pwny Documentation

Release 0.6.0

Author

April 14, 2015

Contents

pwnypack is the official CTF toolkit of Certified Edible Dinosaurs. It aims to provide a set of command line utilities and a python library that are useful when playing hacking CTFs.

The core functionality of *pwnypack* is defined in the modules of the `pwnypack` package. The `pwny` package imports all that functionality into a single namespace for convenience.

Some of the functionality of the `pwnypack` package is also exported through a set of commandline utilities. Run `pwny help` after installing *pwnypack* to get a list of available utilities. You can create convenience symlinks for all the included apps by running `pwny symlink`. Each app has a help function that is accessible using the `-h` parameter.

For some example of how to use *pwnypack*, check the write-ups on the official Certified Edible Dinosaurs website.

Package contents:

CHAPTER 1

pwny package

The `pwny` package provides a convence metapackage that imports the entire public API of *pwnypack* into a single namespace:

```
>>> from pwny import *
>>> enhex(asm('mov rax, 0xced', target=Target(arch=Architecture.x86_64)))
u'b8ed0c0000'
```

For details about what exactly is made available, please consult the documentation of the individual pwnypack modules.

CHAPTER 2

pwnypack package

All the functionality of *pwnypack* is implemented in the modules of this package.

2.1 pwnypack.asm – (Dis)assembler

This module contains functions to assemble and disassemble code for a given target platform.

Currently, the only supported architecture is `x86` (both 32 and 64 bits variants). Assembly is performed by the *nasm* assembler (only supports `nasm` syntax). Disassembly is performed by *ndisasm* (`nasm` syntax) or *capstone* (`intel` & `att` syntax).

class pwnypack.asm.**AsmSyntax**
Bases: enum.IntEnum

This enumeration is used to specify the assembler syntax.

pwnypack.asm.**asm**(*code*, *addr=0*, *syntax=AsmSyntax.nasm*, *target=None*)
Assemble statements into machine readable code.

Parameters

- **code** (*str*) – The statements to assemble.
- **addr** (*int*) – The memory address where the code will run.
- **syntax** (AsmSyntax) – The input assembler syntax.
- **target** (Target) – The target architecture. The global target is used if this argument is `None`.

Returns The assembled machine code.

Return type bytes

Raises

- `SyntaxError` – If the assembler statements are invalid.
- `NotImplementedError` – In an unsupported target platform is specified.

Example

```
>>> from pwny import *
>>> asm('''
...     pop rdi
...     ret
... ''', target=Target(arch=Target.Arch.x86, bits=64))
b'_\xc3'
```

pwnypack.asm.**disasm**(*code*, *addr=0*, *syntax=AsmSyntax.nasm*, *target=None*)

Disassemble machine readable code into human readable statements.

Parameters

- **code** (*bytes*) – The machine code that is to be disassembled.
- **addr** (*int*) – The memory address of the code (used for relative references).
- **syntax** (AsmSyntax) – The output assembler syntax.
- **target** (Target) – The architecture for which the code was written. The global target is used if this argument is `None`.

Returns The disassembled machine code.

Return type list of str

Raises

- `NotImplementedError` – In an unsupported target platform is specified.
- `RuntimeError` – If ndisasm encounters an error.

Example

```
>>> from pwny import *
>>> disasm(b'_\xc3', target=Target(arch=Target.Arch.x86, bits=64))
['pop rdi', 'ret']
```

2.2 pwnypack.codec – Data transformation

This module contains functions that allow you to manipulate, encode or decode strings and byte sequences.

pwnypack.codec.**xor**(*key*, *data*)

Perform cyclical exclusive or operations on `data`.

The `key` can be a an integer *(0 <= key < 256)* or a byte sequence. If the key is smaller than the provided `data`, the `key` will be repeated.

Parameters

- **key** (*int or bytes*) – The key to xor `data` with.
- **data** (*bytes*) – The data to perform the xor operation on.

