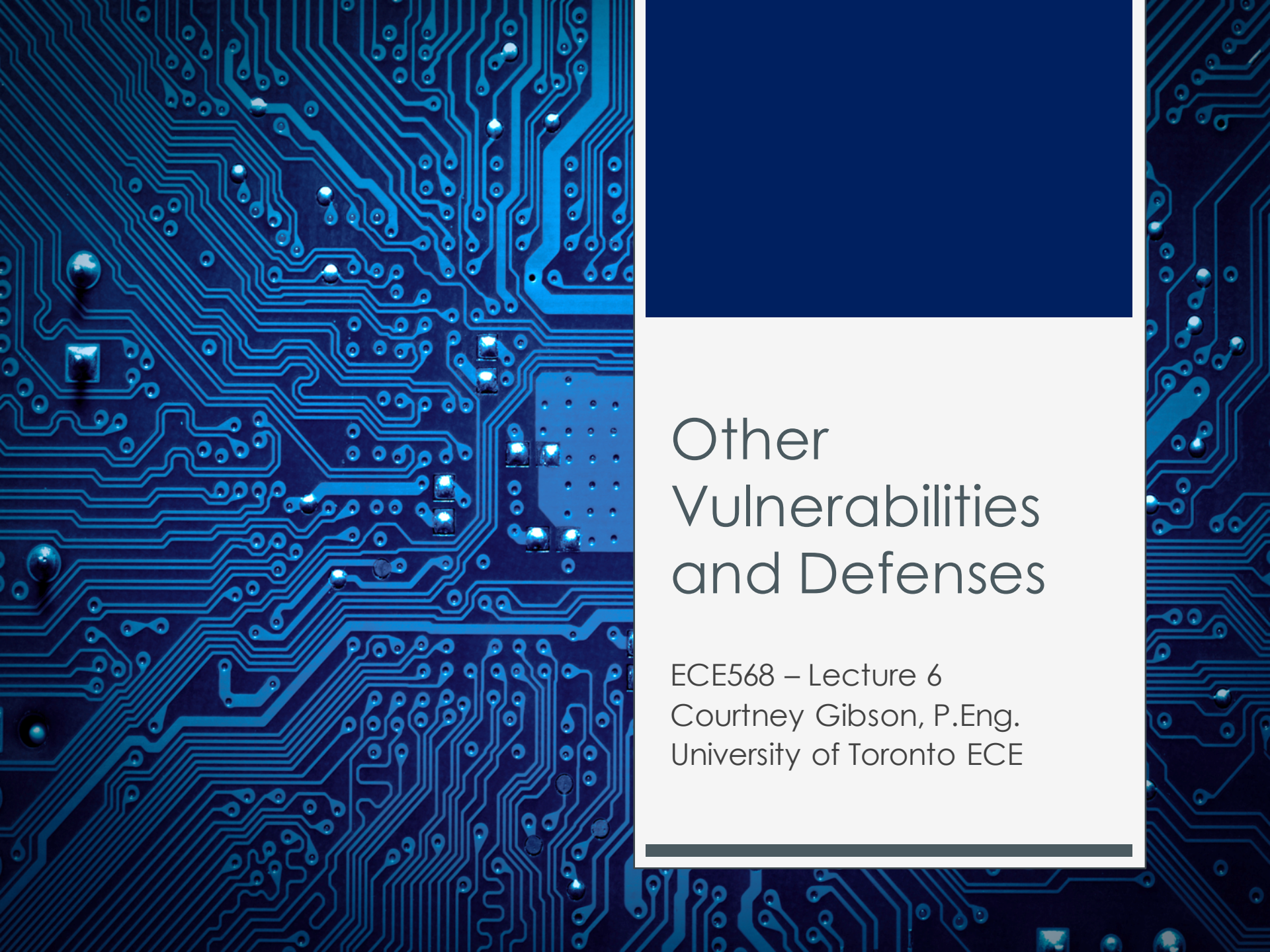


Other Vulnerabilities and Defenses

ECE568 – Lecture 6

Courtney Gibson, P.Eng.
University of Toronto ECE

Outline

Other Common Vulnerabilities

- Return-into-libc attacks
- Function pointer overwrite
- PLT/GOT overwrite
- Integer overflow
- Bad bounds check
- Argument overwrite

Defenses

- Stackguard stack smashing defense
- Address space layout randomization
- Non-executable pages

Other Common Vulnerabilities

Function pointers, dynamic linking, integer overflows, bad bounds checking, argument overwrite

Attacks without Code Injection

Until now, the attacks have involved overwriting the return address to point to injected code

- Is it possible to exploit a program without injecting code?
- What does it mean to exploit a program without injecting code?

