Intro to x86 Part 2: Writing, Compiling and Analyzing x86

Updated slides 2021/2022

Slides derived from content made by Xeno Kovah – 2009/2010

xkovah at gmail

All materials is licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/

You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

Intel vs. AT&T Syntax

- Intel: Destination <- Source(s)
  - Windows. Think algebra or C: y = 2x + 1;
  - mov ebp, esp
  - add esp, 0x14 ; (esp = esp + 0x14)
- AT&T: Source(s) -> Destination
  - *nix/GNU. Think elementary school: 1 + 2 = 3
  - mov %esp, %ebp
  - add $0x14,%esp
  - So registers get a % prefix and immediates get a $
- By default gdb displays in ATT syntax, however most gdb tools for RE/VR display in Intel syntax, as well as many writeups.
- It's important to be versed in both, so you're prepared to encounter either format.

Intel vs AT&T Syntax 2

- In my opinion the hardest-to-read difference is for r/m32 values
- For intel it's expressed as

  `[base + index*scale + disp]`
- For AT&T it's expressed as

  `disp(base, index, scale)`
- Examples:
  - call DWORD PTR [ebx+esi*4-0xe8]
  - call *-0xe8(%ebx,%esi,4)

  - mov eax, DWORD PTR [ebp+0x8]
  - mov 0x8(%ebp), %eax

  - lea eax, [ebx-0xe8]
  - lea -0xe8(%ebx), %eax

Intel vs AT&T Syntax 3

- For instructions which can operate on different sizes, the mnemonic will have an indicator of the size.
  - movb - operates on bytes
  - mov/movw - operates on word (2 bytes)
  - movl - operates on “long” (dword) (4 bytes)
- Intel does indicate size with things like “mov dword ptr [eax], but it’s just not in the actual mnemonic of the instruction

Syscalls- x86

- *Syscalls* offer a method to invoke and use functionality in the operating system.

Place syscall number in eax

↓

Load parameters in registers

↓

Invoke with “INT 0x80”

Parameter #	Register
Parameter 1	EBX
Parameter 2	ECX
Parameter 3	EDX
Parameter 4	ESI
Parameter 5	EDI
Parameter 6	EBP

- 2 pretty important syscalls for us in exploit world- system() and execve()!
- NOTE: The calling convention for syscalls are different on other architectures, this would even include x64 which uses the SYSCALL instruction instead of INT 0x80.

Resource: https://securityboulevard.com/2021/05/linux-x86-assembly-how-to-build-a-hello-world-program-in-nasm/

How to Find Information on Syscalls?

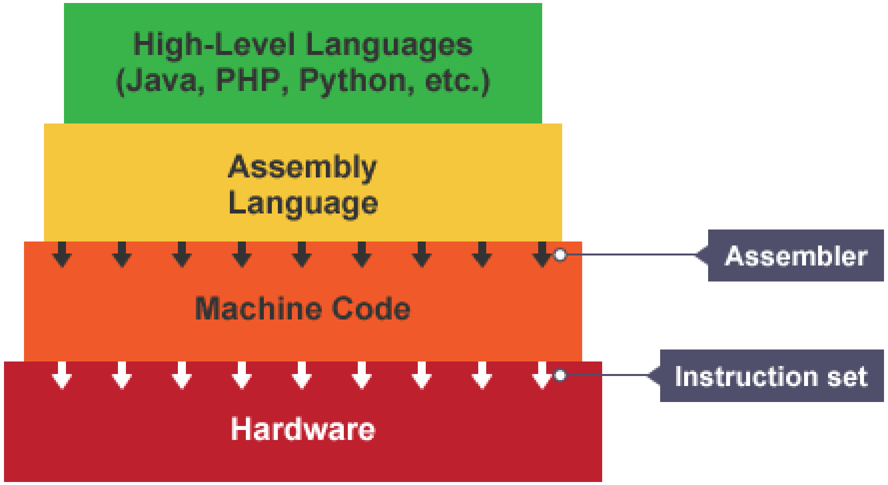
- We need to load a syscall number into EAX for the syscall we want to invoke, but how do we know which numbers relate to what syscall? There are several ways to obtain this information. Here are a few methods:

- Linux Kernel Headers
  - The most absolute source of truth, but also not the easiest method to get what you're looking for
- Web resources
  - https://syscalls.w3challs.com/ is an amazing resource for this
  - Has a table view of syscalls, the parameters it expects, and the register configuration
  - They have several architectures and OSes covered here
  - https://syscalls.w3challs.com/?arch=x86 is the one to look at for Linux 32-bit x86 syscalls.

