CSC 405
Writing Assembly and Binary Patching

Aleksandr Nahapetyan
anahape@ncsu.edu
(Slides adapted from Dr. Kapravelos)

The Netwide Assembler (NASM)

```
section .text
  global _start

  _start:
```

As mentioned in our last lecture, Assembly level programs can be broken down into three distinct **sections**

`.text` contains the actual logic of the program

The Netwide Assembler (NASM)

```
section .text
  global _start

  _start:
```

One thing this section also includes is an entry point for where the code actually begins

This is handled with the global **`_start`**

The Netwide Assembler (NASM)

```
section .bss
  ; variables

section .text
  global _start        ; entry point for program

  _start:              ; starting point
```

Next the **block starting symbol** (`.bss`) section stores the variables that may / may not change during the execution of the program

The Netwide Assembler (NASM)

```
section .bss
  ; variables

section .data
  ; constants

section .text
  global _start       ; entry point for program

  _start:             ; starting point
```

Finally, the **.data** section handles constants that will not change

The Netwide Assembler (NASM)

Let's say we want to print "Hello World" in Assembly…

Our first task is to design a **label** for the String

```
section .data
  hello:

section .text
  global _start       ; entry point for program

  _start:             ; starting point
```

The Netwide Assembler (NASM)

We can use **define byte** (or **db**) to define the String into memory

The `10` afterwards refers to the decimal notation for a **new line**
Without a newline character (10) the shell prompt is being displayed immediately after the string

```
section
  hello: db "Hello World\n"

section .text
  global _start       ; entry point for program

  _start:             ; starting point
```

The Netwide Assembler (NASM)

We could have also omitted the **db** command and placed the characters of the String as raw hexadecimal values

```
section
  hello: 48 65 6C 6C 6F 20 57 6F 72 6C 64 0A

secti
  global _start        ; entry point for program

  _start:              ; starting point
```

The Netwide Assembler (NASM)

```
section .bss
  ; variables

section .data
  hello: db "Hello World", 10

section .text
  global _start ; must be declared for linker (ld)

_start:         ; starting point
```

Now it's time to actually print "Hello World"

The Netwide Assembler (NASM)

```
section .data
  hello: db "Hello World", 10

section .text
  g                                    r program
  _start:                    ; starting point
    mov rax, 1               ; sys_write
```

We'll start by moving **1** into the **rax** general register

1 corresponds in Linux to the **sys_write** command

The Netwide Assembler (NASM)

```
section .data
  hello: db "Hello World", 10

section .text
  global _start
_start:
  mov rax, 1        ; sys_write
```

Note, you'll often see `rax`, `eax`, and `ax` but they all refer to the same thing, only with a smaller bit space

rax	64-bit general register
eax	32-bit general register
ax	16-bit general register

The Netwide Assembler (NASM)

```
section .data
  hello: db "Hello World", 10

section .text
  global _start

_start:
  mov rax, 1        ; sys_write
```

Likewise, we can place other integers into rax to execute different system calls

0	sys_read
1	sys_write
2	sys_open
...	...
41	sys_socket
...	...

https://filippo.io/linux-syscall-table/

The Netwide Assembler (NASM)

```
section .data
  hello: db "Hello World", 10

section .text
  g                                          r program
  _start:                    ; starting point
    mov rax, 1               ; sys_write
    mov rdi, 1               ; stdout
```

Then, we'll write **1** to the **rdi** general register

This time, **1** in **rdi** corresponds to the terminal's standard output

The Netwide Assembler (NASM)

```
section .data
  hello: db "Hello World", 10

section .text
  global _start

_start:                 ; starting point
  mov rax, 1            ; sys_write
  mov rdi, 1            ; stdout
```

Since I'm using rax to establish sys_write, rdi can be viewed as the **argument** for sys_write

