CIS 551 / TCOM 401

Computer and Network Security

Spring 2008

Lecture 2

Announcments

- First project: Due: 8 Feb. 2007 at 11:59 p.m.
- http://www.cis.upenn.edu/~cis551/project1.html
- Group project:
  - 2 or 3 students per group
  - Send e-mail to TA with your group by Jan. 25th

- Plan for Today / Thursday:
  - Designing secure systems
  - Buffer overflows in detail

Building Secure Software

- Source: book by John Viega and Gary McGraw
  - Copy on reserve in the library
  - Strongly recommend buying it if you care about implementing secure software.
- Designing software with security in mind

- What are the security goals and requirements?
  - Risk Assessment
  - Tradeoffs
- Why is designing secure software a hard problem?
- Design principles
- Implementation
- Testing and auditing

Security Goals

- Prevent common vulnerabilities from occurring (e.g. buffer overflows)
- Recover from attacks
  - Traceability and auditing of security-relevant actions
- Monitoring
  - Detect attacks
- Privacy, confidentiality, anonymity
  - Protect secrets
- Authenticity
  - Needed for access control, authorization, etc.
- Integrity
  - Prevent unwanted modification or tampering
- Availability and reliability
  - Reduce risk of DoS

Other Software Project Goals

- Functionality
- Usability
- Efficiency
- Time-to-market
- Simplicity

- Often these conflict with security goals
  - Examples?

- So, an important part of software development is risk assessment/risk management to help determine the design choices made in light of these tradeoffs.

Risk Assessment

- Identify:
  - What needs to be protected?
  - From whom?
  - For how long?
  - How much is the protection worth?
- Refine specifications:
  - More detailed the better (e.g. "Use crypto where appropriate." vs. "Credit card numbers should be encrypted when sent over the network.")
  - How urgent are the risks?
- Follow good software engineering principles, but take into account malicious behavior.

Principles of Secure Software

- What guidelines are there for developing secure software?
- How would you go about building secure software? Class answers:

#1: Secure the Weakest Link

- Attackers go after the easiest part of the system to attack.
  - So improving that part will improve security most.
- How do you identify it?
- Weakest link may not be a software problem.
  - Social engineering
  - Physical security
- When do you stop?

#2: Practice Defense in Depth

- Layers of security are harder to break than a single defense.
- Example: Use firewalls, and virus scanners, and encrypt traffic even if it's behind firewall

#3: Fail Securely

- Complex systems fail.
- Plan for it:
  - Aside: For a great example, see the work of George Candea who's Ph.D. research is about something called "microreboots"
- Sometimes better to crash or abort once a problem is found.
  - Letting a system continue to run after a problem could lead to worse problems.
  - But sometimes this is not an option.
- Good software design should handle failures gracefully
  - For example, handle exceptions

#4: Principle of Least Privilege

- Recall the Saltzer and Schroeder article
- Don't give a part of the system more privileges than it needs to do its job.
  - Classic example is giving root privileges to a program that doesn't need them: mail servers that don't relinquish root privileges once they're up and running on port 25.
  - Another example: Lazy Java programmer that makes all fields public to avoid writing accessor methods.
- Military's slogan: "Need to know"

#5: Compartmentalize

- As in software engineering, modularity is useful to isolate problems and mitigate failures of components.
- Good for security in general: Separation of Duties
  - Means that multiple components have to fail or collude in order for a problem to arise.
  - For example: In a bank the person who audits the accounts can't issue cashier's checks (otherwise they could cook the books).
- Good examples of compartmentalization for secure software are hard to find.
  - Negative examples?

#6: Keep it Simple

- KISS: Keep it Simple, Stupid!
- Einstein: "Make things as simple as possible, but no simpler."
- Complexity leads to bugs and bugs lead to vulnerabilities.
- Failsafe defaults: The default configuration should be secure.
- Ed Felten quote: "Given the choice between dancing pigs and security, users will pick dancing pigs every time."

