pwntools Documentation

Release 2.2

Gallopsled et al

January 05, 2015

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.

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 faster get familiar with how pwntools works.
- 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 access to everything you need to write an exploit.
- Calls `pwnlib.term.init()` to put your terminal in raw mode and implementing functionality to make it look like it is not.
- 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 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 Ubuntu 12.04 and 14.04, but 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.).

Ubuntu

First, add our Personal Package Archive repository.

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

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/osx/binutils-$AR
```

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 http://ftp.gnu.org/gnu/binutils/binutils-$V.tar.gz
wget -nc http://ftp.gnu.org/gnu/binutils/binutils-$V.tar.gz.sig

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

rm -rf binutils-*
```

```
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
sudo $MAKE install
```

Capstone

Capstone is a disassembly library required for gathering ROP gadgets and ROP chain generation.

It's a separate requirement from `binutils` because it's used by Jon Salwan's `ROPgadget` tool which we use under the covers.

In particular, version `2.1.2` should be used. Capstone can be downloaded here, or installed with the steps below.

Ubuntu

```
$ wget -nc http://www.capstone-engine.org/download/2.1.2/capstone-2.1.2_amd64.deb
$ apt-get install capstone-2.1.2_amd64.deb
```

Mac OS X

```
$ brew install capstone
```

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.

```
$ apt-get install python2.7 python2.7-dev python-pip
$ pip install pwntools
```

1.2.3 Latest Version

Alternatively if you prefer to use the latest version from the repository:

```
$ git clone https://github.com/Gallopsled/pwntools
$ cd pwntools
$ pip install -e .
```

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 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.debian.org',21)
>>> conn.recvline()
'220 ...'
>>> conn.send('USER anonymous\r\n')
>>> conn.recvuntil(' ', drop=True)
'331'
>>> conn.recvline()
'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('hello')
>>> c.recv()
'hello'
```

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

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

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

```
>>> sh.interactive()
$ 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 iwth it like a `process` tube.

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

1.3.2 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 = '37130000'.decode('hex')
>>> u32('abcd') == struct.unpack('I', 'abcd')[0]
True
```

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

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

1.3.3 Setting the Target Architecture and OS

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

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

1.3.4 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.

1.3.5 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.

```
>>> asm('mov eax, 0').encode('hex')
'b800000000'
```

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

```
>>> print disasm('6a0258cd80ebf9'.decode('hex'))
   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!

```
>>> asm(shellcraft.setreuid() + shellcraft.dupsh(4)).encode('hex')
'6a3158cd8089c389d96a4658cd806a045b6a0359496a3f58cd8075f86a68682f2f2f73682f62696e89e331c96a0b5899cd80'
```

1.3.6 Misc Tools

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

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

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

1.3.7 ELF Manipulation

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

```
>>> 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)
'ELF'
>>> e.asm(e.address, 'ret')
>>> e.save('/tmp/quiet-cat')
>>> disasm(file('/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.

- **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 `pwnlib.context.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 contining 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 fancyness**
    - *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 evalutaing 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.

1.5.1 asm

Assemble shellcode into bytes

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

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

-f <format>**, --format** <format>
Output format (defaults to hex for ttys, otherwise raw)

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

-c <<opt>>**, --context** <<opt>>
The os/architecture the shellcode will run in (default: linux/i386), choose from: aarch64, alpha, amd64, arm, avr, cris, freebsd, i386, ia64, linux, m68k, mips, mips64, msp430, powerpc, powerpc64, s390, sparc, sparc64, thumb, vax, windows

1.5.2 constgrep

Looking up constants from header files.

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

regex
The regex matching constant you want to find

constant
The constant to find

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

-e <<constant name>>, **--exact** <<constant name>>
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 <<opt>>, **--context** <<opt>>
The os/architecture to find constants for (default: linux/i386), choose from: aarch64, alpha, amd64, arm, avr, cris, freebsd, i386, ia64, linux, m68k, mips, mips64, msp430, powerpc, powerpc64, s390, sparc, sparc64, thumb, vax, windows

1.5.3 cyclic

Cyclic pattern creator/finder

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).

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

1.5.4 disasm

Disassemble bytes into text format

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

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

-c <<opt>>, **--context** <<opt>>
The architecture of the shellcode (default: i386), choose from: powerpc64, aarch64, sparc64, powerpc, mips64, msp430, thumb, amd64, sparc, alpha, s390, i386, m68k, mips, ia64, cris, vax, avr, arm

1.5.5 elfdiff

a

b

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

1.5.6 elfpatch

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

1.5.7 hex

Hex-encodes data provided on the command line or via stdin.

data
Data to convert into hex

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

1.5.8 phd

Pwnlib HexDump

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 <color>
Colorize the output. When 'auto' output is colorized exactly when stdout is a TTY. Default is 'auto'.

1.5.9 shellcraft

Microwave shellcode – Easy, fast and delicious

<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 <<format>>, **--format** <<format>>
Output format (default: hex), choose from {r}aw, {s}tring, {c}-style array, {h}ex string, hex{i}i, {a}ssembly code, {p}reprocssed code

1.5.10 unhex

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

hex
Hex bytes to decode

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

CHAPTER 2

Module Index

Each of the pwntools modules is documented here.

2.1 pwnlib.asm — Assembler functions

Utilities for assembling and disassembling code.

2.1.1 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().

2.1.2 Assembly

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

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

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

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

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

```
>>> asm(shellcraft.sh())
'jhh///sh/bin\x89\xe31\xc9j\x0bX\x99\xcd\x80'
```

2.1.3 Disassembly

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

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

pwnlib.asm.**asm**(*code*, *vma = 0*, ...) → 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.contex`.

To support all these architecture, we bundle the GNU assembler and objcopy with pwntools.

Parameters

- **shellcode** (*str*) – Assembler code to assemble.
- **vma** (*int*) – Virtual memory address of the beginning of assembly

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

Examples

```
>>> asm("mov eax, SYS_select", arch = 'i386', os = 'freebsd')
'\xb8]\x00\x00\x00'
>>> asm("mov eax, SYS_select", arch = 'amd64', os = 'linux')
'\xb8\x17\x00\x00\x00'
>>> asm("mov rax, SYS_select", arch = 'amd64', os = 'linux')
'H\xc7\xc0\x17\x00\x00\x00'
>>> asm("ldr r0, =SYS_select", arch = 'arm', os = 'linux', bits=32)
'\x04\x00\x1f\xe5R\x00\x90\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 (0x900000+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`.

To support all these architecture, we bundle the GNU objcopy and objdump with pwntools.

Parameters

- **data** (*str*) – Bytestring to disassemble.
- **vma** (*int*) – Passed through to the –adjust-vma argument of objdump

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

Examples

```
>>> print disasm('b85d000000'.decode('hex'), arch = 'i386')
   0:   b8 5d 00 00 00          mov    eax,0x5d
>>> print disasm('b817000000'.decode('hex'), arch = 'amd64')
   0:   b8 17 00 00 00          mov    eax,0x17
>>> print disasm('48c7c017000000'.decode('hex'), arch = 'amd64')
   0:   48 c7 c0 17 00 00 00    mov    rax,0x17
>>> print disasm('04001fe552009000'.decode('hex'), arch = 'arm')
   0:   e51f0004        ldr     r0, [pc, #-4]   ; 0x4
   4:   00900052        addseq  r0, r0, r2, asr r0
>>> print disasm('4ff00500'.decode('hex'), arch = 'thumb', bits=32)
   0:   f04f 0005       mov.w   r0, #5
```

pwnlib.asm.**which_binutils**(*util*, ***kwargs*)

Finds a binutils in the PATH somewhere. Expects that the utility is prefixed with the architecture name.

Examples

```
>>> import platform
>>> which_binutils('as', arch=platform.machine())
'.../bin/as'
>>> which_binutils('as', arch='arm')
'.../bin/arm-...-as'
>>> which_binutils('as', arch='powerpc')
'.../bin/powerpc...-as'
>>> which_binutils('as', arch='msp430')
...
Traceback (most recent call last):
...
Exception: Could not find 'as' installed for ContextType(arch = 'msp430')
Traceback (most recent call last):
...
Exception: Could not find 'as' installed for ContextType(arch = 'msp430')
```

2.2 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.

2.3 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.

2.4 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.

