`pwnlib.tubes.process` — Processes

class `pwnlib.tubes.process.process`*(argv=None, shell=False, executable=None, cwd=None, env=None, ignore_environ=None, stdin=-1, stdout=<pwnlib.tubes.process.PTY object>, stderr=-2, close_fds=True, preexec_fn=<function process.<lambda>>, raw=True, aslr=None, setuid=None, where='local', display=None, alarm=None, creationflags=0, *args, **kwargs)* [source]

Bases: `tube`

Spawns a new process, and wraps it with a tube for communication.

Parameters

- **argv** (*list*) – List of arguments to pass to the spawned process.
- **shell** (*bool*) – Set to *True* to interpret *argv* as a string to pass to the shell for interpretation instead of as argv.
- **executable** (*str*) – Path to the binary to execute. If `None`, uses `argv[0]`. Cannot be used with `shell`.
- **cwd** (*str*) – Working directory. Uses the current working directory by default.
- **env** (*dict*) – Environment variables to add to the environment.
- **ignore_environ** (*bool*) – Ignore Python’s environment. By default use Python’s environment iff env not specified.
- **stdin** (*int*) – File object or file descriptor number to use for `stdin`. By default, a pipe is used. A pty can be used instead by setting this to `PTY`. This will cause programs to behave in an interactive manner (e.g.., `python` will show a `>>>` prompt). If the application reads from `/dev/tty` directly, use a pty.
- **stdout** (*int*) – File object or file descriptor number to use for `stdout`. By default, a pty is used so that any stdout buffering by libc routines is disabled. May also be `PIPE` to use a normal pipe.

- **stderr** (*int*) – File object or file descriptor number to use for `stderr`. By default, `STDOUT` is used. May also be `PIPE` to use a separate pipe, although the `pwnlib.tubes.tube.tube` wrapper will not be able to read this data.
- **close_fds** (*bool*) – Close all open file descriptors except stdin, stdout, stderr. By default, `True` is used.
- **preexec_fn** (*callable*) – Callable to invoke immediately before calling `execve`.
- **raw** (*bool*) – Set the created pty to raw mode (i.e. disable echo and control characters). `True` by default. If no pty is created, this has no effect.
- **aslr** (*bool*) –

  If set to `False`, disable ASLR via `personality` (`setarch -R`) and `setrlimit` (`ulimit -s unlimited`).

  This disables ASLR for the target process. However, the `setarch` changes are lost if a `setuid` binary is executed.

  The default value is inherited from `context.aslr`. See `setuid` below for additional options and information.

- **setuid** (*bool*) –

  Used to control *setuid* status of the target binary, and the corresponding actions taken.

  By default, this value is `None`, so no assumptions are made.

  If `True`, treat the target binary as `setuid`. This modifies the mechanisms used to disable ASLR on the process if `aslr=False`. This is useful for debugging locally, when the exploit is a `setuid` binary.

  If `False`, prevent `setuid` bits from taking effect on the target binary. This is only supported on Linux, with kernels v3.5 or greater.

- **where** (*str*) – Where the process is running, used for logging purposes.
- **display** (*list*) – List of arguments to display, instead of the main executable name.
- **alarm** (*int*) – Set a SIGALRM alarm timeout on the process.

- **creationflags** (*int*) – Windows only. Flags to pass to `CreateProcess`.

