

# Code Security

CSE497b - Spring 2007

Introduction Computer and Network Security

Professor Jaeger

[www.cse.psu.edu/~tjaeger/cse497b-s07/](http://www.cse.psu.edu/~tjaeger/cse497b-s07/)

# Coding

- Why do we write programs?
  - Function
- What functions do we enable via our programs?
  - Can some enable vulnerabilities?
  - What are the types of code vulnerabilities?

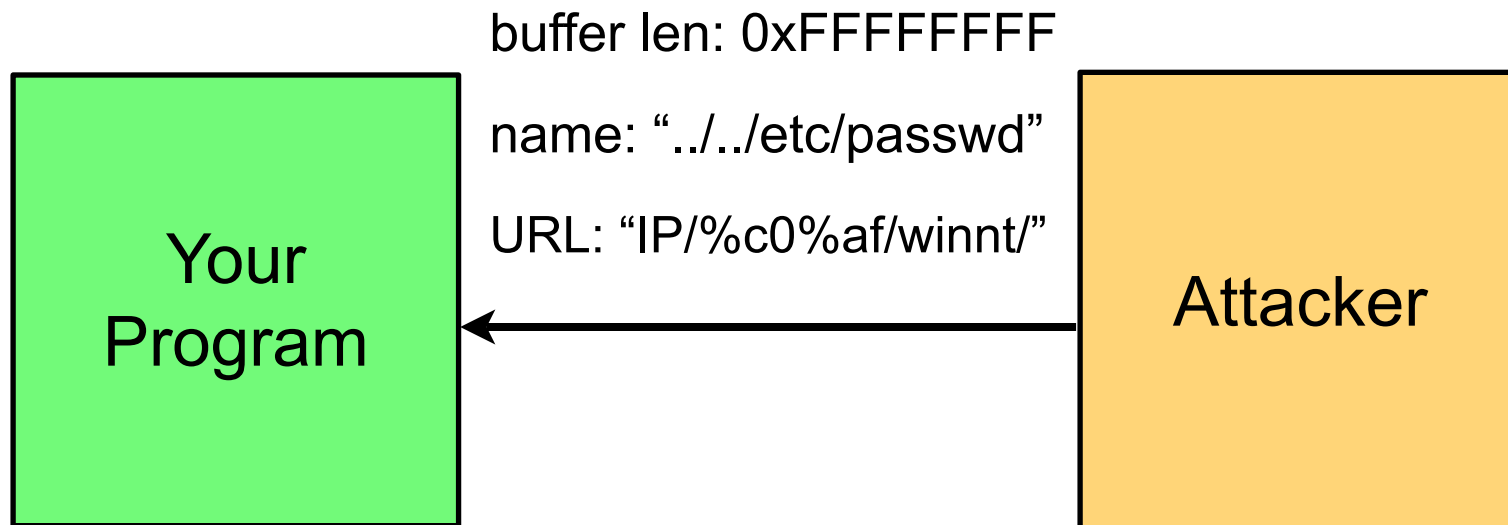


# Is Your Code Secure?

- How do *you* prevent your code from having vulnerabilities?
  - How do you know?

# Types of Problems

- Overflows
- Format Parsing
- Integer Overflow
- Character Parsing



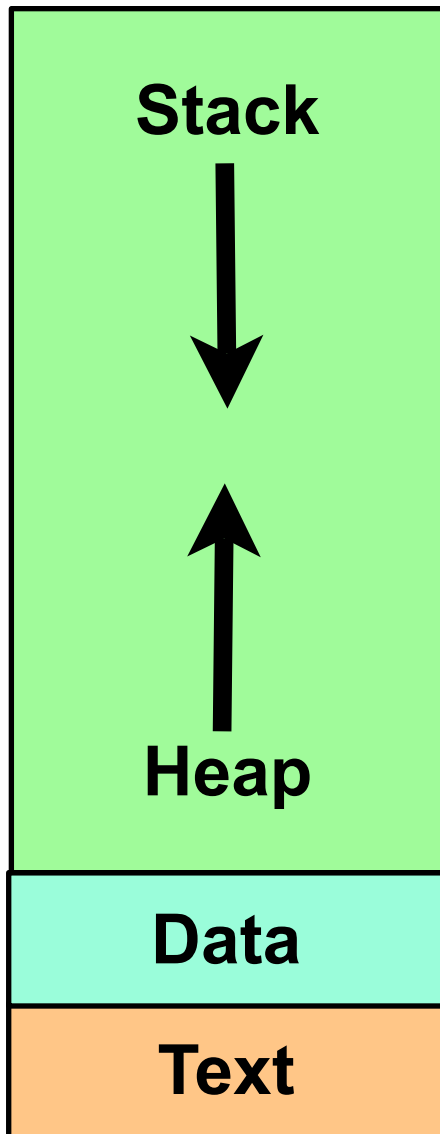
# Buffer Overflow

```
main() {
    int size = 1024, bytes;
    char buf[size];

    ...
    bytes = read(fd, user_data, BLOCK_SIZE);
    ...
    strcpy(buf, user_data);
}
```