#7: Promote Privacy

- Don't reveal more information than necessary
  - Related to least privileges
- Protect personal information
  - Consider implementing a web pages that accepts credit card information.
  - How should the cards be stored?
  - What tradeoffs are there w.r.t. usability?
  - What kind of authentication/access controls are there?

#8: Hiding Secrets is Hard

- The larger the secret, the harder it is to keep
  - That's why placing trust in a cryptographic key is desirable
- Security through obscurity doesn't work
  - Compiling secrets into the binary is a bad idea
  - Code obfuscation doesn't work very well
  - Reverse engineering is not that difficult
  - Software antipirating measures don't work
  - Even software on a "secure" server isn't safe (e.g. source code to Quake was stolen from id software)

#9: Be reluctant to trust

- *Trusted Computing Base*: The set of components that must function correctly in order for the system to be secure.
- The smaller the TCB, the better.
- Trust is transitive
- Be skeptical of code quality
  - Especially when obtained from elsewhere
  - Even when you write it yourself

#10: Use Community Resources

- Software developers are not cryptographers
  - Don't implement your own crypto
  - (e.g. bugs in Netscape's storage of user data)

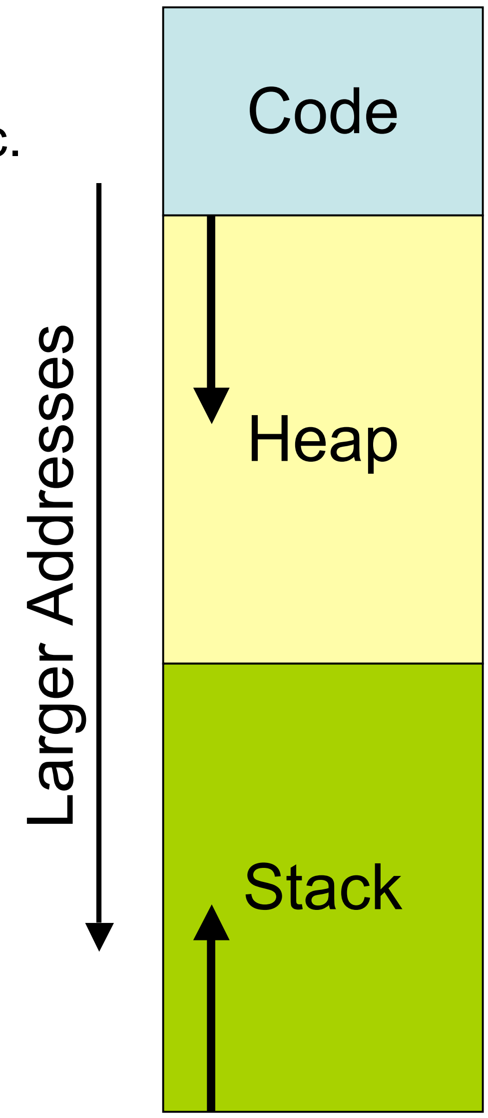
- Make use of CERT, Bugtraq, developer information, etc.

Buffer Overflow Attacks

- > 50% of security incidents reported at CERT are related to buffer overflow attacks
- Problem is access control but at a very fine level of granularity
- C and C++ programming languages don’t do array bounds checks

