CIS 418/518 – Secure Software Engineering
Software Coding – Buffer Overflow

Jagadeesh Nandigam

School of Computing
Grand Valley State University
nandigaj@gvsu.edu

Outline

1. What is a Buffer Overflow?
2. Memory Layout of a C Program
3. Stack and Heap-Based Buffer Overflows
4. Function Activation and Stack Frame Layout
5. Stack Buffer Overflow Example
6. Exploiting Stack Buffer Overflow Vulnerability
7. Buffer Allocation Strategies and Trade-Offs
8. String Buffer Overflow
9. Integer Overflow
10. Buffer Overflow: Countermeasures

What is a Buffer Overflow?

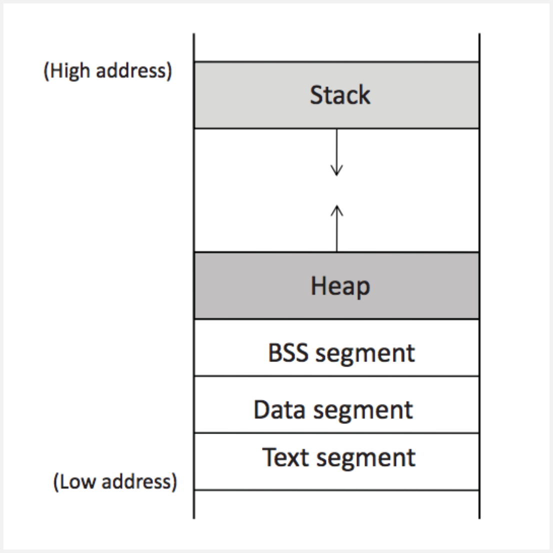
- A *buffer overflow* occurs when a program writes data outside the bounds of allocated memory.
- Use of unsafe languages (like C and C++) combined with poor programming practices is the root cause behind buffer overflow problems.
- Buffer overflow can cause erratic program behavior, memory access errors, incorrect results, program crashes, and execution of arbitrary pieces of code.
- Buffer overflow vulnerabilities can be exploited by a malicious user to overwrite data, alter the control flow of the program, or even execute malicious piece of code to create security problems.

What is a Buffer Overflow?

Let's compile and run this C code to see the potential buffer overflow problem:

```
#include <stdio.h>

int main() {
   int c = 3;
   int d = 4;
   char buf[6];

   printf("Enter some text: ");
   gets(buf); // can cause buf to overflow
   printf("%d %d %s\n", c, d, buf);

   return 0;
}
```

Memory Layout of a C Program



Memory Layout of a C Program

- **Text segment** stores the executable code of the program (read-only).
- **Data segment** stores static/global variables that are initialized by the programmer.
- **BSS segment** stores uninitialized static/global variables. This segment will be filled with zeros by the kernel. BSS stands for *Block Started by Symbol*.
- **Heap segment** is used to provide space for dynamic memory allocation.
- **Stack segment** is used for storing data related to function calls, such as arguments, return address, stack frame pointer, and local variables.
- The BSS, data, and text segments constitute static memory – the sizes of these segments are fixed before the program ever runs.

Memory Layout of a C Program

Determine which segments in the memory contain the values of `x`, `y`, `a`, `b`, `ptr`, `ptr[0]`, and `ptr[1]`?

```
int x = 100;

int main() {
   int a = 2;
   int b = 2.5;
   static int y;

   int *ptr = (int *) malloc(2 * sizeof(int));
   ptr[0] = 5;
   ptr[1] = 6;

   free(ptr);
   return 0;
}
```

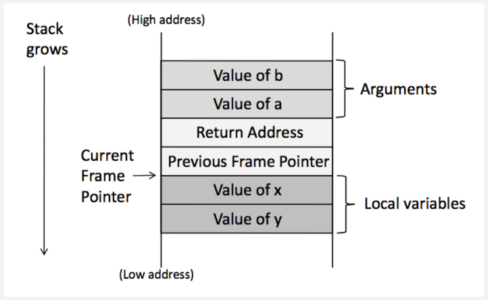
Stack and Heap-Based Buffer Overflows

- Buffer overflow can happen on both stack and heap.
- Heap overflows are harder to exploit for many reasons:
  - Hard to find out which region/variable on heap is security-critical
  - Have to identify a buffer that can overflow in such a way that it overwrites the target variable/region on heap.
- Stack overflows are not as difficult to exploit because there's always something critical to overwrite on the stack – *the return address*.

