İTÜ
Computer Security

Software and Operating System Security

Dr. Şerif Bahtiyar

bahtiyars@itu.edu.tr

Fall 2015

Before Starting

The secret to staying safe online

....more than 40 million people in the US had their personal information stolen in 2014, as well as **54 million in Turkey**, 20 million in Korea, 16 million in Germany and more than 20 million in China....

http://www.bbc.com/future/story/20141010-the-secret-to-staying-safe-online

Outline

- Buffer Overflow
- Basics of Software Security
- Handling Program Input
- Handling Program Output
- Interacting with Operating System
- Writing Safe Program Code

Roots of Security Threats

- Threat: A potential for violation of security. A threat is a possible danger that might exploit a vulnerability.
- Vulnerability: A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy.

```
if ((err = SSLHashSHA1.update(
    &hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
```

SSL Vulnerability

- Attack: An assault on system security that derives from an intelligent threat. It is a deliberate attempt to evade security services and violate the security policy of a system.

Buffer Overflow

- Buffer Overrun (Overflow): A condition at an interface under which more input can be placed into a buffer or data holding area than the capability allocated, overwriting other information.
- Attackers exploit such a condition to crash a system or to insert specifically crafted code that allows them to gain control of the system.
- Overflow attacks is one of the most common attacks seen and results from careless programming in applications.

```
...
char buf[BUFSIZE];
gets(buf);
...
```

- The buffer can be located on the stack, in the heap, or in the data section of the process.

Buffer Overflow

- Testing of programs may not identify the buffer overflow vulnerability, as the test inputs provided would usually reflect the range of inputs the programmers expect users to provide.

- Consequences of buffer overflow:
  - Corruption of data used by the program,
  - Unexpected transfer of control,
  - Memory access violations,
  - Program termination,
  - If it is a part of an attack, run attacker’s code.

Buffer Overflow

```
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);
}
```

(a) Basic buffer overflow C code

```
$ 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)
```

(b) Basic buffer overflow example runs

Buffer Overflow

Memory Address	Before gets(str2)	After gets(str2)	Contains value of
. . . .	. . . .	. . . .	
bffffbf4	34fcffbf 4 . . .	34fcffbf 3 . . .	argv
bffffbf0	01000000	01000000	argc
bffffbec	c6bd0340 . . . @	c6bd0340 . . . @	return addr
bffffbe8	08fcffbf	08fcffbf	old base ptr
bffffbe4	00000000	01000000	valid
bffffbe0	80640140 . d . @	00640140 . d . @	
bffffbdc	54001540 T . . @	4e505554 N P U T	str1[4-7]
bffffbd8	53544152 S T A R	42414449 B A D I	str1[0-3]
bffffbd4	00850408	4e505554 N P U T	str2[4-7]
bffffbd0	30561540 0 V . @	42414449 B A D I	str2[0-3]
. . . .	. . . .	. . . .	

Figure 10.2 Basic Buffer Overflow Stack Values

Buffer Overflow

- To exploit any type of buffer overflow the attacker needs
  - identify a buffer overflow vulnerability
  - understand how that buffer will be stored in the processes memory

- Programming languages and buffer overflow
  - Assembly and machine code (instructions): greatest access to computer resources and programming effort. (vulnerable)
  - High level programming languages (Java, ADA, Python): require high computer resources and no direct access to hardware resources (not vulnerable)
  - Languages like C: have many modern control-structures and data type abstractions, provide access to hardware (vulnerable)

Buffer Overflow
Stack Buffer Overflows

- A stack buffer overflow (stack smashing) occurs when the targeted buffer is located on the stack, usually as a local variable in a function’s stack frame.

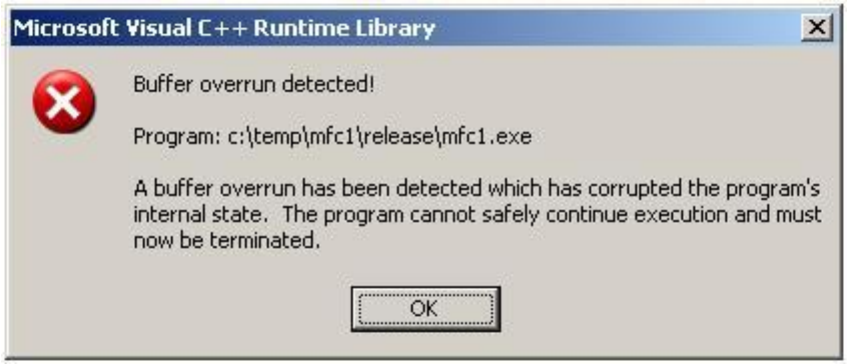
- Morris Internet worm:
  - First being seen in the wild in 1988
  - Uses an unchecked buffer of the C gets() function in the fingerd deamon.

Buffer Overflow
Stack Buffer Overflows

A stack overflow example

- Because local variables are placed below the saved frame pointer and return address, the possibility exists of exploiting a local buffer variable overflow vulnerability to overwrite the values of one or both of these key function linkage values.

- A stack overflow can result in some form of denial-of-service attack.