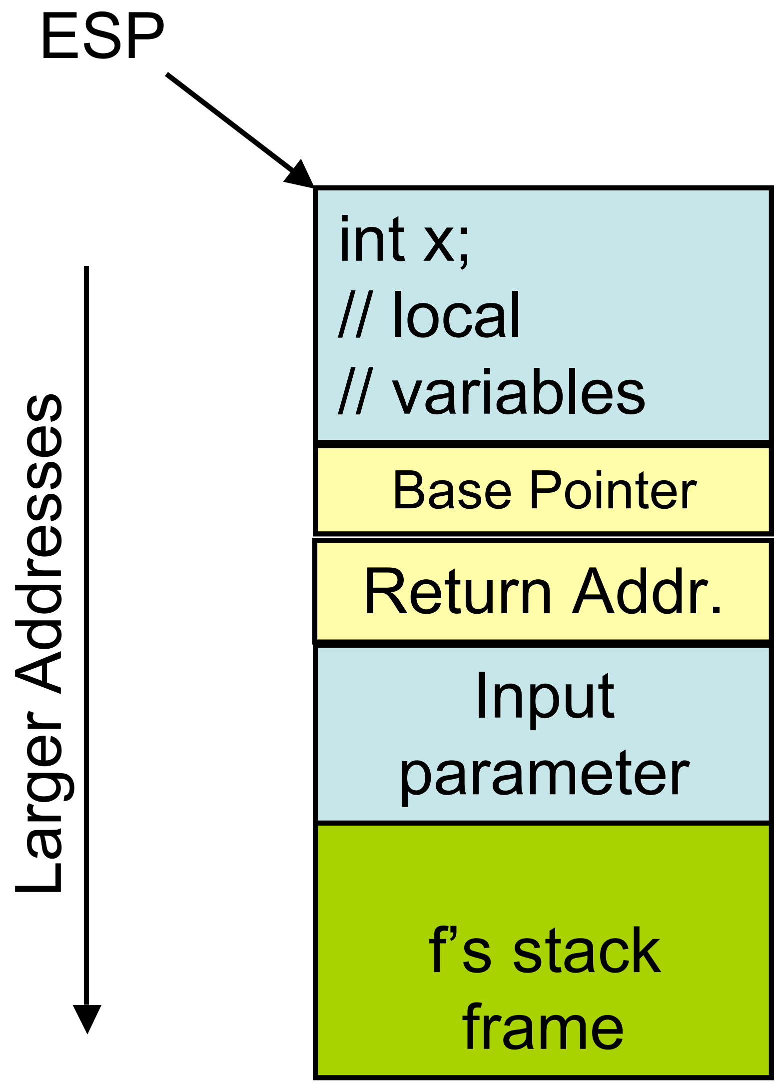
3 parts of C memory model

- The code & data (or "text") segment
  - contains compiled code, constant strings, etc.
- The Heap
  - Stores dynamically allocated objects
  - Allocated via "malloc"
  - Deallocated via "free"
  - C runtime system
- The Stack
  - Stores local variables
  - Stores the return address of a function



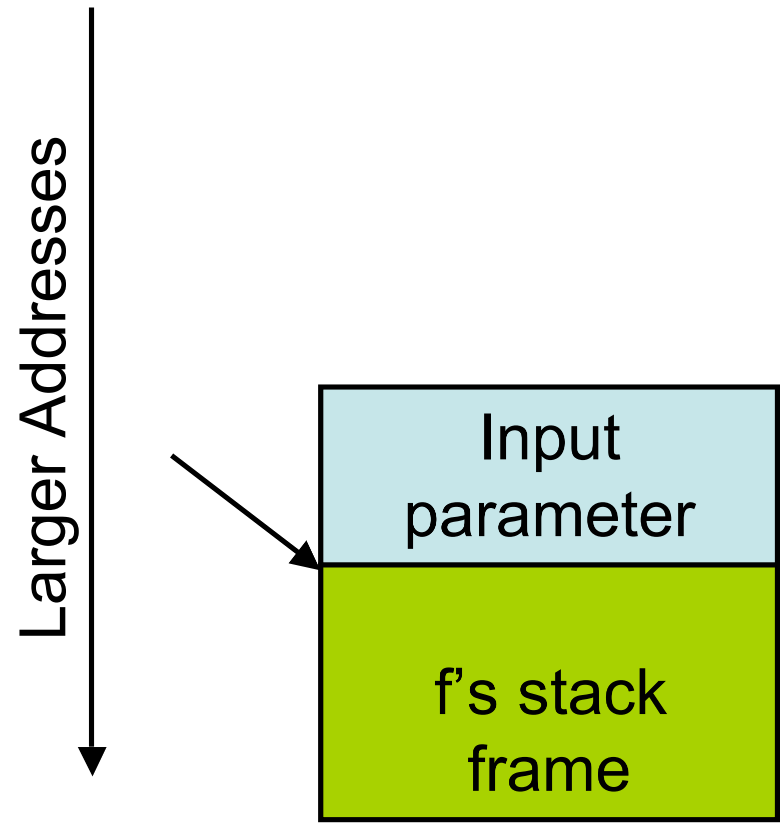
C's Control Stack

```
f() {
  g(parameter);
}

g(char *args) {
  int x;
  // more local
  // variables
  ...
}
```



