SEZG566/SSZG566

Secure Software Engineering

Vulnerabilities in Code

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

T V Rao

DISCLAIMERS

- *The slides presented here are obtained from the authors of the books, product documentations, and from various other contributors. I hereby acknowledge all the contributors for their material and inputs.*
- *I have added and modified slides to suit the requirements of the course.*

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Memory Safety

Memory Safety

Memory safety is the state of being protected from various software bugs and security vulnerabilities when dealing with memory access

Java is memory-safe because of its runtime error detection checks for array bounds and pointer dereferences

C and C++ are not memory-safe, since they allow pointer arithmetic with pointers implemented as direct memory addresses with no provision for bounds checking

Memory Errors

A memory error occurs when an object accessed using a pointer expression is different from the one intended

We can categorize memory errors as

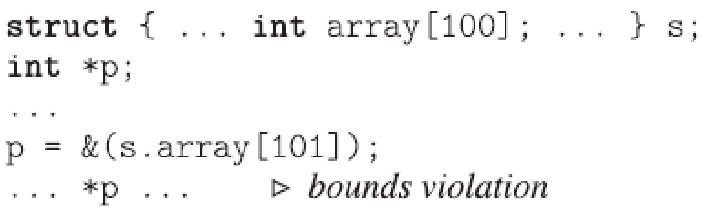
- Spatial Memory Errors
- Temporal Memory Errors

Spatial Memory Errors

A spatial memory error occurs when a pointer pointing outside the bound of its referent is dereferenced

Spatial memory errors include

- Dereferences of uninitialized pointers and non-pointer data
- Valid pointers used with invalid pointer arithmetic where buffer overflows



```
struct { ... int array[100]; ... } s;
int *p;
...
p = &(s.array[101]);
... *p ...        ▷ bounds violation
```

Temporal Memory Errors

A temporal memory error occurs when a the program dereferences a pointer to an object that no longer exists

Spatial memory errors include

- Dangling pointers
- Double frees

```
int *p = malloc(sizeof(int));
*p = 23;
free(p);
printf(“%d\n”,*p); //temporal violation
```

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Buffer Overflows

NIST’s Definition

“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.”

Buffer Overflow: A Well-Known Problem

- A very common attack mechanism
  - from 1988 Morris Worm to Code Red, Slammer, Sasser and many others
- Prevention techniques known
- Still of major concern due to
  - legacy of widely deployed buggy code
  - continued careless programming techniques

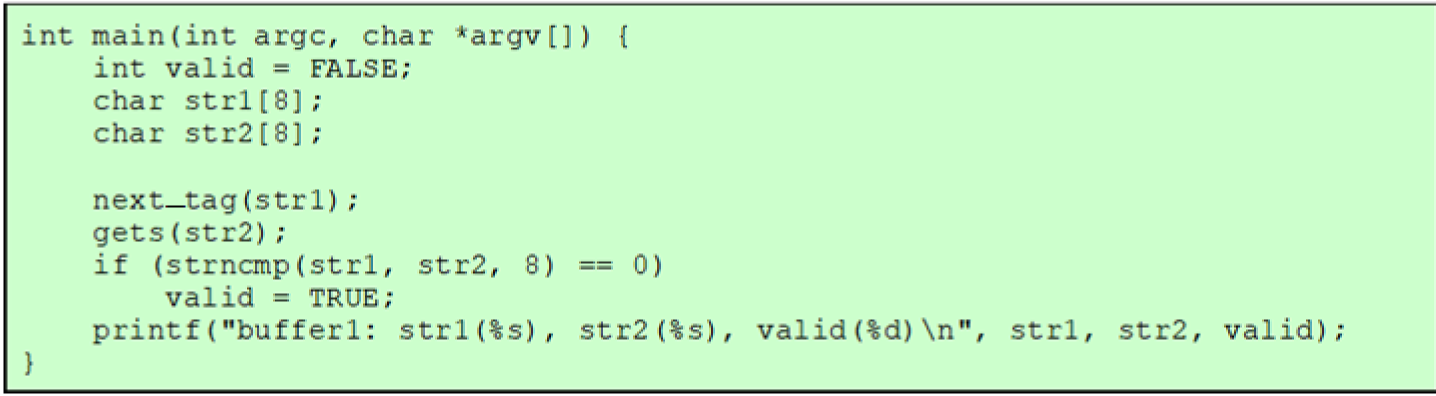
Morris worm

- One of best known worms
  - Affected 6,000 computers in 1988; cost $10-$100 M
- Released by Robert Morris
  - Graduate student at Cornell, son of NSA chief scientist
  - Convicted under Computer Fraud and Abuse Act, sentenced to 3 years of probation and 400 hours of community service
  - Now a computer science professor at MIT
- Worm was intended to propagate slowly and harmlessly measure the size of the Internet. Due to a coding error, it created new copies as fast as it could and overloaded infected machines
- The worm propagated thru buffer overflow attack against a vulnerable version of fingerd on VAX system