Function Activation and Stack Frame Layout

Stack frame layout for the following C function:

```
void func(int a, int b) {
   int x, y;

   x = a + b;
   y = a - b;
}
```

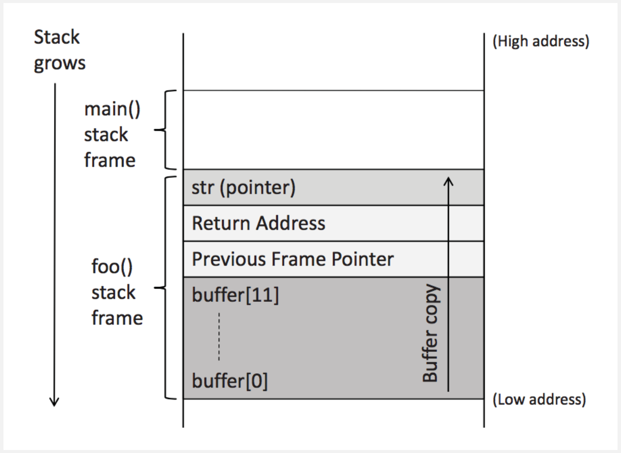


Stack Buffer Overflow Example

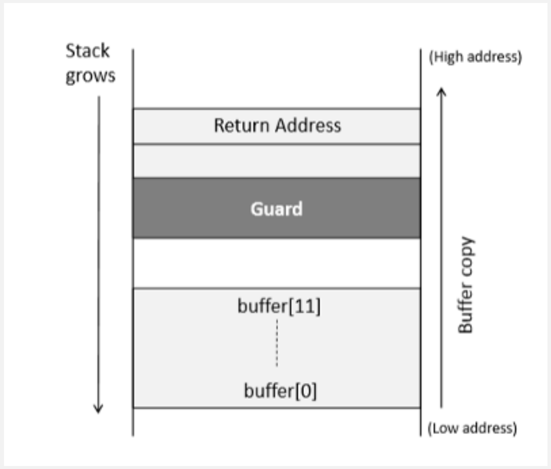
Let's see how the stack segment looks like for this program after the input string is copied to the *buffer* in function foo():

```
#include <string.h>

void foo(char *str) {
   char buffer[12];

   strcpy(buffer,str);
}

int main() {
   char *str = "This is definitely longer than 12";
   foo(str);
   return 1;
}
```

Stack Buffer Overflow Example



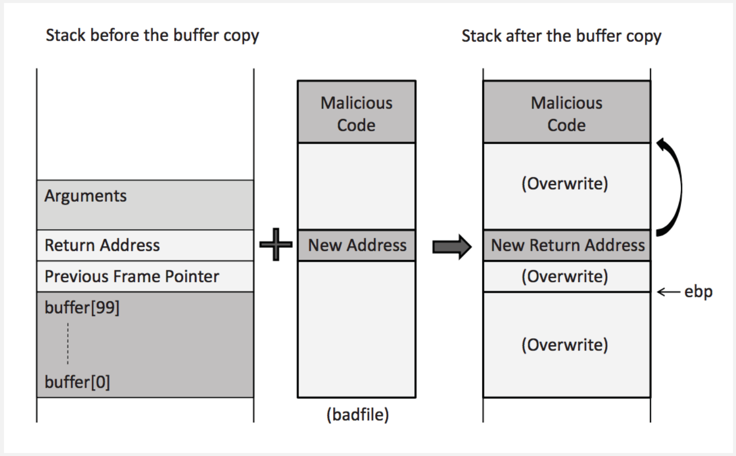
Buffer Overflow Detection with StackGuard



