CIS 4360
Secure Computer Systems

Buffer Overflows

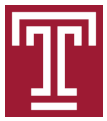
Professor Qiang Zeng

Spring 2017

Some slides are courtesy of Dr. David A. Wheeler

Hard link vs. soft link

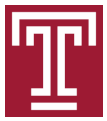
- How are they implemented?
  - When you create a hard link, you add a directory entry that points to the inode of the target file
  - When you create a soft link, you create a file whose metadata (stored in the created file's inode block) records that it is a soft link and the target file's name
- The hard link will never be dangling (there is a reference count in the inode of the target file), while the soft link may get dangling
- The hard link can only point to a file in the same partition, while a soft link can point to any file or dir



Previous Class

How doe the sticky bit improve the security of using the directory /tmp?

Once a user creates a file in /tmp, only the directory owner (root in this case) or the file owner can delete or rename the file

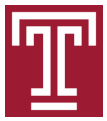


Previous Class

Describe, even with the sticky bit, why /tmp is till insecure to use? What is the lesson?

(1) If the attacker knows the name of the file to be created by a victim, the attacker can create a file with that name to hurt the confidentiality and integrity in case the exclusive flag (O_EXCL) is not used in "open" (e.g., the "stat-open" vul.)
(2) Even if the exclusive flag is used in "open" by the victim, the attacker can launch DoS attacks by creating a file with the same name ahead of the victim

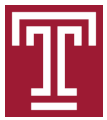
Lesson: try your best not to use publicly writable directory



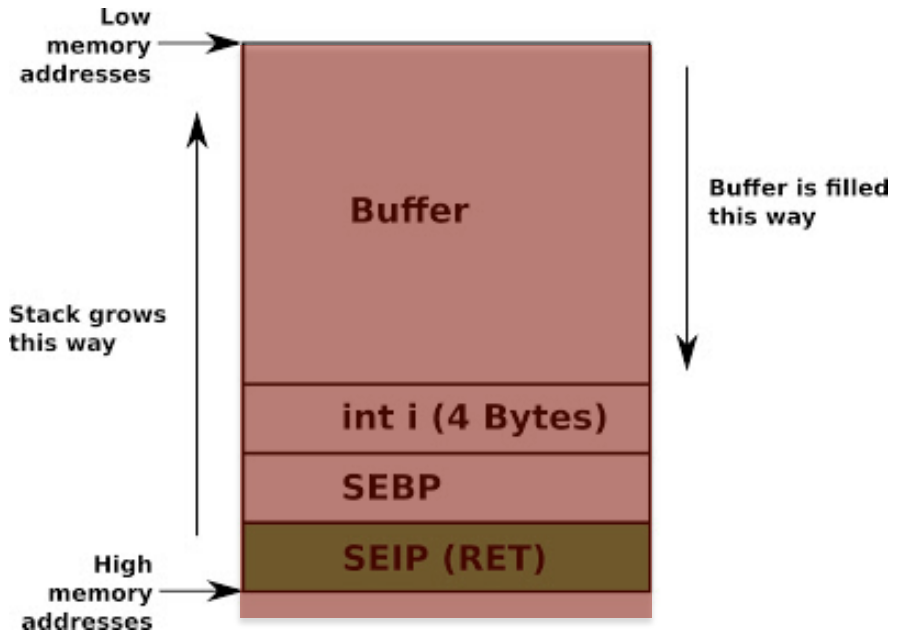
Buffer Overflow

- Also called "Buffer Overrun"
- It occurs when the access to a buffer is out of the bounds of the buffer, e.g.

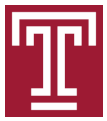
```
char dst[5];
strcpy(dst, "some user provided input");
```



Common and dangerous

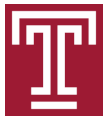


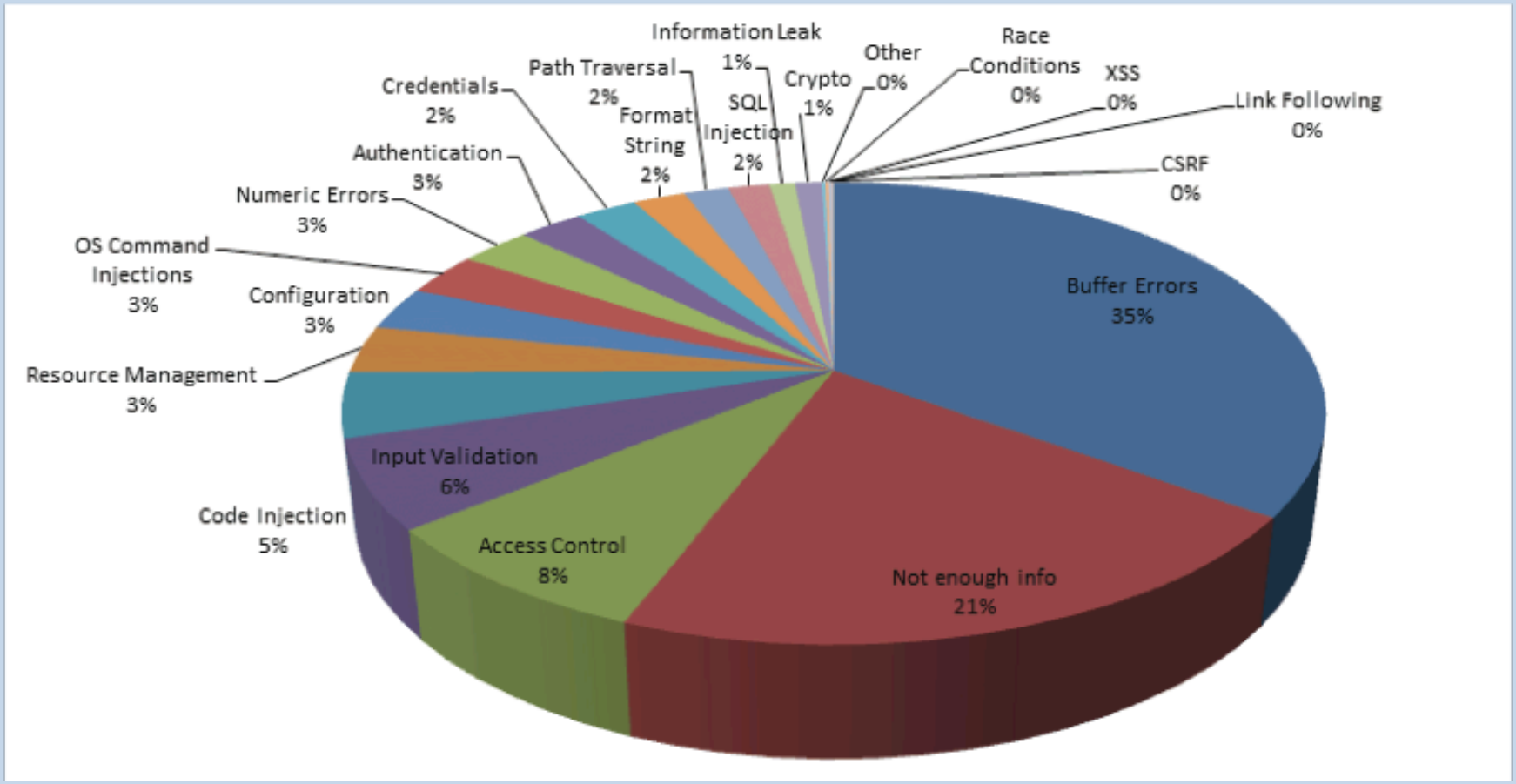
E.g., strcpy(buffer, long-src-str);