Returns The result of the exclusive or operation.

Return type bytes

Examples

```
>>> from pwny import *
>>> xor(5, b'ABCD')
b'DGFA'
>>> xor(5, b'DGFA')
b'ABCD'
>>> xor(b'pwny', b'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
b'15-=51)19=%5=9!)!%=-%!9!)-'
>>> xor(b'pwny', b'15-=51)19=%5=9!)!%=-%!9!)-')
b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

pwnypack.codec.**rot13**(*d*)

Rotate all characters in the alphabets A-Z and a-z by 13 positions in the alphabet. This is a *caesar()* shift of 13 along the fixed alphabets `A-Z` and `a-z`.

Parameters **d** (*str*) – The string to the apply the cipher to.

Returns The string with the rot13 cipher applied.

Return type str

Examples

```
>>> rot13('whax')
'junk'
>>> rot13('junk')
'whax'
```

pwnypack.codec.**caesar**(*shift*, *data*, *shift_ranges=('az', 'AZ')*)

Apply a caesar cipher to a string.

The caesar cipher is a substition cipher where each letter in the given alphabet is replaced by a letter some fixed number down the alphabet.

If `shift` is `1`, *A* will become *B*, *B* will become *C*, etc...

You can define the alphabets that will be shift by specifying one or more shift ranges. The characters will than be shifted within the given ranges.

Parameters

- **shift** (*int*) – The shift to apply.
- **data** (*str*) – The string to apply the cipher to.
- **shift_ranges** (*list of str*) – Which alphabets to shift.

Returns The string with the caesar cipher applied.

Return type str

Examples

```
>>> caesar(16, 'Pwnypack')
'Fmdofqsa'
>>> caesar(-16, 'Fmdofqsa')
'Pwnypack'
>>> caesar(16, 'PWNYpack', shift_ranges=('AZ',))
```

```
    'FMDOpack'
    >>> caesar(16, 'PWNYpack', shift_ranges=('Az',))
    '`g^iFqsA'
```

pwnypack.codec.**enhex**(*d*, *separator=''*)

Convert bytes to their hexadecimal representation, optionally joined by a given separator.

Parameters

- **d** (*bytes*) – The data to convert to hexadecimal representation.
- **separator** (*str*) – The separator to insert between hexadecimal tuples.

Returns The hexadecimal representation of d.

Return type str

Examples

```
    >>> from pwny import *
    >>> enhex(b'pwnypack')
    '70776e797061636b'
    >>> enhex(b'pwnypack', separator=' ')
    '70 77 6e 79 70 61 63 6b'
```

pwnypack.codec.**dehex**(*d*)

Convert a hexadecimal representation of a byte sequence to bytes. All non-hexadecimal characters will be removed from the input.

Parameters **d** (*str*) – The string of hexadecimal tuples.

Returns The byte sequence represented by d.

Return type bytes

Examples

```
    >>> from pwny import *
    >>> dehex('70776e797061636b')
    b'pwnypack'
    >>> dhex('70 77 6e 79 70 61 63 6b')
    b'pwnypack'
```

pwnypack.codec.**enb64**(*d*)

Convert bytes to their base64 representation.

Parameters **d** (*bytes*) – The data to convert to its base64 representation.

Returns The base64 representation of d.

Return type str

Example

```
    >>> from pwny import *
    >>> enb64(b'pwnypack')
    'cHdueXBhY2s='
```

pwnypack.codec.**deb64**(*d*)
Convert a base64 representation back to its original bytes.

Parameters **d** (*str*) – The base64 representation to decode.

Returns The bytes represented by d.

Return type bytes

Example

```
>>> from pwny import *
>>> deb64('cHdueXBhY2s=')
b'pwnypack'
```

pwnypack.codec.**frequency**(*v*)
Perform a frequency analysis on a byte sequence or string.

Parameters **d** (*bytes or str*) – The sequence to analyse.

Returns A dictionary of unique elements in d and how often the occur.

Return type dict

Example