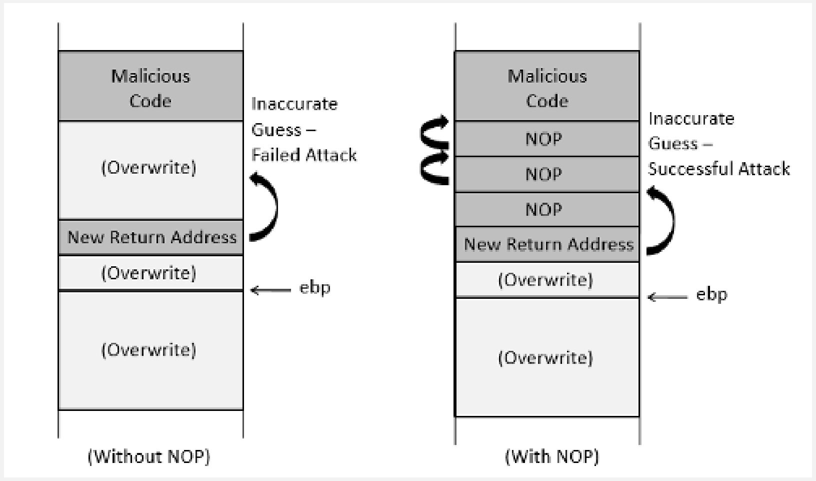
Stack Buffer Overflow Consequences

- If the return address field in the stack frame is modified due to a buffer overflow, several things can happen when the function returns:
  - New address may not map to any physical address – return fails and program crashes.
  - New address maps to a physical address, but the address space is protected – jump will fail and program crashes.
  - New address maps to a physical address, but the content of that location is not a valid machine instruction – return fails and program crashes.
  - New address maps to a physical address with a valid machine instruction and the program will continue running from that instruction, but the the logic of the program will be altered. This scenario gives an attacker to hijack the execution of the program.

Exploiting Stack Buffer Overflow Vulnerability



Exploiting Stack Buffer Overflow Vulnerability



Buffer Allocation Strategies and Trade-Offs

- Static Buffer Allocation
  - Memory for a buffer is allocated once and the buffer retains its initial size for the duration of its existence (within its scope).
  - Simplicity – simpler code surrounding buffer operations and facilitates both manual and automated code review.
  - Less flexibility when an operation requires larger buffer than is currently allocated – refuse to perform the operation or truncate the data and return an error/warning message.
  - Data truncation can introduce a variety of logic and representation errors that can be hard to track down.
  - Potential for wasted resources when the maximum capacity required for a buffer is much larger than the average capacity used.

Buffer Allocation Strategies and Trade-Offs

A simple program using static memory allocation:

```
int main(int argc, char **argv) {
  char str[BUFSIZE];
  int len;
  len = snprintf(str, BUFSIZE, "%s(%d)", argv[0], argc);
  printf("%s\n", str);
  if (len >= BUFSIZE) {
    printf("length truncated (from %d)\n", len);
  }
  return SUCCESS;
}
```

Buffer Allocation Strategies and Trade-Offs

- Dynamic Buffer Allocation
  - Dynamic buffer allocation approach allows for buffers to be resized according to runtime values as required by the program.
  - More flexibility with additional runtime complexity
  - Program's correctness is harder to verify because a runtime value controls the size of the dynamically allocated buffer.
  - Possible to exhaust memory resources with unexpected or malicious input.
  - Performing sanity checks is essential to ensure that an unreasonable amount of memory is not allocated in the process of resizing a buffer.

Buffer Allocation Strategies and Trade-Offs

Previous program rewritten using dynamic memory allocation:

```
int main(int argc, char **argv) {
  char *str;
  int len;
  if ((str = (char *)malloc(BUFSIZE)) == NULL) {
    return FAILURE_MEMORY;
  }
  len = snprintf(str, BUFSIZE, "%s(%d)", argv[0], argc);
  if (len >= BUFSIZE) {
    free(str);
    if ((str = (char *)malloc(len + 1)) == NULL) {
      return FAILURE_MEMORY;
    }
    snprintf(str, len + 1, "%s(%d)", argv[0], argc);
  }
  printf("%s\n", str);
  free(str);
  str = NULL;
  return SUCCESS;
}
```

