Fuzzing — Part 2

Mitchell Adair
utdcsg.org
Outline

Debugging libraries (for Windows)
  o WinAppDbg, PyDBG
    • Examples
    • Pros and con

Fuzzer design
  o Design concepts
  o Fuzzer goals
  o Github
  o Future work
Debugging Libraries
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
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!
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
Let’s see some examples!
Create a debugging object
Load the target executable
Run it

Pretty painless
From the interpreter

```python
>>> 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:

```python
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:

 있지만

 Saúde

 Saúde
Let’s handle some signals. How about access violation?

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

- Wikipedia
PyDBG — Example 2

PS C:\Users\nomnom\utdcsg\fuzzing\part2> C:\Python26\python.exe ./pydbg-2.py
access violation!
access violation!

I_crank_alot.exe has stopped working
A problem caused the program to stop working correctly. Windows will close the program and notify you if a solution is available.

Debug Close program
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)
We can
  o Launch or attach to an application
  o Set our callback handlers
  o 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
Just create a crash_binning object and record the crash with the dbg object passed to the callback handler

```python
def record_crash (self, pydbg, extra=None):
    ...
    Given a PyDbg instantiation that at the current time is assumed to have "crashed" (access violation for example) record various details such as the disassembly around the violating address, the ID of the offending thread, the call stack and the SEH unwind. Store the recorded data in an internal dictionary, binning them by the exception address.
    ...
    @type  pydbg: pydbg
    @param pydbg: Instance of pydbg
    @type  extra: Mixed
    @param extra: (Optional, Def=None) Whatever extra data you want to store with this bin
    ...
```
That’s a pretty powerful 16 lines of code...
Sample output from crash_binning

Registers, assembly, stack trace, SEH

All with a function call, so easy!
Now import multiprocessing
Mutate some files
Launch the target application with the new files
Find bugs 😊
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/
Why not just stick with PyDBG?
  o Rumor has it PyDBG development has become OSX focused
  o It rocks, but it’s a little old and antiquated
  o 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
The WinAppDbg site has some great examples

- [http://winappdbg.sourceforge.net/ProgrammingGuide.html](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
Picking up where we left off with PyDBG

```python
from winappdbg import Debug, EventHandler

class MyEventHandler(EventHandler):
    def __init__(self):
        super(MyEventHandler, self).__init__()  # call our super class

    # these functions will be called if their signal occurs
    def access_violation(self, event):
        self.handleSignal(event)
    def illegal_instruction(self, event):
        self.handleSignal(event)

    def handleSignal(self, event):
        # gather data or handle the signal how we like
        # print registers, stack, etc.
        pass

    # initialize the handler, and the debugger to use it
handler = MyEventHandler()
debug = Debug(handler)
# launch the application, enter the debugging loop.
debug.execl(r'C:\Windows\system32\notepad.exe')
d debug.loop()
```

A custom event handler is optional, but is an easy way to catch any signals of interest.
from winappdbg.win32 import PVOID

# This function will be called when the hooked function is entered.
def wprintf():
    # 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 "wprintf( %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 wprintf.
            address = module.resolve( "wprintfW" )

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

            # Hook the wprintf function.
            event.debug.hook_function( pid, address, wprintf, signature = signature )
from winappdbg.win32 import PVOID

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

Hooks a function, wsprintfW

Catch the load_dll signal

If it's user32.dll, resolve wsprintf, hook it

Print the args
# 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
Hooking a function, \texttt{wsprintfW}

Catch the \texttt{load_dll} signal

If it's \texttt{user32.dll}, resolve \texttt{wsprintf}, hook it

Print the args
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 )
from winappdbg.win32 import PVOID

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( '%s', '\
').count( '%s' )
    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

1. Catch load_dll signal
2. If it’s user32.dll
3. Resolve “wsprintfW”
4. Hook it
5. wsprintf hit at run time

- Hooking a function, wsprintfW
- Catch the load_dll signal
- If it’s user32.dll, resolve wsprintf, hook it
- Print the args
# 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 )
```python
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 )\n% ( 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

1. Catch `load_dll` signal
2. If it’s `user32.dll`
3. Resolve “`wsprintfW`”
4. Hook it
5. `wsprintf` hit at run time
6. Dereference format string
7. Count args
Hooking a function, `wsprintfW`

Catch the `load_dll` signal

If it’s `user32.dll`, resolve `wsprintf`, hook it

Print the args

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

- Mutate Data
- Log Results
- Launch Application
- Monitor Application
Fuzzer Design - Modularity

- Mutate Data
- Launch Application
- Monitor Application
- Log Results

- Mutator.py
- Executor.py
- Fuzzer.py
Part 1 discussed possible values you may want to try