Buffer Overflow Basics

- Caused by programming error
- Allows more data to be stored than capacity available in a fixed sized buffer
  - buffer can be on stack, heap, global data
  - Overwriting adjacent memory locations
    - corruption of program data
    - unexpected transfer of control
    - memory access violation
    - execution of code chosen by attacker

Buffer Overflow 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);
}
```

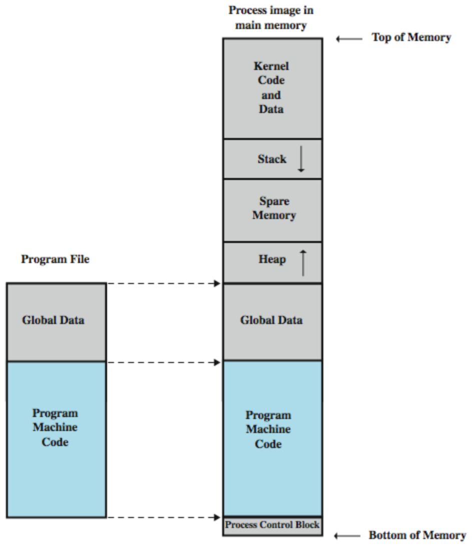
Buffer Overflow - Program Output

```
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);
}
```

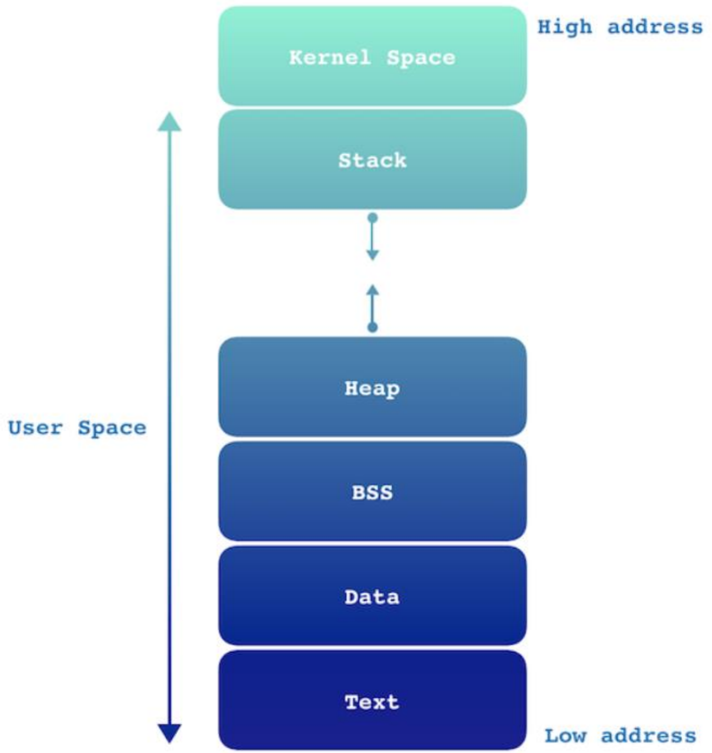
```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

Process in Memory

- Processes are divided into three regions: Text, Data, and Stack.
- The text region includes code (instructions) and read-only data. This region is normally marked read-only and any attempt to write to it will result in a segmentation violation
- The data region contains initialized and uninitialized data. Static variables are stored in this region.
- A procedure call alters the flow of control, when finished performing its task, a function returns control to the statement or instruction following the call. This high-level abstraction is implemented with the help of the stack.
  - The stack is used to dynamically allocate the local variables used in functions, to pass parameters to the functions, and to return values from the function.



Layout in Unix-like OS



Stack

The stack space is located just under the OS kernel space, generally opposite the heap area and grows downwards to lower addresses.

Heap

The Heap is the segment where dynamic memory allocation usually takes place. This area grows upwards to higher memory addresses

BSS (Block Started by Symbol)

Uninitialized data segment

Data

The data segment contains initialized global and static variables

Text

A segment in which a machine language instruction is stored. This segment is a read-only space.

