malduck

Release 3.2.0

CERT Polska

Jun 22, 2020

EXTRACTION TOOLS:

1 Static configuration extractor engine **3**
1.1 Module interface . . . 3
1.2 Internally used classes and routines . . . 6

2 Memory model objects (procmem) **9**
2.1 ProcessMemory (procmem) . . . 9
2.2 ProcessMemoryPE (procmempe) . . . 17
2.3 ProcessMemoryELF (procmemelf) . . . 18
2.4 CuckooProcessMemory (cuckoomem) . . . 18
2.5 IDAProcessMemory (idamem) . . . 18

3 x86 disassembler **19**

4 PE wrapper **21**

5 Yara wrapper **23**

6 Cryptography **25**
6.1 AES . . . 25
6.1.1 AES-CBC mode . . . 25
6.1.2 AES-ECB mode . . . 26
6.1.3 AES-CTR mode . . . 26
6.2 Blowfish (ECB only) . . . 27
6.3 DES/DES3 (CBC only) . . . 27
6.4 Serpent (CBC only) . . . 28
6.5 Rabbit . . . 29
6.6 RC4 . . . 29
6.7 XOR . . . 29
6.8 RSA (BLOB parser) . . . 30
6.9 BLOB struct . . . 30

7 Compression algorithms **33**
7.1 aPLib . . . 33
7.2 gzip . . . 33
7.3 lznt1 (RtlDecompressBuffer) . . . 34

8 Hashing algorithms **35**
8.1 CRC32 . . . 35
8.2 MD5 . . . 35
8.3 SHA1 . . . 35
8.4 SHA224/256/384/512 . . . 35

9 Common bitwise operations 37
9.1 Rotate left/right 37
9.2 Align up/down 37

10 Fixed-integer types 39
10.1 Object properties 39
10.2 UInt64/UInt32/UInt16/UInt8 (QWORD/DWORD/WORD/BYTE) 41
10.3 Int64/Int32/Int16/Int8 41

11 Common string operations (padding, chunks, base64) 43
11.1 chunks/chunks_iter 43
11.2 asciiz/utf16z 43
11.3 enhex/unhex 43
11.4 Padding (null/pkcs7) 44
11.5 Packing/unpacking (p64/p32/p16/p8, u64/u32/u16/u8, bigint) 44
11.6 IPv4 inet_ntoa 46

12 Indices and tables 47

Python Module Index 49

Index 51

Malduck is your ducky companion in malware analysis journeys. It is mostly based on Roach project, which derives many concepts from mlib library created by Maciej Kotowicz. The purpose of fork was to make Roach independent from Cuckoo Sandbox project, but still supporting its internal *procmem* format.

Main goal is to make library for malware researchers, which will be something like pwntools for CTF players.

Malduck provides many improvements resulting from CERT.pl codebase, making malware analysis scripts much shorter and more powerful.

CHAPTER ONE

STATIC CONFIGURATION EXTRACTOR ENGINE

1.1 Module interface

class malduck.extractor.**Extractor**(*parent*)

Base class for extractor modules

Following parameters need to be defined:

- family (see *extractor.ExtractorBase.family*)
- *yara_rules*
- overrides (optional, see *extractor.ExtractorBase.overrides*)

Example extractor code for Citadel:

```
from ripper import Extractor

class Citadel(Extractor):
    family = "citadel"
    yara_rules = ["citadel"]
    overrides = ["zeus"]

    @Extractor.extractor("briankerbs")
    def citadel_found(self, p, addr):
        log.info('[+] `Coded by Brian Krebs` str @ %X' % addr)
        return True

    @Extractor.extractor
    def cit_login(self, p, addr):
        log.info('[+] Found login_key xor @ %X' % addr)
        hit = p.uint32v(addr + 4)
        print(hex(hit))
        if p.is_addr(hit):
            return {'login_key': p.asciiz(hit)}

        hit = p.uint32v(addr + 5)
        print(hex(hit))
        if p.is_addr(hit):
            return {'login_key': p.asciiz(hit)}
```

@**extractor**

Decorator for string-based extractor methods. Method is called each time when string with the same identifier as method name has matched

Extractor can be called for many number-suffixed strings e.g. *$keyex1* and *$keyex2* will call *keyex* method.

@**extractor** (*string_or_method*, *final=False*)
Specialized *@extractor* variant

Parameters

- **string_or_method** (*str*) – If method name doesn't match the string identifier pass yara string identifier as decorator argument
- **final** (*bool*) – Extractor will be called whenever Yara rule has been matched, but always after string-based extractors

@**final**
Decorator for final extractors, called after regular extraction methods.

```
from ripper import Extractor

class Evil(Extractor):
    yara_rules = ["evil"]
    family = "evil"

    ...

    @Extractor.needs_pe
    @Extractor.final
    def get_config(self, p):
        cfg = {"urls": self.get_cncs_from_rsrc(p)}
        if "role" not in self.collected_config:
            cfg["role"] = "loader"
        return cfg
```

@**weak**
Use this decorator for extractors when successful extraction is not sufficient to mark family as matched.

All "weak configs" will be flushed when "strong config" appears.

@**needs_pe**
Use this decorator for extractors that need PE instance. (`malduck.procmem.ProcessMemoryPE`)

@**needs_elf**
Use this decorator for extractors that need ELF instance. (`malduck.procmem.ProcessMemoryELF`)

property collected_config
Shows collected config so far (useful in "final" extractors)

Return type dict

property globals
Container for global variables associated with analysis

Return type dict

handle_yara (*p*, *match*)
Override this if you don't want to use decorators and customize ripping process (e.g. yara-independent, brute-force techniques)

Parameters

- **p** (`malduck.procmem.ProcessMemory`) – ProcessMemory object
- **match** (List[*malduck.yara.YaraMatch*]) – Found yara matches for this family

property log
Logger instance for Extractor methods

Returns `logging.Logger`

property matched
Returns True if family has been matched so far

Return type bool

on_error(*exc*, *method_name*)
Handler for all Exception's throwed by extractor methods.

Parameters

- **exc** (`Exception`) – Exception object
- **method_name** (*str*) – Name of method which throwed exception

push_config(*config*)
Push partial config (used by `Extractor.handle_yara()`)

Parameters **config** (*dict*) – Partial config element

push_procmem(*procmem*, ***info*)
Push procmem object for further analysis

Parameters

- **procmem** (`malduck.procmem.ProcessMemory`) – ProcessMemory object
- **info** – Additional info about object

yara_rules = ()
Names of Yara rules for which handle_yara is called

class malduck.extractor.**ExtractManager**(*modules*)
Multi-dump extraction context. Handles merging configs from different dumps, additional dropped families etc.

Parameters **modules** (`ExtractorModules`) – Object with loaded extractor modules

property config
Extracted configuration (list of configs for each extracted family)

property extractors
Bound extractor modules :rtype: List[Type[`malduck.extractor.Extractor`]]

on_error(*exc*, *extractor*)
Handler for all Exception's thrown by `Extractor.handle_yara()`.

Deprecated since version 2.1.0: Look at `ExtractManager.on_extractor_error()` instead.

Parameters

- **exc** (`Exception`) – Exception object
- **extractor** (`malduck.extractor.Extractor`) – Extractor object which throwed exception

on_extractor_error(*exc*, *extractor*, *method_name*)
Handler for all Exception's thrown by extractor methods (including `Extractor.handle_yara()`).

Override this method if you want to set your own error handler.

Parameters

- **exc** (`Exception`) – Exception object
- **extractor** (`extractor.Extractor`) – Extractor instance
- **method_name** (*str*) – Name of method which throwed exception

push_file (*filepath*, *base=0*)
Pushes file for extraction. Config extractor entrypoint.

Parameters

- **filepath** (*str*) – Path to extracted file
- **base** (*int*) – Memory dump base address

Returns Family name if ripped successfully and provided better configuration than previous files. Returns None otherwise.

push_procmem (*p*, *rip_binaries=False*)
Pushes ProcessMemory object for extraction

Parameters

- **p** (malduck.procmem.ProcessMemory) – ProcessMemory object
- **rip_binaries** – Look for binaries (PE, ELF) in provided ProcessMemory and try to perform extraction using

specialized variants (ProcessMemoryPE, ProcessMemoryELF) :type rip_binaries: bool (default: False) :return: Family name if ripped successfully and provided better configuration than previous procmems.

Returns None otherwise.

property rules
Bound Yara rules :rtype: *malduck.yara.Yara*

class malduck.extractor.**ExtractorModules** (*modules_path=None*)
Configuration object with loaded Extractor modules for ExtractManager

Parameters modules_path (*str*) – Path with module files (Extractor classes and Yara files, default '~/.malduck')

on_error (*exc*, *module_name*)
Handler for all Exception's throwed during module load

Override this method if you want to set your own error handler.