- What is the problem at strcpy?
- How do we fix it?
- How do we know that all have been caught, properly?

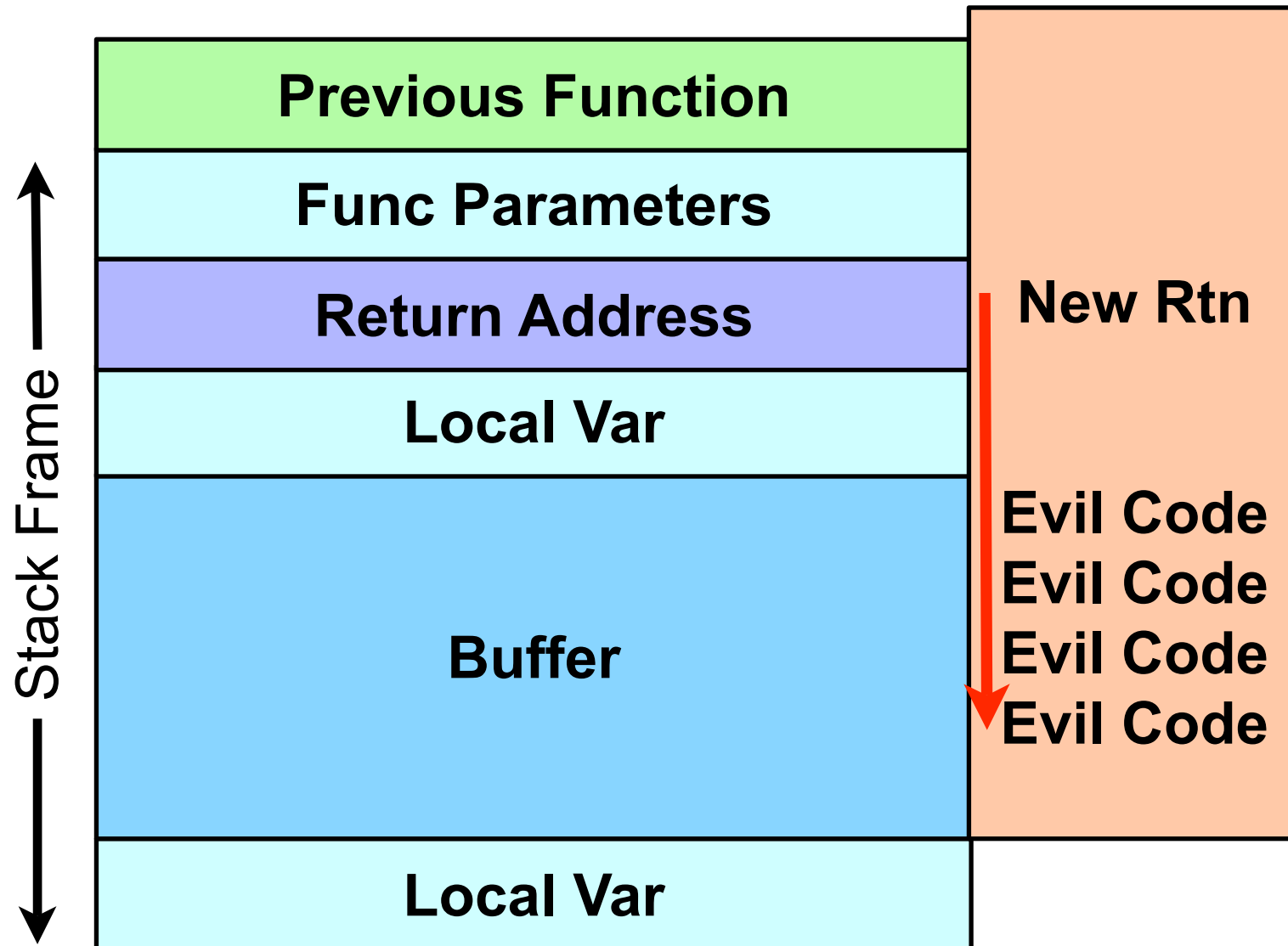
# Address Space Layout and Assumptions



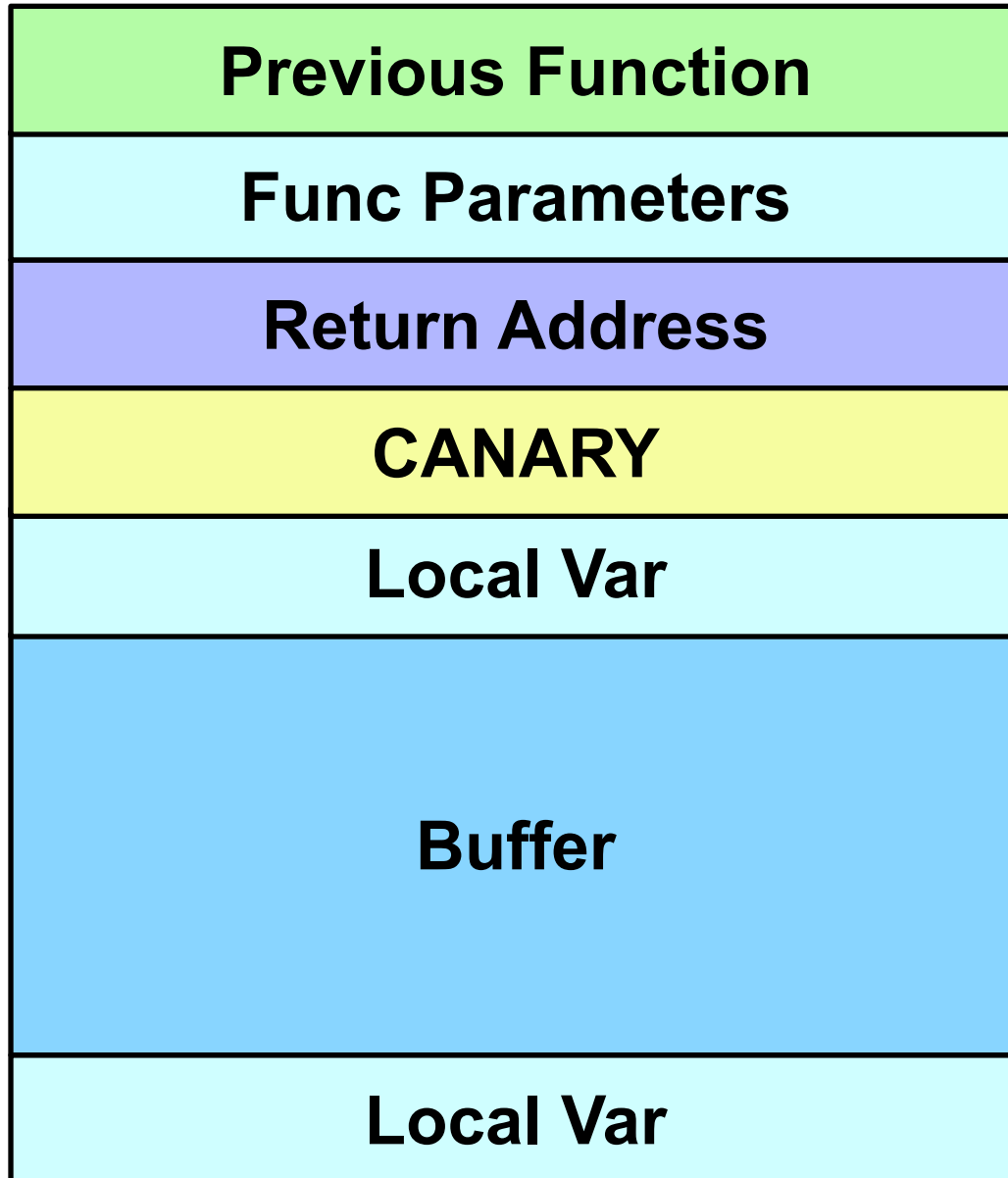
- Depends on the layout of a computer process's memory
  - Note that the stack grows downward
- Depends on lack of type safety in programming language
  - Can write outside a data structure using its reference
- Difference between logical model and implementation
  - Can't assume everything works according to the logical model

# Buffer Overflow

- How it works



# Defenses



- “Canary” on the stack
  - Random value placed between the local vars and the return address
  - If canary is modified, program is stopped
  - Will this address the “basic” buffer overflow?
- Alternative:
  - Non-executable stack
- Are we done?

