CS 33:
Introduction to Computer Organization

Week 6

Admin

- Homework 4 “due” today, Friday, November 6th
- Lab 3: Smashing lab “due” Friday, November 13th
- ...also, midterm 2: Monday, November 16th?

Agenda

- Stack Exploits
- Lab 3: Smashing Lab
- Instruction Level Parallelism (ILP)? Maybe?

Stack Exploits

- Consider a server that hosts publicly accessible server code that performs tasks A, B, and C and an attacker that has access to the server and wants to cause trouble.
- It wouldn't be terribly impressive (or convincing) if the attacker claimed that he/she took over the server and coerced it to perform tasks A, B, or C.
- However, if the attacker could get the code to perform task D (something the was not intended to be performed by the server)...

Stack Exploits

- To do this, an attacker would want to get the server code to execute code of the attacker's choice.
- The attacker cannot directly influence the operations of the CPU or swap out the code contents in memory. It'd be pretty much game over if they could.
- However, very commonly, a user of a program will have the power to provide input to the server, which can affect on the stack.

Stack Exploits

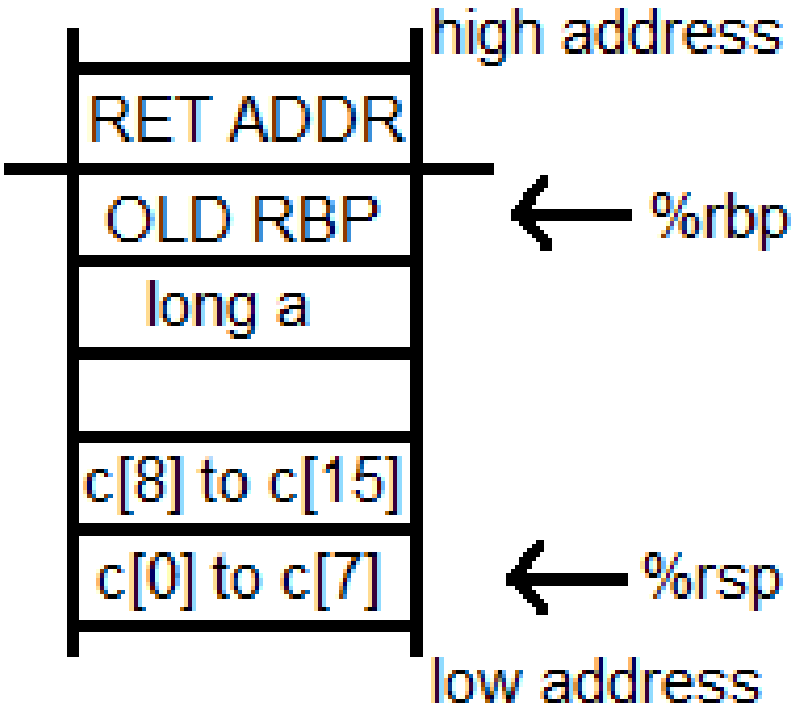
- Consider the following function:

```
int terrible()
{
  long a = 0x7766554433221100;
  char c[16];
  gets(c);
}
```

- When compiled, the stack looked like this:

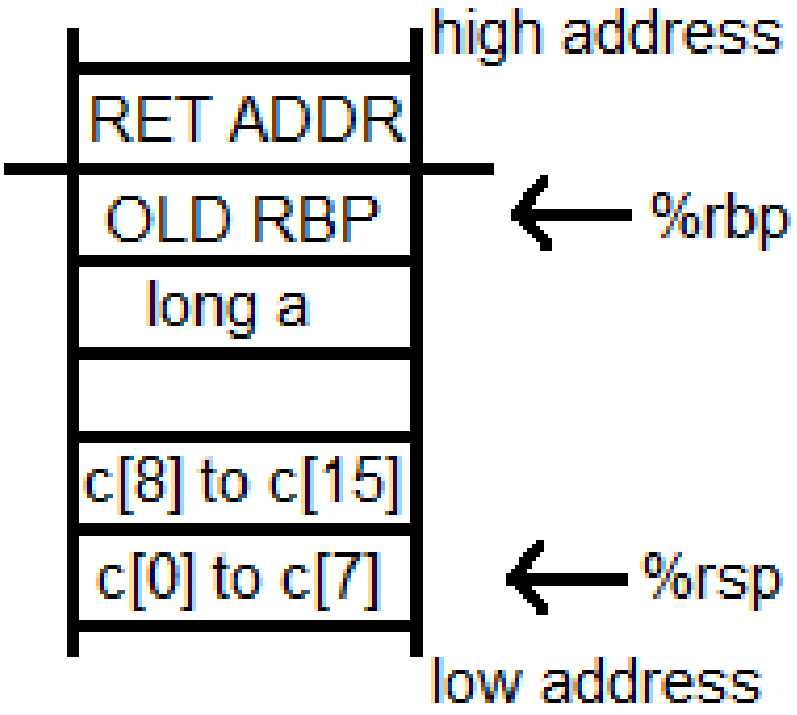
Stack Exploits

- Note: each block is 8 bytes.
- First of all, what's the blank block for?



Stack Exploits

- Note: each block is 8 bytes.
- First of all, what's the blank block for?
  - Alignment purposes. Recall that %rsp must be 16-byte aligned. The assumption is that %rsp was aligned when calling this function.
- The code looks like:



Stack Exploits

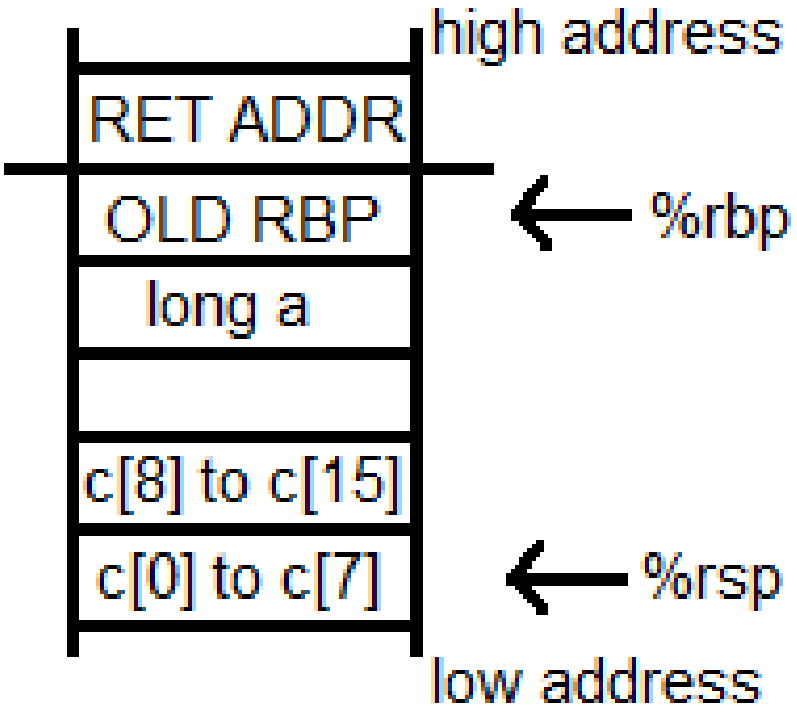
```
Dump of assembler code for function blah (gcc with
no options):
 0x400528 <+0>:     push   %rbp
 0x400529 <+1>:     mov    %rsp,%rbp
 0x40052c <+4>:     sub    $0x20,%rsp
 0x400530 <+8>:     movabs $0x7766554433221100,%rax
 0x40053a <+18>:    mov    %rax,-0x8(%rbp)
 0x40053e <+22>:    lea    -0x20(%rbp),%rax
 0x400542 <+26>:    mov    %rax,%rdi
 0x400545 <+29>:    callq  0x4003b0 <gets@plt>
 0x40054a <+34>:    leaveq
 0x40054b <+35>:    retq
End of assembler dump.
```

Stack Exploits

- The “gets” function takes as an argument, a character pointer. Then, it asks the user to input a character string where it then copies the string into the specified character pointer.
- For example, if you called “get(c)” and the user input provided “ABCD”, then the bytes for “ABCD” would be copied from *c to *c + 3.
- The pointer in this example is -0x20(%rbp) or -32(%rbp).