Parameters

- **exc** (Exception) – Exception object
- **module_name** (*str*) – Name of module which throwed exception

1.2 Internally used classes and routines

class malduck.extractor.extract_manager.**ProcmemExtractManager** (*parent*)
Single-dump extraction context (single family)

collected_config = None
Collected configuration so far (especially useful for "final" extractors)

property config
Returns collected config, but if family is not matched - returns empty dict. Family is not included in config itself, look at *ProcmemExtractManager.family*.

family = None
Matched family

on_extractor_error (*exc*, *extractor*, *method_name*)
Handler for all Exception's throwed by extractor methods.

Parameters

- **exc** (Exception) – Exception object
- **extractor** (extractor.Extractor) – Extractor instance
- **method_name** (*str*) – Name of method which throwed exception

parent = None
Bound ExtractManager instance

push_config (*config*, *extractor*)
Pushes new partial config

If strong config provides different family than stored so far and that family overrides stored family - set stored family Example: citadel overrides zeus

Parameters

- **config** (*dict*) – Partial config object
- **extractor** (*malduck.extractor.Extractor*) – Extractor object reference

push_procmem (*p*, *_matches=None*)
Pushes ProcessMemory object for extraction

Parameters

- **p** (malduck.procmem.ProcessMemory) – ProcessMemory object
- **_matches** (*malduck.yara.YaraMatches*) – YaraMatches object (used internally)

class malduck.extractor.extractor.**ExtractorBase** (*parent*)

property collected_config
Shows collected config so far (useful in "final" extractors)

Return type dict

family = None
Extracted malware family, automatically added to "family" key for strong extraction methods

property globals
Container for global variables associated with analysis

Return type dict

property log
Logger instance for Extractor methods

Returns logging.Logger

property matched
Returns True if family has been matched so far

Return type bool

overrides = []
Family match overrides another match e.g. citadel overrides zeus

parent = None
ProcmemExtractManager instance

push_config(*config*)
Push partial config (used by `Extractor.handle_yara()`)

Parameters **config** (*dict*) – Partial config element

push_procmem(*procmem*, **info*)
Push procmem object for further analysis

Parameters

- **procmem** (`malduck.procmem.ProcessMemory`) – ProcessMemory object
- **info** – Additional info about object

class malduck.extractor.extractor.**MetaExtractor**
Metaclass for Extractor. Handles proper registration of decorated extraction methods

CHAPTER

TWO

MEMORY MODEL OBJECTS (PROCMEM)

2.1 ProcessMemory (procmem)

malduck.**procmem**

alias of *malduck.procmem.procmem.ProcessMemory*

class malduck.procmem.procmem.**ProcessMemory**(*buf*, *base=0*, *regions=None*)

Basic virtual memory representation

Short name: *procmem*

Parameters

- **buf** (*bytes, mmap, memoryview or bytearray object*) – Object with memory contents
- **base** (*int, optional (default: 0)*) – Virtual address of the region of interest (or beginning of buf when no regions provided)
- **regions** (List[*Region*]) – Regions mapping. If set to None (default), buf is mapped into single-region with VA specified in *base* argument

Let's assume that *notepad.exe_400000.bin* contains raw memory dump starting at 0x400000 base address. We can easily load that file to *ProcessMemory* object, using *from_file()* method:

```
from malduck import procmem

with procmem.from_file("notepad.exe_400000.bin", base=0x400000) as p:
    mem = p.readv(...)
    ...
```

If your data are loaded yet into buffer, you can directly use *procmem* constructor:

```
from malduck import procmem

with open("notepad.exe_400000.bin", "rb") as f:
    payload = f.read()

p = procmem(payload, base=0x400000)
```

Then you can work with PE image contained in dump by creating `ProcessMemoryPE` object, using its *from_memory()* constructor method

```
from malduck import procmem

with open("notepad.exe_400000.bin", "rb") as f:
```

(continues on next page)

(continued from previous page)

```
    payload = f.read()

p = procmem(payload, base=0x400000)
ppe = procmempe.from_memory(p)
ppe.pe.resource("NPENCODINGDIALOG")
```

If you want to load PE file directly and work with it in a similar way as with memory-mapped files, just use *image* parameter. It also works with `ProcessMemoryPE.from_memory()` for embedded binaries. Your file will be loaded and relocated in similar way as it's done by Windows loader.

```
from malduck import procmempe

with procmempe.from_file("notepad.exe", image=True) as p:
    p.pe.resource("NPENCODINGDIALOG")
```

addr_region(*addr*)
Returns *Region* object mapping specified virtual address

Parameters **addr** – Virtual address

Return type *Region*

asciiz(*addr*)
Read a null-terminated ASCII string at address.

close(*copy=False*)
Closes opened files referenced by ProcessMemory object

Parameters **copy** – Copy data into string before closing the mmap object (default: False)

disasmv(*addr*, *size*, *x64=False*)
Disassembles code under specified address

Parameters

- **addr** (*int*) – Virtual address
- **size** (*int*) – Size of disassembled buffer
- **x64** (*bool (optional)*) – Assembly is 64bit

Returns `Disassemble`

extract(*modules=None*, *extract_manager=None*)
Tries to extract config from ProcessMemory object

Parameters

- **modules** (*malduck.extractor.ExtractorModules*) – Extractor modules object (optional, loads '~/.malduck' by default)
- **extract_manager** (*malduck.extractor.ExtractManager*) – ExtractManager object (optional, creates ExtractManager by default)

Returns Static configuration(s) (*malduck.extractor.ExtractManager.config*) or None if not extracted

Return type List[dict] or None

findbytesp(*query*, *offset=0*, *length=None*)
Search for byte sequences (e.g., *4? AA BB ?? DD*). Uses *yarap()* internally

New in version 1.4.0: Query is passed to yarap as single hexadecimal string rule. Use Yara-compatible strings only

Parameters

- **query** (*str or bytes*) – Sequence of wildcarded hexadecimal bytes, separated by spaces
- **offset** (*int (optional)*) – Buffer offset where searching will be started
- **length** (*int (optional)*) – Length of searched area

Returns Iterator returning next offsets

Return type Iterator[int]

findbytesv(*query*, *addr=None*, *length=None*)

Search for byte sequences (e.g., *4? AA BB ?? DD*). Uses `yarav()` internally

New in version 1.4.0: Query is passed to yarav as single hexadecimal string rule. Use Yara-compatible strings only

Parameters

- **query** (*str or bytes*) – Sequence of wildcarded hexadecimal bytes, separated by spaces
- **addr** (*int (optional)*) – Virtual address where searching will be started
- **length** (*int (optional)*) – Length of searched area

Returns Iterator returning found virtual addresses

Return type Iterator[int]

Usage example:

```
from malduck import hex

findings = []

for va in mem.findbytesv("4? AA BB ?? DD"):
    if hex(mem.readv(va, 5)) == "4aaabbccdd":
        findings.append(va)
```

findmz(*addr*)

Tries to locate MZ header based on address inside PE image

Parameters **addr** (*int*) – Virtual address inside image

Returns Virtual address of found MZ header or None

findp(*query*, *offset=0*, *length=None*)

Find raw bytes in memory (non-region-wise).

Parameters

- **query** (*bytes*) – Substring to find
- **offset** (*int (optional)*) – Offset in buffer where searching starts
- **length** (*int (optional)*) – Length of searched area

Returns Generates offsets where bytes were found

Return type Iterator[int]

findv (*query*, *addr=None*, *length=None*)
Find raw bytes in memory (region-wise)

Parameters

- **query** (*bytes*) – Substring to find
- **addr** (*int (optional)*) – Virtual address of region where searching starts
- **length** (*int (optional)*) – Length of searched area

Returns Generates offsets where regex was matched

Return type Iterator[int]

classmethod from_file (*filename*, ***kwargs*)
Opens file and loads its contents into ProcessMemory object

Parameters **filename** – File name to load

Return type *ProcessMemory*

It's highly recommended to use context manager when operating on files:

```
from malduck import procmem

with procmem.from_file("binary.dmp") as p:
    mem = p.readv(...)
    ...
```

classmethod from_memory (*memory*, *base=None*, ***kwargs*)
Makes new instance based on another ProcessMemory object.

Useful for specialized derived classes like `CuckooProcessMemory`

Parameters

- **memory** (*ProcessMemory*) – ProcessMemory object to be copied
- **base** (*int*) – Virtual address of region of interest (imgbase)

Return type *ProcessMemory*

int16v (*addr*, *fixed=False*)
Read signed 16-bit value at address.

int32v (*addr*, *fixed=False*)
Read signed 32-bit value at address.

int64v (*addr*, *fixed=False*)
Read signed 64-bit value at address.

int8v (*addr*, *fixed=False*)
Read signed 8-bit value at address.

is_addr (*addr*)
Checks whether provided parameter is correct virtual address :param addr: Virtual address candidate :return: True if it is mapped by ProcessMemory object

iter_regions (*addr=None*, *offset=None*, *length=None*, *contiguous=False*, *trim=False*)
Iterates over Region objects starting at provided virtual address or offset

This method is used internally to enumerate regions using provided strategy.

> **Warning:** If starting point is not provided, iteration will start from the first mapped region. This could be counter-intuitive when length is set. It literally means "get <length> of mapped bytes". If you want to look for regions from address 0, you need to explicitly provide this address as an argument.