# General Overflows

- Don't execute on stack
  - Return-to-libc
    - Overflow return pointer to library function
- Not detectable by canary
  - Overwrite function pointers
  - Overwrite data pointers
  - Extract unauthorized data
- Heap overflow
  - Overwrite other data or function pointers on the heap
- *Result*: A comprehensive solution has been hard to find
  - Address space randomization
  - Type safety

# Printf Vulnerability

- Overwrite memory using printf
- Because previously thought harmless
  - People adopted the shortcut
  - `printf(“%s”, buf) ---> printf(buf)`
- Found an exploit in 1999
  - Found hundreds of live programs to which this exploit applied
- Craft a user input containing %n in the string
  - %n directive writes the number of bytes printed so far
  - back to the corresponding argument
  - assumes that the argument is an integer pointer
- *Can overwrite a word nearly anywhere in the process's address space!*

# Integers

- Integers in theory
  - -infinity to infinity
- In computing
  - Finite representation
    - Different sizes
  - Signed and unsigned
- For signed 8-bit integers
  - $127 + 1 = ??$
- For a signed integer *size* and unsigned *sizeof(buf)*
  - $size < sizeof(buf)$ 
    - if size can be negative
  - `array[size]`
    - negative value can enable arbitrary memory read

# Integer Overflow?

```
char buf[128];  
combine(char *s1, int len1, char *s2, int len2)  
{  
    if (len1 + len2 + 1 <= sizeof(buf)) {  
        strncpy(buf, s1, len1);  
        strncat(buf, s2, len2);  
    }  
}
```

# Integer Overflow to Buffer Overflow

```
char buf[128];
combine(char *s1, int len1, char *s2, int len2)
{
    if (len1 + len2 + 1 <= sizeof(buf)) {
        strncpy(buf, s1, len1);
        strncat(buf, s2, len2);
    }
}
```

- **Suppose**
  - $len1 < sizeof(buf)$
  - $len2 = 0xFFFFFFFF$
  - **strncat expects unsigned len**

# Character Strings

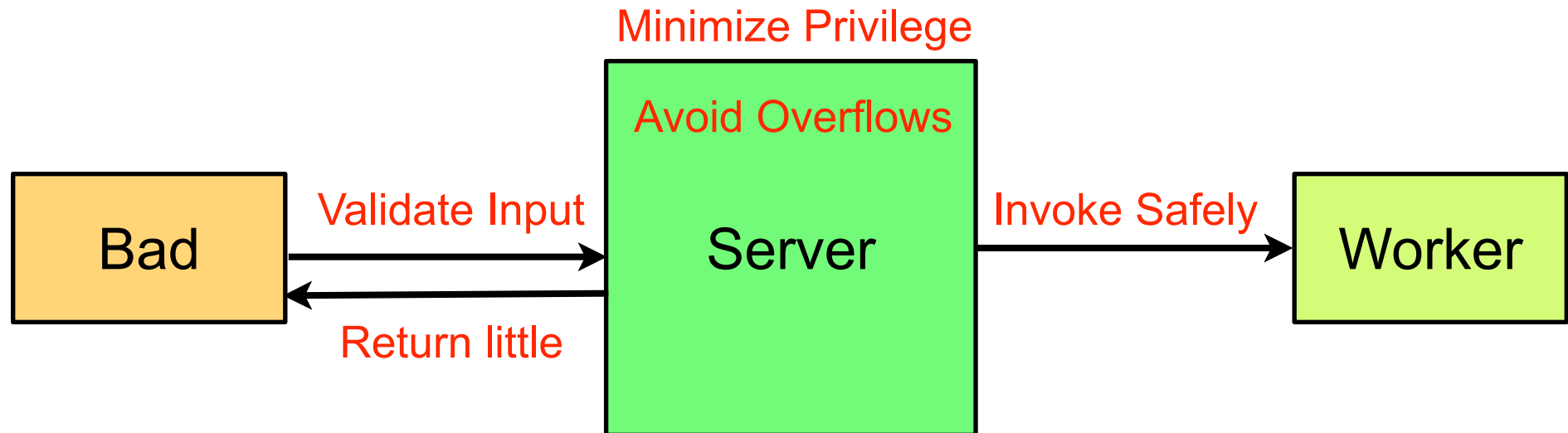
- String formats
  - Unicode
    - ASCII -- 0x00 -- 0x7F
    - non-ASCII -- 0x80 -- 0xF7
    - Also, multi-byte formats
  - Decoding is a challenge
    - URL: [IPaddr]/scripts/..%c0%af../winnt/system32
    - Decodes to /winnt/system32
  - Markus Kuhn's page on Unicode resources for Linux
    - [www.cl.cam.ac.uk/~mgk25/unicode.html](http://www.cl.cam.ac.uk/~mgk25/unicode.html)
- Buffer handling
  - Find problems in our code
  - Has data been validated?