```
>>> frequency('pwnypack')
{'a': 1, 'c': 1, 'k': 1, 'n': 1, 'p': 2, 'w': 1, 'y': 1}
```

2.3 pwnypack.elf – ELF file parsing

This module contains a parser for, and methods to extract information from ELF files.

class pwnypack.elf.**ELF**(*f=None*)
Bases: *pwnypack.target.Target*

A parser for ELF files. Upon parsing the ELF headers, it will not only fill the ELF specific fields but will also populate the inherited *arch*, *bits* and *endian* properties based on the values it encounters.

Parameters **f** (str, file or None) – The (path to) the ELF file to parse.

Example

```
>>> from pwny import *
>>> e = ELF('my-executable')
>>> print(e.machine)
>>> print(e.program_headers)
>>> print(e.section_headers)
>>> print(e.symbols)
```

class Machine
Bases: enum.IntEnum

The target machine architecture.

class ELF.**OSABI**
Bases: enum.IntEnum

Describes the OS- or ABI-specific ELF extensions used by this file.

class ELF.**ProgramHeader**(*elf*, *data*)
Bases: object

Describes how the loader will load a part of a file. Called by the *ELF* class.

Parameters

- **elf** (ELF) – The ELF instance owning this program header.
- **data** – The content of the program header entry.

class Flags
Bases: enum.IntEnum

The individual flags that make up *ELF.ProgramHeader.flags*.

class ELF.ProgramHeader.**Type**
Bases: enum.IntEnum

The segment type.

ELF.ProgramHeader.**align = None**

ELF.ProgramHeader.**filesz = None**

ELF.ProgramHeader.**flags = None**

ELF.ProgramHeader.**memsz = None**

ELF.ProgramHeader.**offset = None**

ELF.ProgramHeader.**paddr = None**

ELF.ProgramHeader.**type = None**

ELF.ProgramHeader.**type_id = None**

ELF.ProgramHeader.**vaddr = None**

class ELF.**SectionHeader**(*elf*, *data*)
Bases: object

Describes a section of an ELF file. Called by the *ELF* class.

Parameters

- **elf** (ELF) – The ELF instance owning this section header.
- **data** – The content of the section header entry.

class Flags
Bases: enum.IntEnum

class ELF.SectionHeader.**Type**
Bases: enum.IntEnum

Describes the section's type

ELF.SectionHeader.**addr = None**

ELF.SectionHeader.**addralign = None**

ELF.SectionHeader.**content**
The contents of this section.

ELF.SectionHeader.**elf** = None

ELF.SectionHeader.**entsize** = None

ELF.SectionHeader.**flags** = None

ELF.SectionHeader.**info** = None

ELF.SectionHeader.**link** = None

ELF.SectionHeader.**name** = None

ELF.SectionHeader.**name_index** = None

ELF.SectionHeader.**offset** = None

ELF.SectionHeader.**size** = None

ELF.SectionHeader.**type** = None

ELF.SectionHeader.**type_id** = None

class ELF.**Symbol**(*elf*, *data*, *strs*)
Bases: object

Contains information about symbols. Called by the *ELF* class.

Parameters

- **elf** (ELF) – The ELF instance owning this symbol.
- **data** – The content of the symbol definition.
- **strs** – The content of the string section associated with the symbol table.

class Binding
Bases: enum.IntEnum

Describes a symbol's binding.

class ELF.Symbol.**SpecialSection**
Bases: enum.IntEnum

Special section types.

class ELF.Symbol.**Type**
Bases: enum.IntEnum

Describes the symbol's type.

class ELF.Symbol.**Visibility**
Bases: enum.IntEnum

Describes the symbol's visibility.

ELF.Symbol.**content**
The contents of a symbol.
Raises TypeError – If the symbol isn't defined until runtime.

ELF.Symbol.**elf** = None

ELF.Symbol.**info** = None

ELF.Symbol.**name** = None

ELF.Symbol.**name_index** = None

ELF.Symbol.**other** = None

ELF.Symbol.**shndx** = None

ELF.Symbol.**size** = None

ELF.Symbol.**type** = None

ELF.Symbol.**type_id** = None

ELF.Symbol.**value** = None

ELF.Symbol.**visibility** = None

class ELF.**Type**
Bases: enum.IntEnum

Describes the object type.

ELF.**abi_version** = None

ELF.**entry** = None

ELF.**f** = None

ELF.**flags** = None

ELF.**get_program_header**(*index*)
Return a specific program header by its index.

Parameters **index** (*int*) – The program header index.

Returns *~ELF.ProgramHeader*: The program header.

Return type :class

Raises KeyError – The specified index does not exist.

ELF.**get_section_header**(*section*)
Get a specific section header by index or name.

Parameters **section** (*int or str*) – The index or name of the section header to return.

Returns *~ELF.SectionHeader*: The section header.

Return type :class

Raises KeyError – The requested section header does not exist.

ELF.**get_symbol**(*symbol*)
Get a specific symbol by index or name.

Parameters **symbol** (*int or str*) – The index or name of the symbol to return.

Returns The symbol.

Return type *ELF.Symbol*

Raises KeyError – The requested symbol does not exist.

ELF.**hsize** = None

ELF.**machine** = None

ELF.**osabi** = None

ELF.**parse_file**(*f*)
Parse an ELF file and fill the class' properties.

Parameters **f** (*file or str*) – The (path to) the ELF file to read.

ELF.**phentsize** = None

ELF.**phnum** = None

ELF.**phoff** = None

ELF.**program_headers**
A list of all program headers.

ELF.**section_headers**
Return the list of section headers.

ELF.**shentsize** = None

ELF.**shnum** = None

ELF.**shoff** = None

ELF.**shstrndx** = None

ELF.**symbols**
Return a list of all symbols.

ELF.**type** = None

2.4 pwnypack.flow – Communication

The Flow module lets you connect to processes or network services using a unified API. It is primarily designed for synchronous communication flows.

It is based around the central *Flow* class which uses a Channel to connect to a process. The *Flow* class then uses the primitives exposed by the Channel to provide a high level API for reading/receiving and writing/sending data.

Examples

