Week 3
Serialization Vulnerabilities

Intro to Offensive Security

Serialization

- Many programming languages offer a way to *serialize* objects in memory
- Serialization gathers up the data from the objects, converts them to a string of bytes, and writes to disk
- Later, the data can be *deserialized* and the original objects can be recreated

Example: Python Pickling

- In Python, serialization is called “pickling”
- import pickle
  foo = pickle.dumps([1,2,3])
  foo is now: "(lp0\nI1\naI2\naI3\na."
- Later, you can unpickle:
  pickle.loads(foo)
  Output: [1, 2, 3]

Serialization in Web Apps

- Recall that HTTP is *stateless*
- If you want to save state from one connection to the next, you store data in a cookie
- A natural thing webapp developers want:
  - Store data from my program in a cookie and then restore it on the next request
- This is a natural use case for serialization and deserialization

The Problem

Warning: The `pickle` module is not secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.

- In general, you can't unpickle untrusted data safely!
- …and cookies are untrusted data (their content is *completely* controlled by the client)

What Can Go Wrong?

- From the basic description it's clear *something* can go wrong here – but what, and how?
- It turns out unpickling untrusted data can result in *arbitrary Python code* being executed (!!)
- To see why we need to look more deeply into pickling

__reduce__, reuse, recycle

- Python knows how to pickle built-in types (lists, strings, numbers, etc.)
- But for your own custom classes, there may be custom behavior needed when pickling
  - E.g., some parts may not be pickleable, like open file descriptors
  - We may need to run some code to reconstruct the object's state properly when unpickling

__reduce__, reuse, recycle

- Python knows how to pickle built-in types (lists, strings, numbers, etc.)
- But for your own custom classes, there may be custom behavior needed when pickling
  - E.g., some parts may not be pickleable, like open file descriptors
  - We may need to ***run some code*** to reconstruct the object's state properly when unpickling

A Malicious __reduce__

```
import pickle
import os
class EvilPickle(object):
    def __reduce__(self):
        return (os.system, ('cat /etc/passwd', ))

pickle_data = pickle.dumps(EvilPickle())

with open("backup.data", "wb") as file:
    file.write(pickle_data)
```

Unpickling EvilPickle

- (Demo)

Maybe YAML is Better?

- Your first instinct may be to try another serialization format
- YAML (Yet Another Markup Language) is a popular choice for data interchange
- Unfortunately, the Python YAML library *also* allows execution of arbitrary code by default

Loading YAML

Warning: It is not safe to call `yaml.load` with any data received from an untrusted source! `yaml.load` is as powerful as `pickle.load` and so may call any Python function. Check the `yaml.safe_load` function though.

Exploiting YAML

```
import yaml
document = "!!python/object/apply:os.system ['cat /etc/passwd']"
yaml.load(document)
```

Safely Loading YAML

- The (easy) fix is to use safe_load
- This will refuse to load anything but basic types like lists, ints, strings, etc.
- (Demo)

Python is Not Unique

- Many other languages have serialization formats with similar issues:
  - Java: ObjectInputStream can deserialize any kind of class available in the namespace of the application
  - PHP: unserialize can create arbitrary objects (known as an *object injection* vulnerability)
  - Ruby: Marshal.load can create objects from the standard library, these can be combined to get full code execution
  - And so on…