revenge

Michael Bann

Dec 23, 2020

OVERVIEW

REVerse ENGineering Environment (`revenge`) was created as a python centric environment for many things reversing related. The idea is to create a cross-platform API to interact with binaries in different ways, simplify reverse engineering, and ultimately achieve a goal faster.

For the time being, `revenge` heavily relies on `frida`. On the plus side, `frida` is a nice cross platform DBI (which is why it was the building block). It also means that any bugs in `frida` will likely affect `revenge` as well.

If you have suggestions for what you would like to see `revenge` do, submit an issue ticket to my github.

CHAPTER

ONE

INSTALLATION

There are two primary ways to install and run `revenge`. You can use python directly, or you can utilize the `docker` image.

Note: Python 2 is NOT supported!

1.1 Python3

1.1.1 Virtual Environment

It's recomended to install `revenge` into a python virtual environment. If you haven't used this before, don't worry, it's easy.

First, install the python virtualenv package:

```
$ sudo apt update && sudo apt install -y virtualenv
```

Next, create a virtual environment for `revenge`:

```
$ virtualenv --python=$(which python3) /opt/revenge
```

Finally, you need to have it activated when you install or run `revenge`. Do this by sourcing the activate script. Note, this may vary depending on what shell you're using, but the base script should be fine for most.:

```
$ source /opt/revenge/bin/activate
```

1.1.2 Option 1 – pypi

The fastest way to get started is to simply pip install revenge.:

```
$ pip3 install revenge
```

1.1.3 Option 2 – git

You can install the very latest version of `revenge` directly from git:

```
$ pip3 install https://github.com/bannsec/revenge/archive/master.zip
```

1.2 Docker

You can use the auto-building docker image with the following:

```
$ sudo docker run -it --rm --privileged bannsec/revenge
```

CHAPTER

TWO

QUICK START

First off, head over to installation to get setup initially. Also, once you're done with quick start, please take a moment to read about the philosophy to get a better understanding of how to use the tool.

2.1 Just Show Me

```
from revenge import Process

# Load up /bin/ls, but don't let it continue
p = Process("/bin/ls", resume=False)

# Optionally, specify argv and envp
p = Process(["/bin/ls", ".."], envp={'var1':'val1'})

# Print out some basic info about the running process
print(p.threads)
print(p.modules)
print(p.memory)

# Resume execution
p.resume()

# Interact with the process
p.stdout(12) # Read 12 bytes of stdout
p.stdout("something") # Read until "something" is in output
p.stderr(12) # Read 12 bytes of stderr
p.stdin(b"hello!\n") # Write to stdin
p.interactive() # Quasi-interactive shell. ctrl-c to exit.
```

Check out the examples for each platform for more quick start ideas.

2.2 A Little Deeper

The two cent starting guide is that everything in `revenge` hangs off the core class called `Process`.

revenge.process.Process

This has traditionally been the starting point for opening applications, however in some cases (Android for the moment) it has become necessary to add a wrapper around Process to get the ball rolling. This is the Device class that is extended for various platforms. In the future, it will likely be the starting point for running an application.

CHAPTER

THREE

PHILOSOPHY

The philosophy for `revenge` is to make all common binary reverse engineering tasks pythonic. With this in mind, you will find that there are many classes. Things that might not even appear to be python classes, may be custom classes behind the scenes. This provides great flexibility in what you can do in consice commands.

Most classes in `revenge` will have a lot of custom overrides. If you're unsure of what to do with a class, try runing `__repr__` or `print` on it, as often times those will produce different and useful results.

It should be noted that, at least for the time being, this application is focused on dynamic reverse engineering. That means, all commands will return information about the process *as it is right now*. Addresses will change and values will change.

CHAPTER

FOUR

NATIVE INTERACTION

The concept of native interaction are simply those actions that are not specific to any platform. This generally entails assembly level, CPU, memory, threading, and any other things that are going to be common among any platform.

4.1 Debugging

While the driving concept behind `revenge` is dynamic binary instrumentation, you can still do some traditional debugging activities.

4.1.1 Breakpoints

Breakpoints in `revenge` are not the normal `int3` or even hardware breakpoints. Instead, `revenge` re-writes the address in question with a small loop that effectively stops it there, while not actually suspending the thread. This allows for setup to be completed or other activites to be run, and DBI to proceed part way through the binary.

Examples

```
# Set a breakpoint at main
process.memory['a.out:main'].breakpoint = True

# Continue execution from main, later
process.memory['a.out:main'].breakpoint = False

# Check if any given point in memory has a breakpoint
process.memory['a.out:main'].breakpoint
```

4.2 Errors

4.2.1 Native Errors

Native errors represent what you would get back from the operating system as an error number. This is usually called `errno`. `revenge` attempts to handle resolution of these errors into more helpful strings automagically with the class *`NativeError`*.

Examples: *`NativeError`*

4.3 Exceptions

4.3.1 Call Exceptions

When using the memory.__call__ method to call a function, the call will be wrapped in try/catch and will return objects of type revenge.native_exception.NativeException.

revenge.native_exception.NativeException

Examples

```
# Assuming that this threw an exception
exception = process.memory[':some_function']('blerg')

# Where did we except?
exception.address

# What type of exception?
exception.type

# Thread context at time of exception, containing registers and such
exception.context
```

4.4 Functions

4.4.1 Args and Return Types

In some cases, `revenge` will be able to identify (or guess) correctly the arugment types and return types for the function. However, in some cases you may need to tell it what to expect.

Examples

```
atof = process.memory[':atof']

# Tell revenge what the return type should be
atof.return_type = revenge.types.Double

# Not needed in this case, but you can tell revenge explicitly the
# parameter type
atof.argument_types = revenge.types.StringUTF8
```

4.4.2 Calling Functions

You can generically call native functions by first creating a memory object for them. Once you have that object, you usually can call it directly. This is due to some backend magic that attempts to identify argument and return types, and if it fails it falls back to integers.

However, sometimes that's not enough, and you need to tell `revenge` what types to send and/or expect back. Luckily, that's fairly strait forward.

Examples

```
import revenge
process = revenge.Process("/bin/ls", resume=False, verbose=False)

# Grab memory object for strlen
strlen = process.memory[':strlen']

# Call it directly on a string
strlen("test")
4

# You can specify the arg types if you need to
abs = process.memory[':abs']
abs(types.Int(-12))
12

# Sometimes you need to define what you're expecting to get in return
atof = process.memory[':atof']
atof.return_type = revenge.types.Double
atof('12.123')
12.123
```

4.4.3 Function Hooking

You can hook and replace functions. For native functions, you can either use an integer (which simply replaces the entire function and returns that integer instead), or a string that contains javascript that will be executed.

For the javascript replace, a special variable is created for you called `original`. You can assume this variable will always be there and will always be the original function you are replacing. This allows you to call down to the original function if needed, either replacing arguments, return types, or simply proxying the call.

To get data back from inside your replacement function, you need to define replace_on_message. That variable needs to be a callable that takes in at least one argument (the return from the script). Otherwise, all return sends will simply be ignored.

Sometimes it's easier to just attach to the entry or exit of the function rather than replacing it. You can do this via the `on_enter` method, in the same way as you would for `replace`. The only difference is that you do not have to worry about calling the function, as that will be done automatically after your code completes.

Examples

```
import revenge
process = revenge.Process("/bin/ls", resume=False, verbose=False)

# Replace function 'alarm' to do nothing and simply return 1
alarm = process.memory[':alarm']
alarm.replace = 1

# Un-replace alarm, reverting it to normal functionality
alarm.replace = None
```

More examples in the code.

revenge.memory.MemoryBytes.replace()

revenge.memory.MemoryBytes.on_enter()

4.4.4 Disassembly

You can disassemble in memory using `revenge` via the memory object.

Examples

```
import revenge
process = revenge.Process("a.out", resume=False, verbose=False)

print(process.memory['a.out:main'].instruction_block)
"""
0x804843a: lea         ecx, [esp + 4]
0x804843e: and         esp, 0xfffffff0
0x8048441: push        dword ptr [ecx - 4]
0x8048444: push        ebp
0x8048445: mov         ebp, esp
0x8048447: push        ebx
0x8048448: push        ecx
0x8048449: sub         esp, 0x10
0x804844c: call        0x8048360
"""

# Or just analyze one instruction at a time
process.memory['a.out:main'].instruction
"""<AssemblyInstruction 0x804843a lea ecx, [esp + 4]>"""
```

4.4.5 Building Functions With C

As of `frida` version 12.7, there is now support for injecting code simply as C. The backend of `frida` takes care of compiling it and injecting. `revenge` now exports this in a super easy to use way through the `create_c_function()` method.

`revenge` extends this also by making it easier to perform function calls anywhere in process space. It does this by creating a run-time function defition based on the current known address of the function. See example.

Examples

```
add = process.memory.create_c_function(r"""
    int eq(int x, int y) {
        return x==y;
    }""")

assert add(4,1) == 5

#
# Runtime function calling
#

# Suppose we want to call strlen, we need to export it as a callable
# function. Since we're compiling C code, the compiler has no idea
# where this function really is, and will throw an exception. However,
# revenge allows you to easily tell the compiler where it is and run as
# if you compiled with the application itself.

# Grab the strlen address
strlen = process.memory[':strlen']

# Setup strlen's argument and return types
strlen.argument_types = types.StringUTF8
strlen.return_type = types.Int

# Main difference is that we're adding a keyword arg to say
# "export/link in strlen here". So long as you've defined the
# MemoryBytes object, this can be anywhere, not just exported symbols.

my_strlen = process.memory.create_c_function(r"""
    int my_strlen(char *s) { return strlen(s); }
    """, strlen=strlen)

assert my_strlen("blerg") == 5
```

4.5 Handles

Handles are an abstraction for the native handles for your given operating system. For details, see examples in the python API doc.

revenge.plugins.handles.Handles

revenge.plugins.handles.Handle

4.6 Memory

4.6.1 Resolve Address

Since we're always dealing with running processes, we need a way to quickly identify locations of things of interest in memory. The primary way to do this is though a location format.

The location format is simply a string that takes the form `<module>:<offset or symbol>`. If no module is specified, the symbol will be resolved in the normal process manner (local->imports).

Resolving specific non-export symbols for libraries can be done with the Symbols class instead.

Examples

```
import revenge
process = revenge.Process("/bin/ls", resume=False, verbose=False)

# Resolve strlen from libc
strlen = process.memory['strlen']

# Resolve symbol test from the main binary
t = process.memory['bin:test']

# Grab memory object directly with address
thing = process.memory[0x12345]

# Write memory directly to address
process.memory[0x12345] = thing
```

4.6.2 Find

One common task is to find something in memory. `revenge` exposes this through the MemoryFind class.

revenge.memory.MemoryFind

Examples

```
import revenge

process = revenge.Process("/bin/ls", resume=False, verbose=False)

f = process.memory.find(types.StringUTF8('/bin/sh'))
"""<MemoryFind found 1 completed>"""

[hex(x) for x in f]
"""['0x7f9c1f3ede9a']"""
```

4.6.3 Read/Write

`revenge` has the ability to read and write to memory. It does this through the MemoryBytes class.

revenge.memory.MemoryBytes

Because of the inherent ambiguities of reading and writing to memory, you must specify the type of thing that you're reading or writing. Both reading and writing are done as a property to the class.

Examples

```
import revenge

# Start up /bin/ls
process = revenge.Process("/bin/ls", resume=False, verbose=False)

# Grab some memory location
mem = process.memory['ls:0x12345']

# Read UTF8 string from that location
mem.string_utf8

# Write UTF8 string to that location
mem.string_utf8 = "Hello world"

# Read signed 32-bit integer
mem.int32

# Write signed 32-bit integer
mem.int32 = -5

# Extract a range of bytes
mem = process.memory[0x12345:0x22222]
mem.bytes

# Write bytes into memory
mem.bytes = b'AB\x13\x37'

# You can write bytes generically if using types
process.memory['ls:0x12345'] = types.Int(12)

# You can use cast to read bytes using a type
assert mem.cast(types.Int32) == mem.int32
```

4.6.4 Memory Pages

We can investigate the memory layout programmatically or visually. We can also modify page permissions.

