Malduck

CERT Polska

Apr 11, 2024

EXTRACTION TOOLS:

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

Example extractor code for Citadel:

```
from malduck import Extractor

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

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

    @Extractor.string
    def cit_login(self, p, addr, match):
        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)}
```

Decorated methods are always called in order:

- @*Extractor.extractor* methods

- *@Extractor.string* methods
- *@Extractor.rule* methods
- *@Extractor.final* methods

@string

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.

You can optionally provide the actual string identifier as an argument if you don't want to name your method after the string identifier.

Signature of decorated method:

```
@Extractor.string
def string_identifier(self, p: ProcessMemory, addr: int, match:␣
↪YaraStringMatch) -> Config:
    # p: ProcessMemory object that contains matched file/dump representation
    # addr: Virtual address of matched string
    # Called for each "$string_identifier" hit
    ...
```

If you want to use same method for multiple different named strings, you can provide multiple identifiers as *@Extractor.string* decorator argument

Extractor methods should return *dict* object with extracted part of configuration, *True* indicating a match or *False*/*None* when family has not been matched.

For strong methods: truthy values are transformed to *dict* with *{"family": self.family}* key.

New in version 4.0.0: Added *@Extractor.string* as extended version of *@Extractor.extractor*

Parameters

strings_or_method (*str, optional*) – If method name doesn't match the string identifier, pass yara string identifier as decorator argument. Multiple strings are accepted

@extractor

Simplified variant of *@Extractor.string*.

Doesn't accept multiple strings and passes only string offset to the extractor method.

```
from malduck import Extractor

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

    @Extractor.extractor("briankerbs")
    def citadel_found(self, p, addr):
        # Called for each $briankerbs hit
        ...

    @Extractor.extractor
    def cit_login(self, p, addr):
        # Called for each $cit_login1, $cit_login2 hit
        ...
```

@rule

Decorator for rule-based extractor methods, called once for rule match after string-based extraction methods.

Method is called each time when rule with the same identifier as method name has matched.

You can optionally provide the actual rule identifier as an argument if you don't want to name your method after the rule identifier.

Rule identifier must appear in *yara_rules* tuple.

Signature of decorated method:

```
@Extractor.rule
def rule_identifier(self, p: ProcessMemory, matches: YaraMatch) -> Config:
    # p: ProcessMemory object that contains matched file/dump representation
    # matches: YaraMatch object with offsets of all matched strings related␣
↪with the rule
    # Called for matched rule named "rule_identifier".
    ...
```

New in version 4.0.0: Added *@Extractor.rule* decorator

```
from malduck import Extractor

class Evil(Extractor):
    yara_rules = ("evil", "weird")
    family = "evil"

    ...

    @Extractor.rule
    def evil(self, p, matches):
        # This will be called each time evil match.
        # `matches` is YaraMatch object that contains information about
        # all string matches related with this rule.
        ...
```

Parameters

string_or_method (*str, optional*) – If method name doesn't match the rule identifier pass yara string identifier as decorator argument

@final

Decorator for final extractor methods, called once for each single rule match after other extraction methods.

Behaves similarly to the @rule-decorated methods but is called for each rule match regardless of the rule identifier.

Signature of decorated method:

```
@Extractor.rule
def rule_identifier(self, p: ProcessMemory) -> Config:
    # p: ProcessMemory object that contains matched file/dump representation
    # Called for each matched rule in self.yara_rules
    ...
```

```python
from malduck import Extractor

class Evil(Extractor):
    yara_rules = ("evil", "weird")
    family = "evil"

    ...

    @Extractor.needs_pe
    @Extractor.final
    def get_config(self, p):
        # This will be called each time evil or weird match
        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.

Changed in version 4.0.0: Method must be decorated first with *@extractor*, *@rule* or *@final* decorator

```python
from malduck import Extractor

class Evil(Extractor):
    yara_rules = ("evil", "weird")
    family = "evil"

    ...