An exploit can occur if an attacker can cause unintended program execution or unintended data modification

Return into **libc**

- One exploit method that doesn't require code injection is to use code already present
- Many **libc** functions have code useful to the attacker
  - e.g., the **`system`** library call looks a lot like shell code:

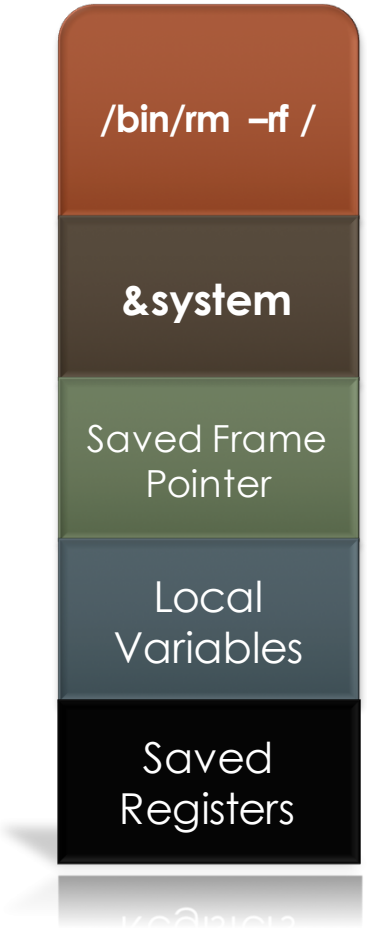
```
int system(const char *string);


// system() executes a command specified in string
// by calling /bin/sh -c string
```

Return into **libc**

Rather than inject shellcode, attacker:

- Changes the return address to point to start of the **`system`** function
- Injects a stack frame on the stack
- Just before return, **sp** points to &system (the orig return address)
- On return, **`system`** executes and expects its arguments at the top of the stack
- Argument contains the string the attacker wants to execute



Attacks without Overwriting the Return Address

Until now, attacks have overwritten return address. Are there other exploit possibilities?

- Function pointer overwrite
- Global offset table overwrite

Function Pointers

A function pointer is a variable that can be dereferenced to call a function:

```
int foo(int arg1) {
  ...
}

int (*fp)(int arg1);   /* define a function pointer */
fp = &foo;     /* assign addr of foo to the pointer */
fp(6);                  /* call foo via the pointer */
```

An adversary can try to overwrite a function pointer

Function Pointer Overwrite

- Common in object-oriented languages (*e.g.*, C++)
- Function pointers are often used in C also
  - Allows mimicing polymorphic features of OO languages (*e.g.*, qsort)
  - Used to support dynamically loaded libraries
  - Very common in OS kernels, where the kernel has to run with different modules or drivers without recompilation
  - Also common in other programs that use modules such as web servers, etc.
- Sometimes a buffer will be beside a function pointer rather a return address
  - By overwriting the function pointer, the attacker can cause execution to be redirected when the program calls the function pointer

Dynamic Linking

Program code needs to call functions such as **`printf`** in dynamic libraries

- These libraries are normally linked into the program at run time, at arbitrary locations, by a dynamic linker
- Typically, both the caller of a library function and the function itself are compiled to be **position independent**
- We need to map the position independent function call to the absolute location of the function's code in the library
  - The dynamic linker performs this mapping
  - It uses two tables: the **Procedure Linkage Table** (PLT) and the **Global Offset Table** (GOT)

PLT/GOT

GOT is a table of pointers to functions:

- Contains the absolute memory location of each of the dynamically-loaded library functions

PLT is a table of code entries:

- One per each library function called by program
  - For example, sprintf@plt
- Somewhat similar to a switch statement
- Each code entry invokes the corresponding function pointer in GOT
  - For example, sprintf@PLT code may invoke "jmp GOT[k]", where k corresponds to the sprintf function index in GOT

Dynamic Linking Mechanism

All calls to dynamic libraries jump to PLT:

- The first time the function is called, the runtime linker is invoked to load the library
- The runtime linker updates the GOT entry, based on where the library is loaded
- Further calls invoke the function in the loaded library via the updated GOT entry
- The PLT/GOT contain commonly used library functions such as **`printf`**, **`fopen`**, **`fclose`**, etc.

