Reverse Engineering: Cheat Sheet

KU Information Security Club · Jayhackers Reverse Engineering Lab | strings for text | gdb for runtime | Ghidra for logic

FIRST LOOK

file f — type, 32/64-bit, stripped or not
`$ file Chal1`

chmod +x f — make it executable before running
`$ chmod +x Chal1`

./f — run it first: note prompts + messages
`$ ./Chal1`

STRINGS — STATIC TEXT

strings f — print every printable run (4+ chars)
`$ strings Chal1`

-n N — only show strings N+ chars long
`$ strings -n 10 Chal1`

-e l / -e L — look for 16-bit / 32-bit wide strings
`$ strings -e L Chal3`

| wc -l — count how many strings came out
`$ strings Chal1 | wc -l`

| less — scroll the output (/word to search)
`$ strings Chal1 | less`

GREP — FILTER THE NOISE

grep "word" — keep only lines containing word
`$ strings Chal1 | grep "flag"`

grep -i — ignore upper/lowercase
`$ strings Chal1 | grep -i "flag"`

grep -c — count matching lines
`$ strings Chal1 | grep -c "flag"`

grep -v — drop lines that match
`$ strings Chal1 | grep -v "jh_"`

grep -o — print only the matched part
`$ grep -o "flag{[^}]*}"`

*Always quote the pattern. Quotes stop the shell from touching { } * and spaces. Too many decoys? Make the pattern more exact.*

WHICH TOOL WHEN?

strings — answer sits in the file as plain text
gdb — answer is computed while it runs
Ghidra — answer is hidden in the logic or data

INSIDE A BINARY (ELF)

.text — the code: every function's instructions

.rodata — read-only data: string literals like "Correct!"

.data — global variables with starting values (arrays!)

.bss — globals that start as zero

.plt / .got — jumps to library calls: puts, scanf, strlen

.symtab — names like main; gone if stripped

strings only sees bytes that are readable text in a row. Code in .text and numbers in .data hide from it.

x86-64 is little-endian: 0x66 stored in 8 bytes is 66 00 00 00 00 00 00 00.

Ghidra game plan

1. Import, Analyze, open main in the Decompiler
2. Data hunt: Symbol Tree > Labels, decode the array
3. Logic hunt: double-click the check function
4. Follow the switch state by state, fill the blanks
5. Run the binary and test your answer

GDB — START & STOP

gdb -q ./f — load a program (doesn't run it yet)
`$ gdb -q ./Chal2`

run / r — start the program

start — run, then stop at the top of main

kill / k — stop the running program

quit / q — leave gdb

set disassembly-flavor intel — easier asm syntax

GDB — TUI LAYOUTS

layout asm — live disassembly above the prompt

layout regs — add a live register window

focus cmd / fs cmd — arrow keys = history, not asm

Ctrl+X A — toggle the TUI on/off

Ctrl+L — redraw a garbled screen

=> marks the NEXT instruction to run (not yet run).

GDB — BREAKPOINTS

break / b — stop at a function or address
`(gdb) b main` `(gdb) b *0x401234`

info break / i b — list breakpoints + their numbers

delete / d N — remove breakpoint N

GDB — STEPPING

nexti / ni — next instruction, steps OVER calls

stepi / si — next instruction, goes INTO calls

continue / c — run until the next breakpoint

finish / fin — run until this function returns

Enter — repeat the last command (ni ni ni...)

GDB — INSPECT & CHANGE

info registers / i r — dump every register
`(gdb) i r eax edx` (just these)

print / p — print a value (decimal)
`(gdb) p $eax`

p/x p/c p/d — print as hex / char / decimal
`(gdb) p/x $eax`

x/d x/s x/10i — examine memory as int / string / code
`(gdb) x/d $rbp-0xc`

disassemble / disas — dump a whole function
`(gdb) disas main`

set $reg = v — overwrite a register
`(gdb) set $eax = 5`

gdb game plan

1. gdb -q ./Chal2, then layout asm
2. b main, then r
3. ni until => sits on the cmp you care about
4. p $eax (and the other operand): what's being compared?
5. r again and type the value you found

GHIDRA — SETUP

File > New Project — Non-Shared, pick a folder

File > Import (I) — add the binary, keep defaults

Double-click it — opens the CodeBrowser

Analyze? > Yes — defaults are fine, click Analyze

GHIDRA — THE WINDOWS

Program Trees — the sections: .text, .data, .rodata

Symbol Tree — Functions = code, Labels = globals

Listing (middle) — raw asm + data, address by address

Decompiler (right) — C-like version of the function

Click a line in one window and the others jump with it.