Examples

```
import revenge
process = revenge.Process("/bin/ls", resume=False, verbose=False)

# Print out memory layout like proc/<pid>/maps
print(process.memory)

"""
 564031418000-56403141d000          r-x  /bin/ls
 56403141d000-56403141e000          rwx  /bin/ls
 56403141e000-564031437000          r-x  /bin/ls
 564031636000-564031638000          r--  /bin/ls
 564031638000-564031639000          rw-  /bin/ls
 564031639000-56403163a000          rw-
 5640326bd000-5640326de000          rw-
 7f07f0000000-7f07f0021000          rw-
 7f07f8000000-7f07f8021000          rw-
 7f07fc272000-7f07fca72000          rw-
 7f07fca73000-7f07fd273000          rw-
 7f07fd274000-7f07fda74000          rw-
 7f07fda75000-7f07fe275000          rw-
 7f07fe275000-7f07fe412000          r-x  /lib/x86_64-linux-gnu/libm-2.27.so
 7f07fe611000-7f07fe612000          r--  /lib/x86_64-linux-gnu/libm-2.27.so
 7f07fe612000-7f07fe613000          rw-  /lib/x86_64-linux-gnu/libm-2.27.so
 7f07fe613000-7f07fe61a000          r-x  /lib/x86_64-linux-gnu/librt-2.27.so
 7f07fe819000-7f07fe81a000          r--  /lib/x86_64-linux-gnu/librt-2.27.so
 7f07fe81a000-7f07fe81b000          rw-  /lib/x86_64-linux-gnu/librt-2.27.so
 7f07fffd5000-7f0800000000          rw-
 7f0800000000-7f0800021000          rw-
 7f0804013000-7f080402a000          r-x  /lib/x86_64-linux-gnu/libresolv-2.27.so
 7f080422a000-7f080422b000          r--  /lib/x86_64-linux-gnu/libresolv-2.27.so
 7f080422b000-7f080422c000          rw-  /lib/x86_64-linux-gnu/libresolv-2.27.so
 7f080422c000-7f080422e000          rw-
 7f080422f000-7f0804a2f000          rw-
 7f0804a2f000-7f0804a49000          r-x  /lib/x86_64-linux-gnu/libpthread-2.27.so
 7f0804c48000-7f0804c49000          r--  /lib/x86_64-linux-gnu/libpthread-2.27.so
 7f0804c49000-7f0804c4a000          rw-  /lib/x86_64-linux-gnu/libpthread-2.27.so
 7f0804c4a000-7f0804c4e000          rw-
 7f0804c4e000-7f0804c51000          r-x  /lib/x86_64-linux-gnu/libdl-2.27.so
 <clipped>
"""

# Loop through the maps programmatically
for m in process.memory.maps:
    print(m)

# Make a page rwx
page = process.memory.maps[0x12345]
page.protection = 'rwx'
```

4.6.5 Allocate Memory

We can allocate and free memory with direct calls to the underlying operating system APIs, or through the memory wrapper.

Examples

```
import revenge
process = revenge.Process("/bin/ls", resume=False, verbose=False)

# Allocate a string in memory
mem = process.memory.alloc_string("Hello!")

# Use it like a pointer
# Free it once you're done
mem.free()

# Allocate some space generically
mem = process.memory.alloc(128)
```

4.7 Modules

For `revenge`, a module is any loaded library or shared library.

4.7.1 Listing Modules

```
# List current modules
print(process.modules)
"""
+--------------------+----------------+-----------+-----------------------------------
↪----------------------------+
|        name        |      base      |    size   | path
↪                            |
+--------------------+----------------+-----------+-----------------------------------
↪----------------------------+
|       test2        | 0x557781b84000 |  0x202000 | /home/user/tmp/test2
↪                            |
|  linux-vdso.so.1   | 0x7ffd3b5ee000 |   0x2000  | linux-vdso.so.1
↪                            |
|    libc-2.27.so    | 0x7fc6a8499000 |  0x3ed000 | /lib/x86_64-linux-gnu/libc-2.27.
↪so                           |
|     ld-2.27.so     | 0x7fc6a888a000 |  0x229000 | /lib/x86_64-linux-gnu/ld-2.27.so
↪                            |
| libpthread-2.27.so | 0x7fc6a827a000 |  0x21b000 | /lib/x86_64-linux-gnu/libpthread-
↪2.27.so                      |
| frida-agent-64.so  | 0x7fc6a6294000 | 0x17ba000 | /tmp/frida-
↪7846ef0864a82f3695599c271bf7b0f1/frida-agent-64.so |
| libresolv-2.27.so  | 0x7fc6a6079000 |  0x219000 | /lib/x86_64-linux-gnu/libresolv-2.
↪27.so                        |
|   libdl-2.27.so    | 0x7fc6a5e75000 |  0x204000 | /lib/x86_64-linux-gnu/libdl-2.27.
↪so                           |
|   librt-2.27.so    | 0x7fc6a5c6d000 |  0x208000 | /lib/x86_64-linux-gnu/librt-2.27.
↪so                           |
```

(continues on next page)

(continued from previous page)

```
|    libm-2.27.so    | 0x7fc6a58cf000 |  0x39e000 | /lib/x86_64-linux-gnu/libm-2.27.
↪so                              |
+--------------------+----------------+-----------+----------------------------------
↪----------------------------+
"""
```

4.7.2 Module Lookup

Instead of enumerating modules, you can look up a module by it's full name, a glob name, or by giving an address.

```
# Get the base address for specific module
hex(process.modules['test2'].base)
0x557781b84000

# Or by glob
process.modules['libc*']
"""<Module libc-2.27.so @ 0x7f282f7aa000>"""

# Or resolve address into corresponding module
process.modules[0x7f282f7ab123]
"""<Module libc-2.27.so @ 0x7f282f7aa000>"""
```

4.7.3 Symbols

Symbols for modules can be resolved and enumerated in a few ways.

Examples

```
# Grab symbol address for main function in my_bin
main = process.modules['a.out'].symbols['main']

# List all symbols from libc
print(process.modules['*libc*'].symbols)

# Grab the GOT entry for printf
process.modules['a.out']['got.printf']

# Grab the PLT entry for printf
printf_plt = process.modules['a.out']['plt.printf']

# Use symbol to get memory
printf_plt.memory

# Call symbol directly
printf_plt()
```

4.7.4 File Format Parsing

Some limited file format parsing is implemented.

Examples

```
# This elf object parses the elf as loaded in memory
elf = process.modules['ls'].elf
```

4.7.5 Loading Libraries

You can dynamically load libraries in `revenge` by using the *`load_library()`* method. This will attempt to load the library using native calls for your platform and return a *`Module`* object.

4.8 Symbols

Symbol support is provided under modules.

4.9 Threads

4.9.1 Enumerating

Enumerating threads is done by calling *process.threads* which is actually a Threads object.

`revenge.threads.Threads`

This threads object will look for the most current thread information every time you call methods on it, so if you're looking to be performant and don't need to refresh the threads, keep the return objects instead of re-enumerating.

Examples

```
threads = process.threads
print(threads)

"""
+--------+---------+----------------+--------------+-------+
|   id   |  state  |       pc       |    module    | Trace |
+--------+---------+----------------+--------------+-------+
| 120204 | waiting | nanosleep+0x40 | libc-2.27.so |   No  |
+--------+---------+----------------+--------------+-------+
"""

# Or you can go through the threads programmatically
for thread in threads:
    print(thread)

# If you know the thread id, you can index to it
thread = process.threads[81921]
```

4.9.2 Tracing

Please see tracing for more information.

4.9.3 Creating

You can easily create new threads using *create()*.

Examples

```
# Create a stupid callback that just spins
func = process.memory.create_c_function("void func() { while ( 1 ) { ; } }")

# Start the thread
t = process.threads.create(func.address)
assert isinstance(t, revenge.threads.thread.Thread)

# View it running
print(process.threads)
```

4.9.4 Return Values

Thread return values are handled by `revenge` in the same way the native operating system does. Specifically, you can call *join()* to get the return value.

Note, if you're attempting to return something outside a standard integer (such as a double or float), you will need to malloc space yourself and save off the value in there, then return the pointer to that space instead.

4.10 Tracing

A core reason for creating `revenge` was to make tracing applications easier. With that in mind, there will be a few different built-in tracers to run.

Note: You can only have one trace running per-thread at a time! This is a function of of DBI works, and not a limitation with revenge specifically.

Tracing is now considered a `Technique`. See *Techniques* for more information

4.10.1 Instruction Tracing

It's often interesting to simply trace an execution path. To do this with `revenge`, you can use the instruction tracing method to get a Tracer object and view your results. You can trace at different levels of granularity by specifying what you want to trace in the keyword arguments.

Examples

```
# Possible tracing options are: call, ret, block, exec, compile
# Default is False for all of them, so specify any combination
trace = process.techniques.NativeInstructionTracer(call=True, ret=True)

# Since trace is a technique, you must apply it
# By default, trace will apply to all threads if not given any arguments
trace.apply()

t = list(trace)[0]

print(t)
"""
call  ls:_init+0x211c                                          libc-2.27.so:__libc_
↪start_main                      0
call   libc-2.27.so:__libc_start_main+0x47                      libc-2.27.so:__cxa_
↪atexit                          1
call    libc-2.27.so:__cxa_atexit+0x54                          libc-2.27.so:on_
↪exit+0xe0                         2
ret      libc-2.27.so:on_exit+0x1a7                              libc-2.27.so:__cxa_
↪atexit+0x59                     3
ret     libc-2.27.so:__cxa_atexit+0xb4                          libc-2.27.so:__libc_
↪start_main+0x4c                 2
call   libc-2.27.so:__libc_start_main+0x76                      ls:_obstack_memory_
↪used+0xc30                      1
call    ls:_obstack_memory_used+0xc5c                           ls:_init
↪                                2
ret      ls:_init+0x16                                           ls:_obstack_memory_
↪used+0xc61                      3
call    ls:_obstack_memory_used+0xc79                           ls:_init+0x21f8
↪                                2
ret      ls:_init+0x21a9                                         ls:_obstack_memory_
↪used+0xc7d                      3
ret     ls:_obstack_memory_used+0xc94                           libc-2.27.so:__libc_
↪start_main+0x78                 2
call   libc-2.27.so:__libc_start_main+0x9a                      libc-2.27.so:_setjmp
↪                                1
ret     libc-2.27.so:__sigsetjmp+0x83                           libc-2.27.so:__libc_
↪start_main+0x9f                 2
call   libc-2.27.so:__libc_start_main+0xe5                      ls:_init+0x738
↪                                1
<clipped>
"""

# Loop through each instruction in the trace
for i in t:
    print(i)

# Remove the trace so you can run a different one
trace.remove()

# Take a slice of the trace
t2 = t[12:24]
```

4.10.2 Timeless Tracing

`revenge` has a technique for timeless tracing. For more information, see *NativeTimelessTracer*.

4.11 Types

`revenge` defines it's own types to better understand what data it is looking at. This means, while in many cases you can pass native python types to methods and fields, sometimes you will need to pass an instantiated type instead.

See *types doc*.

4.11.1 Examples

```
from revenge import types

# Create some ints
i = types.Int32(0)
i2 = types.UInt64(12)

# You can optionally read memory as a type instead of using memory attributes
assert process.memory[0x12345].cast(types.Int32) ==  process.memory[0x12345].int32
```

4.11.2 Structs

The *Struct* type is a little different from the rest of the types. Specifically, it defines a C structure, rather than a specific type. A struct can be defined by itself first, and then "bound" to a memory address.

The behavior of structs is to be used like dictionary objects.

Note: Compilers have NO standardization for struct padding. If your struct is not displaying correctly, check if you need to add type.Padding in between some elements.