PLT/GOT Overwrites

Suppose that an attacker is only able to overwrite a single chosen address location with a chosen value

- Then a good option is to overwrite a GOT function pointer

A binary utility like **`objdump -x`** allows dissassembling an executable

- It provides the location of these structures
- PLT/GOT always appear at a **known** location

Integer Overflows

A server processes packets of variable size:

- The first 2 bytes of the packet store the size of the packet to be processed
- Only packets of size <= 512 bytes should be processed
- What's wrong with the code?
- **Hint:** the third arg of memcpy is unsigned int

```
char* processNext(char* strm)
{
  char buf[512];
  short len = *(short*) strm;
  strm += sizeof(len);
  if (len <= 512) {
    memcpy(buf, strm, len);
    process(buf);
    return strm + len;
  } else {
    return -1;
  }
}
```

Bad Bounds Check

What's wrong with this bounds check?

```
/* Linux 2.4.5/drivers/char/drm/i810_dma.c */
/* [copy arg from user space into d] */
if(copy_from_user(&d, arg, sizeof(arg)))
  return -EFAULT;
if(d.idx > dma->buf_count)
  return -EINVAL;
buf = dma->buflist[d.idx];
copy_from_user(buf_priv->virtual, d.address, d.used);
```

Allows reading arbitrary memory locations

- Similar vulnerabilities have led to remote code execution in the past
  - e.g., the do_brk() function in the Linux 2.4.22 kernel

Argument Overwrite

Instead of changing the execution of a program (i.e. control flow), an attacker can cause unintended data modification

- For example, an attacker can hijack a program by overwriting the argument of a sensitive function such as **exec**

```
char buf[128] = “my_program” ;
char vulnerable[32];
        …
exec(buf);
```

The attacker can corrupt the argument **buf** by overflowing **vulnerable** and have the program execute something else

- Note that the program execution has not changed!

Defenses

Buffer Overflow Defenses

Many of the attacks discussed have depended on overflowing buffers. The most obvious way to defend against buffer overflow vulnerabilities is not to make them:

- Audit code rigourously
- Use a type-safe language with bounds checking
  - *e.g.*, Java, C#
  - Code will be **memory safe**: compiler will enforce the memory access rules of the language

However, this is not always possible:

- Too much legacy code
- Source code is not available
- Performance may be a concern
- Easy to write C code without correct checks

Other Options for Defense?

Buffer overflow attack requires an input string to be copied into a buffer without bounds checking

- Typical attack requires three steps
  - Control over a location such as return address
  - Overwrite location with guessed address
  - Inject and execute shell code

What is needed for these steps to succeed?

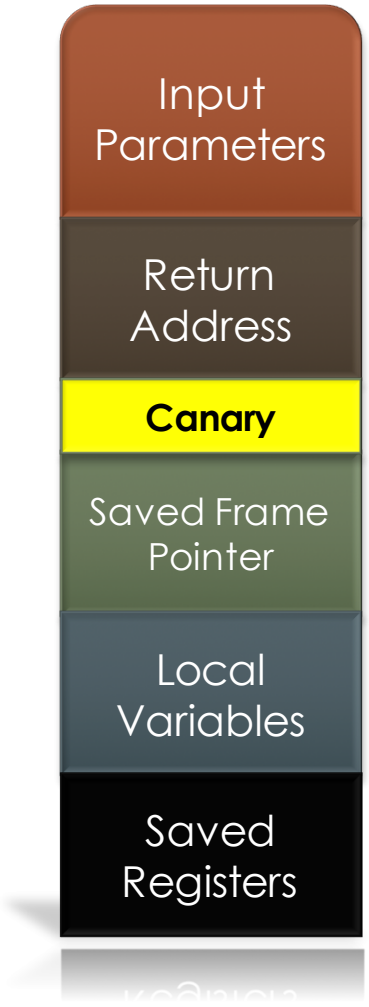
- Return address overwrite
- Target address has to be guessed
- Injected code has to be executable

