# autorop

**Mariusz Skoneczko**

**Apr 03, 2022**

# CONTENTS

Automated solver of classic CTF pwn challenges, with flexibility in mind.

Official documentation can be found at autorop.readthedocs.io.

CHAPTER

ONE

# DISCLAIMER

Do not use this software for illegal purposes. This software is intended to be used in legal Capture the Flag competitions only.

CHAPTER

# TWO

# COMMAND LINE

```
$ autorop
Usage: autorop BINARY [HOST PORT]
```

```
$ autorop tests/bamboofox/ret2libc bamboofox.cs.nctu.edu.tw 11002
[*] '/home/mariusz/Projects/autorop/tests/bamboofox/ret2libc'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[*] Produced pipeline: Classic(Corefile(), OpenTarget(), Puts(False, ['__libc_start_main
↪', 'puts']), Auto(), SystemBinSh())
[*] Pipeline [1/5]: Corefile()
[+] Starting local process 'tests/bamboofox/ret2libc': pid 18833
[*] Process 'tests/bamboofox/ret2libc' stopped with exit code -11 (SIGSEGV) (pid 18833)
...
[*] Switching to interactive mode
Hello!
The address of "/bin/sh" is 0x804a02c
The address of function "puts" is 0xf7e43da0
$ wc -c /home/ctf/flag
57 /home/ctf/flag
```

CHAPTER

# THREE

# API

Importing autorop automatically does a `from pwn import *`, so you can use all of pwntools' goodies.

Central to autorop's design is the pipeline. Most functions take in a `PwnState`, and pass it on to the next function with some attributes changed. `Pipeline` copies* the `PwnState` between each function so mutations are safe. This allows great simplicity and flexibility.

See how the below example neatly manages to "downgrade" the problem from something unique, to something generic that the `Classic` pipeline can handle.

```
from autorop import *

BIN = "./tests/tjctf_2020/stop"


def send_letter_first(tube, data):
    # the binary expects us to choose a letter first, before it takes input unsafely
    tube.sendline("A")
    # send actual payload
    tube.sendline(data)

# create a starting state
s = PwnState(BIN, lambda: process(BIN))
# set an overwriter function, if the buffer overflow input
# is not available immediately
s.overwriter = send_letter_first

# use base classic pipeline, with printf for leaking
pipeline = turnkey.Classic(leak=leak.Printf())
result = pipeline(s)

# switch to interactive shell which we got via the exploit
result.target.interactive()
```

* **Note**: Although most of the attributes are deep-copied, `target` and `_elf` are not.

CHAPTER

FOUR

# INSTALL

1. Install autorop itself. You might want to be in your python virtual environment. After cloning, install with pip:

```
$ git clone https://github.com/mariuszskon/autorop && cd autorop && pip install .
```

2. Make sure corefiles are enabled and are plainly written to the right directory:

```
# sysctl -w kernel.core_pattern=core.%p
```

3. (Optional) Install libc-database into `~/.libc-database` (or your own location then edit `state.libc_database_path`).
4. All done!

## 4.1 autorop

### 4.1.1 autorop package

#### 4.1.1.1 Subpackages

#### autorop.arutil package

#### Submodules

#### autorop.arutil.OpenTarget module

**class** autorop.arutil.OpenTarget.**OpenTarget**

Bases: *autorop.toplevel.Pipe.Pipe*

**__call__**(*state*)

Open a fresh target.

**Parameters**

- **state** (*PwnState*) – The state with the old (if any) `target` and factory for targets
- **target_factory().** –

**Return type** *PwnState*

**Returns** The mutated `PwnState` with a fresh target connection open.

__init__()

Open a fresh target.

This pipe will close the previous target connection and open a new one using `target_factory()`.

### autorop.arutil.addressify module

autorop.arutil.addressify.**addressify**(*data*)

Produce the address from a data leak.