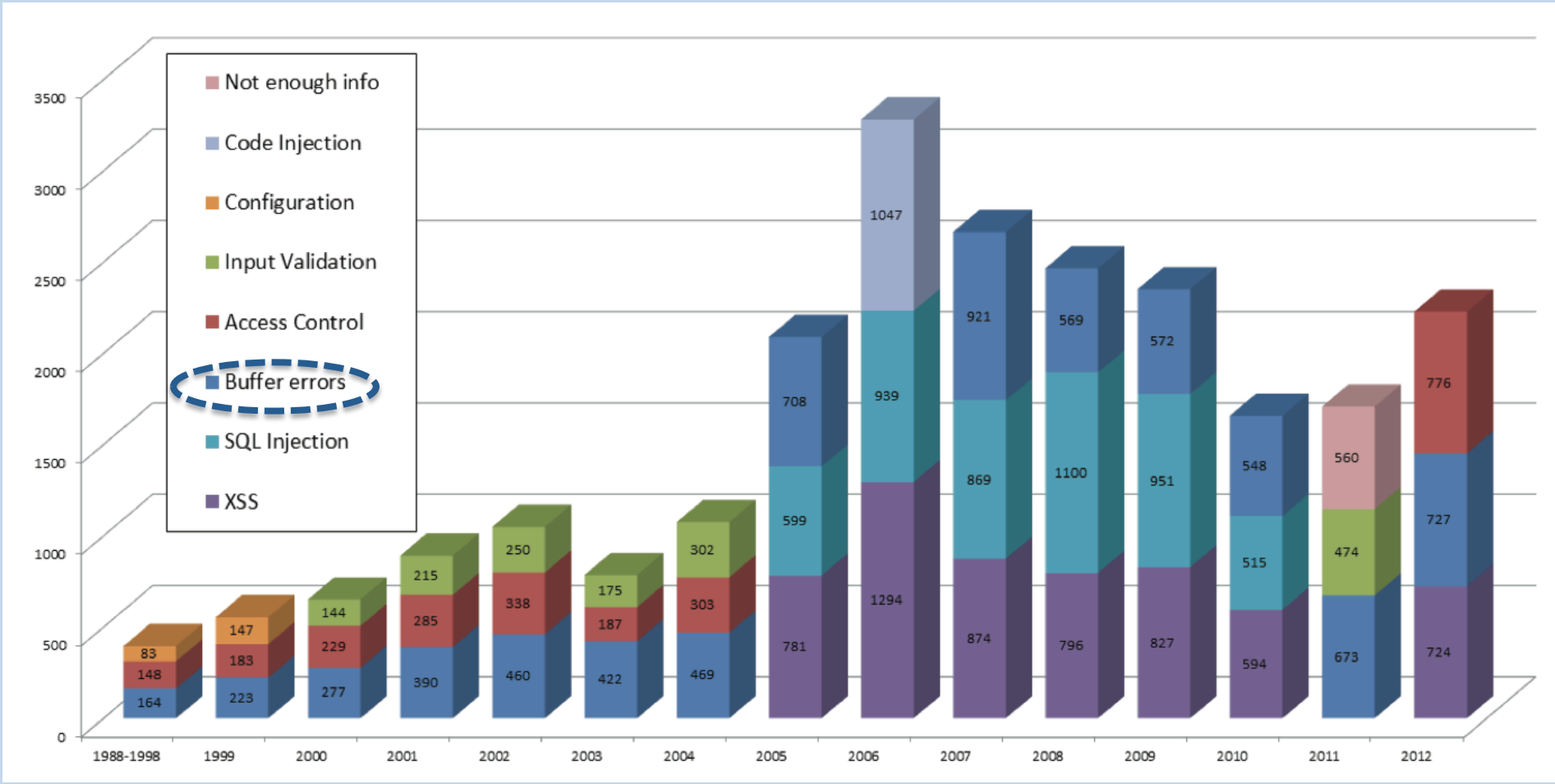
Types of buffer overflows

- Direction
  - Access beyond upper bound: overflow
  - Access beyond lower bound: underflow
  - People usually call both as Buffer Overflows
- The out-of-bound access can be read or write
  - Overread (leak info)
  - Overwrite (manipulate memory content)
- The access can occur on stack or heap
  - Stack-based / heap-based buffer overflows



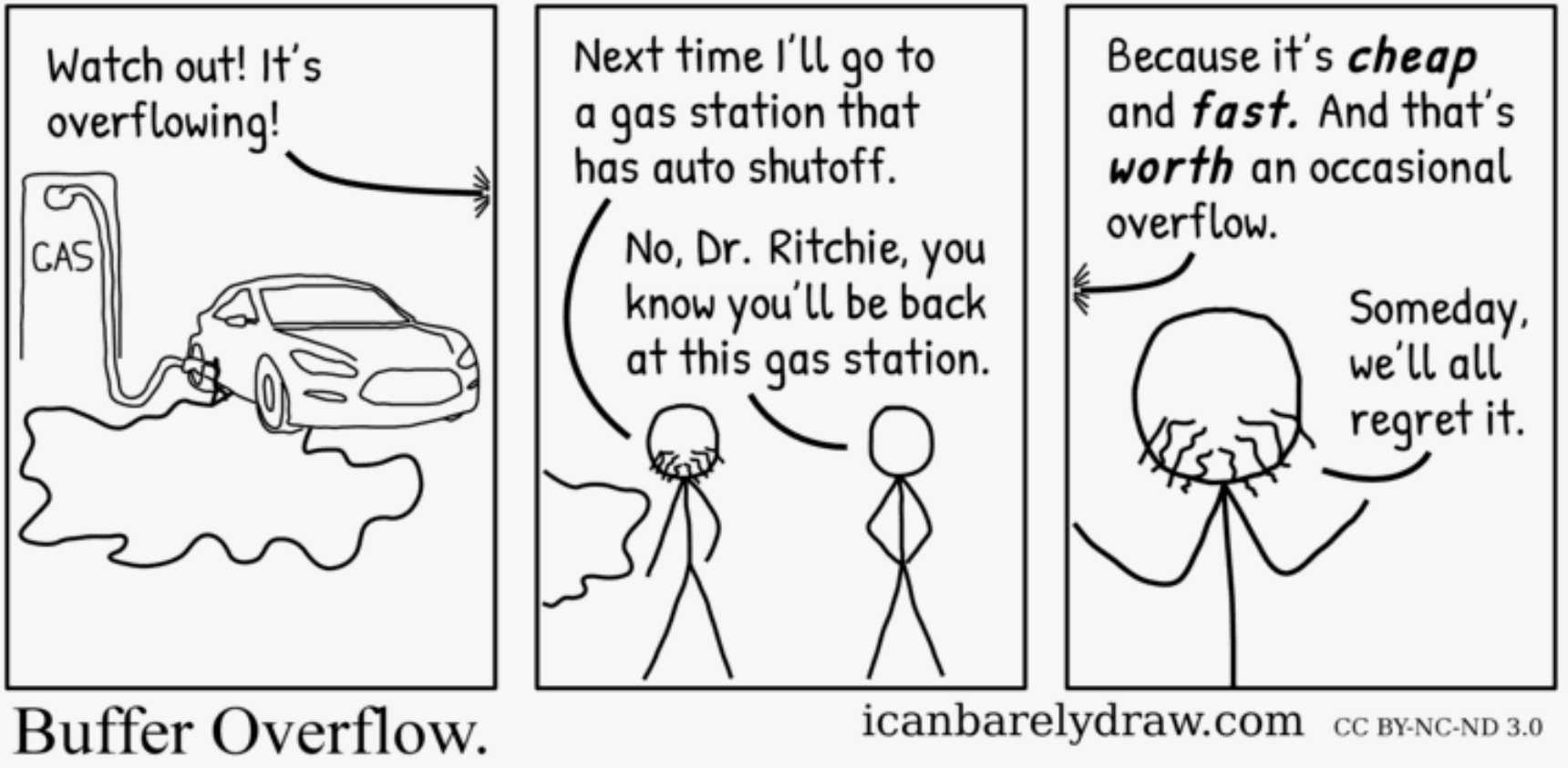


“If buffer overflow vulnerabilities could be effectively eliminated, a very large portion of the most serious security threats would also be eliminated.” C. Cowan



