C/C++ Security Vulnerabilities

/*Secure Coding - WS 2014
LS XXII – Chair for Software Engineering
Technische Universität München*/

Outline

Motivation

How popular are C and C++ languages?

Motivation

How popular are C and C++ languages?

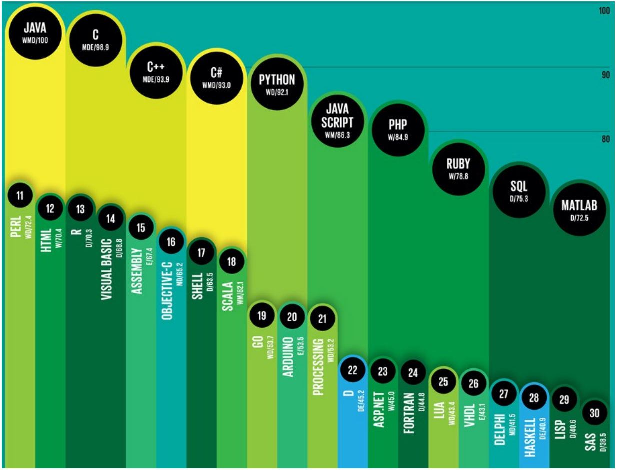


Figure : IEEE Spectrum's 2014 Ranking;
Source `http://spectrum.ieee.org/computing/software/top-10-programming-languages`

Outline

Motivation for Memory Management

C/C++ memory management is hard:

- No array bound checking
- No garbage collector
- Programmer responsible for memory management

Motivation for Memory Management

C/C++ memory management is hard:

- No array bound checking
- No garbage collector
- Programmer responsible for memory management

Typical Bugs:

- Buffer-overflows (e.g. 1988 Morris worm, 2003 Blaster worm, etc.)
- Dangling-pointers (e.g. 2010 attack on Google's corporate network)
- Memory-leaks (e.g. 2014 Heartbleed)

Outline

Questionary

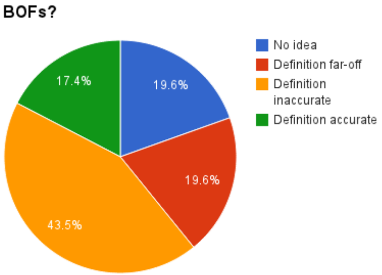


Figure : Stats on the questionary of Week 1 for the definition of Buffer Overflows.

Problems on the Call Stack

What happens if a program writes out of bounds of a static memory buffer?

Problems on the Call Stack

What happens if a program writes out of bounds of a static memory buffer?

Stack buffer overflows

- Alter control flow of program
- Execute arbitrary code → anything can happen

Problems on the Call Stack

What happens if a program writes out of bounds of a static memory buffer?

Stack buffer overflows

- Alter control flow of program
- Execute arbitrary code $\rightarrow$ anything can happen

Why is this possible?

- Mixture of data and control flow information on the stack
- Return addresses may be over-written

What is the stack?

- The stack is a memory area inside of process memory.

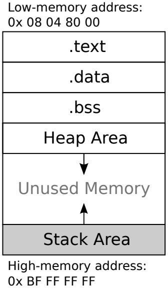


Figure : Memory organization of UNIX processes

What is the stack?

- The stack is a memory area inside of process memory.
- What is process memory?

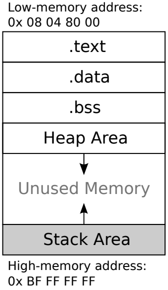


Figure : Memory organization of UNIX processes

What is the stack?

- The stack is a memory area inside of process memory.
- What is process memory?
- Dedicated memory allocated when a program begins execution
- Contains the following memory ares/segments:

`.text` readable & executable segment, containing program instructions

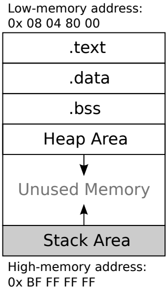


Figure : Memory organization of UNIX processes

What is the stack?

- The stack is a memory area inside of process memory.
- What is process memory?
- Dedicated memory allocated when a program begins execution
- Contains the following memory ares/segments:

`.text` readable & executable segment, containing program instructions

`.data` readable & writable segment, containing initialized global variables

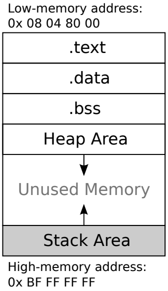


Figure : Memory organization of UNIX processes

What is the stack?