Buffer Allocation Strategies and Trade-Offs

Previous program rewritten using dynamic memory allocation along with sanity checks on the maximum memory allocated:

```
int main(int argc, char **argv) {
  char *str;
  int len;
  if ((str = (char *)malloc(BUFSIZE)) == NULL) {
    return FAILURE_MEMORY;
  }
  len = snprintf(str, BUFSIZE, "%s(%d)", argv[0], argc);
  if (len >= BUFSIZE) {
    free(str);
    if (len >= MAX_ALLOC) {
      return FAILURE_TOOBIG;
    }
    if ((str = (char *)malloc(len + 1)) == NULL) {
      return FAILURE_MEMORY;
    }
    snprintf(str, len + 1, "%s(%d)", argv[0], argc);
  }
  printf("%s\n", str);
  free(str);
  str = NULL;
  return SUCCESS;
}
```

Buffer Allocation Strategies and Trade-Offs

- Dynamic memory allocation can also introduce a variety of errors that are not present with static allocation – *memory leaks*, *use-after-free errors*, *double-free errors*.
- Memory leaks deplete available resources.
- Use-after-free and double-free errors often can cause segmentation faults, corrupt memory management data structures, and potentially introduce buffer overflow vulnerabilities.

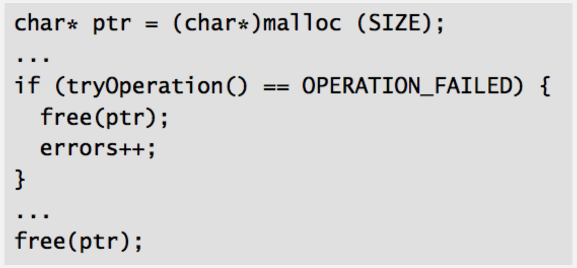
Buffer Allocation Strategies and Trade-Offs

A use-after-free vulnerability in simple error-handling code:

```
char *ptr = (char *) malloc(SIZE);
...
if (tryOperation() == OPERATION_FAILED) {
   free(ptr);
   errors++;
}
...
if (errors > 0) {
   logError("operation aborted before commit", ptr);
}
```

Buffer Allocation Strategies and Trade-Offs

A double-free vulnerability in simple error-handling code:



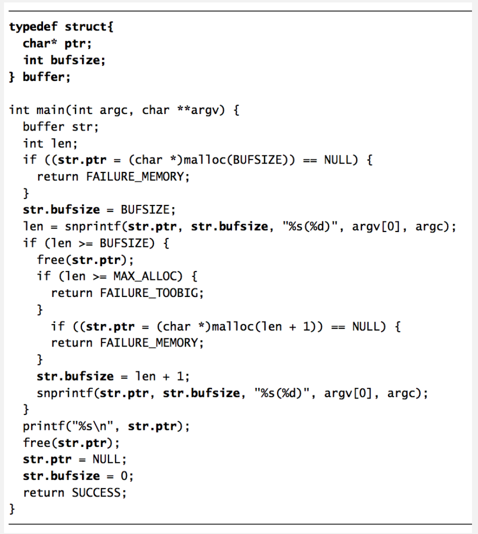
```
char* ptr = (char*)malloc (SIZE);
...
if (tryOperation() == OPERATION_FAILED) {
  free(ptr);
  errors++;
}
...
free(ptr);
```

Buffer Allocation Strategies and Trade-Offs

- Memory-safe languages, such as Java and Ruby, track the size of every buffer and make comparisons against this size when operations are performed on them.
- In unsafe languages, such as C and C++, a better approach to track a buffer and its size is to store them together in a composite data structure (such as a struct or class).
- When manually tracking the buffer size (by storing its size separately), it is important to maintain the size information correctly when performing operations on the buffer.
- A stale buffer size is more dangerous than no buffer size because later operations on the buffer might trust the stored size implicitly.
- Code on the next slide shows an example of how buffer size is stored, tracked, and maintained explicitly.