New in version 3.0.0.

Parameters

- **addr** (*int (default: None)*) – Virtual address of starting point
- **offset** (*int (default: None)*) – Offset of starting point, which will be translated to virtual address
- **length** (*int (default: None, unlimited)*) – Length of queried range in VM mapping context
- **contiguous** (*bool (default: False)*) – If True, break after first gap. Starting point must be inside mapped region.
- **trim** (*bool (default: False)*) – Trim Region objects to range boundaries (addr, addr+length)

Return type Iterator[*Region*]

p2v (*off, length=None*)

Buffer (physical) offset to virtual address translation

Changed in version 3.0.0: Added optional mapping length check

Parameters

- **off** – Buffer offset
- **length** – Expected minimal length of mapping (optional)

Returns Virtual address or None if offset is not mapped

patchp (*offset, buf*)

Patch bytes under specified offset

> **Warning:** Family of *p methods doesn't care about contiguity of regions.
>
> Use *p2v()* and *patchv()* if you want to operate on contiguous regions only

Parameters

- **offset** (*int*) – Buffer offset
- **buf** (*bytes*) – Buffer with patch to apply

Usage example:

```
from malduck import procmempe, aplib

with procmempe("mal1.exe.dmp") as ppe:
    # Decompress payload
    payload = aPLib().decompress(
        ppe.readv(ppe.imgbase + 0x8400, ppe.imgend)
    )
    embed_pe = procmem(payload, base=0)
```

(continues on next page)

(continued from previous page)

```
# Fix headers
embed_pe.patchp(0, b"MZ")
embed_pe.patchp(embed_pe.uint32p(0x3C), b"PE")
# Load patched image into procmempe
embed_pe = procmempe.from_memory(embed_pe, image=True)
assert embed_pe.asciiz(0x1000a410) == b"StrToIntExA"
```

patchv (*addr*, *buf*)

Patch bytes under specified virtual address

Parameters

- **addr** (*int*) – Virtual address
- **buf** (*bytes*) – Buffer with patch to apply

readp (*offset*, *length=None*)

Read a chunk of memory from the specified buffer offset.

> **Warning:** Family of *p methods doesn't care about contiguity of regions.
>
> Use *p2v()* and *readv()* if you want to operate on contiguous regions only

Parameters

- **offset** – Buffer offset
- **length** – Length of chunk (optional)

Returns Chunk from specified location

Return type bytes

readv (*addr*, *length=None*)

Read a chunk of memory from the specified virtual address

Parameters

- **addr** (*int*) – Virtual address
- **length** (*int*) – Length of chunk (optional)

Returns Chunk from specified location

Return type bytes

readv_regions (*addr=None*, *length=None*, *contiguous=True*)

Generate chunks of memory from next contiguous regions, starting from the specified virtual address, until specified length of read data is reached.

Used internally.

Parameters

- **addr** – Virtual address
- **length** – Size of memory to read (optional)
- **contiguous** – If True, readv_regions breaks after first gap

Return type Iterator[Tuple[int, bytes]]

readv_until(*addr*, *s=None*)
Read a chunk of memory until the stop marker

Parameters

- **addr** (*int*) – Virtual address
- **s** (*bytes*) – Stop marker

Return type bytes

regexp(*query*, *offset=0*, *length=None*)
Performs regex on the memory contents (non-region-wise)

Parameters

- **query** (*bytes*) – Regular expression to find
- **offset** (*int (optional)*) – Offset in buffer where searching starts
- **length** (*int (optional)*) – Length of searched area

Returns Generates offsets where regex was matched

Return type Iterator[int]

regexv(*query*, *addr=None*, *length=None*)
Performs regex on the memory contents (region-wise)

Parameters

- **query** (*bytes*) – Regular expression to find
- **addr** (*int (optional)*) – Virtual address of region where searching starts
- **length** (*int (optional)*) – Length of searched area

Returns Generates offsets where regex was matched

Return type Iterator[int]

Warning: Method doesn't match bytes overlapping the border between regions

uint16p(*offset*, *fixed=False*)
Read unsigned 16-bit value at offset.

uint16v(*addr*, *fixed=False*)
Read unsigned 16-bit value at address.

uint32p(*offset*, *fixed=False*)
Read unsigned 32-bit value at offset.

uint32v(*addr*, *fixed=False*)
Read unsigned 32-bit value at address.

uint64p(*offset*, *fixed=False*)
Read unsigned 64-bit value at offset.

uint64v(*addr*, *fixed=False*)
Read unsigned 64-bit value at address.

uint8p(*offset*, *fixed=False*)
Read unsigned 8-bit value at offset.

uint8v(*addr*, *fixed=False*)
Read unsigned 8-bit value at address.

utf16z(*addr*)
Read a null-terminated UTF-16 ASCII string at address.

Parameters **addr** – Virtual address of string

Return type bytes

v2p(*addr*, *length=None*)
Virtual address to buffer (physical) offset translation

Changed in version 3.0.0: Added optional mapping length check

Parameters

- **addr** – Virtual address
- **length** – Expected minimal length of mapping (optional)

Returns Buffer offset or None if virtual address is not mapped

yarap(*ruleset*, *offset=0*, *length=None*)
Perform yara matching (non-region-wise)

Parameters

- **ruleset** (*malduck.yara.Yara*) – Yara object with loaded yara rules
- **offset** (*int (optional)*) – Offset in buffer where searching starts
- **length** (*int (optional)*) – Length of searched area

Return type *malduck.yara.YaraMatches*

yarav(*ruleset*, *addr=None*, *length=None*)
Perform yara matching (region-wise)

Parameters

- **ruleset** (*malduck.yara.Yara*) – Yara object with loaded yara rules
- **addr** (*int (optional)*) – Virtual address of region where searching starts
- **length** (*int (optional)*) – Length of searched area

Return type *malduck.yara.YaraMatches*

class malduck.procmem.procmem.**Region**(*addr*, *size*, *state*, *type_*, *protect*, *offset*)
Represents single mapped region in *ProcessMemory*

contains_addr(*addr*)
Checks whether region contains provided virtual address

contains_offset(*offset*)
Checks whether region contains provided physical offset

property end
Virtual address of region end (first unmapped byte)

property end_offset
Offset of region end (first unmapped byte)

intersects_range(*addr*, *length*)
Checks whether region mapping intersects with provided range

property last
Virtual address of last region byte

property last_offset
Offset of last region byte

p2v(*off*)
Physical offset to translation. Assumes that offset is valid within Region. :param addr: Physical offset :return: Virtual address

to_json()
Returns JSON-like dict representation

trim_range(*addr*, *length=None*)
Returns region intersection with provided range :param addr: Virtual address of starting point :param length: Length of range (optional) :rtype: `Region`

v2p(*addr*)
Virtual address to physical offset translation. Assumes that address is valid within Region. :param addr: Virtual address :return: Physical offset

2.2 ProcessMemoryPE (procmempe)

malduck.**procmempe**
alias of *malduck.procmem.procmempe.ProcessMemoryPE*

class malduck.procmem.procmempe.**ProcessMemoryPE**(*buf*, *base=0*, *regions=None*, *image=False*, *detect_image=False*)
Representation of memory-mapped PE file

Short name: *procmempe*

PE files can be read directly using inherited `ProcessMemory.from_file()` with *image* argument set (look at `from_memory()` method).

property imgend
Address where PE image ends

is_image_loaded_as_memdump()
Checks whether memory region contains image incorrectly loaded as memory-mapped PE dump (image=False).

```
embed_pe = procmempe.from_memory(mem)
if not embed_pe.is_image_loaded_as_memdump():
    # Memory contains plain PE file - need to load it first
    embed_pe = procmempe.from_memory(mem, image=True)
```

is_valid()
Checks whether imgbase is pointing at valid binary header

property pe
Related `PE` object

store()
Store ProcessMemoryPE contents as PE file data.

Return type bytes

2.3 ProcessMemoryELF (procmemelf)

malduck.**procmemelf**
alias of *malduck.procmem.procmemelf.ProcessMemoryELF*

class malduck.procmem.procmemelf.**ProcessMemoryELF**(*buf*, *base=0*, *regions=None*, *image=False*, *detect_image=False*)
Representation of memory-mapped ELF file

Short name: *procmemelf*

ELF files can be read directly using inherited `ProcessMemory.from_file()` with *image* argument set (look at `from_memory()` method).

property elf
Related `ELFFile` object

property imgend
Address where ELF image ends

is_image_loaded_as_memdump()
Uses some heuristics to deduce whether contents can be loaded with *image=True*. Used by *detect_image*

is_valid()
Checks whether imgbase is pointing at valid binary header

2.4 CuckooProcessMemory (cuckoomem)

malduck.**cuckoomem**
alias of *malduck.procmem.cuckoomem.CuckooProcessMemory*

class malduck.procmem.cuckoomem.**CuckooProcessMemory**(*buf*, *base=None*, ***kwargs*)
Wrapper object to operate on process memory dumps in Cuckoo 2.x format.

