Fuzzing – Part 2

Mitchell Adair
utdcsg.org

Outline

- Debugging libraries (for Windows)
  - WinAppDbg, PyDBG
    - Examples
    - Pros and con
- Fuzzer design
  - Design concepts
  - Fuzzer goals
  - Github
  - Future work

Debugging Libraries

PyDBG

- PyDBG
  - “A pure-python win32 debugger interface.”
  - Part of the Paimei reverse engineering framework
    - Awesome
  - Created by Pedram Amini
    - Badass, you should be following him on Twitter etc.

- https://github.com/OpenRCE/pydbg

PyDBG

- So… what can it do?
  - Launch or attach to processes
  - Breakpoints, step into, step over, etc.
  - Get / set memory or register values
  - Give you access to PEB
  - Resolve functions
  - Disassemble
  - Set callbacks for signals, events, breakpoints, etc.
  - Snapshots
  - … (seriously)

- And… you can use it stand-alone, or from **within IDA**!

PyDBG

- How is this different from Immunity, OllyDBG, etc?
  - It's scriptable!
- How about automating…
  - Unpacking
  - Malware analysis
    - General statistics, system calls of interest, etc.
  - Crash analysis
    - Trace my path, save operand values, etc.
  - Fuzzing!
    - Debug a process, set callbacks on signals of interest, log the run…
    - In memory fuzzing with snapshots

PyDBG

- Let's see some examples!

PyDBG – Example 1

- Create a debugging object
- Load the target executable
- Run it

```
1 from pydbg import *
2
3 dbg = pydbg()
4 dbg.load(r"C:\Windows\System32\notepad.exe")
5 dbg.run()
```

- Pretty painless

PyDBG - Callbacks

- From the interpreter

```
>>> help(dbg.set_callback)
Help on method set_callback in module pydbg.pydbg:

set_callback(self, exception_code, callback_func) method of pydbg.pydbg.pydbg instance
    Set a callback for the specified exception (or debug event) code. The prototype of the callback routines is::

        func (pydbg):
            return DBG_CONTINUE     # or other continue status

    You can register callbacks for any exception code or debug event. Look in the source for all event_handler_???
    and exception_handler_??? routines to see which ones have internal processing (internal handlers will still
    pass control to your callback). You can also register a user specified callback that is called on each loop
    iteration from within debug_event_loop(). The callback code is USER_CALLBACK_DEBUG_EVENT and the function
    prototype is::
```

- The entire dbg object is passed to the callback handler
- Some sort of continue status is returned

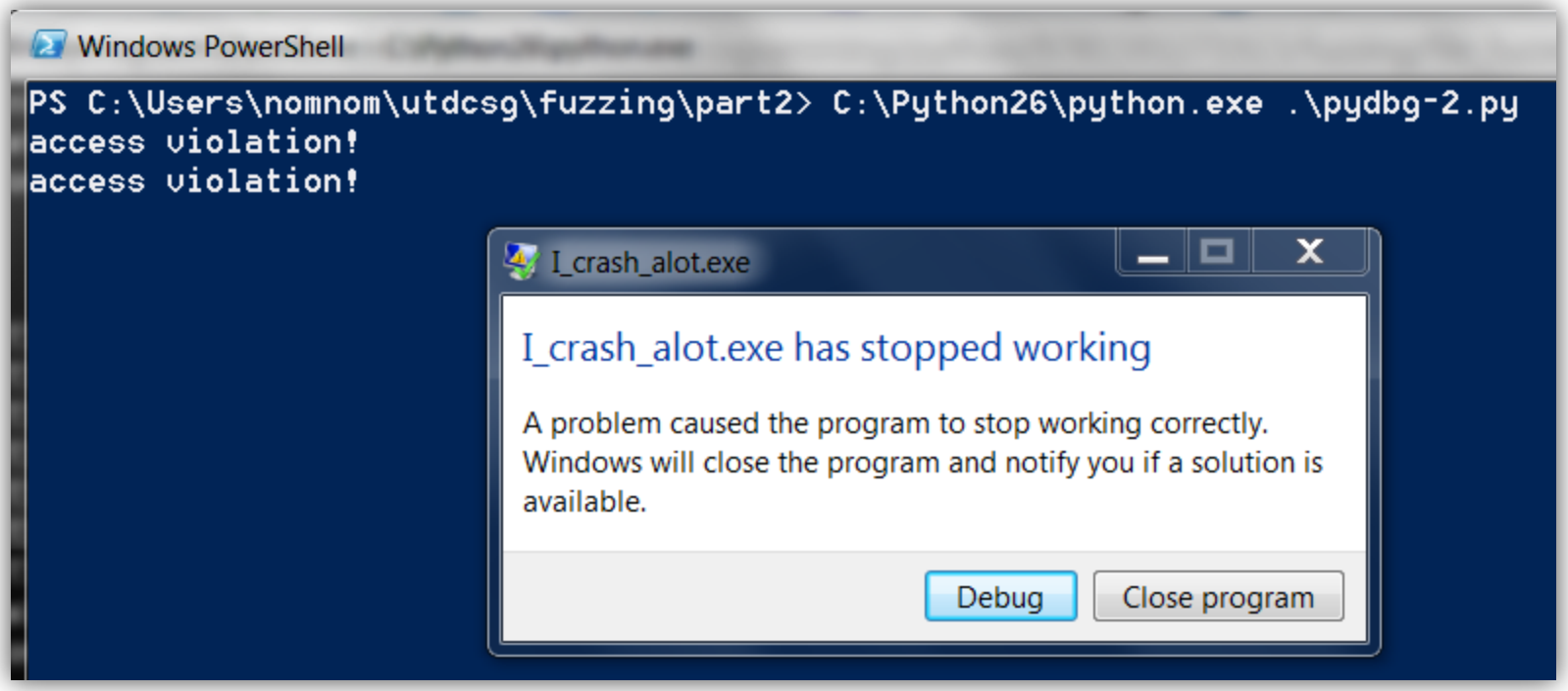
PyDBG – Example 2

- Let's handle some signals. How about access violation

```
1  from pydbg import *
2  from pydbg.defines import *
3
4  def handle_av(dbg):
5      print 'access violation!'
6      return DBG_EXCEPTION_NOT_HANDLED
7
8  dbg = pydbg()
9  dbg.set_callback(EXCEPTION_ACCESS_VIOLATION, handle_av)
10 dbg.load(r"C:\I_crash_alot.exe")
11 dbg.run()
12
13
14
```

- On Microsoft Windows, a process that accesses invalid memory receives the STATUS_ACCESS_VIOLATION exception.
  - Wikipedia

PyDBG – Example 2



PyDBG – Example 2

- Why do we care about access violations?
  - "invalid memory" = ?
  - Virtual memory that does not map to physical memory
  - Virtual memory marked with permissions, and the process does not have permission to perform the operation
    - Memory is read/write/executable
    - Trying to perform a read on non-readable memory… access violation
- We are typically trying to influence pointers, influence length values, overflow boundaries, etc.
- The above usually results in access violations
- Illegal instruction is another good signal (usually means we messed with EIP and it now points to an invalid instruction)

PyDBG – Example 3

- We can
  - Launch or attach to an application
  - Set our callback handlers
  - Run the application
- But… we want to collect as much information as possible from the access violation handler
- Paimei comes with the great util, crash_binning.py that will record lots of useful information

PyDBG – Example 3

- Just create a crash_binning object and record the crash with the dbg object passed to the callback handler

```
64  def record_crash (self, pydbg, extra=None):
65      '''
66      Given a PyDbg instantiation that at the current time is assumed to have "crashed" (access violation for example)
67      record various details such as the disassemly around the violating address, the ID of the offending thread, the
68      call stack and the SEH unwind. Store the recorded data in an internal dictionary, binning them by the exception
69      address.
70
71      @type  pydbg: pydbg
72      @param pydbg: Instance of pydbg
73      @type  extra: Mixed
74      @param extra: (Optional, Def=None) Whatever extra data you want to store with this bin
75      '''
```

PyDBG – Example 3

```
1  from pydbg import *
2  from pydbg.defines import *
3  import utils
4
5  def handle_av(dbg):
6      crash_bin = utils.crash_binning.crash_binning()
7      crash_bin.record_crash(dbg)
8      print crash_bin.crash_synopsis()
9
10     dbg.terminate_process()
11     return DBG_EXCEPTION_NOT_HANDLED
12
13 dbg = pydbg()
14 dbg.set_callback(EXCEPTION_ACCESS_VIOLATION, handle_av)
15 dbg.load(r"C:\I_crash_alot.exe")
16 dbg.run()
```