Resource: https://securityboulevard.com/2021/05/linux-x86-assembly-how-to-build-a-hello-world-program-in-nasm/

Assembling

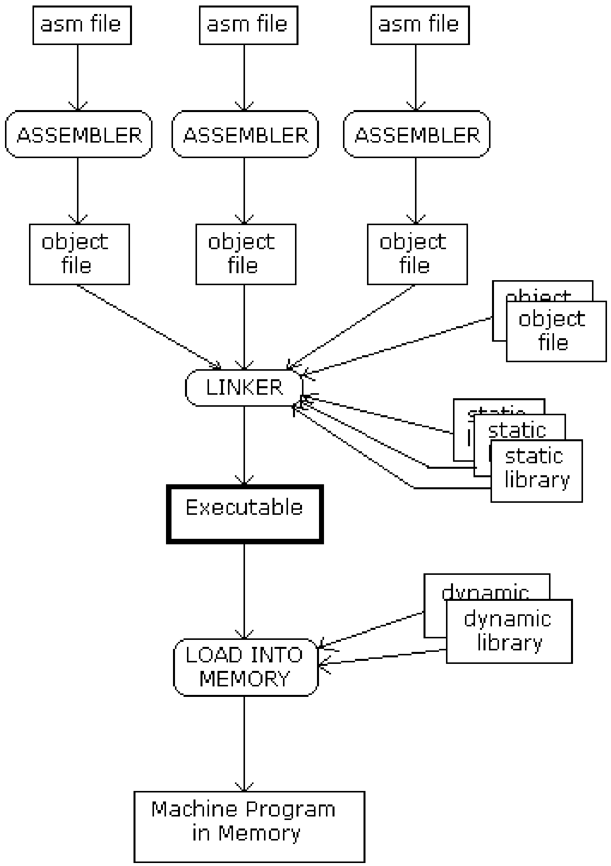
- Assemblers
  - Used to compile assembly language into machine code, aka object files
  - Nasm- Our main focus cus I like Intel ☺
    - ".asm" extension
    - Intel syntax
  - As/GCC
    - ".s" extension
    - Usually in AT&T syntax, so not used as often
    - Can make it use Intel syntax like so:
      - gcc -S -masm=intel test.c



Resources:
https://cs.lmu.edu/~ray/notes/x86assembly/,
https://www.bbc.co.uk/bitesize/guides/zgmpr82/revision/2