"If buffer overflow vulnerabilities could be effectively eliminated, a very large portion of the most serious security threats would also be eliminated." C. Cowan

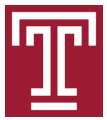
"It is now safe to declare the buffer overflow the vulnerability of the quarter-century." Y. Younan



Almost all the operating system kernels, device drivers, compilers, JVM, and Python interpreters are coded in C and C++

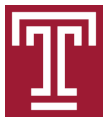
Famous attacks due to buffer overflows

- 1988: Morris worm – took down Internet
  - Includes buffer overflow via gets() in fingerd
- 1999: RSA crypto reference implementation
  - Subverted PGP, OpenSSH, Apache's ModSSL, etc.
- 2001: Code Red worm – buffer overflow in Microsoft's Internet Information Services (IIS) 5.0
- 2003: SQL Slammer worm compromised machines running Microsoft SQL Server 2000
- ~2008: Twilight hack – unlocks Wii consoles
  - Creates an absurdly-long horse name for "The Legend of Zelda: Twilight Princess" that includes a program
- 2014: Heartbleed - buffer overread vul. in OpenSSL lib
  - Half a million web servers are vulnerable to Heartbleed attacks



Background – representation of C string

- In C/C++ programming
  - strlen("Hello") returns 5
  - The space needed by "Hello" is 6
  - Each string needs a \0 to flag the ending
- memcpy(str_dst, str_src, strlen(str_src));
  - Bug 1: buffer overflow when str_dst is too small
  - Bug 2: the ending \0 is not copied



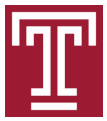
Background - Calling a procedure

- Given this C program:

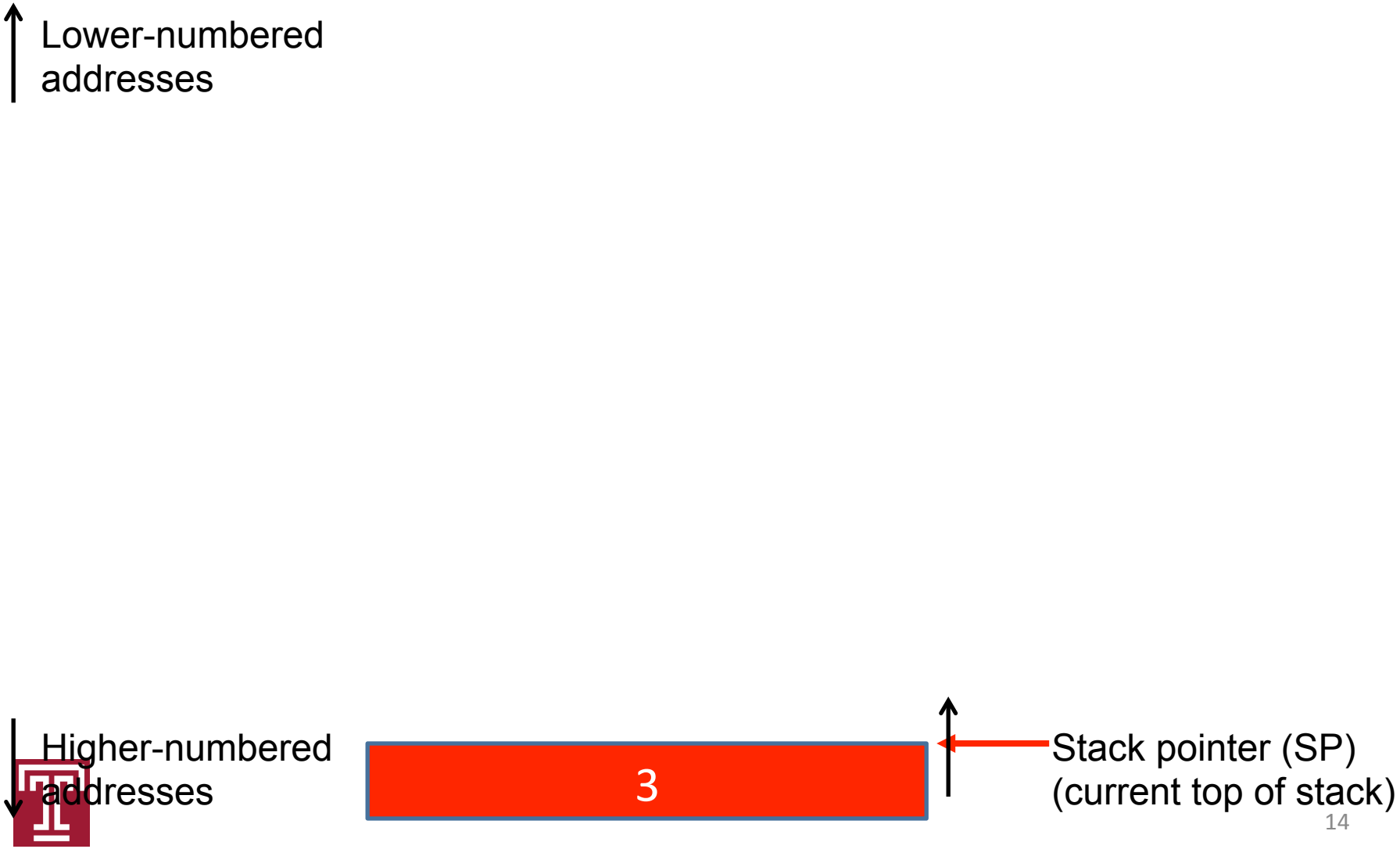
```
void main() {
  f(1,2,3);
  exit(0);
}
```

- The invocation of f() generates code (parameters are pushed from right to left):

