Unexpected Execution: Wild Ways Remote Code Execution can Occur on Python Servers

Graham Bleaney (@GrahamBleaney)
Security Engineer

Ibrahim Mohamed (@the_st0rm)
Security Engineer

FACEBOOK

Agenda

1. Definitions and Motivation
2. Exploits
    1. Explicit Code Execution
    2. Deserialization
    3. `input`
    4. Custom RPC Frameworks
    5. Files on Disk
    6. String Formatting
    7. Execution Where you Least Expect It
3. Conclusion

Part 1: Intro

About Us

Graham Bleaney

- Security Engineer at Facebook
- Work on Python security

Ibrahim Mohamed

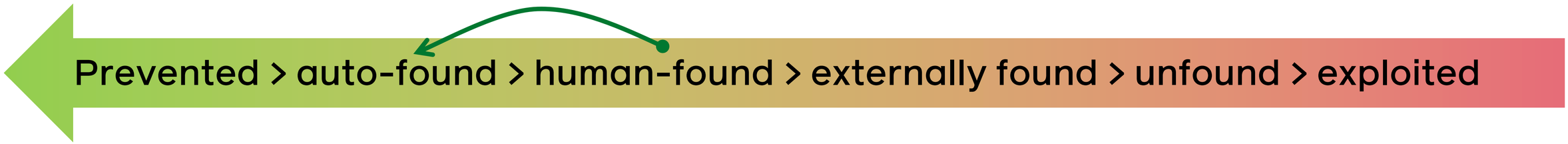
- Security Engineer at Facebook
- Worked on Hack and Python static analysis

Remote Code/Command Execution (RCE): Executing arbitrary instructions on a remote system

RCE Impact

- Potential outcomes:
  - Read private user data
  - Take down the server
  - Steal corporate secrets
  - Serve malware
- We need to prevent it

Preventing RCE – Shift Left



Automatically Finding Bugs – Pysa

- **Open source Python Static Analyzer for security**
- Tracks the flow of data through a program
- Can *automatically* find RCE given:
  - Where user controlled data originates
  - Which functions execute code

Problem: What Python functions execute code?

Talk Contents

- Summary of RCE vectors we've found for:
  - Python code (eg. `eval("print('RCE')")`)
  - Shell commands (eg. `subprocess.call(["echo", "RCE"])`)
- Demos
  - Framed as a webserver
  - Available at: https://github.com/gbleaney/python_security
  - Pause before we reveal the solution to try yourself

Why do you care?

- ...RCE is cool?
- RCE is high severity
- Knowledge -> Writing safer code
- All functions are listed in our GitHub repo
    - Use with Pysa or other static analyzers to protect your code

Part 2: Explicit Code Execution

Executing on the machine

- Standard builtins *intended* to execute code:

```
def entry_point(data: str):
    eval(data)
    exec(data)
```

- Standard library functions *intended* to execute commands:

```
def entry_point(data: str):
    os.system(data)
    subprocess.call(data.split(" "))
    asyncio.subprocess.create_subprocess_shell(data)
```

Oh wow that's a lot

```
def shell_entry_point(args: List[str]):
    program = args[0]
    path = find_in_path(program)
    remaining_args = args[1:]
    command = " ".join(args)
    environment = get_environment()

    subprocess.run(args)
    subprocess.check_call(args)
    subprocess.check_output(args)

    subprocess.Popen(args)
```

```
    os.system(command)


def python_entry_point(source_code: str):
    inperpreter = code.InteractiveInterpreter()
    inperpreter.runsource(source_code)
    inperpreter.runcode(code.compile_command(source_code))

    console = code.InteractiveConsole()
    console.push(source_code)

    test.support.run_in_subinterp(source_code)
    _testcapi.run_in_subinterp(source_code)
    _xxsubinterpreters.run_string(source_code)
```

Is anyone even reading these headings?

```
def python_entry_point(source_code: str):
    interpreter = code.InteractiveInterpreter()
    interpreter.runsource(source_code)
    interpreter.runcode(code.compile_command(source_code))

    console = code.InteractiveConsole()
    console.push(source_code)

    test.support.run_in_subinterp(source_code)
    _testcapi.run_in_subinterp(source_code)
    _xxsubinterpreters.run_string(source_code)
```

