Software Security and Dependability

ENGR5560G

Lecture 10

Software Security
Buffer Overflow

Dr. Khalid A. Hafeez

Spring, 2025

Introduction

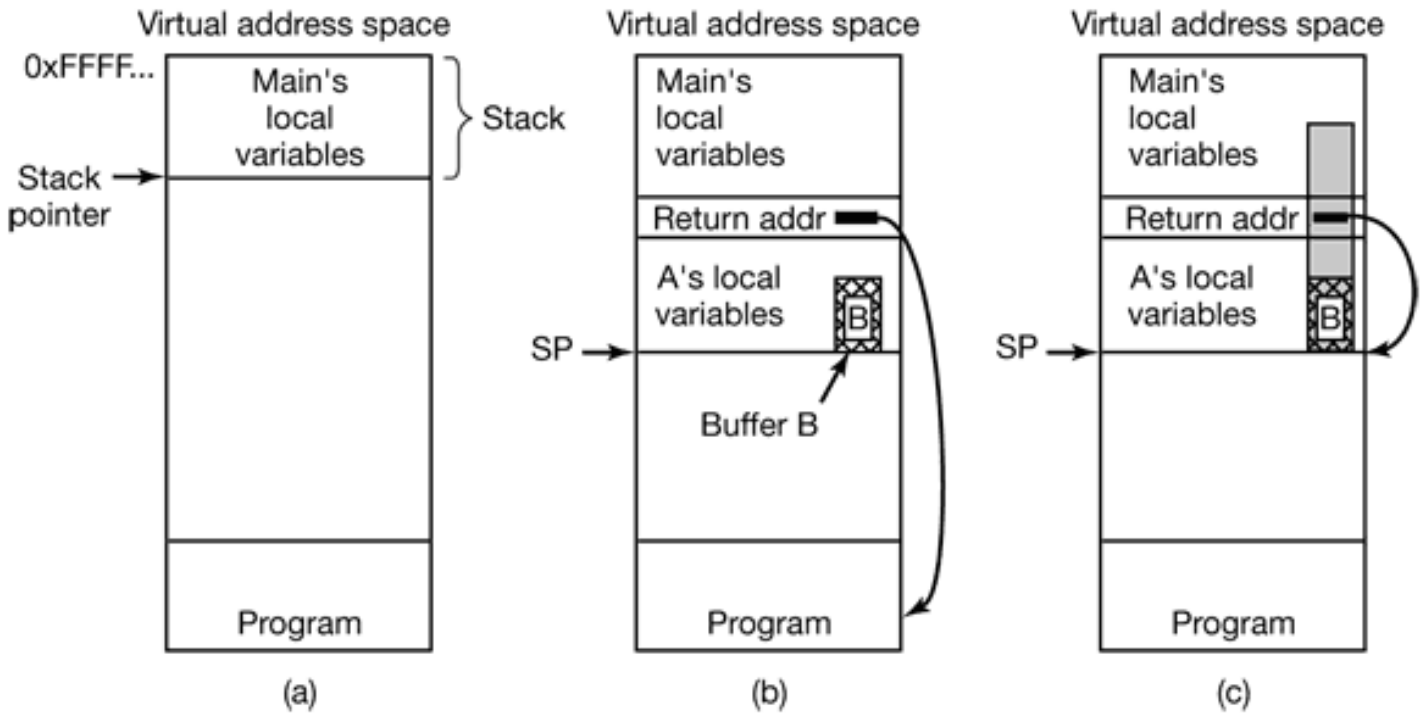
- Buffer Overflow:
  - A very common attack mechanism
  - Prevention techniques are known
  - Still of major concern because of:
    - Legacy of buggy code in widely deployed operating systems and applications
    - Continued careless programming practices by programmers
  - A Brief History of Some Buffer Overflow Attacks

1988	The Morris Internet Worm uses a buffer overflow exploit in "fingerd" as one of its attack mechanisms.
1995	A buffer overflow in NCSA httpd 1.3 was discovered and published on the Bugtraq mailing list by Thomas Lopatic.
1996	Aleph One published "Smashing the Stack for Fun and Profit" in *Phrack* magazine, giving a step by step introduction to exploiting stack-based buffer overflow vulnerabilities.
2001	The Code Red worm exploits a buffer overflow in Microsoft IIS 5.0.
2003	The Slammer worm exploits a buffer overflow in Microsoft SQL Server 2000.
2004	The Sasser worm exploits a buffer overflow in Microsoft Windows 2000/XP Local Security Authority Subsystem Service (LSASS).