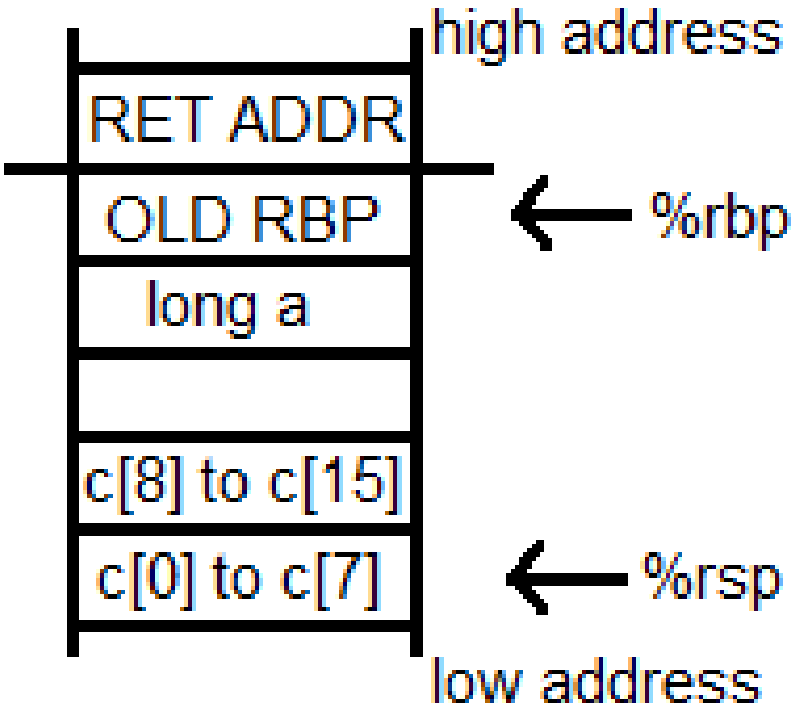
Stack Exploits

- If the user typed “jonathan” (8 characters), these 8 bytes would span from *c to *c + 7
- If the user typed “mr_super_long_name_the_third_esquire”, this would be copied from *c to *c + 35
- However, in the C code, we only specified a character array of size 16.
- “gets” don't care



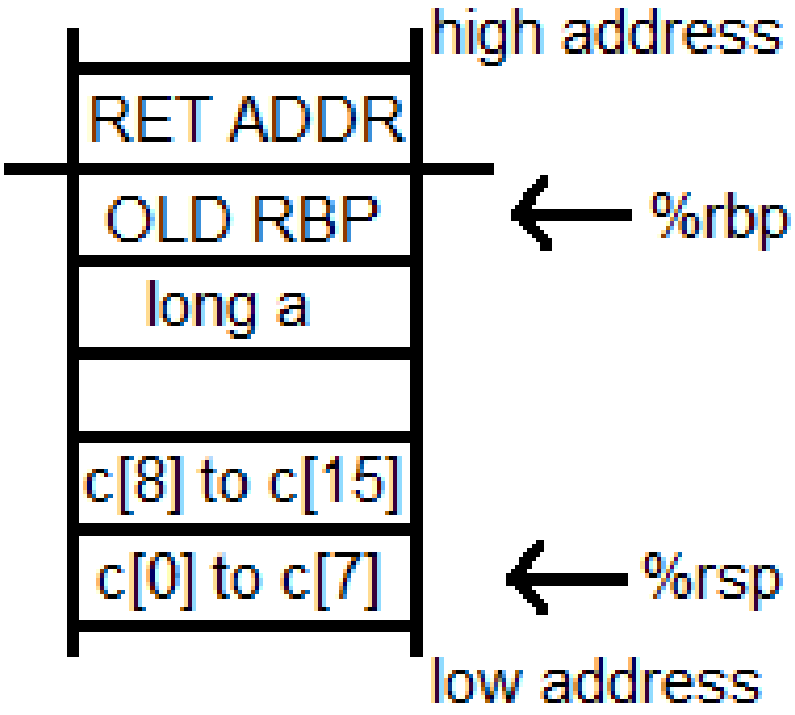
Stack Exploits

- As a result, the user can provide a string to arbitrarily write to *c to *c + x.
- This means, the user could also overwrite long a, the old rbp, and most worryingly, the return address.
- In this instance, what would the user have to write in order to overwrite the return address with 0x400800?



Stack Exploits

- In this instance, what would the user have to write in order to overwrite the return address with 0x400800?
- At least 40 characters (which will wipe out long a and OLD RBP), and then: 0x00, 0x08, 0x40, 0x00 … (assume little endian)
- Suddenly, once the function returns, it will attempt to execute something that it did not intend to.



Stack Exploits

- Now that there is a way to change the return address, how do you get the code to do something that it didn't intend to do.
- Say... delete a file?

Stack Exploits: Jump to Existing Library

- If your code contains: “#include <unistd.h>”, this means that that during runtime, the functions included in that library exist somewhere in the code segment.
- Even if the original code didn't use them, somewhere, there exists many fun functions (or ***fun***ctions) such as:
  - unsigned int alarm(unsigned int)
  - int pause()
  - int chown(const char *path, uid_t owner, gid_t group)
  - int unlink(const char *path) – Deletes a file specified by path.

Stack Exploits: Jump to Existing Library

- Why write the code to implement behavior when the code already exists in the executable?
- Thus, if you know where the “unlink” function is loaded into the executable during runtime, you can change the return address to &unlink and then “return” to unlink.
- How would you know the address of unlink?
- We make the assumption that you have a copy of the code and can therefore run it and see where it will be loaded. Thus, you can use gdb to find and disassemble unlink.

Stack Exploits: Jump to Existing Library

- Can you see a problem with this approach?
- An obvious one is that you're limited to the functions that are provided by the libraries. In particular, you can only call functions provided by the linked libraries. That can be limiting.
- However, there's another, more pressing weakness.
- Hint: The signature of unlink is:
  - int unlink(const char *path);

Stack Exploits: Jump to Existing Library

- Can you see a problem with this approach?
- In order to get unlink to work, we need to specify a pointer to a string (which is the path to the file we want to delete) as an argument.
- In x86-64, we specify arguments via registers. If all we do is overwrite the return address, then we will effectively call unlink on whatever is currently in the %rdi register, which is probably not the path that we want.

Stack Exploits: Jump to Existing Library

- The fact is, this USED to work on x86. Why?

Stack Exploits: Jump to Existing Library

- The fact is, this USED to work on x86. Why?
- x86 specifies its arguments by passing them on to the stack. More specifically, when a function is entered the arguments are expected to be at (%esp + X).
- Thus, we could previous write the argument to the stack.
- Now thanks to the cutting edge 2003 x86-64 technology, this is no longer a possibility.
- Or is it...?

Stack Exploits: Injecting Code

- Knowing that we are no longer able to rely on existing code, we will somehow need to supply our own.
- Then, we can jump to our own code when we return.
- How exactly do we specify code? It's not exactly like the program is going to be asking us for code to inject. Don't be ridiculous.

Stack Exploits: Injecting Code

- Actually... it sort of it
- “gets” gives us direct access to writing into memory; whatever string of bytes is provided as an input is saved into memory/the stack.
- Sure it's not the code segment, but we can provide a string that contains the code that we want to execute,
- Then in the return address, write the address of the code that we provided.