Buffer Allocation Strategies and Trade-Offs

Tracking buffer size explicitly in C:



```
typedef struct{
  char* ptr;
  int bufsize;
} buffer;

int main(int argc, char **argv) {
  buffer str;
  int len;
  if ((str.ptr = (char *)malloc(BUFSIZE)) == NULL) {
    return FAILURE_MEMORY;
  }
  str.bufsize = BUFSIZE;
  len = snprintf(str.ptr, str.bufsize, "%s(%d)", argv[0], argc);
  if (len >= BUFSIZE) {
    free(str.ptr);
    if (len >= MAX_ALLOC) {
      return FAILURE_TOOBIG;
    }
      if ((str.ptr = (char *)malloc(len + 1)) == NULL) {
      return FAILURE_MEMORY;
    }
    str.bufsize = len + 1;
    snprintf(str.ptr, str.bufsize, "%s(%d)", argv[0], argc);
  }
  printf("%s\n", str.ptr);
  free(str.ptr);
  str.ptr = NULL;
  str.bufsize = 0;
  return SUCCESS;
}
```

String Buffer Overflow

- The basic C *string* data structure (a null-terminated character array) is error-prone.
- Many C built-in string-manipulation functions are easy to misuse:
  - `gets()` and its friends
  - `scanf()` and its friends
  - `sprintf()` and its friends
  - `strcpy()` and its friends
  - `strcat()` and its friends
  - and others ...
- The best thing to do is to avoid these unsafe and dangerous functions altogether.

String Buffer Overflow

- C language offers bounded equivalents to the unsafe string-manipulation functions.
- These safer alternatives accept a parameter that limits the amount of data that will be written to the target buffer in an operation.
- These safer functions include `fgets()`, `snprintf()`, `strncpy()`, `strncat()`, and others.
- Proper use of bounded string-manipulation (and other safe) functions is an absolute bare-minimum requirement when building secure programs.

String Buffer Overflow

- Refactoring legacy code to make consistent use of bounded and safe functions is essential to make code more robust and less vulnerable.
- If the bound is specified incorrectly, a bounded function is just as capable of causing a buffer overflow as an unbounded function. The resulting errors are much harder to detect manually or automatically with tools.
- Even when used correctly, bounded functions can truncate the data they operate upon causing a variety of errors. So, perform sanity checks first on the data before using a bounded function.

String Buffer Overflow

- Common pitfalls with bounded functions
  - The destination buffer overflows because the bound depends on the size of the source data rather than the size of the destination buffer.
  - The destination buffer is left without a null terminator, often as a result of an off-by-one error.
  - The destination buffer overflows because its bound is specified as the total size of the buffer rather than the space remaining.
  - The program writes to an arbitrary location in memory because the destination buffer is not null-terminated and the function begins writing at the location of the first null character in the destination buffer.

String Buffer Overflow

- Guidelines to use, as appropriate, for avoiding pitfalls with bounded functions
  - Use a safe bound (depending on the bounded function being used).
  - Manually null-terminate the destination buffer after the function call.
  - Manually null-terminate the source and destination buffers before the call.
  - Insert null terminator in situations in which a string might lose its terminator.
  - Avoid blindly relying upon proper null terminator. Never assume that data from the outside world or from other string operations will be properly null-terminated.
  - A workable option might be to null-terminate the buffer at the last byte of allocated memory.

String Buffer Overflow

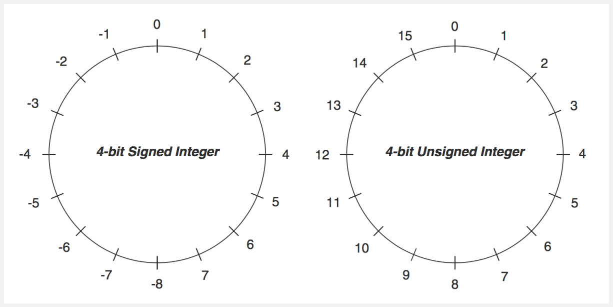
- Bounded string functions can truncate data in the process of copying the data from one location to another.
- Truncated data might have an unexpected meaning or become syntactically or semantically malformed so that subsequent operations on the data produce errors and incorrect behavior.
- An attack might be able to exploit truncation vulnerabilities.