Introduction

- Buffer Overflow (Buffer Overrun):
  - It is defined in the NIST Glossary of Key Information Security Terms as:
    - "A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system."

(a) Main program is running
(b) Program A is called
(c) Buffer overflow attack



Buffer Overflow Basics

- 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 Basics

- Example:

```
int main(int argc, char *argv[]) {
 int valid = FALSE;
 char str1[8];
 char str2[8];
 next_tag(str1);
 gets(str2);
 if (strncmp(str1, str2, 8) == 0)
   valid = TRUE;
   printf("buffer1: str1(%s), str2(%s), valid(%d)\n",
                                   str1, str2, valid);
 }
```

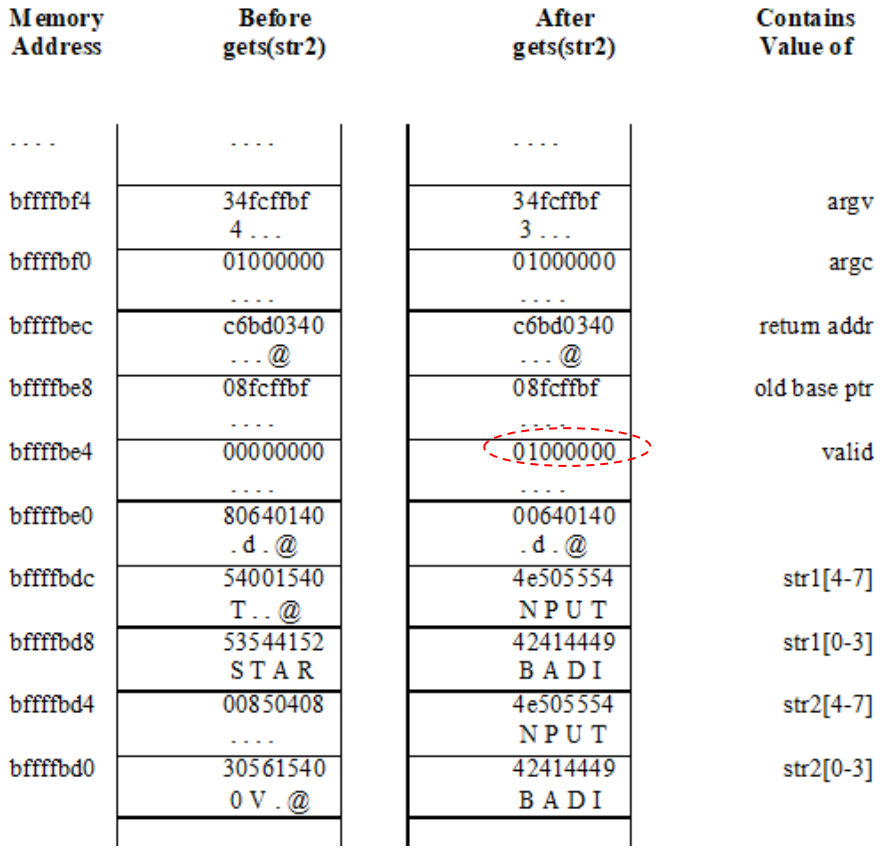
```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)

$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)

$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

the function next_tag(str1) to copy into str1 some expected tag value (assume START)

Problem:
gets() function does not include any checking on the amount of data copied



Buffer Overflow Attacks

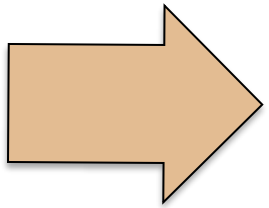
- To exploit a buffer overflow, an attacker needs:
  - To identify a buffer overflow vulnerability in some program that can be triggered using externally sourced data under the attacker's control
  - To understand how that buffer is stored in memory and determine potential for corruption
- Identifying 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

Buffer Overflow Basics