Examples

```
>>> p = process('python')
>>> p.sendline(b"print('Hello world')")
>>> p.sendline(b"print('Wow, such data')")
>>> b'' == p.recv(timeout=0.01)
True
>>> p.shutdown('send')
>>> p.proc.stdin.closed
True
>>> p.connected('send')
False
>>> p.recvline()
b'Hello world\n'
>>> p.recvuntil(b',')
b'Wow,'
>>> p.recvregex(b'.*data')
b' such data'
>>> p.recv()
b'\n'
>>> p.recv()
Traceback (most recent call last):
...
EOFError
```

```
>>> p = process('cat')
>>> d = open('/dev/urandom', 'rb').read(4096)
>>> p.recv(timeout=0.1)
b''
>>> p.write(d)
>>> p.recvrepeat(0.1) == d
True
>>> p.recv(timeout=0.1)
b''
>>> p.shutdown('send')
>>> p.wait_for_close()
>>> p.poll()
0
```

```
>>> p = process('cat /dev/zero | head -c8', shell=True, stderr=open('/dev/null', 'w+b'))
>>> p.recv()
b'\x00\x00\x00\x00\x00\x00\x00\x00'
```

```
>>> p = process(['python','-c','import os; print(os.read(2,1024).decode())'],
...             preexec_fn = lambda: os.dup2(0,2))
>>> p.sendline(b'hello')
>>> p.recvline()
b'hello\n'
```

```
>>> stack_smashing = ['python','-c','open("/dev/tty","wb").write(b"stack smashing
detected")']
>>> process(stack_smashing).recvall()
b'stack smashing detected'
```

```
>>> process(stack_smashing, stdout=PIPE).recvall()
b''
```

```
>>> getpass = ['python','-c','import getpass; print(getpass.getpass("XXX"))']
>>> p = process(getpass, stdin=PTY)
>>> p.recv()
b'XXX'
>>> p.sendline(b'hunter2')
>>> p.recvall()
b'\nhunter2\n'
```

```
>>> process('echo hello 1>&2', shell=True).recvall()
b'hello\n'
```

```
>>> process('echo hello 1>&2', shell=True, stderr=PIPE).recvall()
b''
```

```
>>> a = process(['cat', '/proc/self/maps']).recvall()
>>> b = process(['cat', '/proc/self/maps'], aslr=False).recvall()
>>> with context.local(aslr=False):
...     c = process(['cat', '/proc/self/maps']).recvall()
>>> a == b
False
>>> b == c
True
```

```
>>> process(['sh','-c','ulimit -s'], aslr=0).recvline()
b'unlimited\n'
```

```
>>> io = process(['sh','-c','sleep 10; exit 7'], alarm=2)
>>> io.poll(block=True) == -signal.SIGALRM
True
```

```
>>> binary = ELF.from_assembly('nop', arch='mips')
>>> p = process(binary.path)
>>> binary_dir, binary_name = os.path.split(binary.path)
>>> p = process('./{}'.format(binary_name), cwd=binary_dir)
>>> p = process(binary.path, cwd=binary_dir)
>>> p = process('./{}'.format(binary_name), cwd=os.path.relpath(binary_dir))
>>> p = process(binary.path, cwd=os.path.relpath(binary_dir))
```

`__getattr__`(*attr*) [source]

Permit pass-through access to the underlying process object for fields like `pid` and `stdin`.

`__init__`(*argv=None, shell=False, executable=None, cwd=None, env=None, ignore_environ=None, stdin=-1, stdout=<pwnlib.tubes.process.PTY object>, stderr=-2, close_fds=True, preexec_fn=<function process.<lambda>>, raw=True, aslr=None, setuid=None, where='local', display=None, alarm=None, creationflags=0, *args, **kwargs*) [source]

`__on_enoexec`(*exception*) [source]

We received an ‘exec format’ error (ENOEXEC)

This implies that the user tried to execute e.g. an ARM binary on a non-ARM system, and does not have binfmt helpers installed for QEMU.

`__preexec_fn`() [source]

Routine executed in the child process before invoking execve().

Handles setting the controlling TTY as well as invoking the user- supplied preexec_fn.

`__pty_make_controlling_tty`(*tty_fd*) [source]

This makes the pseudo-terminal the controlling tty. This should be more portable than the pty.fork() function. Specifically, this should work on Solaris.

`_validate`(*cwd, executable, argv, env*) [source]

Perform extended validation on the executable path, argv, and envp.

Mostly to make Python happy, but also to prevent common pitfalls.

`can_recv_raw`(*timeout*)→ bool [source]

Should not be called directly. Returns True, if there is data available within the timeout, but ignores the buffer on the object.

`close`() [source]

Closes the tube.

`communicate`(*stdin=None*)→ str [source]

Calls `subprocess.Popen.communicate()` method on the process.

`connected_raw`(*direction*) [source]

connected(direction = 'any') -> bool

Should not be called directly. Returns True iff the tube is connected in the given direction.

`fileno`()→ int [source]

Returns the file number used for reading.

`kill`() [source]

Kills the process.

`leak`(*address, count=1*) [source]

Leaks memory within the process at the specified address.

Parameters

- **address** (*int*) – Address to leak memory at
- **count** (*int*) – Number of bytes to leak at that address.

Example

```
>>> e = ELF(which('bash-static'))
>>> p = process(e.path)
```

In order to make sure there’s not a race condition against the process getting set up…

```
>>> p.sendline(b'echo hello')
>>> p.recvuntil(b'hello')
b'hello'
```

Now we can leak some data!

```
>>> p.leak(e.address, 4)
b'\x7fELF'
```

`libs`()→ dict [source]

Return a dictionary mapping the path of each shared library loaded by the process to the address it is loaded at in the process’ address space.

`poll`(*block=False*)→ int [source]

Parameters **block** (*bool*) – Wait for the process to exit

Poll the exit code of the process. Will return None, if the process has not yet finished and the exit code otherwise.