```
>>> from pwny import *
>>> f = Flow.connect_tcp('ced.pwned.systems', 80)
>>> f.writelines([
...     b'GET / HTTP/1.0',
...     b'Host: ced.pwned.systems',
...     b'',
... ])
>>> line = f.readline().strip()
>>> print(line == b'HTTP/1.0 200 OK')
True
>>> f.until(b'\r\n\r\n')
>>> f.read_eof(echo=True)
... lots of html ...
```

```
>>> from pwny import *
>>> f = Flow.execute('cat')
>>> f.writeline(b'hello')
>>> f.readline(echo=True)
```

class pwnypack.flow.**ProcessChannel**(*executable*, *argument...*, *redirect_stderr=False*)
Bases: object

This channel type allows controlling processes. It uses python's subprocess.Popen class to execute a process and allows you to communicate with it.

Parameters

- **executable** (*str*) – The executable to start.
- **argument...** (*list of str*) – The arguments to pass to the executable.
- **redirect_stderr** (*bool*) – Whether to also capture the output of stderr.

close()
Wait for the subprocess to exit.

kill()
Terminate the subprocess.

read(*n*)
Read *n* bytes from the subprocess' output channel.

Parameters **n** (*int*) – The number of bytes to read.

Returns *n* bytes of output.

Return type bytes

Raises EOFError – If the process exited.

write(*data*)
Write *n* bytes to the subprocess' input channel.

Parameters **data** (*bytes*) – The data to write.

Raises EOFError – If the process exited.

class pwnypack.flow.**SocketChannel**(*sock*)
Bases: object

This channel type allows controlling sockets.

Parameters **socket** (*socket.socket*) – The (already connected) socket to control.

close()
Close the socket gracefully.

kill()
Shut down the socket immediately.

read(*n*)
Receive *n* bytes from the socket.

Parameters **n** (*int*) – The number of bytes to read.

Returns *n* bytes read from the socket.

Return type bytes

Raises EOFError – If the socket was closed.

write(*data*)
Send *n* bytes to socket.

Parameters **data** (*bytes*) – The data to send.

Raises EOFError – If the socket was closed.

class pwnypack.flow.**TCPSocketChannel**(*host*, *port*)
Bases: *pwnypack.flow.SocketChannel*

Convenience subclass of *SocketChannel* that allows you to connect to a TCP hostname / port pair easily.

Parameters

- **host** (*str*) – The hostname or IP address to connect to.
- **port** (*int*) – The port number to connect to.

class pwnypack.flow.**Flow**(*channel*, *echo=False*)
Bases: object

The core class of *Flow*. Takes a channel and exposes synchronous utility functions for communications.

Usually, you'll use the convenience classmethods *connect_tcp()* or *execute()* instead of manually creating the constructor directly.

Parameters

- **channel** (Channel) – A channel.
- **echo** (*bool*) – Whether or not to echo all input / output.

close()
Gracefully close the channel.

classmethod connect_tcp(*host*, *port*, *echo=False*)
Set up a *TCPSocketChannel* and create a *Flow* instance for it.

Parameters

- **host** (*str*) – The hostname or IP address to connect to.
- **port** (*int*) – The port number to connect to.
- **echo** (*bool*) – Whether to echo read/written data to stdout by default.

Returns *Flow*: A Flow instance initialised with the TCP socket channel.

Return type :class

classmethod execute(*executable*, **arguments*, ***kwargs*)
execute(executable, argument..., redirect_stderr=False, echo=False):

Set up a *ProcessChannel* and create a *Flow* instance for it.

Parameters

- **executable** (*str*) – The executable to start.
- **argument...** (*list of str*) – The arguments to pass to the executable.
- **redirect_stderr** (*bool*) – Whether to also capture the output of stderr.