**Parameters** **data** (bytes) – Raw bytes that were leaked.

**Return type** int

**Returns** The address which was part of the leak.

### autorop.arutil.align_call module

autorop.arutil.align_call.**align_call**(*rop*, *func*, *args*)

Align the stack prior to making a rop call to it.

**Parameters**

- **rop** (ROP) – Current rop chain, just before making the call to the function.
- **func** (str) – Symbol name of the function to call.
- **args** (List[int]) – Arguments to pass to the function.

**Return type** ROP

**Returns** Reference to the mutated `rop`, performing the function call ensuring the stack is aligned.

### autorop.arutil.align_rop module

autorop.arutil.align_rop.**align_rop**(*rop*, *n*)

Pad `rop` to `n` words using `ret` instructions.

**Parameters**

- **rop** (ROP) – The rop chain to pad.
- **n** (int) – the minimum size of the rop chain after padding, in words.

**Return type** ROP

**Returns** Reference to the mutated rop chain `rop`, which is padded to be at least `n` bytes long.

### autorop.arutil.debug_requests module

autorop.arutil.debug_requests.**debug_requests**(*r*)

Print debugging information on a HTTP response made with `requests`.

**Parameters** **r** (`Response`) – The response whose contents are to be logged.

**Return type** `None`

### autorop.arutil.leak_helper module

autorop.arutil.leak_helper.**leak_helper**(*state*, *leaker*, *symbols*, *offset=0*)

Leak libc addresses using a leaking function.

This function leaks the libc addresses of `symbols` using rop chain built by `leaker`, placing them in `state.leaks`. `leaker` msut separate leaks using newlines.

**Parameters**

- **state** (*PwnState*) – The current `PwnState` with the following set
  - `target_factory`: Producer of target to exploit.
  - `_elf`: pwntools `ELF` of `state.binary_name`.
  - `overwriter`: Function which writes rop chain to the "right place".
  - `vuln_function`: Name of vulnerable function in binary, which we can return to repeatedly.
- **leaker** (`Callable[[ROP, int], ROP]`) – function which reads arbitrary memory, newline terminated.
- **symbols** (`Iterable[str]`) – what libc symbols we need to leak.
- **offset** (`int`) – offset, in bytes, from the start of the GOT address of each symbol at which to begin leak, treating previous bytes as zeroes (this is helpful if the leaker function terminates on a zero byte)

**Return type** *PwnState*

**Returns**

Mutated `PwnState`, with the following updated

- `target`: The instance of target from which we got a successful leak. Hopefully it can still be interacted with.
- `leaks`: Updated with `"symbol": address` pairs for each function address of libc that was leaked.

### autorop.arutil.load_libc module

autorop.arutil.load_libc.**load_libc**(*state*)

Load the libc specified in the given state into a pwntools' ELF.

**Parameters** **state** (*PwnState*) – The state, with the following set

- libc: Path to target's libc.
- libc_base: Base address of libc, or None if unknown.

**Return type** ELF

**Returns** Loaded ELF of the libc with attributes set as expected.

### autorop.arutil.pad_rop module

autorop.arutil.pad_rop.**pad_rop**(*rop*, *n*)

Append n ret instructions to rop.

**Parameters**

- **rop** (ROP) – The rop chain to pad.
- **n** (int) – The number of ret instructions to pad rop with.

**Return type** ROP

**Returns** Reference to mutated rop chain rop, which has had exactly n ret instructions appended to it.

### autorop.arutil.pretty_function module

autorop.arutil.pretty_function.**pretty_function**(*name*, *args*)

Produce a pretty textual description of a function call.

Produce a string describing a function call. This is of the form: name(args[0], args[1], ...)

**Parameters**

- **name** (str) – Name of function.
- **args** (Iterable[Any]) – The arguments passed to said function.

**Return type** str

**Returns** Textual description of function call to the function name with the provided arguments.

### Module contents

### autorop.assertion package

### Submodules

### autorop.assertion.have_shell module

autorop.assertion.have_shell.**have_shell**(*tube*)

Check if the given tube is a shell, using a simple heuristic.