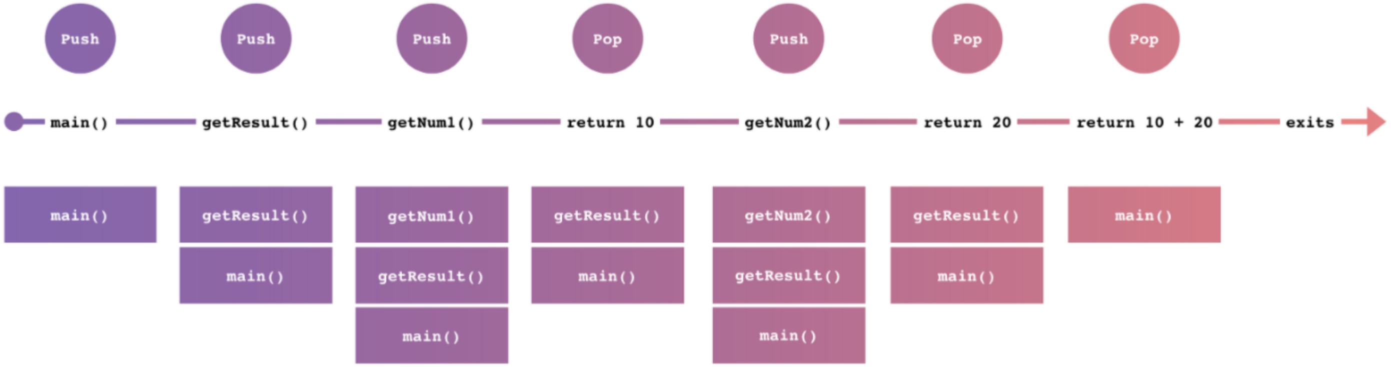
https://medium.com/@shoheiyokoyama/understanding-memory-layout-4ef452c2e709

Consider a simple program

```
int main() {
  int result = getResult();
}

int getResult() {
  int num1 = getNum1();
  int num2 = getNum2();
  return num1 + num2;
}

int getNum1() {
  return 10;
}

int getNum2() {
  return 20;
}
```

https://medium.com/@shoheiyokoyama/understanding-memory-layout-4ef452c2e709

Sequence of Stack contents



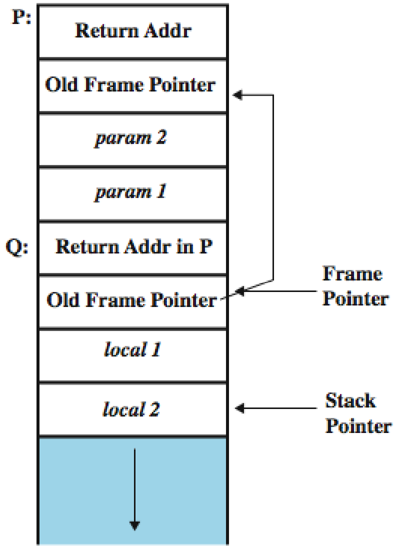
https://medium.com/@shoheiyokoyama/understanding-memory-layout-4ef452c2e709

Stack Frame Structure

The general process of one function P calling another function Q can be summarized as follows.
The calling function P
1. Pushes the parameters for the called function onto the stack (typically in reverse order of declaration)
2. Executes the call instruction to call the target function, which pushes the return address onto the stack

The called function Q
3. Pushes the current frame pointer value (which points to the calling routine's stack frame) onto the stack
4. Sets the frame pointer to be the current stack pointer value (that is the address of the old frame pointer), which now identifies the new stack frame location for the called function
5. Allocates space for local variables by moving the stack pointer down to leave sufficient room for them
6. Runs the body of the called function
7. As it exits it first sets the stack pointer back to the value of the frame pointer (effectively discarding the space used by local variables)
8. Pops the old frame pointer value (restoring the link to the calling routine's stack frame)
9. Executes the return instruction which pops the saved address off the stack and returns control to the calling function

Lastly, the calling function
10. Pops the parameters for the called function off the stack
11. Continues execution with the instruction following the function call.



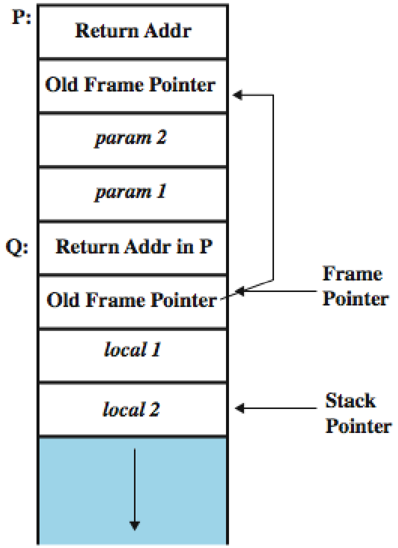
Exploiting the Buffer-Overflow

To fully exploit a stack buffer-overflow vulnerability,

- Inject the malicious code: need to be able to inject the malicious code into the memory of the target process. This can be done if attacker can control the contents of the buffer in the targeted program.
- Jump to the malicious code: With the malicious code already in the memory, if the targeted program can jump to the starting point of the malicious code, the attacker will be in control.

