Introduction to Secure Coding Principles

Revision Date: 2018-11-20

Secure Coding Principles: Get the exercises

Go to:

https://github.com/bryanstephenson/Secure-Coding-Principles

Download file “exercises.tgz” and then run “tar xzvf exercises.tgz”

Do not look at the files (yet).

Agenda

Evolution of the threat landscape

Foundation Concepts

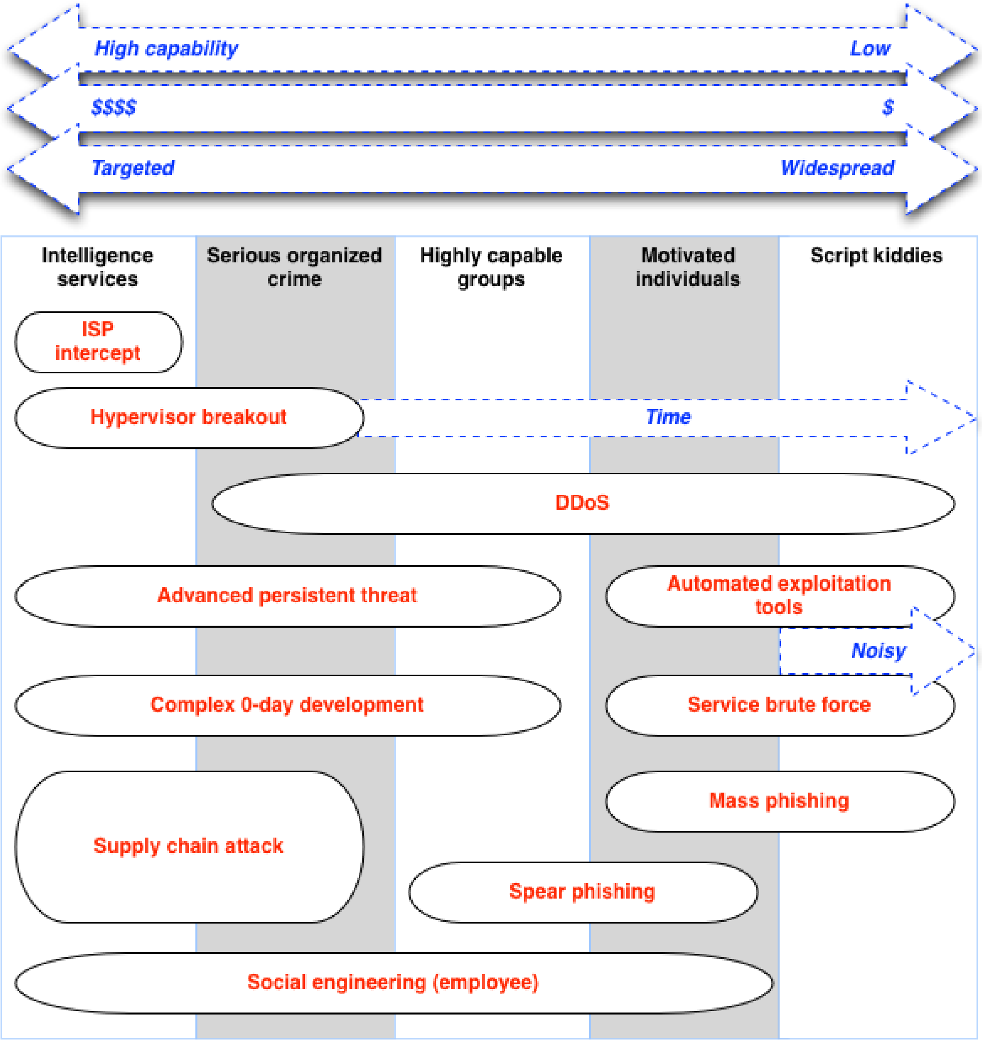
Attack and remediation exercises

Bandit primer

Additional Security Principles

Intro

Hackers in the 1990’s


High capability
Low
$$$$
$
Targeted
Widespread
Intelligence services
Serious organized crime
Highly capable groups
Motivated individuals
Script kiddies
ISP intercept
Hypervisor breakout
Time
DDoS
Advanced persistent threat
Automated exploitation tools
Noisy
Complex 0-day development
Service brute force
Mass phishing
Supply chain attack
Spear phishing
Social engineering (employee)

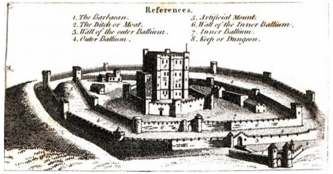
Foundation Concepts

Chained Attacks

Foundation Concepts

Chained Attacks

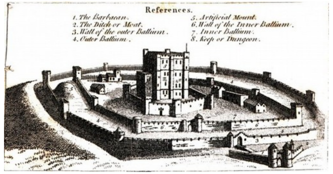
Defense in Depth



Foundation Concepts

Chained Attacks

Defense in Depth

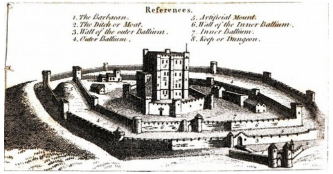


Least Privilege

Foundation Concepts

Chained Attacks

Defense in Depth



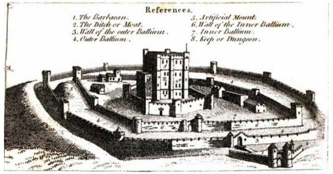
Least Privilege

Validate all User Input

Foundation Concepts

Chained Attacks

Defense in Depth



Least Privilege

Validate all User Input

Negative Testing

First exercise: SQL Injection

Example program in directory sql_injection/

Look at vulnerable_sql.py which is a very simple program which authenticates a user against a table of usernames and passwords in the database.

It uses the sqlite3 database built into Python.

It uses command line for input to make the training exercise easier. A real program would probably have a web user interface.

Main program

Asks the user for username and password then authenticates the user.

```
user = raw_input("Welcome to the system. To login please enter your user name: ")
password = raw_input("Please enter your password: ")

result = authenticate_user(user, password)

if result == True:
  print '\nSuccessful login for user {0}'.format(user)
else:
  print '\nYour user name and password were not found in the system. Please try
again.'
```

authenticate_user(user, password)

Asks the user for username and password then authenticates the user.

```
# Returns True if the provided user and password match an entry in the Users
# table of the database and False otherwise.

# This is the query which we will attack:

query = "SELECT username FROM Users WHERE username='{}' AND
password='{}';".format(user, password)

cursor.execute(query)
```

Perform SQL Injection attack

Spend a few minutes thinking like an attacker and try to attack the query. Put on your “Black Hat”.

Goal of the attack: login as a user without knowing or using the user’s password.