`readmem`(*address*, *count=1*) [source]

Leaks memory within the process at the specified address.

Parameters

- **address** (*int*) – Address to leak memory at
- **count** (*int*) – Number of bytes to leak at that address.

Example

```
>>> e = ELF(which('bash-static'))
>>> p = process(e.path)
```

In order to make sure there's not a race condition against the process getting set up…

```
>>> p.sendline(b'echo hello')
>>> p.recvuntil(b'hello')
b'hello'
```

Now we can leak some data!

```
>>> p.leak(e.address, 4)
b'\x7fELF'
```

`recv_raw`(*numb*)→ str [source]

Should not be called directly. Receives data without using the buffer on the object.

Unless there is a timeout or closed connection, this should always return data. In case of a timeout, it should return None, in case of a closed connection it should raise an `exceptions.EOFError`.

`send_raw`(*data*) [source]

Should not be called directly. Sends data to the tube.

Should return `exceptions.EOFError`, if it is unable to send any more, because of a closed tube.

`settimeout_raw`(*timeout*) [source]

Should not be called directly. Sets the timeout for the tube.

`shutdown_raw`(*direction*) [source]

Should not be called directly. Closes the tube for further reading or writing.

`writemem`(*address*, *data*) [source]

Writes memory within the process at the specified address.

Parameters

- **address** (*int*) – Address to write memory
- **data** (*bytes*) – Data to write to the address

Example

Let's write data to the beginning of the mapped memory of the ELF.

```
>>> context.clear(arch='i386')
>>> address = 0x100000
>>> data = cyclic(32)
>>> assembly = shellcraft.nop() * len(data)
```

Wait for one byte of input, then write the data to stdout

```
>>> assembly += shellcraft.write(1, address, 1)
>>> assembly += shellcraft.read(0, 'esp', 1)
>>> assembly += shellcraft.write(1, address, 32)
>>> assembly += shellcraft.exit()
>>> asm(assembly)[32:]
b'j\x01[\xb9\xff\xff\xef\xff\xf7\xd1\x89\xdaj\x04X\xcd\x801\xdb\x89\xe1j\x01Zj\x03X\xcd\x
Zj\x04X\xcd\x801\xdbj\x01X\xcd\x80'
```

Assemble the binary and test it

```
>>> elf = ELF.from_assembly(assembly, vma=address)
>>> io = elf.process()
>>> _ = io.recvuntil(b'\x90')
>>> _ = io.writemem(address, data)
>>> io.send(b'X')
>>> io.recvall()
b'aaaabaaacaaadaaaeaaafaaagaaahaaa'
```

_setuid [source]

Whether setuid is permitted

_stop_noticed= *0* [source]

Have we seen the process stop? If so, this is a unix timestamp.

alarm [source]

Alarm timeout of the process

argv [source]

Arguments passed on argv

aslr [source]

Whether ASLR should be left on

property **`corefile`** [source]

Returns a corefile for the process.

If the process is alive, attempts to create a coredump with GDB.

If the process is dead, attempts to locate the coredump created by the kernel.

property **`cwd`** [source]

Directory that the process is working in.

Example

```
>>> p = process('sh')
>>> p.sendline(b'cd /tmp; echo AAA')
>>> _ = p.recvuntil(b'AAA')
>>> p.cwd == '/tmp'
True
>>> p.sendline(b'cd /proc; echo BBB;')
>>> _ = p.recvuntil(b'BBB')
>>> p.cwd
'/proc'
```

property **`elf`** [source]

Returns an ELF file for the executable that launched the process.

`env` [source]

Environment passed on envp

`executable` [source]

Full path to the executable

property `libc` [source]

Returns an ELF for the libc for the current process. If possible, it is adjusted to the correct address automatically.

Example:

```
>>> p = process("/bin/cat")
>>> libc = p.libc
>>> libc
ELF('/lib64/libc-...so')
>>> p.close()
```

`proc`= *None* [source]

`subprocess.Popen` object that backs this process

property `program` [source]

Alias for `executable`, for backward compatibility.

Example

```
>>> p = process('/bin/true')
>>> p.executable == '/bin/true'
True
>>> p.executable == p.program
True
```

`pty` [source]

Which file descriptor is the controlling TTY

`raw` [source]

Whether the controlling TTY is set to raw mode

property **`stderr`** [source]

Shorthand for `self.proc.stderr`

See: `process.proc`

property **`stdin`** [source]

Shorthand for `self.proc.stdin`

See: `process.proc`

property **`stdout`** [source]

Shorthand for `self.proc.stdout`

See: `process.proc`