Example

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

2.5 `pwnlib.context` — Setting runtime variables

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

Global `context` object, used to store commonly-used pwntools settings. In most cases, the context is used to infer default variables values. For example, `pwnlib.asm.asm()` can take an `os` parameter as a keyword argument. If it is not supplied, the `os` specified by `context` is used instead. Consider it a shorthand to passing `os=` and `arch=` to every single function call.

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

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

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

```
#!/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')
>>> 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 pwnlib.asm.asm('nop').encode('hex')
>>> 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
```

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():
...     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.

`ContextType.__call__`(***kwargs*)
Alias for `pwnlib.context.ContextType.update()`

`ContextType.arch`
Target machine 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'
Traceback (most recent call last):
 ...
AttributeError: arch must be one of ['aarch64', ..., 'thumb']
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
```

ContextType.**architectures** = OrderedDict([('powerpc64', {'bits': 64, 'endian': 'big'}), ('aarch64', {'bits': 64, 'end

Keys are valid values for `pwnlib.context.ContextType.arch()`. Values are defaults which are set when `pwnlib.context.ContextType.arch` is set

ContextType.**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
Traceback (most recent call last):
...
AttributeError: bits must be >= 0 (-1)
Traceback (most recent call last):
...
AttributeError: bits must be >= 0 (-1)
```

ContextType.**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
Traceback (most recent call last):
...
AttributeError: bits must be >= 0 (0)
Traceback (most recent call last):
...
AttributeError: bits must be >= 0 (0)
```

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

Examples

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

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

Examples

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

ContextType.**defaults = {'newline': '\n', 'arch': 'i386', 'log_level': 20, 'bits': 32, 'signed': False, 'timeout': 104857**
Default values for pwnlib.context.ContextType

ContextType.**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'
Traceback (most recent call last):
 ...
AttributeError: endian must be one of ['be', 'big', 'eb', 'el', 'le', 'little']
Traceback (most recent call last):
 ...
AttributeError: endian must be one of ['be', 'big', 'eb', 'el', 'le', 'little']
```

ContextType.**endianness**
Legacy alias for endian.

Examples

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

ContextType.**endiannesses = OrderedDict([('little', 'little'), ('big', 'big'), ('el', 'little'), ('le', 'little'), ('be', 'big'), ('e**
Valid values for endian

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

```
>>> print context.timeout
1.0
```

ContextType.**log_level**
Sets the verbosity of pwntools logging mechanism.

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'
Traceback (most recent call last):
...
AttributeError: log_level must be an integer or one of ['CRITICAL', 'DEBUG', 'ERROR', 'INFO'
Traceback (most recent call last):
...
AttributeError: log_level must be an integer or one of ['CRITICAL', 'DEBUG', 'ERROR', 'INFO'
```

ContextType.**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'
Traceback (most recent call last):
...
AttributeError: os must be one of ['freebsd', 'linux', 'windows']
Traceback (most recent call last):
...
AttributeError: os must be one of ['freebsd', 'linux', 'windows']
```

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

ContextType.**reset_local**()
Deprecated. Use clear().

ContextType.**sign**
Alias for signed

ContextType.**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'
Traceback (most recent call last):
...
AttributeError: signed must be one of ['no', 'signed', 'unsigned', 'yes'] or a non-string tr
Traceback (most recent call last):
...
AttributeError: signed must be one of ['no', 'signed', 'unsigned', 'yes'] or a non-string tr
```

ContextType.**signedness**
Alias for `signed`

ContextType.**signednesses = {'yes': True, 'unsigned': False, 'signed': True, 'no': False}**
Valid string values for `signed`

ContextType.**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.

ContextType.**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')
```

ContextType.**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)
>>> _=(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.

2.6 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.

2.6.1 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, (data or '').encode('hex')))
    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*)

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.

bases()
Resolve base addresses of all loaded libraries.

Return a dictionary mapping library path to its base address.

dynamic

Returns Pointer to the `.DYNAMIC` area.

elfclass
32 or 64

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

link_map
Pointer to the runtime link_map object

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

Parameters

- **symb** (*str*) – Named routine to look up
- **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`.

`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.

2.7 `pwnlib.elf` — Working with ELF binaries

Exposes functionality for manipulating ELF files

`pwnlib.elf.`**load**(**args*, ***kwargs*)
Compatibility wrapper for pwntools v1

class pwnlib.elf.**ELF**(*path*)

Encapsulates information about an ELF file.

Variables

- **path** – Path to the binary on disk
- **symbols** – Dictionary of {name: address} for all symbols in the ELF
- **plt** – Dictionary of {name: address} for all functions in the PLT
- **got** – Dictionary of {name: address} for all function pointers in the GOT
- **libs** – Dictionary of {path: address} for each shared object required to load the ELF

Example:

```
bash = ELF(which('bash'))
hex(bash.symbols['read'])
# 0x41dac0
hex(bash.plt['read'])
# 0x41dac0
u32(bash.read(bash.got['read'], 4))
# 0x41dac6
print disasm(bash.read(bash.plt['read'],16), arch='amd64')
# 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    0xffffffffffffa60
```

address

Address of the lowest segment loaded in the ELF. When updated, cascades updates to segment vaddrs, section addrs, symbols, plt, and got.

```
>>> bash = ELF(which('bash'))
>>> old = bash.symbols['read']
>>> bash.address += 0x1000
>>> bash.symbols['read'] == old + 0x1000
True
```

asm(*address*, *assembly*)

Assembles the specified instructions and inserts them into the ELF at the specified address.

The resulting binary can be saved with ELF.save()

bss(*offset=0*)

Returns an index into the .bss segment

disasm(*address*, *n_bytes*)

Returns a string of disassembled instructions at the specified virtual memory address

dwarf

DWARF info for the elf

elfclass

ELF class (32 or 64).

Note: Set during ELFFile._identify_file

elftype

ELF type (EXEC, DYN, etc)

entry

Entry point to the ELF

entrypoint
Entry point to the ELF

executable_segments
Returns: list of all segments which are executable.

get_data()
Retrieve the raw data from the ELF file.

```
>>> bash = ELF(which('bash'))
>>> fd   = open(which('bash'))
>>> bash.get_data() == fd.read()
True
```

non_writable_segments
Returns: list of all segments which are NOT writeable

offset_to_vaddr(*offset*)
Translates the specified offset to a virtual address.

Parameters **offset** (*int*) – Offset to translate

Returns Virtual address which corresponds to the file offset, or None

Examples

```
>>> bash = ELF(which('bash'))
>>> bash.address == bash.offset_to_vaddr(0)
True
>>> bash.address += 0x123456
>>> bash.address == bash.offset_to_vaddr(0)
True
```

read(*address*, *count*)
Read data from the specified virtual address

Parameters

- **address** (*int*) – Virtual address to read
- **count** (*int*) – Number of bytes to read

Returns A string of bytes, or None

Examples

```
>>> bash = ELF(which('bash'))
>>> bash.read(bash.address+1, 3)
'ELF'
```

save(*path*)
Save the ELF to a file

```
>>> bash = ELF(which('bash'))
>>> bash.save('/tmp/bash_copy')
>>> copy = file('/tmp/bash_copy')
>>> bash = file(which('bash'))
>>> bash.read() == copy.read()
True
```

search (*needle*, *writable* = *False*) → str generator
Search the ELF's virtual address space for the specified string.

Parameters

- **needle** (*str*) – String to search for.
- **writable** (*bool*) – Search only writable sections.

Returns An iterator for each virtual address that matches.

Examples

```
>>> bash = ELF(which('bash'))
>>> bash.address + 1 == next(bash.search('ELF'))
True

>>> sh = ELF(which('bash'))
>>> # /bin/sh should only depend on libc
>>> libc_path = [key for key in sh.libs.keys() if 'libc' in key][0]
>>> libc = ELF(libc_path)
>>> # this string should be in there because of system(3)
>>> len(list(libc.search('/bin/sh'))) > 0
True
```

section (*name*)
Gets data for the named section

Parameters **name** (*str*) – Name of the section

Returns String containing the bytes for that section

sections
A list of all sections in the ELF

segments
A list of all segments in the ELF

start
Entry point to the ELF

vaddr_to_offset (*address*)
Translates the specified virtual address to a file address

Parameters **address** (*int*) – Virtual address to translate

Returns Offset within the ELF file which corresponds to the address, or None.