If you get stuck, look at hint files one at a time starting with hint_1, then hint_2, then hint_3, then hint_4. If you still have trouble with the attack, examine the “attack.answer” file which shows precisely how to attack this code and use it to attack the code so you see and understand how SQL injection happens.

Do not look at the “safe_sql.py” file (yet).

Example with Detailed Explanation

```
query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

cursor.execute(query)
```

|

Example

Attacker enters user: admin' or 1=1 --
password: x

```
query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

cursor.execute(query)
```

Example

Attacker enters user: admin' or 1=1 --

query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

admin'

cursor.execute(query)

Example

Attacker enters user: admin' or 1=1 --

```
query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

cursor.execute(query)
```

admin   specifies a user

'   terminates the string

Example

Attacker enters user: admin' or 1=1 --

```
query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

cursor.execute(query)
```

admin' specifies a user

"or 1=1"

Example

Attacker enters user: admin' or 1=1 --

query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

cursor.execute(query)

admin' specifies a user

"or 1=1" extends the logic of the where clause

Example

Attacker enters user: admin' or 1=1 --

```
query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

cursor.execute(query)
```

admin' specifies a user

"or 1=1" extends the logic of the where clause

"--"

Example

Attacker enters user: admin' or 1=1 --

```
query = "SELECT username FROM Users WHERE username='{}' AND
password='{}';".format(user, password)

cursor.execute(query)
```

admin' specifies a user

"or 1=1" extends the logic of the where clause

"--" comments out the password part of this query!

Example

Attacker enters user: admin’ or 1=1 --

```
query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

cursor.execute(query)
```

admin’ specifies a user

“or 1=1” extends the logic of the where clause

“--” comments out the password part of this query

SELECT username FROM Users where username=’admin’ or 1=1 -- AND password=’x’

Example

Attacker enters user: admin' or 1=1 --

```
query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

cursor.execute(query)
```

admin' specifies a user

"or 1=1" extends the logic of the where clause

"--" comments out the password part of this query

SELECT username FROM Users where username='admin' or 1=1 -- AND password='x'

"username = 'admin' or 1=1"

Example

Attacker enters user: admin' or 1=1 --

```
query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

cursor.execute(query)
```

admin' specifies a user

"or 1=1" extends the logic of the where clause

"--" comments out the password part of this query

SELECT username FROM Users where username='admin' or 1=1 -- AND password='x'

"username = 'admin' or 1=1" evaluates to TRUE

Example

Attacker enters user: admin' or 1=1 --

```
query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

cursor.execute(query)
```

admin' specifies a user

"or 1=1" extends the logic of the where clause

"--" comments out the password part of this query

SELECT username FROM Users where username='admin' or 1=1 -- AND password='x'

SELECT username FROM Users WHERE username='admin' or TRUE

Preventing SQL Injection

ALWAYS use parameterized queries

```
query = 'SELECT username FROM Users WHERE username = ? AND
  password = ?'

cursor.execute(query, [user, password])
```

Use SQLAlchemy or another ORM when possible

SDG: ***https://security.openstack.org/guidelines/dg_parameterize-database-queries.html***

Preventing SQL Injection

Exercise: Improve the program so it is not vulnerable to SQL injection. Put on your “White Hat”.

Copy the vulnerable_sql.py file and edit the copy.

Run the attack to verify the program is no longer vulnerable.

SDG: https://security.openstack.org/guidelines/dg_parameterize-database-queries.html

Command Injection Attack

Example program in directory command_injection/

Do not look at other files (yet).

Look at vulnerable_code.py which is a very simple program which provides a simple file sharing service. Users can list the directories and view the files they have stored in the service.

User “a” has password “a” to make logging in and trying attacks fast and easy.

Uses the sqlite3 database built into Python.

Uses command line for input to make the training easier. A real program would have a web user interface.

Command Injection Attack

Vulnerable code in function list_files():

```
try:
  subprocess.Popen(command_string, shell=True)
except:
  print 'I could not find that directory.'
```

Perform a command injection attack on this code. Goal of the attack: run commands of your choice on the system, such as “cat /etc/passwd”.

If you get stuck, look at hint files one at a time: first hint_1, then hint_2.

Do not look at the “command_injection.answer” file or the “safe_code.py” file (yet).

Running Linux commands

```
subprocess.Popen(‘ls -al /’)
subprocess.Popen([‘ls’, ‘-al’, ‘/’])
subprocess.Popen(‘ls -al /’, shell=True)
```

Running Linux commands

subprocess.Popen('ls -al /')   **X - doesn't work**

subprocess.Popen(['ls', '-al', '/'])   **✓ - good**

subprocess.Popen('ls -al /', shell=True)

X - dangerous!!

Why is this unsafe?

Linux shell metacharacters for example **> ‘ ; &&**

cause the Linux shell to perform *special actions*

Subprocess.Popen with shell=True enables the use of metacharacters

Avoiding command injection attacks

Use Linux commands parameterized with no shell:

subprocess.check_output([‘ls’, user_dir])

Now:

The only command that can execute is ‘ls’

A user with shell metacharacters in the name will be evaluated as a normal string, not special shell characters

- *some functions like os.system call commands directly on a shell - Avoid using them*
- *beware the wrapper functions!*

SDG: https://security.openstack.org/guidelines/dg_use-subprocess-securely.html

Preventing Command Injection

Exercise: Improve the program so it is not vulnerable to command injection. Two places use subprocess.Popen insecurely so they both need changed.

Run the attack to verify the program is no longer vulnerable.

SDG: https://security.openstack.org/guidelines/dg_use-subprocess-securely.html

Path Attack

Example program in directory path_attack/

Do not look at other files (yet).

vulnerable_code.py is a very simple program which which provides a simple file sharing service. Users can list the directories and view the files they have stored in the service.

User “a” has password “a” to make logging in and trying attacks fast and easy.

Uses the sqlite3 database built into Python.

Uses command line for input to make the training easier. A real program would have a web user interface.

Path Attack

Vulnerable code in function display_file():

```
command_option = 'user_file_storage/{0}/{1}'.format(user, filename)
try:
  file_text = subprocess.check_output(['cat', command_option])
  print file_text
except:
  print 'I could not find that file.'
```

Perform a path attack on this code. Goal of the attack: as user “a” try to view a file of user “alice”.

If you get stuck, look at hint files one at a time starting with hint_1, then hint_2.

Do not look at the “path.attack.answer” file or the “safe_code.py” file (yet).

Path Attack answers

Steal Alice's secret lasagna recipe by entering option 2 and filename:

../alice/recipes/lasagna

Or examine files on the system:

Enter the name of the file you want to see: ../../../../../../../../../../../etc/passwd

Fix the code to not be vulnerable to Path Attacks

Copy the vulnerable file to a new name and edit this new file.