2.5 IDAProcessMemory (idamem)

malduck.**idamem**
alias of *malduck.procmem.idamem.IDAProcessMemory*

class malduck.procmem.idamem.**IDAProcessMemory**
ProcessMemory representation operating in IDAPython context [BETA]

CHAPTER THREE

X86 DISASSEMBLER

class malduck.disasm.**Disassemble**

disassemble(*data*, *addr*, *x64=False*)

Disassembles data from specific address

short: disasm

Parameters

- **data** (*bytes*) – Block of data to disasseble
- **addr** (*int*) – Virtual address of data
- **x64** (*bool (default=False)*) – Disassemble in x86-64 mode?

Returns Returns list of instructions

Return type List[*Instruction*]

class malduck.disasm.**Instruction**(*mnem=None*, *op1=None*, *op2=None*, *op3=None*, *addr=None*, *x64=False*)

Represents single instruction in *Disassemble*

short: insn

Properties correspond to the following elements of instruction:

```
00400000  imul    ecx,   edx,   0
[addr]    [mnem]  [op1], [op2], [op3]
```

Usage example:

```
def get_move_value(self, p, hit, *args):
    # find move value of `mov eax, x`
    for ins in p.disasmv(hit, 0x100):
        if ins.mnem == 'mov' and ins.op1.value == 'eax':
            return ins.op2.value
```

See also:

malduck.procmem.ProcessMemory.disasmv()

property addr

Instruction address

property op1

First operand

property op2
Second operand

property op3
Third operand

class malduck.disasm.**Operand**(*op*, *x64*)
Operand object for single *Instruction*

property is_imm
Is it immediate operand?

property is_mem
Is it memory operand?

property is_reg
Is it register operand?

property mem
Returns *Memory* object for memory operands

property reg
Returns register used by operand.

For memory operands, returns base register or index register if base is not used. For immediate operands or displacement-only memory operands returns None.

Return type str

property value
Returns operand value or displacement value for memory operands

Return type str or int

class malduck.disasm.**Memory**(*size*, *base*, *scale*, *index*, *disp*)

property base
Alias for field number 1

property disp
Alias for field number 4

property index
Alias for field number 3

property scale
Alias for field number 2

property size
Alias for field number 0

CHAPTER FOUR

PE WRAPPER

class malduck.pe.**PE**(*data*, *fast_load=False*)
Wrapper around `pefile.PE`, accepts either bytes (raw file contents) or `ProcessMemory` instance.

directory(*name*)
Get pefile directory entry by identifier

Parameters **name** – shortened pefile directory entry identifier (e.g. 'IMPORT' for 'IMAGE_DIRECTORY_ENTRY_IMPORT')

Return type `pefile.Structure`

property dos_header
Dos header

property file_header
File header

property headers_size
Estimated size of PE headers (first section offset). If there are no sections: returns 0x1000 or size of input if provided data are shorter than single page

property is32bit
Is it 32-bit file (PE)?

property is64bit
Is it 64-bit file (PE+)?

property nt_headers
NT headers

property optional_header
Optional header

resource(*name*)
Retrieves single resource by specified name or type

Parameters **name** (*int or str or bytes*) – String name (e2) or type (e1), numeric identifier name (e2) or RT_* type (e1)

Return type bytes or None

resources(*name*)
Finds resource objects by specified name or type

Parameters **name** (*int or str or bytes*) – String name (e2) or type (e1), numeric identifier name (e2) or RT_* type (e1)

Return type Iterator[bytes]

section (*name*)
Get section by name

Parameters **name** (*str or bytes*) – Section name

property sections
Sections

structure (*rva, format*)
Get internal pefile Structure from specified rva

Parameters **format** – `pefile.Structure` format (e.g. `pefile.PE.__IMAGE_LOAD_CONFIG_DIRECTORY64_format__`)

Return type `pefile.Structure`

validate_import_names ()
Returns True if the first 8 imported library entries have valid library names

validate_padding ()
Returns True if area between first non-bss section and first 4kB doesn't have only null-bytes

validate_resources ()
Returns True if first level of resource tree looks consistent

CHAPTER

FIVE

YARA WRAPPER

class malduck.yara.**Yara**(*rule_paths=None*, *name='r'*, *strings=None*, *condition='any of them'*)

Represents Yara ruleset. Rules can be compiled from set of files or defined in code (single rule only).

Most simple rule (with default identifiers left):

```
from malduck.yara import Yara, YaraString

Yara(strings="MALWR").match(data=b"MALWRMALWARMALWR").r.string == [0, 11]
```

Example of more complex rule defined in Python:

```
from malduck.yara import Yara, YaraString

ruleset = Yara(name="MalwareRule",
strings={
    "xor_stub": YaraString("This program cannot", xor=True, ascii=True),
    "code_ref": YaraString("E2 34 ?? C8 A? FB", type=YaraString.HEX),
    "mal1": "MALWR",
    "mal2": "MALRW"
}, condition="( $xor_stub and $code_ref ) or any of ($mal*)")

# If mal1 or mal2 are matched, they are grouped into "mal"

# Print appropriate offsets

match = ruleset.match(data=b"MALWR MALRW")

if match:
    # ["mal1", "mal", "mal2"]
    print(match.MalwareRule.keys())
    if "mal" in match.MalwareRule:
        # Note: Order of offsets for grouped is arbitrary
        print("mal*", match.MalwareRule["mal"])
```

Parameters

- **rule_paths** (*dict*) – Dictionary of {"namespace": "rule_path"}. See also `Yara.from_dir()`.
- **name** (*str*) – Name of generated rule (default: "r")
- **strings** (dict or str or *YaraString*) – Dictionary representing set of string patterns ({"string_identifier": YaraString or plain str})
- **condition** (*str*) – Yara rule condition (default: "any of them")

static from_dir(*path*, *recursive=True*, *followlinks=True*)
Find rules (recursively) in specified path. Supported extensions: *.yar, *.yara

Parameters

- **path** (*str*) – Root path for searching
- **recursive** (*bool*) – Search recursively (default: enabled)
- **followlinks** (*bool*) – Follow symbolic links (default: enabled)

Return type *Yara*

match(*offset_mapper=None*, ***kwargs*)
Perform matching on file or data block

Parameters

- **filepath** (*str*) – Path to the file to be scanned
- **data** (*str*) – Data to be scanned
- **offset_mapper** (*function*) – Offset mapping function. For unmapped region, should returned None. Used by `malduck.procmem.ProcessMemory.yarav()`

Return type *YaraMatches*

class malduck.yara.**YaraString**(*value*, *type=0*, ***modifiers*)
Formatter for Yara string patterns

Parameters

- **value** (*str*) – Pattern value
- **type** (*YaraString.TEXT* / *YaraString.HEX* / *YaraString.REGEX*) – Pattern type (default is *YaraString.TEXT*)
- **modifiers** – Yara string modifier flags

HEX = 1
Hexadecimal string (*"aa bb cc dd"* => *'{ aa bb cc dd }'*)

REGEX = 2
Regex string (*'value'* => *'/value/'*)

TEXT = 0
Text string (*'value'* => *'"value"'*)

class malduck.yara.**YaraMatches**(*match_results*, *offset_mapper=None*)
Represented matching results. Returned by *Yara.match()*.

Rules can be referenced by both attribute and index.

keys()
List of matched rule identifiers

class malduck.yara.**YaraMatch**(*match*, *offset_mapper=None*)
Represented matching results for rules. Returned by *YaraMatches.<rule>*.

Strings can be referenced by both attribute and index.

get(*item*)
Get matched string offsets or empty list if not matched

keys()
List of matched string identifiers

CHAPTER

SIX

CRYPTOGRAPHY

Common cryptography algorithms used in malware.

6.1 AES

AES (Advanced Encryption Standard) block cipher.

Supported modes: CBC, ECB, CTR.

```
from malduck import aes

key = b'A'*16
iv = b'B'*16
plaintext = b'data'*16
ciphertext = aes.cbc.encrypt(key, iv, plaintext)
```

6.1.1 AES-CBC mode

`malduck.aes.cbc.`**`encrypt`**(*key*, *iv*, *data*)

Encrypts buffer using AES algorithm in CBC mode.

Parameters

- **key** (*bytes*) – Cryptographic key (128, 192 or 256 bits)
- **iv** (*bytes*) – Initialization vector
- **data** (*bytes*) – Buffer to be encrypted

Returns Encrypted data

Return type bytes

`malduck.aes.cbc.`**`decrypt`**(*key*, *iv*, *data*)

Decrypts buffer using AES algorithm in CBC mode.

Parameters

- **key** (*bytes*) – Cryptographic key (128, 192 or 256 bits)
- **iv** (*bytes*) – Initialization vector
- **data** (*bytes*) – Buffer to be decrypted

Returns Decrypted data

Return type bytes

6.1.2 AES-ECB mode

`malduck.aes.ecb.encrypt` (*key*, *data*)
Encrypts buffer using AES algorithm in ECB mode.