- The stack is a memory area inside of process memory.
- What is process memory?
- Dedicated memory allocated when a program begins execution
- Contains the following memory ares/segments:

`.text` readable & executable segment, containing program instructions

`.data` readable & writable segment, containing initialized global variables

`.bss` readable & writable segment, containing uninitialized global variables

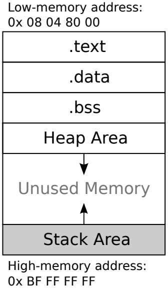


Figure : Memory organization of UNIX processes

What is the stack?

- The stack is a memory area inside of process memory.
- What is process memory?
- Dedicated memory allocated when a program begins execution
- Contains the following memory ares/segments:

`.text` readable & executable segment, containing program instructions

`.data` readable & writable segment, containing initialized global variables

`.bss` readable & writable segment, containing uninitialized global variables

`Heap` area used for dynamic memory allocation; grows “downward” to higher-memory addresses

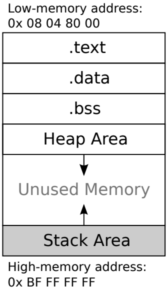


Figure : Memory organization of UNIX processes

What is the stack?

- The stack is a memory area inside of process memory.
- What is process memory?
- Dedicated memory allocated when a program begins execution
- Contains the following memory ares/segments:

`.text` readable & executable segment, containing program instructions

`.data` readable & writable segment, containing initialized global variables

`.bss` readable & writable segment, containing uninitialized global variables

`Heap` area used for dynamic memory allocation; grows "downward" to higher-memory addresses

`Stack` area used for allocation of memory associated to function calls; grows "upward" to lower-memory addresses

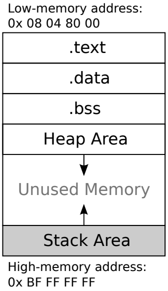


Figure : Memory organization of UNIX processes

Stack Structure

- The stack comprises of stack frames:

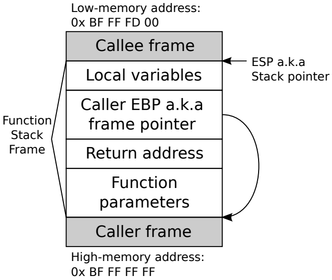


Figure : Structure of a stack frame

Stack Structure

- The stack comprises of stack frames:
  - New stack frame allocated at top of stack / function call
  - Top stack frame is removed when function returns

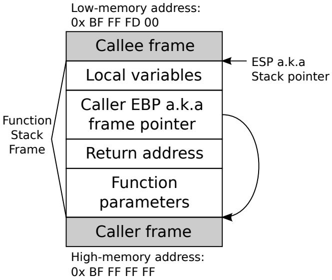


Figure : Structure of a stack frame

Stack Structure

- The stack comprises of stack frames:
  - New stack frame allocated at top of stack / function call
  - Top stack frame is removed when function returns
- Each stack frame holds at least the following information (bottom-to-top):

  1. Current function parameters

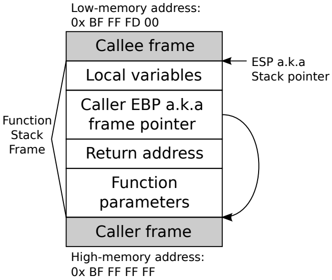


Figure : Structure of a stack frame

Stack Structure

- The stack comprises of stack frames:
  - New stack frame allocated at top of stack / function call
  - Top stack frame is removed when function returns
- Each stack frame holds at least the following information (bottom-to-top):

1. Current function parameters
2. Return address pop-ed in EIP when function finishes execution, (normally located in `.text` segment)

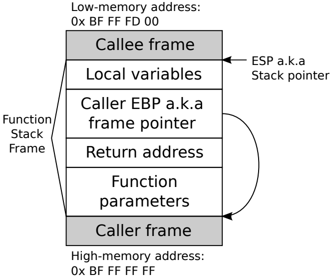


Figure : Structure of a stack frame

Stack Structure

- The stack comprises of stack frames:
  - New stack frame allocated at top of stack / function call
  - Top stack frame is removed when function returns
- Each stack frame holds at least the following information (bottom-to-top):