    @Extractor.weak
    @Extractor.extractor
    def dga_seed(self, p, hit):
        # Even if we're able to get the DGA seed, extractor won't produce config
        # until is_it_really_evil match as well
        dga_config = p.readv(hit, 128)
        seed = self._get_dga_seed(dga_config)
        if seed is not None:
            return {"dga_seed": seed}

    @Extractor.final
    def is_it_really_evil(self, p):
        # If p starts with 'evil', we can produce config
        return p.read(p.imgbase, 4) == b'evil'
```

@needs_pe

Use this decorator for extractors that need PE instance. (p is guaranteed to be `malduck.procmem.ProcessMemoryPE`)

Changed in version 4.0.0: Method must be decorated first with *@extractor*, *@rule* or *@final* decorator

@needs_elf

Use this decorator for extractors that need ELF instance. (p is guaranteed to be `malduck.procmem.ProcessMemoryELF`)

Changed in version 4.0.0: Method must be decorated first with *@extractor*, *@rule* or *@final* decorator.

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

handle_match(*p*, *match*)

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

Called for each rule hit listed in Extractor.yara_rules.

Overriding this method means that all Yara hits must be processed within this method. Ripped configurations must be reported using *push_config()* method.

Parameters

- **p** (`malduck.procmem.ProcessMemory`) – ProcessMemory object
- **match** (`malduck.yara.YaraRuleMatch`) – Found yara matches for currently matched rule

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 exceptions raised by extractor methods.

Parameters

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

overrides = []

Family match overrides another match e.g. citadel overrides zeus

push_config(*config*)

Push partial config (used by *Extractor.handle_match()*)

Parameters
config (*dict*) – Partial config element

push_procmem(*procmem: ProcessMemory*, ***info*)

Push extracted 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_match is called

class malduck.extractor.**ExtractManager**(*modules: ExtractorModules*)

Multi-dump extraction context. Handles merging configs from different dumps, additional dropped families etc.

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

carve_procmem(*p: ProcessMemory*) → List[ProcessMemoryBinary]

Carves binaries from ProcessMemory to try configuration extraction using every possible address mapping.

property config: List[Dict[str, Any]]

Extracted configuration (list of configs for each extracted family)

property extractors: List[Type[*Extractor*]]

Bound extractor modules :rtype: List[Type[*malduck.extractor.Extractor*]]

match_procmem(*p: ProcessMemory*) → YaraRulesetMatch

Performs Yara matching on ProcessMemory using modules bound with current ExtractManager.

on_error(*exc: Exception*, *extractor: Extractor*) → None

Handler for all exceptions raised 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 raised the exception

on_extractor_error(*exc: Exception*, *extractor: Extractor*, *method_name: str*) → None

Handler for all exceptions raised 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 raised the exception

push_file(*filepath: str*, *base: int = 0*) → str | None

Pushes file for extraction. Config extractor entrypoint.

Parameters

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

Returns

Detected family if configuration looks better than already stored one

push_procmem(*p: ProcessMemory*, *rip_binaries: bool = False*) → str | None

Pushes ProcessMemory object for extraction

Parameters

- **p** (malduck.procmem.ProcessMemory) – ProcessMemory object
- **rip_binaries** (*bool (default: False)*) – Look for binaries (PE, ELF) in provided ProcessMemory and try to perform extraction using specialized variants (ProcessMemoryPE, ProcessMemoryELF)

Returns

Detected family if configuration looks better than already stored one

property rules: *Yara*

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

class malduck.extractor.**ExtractorModules**(*modules_path: str | None = None*)

Configuration object with loaded Extractor modules for ExtractManager

Parameters

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

compare_family_overrides(*first: str*, *second: str*) → int

Checks which family supersedes which. Relations can be transitive, so ExtractorModules builds all possible paths and checks the order. If there is no such relationship between families, function returns None.

on_error(*exc: Exception*, *module_name: str*) → None

Handler for all exceptions raised 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 raised the exception

1.2 Internally used classes and routines

class malduck.extractor.extract_manager.**ExtractionContext**(*parent: ExtractManager*)

Single-dump extraction context (single family)

collected_config: Dict[str, Any]

Collected configuration so far (especially useful for "final" extractors)

property config: Dict[str, Any]

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

property family: str | None

Matched family

on_extractor_error(*exc: Exception*, *extractor: Extractor*, *method_name: str*) → None

Handler for all exceptions raised by extractor methods.

Parameters

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

parent

Bound ExtractManager instance

push_config(*config: Dict[str, Any]*, *extractor: Extractor*) → None

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: ProcessMemory*, *_matches: YaraRulesetMatch | None = None*) → None

Pushes ProcessMemory object for extraction

Parameters

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

CHAPTER

TWO

MEMORY MODEL OBJECTS (PROCMEM)

2.1 ProcessMemory (procmem)

malduck.**procmem**

alias of *ProcessMemory*

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

Basic virtual memory representation

Short name: *procmem*

Parameters

- **buf** (*bytes, mmap, memoryview, bytearray or MemoryBuffer 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