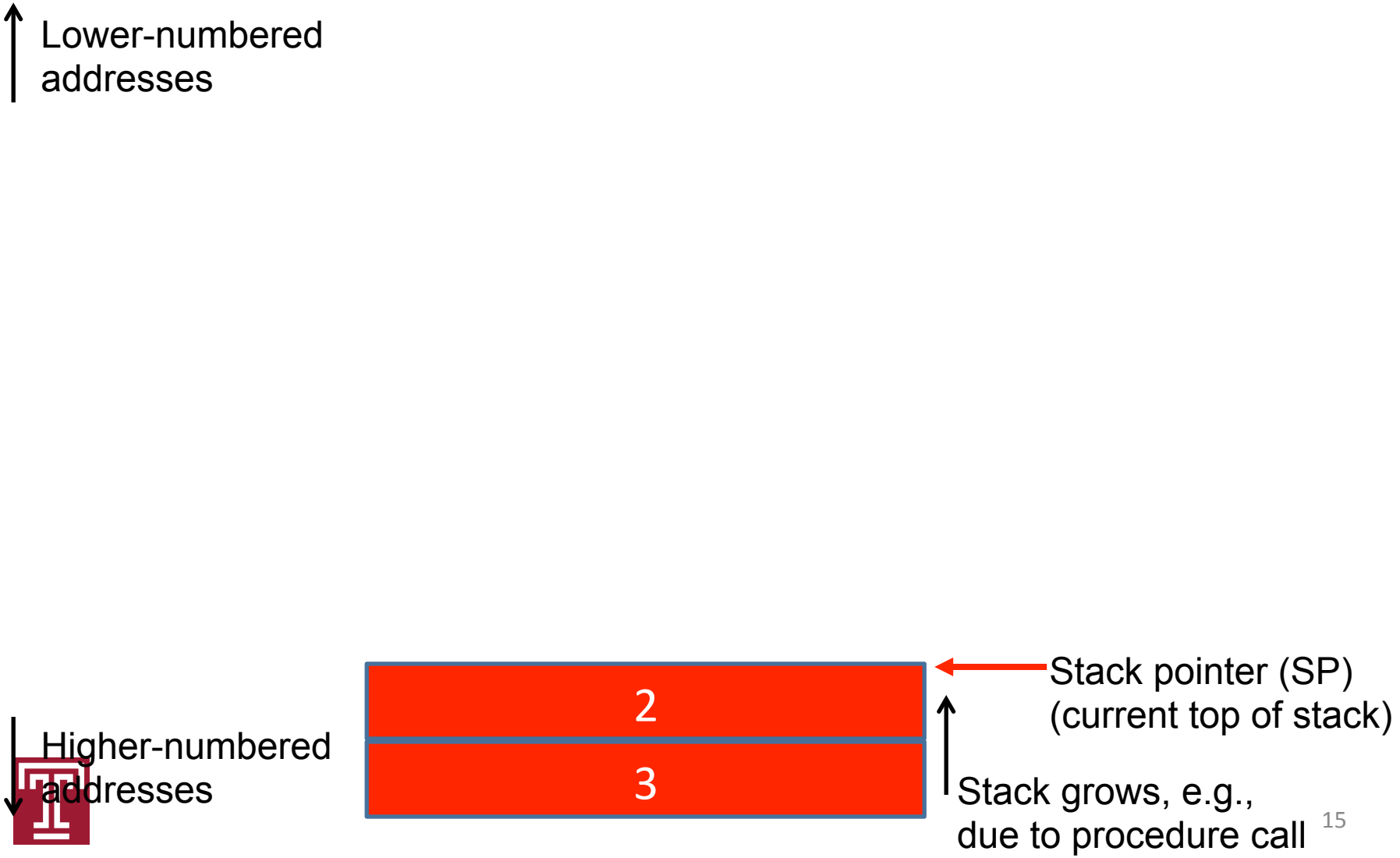
```
pushl $3 ; // constant 3
pushl $2 ;
pushl $1
call f // push the IP on stack and jump to f()
```