Remote remote code execution

- Libraries *intended* to execute commands on *other* machines:

```
from paramiko.client import SSHClient

def run_ssh(command)
    client = SSHClient()
    client.load_system_host_keys()
    client.connect('ssh.example.com')
    stdin, stdout, stderr = client.exec_command(command)
```

Source: http://docs.paramiko.org/en/stable/api/client.html#paramiko.client.SSHClient

Not the only one

```
def run_ssh(args: List[str])
    command = " ".join(args)

    # paramiko
    #   Source: http://docs.paramiko.org/en/stable/api/client.html#paramiko.client.SSHClient
    from paramiko.client import SSHClient
    client = SSHClient()
    client.load_system_host_keys()
    client.connect('ssh.example.com’)
    stdin, stdout, stderr = client.exec_command(command)

    # pexpect
    #   Source: https://pexpect.readthedocs.io/en/stable/api/pxssh.html
```

```
expect.login(
    hostname="ssh.example.com",
    username="username",
    password="password"
)
expect.command(command)

# netmiko (based on paramiko)
#  Source: https://pypi.org/project/netmiko/
from netmiko import ConnectHandler
net_connect = ConnectHandler(
    host="ssh.example.com",
    username="username",
    password="password"
)
output = net_connect.send_command(command)
```

Is that a bug, or a dust mite?

```python
def run_ssh(args: List[str]):
    command = " ".join(args)

    # paramiko
    # Source: https://docs.paramiko.org/en/stable/api/client.html#paramiko.client.SSHClient
    from paramiko.client import SSHClient
    client = SSHClient()
    client.load_system_host_keys()
    client.connect('ssh.example.com')
    stdin, stdout, stderr = client.exec_command(command)

    # pexpect
    # Source: https://pexpect.readthedocs.io/en/stable/api/pxssh.html
    from pexpect import pxssh
    s = pxssh.pxssh()
    s.login("ssh.example.com", "username", "password")
    s.sendline(command)

    # fabric
    # Source: https://docs.fabfile.org/en/1.12.1/tutorial.html
    from fabric.api import run
    run(command)

    # spur
    # Source: https://pypi.org/project/spur/
    import spur
    shell = spur.SshShell(
        hostname="ssh.example.com",
        username="username",
        password="password"
    )
    with shell:
        result = shell.run(args)

    # asyncssh
    # Source: https://asyncssh.readthedocs.io/en/latest/
    import asyncssh

    async with asyncssh.connect(
        'ssh.example.com',
        username="username",
        password="password"
    ) as conn:
        result = await conn.run(command)

    # ssh2-python
    # Source: https://pypi.org/project/ssh2-python/
    from ssh2.session import Session
    import socket

    sock = socket.socket(
        socket.AF_INET,
        socket.SOCK_STREAM
    )
    sock.connect(('ssh.example.com', 22))

    session = Session()
    session.handshake(sock)

    session.userauth_password("username", "password")

    channel = session.open_session()
    channel.execute(command)
    channel.wait_eof()
    channel.close()
    channel.wait_closed()

    # twisted.conch
    # Source: https://twistedmatrix.com/documents/current/conch/examples/index.html
    # https://stackoverflow.com/questions/22196373/sshcommandclientendpoint-twisted-how-to-execute-more-than-one-commands
    # https://twistedmatrix.com/documents/current/conch/howto/conch_client.html
    from twisted.conch.endpoints import SSHCommandClientEndpoint
    from twisted.internet.protocol import Factory
    from twisted.internet import reactor

    endpoint = SSHCommandClientEndpoint.newConnection(
        reactor,
        command,
        "username",
        "ssh.example.com",
        22,
        password="password"
    )
    factory = Factory()
    d = endpoint.connect(factory)
    d.addCallback(lambda protocol: protocol.finished)

    # trigger
    # Source: https://trigger.readthedocs.io/en/latest/examples.html#execute-commands-asynchronously-using-twisted
    from trigger.netdevices import NetDevices
    nd = NetDevices()
    dev = nd.find('ssh.example.com')
    dev.execute(command)

    # parallel-ssh
    # Source: https://github.com/ParallelSSH/parallel-ssh
    from pssh.clients import SSHClient

    client = SSHClient('ssh.example.com')
    host_out = client.run_command(command)

    # scrapli (transport via paramiko, ssh2, asyncssh)
    # Source: https://github.com/carlmontanari/scrapli
    from scrapli import Scrapli

    conn = Scrapli(
        host='ssh.example.com',
        auth_username="username",
        auth_password="password"
    )
    conn.open()
    conn.send_command(command)

    # redexpect (based on ssh2-python)
    # Source: https://github.com/Red-M/RedExpect/blob/master/examples/run_whoami.py
    import redexpect
    expect = redexpect.RedExpect()
    expect.login(
        hostname="ssh.example.com",
        username="username",
        password="password"
    )
    expect.command(command)

    # netmiko (based on paramiko)
    # Source: https://pypi.org/project/netmiko/
    from netmiko import ConnectHandler
    net_connect = ConnectHandler(
        host="ssh.example.com",
        username="username",
        password="password"
    )
    output = net_connect.send_command(command)
```