Linking

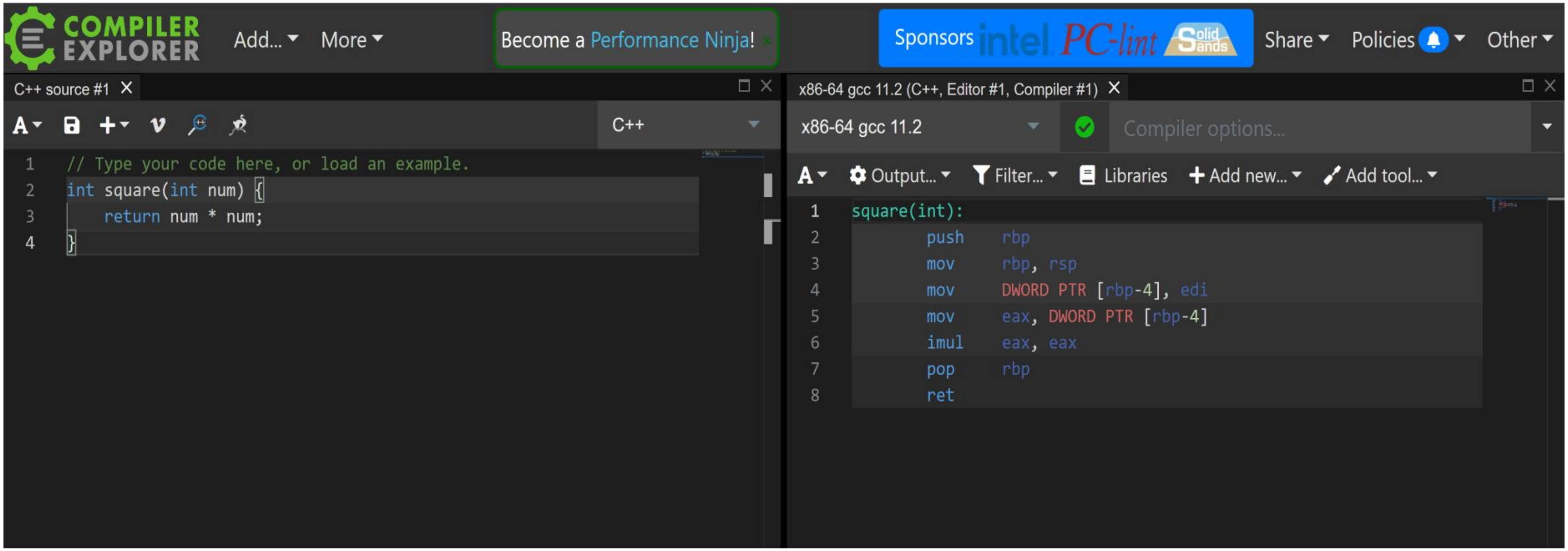
- Linkers
  - Used to link together all the object files created from the assembly stage to form an executable
  - There are 2 types of linking
    - Dynamic linking- Shareable libraries are simply referenced at compile time to be loaded later.
      - Requires less space for the executable and multiple programs can share one reference to the required library code
    - Static linking- All object files required during execution, including shareable libraries, are included during compile time
      - Makes the binary larger but more portable. Since all libraries are included with the executable, this avoids errors/failures related to bad library references between machines.
  - Ld is the tool we will be using to link our objects



https://www.geeksforgeeks.org/linker/

Godbolt

- If you are interested in trying out other ways to compile assembly or other archtiectures you can visit godbolt.org!
- You can provide it a C/C++ program and it will output the assembly for that code
- This is a helpful tool for learning other architectures!



x86 Program Layout

```
global _start ; this tells the linker where to go when our program starts

section .data ; this is where we would put strings and other variables
         <variables>

section .text ; this is the section where we put our instructions
_start: ; _start label indicating what instructions we begin executing at
         <instructions> ; where we put assembly we want to run from start
```

- You can declare strings in the .data section using the following format
  - `<string_label>: db “string contents”`
  - Refer to strings by their label when you need to reference them in your assembly
  - db = data in bytes, 8 bits
    - Other data types (dd- double word, 32 bits, dw- word, 16 bits)
- You can use semicolons to write comments next to your lines of assembly!

Build x86 using nasm

- After we write a .asm file how do we get it to execute?
- Commands for building with nasm
  - nasm -f elf32 exit_nasm.asm
    - You can see other architectures nasm will compile for by passing the “–hf” option
  - ld -m elf_i386 exit_nasm.o -o exit_nasm
  - Check your work!
    - objdump -d -M intel exit_nasm
      - Option “-d” tells objdump you want to see the disassembly
      - the “-M” option allows you to specify the syntax you would like to see your dissassembly in. Since we are writing our assembly in intel, this would be our preferred output syntax for comparison!

Example exit(0)

Let’s write some assembly in a .asm file that will only do the exit syscall!

```
global _start

section .data ; this is where we would put strings etc.
exit_code: db 0 ; change this value to change the exit code

section .text ; this is the section where we put our instructions
_start: ; the starting point for our code to begin executing
         mov al, 1 ; move syscall number into eax
         mov ebx, [exit_code] ; load error code as param 1
         int 0x80 ; interrupt/trap to call the syscall
```

- Open a file with a .asm extension and copy the code above
- Compile using nasm commands
- After linking your executable, run it!
- You can check the exit code by typing “$?” and hitting enter in the terminal
- Now change the value of `exit_code` loaded into ebx and check your exit code again! Did it change?