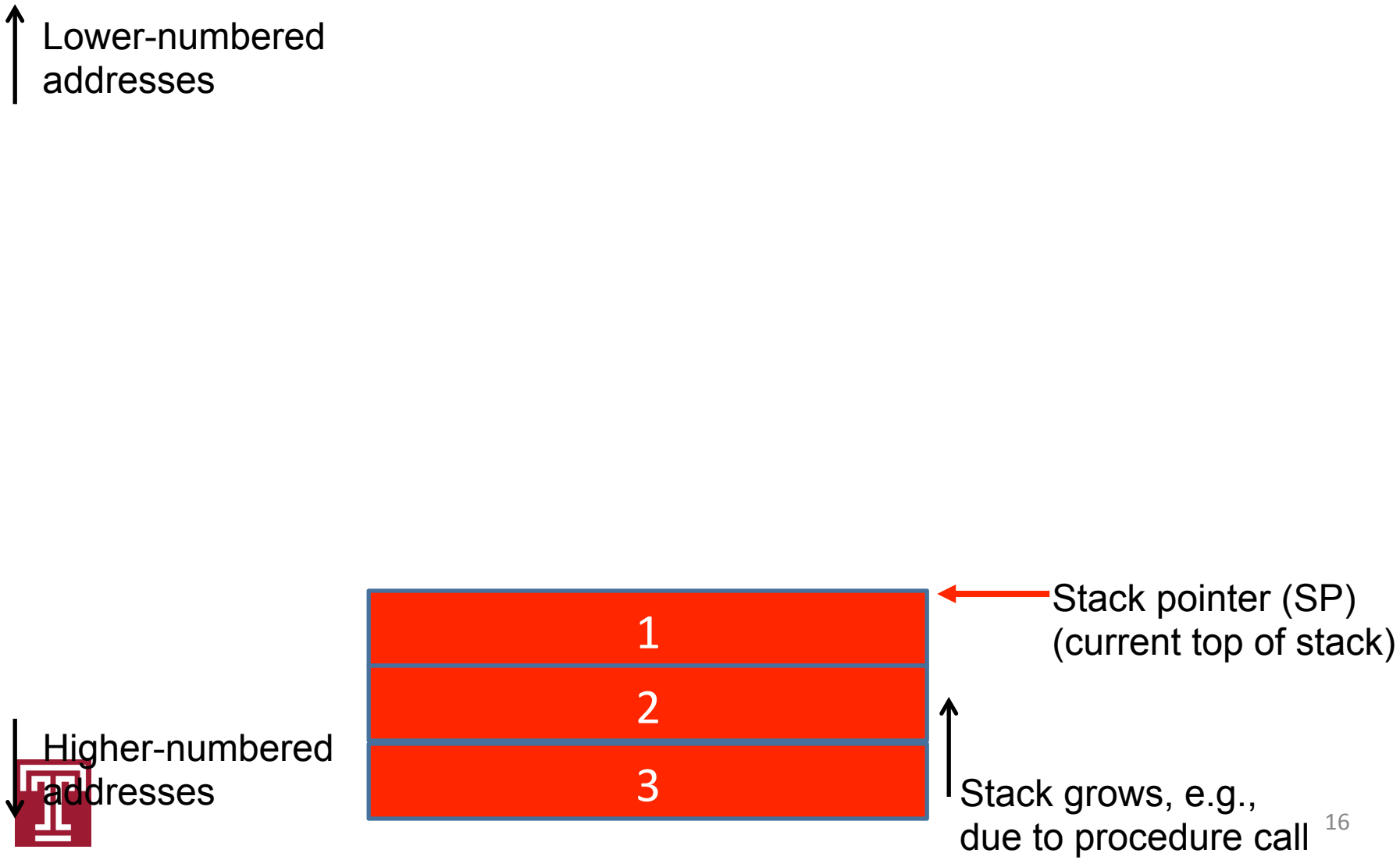
Background: Stack



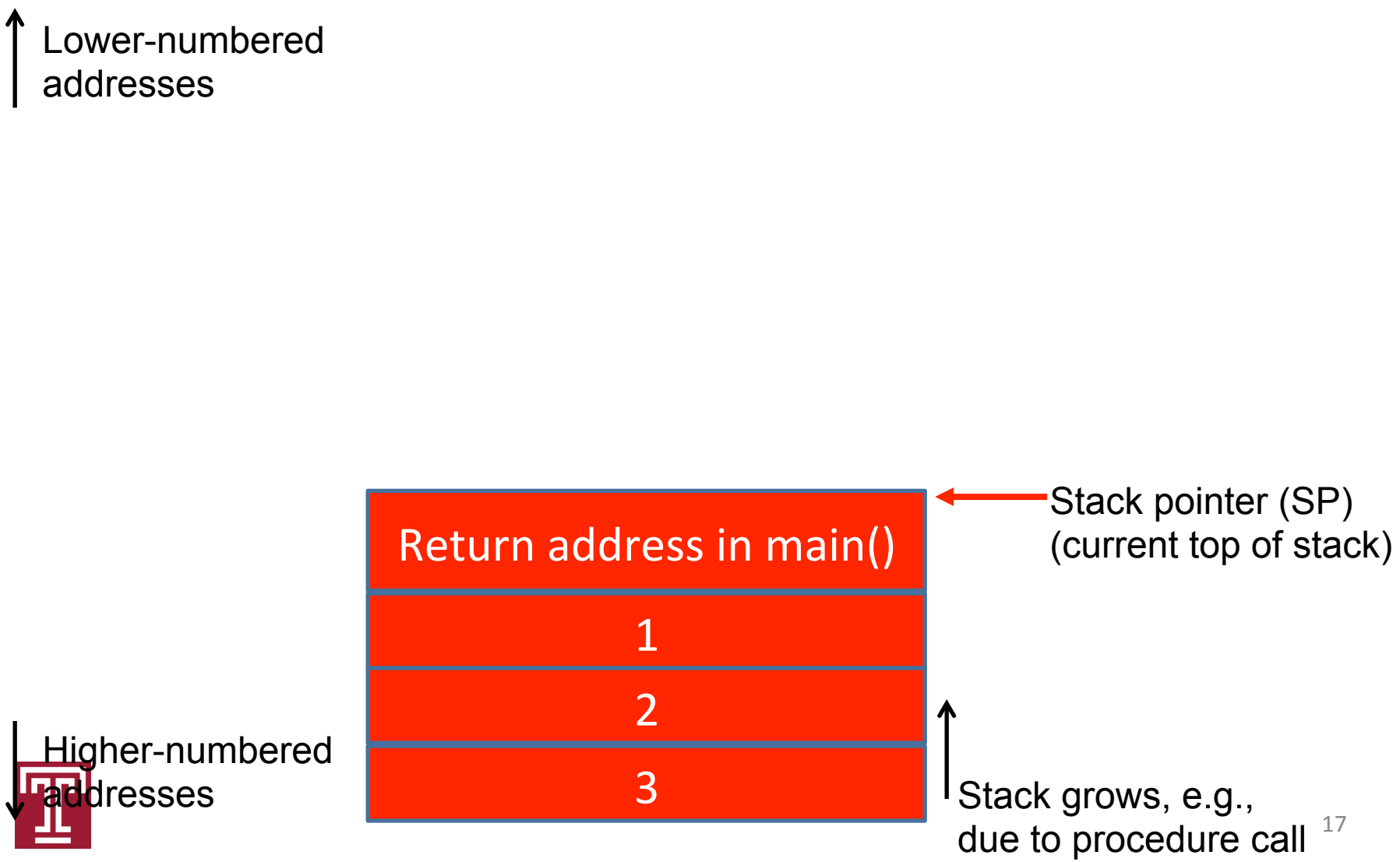
Background: Stack



Background: Stack



Background: Stack



Background: Function prologue

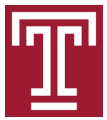
- Imagine f() has local variables, e.g. in C:

```
void f(int a, int b, int c) {
  char buffer1[5];
  char buffer2[10];
  strcpy(buffer2, "This is a very long string!!!!!!!");
}
```

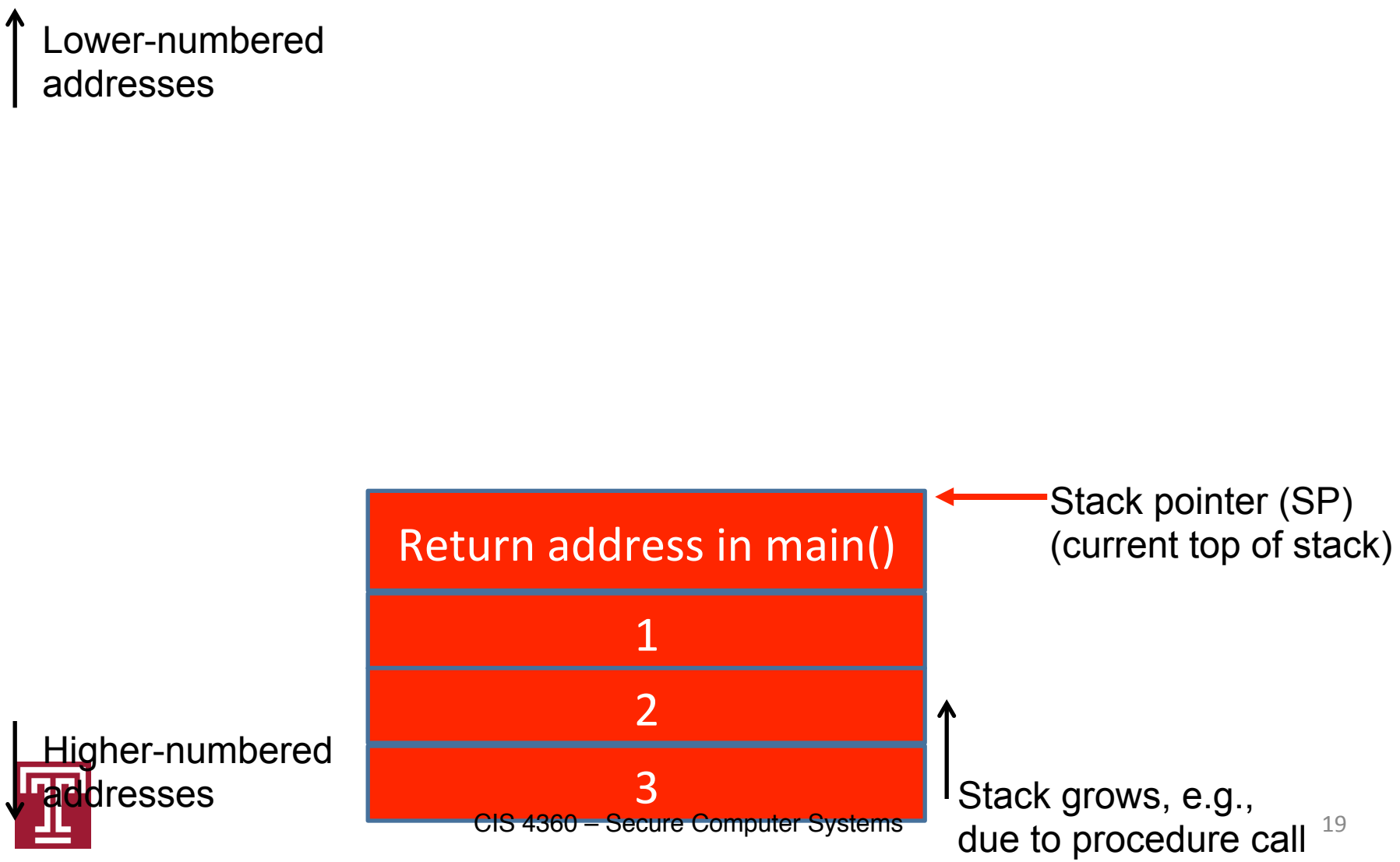
- Typical x86-32 assembly on entry of f() ("prologue"):

```
pushl %ebp  ; Push old frame pointer (FP)
movl %esp,%ebp ; New FP is old SP
subl $20,%esp ; New SP is after local vars
  ; "$20" is calculated to be >= local var space
```