- That's a pretty powerful 16 lines of code…

```
PS C:\Users\nomnom\utdcsg\fuzzing\part2> python .\pydbg-3.py
I_crash_alot.exe:004013d3 mov eax,[eax] from thread 3984 caused access violation
when attempting to read from 0xdeadbeef

CONTEXT DUMP
  EIP: 004013d3 mov eax,[eax]
  EAX: deadbeef (3735928559) -> N/A
  EBX: 7efde000 (2130567168) -> N/A
  ECX: 00000001 (         1) -> N/A
  EDX: 0008e3c8 (    582600) -> uuPuuLuwu@xwP$u ~[u4(uP$u (heap)
  EDI: 00000000 (         0) -> N/A
  ESI: 00000000 (         0) -> N/A
  EBP: 0028ff28 (   2686760) -> h( (stack)
  ESP: 0028ff10 (   2686736) -> p@ (stack)
  +00: 00401970 (   4200816) -> N/A
  +04: 006d4bd0 (   7162832) -> "C:\I_crash_alot.exe" (heap)
  +08: 00000015 (        21) -> N/A
  +0c: 7efde000 (2130567168) -> N/A
  +10: 7efde000 (2130567168) -> N/A
  +14: 00000000 (         0) -> N/A

disasm around:
        0x004013b5 lea esi,[esi+0x0]
        0x004013b8 mov eax,0x0
        0x004013bd jmp 0x4013a6
        0x004013bf nop
        0x004013c0 push ebp
        0x004013c1 mov ebp,esp
        0x004013c3 and esp,0xfffffff0
        0x004013c6 sub esp,0x10
        0x004013c9 call 0x401a00
        0x004013ce mov eax,0xdeadbeef
        0x004013d3 mov eax,[eax]
        0x004013d5 mov [esp+0xc],eax
        0x004013d9 mov eax,[esp+0xc]
        0x004013dd leave
        0x004013de ret
        0x004013df nop
        0x004013e0 push ebp
        0x004013e1 xor eax,eax
        0x004013e3 mov ebp,esp
        0x004013e5 pop ebp
        0x004013e6 ret

stack unwind:
        I_crash_alot.exe:004010db
        I_crash_alot.exe:00401178
        kernel32.dll:76a8339a
        ntdll.dll:77b29ef2
        ntdll.dll:77b29ec5

SEH unwind:
        ffffffff -> ntdll.dll:77b671d5 mov edi,edi
```

PyDBG – Example 3

- Sample output from crash_binning
- Registers, assembly, stack trace, SEH
- All with a function call, so easy!

PyDBG

- Now import multiprocessing
- Mutate some files
- Launch the target application with the new files
- Find bugs ☺

Debugging Libraries

- WinAppDbg
- "The *WinAppDbg* python module allows developers to quickly code instrumentation scripts in **Python** under a **Windows** environment."
- "It uses **ctypes** to wrap many Win32 API calls related to debugging..."
- "The intended audience are QA engineers and software security auditors wishing to test or fuzz Windows applications with quickly coded Python scripts."
- http://winappdbg.sourceforge.net/

WinAppDbg

- Why not just stick with PyDBG?
  - Rumor has it PyDBG development has become OSX focused
  - It rocks, but it's a little old and antiquated
  - Might have to write some wrappers, depending on your usage
- WinAppDbg is *only* windows, but it has a *ton* of stuff to work with
- If you're doing heavy PE work WinAppDbg might be the way to go

WinAppDbg

- The WinAppDbg site has some great examples
  - http://winappdbg.sourceforge.net/ProgrammingGuide.html
  - Instrumentation
    - Enumerating processes, loading a DLL into a process, control windows
  - Debugging
    - Starting and attaching, handling events, breakpoints, etc.
  - Win32 API wrappers
    - Enumerating heap blocks, modules and device drivers
  - Misc
    - Dump process memory, find alphanumeric jump addresses, etc.
- We'll compare WinAppDbg with our last PyDBG example, then show one more interesting example

WinAppDbg – Example 1

- Picking up where we left off with PyDBG

A custom event handler is optional, but is an easy way to catch any signals of interest

```
1  from winappdbg import Debug, EventHandler
2
3  # create our custom event handler
4  class MyEventHandler(EventHandler):
5      def __init__(self):
6          super(MyEventHandler, self).__init__()   # call our super class
7
8      # these functions will be called if their signal occurs
9      def access_violation(self, event):
10         self.handleSignal(event)
11     def illegal_instruction(self, event):
12         self.handleSignal(event)
13
14     def handleSignal(self, event):
15         # gather data or handle the signal how we like
16         # print registers, stack, etc.
17         pass
18
19 # initialize the handler, and the debugger to use it
20 handler = MyEventHandler()
21 debug = Debug(handler)
22 # launch the application, enter the debugging loop
23 debug.execl(r'C:\Windows\system32\notepad.exe')
24 debug.loop()
```

```
from winappdbg.win32 import PVOID


# This function will be called when the hooked function is entered.
def wsprintf( event, ra, lpOut, lpFmt ):

    # Get the format string.
    process = event.get_process()
    lpFmt   = process.peek_string( lpFmt, fUnicode = True )

    # Get the vararg parameters.
    count      = lpFmt.replace( '%%', '%' ).count( '%' )
    thread     = event.get_thread()
    if process.get_bits() == 32:
        parameters = thread.read_stack_dwords( count, offset = 3 )
    else:
        parameters = thread.read_stack_qwords( count, offset = 3 )

    # Show a message to the user.
    showparams = ", ".join( [ hex(x) for x in parameters ] )
    print "wsprintf( %r, %s );" % ( lpFmt, showparams )


class MyEventHandler( EventHandler ):

    def load_dll( self, event ):

        # Get the new module object.
        module = event.get_module()

        # If it's user32...
        if module.match_name("user32.dll"):

            # Get the process ID.
            pid = event.get_pid()

            # Get the address of wsprintf.
            address = module.resolve( "wsprintfW" )

            # This is an approximated signature of the wsprintf function.
            # Pointers must be void so ctypes doesn't try to read from them.
            # Varargs are obviously not included.
            signature = ( PVOID, PVOID )

            # Hook the wsprintf function.
            event.debug.hook_function( pid, address, wsprintf, signature = signature)
```

Example 2

- Hooking a function, wsprintfW
- Catch the load_dll signal
- If it's user32.dll, resolve wsprintf, hook it
- Print the args

```
from winappdbg.win32 import PVOID


# This function will be called when the hooked function is entered.
def wsprintf( event, ra, lpOut, lpFmt ):

    # Get the format string.
    process = event.get_process()
    lpFmt   = process.peek_string( lpFmt, fUnicode = True )

    # Get the vararg parameters.
    count      = lpFmt.replace( '%%', '%' ).count( '%' )
    thread     = event.get_thread()
    if process.get_bits() == 32:
        parameters = thread.read_stack_dwords( count, offset = 3 )
    else:
        parameters = thread.read_stack_qwords( count, offset = 3 )

    # Show a message to the user.
    showparams = ", ".join( [ hex(x) for x in parameters ] )
    print "wsprintf( %r, %s );" % ( lpFmt, showparams )


class MyEventHandler( EventHandler ):

    def load_dll( self, event ):

        # Get the new module object.
        module = event.get_module()

        # If it's user32...
        if module.match_name("user32.dll"):

            # Get the process ID.
            pid = event.get_pid()

            # Get the address of wsprintf.
            address = module.resolve( "wsprintfW" )

            # This is an approximated signature of the wsprintf function.
            # Pointers must be void so ctypes doesn't try to read from them.
            # Varargs are obviously not included.
            signature = ( PVOID, PVOID )

            # Hook the wsprintf function.
            event.debug.hook_function( pid, address, wsprintf, signature = signature)
```

1. Catch load_dll signal

Example 2

- Hooking a function, wsprintfW
- Catch the load_dll signal
- If it's user32.dll, resolve wsprintf, hook it
- Print the args

```
from winappdbg.win32 import PVOID


# This function will be called when the hooked function is entered.
def wsprintf( event, ra, lpOut, lpFmt ):

    # Get the format string.
    process = event.get_process()
    lpFmt   = process.peek_string( lpFmt, fUnicode = True )

    # Get the vararg parameters.
    count      = lpFmt.replace( '%%', '%' ).count( '%' )
    thread     = event.get_thread()
    if process.get_bits() == 32:
        parameters = thread.read_stack_dwords( count, offset = 3 )
    else:
        parameters = thread.read_stack_qwords( count, offset = 3 )

    # Show a message to the user.
    showparams = ", ".join( [ hex(x) for x in parameters ] )
    print "wsprintf( %r, %s );" % ( lpFmt, showparams )


class MyEventHandler( EventHandler ):

    def load_dll( self, event ):

        # Get the new module object.
        module = event.get_module()

        # If it's user32...
        if module.match_name("user32.dll"):

            # Get the process ID.
            pid = event.get_pid()

            # Get the address of wsprintf.
            address = module.resolve( "wsprintfW" )

            # This is an approximated signature of the wsprintf function.
            # Pointers must be void so ctypes doesn't try to read from them.
            # Varargs are obviously not included.
            signature = ( PVOID, PVOID )

            # Hook the wsprintf function.
            event.debug.hook_function( pid, address, wsprintf, signature = signature)
```

1. Catch load_dll signal

2. If it's user32.dll

Example 2

- Hooking a function, wsprintfW
- Catch the load_dll signal
- If it's user32.dll, resolve wsprintf, hook it
- Print the args

Example 2

```
from winappdbg.win32 import PVOID


# This function will be called when the hooked function is entered.
def wsprintf( event, ra, lpOut, lpFmt ):

    # Get the format string.
    process = event.get_process()
    lpFmt   = process.peek_string( lpFmt, fUnicode = True )

    # Get the vararg parameters.
    count      = lpFmt.replace( '%%', '%' ).count( '%' )
    thread     = event.get_thread()
    if process.get_bits() == 32:
        parameters = thread.read_stack_dwords( count, offset = 3 )
    else:
        parameters = thread.read_stack_qwords( count, offset = 3 )

    # Show a message to the user.
    showparams = ", ".join( [ hex(x) for x in parameters ] )
    print "wsprintf( %r, %s );" % ( lpFmt, showparams )


class MyEventHandler( EventHandler ):

    def load_dll( self, event ):

        # Get the new module object.
        module = event.get_module()

        # If it's user32...
        if module.match_name("user32.dll"):

            # Get the process ID.
            pid = event.get_pid()

            # Get the address of wsprintf.
            address = module.resolve( "wsprintfW" )

            # This is an approximated signature of the wsprintf function.
            # Pointers must be void so ctypes doesn't try to read from them.
            # Varargs are obviously not included.
            signature = ( PVOID, PVOID )

            # Hook the wsprintf function.
            event.debug.hook_function( pid, address, wsprintf, signature = signature)
```