Examples

```
# Create a struct
my_struct = types.Struct()
my_struct.add_member('member_1', types.Int)
my_struct.add_member('member_2', types.Pointer)

# Alternatively, add them IN ORDER via dict setter
my_struct = types.Struct()
my_struct['member_1'] = types.Int
my_struct['member_2'] = types.Pointer

# Use cast to bind your struct to a location
my_struct = process.memory[0x12345].cast(my_struct)

# Or set memory property directly
my_struct.memory = process.memory[0x12345]
```

(continues on next page)

(continued from previous page)

```
# Read out the values
my_struct['member_1']
my_struct['member_2']

# Write in some new values (this will auto-cast based on struct def)
my_struct['member_1'] = 12

# Print out some detail about it
print(my_struct)
"""
struct {
  test1 = -18;
  test2 = 3;
  test3 = 26;
  test4 = 4545;
  test5 = 3;
  test6 = 5454;
}
"""

# There's also a short-hand way to get space for your struct on the heap
struct = process.memory.alloc_struct(struct)

# It's bound to that address now, use it as above.
# Using this struct as an argument to a function call, you will likely
# want to wrap it as a pointer.
func = process.memory[<something>]
func(types.Pointer(my_struct))
```

4.11.3 Telescope

The *Telescope* class is a meta-ish class that holds other types. Specifically, it's goal is to address the question of how to describe and handle the concept of "telescoping" variables. With this in mind, often you do not create this directly, but will get it from certain tracer techniques.

Interaction with this class is effectively using the `thing` and `next` properties. Where `thing` is a holder for whatever the current thing is and `next` is the next one. Also, `type` will help inform you what to expect in the variable.

Example

```
# Telescope down into address 0x12345
scope = revenge.types(process, 0x12345)
scope.thing
scope.next
```

CHAPTER

FIVE

RELEASE NOTES

5.1 Version 0.20

- Introducing new *Angr* plugin that allows you to pick up an angr state at virtually any point in execution
- *Thread* now shows breakpoint register information when at a breakpoint instead of actual internal state
- You can now register a plugin to specifically be a *Thread* plugin the same way as modules
- Better Windows handling
  - Automatically breaks at process exit (like linux)
  - Unbuffers stdout (like linux)
  - Exposed *revenge.modules.Module.pe* to manually use PEFile
  - Implemented *entrypoint* finding
  - Properly handle radare2 not being installed
- Created *revenge.process.Process.resume()* to generically allow resuming of all paused threads
- Created new technique for *NativeInstructionCounter* to more easily allow counting instructions executed
- General updates and bug fixes

5.2 Version 0.19

- Added exception catching for the main thread. Any exceptions encountered will now be added to *exceptions*
- You can now expect output by supplying a string or bytes to *stdout()* or *stderr()*
- Added ability to *kill()* your thread more easily
- Modules can now have plugins registered with *_register_plugin()*
- The radare2 plugin is now a Module plugin
- Added initial DWARF decompiler
- All remote file loads will use a local cache, speeding up access times
- Backend updates to batch sending and timeless tracer
- Updated for frida api changing

5.3 Version 0.18

- Added ability to programmatically talk to *stdin()*, *stdout()*, and *stderr()*
- Added new plugin for enhancing reversing with *Radare2*
  - Ability to *highlight()* execution paths for view in *V* and *VV* modes
  - Integrated ghidra decompiler
- Added *Decompiler* plugin to allow for requesting decompiled code and doing thing such as highlighting paths
- Added plugin to support enumerating/reading and writing to *Handles*
- Added helper to discover what file an address belongs to as well as it's relative offset from the beginning of that file: *lookup_offset()*

5.4 Version 0.17

- Added support for *ARMContext* (Android on ARM emulator works now)
- Drastically improved performance for *NativeTimelessTracer*
- Updates to `contexts`
  - Tracking changed registers in *changed_registers*
  - Auto highlighting changed registers when printing cpu context
  - Consolidated and simplified handling of CPU contexts
- Lookups of the form "mod:sym:offset" work now
- New *LocalDevice* class
- Bunch of restructuring to eventually support multiple engines

5.5 Version 0.16

- Initial *NativeTimelessTracer* implementation is here! For more information, checkout *NativeTimelessTracer*
- Exposed frida's `on_enter()` to allow for more easily monitoring functions rather than replacing them
- Overhaul of *Telescope*
  - Implemented int/hex/bitand and rshift
  - Telescopes are now implemented via hash consing. This is drastically reduces the memory utilization when using the new NativeTimelessTracer.
  - Refactor of underlying js code for handling telescoping
- CPU Contexts now handle and print telescoping register values
- *NativeException* now telescopes the CPU reigsters when returning an exception
- Updated travis tests to enable testing on Android 10
- Updated coveralls to merge results

5.6 Version 0.15

- Implemented ability to call native function in it's own thread, instead of from frida's core thread
  - This will be done transparently, but can be done manually by calling `revenge.memory.MemoryBytes._call_as_thread()`
- Implemented *Techniques* to make common sets of actions more generic
- `InstructionTracer` is now *NativeInstructionTracer*
- *NativeInstructionTracer* now supports two new options
  - `include_function` allows you to specify a specific function to trace. This will cause revenge to ignore any trace before or after that function call.
  - `exclude_ranges` allows you to specify ranges of memory to be ignored from the trace
- Created *NativeError* class to generically handle `errno`.
- *Technique* mixin now also has optional method of `_technique_code_range()` that will get passed any known revenge/frida specific code ranges that can be ignored
- *Thread* changes
  - Implemented *join()* to allow for retrieving thread exit codes
  - Threads will now have *pthread_id* attribute if they were spawned on Linux.
  - Bugfix in *create()*
- Implemented `batch_send` js include to make it easier to handle pushing lots of data back

5.7 Version 0.14

- argv and envp options added to `Process` spawning
- Added *revenge.threads.Threads.create()* to simplify kicking off a thread
- Simplified symbol resolution, you can now use `process.memory['symbol']` directly as well as `process.memory['symbol+offset']`
- threads is now a submodule
- Can now create dummy thread for hidden Frida thread
- CPUContexts have been moved to `revenge.cpu.contexts`
- Tracer assembly has been moved to `revenge.cpu.assembly`

5.8 Version 0.13

- Implemented Frida's new `CModule` support as `create_c_function()`.
  - Also added support to make calling dynamic functions easier by passing them as kwargs to the constructor. See examples in code doc.
- Added `js_include` option to `run_script_generic()` to enable javascript library/code reuse type things
- Implemented `telescope.js` and *Telescope* for initial telescoping variable support

- `revenge.device_types` is now called *devices*.
- Added `quit()` to enable closing the process explicitly.
- Travis test cases are a bit more stable now.
- Implemented `_from_frida_find_json()` to allow for loading of MemoryRange objects directly from Frida json.

5.9 Version 0.12

- Added `__call__` to *Symbol* allowing for `symbol()` function call directly from the symbol class.
- Added Symbol *memory()* as a shortcut to get the MemoryBytes object for said symbol.
- Implemented new type for *Struct*. It's now much easier to both define, set, and read memory structures.
- Implemented *Memory* `__setitem__`, allowing for setting memory implicitly based on type. Example:

```
process.memory[0x12345] = types.Int16(5)
```

- Implemented MemoryBytes *cast()*, allowing for more programmatic retrieval based on type.
- Stability improvements

5.10 Version 0.11

- Updated `revenge.threads.Threads.__repr__()` to use descriptive addresses
- Added 0.5 second cache to *Modules* to improve performance.
- Many updates to `revenge.tracer.instruction_tracer.Trace.__str__()` to improve readabiliy (descriptive addrs, indentation, programmatic spacing)
- Implemented *plt()* to identify the base of the Procedure Lookup Table in ELF.
- Implemented and incorporated GOT and PLT symbols into *symbols()*. They will also now resolve on traces i.e.: symbol['got.printf'] or symbol['plt.printf']
- Symbols returned from *symbols()* are now actually an object: *Symbol*.
- Updated slice for `Trace` so that trace[:12], for instance, now returns a new Trace object with those instructions instead of just a list.
- entrypoint_rebased no longer exists. Now, just use `entrypoint()`
- Tests/docs updates

5.11 Version 0.10

- Added *revenge.memory.MemoryBytes.argument_types()* to allow a single or list/tuple of argument types for the function
- Added *revenge.memory.MemoryBytes.replace()* javascript string option. Now, you also have the option to set the replace to a javascript string that will replace the given function.
- Added `original` global variable for `MemoryBytes.replace` to allow you to more easily chain a call into the original native function.
- Aliased *revenge.memory.MemoryBytes.implementation()* to `MemoryBytes.replace` to standardize the naming convention with `JavaClass.implementation`.

CHAPTER

SIX

TECHNIQUES

In `revenge` parlance, a *`Technique`* is a high level set of actions that should be performed on the running binary. For instance, instead of manually deciding how you need to hide yourself from debugging checks, a technique could be applied that specifically attempts to do that. They're effectively high level batches of actions to be parformed with a single goal in mind.

6.1 Types of Techniques

There are two main types of techniques, due to how these techniques may be implemented:

6.1.1 Stalk Techniques

These techniques require the use of per-thread stalking. The important thing to note here is that you can only have one stalker running on a thread at a time. That means, you can only use one of these techniques at a time.

6.1.2 Replace Techniques

These techniques utilize binary re-writing prior to thread execution. The important thing to realize here is that these changes will affect all threads, since they are not thread specific. You can, however, have as many of these techniques running at a time as you like, since they do not take up a stalker context.

6.2 General Technique Usage

Specifics of technique usage may vary from technique to technique, the general usage remains the same. The steps are:

1. Start your process
2. Select the technique from `process.techniques.<technique>()`
3. *`apply()`* your technique

   a. If it is a stalking technique, you may want to provide the threads to the apply function
4. (optionally) *`remove()`* the technique at some point.

6.2.1 Techniques for specific calls

It's possible to apply a technique for specific calls. For instance, where you would use a native call to a function like `time(0)`, you can also provide a `techniques` argument with a single (or list) of techniques to apply to the specific call.

Details can be found under *`MemoryBytes`* documentation.

6.3 Implemented Techniques

For a list of techniques and more information, see *Techniques*.

6.4 How To Create a Technique

Creating a new technique is relatively strait forward:

1. Create a new submodule in `revenge/techniques`
2. Create your technique class by extending `revenge.techniques.Technique`
3. Implement `apply` and `remove` methods
4. Make sure `TYPE` is defined in your class
5. In the `__init__.py`, be sure that you expose the `Technique` you created. It can be any name, so long as the class instantiator is visible.
6. docs and tests

`revenge` will auto-discover the technique at runtime and expose it.

CHAPTER

SEVEN

ENGINES

Note: Engines concept is currently in development.

To support diversification and not be completely tied to one tool, `revenge` has introduced the concept of *`engines`*. The `engine` is basically the underlying driver that supports running `revenge`. Initially, this `engine` has been the impressive `frida` DBI. However, in some cases either `frida` doesn't yet support what we would like to do, or other technologies (such as an emulator), might be a better fit.

To select an engine, simply provide the `engine` keyword when instantiating your `Process` object. This will tell `revenge` to use the given engine.

CHAPTER

EIGHT

PLUGINS

8.1 angr

The angr plugin is being written to help expose features of angr to dynamic reversing.

8.1.1 Requirements

The current requirements to use the *angr* plugin are:

- Having the base angr installed
- Having *angr-targets* installed

8.1.2 Setup

As of writing, the version of *angr* in pypi is very out of date and will not work correctly for *revenge*. You will need to perform a dev install until *angr* pushes a new build.

```
git clone --depth=1 https://github.com/angr/angr-dev.git
cd angr-dev
./setup.sh -e angr -i
```

Note the *-e*. Choose whichever python virtual environment you have *revenge* installed in.

angr also has pre-built docker containers available which alleviate build issues.

8.1.3 Usage

Thread Plugin

As a thread plugin, *angr* gets exposed as a property of `Thread`. The primary use case of this is to allow seamless Symbion integration. When requesting objects, the plugin will automatically configure those objects to use *revenge* as a concrete backer as well as provide additional relocation support that isn't available directly by *Symbion*.

In English, this means you can execute to interesting points in your code using *revenge*, then easily get an *angr* state object that will pick up right at that point.

Basic Example

```
# Set process breakpoint somewhere interesting
process.memoery[interesting].breakpoint = True

# Once you hit that interesting point, grab your thread
thread = list(process.threads)[0]

# Now easily grab an angr state as if angr was already at this point in
# execution
state = thread.angr.state

assert state.pc == thread.pc

# Other helpful things
thread.angr.project
thread.angr.simgr
```

For more info, see the *Angr* API.