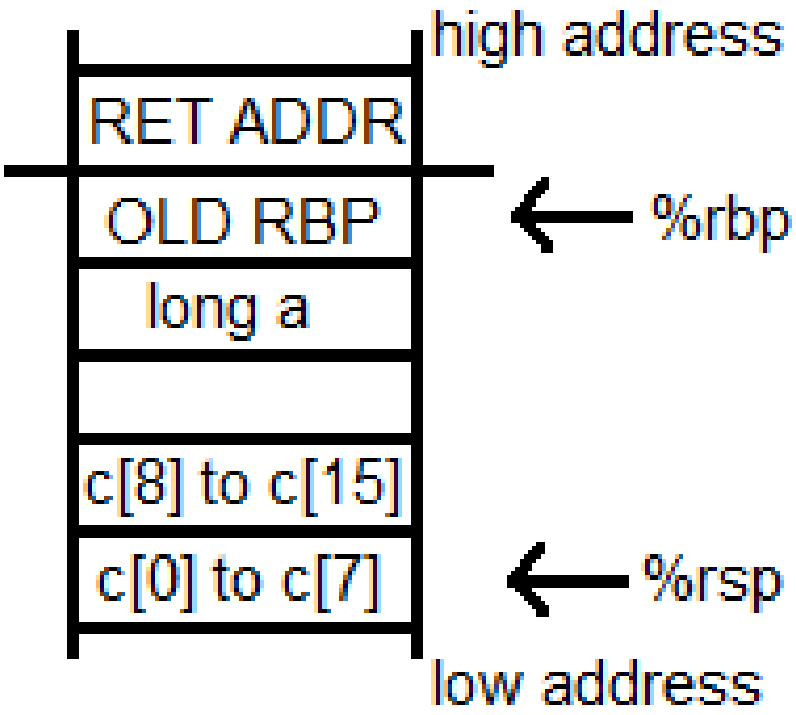
Stack Exploits: Injecting Code

- Let's say as our vicious attack to take over the server is to execute the following instructions:

```
48 89 f8      mov    %rdi,%rax
48 83 c0 01   add    $0x1,%rax
```

- Go big or go home.
- Consider the previous “gets” example:

Stack Exploits: Injecting Code



- The function gets(c) is called.
- Say %rsp = 0x7fffffffe1d0.
- Then the memory before calling “gets” looks like this:

Stack Exploits: Injecting Code

- **(gdb) x/40xw $rsp**
- **Stack frame for “terrible” before calling gets.**
- **%rsp is 0x7fffffffe1d0 (and &c).**

```
0x7fffffffe1d0: 0x00000000  0x00000000  0x00000000  0x00000000
0x7fffffffe1e0: 0x00400570  0x00000000  0x33221100  0x77665544
0x7fffffffe1f0: 0xffffe200  0x00007fff  0x00400560  0x00000000
```

- **First, some notes about this little endian GDB format.**

Stack Exploits: Injecting Code

- (gdb) x/40xw $rsp
- Stack frame for “terrible” before calling gets.
- %rsp is 0x7fffffffe1d0 (and &c).

- 0x7fffffffe1d0: 0x00000000 0x00000000 0x00000000 0x00000000
  0x7fffffffe1e0: 0x00400570 0x00000000 0x33221100 0x77665544
  0x7fffffffe1f0: 0xffffe200 0x00007fff 0x00400560 0x00000000

- Each column is a 4-byte word.
- 0x00400570 is the 4-byte word at address 0x7fffffffe1e0.
- 0x00000000 is the word at address 0x7fffffffe1e4
- 0x33221100 is the word at address 0x7fffffffe1e8
- 0x77665544 is the word at address 0x7fffffffe1ec

Stack Exploits: Injecting Code

- (gdb) x/40xw $rsp
- Stack frame for “terrible” before calling gets.
- %rsp is 0x7fffffffe1d0 (and &c).

- 0x7fffffffe1d0: 0x00000000 0x00000000 0x00000000 0x00000000
  0x7fffffffe1e0: 0x00400570 0x00000000 0x33221100 0x77665544
  0x7fffffffe1f0: 0xffffe200 0x00007fff 0x00400560 0x00000000

- Each word is presented as the actual value. In a little endian machine.
- 0x70 is the byte at address 0x7fffffffe1e0.
- 0x05 is the byte at address 0x7fffffffe1e1
- 0x40 is the byte at address 0x7fffffffe1e2
- 0x00 is the byte at address 0x7fffffffe1e3

Stack Exploits: Injecting Code

- **(gdb) x/40xw $rsp**
- **Stack frame for “terrible” before calling gets.**
- **%rsp is 0x7fffffffe1d0 (and &c).**

- **0x7fffffffe1d0: 0x00000000 0x00000000 0x00000000 0x00000000**
  0x7fffffffe1e0: 0x00400570 0x00000000 0x33221100 0x77665544
  0x7fffffffe1f0: 0xffffe200 0x00007fff 0x00400560 0x00000000

- **Remember, we want to execute:**
  48 89 f8 mov %rdi,%rax
  48 83 c0 01 add $0x1,%rax

- **Because of gets(c), we have the power to write bytes into the 'c' buffer.**

Stack Exploits: Injecting Code

```
Dump of assembler code for function blah (gcc with
no options):
 0x400528 <+0>:     push   %rbp
 0x400529 <+1>:     mov    %rsp,%rbp
 0x40052c <+4>:     sub    $0x20,%rsp
 0x400530 <+8>:     movabs $0x7766554433221100,%rax
 0x40053a <+18>:    mov    %rax,-0x8(%rbp)
 0x40053e <+22>:    lea    -0x20(%rbp),%rax
 0x400542 <+26>:    mov    %rax,%rdi
 0x400545 <+29>:    callq  0x4003b0 <gets@plt>
 0x40054a <+34>:    leaveq
 0x40054b <+35>:    retq
End of assembler dump.
```

Stack Exploits: Injecting Code

- We need to do two things:
  - Inject the code.
  - Overwrite the return address to point to the injected code.
- One way of doing this is to inject the code with “gets” at the beginning of “c” and then have the return address point to address of buffer “c”.
- In this example, the return address is at *c + 40 (&c is a -0x20(%rbp). Since %rbp was the first thing pushed to the stack, the return address is at 0x8(%rbp). Thus, the return address is *c + 0x20 + 0x8)

Stack Exploits: Injecting Code

- (gdb) x/40xw $rsp
- Stack frame for “terrible” before calling gets.
- %rsp is 0x7fffffffe1d0 (and &c).

- 0x7fffffffe1d0: 0x00000000 0x00000000 0x00000000 0x00000000
  0x7fffffffe1e0: 0x00400570 0x00000000 0x33221100 0x77665544
  0x7fffffffe1f0: 0xffffe200 0x00007fff 0x00400560 0x00000000

- Red: The return address.
- Blue: The first 4-bytes of c (&c is ...e1d0). The return address is at ...e1d0 + 0x20 + 0x8 = e1f8.

Stack Exploits: Injecting Code

- Thus, if we provided the exploit string:

```
48 89 f8 48 83 c0 01 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 d0 e1 ff ff ff 7f 00 00
```

- 48 89 f8 48 83 c0 01 is the code we want to execute.
- d0 e1 ff ff ff 7f 00 00 is the return address.
- After the “gets”, the stack will look like this:

Stack Exploits: Injecting Code

- (gdb) x/40xw $rsp
- Stack frame for “terrible” AFTER calling gets with our exploit string.
- %rsp is 0x7fffffffe1d0.

```
0x7fffffffe1d0: 0x48f88948  0x0001c083  0x00000000  0x00000000
0x7fffffffe1e0: 0x00000000  0x00000000  0x00000000  0x00000000
0x7fffffffe1f0: 0x00000000  0x00000000  0xffffe1d0  0x00007fff
```

- Now when we return, the code will “return” to 0x7fffffffe1d0 and execute accordingly:

```
Dump of assembler code for function... huh? Was this here before?:
   0x7fffffffe1d0 <+0>:   48 89 f8        mov    %rdi,%rax
   0x7fffffffe1d3 <+3>:   48 83 c0 01     add    $0x1,%rax
End of assembler dump. Man I'm losing it.
```

Stack Exploits: Protection From

- How do we handle/prevent such an exploit?
- There are two aspects that allow an exploit like this:
  1. The ability to write beyond an allocated buffer (ie, the buffer overrun) to overwrite return addresses.
  2. The ability to execute code of the attacker's specification.
- We can try to protect our system by preventing both aspects.