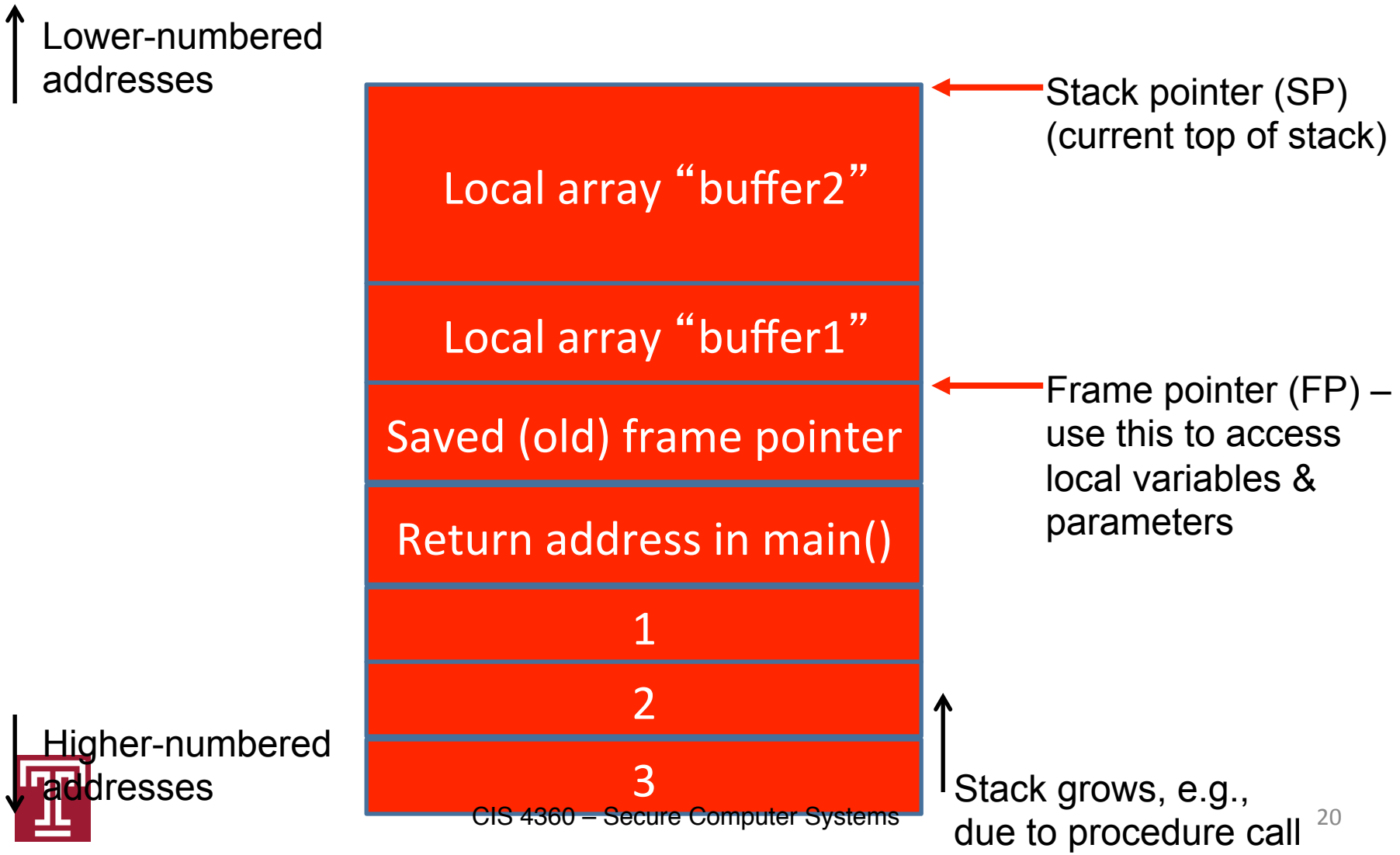
In the assembly above, ";" introduces a comment to end of line



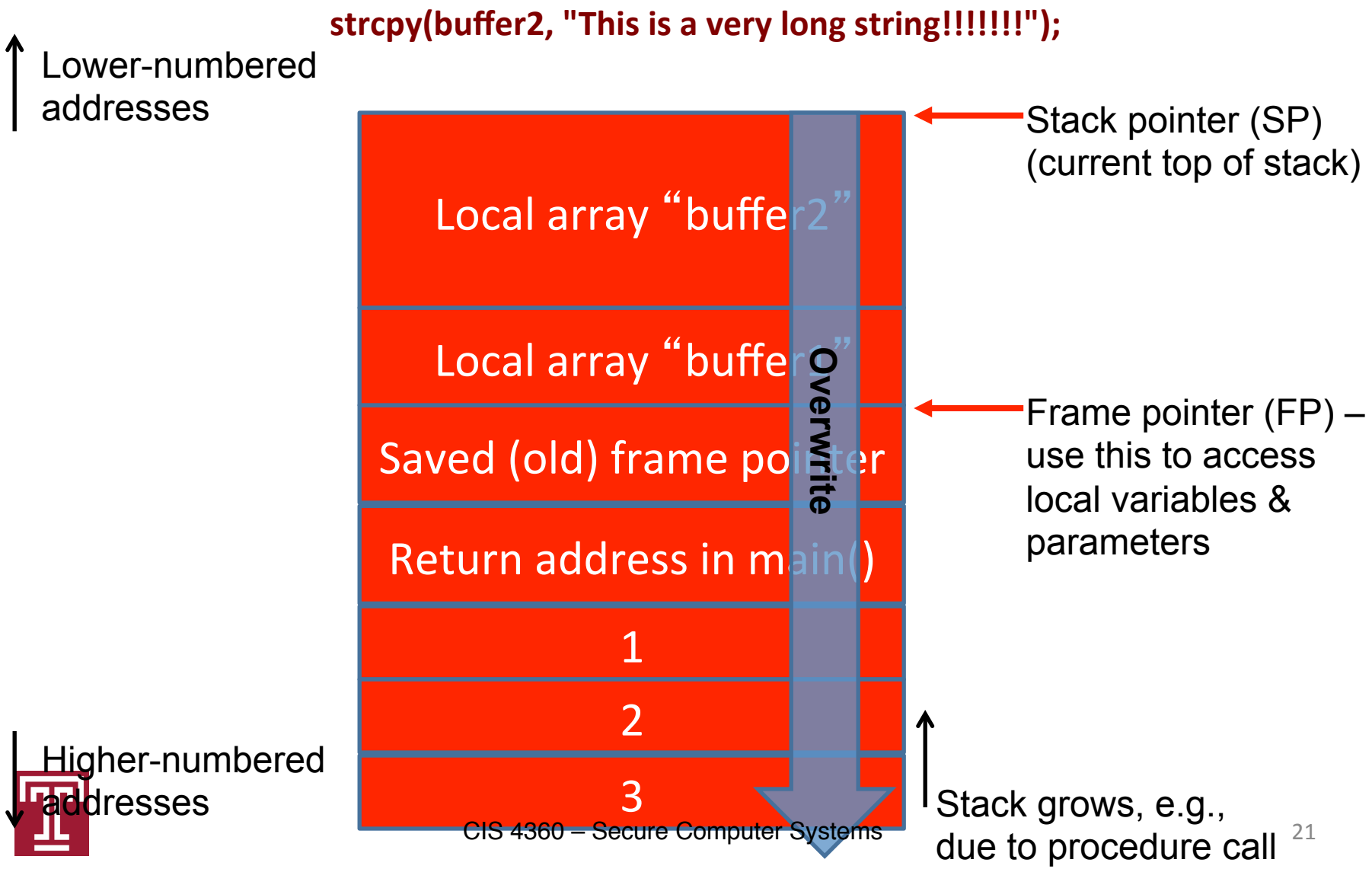
Immediately after call instruction



After prologue

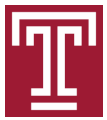


Overflowing buffer2



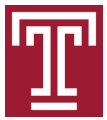
What happens if we write past the end of buffer2?