  1. Current function parameters
  2. Return address pop-ed in EIP when function finishes execution, (normally located in `.text` segment)
  3. The caller EBP, points to EBP in caller stack frame

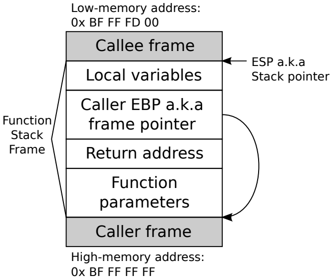


Figure : Structure of a stack frame

Stack Structure

- The stack comprises of stack frames:
  - New stack frame allocated at top of stack / function call
  - Top stack frame is removed when function returns
- Each stack frame holds at least the following information (bottom-to-top):

  1. Current function parameters
  2. Return address pop-ed in EIP when function finishes execution, (normally located in `.text` segment)
  3. The caller EBP, points to EBP in caller stack frame
  4. Local function parameters (including statically allocated buffers)

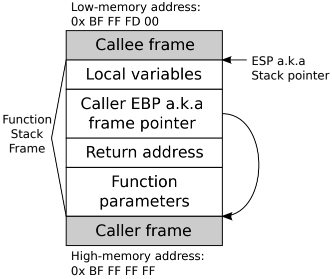


Figure : Structure of a stack frame

Video: How the Stack Evolves

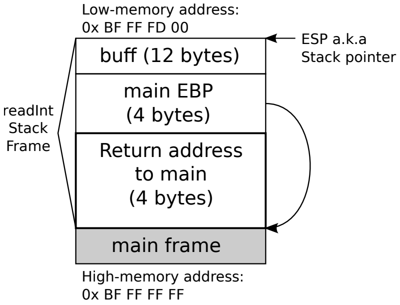
- Step-by-step debugging session (in Visual Studio 2010) of the following C program:

```c
/* returns sum of 4 parameters plus 2 */
int proc(int a, int b, int c, int d) {
  int e,f;
  e=2;
  f=a+b+c+d+e;
  return f;
}
/* entry point */
int main() {
  int a,b,c;
  a=6;
  b=5;
  c=proc(a,b,8,7);
  return c;
}
```

- GOAL: Learn how C code affects the stack
- https://www.dropbox.com/s/vk91vtulxbkj8mv/video-fixed-sum.mp4

Stack Buffer Overflows

- What is overwritten on the stack when a buffer overflow occurs?

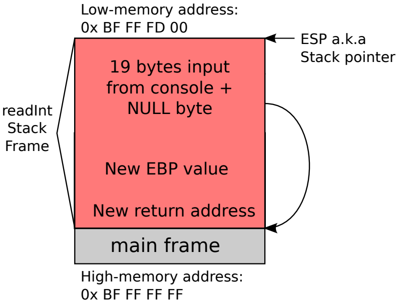



```
int readInt() {
  char buf[12];
  cin >> buf ;
  return atoi(buf);
}

int main() {
  /* ... */
  int a = readInt();
  /* ... */
}
```

Stack Buffer Overflows

- What is overwritten on the stack when a buffer overflow occurs?

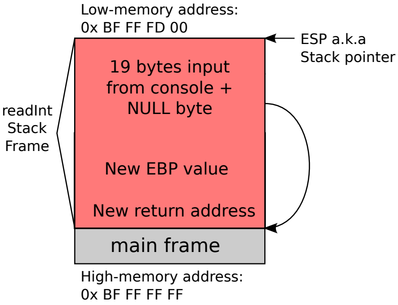



```
int readInt() {
  char buf[12];
  cin >> buf ;
  return atoi(buf);
}

int main() {
  /* ... */
  int a = readInt();
  /* ... */
}
```

Stack Buffer Overflows

- What is overwritten on the stack when a buffer overflow occurs?



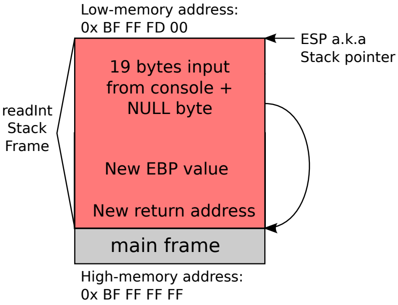

```
int readInt() {
  char buf[12];
  cin >> buf ;
  return atoi(buf);
}

int main() {
  /* ... */
  int a = readInt();
  /* ... */
}
```

- Where could the new return address point to?