Parameters

- **key** (*bytes*) – Cryptographic key (128, 192 or 256 bits)
- **data** (*bytes*) – Buffer to be encrypted

Returns Encrypted data

Return type bytes

`malduck.aes.ecb.decrypt` (*key*, *data*)
Decrypts buffer using AES algorithm in ECB mode.

Parameters

- **key** (*bytes*) – Cryptographic key (128, 192 or 256 bits)
- **data** (*bytes*) – Buffer to be decrypted

Returns Decrypted data

Return type bytes

6.1.3 AES-CTR mode

`malduck.aes.ctr.encrypt` (*key*, *nonce*, *data*)
Encrypts buffer using AES algorithm in CTR mode.

Parameters

- **key** (*bytes*) – Cryptographic key (128, 192 or 256 bits)
- **nonce** (*bytes*) – Initial counter value, big-endian encoded
- **data** (*bytes*) – Buffer to be encrypted

Returns Encrypted data

Return type bytes

`malduck.aes.ctr.decrypt` (*key*, *nonce*, *data*)
Decrypts buffer using AES algorithm in CTR mode.

Parameters

- **key** (*bytes*) – Cryptographic key (128, 192 or 256 bits)
- **nonce** (*bytes*) – Initial counter value, big-endian encoded
- **data** (*bytes*) – Buffer to be decrypted

Returns Decrypted data

Return type bytes

6.2 Blowfish (ECB only)

Blowfish block cipher.

Supported modes: ECB.

```
from malduck import blowfish

key = b'blowfish'
plaintext = b'data'*16
ciphertext = blowfish.ecb.encrypt(key, plaintext)
```

`malduck.blowfish.ecb.`**`encrypt`**(*key*, *data*)
Encrypts buffer using Blowfish algorithm in ECB mode.

Parameters

- **key** (*bytes*) – Cryptographic key (4 to 56 bytes)
- **data** (*bytes*) – Buffer to be encrypted

Returns Encrypted data

Return type bytes

`malduck.blowfish.ecb.`**`decrypt`**(*key*, *data*)
Decrypts buffer using Blowfish algorithm in ECB mode.

Parameters

- **key** (*bytes*) – Cryptographic key (4 to 56 bytes)
- **data** (*bytes*) – Buffer to be decrypted

Returns Decrypted data

Return type bytes

6.3 DES/DES3 (CBC only)

Triple DES block cipher.

Fallbacks to single DES for 8 byte keys.

Supported modes: CBC.

```
from malduck import des3

key = b'des3des3'
iv = b'3des3des'
plaintext = b'data'*16
ciphertext = des3.cbc.decrypt(key, plaintext)
```

`malduck.des3.cbc.`**`encrypt`**(*key*, *iv*, *data*)
Encrypts buffer using DES/DES3 algorithm in CBC mode.

Parameters

- **key** (*bytes*) – Cryptographic key (16 or 24 bytes, 8 bytes for single DES)
- **iv** (*bytes*) – Initialization vector

- **data** (*bytes*) – Buffer to be encrypted

Returns Encrypted data

Return type bytes

`malduck.des3.cbc.decrypt(key, iv, data)`
Decrypts buffer using DES/DES3 algorithm in CBC mode.

Parameters

- **key** (*bytes*) – Cryptographic key (16 or 24 bytes, 8 bytes for single DES)
- **iv** (*bytes*) – Initialization vector
- **data** (*bytes*) – Buffer to be decrypted

Returns Decrypted data

Return type bytes

6.4 Serpent (CBC only)

Serpent block cipher.

Supported modes: CBC

```
from malduck import serpent

key = b'a'*16
iv = b'b'*16
plaintext = b'data'*16
ciphertext = serpent.cbc.encrypt(key, plaintext, iv=iv)
```

`malduck.serpent.cbc.encrypt(key, data, iv=None)`
Encrypts buffer using Serpent algorithm in CBC mode.

Parameters

- **key** (*bytes*) – Cryptographic key (4-32 bytes, must be multiple of four)
- **data** (*bytes*) – Buffer to be encrypted
- **iv** (*bytes, optional*) – Initialization vector (defaults to *b"" * 16*)

Returns Encrypted data

Return type bytes

`malduck.serpent.cbc.decrypt(key, data, iv=None)`
Decrypts buffer using Serpent algorithm in CBC mode.

Parameters

- **key** (*bytes*) – Cryptographic key (4-32 bytes, must be multiple of four)
- **data** (*bytes*) – Buffer to be decrypted
- **iv** (*bytes, optional*) – Initialization vector (defaults to *b"" * 16*)

Returns Decrypted data

Return type bytes

6.5 Rabbit

Rabbit stream cipher.

```
from malduck import rabbit

key = b'a'*16
plaintext = b'data'*16
ciphertext = rabbit(key, plaintext)
```

`malduck.`**`rabbit`**(*key*, *iv*, *data*)

Encrypts/decrypts buffer using Rabbit algorithm

Parameters

- **key** (*bytes*) – Cryptographic key (16 bytes)
- **iv** (*bytes*) – Initialization vector (8 bytes)
- **data** (*bytes*) – Buffer to be encrypted/decrypted

Returns Encrypted/decrypted data

Return type bytes

6.6 RC4

RC4 stream cipher.

```
from malduck import rc4

key = b'a'*16
plaintext = b'data'*16
ciphertext = rc4(key, plaintext)
```

`malduck.`**`rc4`**(*key*, *data*)

Encrypts/decrypts buffer using RC4 algorithm

Parameters

- **key** (*bytes*) – Cryptographic key (from 3 to 256 bytes)
- **data** (*bytes*) – Buffer to be encrypted/decrypted

Returns Encrypted/decrypted data

Return type bytes

6.7 XOR

XOR "stream cipher".

```
from malduck import xor

key = b'a'*16
xored = b'data'*16
unxored = xor(key, xor)
```

`malduck.xor(key, data)`
XOR encryption/decryption

Parameters

- **key** (*int (single byte) or bytes*) – Encryption key
- **data** (*bytes*) – Buffer containing data to decrypt

Returns Encrypted/decrypted data

Return type bytes

6.8 RSA (BLOB parser)

`malduck.rsa`
alias of *malduck.crypto.rsa.RSA*

`class malduck.crypto.rsa.RSA`

`static export_key(n, e, d=None, p=None, q=None, crt=None)`
Constructs key from tuple of RSA components

Parameters

- **n** – RSA modulus n
- **e** – Public exponent e
- **d** – Private exponent d
- **p** – First factor of n
- **q** – Second factor of n
- **crt** – CRT coefficient q

Returns RSA key in PEM format

Return type bytes

`static import_key(data)`
Extracts key from buffer containing *PublicKeyBlob* or *PrivateKeyBlob* data

Parameters **data** (*bytes*) – Buffer with *BLOB* structure data

Returns RSA key in PEM format

Return type bytes

6.9 BLOB struct

`class malduck.crypto.winhdr.BLOBHEADER`
Windows BLOBHEADER structure

See also:

BLOBHEADER structure description (Microsoft Docs): https://docs.microsoft.com/en-us/windows/win32/api/wincrypt/ns-wincrypt-publickeystruc

class malduck.crypto.aes.**PlaintextKeyBlob**

BLOB object (*PLAINTEXTKEYBLOB*) for *CALG_AES*

See also:

malduck.crypto.BLOBHEADER

export_key()

Exports key from structure

Returns Tuple (*algorithm*, *key*). *Algorithm* is one of: "AES-128", "AES-192", "AES-256"

Return type Tuple[str, bytes]

parse(*buf*)

Parse structure from buffer

Parameters **buf** (io.BytesIO) – Buffer with structure data

class malduck.crypto.rsa.**PublicKeyBlob**

class malduck.crypto.rsa.**PrivateKeyBlob**

CHAPTER

SEVEN

COMPRESSION ALGORITHMS

7.1 aPLib

`malduck.aplib(buf, length=None, headerless=False)`

aPLib decompression

Changed in version 2.0: *length* argument is deprecated

```
from malduck import aplib

# Headerless compressed buffer
aplib(b'T\x00he quick\xecb\x0erown\xcef\xaex\x80jumps\xed\xe4veur`t?
↪lazy\xead\xfeg\xc0\x00')
# Header included
aplib(b
↪'AP32\x18\x00\x00\x00\r\x00\x00\x00\xbc\x9ab\x9b\x0b\x00\x00\x00\x85\x11J\rh8el\x8eo␣
↪wnr\xecd\x00')
```

Parameters

- **buf** (*bytes*) – Buffer to decompress
- **headerless** (bool (default: *True*)) – Force headerless decompression (don't perform 'AP32' magic detection)

Return type bytes

7.2 gzip

`malduck.gzip(buf)`

gzip/zlib decompression

```
from malduck import gzip, unhex

# zlib decompression
gzip(unhex(b'789ccb48cdc9c95728cf2fca4901001a0b045d'))
# gzip decompression (detected by 1f8b08 prefix)
gzip(unhex(b
↪'1f8b08082199b75a0403312d3100cb48cdc9c95728cf2fca49010085114a0d0b000000'))
```

