting file system access and using a restricted file permission profile.

Incorrect

An example of a path traversal vulnerability in a web application:

```python
import os
from flask import Flask, request, send_file

app = Flask(__name__)

@app.route('/')
def cat_picture():
    image_name = request.args.get('image_name')
    if not image_name:
        return 404
    return send_file(os.path.join(os.getcwd(), image_name))

if __name__ == '__main__':
    app.run(debug=True)
```

An attacker could exploit this by making a request like:

```
curl http://example.com/?image_name=../../../../../../../../etc/passwd
```

Correct

Use the following method to restrict access to files within a specific directory:

```python
import os

def is_safe_path(basedir, path, follow_symlinks=True):
    if follow_symlinks:
        matchpath = os.path.realpath(path)
    else:
        matchpath = os.path.abspath(path)
    return basedir == os.path.commonpath([basedir, matchpath])
```

```
def main(args):
    for arg in args:
        if is_safe_path(os.getcwd(), arg):
            print(f"safe: {arg}")
        else:
            print(f"unsafe: {arg}")

if __name__ == "__main__":
    import sys
    main(sys.argv[1:])
```

Another approach is to use an indirect mapping between a unique identifier and a file path:

COPY

```
localfiles = {
    "01": "/var/www/img/001.png",
    "02": "/var/www/img/002.png",
    "03": "/var/www/img/003.png",
}

def get_file(file_id):
    return open(localfiles[file_id])
```

Use Subprocess Securely

Running commands via subprocess is common but can be insecure if not handled correctly.

Incorrect

```
def ping(myserver):
    return subprocess.check_output(f'ping -c 1 {myserver}',
shell=True)

# >>> ping('8.8.8.8')
# '64 bytes from 8.8.8.8: icmp_seq=1 ttl=58 time=5.82 ms\n'
```

This code is vulnerable to shell injection:

COPY

```
ping('8.8.8.8; rm -rf /')
```

Correct

COPY

```
def ping(myserver):
    args = ['ping', '-c', '1', myserver]
    return subprocess.check_output(args, shell=False)
```

By passing a list of strings, the shell does not process user input as commands:

COPY

```
ping('8.8.8.8; rm -rf /')
# ping: unknown host 8.8.8.8; rm -rf /
```

Using subprocess correctly ensures user-provided input is interpreted safely.

Parameterize Database Queries

Often we write code that interacts with a database using parameters provided by the application's users. These parameters include credentials, resource identifiers, and other user-supplied data.

Care must be taken when dynamically creating database queries to prevent them from being subverted by user-supplied malicious input. This is generally referred to as SQL injection (SQLi). SQL injection works because the user input changes the logic of the SQL query, resulting in behavior that is not intended by the application developer.

The results of a successful SQL injection attack can include disclosure of sensitive information such as user passwords, modification or deletion of data, and gaining execution privileges, which would allow an attacker to run arbitrary commands on the database server.

SQL injection can typically be mitigated by using some combination of prepared statements, stored procedures, and escaping of user-supplied input. Most secure web applications will use all three, and we have described their use below.

SQLAlchemy

Incorrect

This example uses the built-in parameter substitution mechanism `%` to insert a value into the query string. This will perform an unsafe literal insertion and not provide any escaping.

COPY

```
import sqlalchemy

connection = engine.connect()
myvar = 'jsmith'  # our intended usage
myvar = 'jsmith or 1=1'  # this will return all users
myvar = 'jsmith; DROP TABLE users'  # this drops (removes) the users
table
query = "select username from users where username = %s" % myvar
result = connection.execute(query)
```

```
for row in result:
    print("username:", row['username'])
connection.close()
```

Correct

This example uses SQLAlchemy’s built-in parameter substitution mechanism to safely replace the `:name` variable with a provided value.

COPY

```
import sqlalchemy

connection = engine.connect()
myvar = 'jsmith'  # our intended usage
myvar = 'jsmith or 1=1'  # only matches this odd username
query = "select username from users where username = :name"
result = connection.execute(query, name=myvar)
for row in result:
    print("username:", row['username'])
connection.close()
```

MySQL

Incorrect

Without using any escaping mechanism, potentially unsafe queries can be created.