- Programming Language History:
  - At the machine level data manipulated by machine instructions executed by the computer processor are stored in either the processor's registers or in memory
  - Assembly language programmer is responsible for the correct interpretation of any saved data value

Modern high-level languages have a strong notion of type and valid operations

- **Not vulnerable to buffer overflows**
- **Does incur overhead, some limits on use**



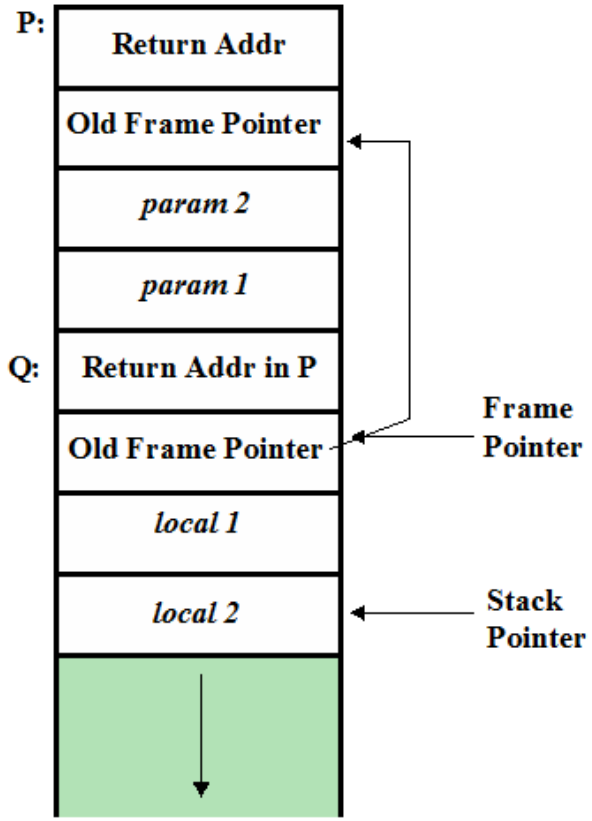
C and related languages have high-level control structures, but also allow direct access to memory

- **Hence are vulnerable to buffer overflow**
- **Have a large legacy of widely used, unsafe, and hence vulnerable code**

Stack Overflows

- Occur when buffer is located on stack
  - Also referred to as stack smashing
  - Used by Morris Worm through the use of C ***gets()*** function
  - Exploits included an unchecked buffer overflow
- Are still being widely exploited

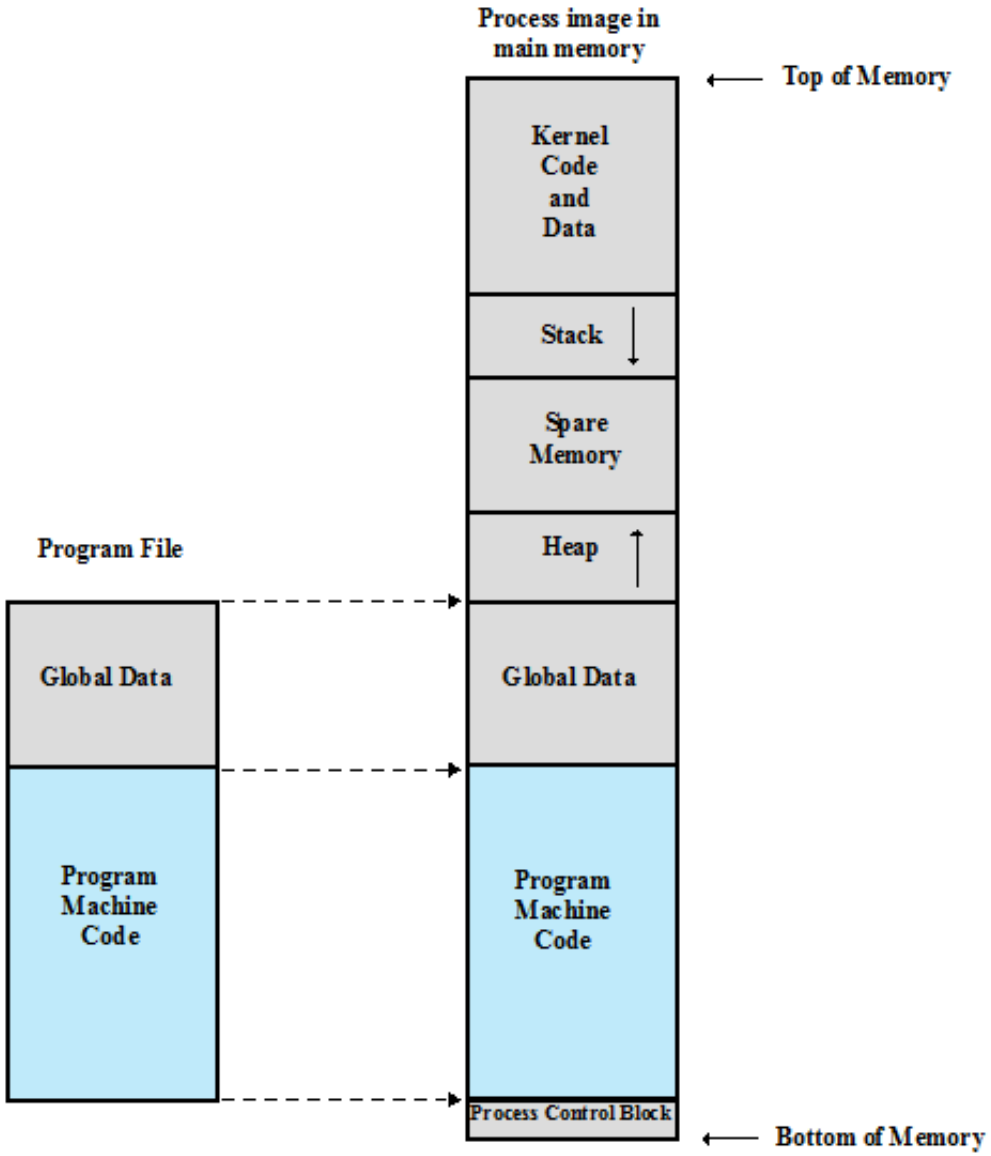
- Stack frame
  - When one function calls another it needs somewhere to save the return address
  - Also needs locations to save the parameters to be passed in to the called function and to possibly save register values
- The possibility of overwriting the saved frame pointer and return address forms the core of a stack overflow attack.



Example: Stack Frame with Functions P calling Q

Stack Overflows

- Stack Overflow Example:
  - Program loading into process memory



Stack Overflows- Example 1