8.2 Decompiler

The `decompiler` plugin is an abstraction around the concept of decompiling code. While it registers as a single plugin, the actual decompiler backend is flexible and can be extended with new decompilers. When `revenge` starts up, a decompiler will be selected from those that `revenge` can identify that you have on your system.

8.2.1 General Usage

Here's a basic example. For more examples, see the code docs under *Decompiler*.

```
# Attempt to decompile an address
decomp = process.decompiler.decompile_address(0x1234)

# Attempt to decompile a function
decomp = process.decompiler.decompile_function(0x1234)
```

See notes for each decompiler engine about possible caveats.

8.2.2 Engines

- *DWARF* (priority 100)
- radare2_ghidra_decompiler (priority 70)

8.2.3 Building A Decompiler

To build a decompiler engine (building the decompiler is WAY beyond this little documentation), you must extend the *DecompilerBase* class. The calls to decompile MUST return an instance of *Decompiled*, which in turn must have 0 or more populated *DecompiledItem* instances.

On initialization of your decompiler, if it's valid for the current configuration, register it as an option with `process.decompiler._register_decompiler`.

The priority is mostly a way to select from multiple competing decompilers. The higher the number the higher priority.

8.3 DWARF

DWARF is a format for debugging info relating to ELF files. Standard compilations of binaries do not contain DWARF info. However, when you compile binaries with this info (generally with the *-g* flag), much more useful inforamtion is available. This plugin attempts to expose that information.

8.3.1 General Interaction

General interaction with the DWARF plugin is via the *modules*. For instance:

```
bin = process.modules['bin']
dwarf = bin.dwarf
```

8.3.2 Functions

Functions are enumerated and exposed via the *functions* property. You can utilize the *lookup_function()* method to resolve an address to it's function.

8.3.3 Source Lookup

The DWARF plugin can assist with looking up what the corresponding file and line number would be for a given address. As with all things in `revenge` this address is the current loaded address, rather than a base address. This lookup can be done via *lookup_file_line()*.

You can also ask DWARF to "decompile" an address for you. Note, this isn't actually decompiling, but the names are kept the same to avoid confusion. Instead of actually decompiling, the plugin will attempt to lookup the source address and line for your running address, and then lookup the corresponding source code for it. You must ensure you have told the plugin where your source directories are by using *add_source_path()*. Lookups for a source address can be done via *decompile_address()* and *decompile_function()*.

8.4 Radare2

The *radare2* plugin will attempt to utilize *radare2* to enrich local reversing information. It also exposes the ability to connect to a remote *radare2* instance and push enrichment data there.

8.4.1 Connecting

If `revenge` identifies that *radare2* is installed, the plugin will automatically load and start up a base instance of *radare2* for the given binary. By default, it will NOT perform auto analysis, since this can be expensive and time consuming.

Connecting to a remote instance can be done with the *`connect()`* method.

8.4.2 Highlighting

One thing that can be very helpful when analyzing code paths is to graphically *`highlight()`* them. This allows you to more easily see where a path travelled. Further, this becomes helpful when trying to identify where your test cases (or fuzzer) has covered in your code. While it can be done programmatically, this plugin exposes an easy way to view (in *radare2*) the paths covered.

Whereas other methods in this plugin can be used without a remote connection, highlighting likely makes the most sense when connected to a remote *radare2* session.

Example

```
# Startup r2 in a separate window
# r2 -A ./whatever
# In that window, start up the HTTP server
# =h& 12345

# Connect up to that session from your revenge session
process.radare2.connect("http://127.0.0.1:12345")

# Setup a timeless tracer
timeless = process.techniques.NativeTimelessTracer()
timeless.apply()
t = list(timeless)[0]

# Assuming you need to send some input to this program
process.memory[process.entrypoint].breakpoint = False
process.stdin("some input\n")

# Now that our trace is populated, send that data off to our r2 session
process.radare2.highlight(t)

# You can also use r2 for loaded libraries
libc = process.memory['*libc*']
libc.radare2

# In your other r2, you should now see highlights for this path in the
# Visual mode and the Very Visual mode
```

Plugins are a means for `revenge` to expose support in a general way. Plugins are dynamically loaded at runtime based on the current engine and compatability of the process for this plugin.

8.5 Building a Plugin

To build a plugin, you must extend the *Plugin* class. The general layout is:

- Create new submodule under `revenge.plugins`. This will be the core of the plugin and should have __no__ dependencies on any specific engine
- Create a submodule under `revenge.engines.<engine>`. This should extend the plugin class created above, and fill in any engine specific properties. NOTE: it's possible to create a plugin that is completely independent of any engine. In this case, the submodule here would simply extend the plugin class you created in step 1 and do nothing.
- Implement *_is_valid*. This property is called after instantiation to allow the plugin to determine if it wants to register in the current environment or not.

That's it. You should now have a working plugin.

8.6 Registering a Plugin

Your plugin will automatically register to the base process object if you return True for *_is_valid*. However, you can also dynamically register your plugin in a few different locations.

8.6.1 Registering a Module Plugin

Module plugins end up instantiated under Module.<plugin>. For instance:

```
# "plugin" here is where your plugin would end up
# It will get instantiated with the module that it is called from
process.modules['my_process'].plugin
```

If your plugin would likely be specific per module, you can register it as a module plugin. To do this, simply call *revenge.modules.Modules._register_plugin()* with your class instantiator as well as a name for the plugin. If successful, your plugin will now show up under the module object.

Example of how to do this can be found in the API docs.

8.6.2 Registering a Thread Plugin

Thread plugin registration works exactly the same way that module registration works. See *revenge.threads.Threads._register_plugin()*.

CHAPTER

NINE

WRITEUPS

Here's some write-ups done using `revenge`.

- DEFCON Quals 2019: VeryAndroidoso
- CSAW 2019: beleaf
- Patriot 2020: malloc

CHAPTER

TEN

NATIVE

10.1 CPU

10.1.1 CPUContextBase

This is the base mix-in class when defining new CPUs to support.

class revenge.cpu.contexts.**CPUContextBase**(*process*, *diff=None*, ***registers*)

Bases: object

property changed_registers

What registers were changed with this step?

Type list

pc

sp

10.1.2 CPUContext

The CPUContext represents the state of the CPU. The following is the base generator of contexts.

revenge.cpu.contexts.**CPUContext**(*process*, **args*, ***kwargs*)

x64

class revenge.cpu.contexts.x64.**X64Context**(*process*, *diff=None*, ***registers*)

Bases: *revenge.cpu.contexts.CPUContextBase*

REGS = ['rip', 'rsp', 'rbp', 'rax', 'rbx', 'rcx', 'rdx', 'rsi', 'rdi', 'r8', 'r9', 'r1

REGS_ALL = {'ah': '(self.rax>>8) & 0xff', 'al': 'self.rax & 0xff', 'ax': 'self.rax

r10

r11

r12

r13

r14

r15

r8

r9

rax

rbp

rbx

rcx

rdi

rdx

rip

rsi

rsp

x86

class revenge.cpu.contexts.x86.**X86Context**(*process*, *diff=None*, ***registers*)

Bases: *revenge.cpu.contexts.CPUContextBase*

REGS = ['eip', 'esp', 'ebp', 'eax', 'ebx', 'ecx', 'edx', 'esi', 'edi']

REGS_ALL = {'ah': '(self.eax>>8) & 0xff', 'al': 'self.eax & 0xff', 'ax': 'self.eax

eax

ebp

ebx

ecx

edi

edx

eip

esi

esp

arm

class revenge.cpu.contexts.arm.**ARMContext**(*process*, *diff=None*, ***registers*)

Bases: *revenge.cpu.contexts.CPUContextBase*

REGS = ['pc', 'sp', 'r0', 'r1', 'r2', 'r3', 'r4', 'r5', 'r6', 'r7', 'r8', 'r9', 'r10',

REGS_ALL = {}

lr

pc

r0

r1

r10

r11

r12

r2

r3

r4

r5

r6

r7

r8

r9

sp

10.2 Assembly

Abstraction for the assembly instructions.

10.2.1 Assembly Instruction

class revenge.cpu.**AssemblyInstruction**(*process*, *address=None*)

Bases: object

Represents an assembly instruction.

property address

Address where this instruction is located.

Type *Pointer*

property address_next

Address of instruction following this one.

Type *Pointer*

property args_str

Operation arguments as a string.

Type str

property args_str_resolved

Attempt to resolve addresses in the args str into symbols.

Type str

classmethod from_frida_dict(*process*, *d*)

Builds this assembly instruction from a frida dictionary, ala Instruction.parse()

property groups

List of descriptive groups that this instruction belongs to.

Type list

property mnemonic

Operation mnemonic.

Type str

property operands
List of operands.

Type list

property registers_read
List of registers that are read by this instruction.

Type list

property registers_written
List of registers written by this instruction.

Type list

property size
Size of this instruction in bytes.

Type int

10.2.2 Assembly Block

class revenge.cpu.assembly.instruction.**AssemblyBlock**(*process*, *address*)
Bases: object

Represents an assembly block.

10.3 Devices

10.3.1 AndroidDevice

revenge.devices.AndroidDevice.**applications**

revenge.devices.AndroidDevice.**arch**
Returns the arch of this android device.

revenge.devices.AndroidDevice.**frida_server_running**

revenge.devices.AndroidDevice.**platform**

revenge.devices.AndroidDevice.**processes**
Currently running processes

Type list

revenge.devices.AndroidDevice.**version**
Returns android version for this device.

Type str

10.3.2 LocalDevice

Connect to whatever this is locally running on.

param engine What engine to use? Defualt: frida

type engine str, optional

`revenge.devices.LocalDevice.`**`platform`**

`revenge.devices.LocalDevice.`**`processes`**

class `revenge.devices.`**BaseDevice**
Bases: `object`

Process(**args*, ***kwargs*)

property platform
What platform is this?

Type str

property processes
Currently running processes

Type list

resume(*pid*)
Resume a given process.

suspend(*pid*)
Suspend a given process.

10.4 Process

class `revenge.devices.process.process.`**Process**(*name*, *pid*, *ppid=None*)
Bases: `object`

Describes a process on this device.

Parameters

- **name** (*str*) – What is the name of this process
- **pid** (*int*) – Process ID
- **ppid** (*int, optional*) – Process Parent ID

property name
Process name.

Type str

property pid
Process ID

Type int

property ppid
Process Parent ID

Type int

10.5 Processes

class revenge.devices.process.processes.**Processes**(*processes=None*)
Bases: object

List of process objects.

Parameters **processes** (*list, optional*) – List of processes.

Examples

```
# List the process objects
list(procs)
```

10.6 Errors

class revenge.native_error.**NativeError**(*process*, *errno=None*)
Bases: object

Represents a error as defined by the native operating system.

Parameters

- **process** (*revenge.Process*) – Process object
- **errno** (*int, optional*) – The error number for this error

This object currently supports Linux type "errno" numbers.

Examples

```
# Normally you will be given the object, but you can
# instantiate it yourself as well
e = NativeError(process, 0)

print(e)
"Success"

assert e.description == "Success"
```

property description
String description of this error.

Type str

property errno
Error number for this error.

Type int

10.7 Exceptions

class revenge.native_exception.**NativeException**(*context*, *backtrace=None*, *type=None*, *memory_operation=None*, *memory_address=None*)

Bases: object

TYPES = ['abort', 'access-violation', 'illegal-instruction', 'arithmetic', 'breakpoint

property address
Address of this exception.

Type int

property memory_address
Address of memory exception.

Type int

property memory_operation
Type of memory operation performed at exception.

Enum: read, write, execute

Type str

property type
What type of native exception? One of abort, access-violation, illegal-instruction, arithmetic, breakpoint, system

Type str

10.8 Functions

10.8.1 Functions

class revenge.functions.**Functions**(*process*)

Bases: object

Represents functions.