Arch/ABI	arg1	arg2	arg3	arg4	arg5	arg6	arg7	Notes
alpha	a0	a1	a2	a3	a4	a5	-	
arc	r0	r1	r2	r3	r4	r5	-	
arm/OABI	r0	r1	r2	r3	r4	r5	r6	
arm/EABI	r0	r1	r2	r3	r4	r5	r6	
arm64	x0	x1	x2	x3	x4	x5	-	
blackfin	R0	R1	R2	R3	R4	R5	-	
i386	ebx	ecx	edx	esi	edi	ebp	-	
ia64	out0	out1	out2	out3	out4	out5	-	
loongarch	a0	a1	a2	a3	a4	a5	a6	
m68k	d1	d2	d3	d4	d5	a0	-	
microblaze	r5	r6	r7	r8	r9	r10	-	
mips/o32	a0	a1	a2	a3	-	-	-	1
mips/n32,64	a0	a1	a2	a3	a4	a5	-	
nios2	r4	r5	r6	r7	r8	r9	-	
parisc	r26	r25	r24	r23	r22	r21	-	
powerpc	r3	r4	r5	r6	r7	r8	r9	
powerpc64	r3	r4	r5	r6	r7	r8	-	
riscv	a0	a1	a2	a3	a4	a5	-	
s390	r2	r3	r4	r5	r6	r7	-	
s390x	r2	r3	r4	r5	r6	r7	-	
superh	r4	r5	r6	r7	r0	r1	r2	
sparc/32	o0	o1	o2	o3	o4	o5	-	
sparc/64	o0	o1	o2	o3	o4	o5	-	
tile	R00	R01	R02	R03	R04	R05	-	
x86-64	rdi	rsi	rdx	r10	r8	r9	-	
x32	rdi	rsi	rdx	r10	r8	r9	-	
xtensa	a6	a3	a4	a5	a8	a9	-	

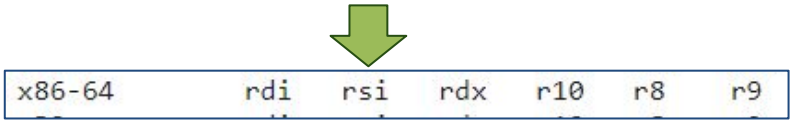
https://man7.org/linux/man-pages/man2/syscall.2.html

The Netwide Assembler (NASM)

```
section .data
  hello: db "Hello World", 10

section .text
  global _start       ; entry point for program
```

Next, we specify the message we intend to write to the terminal by using our label from **.data**




```
  mov r                ; stdout
  mov rsi, hello       ; message to write
```

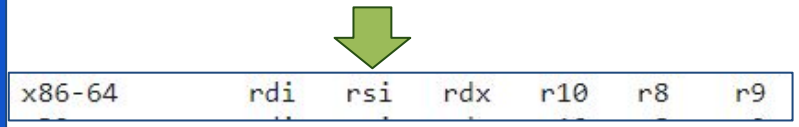
The Netwide Assembler (NASM)

```
section .data
  hello: db "Hello World", 10

section .text
  global  start        ; entry point for program
  mov r                ; stdout
  mov rsi, hello       ; message to write
```

However, Assembly isn't smart enough to know **how much** to print

All we said was **where to start printing from** (remember, variables are simply memory addresses)

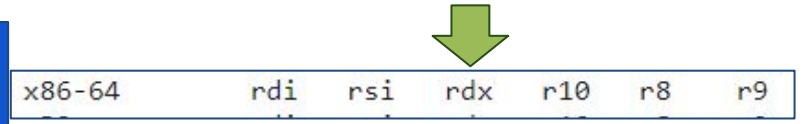


The Netwide Assembler (NASM)

```
section .data
  hello: db "Hello World", 10

section .text
  global _start       ; entry point for program
```

So, we need to specify to the program **how many bytes to read** from the memory address of **`hello`** by giving **`sys_write`** the memory address as a parameter



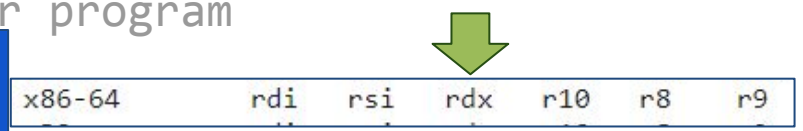

```
    mov r[illegible]ello    ; message to write
    mov rdx, 12             ; message length
```