Stack frame:

- *Calling function*: needs a data structure to store the "return" address and parameters to be passed
- *Called function*: needs a place to store its local variables somewhere different for every call



Vulnerable Program

```
void function(int a, int b, int c) {
char buffer[20];
gets(buffer);
}

void main() {
  function(1,2,3);
  ….
  do more
  …
}
```

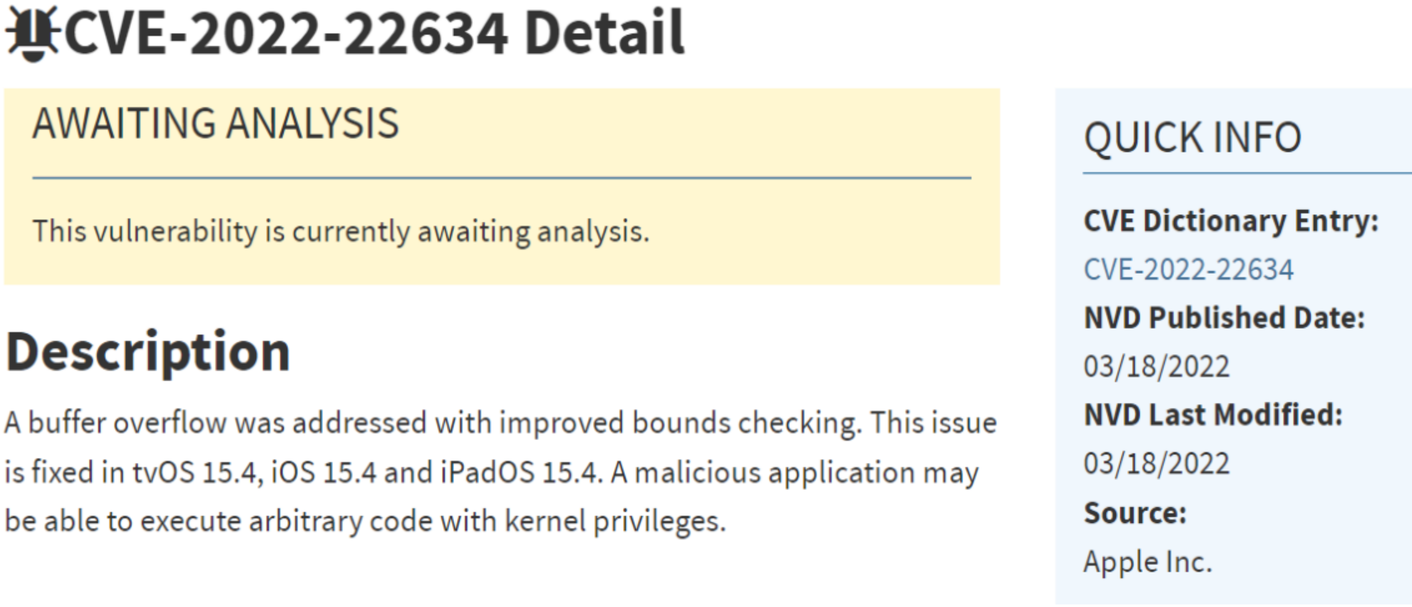
- Assuming the stack starts at address 0xFF, and that S stands for the code attackers want to execute the stack should then look like this:

```
bottom of   DDDDDDDDEEEEEEEEEEEE   EEEE   FFFF   FFFF   FFFF   FFFF      top of
memory      89ABCDEF0123456789AB   CDEF   0123   4567   89AB   CDEF      memory
            buffer                 sfp    ret    a      b      c

<------    [SSSSSSSSSSSSSSSSSSSS][SSSS][0xD8][0x01][0x02][0x03]
            ^                             |
            |_____________________________|
top of                                                                   bottom of
stack                                                                        stack
```

Arc Injection (return-into-libc)

- Arc injection transfers control to code that already exists in the program’s memory space
- refers to how exploits insert a new arc (control-flow transfer) into the program’s control-flow graph as opposed to injecting code.
- can install the address of an existing function (such as **system()** or **exec()**, which can be used to execute programs on the local system
- Sophisticated attacks possible using this technique
- “Exploit” code pre-installed in code segment; No code is injected
- Memory based protection schemes cannot prevent arc injection
- Does not require larger overflows
- The original frame can be restored to prevent detection

Buffer Overflows persist



CVE-2022-22634 Detail

AWAITING ANALYSIS

This vulnerability is currently awaiting analysis.