Strace/Ltrace

- Ltrace traces function calls to methods defined in other libraries imported to your binary
  - Ltrace gives you the order in which library functions are executing, so that can be helpful to see the flow of the program
  - Sometimes you can use this tool to get a list of functions and their addresses for which you might like to overwrite for your exploit
    - there are some security mitigations that might limit this approach
- Strace traces system calls and signals that are executing/occurring in your binary
  - You can use strace to do some reversing on binaries doing things with networking and file paths to get a big picture of what the binary is trying to do when it interacts with the outside world
    - I’ve used this to troubleshoot if my exploit was invoking a shell with system()/execve()

File- command to see information stored file header

- Different file types have associated magic bytes at the front
  - Past CTFs have used this to obfuscate the actual contents of files
- Use this command to get an idea of what you're up against
  - What architecture is the program written in?
    - ARM? X86?
    - 32 bit or 64 bit?
  - Is the file stripped? AKA does it contain debugging symbols?
    - Why is this important? Makes it easier to reverse a binary if you can get an idea of what the functions are doing because of their names
    - Symbols make it easier to reverse and make the file larger- symbols can be removed with the tool “strip”
  - Is the file statically/dynamically linked?
    - Sometimes static linking can make reversing a little more difficult if symbols are stripped
      - if dynamically linked you can at least get an idea of what dynamic functions a binary is calling, you can use tool “ldd” to see the libraries imported
      - IDA Flirt keeps can help you identify these functions though

objdump - display information from object files

- Where ”object file” can be an intermediate file created during compilation but before linking, or a fully linked executable
  - For our purposes means any ELF file - the executable format standard for Linux
- The main thing we care about is -d to disassemble a file.
- Can override the output syntax with “-M intel”
  - Good for getting an alternative perspective on what an instruction is doing, while learning AT&T syntax

hexdump & xxd

- Sometimes useful to look at a hexdump to see opcodes/operands or raw file format info
- hexdump, hd - ASCII, decimal, hexadecimal, octal dump
  - hexdump -C for “canonical” hex & ASCII view
  - Use for a quick peek at the hex
    - Even quicker peek, say you just want to look at the header of the file for “magic bytes”, pipe output to “less”
      - Hexdump exit_nasm | less
- xxd - make a hexdump or do the reverse
  - Use as a quick and dirty hex editor
  - xxd exit_nasm > exit.dump
  - Edit hello.dump
  - xxd -r exit.dump > exit_nasm

Homework 1: Lab0

- Go into the lab0, compile and run the following commands on the binary file exit_nasm:
  - file
  - strace
  - objdump
  - hexdump (pipe to less)
  - xxd
    - Create a hexdump then reverse it back to a binary

Strace

- Strace traces system calls and signals that are executing/occurring in your binary
- Run the command strace with your binary to see what syscalls your program is calling Command “strace ./exit_nasm”
  - What do you see?

```
user@ubuntu:~/Documents/assembly_labs/lab0/nasm$ strace ./exit_nasm
execve("./exit_nasm", ["./exit_nasm"], 0x7fffe79b9a80 /* 52 vars */) = 0
strace: [ Process PID=198085 runs in 32 bit mode. ]
exit(3)                                 = ?
+++ exited with 3 +++
user@ubuntu:~/Documents/assembly_labs/lab0/nasm$
```

objdump -d exit_nasm

```
exit_nasm:     file format elf32-i386


Disassembly of section .text:

08049000 <_start>:
 8049000:   b0 01                  mov    $0x1,%al
 8049002:   8b 1d 00 a0 04 08      mov    0x804a000,%ebx
 8049008:   cd 80                  int    $0x80
```

objdump -d -M intel exit_nasm

```
exit_nasm:     file format elf32-i386


Disassembly of section .text:

08049000 <_start>:
 8049000:   b0 01               mov    al,0x1
 8049002:   8b 1d 00 a0 04 08   mov    ebx,DWORD PTR ds:0x804a000
 8049008:   cd 80               int    0x80
```