The Netwide Assembler (NASM)

```
section .data
  hello: db "Hello World", 10
  helloLen: equ $-hello

section .text
  global _start          ; entry point for program
    mov r                ; stdout
    mov rsi, hello       ; message to write
    mov rdx, helloLen    ; message length
```

Some additional witchcraft could be done to calculate the length of a String by referencing the **current memory location** of `hello` (**$**) and calculating its **offset** (length) in memory (**-**)

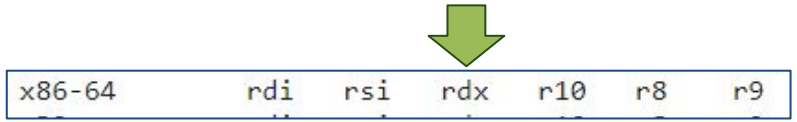


The Netwide Assembler (NASM)

```
section .data
  hello: db "Hello World", 10

section .text
  global _start       ; entry point for program

  _start:             ; starting point
    mov rax, 1        ; sys_write
    mov rdi, 1        ; stdout
    mov rsi, hello    ; message to write
    mov rdx, 12       ; message length
```



We don't need the rest of these registers for our application, but the Linux manual for **`syscall`** also notes where additional parameters can be found

The Netwide Assembler (NASM)

```
section .data
  hello: db "Hello World", 10

section .text
  global _start       ; entry point for program

  _start:             ; starting point
    mov rax, 1        ; sys_write
    mov rdi, 1        ; stdout
    mov rsi, hello    ; message to write
    mov rdx, 12       ; message length
    syscall           ; execute rax
```

Now that we've loaded everything needed into memory, we can finally tell the CPU to call `sys_write`

The Netwide Assembler (NASM)

```
_start:                 ; starting point
   mov rax, 1           ; sys_write
   mov rdi, 1           ; stdout
   mov rsi, hello       ; message to write
   mov rdx, 12          ; message length
   syscall              ; execute rax

   mov rax, 60          ; sys_exit
   mov rdi, 0           ; error code 0 (success)
   syscall              ; execute rax
```

This last bit of instruction is to "correctly" end our program, because the CPU expects a **sys_exit** system call

“Compiling” Assembly

Similar to other languages, Assembly needs to be translated into machine code

```
> nasm -f elf64 hello.asm
```

We can use NASM to generate our 64-bit binary (In `elf` format specifically, we'll talk more about it next lecture)

We need to do one final task: **link** our binary to an executable file

```
> ld -o hello hello.o
> ./hello
Hello World
```

~~The Netwide~~ GNU Assembler (GAS)

```
.data
  hello: .string "Hello World\n"

.text
.global _start          # entry point for program

  _start:               # starting point
    mov $1,      %rax   # sys_write
    mov $1,      %rdi   # stdout
    mov $hello,  %rsi   # message to write
    mov $12,     %rdx   # message length
    syscall             # execute rax

    mov $60,     %rax   # sys_exit
    mov $0,      %rdi   # error code 0 (success)
    syscall             # execute rax
```

Now in AT&T (Linux format)!

“Compiling” Assembly

If we're using AT&T syntax then NASM won't work!

However, we can utilize **`gcc`** to do the exact same thing

```
> gcc -c -no-pie hello.s -o hello.o
> ld -o hello hello.o
> ./hello
Hello World
```

`-c` = generate an object, but don't link

`-no-pie` = disable Position Independent Executable (PIE), which is a security feature that randomizes the base address of the program

In-class practice / Tooling check

- Write and assemble (compile) an x86 “Hello, World” binary
  - Ensure you have the right Linux setup
- What happens if you pass a 0 for $RSI?
- What happens if you pass a value greater than 12?
- How could you bypass the need to have a data section?