Integer Overflow

- An *integer overflow* occurs when an integral value is increased or decreased beyond its capacity.
- Integer overflow errors are frequently an attacker's springboard to a buffer overflow.
- Buffer overflows are all about numbers.
- Numbers determine how much memory to allocate, control the bounds for string operations, and are used to calculate offsets for indexing memory.
- Operations that might carry a variable beyond its maximum or minimum value can be a prelude to a buffer overflow.
- Common Problems
  - Wrap-around errors
  - Truncation and sign extension errors
  - Conversion errors between signed and unsigned

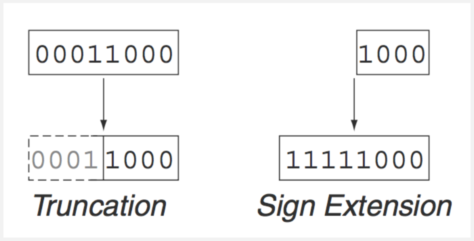
Integer Overflow

- Wrap-around errors occur when an integer value is increased or decreased to a value outside the range its data type is capable of representing.
- The following number wheels show the range of values that can be represented by signed and unsigned 4-bit integers (assuming two's complement representation).



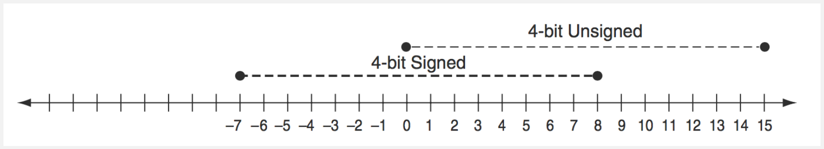
Integer Overflow

- Truncation errors occur when an integer data type with a larger number of bits is converted to a data type with fewer bits.
- Sign extension errors occur when a signed integer is converted from a smaller number of bits to a larger number of bits, the extra bits are filled in so that the new number retains the same sign.



Integer Overflow

- Both signed and unsigned data types (of same size in terms of number of bits) are capable of representing the same number of values.
- But there is only partial overlap between the range of numbers that the two types can express.
- Sign errors occur when a value is converted from signed to unsigned or vice versa.



Integer Overflow

- Methods to detect and prevent integer overflow
  - Use unsigned types, especially when used to allocate or index memory.
  - Declare signedness of an integral type explicitly to avoid bad assumptions.
  - Restrict numeric user input by imposing reasonable maximums and minimums.
  - Sanity-check values used to allocate and access memory.
  - Respect compiler warnings because they provide a great deal of assistance in tracking down potential integer overflow errors. If your program requires a lot of type casts to compile without warnings, you've got trouble.
  - Invest time in understanding integer conversion rules.
  - Verify pre- and post-conditions for operators that can overflow.

Buffer Overflow: Countermeasures

- Use safer functions in a programming language
- Use safer programming languages
    - Safe languages (such as Java, C#, Ruby, and Python) virtually eliminate buffer overflow errors at a cost (safety vs performance/flexibility).
    - Safe languages offer two properties to ensure that programs respect allocation bounds – *memory safety* and *type safety*.
    - Safe vs. unsafe languages – *concept of trust*
    - Unsafe languages implicitly trust the programmer, while safe languages explicitly limit the operations that they allow in exchange for the capability to prevent programs from making potentially damaging mistakes.

Buffer Overflow: Countermeasures

- Use a safe dialect of an unsafe language
- Use safer dynamic link libraries
- Use of static code analyzers/tols to detect potentially unsafe code
- Compile-time instrumentation:Stackshield and StackGuard
- Operating system: Address Space Layout Randomization (ASLR) to randomize the layout of the program memory
- Hardware architecture - NX (No-eXecute) bit to mark areas of memory as non-executable

References

- Brian Chess and Jacob West, "Secure Programming with Static Analysis", Chapters 6 & 7, Addison Wesley, 2007.
- Wenliang Du, "Computer Security: A Hands-on Approach", Chapter 4 on "Buffer Overflow Attack", CreateSpace eStore, 2017.