Stack Exploits: Protection From

- How do you protect against:
  - The ability to write beyond an allocated buffer (ie, the buffer overrun)

Stack Exploits: Protection From

- Method 1: Don't use C.
- The issue of overrunning a buffer is archaic and well-documented. Modern languages automate and abstract away many of C's low level behavior such as pointer manipulation and maintaining proper array access.
- Modern languages tend to maintain the length of buffers and will crash if you attempt to access an area out of range.
- What's a weakness of this approach?

Stack Exploits: Protection From

- What's an issue of the “don't use C” approach?
- Depending on your preference of programming languages, this could be a blessing.
- However, there's a reason that C is still the predominant language for low level OS code. You have more direct control over memory, I/O, and etc. and more control over things that other languages don't trust the user with.
- C is fast.
- Please don't stop using C.

Stack Exploits: Protection From

- Method 2: Write better code. Use safer functions.
- If you try to compile code using “gets”, it's either unrecognized or you get this:
  - “warning: the `gets' function is dangerous and should not be used.”
  - The compiler just called a function “dangerous”. Time to sleep with one eye open.
- Use functions that allow you to specify a maximum length of bytes such as fgets() and read().
- If you write a program with no potential for buffer overrun, problem solved.

Stack Exploits: Protection From

- Method 3: “Canaries” or their cooler name, “Sentinels”
- For each buffer, allocate extra space before and after the buffer.
- Consider an char c[4] where &c = 0x100:

ADDR:	0xFF	0x100	0x101	0x102	0x103	0x104
VALUE:	0xCA	0x00	0x11	0x22	0x33	0xCA

Stack Exploits: Protection From

- Consider an char c[4] where &c = 0x100:

ADDR:	0xFF	0x100	0x101	0x102	0x103	0x104
VALUE:	0xCA	0x00	0x11	0x22	0x33	0xCA

- The bytes before and after are special canary bytes whose values change between invocations of the program. The compiler generates code that maintains the canaries.
- Then, before the function returns, check to make sure the canaries are still preserved. If they have been changed you've had a buffer overrun or have written to a bad spot.
- What are some weaknesses of this approach?

Stack Exploits: Protection From

- The program takes up more memory:
  - More space allocated in the stack
  - More instructions necessary for execution
- The program is slower:
  - Having to constantly check that the canaries are alive.
- It's not foolproof. If an attacker knows you're using canaries, they can try to guess the canaries when overrunning the buffer.

Stack Exploits: Protection From

- How do you protect against
  - The ability to execute code of the attacker's specification.
- There's not a whole lot the programmer can do about this one, but what are some techniques that the compiler and machine can do?

Stack Exploits: Protection From

- Address Space Layout Randomization.
  - Each time a program is executed, the addresses/locations of things such as the stack and the code are slightly different each time.
- Both of the exploits mentioned above require knowledge of where key things are located in memory (for example, &c)
- However, what if &c or &unlink were slightly different each time we ran the code? We would no longer be able to reliably “return” to them.

Stack Exploits: Protection From

- NX bit AKA execution permissions.
- The real danger in the buffer overflow is when the user can execute arbitrary code.
- A user is usually only able to write arbitrary code to the stack or the heap.
- Thus, one solution is to maintain permissions on the different areas of memory and only allow for the code to execute instructions located in the text (code) segment of memory.

Stack Exploits: Protection From

- NX bit AKA execution permissions.
- For example:
  - Stack + Heap: Read/Write but no Execute
  - Text (code): Read/Execute but no Write
- Now, the only place an attacker could write his own exploit code is the Stack + Heap, but nothing written there can be executed.
- This doesn't prevent the “return to unlink” style attack, but we've prevent the user from executing arbitrary code, right? Right?

Stack Exploits: Return Oriented Programming

- The bytes that comprise the program's code can be interpreted differently from their original intent.
- For example, recall that ret has byte code 'C3'
- Say we have the code snippet:

```
   0x...fe8 <+120>:    8d 04 39          lea    (%rcx,%rdi,1),%eax
   0x...feb <+123>:    48 39 c3          cmp    %rax,%rbx
   0x...fee <+126>:    0f 82 c4 00 00 00       jb
0x7ffff7dee0b8 <add_to_global+328>
```

Stack Exploits: Return Oriented Programming

- Say we have the code snippet:

```
   0x...fe8 <+120>:    8d 04 39          lea    (%rcx,%rdi,1),%eax
   0x...feb <+123>:    48 39 c3          cmp    %rax,%rbx
   0x...fee <+126>:    0f 82 c4 00 00 00       jb
0x7ffff7dee0b8 <add_to_global+328>
```

- Hey look, it's the return instruction at 0x...fed!
- If we interpreted 0x...fed as the starting point of an instruction, we would instead be calling the return function where there wasn't originally a ret.

Stack Exploits: Return Oriented Programming

- This idea is the crux of Return Oriented Programming.
- If we can't execute the stack, we're limited to the functions provided by the source code + linked libraries that are in the executable segments of memory.

Stack Exploits: Return Oriented Programming

- This idea is the crux of Return Oriented Programming.
- If we can't execute the stack, we're limited to the ~~functions~~ bytes provided by the source code + linked libraries that are in the executable segments of memory.

Stack Exploits: Return Oriented Programming

- Find bytes in the code that form meaningful instructions that end in the ret instruction.
- These snippets will be our new “functions”.
- Then, write the addresses of the sequence of functions you want to execute to the stack.
- If you can find enough meaningful instructions to do what you want, you've just created arbitrary code out of a limited set of code.

Stack Exploits: Return Oriented Programming

- For example, pretend that:
  - The ret instruction is represented by byte “RR”
  - You want to execute the following code:
  - CC AA BB movq %rdi, %rax
  - CC DD EE FF GG HH addq $1, %rax
  - No, this is not the mythical “octodecimal” representation. These are just fake placeholder bytes.
  - You find a snippet of the provided library code that does this:

Stack Exploits: Return Oriented Programming

```
...
0x4006af <+16>:    JJ EE            mov %eax,%ecx
0x4006b1 <+18>:    BB VV            sar %cl,%edx
0x4006b3 <+20>:    JJ QQ            mov %edx,%eax
0x4006b5 <+22>:    JJ RR TT         mov %eax,-0x4(%rbp)
0x4006b8 <+25>:    LL AA VV CC AA   mov $0x1f,%eax
0x4006bd <+30>:    BB RR GG         sub -0x18(%rbp),%eax
0x4006c0 <+33>:    JJ AA BB         mov %eax,-0x8(%rbp)
0x4006c3 <+36>:    LL CC DD         mov -0x8(%rbp),%eax
0x4006c6 <+39>:    EE FF GG HH RR   mov $0xffffffff,%edx
0x4006cb <+44>:    JJ CC            mov %eax,%ecx
…
```

- Well, I don't see the instructions we're looking for...

Stack Exploits: Return Oriented Programming

```
...
0x4006af <+16>:    JJ EE              mov %eax,%ecx
0x4006b1 <+18>:    BB VV              sar %cl,%edx
0x4006b3 <+20>:    JJ QQ              mov %edx,%eax
0x4006b5 <+22>:    JJ RR TT           mov %eax,-0x4(%rbp)
0x4006b8 <+25>:    LL AA VV CC AA     mov $0x1f,%eax
0x4006bd <+30>:    BB RR GG           sub -0x18(%rbp),%eax
0x4006c0 <+33>:    JJ AA BB           mov %eax,-0x8(%rbp)
0x4006c3 <+36>:    LL CC DD           mov -0x8(%rbp),%eax
0x4006c6 <+39>:    EE FF GG HH RR     mov $0xffffffff,%edx
0x4006cb <+44>:    JJ CC              mov %eax,%ecx
…
```

- Well, I don't see the instructions we're looking for...
- But this looks promising