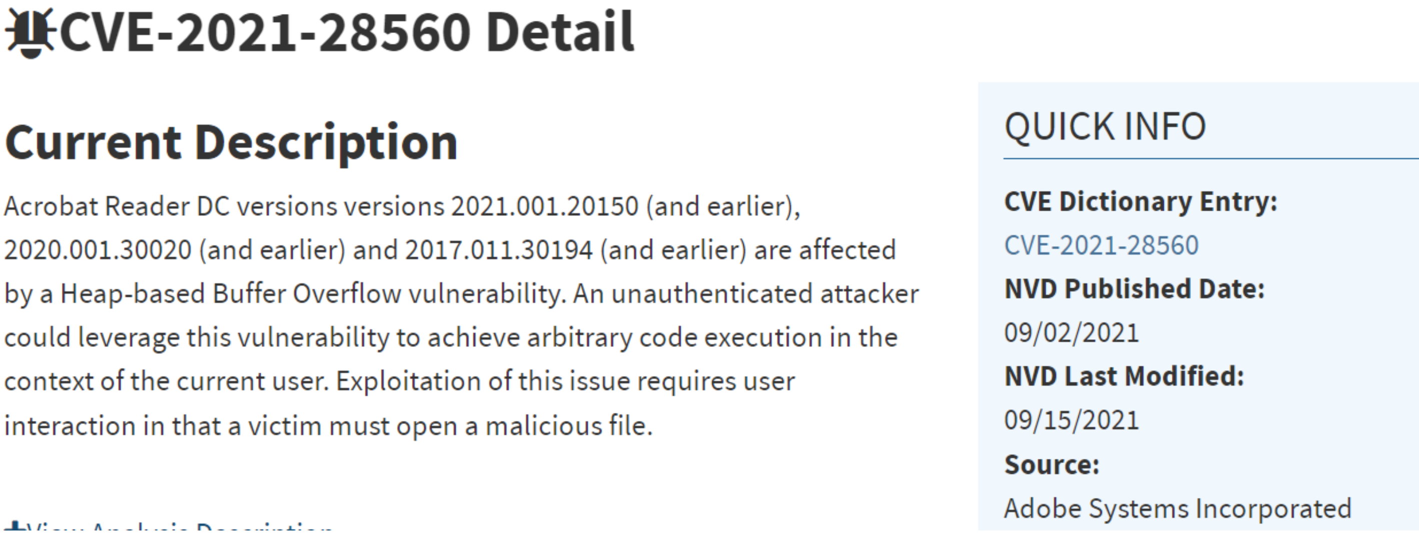
Description

A buffer overflow was addressed with improved bounds checking. This issue is fixed in tvOS 15.4, iOS 15.4 and iPadOS 15.4. A malicious application may be able to execute arbitrary code with kernel privileges.

QUICK INFO

CVE Dictionary Entry:
CVE-2022-22634
NVD Published Date:
03/18/2022
NVD Last Modified:
03/18/2022
Source:
Apple Inc.

Buffer Overflows persist



CVE-2021-28560 Detail

Current Description

Acrobat Reader DC versions versions 2021.001.20150 (and earlier), 2020.001.30020 (and earlier) and 2017.011.30194 (and earlier) are affected by a Heap-based Buffer Overflow vulnerability. An unauthenticated attacker could leverage this vulnerability to achieve arbitrary code execution in the context of the current user. Exploitation of this issue requires user interaction in that a victim must open a malicious file.

QUICK INFO

CVE Dictionary Entry:
CVE-2021-28560
NVD Published Date:
09/02/2021
NVD Last Modified:
09/15/2021
Source:
Adobe Systems Incorporated

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Buffer Overflow Defenses

Buffer Overflow Defenses

Buffer overflows are widely exploited

Large amount of vulnerable code in use

- despite cause and countermeasures known

Two broad defense approaches

- compile-time - harden new programs
- run-time - handle attacks on existing programs

Compile-Time Defenses

- Aim to prevent or detect buffer overflows
- Possibilities include
  - Choose a high-level language that does not permit buffer overflows
  - Encourage safe coding standards
  - Use safe standard libraries
  - Include additional code to detect corruption of the stack frame

Compile-Time Defenses: Programming Language

- Use a modern high-level languages with strong typing
  - not vulnerable to buffer overflow
  - compiler enforces range checks and permissible operations on variables
- Flexibility & Safety come with cost in resource use
  - at compile time
  - additional checks at run time
- Add restrictions on access to hardware
  - still need some code(e.g. device drivers) in C like languages