Examples

```
>>> bash = ELF(which('bash'))
>>> 0 == bash.vaddr_to_offset(bash.address)
True
>>> bash.address += 0x123456
>>> 0 == bash.vaddr_to_offset(bash.address)
True
```

writable_segments
Returns: list of all segments which are writeable

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)
'ELF'
>>> bash.write(bash.address, "HELO")
>>> bash.read(bash.address, 4)
'HELO'
```

2.8 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.

2.9 pwnlib.gdb — Working with GDB

pwnlib.gdb.**attach**(*target*, *execute = None*, *exe = None*, *arch = None*) → None
Start GDB in a new terminal and attach to *target*. `pwnlib.util.proc.pidof()` is used to find the PID of *target* except when *target* is a `(host, port)`-pair. In that case *target* is assumed to be a GDB server.

If it is running locally and *exe* is not given we will try to find the path of the target binary from parsing the command line of the program running the GDB server (e.g. qemu or gdbserver). Notice that if the PID is known (when *target* is not a GDB server) *exe* will be read from `/proc/<pid>/exe`.

If *gdb-multiarch* is installed we use that or 'gdb' otherwise.

Parameters

- **target** – The target to attach to.
- **execute** (*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

Returns `None`

pwnlib.gdb.**debug**(*args*) → tube
Launch a GDB server with the specified command line, and launches GDB to attach to it.

Parameters

- **args** – Same args as passed to pwnlib.tubes.process
- **ssh** – Remote ssh session to use to launch the process. Automatically sets up port forwarding so that gdb runs locally.

Returns A tube connected to the target process

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
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='bandit.labs.overthewire.org',user='bandit0',password='bandit0')
...     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'])
'0x7ffff7634660'
```

2.10 pwnlib.log and — Logging stuff

Logging module for printing status during an exploit, and internally within pwntools.

2.10.1 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 bruteforcing).:

```
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 context.log_level on the global context object.:

```
log.info("No you see me")
context.log_level = 'error'
log.info("Now you don't")
```

2.10.2 Pwnlib Developers

A module-specific logger can be imported into the module via:

```
log = logging.getLogger(__name__)
```

This provides an easy way to filter logging programmatically or via a configuration file for debugging.

There's no need to expressly import this log module.

When using progress, you should use the with keyword to manage scoping, to ensure the spinner stops if an exception is thrown.

class pwnlib.log.**Logger**(**args*, ***kwargs*)

Specialization of logging.Logger which uses pwnlib.context.context.log_level to infer verbosity.

Also adds some pwnlib flavor via:

- progress()
- success()
- failure()
- indented()

Adds pwnlib-specific information for coloring and indentation to the log records passed to the logging.Formatter.

Internal:

Permits prepending a string to each message, by means of msg_prefix. This is leveraged for progress messages.

debug (*message*)
Logs a debug message.

error (*message*)
To be called outside an exception handler.

Logs an error message, then raises a PwnlibException.

failure (*message*)
Logs a failure message. If the Logger is animated, the animation is stopped.

indented (*message*, *level=logging.INFO*)
Log an info message without the line prefix.

Parameters **level** (*int*) – Alternate log level at which to set the indented message.

info (*message*)
Logs an info message.

info_once (*message*)
Logs an info message. The same message is never printed again.

progress (*self*) → Logger
Creates a Logger with a progress animation, which can be stopped via success(), and failure().

The Logger returned is also a scope manager. Using scope managers ensures that the animation is stopped, even if an exception is thrown.

::

with log.progress('Trying something...') as p:

for i in range(10): p.status("At %i" % i) time.sleep(0.5)

x = 1/0

success (*message*)
Logs a success message. If the Logger is animated, the animation is stopped.

warn (*message*)
Logs a warning message.

warn_once (*message*)
Logs a warning message. The same message is never printed again.

2.11 pwnlib.memleak — Helper class for leaking memory

class pwnlib.memleak.**MemLeak** (*f*, *search_range=20*, *reraise=True*)
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
'\x7fELF'
>>> hex(leaker.d(0))
'0x464c457f'
>>> hex(leaker.clearb(1))
'0x45'
>>> hex(leaker.d(0))
leaking 0x1
'0x464c457f'
```

b (*addr*, *ndx* = *0*) → int
Leak byte at `((uint8_t*) addr)[ndx]`

Examples

```
>>> import string
>>> data = string.ascii_lowercase
>>> 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:'a'}
>>> l.n(0,1) == 'a'
True
>>> l.clearb(0) == unpack('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:'a', 1: 'b', 2: 'c', 3: 'd'}
>>> l.n(0, 4) == 'abcd'
True
>>> l.cleard(0) == unpack('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: '')
>>> c.cache = {x:'x' for x in range(0x100, 0x108)}
>>> c.clearq(0x100) == unpack('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:'a', 1: 'b'}
>>> l.n(0, 2) == 'ab'
True
>>> l.clearw(0) == unpack('ab', 16)
True
```

```
>>> l.cache
{}
```

d(*addr*, *ndx* = *0*) → int
Leak dword at `((uint32_t*) addr)[ndx]`

Examples

```
>>> import string
>>> data = string.ascii_lowercase
>>> l = MemLeak(lambda a: data[a:a+8], reraise=False)
>>> l.d(0) == unpack('abcd', 32)
True
>>> l.d(22) == unpack('wxyz', 32)
True
>>> l.d(23) is None
True
```

field(*address*, *obj*)
call(address, field) => int or str

Leak an entire structure, or structure field.

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`.

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
>>> l = MemLeak(lambda a: data[a:a+4], reraise=False)
>>> l.n(0,1) == 'a'
True
>>> l.n(0,26) == data
True
>>> len(l.n(0,26)) == 26
True
>>> l.n(0,27) is None
True
```

q(*addr*, *ndx* = *0*) → int
Leak qword at `((uint64_t*) addr)[ndx]`

Examples

```
>>> import string
>>> data = string.ascii_lowercase
>>> l = MemLeak(lambda a: data[a:a+16], reraise=False)
>>> l.q(0) == unpack('abcdefgh', 64)
True
>>> l.q(18) == unpack('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 = "Hello\x00World"
>>> l = MemLeak(lambda a: data[a:a+4], reraise=False)
>>> l.s(0) == "Hello"
True
>>> l.s(5) == ""
True
>>> l.s(6) == "World"
True
>>> l.s(999) == ""
True
```

setb(*addr*, *val*, *ndx=0*)
Sets byte at `((uint8_t*)addr)[ndx]` to *val* in the cache.

Examples

```
>>> l = MemLeak(lambda x: '')
>>> l.cache == {}
True
>>> l.setb(33, 0x41)
>>> l.cache == {33: '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: '')
>>> l.cache == {}
True
>>> l.sets(0, 'H\x00ello')
>>> l.cache == {0: 'H', 1: '\x00', 2: 'e', 3: 'l', 4: 'l', 5: 'o', 6: '\x00'}
True
```

setw(*addr*, *val*, *ndx=0*)
Sets word at `((uint16_t*)addr)[ndx]` to *val* in the cache.

Examples

```
>>> l = MemLeak(lambda x: '')
>>> l.cache == {}
True
>>> l.setw(33, 0x41)
>>> l.cache == {33: 'A', 34: '\x00'}
True
```

w(*addr*, *ndx = 0*) → int
Leak word at `((uint16_t*) addr)[ndx]`

Examples

```
>>> import string
>>> data = string.ascii_lowercase
>>> l = MemLeak(lambda a: data[a:a+4], reraise=False)
>>> l.w(0) == unpack('ab', 16)
True
>>> l.w(24) == unpack('yz', 16)
True
>>> l.w(25) is None
True
```

2.12 `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 recieved.

Parameters **n** (*int*) – Number of seconds to sleep.

2.13 pwnlib.rop — Return Oriented Programming

Return Oriented Programming

class pwnlib.rop.**ROP**(*elfs*, *base=None*)

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']
str(rop)
# '\xfc\x82\x04\x08\xef\xbe\xad\xde\x00\x00\x00\x00\xa8\x96\x04\x08'
```

__getattr__(*attr*)

Helper to make finding ROP gadets easier.

Also provides a shorthand for .call(): ` rop.function(args) ==> rop.call(function, args) `

```
>>> 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     == rop.search(move=rop.align)
True
>>> rop.ret_8   == rop.search(move=8)
True
>>> rop.ret     != None
True
```

__str__()

Returns: Raw bytes of the ROP chain

build(*base=None*)

Build the ROP chain into a list (addr, int/string, bool), where the last value is True iff the value was an internal reference.

It is guaranteed that the individual parts are next to each other.

If there is no base available, then the returned addresses are indexed from 0.