NR	syscall name	references	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)	arg3 (%r10)	arg4 (%r8)	arg5 (%r9)
1	write	man/ cs/	0x01	unsigned int fd	const char *buf	size_t count	-	-	-
60	exit	man/ cs/	0x3c	int error_code	-	-	-	-	-
62	kill	man/ cs/	0x3e	pid_t pid	int sig	-	-	-	-

In-class practice / Tooling check

- Write and assemble (compile) an x86 “Hello, World” binary
  - Ensure you have the right Linux setup
- What happens if you pass a 0 for $RSI?
  - Nothing. You could figure this out by looking at the source code but this is way easier!
- What happens if you pass a value greater than 12?
- How could you bypass the need to have a data section?

In-class practice / Tooling check

- Write and assemble (compile) an x86 “Hello, World” binary
  - Ensure you have the right Linux setup
- What happens if you pass a 0 for $RSI?
  - Nothing. You could figure this out by looking at the source code but this is way easier!
- What happens if you pass a value greater than 12?
  - Who knows? Leak secrets, maybe?
- How could you bypass the need to have a data section?

In-class practice / Tooling check

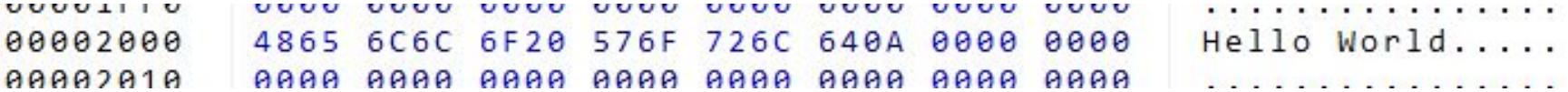
- Write and assemble (compile) an x86 “Hello, World” binary
  - Ensure you have the right Linux setup
- What happens if you pass a 0 for $RSI?
  - Nothing. You could figure this out by looking at the source code but this is way easier!
- What happens if you pass a value greater than 12?
  - Who knows? Leak secrets, maybe?
- How could you bypass the need to have a data section?
  - Put things on the stack!

In-class practice / Tooling check

- Write and assemble (compile) an x86 “Hello, World” binary
  - Ensure you have the right Linux setup
- What happens if you pass a 0 for $RSI?
  - Nothing. You could figure this out by looking at the source code but this is way easier!
- What happens if you pass a value greater than 12?
  - Who knows? Leak secrets, maybe?
- How could you bypass the need to have a data section?
  - Put things on the stack!
  - Put things in the code!

Changing the World

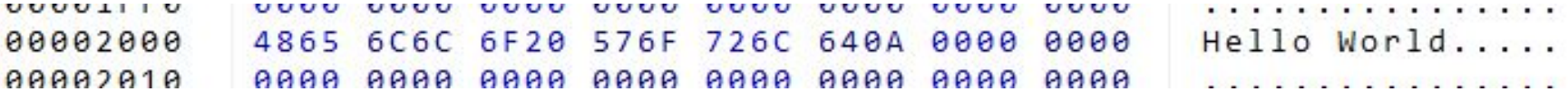
Whether you used Intel or AT&T syntax, the end result is the same: a binary object that stores "Hello World" in binary




```
hello: 48 65 6C 6C 6F 20 57 6F 72 6C 64 0A
```

Changing the World

Whether you used Intel or AT&T syntax, the end result is the same: a binary object that stores "Hello World" in binary

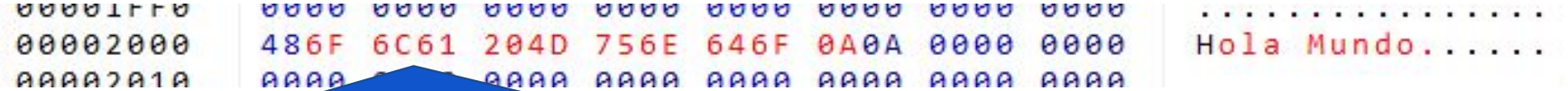



```
hello: 48 65 6C 6C 6F 20 57 6F 72 6C 64 0A
```

But that also means if we change this section in memory, we can **~change the world~**