Parameters **buf** (*bytes*) – Buffer to decompress

Return type bytes

7.3 lznt1 (RtlDecompressBuffer)

malduck.**lznt1**(*buf*)

Implementation of LZNT1 decompression. Allows to decompress data compressed by RtlCompressBuffer

```
from malduck import lznt1

lznt1(b"°\compress\edtestdataalot")
```

Parameters **buf** (*bytes*) – Buffer to decompress

Return type bytes

CHAPTER EIGHT

HASHING ALGORITHMS

8.1 CRC32

malduck.**crc32**(*val*)

Computes CRC32 checksum for provided data

Changed in version 3.0.0: Guaranteed to be unsigned on both Py2/Py3

8.2 MD5

malduck.**md5**(*s*)

8.3 SHA1

malduck.**sha1**(*s*)

8.4 SHA224/256/384/512

malduck.**sha224**(*s*)

malduck.**sha256**(*s*)

malduck.**sha384**(*s*)

malduck.**sha512**(*s*)

CHAPTER

NINE

COMMON BITWISE OPERATIONS

9.1 Rotate left/right

`malduck.bits.rol(value, count, bits=32)`
Bitwise rotate left

Parameters

- **value** – Value to rotate
- **count** – Number of bits to rotate
- **bits** – Bit-length of rotated value (default: 32-bit, DWORD)

See also:

malduck.ints.IntType.rol()

`malduck.bits.ror(value, count, bits=32)`
Bitwise rotate right

Parameters

- **value** – Value to rotate
- **count** – Number of bits to rotate
- **bits** – Bit-length of rotated value (default: 32-bit, DWORD)

See also:

malduck.ints.IntType.ror()

9.2 Align up/down

`malduck.bits.align(value, round_to)`
Rounds value up to provided alignment

`malduck.bits.align_down(value, round_to)`
Rounds value down to provided alignment

CHAPTER

TEN

FIXED-INTEGER TYPES

10.1 Object properties

class malduck.ints.**IntType**

Fixed-size variant of long type with C-style operators and casting

Supports ctypes-like multiplication for unpacking tuple of values

- **Unsigned types:** *UInt64* (QWORD), *UInt32* (DWORD), *UInt16* (WORD), *UInt8* (BYTE or CHAR)
- **Signed types:** *Int64*, *Int32*, *Int16*, *Int8*

IntTypes are derived from `long` type, so they are fully compatible with other numeric types

```
res = u32(0x8080FFFF) << 16 | 0xFFFF
> 0xFFFFFFFF
res = Int32(res)
> -1
```

Using IntTypes you don't need to mask everything with 0xFFFFFFFF, only if you remember about appropriate casting.

```
from malduck import DWORD

def rol7_hash(name: bytes):
    hh = 0
    for c in name:
        hh = DWORD(x).rol(7) ^ c
    return x

def sdbm_hash(name: bytes):
    hh = 0
    for c in name:
        hh = DWORD(c) + (hh << 6) + (hh << 16) - hh
    return hh
```

Type coercion between native and fixed integers depends on LHS type:

```
UInt32 = UInt32 + int
int = int + UInt32
```

IntTypes can be multiplied like ctypes classes for unpacking tuple of values:

```
values = (BYTE * 3).unpack('\x01\x02\x03')

values -> (1, 2, 3)
```

pack()
Pack value into bytes with little-endian order

pack_be()
Pack value into bytes with big-endian order

rol(*other*)
Bitwise rotate left

ror(*other*)
Bitwise rotate right

classmethod unpack(*other*, *offset=0*, *fixed=True*)
Unpacks single value from provided buffer with little-endian order

Parameters

- **other** (*bytes*) – Buffer object containing value to unpack
- **offset** (*int*) – Buffer offset
- **fixed** (*bool (default: True)*) – Convert to fixed-size integer (IntType instance)

Return type IntType instance or None if there are not enough data to unpack

> **Warning:** Fixed-size integer operations are 4-5 times slower than equivalent on built-in integer types

classmethod unpack_be(*other*, *offset=0*, *fixed=True*)
Unpacks single value from provided buffer with big-endian order

Parameters

- **other** (*bytes*) – Buffer object containing value to unpack
- **offset** (*int*) – Buffer offset
- **fixed** (*bool (default: True)*) – Convert to fixed-size integer (IntType instance)

Return type IntType instance or None if there are not enough data to unpack

> **Warning:** Fixed-size integer operations are 4-5 times slower than equivalent on built-in integer types

class malduck.ints.**IntTypeBase**
Base class representing all IntType instances

class malduck.ints.**MultipliedIntTypeBase**
Base class representing all MultipliedIntType instances

class malduck.ints.**MetaIntType**
Metaclass for IntType classes. Provides ctypes-like behavior e.g. (QWORD*8).unpack(…) returns tuple of 8 QWORDs

property invert_mask
Mask for sign bit

property mask
Mask for potentially overflowing operations

10.2 UInt64/UInt32/UInt16/UInt8 (QWORD/DWORD/WORD/BYTE)

malduck.**QWORD**
alias of *malduck.ints.UInt64*

malduck.**DWORD**
alias of *malduck.ints.UInt32*

malduck.**WORD**
alias of *malduck.ints.UInt16*

malduck.**BYTE**
alias of *malduck.ints.UInt8*

class malduck.ints.**UInt64**

class malduck.ints.**UInt32**

class malduck.ints.**UInt16**

class malduck.ints.**UInt8**

10.3 Int64/Int32/Int16/Int8

class malduck.ints.**Int64**

class malduck.ints.**Int32**

class malduck.ints.**Int16**

class malduck.ints.**Int8**

CHAPTER

ELEVEN

COMMON STRING OPERATIONS (PADDING, CHUNKS, BASE64)

Supports most common string operations e.g.:

- **packing/unpacking:** `p64(),p32(),p16(),p8()`

  `u64(),u32(),u16(),u8()`

- chunks: `chunks_iter(),chunks()`

11.1 chunks/chunks_iter

`malduck.`**`chunks_iter`**`(`*s*, *n*`)`
Yield successive n-sized chunks from s.

`malduck.`**`chunks`**`(`*s*, *n*`)`
Return list of successive n-sized chunks from s.

11.2 asciiz/utf16z

`malduck.`**`asciiz`**`(`*s*`)`
Treats s as null-terminated ASCII string

Parameters **s** (*bytes*) – Buffer containing null-terminated ASCII string

`malduck.`**`utf16z`**`(`*s*`)`
Treats s as null-terminated UTF-16 ASCII string

Parameters **s** (*bytes*) – Buffer containing null-terminated UTF-16 string

Returns ASCII string without ‘’ terminator

Return type bytes

11.3 enhex/unhex

`malduck.`**`enhex`**`(`*s*`)`
Changed in version 2.0.0: Renamed from `malduck.hex()`

`malduck.`**`unhex`**`(`*s*`)`

`malduck.`**`uleb128`**`(`*s*`)`
Unsigned Little-Endian Base 128

`malduck.base64(s)`
Base64 encoder/decoder

11.4 Padding (null/pkcs7)

`malduck.pad(s, block_size)`
Padding PKCS7/NULL

`malduck.unpad(s)`
Unpadding PKCS7/NULL

11.5 Packing/unpacking (p64/p32/p16/p8, u64/u32/u16/u8, bigint)

`malduck.uint64(other, offset=0, fixed=True)`
Unpacks single value from provided buffer with little-endian order

Parameters

- **other** (*bytes*) – Buffer object containing value to unpack
- **offset** (*int*) – Buffer offset
- **fixed** (*bool (default: True)*) – Convert to fixed-size integer (IntType instance)

Return type IntType instance or None if there are not enough data to unpack

> **Warning:** Fixed-size integer operations are 4-5 times slower than equivalent on built-in integer types

`malduck.uint32(other, offset=0, fixed=True)`
Unpacks single value from provided buffer with little-endian order

Parameters

- **other** (*bytes*) – Buffer object containing value to unpack
- **offset** (*int*) – Buffer offset
- **fixed** (*bool (default: True)*) – Convert to fixed-size integer (IntType instance)

Return type IntType instance or None if there are not enough data to unpack

> **Warning:** Fixed-size integer operations are 4-5 times slower than equivalent on built-in integer types

`malduck.uint16(other, offset=0, fixed=True)`
Unpacks single value from provided buffer with little-endian order

Parameters

- **other** (*bytes*) – Buffer object containing value to unpack
- **offset** (*int*) – Buffer offset
- **fixed** (*bool (default: True)*) – Convert to fixed-size integer (IntType instance)

Return type IntType instance or None if there are not enough data to unpack

> **Warning:** Fixed-size integer operations are 4-5 times slower than equivalent on built-in integer types

malduck.**uint8**(*other*, *offset=0*, *fixed=True*)

Unpacks single value from provided buffer with little-endian order

Parameters

- **other** (*bytes*) – Buffer object containing value to unpack
- **offset** (*int*) – Buffer offset
- **fixed** (*bool (default: True)*) – Convert to fixed-size integer (IntType instance)