Examples

```
# This is meant to be used like a dictionary

# Lookup MemoryBlock for function main
main = functions["main"]

# Lookup what function an address belongs to
assert functions[main.address] == b"main"

# Add function info
functions["func1"] = process.memory[<func1 range here>]

# Loop through function names
for name in functions:
```

(continues on next page)

(continued from previous page)

```
    pass

# Print out functions as table
print(functions)

# Check if a function name exists
assert "main" in functions

# Check if an address belongs to one of the known functions
assert 0x12345 in functions

# Not sure why you'd want to do this, but you can
functions[0x1000:0x2000] = "some_function"
```

lookup_address (*address*)

Lookup a function based on address.

Parameters **address** (*int, MemoryBytes*) – Address to lookup

Returns Name of function or None

Return type bytes

Examples

```
functions.lookup_address(0x12345) == b"some_function"
```

lookup_name (*name*)

Lookup MemoryBytes for a given name.

Parameters **name** (*str, bytes*) – Name of function

Returns Corresponding MemoryBytes object or None.

Return type *MemoryBytes*

Examples

```
main = functions.lookup_name("main")
```

set_function (*name*, *memory_bytes*)

Adds a function entry. Usually not done manually…

Parameters

- **name** (*str, bytes*) – Name of function
- **memory_bytes** (*MemoryBytes*) – MemoryBytes for function

10.9 Process

class revenge.process.**Process**(*target*, *resume=False*, *verbose=False*, *load_symbols=None*, *envp=None*, *engine=None*, *ignore_exceptions=False*)

Bases: object

Represents a process.

Args:

target (str, int, list): File name or pid to attach to. If target is a list, it will be set as argv.

resume (bool, optional): Resume the binary if need be after loading? verbose (bool, optional): Enable verbose logging load_symbols (list, optional): Only load symbols from those modules

in the list. Saves some startup time. Can use glob ('libc*')

envp (dict, optional): Specify what you want the environment pointer list to look like. Defaults to whatever the current envp is.

engine (revenge.engines.Engine): Instantiated Engine for this process

ignore_exceptions (bool): Should we not attempt to generically catch process exceptions? Default is False.

Examples:

```
# Kick off ls
p = revenge.Process("/bin/ls")

# Kick off ls for /tmp with custom environment
p = revenge.Process(["/bin/ls","/tmp/"], envp={'var1':'thing1'})

#
# Interaction
#

# Write to stdin
p.stdin(b"hello
```

")

Read from stdout p.stdout(16)

Read up to expected output in stdout p.stdout("expected")

Interact like a shell p.interactive()

property BatchContext

Returns a BatchContext class for this process.

Example

```
with process.BatchContext() as context:
    something(context=context)
```

Represents a context used to send many commands to a frida script.

Parameters

- **process** (*revenge.Process*) – Process this batch is running under.
- **send_buffer_size** (*int, optional*) – How big of a buffer to have before sending. (default: 1024)
- **return_buffer_size** (*int, optional*) – How big of a buffer to have before returning (default: 1024) If -1, do not return anything.
- **on_message** (*callable, optional*) – Callable to be called when we recieve information back. By default, returned information will be dropped.
- **run_script_generic** (*callable, optional*) – Which run_script_generic to use for calling? (default: process.run_script_generic)
- **handler_pre** (*str, optional*) – Something to optionally run before iterating over the strings provided.
- **handler_post** (*str, optional*) – Something to optionally run after iterating over the strings provided.

Example

```
with process.BatchContext():
    for i in range(255):
        do_something
```

This Context will simply queue up a bunch of strings, which will be fed into the thread and executed sequentially.

property alive
Is this process still alive?

Type bool

property arch
What architecture? (x64, ia32, arm, others?)

Type str

property argv
argv for this process instantitation.

Type list

property bits
How many bits is the CPU?

Type int

property device
What device is this process associated with?

Type *revenge.devices.BaseDevice*

property device_platform
Wrapper to discover the device's platform.

property endianness
Determine which endianness this binary is. (little, big)

property engine
The current engine revenge is using.

property entrypoint
Returns the entrypoint for this running program.

Type int

property file_name
The base file name.

Type str

property file_type
Guesses the file type.

interactive()
Go interactive. Return back to your shell with ctrl-c.

property pid

quit()
Call to quit your session without exiting. Do NOT continue to use this object after.

If you spawned the process, it will be killed. If you attached to the process, frida will be cleaned out, detatched, and the process should continue normally.

resume()
Resume execution of any current breakpoint hit or suspended thread.

stderr(*n*)
Read n bytes from stderr.

Parameters **n** (*int, str, bytes*) – Number of bytes to read or string to expect. If no value is given, it's presumed you are trying to read all currently queued output.

Returns Output of stderr

Return type bytes

stdin(*thing*)
Write thing to stdin.

Parameters **thing** (*str, bytes*) – If str, it will be encoded as latin-1.

Note: There's no newline auto appended. Remember to add one if you want it.

stdout(*n*)
Read n bytes from stdout.

Parameters **n** (*int, str, bytes*) – Number of bytes to read or string to expect. If no value is given, it's presumed you are trying to read all currently queued output.

Returns Output of stdout

Return type bytes

property target
Target for this session.

Type str, int

target_type(*x*)

property verbose
Output extra debugging information.

Type bool

10.10 Memory

10.10.1 Memory

class revenge.memory.**Memory**(*engine*)
Bases: object

Class to simplify getting and writing things to memory.

Examples

```
# Read a signed 8-bit int from address
memory[0x12345].int8

# Returns MemoryBytes object for memory
memory[0x12345:0x12666]

# Write int directly to memory
memory[0x12345] = types.Int8(12)

# Write string directly to memory
memory[0x12345] = types.StringUTF8("hello!")
```

alloc(*size*)
Allocate size bytes of memory and get a MemoryBytes object back to use it.

Parameters **size** (*int*) – How many bytes to allocate.

Returns Object for the new memory location.

Return type *revenge.memory.MemoryBytes*

alloc_string(*s*, *encoding='latin-1'*)
Short-hand to run alloc of appropriate size, then write in the string.

Parameters

- **s** (*bytes, str*) – String to allocate
- **encoding** (*str, optional*) – How to encode the string if passed in as type str.

alloc_struct(*struct*)
Short-hand to alloc appropriate space for the struct and write it in.

Parameters **struct** (revenge.types.Struct) – The struct to write into memory.

Returns The original struct, but now bound to the new memory location.

Return type *revenge.types.Struct*

describe_address(*address*, *color=False*)
Takes in address and attempts to return a better description of what's there.

Parameters

- **address** (*int*) – What address to describe
- **color** (*bool, optional*) – Should the description be colored? (default: False)

Returns description of the address

Return type str

find(**args*, ***kwargs*)
Search for thing in memory. Must be one of the defined types.

property maps
Return a list of memory ranges that are currently allocated.

10.10.2 MemoryBytes

class revenge.memory.**MemoryBytes**(*engine*, *address*, *address_stop=None*)
Bases: object

Abstracting what memory location is.

Parameters

- **engine** (revenge.engines.Engine) – The engine this is tied to.
- **address** (*int*) – Starting address of the memory location.
- **address_stop** (*int, optional*) – Optional stopping memory location.

Examples

```
# Trace specifically the function "win"
win = process.memory['a.out:win']
trace = process.techniques.NativeInstructionTracer(exec=True)

# This will populate the trace
win("input", techniques=trace)
print(trace)
```

property address
Address of this MemoryBytes.

Type *Pointer*

property address_stop
Stop address of this MemoryBytes.

Type *Pointer*

property argument_types
Returns the registered arguments types for this function or None if none have been found/registered.

Type tuple

property breakpoint
Does this address have an active breakpoint?

Type bool

property bytes
Return this as raw bytes.

Type bytes

cast(*cast_to*)
Returns this memory cast to whatever type you give it.

Examples

```
ptr = memory.cast(types.Pointer)

struct = types.Struct()
struct.add_member('my_int', types.Int)
struct.add_member('my_pointer', types.Pointer)
struct = memory.cast(struct)
```

property double
Read as double val

property float
Read as float val

free()
bool: Free this memory location. This is only valid if this memory location has been allocated by us.

property implementation

property instruction
Returns an assembly instruction parsed from what is in memory at this location.

Type *AssemblyInstruction*

property instruction_block
Returns an AssemblyBlock starting at this instruction.

Type *AssemblyBlock*

property int16
Signed 16-bit int

property int32
Signed 32-bit int

property int64
Signed 64-bit int

property int8
Signed 8-bit int

property name
Descriptive name for this address. Optional.

Type str

property pointer
Read as pointer val

property replace

property replace_on_message
Optional callable to be called if/when something inside the function replace sends data back.

Example

```
# If you just wanted to print out the messages that came back
def on_message(x,y):
    print(x,y)

strlen.replace_on_message = on_message
```

Type callable

property return_type
What's the return type for this? Only valid if this is a function.

property size
Size of this MemoryBytes. Only valid if it was generated as a slice, alloc or something else that has known size.

Type int

property string_ansi
Read as ANSI string

property string_utf16
Read as utf-16 string

property string_utf8
Read as utf-8 string

property struct
Write as a struct.

Example

```
struct = types.Struct()
struct.add_member('test1', types.Int32(-5))
struct.add_member('test2', types.Int8(-12))
struct.add_member('test3', types.UInt16(16))
process.memory[0x12345].struct = struct

# Or
process.memory[0x12345] = struct
```

property uint16
Unsigned 16-bit int

property uint32
Unsigned 32-bit int

property uint64
Unsigned 64-bit int

property uint8
Unsigned 8-bit int

10.10.3 MemoryRange

class revenge.memory.**MemoryRange**(*engine*, *base*, *size*, *protection*, *file=None*)
Bases: object

property base
Base address for this range.

Type int

property executable
Is this range executable?

Type bool

property file
File backing this memory range, or None.

Type str

property file_offset
Offset into backing file or None.

Type int

property protection
Protection for this range.

Type str

property readable
Is this range readable?

Type bool

set_protection(*read*, *write*, *execute*)
Sets the protection for this memory page.

Parameters

- **read** (*bool*) – Allow read?
- **write** (*bool*) – Allow write?
- **execute** (*bool*) – Allow execute?

This will call appropriate mprotect or similar. This can be done implicitly from the .protection property.

property size
Size for this range.

Type int

property writable
Is this range writable?

Type bool

10.10.4 MemoryFind

class revenge.memory.**MemoryFind**(*engine*, *thing*, *ranges=None*)
Bases: object

property completed
Is this search completed?

Type bool

property ranges

property search_string
The search string for this thing.

sleep_until_completed()
This call sleeps and only returns once the search is completed.

property thing
What we're looking for.

10.11 Modules

10.11.1 Modules

class revenge.modules.**Modules**(*process*)
Bases: object

_flush_cache()
Make sure the next time we're hit is a full one.

_register_plugin(*plugin*, *name*)
Registers this plugin to be exposed as a module plugin.

Parameters

- **plugin** (*callable*) – A class constructor. Must take an argument for the current module
- **name** (*str*) – What will this be called?

The plugin will be instantiated at most once per module instance, and done only when referenced.

Examples

```
class MyPlugin:
    @classmethod
    def _modules_plugin(klass, module):
        self = klass()
        self._module = module
        return self

process.modules._register_plugin(MyPlugin._modules_plugin, "myplugin")

# This first call will instantiate the plugin
process.modules['proc_name'].myplugin
```

load_library(*library*)
Dynamically load a library into the program.

Parameters **library** (*str*) – The full path to the library on the process machine

Returns ReturRns the new loaded module or None on error.

Return type *revenge.modules.Module*

Examples

```
selinux = process.modules.load_library("/lib/x86_64-linux-gnu/libselinux.so.1
↪")
```

This will eventually be implemented across all platforms. For now, it only works on linux platforms.

lookup_offset(*symbol*)
Lookup raw file offset to symbol.

Returns (module_name, offset) or None if cannot resolve

Return type tuple

See examples from modules.lookup_symbol

lookup_symbol(*symbol*)
Generically resolve a symbol.

Examples

resolve_symbol(":strlen") -> returns address of strlen resolved globally. resolve_symbol("strlen") -> equivalent to above resolve_symbol("strlen+0xf") -> strlen offset by 0xf resolve_symbol("a.out:main") -> returns address of main resolved to a.out. resolve_symbol(0x12345) -> returns symbol at that address.

property modules
Return list of modules.

