# Software Security

CS 4235/6035

# Security Flaws

- These flaws occur as a consequence of insufficient checking and validation of data and error codes in programs
- Awareness of these issues is a critical initial step in writing more secure program code
- Emphasis should be placed on the need for software developers to address these known areas of concern

# Defensive Programming

- Designing and implementing software so that it continues to function even when under attack
- Requires attention to all aspects of program execution, environment, and type of data it processes
- Software is able to detect erroneous conditions resulting from some attack
- Key rule is to never assume anything, check all assumptions and handle any possible error states

# Defensive Programming

- Programmers often make assumptions about the type of inputs a program will receive and the environment it executes in
  - Assumptions need to be validated by the program and all potential failures handled gracefully and safely
- Requires a changed mindset to traditional programming practices
  - Programmers have to understand how failures can occur and the steps needed to reduce the chance of them occurring in their programs

# Input Size & Buffer Overflow

- Programmers often make assumptions about the maximum expected size of input
  - Allocated buffer size is not confirmed
  - Resulting in buffer overflow
- Testing may not identify vulnerability
  - Test inputs are unlikely to include large enough inputs to trigger the overflow
- Safe coding treats all input as dangerous

# Validating Input Syntax

It is necessary to ensure that data conform with any assumptions made about the data before subsequent use

Input data should be compared against what is wanted

Alternative is to compare the input data with known dangerous values

By only accepting known safe data the program is more likely to remain secure

# Input Fuzzing

Software testing technique that uses randomly generated data as inputs to a program

- Range of inputs is very large
- Intent is to determine if the program or function correctly handles abnormal inputs
- Simple, free of assumptions, cheap

# Writing Safe Program Code

- Second component is processing of data by some algorithm to solve required problem
- High-level languages are typically compiled and linked into machine code which is then directly executed by the target processor

**Security issues:**

- Correct algorithm implementation
- Correct machine instructions for algorithm
- Valid manipulation of data

# Race Conditions

- Without synchronization of accesses it is possible that values may be corrupted or changes lost due to overlapping access, use, and replacement of shared values
- Arise when writing concurrent code whose solution requires the correct selection and use of appropriate synchronization primitives
- Deadlock
  - Processes or threads wait on a resource held by the other
  - One or more programs has to be terminated

# Preventing Race Conditions

- Programs may need to access a common system resource
- Need suitable synchronization mechanisms
  - Most common technique is to acquire a lock on the shared file
- Lockfile
  - Process must create and own the lockfile in order to gain access to the shared resource
  - Concerns
    - If a program chooses to ignore the existence of the lockfile and access the shared resource the system will not prevent this
    - All programs using this form of synchronization must cooperate
    - Implementation

# Handling Program Output

- Final component is program output
  - May be stored for future use, sent over net, displayed
  - May be binary or text
- Important from a program security perspective that the output conform to the expected form and interpretation
- Programs must identify what is permissible output content and filter any possibly untrusted data to ensure that only valid output is displayed
- Character set should be specified

# Buffer Overflow

- A very common attack mechanism
  - First widely used by the Morris Worm in 1988
- Prevention techniques known
- Still of major concern
  - Legacy of buggy code in widely deployed operating systems and applications
  - Continued careless programming practices by programmers

# Buffer Overflow Basics

- Programming error when a process attempts to store data beyond the limits of a fixed-sized buffer
- Overwrites adjacent memory locations
  - Locations could hold other program variables, parameters, or program control flow data
- Buffer could be located on the stack, in the heap, or in the data section of the process

**Consequences:**

- **Corruption of program data**
- **Unexpected transfer of control**
- **Memory access violations**
- **Execution of code chosen by attacker**

# Buffer Overflow Attacks

- To exploit a buffer overflow an attacker needs:
  - To identify a buffer overflow vulnerability in some program that can be triggered using externally sourced data under the attacker's control
  - To understand how that buffer is stored in memory and determine potential for corruption
- Identifying vulnerable programs can be done by:
  - Inspection of program source
  - Tracing the execution of programs as they process oversized input
  - Using tools such as fuzzing to automatically identify potentially vulnerable programs

# Stack Buffer Overflows

- Occur when buffer is located on stack
  - Also referred to as *stack smashing*
  - Used by Morris Worm
  - Exploits included an unchecked buffer overflow
- Are still being widely exploited
- Stack frame
  - When one function calls another it needs somewhere to save the return address
  - Also needs locations to save the parameters to be passed in to the called function and to possibly save register values