Stack Buffer Overflows

- What is overwritten on the stack when a buffer overflow occurs?




```
int readInt() {
  char buf[12];
  cin >> buf ;
  return atoi(buf);
}

int main() {
  /* ... */
  int a = readInt();
  /* ... */
}
```

- Where could the new return address point to?
- Wherever attacker wants (see arc-injection and return-oriented programming)

Attacks Exploiting Buffer Overflows

1988 Morris worm - buffer overflow in Unix *finger* service

2001 Code Red worm - buffer overflow in Microsoft IIS 5.0

2003 SQL Slammer worm - buffer overflow in Microsoft SQL Server 2000

2003 Blaster worm - buffer overflow in DCOM RPC service of Windows 2000/XP

Cat and mouse game of the past decade

Buffer-overflow:

- → Code injection → Non-executable stack (NX bit, DEP, W⊕X)
- → Return-to-libc → ASLR, ASCII ARMOR, etc.
- → Jump-/Return- Oriented Programming (solutions in research phase)

Outline

String Errors

- Where do program inputs come from?

String Errors

- Where do program inputs come from?
    - Command-line arguments
    - Environment variables
    - GUI and console
    - Files (e.g. XML)
    - Network connections

String Errors

- Where do program inputs come from?
  - Command-line arguments
  - Environment variables
  - GUI and console
  - Files (e.g. XML)
  - Network connections
- What is a memory buffer in C/C++?

String Errors

- Where do program inputs come from?
  - Command-line arguments
  - Environment variables
  - GUI and console
  - Files (e.g. XML)
  - Network connections
- What is a memory buffer in C/C++?

  $\rightarrow$ An array (of bytes, words, double-words)

String Errors

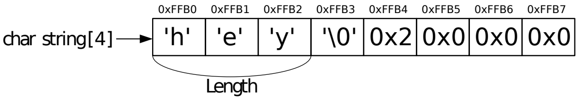
- Where do program inputs come from?
  - Command-line arguments
  - Environment variables
  - GUI and console
  - Files (e.g. XML)
  - Network connections
- What is a memory buffer in C/C++?

  → An array (of bytes, words, double-words)
- What is a string in C?

String Errors

- Where do program inputs come from?
  - Command-line arguments
  - Environment variables
  - GUI and console
  - Files (e.g. XML)
  - Network connections
- What is a memory buffer in C/C++?

  → An array (of bytes, words, double-words)
- What is a string in C?



String Manipulation Errors

Where do string manipulation errors come from?

String Manipulation Errors

Where do string manipulation errors come from?

- Improperly bounded string copying
- Off-by-one errors
- Null-termination errors
- String truncation
- String errors without functions

String Manipulation Errors

Where do string manipulation errors come from?

- Improperly bounded string copying
- Off-by-one errors
- Null-termination errors
- String truncation
- String errors without functions

NOTE: previous manipulation errors are applicable to other C/C++ memory buffers (not only strings)

Improperly Bounded String Copies (1)

- *Improperly bounded string copies* refer to usage of functions which allow writing characters out of string bounds
- Problem reading user input with `gets` in C?

```
char *gets(char *dest) {
  int c = getchar();
  char *p = dest;
  while (c != EOF && c != '\n') {
    *p++ = c;
    c = getchar();
  }
  *p = '\0';
  return dest;
}
```

Improperly Bounded String Copies (1)

- *Improperly bounded string copies* refer to usage of functions which allow writing characters out of string bounds
- Problem reading user input with `gets` in C?

```c
char *gets(char *dest) {
  int c = getchar();
  char *p = dest;
  while (c != EOF && c != '\n') {
    *p++ = c;
    c = getchar();
  }
  *p = '\0';
  return dest;
}
```

- Yes. There's no way to indicate the size of `dest`
- Malicious user can write outside of allocated buffer

Improperly Bounded String Copies (2)