Compile-Time Defenses: Safe Coding Techniques

If possible, avoid using potentially unsafe languages e.g. C

Programmer must explicitly write safe code

– by design with new code

– ***extensive after code review*** of existing code, (e.g., OpenBSD)

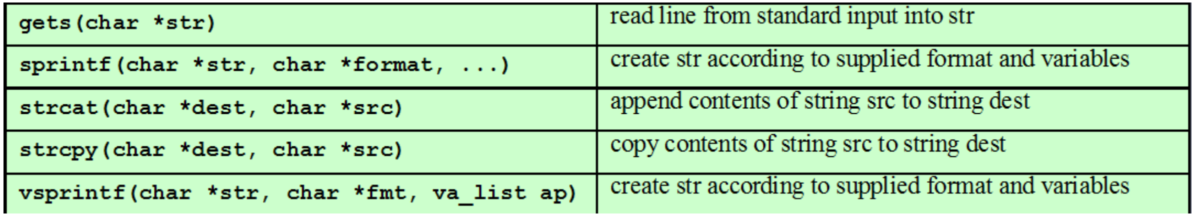
Buffer overflow safety a subset of general safe coding techniques

Allow for graceful failure ***(know how things may go wrong)***

– check for sufficient space in any buffer

Compile-Time Defenses: Safe Coding Techniques

Common Unsafe C Functions



`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

Compile-Time Defenses: Language Extension, Safe Libraries

Proposals for safety extensions (library replacements) to C

- performance penalties
- must compile programs with special compiler

Several safer standard library variants

- new functions, e.g. strlcpy()
- safer re-implementation of standard functions as a dynamic library, e.g. Libsafe

C String Library (SafeStr)

- The C String Library (SafeStr) from Messier and Viega provides a rich string-handling library for C that has secure semantics yet is interoperable with legacy library code in a straightforward manner
  - The SafeStr library uses a dynamic approach for C that automatically resizes strings as required.
  - SafeStr accomplishes this by reallocating memory and moving the contents of the string whenever an operation requires that a string grow in size.
  - As a result, buffer overflows should not result from using the library
  - The SafeStr library uses a dynamic approach for C that automatically resizes strings as required. SafeStr accomplishes this by reallocating memory and moving the contents of the string whenever an operation requires that a string grow in size. As a result, buffer overflows should not result from using the library

Compile-Time Defenses: Stack Protection

- Stackguard: add function entry and exit code to check stack for signs of corruption
  - Use random canary
  - e.g. Stackguard, Win/GS, GCC
  - check for overwrite between local variables and saved frame pointer and return address
  - abort program if change found
  - issues: recompilation, debugger support
- Or save/check safe copy of return address (in a safe, non-corruptible memory area), e.g. Stackshield, RAD (Return Address Defender)

Run-Time Defenses: Non Executable Address Space

- Many BO attacks copy machine code into buffer and xfer ctrl to it
- Use virtual memory support to make some regions of memory non-executable (to avoid exec of attacker's code)
  - e.g. stack, heap, global data
  - need h/w support in MMU (memory management unit)
  - long existed on SPARC/Solaris systems
  - recent on x86 Linux/Unix/Windows systems
- Issues: support for executable stack code

Run-Time Defenses: Address Space Randomization

Manipulate location of key data structures

- stack, heap, global data: change address by 1 MB
- using random shift for each process
- have large address range on modern systems means wasting some has negligible impact

Randomize location of heap buffers and location of standard library functions

Run-Time Defenses: Guard Pages

- Place guard pages between critical regions of memory (or between stack frames)
  - flagged in MMU (mem mgmt unit) as illegal addresses
  - any access aborts process
- Can even place between stack frames and heap buffers
  - at execution time and space cost

Source Code Analysis Tools

Source Code Analysis Tools (security analyzers) are automated tools for helping analysts find security-related problems in software

- use data- and control-flow analysis to find subtler bugs and to reduce false alarms

Some vulnerabilities (e.g. use of strcpy()) can be detected with high accuracy, others are harder to detect, and, in fact, one can always devise vulnerabilities that are undetectable altogether.