Type list

10.11.2 Module

class revenge.modules.**Module**(*process*, *name*, *base*, *size*, *path*)
Bases: object

property base
Base address this module is loaded at.

Type int

property elf
Returns ELF object, if applicable, otherwise None.

property file
Opened file reader to a local copy of this module.

Type io.BufferReader

property name
Module name.

Type str

property path
Module path.

Type str

property pe
Returns PE object, if applicable, otherwise None.

property plt
Location of PLT for this module. Returns None if not known.

Type int

property size
Size of this module.

Type int

property symbols
symbol name -> address for this binary.

Type dict

10.12 Symbols

10.12.1 Symbol

class revenge.symbols.**Symbol**(*process*, *name=None*, *address=None*)
Bases: object

Represents a binary symbol.

Parameters

- **process** – Process object
- **name** (*str, optional*) – Name of this symbol
- **address** (*int, optional*) – Address of this symbol

property address
Address of this Symbol.

Type int

property memory
Convenience property to grab a memory object for this symbol.

Type *revenge.memory.MemoryBytes*

property name
Name of this symbol.

Type str

startswith(*x*)
Passthrough to check if the symbol name starts with some string.

Returns bool

10.13 Threads

10.13.1 Threads

Threads class object is what you get when you request `Process.threads`.

class revenge.threads.**Threads**(*process*)
Bases: object

_register_plugin(*plugin*, *name*)
Registers this plugin to be exposed as a thread plugin.

Parameters

- **plugin** (*callable*) – A class constructor. Must take an argument for the current thread
- **name** (*str*) – What will this be called?

The plugin will be instantiated at most once per thread instance, and done only when referenced.

Examples

```
class MyPlugin:
    @classmethod
    def _thread_plugin(klass, thread):
        self = klass()
        self._thread = module
        return self

process.threads._register_plugin(MyPlugin._thread_plugin, "myplugin")

# This first call will instantiate the plugin
process.threads[1234].myplugin
```

create(*callback*)
Create and start a new thread on the given callback.

Parameters **callback** – Pointer to function to start the thread on. This can be created via CModule, NativeCallback or use an existing function in the binary

Returns The new thread that was created or None if either the thread create failed or the thread finished before this method returned.

Return type *revenge.threads.Thread*

Example

```
# Create a stupid callback that just spins
func = process.memory.create_c_function("void func() { while ( 1 ) { ; } }")

# Start the thread
t = process.threads.create(func.address)
assert isinstance(t, revenge.threads.thread.Thread)

# View it running
print(process.threads)
```

(continues on next page)

(continued from previous page)

```
# Grab the return value (in this case the thread won't end though)
return_val = t.join()
```

property threads
Current snapshop of active threads.

10.13.2 Thread

The Thread class is an actual description of the thread itself.

class revenge.threads.**Thread**(*process*, *info*)
Bases: object

Defines a process thread.

Parameters **info** (*dict*) – frida thread info dict

Examples

```
# Grab your thread
thread = process.threads[tid]

# Wait for this thread to return
thread.join()

# Check out any exceptions that may have been thrown on this thread
thread.exceptions

# Check out the attached trace object
thread.trace
```

property breakpoint
Is this thread at a breakpoint?

Type bool

property context
The current context for this thread.

Type revenge.cpu.contexts.CPUContext

property exceptions
Exceptions that have been caught generically for this thread.

Type list

property id
Thread ID

Return type int

join()
Traditional thread join. Wait for thread to exit and return the thread's return value.

kill()
Attempts to kill this thread.

Note: If you're having trouble killing the thread, be sure your thread is killable.

For pthreads, that means: pthread_setcancelstate(0, 0); pthread_setcanceltype(1,0)

property module
What module is the thread's program counter in? i.e.: libc-2.27.so.

Return type `str`

property pc
The current program counter/instruction pointer.

Return type `int`

property state
Thread state, such as 'waiting', 'suspended'

Return type `str`

property trace
Returns Trace object if this thread is currently being traced, otherwise None.

Type revenge.tracer.instruction_tracer.Trace

10.14 Tracing

This is handled now as a technique. See *NativeInstructionTracer*

10.15 Native Types

class revenge.types.**Basic**
Bases: *revenge.types.BasicBasic*

property js
String that can be fed into js.

class revenge.types.**BasicBasic**
Bases: `object`

property memory
Instantiate this type to an active memory location for getting and setting.

Examples

```
struct = types.Struct()
struct.add_member('my_int', types.Int)
struct.add_member('my_pointer', types.Pointer)

struct.memory = 0x12345
# OR
struct.memory = 'a.out:symb'
# OR
struct.memory = process.memory[<whatever>]
```

class revenge.types.**Char**
Bases: *revenge.types.Int8*

type = 'char'

class revenge.types.**Double**(*x=0, /*)
Bases: *revenge.types.Float*

ctype = 'double'

sizeof = 8

type = 'double'

class revenge.types.**Float**(*x=0, /*)
Bases: *revenge.types.FloatBasic*, float

ctype = 'float'

property js

sizeof = 4

type = 'float'

class revenge.types.**FloatBasic**
Bases: object

class revenge.types.**Int**
Bases: *revenge.types.Int32*

type = 'int'

class revenge.types.**Int16**
Bases: *revenge.types.Basic*, int

ctype = 'short'

sizeof = 2

type = 'int16'

class revenge.types.**Int32**
Bases: *revenge.types.Basic*, int

ctype = 'int'

sizeof = 4

type = 'int32'

class revenge.types.**Int64**
Bases: *revenge.types.Basic*, int

ctype = 'long'

property js
String that can be fed into js.

sizeof = 8

type = 'int64'

class revenge.types.**Int8**
Bases: *revenge.types.Basic*, int

ctype = 'char'

sizeof = 1

type = 'int8'

class revenge.types.**Long**
Bases: *revenge.types.Int64*

type = 'long'

class revenge.types.**Padding**(*size*)
Bases: *revenge.types.BasicBasic*

Defines the spacing between struct entries.

Example

```
struct = types.Struct()
struct['one'] = types.Int8
struct['pad1'] = types.Padding(3)
struct['two'] = types.Int16
```

class revenge.types.**Pointer**
Bases: *revenge.types.UInt64*

ctype = 'void *'

property js
String that can be fed into js.

property sizeof
int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

type = 'pointer'

class revenge.types.**Short**
Bases: *revenge.types.Int16*

type = 'short'

class revenge.types.**StringUTF16**
Bases: *revenge.types.BasicBasic*, str

property js

property sizeof

type = 'utf16'

class revenge.types.**StringUTF8**
Bases: *revenge.types.BasicBasic*, str

ctype = 'char *'

property js

property sizeof

type = 'utf8'

class revenge.types.**Struct**

Bases: *revenge.types.Pointer*

Defines a C structure.

Examples

```
# Create a struct
my_struct = types.Struct()
my_struct.add_member('member_1', types.Int)
my_struct.add_member('pad1', types.Padding(1))
my_struct.add_member('member_2', types.Pointer)

# Alternatively, add them IN ORDER via dict setter
my_struct = types.Struct()
my_struct['member_1'] = types.Int
my_struct['member_2'] = types.Pointer

# Use cast to bind your struct to a location
my_struct = process.memory[0x12345].cast(my_struct)

# Or set memory property directly
my_struct.memory = process.memory[0x12345]

# Read out the values
my_struct['member_1']
my_struct['member_2']

# Write in some new values (this will auto-cast based on struct def)
my_struct['member_1'] = 12

# Allocate a struct and use it in a function call
my_struct = process.memory.alloc_struct(my_struct)
process.memory[<some function>](types.Pointer(my_struct))
```

add_member(*name*, *value=None*)

Adds given member to the end of this current structure.

Parameters

- **name** (*str*) – Name of the Struct member
- **value** (*revenge.types.all_types*) – Type and/or value for member.

Examples

```
s = revenge.types.Struct()
s.add_member('my_int', revenge.types.Int(12))

# Or, just the definition
s = revenge.types.Struct()
s.add_member('my_int', revenge.types.Int)
```

property members

property name

property sizeof
int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

class revenge.types.**Telescope**(*process*, *address=None*, *data=None*)
Bases: *revenge.types.BasicBasic*

property address
Address of this variable.

Type int

property description
String representational description of this telescope.

Type str

property memory_range
Information about the memory range this is in.

property next
Next step in the telescope.

property thing
Whatever this part of the telescope is.

class revenge.types.**UChar**
Bases: *revenge.types.UInt8*

type = 'uchar'

class revenge.types.**UInt**
Bases: *revenge.types.UInt32*

type = 'uint'

class revenge.types.**UInt16**
Bases: *revenge.types.Basic*, int

ctype = 'unsigned short'

sizeof = 2

type = 'uint16'

class revenge.types.**UInt32**
Bases: *revenge.types.Basic*, int

ctype = 'unsigned int'

sizeof = 4

type = 'uint32'

class revenge.types.**UInt64**
Bases: *revenge.types.Basic*, int

ctype = 'unsigned long'

property js
String that can be fed into js.

sizeof = 8

type = 'uint64'

class revenge.types.**UInt8**
Bases: *revenge.types.Basic*, int

ctype = 'unsigned char'

sizeof = 1

type = 'uint8'

class revenge.types.**ULong**
Bases: *revenge.types.UInt64*

type = 'ulong'

class revenge.types.**UShort**
Bases: *revenge.types.UInt16*

type = 'ushort'

revenge.types.**require_process**(*func*)

CHAPTER

ELEVEN

TECHNIQUES

class revenge.techniques.**Technique**(*process*)
Bases: object

This is a base mix-in class. To implement a technique, you need to extend this class.

TYPE = None

TYPES = ('stalk', 'replace')

apply(*threads=None*)
Applies this technique, optionally to the given threads.

remove()
Removes this technique.

property threads
Threads that are being traced by this object.

Type list

class revenge.techniques.**Techniques**(*process*)
Bases: object

append(*item*)

11.1 Stalk Techniques

11.1.1 NativeInstructionCounter

The NativeInstructionCounter's purpose is to allow you to more easily count the number of instructions executed. Moreover, it also allows you to count other things, such as the number of calls, returns, or blocks. This can be useful when performing execution length analysis to find new paths.

NativeInstructionCounter

class revenge.techniques.native_instruction_counter.**NativeInstructionCounter**(*process*, *from_modules=Non*, *call=False*, *ret=False*, *exec=False*, *block=False*, *compile=False*, *exclude_ranges=None*)

Bases: *revenge.techniques.Technique*

Counts instructions executed.

Parameters

- **process** – Base process instantiation
- **from_modules** (*list, optional*) – Restrict counting to those instructions that start from one of the listed modules.
- **call** (*bool, optional*) – Count calls
- **ret** (*bool, optional*) – Count rets
- **exec** (*bool, optional*) – Count all instructions
- **block** (*bool, optional*) – Count blocks
- **compile** (*bool, optional*) – Count Frida instruction compile
- **exclude_ranges** (*list, optional*) – [low, high] range pairs to exclude any trace items from.

Examples

```
# With no args, it will count individual assembly instructions
# executed
counter = process.techniques.NativeInstructionCounter()

# Need to apply it to use it
counter.apply()

# Resume the process to get execution going again
process.resume()

# Some point later, print out the count
print(counter)

### Can also be used as technique for specific call
strlen = process.memory["strlen"]
counter = process.techniques.NativeInstructionCounter()
strlen("hello", techniques=counter)
print(counter)
```

TYPE = 'stalk'

apply(*threads=None*)
Applies this technique, optionally to the given threads.

remove()
Removes this technique.

Counter

class revenge.techniques.native_instruction_counter.**Counter**(*thread*, *script*)
Bases: object

property count
Number of instructions executed.

Type int

stop()
Stop tracing.

11.1.2 NativeInstructionTracer

The NativeInstructionTracer's purpose is to centralize a means of simple stalking. You can specify what types of instructions to return back, and they will be returned in a python class object to help with analysis.