Run a path attack on the fixed code to prove it is not vulnerable.

Fix the code to not be vulnerable to Path Attacks

```
# Block path attacks in the filename user input by removing "../" text strings

filename = requested_filename.replace('../', '')
```

Temp File Attacks

In Directory temp_file_attack

Look at the file vulnerable_code.py.

Do not look at other files (yet).

In Directory temp_file_attack

```
script = """
echo "Welcome to the system. Please set a password for future access: "
read password
echo $password > password.secret
"""

with open("/tmp/my_script.sh", "w") as f:
    f.write(script)

os.system("bash /tmp/my_script.sh")
```

Temp File Attack

Attack the vulnerable code by messing around on the filesystem (do not attack with user input as in earlier exercises).

Goal of the attack: prevent the program from operating properly.

Look at hint files if you need to.

Temp File Attack answer

Attack the vulnerable code by creating a file /tmp/my_script.sh and making it read only.

```
touch /tmp/my_script.sh
chmod 400 /tmp/my_script.sh
```

This causes the program to be unable to open the file for writing.

A Real Example

```
def start_oozie_process(pctx, instance):
    with instance.remote() as r:

        if c_helper.is_mysql_enabled(pctx, instance.node_group.cluster):
            _start_mysql(r)

            sql_script = files.get_file_text(
                      'plugins/vanilla/hadoop2/resources/create_oozie_db.sql')

            r.write_file_to('/tmp/create_oozie_db.sql', sql_script)

            _oozie_create_db(r)

def _oozie_create_db(remote):
    LOG.debug("Creating Oozie DB Schema...")
    remote.execute_command('mysql -u root < /tmp/create_oozie_db.sql')
```

Attacker controls commands being run

An attacker may control this file and trick the program into running commands of the attacker’s choice. The attacker creates the file with permissions which allow the program to open it for writing, then changes the contents of the file just before the program executes it.

Safe usage of temporary files

Use one of these functions:

- **tempfile.mkstemp or tempfile.mkdtemp** - these functions guarantee a file (mkstemp) or directory (mkdtemp) will be created (**no DoS possible**)
- mkstemp and mkdtemp also take care of file permissions: the resources they create are **only accessible by the calling user**
  - Note: the developer is responsible for removing the temp file or directory when finished.
- **tempfile.TemporaryFile** and **tempfile.NamedTemporaryFile** use mkstemp and delete the file when the program exists (not when file is closed).

SDG: https://security.openstack.org/guidelines/dg_using-temporary-files-securely.html
Python Docs: https://docs.python.org/3/library/tempfile.html

Secure Development Guidelines

Correct

The Python standard library provides a number of secure ways to create temporary files and directories. The following are examples of how you can use them.

Creating files:

```
import os
import tempfile

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

# Clean up a NamedTemporaryFile on your own
# delete=True means the file will be deleted on close
tmp = tempfile.NamedTemporaryFile(delete=True)
try:
    # do stuff with temp
    tmp.write('stuff')
finally:
    tmp.close()  # deletes the file

# Handle opening the file yourself. This makes clean-up
# more complex as you must watch out for exceptions
fd, path = tempfile.mkstemp()
try:
    with os.fdopen(fd, 'w') as tmp:
        # do stuff with temp file
        tmp.write('stuff')
finally:
    os.remove(path)
```

Exercise: Safe usage of temp files

There are many issues with this code.

For the first part of the exercise keep using the script /tmp/my_script.sh and protect the script file in /tmp.

If you get stuck, look at coding_hint_1 and coding_hint_2. The answer for this is in safe_code.py

Another Real Example – Making a temporary file

```
# Add a random integer in case there is corruption due to the script being
# run concurrently

temp_file = "/tmp/tmp_swift_data" + str(randint(0, 999)) + ".txt"

file_to_write = open(temp_file, "w")

file_to_write.write(json.dumps(node_data, indent=4,
                               separators=(',', ': ')))

file_to_write.close()
```

Issue 1: Application DoS

What happens if /tmp/tmp_swift_dataNNN already exists and can't be written? (eg. an attacker has already created the file and made it read only)

This creates a Denial of Service condition where the application either fails to run or doesn't run as expected.

Issue 2: Possible symlink attack

An attacker may use a technique called a Symlink Race* to trick the application into writing the data at an arbitrary location the application has access to!

* ***https://en.wikipedia.org/wiki/Symlink_race***

Issue 3: Possible data leakage

The /tmp directory is accessible to all users on a Linux system. Unless care is taken, the file will be created using the default umask, which will often allow any user to read the file (and possibly sensitive data).

Intro to Bandit

What is Bandit?

Fast static code scanner

Created by members of the OpenStack Security Project

Open source

Won’t find all bugs, but helps

Goal: to give developers a quick and easy way to scan their code for common security mistakes.

Bandit: easy to setup

1. (optional) virtualenv <virtualenv_name>
2. (optional) source <virtualenv_name>/bin/activate
3. pip install bandit
4. “bandit -r path_to_project -ll -ii”

Bandit Output

Metrics:

Total lines of code: 15
Total lines skipped (#nosec): 0

hardcoded_tmp_directory: Probable insecure usage of temp file/directory.

Severity: MEDIUM
Confidence: MEDIUM
File: examples/hardcoded-tmp.py

```
        f = open('/tmp/abc', 'w')
        f.write('def')
        f.close()
```

Plugin name and description - each plugin has documentation describing the test and why it's important

Bandit checks for usage of assert

- Be careful when using assert statements because running Python with optimizations enabled removes all assert statements.
- Use asserts to help ensure your code is stable during development, but don’t use them for code logic which is needed when the code is run in production.
- Expecting asserts to be used in production code causes things like https://github.com/IdentityPython/pysaml2/issues/451, a severe vulnerability which allows authentication bypass for any user.

Run Bandit on one or two vulnerable programs

- pip install bandit
- bandit <filename>.py

Known Unsafe Libraries/Functions

Use Safe Libraries

Unsafe libraries	Safe Alternatives
Pickle cPickle marshal	JSON
telnetlib	SSH, possibly with Pexpect : http://pexpect.readthedocs.io/en/latest/
xml, lxml	Defused XML : https://pypi.python.org/pypi/defusedxml
random (may be safe if not used for security)	os.urandom() or RAND_bytes function in OpenSSL library : https://www.openssl.org/

SDG: https://security.openstack.org/guidelines/dg_avoid-dangerous-input-parsing-libraries.html, https://security.openstack.org/guidelines/dg_strong-crypto.html

Pickle is unsafe

From https://blog.nelhage.com/2011/03/exploiting-pickle/

- Pickle can represent arbitrary objects, including subprocess.Popen.
- Pickle allows the objects to declare how they should be pickled by defining a __reduce__ method.
- This means an attacker can construct a pickle that will execute /bin/sh.
- Therefore never un-pickle data from a source that is not trusted.