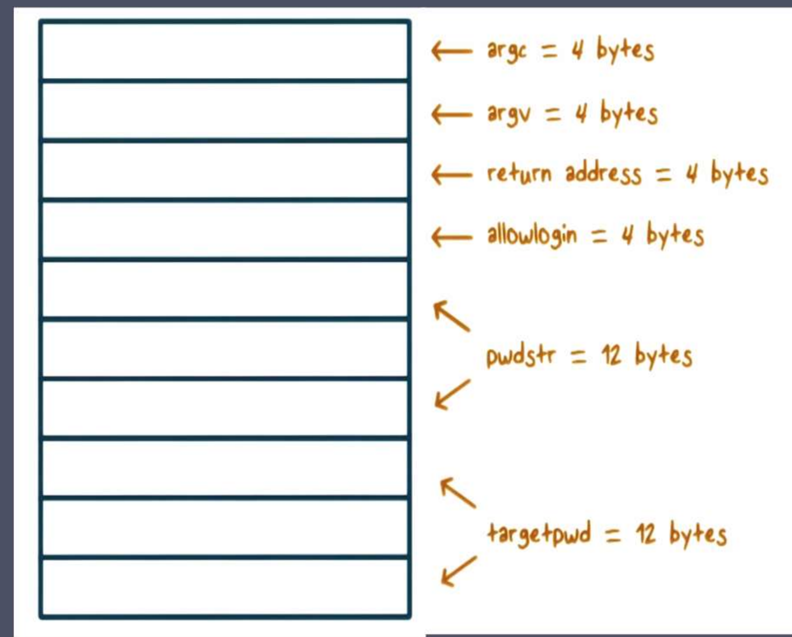
# Understanding the Stack Frame

# A Vulnerable Password Checking Program

```
#include <stdio.h>
#include <strings.h>

int main(int argc, char *argv[]) {
        int allow_login = 0;
        char pwdstr[12];
        char targetpwd[12] = "MyPwd123";
        gets(pwdstr);
        if (strncmp(pwdstr,targetpwd, 12) == 0)
                allow_login = 1;

        if (allow_login == 0)
                printf("Login request rejected");
        else
                printf("Login request allowed");
}
```

```c
#include <stdio.h>
#include <strings.h>

int main(int argc, char *argv[]) {
        int allow_login = 0;
        char pwdstr[12];
        char targetpwd[12] = "MyPwd123";
        gets(pwdstr);
        if (strncmp(pwdstr,targetpwd, 12) == 0)
                allow_login = 1;

        if (allow_login == 0)
                printf("Login request rejected");
        else
                printf("Login request allowed");
}
```