- Overwrites whatever is past buffer2!
  - As you go further, overwrite *higher* addresses
- Impact depends on system details
- In our example, can overwrite:
  - Local values (buffer1)
  - Saved frame pointer
  - Return value (changing what we return to)
  - Parameters to function
  - Previous frames

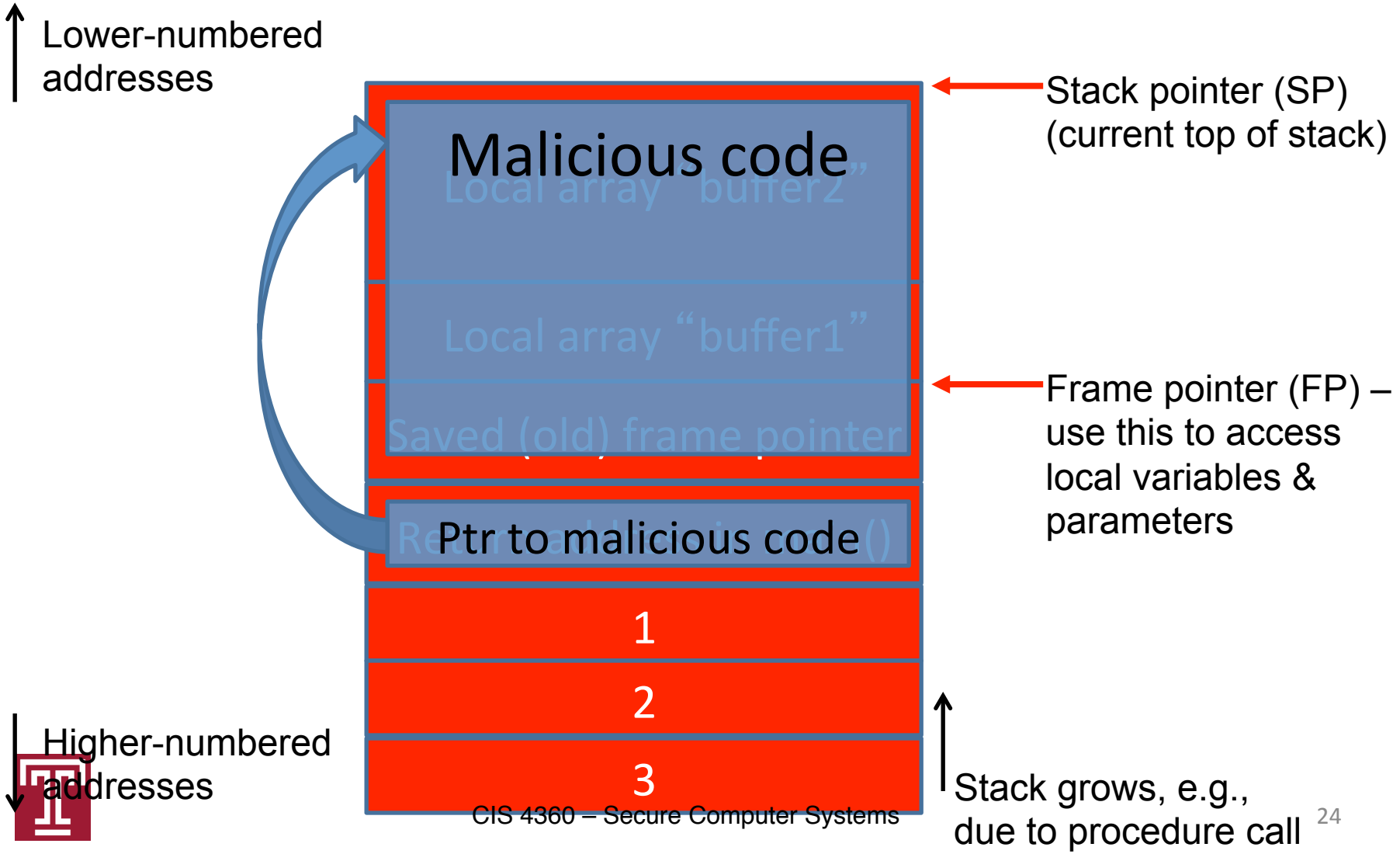


Inserting code in the buffer overflow attack (e.g., shell code)

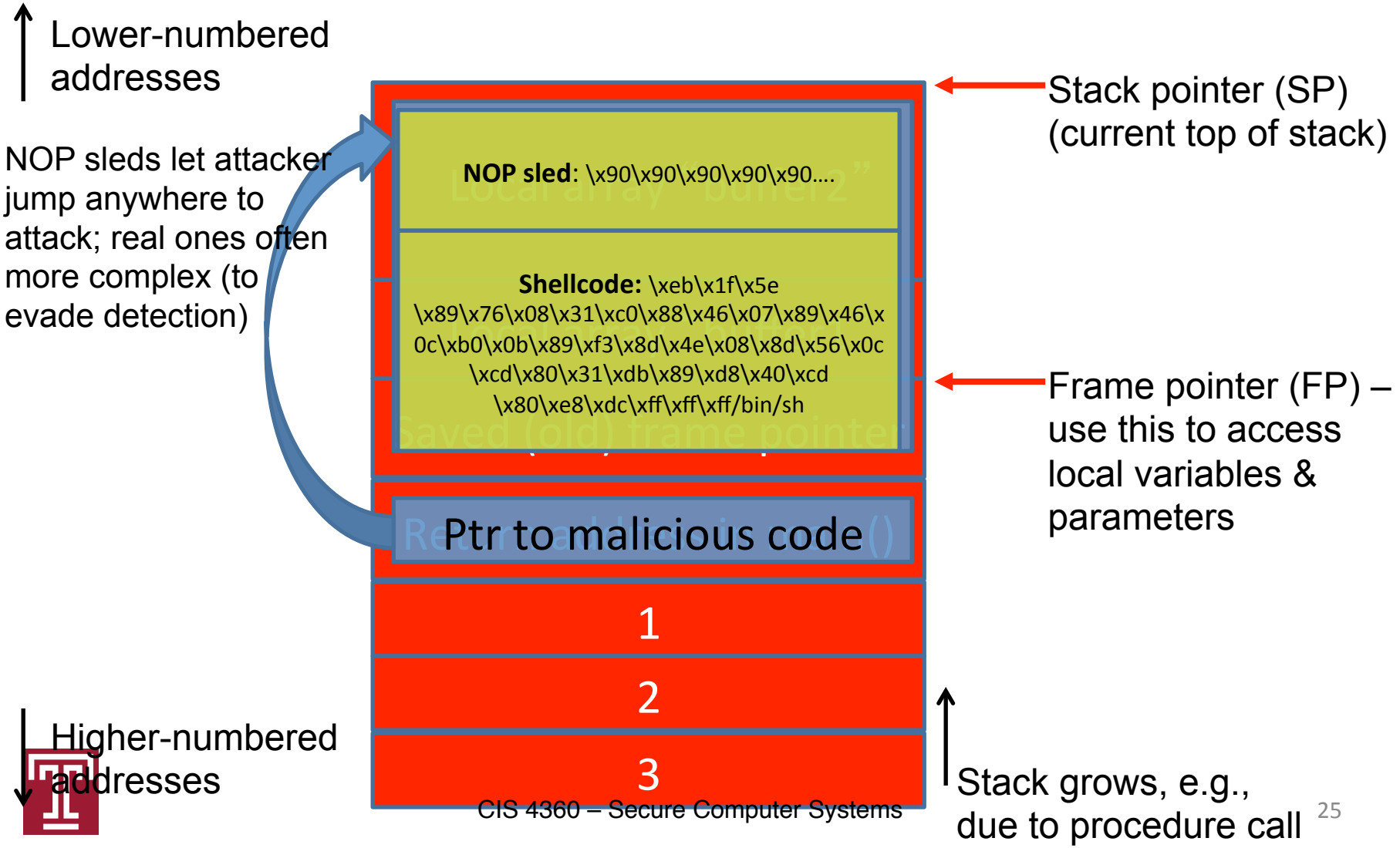
- Attacker can also include machine code that they want us to run
- If setting the "return" value to point to this malicious code, the victim will run that code on return
- Significant portion of "Smashing the Stack" paper describes how to insert such code