```python
from malduck import procmem, procmempe

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

```python
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 owned by this object.

If copy is False (default): invalidates the object.

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

disasmv(*addr*, *size=None*, *x64=False*, *count=None*)

Disassembles code under specified address

Changed in version 4.0.0: Returns iterator instead of list of instructions

Parameters

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

Returns
`List[Instruction]`

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=None*, *length=None*)

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

If offset is None, looks for match from the beginning of memory

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

If addr is None, looks for match from the beginning of memory

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"):
```

(continues on next page)

(continued from previous page)

```
    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=None*, *length=None*)

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

If offset is None, looks for substring from the beginning of memory

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)

If addr is None, looks for substring from the beginning of memory

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 (optional, default is provided by specialized class)*) – Virtual address of region of interest (imgbase)

Return type

ProcessMemory

int16p(*offset*, *fixed=False*)

Read signed 16-bit value at offset.

int16v(*addr*, *fixed=False*)

Read signed 16-bit value at address.

int32p(*offset*, *fixed=False*)

Read signed 32-bit value at offset.

int32v(*addr*, *fixed=False*)

Read signed 32-bit value at address.

int64p(*offset*, *fixed=False*)

Read signed 64-bit value at offset.

int64v(*addr*, *fixed=False*)

Read signed 64-bit value at address.

int8p(*offset*, *fixed=False*)

Read signed 8-bit value at offset.

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*]

property length

Returns length of raw memory contents :rtype: int

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)
    )
```

(continues on next page)