Return type IntType instance or None if there are not enough data to unpack

> **Warning:** Fixed-size integer operations are 4-5 times slower than equivalent on built-in integer types

malduck.**u64**(*other*, *offset=0*, *fixed=True*)

Unpacks single value from provided buffer with little-endian order

Parameters

- **other** (*bytes*) – Buffer object containing value to unpack
- **offset** (*int*) – Buffer offset
- **fixed** (*bool (default: True)*) – Convert to fixed-size integer (IntType instance)

Return type IntType instance or None if there are not enough data to unpack

> **Warning:** Fixed-size integer operations are 4-5 times slower than equivalent on built-in integer types

malduck.**u32**(*other*, *offset=0*, *fixed=True*)

Unpacks single value from provided buffer with little-endian order

Parameters

- **other** (*bytes*) – Buffer object containing value to unpack
- **offset** (*int*) – Buffer offset
- **fixed** (*bool (default: True)*) – Convert to fixed-size integer (IntType instance)

Return type IntType instance or None if there are not enough data to unpack

> **Warning:** Fixed-size integer operations are 4-5 times slower than equivalent on built-in integer types

malduck.**u16**(*other*, *offset=0*, *fixed=True*)

Unpacks single value from provided buffer with little-endian order

Parameters

- **other** (*bytes*) – Buffer object containing value to unpack
- **offset** (*int*) – Buffer offset
- **fixed** (*bool (default: True)*) – Convert to fixed-size integer (IntType instance)

Return type IntType instance or None if there are not enough data to unpack

> **Warning:** Fixed-size integer operations are 4-5 times slower than equivalent on built-in integer types

`malduck.`**`u8`**(*other*, *offset=0*, *fixed=True*)
Unpacks single value from provided buffer with little-endian order

Parameters

- **`other`** (*`bytes`*) – Buffer object containing value to unpack
- **`offset`** (*`int`*) – Buffer offset
- **`fixed`** (*`bool (default: True)`*) – Convert to fixed-size integer (IntType instance)

Return type IntType instance or None if there are not enough data to unpack

> **Warning:** Fixed-size integer operations are 4-5 times slower than equivalent on built-in integer types

`malduck.`**`p64`**(*v*)

`malduck.`**`p32`**(*v*)

`malduck.`**`p16`**(*v*)

`malduck.`**`p8`**(*v*)

`malduck.`**`bigint`**(*s*, *bitsize*)

11.6 IPv4 inet_ntoa

`malduck.`**`ipv4`**(*s*)

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

malduck.bits, 37
malduck.compression, 33
malduck.crypto, 25
malduck.disasm, 19
malduck.extractor, 3
malduck.hash, 35
malduck.ints, 39
malduck.pe, 21
malduck.procmem, 9
malduck.string, 43
malduck.yara, 23

INDEX

A

addr() (*malduck.disasm.Instruction property*), 19
addr_region() (*malduck.procmem.procmem.ProcessMemory method*), 10
align() (*in module malduck.bits*), 37
align_down() (*in module malduck.bits*), 37
aplib() (*in module malduck*), 33
asciiz() (*in module malduck*), 43
asciiz() (*malduck.procmem.procmem.ProcessMemory method*), 10

B

base() (*malduck.disasm.Memory property*), 20
base64() (*in module malduck*), 43
bigint() (*in module malduck*), 46
BLOBHEADER (*class in malduck.crypto.winhdr*), 30
BYTE (*in module malduck*), 41

C

chunks() (*in module malduck*), 43
chunks_iter() (*in module malduck*), 43
close() (*malduck.procmem.procmem.ProcessMemory method*), 10
collected_config (*malduck.extractor.extract_manager.ProcmemExtractManager attribute*), 6
collected_config() (*malduck.extractor.Extractor property*), 4
collected_config() (*malduck.extractor.extractor.ExtractorBase property*), 7
config() (*malduck.extractor.extract_manager.ProcmemExtractManager property*), 6
config() (*malduck.extractor.ExtractManager property*), 5
contains_addr() (*malduck.procmem.procmem.Region method*), 16
contains_offset() (*malduck.procmem.procmem.Region method*), 16
crc32() (*in module malduck*), 35
cuckoomem (*in module malduck*), 18
CuckooProcessMemory (*class in malduck.procmem.cuckoomem*), 18

D

decrypt() (*in module malduck.aes.cbc*), 25
decrypt() (*in module malduck.aes.ctr*), 26
decrypt() (*in module malduck.aes.ecb*), 26
decrypt() (*in module malduck.blowfish.ecb*), 27
decrypt() (*in module malduck.des3.cbc*), 28
decrypt() (*in module malduck.serpent.cbc*), 28
directory() (*malduck.pe.PE method*), 21
disasmv() (*malduck.procmem.procmem.ProcessMemory method*), 10
Disassemble (*class in malduck.disasm*), 19
disassemble() (*malduck.disasm.Disassemble method*), 19
disp() (*malduck.disasm.Memory property*), 20
dos_header() (*malduck.pe.PE property*), 21
DWORD (*in module malduck*), 41

E

elf() (*malduck.procmem.procmemelf.ProcessMemoryELF property*), 18
encrypt() (*in module malduck.aes.cbc*), 25
encrypt() (*in module malduck.aes.ctr*), 26
encrypt() (*in module malduck.aes.ecb*), 26
encrypt() (*in module malduck.blowfish.ecb*), 27
encrypt() (*in module malduck.des3.cbc*), 27
encrypt() (*in module malduck.serpent.cbc*), 28
end() (*malduck.procmem.procmem.Region property*), 16
end_offset() (*malduck.procmem.procmem.Region property*), 16
enhex() (*in module malduck*), 43
export_key() (*malduck.crypto.aes.PlaintextKeyBlob method*), 31
export_key() (*malduck.crypto.rsa.RSA static method*), 30
extract() (*malduck.procmem.procmem.ProcessMemory method*), 10

ExtractManager (*class in malduck.extractor*), 5
Extractor (*class in malduck.extractor*), 3
extractor() (*malduck.extractor.Extractor method*), 3
ExtractorBase (*class in malduck.extractor.extractor*), 7
ExtractorModules (*class in malduck.extractor*), 6
extractors() (*malduck.extractor.ExtractManager property*), 5

F

family (*malduck.extractor.extract_manager.ProcmemExtractManager attribute*), 6
family (*malduck.extractor.extractor.ExtractorBase attribute*), 7
file_header() (*malduck.pe.PE property*), 21
final() (*malduck.extractor.Extractor method*), 4
findbytesp() (*malduck.procmem.procmem.ProcessMemory method*), 10
findbytesv() (*malduck.procmem.procmem.ProcessMemory method*), 11
findmz() (*malduck.procmem.procmem.ProcessMemory method*), 11
findp() (*malduck.procmem.procmem.ProcessMemory method*), 11
findv() (*malduck.procmem.procmem.ProcessMemory method*), 11
from_dir() (*malduck.yara.Yara static method*), 23
from_file() (*malduck.procmem.procmem.ProcessMemory class method*), 12
from_memory() (*malduck.procmem.procmem.ProcessMemory class method*), 12

G

get() (*malduck.yara.YaraMatch method*), 24
globals() (*malduck.extractor.Extractor property*), 4
globals() (*malduck.extractor.extractor.ExtractorBase property*), 7
gzip() (*in module malduck*), 33

H

handle_yara() (*malduck.extractor.Extractor method*), 4
headers_size() (*malduck.pe.PE property*), 21
HEX (*malduck.yara.YaraString attribute*), 24

I

idamem (*in module malduck*), 18
IDAProcessMemory (*class in malduck.procmem.idamem*), 18
imgend() (*malduck.procmem.procmemelf.ProcessMemoryELF property*), 18
imgend() (*malduck.procmem.procmempe.ProcessMemoryPE property*), 17
import_key() (*malduck.crypto.rsa.RSA static method*), 30
index() (*malduck.disasm.Memory property*), 20
Instruction (*class in malduck.disasm*), 19
Int16 (*class in malduck.ints*), 41
int16v() (*malduck.procmem.procmem.ProcessMemory method*), 12
Int32 (*class in malduck.ints*), 41
int32v() (*malduck.procmem.procmem.ProcessMemory method*), 12
Int64 (*class in malduck.ints*), 41
int64v() (*malduck.procmem.procmem.ProcessMemory method*), 12
Int8 (*class in malduck.ints*), 41
int8v() (*malduck.procmem.procmem.ProcessMemory method*), 12
intersects_range() (*malduck.procmem.procmem.Region method*), 16
IntType (*class in malduck.ints*), 39
IntTypeBase (*class in malduck.ints*), 40
invert_mask() (*malduck.ints.MetaIntType property*), 40
ipv4() (*in module malduck*), 46
is32bit() (*malduck.pe.PE property*), 21
is64bit() (*malduck.pe.PE property*), 21
is_addr() (*malduck.procmem.procmem.ProcessMemory method*), 12
is_image_loaded_as_memdump() (*malduck.procmem.procmemelf.ProcessMemoryELF method*), 18
is_image_loaded_as_memdump() (*malduck.procmem.procmempe.ProcessMemoryPE method*), 17
is_imm() (*malduck.disasm.Operand property*), 20
is_mem() (*malduck.disasm.Operand property*), 20
is_reg() (*malduck.disasm.Operand property*), 20
is_valid() (*malduck.procmem.procmemelf.ProcessMemoryELF method*), 18
is_valid() (*malduck.procmem.procmempe.ProcessMemoryPE method*), 17
iter_regions() (*malduck.procmem.procmem.ProcessMemory method*), 12