GHIDRA — MOVING AROUND

Filter box — bottom of Symbol Tree: type part of a name

G — go to an address or name

Double-click — follow a call, label, or variable

Alt+Left — go back

Show References to — right-click a name: who uses it?

Search > For Strings — everything Ghidra sees as text

GHIDRA — FINDING A GLOBAL ARRAY

Where — Symbol Tree > Labels, or Program Trees > .data

Look for — a name that sounds like storage (vault, key, flag)

Default view — often undefined1[N]: just raw bytes. Expand it with the + arrow

Spot the width — value, then zeros, repeating? Count the gap: 66 + seven 00s = 8-byte entries

Confirm it — code that reads it uses index * 8 → 8 bytes each

Retype (optional) — T, pick any 8-byte type (longlong), size = bytes ÷ 8

See every value — Edit > Tool Options > Listing Fields > Array Options > Elements Per Line = 1

Decode — hex → ASCII with the table on page 2, in order

GHIDRA — READING A FUNCTION

Start at main — Symbol Tree > Functions > main

Follow calls — double-click a call to open it; check_password calls check_step

L / Ctrl+L — rename / retype a variable

Convert > Char — right-click 0x61 to see 'a'

State machines — find the starting state, then follow what each case returns as the next state, not the order the cases are written in

Rename as you go: local_c → state, param_1 → password. Write _ _ _ _ _ _ _ and fill in each index.

x86-64 Assembly Primer *what the instructions in gdb and Ghidra mean*

WHAT IS A REGISTER?

A register is a small variable built into the CPU. Almost every instruction reads or writes one. These are the ones you'll see most:

eax — holds a function's answer after it returns

edi, esi — the 1st and 2nd inputs passed to a function

edx, ecx — extra scratch space (also 3rd / 4th inputs)

rbp, rsp — used to find local variables in memory

rip — the address of the next instruction to run

MEMORY & LOCAL VARIABLES

[rbp-0xc] — the value in memory at (rbp minus 0xc)

Square brackets mean "go to this address and use what's stored there." Local variables live on the stack, so you'll see them as [rbp-something] or [rsp+something].

TWO WAYS TO WRITE IT

gdb and Ghidra can show the same instruction two ways. Both of these mean eax = 5:

Intel — mov eax, 5 destination comes first

AT&T — mov $5, %eax destination comes last

AT&T (gdb's default) puts % before registers and $ before numbers, and writes [rbp-0xc] as -0xc(%rbp). To switch gdb to Intel: set disassembly-flavor intel

COMMON INSTRUCTIONS

Written Intel style: the first thing listed is what changes.

mov a, b — copy b into a
`mov eax, 5` → eax = 5

movzx a, b — copy one small value (like a char) into a bigger register, filling the rest with 0s
`movzx eax, BYTE PTR [rdi]` → eax = first char of a string

lea a, [b] — put an address into a (a pointer, not the value there)
`lea rdi, [rip+0xd81]` → rdi = address of a string

add a, b — add b to a
`add eax, 1` → eax = eax + 1

sub a, b — subtract b from a
`sub eax, 2` → eax = eax − 2

imul a, b — multiply a by b
`imul eax, 3` → eax = eax × 3

xor a, a — set a to 0 (a common shortcut)
`xor eax, eax` → eax = 0

push a — save a copy of a onto the stack

pop a — take the top value off the stack and put it in a

call f — run function f, then come back to the next line
`call puts@plt` → puts(...)

ret — finish this function and go back to whoever called it

nop, endbr64 — do nothing: safe to ignore

name@plt = a library function like puts, scanf, or strlen.

WHAT IS A JUMP?

The CPU normally runs instructions top to bottom. A jump tells it to continue from a different address instead. That is how if statements and loops work in assembly.

Most jumps come right after a cmp. If the condition is true, the CPU jumps. If not, it just runs the next line.

COMPARING & JUMPING

cmp a, b — compare a and b; the next jump uses the result

test a, a — check whether a is 0

jmp X — always jump to X

je X — jump to X if they were equal

jne X — jump to X if they were NOT equal

jg / **jl X** — jump if a was greater / less than b

jge / **jle X** — jump if greater-or-equal / less-or-equal

A jump to an EARLIER address means a loop.

How an if statement looks

C: if (input == pin) { puts("Completed!"); }
cmp eax, edx compare the PIN and the input
jne fail not equal → skip ahead to fail
call puts equal → print "Completed!"
Note: the jump checks the opposite (jne for ==) because it skips the code inside the if.

ASCII (hex)

A–Z 0x41–0x5a a–z 0x61–0x7a
0–9 0x30–0x39 space 0x20
{ } 0x7b 0x7d _ 0x5f ! 0x21
Tip: lowercase = uppercase + 0x20