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

call(*resolvable*, *arguments=()*)

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()
Build the ROP chain

Returns str containging raw ROP bytes

dump()
Dump the ROP chain in an easy-to-read manner

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/str*) – The raw value to put onto the rop chain.

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

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 tuple of (address, info) in the same format as self.gadgets.items().

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.

2.14 `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.

Example

```
>>> print shellcraft.i386.nop().strip('\n')
    nop
>>> print shellcraft.i386.linux.sh()
    /* push '/bin///sh\x00' */
    push 0x68
    push 0x732f2f2f
    push 0x6e69622f
...
```

2.14.1 Submodules

pwnlib.shellcraft.amd64 — Shellcode for AMD64

pwnlib.shellcraft.amd64

Shellcraft module containing generic Intel x86_64 shellcodes.

pwnlib.shellcraft.amd64.**infloop**()
A two-byte infinite loop.

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.

Example

```
>>> print shellcraft.amd64.mov('eax','ebx').rstrip()
    mov eax, ebx
>>> print shellcraft.amd64.mov('eax', 0).rstrip()
    xor eax, eax
>>> print shellcraft.amd64.mov('ax', 0).rstrip()
    xor ax, ax
>>> print shellcraft.amd64.mov('rax', 0).rstrip()
    xor eax, eax
>>> print shellcraft.amd64.mov('al', 'ax').rstrip()
    /* moving ax into al, but this is a no-op */
>>> print shellcraft.amd64.mov('bl', 'ax').rstrip()
    mov bl, al
>>> print shellcraft.amd64.mov('ax', 'bl').rstrip()
    movzx ax, bl
>>> print shellcraft.amd64.mov('eax', 1).rstrip()
    push 0x1
    pop rax
>>> print shellcraft.amd64.mov('rax', 0xdead00ff).rstrip()
    mov eax, 0x1010101
    xor eax, 0xdfac01fe
>>> print shellcraft.amd64.mov('rax', 0x11dead00ff).rstrip()
    mov rax, 0x101010101010101
    push rax
    mov rax, 0x1010110dfac01fe
```

```
xor [rsp], rax
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.**push**(*value*)
Pushes a value onto the stack without using null bytes or newline characters.

Parameters **value** (*int,str*) – The value or register to push

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 '\x00' */
    push 1
    dec byte ptr [rsp]
>>> print shellcraft.amd64.pushstr('a').rstrip()
    /* push 'a\x00' */
    push 0x61
>>> print shellcraft.amd64.pushstr('aa').rstrip()
    /* push 'aa\x00' */
    push 0x...
    xor dword ptr [rsp], 0x...
>>> print shellcraft.amd64.pushstr('aaa').rstrip()
    /* push 'aaa\x00' */
    push 0x...
    xor dword ptr [rsp], 0x...
>>> print shellcraft.amd64.pushstr('aaaa').rstrip()
    /* push 'aaaa\x00' */
    push 0x61616161
>>> print shellcraft.amd64.pushstr('aaa\xc3').rstrip()
    /* push 'aaa\xc3\x00' */
    push 0x...
    xor dword ptr [rsp], 0x...
>>> print shellcraft.amd64.pushstr('aaa\xc3', append_null = False).rstrip()
    /* push 'aaa\xc3' */
    push 0x...
>>> print shellcraft.amd64.pushstr('\xc3').rstrip()
    /* push '\xc3\x00' */
    push 0x...
    xor dword ptr [rsp], 0x...
>>> print shellcraft.amd64.pushstr('\xc3', append_null = False).rstrip()
    /* push '\xc3' */
    push 0x...c3
>>> with context.local():
...     context.arch = 'amd64'
```

```
... print enhex(asm(shellcraft.pushstr("/bin/sh")))
48b8010101010101010150 48b82e63686f2e72690148310424
>>> with context.local():
...     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.**ret**(*return_value=None*)
A single-byte RET instruction.

Parameters **return_value** – Value to return

pwnlib.shellcraft.amd64.**trap**()
A trap instruction.

pwnlib.shellcraft.amd64.linux

Shellcraft module containing Intel x86_64 shellcodes for Linux.

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='rbp'*)
Writes a string to a file descriptor

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 /bin/sh

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

Example

```
>>> print pwnlib.shellcraft.amd64.linux.syscall('SYS_execve', 1, 'rsp', 2, 0).rstrip()
    /* call execve(1, 'rsp', 2, 0) */
    push 0x1
    pop rdi
```

```
    mov rsi, rsp
    push 0x2
    pop rdx
    xor r10d, r10d
    push 0x3b
    pop rax
    syscall
>>> print pwnlib.shellcraft.amd64.linux.syscall('SYS_execve', 2, 1, 0, -1).rstrip()
    /* call execve(2, 1, 0, -1) */
    push 0x2
    pop rdi
    push 0x1
    pop rsi
    push -1
    pop r10
    push 0x3b
    pop rax
    cdq /* Set rdx to 0, rax is known to be positive */
    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') */
    /* moving rdi into rdi, but this is a no-op */
    /* moving rsi into rsi, but this is a no-op */
    /* moving rax into rax, but this is a no-op */
    syscall
>>> print pwnlib.shellcraft.amd64.linux.syscall('rbp', None, None, 1).rstrip()
    /* call syscall('rbp', ?, ?, 1) */
    push 0x1
    pop rdx
    mov rax, rbp
    syscall
```

pwnlib.shellcraft.arm — Shellcode for ARM

pwnlib.shellcraft.arm

Shellcraft module containing generic ARM little endian shellcodes.

pwnlib.shellcraft.arm.**infloop**()
An infinite loop.

pwnlib.shellcraft.arm.**mov**(*dst*, *src*)
Returns THUMB code for moving the specified source value into the specified destination register.

pwnlib.shellcraft.arm.**nop**()
A nop instruction.

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.**to_thumb**()
Go from ARM to THUMB mode.

pwnlib.shellcraft.arm.linux

Shellcraft module containing ARM shellcodes for Linux.

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.**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 /bin/sh

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.**infloop**()
A two-byte infinite loop.

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.

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()
    push 0x11
    pop ax
    inc esp
    inc esp
>>> print shellcraft.i386.mov('al', 'ax').rstrip()
    /* moving ax into al, but this is a no-op */
>>> print shellcraft.i386.mov('bl', 'ax').rstrip()
    mov bl, al
>>> print shellcraft.i386.mov('ax', 'bl').rstrip()
    movzx ax, bl
>>> print shellcraft.i386.mov('eax', 1).rstrip()
    push 0x1
    pop eax
>>> print shellcraft.i386.mov('eax', 0xdead00ff).rstrip()
    mov eax, 0x1010101
    xor eax, 0xdfac01fe
```

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.i386.**nop**()
A single-byte nop instruction.

pwnlib.shellcraft.i386.**push**(*value*)
Pushes a value onto the stack without using null bytes or newline characters.

Parameters **value** (*int,str*) – The value or register to push

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' */
    push 0x61616161
>>> print shellcraft.i386.pushstr('\xc3').rstrip()
    /* push '\xc3\x00' */
    push 0x1010101
    xor dword ptr [esp], 0x10101c2
>>> print shellcraft.i386.pushstr('\xc3', append_null = False).rstrip()
    /* push '\xc3' */
    push 0x...c3
>>> with context.local():
...     context.arch = 'i386'
...     print enhex(asm(shellcraft.pushstr("/bin/sh")))
6801010101813424 2e726901682f62696e
>>> 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.**ret**(*return_value=None*)

A single-byte RET instruction.

Parameters **return_value** – Value to return

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.**trap**()
A trap instruction.

pwnlib.shellcraft.i386.linux

Shellcraft module containing Intel i386 shellcodes for Linux.

pwnlib.shellcraft.i386.linux.**acceptloop_ipv4**(*port*)
Args: port Waits for a connection. Leaves socket in EBP. ipv4 only

pwnlib.shellcraft.i386.linux.**dup**(*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='ebp'*)
Writes a string to a file descriptor

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**(*size*, *port=None*)
Findpeer + stager

pwnlib.shellcraft.i386.linux.**getdents**(*in_fd='ebp'*, *size=255*, *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.**i386_to_amd64**()
Returns code to switch from i386 to amd64 mode.