(continued from previous page)

```
    embed_pe = procmem(payload, base=0)
    # 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

Patched address range must be within single region, ValueError is raised otherwise.

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.

Changed in version 3.0.0: Contents of contiguous regions are merged into single string

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*)

Read a chunk of memory until the stop marker

Parameters

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

Return type
bytes

`regexp`(*query*, *offset=None*, *length=None*)

Performs regex on the memory contents (non-region-wise)

If offset is None, looks for match from the beginning of memory

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)

If addr is None, looks for match from the beginning of memory

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=None*, *length=None*, *extended=False*)

Perform yara matching (non-region-wise)

If offset is None, looks for match from the beginning of memory

Changed in version 4.0.0: Added *extended* option which allows to get extended information about matched strings and rules. Default is False for backwards compatibility.

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

- **extended** (*bool (optional, default False)*) – Returns extended information about matched strings and rules

Return type
malduck.yara.YaraMatches

yarav(*ruleset*, *addr=None*, *length=None*, *extended=False*)

Perform yara matching (region-wise)

If addr is None, looks for match from the beginning of memory

Changed in version 4.0.0: Added *extended* option which allows to get extended information about matched strings and rules. Default is False for backwards compatibility.

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
- **extended** (*bool (optional, default False)*) – Returns extended information about matched strings and rules

Return type
`malduck.yara.YaraRulesetOffsets` or `malduck.yara.YaraRulesetMatches` if extended is set to True

class malduck.procmem.procmem.**Region**(*addr: int*, *size: int*, *state: int*, *type_: int*, *protect: int*, *offset: int*)

Represents single mapped region in *ProcessMemory*

contains_addr(*addr: int*) → bool

Checks whether region contains provided virtual address

contains_offset(*offset: int*) → bool

Checks whether region contains provided physical offset

property end: int

Virtual address of region end (first unmapped byte)

property end_offset: int

Offset of region end (first unmapped byte)

intersects_range(*addr: int*, *length: int*) → bool

Checks whether region mapping intersects with provided range

property last: int

Virtual address of last region byte

property last_offset: int

Offset of last region byte

p2v(*off: int*) → int

Physical offset to translation. Assumes that offset is valid within Region. :param off: Physical offset :return: Virtual address

to_json() → Dict[str, int | str | None]

Returns JSON-like dict representation

trim_range(*addr: int*, *length: int* | *None = None*) → *Region* | None

Returns region intersection with provided range :param addr: Virtual address of starting point :param length: Length of range (optional) :rtype: `Region`

v2p(*addr: int*) → int

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 `ProcessMemoryPE`

class `malduck.procmem.procmempe.`**ProcessMemoryPE**(*buf: bytes* | *bytearray* | *mmap* | *MemoryBuffer*, *base: int = 0*, *regions: List[Region]* | *None = None*, *image: bool = False*, *detect_image: bool = False*)

Representation of memory-mapped PE file

Short name: *procmempe*

Parameters

- **buf** (bytes, mmap, memoryview, bytearray or `MemoryBuffer()` object) – A memory object containing the PE to be loaded
- **base** (*int, optional (default: 0)*) – Virtual address of the region of interest (or beginning of buf when no regions provided)
- **image** (*bool, optional (default: False)*) – The memory object is a dump of memory-mapped PE
- **detect_image** (*bool, optional (default: False)*) – Try to automatically detect if the input buffer is memory-mapped PE using some heuristics

File *memory_dump* contains a 64bit memory-aligned PE dumped from address *0x140000000*, in order to load it into procmempe and access the *pe* field all we have to do is initialize a new object with the file data:

```
from malduck import procmempe

with open("memory_dump", "rb") as f:
    data = f.read()

pe_dump = procmempe(buf=data, base=0x140000000, image=True)
print(pe_dump.pe.is64bit)
```

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

```
pe_dump = procmempe.from_file("140000000_1d5bdc3dbe71a7bd", image=True)
print(pe_dump.pe.sections)
```

property imgend: int

Address where PE image ends

is_image_loaded_as_memdump() → bool

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() → bool

Checks whether imgbase is pointing at valid binary header

property pe: *PE*

Related PE object

store() → bytes

Store ProcessMemoryPE contents as PE file data.

Return type
bytes

2.3 ProcessMemoryELF (procmemelf)

malduck.**procmemelf**

alias of *ProcessMemoryELF*

class malduck.procmem.procmemelf.**ProcessMemoryELF**(*buf: bytes | bytearray | mmap | MemoryBuffer, base: int = 0, regions: List[Region] | None = None, image: bool = False, detect_image: bool = 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: ELFFile

Related `ELFFile` object

property imgend: int

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() → bool

Checks whether imgbase is pointing at valid binary header

2.4 CuckooProcessMemory (cuckoomem)

malduck.**cuckoomem**

alias of *CuckooProcessMemory*

class malduck.procmem.cuckoomem.**CuckooProcessMemory**(*buf: bytes | bytearray | mmap | MemoryBuffer*, *base: int | None = None*, ***_*)

Wrapper object to operate on process memory dumps in Cuckoo 2.x format.

2.5 IDAProcessMemory (idamem)

malduck.**idamem**

alias of *IDAProcessMemory*

class malduck.procmem.idamem.**IDAProcessMemory**

ProcessMemory representation operating in IDAPython context

Short name: *idamem*

Initialize by creating the object within IDAPython context and then use like a normal procmem object:

```
from malduck import idamem, xor

ida = idamem()
decrypted_data = xor(b"KEYZ", ida.readv(0x0040D320, 128))
some_wide_string = ida.utf16z(0x402010).decode("utf-8")
```

CHAPTER

THREE

X86 DISASSEMBLER

class malduck.disasm.**Disassemble**

disassemble(*data: bytes*, *addr: int*, *x64: bool = False*, *count: int = 0*) → Iterator[*Instruction*]

Disassembles data from specific address

Changed in version 4.0.0: Returns iterator instead of list of instructions, accepts maximum number of instructions to disassemble

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?
- **count** (*int (default=0)*) – Number of instructions to disassemble

Returns

Returns iterator of instructions

Return type

Iterator[*Instruction*]

class malduck.disasm.**Instruction**(*mnem: str | None = None*, *op1: Operand | None = None*, *op2: Operand | None = None*, *op3: Operand | None = None*, *addr: int | None = None*, *x64: bool = 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: int | None

Instruction address

property op1: *Operand* **| None**

First operand

property op2: *Operand* **| None**

Second operand

property op3: *Operand* **| None**

Third operand

class malduck.disasm.**Operand**(*op: X86Op*, *x64: bool*)

Operand object for single *Instruction*

property is_imm: bool

Is it immediate operand?

property is_mem: bool

Is it memory operand?

property is_reg: bool

Is it register operand?

property mem: *Memory* **| None**

Returns *Memory* object for memory operands

property reg: str | int | None

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: str | int

Returns operand value or displacement value for memory operands

Return type
str or int or None

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

base

Alias for field number 1

disp

Alias for field number 4

index

Alias for field number 3

scale

Alias for field number 2

size

Alias for field number 0

CHAPTER

FOUR

PE WRAPPER

class malduck.pe.**PE**(*data: ProcessMemory | bytes*, *fast_load: bool = False*)

Wrapper around `pefile.PE`, accepts either bytes (raw file contents) or `ProcessMemory` instance.

directory(*name: str*) → Any

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: Any

Dos header

property file_header: Any

File header

property headers_size: int

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: Any

Is it 32-bit file (PE)?

property is64bit: Any

Is it 64-bit file (PE+)?

property nt_headers: Any

NT headers

property optional_header: Any

Optional header

resource(*name: int | str | bytes*) → bytes | None

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: int | str | bytes*) → Iterator[bytes]

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: str | bytes*) → Any

Get section by name

Parameters

name (*str or bytes*) – Section name

property sections: list

Sections

structure(*rva: int*, *format: Any*) → Any

Get internal pefile Structure from specified rva

Parameters

- **rva** – Relative virtual address of structure
- **format** – `pefile.Structure` format (e.g. `pefile.PE.__IMAGE_LOAD_CONFIG_DIRECTORY64_format__`)

Return type

`pefile.Structure`

validate_import_names() → bool

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

validate_padding() → bool

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

validate_resources() → bool

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 strings is undetermined
        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*, *extended=False*, ***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()`
- **extended** (*bool (optional, default False)*) – Returns extended information about matched strings and rules

Return type

`malduck.yara.YaraRulesetOffsets` or `malduck.yara.YaraRulesetMatches` if extended is set to True

class malduck.yara.**YaraString**(*value*, *type=YaraStringType.TEXT*, ***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

malduck.yara.**YaraMatches**

alias of `YaraRulesetOffsets`

malduck.yara.**YaraMatch**

alias of `YaraRuleOffsets`

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: bytes*, *iv: bytes*, *data: bytes*) → bytes

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: bytes*, *iv: bytes*, *data: bytes*) → bytes

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: bytes*, *data: bytes*) → bytes

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: bytes*, *data: bytes*) → bytes

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: bytes*, *nonce: bytes*, *data: bytes*) → bytes

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: bytes*, *nonce: bytes*, *data: bytes*) → bytes

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: bytes*, *data: bytes*) → bytes

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: bytes*, *data: bytes*) → bytes

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 Camellia

Camellia block cipher.

Supported modes: ECB, CBC, CTR, CFB, OFB.