Buffer Overflow
Stack Buffer Overflows

```
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*", "414243444546474851525354555657586162636465666768
08fcffbf948304080a4e4e4e4e0a");' | ./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

Buffer Overflow

Shellcode is machine code or series of binary values corresponding to the machine instructions and data values that implement the attacker’s desired functionality.

- It is an essential component of many buffer overflow attacks to transfer the execution to the code supplied by the attacker and often saved in the buffer being overflowed.
- Specific to particular processor architecture and operating system.

Buffer Overflow

- Buffer overflow defenses
  - Compile time defenses: aim to harden programs to resist attacks in new programs.
  - Run time defenses: aim to detect and abort attacks in existing programs.
- To prevent buffer overflow
  - Use a dynamically sized buffer to ensure that sufficient space is available
  - Space requested does not exceed available memory for dynamic sizes
  - Process the input in buffer sized blocks
  - Discard excess input
  - Terminate the program

Basics of Software Security

Many security vulnerabilities results from poor programming practices.

Table 11.1 CWE/SANS TOP 25 Most Dangerous Software Errors

Software Error Category: Insecure Interaction Between Components
Failure to Preserve Web Page Structure ("Cross-site Scripting")
Failure to Preserve SQL Query Structure (aka "SQL Injection")
Cross-Site Request Forgery (CSRF)
Unrestricted Upload of File with Dangerous Type
Failure to Preserve OS Command Structure (aka "OS Command Injection")
Information Exposure Through an Error Message
URL Redirection to Untrusted Site ("Open Redirect")
Race Condition
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")
Improper Control of Filename for Include/Require Statement in PHP Program ("PHP File Inclusion")
Buffer Access with Incorrect Length Value
Improper Check for Unusual or Exceptional Conditions
Improper Validation of Array Index
Integer Overflow or Wraparound
Incorrect Calculation of Buffer Size
Download of Code Without Integrity Check
Allocation of Resources Without Limits or Throttling
Software Error Category: Porous Defenses
Improper Access Control (Authorization)
Reliance on Untrusted Inputs in a Security Decision
Missing Encryption of Sensitive Data
Use of Hard-coded Credentials
Missing Authentication for Critical Function
Incorrect Permission Assignment for Critical Resource
Use of a Broken or Risky Cryptographic Algorithm

Basics of Software Security

- Software quality and reliability is concerned with the accidental failure of a program as a result of
  - unanticipated input,
  - system interaction,
  - use of incorrect code.

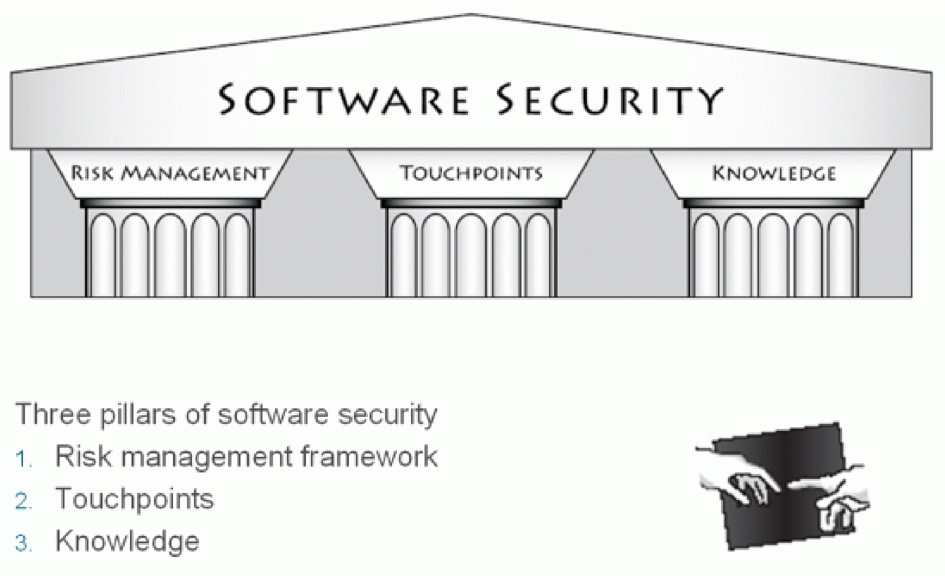
- To improve software quality

  Use some form of structured design and testing to identify and eliminate bugs.