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.**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 /bin/sh

pwnlib.shellcraft.i386.linux.**stager**(*sock*, *size*)
Recives a fixed sized payload into a mmaped buffer Useful in conjuncion with findpeer.

pwnlib.shellcraft.i386.linux.**syscall**(*syscall=None*, *arg0=None*, *arg1=None*, *arg2=None*, *arg3=None*, *arg4=None*)

Args: [syscall_number, *args] Does a syscall

Example

```
>>> print pwnlib.shellcraft.i386.linux.syscall('SYS_execve', 1, 'esp', 2, 0).rstrip()
    /* call execve(1, 'esp', 2, 0) */
    push 0x1
    pop ebx
    mov ecx, esp
    push 0x2
    pop edx
    xor esi, esi
    push 0xb
    pop eax
    int 0x80
>>> print pwnlib.shellcraft.i386.linux.syscall('SYS_execve', 2, 1, 0, 20).rstrip()
    /* call execve(2, 1, 0, 20) */
    push 0x2
    pop ebx
    push 0x1
    pop ecx
    push 0x14
    pop esi
    push 0xb
    pop eax
    cdq /* Set edx to 0, eax is known to be positive */
    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') */
    /* moving ebx into ebx, but this is a no-op */
    /* moving ecx into ecx, but this is a no-op */
    /* moving eax into eax, but this is a no-op */
    int 0x80
>>> print pwnlib.shellcraft.i386.linux.syscall('ebp', None, None, 1).rstrip()
    /* call syscall('ebp', ?, ?, 1) */
    push 0x1
    pop edx
    mov eax, ebp
    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.thumb — Shellcode for Thumb Mode

pwnlib.shellcraft.thumb

Shellcraft module containing generic thumb little endian shellcodes.

pwnlib.shellcraft.thumb.**infloop**()
An infinite loop.

pwnlib.shellcraft.thumb.**mov**(*dst*, *src*)
Returns THUMB code for moving the specified source value into the specified destination register.

pwnlib.shellcraft.thumb.**nop**()
A nop instruction.

pwnlib.shellcraft.thumb.**pushstr**(*string*, *append_null=True*)
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

>>>> with context.local(): ... context.arch = 'thumb' ... print enhex(asm(shellcraft.pushstr('HellonWorld!', True))) 81ea010102b4dff8041001e0726c642102b4dff8041001e06f0a576f02b4dff8041001e048656c6c02b4 >>>> with context.local(): ... context.arch = 'thumb' ... print enhex(asm(shellcraft.pushstr('',

True))) 81ea010102b4 >>>> with context.local(): ... context.arch = 'thumb' ... print enhex(asm(shellcraft.pushstr('x00', False))) 81ea010102b4

pwnlib.shellcraft.thumb.**ret**(*return_value=None*)
A single-byte RET instruction.

Parameters **return_value** – Value to return

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.**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.**findpeer**(*port*)
Finds a connected socket. If port is specified it is checked against the peer port. Resulting socket is left in r6.

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.**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'.

pwnlib.shellcraft.thumb.linux.**sh**()
Execute /bin/sh

2.15 pwnlib.term — Terminal handling

pwnlib.term.**can_init**()
This function returns True iff stdout is a tty and we are not inside a REPL.

pwnlib.term.**init**()
Calling this function will take over the terminal (if `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()`.

2.16 pwnlib.timeout — Timeout handling

Timeout encapsulation, complete with countdowns and scope managers.

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):
...     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):
...     for i in range(5):
...         print round(t.timeout,1)
...         time.sleep(0.1)
0.5
0.5
0.5
0.5
0.5
>>> print t.timeout
5.0
```

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.

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

local (*timeout*)
Scoped timeout setter. Sets the timeout within the scope, and restores it when leaving the scope.

maximum = 1048576.0
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`.

timeout_change()
Callback for subclasses to hook a timeout change.

2.17 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.

2.17.1 Sockets

class pwnlib.tubes.remote.**remote**(*host*, *port*, *fam='any'*, *typ='tcp'*, *timeout=pwnlib.timeout.Timeout.default*, *ssl=False*, *sock=None*)
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
- **sock** (*socket*) – Socket to inherit, rather than connecting

Examples

```
>>> r = remote('google.com', 443, ssl=True)
>>> r.send('GET /\r\n\r\n')
>>> r.recvn(4)
'HTTP'
>>> r = remote('127.0.0.1', 1)
Traceback (most recent call last):
```

```
...
PwnlibException: Could not connect to 127.0.0.1 on port 1
>>> import socket
>>> s = socket.socket()
>>> s.connect(('google.com', 80))
>>> s.send('GET /' + '\r\n'*2)
9
>>> r = remote.fromsocket(s)
>>> r.recvn(4)
'HTTP'
Traceback (most recent call last):
...
PwnlibException: Could not connect to 127.0.0.1 on port 1
```

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='0.0.0.0'*, *fam='any'*, *typ='tcp'*, *timeout=pwnlib.timeout.Timeout.default*)
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.
- **bindaddr** (*str*) – The address to bind 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

wait_for_connection()
Blocks until a connection has been established.

class pwnlib.tubes.sock.**sock**
Bases: pwnlib.tubes.tube.tube

Methods available exclusively to sockets.

2.17.2 Processes

class pwnlib.tubes.process.**process**(*args*, *shell=False*, *executable=None*, *cwd=None*, *env=None*, *timeout=pwnlib.timeout.Timeout.default*)
Bases: pwnlib.tubes.tube.tube

Implements a tube which talks to a process on stdin/stdout/stderr.

Examples

```
>>> context.log_level='error'
>>> p = process(which('python2'))
>>> p.sendline("print 'Hello world'")
>>> p.sendline("print 'Wow, such data'");
>>> '' == p.recv(timeout=0.01)
True
>>> p.shutdown('send')
>>> p.proc.stdin.closed
True
>>> p.connected('send')
False
>>> p.recvline()
'Hello world\n'
>>> p.recvuntil(',')
'Wow,'
>>> p.recvregex('.*data')
' such data'
>>> p.recv()
'\n'
>>> p.recv()
Traceback (most recent call last):
...
EOFError
Traceback (most recent call last):
...
EOFError
```

communicate(*stdin = None*) → str
Calls `subprocess.Popen.communicate()` method on the process.

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.

2.17.3 SSH

class pwnlib.tubes.ssh.**ssh**(*user*, *host*, *port=22*, *password=None*, *key=None*, *keyfile=None*, *proxy_command=None*, *proxy_sock=None*, *timeout=pwnlib.timeout.Timeout.default*)

__call__(*attr*)
Permits function-style access to run commands over SSH

Examples

```
>>> with ssh(host='localhost',
...          user='demouser',
...          password='demopass') as s:
...     print repr(s('echo hello'))
'hello'
```

__getattr__(*attr*)
Permits member access to run commands over SSH

Examples

```
>>> with ssh(host='localhost',
...          user='demouser',
...          password='demopass') as s:
...     print s.echo('hello')
...     print s.whoami()
...     print s.echo(['huh','yay','args'])
hello
demouser
huh yay args
```

__getitem__(*attr*)
Permits indexed access to run commands over SSH

Examples

```
>>> with ssh(host='localhost',
...          user='demouser',
...          password='demopass') as s:
...     print s['echo hello']
hello
```

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()
>>> with ssh(host='localhost',
...          user='demouser',
...          password='demopass') as s:
...     a = s.connect_remote('localhost', l.lport)
...     b = l.wait_for_connection()
...     a.sendline('Hello')
...     print repr(b.recvline())
'Hello\n'
```

connected()
Returns True if we are connected.

Example

```
>>> with ssh(host='localhost',
...          user='demouser',
...          password='demopass') as s:
...      print s.connected()
...      s.close()
...      print s.connected()
True
False
```

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 file('/tmp/bar','w+') as f:
...     f.write('Hello, world')
>>> with ssh(host='localhost',
...          user='demouser',
...          password='demopass') as s:
...          print s.download_data('/tmp/bar')
Hello, world
```

download_dir(*local*, *remote=None*)
Recursively uploads a directory onto 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.