Tools have tradeoff between false alarms (also known as false positives) and missed vulnerabilities (also known as false negatives)

- can be configured to make a tool more sensitive (decreasing false negatives while increasing false positives) or make it less sensitive (increasing false negatives while decreasing false positives)

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Heap & Integer Vulnerabilities

Heap Overflow

- Possible to attack buffer located in heap
  - typically located above program code and global data and grows up in memory (while stack grows down towards it)
  - memory requested by programs to use in dynamic data structures, e.g. linked lists
- No return address
  - hence no easy transfer of control
- May have function pointers that can be exploited
  - Typically for custom processing of data, e.g. decoding a compressed image
  - or manipulate management data structures
- Defenses: non executable or random heap

Heap Overflow Example

```
/* record type to allocate on heap */
typedef struct chunk {
   char inp[64];                   /* vulnerable input buffer */
   void (*process)(char *);      /* pointer to function */
} chunk_t;

void showlen(char *buf) {
   int len; len = strlen(buf);
   printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[]) {
   chunk_t *next;
   setbuf(stdin, NULL);
   next = malloc(sizeof(chunk_t));
   next->process = showlen;
   printf("Enter value: ");
   gets(next->inp);
   next->process(next->inp);
   printf("buffer5 done\n");
}
```

Integer Security

Integers represent a source of vulnerabilities in C and C++ programs.

Integer range checking has not been systematically done in many C and C++ software

- Security flaws involving integers exist
- Some of these are likely to be vulnerabilities

Integers in C and C++ are either signed or unsigned.

- For each signed type there is an equivalent unsigned type.
- Signed integers are used to represent positive and negative values.
  - On a computer using two's complement arithmetic, a signed integer ranges from -2^{n-1} through 2^{n-1}-1.
- Unsigned integer values range from zero to a maximum
  - This maximum value can be calculated as 2^{n-1}, where n is the number of bits used to represent the unsigned type.

Integer Conversions

From unsigned	To	Method
char	char	Preserve bit pattern; high-order bit becomes sign bit
char	short	Zero-extend
char	long	Zero-extend
char	unsigned short	Zero-extend
char	unsigned long	Zero-extend
short	char	Preserve low-order byte
short	short	Preserve bit pattern; high-order bit becomes sign bit
short	long	Zero-extend
short	unsigned char	Preserve low-order byte
long	char	Preserve low-order byte
long	short	Preserve low-order word
long	long	Preserve bit pattern; high-order bit becomes sign bit
long	unsigned char	Preserve low-order byte
long	unsigned short	Preserve low-order word

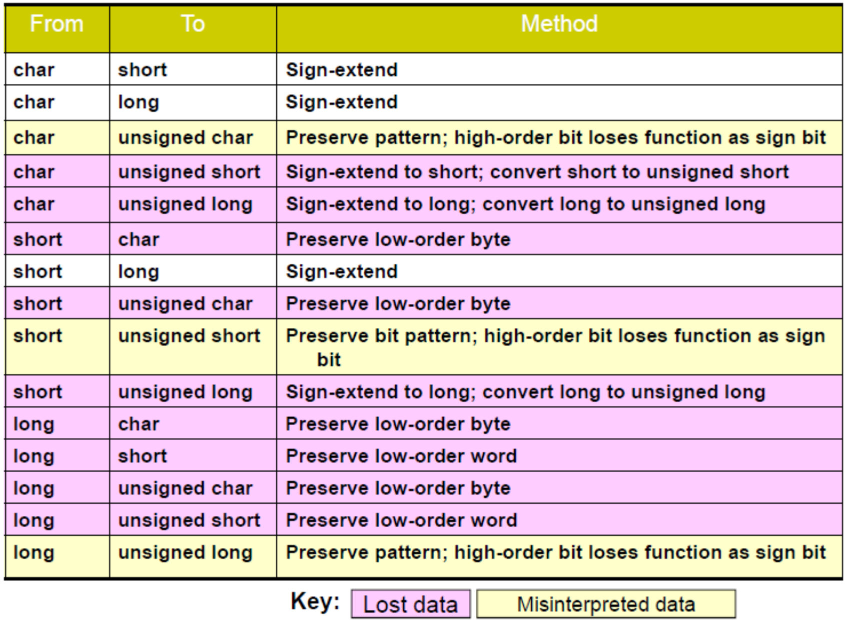
Key: Lost data | Misinterpreted data

Type conversions may occur in C and C++

- explicitly as a cast or
- implicitly as C language can perform operations on mixed types.

Conversions can lead to lost or misinterpreted data.