K

keys() (*malduck.yara.YaraMatch method*), 24
keys() (*malduck.yara.YaraMatches method*), 24

L

last() (*malduck.procmem.procmem.Region property*), 16

last_offset() (*malduck.procmem.procmem.Region property*), 17
log() (*malduck.extractor.Extractor property*), 4
log() (*malduck.extractor.extractor.ExtractorBase property*), 7
lznt1() (*in module malduck*), 34

M

malduck.bits (*module*), 37
malduck.compression (*module*), 33
malduck.crypto (*module*), 25
malduck.disasm (*module*), 19
malduck.extractor (*module*), 3
malduck.hash (*module*), 35
malduck.ints (*module*), 39
malduck.pe (*module*), 21
malduck.procmem (*module*), 9
malduck.string (*module*), 43
malduck.yara (*module*), 23
mask() (*malduck.ints.MetaIntType property*), 41
match() (*malduck.yara.Yara method*), 24
matched() (*malduck.extractor.Extractor property*), 5
matched() (*malduck.extractor.extractor.ExtractorBase property*), 7
md5() (*in module malduck*), 35
mem() (*malduck.disasm.Operand property*), 20
Memory (*class in malduck.disasm*), 20
MetaExtractor (*class in malduck.extractor.extractor*), 8
MetaIntType (*class in malduck.ints*), 40
MultipliedIntTypeBase (*class in malduck.ints*), 40

N

needs_elf() (*malduck.extractor.Extractor method*), 4
needs_pe() (*malduck.extractor.Extractor method*), 4
nt_headers() (*malduck.pe.PE property*), 21

O

on_error() (*malduck.extractor.ExtractManager method*), 5
on_error() (*malduck.extractor.Extractor method*), 5
on_error() (*malduck.extractor.ExtractorModules method*), 6
on_extractor_error() (*malduck.extractor.extract_manager.ProcmemExtractManager method*), 6
on_extractor_error() (*malduck.extractor.ExtractManager method*), 5
op1() (*malduck.disasm.Instruction property*), 19
op2() (*malduck.disasm.Instruction property*), 19
op3() (*malduck.disasm.Instruction property*), 20
Operand (*class in malduck.disasm*), 20
optional_header() (*malduck.pe.PE property*), 21
overrides (*malduck.extractor.extractor.ExtractorBase attribute*), 7

P

p16() (*in module malduck*), 46
p2v() (*malduck.procmem.procmem.ProcessMemory method*), 13
p2v() (*malduck.procmem.procmem.Region method*), 17
p32() (*in module malduck*), 46
p64() (*in module malduck*), 46
p8() (*in module malduck*), 46
pack() (*malduck.ints.IntType method*), 40
pack_be() (*malduck.ints.IntType method*), 40
pad() (*in module malduck*), 44
parent (*malduck.extractor.extract_manager.ProcmemExtractManager attribute*), 7
parent (*malduck.extractor.extractor.ExtractorBase attribute*), 7
parse() (*malduck.crypto.aes.PlaintextKeyBlob method*), 31
patchp() (*malduck.procmem.procmem.ProcessMemory method*), 13
patchv() (*malduck.procmem.procmem.ProcessMemory method*), 14
PE (*class in malduck.pe*), 21
pe() (*malduck.procmem.procmempe.ProcessMemoryPE property*), 17
PlaintextKeyBlob (*class in malduck.crypto.aes*), 30
PrivateKeyBlob (*class in malduck.crypto.rsa*), 31
ProcessMemory (*class in malduck.procmem.procmem*), 9
ProcessMemoryELF (*class in malduck.procmem.procmemelf*), 18
ProcessMemoryPE (*class in malduck.procmem.procmempe*), 17
procmem (*in module malduck*), 9
procmemelf (*in module malduck*), 18
ProcmemExtractManager (*class in malduck.extractor.extract_manager*), 6
procmempe (*in module malduck*), 17
PublicKeyBlob (*class in malduck.crypto.rsa*), 31
push_config() (*malduck.extractor.extract_manager.ProcmemExtractManager method*), 7
push_config() (*malduck.extractor.Extractor method*), 5
push_config() (*malduck.extractor.extractor.ExtractorBase method*), 7
push_file() (*malduck.extractor.ExtractManager method*), 5
push_procmem() (*malduck.extractor.extract_manager.ProcmemExtractManager*

method), 7
push_procmem() (malduck.extractor.ExtractManager method), 6
push_procmem() (malduck.extractor.Extractor method), 5
push_procmem() (malduck.extractor.extractor.ExtractorBase method), 8

Q

QWORD (in module malduck), 41

R

rabbit() (in module malduck), 29
rc4() (in module malduck), 29
readp() (malduck.procmem.procmem.ProcessMemory method), 14
readv() (malduck.procmem.procmem.ProcessMemory method), 14
readv_regions() (malduck.procmem.procmem.ProcessMemory method), 14
readv_until() (malduck.procmem.procmem.ProcessMemory method), 14
reg() (malduck.disasm.Operand property), 20
REGEX (malduck.yara.YaraString attribute), 24
regexp() (malduck.procmem.procmem.ProcessMemory method), 15
regexv() (malduck.procmem.procmem.ProcessMemory method), 15
Region (class in malduck.procmem.procmem), 16
resource() (malduck.pe.PE method), 21
resources() (malduck.pe.PE method), 21
rol() (in module malduck.bits), 37
rol() (malduck.ints.IntType method), 40
ror() (in module malduck.bits), 37
ror() (malduck.ints.IntType method), 40
RSA (class in malduck.crypto.rsa), 30
rsa (in module malduck), 30
rules() (malduck.extractor.ExtractManager property), 6

S

scale() (malduck.disasm.Memory property), 20
section() (malduck.pe.PE method), 21
sections() (malduck.pe.PE property), 22
sha1() (in module malduck), 35
sha224() (in module malduck), 35
sha256() (in module malduck), 35
sha384() (in module malduck), 35
sha512() (in module malduck), 35
size() (malduck.disasm.Memory property), 20
store() (malduck.procmem.procmempe.ProcessMemoryPE method), 17
structure() (malduck.pe.PE method), 22

T

TEXT (malduck.yara.YaraString attribute), 24
to_json() (malduck.procmem.procmem.Region method), 17
trim_range() (malduck.procmem.procmem.Region method), 17

U

u16() (in module malduck), 45
u32() (in module malduck), 45
u64() (in module malduck), 45
u8() (in module malduck), 46
UInt16 (class in malduck.ints), 41
uint16() (in module malduck), 44
uint16p() (malduck.procmem.procmem.ProcessMemory method), 15
uint16v() (malduck.procmem.procmem.ProcessMemory method), 15
UInt32 (class in malduck.ints), 41
uint32() (in module malduck), 44
uint32p() (malduck.procmem.procmem.ProcessMemory method), 15
uint32v() (malduck.procmem.procmem.ProcessMemory method), 15
UInt64 (class in malduck.ints), 41
uint64() (in module malduck), 44
uint64p() (malduck.procmem.procmem.ProcessMemory method), 15
uint64v() (malduck.procmem.procmem.ProcessMemory method), 15
UInt8 (class in malduck.ints), 41
uint8() (in module malduck), 45
uint8p() (malduck.procmem.procmem.ProcessMemory method), 15
uint8v() (malduck.procmem.procmem.ProcessMemory method), 15
uleb128() (in module malduck), 43
unhex() (in module malduck), 43
unpack() (malduck.ints.IntType class method), 40
unpack_be() (malduck.ints.IntType class method), 40
unpad() (in module malduck), 44
utf16z() (in module malduck), 43
utf16z() (malduck.procmem.procmem.ProcessMemory method), 16

V

v2p() (malduck.procmem.procmem.ProcessMemory method), 16
v2p() (malduck.procmem.procmem.Region method), 17

validate_import_names() (*malduck.pe.PE method*), 22
validate_padding() (*malduck.pe.PE method*), 22
validate_resources() (*malduck.pe.PE method*), 22
value() (*malduck.disasm.Operand property*), 20

W

weak() (*malduck.extractor.Extractor method*), 4
WORD (*in module malduck*), 41

X

xor() (*in module malduck*), 29

Y

Yara (*class in malduck.yara*), 23
yara_rules (*malduck.extractor.Extractor attribute*), 5
YaraMatch (*class in malduck.yara*), 24
YaraMatches (*class in malduck.yara*), 24
yarap() (*malduck.procmem.procmem.ProcessMemory method*), 16
YaraString (*class in malduck.yara*), 24
yarav() (*malduck.procmem.procmem.ProcessMemory method*), 16