- **echo** (*bool*) – Whether to echo read/written data to stdout by default.

Returns *Flow*: A Flow instance initialised with the process channel.

Return type :class

kill()
Terminate the channel immediately.

read(*n*, *echo=None*)
Read *n* bytes from the channel.

Parameters

- **n** (*int*) – The number of bytes to read from the channel.
- **echo** (*bool*) – Whether to write the read data to stdout.

Returns *n* bytes of data.

Return type bytes

Raises EOFError – If the channel was closed.

read_eof(*echo=None*)
Read until the channel is closed.

Parameters **echo** (*bool*) – Whether to write the read data to stdout.

Returns The read data.

Return type bytes

read_until(*s*, *echo=None*)
Read until a certain string is encountered..

Parameters

- **s** (*bytes*) – The string to wait for.
- **echo** (*bool*) – Whether to write the read data to stdout.

Returns The data up to and including *s*.

Return type bytes

Raises EOFError – If the channel was closed.

readline(*echo=None*)
Read 1 line from channel.

Parameters **echo** (*bool*) – Whether to write the read data to stdout.

Returns The read line which includes new line character.

Return type bytes

Raises EOFError – If the channel was closed before a line was read.

readlines(*n*, *echo=None*)
Read *n* lines from channel.

Parameters

- **n** (*int*) – The number of lines to read.
- **echo** (*bool*) – Whether to write the read data to stdout.

Returns *n* lines which include new line characters.

Return type list of bytes

Raises EOFError – If the channel was closed before *n* lines were read.

until(*s*, *echo=None*)
Read until a certain string is encountered..

Parameters

- **s** (*bytes*) – The string to wait for.
- **echo** (*bool*) – Whether to write the read data to stdout.

Returns The data up to and including *s*.

Return type bytes

Raises EOFError – If the channel was closed.

write(*data*, *echo=None*)
Write data to channel.

Parameters

- **data** (*bytes*) – The data to write to the channel.
- **echo** (*bool*) – Whether to echo the written data to stdout.

Raises `EOFError` – If the channel was closed before all data was sent.

writeline(*line=b''*, *echo=None*)
Write a byte sequences to the channel and terminate it with carriage return and line feed.

Parameters

- **line** (*bytes*) – The line to send.
- **echo** (*bool*) – Whether to echo the written data to stdout.

Raises `EOFError` – If the channel was closed before all data was sent.

writelines(*lines*, *echo=None*)
Write a list of byte sequences to the channel and terminate them with carriage return and line feed.

Parameters

- **lines** (*list of bytes*) – The lines to send.
- **echo** (*bool*) – Whether to echo the written data to stdout.

Raises `EOFError` – If the channel was closed before all data was sent.

2.5 pwnypack.packing – Data (un)packing

pwnypack.packing.**pack**(*fmt*, *v1*, *v2*, ..., *endian=None*, *target=None*)
Return a string containing the values v1, v2, ... packed according to the given format. The actual packing is performed by `struct.pack` but the byte order will be set according to the given *endian*, *target* or byte order of the global target.

Parameters

- **fmt** (*str*) – The format string.
- **v1,v2,...** – The values to pack.
- **endian** (*Endian*) – Override the default byte order. If `None`, it will look at the byte order of the `target` argument.
- **target** (*Target*) – Override the default byte order. If `None`, it will look at the byte order of the global `target`.

Returns The provided values packed according to the format.

Return type bytes

pwnypack.packing.**unpack**(*fmt*, *data*, *endian=None*, *target=None*)