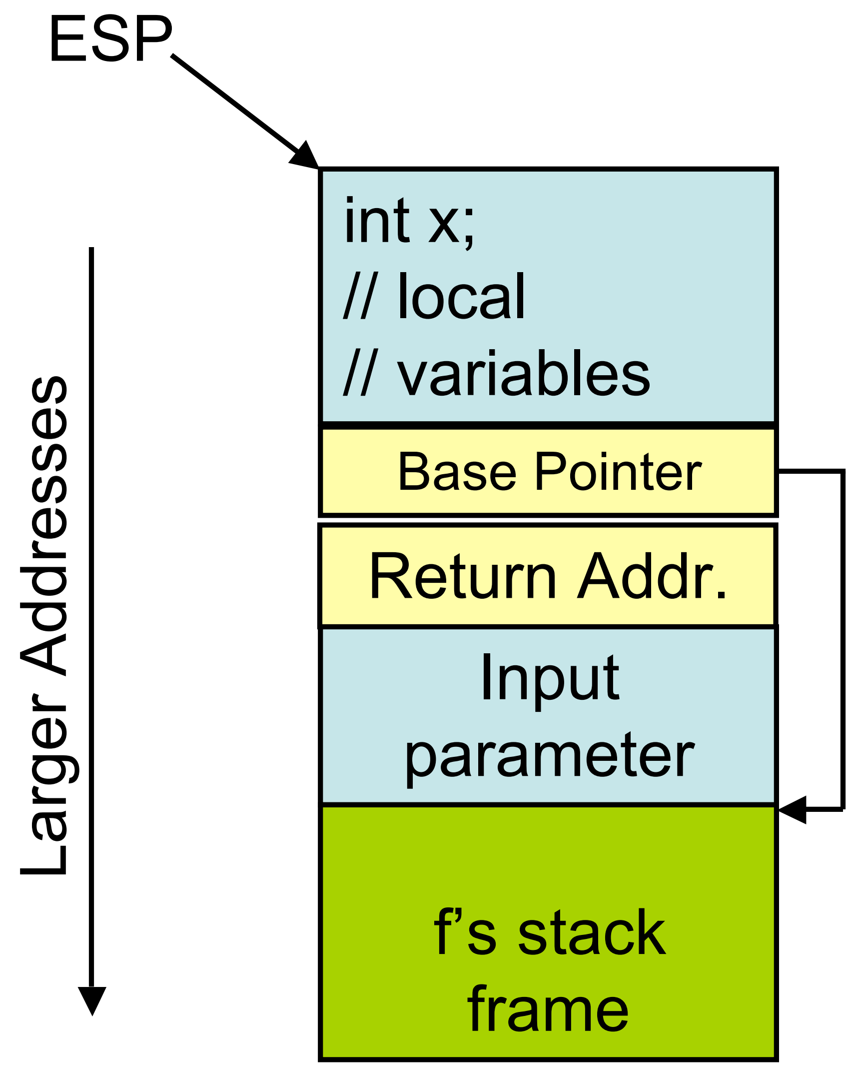
C's Control Stack

```
f() {
  g(parameter);
}

g(char *args) {
  int x;
  // more local
  // variables
  ...
}
```