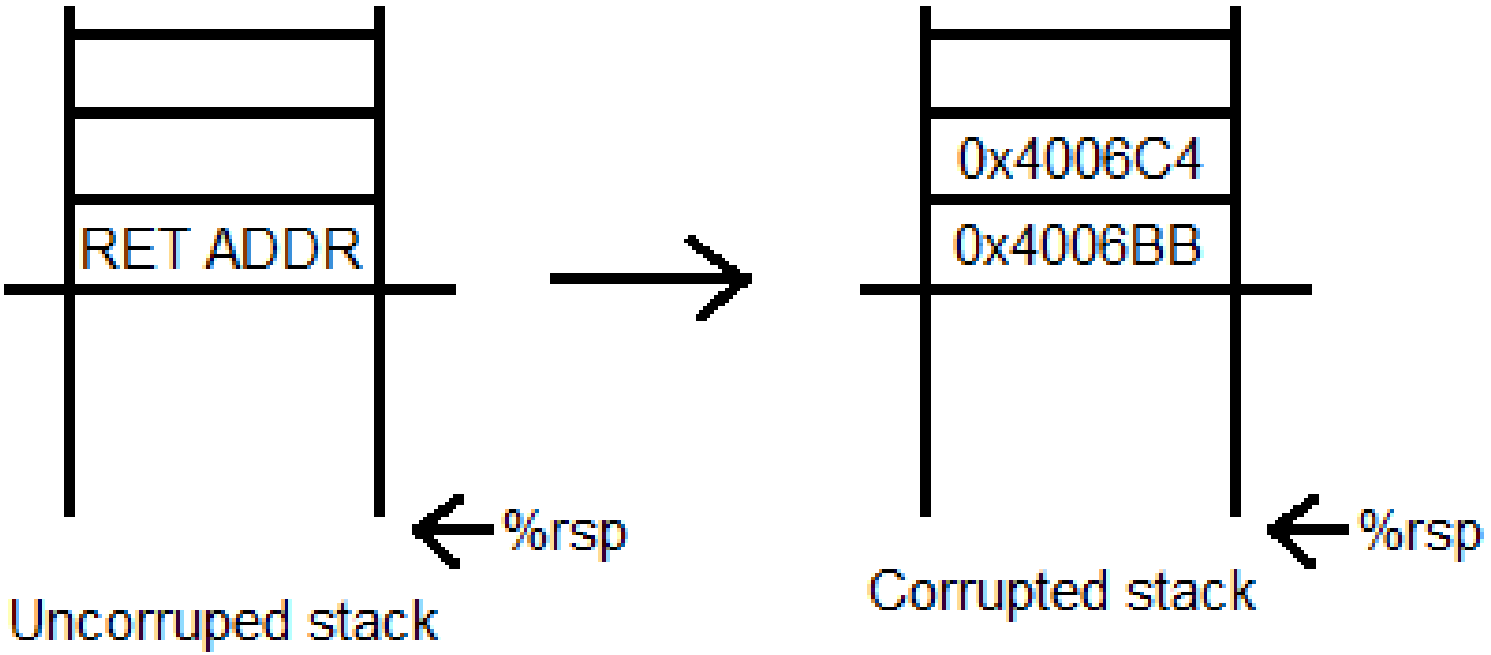
Stack Exploits: Return Oriented Programming

```
...
0x4006b8 <+25>:    LL AA VV CC AA   mov $0x1f,%eax
0x4006bd <+30>:    BB RR GG         sub -0x18(%rbp),%eax
0x4006c0 <+33>:    JJ AA BB         mov %eax,-0x8(%rbp)
0x4006c3 <+36>:    LL CC DD         mov -0x8(%rbp),%eax
0x4006c6 <+39>:    EE FF GG HH RR   mov $0xffffffff,%edx
…
```

- If we treated 0x4006bb as the starting point for an instruction, we'd have:
  - 0x4006bb: CC AA BB movq %rdi, %rax
  - 0x4006be: RR ret
- If we treated 0x4006c4 as the starting point for an instruction, we'd have:
  - 0x4006c4: CC DD EE FF GG HH addq $1, %rax
  - 0x4006ca: RR ret

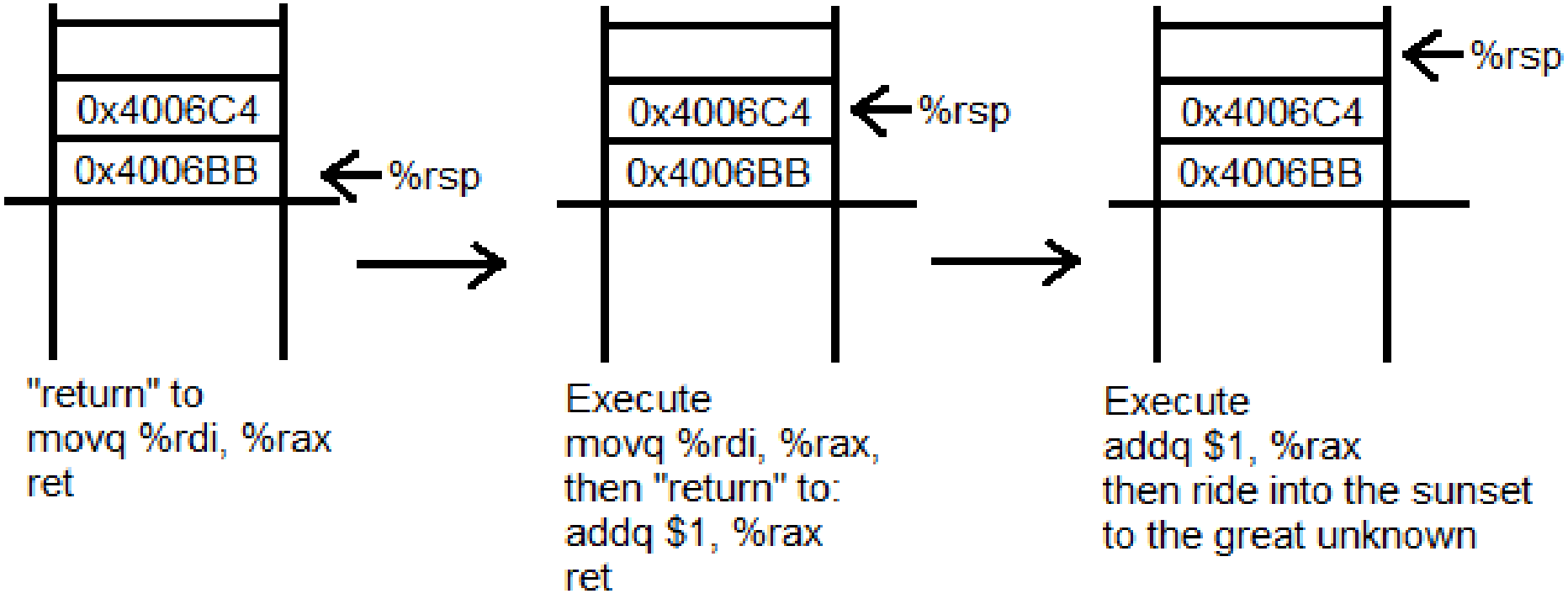
Stack Exploits: Return Oriented Programming

- Assuming we have a buffer overrun exploit to... exploit, we can prepare the stack like this:



- Now, when we return from the function, we jump to 0x4006BB

Stack Exploits: Return Oriented Programming



- Hence: “return oriented”.
- Your “code” now consists of the sequence of return addresses that you wrote to the stack.

Stack Exploits: Return Oriented Programming

- Note: There's a reason I had to completely fabricate an example.
- This is very difficult and is often done via automated analysis and requires a sufficiently large program.
- Let this be a hint as to whether or not you should turn towards Return Oriented Programming for Lab 3.
- Speaking of...

Lab 3: Smashing Lab

- Your task (an overview, for more details, consult the specs):
- 1. Download the thttpd server code.
- 2. Apply the “patch” to src/thttpd.c
  - Patches are generally used to modify source code in order to fix bugs.
  - The format of the posted patch is correctly formatted for you to use the Linux “patch” command if you're so inclined.
  - However...

Lab 3: Smashing Lab

- DISCLAIMER:
- As of writing these slides, the professor has not yet updated the Fall '15 version of Smashing Lab.
- As a result, these slides are based on the Spring '15, 32-bit version of Smashing Lab. The general techniques will likely be the same but the specifics such as compilation flags will not!
- I will modify the slides once the new lab is posted.