Instruction Tracer

class revenge.techniques.tracer.**NativeInstructionTracer**(*process*, *from_modules=None*, *call=False*, *ret=False*, *exec=False*, *block=False*, *compile=False*, *callback=None*, *exclude_ranges=None*, *include_function=None*)
Bases: *revenge.techniques.Technique*

Parameters

- **process** – Base process instantiation
- **from_modules** (*list, optional*) – Restrict trace returns to those that start from one of the listed modules.
- **call** (*bool, optional*) – Trace calls
- **ret** (*bool, optional*) – Trace rets
- **exec** (*bool, optional*) – Trace all instructions
- **block** (*bool, optional*) – Trace blocks
- **compile** (*bool, optional*) – Trace on Frida instruction compile
- **callback** (*callable, optional*) – Callable to call with list of new instructions as they come in. First arg will be the thread id.
- **exclude_ranges** (*list, optional*) – [low, high] range pairs to exclude any trace items from.

- **include_function** (*optional*) – resolvable function name or memorybytes object. starts tracing when function is entered and stops tracing when function is exited (call/ret)

Examples

```
#
# Trace all instructions in process except for those in a given range
# Apply this to the entire program execution
#

trace = process.techniques.NativeInstructionTracer(exec=True, exclude_
↪ranges=[[0x12345, 0x424242]])

# Apply this to the whole program and run
trace.apply()
process.memory[process.entrypoint].breakpoint = False

# Print out the trace
print(trace)

#
# Trace only blocks starting from a given function call downwards.
# Utilize this technique only on a specific call, rather than full program␣
↪execution
#

trace = process.techniques.NativeInstructionTracer(exec=True, include_function=
↪'my_func')
# or
my_func = process.memory['my_func']
trace = process.techniques.NativeInstructionTracer(exec=True, include_function=my_
↪func)

my_func(1,2,3, techniques=trace)

# Trace object should be populated now
print(trace)
```

TYPE = 'stalk'

apply(*threads=None*)
Applies this technique, optionally to the given threads.

remove()
Removes this technique.

Trace

class revenge.techniques.tracer.**Trace**(*process*, *tid*, *script*, *callback=None*)
Bases: object

append(*item*)

stop()
Stop tracing.

wait_for(*address*)
Don't return until the given address is hit in the trace.

Trace Item

class revenge.techniques.tracer.**TraceItem**(*process*, *item*)
Bases: object

property type

11.1.3 NativeTimelessTracer

The NativeTimelessTracer's purpose is to provide a standard means of performing timeless tracing. It is similar in concept to other timeless debuggers such as qira and rr.

Caveats

The major caveat for now is that it is going to be substantially slower than the other timeless debugger options. Performance will hopefully be improved in future releases.

Since this is a newer feature, it is currently only tested against:

- Linux (i386 and x64)

Why?

Timeless Debugging (or really, timeless tracing for revenge) is helpful for more thoroughly inspecting what happens during program execution. Instead of re-running an application and setting different break points each time, the tracer will attempt to gather all relevant information at each instruction step so that you can go forwards and backwards in time of the binary execution (thus, "timeless").

revenge's implementation has a goal of being platform and architecture independent. Meaning, the same syntax you would use to timeless trace on an amd64 Windows machine should work on an i386 MacOS or an ARM Linux.

Also, due to revenge's modularity, the timeless tracer will be used as a core component to other techniques and analysis engines, making it a building block, not an endpoint.

How do I use it?

The timeless tracer can be run just like any other *Technique*. Once the trace is acquired, you can manually look through it, or use an Analysis module (coming soon).

NativeTimelessTracer

class revenge.techniques.native_timeless_tracer.**NativeTimelessTracer**(*process*)
Bases: *revenge.techniques.Technique*

Performs timeless tracing.

Examples

```
#
# Global apply
#

# Setup the tracer
timeless = process.techniques.NativeTimelessTracer()
timeless.apply()

# Continue execution
process.memory[process.entrypoint].breakpoint = False

# Keep checking back to your trace
print(timeless)

# Grab your specific trace
t = list(timeless)[0]
print(t)
print(t[-50:])

# Look at the trace items individually
ti = t[0]
print(ti.context)
print(ti.context.rax)

#
# Call apply
#

# Also can apply this per-call
time = process.memory['time']
time(0, techniques=timeless)
```

TYPE = 'stalk'

apply(*threads=None*)
Applies this technique, optionally to the given threads.

remove()
Removes this technique.

NativeTimelessTrace

class revenge.techniques.native_timeless_tracer.**NativeTimelessTrace**(*process*, *thread*)

Bases: object

start()
Start tracing.

stop()
Stop tracing.

wait_for(*address*)
Don't return until the given address is hit in the trace.

NativeTimelessTraceItem

class revenge.techniques.native_timeless_tracer.**NativeTimelessTraceItem**(*process*, *context=None*, *depth=None*, *previous=None*)

Bases: `object`

property context

classmethod from_snapshot(*process*, *snapshot*, *previous=None*)

Creates a NativeTimelessTraceItem from a snapshot returned by timeless_snapshot()

Parameters

- **process** (*revenge.Process*) – Process object
- **snapshot** (*dict*) – Timeless snapshot dictionary
- **previous** (NativeTimelessTraceItem, *optional*) – Previous timeless trace item to use for differential generation

property instruction

Returns the assembly instruction object for this item.

11.1.4 About

Stalk techniques require stalking individual threads.

Pros:

- High granularity
- Can follow unexpected paths and behaviors

Cons:

- Can only have 1 running per thread
- Possibly more overhead than replace techniques

11.2 Replace Techniques

11.2.1 About

Replace techniques take advantage of rewriting parts of the binary prior to it being executed.

Pros:

- Can have as many of these running as needed (so long as they don't overlap)
- Generally more performant and reliable than stalking

Cons:

- Cannot as easily follow unexpected code paths
- Less granular in some cases

CHAPTER

TWELVE

ENGINES

class revenge.engines.**Engine**(*klass*, *device*, **args*, ***kwargs*)

Bases: object

Base for Revenge Engines.

property device

What device is this process associated with?

Type *revenge.devices.BaseDevice*

resume(*pid*)

Resume execution.

start_session()

This call is responsible for getting the engine up and running.

CHAPTER

THIRTEEN

PLUGINS

13.1 angr

class revenge.plugins.angr.**Angr**(*process*, *thread=None*)

Bases: *revenge.plugins.Plugin*

Use angr to enrich your reversing.

Examples

```
# Grab current location
thread = list(process.threads)[0]

# Load options and state options can be configured
# They use the same name but are exposed as attributes here
# If you SET any of these, the project will be re-loaded next
# time you ask for an object. It will NOT affect the current
# object instance you have.
thread.angr.load_options
thread.angr.support_selfmodifying_code
thread.angr.use_sim_procedures
thread.angr.add_options

# Ask for a simgr for this location
simgr = thread.angr.simgr

# Whoops, we wanted self modifying code!
thread.angr.support_selfmodifying_code = True
simgr = thread.angr.simgr

# Use this as you normally would
simgr.explore(find=winner)
```

property exclude_sim_procedures_list

Which procedures should angr not wrap?

Type bool

property load_options

angr load_options

property project

Returns the angr project for this file.

property simgr
Returns an angr simgr object for the current state.

property state
Returns a state object for the current thread state.

property support_selfmodifying_code
Should angr support self modifying code?

Type bool

property use_sim_procedures
Should angr use sim procedures?

Type bool

13.2 Decompiler

Note: The decompiler should be called as a plugin from process.decompiler.

13.2.1 DecompilerBase

class revenge.plugins.decompiler.**DecompilerBase**(*process*)
Bases: object

Use this to decompile things.

Examples

```
# Attempt to get corresponding source code from address 0x12345
process.decompiler[0x12345]

# Decompile a function
decomp = process.decompiler.decompile_function(0x12345)
# Or alternatively, specify it as a string to getitem
decomp = process.decompiler["my_func"]

# Programmatically iterate through it
for item in decomp:
    x = decomp[item]
    # stuff

# Or print it out to the screen
print(decomp)

# See decomp.highlight() as well.
```

decompile_address(*address*)
Lookup the corresponding decompiled code for a given address.

Parameters **address** (*int*) – The address to look up decompiled code.

Returns Decompiled output or None if no corresponding decompile was found.

Return type revenge.plugins.decompiler.decompiled.Decompiled

decompile_function(*address*)
Lookup the corresponding decompiled code for a given function.

Parameters **address** (*int*) – The start of the function to decompile.

Returns Decompiled output or None if no corresponding decompile was found.

Return type revenge.plugins.decompiler.decompiled.Decompiled

13.2.2 Decompiler

class revenge.plugins.decompiler.**Decompiler**(*process*)
Bases: *revenge.plugins.Plugin*

Use this to decompile things.

Examples

```
# Attempt to get corresponding source code from address 0x12345
process.decompiler[0x12345]

# Decompile a function
decomp = process.decompiler.decompile_function(0x12345)
# Or alternatively, specify it as a string to getitem
decomp = process.decompiler["my_func"]

# Programmatically iterate through it
for item in decomp:
    x = decomp[item]
    # stuff

# Or print it out to the screen
print(decomp)

# See decomp.highlight() as well.
```

decompile_address(*address*)
Lookup the corresponding decompiled code for a given address.

Parameters **address** (*int*) – The address to look up decompiled code.

Returns Decompiled output or None if no corresponding decompile was found.

Return type revenge.plugins.decompiler.decompiled.Decompiled

decompile_function(*address*)
Lookup the corresponding decompiled code for a given function.

Parameters **address** (*int*) – The start of the function to decompile.

Returns Decompiled output or None if no corresponding decompile was found.

Return type revenge.plugins.decompiler.decompiled.Decompiled

property imp
The underlying implementation.

This will be guessed automatically based on what decompilers are discovered. You can also instantiate your own and assign it directly to imp.

Type *revenge.plugins.decompiler.DecompilerBase*

13.2.3 Decompiled

class revenge.plugins.decompiler.**Decompiled**(*process*, *file_name=None*)
Bases: object

highlight(*thing*, *color=None*)
Highlight everything in thing with color.

Parameters

- **thing** (*int, list, tuple, trace*) – Addresses of things to highlight
- **color** (*str, optional*) – Color to use (see DecopmiledItem.highlight) default: green

Examples

```
# Create a timeless trace
timeless = process.techniques.NativeTimelessTracer()
timeless.apply()
t = list(timeless)[0]

# Decompile your function, this can be done at any time
decomp = process.decompiler.decompile_function(0x12345)

# Let your program run to grab the trace
process.memory[process.entrypoint].breakpoint = False

# Apply the trace to your decomp
decomp.highlight(t)

# You can keep the same decomp and apply traces from different timeless runs␣
↪as well
# For instance, if you had a second trace called t2, this would overlay that␣
↪trace
decomp.highlight(t2)
```

The things to highlight here must be valid in the current instance of revenge. This means, if your binary has ASLR, these must be the CURRENT addresses, with ASLR applied. Highlight will adjust the locations as needed.

13.2.4 DecompiledItem

class revenge.plugins.decompiler.**DecompiledItem**(*process*, *file_name=None*, *address=None*, *src=None*, *highlight=None*)
Bases: object

property address
Address of this decompiled instruction.

Type int

property highlight
Color to highlight this instruction (or None).

Valid options are: ['BLACK', 'BLUE', 'CYAN', 'GREEN', 'LIGHTBLACK_EX', 'LIGHTBLUE_EX', 'LIGHTCYAN_EX', 'LIGHTGREEN_EX', 'LIGHTMAGENTA_EX', 'LIGHTRED_EX', 'LIGHTWHITE_EX', 'LIGHTYELLOW_EX', 'MAGENTA', 'RED', 'WHITE', 'YELLOW']

Type str

property src
Pseudo source for this instruction.

Type str

13.3 Dwarf

13.3.1 Dwarf

class revenge.plugins.dwarf.**Dwarf**(*process*, *module=None*)
Bases: *revenge.plugins.Plugin*

Lookup Dwarf debugging information from the file.

Examples

```
dwarf = process.modules['*libc'].dwarf

# Show all known function names and their address and size
print(dwarf.functions)

# Print the first instruction block in main
print(dwarf.functions['main'].instruction_block)
```

add_source_path(*path*)
Adds the given path to the list of directories to look for source code in.