Let's look at how to detect or prevent each of these steps…

Defending Against Stack Smashing

Recent protection techniques will prevent the return address from being overwritten

- **Stackshield**
  - Put return addresses on a separate stack with no other data buffers there
- **Stackguard**
  - On a function call, a random **canary** value is placed just before the return address
  - Just before the function returns, the code checks the canary value and, if the value has changed, the program is halted
  - MS VC++ compiler supports it with the *GS flag*
  - Recent GCC compilers support it
  - Does the canary stop format string attacks?



Defending Against Stack Smashing

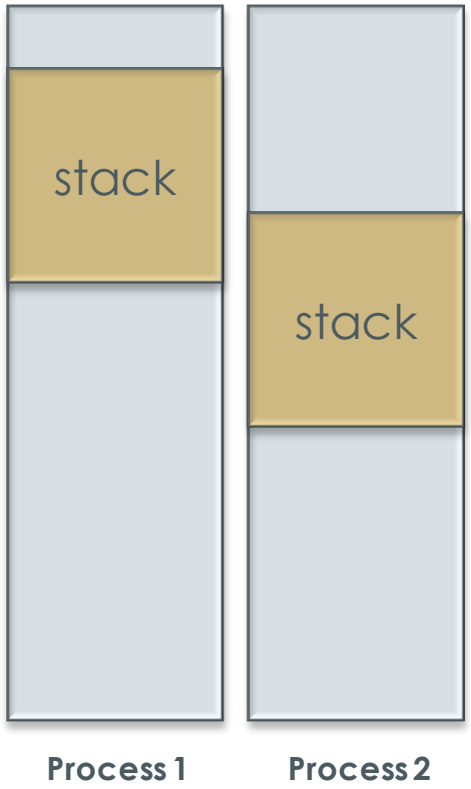
Run Time Checking: Libsafe (Avaya Labs)

- Dynamically loaded library
  - Overrids **libc.so** (usually via /etc/ld.so.preload)
  - Done at runtime, so doesn't need program recompilation or source-code changes
- Intercepts calls to dangerous functions such as **`strcpy`**
  - Validates sufficient space in current stack frame

Address-Space Layout Randomization (ASLR)

Recall that the target address (*e.g.*, the buffer's location on the stack) has to be guessed:

- With ASLR, the OS maps the stack of each process at a **randomly** selected location with each invocation
  - An attacker will not be able to easily guess the target address
  - Application will crash rather than executing the attacker's code
  - ASLR also randomizes location of dynamically loaded libraries, making it harder to perform return-into-libc attacks or GOT overwrites
- Linux 2.6 and Windows Vista use ASLR



Non-Executable Pages (NX)

- If stack is made non-executable, then shellcode on the stack will not execute
  - Recent Intel, AMD processors allow non-executable pages
    - Page tables have NX protection bit
  - Requires support from OS
    - NX implemented in Windows XP SP2 patch
- However, non-injection attacks are still possible
  - E.g., return-into-libc attacks, argument overwrite attacks