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_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 *
>>> with ssh(host='localhost',
...          user='demouser',
...          password='demopass') as s:
...      l = s.listen_remote()
...      a = remote('localhost', l.port)
...      b = l.wait_for_connection()
...      a.sendline('Hello')
...      print repr(b.recvline())
'Hello\n'
```

run(*process*, *tty = False*, *wd = None*, *env = None*, *timeout = Timeout.default*) → ssh_channel
Open a new channel with a specific process inside. If *tty* is True, then a TTY is requested on the remote server.

Return a `pwnlib.tubes.ssh.ssh_channel` object.

Examples

```
>>> with ssh(host='localhost',
...          user='demouser',
...          password='demopass') as s:
...      py = s.run('python -i')
...      _ = py.recvuntil('>>> ')
...      py.sendline('print 2+2')
...      py.sendline('exit')
...      print repr(py.recvline())
'4\n'
```

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

```
>>> with ssh(host='localhost',
...          user='demouser',
...          password='demopass') as s:
...      print s.run_to_end('echo Hello; exit 17')
('Hello\n', 17)
```

set_working_directory(*wd=None*)
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

Parameters **wd** (*string*) – Working directory. Default is to auto-generate a directory based on the result of running 'mktemp -d' on the remote machine.

Examples

```
>>> with ssh(host='localhost',
...          user='demouser',
...          password='demopass') as s:
...      cwd = s.set_working_directory()
...      print '' == s.ls()
...      print s.pwd() == cwd
True
True
```

shell(*shell = None*, *tty = False*, *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

```
>>> with ssh(host='localhost',
...          user='demouser',
...          password='demopass') as s:
...      sh = s.shell('/bin/sh')
...      sh.sendline('echo Hello; exit')
...      print 'Hello' in sh.recvall()
True
```

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.

Examoles:

```
>>> with ssh(host='localhost',
...          user='demouser',
...          password='demopass') as s:
...      s.upload_data('Hello, world', '/tmp/foo')
...      print file('/tmp/foo').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.

Args: 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.

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_connecter**
Bases: pwnlib.tubes.sock.sock

class pwnlib.tubes.ssh.**ssh_listener**
Bases: pwnlib.tubes.sock.sock

2.17.4 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__()

__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.

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('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('clean me up')
>>> t.clean(0)
'clean me up'
>>> len(t.buffer)
0
```

clean_and_log(*timeout = 0.05*)

Works exactly as pwnlib.tubes.tube.tube.clean(), but logs recieved data with pwnlib.log.info().

Returns All data received

Examples

```
>>> def recv(n, data=['', 'hooray_data']):
...     while data: return data.pop()
>>> t = tube()
>>> t.recv_raw      = recv
>>> t.connected_raw = lambda d: True
>>> t.fileno        = lambda: 1234
>>> with context.local(log_level='info'):
...     data = t.clean_and_log()
    'hooray_data'
>>> data
'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
>>> def recvone(n, data=['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 x
>>> def recvone(n, data=['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
>>> _=(a.connect_output(b), time.sleep(0.1))
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
>>> _=map(t.connected, ('any', 'in', 'read', 'recv', 'out', 'write', 'send'))
any
recv
recv
recv
send
send
send
>>> t.connected('bad_value')
Traceback (most recent call last):
...
KeyError: "direction must be in ['any', 'in', 'out', 'read', 'recv', 'send', 'write']"
Traceback (most recent call last):
...
KeyError: "direction must be in ['any', 'in', 'out', 'read', 'recv', 'send', 'write']"
```

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.

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.

newline = '\n'
Delimiter to use for sendline(), recvline(), and related functions.

recv(*numb = 4096*, *timeout = default*) → str
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 string 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: 'Hello, world'
>>> t.recv() == 'Hello, world'
True
>>> t.unrecv('Woohoo')
>>> t.recv() == 'Woohoo'
True
>>> with context.local(log_level='debug'):
...     _ = t.recv()
[...] Received 0xc bytes:
    'Hello, world'
```

recvall() → str
Receives data until EOF is reached.

recvline(*keepends = True*) → str
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: 'Foo\nBar\r\nBaz\n'
>>> t.recvline()
'Foo\n'
>>> t.recvline()
```

```
'Bar\r\n'
>>> t.recvline(keepends = False)
'Baz'
>>> t.newline = '\r\n'
>>> t.recvline(keepends = False)
'Foo\nBar'
```

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: "Hello\nWorld\nXylophone\n"
>>> t.recvline_contains('r')
'World'
>>> f = lambda n: "cat dog bird\napple pear orange\nbicycle car train\n"
>>> t = tube()
>>> t.recv_raw = f
>>> t.recvline_contains('pear')
'apple pear orange'
>>> t = tube()
>>> t.recv_raw = f
>>> t.recvline_contains(('car', 'train'))
'bicycle car train'
```

recvline_endswith(*delims*, *keepends = False*, *timeout = default*) → str
Keep recieving lines until one is found that starts with one of *delims*. Returns the last line recieved.

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: 'Foo\nBar\nBaz\nKaboodle\n'
>>> t.recvline_endswith('r')
'Bar'
>>> t.recvline_endswith(tuple('abcde'), True)
'Kaboodle\n'
>>> t.recvline_endswith('oodle')
'Kaboodle'
```

recvline_pred(*pred*, *keepends = False*) → str
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: "Foo\nBar\nBaz\n"
>>> t.recvline_pred(lambda line: line == "Bar\n")
'Bar'
>>> t.recvline_pred(lambda line: line == "Bar\n", keepends=True)
'Bar\n'
>>> t.recvline_pred(lambda line: line == 'Nope!', timeout=0.1)
''
```

recvline_regex(*regex*, *exact=False*, *keepends=False*, *timeout=pwnlib.timeout.Timeout.default*)
recvregex(regex, exact = False, keepends = False, timeout = default) -> str

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_startswith(*delims*, *keepends = False*, *timeout = default*) → str
Keep recieving lines until one is found that starts with one of *delims*. Returns the last line recieved.

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: "Hello\nWorld\nXylophone\n"
>>> t.recvline_startswith(tuple('WXYZ'))
'World'
>>> t.recvline_startswith(tuple('WXYZ'), True)
'Xylophone\n'
>>> t.recvline_startswith('Wo')
'World'
```

recvlines(*numlines*, *keepends = False*, *timeout = default*) → str list
Recieve 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: '\n'
>>> t.recvlines(3)
['', '', '']
>>> t.recv_raw = lambda n: 'Foo\nBar\nBaz\n'
>>> t.recvlines(3)
['Foo', 'Bar', 'Baz']
>>> t.recvlines(3, True)
['Foo\n', 'Bar\n', 'Baz\n']
```

recvn (*numb*, *timeout* = *default*) → str

Recieves 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 = 'hello world'
>>> t.recv_raw = lambda *a: data
>>> t.recvn(len(data)) == data
True
>>> t.recvn(len(data)+1) == data + data[0]
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 'a'
>>> t.recvn(10, timeout=0.05)
''
>>> t.recvn(10, timeout=0.05)
'aaaaaaaaaa'
```

recvpred (*pred*, *timeout* = *default*) → str

Receives one byte at a time from the tube, until `pred(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 string containing bytes received from the socket, or `''` if a timeout occurred while waiting.

recvregex (*regex*, *exact* = *False*, *timeout* = *default*) → str

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.

recvrepeat ()

Receives data until a timeout or EOF is reached.

Examples

```
>>> data = [
... 'd',
... '', # simulate timeout
... 'c',
... 'b',
... 'a',
... ]
>>> def delayrecv(n, data=data):
...     return data.pop()
>>> t = tube()
>>> t.recv_raw = delayrecv
>>> t.recvrepeat(0.2)
'abc'
>>> t.recv()
'd'
```

recvuntil (*delims*, *timeout* = *default*) → str

Recieve 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** (*str,tuple*) – String of delimiters characters, or list of delimiter 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: "Hello World!"
>>> t.recvuntil(' ')
'Hello '
>>> _=t.clean(0)
>>> # Matches on 'o' in 'Hello'
>>> t.recvuntil(tuple(' Wor'))
'Hello'
>>> _=t.clean(0)
>>> # Matches expressly full string
>>> t.recvuntil(' Wor')
'Hello Wor'
>>> _=t.clean(0)
>>> # Matches on full string, drops match
>>> t.recvuntil(' Wor', drop=True)
'Hello'

>>> # Try with regex special characters
>>> t = tube()
>>> t.recv_raw = lambda n: "Hello|World"
>>> t.recvuntil('|', drop=True)
'Hello'
```

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('hello')
'hello'
```

sendafter(*delim*, *data*, *timeout* = *default*) → str
A combination of `recvuntil(delim, 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('hello')
'hello\n'
>>> t.newline = '\r\n'
>>> t.sendline('hello')
'hello\r\n'
```

sendlineafter (*delim*, *data*, *timeout* = *default*) → str
A combination of `recvuntil(delim, timeout)` and `sendline(data)`.