1. Catch load_dll signal
2. If it's user32.dll
3. Resolve "wsprintfW"

- Hooking a function, wsprintfW
- Catch the load_dll signal
- If it's user32.dll, resolve wsprintf, hook it
- Print the args

Example 2

```
from winappdbg.win32 import PVOID


# This function will be called when the hooked function is entered.
def wsprintf( event, ra, lpOut, lpFmt ):

    # Get the format string.
    process = event.get_process()
    lpFmt   = process.peek_string( lpFmt, fUnicode = True )

    # Get the vararg parameters.
    count      = lpFmt.replace( '%%', '%' ).count( '%' )
    thread     = event.get_thread()
    if process.get_bits() == 32:
        parameters = thread.read_stack_dwords( count, offset = 3 )
    else:
        parameters = thread.read_stack_qwords( count, offset = 3 )

    # Show a message to the user.
    showparams = ", ".join( [ hex(x) for x in parameters ] )
    print "wsprintf( %r, %s );" % ( lpFmt, showparams )


class MyEventHandler( EventHandler ):

    def load_dll( self, event ):

        # Get the new module object.
        module = event.get_module()

        # If it's user32...
        if module.match_name("user32.dll"):

            # Get the process ID.
            pid = event.get_pid()

            # Get the address of wsprintf.
            address = module.resolve( "wsprintfW" )

            # This is an approximated signature of the wsprintf function.
            # Pointers must be void so ctypes doesn't try to read from them.
            # Varargs are obviously not included.
            signature = ( PVOID, PVOID )

            # Hook the wsprintf function.
            event.debug.hook_function( pid, address, wsprintf, signature = signature)
```

1. Catch load_dll signal
2. If it's user32.dll
3. Resolve "wsprintfW"
4. Hook it

- Hooking a function, wsprintfW
- Catch the load_dll signal
- If it's user32.dll, resolve wsprintf, hook it
- Print the args

```
from winappdbg.win32 import PVOID


# This function will be called when the hooked function is entered.
def wsprintf( event, ra, lpOut, lpFmt ):

    # Get the format string.
    process = event.get_process()
    lpFmt   = process.peek_string( lpFmt, fUnicode = True )

    # Get the vararg parameters.
    count      = lpFmt.replace( '%%', '%' ).count( '%' )
    thread     = event.get_thread()
    if process.get_bits() == 32:
        parameters = thread.read_stack_dwords( count, offset = 3 )
    else:
        parameters = thread.read_stack_qwords( count, offset = 3 )

    # Show a message to the user.
    showparams = ", ".join( [ hex(x) for x in parameters ] )
    print "wsprintf( %r, %s );" % ( lpFmt, showparams )


class MyEventHandler( EventHandler ):

    def load_dll( self, event ):

        # Get the new module object.
        module = event.get_module()

        # If it's user32...
        if module.match_name("user32.dll"):

            # Get the process ID.
            pid = event.get_pid()

            # Get the address of wsprintf.
            address = module.resolve( "wsprintfW" )

            # This is an approximated signature of the wsprintf function.
            # Pointers must be void so ctypes doesn't try to read from them.
            # Varargs are obviously not included.
            signature = ( PVOID, PVOID )

            # Hook the wsprintf function.
            event.debug.hook_function( pid, address, wsprintf, signature = signature)
```

5. wsprintf hit at run time

1. Catch load_dll signal

2. If it's user32.dll

3. Resolve "wsprintfW"

4. Hook it

Example 2

- Hooking a function, wsprintfW
- Catch the load_dll signal
- If it's user32.dll, resolve wsprintf, hook it
- Print the args

Example 2

```
from winappdbg.win32 import PVOID


# This function will be called when the hooked function is entered.
def wsprintf( event, ra, lpOut, lpFmt ):

    # Get the format string.
    process = event.get_process()
    lpFmt   = process.peek_string( lpFmt, fUnicode = True )

    # Get the vararg parameters.
    count      = lpFmt.replace( '%%', '%' ).count( '%' )
    thread     = event.get_thread()
    if process.get_bits() == 32:
        parameters = thread.read_stack_dwords( count, offset = 3 )
    else:
        parameters = thread.read_stack_qwords( count, offset = 3 )

    # Show a message to the user.
    showparams = ", ".join( [ hex(x) for x in parameters ] )
    print "wsprintf( %r, %s );" % ( lpFmt, showparams )


class MyEventHandler( EventHandler ):

    def load_dll( self, event ):

        # Get the new module object.
        module = event.get_module()

        # If it's user32...
        if module.match_name("user32.dll"):

            # Get the process ID.
            pid = event.get_pid()

            # Get the address of wsprintf.
            address = module.resolve( "wsprintfW" )

            # This is an approximated signature of the wsprintf function.
            # Pointers must be void so ctypes doesn't try to read from them.
            # Varargs are obviously not included.
            signature = ( PVOID, PVOID )

            # Hook the wsprintf function.
            event.debug.hook_function( pid, address, wsprintf, signature = signature)
```

5. wsprintf hit at run time

6. Dereference format string

1. Catch load_dll signal

2. If it's user32.dll

3. Resolve "wsprintfW"

4. Hook it

- Hooking a function, wsprintfW
- Catch the load_dll signal
- If it's user32.dll, resolve wsprintf, hook it
- Print the args

Example 2

```
from winappdbg.win32 import PVOID


# This function will be called when the hooked function is entered.
def wsprintf( event, ra, lpOut, lpFmt ):

    # Get the format string.
    process = event.get_process()
    lpFmt   = process.peek_string( lpFmt, fUnicode = True )

    # Get the vararg parameters.
    count      = lpFmt.replace( '%%', '%' ).count( '%' )
    thread     = event.get_thread()
    if process.get_bits() == 32:
        parameters = thread.read_stack_dwords( count, offset = 3 )
    else:
        parameters = thread.read_stack_qwords( count, offset = 3 )

    # Show a message to the user.
    showparams = ", ".join( [ hex(x) for x in parameters ] )
    print "wsprintf( %r, %s );" % ( lpFmt, showparams )


class MyEventHandler( EventHandler ):

    def load_dll( self, event ):

        # Get the new module object.
        module = event.get_module()

        # If it's user32...
        if module.match_name("user32.dll"):

            # Get the process ID.
            pid = event.get_pid()

            # Get the address of wsprintf.
            address = module.resolve( "wsprintfW" )

            # This is an approximated signature of the wsprintf function.
            # Pointers must be void so ctypes doesn't try to read from them.
            # Varargs are obviously not included.
            signature = ( PVOID, PVOID )

            # Hook the wsprintf function.
            event.debug.hook_function( pid, address, wsprintf, signature = signature)
```

5. wsprintf hit at run time
6. Dereference format string
7. Count args
1. Catch load_dll signal
2. If it's user32.dll
3. Resolve "wsprintfW"
4. Hook it

- Hooking a function, wsprintfW
- Catch the load_dll signal
- If it's user32.dll, resolve wsprintf, hook it
- Print the args

Example 2

```
from winappdbg.win32 import PVOID


# This function will be called when the hooked function is entered.
def wsprintf( event, ra, lpOut, lpFmt ):

    # Get the format string.
    process = event.get_process()
    lpFmt   = process.peek_string( lpFmt, fUnicode = True )

    # Get the vararg parameters.
    count      = lpFmt.replace( '%%', '%' ).count( '%' )
    thread     = event.get_thread()
    if process.get_bits() == 32:
        parameters = thread.read_stack_dwords( count, offset = 3 )
    else:
        parameters = thread.read_stack_qwords( count, offset = 3 )

    # Show a message to the user.
    showparams = ", ".join( [ hex(x) for x in parameters ] )
    print "wsprintf( %r, %s );" % ( lpFmt, showparams )


class MyEventHandler( EventHandler ):

    def load_dll( self, event ):

        # Get the new module object.
        module = event.get_module()

        # If it's user32...
        if module.match_name("user32.dll"):

            # Get the process ID.
            pid = event.get_pid()

            # Get the address of wsprintf.
            address = module.resolve( "wsprintfW" )

            # This is an approximated signature of the wsprintf function.
            # Pointers must be void so ctypes doesn't try to read from them.
            # Varargs are obviously not included.
            signature = ( PVOID, PVOID )

            # Hook the wsprintf function.
            event.debug.hook_function( pid, address, wsprintf, signature = signature)
```

5. wsprintf hit at run time
6. Dereference format string
7. Count args
8. Read off stack, print args
1. Catch load_dll signal
2. If it's user32.dll
3. Resolve "wsprintfW"
4. Hook it

- Hooking a function, wsprintfW
- Catch the load_dll signal
- If it's user32.dll, resolve wsprintf, hook it
- Print the args

WinAppDbg

- Way too many great examples on their site to go into
  - Hooking functions
  - Watching variables
  - Watching buffers
  - Etc… very powerfull
- If you want to automate anything PE related, this is a great library to look into