Lab 3: Smashing Lab

- Here's a snippet:

```
@@ -999,7 +999,7 @@ static void
 read_config( char* filename )
     {
     FILE* fp;
-    char line[10000];
+     char line[100];
```

- The format of patch files is such that the lines marked by “-” indicate a line that should be removed from src/thttpd.c. The line marked by “+” is a line that should be added.

Lab 3: Smashing Lab

- As you can see, we're simply replacing two lines of code in read_config, so feel free to do that by hand.
- 2.5. Configure the make script.

```
./configure \
   CFLAGS='-m32' \
   LDFLAGS="-Xlinker --rpath=/usr/local/cs/gcc-$(gcc -dumpversion)/lib"
```

Lab 3: Smashing Lab

- 3. Compile the server
  - You will compile it using the make script and under three different levels of stack-protection
    - make clean
    - make CFLAGS='-m32 -g3 -O2 -fno-inline -fstack-protector-strong'
    - mv src/thttpd src/thttpd-sp
- 4. Run the server.
  - “src/thttpd-no -p 10000 -D”
  - This step is only necessary to check to make sure that it is working correctly.

Lab 3: Smashing Lab

Here's a freebie. Make a file called compile.sh and include this code

```
#!/bin/sh

./configure \
   CFLAGS='-m32' \
   LDFLAGS="-Xlinker --rpath=/usr/local/cs/gcc-$(gcc
-dumpversion)/lib"

make clean
make CFLAGS='-m32 -g3 -O2 -fno-inline -fstack-protector-strong'
mv src/thttpd src/thttpd-sp

make clean
make CFLAGS='-m32 -g3 -O2 -fno-inline -fsanitize=address'
mv src/thttpd src/thttpd-as

make clean
make CFLAGS='-m32 -g3 -O2 -fno-inline'
mv src/thttpd src/thttpd-no
```

Lab 3: Smashing Lab

- This creates a bash script that will run all of the commands listed inside, in the order that they appear. Use:
  - chmod +777 compile.sh
- ...to change the permissions of the file so that it is executable.
- Then, you can use:
  - ./compile.sh
- To compile everything at once. Modify as you see fit.

Lab 3: Smashing Lab

- Note, the server is invoked with several flags:
- -p 10000
  - This specifies a port number (in this example 10000). The actual details are unimportant, but this is a networking concept. This is necessary to run.
- -D
  - This runs the server in the foreground. If you don't include '-D', the server will start in the background and you'll have to kill it manually by hunting down the process id.
- -C config.txt
  - This runs the server with the configuration file “config.txt”. This is how you will perform your exploits/crashes.

Lab 3: Smashing Lab

- 5. Crash the running servers by invoking them in a special way.
  - This is a little silly. If we have control over starting/stopping the server, we probably already have the power to cause trouble.
  - This “attack” is to run “src/thttpd-no …” in a way where it crashes.
  - Hey, that's easy.

Lab 3: Smashing Lab

- src/thttpd-no -p 10000 -D awerkhdvkhasdf
- That works right?
- We're looking for a particular type of crash.
- We're looking for a crash that was a result of corrupting the stack.
- Hopefully, this will crash the code in a way where you can collect a backtrace when it crashes.

Lab 3: Smashing Lab

- How do you collect a backtrace?
- In gdb, when you run the server under the conditions where it would crash, once it crashes, use the gdb command:
  - backtrace (or bt)
- If the program crashes via an abort, you will not be able to generate the backtrace. Therefore aim for the Segmentation Fault. However, aborts are also considered acceptable.

Lab 3: Smashing Lab

- 6. Generate assembly for handle_read in order to determine the differing techniques used by -fstack-protector-strong and -fsanitize=address to protect the stack.
  - Pretty straightforward.
  - Sorry if this part triggers any flashbacks to Pexex Lab.
- Hey look, it's our good friend “-fsanitize” again.
- Both of these techniques are described in the links on the top of the specs page.

Lab 3: Smashing Lab

- 7. Finally, build an exploit for the thttpd-no case that allows you to delete a file called “target.txt”.
  - Note, despite what the previous section was about, your exploit should not be focused on handle_read which handles the case where a client issues a URL request to the server.
  - This is going to be an exploit in read_config where the patch introduced a bug.
  - This means the exploit ought to delete the file when starting up the server correctly (or incorrectly as it were).

Lab 3: Smashing Lab

- General techniques:
  - Somehow, you are going to need to overrun a buffer in read_config.
  - Unlike in the crashing cases, you will need to execute your own code.
  - This means somehow overwriting the return address to point to code that you supply that corresponds with deleting the file target.txt.

Lab 3: Smashing Lab

- Examine the code in read_config to determine where the buffer exploit occurs, how to exploit it, and where the buffer is.
- Examine the assembly and run the code under gdb to determine the address of the buffer and the address of the return address (AKA the difference between the beginning of the buffer and the return address).
- Produce assembly bytecode that corresponds to an attack string that will work when injected into the buffer.

Lab 3: Smashing Lab

- ASLR/Stack randomization
  - This means that the stack addresses will change.
  - However... the relative distances between variables and pointers will remain the same, it's just that the base offset of the stack will be different. Use this to your advantage.
  - For example, say a buffer buf is located at address 0x100 while the return address is located at 0x110. If the stack is randomized so that buf is at 0x135, then the return address will be located at 0x145.

Lab 3: Smashing Lab

- A comment on stack randomization:
- By default, gdb disables stack randomization so an exploit that works in gdb may not work out in the wilderness.
- You can turn on stack randomization in gdb with the following:
  - set disable-randomization off
  - You're turning off the disabling of randomization.

Lab 3: Smashing Lab

- NX bit
  - Remember from class, typically you can read/write to the stack and read/execute the text segment. That means anywhere we can write, we can't execute.
- As far as we know, there's no way around this one. In order to turn the NX bit off, include:
- “-z execstack” as one of the flags when compiling.
- Or use the “execstack -s thttpd-no” command to change the executable thttpd-no so that the stack is executable.

Lab 3: Smashing Lab

- In addition to the commands you used for PexexLab, you may also find the following instructions useful:
  - p <var_name> (print the variable)
  - p &<var_name> (print the address of the variable)
  - p will sometimes return <optimized_out> in which case there is no address/value that corresponds to the variable at the moment.

Lab 3: Smashing Lab

- As some part of lab you will find it necessary to have the vulnerable server read in your exploit code in the form of bytecode, one character at a time.
- The first step is to generate the bytes that correspond to the assembly instructions you want to run.

Lab 3: Smashing Lab

- One method is to write the sequence of assembly instructions into a file (say insns.s) and do the following:
- gcc -c insns.s
- objdump -d insns.o
- This will show the bytes that correspond to each instruction. You can then copy those bytes into a file.

Lab 3: Smashing Lab

- Unfortunately, a file that is a sequence of written hexadecimal bytes:
  - “78 A1 90 90 90 ...”
- ...will be read by fgets as a sequence of characters {'7', '8', ' ', ...}, when in fact you'd like these to be interpreted as hex bytes.
- The following are a few tools that will allow you to do this.

Lab 3: Smashing Lab

- xxd:
  - Creates a hex dump or the reverse
  - xxd text.txt
  - text.txt is an ascii character file. Will return the hexdump in the form of:
  - 0000000: ABBA CABB 0909 AE09 …
  - 0000010: DEAD BEEF FEED …
  - xxd -r hex.txt will take in a text file of the above format and convert that to a hex string.

Lab 3: Smashing Lab

- hexedit:
  - I know it exists, I'd recommend looking it up, but SEASnet does not seem to have it. This is an option for people who have their own Linux virtual machines or partitions.
- hex2raw:
  - Find a copy located here:
  - http://www.seas.ucla.edu/~uentao/hex2raw
  - It was a utility from one of Reinman's labs that allows you convert from hex to raw.