From https://blog.nelhage.com/2011/03/exploiting-pickle/

Pickle Exploit

From https://blog.nelhage.com/2011/03/exploiting-pickle/

```
import cPickle
import subprocess
import base64

class Exploit(object):
  def __reduce__(self):
    fd = 20
    return (subprocess.Popen,
            (('/bin/sh',), # args
             0,            # bufsize
             None,         # executable
             fd, fd, fd    # std{in,out,err}
             ))

print base64.b64encode(cPickle.dumps(Exploit()))
```

Use Safe Functions

Unsafe Functions	**Safe Alternatives**
eval, exec	Find another way; perhaps use a parser or ast.literal_eval instead
yaml.load	yaml.safeload
hashlib.md5, hashlib.sha1	hashlib.sha384, hashlib.sha512
os.system	subprocess.Popen without “shell=True”
mark_safe (may be safe depending on usage)	Review the code carefully for XSS issues

If you aren’t sure about a library or function please read the docs or ask the security team

SDG: ***https://security.openstack.org/guidelines/dg_avoid-dangerous-input-parsing-libraries.html,***
https://security.openstack.org/guidelines/dg_strong-crypto.html

Sensitive Info in Logs

Why would you have sensitive information in logs?

Developers put them in to stabilize something and then forget to take them out

Might think “It’s OK to log this in DEBUG”

Belief that logs are secure - they may be, but you don’t know where they’re going to get distributed and who will be able to view them

Put sensitive information in __str__() or __repr__() which end up logged

Things not to log

Do not log (or output) anything an attacker would go after:

- Passwords, even failed passwords
- tokens, keys, or any other credential
- PII (Personally Identifiable Information): Identification numbers, birth dates, phone numbers, etc.

SDG: https://security.openstack.org/guidelines/dg_protect-sensitive-data-in-files.html

Be “Secure by Default” after installation

After an initial installation all default configuration settings should be set to the most secure option which does not cause issues.

Keep settings which were in place before an upgrade even if they are not the most secure settings.

Clearly document the risks of changing configuration settings so that administrators can make an informed decision.

When installing ensure the system starts with a secure root of trust like a password entered when running an installation script.

Spot the Bug

Example: Encrypted web application data

```
logger.debug("user: {}, password: {},
        source_ip: {},
        session_contents: {},
        session_duration: {}".format(
  user.id, user.password, user.ip, user.session.data, user.session.length ))
```

Spot the Bug - Explanation

user.id -- **ok** (usually); user ID is usually public by design. Consider logging a hash of the user ID.

user.password -- **bad**, we should never log passwords

user.ip -- **danger**, depending on jurisdiction (PII in Germany and UK)

user.session.data -- **bad**, unencrypted session data may be sensitive

user.session.length -- **ok**, not sensitive data

Bonus Exercise: Fix the temp file demo program

For the second part of the exercise lose the script entirely. Use file handling functions to create the password.secret file.

An answer to this is in part2.py.

A better answer is in part3.py

A note on using passwords

Use a service to do your authentication for you if at all possible.

If you must store a password from a user, use a crypto library that handles the details and stores it for you.

If you really need to store the password yourself without a library to store it for you then:

- Use strict permissions for the file (mode 600) and directory (mode 700) containing the file.
- Hash the password before storing it anywhere other than memory.
- Have more people review the code more thoroughly than typical code.

Additional Information

Security Resources (1/3)

More information about attacks

https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

http://cwe.mitre.org/top25/

Developer best practices

https://security.openstack.org/#secure-development-guidelines

https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide

https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard

For web developers

https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

Security Resources (2/3)

Cryptography

- https://www.owasp.org/index.php/Guide_to_Cryptography
- https://cryptography.io - Python crypto library

CWE (Common Weakness Enumeration)

- https://cwe.mitre.org/data/definitions/699.html – Development Concepts
- https://cwe.mitre.org/data/definitions/1008.html – Architectural Concepts

Security Resources (3/3)

- Secure Programming for Linux and Unix HOWTO: http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/
- Writing Secure Code: Practical Strategies and Proven Techniques for Building Secure Applications in a Networked World (Developer Best Practices) Second Edition, by Michael Howard and David LeBlanc
- Practical Cryptography, by Niels Ferguson and Bruce Schneier
- The Shellcoder's Handbook, Second Edition, by Chris Anley, John Heasman, Felix Lindner, and Gerardo Richarte
- The Web Application Hacker's Handbook : Discovering and Exploiting Security Flaws, by Dafydd Stuttard and Marcus Pinto
- 19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them, by Michael Howard, David LeBlanc, and John Viega
- Hacking, Second Edition, by Jon Erickson
- The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities (2 Volume set), 1st Edition, by Mark Dowd , John McDonald, and Justin Schuh.

Introduction to
Secure Coding Principles

Revision Date: 2018-11-20

You will get all the slides and code examples that we run here.

We could talk for days about secure coding techniques. This attempts to teach the most important things to get started.

More resources are available and at the end there are pointers to them.

Please fill out the survey and rate this workshop because this helps to improve the material. I've taught hundreds of developers over the years and the material has evolved a lot.

I don't want to assume too much knowledge so some of this may be well known to many of you. Please don't be insulted. Those of you with more experience in this area please share your expertise during discussions.

Everyone, please interrupt me at any time with questions.

Secure Coding Principles: Get the exercises

Go to:

https://github.com/bryanstephenson/Secure-Coding-Principles

Download file “exercises.tgz” and then run “tar xzvf exercises.tgz”

Do not look at the files (yet).