```
from malduck import camellia

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

6.3.1 Camellia-ECB mode

`malduck.camellia.ecb.encrypt(key: bytes, data: bytes) → bytes`

Encrypts buffer using Camellia 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.camellia.ecb.decrypt(key: bytes, data: bytes) → bytes`

Decrypts buffer using Camellia 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

6.3.2 Camellia-CBC mode

`malduck.camellia.cbc.encrypt(key: bytes, iv: bytes, data: bytes) → bytes`

Encrypts buffer using Camellia 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.camellia.cbc.decrypt`(*key: bytes*, *iv: bytes*, *data: bytes*) → bytes

Decrypts buffer using Camellia 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

6.3.3 Camellia-CTR mode

`malduck.camellia.ctr.encrypt`(*key: bytes*, *nonce: bytes*, *data: bytes*) → bytes

Encrypts buffer using Camellia 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.camellia.ctr.decrypt`(*key: bytes*, *nonce: bytes*, *data: bytes*) → bytes

Decrypts buffer using Camellia 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

6.3.4 Camellia-CFB mode

`malduck.camellia.cfb.encrypt`(*key: bytes*, *iv: bytes*, *data: bytes*) → bytes

Encrypts buffer using Camellia algorithm in CFB 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.camellia.cfb.decrypt`(*key: bytes*, *iv: bytes*, *data: bytes*) → bytes

Decrypts buffer using Camellia algorithm in CFB 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

6.3.5 Camellia-OFB mode

`malduck.camellia.ofb.encrypt`(*key: bytes*, *iv: bytes*, *data: bytes*) → bytes

Encrypts buffer using Camellia algorithm in OFB 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.camellia.ofb.decrypt`(*key: bytes*, *iv: bytes*, *data: bytes*) → bytes

Decrypts buffer using Camellia algorithm in OFB 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

6.4 ChaCha20

ChaCha20 stream cipher.

Assumes empty nonce if none given.

```
from malduck import chacha20

key = b'chachaKeyHereNow' * 2
nonce = b'\x01\x02\x03\x04\x05\0x6\0x7'
plaintext = b'data'*16
ciphertext = chacha20.decrypt(key, plaintext, nonce)
```

`malduck.chacha20.`**`encrypt`**(*key: bytes*, *data: bytes*, *nonce: bytes | None = None*) → bytes
Encrypts buffer using ChaCha20 algorithm.

Parameters

- **key** (*bytes*) – Cryptographic key (32 bytes)
- **data** (*bytes*) – Buffer to be encrypted
- **nonce** (*bytes, optional*) – Nonce value (8/12 bytes, defaults to *b"\x00"*8*)

Returns
Encrypted data

Return type
bytes

`malduck.chacha20.`**`decrypt`**(*key: bytes*, *data: bytes*, *nonce: bytes | None = None*) → bytes
Decrypts buffer using ChaCha20 algorithm.

Parameters

- **key** (*bytes*) – Cryptographic key (32 bytes)
- **data** (*bytes*) – Buffer to be decrypted
- **nonce** (*bytes, optional*) – Nonce value (8/12 bytes, defaults to *b"\x00"*8*)

Returns
Decrypted data

Return type
bytes

6.5 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.encrypt(key, plaintext)
```

`malduck.des3.cbc.`**`encrypt`**(*key: bytes*, *iv: bytes*, *data: bytes*) → bytes

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: bytes*, *iv: bytes*, *data: bytes*) → bytes

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

6.6 Salsa20

Salsa20 stream cipher.

Assumes empty nonce if none given.

```
from malduck import salsa20

key = b'salsaFTW' * 4
nonce = b'\x01\x02\x03\x04\x05\0x6\0x7'
```

(continues on next page)

(continued from previous page)