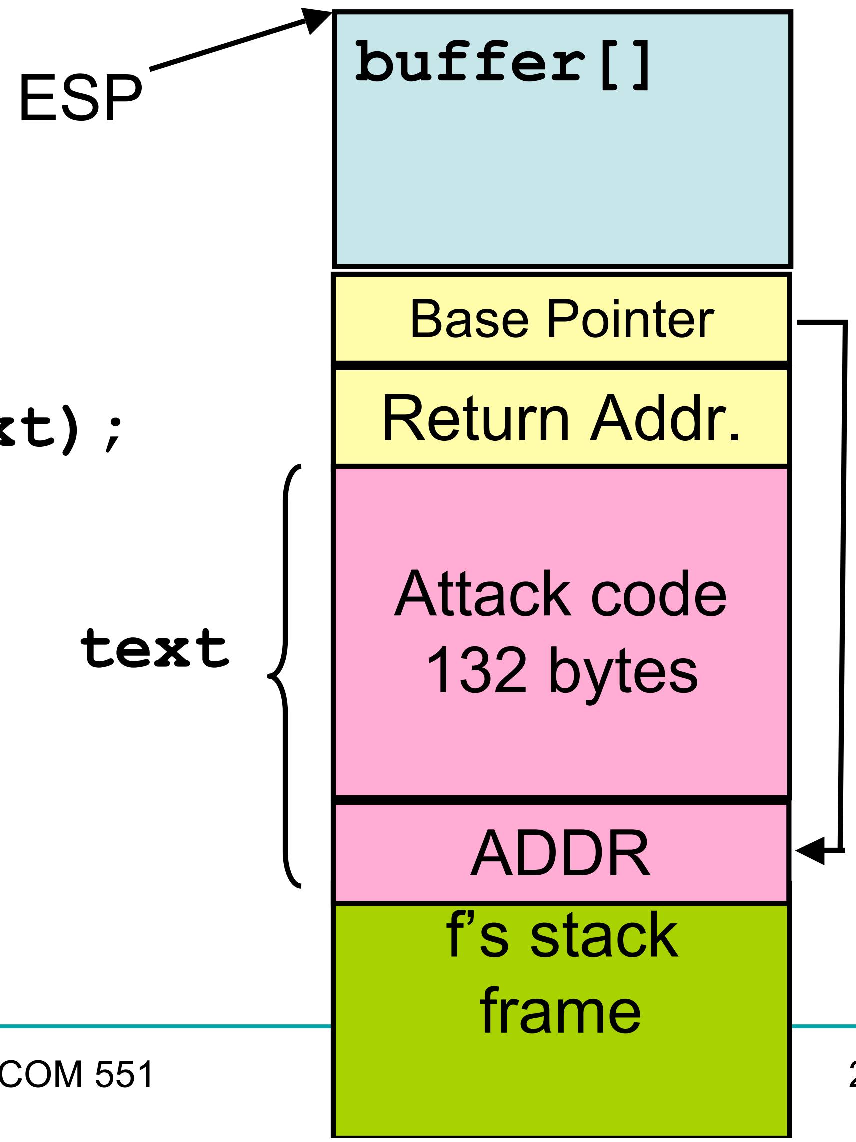
C’s Control Stack

```
f() {
  g(parameter);
}

g(char *args) {
  int x;
  // more local
  // variables
  ...
}
```