```python
class Mutator:
    """iterate over the contents of a given file, mutate the contents """

    def yieldNext(self):
        """yield a new file with mutated contents """

        for offset in range(self.file_contents):
            for value in self.mutation_values:
                # replace bytes

                newfile = open('name', 'wb')
                newfile.write(replaced_bytes)
                newfile.close()

                yield('name')
```

Yield is a nice python feature

Sole job is to mutate the bytes, any changes in possible values can easily be handled here
My actual executor

Continually check queue for new jobs

When one is available, call execute

Create a new pydbg instance, setup callbacks, execute
1. Establish timeout and queues

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

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

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

    def handle_av(self, 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
```

1. Establish timeout and queues
2. Wait for new job
3. Execute job
class Executor():
    def __init__(self, timeout, queue_in, queue_out):
        self.timeout = timeout
        self.queue_in = queue_in
        self.queue_out = queue_out
        self.executeLoop()
        self.obj = None

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

    def handle_av(self, 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

1. Establish timeout and queues
2. Wait for new job
3. Execute job
4. Check timeout
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
handle_av we’ve seen, uses crash_binning to capture relevant data

timeout_callback is a custom callback. Every iteration of the main debugging loop, it gets called. An easy way to implement a max timeout
class Fuzzer():
    def __init__(self, max_processes, logfile, save_directory):
        self.q_to = Queue()
        self.q_from = Queue()
        self.processes = []
        self.max_processes = max_processes
        self.save_directory = save_directory
        self.mutator = None
        
        # open the logfile
        try:
            log = open(logfile, 'w')
        except:
            print('[!] Unable to open logfile', logfile
        exit(1)
        self.log = log

    def start(self, command, original_file, timeout, temp_directory, mutation_type):
        # create the consumers
        for i in range(self.max_processes):
            process = Process(target=Executor, args=(timeout, self.q_to, self.q_from))
            self.processes.append(process)
            process.start()
        
        # create the thread to get consumer output
        monitor_thread = Thread(target=self.monitor)
        monitor_thread.start()

        # create the mutator
        mutator = Mutator(original_file, temp_directory, mutation_type)
        
        for counter, (offset, value_index, value_type, new_file) in enumerate(mutator.createNext()):
            while not self.q_to.empty():
                sleep(.1)

            self.q_to.put({'command':command, 'args':'%s%new_file', 'offset':offset,
                            'value_index':value_index, 'value_type':value_type, 'new_file':new_file})
            
        self.stop()

    def stop(self):
        # shutdown

    def monitor(self):
        # check self.q_from for output and log it
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
1. For each file mutation in mutator
1. For each file mutation in mutator
2. Yield a new mutated file

Architecture

Mutator.py
Fuzzer.py
queue
Executor 1
Executor 2
Executor n
1. For each file mutation in mutator

2. Yield a new mutated file

3. Add the new job to the in_queue

Executor 1

Executor 2

Executor n
1. For each file mutation in mutator

2. Yield a new mutated file

3. Add the new job to the in_queue

4. Execute, and monitor the job
1. For each file mutation in mutator
2. Yield a new mutated file
3. Add the new job to the in_queue
4. Execute, and monitor the job
5. Return the results to the out_queue
Fuzzer.py

Mutator.py

Queue

Fuzzer.py

1. For each file mutation in mutator

2. Yield a new mutated file

3. Add the new job to the in_queue

4. Execute, and monitor the job

5. Return the results to the out_queue

6. Log results

Executor 1

Executor 2

... 

Executor n
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 ;)}
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 ;)

Fuzzer++
Where can I find some sample files?

- Google.com, with the filter “filetype:xyz”
- ie. “filetype:zip”
- [http://samples.mplayerhq.hu/](http://samples.mplayerhq.hu/)
  - Be careful!
Gray Hat Python: Python Programming for Hackers and Reverse Engineers

Fuzzing: Brute Force Vulnerability Discovery
  - http://fuzzing.org/