```
void hello(char *tag)
{
   char inp[16];

   printf("Enter value for %s: ", tag);
   gets(inp);
   printf("Hello your %s is %s\n", tag, inp);
}
```

(a) Basic stack overflow C code

```
$ cc -g -o buffer2 buffer2.c

$ ./buffer2
Enter value for name: Bill and Lawrie
Hello your name is Bill and Lawrie
buffer2 done

$ ./buffer2
Enter value for name: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Segmentation fault (core dumped)

$ perl -e 'print pack("H*","41424344454647485152535455565758616263646566676808fcffbf948304080a4e4e4e4e0a");' | ./buffer2
Enter value for name:
Hello your Re?pyy]uEA is ABCDEFGHQRSTUVWXabcdefguyu
Enter value for Kyyu:
Hello your Kyyu is NNNN
Segmentation fault (core dumped)
```

(b) Basic stack overflow example runs

Memory Address	Before gets(inp)	After gets(inp)	Contains Value of
. . . .	. . . .	. . . .	
bffffbe0	3e850408 > . . .	00850408	tag
bffffbdc	f0830408	94830408	return addr
bffffbd8	e8fbffbf	e8ffffbf	old base ptr
bffffbd4	60840408 ` . . .	65666768 e f g h	
bffffbd0	30561540 0 V . @	61626364 a b c d	
bffffbcc	1b840408	55565758 U V W X	inp[12-15]
bffffbc8	e8fbffbf	51525354 Q R S T	inp[8-11]
bffffbc4	3cfcffbf < . . .	45464748 E F G H	inp[4-7]
bffffbc0	34fcffbf 4 . . .	41424344 A B C D	inp[0-3]

Stack Overflows- Example 2