```
plaintext = b'data' * 16
ciphertext = salsa20.decrypt(key, plaintext, nonce)
```

`malduck.salsa20.encrypt`(*key: bytes*, *data: bytes*, *nonce: bytes | None = None*) → bytes

Encrypts buffer using Salsa20 algorithm.

Parameters

- **key** (*bytes*) – Cryptographic key (16/32 bytes)
- **data** (*bytes*) – Buffer to be encrypted
- **nonce** (*bytes, optional*) – Nonce value (8 bytes, defaults to *b"\x00"*8*)

Returns

Encrypted data

Return type

bytes

`malduck.salsa20.decrypt`(*key: bytes*, *data: bytes*, *nonce: bytes | None = None*) → bytes

Decrypts buffer using Salsa20 algorithm.

Parameters

- **key** (*bytes*) – Cryptographic key (16/32 bytes)
- **data** (*bytes*) – Buffer to be decrypted
- **nonce** (*bytes, optional*) – Nonce value (8 bytes, defaults to *b"\x00"*8*)

Returns

Decrypted data

Return type

bytes

6.7 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: bytes*, *data: bytes*, *iv: bytes | None = None*) → bytes

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"\x00" * 16*)

Returns
Encrypted data

Return type
bytes

`malduck.serpent.cbc.decrypt(key: bytes, data: bytes, iv: bytes | None = None) → bytes`

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"\x00" * 16*)

Returns
Decrypted data

Return type
bytes

6.8 Rabbit

Rabbit stream cipher.

```
from malduck import rabbit

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

`malduck.rabbit(key: bytes, iv: bytes, data: bytes) → bytes`

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

6.9 RC4

RC4 stream cipher.

```
from malduck import rc4

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

`malduck.rc4`(*key: bytes*, *data: bytes*) → bytes

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

6.10 XOR

XOR "stream cipher".

```
from malduck import xor

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

`malduck.xor`(*key: int | bytes*, *data: bytes*) → bytes

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

6.11 RSA (BLOB parser)

`malduck.rsa`

alias of *RSA*

class `malduck.crypto.rsa.RSA`

static **export_key**(*n: int*, *e: int*, *d: int | None = None*, *p: int | None = None*, *q: int | None = None*, *crt: int | None = None*) → bytes

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: bytes*) → bytes | None

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

6.12 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() → Tuple[str, bytes] | None

Exports key from structure or returns None if no key was imported

Returns

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

Return type

Tuple[str, bytes]

parse(*buf: BytesIO*) → None

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: bytes*, *headerless: bool = True*) → bytes | None

aPLib decompression

```
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: bytes*) → bytes

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: bytes*) → bytes

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