Fuzzer Design

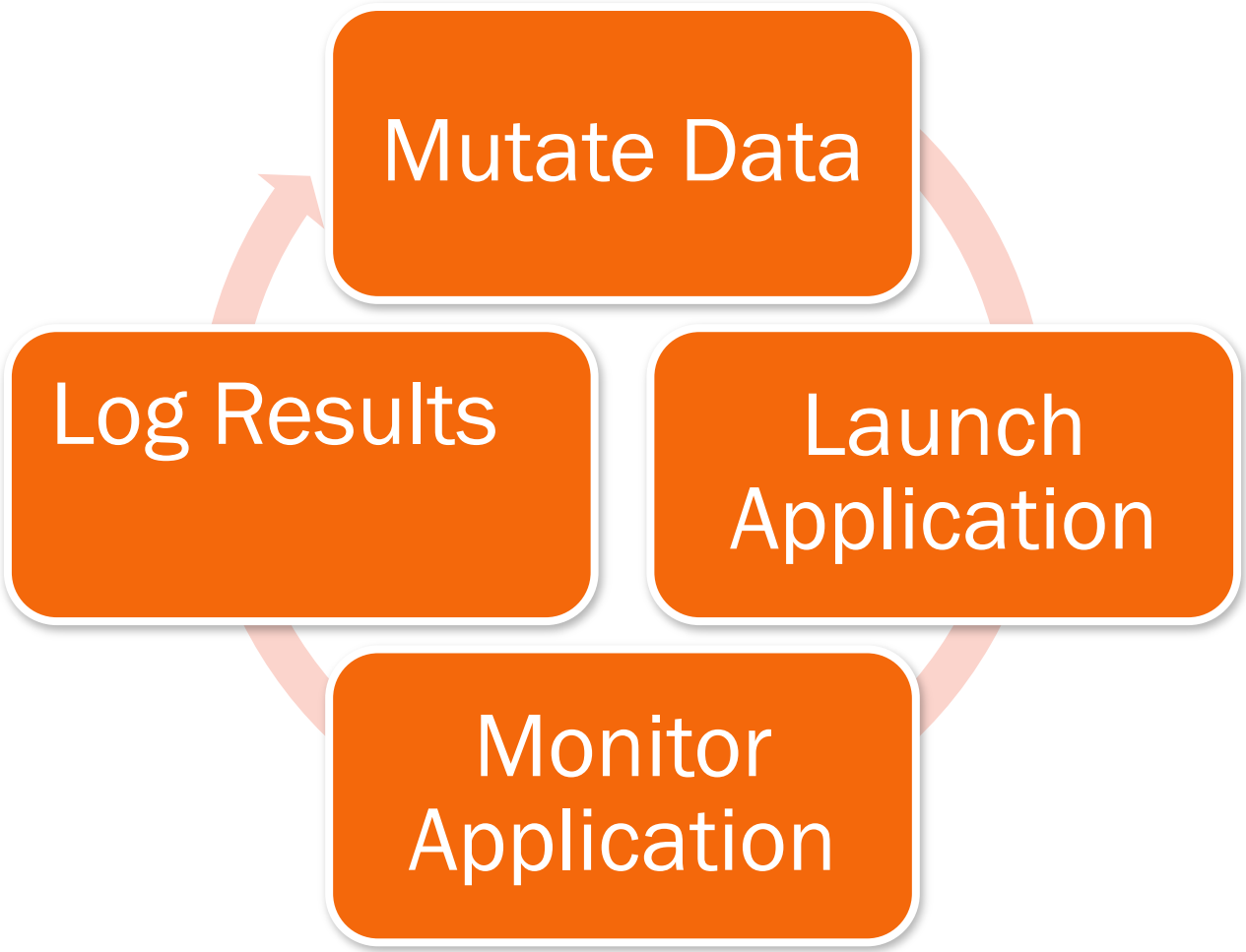
Fuzzer Design

- Design goals
  - Modularity
    - Ex: generator, executor, monitor
  - Reusability
    - A new target program or file type should make little to no difference
  - Speed
    - A large file might have hundreds of thousands of mutations
    - Multiprocessing or a distributed architecture is helpful
  - False negatives
    - We don't want to miss anything…

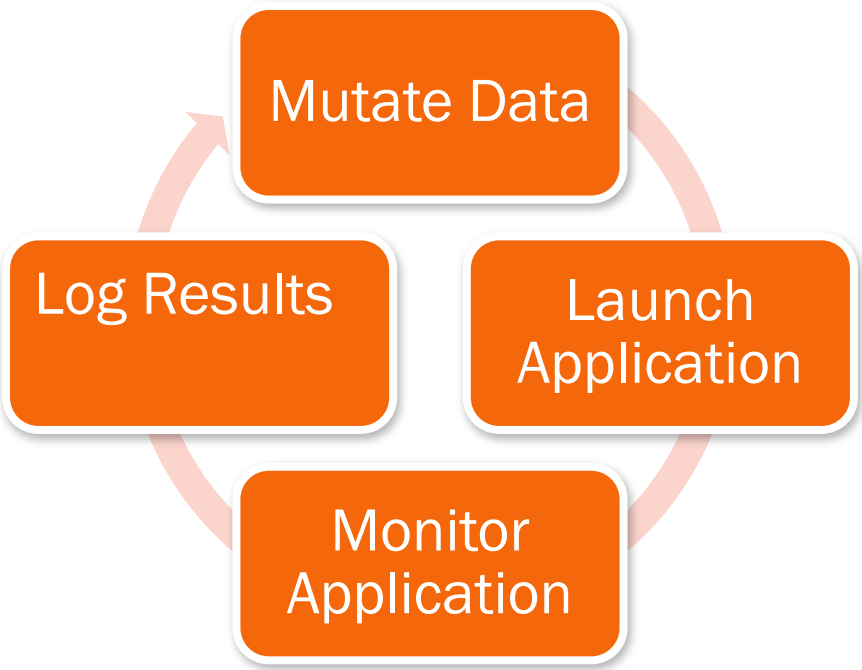
Fuzzer Design - Modularity

- What are the general tasks performed during fuzzing?
  - Generating mutated data
  - Launching the target application
  - Sending the data to the application
  - Monitoring the application for signals of interest
  - Logging results
  - …more?

Fuzzer Design - Modularity



Fuzzer Design - Modularity



Mutator.py

?

Executor.py

Fuzzer.py

Mutatory.py

- Part 1 discussed possible values you may want to try

```
1  class Mutator():
2      ''' itterate over the contents of a given file, mutate the contents '''
3
4      def yieldNext(self):
5          ''' yield a new file with mutated contents '''
6
7          for offset in range(self.file_contents):
8              for value in self.mutation_values:
9                  # replace bytes
10
11                 newfile = open('name', 'wb')
12                 newfile.write(replaced_bytes)
13                 newfile.close()
14
15                 yield('name')
16
```

- Yield is a nice python feature
- Sole job is to mutate the bytes, any changes in possible values can easily be handled here

```python
class Executor():
    def __init__(self, timeout, queue_in, queue_out):
        self.timeout   = timeout
        self.queue_in  = queue_in
        self.queue_out = queue_out
        self.enterLoop()
        self.obj = None

    def enterLoop(self):
        while True:
            try:
                obj = self.queue_in.get_nowait()
            except:
                sleep(.1)
                continue

            if obj == 'STOP':
                break

            self.obj = obj
            self.execute(obj)

            if not 'crash' in self.obj:
                self.obj['crash'] = False
                self.obj['output'] = None

            self.queue_out.put(self.obj)

    def execute(self, q):
        dbg = pydbg()
        dbg.set_callback(EXCEPTION_ACCESS_VIOLATION, self.handle_av)
        dbg.set_callback(USER_CALLBACK_DEBUG_EVENT, self.timeout_callback)
        dbg.load(q['command'], command_line=q['args'])
        dbg.start_time = time()
        dbg.run()

    def timeout_callback(self, dbg):
        if time() - dbg.start_time > self.timeout:
            dbg.terminate_process()
            return DBG_CONTINUE

    def handle_av(self, dbg):
        crash_bin = utils.crash_binning.crash_binning()
        crash_bin.record_crash(dbg)
        self.obj['crash'] = True
        self.obj['output'] = crash_bin.crash_synopsis()

        dbg.terminate_process()
        return DBG_EXCEPTION_NOT_HANDLED
```

Executor.py

- My actual executor
- Continually check queue for new jobs
- When one is available, call execute
- Create a new pydbg instance, setup callbacks, execute

```
6  class Executor():
7      def __init__(self, timeout, queue_in, queue_out):
8          self.timeout   = timeout
9          self.queue_in  = queue_in
10         self.queue_out = queue_out
11         self.enterLoop()
12         self.obj = None
13
14     def enterLoop(self):
15         while True:
16             try:
17                 obj = self.queue_in.get_nowait()
18             except:
19                 sleep(.1)
20                 continue
21
22             if obj == 'STOP':
23                 break
24
25             self.obj = obj
26             self.execute(obj)
27
28             if not 'crash' in self.obj:
29                 self.obj['crash'] = False
30                 self.obj['output'] = None
31
32             self.queue_out.put(self.obj)
33
34     def execute(self, q):
35         dbg = pydbg()
36         dbg.set_callback(EXCEPTION_ACCESS_VIOLATION, self.handle_av)
37         dbg.set_callback(USER_CALLBACK_DEBUG_EVENT, self.timeout_callback)
38         dbg.load(q['command'], command_line=q['args'])
39         dbg.start_time = time()
40         dbg.run()
41
42     def timeout_callback(self, dbg):
43         if time() - dbg.start_time > self.timeout:
44             dbg.terminate_process()
45             return DBG_CONTINUE
46
47     def handle_av(self, dbg):
48         crash_bin = utils.crash_binning.crash_binning()
49         crash_bin.record_crash(dbg)
50         self.obj['crash'] = True
51         self.obj['output'] = crash_bin.crash_synopsis()
52
53         dbg.terminate_process()
54         return DBG_EXCEPTION_NOT_HANDLED
```