Unpack the string (presumably packed by pack(fmt, ...)) according to the given format. The actual unpacking is performed by `struct.unpack` but the byte order will be set according to the given *endian*, *target* or byte order of the global target.

Parameters

- **fmt** (*str*) – The format string.

- **data** (*bytes*) – The data to unpack.
- **endian** (*Endian*) – Override the default byte order. If `None`, it will look at the byte order of the `target` argument.
- **target** (*Target*) – Override the default byte order. If `None`, it will look at the byte order of the global `target`.

Returns The unpacked values according to the format.

Return type list

`pwnypack.packing.pack_size`(*fmt*, *endian=None*, *target=None*)

`pwnypack.packing.P`(*value*, *bits=None*, *endian=None*, *target=None*)

Pack an unsigned pointer for a given target.

Parameters

- **value** (*int*) – The value to pack.
- **bits** (*Bits*) – Override the default word size. If `None` it will look at the word size of `target`.
- **endian** (*Endian*) – Override the default byte order. If `None`, it will look at the byte order of the `target` argument.
- **target** (*Target*) – Override the default byte order. If `None`, it will look at the byte order of the global `target`.

`pwnypack.packing.p`(*value*, *bits=None*, *endian=None*, *target=None*)

Pack a signed pointer for a given target.

Parameters

- **value** (*int*) – The value to pack.
- **bits** (*pwnypack.target.Target.Bits*) – Override the default word size. If `None` it will look at the word size of `target`.
- **endian** (*Endian*) – Override the default byte order. If `None`, it will look at the byte order of the `target` argument.
- **target** (*Target*) – Override the default byte order. If `None`, it will look at the byte order of the global `target`.

`pwnypack.packing.U`(*data*, *bits=None*, *endian=None*, *target=None*)

Unpack an unsigned pointer for a given target.

Parameters

- **data** (*bytes*) – The data to unpack.
- **bits** (*pwnypack.target.Target.Bits*) – Override the default word size. If `None` it will look at the word size of `target`.
- **endian** (*Endian*) – Override the default byte order. If `None`, it will look at the byte order of the `target` argument.
- **target** (*Target*) – Override the default byte order. If `None`, it will look at the byte order of the global `target`.

Returns The pointer value.

Return type int

pwnypack.packing.**u**(*data*, *bits=None*, *endian=None*, *target=None*)
Unpack a signed pointer for a given target.

Parameters

- **data** (*bytes*) – The data to unpack.
- **bits** (*pwnypack.target.Target.Bits*) – Override the default word size. If `None` it will look at the word size of `target`.
- **endian** (*Endian*) – Override the default byte order. If `None`, it will look at the byte order of the `target` argument.
- **target** (*Target*) – Override the default byte order. If `None`, it will look at the byte order of the global `target`.

Returns The pointer value.

Return type int

pwnypack.packing.**p8**(*value*, *endian=None*, *target=None*)
Pack signed 8 bit integer. Alias for `pack('b', ...)`.

pwnypack.packing.**P8**(*value*, *endian=None*, *target=None*)
Pack unsigned 8 bit integer. Alias for `pack('B', ...)`.

pwnypack.packing.**u8**(*data*, *endian=None*, *target=None*)
Unpack signed 8 bit integer. Alias for `unpack('b', ...)`.

pwnypack.packing.**U8**(*data*, *endian=None*, *target=None*)
Unpack unsigned 8 bit integer. Alias for `unpack('B', ...)`.

pwnypack.packing.**p16**(*value*, *endian=None*, *target=None*)
Pack signed 16 bit integer. Alias for `pack('h', ...)`.