Prevention and Mitigation

- Don’t pass user-controlled input
- Prefer the functions that take a `list`, rather than a `str`
- Don’t try to make `eval` safe with tricks like `{'__builtins__':{}}`

Source: https://nedbatchelder.com/blog/201206/eval_really_is_dangerous.html

Part 3: Deserialization

Why so serial?

- Need to serialize/deserialize when data leaves/enters the program
  - Transmission
  - Storage
- Standard cross-language protocols:
  - JSON
  - YAML
  - XML
- Python-specific protocols:
  - Pickle
  - Marshal

The Holy Grail: Recreate *Any* Object

- Simple types (`list`, `str`, `dict`) are are easy to serialize to any format (eg. JSON)
  - What about arbitrary objects?
- `object.__reduce__()`
  - Return a `str` or `tuple` indicating how to recreate the object

```
class SomeClass:
  def __reduce__(self):
    return (function_to_call, (args, to, provide))
```

```
new_object = function_to_call(args, to, provide)
```

Source: https://docs.python.org/3/library/pickle.html#object.__reduce__

Demo

Vulnerable Libraries

- `pickle` (and libraries like `dill` and `shelve` that wrap it)
- PyYAML (deprecated in `load`, need to use "`unsafe`" APIs now)
- `jsonpickle`
- `marshal`?

Prevention and Mitigation

- Only deserialize trusted data
  - Sign data passing through untrusted parties
- Prefer simpler serialization formats for untrusted data
  - `json`
  - `msgpack`

Part 4: `input`

Parsing made easy

- Easiest way to convert arbitrary strings to the right data type?
  - `eval`!

```
>>> eval("1")
1 # int
```

```
>>> eval("'one'")
'one' # str
```

~~Parsing~~ Exploits made easy

- How does `input()` work (Python 2.x only)?
  - `eval`!

```
>>> print "Input: %s" % input()
1
Input: 1 # int
```

```
>>> print "Input: %s" % input()
1+1
Input: 2 # oops
```

Demo

Prevention and Mitigation

- Run Python 3
- Use `raw_input()`

Part 5: Custom RPC Framework

Demo

Why write this?

- Remote procedure calls
- Augmenting simple deserialization libraries

Prevention and Mitigation

- Don’t expose this code to untrusted data
- Allowlist callable functions

Part 6: Files on Disk

Controlling Files on Disk

- Python imports perform two operations:
  - Searches for the named module
  - Binds the module to the scope

```python
# arithmetic.py
def sum(a: int, b: int) -> int:
    return a + b
```

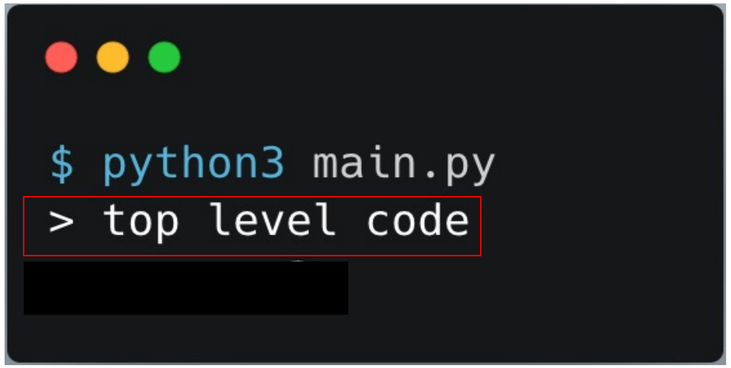
```python
# main.py
import arithmetic
res = arithmetic.sum(1,2)
print("res = ", res) # res = 3
```

```
$ python3 main.py
> res =  3
```