```
void getinp(char *inp, int siz)
{
   puts("Input value: ");
   fgets(inp, siz, stdin);
   printf("buffer3 getinp read %s\n", inp);
}

void display(char *val)
{
   char tmp[16];
   sprintf(tmp, "read val: %s\n", val);
   puts(tmp);
}

int main(int argc, char *argv[])
{
   char buf[16];
   getinp(buf, sizeof(buf));
   display(buf);
   printf("buffer3 done\n");
}
```

(a) Another stack overflow C code

```
$ cc -o buffer3 buffer3.c

$ ./buffer3
Input value:
SAFE
buffer3 getinp read SAFE
read val: SAFE
buffer3 done

$ ./buffer3
Input value:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
buffer3 getinp read XXXXXXXXXXXXXXXX
read val: XXXXXXXXXXXXXXXX

buffer3 done
Segmentation fault (core dumped)
```

(b) Another stack overflow example runs

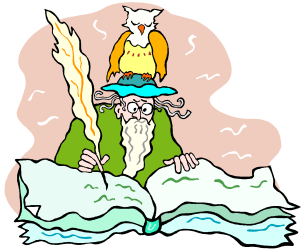
Stack Overflows

- Some Common Unsafe C Standard Library Routines :

`gets(char *str)`	read line from standard input into str
`sprintf(char *str, char *format, ...)`	create str according to supplied format and variables
`strcat(char *dest, char *src)`	append contents of string src to string dest
`strcpy(char *dest, char *src)`	copy contents of string src to string dest
`vsprintf(char *str, char *fmt, va_list ap)`	create str according to supplied format and variables

Shellcode

- Shellcode:
  - It is a code supplied by attacker
    - Often saved in buffer being overflowed
    - Traditionally transferred control to a user command-line interpreter (shell)
  - Usually is a machine code
    - Specific to processor and operating system
    - Traditionally needed good assembly language skills to create
    - More recently a number of sites and tools have been developed that automate this process



  - Metasploit Project
    - Provides useful information to people who perform penetration testing, IDS signature development, and exploit research

Stack Overflow Variants

Target program can be:

- A trusted system utility
- Network service daemon
- Commonly used library code

Shellcode functions

- **Launch a remote shell when connected to**
- **Create a reverse shell that connects back to the hacker**
- **Use local exploits that establish a shell**
- **Flush firewall rules that currently block other attacks**
- **Break out of a chroot (restricted execution) environment, giving full access to the system**

Buffer Overflow Defenses

- Buffer overflows are widely exploited:
  - **Two broad defense approaches**
    - Compile-time defenses, which aim to harden programs to resist attacks in new programs
    - Run-time defenses, which aim to detect and abort attacks in existing programs

Buffer Overflow Defenses - Compile-Time Defenses

1. Choice of 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
  - Flexibility and safety comes at a cost in resource use
  - Distance from the underlying machine language and architecture means that access to some instructions and hardware resources is lost
  - Limits their usefulness in writing code, such as device drivers, that must interact with such resources

Buffer Overflow Defenses - Compile-Time Defenses

2. 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
  - An example of this is the OpenBSD project
    - Programmers have audited the existing code base, including the operating system, standard libraries, and common utilities
      - This has resulted in what is widely regarded as one of the safest operating systems in widespread use

Buffer Overflow Defenses - Compile-Time Defenses

- Example: Unsafe C Code
  - Example of an unsafe byte copy function. This code copies ***len*** bytes out of the ***from*** array into the ***to*** array starting at position ***pos*** and returning the end position.

- This function is given no information about the actual size of the destination buffer ***to*** and hence is unable to ensure an overflow does not occur.