Basics of Software Security

In software security,

- The attacker chooses the probability distribution, targeting specific bugs that result in a failure.
- The bugs are triggered by often very unlikely inputs and common tests do not identify them.

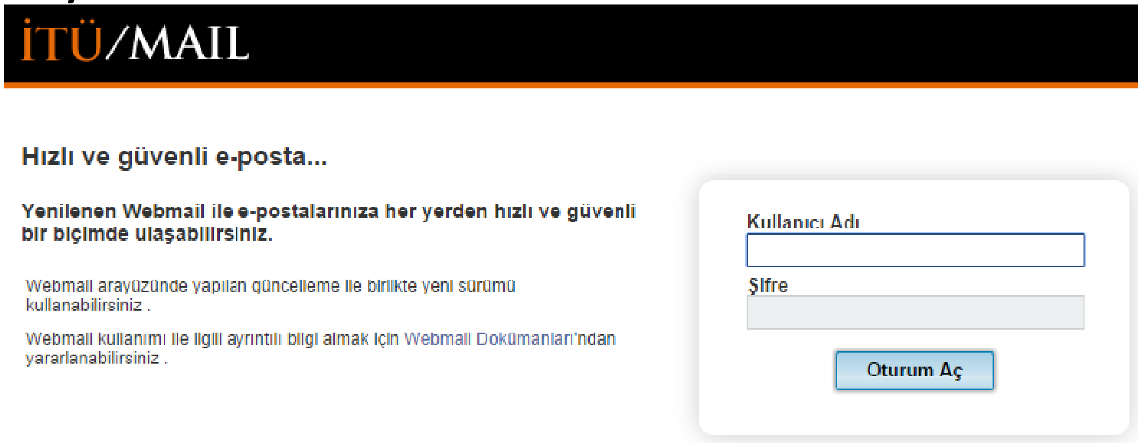


Basics of Software Security

- Software security assurance is a process that helps design and implement software that protects the data and resources contained in and controlled by that software.
- Software security assurance includes
  - A security evaluation
  - Security requirements for software
  - Security requirements for software development, operations, maintenance processes
  - Evaluation for each software audit and review
  - A configuration management and corrective action process
  - Adequate physical security for software

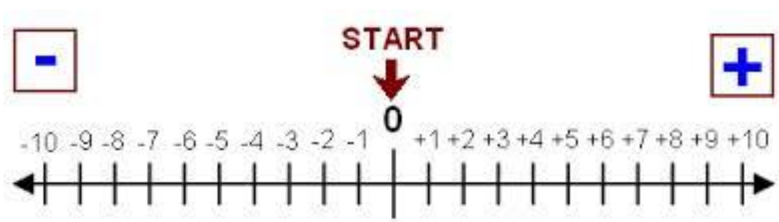
Handling Program Input

- Incorrect handling of program input is one of the most common failings in software security.
- Input is any source of data from outside, such as
  - data read from keyboard, mouse, file, network
  - execution environment
- Input data and their source must be
  - identified and explicitly verified



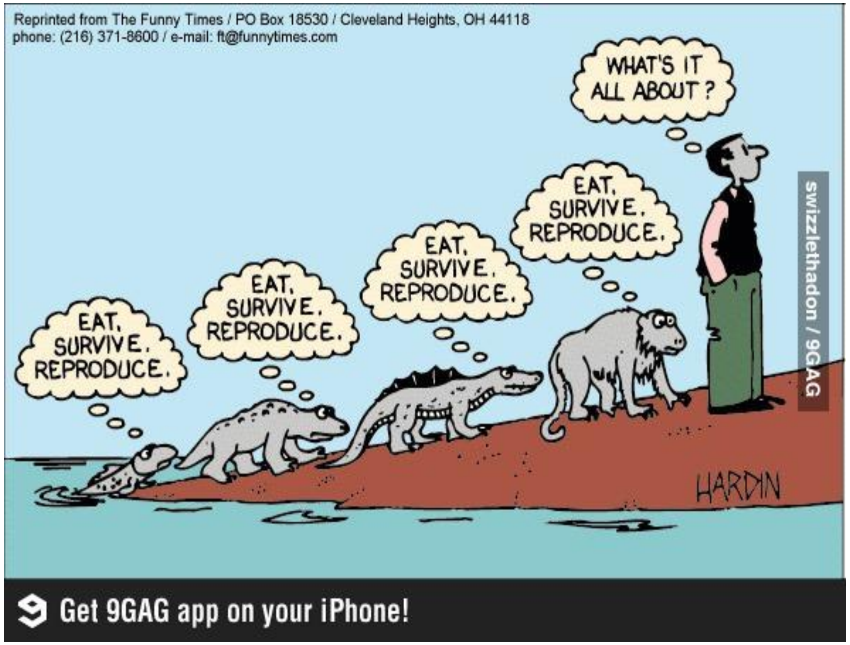
Handling Program Input

- Meaning and interpretation of input is a key concern for programs.
- Input data may be broadly classified as textual or binary.
- Interpretation of the raw binary values may represent integers, floating-point numbers, character strings, or some more complex structures.



Handling Program Input

- Beyond identifying which characters are input, their meaning must be identified.
  - Filename
  - URL
  - E-mail address
  - ….
- Failure to identify the meaning could result in a vulnerability that permits an attacker to influence the operation of the program, with possibly serious consequences.



Handling Program Input

- Injection attack refers to a wide variety of program flaws related to invalid handling of input data.
  - OWASP describes 25 different injection attacks (https://www.owasp.org/index.php/Category:Injection)

- These attacks occurs when program input data can accidentally or deliberately influence the flow of execution of the program.
- Flaws related to invalid handling of input data
  - influences program execution
  - passed as a parameter to a helper program or other utility or subsystem
  - most often occurs in scripting languages, such as Web CGI scripts to process data supplied from HTML formats

Handling Program Input

Code Injection

- The input includes code that is executed by the attacked system.
- This type of attack is widely exploited.

Command Injection

- The input is used in the construction of a command that is subsequently executed by the system with the privileges of the Web server.
- The problem caused by insufficient checking of program input.

Handling Program Input