COPY

```
import MySQLdb

query = "select username from users where username = '%s'" % name
con = MySQLdb.connect('localhost', 'testuser', 'test623', 'testdb')

with con:
    cur = con.cursor()
    cur.execute(query)
```

Better

In this example, the query is created using Python's standard, unsafe `%` operator. MySQL's `escape_string` method is used to perform escaping on the user input string prior to inclusion in the string.

COPY

```
import MySQLdb

query = "select username from users where username = '%s'" %
MySQLdb.escape_string(name)
con = MySQLdb.connect('localhost', 'testuser', 'test623', 'testdb')

with con:
    cur = con.cursor()
    cur.execute(query)
```

Correct

The correct way to do this using a parameterized query might look like the following:

COPY

```
import MySQLdb

query = "select username from users where username = %s"
con = MySQLdb.connect('localhost', 'testuser', 'test623', 'testdb')

with con:
    cur = con.cursor()
    cur.execute(query, (username_value,))
```

This works because the logic of the query is compiled before the user input is considered.

PostgreSQL (Psycopg2)

Incorrect

This example uses Python's unsafe default parameter substitution mechanism to build a query string. This will not perform any escaping, unlike the correct example below. The string is processed and passed as a single parameter to `execute`.

COPY

```
import psycopg2

conn = psycopg2.connect("dbname=test user=postgres")
cur = conn.cursor()
cur.execute("select username from users where username = '%s'" % name)
```

Correct

This example uses Psycopg2's parameter substitution mechanism to build a query string. Despite the use of `%` to indicate the substitution token, it is not the same as Python's built-in string operator `%`. Note the value(s) are passed as parameters to `execute` separately.

COPY

```
import psycopg2

conn = psycopg2.connect("dbname=test user=postgres")
cur = conn.cursor()
cur.execute("select username from users where username = %s", (name,))
```

Protect Sensitive Data in Config Files from Disclosure

It is preferable to avoid storing sensitive information in configuration files, but there are occasions where this is unavoidable. For those situations, `oslo.config` provides a useful mechanism by which those sensitive pieces of information can be sanitized and protected.

In order to trigger this sanitization, a `secret=True` flag must be added to the `cfg.StrOpt()` function when registering the Oslo configuration.

An example of this practice is provided below.

Incorrect

In the example below, the password `secrets!` will be loaded through the `cfg.StrOpt()` function that could otherwise be logged and disclosed to anyone with access to the log file (legitimate or not).

COPY

```
cfg.StrOpt('password',
           help='Password of the host.')
```

Correct

A correct code example:

COPY

```
cfg.StrOpt('password',
           help='Password of the host.',
           secret=True)
```

Consequences

If sensitive information options are logged without being marked as secret, that sensitive information would be exposed whenever the logger debug flag is activated.

Example Log Entries

```
2015-02-18 20:46:48.928 25351 DEBUG nova.openstack.common.service
[-] Full set of CONF: _wait_for_exit_or_signal
/usr/lib/python2.7/dist-packages/nova/openstack/common/service.py:166
2015-02-18 20:46:48.937 25351 DEBUG nova.openstack.common.service [-]
Configuration options gathered from: log_opt_values
/usr/lib/python2.7/dist-packages/oslo/config/cfg.py:1982
2015-02-18 20:46:51.482 25351 DEBUG nova.openstack.common.service [-]
host.ip                 = 127.0.0.1 log_opt_values
/usr/lib/python2.7/dist-packages/oslo/config/cfg.py:2002
2015-02-18 20:46:51.491 25351 DEBUG nova.openstack.common.service [-]
host.port               = 443 log_opt_values /usr/lib/python2.7/dist-
packages/oslo/config/cfg.py:2002
2015-02-18 20:46:51.502 25351 DEBUG nova.openstack.common.service [-]
host.username           = root log_opt_values /usr/lib/python2.7/dist-
packages/oslo/config/cfg.py:2002
2015-02-18 20:46:51.486 25351 DEBUG nova.openstack.common.service [-]
host.password           = secrets! log_opt_values
/usr/lib/python2.7/dist-packages/oslo/config/cfg.py:2002
```

Use Secure Channels for Transmitting Data

Data in transit over networks should be protected wherever possible. Although some data may not appear to have strong confidentiality or integrity requirements, it is best practice to secure it.