Changing the World

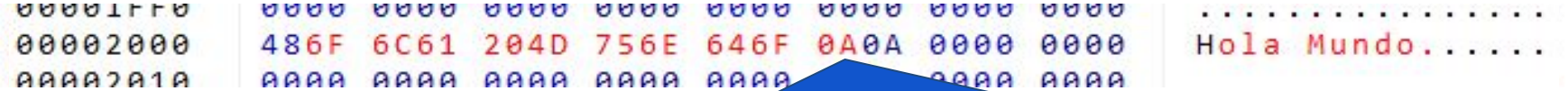
Whether you used Intel or AT&T syntax, the end result is the same: a binary object that stores "Hello World" in binary



By directly manipulating the binary, we can replace "Hello World" with any text that fits into 12 bytes

Changing the World

Whether you used Intel or AT&T syntax, the end result is the same: a binary object that stores "Hello World" in binary



Note "Hola Mundo" is only 10 characters, so I also injected another new line character into the code

In-class practice

- You can use HexEd.it to patch binaries in your browser!
  - Go to https://go.ncsu.edu/csc405-s26-02
  - Try out changing “Hello World” with `./hello`
  - Get `./check` to print “Access Granted”

In-class practice

- You can use HexEd.it to patch binaries in your browser!
  - Go to https://go.ncsu.edu/csc405-s26-02
  - Try out changing “Hello World” with `./hello`
  - Get `./check` to print “Access Granted”
    - Change Access Denied to Access Granted

In-class practice

- You can use HexEd.it to patch binaries in your browser!
  - Go to https://go.ncsu.edu/csc405-s26-02
  - Try out changing “Hello World” with `./hello`
  - Get `./check` to print “Access Granted”
    - Change Access Denied to Access Granted
    - Fix the bug of not having a null-terminated string

In-class practice

- You can use HexEd.it to patch binaries in your browser!
  - Go to https://go.ncsu.edu/csc405-s26-02
  - Try out changing “Hello World” with `./hello`
  - Get `./check` to print “Access Granted”
    - Change Access Denied to Access Granted
    - Fix the bug of not having a null-terminated string
    - Bypass the check! `75 27 (jne)` → `90 90 (nop nop)`

In-class practice

- You can use HexEd.it to patch binaries in your browser!
  - Go to https://go.ncsu.edu/csc405-s26-02
  - Try out changing “Hello World” with `./hello`
  - Get `./check` to print “Access Granted”
    - Change Access Denied to Access Granted
    - Fix the bug of not having a null-terminated string
    - Bypass the check! `75 27 (jne)` → `90 90 (nop nop)`

```
alex@mithrun ~/A/c/02-assembly (main)> chmod +x check_nop
alex@mithrun ~/A/c/02-assembly (main)> ./check_nop
Enter password: isajdfoaisjdofijasdoifj
Access granted!
alex@mithrun ~/A/c/02-assembly (main)>
```

Ensuring Program Integrity

This leads to a new question:

How do I ensure a program has not been tampered with?

Ensuring Program Integrity

This leads to a new question:

How do I ensure a program has not been tampered with?

Answer:

We can calculate a **checksum** of the program's original binary

Ensuring Program Integrity

This leads to a new question:

How do I ensure a program has not been tampered with?

Answer:

We can calculate a **checksum** of the program's original binary

```
> sha256sum hello
6688884c7518fb722e560c2b29866c5bbf97228e10d98966cd17fa4470da224c  hello
– or –
> md5sum hello
5c0499e5aec8b99a22e4723cbdc5c199  hello
```

We can then save this checksum to always ensure the program has not been tampered with

Ensuring Program Integrity

This leads to a new question:

How do I ensure a program has not been tampered with?

Answer:

We can calculate a **checksum** of the program's original binary

```
> sha256sum hello
6688884c7518fb722e560c2b29866c5bbf97228e10d98966cd17fa4470da224c  hello
— We edit the hello binary —
> sha256sum hello
6c2ff4ed235045a645f188630ac59ca3e826a94e0468b2f1896d6fe85ac350a6  hello
```

Security Zen - World's First MIDI Shellcode

https://psi3.ru/blog/swl01u/