```
int copy_buf(char *to, int pos, char *from, int len)
{
   int i;

   for (i=0; i<len; i++) {
      to[pos] = from[i];
      pos++;
   }
   return pos;
}
```

(a) Unsafe byte copy

- Example of an unsafe byte input function. It reads the length of binary data expected and then reads that number of bytes into the destination buffer.
- The problem is that this code is not given any information about the size of the buffer and hence is unable to check for possible overflow.

```
short read_chunk(FILE fil, char *to)
{
   short len;
   fread(&len, 2, 1, fil);..... ..... /* read length of binary data */
   fread(to, 1, len, fil);..... ..... /* read len bytes of binary data
   return len;
}
```

(b) Unsafe byte input

Buffer Overflow Defenses - Compile-Time Defenses

3. Language Extensions and Safe Libraries

- C compilers can automatically insert range checks for statically allocated arrays
- But handling dynamically allocated memory is more problematic because the ***size*** information is not available at compile time
  - Requires a new extension and the use of new library routines
    - Old programs and libraries need to be recompiled
- Concern with **C** is the use of unsafe standard library routines (like string manipulation routines)
  - One approach has been to replace these with safer variants
    - Libsafe is an example which is implemented as a dynamic library arranged to load before the existing standard libraries

Buffer Overflow Defenses - Compile-Time Defenses

4. Stack Protection Mechanisms

- Add function entry and exit code to check the stack for signs of corruption
- Use random canary value below the old frame pointer address
  - The canary value needs to be unpredictable and should be different on different systems
- Use Stackshield and Return Address Defender (RAD)
  - They are GCC compiler extensions that include additional function entry and exit code
    - Function entry writes a copy of the return address to a safe region of memory
    - Function exit code checks the return address in the stack frame against the saved copy
    - If change is found, aborts the program

Buffer Overflow Defenses - Run-Time Defenses

1. Executable Address Space Protection
   - Can be done by blocking the execution of code on the stack
   - Use virtual memory support to make some regions of memory (like the stack) non-executable
     - Requires support from memory management unit (MMU)
     - Long existed on SPARC / Solaris systems
     - Recently added on x86 Linux/Unix/Windows systems
   - Issue
     - Support for executable stack code

Buffer Overflow Defenses - Run-Time Defenses

2. 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

Buffer Overflow Defenses - Run-Time Defenses

3. 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

Software Security Issues

- Many vulnerabilities result from poor programming practices
- Consequence from insufficient checking and validation of data and error codes
  - Awareness of these issues is a critical initial step in writing more secure program code
- Software error categories:
  - Insecure interaction between components
  - Risky resource management
  - Porous defenses

Software Security Issues

- CWE/SANS Top 25 Most Dangerous Software Errors

Software Error Category: Insecure Interaction Between Components
Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
Unrestricted Upload of File with Dangerous Type
Cross-Site Request Forgery (CSRF)
URL Redirection to Untrusted Site ('Open Redirect')
Software Error Category: Risky Resource Management
Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
Download of Code Without Integrity Check
Inclusion of Functionality from Untrusted Control Sphere
Use of Potentially Dangerous Function
Incorrect Calculation of Buffer Size
Uncontrolled Format String
Integer Overflow or Wraparound
Software Error Category: Porous Defenses
Missing Authentication for Critical Function
Missing Authorization
Use of Hard-coded Credentials
Missing Encryption of Sensitive Data
Reliance on Untrusted Inputs in a Security Decision
Execution with Unnecessary Privileges
Incorrect Authorization
Incorrect Permission Assignment for Critical Resource
Use of a Broken or Risky Cryptographic Algorithm