Vulnerability Databases

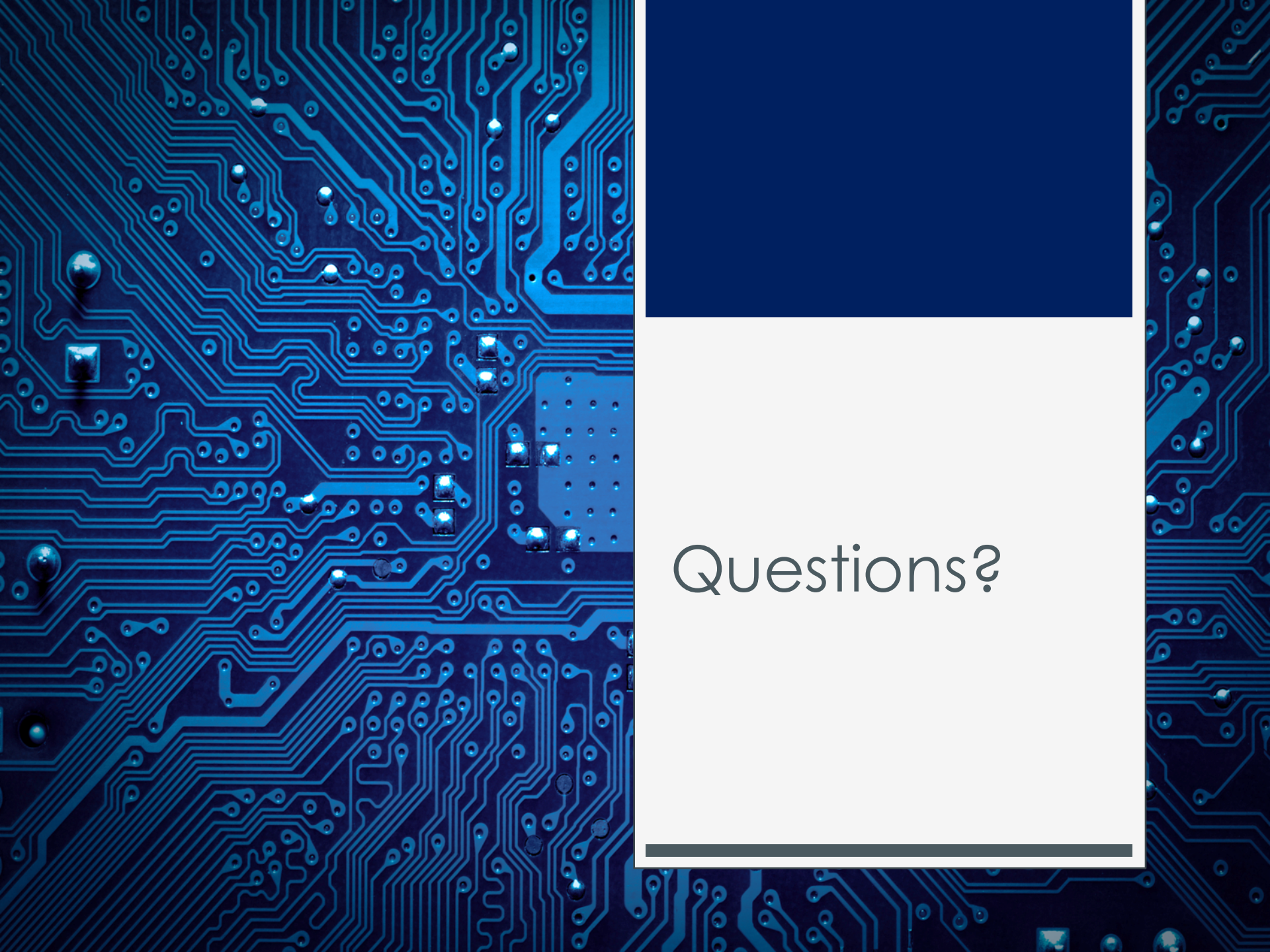
To aid computer administrators, there are several large databases of vulnerabilities on the Internet:

- **National Vulnerability Database:** http://nvd.nist.gov
- **CERT:** http://www.cert.org
- **SecurityFocus:** http://www.securityfocus.com/vulnerabilities
- **Bugtraq:** http://www.securityfocus.com/archive/
- **OSVDB:** http://www.osvdb.org

For any program and version, one can query these databases and get a description of the vulnerability

Conclusion

- Easy to make a mistake, end up with a vulnerability
  - Exploiting them takes a bit of work, but is not beyond someone who knows what they are doing
- Certain vulnerabilities can be removed by moving to safer languages
  - A lot of vulnerabilities result from uses of pointers and running off the end of arrays
  - Java doesn't allow the use of pointers, does array bounds checking automatically and has a stronger type system
- However, the only real defense is to be aware of what vulnerabilities exist, to be extra careful when creating code and let others audit your code


Questions?