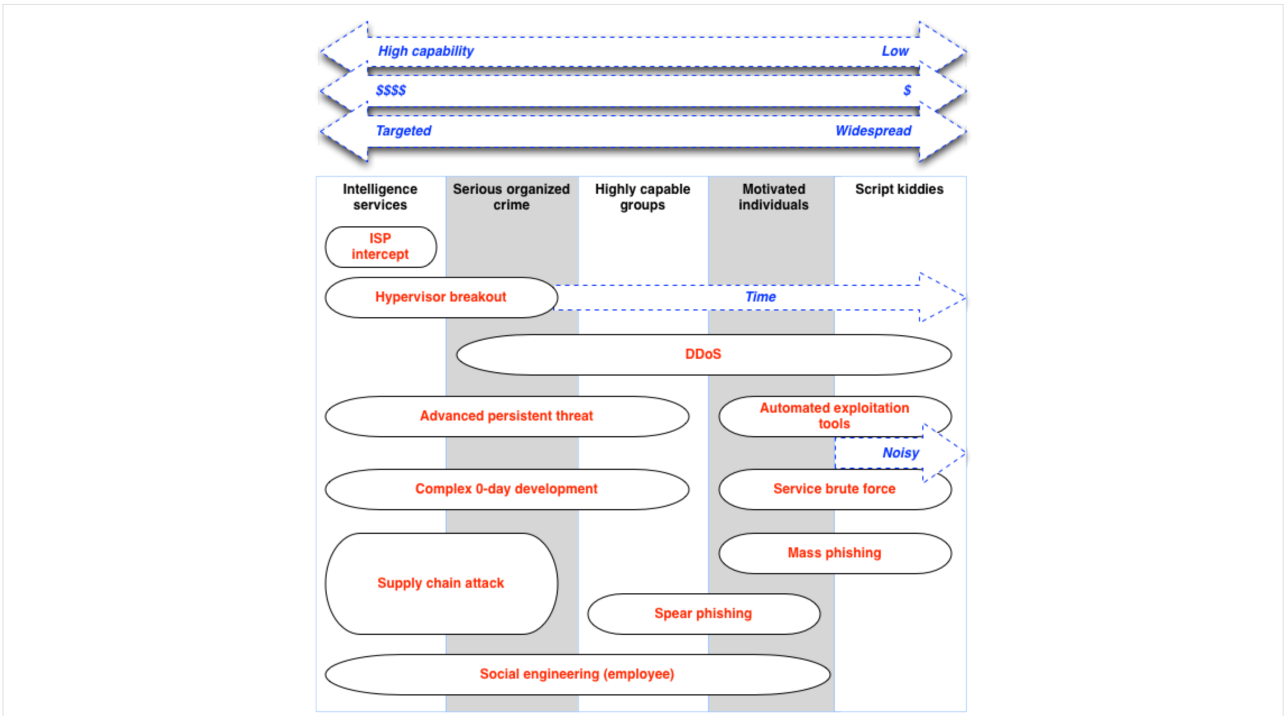
Agenda

- Evolution of the threat landscape
- Foundation Concepts
- Attack and remediation exercises
- Bandit primer
- Additional Security Principles

Intro

Hackers in the 1990’s

This is Angelina Jolie and her first husband in the movie Hackers from the 1990s.
Adversaries were often smart kids looking for notoriety.
But it’s not the 90’s anymore!



Now we and our users face formidable adversaries.
We need to make it difficult and expensive to hack our systems.
This class teaches many of the basic things we need to do, but there is much more we could talk about.

Foundation Concepts

Chained Attacks

Chained attacks: Phish a backup operator -> Backup server -> Database server -> Information they seek

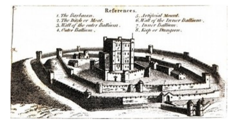
Defense in depth: Example - network separation, public facing stuff on a separate network from internal things

You can watch for these symbols on future slides to see what concepts are being demonstrated.

Foundation Concepts

Chained Attacks

Defense in Depth

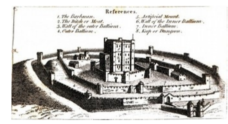


Chained attacks: Phish a backup operator -> Backup server -> Database server -> Information they seek
Defense in depth: Example - network separation, public facing stuff on a separate network from internal things
You can watch for these symbols on future slides to see what concepts are being demonstrated.

Foundation Concepts

Chained Attacks

Defense in Depth



Least Privilege

Chained attacks: Phish a backup operator -> Backup server -> Database server -> Information they seek

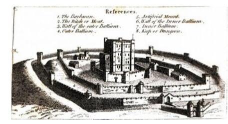
Defense in depth: Example - network separation, public facing stuff on a separate network from internal things

You can watch for these symbols on future slides to see what concepts are being demonstrated.

Foundation Concepts

Chained Attacks

Defense in Depth



Least Privilege

Validate all User Input

Chained attacks: Phish a backup operator -> Backup server -> Database server -> Information they seek

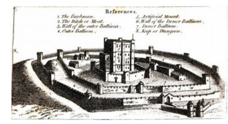
Defense in depth: Example - network separation, public facing stuff on a separate network from internal things

You can watch for these symbols on future slides to see what concepts are being demonstrated.

Foundation Concepts

Chained Attacks

Defense in Depth



Least Privilege

Validate all User Input

Negative Testing

Most programs undergo positive testing to verify the the intended functionality works.

Not all program undergo negative testing to verify that they do not enable unintended and undesirable functionality.

In general we should focus more on negative testing than we do today.

First exercise: SQL Injection

Example program in directory sql_injection/

Look at vulnerable_sql.py which is a very simple program which authenticates a user against a table of usernames and passwords in the database.

It uses the sqlite3 database built into Python.

It uses command line for input to make the training exercise easier. A real program would probably have a web user interface.

Main program

Asks the user for username and password then authenticates the user.

```
user = raw_input("Welcome to the system. To login please enter your user name: ")
password = raw_input("Please enter your password: ")

result = authenticate_user(user, password)

if result == True:
  print '\nSuccessful login for user {0}'.format(user)
else:
  print '\nYour user name and password were not found in the system. Please try again.'
```

authenticate_user(user, password)

Asks the user for username and password then authenticates the user.

```
# Returns True if the provided user and password match an entry in the Users
# table of the database and False otherwise.

# This is the query which we will attack:

query = "SELECT username FROM Users WHERE username='{}' AND
password='{}';".format(user, password)

cursor.execute(query)
```

Perform SQL Injection attack

Spend a few minutes thinking like an attacker and try to attack the query. Put on your “Black Hat”.

Goal of the attack: login as a user without knowing or using the user’s password.

If you get stuck, look at hint files one at a time starting with hint_1, then hint_2, then hint_3, then hint_4. If you still have trouble with the attack, examine the “attack.answer” file which shows precisely how to attack this code and use it to attack the code so you see and understand how SQL injection happens.

Do not look at the “safe_sql.py” file (yet).