1. Establish timeout and queues

Executor.py

- My actual executor
- Continually check queue for new jobs
- When one is available, call execute
- Create a new pydbg instance, setup callbacks, execute

```
6  class Executor():
7      def __init__(self, timeout, queue_in, queue_out):
8          self.timeout   = timeout
9          self.queue_in  = queue_in
10         self.queue_out = queue_out
11         self.enterLoop()
12         self.obj = None
13
14     def enterLoop(self):
15         while True:
16             try:
17                 obj = self.queue_in.get_nowait()
18             except:
19                 sleep(.1)
20                 continue
21
22             if obj == 'STOP':
23                 break
24
25             self.obj = obj
26             self.execute(obj)
27
28             if not 'crash' in self.obj:
29                 self.obj['crash'] = False
30                 self.obj['output'] = None
31
32             self.queue_out.put(self.obj)
33
34     def execute(self, q):
35         dbg = pydbg()
36         dbg.set_callback(EXCEPTION_ACCESS_VIOLATION, self.handle_av)
37         dbg.set_callback(USER_CALLBACK_DEBUG_EVENT, self.timeout_callback)
38         dbg.load(q['command'], command_line=q['args'])
39         dbg.start_time = time()
40         dbg.run()
41
42     def timeout_callback(self, dbg):
43         if time() - dbg.start_time > self.timeout:
44             dbg.terminate_process()
45             return DBG_CONTINUE
46
47     def handle_av(self, dbg):
48         crash_bin = utils.crash_binning.crash_binning()
49         crash_bin.record_crash(dbg)
50         self.obj['crash'] = True
51         self.obj['output'] = crash_bin.crash_synopsis()
52
53         dbg.terminate_process()
54         return DBG_EXCEPTION_NOT_HANDLED
```

1. Establish timeout and queues

2. Wait for new job

Executor.py

- My actual executor
- Continually check queue for new jobs
- When one is available, call execute
- Create a new pydbg instance, setup callbacks, execute

Executor.py

```
6  class Executor():
7      def __init__(self, timeout, queue_in, queue_out):
8          self.timeout   = timeout
9          self.queue_in  = queue_in
10         self.queue_out = queue_out
11         self.enterLoop()
12         self.obj = None
13
14     def enterLoop(self):
15         while True:
16             try:
17                 obj = self.queue_in.get_nowait()
18             except:
19                 sleep(.1)
20                 continue
21
22             if obj == 'STOP':
23                 break
24
25             self.obj = obj
26             self.execute(obj)
27
28             if not 'crash' in self.obj:
29                 self.obj['crash'] = False
30                 self.obj['output'] = None
31
32             self.queue_out.put(self.obj)
33
34     def execute(self, q):
35         dbg = pydbg()
36         dbg.set_callback(EXCEPTION_ACCESS_VIOLATION, self.handle_av)
37         dbg.set_callback(USER_CALLBACK_DEBUG_EVENT, self.timeout_callback)
38         dbg.load(q['command'], command_line=q['args'])
39         dbg.start_time = time()
40         dbg.run()
41
42     def timeout_callback(self, dbg):
43         if time() - dbg.start_time > self.timeout:
44             dbg.terminate_process()
45             return DBG_CONTINUE
46
47     def handle_av(self, dbg):
48         crash_bin = utils.crash_binning.crash_binning()
49         crash_bin.record_crash(dbg)
50         self.obj['crash'] = True
51         self.obj['output'] = crash_bin.crash_synopsis()
52
53         dbg.terminate_process()
54         return DBG_EXCEPTION_NOT_HANDLED
```

1. Establish timeout and queues

2. Wait for new job

3. Execute job

- My actual executor
- Continually check queue for new jobs
- When one is available, call execute
- Create a new pydbg instance, setup callbacks, execute

Executor.py

```
6  class Executor():
7      def __init__(self, timeout, queue_in, queue_out):
8          self.timeout   = timeout
9          self.queue_in  = queue_in
10         self.queue_out = queue_out
11         self.enterLoop()
12         self.obj = None
13
14     def enterLoop(self):
15         while True:
16             try:
17                 obj = self.queue_in.get_nowait()
18             except:
19                 sleep(.1)
20                 continue
21
22             if obj == 'STOP':
23                 break
24
25             self.obj = obj
26             self.execute(obj)
27
28             if not 'crash' in self.obj:
29                 self.obj['crash'] = False
30                 self.obj['output'] = None
31
32             self.queue_out.put(self.obj)
33
34     def execute(self, q):
35         dbg = pydbg()
36         dbg.set_callback(EXCEPTION_ACCESS_VIOLATION, self.handle_av)
37         dbg.set_callback(USER_CALLBACK_DEBUG_EVENT, self.timeout_callback)
38         dbg.load(q['command'], command_line=q['args'])
39         dbg.start_time = time()
40         dbg.run()
41
42     def timeout_callback(self, dbg):
43         if time() - dbg.start_time > self.timeout:
44             dbg.terminate_process()
45             return DBG_CONTINUE
46
47     def handle_av(self, dbg):
48         crash_bin = utils.crash_binning.crash_binning()
49         crash_bin.record_crash(dbg)
50         self.obj['crash'] = True
51         self.obj['output'] = crash_bin.crash_synopsis()
52
53         dbg.terminate_process()
54         return DBG_EXCEPTION_NOT_HANDLED
```

1. Establish timeout and queues

2. Wait for new job

3. Execute job

4. Check timeout

- My actual executor
- Continually check queue for new jobs
- When one is available, call execute
- Create a new pydbg instance, setup callbacks, execute

Executor.py

```
6  class Executor():
7      def __init__(self, timeout, queue_in, queue_out):
8          self.timeout   = timeout
9          self.queue_in  = queue_in
10         self.queue_out = queue_out
11         self.enterLoop()
12         self.obj = None
13
14     def enterLoop(self):
15         while True:
16             try:
17                 obj = self.queue_in.get_nowait()
18             except:
19                 sleep(.1)
20                 continue
21
22             if obj == 'STOP':
23                 break
24
25             self.obj = obj
26             self.execute(obj)
27
28             if not 'crash' in self.obj:
29                 self.obj['crash'] = False
30                 self.obj['output'] = None
31
32             self.queue_out.put(self.obj)
33
34     def execute(self, q):
35         dbg = pydbg()
36         dbg.set_callback(EXCEPTION_ACCESS_VIOLATION, self.handle_av)
37         dbg.set_callback(USER_CALLBACK_DEBUG_EVENT, self.timeout_callback)
38         dbg.load(q['command'], command_line=q['args'])
39         dbg.start_time = time()
40         dbg.run()
41
42     def timeout_callback(self, dbg):
43         if time() - dbg.start_time > self.timeout:
44             dbg.terminate_process()
45             return DBG_CONTINUE
46
47     def handle_av(self, dbg):
48         crash_bin = utils.crash_binning.crash_binning()
49         crash_bin.record_crash(dbg)
50         self.obj['crash'] = True
51         self.obj['output'] = crash_bin.crash_synopsis()
52
53         dbg.terminate_process()
54         return DBG_EXCEPTION_NOT_HANDLED
```

1. Establish timeout and queues

2. Wait for new job

3. Execute job

4. Check timeout

5. Handle av

- My actual executor
- Continually check queue for new jobs
- When one is available, call execute
- Create a new pydbg instance, setup callbacks, execute

Executor.py

```
42      def timeout_callback(self, dbg):
43          if time() - dbg.start_time > self.timeout:
44              dbg.terminate_process()
45              return DBG_CONTINUE
46
47      def handle_av(self, dbg):
48          crash_bin = utils.crash_binning.crash_binning()
49          crash_bin.record_crash(dbg)
50          self.obj['crash'] = True
51          self.obj['output'] = crash_bin.crash_synopsis()
52
53          dbg.terminate_process()
54          return DBG_EXCEPTION_NOT_HANDLED
```

- handle_av we've seen, uses crash_binning to capture relevant data
- timeout_callback is a custom callback. Every itteration of the main debugging loop, it gets called. An easy way to implement a max timeout

```
13 class Fuzzer():
14     def __init__(self, max_processes, logfile, save_directory):
15         self.q_to = Queue()
16         self.q_from = Queue()
17         self.processes = []
18         self.max_processes = max_processes
19         self.save_directory = save_directory
20         self.mutator = None
21 
22         # open the logfile
23         try:
24             log = open(logfile, 'w')
25         except:
26             print '[!] Unable to open logfile', logfile
27             exit(1)
28         self.log = log
29 
30 
31     def start(self, command, original_file, timeout, temp_directory, mutation_type):
32         # create the consumers
33         for i in range(self.max_processes):
34             process = Process(target=Executor, args=(timeout, self.q_to, self.q_from))
35             self.processes.append(process)
36             process.start()
37 
38         # create the thread to get consumer output
39         monitor_thread = Thread(target=self.monitor)
40         monitor_thread.start()
41 
42         # create the mutator
43         mutator = Mutator(original_file, temp_directory, mutation_type)
44 
45         for counter, (offset, value_index, value_type, new_file) in enumerate(mutator.createNext()):
46             while not self.q_to.empty():
47                 sleep(.1)
48 
49             self.q_to.put({'command':command, 'args':'%s'%new_file, 'offset':offset,
50                            'value_index':value_index, 'value_type':value_type, 'new_file':new_file})
51 
52         self.stop()
53 
54     def stop(self):
55         # shutdown
56 
57     def monitor(self):
58         # check self.q_from for output and log it
```