When building any application that communicates over a network, we have to assume that we do not control the network that the data travels over. We should consider that the network may have hostile actors who will attempt to view or change the data that we are transmitting.

Clear Example

OpenStack API calls often contain credentials or tokens that are very sensitive. If they are sent in plaintext, they may be modified or stolen.

It is very important that API calls are protected from malicious third parties viewing them or tampering with their content - even for communications between services on an internal network.

Less Obvious Example

Consider a server process that reports the current number of stars in the sky and sends the data over the network to clients using a simple webpage. There is no strong confidentiality requirement for this; the data is not secret. However, integrity is important. An attacker on the network could alter the communications going from the server to clients and inject malicious traffic such as browser exploits into the HTTP stream, thus compromising vulnerable clients.

Incorrect

COPY

```
cfg.StrOpt('protocol',
           default='http',
           help='Default protocol to use when connecting to glance.')
```

Correct

COPY

```
cfg.StrOpt('protocol',
           default='https',
           help='Default protocol to use when connecting to glance.')
```

Escape User Input to Prevent XSS Attacks

These days, almost every service we create has some form of web interface, be it for administration, monitoring, or for the core functionality of the service. These interfaces are becoming ever more complex and dynamic, and increasingly interactive. There is a risk, however, when increasing the interactivity of these web services, that we inadvertently allow a user to supply data which can corrupt or disrupt the normal running of that service.

Cross-Site Scripting (XSS) is a class of vulnerability whereby an attacker is able to present active web content to a web service, which is subsequently echoed back to a user and executed by the browser. This content can be as seemingly benign as an embarrassing image or text, or as malign as browser-based exploits intended to steal and utilize the user's web session, or even compromise the user's web browser and take control of their client system.

There are three main classes of XSS issues: Persistent, Reflected, and DOM-Based. Persistent XSS issues are those where user input is stored by the server, either in a database or on the filesystem. This data is subsequently retrieved and rendered into the web page as if it were legitimate content. Reflected XSS issues are where user input is presented to the server and then immediately reflected back to the client. This data is never stored by the server. DOM-Based XSS issues occur entirely within the browser.

Incorrect

In the below example, user-supplied data from the query string is written directly into the HTML document and is presented to the user.

COPY

```
def index(request):
    name = request.GET.get('name')
    return f"<html><body>Hello {name}!</body></html>"
```

Correct

In the below example, the same user-supplied data is escaped before being written into the document, thus preventing the user from adding HTML elements to the document.

COPY

```
from html import escape

def index(request):
    name = request.GET.get('name')
    safe_name = escape(name)
    return f"<html><body>Hello {safe_name}!</body></html>"
```

In addition to the use of escaping, frameworks and templating engines can also be useful in preventing XSS attacks.

Resources

- https://docs.openstack.org/security-guide/compute/hardening-the-virtualization-layers.html

Devops DevSecOps openstack #OpenStack development infrastructure

Written by

RR **Reza Rashidi** Add your bio

Published on

DevSecOpsGuides Add blog description

MORE ARTICLES

RR **Reza Rashidi**

Attacking CI/CD

In CI/CD (Continuous Integration/Continuous Deployment) environments, several methods and attacks ca...

RR **Reza Rashidi**

Attacking Pipeline

DevOps pipelines, which integrate and automate the processes of software development and IT operatio...

RR **Reza Rashidi**

Attacking Policy

Open Policy Agent (OPA) is a versatile tool used to enforce policies and ensure compliance within a ...