Example with Detailed Explanation

```
query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

cursor.execute(query)
```

|

Example

Attacker enters user: admin' or 1=1 --
password: x

```
query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

cursor.execute(query)
```

Example

Attacker enters user: admin' or 1=1 --

```
query = "SELECT username FROM Users WHERE username='{}' AND
password='{}';".format(user, password)
```

admin'

```
cursor.execute(query)
```

Example

Attacker enters user: admin' or 1=1 --

query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

admin   specifies a user
'   terminates the string

cursor.execute(query)

Example

Attacker enters user: ***admin' or 1=1 --***

```
query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

cursor.execute(query)
```

admin' specifies a user

"or 1=1"

Example

Attacker enters user: admin' or 1=1 --

```
query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

cursor.execute(query)
```

admin' specifies a user

"or 1=1" extends the logic of the where clause

Example

Attacker enters user: admin' or 1=1 --

```
query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

cursor.execute(query)
```

admin' specifies a user

"or 1=1" extends the logic of the where clause

"--"

Example

Attacker enters user: admin' or 1=1 --

```
query = "SELECT username FROM Users WHERE username='{}' AND
password='{}';".format(user, password)

cursor.execute(query)
```

admin' specifies a user

"or 1=1" extends the logic of the where clause

"--" comments out the password part of this query!

Example

Attacker enters user: admin' or 1=1 --

```
query = "SELECT username FROM Users WHERE username='{}' AND
password='{}';".format(user, password)

cursor.execute(query)
```

admin' specifies a user

"or 1=1" extends the logic of the where clause

"--" comments out the password part of this query

SELECT username FROM Users where username='admin' or 1=1 -- AND password='x'

So what will happen with this query?

Example

Attacker enters user: admin' or 1=1 --

```
query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

cursor.execute(query)
```

admin' specifies a user

"or 1=1" extends the logic of the where clause

"--" comments out the password part of this query

SELECT username FROM Users where username='admin' or 1=1 -- AND password='x'

"username = 'admin' or 1=1"

So what will happen with this query?

Example

Attacker enters user: admin' or 1=1 --

query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

cursor.execute(query)

admin' specifies a user

"or 1=1" extends the logic of the where clause

"--" comments out the password part of this query

SELECT username FROM Users where username='admin' or 1=1 -- AND password='x'

"username = 'admin' or 1=1" evaluates to TRUE

So what will happen with this query?

Example

Attacker enters user: admin' or 1=1 --

query = "SELECT username FROM Users WHERE username='{}' AND password='{}';".format(user, password)

cursor.execute(query)

admin' specifies a user

"or 1=1" extends the logic of the where clause

"--" comments out the password part of this query

SELECT username FROM Users where username='admin' or 1=1 -- AND password='x'

SELECT username FROM Users WHERE username='admin' or TRUE

So what will happen with this query?

Preventing SQL Injection

ALWAYS use parameterized queries

```
query = 'SELECT username FROM Users WHERE username = ? AND
   password = ?'

cursor.execute(query, [user, password])
```

Use SQLAlchemy or another ORM when possible

SDG: ***https://security.openstack.org/guidelines/dg_parameterize-database-queries.html***

Check the query result to ensure uniqueness. If you have two results querying the same username the program should probably fail gracefully.

It is highly recommended to use an Object Relational Mapping tool like SQLAlchemy which will ensure parameterized queries and always used.

Preventing SQL Injection

Exercise: Improve the program so it is not vulnerable to SQL injection. Put on your "White Hat".

Copy the vulnerable_sql.py file and edit the copy.

Run the attack to verify the program is no longer vulnerable.

SDG: ***https://security.openstack.org/guidelines/dg_parameterize-database-queries.html***

Command Injection Attack

Example program in directory command_injection/

Do not look at other files (yet).

Look at vulnerable_code.py which is a very simple program which provides a simple file sharing service. Users can list the directories and view the files they have stored in the service.

User “a” has password “a” to make logging in and trying attacks fast and easy.

Uses the sqlite3 database built into Python.

Uses command line for input to make the training easier. A real program would have a web user interface.

Command Injection Attack

Vulnerable code in function list_files():

```
try:
  subprocess.Popen(command_string, shell=True)
except:
  print 'I could not find that directory.'
```

Perform a command injection attack on this code. Goal of the attack: run commands of your choice on the system, such as “cat /etc/passwd”.

If you get stuck, look at hint files one at a time: first hint_1, then hint_2.

Do not look at the “command_injection.answer” file or the “safe_code.py” file (yet).

Running Linux commands

```
subprocess.Popen(‘ls -al /’)
subprocess.Popen([‘ls’, ‘-al’, ‘/’])
subprocess.Popen(‘ls -al /’, shell=True)
```

Rank these best to worst

Running Linux commands

```
subprocess.Popen('ls -al /')
```
X - doesn't work

```
subprocess.Popen(['ls', '-al', '/'])
```
✓ - good

```
subprocess.Popen('ls -al /', shell=True)
```
X - dangerous!!

Top one won't execute and the third one is really unsafe.

Why is this unsafe?

Linux shell metacharacters for example **> ‘ ; &&**

cause the Linux shell to perform *special actions*

Subprocess.Popen with shell=True enables the use of metacharacters

Avoiding command injection attacks

Use Linux commands parameterized with no shell:

subprocess.check_output([‘ls’, user_dir])

Now:

The only command that can execute is ‘ls’

A user with shell metacharacters in the name will be evaluated as a normal string, not special shell characters

- *some functions like os.system call commands directly on a shell - Avoid using them*
- *beware the wrapper functions!*

SDG: [***https://security.openstack.org/guidelines/dg_use-subprocess-securely.html***](https://security.openstack.org/guidelines/dg_use-subprocess-securely.html)

Keep the user input out of the command part of the arguments.

Preventing Command Injection

Exercise: Improve the program so it is not vulnerable to command injection. Two places use subprocess.Popen insecurely so they both need changed.

Run the attack to verify the program is no longer vulnerable.

SDG: [***https://security.openstack.org/guidelines/dg_use-subprocess-securely.html***](https://security.openstack.org/guidelines/dg_use-subprocess-securely.html)

Path Attack

Example program in directory path_attack/

Do not look at other files (yet).

vulnerable_code.py is a very simple program which which provides a simple file sharing service. Users can list the directories and view the files they have stored in the service.

User “a” has password “a” to make logging in and trying attacks fast and easy.

Uses the sqlite3 database built into Python.

Uses command line for input to make the training easier. A real program would have a web user interface.

Path Attack

Vulnerable code in function display_file():

```
command_option = 'user_file_storage/{0}/{1}'.format(user, filename)
try:
  file_text = subprocess.check_output(['cat', command_option])
  print file_text
except:
  print 'I could not find that file.'
```

Perform a path attack on this code. Goal of the attack: as user “a” try to view a file of user “alice”.

If you get stuck, look at hint files one at a time starting with hint_1, then hint_2.
Do not look at the “path.attack.answer” file or the “safe_code.py” file (yet).

Path Attack answers

Steal Alice’s secret lasagna recipe by entering option 2 and filename:

../alice/recipes/lasagna

Or examine files on the system:

Enter the name of the file you want to see: ../../../../../../../../../../../../etc/passwd

Fix the code to not be vulnerable to Path Attacks

Copy the vulnerable file to a new name and edit this new file.