sendlinethen (*delim*, *data*, *timeout* = *default*) → str
A combination of `sendline(data)` and `recvuntil(delim, timeout)`.

sendthen (*delim*, *data*, *timeout* = *default*) → str
A combination of `send(data)` and `recvuntil(delim, 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
>>> _=map(t.shutdown, ('in', 'read', 'recv', 'out', 'write', 'send'))
recv
recv
recv
send
send
send
>>> t.shutdown('bad_value')
Traceback (most recent call last):
...
KeyError: "direction must be in ['in', 'out', 'read', 'recv', 'send', 'write']"
Traceback (most recent call last):
...
KeyError: "direction must be in ['in', 'out', 'read', 'recv', 'send', 'write']"
```

shutdown_raw (*direction*)
Should not be called directly. Closes the tube for further reading or writing.

spawn_process (**args*, ***kwargs*)
Spawns a new process having this tube as stdin, stdout and stderr.

Takes the same arguments as `subprocess.Popen`.

timeout_change()
Informs the raw layer of the tube that the timeout has changed.

Should not be called directly.

Inherited from Timeout.

unrecv(*data*)
Puts the specified data back at the beginning of the receive buffer.

Examples

```
>>> t = tube()
>>> t.recv_raw = lambda n: 'hello'
>>> t.recv()
'hello'
>>> t.recv()
'hello'
>>> t.unrecv('world')
>>> t.recv()
'world'
>>> t.recv()
'hello'
```

wait_for_close()
Waits until the tube is closed.

2.18 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

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.

pwnlib.ui.**pause**(*n=None*)
Waits for either user input or a specific number of seconds.

2.19 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.

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.

Returns A random user agent string selected from `getall()`.

```
>>> import random as randommod
>>> randommod.seed(1)
>>> random()
'Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; FunWebProducts; FunWebProducts-MyTotalSearch
```

2.20 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.

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.

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('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('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('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('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('123456789')
563
```

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('123456789')
1443
```

pwnlib.util.crc.**crc_12_3gpp**(*data*) → int
Calculates the crc_12_3gpp 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-bits.12

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print crc_12_3gpp('123456789')
3503
```

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.crc-12-cdma2000

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print crc_12_cdma2000('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('123456789')
3931
```

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('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('123456789')
2093
```

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('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('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('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('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('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('123456789')
19462
```

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('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('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('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('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('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('123456789')
54862
```

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('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('123456789')
28561
```

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('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('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('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('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('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('123456789')
2215682
```

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('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('123456789')
2040760
```

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('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('123456789')
3421780262
```

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('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('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('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('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('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('123456789')
806403967
```

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-bits.3

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print crc_3_rohc('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('123456789')
910907393606
```

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-bits.4

Parameters **data** (*str*) – The data to checksum.

Example

```
>>> print crc_4_itu('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('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('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('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('123456789')
7800480153909949255
```

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('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('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('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('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('123456789')
38
```

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('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('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('123456789')
83
```

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('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('123456789')
749237524598872659187218
```

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('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('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('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('123456789')
151
```

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('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('123456789')
161
```

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('123456789')
161
```

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('123456789')
208
```

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('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('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('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('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('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('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('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('123456789')
12739
```

2.21 pwnlib.util.cyclic — Generation of unique sequences

pwnlib.util.cyclic.**cyclic**(*length* = *None*, *alphabet* = *string.ascii_lowercase*, *n* = *4*) → list/str

A simple wrapper over de_bruijn(). This function returns a 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.

Example

```
>>> cyclic(alphabet = "ABC", n = 3)
'AAABAACABBABCACBACCBBBCBCCC'
>>> cyclic(20)
'aaaabaaacaaadaaaeaaa'
>>> alphabet, n = range(30), 3
>>> len(alphabet)**n, len(cyclic(alphabet = alphabet, n = n))
(27000, 27000)
```

pwnlib.util.cyclic.**cyclic_find**(*subseq*, *alphabet* = *string.ascii_lowercase*, *n* = *None*) → int

Calculates the position of a substring into a De Bruijn sequence.

Parameters

- **subseq** – The subsequence to look for. This can either 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.
- **n** (*int*) – The length of subsequences that should be unique.

Examples

```
>>> cyclic_find(cyclic(1000)[514:518])
514
```

pwnlib.util.cyclic.**de_bruijn**(*alphabet = string.ascii_lowercase*, *n = 4*) → 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.

2.22 pwnlib.util.fiddling — Utilities bit fiddling

pwnlib.util.fiddling.**b64d**(*s*) → str
Base64 decodes a string

Example

```
>>> b64d('dGVzdA==')
'test'
```

pwnlib.util.fiddling.**b64e**(*s*) → str
Base64 encodes a string

Example

```
>>> b64e("test")
'dGVzdA=='
```

pwnlib.util.fiddling.**bits**(*s*, *endian = 'big'*, *zero = 0*, *one = 1*) → list
Converts the argument 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("test"))
17
```

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("bits_str", endian = "little")
'0100011010010110001011101100111011111010110011100010111001001110'
```

pwnlib.util.fiddling.**bitswap**(*s*) → str

Reverses the bits in every byte of a given string.

Example

```
>>> bitswap("1234")
'\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.**enhex**(*x*) → str

Hex-encodes a string.

Example

```
>>> enhex("test")
'74657374'
```

pwnlib.util.fiddling.**hexdump_iter**(*s*, *width=16*, *skip=True*, *hexii=False*, *begin=0*, *style=None*, *highlight=None*)

hexdump_iter(s, width = 16, skip = True, hexii = False, begin = 0, style = {}, highlight = []) -> str generator

Return a hexdump-dump of a string as a generator of lines.

Parameters

- **s** (*str*) – The string to dump
- **width** (*int*) – The number of characters per line
- **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.

Returns A hexdump-dump in the form of a string.

`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.`**`randoms`**(*count*, *alphabet = string.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)
'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])
'\x80'
>>> unbits([1], endian = 'little')
'\x01'
>>> unbits(bits('hello'), endian = 'little')
'\x16\xa666\xf6'
```

`pwnlib.util.fiddling.`**`unhex`**(*s*) → str
Hex-decodes a string.

Example

```
>>> unhex("74657374")
'test'
```

`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'
Traceback (most recent call last):
    ...
ValueError: Invalid input to urldecode
```

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('lol', 'hello', 42)
'. ***'
```

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("test")
('\x01\x01\x01\x01', 'udru')
```

2.23 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

2.24 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 d starts at piece *i*. `None` is equivalent to `(0, 1)`.

Returns A string *s* such that `func(s)` returns `True` or `None` if the search space was exhausted.

Example

```
>>> bruteforce(lambda x: x == 'hello', string.lowercase, length = 10)
'hello'
>>> bruteforce(lambda x: x == 'hello', 'hllo', 5) is None
True
```

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]
```

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)
>>> i.next()
5
>>> i = iter([1, 2, 3, 4, 5])
>>> consume(2, i)
>>> list(i)
[3, 4, 5]
```

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** – 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)
>>> i.next()
1
>>> i.next()
2
>>> i.next()
3
>>> i.next()
Traceback (most recent call last):
    ...
StopIteration
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, imap(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
>>> i.next()
0
>>> i = count()
>>> nth(4, i)
4
>>> i.next()
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 whoose 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

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)
>>> take(2, i)
Traceback (most recent call last):
    ...
TypeError: f() takes exactly 0 arguments (1 given)
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.

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]
>>> 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.**ifilter**()
Alias for itertools.ifilter()

pwnlib.util.iters.**ifilterfalse**()
Alias for itertools.ifilterfalse()

pwnlib.util.iters.**imap**()
Alias for itertools.imap()

pwnlib.util.iters.**islice**()
Alias for itertools.islice()

pwnlib.util.iters.**izip**()
Alias for itertools.izip()

pwnlib.util.iters.**izip_longest**()
Alias for itertools.izip_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()

2.25 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)
>>> foo.next()
3
>>> foo.next()
4
>>> foo.next()
6
```

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]]
```

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]]
```

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'
```

2.26 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")
'\x7f\x00\x00\x01'
```

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)
...     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/l
```

pwnlib.util.misc.**read**(*path*) → str
Open file, return content.

Examples

```
>>> read('pwnlib/util/misc.py').split('\n')[0]
'import socket, re, os, stat, errno, string, base64, logging'
```

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', '
>>> sizes
{'ch': 8, 'cl': 8, 'ah': 8, 'edi': 32, 'al': 8, 'cx': 16, 'ebp': 32, 'ax': 16, 'edx': 32, 'ebx':
>>> bigger
{'ch': ['ecx', 'cx', 'ch'], 'cl': ['ecx', 'cx', 'cl'], 'ah': ['eax', 'ax', 'ah'], 'edi': ['edi']
>>> smaller
{'ch': [], 'cl': [], 'ah': [], 'edi': ['di'], 'al': [], 'cx': ['cl', 'ch'], 'ebp': ['bp'], 'ax':
```

pwnlib.util.misc.**run_in_new_terminal**(*command*, *terminal* = *None*) → None

Run a command in a new terminal.