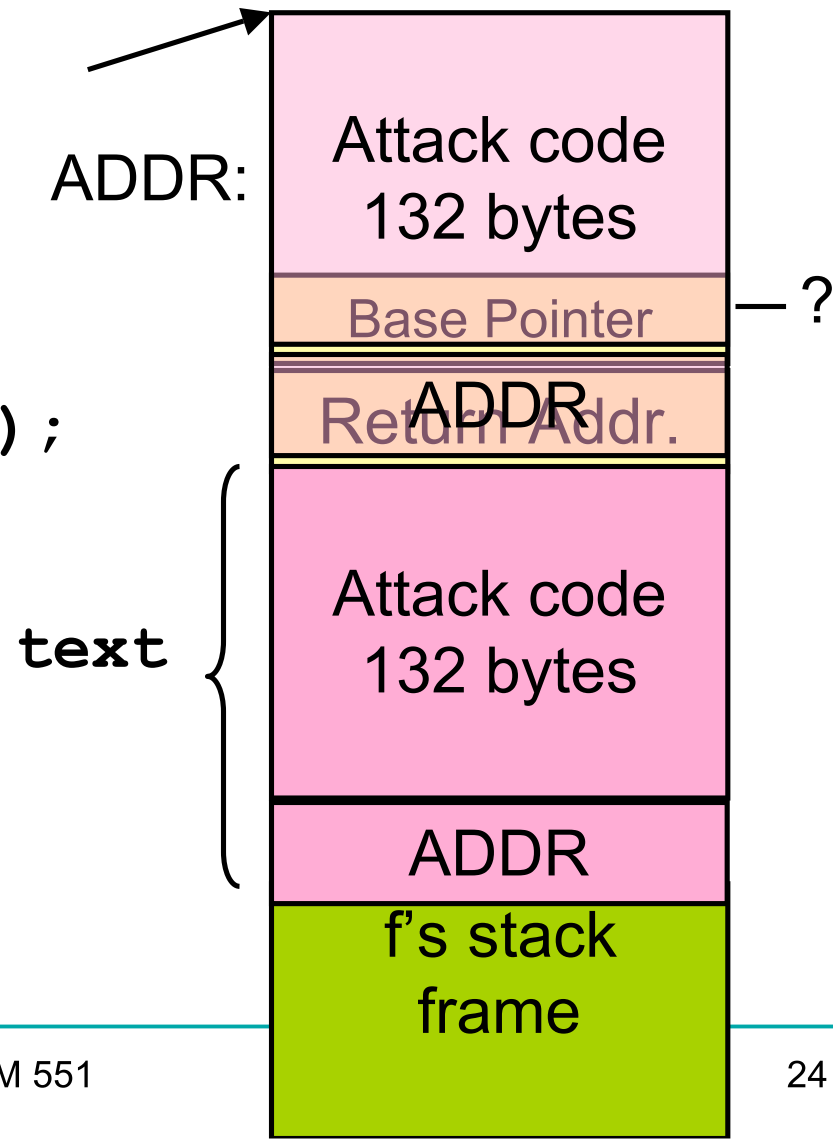
Buffer Overflow Example

```
g(char *text) {
  char buffer[128];
  strcpy(buffer, text);
}
```



Buffer Overflow Example

```
g(char *text) {
  char buffer[128];
  strcpy(buffer, text);
}
```



Details: C calling conventions

```
int function(int a, int b, int c) {
  char buffer1[4];
  int ans = a + b + c;
  char buffer2[10];
  return ans;
}

int main() {
  return function(1,2,3);
}
```

Resulting Assembly (1)

```
    .file       "example.c"
    .text
.globl function
    .type       function, @function
function:
    pushl       %ebp                          // Set up stack frame
    movl %esp,  %ebp
    subl $32,   %esp                          // Allocate local storage
    movl 12(%ebp),%eax
    addl 8(%ebp), %eax                        // ans = a + b + c
    addl 16(%ebp),%eax
    movl %eax, -4(%ebp)
    movl -4(%ebp), %eax                       // %eax holds the return value
    leave                                     // Tear down stack frame
    ret                                       // Pop return address & jump to it
    .size       function, .-function
```