Run a path attack on the fixed code to prove it is not vulnerable.

Fix the code to not be vulnerable to Path Attacks

```
# Block path attacks in the filename user input by removing "../" text strings

filename = requested_filename.replace('../', '')
```

Temp File Attacks

In Directory temp_file_attack

Look at the file vulnerable_code.py.

Do not look at other files (yet).

In Directory temp_file_attack

```
script = """
echo "Welcome to the system. Please set a password for future access: "
read password
echo $password > password.secret
"""

with open("/tmp/my_script.sh", "w") as f:
    f.write(script)

os.system("bash /tmp/my_script.sh")
```

Temp File Attack

Attack the vulnerable code by messing around on the filesystem (do not attack with user input as in earlier exercises).

Goal of the attack: prevent the program from operating properly.

Look at hint files if you need to.

Temp File Attack answer

Attack the vulnerable code by creating a file /tmp/my_script.sh and making it read only.

```
touch /tmp/my_script.sh
chmod 400 /tmp/my_script.sh
```

This causes the program to be unable to open the file for writing.

A Real Example

```
def start_oozie_process(pctx, instance):
    with instance.remote() as r:

        if c_helper.is_mysql_enabled(pctx, instance.node_group.cluster):
            _start_mysql(r)

            sql_script = files.get_file_text(
                    'plugins/vanilla/hadoop2/resources/create_oozie_db.sql')

            r.write_file_to('/tmp/create_oozie_db.sql', sql_script)

            _oozie_create_db(r)

def _oozie_create_db(remote):
    LOG.debug("Creating Oozie DB Schema...")
    remote.execute_command('mysql -u root < /tmp/create_oozie_db.sql')
```

If you discover something like this what do you do?

Talk about “Responsible Disclosure”

Attacker controls commands being run

An attacker may control this file and trick the program into running commands of the attacker’s choice. The attacker creates the file with permissions which allow the program to open it for writing, then changes the contents of the file just before the program executes it.

Safe usage of temporary files

Use one of these functions:

- **tempfile.mkstemp or tempfile.mkdtemp** - these functions guarantee a file (mkstemp) or directory (mkdtemp) will be created (**no DoS possible**)
- mkstemp and mkdtemp also take care of file permissions: the resources they create are **only accessible by the calling user**
  - Note: the developer is responsible for removing the temp file or directory when finished.
- **tempfile.TemporaryFile** and **tempfile.NamedTemporaryFile** use mkstemp and delete the file when the program exists (not when file is closed).

SDG: ***https://security.openstack.org/guidelines/dg_using-temporary-files-securely.html***
Python Docs: ***https://docs.python.org/3/library/tempfile.html***

The OpenStack Secuce Development Guidelines or SDG provide many great examples of how to do things securely. The python documentation is also helpful. Example on the next slide.

Secure Development Guidelines

Correct

The Python standard library provides a number of secure ways to create temporary files and directories. The following are examples of how you can use them.

Creating files:

```
import os
import tempfile

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

# Clean up a NamedTemporaryFile on your own
# delete=True means the file will be deleted on close
tmp = tempfile.NamedTemporaryFile(delete=True)
try:
    # do stuff with temp
    tmp.write('stuff')
finally:
    tmp.close()  # deletes the file

# Handle opening the file yourself. This makes clean-up
# more complex as you must watch out for exceptions
fd, path = tempfile.mkstemp()
try:
    with os.fdopen(fd, 'w') as tmp:
        # do stuff with temp file
        tmp.write('stuff')
finally:
    os.remove(path)
```

Example of OpenStack SDG information.

Exercise: Safe usage of temp files

There are many issues with this code.

For the first part of the exercise keep using the script /tmp/my_script.sh and protect the script file in /tmp.

If you get stuck, look at coding_hint_1 and coding_hint_2. The answer for this is in safe_code.py

Another Real Example – Making a temporary file

```
# Add a random integer in case there is corruption due to the script being
# run concurrently

temp_file = "/tmp/tmp_swift_data" + str(randint(0, 999)) + ".txt"

file_to_write = open(temp_file, "w")

file_to_write.write(json.dumps(node_data, indent=4,
                        separators=(',', ': ')))

file_to_write.close()
```

Issues?

We have reduced the chance of a file name collision but not eliminated it.

Insecure usage of the /tmp/ directory.

Issue 1: Application DoS

What happens if /tmp/tmp_swift_dataNNN already exists and can't be written? (eg. an attacker has already created the file and made it read only)

This creates a Denial of Service condition where the application either fails to run or doesn't run as expected.

Issue 2: Possible symlink attack

An attacker may use a technique called a Symlink Race* to trick the application into writing the data at an arbitrary location the application has access to!

* *https://en.wikipedia.org/wiki/Symlink_race*

The attacker may notice the pattern of files being created in /tmp/ and create all 1000 file names and make them all read only, preventing the application from creating any temporary files.

Issue 3: Possible data leakage

The /tmp directory is accessible to all users on a Linux system. Unless care is taken, the file will be created using the default umask, which will often allow any user to read the file (and possibly sensitive data).

Intro to Bandit

What is Bandit?

Fast static code scanner

Created by members of the OpenStack Security Project

Open source

Won’t find all bugs, but helps

Goal: to give developers a quick and easy way to scan their code for common security mistakes.

Scans keystone code in about 20 seconds

Bandit: easy to setup

1. (optional) virtualenv <virtualenv_name>
2. (optional) source <virtualenv_name>/bin/activate
3. pip install bandit
4. “bandit -r path_to_project -ll -ii”

We will use Bandit in a few slides.

Bandit Output

Metrics:

Total lines of code: 15
Total lines skipped (#nosec): 0

hardcoded_tmp_directory: Probable insecure usage of temp file/directory.

Severity: MEDIUM
Confidence: MEDIUM
File: examples/hardcoded-tmp.py

```
f = open('/tmp/abc', 'w')
f.write('def')
f.close()
```

Plugin name and description - each plugin has documentation describing the test and why it's important

Bandit checks for usage of assert

- Be careful when using assert statements because running Python with optimizations enabled removes all assert statements.
- Use asserts to help ensure your code is stable during development, but don't use them for code logic which is needed when the code is run in production.
- Expecting asserts to be used in production code causes things like https://github.com/IdentityPython/pysaml2/issues/451, a severe vulnerability which allows authentication bypass for any user.