```
int main(char* argc, char** argv) {
        char cmd[CMD_MAX] = "/usr/bin/cat ";
        strcat(cmd, argv[1]);
        system(cmd);
}
```

- *The program accepts a filename as a command line argument, and displays the contents of the file.*
- *if an attacker passes a string of the form ";rm -rf /", then the call to system() fails to execute cat due to a lack of arguments and then plows on to recursively delete the contents of the root partition.*

Handling Program Input

SQL Injection

- The user-supplied input is used to construct a SQL request to retrieve information from a database.
- Must check and validate input

```
SELECT UserList.Username
FROM UserList
WHERE UserList.Username = 'Username'
AND UserList.Password = 'Password'
```

```
SELECT UserList.Username
FROM UserList
WHERE UserList.Username = 'Username'
AND UserList.Password = 'password' OR '1'='1'
```

Handling Program Input

Cross-Site Scripting (XSS) Attacks

- Input from one user is later output to another user.
- Commonly seen scripted Web applications
- With script code that can be JavaScript, ActiveX, VBScript, Flash, ...
- Assumed that all content from one site is equally trusted and permitted to interact with other sites

To Prevent XSS Attacks

- Identify other programs that could not be trusted.
- If it is necessary to trust other programs, filter their output.
- Ensure that untrusted sources were not permitted to direct output.

Handling Program Input

Validating Input Syntax

- Ensure data conform to assumptions, eg. HTML, email, printable
- Compare against what is wanted, accept only valid data
- Alternative: compare input data with known dangerous values
- To validate inputs regular expressions are used
  - Patterns of characters describe allowable input
  - Details vary between languages
- The input data have the possibility of multiple encodings

  The input data must first be transformed into a single, standard, minimal representation known as canonicalization, such as unicode.

Handling Program Input

Input Fuzzing

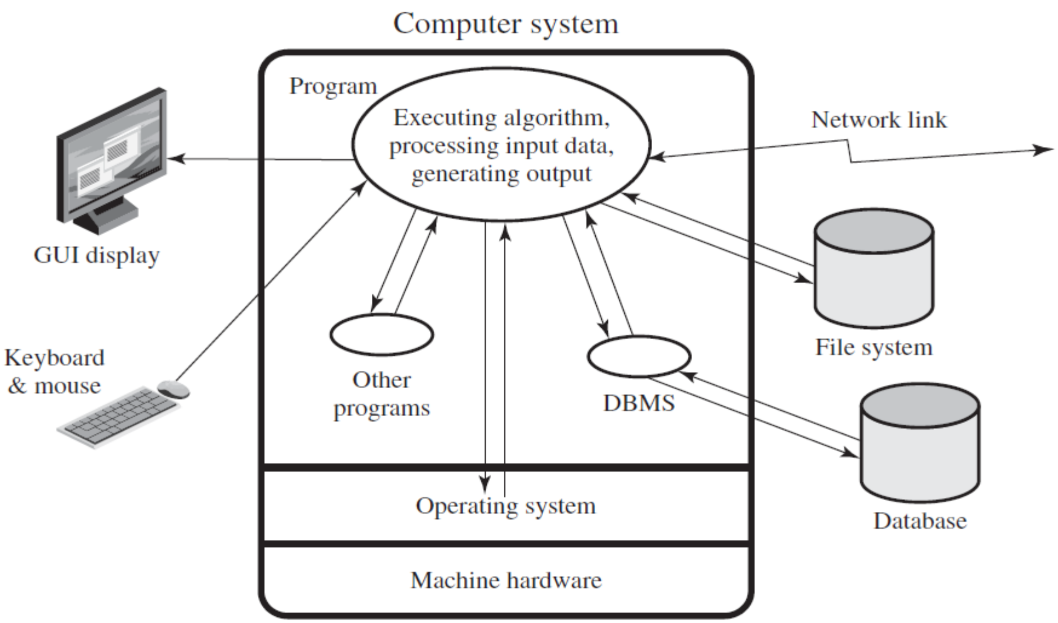
- A software testing technique that uses randomly generated data as inputs to a program.
- Advantage is simplicity and freedom from assumptions about expected input.