Lab 3: Smashing Lab

- hex2raw:
  - In order to use this, you must first change the permissions so that you can execute it. You can do so by using "chmod 0777 hex2raw" or "chmod +x hex2raw"
  - If your hexadecimal code is in hex.txt and you want to convert to a file called raw.txt, invoke the command as follows:
  - "cat hex.txt | ./hex2raw > raw.txt"
  - cat will print out contents of hex.txt
  - ...which will be “piped” in as the input to hex2raw.

Lab 3: Smashing Lab

- When you're dealing with assembly, how exactly can do you do a high level operation such as deleting a file?
- In fact, doing such a thing is a task that is usually something that only the OS can do.
- There is a class of low level functions that allows user code to coerce the OS to do work for it: syscalls.

Lab 3: Smashing Lab

- The syscalls include functionality for stuff like reading files, deleting files (under the name of “unlink”), and etc..
- In C, you would call the “unlink(const char *)” function, but there is also an assembly instruction method for performing syscalls.
- Hint: google “x86 assembly syscall”

Lab 3: Smashing Lab

- Handling randomness.
- The main difficulty with an exploit like this is that in your exploit string, you must specify the initial return address that will jump to your exploit code.
- When the stack is randomized, the absolute addresses will change. In one instance buffer buf will be located at 0x7FFF0000 and in one instance, buffer buf may be located at 0x7FFF0010.

Lab 3: Smashing Lab

- Thus, if you attempted to write your code such that your return address was 0x7FFF0010, it would only work in the instances where 0x7FFF0010 happens to be where buf currently is. Or will it?
- Say your exploit code is:
  - b8 0b 00 00 00 mov $0xb,%eax
- Say you know that your buffer is 0x40 bytes and you know that the randomization will position the buffer between 0x100 and 0x120

Lab 3: Smashing Lab

- Instead of writing b8 0b 00 00 00 to the beginning of the buffer, you can first write some throwaway bytes in the buffer, say 0x20 bytes.
- Thus, if the buffer is at 0x100, then your exploit code begins at 0x120. If the buffer is at 0x120, then your exploit code begins at 0x140.
- Now, if you jump to 0x120, you'll either hit the exploit code or be at an address that is lower than the exploit code.

Lab 3: Smashing Lab

- Now, you just need some way of writing those throwaway bytes so that %rip will execute them, do nothing, and keep going until we hit the exploit code.
- If only we had some sort of operation that does nothing and iterates the %rip.
- Some sort of instruction that does no operation. A no-operation instruction, if you will...

Lab 3: Smashing Lab

- The byte 0x90 corresponds to the nop in x86.
- If the instruction pointer hits a nop, it will execute it, doing nothing and then move on to the next instruction.
- Thus, if you pad your exploit code with a bunch of nops before the actual exploit, if you land anywhere in the string of nops, you'll simply fall through to the exploit code. This increases the chance of a successful jump to your exploit code in the face of randomness.
- This is called a “nop sled”.

Lab 3: Smashing Lab

- Great. We're ready to go. We'll pad our exploit code with a nop sled to increase the probability of landing in a good place. We've conquered randomness!
- Now, how much variability is proved by ASLR?

Lab 3: Smashing Lab

- According to Professor Wikipedia:
  - 2^N is the random offset applied to the positioning.
  - “In many systems, 2^N can be in the thousands or millions; on modern 64-bit systems, these numbers typically reach the millions at least.”

Lab 3: Smashing Lab

- According to Professor Wikipedia:
  - 2^N is the random offset applied to the positioning.
  - “In many systems, 2^N can be in the thousands or millions; on modern 64-bit systems, these numbers typically reach the millions at least.”
- On second thought, lets just turn randomization off.

Instruction Level Parallelism

- The optimizations presented in chapter 5 are all serial optimizations that can marginally improve execution time.
- Marginal improvement can still be significant depending on how much time is spent on the section that was improved.

Instruction Level Parallelism

- However, anything a traditional program does is limited by the fact that a single program or a single thread is expected to the entire work of a program.
- If we have to iterate over a length 100 array and assign a value to each element, there's not a whole lot a single thread can to do get around this.
- But what if we have two processors dedicated to the same task?

Instruction Level Parallelism

- If instead, we have two processors each working on half of the data, we can finish the array assignment in half of the time, which provides a speedup of 2, which was already faster than the best loop unrolling case.
- Plus, the performance improvement provided by splitting the task among independent agents doesn't degrade nearly as quickly. If we had 100 processors, we could complete the task in the time it takes a processor to execute a single instruction.

Instruction Level Parallelism

- It's this observation that has driven the industry towards parallelism rather than simply increasing clock speed when it comes to improving performance.
- Why can't we just keep increasing the clock speed?

Instruction Level Parallelism

- Power Wall:
  - As the clock rate increases, the power needed to operate everything increases. In addition to requiring a lot of power, much of it is leaked/dissipated as heat.
- Memory Wall:
  - Recall from class that processor speeds have increased significantly since the 80's.
  - Also recall the RAM access time has remained essentially the same.

Instruction Level Parallelism

- Memory Wall:
  - It doesn't help to reduce the processor's speed from 1 ns per instruction to .5 ns per instruction if accessing RAM takes 200 ns and you have a reasonable proportion of memory accesses.
  - The bottleneck is still in RAM access and via Amdahl's Law, the bottleneck is the thing we'd need to reduce in order to get a significant improvement.

Instruction Level Parallelism

- Parallelism is such a powerful method of achieving speed-up that we parallelize everything:
  - Multiple computers running in parallel to form server clusters
  - Multiple processors running in parallel to run different programs simultaneously
  - Multiple threads/processes running in parallel.
  - Multiple instructions within the same thread being executed in parallel.

Instruction Level Parallelism

- Instruction Level Parallelism (or ILP) is that last bullet point.
- To help understand the ways that this is done, it's easier to think in terms of the nebulous “micro-operations”.
- But what are they?

Instruction Level Parallelism

- Recall that the x86 ISA follows the CISC school of thought (Complex Instruction Set Computing).
- This means that the x86 ISA defines many different complicated instructions and is thus a “complex instruction set”.
- Hence: all of the different addressing modes, a million different conditional operations, billions of different operand combinations for “muls”
- Why is something like this a good thing?

Instruction Level Parallelism

- When each instruction is powerful and can do a lot of things, this means that code is more compact. This was useful back in the day when memory constraints were a real problem.
- Each instruction is also closer to the high level languages that we, the users, know and love.
- Ultimately, the ISA defines many instructions so that we only need to execute a few instructions (a “few”).
- Why might CISC be undesirable?

Instruction Level Parallelism

- Having instructions that are more complicated comes at the cost of being slower to execute.
- As the machine is expected to execute more and more complicated instructions, the complexity of the architecture design goes up.
- Having extremely complicated instructions also made it difficult to implement ILP.
- As a result, everything started moving towards RISC or Reduced Instruction Set Computing.

Instruction Level Parallelism

- But what about poor x86?
- ...and for that matter, if CISC isn't used any more, why is x86 so prominent?

Instruction Level Parallelism

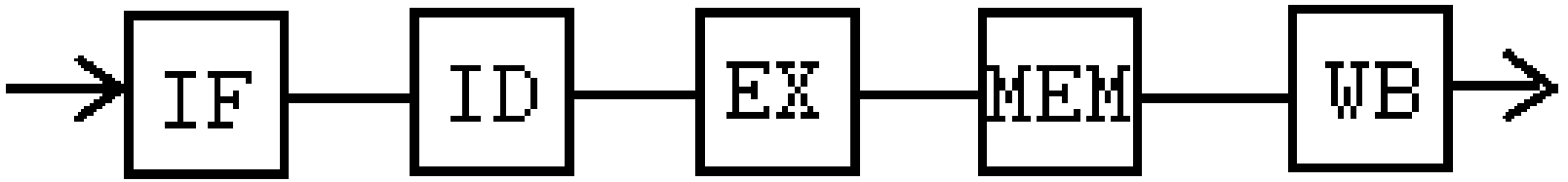
- Mostly historical. After everything x86 has done for us, it'd be cruel to leave it out in the cold.
- But then, how can we do ILP with the CISC x86?
- Secretly have another compilation stage, one that compiles normal x86 into the micro operations
- Essentially, it compiles CISC x86 into RISC-like x86 micro-operations.