pwnypack.packing.**P16**(*value*, *endian=None*, *target=None*)
Pack unsigned 16 bit integer. Alias for `pack('H', ...)`.

pwnypack.packing.**u16**(*data*, *endian=None*, *target=None*)
Unpack signed 16 bit integer. Alias for `unpack('h', ...)`.

pwnypack.packing.**U16**(*data*, *endian=None*, *target=None*)
Unpack unsigned 16 bit integer. Alias for `unpack('H', ...)`.

pwnypack.packing.**p32**(*value*, *endian=None*, *target=None*)
Pack signed 32 bit integer. Alias for `pack('l', ...)`.

pwnypack.packing.**P32**(*value*, *endian=None*, *target=None*)
Pack unsigned 32 bit integer. Alias for `pack('L', ...)`.

pwnypack.packing.**u32**(*data*, *endian=None*, *target=None*)
Unpack signed 32 bit integer. Alias for `unpack('l', ...)`.

pwnypack.packing.**U32**(*data*, *endian=None*, *target=None*)
Unpack unsigned 32 bit integer. Alias for `unpack('L', ...)`.

pwnypack.packing.**p64**(*value*, *endian=None*, *target=None*)
Pack signed 64 bit integer. Alias for `pack('q', ...)`.

pwnypack.packing.**P64**(*value*, *endian=None*, *target=None*)
Pack unsigned 64 bit integer. Alias for `pack('Q', ...)`.

pwnypack.packing.**u64**(*data*, *endian=None*, *target=None*)
Unpack signed 64 bit integer. Alias for `unpack('q', ...)`.

pwnypack.packing.**U64**(*data*, *endian=None*, *target=None*)
Unpack unsigned 64 bit integer. Alias for `unpack('Q', ...)`.

2.6 pwnypack.rop – ROP gadgets

The ROP module contains a function to find gadgets in ELF binaries that can be used to create ROP chains.

pwnypack.rop.**find_gadget**(*elf*, *gadget*, *align=1*, *unique=True*)
Find a ROP gadget in a the executable sections of an ELF executable or library. The ROP gadget can be either a set of bytes for an exact match or a (bytes) regular expression. Once it finds gadgets, it uses the capstone engine to verify if the gadget consists of valid instructions and doesn't contain any call or jump instructions.

Parameters

- **elf** (*ELF*) – The ELF instance to find a gadget in.
- **gadget** (*bytes or regexp*) – The gadget to find.
- **align** (*int*) – Make sure the gadget starts at a multiple of this number
- **unique** (*bool*) – If true, only unique gadgets are returned.

Returns

A dictionary containing a description of the found gadget. Contains the following fields:

- section: The section the gadget was found in.
- offset: The offset inside the segment the gadget was found at.
- addr: The virtual memory address the gadget will be located at.
- gadget: The machine code of the found gadget.
- asm: A list of disassembled instructions.

Return type dict

2.7 pwnypack.target – Target definition

The *Target* class describes the architecture of a targeted machine, executable or environment. It encodes the generic architecture, the word size, the byte order and an architecture dependant mode.

It is used throughout *pwnypack* to determine how data should be interpreted or emitted.

class pwnypack.target.**Target**(*arch=None*, *bits=None*, *endian=None*, *mode=0*)
Bases: object

class Arch
Bases: enum.Enum

Describes the general architecture of a target.

class Target.**Bits**
Bases: enum.IntEnum

The target architecture's word size.

class Target.**Endian**
Bases: enum.IntEnum

The target architecture's byte order.

class Target.**Mode**
Bases: enum.IntEnum

Architecture dependant mode flags.

Target.**arch**
The target's architecture. One of *Target.Arch*.

Target.**assume**(*other*)
Assume the identity of another target. This can be useful to make the global target assume the identity of an ELF executable.

Parameters **other** (*Target*) – The target whose identity to assume.

Example