- Inputs may be generated according to templates but disadvantage is that the templates incorporate assumptions about the input so some bugs triggered by other forms would be missed.

Limitations

- Only identifies simple types of faults
- If a bug is triggered only with a small number of inputs, fuzzing is unlikely to locate it.

Handling Program Output

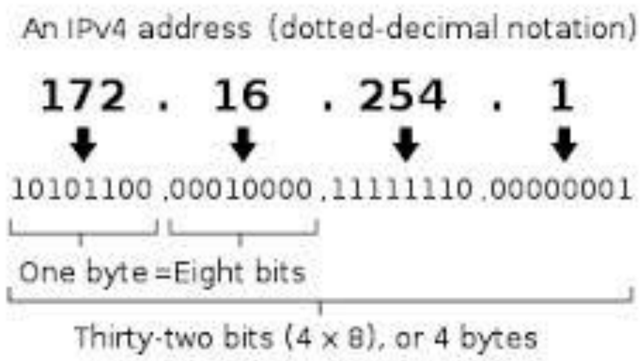
Program Output: It is the generation of output as a result of the processing of input and other interactions.



The target of compromise is not program generating the output but rather the program or device used

Handling Program Output

- Purpose of program outputs
  - Stored for future use
  - Transmitted over networks
  - Displayed to user



- A simple categorization
  - Binary : Complex structures such as network protocol structures
  - Textual : Some structured output such as HTML

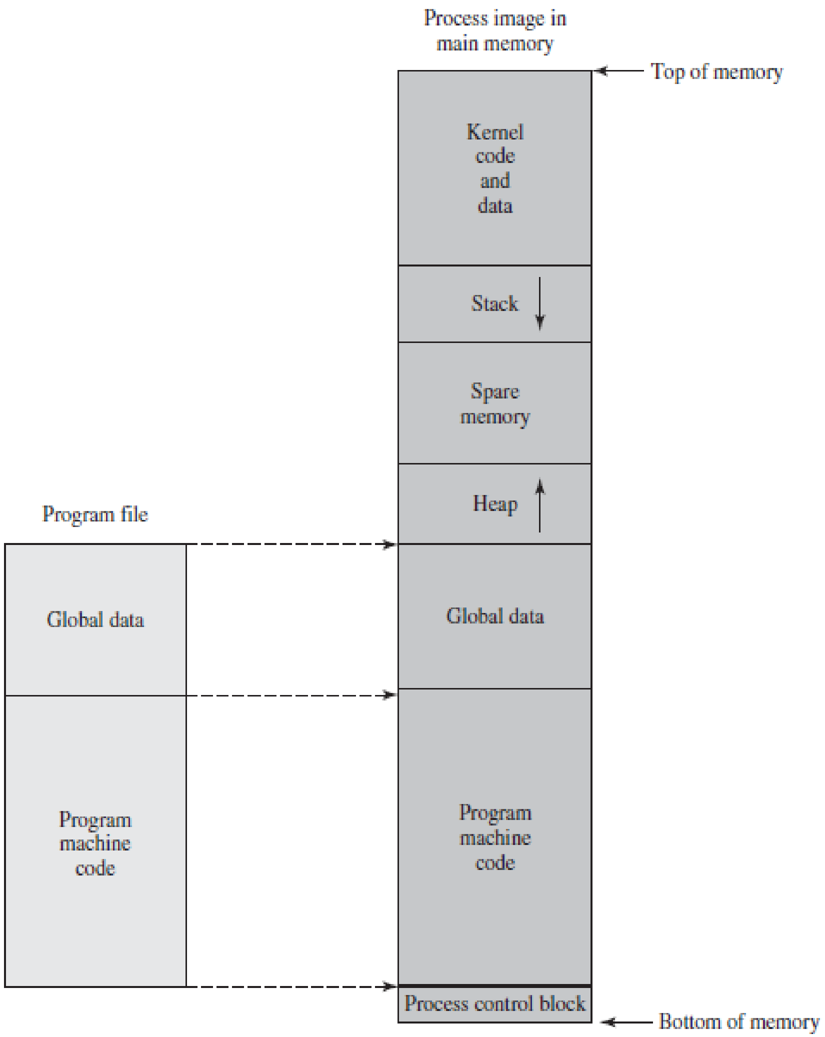
Handling Program Output

Principles of handling program outputs

- **P1** (conform expected form)**:** Output does conform to the expected form and interpretation.
- **P2** (validate third-party data)**:** Any programs that gather and rely on third-party data have to be responsible for ensuring that any subsequent use of such data is safe and does not violate the user's assumptions.
- **P3** (be careful with encoding)**:** Different character sets allow different encodings of meta characters, which may change the interpretation of what is valid output.

Interacting with Operating System

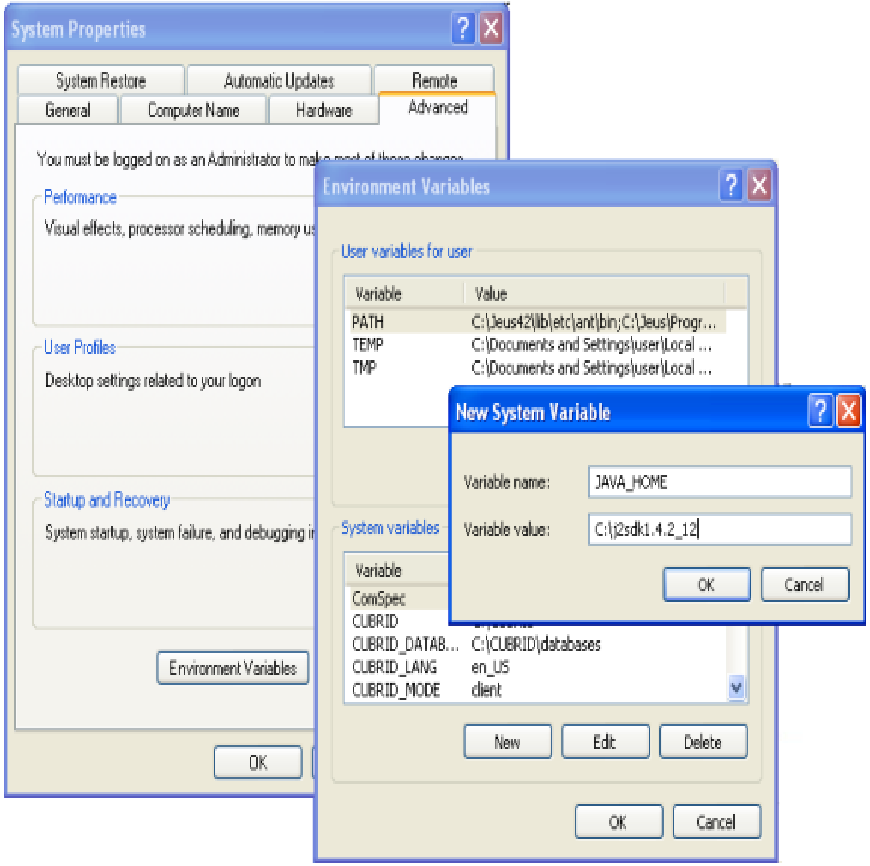
- Programs run under the control of Operating System
- Operating System
  - Mediates access to resources
  - Share the resources
  - Construct an executing environment
- Systems have multiple users with different access permissions
- Programs need to access shared resources that are significant for software security.



Interacting with Operating System

Environment Variables

- Collection of string values inherited by each process from its parent.
- The request to execute a new program can specify a new collection of values.
- The variables provide untrusted input to programs.
- Privileged shell scripts are targeted -> difficult to write safe and correct scripts



Interacting with Operating System

- Well known environment variables: PATH and LD_LIBRARY_PATH.
- Can be used to attack the system.
- Example