Parameters **path** (*str, bytes*) – Path to add to our search

property base_address
What is the binary's defined base address.

Type int

decompile_address(*address*)
Lookup the corresponding decompiled code for a given address.

Parameters **address** (*int*) – The address to look up decompiled code.

Returns Decompiled output or None if no corresponding decompile was found.

Return type revenge.plugins.decompiler.decompiled.Decompiled

decompile_function(*address*)

property decompiler
'Decompiler' using dwarf.

property functions
Dictionary of function_name -> MemoryBytes.

Type dict

property has_debug_info
Does this module actually have debugging info?

Type bool

lookup_file_line(*address*)
Given the address, try to resolve what the source file name and line are

Parameters **address** (*int*) – Address to lookup file line info

Returns (filename,line) or None, None if it wasn't found.

Return type tuple

Example

```
mybin = process.module['mybin']
filename, line = mybin.dwarf.lookup_file_line(mybin.dwarf.functions[b'main'].
↪address)
```

lookup_function(*address*)
Lookup corresponding function that contains this address.

Parameters **address** (*int*) – Address inside function

Returns The name of the function or None if lookup fails.

Return type bytes

13.4 Java

13.4.1 Java

class revenge.plugins.java.**Java**(*process*)
Bases: *revenge.plugins.Plugin*

property classes
Returns java classes object.

Type JavaClasses

find_active_instance(*klass*, *invalidate_cache=False*)
Look through memory and finds an active instance of the given klass.

Parameters

- **klass** (*str, JavaClass*) – The class we want to find already in memory.
- **invalidate_cache** (*bool, optional*) – Throw away any current cache. This should normally not be needed.

Returns Returns JavaClass instance with approrpiate handle server. This means you can use the object without instantiating it yourself.

Example

```
MainActivity = p.java.find_active_class("ooo.defcon2019.quals.veryandroidoso.
↪MainActivity")
MainActivity.parse("test")
```

13.4.2 JavaClass

class revenge.plugins.java.java_class.**JavaClass**
Bases: object

13.5 Handles

class revenge.plugins.handles.**Handles**(*process*)
Bases: *revenge.plugins.Plugin*

Manage process handles.

Examples

```
# Grab a specific handle
handle = process.handles[4]

# Print out details about handles
print(process.handles)
```

values()

class revenge.plugins.handles.**Handle**(*process*, *handle*, *name=None*)
Bases: object

Describes a handle.

Parameters

- **process** (*revenge.Process*) – Corresponding process.
- **handle** (*int*) – The handle identifier.
- **name** (*str, optional*) – File backing this handle.

Examples

```
handle = process.handles[4]

# What file/pipe/thing is this a handle to?
print(handle.name)

# Read 32 bytes from the beginning of the handle
stuff = handle.read(32, 0)

# Read 16 bytes from the current pointer
stuff = handle.read(16)
```

(continues on next page)

(continued from previous page)

```
# Write something to the handle
handle.write(b"something")

# Write something to the handle at offset 4
handle.write(b"something", 4)

# Check the read/write ability on this handle
handle.readable
handle.writable
```

property handle
The actual handle identifier. This is what the OS uses to identify the handle.

Type int

property name
Name or path to file backing this handle.

Type str

property position
Current position in this handle.

Type int

read(*n*, *position=None*)
Reads n bytes, optionally from a given position.

Parameters

- **n** (*int*) – How many bytes to read?
- **position** (*int, optional*) – Where to read from? Absolute.

Returns Data read from fd or None if there was an error

Return type bytes

When given position argument, this call will return the fd to it's original position after reading.

property readable
Is this handle readable?

Type bool

property writable
Is this handle writable?

Type bool

write(*thing*, *position=None*)
Writes thing into the handle, optionally from a given position.

Parameters

- **thing** (*str, bytes*) – What to write
- **position** (*int, optional*) – Where to write from? Absolute.

Returns Number of bytes written.

Return type int

13.6 Radare2

class revenge.plugins.radare2.**Radare2**(*process*, *module=None*)

Bases: *revenge.plugins.Plugin*

Use radare2 to enrich reversing information.

Examples

```
#
# Normal enrichment works without connection
# Radare2 plugin can enrich a remote instance of r2 with more
# information as well.

# In different window, open r2
# r2 -A /bin/ls
# Start up web server
# =h& 12345

# Connect up to it with revenge
process.radare2.connect("http://127.0.0.1:12345")

# Highlight paths that have executed
timeless = process.techniques.NativeTimelessTracer()
timeless.apply()

# Do whatever
t = list(timeless)[0]

process.radare2.highlight(t)
```

analyze()

Ask radare2 to run some auto analysis on this file.

Note: This is NOT run by default due to the fact that it may take a while to run. If you connect to a remote session that has already run analysis, you do NOT need to run this.

property base_address

connect(*web_server*)

Connect to a separate session to work in tandem.

Parameters **web_server** (*str*) – Web server to connect to.

Examples

```
# On existing r2 instance, start web listener on port 12345
# =h& 12345

# Now tell this r2 plugin to connect to it
process.radare2.connect("http://127.0.0.1:12345")
```

property decompiler

Either returns an instance of a decompiler (if one is valid) or None.

disconnect()
Disconnect from web server.

property file

highlight(*what*)
Highlights an instruction or list of instructions.

Parameters **what** (*int, list, tuple*) – Address to highlight.

Note: The addresses should be instantiated from this revenge process. Highlight will determine the correct offset to use for highlighting automatically.

This is likely only useful when you have connected to a remote r2 session as you won't see the color locally.

class revenge.plugins.**Plugin**
Bases: object

Base mix-in for plugins.

property _is_valid
Is the plugin valid for this configuration/should it be loaded?

CHAPTER

FOURTEEN

ANDROID

Anything that you can do with the native process class, you can do on Android. The only difference is that, for android, a `Process.java` class is also exposed to allow higher level interaction with java classes.

14.1 Java Classes

14.1.1 Class Enumeration

Java classes are at the core of Android applications. `revenge` exposes a way to enumerate the currently loaded classes.

`revenge` performs reflective inspection of the java classes. This means you will be able to use tab completion in ipython for methods and fields, as well as see a definition of the method or field in it's `__repr__`.

Note: The loaded classes might change during program execution as the program itself loads and instantiates new classes. The classes method is always a snapshot of the currently loaded classes list.

Examples

```
# List all currently loaded classes
process.java.classes

# Grab the Math class specifically
Math = process.java.classes['java.lang.Math']

# Or use globs
Math = process.java.classes['java.l*.Math']

# See what fields/methods exist
dir(Math)
```

14.1.2 Calling Methods

`revenge` makes directly calling methods from python easy.

Examples

```
# Grab the android logging class
log = process.java.classes['android.util.Log']

# Simply call the method with the required arguments
# Ending with () tells revenge to actually do the call
log.w("Hello", "world!")()
```

14.1.3 Method Override

You can easily override any method's definition. This uses Frida and thus, you will have to actually write your override in javascript.

Examples

```
# Grab the math class
Math = process.java.classes['java.lang.Math']

# Override the random implementation to be not-so-random
Math.random.implementation = "function () { return 12; }"

# Validate that it's our code
Math.random()()
12

# Remove override and check that original functionality is back
Math.random.implementation = None
Math.random()()
0.8056030012322106
```

14.1.4 Instantiated Classes

If an application is saving state in the java class, you may want to interact specifically with the class instance, rather than just a generic class. You can do this by finding the instance.

Examples

```
# Grab the class
MainActivity = process.java.classes['*myapp*MainActivity']

# Find the active instance
M = process.java.find_active_instance(MainActivity)

# Call the method on that specific running instance
M.some_method()()
```

14.1.5 Batch Calling

Batch calling is the same concept as batch calling for the native process. The idea is, since the time it takes to send commands from python into the application and back can be rather slow, we open up a context where we can feed in a bunch of commands at once. Instead of getting the results back one by one per call, we get them back in bulk to a message handler that has the resonsibility to deal with it.

To use batch contexts, you will need to instantiate them inside a `with` context. Then provide the context to the calling method so it knows to use that context.

For CTFs, this is generally used on challenges that require some level of brute foricing of the flag.

Examples

Coming soon..

14.2 Setup

Initially setting up `revenge` to work with an android emulator involves using the devices. For this doc, I'll assume that you already have an android running, in either emulator or physical form.

Note: You must have root access to the device on which you wish to run `revenge`.

14.2.1 Base Connection

Base interactions for `revenge` will go through the device object. Instantiating this object will attempt to automatically install, run and connect to the latest version of frida server for your android.

Examples

```
from revenge import devices

# Connect to the first usb device adb finds
android = devices.AndroidDevice(type="usb")
"<AndroidDevice emulator-5554>"

# Connect to device with the given id
android = devices.AndroidDevice(id="emulator-5554")
"<AndroidDevice emulator-5554>"
```

14.2.2 Installing/Removing APKs

A convenience method exists to install and uninstall apks directly from `revenge`.

Examples

```
android.install("something.apk")
android.uninstall("com.blerg.something")
android.uninstall(android.applications['*something*'])
```

14.2.3 Shell

You can drop into an interactive shell.

Examples

```
android.shell()
```

14.2.4 List Processes/Applications

You can list both running processes and running applications. Applications have their own class.

`revenge.devices.android.applications.AndroidApplications`

Examples

```
android.device.enumerate_processes()
"""
<clip>
 Process(pid=1502, name="tombstoned"),
 Process(pid=1503, name="android.hardware.biometrics.fingerprint@2.1-service"),
 Process(pid=1506, name="iptables-restore"),
 Process(pid=1507, name="ip6tables-restore"),
 Process(pid=1604, name="dhcpclient"),
 Process(pid=1607, name="sh"),
 Process(pid=1608, name="sleep"),
 Process(pid=1619, name="ipv6proxy"),
 Process(pid=1622, name="hostapd"),
 Process(pid=1624, name="dhcpserver"),
 Process(pid=1633, name="system_server"),
 Process(pid=1740, name="com.android.inputmethod.latin"),
 Process(pid=1748, name="com.android.systemui"),
 Process(pid=1790, name="webview_zygote32"),
 Process(pid=1846, name="wpa_supplicant"),
 Process(pid=1851, name="com.android.phone"),
<clip>
"""

# List applications
list(android.applications)
```

(continues on next page)

(continued from previous page)

```
"""
<clip>
 Application(identifier="com.android.dialer", name="Phone", pid=2084),
 Application(identifier="com.android.gallery3d", name="Gallery"),
 Application(identifier="com.android.emulator.smoketests", name="Emulator Smoke Tests
↪"),
 Application(identifier="android.ext.services", name="Android Services Library",␣
↪pid=2566),
 Application(identifier="com.android.packageinstaller", name="Package installer"),
 Application(identifier="com.svox.pico", name="Pico TTS"),
 Application(identifier="com.android.proxyhandler", name="ProxyHandler"),
 Application(identifier="com.android.inputmethod.latin", name="Android Keyboard (AOSP)
↪", pid=1740),
 Application(identifier="org.chromium.webview_shell", name="WebView Shell"),
 Application(identifier="com.android.managedprovisioning", name="Work profile setup"),
<clip>
"""
```

14.2.5 Running Applications

You can spawn and attach to applications via command-line.

Examples

```
# Launch application and retrieve corresponding revenge.Process instance
p = android.spawn("com.android.email", gated=False, load_symbols="*dex")
<Process <pre-initialized>:4335>

calc = android.applications['*calc*']
p = android.spawn(calc, gated=False, load_symbols="*dex")

# If the app is already running, you can just attach
p = android.attach("*calc*", load_symbols="*dex")
```

CHAPTER

FIFTEEN

JAVA

This is theoretically supported. However, it is currently only tested in the context of Android applications (which utilize Java).

CHAPTER

SIXTEEN

LINUX

CHAPTER

SEVENTEEN

MAC OS

Placing this here for completeness. Theoretically `revenge` should function on a Mac, but it is untested.

CHAPTER

EIGHTEEN

WINDOWS

PYTHON MODULE INDEX

r

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X