Controlling Files on Disk

```python
# arithmetic.py
print("top level code")
def sum(a: int, b: int) -> int:
    return a + b
```

```python
# main.py
import arithmetic
```



Controlling Files on Disk

- User-controlled file write (path + content)
- Python imports execute top level code
- Importing user-controlled modules/files leads to RCE

Vulnerable Code – import arbitrary module

```
def write_file(path, content):
    with open(path, "w") as f:
        f.write(content)

def import_user_module():
    import helper
    # Do stuff
```

Demo

Controlling Search path control

- Module search depends on `sys.path`
- Controlling the search path

```
import sys
sys.path = ["dir2", "dir1", "."]
print(sys.path) # [‘dir2', ‘dir1', '.']

import arithmetic
```

Vulnerable Code

```
sys.path.insert(0, user_controlled_value)
...
import requests
```

Prevention and Mitigation

- Do not import untrusted modules/code
- Separate the location of uploaded user data and your code
- Avoid untrusted locations in your search path

Part 7: String Formatting

String Formatting

- `str.format`
  - Format string contains literals and replacement fields with `{}`
  - Replaces fields with content

```
name = "PyCon2021"
fmt = "Conference: {name}"
fmt.format(name=name)
# 'Conference: PyCon2021'
```

String Formatting

- `str.format` looks innocent but control over the string can lead to:
  - Leaking data at a *minimum*
  - RCE in the right settings

```
name = "PyCon2021"
fmt = "Conference: {name}"
fmt.format(name=name)
# 'Conference: PyCon2021'
```

String Formatting – Controlling the format

```
class logMsg(object):
  def __init__(self, msg, lvl):
    self.msg = msg
    self.lvl = lvl

fmt = "{obj.lvl}: {obj.msg}"
fmt.format(obj=logMsg("test", 1))
# '1: test'
```

String Formatting – Controlling the format

```
CONFIG = { 'SECRET_KEY': 'super secret key'}


class logMsg(object):
  def __init__(self, msg, lvl):
    self.msg = msg
    self.lvl = lvl

fmt = "{obj.__init__.__globals__[CONFIG][SECRET_KEY]}:
{obj.msg}"
fmt.format(obj=logMsg("test", 1))
# 'super secret key: test'
```

String Formatting

- Template engines – fancier string formatting

```
import jinja2
t = jinja2.Template("""
My favorite numbers:
{% for n in range(1,10) %}
{{n}}
{% endfor %}
""")
t.render()
```

```
> My favorite numbers:
1
2
3
4
5
6
7
8
9
```

Demo

Vulnerable Libraries

- Jinja2
- Tornado
- Mako
- Chameleon
- Cheetah
- Genshi
- Trender
- Chevron
- Airspeed

Source: https://github.com/github/securitylab/issues/93

Prevention and Mitigation

- Do not let untrusted input to be part of your template
- For Jinja templates use SandboxedEnvironment for parsing user-controlled templates

Part 8: Execution Where you Least Expect It

Vulnerable Code - CVE-2021-3177

```
from ctypes import *
c_double.from_param(untrusted_data)
```

PoC - CVE-2021-3177

```
from ctypes import *
c_double.from_param(1e300)
*** buffer overflow detected ***: terminated
Aborted
```

Source: https://bugs.python.org/issue42938

Vulnerable Code – type hints

```
from typing import get_type_hints

class C:
    member: int = 1

get_type_hints(C)
```

PoC – type hints

```
from typing import get_type_hints

class C:
    member: "print('test')" = 2

get_type_hints(C)
```

Part 9: Conclusion

Conclusion

- Know the APIs you are using
- Use static analysis

Product security processes - Pysa

- Most techniques discussed here have coverage in Pysa
- Open source
- Supports multi-million line codebases
- Try it with our quickstart: https://pyre-check.org/docs/pysa-quickstart/

Want more?

- Visit: https://github.com/gbleaney/python_security
  - Demos
  - Full list of known sinks
- Did we miss something?
  - Send a PR
  - Tweet at @GrahamBleaney or @the_st0rm

Thank you!

- Jennifer Martinez
- Ted Reed
- Blake Matheny
- Stephanie Ding
- Maxime Arthaud
- Lorenzo Fontana
- David Molnar
- Dominik Gabi
- Pieter Hooimeijer
- Collin Greene
- Margarita Zolotova
- Gabby Curtis
- Mike Hollander
- Edward Qiu

FACEBOOK