Quick GDB Overview

- We are using pwndbg as a gdb plugin!
  - Pwndbg- “GDB plug-in that makes debugging with GDB suck less, with a focus on features needed by low-level software developers, hardware hackers, reverse-engineers and exploit developers.”
  - Find out more information about pwndbg here:
    - https://github.com/pwndbg/pwndbg
    - https://blog.xpnsec.com/pwndbg/
- Run the command “gdb ./exit_nasm” in a terminal in your dev machine. The pwndbg plugin will automatically be loaded!
- Run the program to the first instruction and break by running the command “starti”.
  - This will stop the program on your first instruction under the _start label

Quick GDB Overview

“Starti” command breaks on the first mov instruction for our exit() syscall instructions

```
user@ubuntu:~/Documents/assembly_labs/lab0/nasm$ gdb ./exit_nasm
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
pwndbg: loaded 190 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./exit_nasm...
(No debugging symbols found in ./exit_nasm)
pwndbg> starti
Starting program: /home/user/Documents/assembly_labs/lab0/nasm/exit_nasm

Program stopped.
0x08049000 in _start ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
─────────────────────────────────[ REGISTERS ]─────────────────────────────────
 EAX  0x0
 EBX  0x0
 ECX  0x0
 EDX  0x0
 EDI  0x0
 ESI  0x0
 EBP  0x0
 ESP  0xffffd150 ◂— 0x1
 EIP  0x8049000 (_start) ◂— mov    al, 1
────────────────────────────────────────────────────────────────────────────────
 ► 0x8049000 <_start>       mov    al, 1
   0x8049002 <_start+2>     mov    ebx, dword ptr [exit_code]    <0x804a000>
   0x8049008 <_start+8>     int    0x80
   0x804900a                add    byte ptr [eax], al
   0x804900c                add    byte ptr [eax], al
   0x804900e                add    byte ptr [eax], al
   0x8049010                add    byte ptr [eax], al
   0x8049012                add    byte ptr [eax], al
   0x8049014                add    byte ptr [eax], al
   0x8049016                add    byte ptr [eax], al
   0x8049018                add    byte ptr [eax], al
────────────────────────────────────────────────────────────────────────────────
00:0000│ esp 0xffffd150 ◂— 0x1
01:0004│     0xffffd154 —▸ 0xffffd30c ◂— '/home/user/Documents/assembly_labs/lab0/nasm/exit_nasm'
02:0008│     0xffffd158 ◂— 0x0
03:000c│     0xffffd15c —▸ 0xffffd343 ◂— 'SHELL=/bin/bash'
04:0010│     0xffffd160 —▸ 0xffffd353 ◂— 'SESSION_MANAGER=local/ubuntu:@/tmp/.ICE-unix/3793,unix/ubuntu:/tmp/.ICE-unix/3793'
05:0014│     0xffffd164 —▸ 0xffffd3a5 ◂— 'QT_ACCESSIBILITY=1'
06:0018│     0xffffd168 —▸ 0xffffd3b8 ◂— 'COLORTERM=truecolor'
07:001c│     0xffffd16c —▸ 0xffffd3cc ◂— 'XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg'
────────────────────────────────────────────────────────────────────────────────
 ► f 0 0x8049000 _start
────────────────────────────────────────────────────────────────────────────────
pwndbg> 
```