Stack:
One possible result after attack



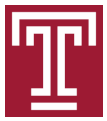
Stack:
One possible result after attack



Other types of attacks possible with a stack buffer overflow

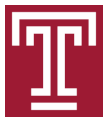
- Make "return" point to *existing* code that the attacker wants us to run *now*
  - Return to libc
  - Return-oriented programming
- Change value of adjacent local variables, e.g., a function pointer or a critical bool variable
- Change value of parameters

... and so on



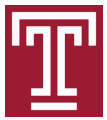
Smashing elsewhere

- "Heap" contains dynamically-allocated data
  - "new" (Java/C++), malloc (C), etc.
- "Data" contains global data
- If attacker can overwrite beyond buffer, he can control other values (e.g., stored afterwards)
  - Values of other structures
  - Heap: Heap maintenance data (e.g., what's free/used)



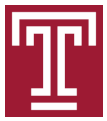
Obvious solution in C

- "Obvious" solution when using C is to always check bounds
  - E.g., if (strlen(src) + 1 > dst_buffer_size) return error;
- However…



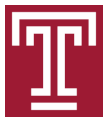
Many C functions don't check bounds

- gets(3) – reads input without checking. Don't use it!
- strcpy(3) – strcpy(dest, src) copies from src to dest
  - If src longer than dest buffer, keeps writing!
- strcat(3) – strcat(dest, src) appends src to dest
  - If src + data in dest longer than dest buffer, keeps writing!
- scanf() family of input functions – e.g., scanf("%s", buf);
  - scanf(3), fscanf(3), sscanf(3), vscanf(3), vsscanf(3), vfscanf(3)
  - Many options don't control max length (e.g., bare "%s")
- It's not just functions; ordinary loops can overflow



Do not use strncpy()

- First thing first, do not use strncpy(dst, src, n)
  - Very weird behaviors
  - copy at most n characters from src to dst (the ending null is not guaranteed)
  - extra space will be padded with 0's (so inefficient)

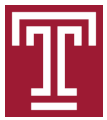


Solution 1: strlcpy/strlcat -- OpenBSD only

- Example code:

```
char    *dir, *file, pname[MAXPATHLEN];
if (strlcpy(pname, dir, sizeof(pname)) >= sizeof(pname))
        goto    toolong;
if (strlcat(pname, file, sizeof(pname)) >= sizeof(pname))
        goto    toolong;
```

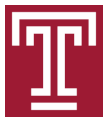
- A variant of strcpy/strcat that truncates the result to fit in the destination buffer
  - Always \0-terminates if dest has any space
  - strlcpy doesn't \0-fill the extra dst space
  - Easy to detect if terminates "in the middle" => Returns "bytes would have written"
  - if (strlcpy(dest, src, destsize) >= destsize) … // truncation detected!



Solution 2: Best solution – snprintf

- int snprintf(char * s, size_t *n*, const char * format, ...);
  - Writes output to buffer "s" up to *n* bytes
  - Always writes \0 at end if n>=1 (hooray!)
  - Returns "length that would have been written (excluding the ending null) if n is large enough" or negative if error, so result-checking can be slightly annoying
- Sample:

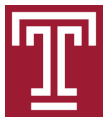
```
len = snprintf(buf, n, "%s", src);
if (len < 0 || len >= n) … // handle error/truncation
```



Question

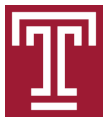
Write the equivalent code of “n = strlcpy(dst, src, len);”

```
n = snprintf(dst, len, "%s",      src);
```



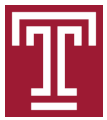
Solution 3: C++ std::string class (resize)

- If using C++, avoid using char* strings
- Instead, use std::string class
  - Automatically resizes
  - Avoids buffer overflow
  - E.g., str1 = str2; str1.append(str2);



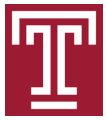
The System Has Done the Following…

- Canary
  - Insert a secret value (cookie) between the return address and the buffer on the stack at call entry, and check the canary integrity at the call exit
  - If the canary is corrupted, a buffer overflow is detected
  - Very effective against stack based overflows
- DEP: Data Execution Prevention
  - Data on stack and heap cannot be executed
  - Prevent injected code from being executed
- ASLR (Address Space Layout Randomization)
  - Scramble the locations of libraries and executables
  - So attackers do not know how to reuse their code



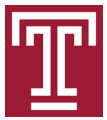
The Compiler Can Do the Following…

- Enabled by default in may systems, such as Ubuntu
- -fstack-protector --param=ssp-buffer-size=4
  - Any function that has a buffer whose size >=4 will be protected with canary checking
- -D_FORTIFY_SOURCE=2
  - Add overflow compile-time/runtime checking for strcpy/ memcpy
  - Needs –O1 to take effect; -O2 is preferred
- To disable the protection above (very bad; but sometimes you want to be crazy)
  - -fno-stack-protector -D_FORTIFY_SOURCE=0



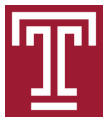
Debugging Tools

- Valgrind –memcheck: how to use it?
  - $valgrind ./your-exe
- Google's "Address sanitizer"
  - faster but not so powerful as Valgrind



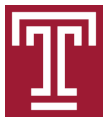
Recap

- Buffer overflows can be devastating
  - C/C++/Objective-C vulnerable to them
  - Most other languages not natively vulnerable
  - But many components/languages in C/C++
- The system has done a lot for us (but it is insufficient)
  - Canary
  - DEP
  - ASLR
- What you should do
  - Always enforce bounds checking in your code
  - Use snprintf/strlcpy/strlcat in your code
  - Do not use strncpy



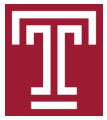
Writing Assignments

- What concrete attacks can be launched by exploiting a buffer overflow attack? List at least four.
- Compare strcpy, strncpy, strlcpy, and snprintf
- Describe Canary, DEP, and ASLR



References

- Buffer Overflows and friends (one of the series lectures of “Secure programming HOWTO”)
  - http://www.dwheeler.com/secure-class/
  - https://www.dwheeler.com/secure-class/presentations/Secure-Software-3-Buffer-Overflow.ppt
- Secure Coding in C and C++: Strings and Buffer Overflows
  - http://www.informit.com/articles/article.aspx?p=2036582&seqNum=6
- Stack Smashing Protector
  - http://wiki.osdev.org/Stack_Smashing_Protector



Released under CC BY-SA 3.0

- This presentation is released under the Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) license
- You are free:
  - to Share — to copy, distribute and transmit the work
  - to Remix — to adapt the work
  - to make commercial use of 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 or similar license to this one
- These conditions can be waived by permission from the copyright holder
  - dwheeler at dwheeler dot com
- Details at: http://creativecommons.org/licenses/by-sa/3.0/
- Attribute me as "David A. Wheeler"