- How would you fix gets if you could?

```c
char *gets(char *dest) {
  int c = getchar();
  char *p = dest;
  while (c != EOF && c != '\n') {
    *p++ = c;
    c = getchar();
  }
  *p = '\0';
  return dest;
}
```

Improperly Bounded String Copies (2)

- How would you fix gets if you could?

```
char *gets(char *dest, int dest_length) {
  if (dest_length <= 0)
    return dest;
  int i = 0;
  int c = getchar();
  char *p = dest;
  while (c != EOF && c != '\n' && i < dest_length) {
    *p++ = c;
    c = getchar();
    i++;
  }
  *p = '\0';
  return dest;
}
```

Improperly Bounded String Copies (3)

- Is there a problem with `strcpy` and `strcat` in this code?

```c
int main(int argc, char *argv[]) {
  char fullname[62];
  if (argc < 3)
    return 1;
  strcpy(fullname, argv[1]);
  strcat(fullname, " ");
  strcat(fullname, argv[2]);
  printf("fullname: %s\n", fullname);
  return 0;
}
```

Improperly Bounded String Copies (3)

- Is there a problem with `strcpy` and `strcat` in this code?

```
int main(int argc, char *argv[]) {
  char fullname[62];
  if (argc < 3)
    return 1;
  strcpy(fullname, argv[1]);
  strcat(fullname, " ");
  strcat(fullname, argv[2]);
  printf("fullname: %s\n", fullname);
  return 0;
}
```

- Yes. No way to specify the size of the destination buffer
- Malicious user can write outside of allocated buffer

Improperly Bounded String Copies (4)

- How would you fix the code without modifying the implementation of `strcpy` and `strcat`?

```c
int main(int argc, char *argv[]) {
  char fullname[62];
  if (argc < 3)
    return 1;
  strcpy(fullname, argv[1]);
  strcat(fullname, " ");
  strcat(fullname, argv[2]);
  printf("fullname: %s\n", fullname);
  return 0;
}
```

Improperly Bounded String Copies (4)

- How would you fix the code without modifying the implementation of `strcpy` and `strcat`?

```c
int main(int argc, char *argv[]) {
  if (argc < 3)
    return 1;
  char *fullname = (char *) malloc(
    strlen(argv[1]) + strlen(argv[2]) + 2);
  strcpy(fullname, argv[1]);
  strcat(fullname, " ");
  strcat(fullname, argv[2]);
  printf("fullname: %s\n", fullname);
  return 0;
}
```

Off-by-One Errors

- *Off-by-one errors* refer to deficient string accesses by 1 offset
- How many off-by-one errors do you spot in the following code?

```c
int main(int argc, char *argv[]) {
  char source[10];
  strcpy(source, "0123456789");
  char *dest = (char *) malloc(strlen(source));
  int i = 0;
  do {
    i++;
    dest[i] = source[i];
  } while (i < 10);
  dest[i] = '\0';
  printf("dest = %s", dest);
}
```

Off-by-One Errors

- *Off-by-one errors* refer to deficient string accesses by 1 offset
- How many off-by-one errors do you spot in the following code?

```
1  int main(int argc, char *argv[]) {
2    char source[10];
3    strcpy(source, "0123456789");
4    char *dest = (char *) malloc(strlen(source));
5    int i = 0;
6    do {
7      i++;
8      dest[i] = source[i];
9    } while (i < 10);
10   dest[i] = '\0';
11   printf("dest = %s", dest);
12 }
```

Line 3: `strcpy` adds `'\0'` after the last character of the source

Line 4: `malloc` should allocate 1 byte more, to account for the null byte

Line 7: `i` should be incremented after the assignment in line 8

Line 10: after the loop ends `i` is equal to 10

Null-Termination Errors

- *Null-termination errors* refer to situations where the '\0' byte termination is missing or misplaced
- What are the values of `a`, `b` and `c` at the end of the following code?

```c
char a[16];
char b[16];
char c[32];
strcpy(a, "0123456789abcdef");
strcpy(b, "0123456789abcde");
strcpy(c, a);
strcat(c, b);
```

Null-Termination Errors

- *Null-termination errors* refer to situations where the '\0' byte termination is missing or misplaced
- What are the values of `a`, `b` and `c` at the end of the following code?

```
1      char a[16];
2      char b[16];
3      char c[32];
4      strcpy(a, "0123456789abcdef");
5      strcpy(b, "0123456789abcde");
6      strcpy(c, a);
7      strcat(c, b);
```