```
#!/bin/bash
user=`echo $1    |sed 's/@.*$//'`
grep $user /var/local/accounts/ipaddrs
```

Takes identity of a user, strips any domain specification if included, and then retrieves mapping for that user to an IP address.

Calls two separate programs: sed and grep.

Attacker has to redefine PATH variable to include a directory they control. Then when this script is run, attacker's program is called instead of standard system version.

Interacting with Operating System

- Example (Continue)

```
#!/bin/bash
PATH="/sbin:/bin:/usr/sbin:/usr/bin"
export PATH
user=`echo $1   |sed 's/@.*$//'`
grep $user /var/local/accounts/ipaddrs
```

Previous attack is prevented but another attack is possible!

Assignment of new value to PATH variable is interpreted as a command to execute program PATH with list of directories as its argument. If attacker has changed PATH variable to include directory with an attack program PATH, then this will be executed when script is run.

To prevent, use a compiled wrapper program. If program executes another program, it is still vulnerable against attacks regarding PATH environment!

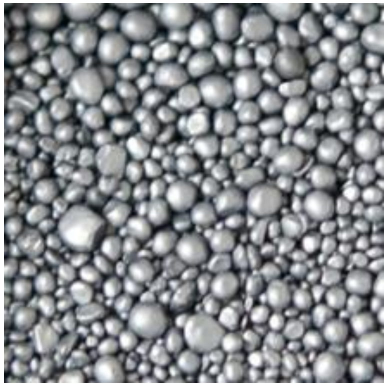
Interacting with Operating System

Using Least Privileges

- Consequence of some flaws is that attacker can execute code with privileges and access rights of compromised program or service.
- If privileges are greater than those available already to attacker, then this results in privilege escalation. Significant step in an attack.
- Normally when a user runs a program, it executes with the same privileges and access rights as that user.
- Programs should execute with the least amount of privileges needed to complete their functions, which is known as the principle of least privileges.

Interacting with Operating System

- A common deficiency found with many privileged programs is to have ownership of all associated files and directories.
  - This violates the principle of least privilege.
  - Any privileged program have to modify only those files and directories necessary.

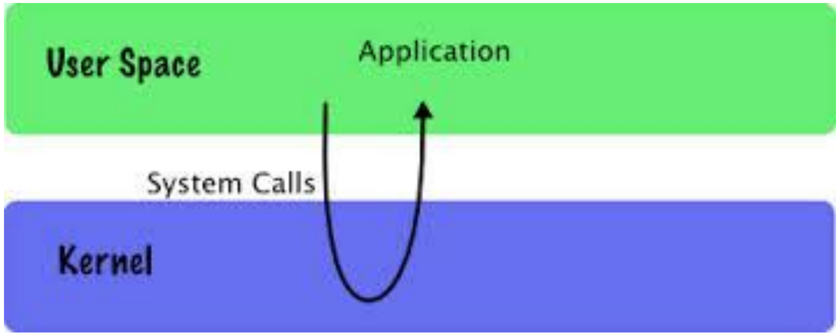


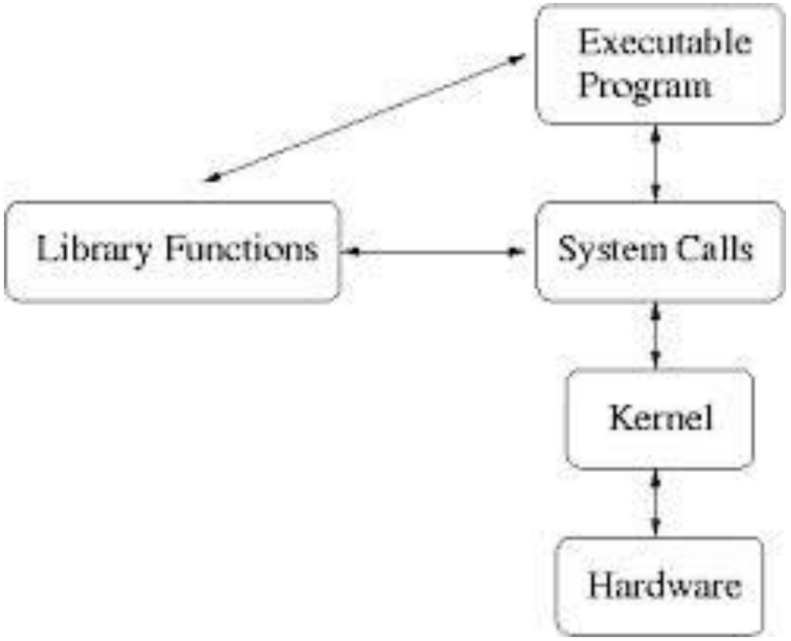
- Good defensive program requires to be partitioned into smaller modules, each granted privilege they require, only when they need.

Interacting with Operating System

Systems Calls and Standard Library Functions

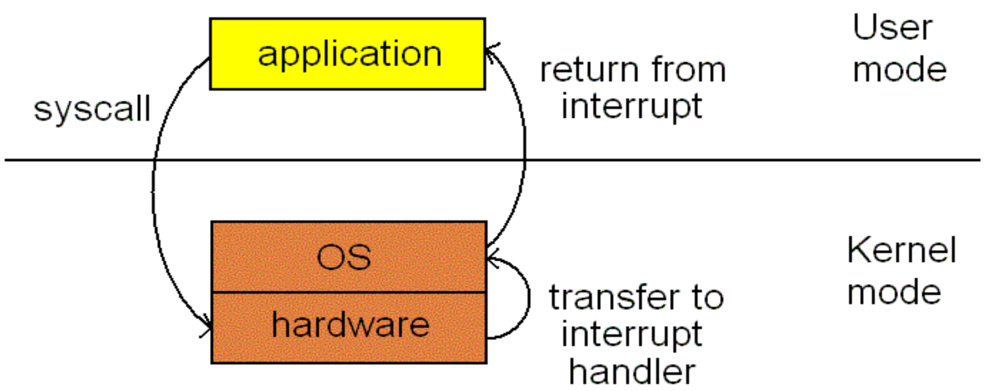
- Except some very small systems, no computer program contains all of code it needs to execute.
- Programs use system calls and standard library functions for common operations.





Interacting with Operating System

- When using these calls and functions, programmers commonly make assumptions about how they actually operate.
- OS and library functions attempt to manage their resources in a manner that provides the best performance to all the programs running on the system. Thus, requests for services
  - Buffered,
  - Resequenced,
  - Modified



- BUT, these optimizations may conflict with goals of the program.

Interacting with Operating System

Example