Run Bandit on one or two vulnerable programs

- pip install bandit
- bandit <filename>.py

Known Unsafe Libraries/Functions

Use Safe Libraries

Unsafe libraries	Safe Alternatives
Pickle cPickle marshal	JSON
telnetlib	SSH, possibly with Pexpect : http://pexpect.readthedocs.io/en/latest/
xml, lxml	Defused XML : https://pypi.python.org/pypi/defusedxml
random (may be safe if not used for security)	os.urandom() or RAND_bytes function in OpenSSL library : https://www.openssl.org/

SDG: ***https://security.openstack.org/guidelines/dg_avoid-dangerous-input-parsing-libraries.html, https://security.openstack.org/guidelines/dg_strong-crypto.html***

xml.etree is vulnerable to a logic bomb.

Telnet sends all traffic unencrypted, including passwords.

Don’t use marshal or pickle to load data; use JSON instead.

http://nadiana.com/python-pickle-insecure

Pickle is unsafe

From https://blog.nelhage.com/2011/03/exploiting-pickle/

- Pickle can represent arbitrary objects, including subprocess.Popen.
- Pickle allows the objects to declare how they should be pickled by defining a __reduce__ method.
- This means an attacker can construct a pickle that will execute /bin/sh.
- Therefore never un-pickle data from a source that is not trusted.

From https://blog.nelhage.com/2011/03/exploiting-pickle/

From https://blog.nelhage.com/2011/03/exploiting-pickle/

Pickle Exploit

From https://blog.nelhage.com/2011/03/exploiting-pickle/

```
import cPickle
import subprocess
import base64

class Exploit(object):
  def __reduce__(self):
    fd = 20
    return (subprocess.Popen,
            (('/bin/sh',), # args
             0,            # bufsize
             None,         # executable
             fd, fd, fd    # std{in,out,err}
             ))

print base64.b64encode(cPickle.dumps(Exploit()))
```

From https://blog.nelhage.com/2011/03/exploiting-pickle/

The attacker sends this many times, and eventually gets FD 20.

Then they send junk until the server errors out and closes the connection.

Now they have a remote shell and can execute arbitrary commands.

Use Safe Functions

Unsafe Functions	Safe Alternatives
eval, exec	Find another way; perhaps use a parser or ast.literal_eval instead
yaml.load	yaml.safeload
hashlib.md5, hashlib.sha1	hashlib.sha384, hashlib.sha512
os.system	subprocess.Popen without “shell=True”
mark_safe (may be safe depending on usage)	Review the code carefully for XSS issues

If you aren’t sure about a library or function please read the docs or ask the security team

SDG: https://security.openstack.org/guidelines/dg_avoid-dangerous-input-parsing-libraries.html, https://security.openstack.org/guidelines/dg_strong-crypto.html

Eval is very powerful. It is nearly impossible to make malicious input safe to run. There are too many things that would need to be checked.

Sometimes operator can be used instead.

Sensitive Info in Logs

Why would you have sensitive information in logs?

Developers put them in to stabilize something and then forget to take them out

Might think “It’s OK to log this in DEBUG”

Belief that logs are secure - they may be, but you don’t know where they’re going to get distributed and who will be able to view them

Put sensitive information in __str__() or __repr__() which end up logged

Things not to log

Do not log (or output) anything an attacker would go after:

- Passwords, even failed passwords
- tokens, keys, or any other credential
- PII (Personally Identifiable Information): Identification numbers, birth dates, phone numbers, etc.

SDG: https://security.openstack.org/guidelines/dg_protect-sensitive-data-in-files.html

Be “Secure by Default” after installation

After an initial installation all default configuration settings should be set to the most secure option which does not cause issues.

Keep settings which were in place before an upgrade even if they are not the most secure settings.

Clearly document the risks of changing configuration settings so that administrators can make an informed decision.

When installing ensure the system starts with a secure root of trust like a password entered when running an installation script.

We may not always want to select the most secure configuration setting for the defaults. Why not?

The most secure setting might cause severe functional issues like filling up a file system, or very poor performance.

We need to make intelligent trade-offs here as with any other engineering problem, and clearly document for users the implications of changing the configuration settings so that they can make informed decisions which are best for their particular environment. That particular environment can vary immensely even within the same organization The settings for a test and development cloud may be very different from the settings for a production cloud running sensitive business-critical applications.

Spot the Bug

Example: Encrypted web application data

```
logger.debug("user: {}, password: {},
        source_ip: {},
        session_contents: {},
        session_duration: {}".format(
  user.id, user.password, user.ip, user.session.data, user.session.length ))
```

Explanations on next slide

Spot the Bug - Explanation

user.id -- **ok** (usually); user ID is usually public by design. Consider logging a hash of the user ID.

user.password -- **bad**, we should never log passwords

user.ip -- **danger**, depending on jurisdiction (PII in Germany and UK)

user.session.data -- **bad**, unencrypted session data may be sensitive

user.session.length -- **ok**, not sensitive data

Bonus Exercise: Fix the temp file demo program

For the second part of the exercise lose the script entirely. Use file handling functions to create the password.secret file.

An answer to this is in part2.py.

A better answer is in part3.py

A note on using passwords

Use a service to do your authentication for you if at all possible.

If you must store a password from a user, use a crypto library that handles the details and stores it for you.

If you really need to store the password yourself without a library to store it for you then:

- Use strict permissions for the file (mode 600) and directory (mode 700) containing the file.
- Hash the password before storing it anywhere other than memory.
- Have more people review the code more thoroughly than typical code.

Additional Information

Security Resources (1/3)

More information about attacks

- https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
- http://cwe.mitre.org/top25/

Developer best practices

- https://security.openstack.org/#secure-development-guidelines
- https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide
- https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard

For web developers

- https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet
- https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

Security Resources (2/3)

Cryptography

- https://www.owasp.org/index.php/Guide_to_Cryptography
- https://cryptography.io - Python crypto library

CWE (Common Weakness Enumeration)

- https://cwe.mitre.org/data/definitions/699.html – Development Concepts
- https://cwe.mitre.org/data/definitions/1008.html – Architectural Concepts

Security Resources (3/3)

- Secure Programming for Linux and Unix HOWTO: http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/
- Writing Secure Code: Practical Strategies and Proven Techniques for Building Secure Applications in a Networked World (Developer Best Practices) Second Edition, by Michael Howard and David LeBlanc
- Practical Cryptography, by Niels Ferguson and Bruce Schneier
- The Shellcoder's Handbook, Second Edition, by Chris Anley, John Heasman, Felix Lindner, and Gerardo Richarte
- The Web Application Hacker's Handbook : Discovering and Exploiting Security Flaws, by Dafydd Stuttard and Marcus Pinto
- 19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them, by Michael Howard, David LeBlanc, and John Viega
- Hacking, Second Edition, by Jon Erickson
- The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities (2 Volume set), 1st Edition, by Mark Dowd , John McDonald, and Justin Schuh.