Register contents

Current instruction and instructions to follow

Visualization ofthe stack

GDB

- We can examine memory at the exit code variable in our text section at address 0x804a000
- You can examine memory in bytes, words, in hex, as strings. You can also decide how many of each type you would like to print.
- Format: x/<quantity><type> <location>
  - Type you can have:
    - s- string
    - x- hex
    - d- dword
    - w- word
    - i- instructions
      - It will try to disassemble at the address you provide
    - You can find other types here: https://sourceware.org/gdb/onlinedocs/gdb/Memory.html
  - Location can be an address or a register

```
pwndbg> x/10x 0x804a000
0x804a000:      0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804a008:      0x00    0x00
pwndbg> x/10b 0x804a000
0x804a000:      0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804a008:      0x00    0x00
pwndbg> x/20x 0x804a000
0x804a000:      0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804a008:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804a010:      0x00    0x00    0x00    0x00
pwndbg> x/20w 0x804a000
0x804a000:      0x00000003      0x00000000      0x00000000      0x00000000
0x804a010:      0x00000000      0x00000000      0x08049000      0x00000000
0x804a020:      0x00010003      0x00000000      0x0804a000      0x00000000
0x804a030:      0x00020003      0x00000001      0x00000000      0x00000000
0x804a040:      0xfff10004      0x0000000f      0x0804a000      0x00000000
pwndbg> x/20d 0x804a000
0x804a000:      3       0       0       0
0x804a010:      0       0       134516736       0
0x804a020:      65539   0       134520832       0
0x804a030:      131075  1       0       0
0x804a040:      -983036 15      134520832       0
pwndbg> x/20s 0x804a000
0x804a000:      "\003"
0x804a002:      ""
0x804a003:      ""
0x804a004:      ""
0x804a005:      ""
0x804a006:      ""
0x804a007:      ""
0x804a008:      ""
0x804a009:      ""
0x804a00a:      ""
0x804a00b:      ""
0x804a00c:      ""
0x804a00d:      ""
0x804a00e:      ""
0x804a00f:      ""
0x804a010:      ""
0x804a011:      ""
0x804a012:      ""
0x804a013:      ""
0x804a014:      ""
pwndbg>
```

```
pwndbg> x/20x $esp
0xffffd150:     0x01    0x00    0x00    0x00    0x0c    0xd3    0xff    0xff
0xffffd158:     0x00    0x00    0x00    0x00    0x43    0xd3    0xff    0xff
0xffffd160:     0x53    0xd3    0xff    0xff
```

Quick GDB Overview

- Useful gdb commands
  - h[elp]- internal navigation of available commands
  - s- step over (don’t go inside function calls, go t the next instruction following the return of the function)
  - si- step into (step inside a function call)
  - c[ontinue]- run until the next breakpoint or until the program ends
  - disass[embly] <location>- print the disassembly for a given location (you can provide an address or function name)
  - i[nfo] b- print breakpoints
  - B[reak] <location>- set a breakpoint on a location (to break on an address the address needs to be proceeded by a “*”
    - Example- b *0x8049000
  - Backtrace- see the call stack, what functions were called to get to the instruction you are on currently
- Run some exploratory commands
  - vmmap- virtual and physical memory analysis tool for processes
  - checksec- see security features of the binary (we will learn about these features later in the course!)
  - Any others?

GDB Other

- Other gdb settings
  - “set disassembly-flavor <syntax>” - use intel syntax rather than AT&T or vice versa
    - Intel is usually the default for RE/VR specific gdb plugins
- Other gdb plugins
  - PEDA- https://github.com/longld/peda
  - GEF- https://github.com/hugsy/gef
  - To allow switching to different assembly plugins
    - https://infosecwriteups.com/pwndbg-gef-peda-one-for-all-and-all-for-one-714d71bf36b8

HW: Let's Write a Hello World

- Try building an assembly program to print Hello World to your screen!
  - Hint: use write syscall for printing!
- Can you figure out a way to get the write syscall to print us a newline in our assembly?
- Resources if you need hints!
  - https://jameshfisher.com/2018/03/10/linux-assembly-hello-world/
  - https://securityboulevard.com/2021/05/linux-x86-assembly-how-to-build-a-hello-world-program-in-nasm/

```
int main(){
        int size_hello = 12;
        char* hello = “Hello World!”;
        write(1, goodbye, size_goodbye);
        exit(0);
}
```

HW: Try Hello World Variations

- Take the C code in the last slide and compile with GCC (32 bit) then compare your assembly program with your C program
  - Run strace on both. Any differences in syscalls?
  - Run objdump on both. What differences do you see?
  - Run file on both. Any differences?
- Lab1: Write a helloworld with no jumps or calls. Make sure to execute the exit() syscall!
- Lab3: Try modifying your helloworld to create a call to exit using the call instruction!