```
patterns = [10101010, 01010101, 11001100, 00110011, 00000000, 11111111,
...]
open file for writing
for each pattern
    seek to start of file
    overwrite file contents with pattern
close file
remove file
```

(a) Initial secure file shredding program algorithm

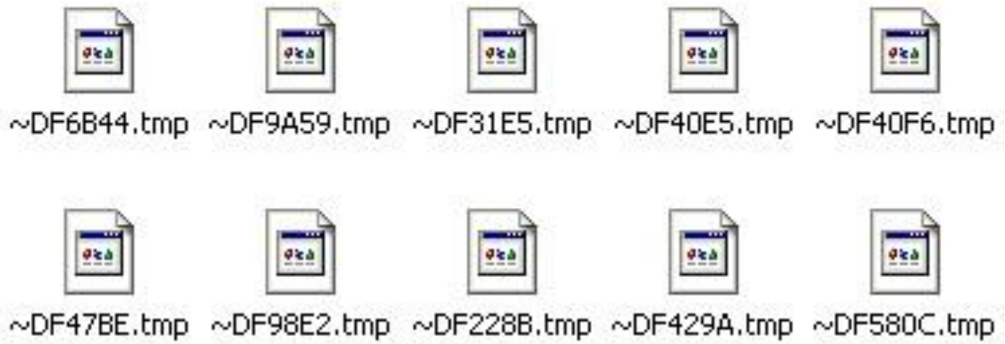
```
patterns = [10101010, 01010101, 11001100, 00110011, 00000000, 11111111,
...]
open file for update
for each pattern
    seek to start of file
    overwrite file contents with pattern
    flush application write buffers
    sync file system write buffers with device
close file
remove file
```

(b) Better secure file shredding program algorithm

Interacting with Operating System

Temporary File Use

- Many programs use temporary files often in common area.
- Most operating systems provide well-known locations for placing temporary files and standard functions for naming and creating them.
- The critical issue is that they are unique and not accessed by other processes.



Interacting with Operating System

- An attacker attempt to guess the file that a privileged program will use -> denial of service attack.
- Secure temporary file creation and use requires the use of a random temporary file name.
  - The creation of temporary file should be done using an atomic system primitive.
  - This prevents the race condition and hence the potential exploit of this file.
- The standard C function mkstemp() is suitable; however, the older functions tmpfile(), tmpnam(), and tempnam() are all insecure unless used with care.

Writing Safe Program Code

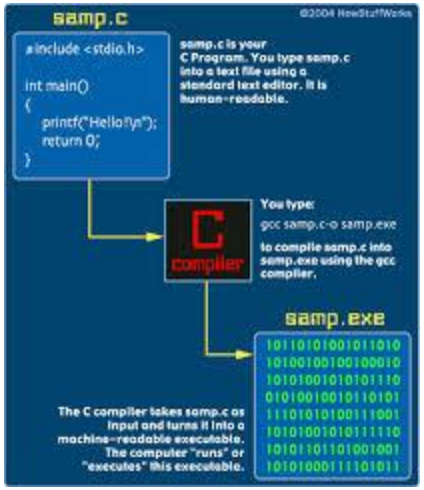
Writing secure, safe code requires

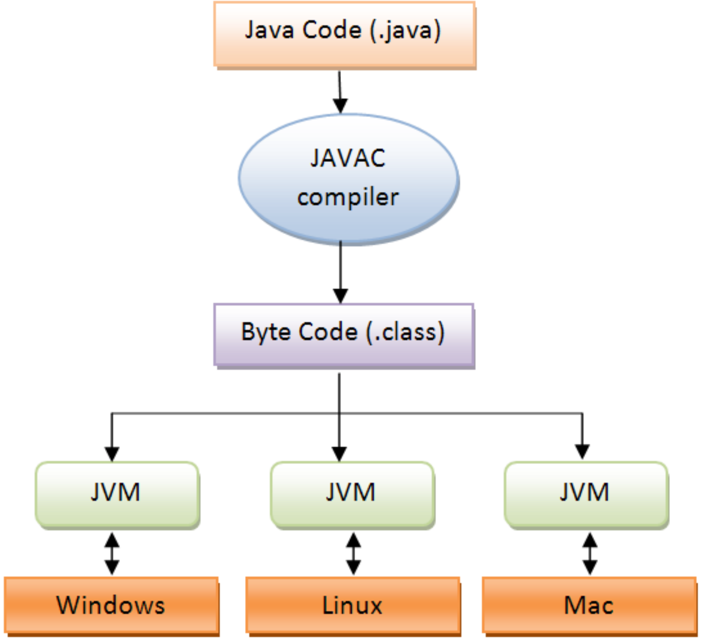
- Attention to all aspects of how a program executes,
- The environment it executes in,
- The type of data it process,
- Noting can be assumed,
- All potential errors must be checked.
- Known as defensive programming or secure programming

Writing Safe Program Code

High-level languages:

- Compiled->linked into machine code -> executed (C)
- Compiled -> intermediate language -> interpreted by suitable program (JAVA)





Writing Safe Program Code

The key issues (software security perspective):

- Whether the implemented algorithm correctly solves the specified problem,
- Whether the machine instructions executed correctly represent the high-level algorithm specification,
- Whether the manipulation of data values in variables is valid and meaningful.

Writing Safe Program Code

Correct Algorithm Implementation

- Good program development technique is significant for software security.
- The consequence of a deficiency in the design or implementation of the algorithm is a bug in the program that could be exploited, such as TCP session spoof or hijack attack.

Writing Safe Program Code

- The implementation flaws permits some attacks, such as the initial sequence numbers used by many TCP/IP implementations are predictable.
- If an interpreter does not correctly implement the specified code, such as incorrect Java Virtual Machine interpretation, it could result in bugs that an attacker might exploit.

Writing Safe Program Code

Correct Machine Language

- Ensures machine instructions are correctly implemented for high-level language code
  – Problem1: Often ignored by developers
  – Problem2: Assume compiler or interpreter work correctly
- Requires comparing machine code with original source code that is slow and difficult

Writing Safe Program Code

- The development of trusted computer systems with very high assurance level is the one area.

- Common Criteria assurance level of EAL 7 requires validation of correspondence among design, source code, and object code.

Writing Safe Program Code

Correct Interpretation of Data Values

- All data on a computer are stored as groups of binary bits.
- Interpretation depends on
  - Program operations used
  - Specific machine instructions used
- Languages provides different capabilities for restricting or validating data use
  - Strongly typed languages are more limited but safer
  - Others are flexible but less secure, such as C

Writing Safe Program Code

Correct Use of Memory

- In many applications, memory must be allocated when needed and released (dynamic memory allocation).
- Memory leak: If a program fails to correctly manage the memory, available memory on the heap is exhausted.
- An attacker can implement a denial of service attack by using memory leaks of targeted program.
- Many older languages, like C, have no explicit support for dynamic memory allocation.
- Modern languages like Java and C++ handle dynamic allocation automatically.

Writing Safe Program Code

An Example for Memory Leak

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

#define  LOOPS    10
#define  MAXSIZE  256

int main(int argc, char **argv)
{
     int count = 0;
     char *pointer = NULL;

     for(count=0; count<LOOPS; count++) {
          pointer = (char *)malloc(sizeof(char) * MAXSIZE);
     }

     free(pointer);

     return count;
}
```