```
from malduck import lznt1

lznt1(b"°compressedtestdataalot")
```

Parameters
buf (*bytes*) – Buffer to decompress

Return type
bytes

CHAPTER

EIGHT

HASHING ALGORITHMS

8.1 CRC32

malduck.**crc32**(*val: bytes*) → int

Computes CRC32 checksum for provided data

8.2 MD5

malduck.**md5**(*s: bytes*) → bytes

8.3 SHA1

malduck.**sha1**(*s: bytes*) → bytes

8.4 SHA224/256/384/512

malduck.**sha224**(*s: bytes*) → bytes

malduck.**sha256**(*s: bytes*) → bytes

malduck.**sha384**(*s: bytes*) → bytes

malduck.**sha512**(*s: bytes*) → bytes

CHAPTER

NINE

COMMON BITWISE OPERATIONS

9.1 Rotate left/right

malduck.bits.**rol**(*value: int*, *count: int*, *bits: int = 32*) → int

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: int*, *count: int*, *bits: int = 32*) → int

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: int*, *round_to: int*) → int

Rounds value up to provided alignment

malduck.bits.**align_down**(*value: int*, *round_to: int*) → int

Rounds value down to provided alignment

CHAPTER

TEN

FIXED-INTEGER TYPES

10.1 Object properties

class malduck.ints.**IntType**(*value: Any*)

Fixed-size variant of int 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 `int` 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() → bytes

Pack value into bytes with little-endian order

pack_be() → bytes

Pack value into bytes with big-endian order

rol(*other*) → *IntType*

Bitwise rotate left

ror(*other*) → *IntType*

Bitwise rotate right

classmethod **unpack**(*other: bytes*, *offset: int = 0*, *fixed: bool = True*) → *IntType* | int | None

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: bytes*, *offset: int = 0*, *fixed: bool = True*) → *IntType* | int | None

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: int

Mask for sign bit

property mask: int

Mask for potentially overflowing operations

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

malduck.**QWORD**

alias of *UInt64*

malduck.**DWORD**

alias of *UInt32*

malduck.**WORD**

alias of *UInt16*

malduck.**BYTE**

alias of *UInt8*

class malduck.ints.**UInt64**(*value: Any*)

class malduck.ints.**UInt32**(*value: Any*)

class malduck.ints.**UInt16**(*value: Any*)

class malduck.ints.**UInt8**(*value: Any*)

10.3 Int64/Int32/Int16/Int8

class malduck.ints.**Int64**(*value: Any*)

class malduck.ints.**Int32**(*value: Any*)

class malduck.ints.**Int16**(*value: Any*)

class malduck.ints.**Int8**(*value: Any*)

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: T*, *n: int*`)` → Iterator[T]

Yield successive n-sized chunks from s.

`malduck.`**`chunks`**`(`*s: T*, *n: int*`)` → List[T]

Return list of successive n-sized chunks from s.

11.2 asciiz/utf16z

`malduck.`**`asciiz`**`(`*s: bytes*`)` → bytes

Treats s as null-terminated ASCII string

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

`malduck.`**`utf16z`**`(`*s: bytes*`)` → bytes

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: bytes) → bytes`

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

`malduck.unhex(s: str | bytes) → bytes`

`malduck.uleb128(s: bytes) → Tuple[int, int] | None`

Unsigned Little-Endian Base 128

`malduck.base64(s: str | bytes) → bytes`

Base64 encoder/decoder

11.4 Padding (null/pkcs7)

`malduck.pad(s: bytes, block_size: int) → bytes`

Padding PKCS7/NULL

`malduck.unpad(s: bytes) → bytes`

Unpadding PKCS7/NULL

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

`malduck.uint64(other: bytes, offset: int = 0, fixed: bool = True) → IntType | int | None`

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: bytes, offset: int = 0, fixed: bool = True) → IntType | int | None`

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: bytes*, *offset: int = 0*, *fixed: bool = True*) → *IntType* | int | None

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: bytes*, *offset: int = 0*, *fixed: bool = True*) → *IntType* | int | None

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: bytes*, *offset: int = 0*, *fixed: bool = True*) → *IntType* | int | None

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: bytes*, *offset: int = 0*, *fixed: bool = True*) → *IntType* | int | None

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: bytes, offset: int = 0, fixed: bool = True)` → IntType | int | None

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: bytes, offset: int = 0, fixed: bool = True)` → IntType | int | None

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.unpack(other: bytes, size: int | None = None)` → int

Unpacks bigint value from provided buffer with little-endian order

New in version 4.0.0: Use bigint.unpack instead of bigint() method

Parameters

- **other** (*bytes*) – Buffer object containing value to unpack

- **size** (*bytes, optional*) – Size of bigint in bytes

Return type
int

`malduck.bigint.pack(other: int, size: int | None = None)` → bytes

Packs bigint value into bytes with little-endian order

New in version 4.0.0: Use bigint.pack instead of bigint() method

Parameters

- **other** (*int*) – Value to be packed
- **size** (*bytes, optional*) – Size of bigint in bytes

Return type
bytes

`malduck.bigint.unpack_be(other: bytes, size: int | None = None)` → int

Unpacks bigint value from provided buffer with big-endian order

Parameters

- **other** (*bytes*) – Buffer object containing value to unpack
- **size** (*bytes, optional*) – Size of bigint in bytes

Return type
int

`malduck.bigint.pack_be(other: int, size: int | None = None)` → bytes

Packs bigint value into bytes with big-endian order

New in version 4.0.0: Use bigint.pack instead of bigint() method

Parameters

- **other** (*int*) – Value to be packed
- **size** (*bytes, optional*) – Size of bigint in bytes

Return type
bytes

11.6 IPv4 inet_ntoa

`malduck.ipv4(s: bytes | int)` → str | None

Decodes IPv4 address and returns dot-decimal notation

Parameters
s (*int or bytes*) – Buffer or integer value to be decoded as IPv4

Return type
str

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

F

G

H

I

Q

R

S

T

U

V

W

X

Y