**Parameters** **tube** (tube) – The connection to check if we have a shell.

**Return type** bool

**Returns** True if the heuristic does think there is a shell, False otherwise.

### Module contents

### autorop.bof package

### Submodules

### autorop.bof.Corefile module

**class** autorop.bof.Corefile.**Corefile**

Bases: *autorop.toplevel.Pipe.Pipe*

**__call__**(*state*)

Transform the given PwnState to have a buffer overflow overwriter.

**Parameters** **state** (*PwnState*) – The current PwnState with the following set

- binary_name: Path to binary.
- bof_ret_offset: (optional) If not None, skips corefile generation step.
- overwriter: Function which writes rop chain to the "right place".

**Return type** *PwnState*

**Returns**

Mutated PwnState, with the following updated

- bof_ret_offset: Updated if it was not set before.
- overwriter: Now calls the previous overwriter but with bof_ret_offset padding bytes prepended to the data given, and reading the same number of lines as were observed at the crash.

**__init__**()

Find offset to the return address via buffer overflow using corefile.

This pipe not only finds the offset from the input to the return address on the stack, but also sets state.overwriter to be a function that correctly overwrites starting at the return address.

You can avoid active corefile generation by setting state.bof_ret_offset yourself - in this case, the state.overwriter is set appropriately.

### Module contents

### autorop.call package

### Submodules

### autorop.call.Custom module

**class** autorop.call.Custom.**Custom**(*func_name*, *args=[]*, *align=False*)

Bases: *autorop.toplevel.Pipe.Pipe*

**__call__**(*state*)

Perform the call on the `target` in `PwnState`.

**Parameters** **state** (*PwnState*) – The current `PwnState`.

**Return type** *PwnState*

**Returns** The same `PwnState`, but with the `state.overwriter` called with the generated rop chain.

**__init__**(*func_name*, *args=[]*, *align=False*)

Call an arbitrary function using rop chain.

Call an arbitrary function using rop chain. This is basically a thin wrapper around using ROP in pwntools.

**Parameters**

- **func_name** (`str`) – Symbol in executable which we can return to.
- **args** (`List[Any]`) – Optional list of arguments to pass to function.
- **align** (`bool`) – Whether the call should be stack aligned or not.

**Returns** Function which takes a `PwnState`, doing the call, and returns reference to the new `PwnState`.

### autorop.call.SystemBinSh module

**class** autorop.call.SystemBinSh.**SystemBinSh**

Bases: *autorop.toplevel.Pipe.Pipe*

**__call__**(*state*)

Call `system("/bin/sh")` on the current `state.target`.

**Parameters** **state** (*PwnState*) – The current `PwnState` with the following set

- `target`: What we want to exploit.
- `_elf`: pwntools `ELF` of `state.binary_name`.
- `libc`: Path to `target`'s libc.
- `libc_base`: Base address of `libc`.
- `vuln_function`: Name of vulnerable function in binary, which we can return to repeatedly.
- `overwriter`: Function which writes rop chain to the "right place".

**Return type** *PwnState*

**Returns** The given `PwnState`.

`__init__()`

Call `system("/bin/sh")` via a rop chain.

Call `system("/bin/sh")` using a rop chain built from `state.libc` and written by `state.overwriter`.

### Module contents

### autorop.leak package

### Submodules

### autorop.leak.Printf module

**class** `autorop.leak.Printf.`**Printf**(*short=False*, *leak_symbols=['__libc_start_main', 'printf']*)

Bases: *autorop.toplevel.Pipe.Pipe*

`__call__`(*state*)

Transform the given state with the results of the leak via `printf`.

**Parameters** **state** (*PwnState*) – The current `PwnState`.

**Return type** *PwnState*

**Returns**

The mutated `PwnState`, with the following updated

- `target`: The fresh instance of target from which we got a successful leak. Hopefully it can still be interacted with.
- `leaks`: Updated with `"symbol": address` pairs for each function address of libc that was leaked.