Improper Restriction of Excessive Authentication Attempts
Use of a One-Way Hash without a Salt

Security Flaws

- Critical Web application security flaws includes five related to insecure software code
  - Invalidated input
  - Cross-site scripting
  - Buffer overflow
  - Injection flaws
  - Improper error handling

- These flaws occur because 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
  - Software developers should address these known areas of concern

Reducing Software Vulnerabilities

- NIST (NISTIR 8151) presents a range of approaches to reduce the number of software vulnerabilities:
  - Stop vulnerabilities before they occur by using improved methods for specifying and building software
  - Find vulnerabilities before they can be exploited by using better and more efficient testing techniques
  - Reduce the impact of vulnerabilities by building more resilient architectures

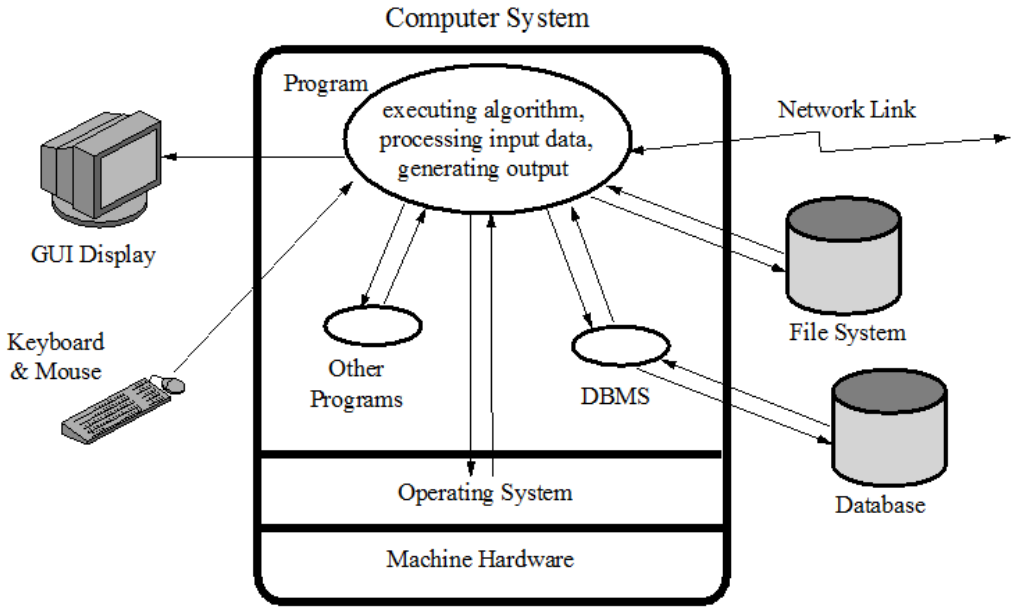
Software Security, Quality, and Reliability

- Software Quality and Reliability:
  - Concerned with the accidental failure of program as a result of some theoretically random, unanticipated input, system interaction, or use of incorrect code
  - Improve using structured design and testing to identify and eliminate as many bugs as possible from a program
  - Concern is not how many bugs, but how often they are triggered

- Software Security:
  - Attacker chooses the probability distribution, specifically targeting bugs that result in a failure that can be exploited by the attacker
  - Triggered by inputs that differ dramatically from what is usually expected
  - Unlikely to be identified by common testing approaches

Defensive or Secure Programming

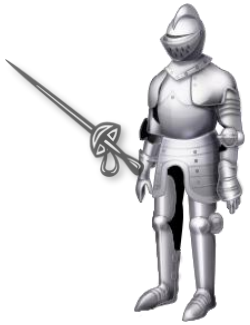
- Designing and implementing software so that it continues to function even when under attack
- Requires attention to all aspects of program execution, environment, and type of data it processes
- Software is able to detect erroneous conditions resulting from some attack
  - So, either continue executing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



Abstract view of a program

Defensive or Secure 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 must understand how failures can occur and the steps needed to reduce the chance of them occurring in their programs
- Conflicts with business pressures to keep development times as short as possible to maximize market advantage



Security by Design

- Security and reliability are common design goals in most engineering disciplines
- Software development not as mature
- Recent years have seen increasing efforts to improve secure software development processes
- Software Assurance Forum for Excellence in Code (SAFECode)
  - Develop publications outlining industry best practices for software assurance and providing practical advice for implementing proven methods for secure software development

Handling Program Input

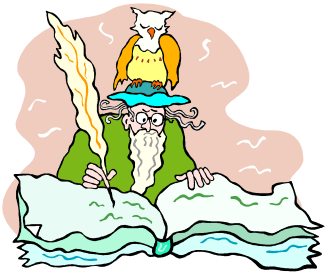
- Incorrect handling of program input is a very common failing
- Input is any source of data from outside and whose value is not explicitly known by the programmer when the code was written
- Must identify all data sources
- Explicitly validate assumptions on size and type of values before use

Handling Program Input

- 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

Handling Program Input

- Interpretation of Program Input:
  - Program input may be binary or text
    - Binary interpretation depends on encoding and is usually application specific
  - There is an increasing variety of character sets being used
    - Care is needed to identify just which set is being used and what characters are being read
  - Failure to validate may result in an exploitable vulnerability
  - 2014 Heartbleed OpenSSL bug is an example of a failure to check the validity of a binary input value



Handling Program Input

- Injection Attacks:
  - Flaws relating to invalid handling of input data, specifically when program input data can accidentally or deliberately influence the flow of execution of the program
  - Most often occur in scripting languages
    - Encourage reuse of other programs and system utilities where possible to save coding effort
    - Often used as Web CGI (Common Gateway Interface) scripts