```
>>> from pwny import *
>>> target.assume(ELF('my-executable'))
```

Target.**bits**
The target architecture word size. One of *Target.Bits*.

Target.**endian**
The target architecture byte order. One of *Target.Endian*.

Target.**mode**
The target architecture dependant flags. OR'ed values of *Target.Mode*.

2.8 pwnypack.util – Utility functions

The util module contains various utility functions.

pwnypack.util.**cycle**(*length*, *width=4*)
Generate a de Bruijn sequence of a given length (and width). A de Bruijn sequence is a set of varying repetitions where each sequence of *n* characters is unique within the sequence. This type of sequence can be used to easily find the offset to the return pointer when exploiting a buffer overflow.

Parameters

- **length** (*int*) – The length of the sequence to generate.
- **width** (*int*) – The width of each element in the sequence.

Returns The sequence.

Return type str

Example

```
>>> from pwny import *
>>> cycle(80)
AAAABAAACAAADAAAEAAAFAAAGAAAHAAAIAAAJAAAKAAALAAAMAAANAAAOAAAPAAAQAAARAAASAAATAAA
```

pwnypack.util.**cycle_find**(*key*, *width=4*)
Given an element of a de Bruijn sequence, find its index in that sequence.

Parameters

- **key** (*str*) – The piece of the de Bruijn sequence to find.
- **width** (*int*) – The width of each element in the sequence.

Returns The index of `key` in the de Bruijn sequence.

Return type int

`pwnypack.util.`**reghex**(*pattern*)

Compile a regular hexpression (a short form regular expression subset specifically designed for searching for binary strings).

A regular hexpression consists of hex tuples interspaced with control characters. The available control characters are:

- `?`: Any byte (optional).
- `.`: Any byte (required).
- `?{n}`: A set of 0 up to *n* bytes.
- `.{n}`: A set of exactly *n* bytes.
- `*`: Any number of bytes (or no bytes at all).
- `+`: Any number of bytes (at least one byte).

Parameters **pattern** (*str*) – The reghex pattern.

Returns A regular expression as returned by `re.compile()`.

Return type regexp

CHAPTER 3

Indices and tables

- genindex
- modindex

p

A

B

C

D

E

F

G

H

I

K

W

X