If the program doesn't crash on line 7 then the values are:

- `a = ''0123456789abcdef0123456789abcde''`
- `b = ''0123456789abcde''`
- `c = ''0123456789abcdef0123456789abcde0123456789abcde''`

Null-Termination Errors

- *Null-termination errors* refer to situations where the '\0' byte termination is missing or misplaced
- What are the values of `a`, `b` and `c` at the end of the following code?

```
1    char a[16];
2    char b[16];
3    char c[32];
4    strcpy(a, "0123456789abcdef");
5    strcpy(b, "0123456789abcde");
6    strcpy(c, a);
7    strcat(c, b);
```

If the program doesn't crash on line 7 then the values are:

- `a = ‘‘0123456789abcdef0123456789abcde’’`
- `b = ‘‘0123456789abcde’’`
- `c = ‘‘0123456789abcdef0123456789abcde0123456789abcde’’`

Why?

Null-Termination Errors

- *Null-termination errors* refer to situations where the '\0' byte termination is missing or misplaced
- What are the values of `a`, `b` and `c` at the end of the following code?

```c
1 char a[16];
2 char b[16];
3 char c[32];
4 strcpy(a, "0123456789abcdef");
5 strcpy(b, "0123456789abcde");
6 strcpy(c, a);
7 strcat(c, b);
```

If the program doesn't crash on line 7 then the values are:

- `a = ''0123456789abcdef0123456789abcde''`
- `b = ''0123456789abcde''`
- `c = ''0123456789abcdef0123456789abcde0123456789abcde''`

Why? ... because line 5 overwrites the '\0' written on the first byte of `b` by line 4

String Truncation

- *String truncation* occurs when a source string is (bounded) copied into a smaller destination string
- May cause improper behavior of the program
- What is the problem with the following program?

```
int main(int argc, char *argv[]){
  char buf[12];
  strncpy(buf, argv[1], sizeof(buf));
  return 0;
}
```

String Truncation

- *String truncation* occurs when a source string is (bounded) copied into a smaller destination string
- May cause improper behavior of the program
- What is the problem with the following program?

```c
int main(int argc, char *argv[]){
 char buf[12];
 strncpy(buf, argv[1], sizeof(buf));
 return 0;
}
```

- No null character at the end of buf if input longer than 11.
- This might cause reading overflows later in the program.

String Errors without Functions

- We've seen gets, strcpy and strcat are bad because they don't properly bound string writes
- However, errors could still occur if we don't use them or any other functions
- What input argument could make the following program crash?

```c
int main(int argc, char *argv[]){
  int i = 0;
  char buf[128];
  char *arg1 = argv[1];
  while (arg1[i] != '\0') {
    buf[i] = arg1[i++];
  }
  buff[i] = '\0';
}
```

String Errors without Functions

- We've seen `gets`, `strcpy` and `strcat` are bad because they don't properly bound string writes
- However, errors could still occur if we don't use them or any other functions
- What input argument could make the following program crash?

```
int main(int argc, char *argv[]){
  int i = 0;
  char buf[128];
  char *arg1 = argv[1];
  while (arg1[i] != '\0') {
    buf[i] = arg1[i++];
  }
  buff[i] = '\0';
}
```

- Any input argument longer than 127 characters

Outline

Dynamic Memory Management Errors

- Previous examples used statically allocated memory on stack
- Does this mean that if we dynamically allocate memory we are safe?

Dynamic Memory Management Errors

- Previous examples used statically allocated memory on stack
- Does this mean that if we dynamically allocate memory we are safe?
- No. There are many common errors involving dynamic memory management:
  - initialization errors
  - failing to check return values
  - freeing memory multiple times
  - failure to distinguish scalars and arrays
  - etc.
- In the following we will see examples

Initialization Errors

- What does the following code print out?

```c
int main(int argc, char* argv[]) {
  int i, j, n = 3;
  for (j = 0; j < n; j++){
    int *x = malloc(n * sizeof(int));
    for (i = 1; i < n; i++) {
      x[i] += x[i-1] + i;
    }
    printf("%d\n", x[n-1]);
    free(x);
  }
  return 0;
}
```