Resulting Assembly (2)

```
.globl main
   .type       main, @function
main:
   leal 4(%esp), %ecx
   andl $-16, %esp          // Align the stack on 16-byte boundary
   pushl -4(%ecx)
   pushl %ebp
   movl %esp, %ebp          // Set up stack frame
   pushl %ecx               // Save caller-save register
   subl $12, %esp
   movl $3, 8(%esp)         // Push arguments onto the stack
   movl $2, 4(%esp)
   movl $1, (%esp)
   call function            // Push return address, jump to function:
   addl $12, %esp           // Pop arguments off the stack
   popl %ecx                // Restore caller-save register
   popl %ebp                // Tear down stack frame
   leal -4(%ecx), %esp      // Undo stack alignment
   ret
```

Project hints

- Use plus.seas.upenn.edu
  - minus.seas.upenn.edu still has stack protection turned on
  - 'uname -a' will give you some useful information about which machine you're connected to
- GCC has changed significantly since the Aleph One tutorial was written:
  - 16 bit vs. 32 bit architecture
  - GCC uses arithmetic with %esp and movl instructions instead of pushl when pushing arguments onto the stack
  - GCC now automatically allocates 8 bytes of "free" space in each stack frame.
  - Syntax of inline assembly is different

Constructing a Payload

- Idea: Overwrite the return address on the stack
  - Value overwritten is an address of some code in the "payload"
  - The processor will jump to the instruction at that location
  - It may be hard to figure out precisely the location in memory

- You can increase the size of the "target" area by padding the code with no-op instructions
- You can increase the chance over overwriting the return address by putting many copies of the target address on the stack

[NOP]…[NOP]{attack code} {attack data}[ADDR]…[ADDR]

More About Payloads

- How do you construct the attack code to put in the payload?
  - You use a compiler!
  - Gcc + gdb + options to spit out assembly (hex encoded)
- What about the padding?
  - NOP on the x86 has the machine code 0x90
- How do you guess the ADDR to put in the payload?
  - Some guesswork here
  - Figure out where the first stack frame lives: OS & hardware platform dependent, but easy to figure out
  - Look at the program -- try to guess the stack depth at the point of the buffer overflow vulnerability.
  - Intel is little endian -- so if ADDR is:
    0xbf9ae358 you actually need to put the following words in the payload: 0x58 0xe3 0x9a 0xbf

Finding Buffer Overflows

- The #1 source of vulnerabilities in software
- Caused because C and C++ are not safe languages
  - They use a “null” terminated string representation:

    `“HELLO!\0”`

  - Standard library routines *assume* that strings will have the null character at the end.
  - Bad defaults: the library routines don’t check inputs
- Easy to accidentally get wrong
- …even easier to maliciously attack

Buffer overflows in library code

- Basic problem is that the library routines look like this:

```
void strcopy(char *src, char *dst) {
  int i = 0;
  while (src[i] != "\0") {
    dst[i] = src[i];
    i = i + 1;
  }
}
```

- If the memory allocated to `dst` is smaller than the memory needed to store the contents of `src`, a buffer overflow occurs.

If you must use C/C++

- Avoid the (long list of) broken library routines:
  - strcpy, strcat, sprintf, scanf, sscanf, *gets*, read, …
- Use (but be careful with) the "safer" versions:
  - e.g. strncpy, snprintf, fgets, …
- *Always* do bounds checks
  - One thing to look for when reviewing/auditing code
- Be careful to manage memory properly
  - Dangling pointers often crash program
  - Deallocate storage (otherwise program will have a memory leak)

- Be aware that doing all of this is difficult.

Tool support for C/C++

- Extensions to gcc that do array bounds checking
- Link against "safe" versions of libc (e.g. libsafe)
- Test programs with tools such as Purify or Splint
- Compile programs using tools such as:
  - Stackguard and Pointguard (Cowan et al., immunix.org)
  - gcc's -fstack-guard and -mudflap options
- Research compilers:
  - Ccured (Necula et al.)
  - Cyclone (Morrisett et al.)
- Binary rewriting techniques
  - Software fault isolation (Wahbe et al.)

Defeating Buffer Overflows

- Use a typesafe programming language
  - Java/C# are not vulnerable to these attacks
- Some operating systems move the start of the stack on a per-process basis:
  - E.g. eniac-l