Start the consumers

Start the monitor thread

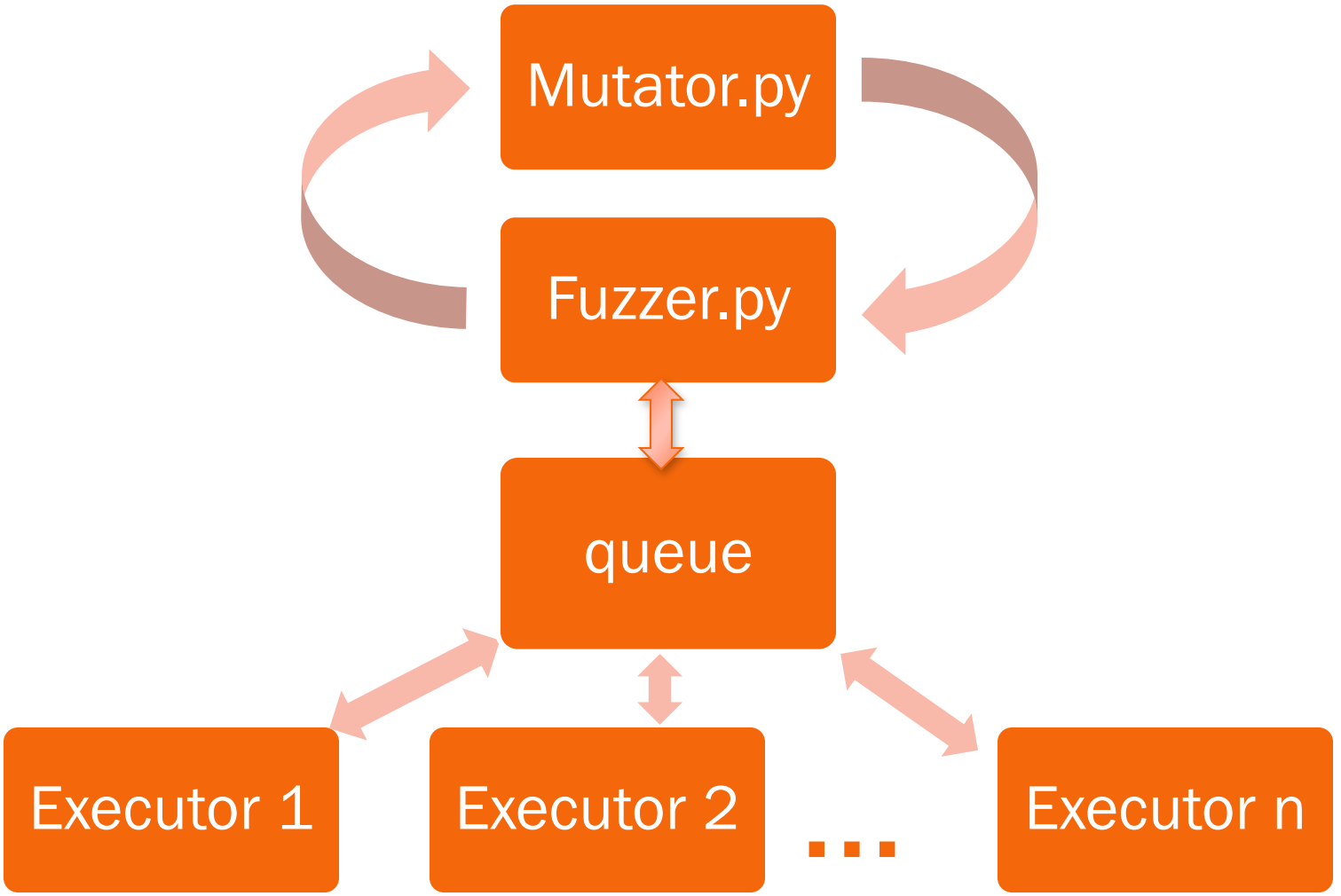
When the queue is empty, put a new job

Fuzzer.py

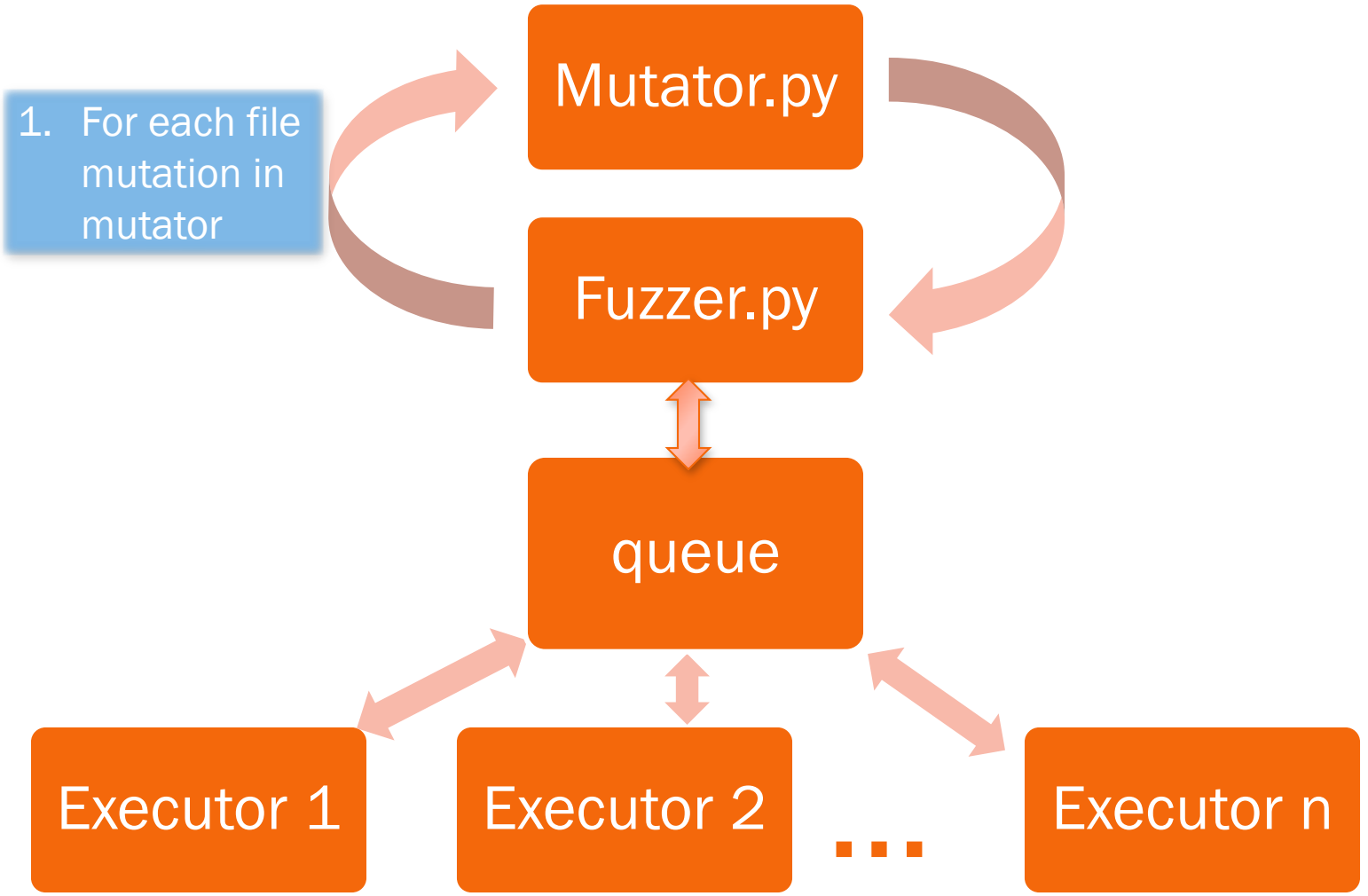
https://github.com/rmadair/fuzzer

- Feel free to grab my *work in progress* from the above link
- (I will update the site after the presentation)
- Producer / Consumer model
- Multiprocessing
- All in about 260 lines of python