If X11 is detected, the terminal will be launched with `x-terminal-emulator`.

If X11 is not detected, a new tmux pane is opened if possible.

Parameters

- **command** (*str*) – The command to run.
- **terminal** (*str*) – Which terminal to use.
- **args** (*list*) – Arguments to pass to the terminal

Returns None

pwnlib.util.misc.**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.

Examples

```
>>> print sh_string('foobar')
foobar
>>> print sh_string('foo bar')
'foo bar'
>>> print sh_string("foo'bar")
"foo'bar"
>>> print sh_string("foo\\bar")
'foo\bar'
>>> print sh_string("foo\\'bar")
"foo\\'bar"
>>> print sh_string("foo\x01'bar")
"$( (echo Zm9vASdiYXI=|(base64 -d||openssl enc -d -base64)||echo -en 'foo\x01\x27bar') 2>/dev/nu
>>> print subprocess.check_output("echo -n " + sh_string("foo\\'bar"), shell = True)
foo\'bar
```

pwnlib.util.misc.**size**(*n*, *abbriv* = *'B'*, *si* = *False*) → str

Convert the length of a bytestream to human readable form.

Parameters **n** (*int,str*) – The length to convert to human readable form

Example

```
>>> size(451)
'451B'
>>> size(1000)
'1000B'
>>> size(1024)
'1.00KB'
>>> 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']
```

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 occurence or `None` is returned.

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')
'/bin/sh'
```

pwnlib.util.misc.**write**(*path*, *data=''*, *create_dir=False*)
Create new file or truncate existing to zero length and write data.

2.27 pwnlib.util.net — Networking interfaces

pwnlib.util.net.**getifaddrs**() → dict list
A wrapper for libc's `getifaddrs`.

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.

`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.

2.28 `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)
'\x00'
>>> p32(0xdeadbeef)
'\xef\xbe\xad\xde'
>>> p32(0xdeadbeef, endian='big')
'\xde\xad\xbe\xef'
>>> with context.local(endian='big'): p32(0xdeadbeef)
'\xde\xad\xbe\xef'
```

Make a frozen packer, which does not change with context.

```
>>> p=make_packer('all')
>>> p(0xff)
'\xff'
>>> p(0x1ff)
'\xff\x01'
>>> with context.local(endian='big'): print repr(p(0x1ff))
'\xff\x01'
```

`pwnlib.util.packing.`**`flat`**(**args*, *preprocessor = None*, *word_size = None*, *endianness = None*, *sign = None*)

Flattens the arguments into a string.

This function takes an arbitrary number of arbitrarily nested lists and tuples. 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.

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 uded.
- **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

```
>>> flat(1, "test", [["AB"]*2]*3], endianness = 'little', word_size = 16, sign = False)
'\x01\x00testABABABABABAB'
>>> flat([1, [2, 3]], preprocessor = lambda x: str(x+1))
'234'
```

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.
- **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, 'little', 'unsigned')
>>> p
<function _p32lu at 0x...>
>>> p(42)
'*\x00\x00\x00'
>>> p(-1)
Traceback (most recent call last):
    ...
error: integer out of range for 'I' format code
>>> make_packer(33, 'little', 'unsigned')
<function <lambda> at 0x...>
Traceback (most recent call last):
    ...
error: integer out of range for 'I' format code
```

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.
- **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, 'little', 'unsigned')
>>> u
<function _u32lu at 0x...>
>>> hex(u('/bin'))
'0x6e69622f'
>>> u('abcde')
Traceback (most recent call last):
    ...
error: unpack requires a string argument of length 4
>>> make_unpacker(33, 'little', 'unsigned')
<function <lambda> at 0x...>
Traceback (most recent call last):
    ...
error: unpack requires a string argument of length 4
```

pwnlib.util.packing.**p16**(*number*, ***kwargs*) → str
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 string

pwnlib.util.packing.**p32**(*number*, ***kwargs*) → str
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 string

pwnlib.util.packing.**p64**(*number*, ***kwargs*) → str
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 string

pwnlib.util.packing.**p8**(*number*, ***kwargs*) → str
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 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’.
- **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)
'ABC'
>>> pack(0x414243, 24, 'little', True)
'CBA'
>>> pack(0x814243, 24, 'big', False)
'\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)
'\x00\x81BC'
>>> pack(-1, 'all', 'little', True)
'\xff'
>>> pack(-256, 'all', 'big', True)
'\xff\x00'
>>> pack(0x0102030405, 'all', 'little', True)
'\x05\x04\x03\x02\x01'
Traceback (most recent call last):
    ...
ValueError: pack(): number does not fit within word_size
```

pwnlib.util.packing.**routine**(*number*, *endianness=None*, *sign=None*, ***kwargs*)
u32(number, **kwargs) -> int

Unpacks an 32-bit integer

Parameters

- **data** (*str*) – 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.**u16**(*number*, ***kwargs*) → int
Unpacks an 16-bit integer

Parameters

- **data** (*str*) – 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*, ***kwargs*) → int
Unpacks an 32-bit integer

Parameters

- **data** (*str*) – 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*, ***kwargs*) → int
Unpacks an 64-bit integer

Parameters

- **data** (*str*) – 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*, ***kwargs*) → int
Unpacks an 8-bit integer

Parameters

- **data** (*str*) – 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”.
- **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('\xaa\x55', 16, 'little', False))
'0x55aa'
>>> hex(unpack('\xaa\x55', 16, 'big', False))
'0xaa55'
>>> hex(unpack('\xaa\x55', 16, 'big', True))
'-0x55ab'
>>> hex(unpack('\xaa\x55', 15, 'big', True))
'0x2a55'
>>> hex(unpack('\xff\x02\x03', 'all', 'little', True))
'0x302ff'
>>> hex(unpack('\xff\x02\x03', 'all', 'big', True))
'-0xfdfd'
```

pwnlib.util.packing.**unpack_many**(*data*, *word_size=None*, *endianness=None*, *sign=None*, ***kwargs*)

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.

Parameters

- **number** (*int*) – String to convert
- **word_size** (*int*) – Word size of the converted integers or the string "all".
- **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

```
>>> map(hex, unpack_many('\xaa\x55\xcc\x33', 16, 'little', False))
['0x55aa', '0x33cc']
>>> map(hex, unpack_many('\xaa\x55\xcc\x33', 16, 'big', False))
['0xaa55', '0xcc33']
>>> map(hex, unpack_many('\xaa\x55\xcc\x33', 16, 'big', True))
['-0x55ab', '-0x33cd']
>>> map(hex, unpack_many('\xff\x02\x03', 'all', 'little', True))
['0x302ff']
>>> map(hex, unpack_many('\xff\x02\x03', 'all', 'big', True))
['-0xfdfd']
```

2.29 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*.

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`.

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.

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.

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.

pwnlib.util.proc.**name**(*pid*) → str

Parameters **pid** (*int*) – PID of the process.

Returns Name of process as listed in `/proc/<pid>/status`.

Example

```
>>> name(os.getpid()) == os.path.basename(sys.argv[0])
True
```

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.

pwnlib.util.proc.**starttime**(*pid*) → float

Parameters **pid** (*int*) – PID of the process.

Returns The time (in seconds) the process started after system boot

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.

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*) → None

Sleeps until the process with PID *pid* is being traced.

Parameters **pid** (*int*) – PID of the process.

Returns None

2.30 pwnlib.util.safeeval — Safe evaluation of python code

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.

```
>>> 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
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.

```
>>> 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
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

2.31 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    = 'http://httpbin.org/robots.txt'
>>> with context.local(log_level='ERROR'):
...     result = wget(url)
>>> result
'User-agent: *\nDisallow: /deny\n'
>>> with context.local(log_level='ERROR'):
...     _ = wget(url, True)
>>> result == file('robots.txt').read()
True
```

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

p

Symbols

A

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X