# Names

- Problem
  - Multiple ways of naming lots of things
    - Files
      - /x/data or /y/z/../../../../x/data or /y/z/%2e%2e/x/data or links
    - Lots of others DNS names, IP addresses, etc.
- Canonicalization
  - Conversion to a single, “standard” name
  - Name to object mapping does not change between check and use
- Rules of thumb
  - Do not rely on names -- or anything -- from remote user
  - Convert them -- correctly -- to canonical format
  - Canonicalize at the system level (inodes instead of filenames)

# Secure Programming

- David Wheeler's Secure Programming for Linux and UNIX
  - Validate all input; Only execute application-defined inputs!
  - Avoid buffer overflows (and other code injection problems)
  - Minimize process privileges
  - Carefully invoke other resources
  - Send information back carefully



# Fix This

```
main() {  
    int size = 1024, bytes;  
    char buf[size];  
  
    ...  
    bytes = read(fd, user_data, BLOCK_SIZE);  
  
    ...  
    strcpy(buf, user_data);  
}
```

# Type Safety

- Many problems are due to lack of type safety in C
  - Buffer overflows
  - Printf vulnerability
- However, many problems persist regardless
  - Integer overflow
  - Character and name
- Must handle user input carefully!
  
- **BUT:** Type safety and resolution of such programming errors would be helpful

- Java is a type-safe language similar to C++
  - In theory, buffer overflows and printf vulnerabilities are not possible
- In practice, not so easy
  - Must provide a type-safe bytecode
  - Researchers found that memory errors could enable violation in Java type safety
    - Took a lot of heat
- Meanwhile, Java and others do not address other problems via type safety
  - Integer overflow
  - Characters
  - Names
  - Using untrusted inputs (e.g., execute)

# Type Safe C? Etc.

- Variants of the C language offer type safety
  - Cyclone
    - Check whether pointers violate type safety dynamically
      - like Java, but other options
- Or tools to verify type safety
  - Ccured
    - Identifies all pointers that are not used in a type safe manner
    - Up to the programmer to address the others



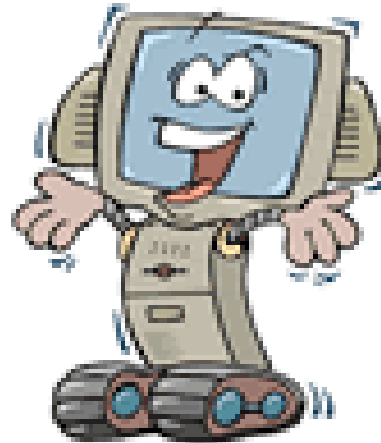
# C Source Code Analysis

- A number of tools have been developed to find bugs in C programs
- Lint/Splint
  - Finds buffer overflows
  - Some features integrated into GCC (-Wall)
- CQUAL
  - Finds malicious user input
  - Propagates type specifications (tainted)
  - Like taint mode in Perl
- Others
  - Prefix/Prefast: Finds simple C errors (multiple frees)
  - Meta-Compilation: Finds C errors based on automata specs
  - SLAM: Powerful, iterative refinement

# Source Code Analysis

- Can we prove a C program correct?

# Correct!



# Take Away

- Writing programs that do not contain a vulnerability is difficult
- Lack of type safety causes some problems
  - Buffer overflows
  - Not easy to eradicate completely
- But, other problems are present, *even in type safe languages*
  - Integer overflows
  - Input handling
- Source code analysis is helpful, but we cannot prove a program to be correct, *in general*