Writing Safe Program Code

An Example for Memory Leak

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

#define  LOOPS    10
#define  MAXSIZE  256

int main(int argc, char **argv)
{
     int count = 0;
     char *pointer = NULL;

     for(count=0; count<LOOPS; count++) {
          pointer = (char *)malloc(sizeof(char) * MAXSIZE);
     }

     free(pointer);

     return count;
}
```

We have 10 allocations of size MAXSIZE.

Every allocation, with the exception of the last, is lost.

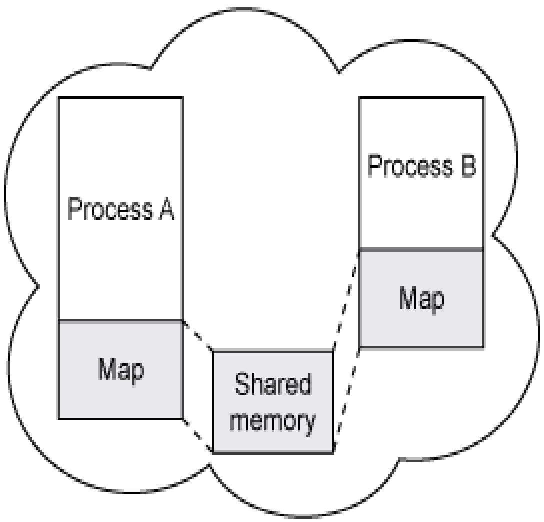
If no pointer is pointed to the allocated block, it is unrecoverable during program execution.

A simple fix to this trivial example is to place the free() call inside of the 'for' loop.

Writing Safe Program Code

Race Conditions and Shared Memory

- Shared memory is the memory where multiple process or threads can access.
- Race condition occurs when multiple processes and threads compete to gain uncontrolled access to some resources.
  - Needs synchronization primitives to solve race conditions.
- Incorrect synchronization leads to deadlock, where each process waits another for a resource.
- Denial of service attack is possible if deadlock conditions are known by attackers.



Summary

- Buffer overflow
- Some buffer overflow attacks
- Defenses
- Basics of software security
- Handling program input and output
- Interacting with OS
- Writing safe program code