`__init__`(*short=False*, *leak_symbols=['__libc_start_main', 'printf']*)

Leak libc addresses using `printf`.

This returns a callable which opens a new target, and leaks the addresses of (by default) `__libc_start_main` and `printf` using `printf`, placing them in `state.leaks`.

**Parameters**

- **short** (`bool`) – Whether the attack should be minimised i.e. leak only one address.
- **leak_symbols** (`Iterable[str]`) – What symbols should be leaked.

### autorop.leak.Puts module

class autorop.leak.Puts.**Puts**(*short=False*, *leak_symbols=['__libc_start_main', 'puts']*)

Bases: *autorop.toplevel.Pipe.Pipe*

**__call__**(*state*)

Transform the given state with the results of the leak via `printf`.

**Parameters** **state** (*PwnState*) – The current `PwnState`.

**Return type** *PwnState*

**Returns**

The mutated `PwnState`, with the following updated

- `target`: The fresh instance of target from which we got a successful leak. Hopefully it can still be interacted with.
- `leaks`: Updated with the `"symbol": address` pairs for each function address of libc that was leaked.

**__init__**(*short=False*, *leak_symbols=['__libc_start_main', 'puts']*)

Leak libc addresses using `puts`.

This returns a callable which opens a new target, and leaks the addresses of (by default) `__libc_start_main` and `puts` using `puts`, placing them in `state.leaks`.

**Parameters**

- **short** (`bool`) – Whether the attack should be minimised i.e. leak only one address.
- **leak_symbols** (`Iterable[str]`) – What symbols should be leaked.

**Returns**

Function which takes the state, and returns the mutated `PwnState`, with the following updated

- `target`: The fresh instance of target from which we got a successful leak. Hopefully it can still be interacted with.
- `leaks`: Updated with `"symbol": address` pairs for each address that was leaked.

### Module contents

### autorop.libc package

### Submodules

### autorop.libc.Auto module

class autorop.libc.Auto.**Auto**

Bases: *autorop.toplevel.Pipe.Pipe*

**__call__**(*state*)

Perform the libc acquisition using `state.libc_getter`.

**Parameters** **state** (*PwnState*) – The current `PwnState` with at least the following set

- `libc_getter`: What to use to get libc. This might have its own requirements for attributes set in `state`.

**Return type** *PwnState*

**Returns** Mutated `PwnState`, with updates from `libc_getter`.

**__init__**()

Acquire libc using configured service.

We can programmatically find and download libc based on function address leaks (two or more preferred). This pipe will set `state.libc`, including setting `state.libc.address` for ready-to-use address calculation.

### autorop.libc.Database module

**class** autorop.libc.Database.**Database**

Bases: *autorop.toplevel.Pipe.Pipe*

**__call__**(*state*)

Acquire libc version using local installation of libc-database

**Parameters** **state** (*PwnState*) – The current `PwnState` with the following set

- `leaks`: Leaked symbols of libc.
- `libc_database_path`: Path to libc-database installation.

**Return type** *PwnState*

**Returns**

Mutated `PwnState`, with the following updated

- `libc`: Path to `target`'s libc.
- `libc_base`: Base address of `libc`.

**__init__**()

Acquire libc version using local installation of libc-database

We can programmatically find libc based on function address leaks (two or more preferred). This pipe will set `state.libc`, including setting `state.libc.address` for ready-to-use address calculation.

### autorop.libc.Rip module

**class** autorop.libc.Rip.**Rip**

Bases: *autorop.toplevel.Pipe.Pipe*

**__call__**(*state*)

Acquire libc version using https://libc.rip.

We can programmatically find and download libc based on function address leaks (two or more preferred). This function sets `state.libc`, including setting `state.libc.address` for ready-to-use address calculation.

**Parameters** **state** (*PwnState*) – The current `PwnState` with the following set

- `leaks`: Leaked symbols of libc.

**Return type** *PwnState*

**Returns**

Mutated `PwnState`, with the following updated

- `libc`: Path to `target`'s libc, according to https://libc.rip.
- `libc_base`: Base address of `libc`.