Instruction Level Parallelism

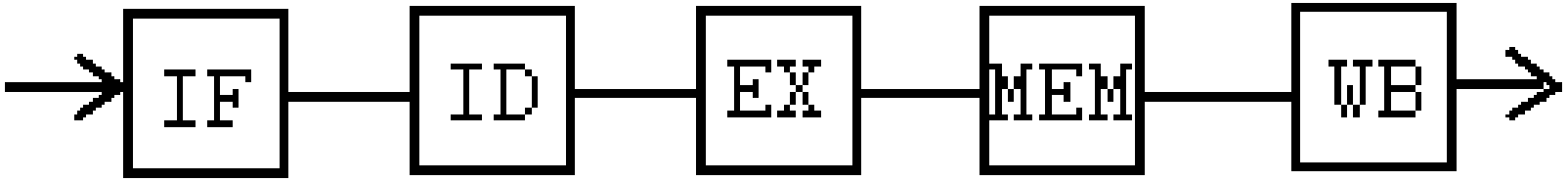
- RISC-like instructions are simple and generally follow a few principles:
  - Operation come in a few basic forms. None of that suffix nonsense.
  - In order to load from memory, we have to directly load it with a load operation.
  - We can't directly add to or from a memory address.

Instruction Level Parallelism

- In order to execute a single simple micro operation, it goes through several steps or “stages” to complete. In the common academic example, there are 5 stages.
- This is known as the pipeline.

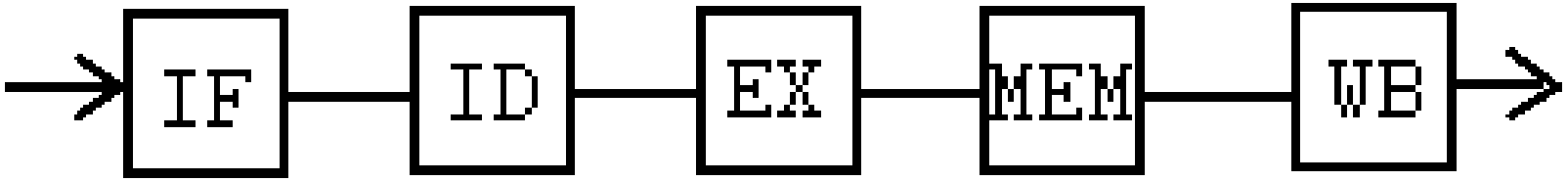


Instruction Level Parallelism



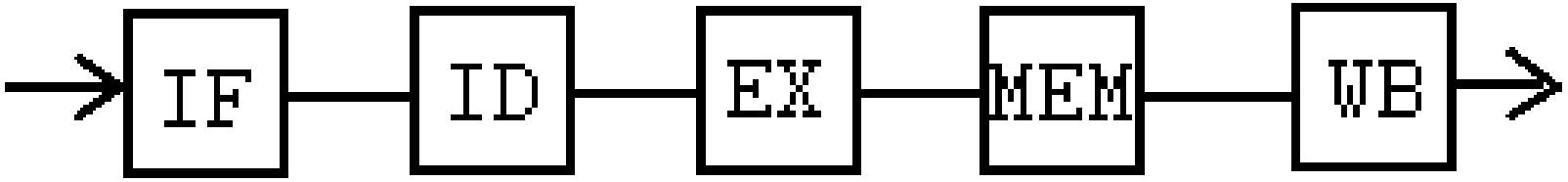
- Each stage corresponds to a physical component in the processor.
- IF (Instruction Fetch): Takes as input the address of the instruction (%rip). Then it finds the actual instruction in memory.
- ID (Instruction Decode/Register Read): Given the instruction bytes, get the appropriate values from memory.

Instruction Level Parallelism



- EX (Execution): Execute any arithmetic operation that the instruction needs to do.
- MEM (Memory): Read/write from memory if necessary.
- WB (Write back): Write to register if necessary.

Instruction Level Parallelism

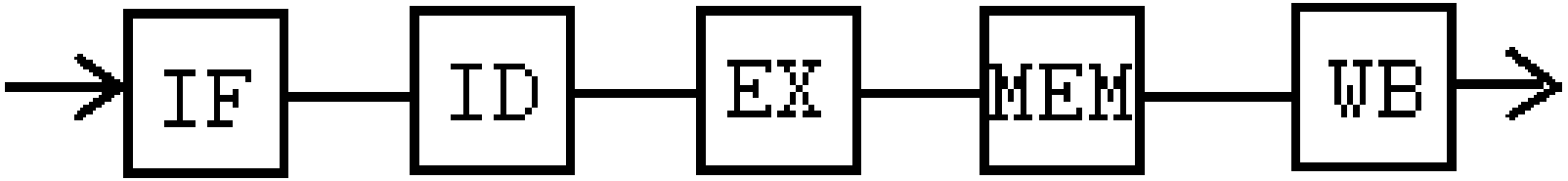


- Assume that each block takes a single cycle to execute. A single instruction would take 5 cycles to complete.
- The latency of each instruction is 5 cycles.
- Our throughput of this system is 1 instruction per every 5 cycles.

Instruction Level Parallelism

- Consider the execution of: add %eax, %ebx
- Step 1. The instruction address is used to fetch the instruction
- Step 2. The values from %eax and %ebx are loaded
- Step 3. %eax is added to %ebx.
- Step 4. The add instruction doesn't use memory
- Step 5. Save the value back into %ebx

Instruction Level Parallelism



- At any given point in time, we are only using one of five available components.
- This is an opportunity to execute multiple instructions at the same time.

Instruction Level Parallelism

- Say we have the following instruction sequence:
  - 1. add %eax, %ebx
  - 2. sub %ecx, %edx
  - 3. xor %esi, %edi
- Say the execution of this snippet begins at time 0.
- At time 0: the add would be in the IF stage.
- At time 1: the add would be in the ID stage and the sub would be in the IF stage.
- At time 2: the add would be in the EX stage...

Instruction Level Parallelism

- Based on this, how long does each instruction take to complete?

Instruction Level Parallelism

- Based on this, how long does each instruction take to complete?
  - It still takes five cycles for any given instruction to complete. The latency has not changed.
- However, how many instructions are completed per cycle?

Instruction Level Parallelism

- Based on this, how long does each instruction take to complete?
  - It still takes five cycles for any given instruction to complete. The latency has not changed.
- However, how many instructions are completed per cycle?
  - 1 instruction is completed per 1 cycle rather than 5 cycles. The throughput has increased from 1/5 instructions per cycle to 1 instruction per cycle. THAT is a huge gain.

Instruction Level Parallelism

- This example of ILP can simultaneously execute 5 instructions at any given time.
- ...or is this perhaps a little too idealized?
- What are some cases where we wouldn't be able to achieve this optimal case?

Instruction Level Parallelism

- Data Hazard: Without pipelining, when each instruction begins, it can safely assume that the previous instruction has completed. With pipelining, that assumption is no longer true.
- Consider:
- add %eax, %ebx
- add %ebx, %ecx
- The first add will not have saved the new value to %ebx by the time the second add needs to read from it.

Instruction Level Parallelism

- Control Hazard:
- When a conditional operation enters the pipeline, it's expected that another operation will enter the pipeline in the next cycle.
- But what instruction should that be if the first operation was conditional?
- We guess using branch prediction, but if we guess wrong, we wasted time doing unnecessary work.

Current State of ILP

- This is just the tip of the iceberg:
  - Multi-issue/Superscalar processors: pipelined as usual, but each stage can accommodate multiple instructions at once.
  - Deeper pipelines (modern processors have 12-19 stages)
  - Out of Order (OoO), that is avoiding data dependencies by executing the instructions out of order as possible.

Current State of ILP

- The pipeline processor covered in these slide are a basic overview as to how a single processor or a single execution unit can be used in parallel.
- From here on out, the book assumes multi-issue and out-of-order processors.
- This assumption is key to understanding the dataflow diagrams in 5.7

End of

The Sixth Week

-Four Weeks Remain-