Initialization Errors

- What does the following code print out?

```
int main(int argc, char* argv[]) {
  int i, j, n = 3;
  for (j = 0; j < n; j++){
    int *x = malloc(n * sizeof(int));
    for (i = 1; i < n; i++) {
      x[i] += x[i-1] + i;
    }
    printf("%d\n", x[n-1]);
    free(x);
  }
  return 0;
}
```

- `malloc` does not initialize memory to zero, it recycles freed memory
- The code could print any 3 bytes depending on memory manager and state of the OS

Initialization Errors

- What does the following code print out?

```c
int main(int argc, char* argv[]) {
  int i, j, n = 3;
  for (j = 0; j < n; j++){
    int *x = malloc(n * sizeof(int));
    for (i = 1; i < n; i++) {
      x[i] += x[i-1] + i;
    }
    printf("%d\n", x[n-1]);
    free(x);
  }
  return 0;
}
```

- `malloc` does not initialize memory to zero, it recycles freed memory
- The code could print any 3 bytes depending on memory manager and state of the OS
- Does this sound familiar? Any recent attacks which benefit from this?

Initialization Errors

- What does the following code print out?

```
int main(int argc, char* argv[]) {
  int i, j, n = 3;
  for (j = 0; j < n; j++){
    int *x = malloc(n * sizeof(int));
    for (i = 1; i < n; i++) {
      x[i] += x[i-1] + i;
    }
    printf("%d\n", x[n-1]);
    free(x);
  }
  return 0;
}
```

- `malloc` does not initialize memory to zero, it recycles freed memory
- The code could print any 3 bytes depending on memory manager and state of the OS
- Does this sound familiar? Any recent attacks which benefit from this?
- Heartbleed (`malloc` allocated the buffer sent to the requester)

Heartbleed

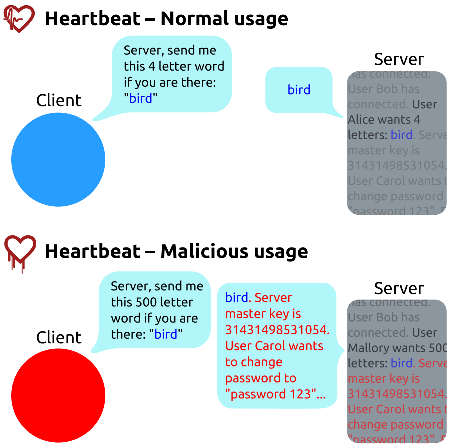


Figure : Simplified Heartbleed Explanation; Source Wikipedia

Patch?

Heartbleed

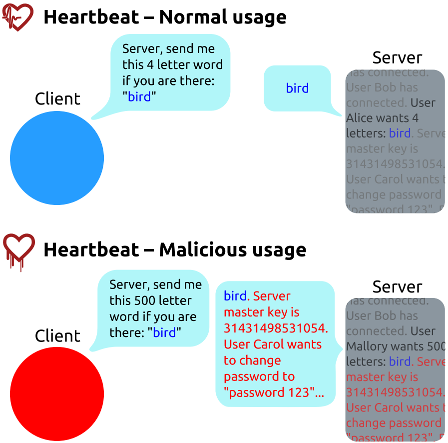


Figure : Simplified Heartbleed Explanation; Source Wikipedia

```
if (1 + 2 + payload + 16 > s->s3->rrec.length)
   return 0; //silently discard per RFC 6520 sec. 4
```

Failing to Check Return Values

- We've fixed this code before. Anything still wrong with it?

```
int main(int argc, char *argv[]) {
  if (argc < 3)
    return 1;
  char *fullname = (char *) malloc(
    strlen(argv[1]) + strlen(argv[2]) + 2);
  strcpy(fullname, argv[1]);
  strcat(fullname, " ");
  strcat(fullname, argv[2]);
  printf("fullname: %s\n", fullname);
  return 0;
}
```

Failing to Check Return Values

- We've fixed this code before. Anything still wrong with it?

```
int main(int argc, char *argv[]) {
  if (argc < 3)
    return 1;
  char *fullname = (char *) malloc(
    strlen(argv[1]) + strlen(argv[2]) + 2);
  strcpy(fullname, argv[1]);
  strcat(fullname, " ");
  strcat(fullname, argv[2]);
  printf("fullname: %s\n", fullname);
  return 0;
}
```