Stack



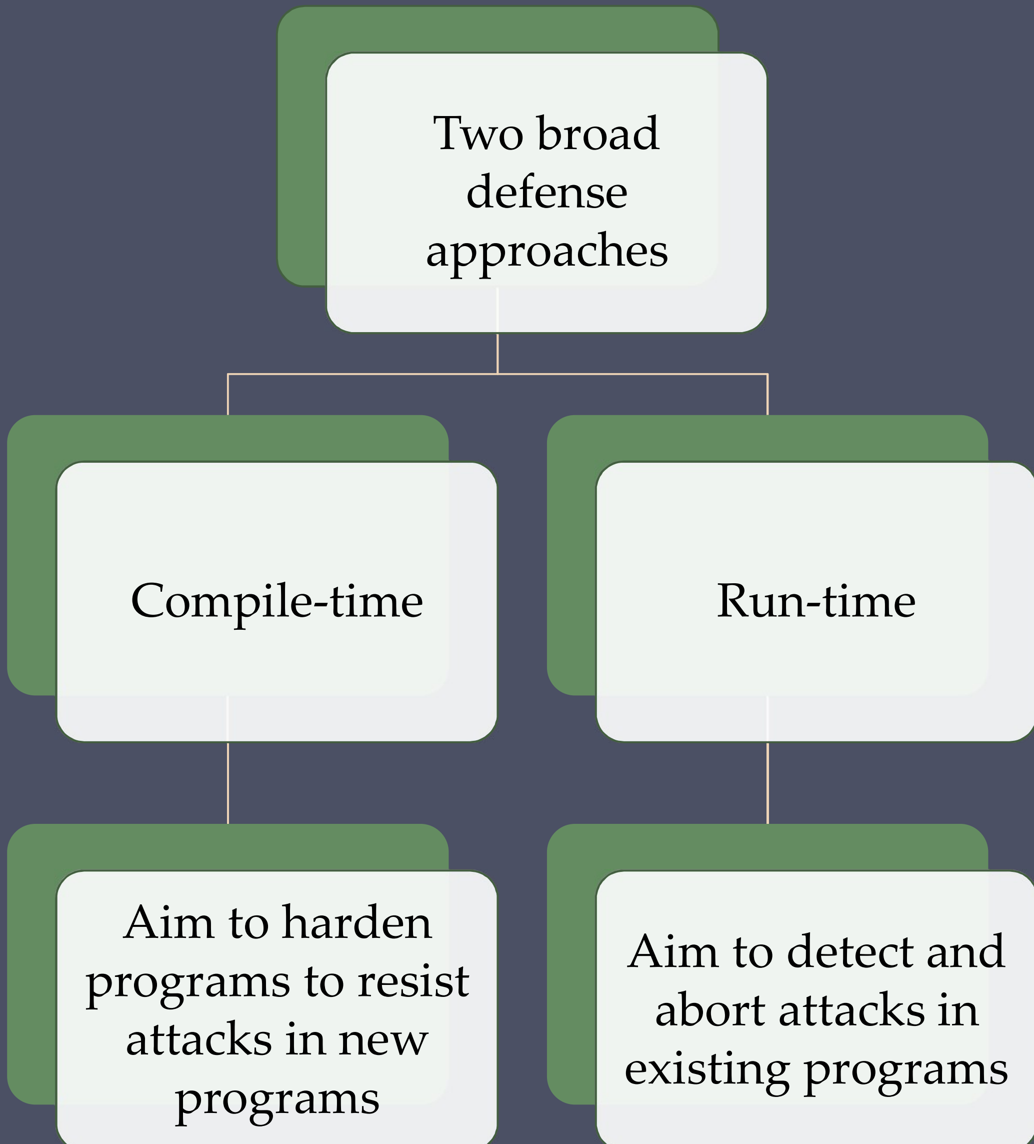

Potential attacks

- Any password of length greater than **12 bytes that ends in '123'**
- Any password of length greater than **16 bytes that begins with 'MyPwd123'**
- Any password of length greater than **8 bytes**

overwrite the "allowlogin" to be 1

# Buffer Overflow Defenses

- Buffer overflows are widely exploited



# Compile-Time Defenses: Programming Language

- Use a modern high-level language
  - Not vulnerable to buffer overflow attacks
  - Compiler enforces range checks and permissible operations on variables

| Disadvantages |
| --- |
| • Additional code must be executed at run time to impose checks<br>• Flexibility and safety comes at a cost in resource use<br>• Distance from the underlying machine language and architecture means that access to some instructions and hardware resources is lost<br>• Limits their usefulness in writing code, such as device drivers, that must interact with such resources |

# Compile-Time Defenses: Safe Coding Techniques

- C designers placed much more emphasis on space efficiency and performance considerations than on type safety
  - Assumed programmers would exercise due care in writing code
- Programmers need to inspect the code and rewrite any unsafe coding
- Java, Ada, Python safer than C

They do job collection for you.

# Compile-Time Defenses: Language Extensions/Safe Libraries

- Handling dynamically allocated memory is more problematic because the size information is not available at compile time

- Concern with C is use of unsafe standard library routines
  - One approach has been to replace these with safer variants
    - Libsafe is an example
    - Library is implemented as a dynamic library arranged to load before the existing standard libraries

# Run-Time Defenses: Executable Address Space Protection

**Use virtual memory support to make some regions of memory non-executable**

- Requires support from memory management unit (MMU)
- Long existed on SPARC / Solaris systems
- Recent on x86 Linux/Unix/Windows systems

**Issues**

- Support for executable stack code
- Special provisions are needed

# Run-Time Defenses: Address Space Randomization

- Manipulate location of key data structures
  - Stack, heap, global data
  - Using random shift for each process
  - Large address range on modern systems means wasting some has negligible impact
- Randomize location of heap buffers
- Random location of standard library functions

# Run-Time Defenses: Guard Pages

- Place guard pages between critical regions of memory
  - Flagged in MMU as illegal addresses
  - Any attempted access aborts process
- Further extension places guard pages Between stack frames and heap buffers
  - Cost in execution time to support the large number of page mappings necessary

# Summary

- Software security issues
  - Introducing software security and defensive programming
- Writing safe program code
  - Correct algorithm implementation
  - Ensuring that machine language corresponds to algorithm
  - Correct interpretation of data values
  - Correct use of memory
  - Preventing race conditions with shared memory
- Handling program output
- Handling program input
  - Input size and buffer overflow
  - Interpretation of program input
  - Validating input syntax
  - Input fuzzing
- Interacting with the operating system and other programs
  - Environment variables
  - Using appropriate, least privileges
  - Systems calls and standard library functions
  - Preventing race conditions with shared system resources
  - Safe temporary file use
  - Interacting with other programs