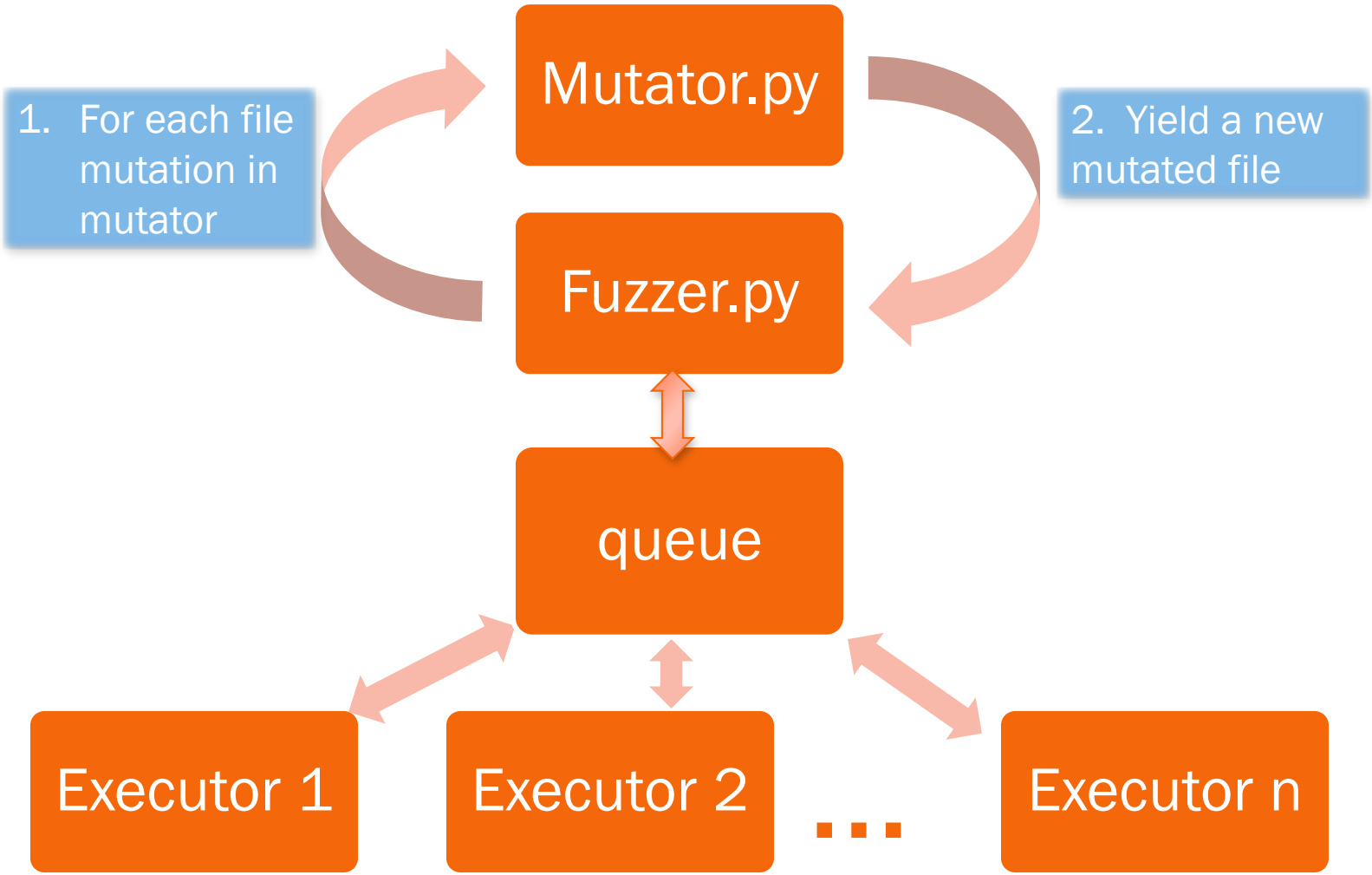
Architecture



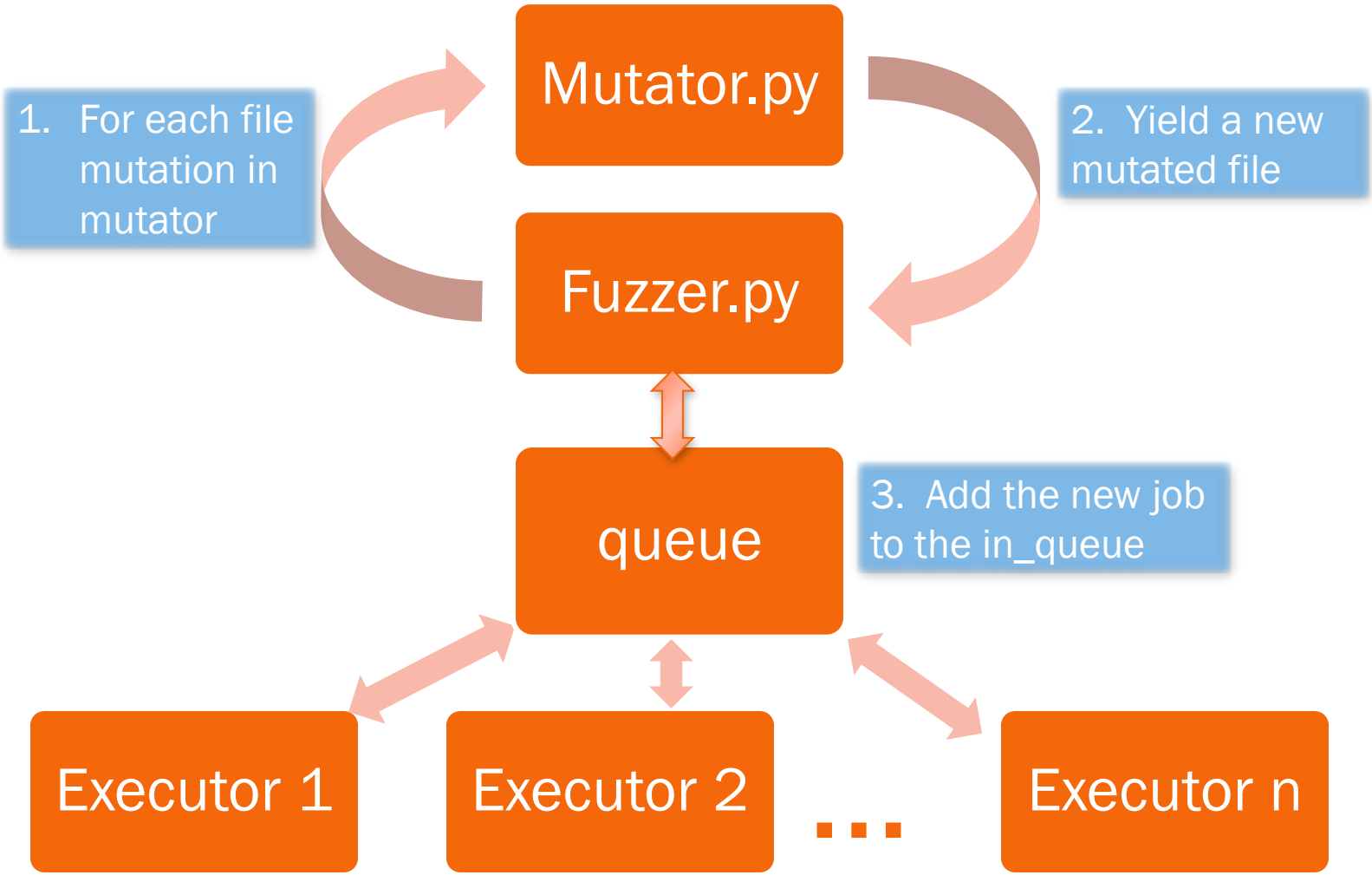
Architecture



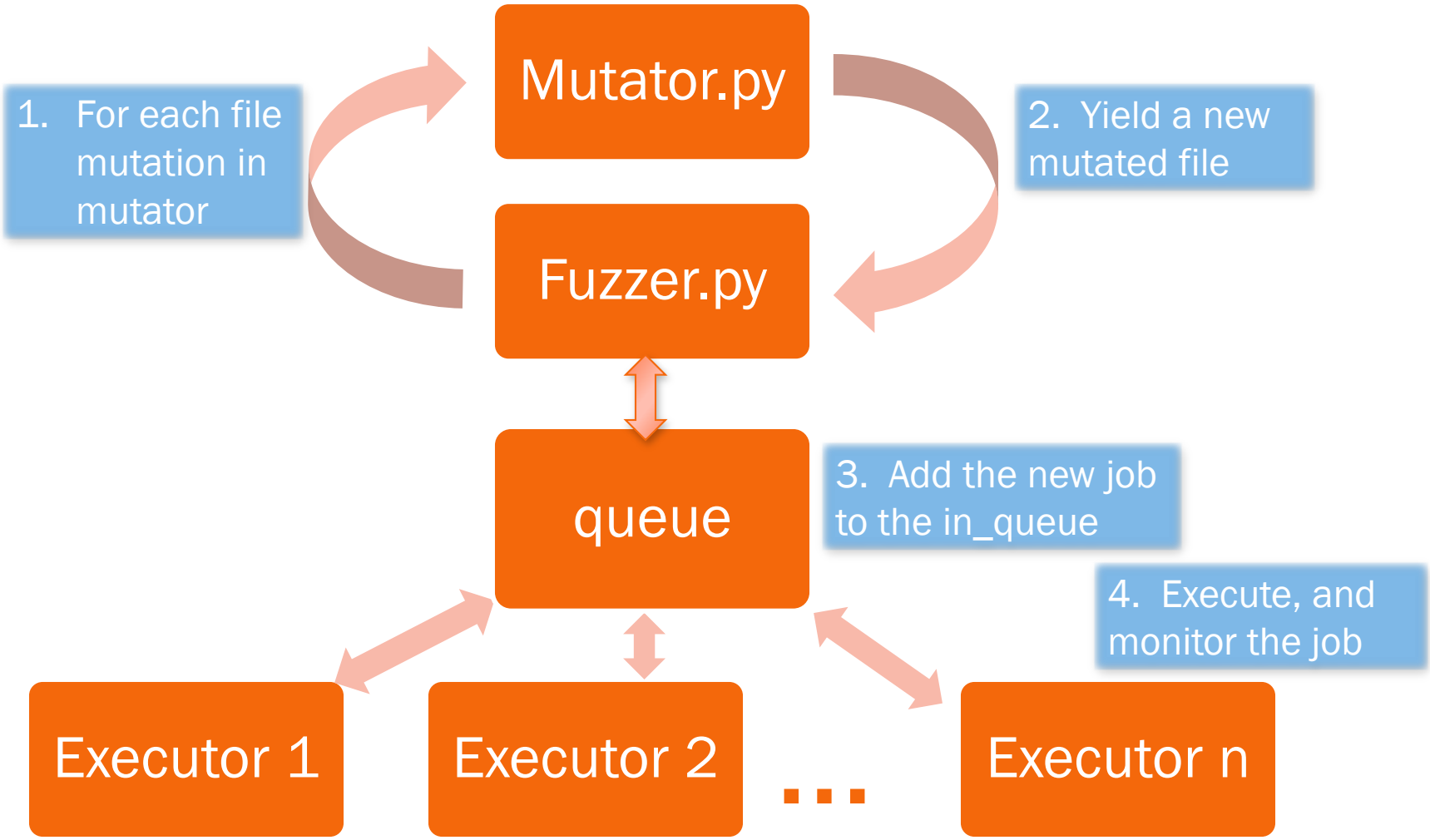
Architecture



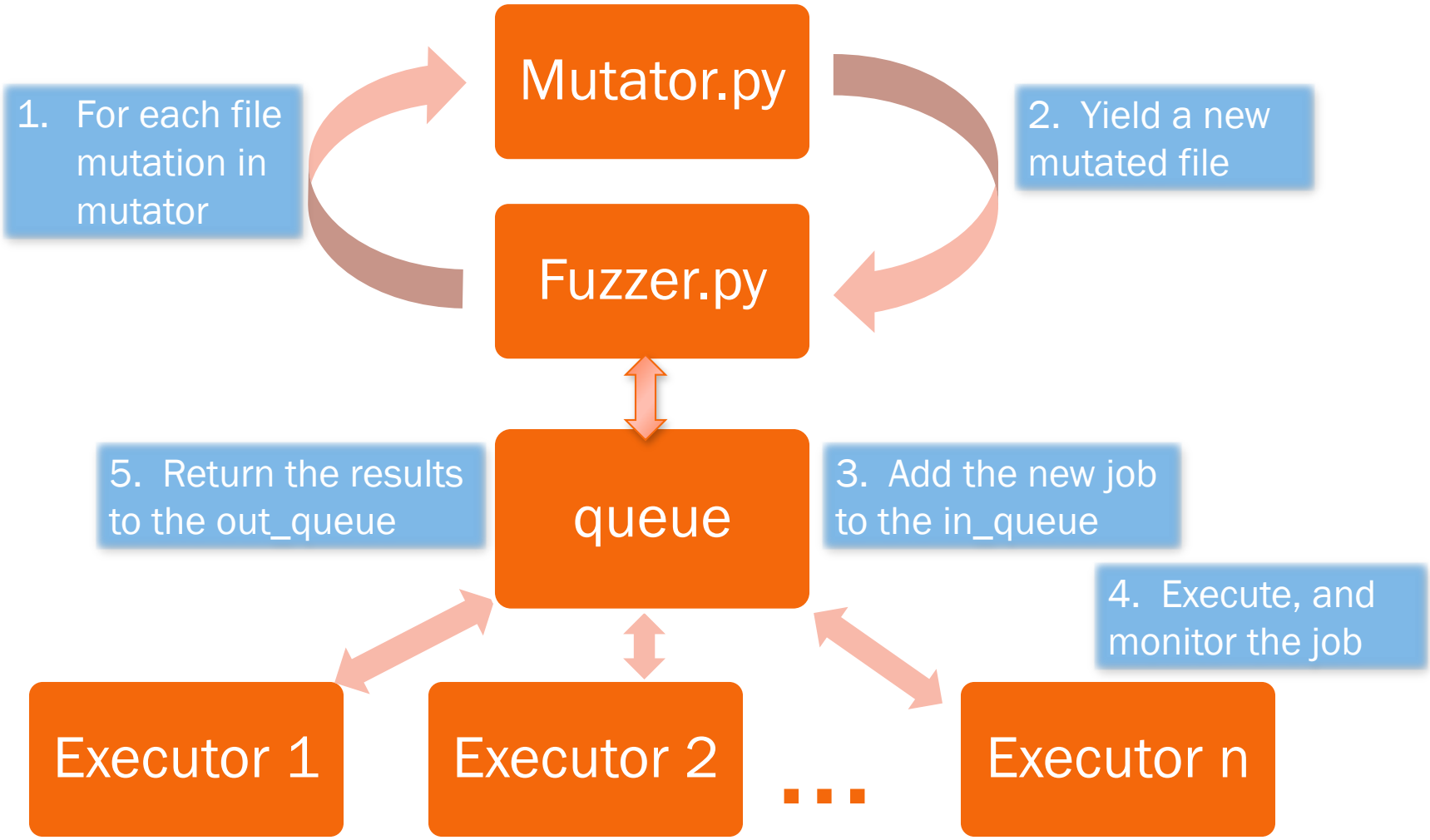
Architecture



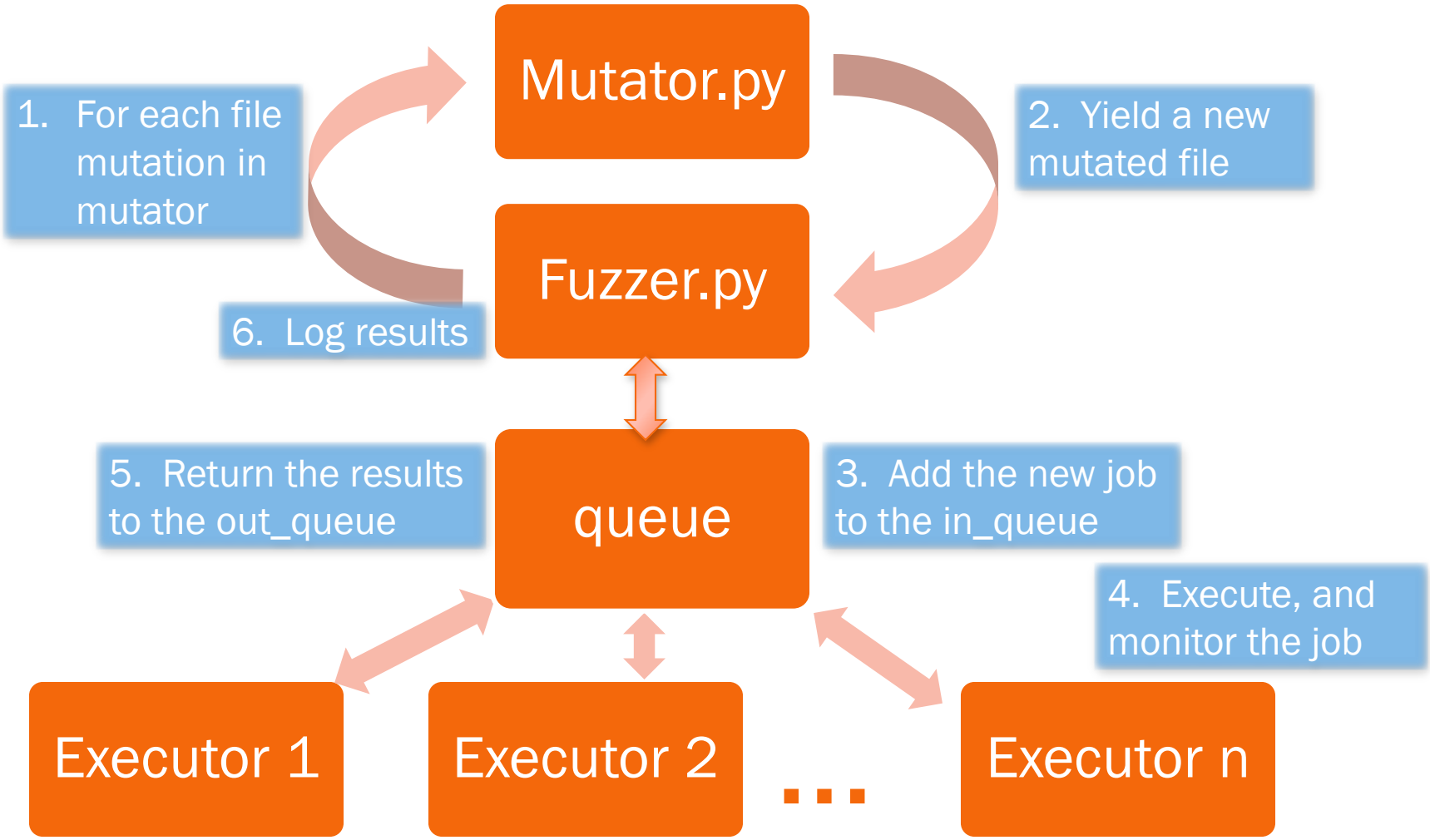
Architecture



Architecture



Architecture



Architecture

- There is actually an incoming queue and an outgoing queue as shown in the fuzzer.py slide, but it took me long enough to get that graphic, I'm not changing it ;)

Fuzzer++

- How can we improve our fuzzer, increase our odds?
- Code coverage would be a nice feature
  - PyDBG and WinAppDbg both support process “stalking”
  - Used to determine the first time a basic block or something specific is hit
    - Enumerate basic blocks ahead of time, count ones hit during execution
    - Find common pitfalls, track code coverage, etc.
- Cluster instead of consumer producer?
- Support specific file format fields?
  - Just use Peach ;)

Sample Files

- Where can I find some sample files?
  - Google.com, with the filter “filetype:xyz”
  - ie. “filetype:zip”
  - http://samples.mplayerhq.hu/
  - http://www.filecrop.com/
    - Be careful!

Resources

- Gray Hat Python: Python Programming for Hackers and Reverse Engineers
  - http://www.amazon.com/Gray-Hat-Python-Programming-Engineers/dp/1593271921
- Fuzzing: Brute Force Vulnerability Discovery
  - http://fuzzing.org/