Handling Program Input

- 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 (whitelisting)

Alternative is to compare the input data with known dangerous values

(blacklisting)

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

Handling Program Input

- Validating Input Syntax:
  - Alternate Encodings
    - Growing requirement to support users around the globe and to interact with them using their own languages
      - May have multiple means of encoding text
    - Unicode used for internationalization
      - Uses 16-bit value for characters
      - UTF-8 encodes as 1-to- 4 byte sequences
      - Many Unicode decoders accept any valid equivalent sequence
    - Canonicalization
      - Transforming input data into a single, standard, minimal representation
      - Once this is done the input data can be compared with a single representation of acceptable input values

Handling Program Input

- Validating Numeric Input:
  - Additional concern when input data represents numeric values
  - Internally stored in fixed sized value
    - 8, 16, 32, 64-bit integers
    - Floating point numbers depend on the processor used
    - Values may be signed or unsigned
  - Must correctly interpret text form and process consistently
    - Have issues comparing signed to unsigned
    - Could be used to thwart buffer overflow check

Handling Program Input

- Validating Numeric Input:
  - Input Fuzzing
    - Developed by Professor Barton Miller at the University of Wisconsin Madison in 1989
    - 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
      - Assists with reliability as well as security
    - Can also use templates to generate classes of known problem inputs
      - Disadvantage is that bugs triggered by other forms of input would be missed
    - Combination of approaches is needed for reasonably comprehensive coverage of the inputs

Writing Safe Program Code

- Consider 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 values in variables as stored in registers or memory

Writing Safe Program Code

- Correct Algorithm Implementation:

Issue of good program development technique

- Algorithm may not correctly handle all problem variants
- Consequence of deficiency is a bug in the resulting program that could be exploited

Initial sequence numbers used by many TCP/IP implementations are too predictable

- Combination of the sequence number as an identifier and authenticator of packets and the failure to make them sufficiently unpredictable enables the attack to occur

Another variant is when the programmers deliberately include additional code in a program to help test and debug it

- Often code remains in production release of a program and could inappropriately release information
- May permit a user to bypass security checks and perform actions they would not otherwise be allowed to perform
- This vulnerability was exploited by the Morris Internet Worm

Writing Safe Program Code

- Ensuring Machine Language Corresponds to Algorithm:
  - Issue is ignored by most programmers
    - Assumption is that the compiler or interpreter generates or executes code that validly implements the language statements
  - Requires comparing machine code with original source
    - Slow and difficult
  - Development of computer systems with very high assurance level is the one area where this level of checking is required
    - Specifically Common Criteria assurance level of EAL 7

Writing Safe Program Code

- Correct Data Interpretation:
  - Data stored as bits/bytes in computer
    - Grouped as words or longwords
    - Accessed and manipulated in memory or copied into processor registers before being used
    - Interpretation depends on machine instruction executed
  - Different languages provide different capabilities for restricting and validating interpretation of data in variables
    - Strongly typed languages are more limited, safer
    - Other languages (such as C) allow more liberal interpretation of data and permit program code to explicitly change their interpretation

Writing Safe Program Code

- Preventing 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
    - Various processes or threads wait on a resource held by the other
    - One or more programs must be terminated