`__init__`()

Acquire libc version using https://libc.rip.

We can programmatically find and download libc based on function address leaks (two or more preferred). This pipe will set `state.libc`, including setting `state.libc.address` for ready-to-use address calculation.

## Module contents

## autorop.toplevel package

## Submodules

## autorop.toplevel.Pipe module

`class autorop.toplevel.Pipe.Pipe`(*params*)

Bases: `object`

`__call__`(*state*)

Call self as a function.

**Return type** *PwnState*

`__init__`(*params*)

Create a "pipe" which operates on a `PwnState`.

Pipes are abstractions that perform a single logical "step" on a `PwnState`, returning the modified `PwnState`.

**Parameters** **params** (`Iterable[Any]`) – The initialisation parameters which describe this pipe.

**Returns** A pipe, which takes and returns a single `PwnState`.

## autorop.toplevel.Pipeline module

`class autorop.toplevel.Pipeline.Pipeline`(**pipes*)

Bases: *autorop.toplevel.Pipe.Pipe*

`__call__`(*state*)

Execute the pipeline.

Execute the saved pipeline sequentially, making a copy of `PwnState` before each function call.

**Parameters** **state** (*PwnState*) – The state to give to the first function.

**Return type** *PwnState*

**Returns** The state returned by the last function.

__init__(**pipes*)

Produce a pipeline to put PwnState through a sequence of Pipes.

Produce a state-copying function pipeline, which executes funcs sequentially, with the output of each function serving as the input to the next function.

The state is copied on every call, for future black magic caching reasons. This means that every function receives its own copy.

**Parameters** **pipes** (*Pipe*) – Functions which operate on the PwnState and return another PwnState.

**Returns** Pipe which puts PwnState through funcs and returns the PwnState returned by the last function.

### autorop.toplevel.PwnState module

class autorop.toplevel.PwnState.**LibcGetter**(**args*, ***kwds*)

Bases: typing_extensions.Protocol

__call__(*state*)

Perform an operation on the state, likely getting the libc based on leaks.

**Return type** *PwnState*

__init__(**args*, ***kwargs*)

class autorop.toplevel.PwnState.**OverwriterFunction**(**args*, ***kwds*)

Bases: typing_extensions.Protocol

__call__(*_OverwriterFunction__t*, *_OverwriterFunction__data*)

Function which writes rop chain to the "right place"

Function which writes rop chain to e.g. the return address. It might be as simple as prepending some padding, or it might need to do format string attacks.

**Parameters**

- **__t** – Where to write the data to.
- **__data** – The data to write at the "right place".

**Return type** Any

**Returns** Anything it likes, the result is ignored.

__init__(**args*, ***kwargs*)

class autorop.toplevel.PwnState.**PwnState**(*binary_name*, *target_factory*, *libc_getter=None*, *vuln_function='main'*, *libc_database_path='/home/docs/.libc-database'*, *libc=None*, *libc_base=None*, *bof_ret_offset=None*, *overwriter=<function default_overwriter>*, *leaks=<factory>*, *target=None*, *_elf=None*)

Bases: object

Class for keeping track of our exploit development.

__init__(*binary_name*, *target_factory*, *libc_getter=None*, *vuln_function='main'*, *libc_database_path='/home/docs/.libc-database'*, *libc=None*, *libc_base=None*, *bof_ret_offset=None*, *overwriter=<function default_overwriter>*, *leaks=<factory>*, *target=None*, *_elf=None*)

**binary_name: str**

Path to the binary to exploit.

**bof_ret_offset: Optional[int] = None**

Offset to return address via buffer overflow.

**leaks: Dict[str, int]**

Leaked symbols of `libc`.

**libc: Optional[str] = None**

Path to `target`'s libc.

**libc_base: Optional[int] = None**

Base address of `target`'s libc.

**libc_database_path: str = '/home/docs/.libc-database'**

Path to local installation of libc-database, if using it.