Integer Conversions



From	To	Method
char	short	Sign-extend
char	long	Sign-extend
char	unsigned char	Preserve pattern; high-order bit loses function as sign bit
char	unsigned short	Sign-extend to short; convert short to unsigned short
char	unsigned long	Sign-extend to long; convert long to unsigned long
short	char	Preserve low-order byte
short	long	Sign-extend
short	unsigned char	Preserve low-order byte
short	unsigned short	Preserve bit pattern; high-order bit loses function as sign bit
short	unsigned long	Sign-extend to long; convert long to unsigned long
long	char	Preserve low-order byte
long	short	Preserve low-order word
long	unsigned char	Preserve low-order byte
long	unsigned short	Preserve low-order word
long	unsigned long	Preserve pattern; high-order bit loses function as sign bit

Key: 🟪 Lost data 🟨 Misinterpreted data

Conversion anomaly

```
unsigned int n = ULONG_MAX;

char c = -1;

if (c == n) {

    printf("-1 = 4,294,967,295 ?\n");

}
```

```
#define BUFF_SIZE 10
int main(int argc, char* argv[]){
int len;
char buf[BUFF_SIZE];
len = atoi(argv[1]);
if (len < BUFF_SIZE) {
    memcpy(buf, argv[2], len);
  }
}
```

SafeInt Class

- SafeInt is a C++ template class written by David LeBlanc.
- Implements a precondition approach that tests the values of operands before performing an operation to determine if an error will occur.
- The class is declared as a template, so it can be used with any integer type.
- Every operator has been overridden except for the subscript **operator[]**

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Format String Vulnerabilities

Format Strings

- Printf (stands for "print formatted") format string are control parameter used by a class of functions in the string-processing libraries.
- The format string is written in a simple template language, and specifies a method for rendering an arbitrary number of varied data type parameters into a string
- e.g.
  - printf ("a has value %d, b has value %d, c is at address: %08x\n", a, b, &c);

https://owasp.org/www-community/attacks/Format_string_attack#
http://www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf

Format Strings Abuse

- Consider e.g.
  - printf ("a has value %d, b has value %d, c is at address: %08x\n", a, b);
- Here the format string asks for 3 arguments, but the program actually provides only two (i.e. *a* and *b*). This program passes the compiler.
  - The function printf() is defined as function with variable length of arguments. Therefore, by looking at the number of arguments, everything looks fine.
  - To find the miss-match, compilers needs to understand how printf() works and what the meaning of the format string is. However, compilers usually do not do this kind of analysis.
  - Sometimes, the format string is not a constant string, it is generated during the execution of the program. Therefore, there is no way for the compiler to find the miss-match in this case.

Format Strings Abuse

- The function printf() fetches the arguments from the stack. If the format string needs 3 arguments, it will fetch 3 data items from the stack.
- In a mis-match case, it will fetch some data that do not belong to this function call.
- Crashing the program
  - printf ("%s%s%s%s%s%s%s%s%s%s%s%s");
  - For each %s, printf() will fetch a number from the stack, treat this number as an address, and print out the memory contents pointed by this address as a string, until a NULL character (i.e., number 0, not character 0) is encountered.
  - Since the number fetched by printf() might not be an address, the memory pointed by this number might not exist, if so the program will crash.
  - It is also possible that the number happens to be a good address.

Format Strings Abuse

- Viewing the stack
  - printf ("%08x %08x %08x %08x %08x\n");
    - This instructs the printf-function to retrieve five parameters from the stack and display them as 8-digit padded hexadecimal numbers.
    - So a possible output may look like:
      - 40012980 080628c4 bffff7a4 00000005 08059c04

Consider the program

```
int main(int argc, char *argv[])
{ char user_input[100];
... ... /* other variable definitions and
statements */
scanf("%s", user_input); /* getting a string
from user */
printf(user_input); /* Vulnerable place */
return 0;
}
```

- If the attacker provides user input of "\x10\x01\x48\x08 %x %x %x %x %s" , the program will print contents at the address 0x10014808

Format Strings Abuse

- Writing an integer in the process memory
  - %n: The number of characters written so far is stored into the integer indicated by the corresponding argument. Consider the code that writes 5 to i:
    - int i;
    - printf ("12345%n", &i);

- Using the same approach as that for viewing memory at any location, we can cause printf() to write an integer into any location. Just replace the %s in the previous example with %n, and the contents at the address 0x10014808 will be overwritten.

Format Strings Abuse

- Using this attack, attackers can do the following:
  - Overwrite important program flags that control access privileges ∗ Overwrite return addresses on the stack, function pointers, etc.
    - However, the value written is determined by the number of characters printed before the %n is reached.
  - Is it really possible to write arbitrary integer values?
    - Use dummy output characters. To write a value of 1000, a simple padding of 1000 dummy characters would do.
  - To avoid long format strings, we can use a width specification of the format indicators.
- Countermeasures
  - Address randomization: just like the countermeasures used to protect against buffer-overflow attacks, address randomization makes it difficult for the attackers to find out what address they want to read/write

Computer Security: Principles and Practice by William Stallings, and Lawrie Brown Pearson, 2008.
Secure Coding in C and C++ by Robert C Seacord, Addison Wesley 2013
sei.cmu.edu/cert
www.owasp.com
www.cigital.com

Thank You!