**libc_getter: Optional[*autorop.toplevel.PwnState.LibcGetter*] = None**

Which libc acquisition service should be used. `libc.Database` is faster, but requires local installation. Automatically set to `libc.Database` if available in `libc_database_path`, otherwise `libc.Rip`.

**overwriter**(*data*)

Function which writes rop chain to the "right place"

**Return type** None

**target: Optional[pwnlib.tubes.tube.tube] = None**

Current target, if any. Produced from calling `target_factory`.

**target_factory: *autorop.toplevel.PwnState.TargetFactory***

What we want to exploit (can be local, or remote). You need to pass in something that can produce a pwntools' tube for the actual target. This may be called multiple times to try multiple exploits.

**vuln_function: str = 'main'**

Name of vulnerable function in binary, which we can return to repeatedly.

class autorop.toplevel.PwnState.**TargetFactory**(**args*, ***kwds*)

Bases: `typing_extensions.Protocol`

**__call__**()

Produce a fresh pwntools' tube of the target.

Create a `tube` of the desired target. This may be called multiple times to try multiple different exploits.

**Return type** tube

**Returns** Instance of target to exploit.

**__init__**(**args*, ***kwargs*)

autorop.toplevel.PwnState.**default_overwriter**(*t*, *data*)

Function which writes data via `t.sendline(data)`

**Return type** None

### autorop.toplevel.constants module

autorop.toplevel.constants.**CLEAN_TIME = 0.5**

pwntools `tube.clean(CLEAN_TIME)`, for removing excess output

autorop.toplevel.constants.**STACK_ALIGNMENT = 16**

Stack alignment, in bytes Ubuntu et al. on x86_64 require it (https://ropemporium.com/guide.html#Common%20pitfalls) and some 32 bit binaries perform `and esp, 0xfffffff0` in `main`

### Module contents

### autorop.turnkey package

### Submodules

### autorop.turnkey.Classic module

**class** autorop.turnkey.Classic.**Classic**(*find=Corefile()*, *leak=Puts(False, ['__libc_start_main', 'puts'])*, *lookup=Auto()*, *shell=SystemBinSh()*)

Bases: *autorop.toplevel.Pipeline.Pipeline*

**__init__**(*find=Corefile()*, *leak=Puts(False, ['__libc_start_main', 'puts'])*, *lookup=Auto()*, *shell=SystemBinSh()*)

Perform a "classic" attack against a binary.

Launch a find-leak-lookup-shell attack against a binary. I made up this term. But it is a common pattern in CTFs.

- Find: Find the vulnerability (e.g. how far we need to write to overwrite return address due to a buffer overflow).
- Leak: Find out important stuff about our context (e.g. addresses of symbols in libc, PIE offset, etc.).
- Lookup: Get data from elsewhere (e.g. download appropriate libc version).
- Shell: Spawn a shell (e.g. via ret2libc, or via syscall).

The default parameters perform a ret2libc attack on a non-PIE/non-ASLR target (at most one of these is fine, but not both), leaking with `puts`. You can set `state._elf.address` yourself and it might work for PIE and ASLR. We use find the libc automatically (likely using libc.rip), and then spawn a shell on the target.

**Parameters**

- **find** (*Pipe*) – "Finder" of vulnerability. *autorop.bof* may be of interest.
- **leak** (*Pipe*) – "Leaker". *autorop.leak* may be of interest.
- **lookup** (*Pipe*) – "Lookup-er" of info. *autorop.libc* may be of interest.
- **shell** (*Pipe*) – Spawner of shell. *autorop.call* may be of interest.

**Returns** Function which takes a `PwnState`, and returns the new `PwnState`.

#### Module contents

#### 4.1.1.2 Submodules

#### 4.1.1.3 autorop.cli module

autorop.cli.**main**()

**Return type** None

#### 4.1.1.4 Module contents

CHAPTER

FIVE

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## a

# INDEX

## Symbols



## A

## B



## C



## D



## H



## L



## M

## O



## P



## R



## S



## T



## V