- `malloc` return value is not checked
- `malloc` returns NULL if the requested space cannot be allocated
- Referencing the NULL memory pointer will probably lead to a crash

Failing to Check Return Values

- We've fixed this code before. Anything still wrong with it?

```
int main(int argc, char *argv[]) {
  if (argc < 3)
    return 1;
  char *fullname = (char *) malloc(
    strlen(argv[1]) + strlen(argv[2]) + 2);
  strcpy(fullname, argv[1]);
  strcat(fullname, " ");
  strcat(fullname, argv[2]);
  printf("fullname: %s\n", fullname);
  return 0;
}
```

- `malloc` return value is not checked
- `malloc` returns NULL if the requested space cannot be allocated
- Referencing the NULL memory pointer will probably lead to a crash
- How about the `new` operator in C++? Does it return NULL?

Failing to Check Return Values

- We've fixed this code before. Anything still wrong with it?

```c
int main(int argc, char *argv[]) {
  if (argc < 3)
    return 1;
  char *fullname = (char *) malloc(
    strlen(argv[1]) + strlen(argv[2]) + 2);
  strcpy(fullname, argv[1]);
  strcat(fullname, " ");
  strcat(fullname, argv[2]);
  printf("fullname: %s\n", fullname);
  return 0;
}
```

- `malloc` return value is not checked
- `malloc` returns NULL if the requested space cannot be allocated
- Referencing the NULL memory pointer will probably lead to a crash
- How about the `new` operator in C++? Does it return NULL?
- No. It throws a `bad_alloc` exception (use `try-catch` block)

Referencing Freed Memory

- Assuming `head` is a pointer to a linked-list structure
- What is wrong with this code?

```
for (p = head; p != NULL; p = p->next)
  free(p);
```

Referencing Freed Memory

- Assuming `head` is a pointer to a linked-list structure
- What is wrong with this code?

```
for (p = head; p != NULL; p = p->next)
  free(p);
```

- `p` is being used in the loop condition after being freed
- `free` does not set memory to zero
- However, there are no guarantees that another application will not allocate and change the memory pointed to by `p` between 2 iterations of the `for` loop

Referencing Freed Memory

- Assuming head is a pointer to a linked-list structure
- What is wrong with this code?

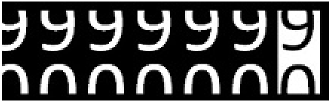
```c
for (p = head; p != NULL; p = p->next)
  free(p);
```

- p is being used in the loop condition after being freed
- free does not set memory to zero
- However, there are no guarantees that another application will not allocate and change the memory pointed to by p between 2 iterations of the for loop
- Very dangerous if p was being used as a pointer to a function

Outline

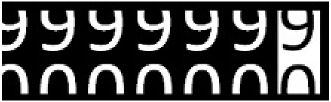
Integer Errors

- A growing and underestimated source of vulnerabilities in C/C++
- Primary issue is ignoring boundary conditions
- Integers in software are not infinite like in mathematics



Integer Errors

- A growing and underestimated source of vulnerabilities in C/C++
- Primary issue is ignoring boundary conditions
- Integers in software are not infinite like in mathematics



- Integer errors lead to the following types of vulnerabilities:
  - integer overflows
  - sign errors
  - truncation errors
  - logic errors
- In the following we will see examples

Integer representation

- **Negative numbers**: major issue in digital integer representation
- Which representation methods do you know?

Integer representation

- **Negative numbers**: major issue in digital integer representation
- Which representation methods do you know?
  - **Signed-magnitude**: 1st bit is sign, remaining bits are magnitude
    `5 = 0101, -5 = 1101`
  - **One's complement**: negative nr. $=$ bit negation of positive nr.
    `5 = 0101, -5 = 1010`
  - **Two's complement**: negative nr. $=$ bit negation of positive nr. $+ 1$
    `5 = 0101, -5 = 1011`
- What about the representation of zero?

Integer representation

- **Negative numbers**: major issue in digital integer representation
- Which representation methods do you know?
  - **Signed-magnitude**: 1st bit is sign, remaining bits are magnitude
    `5 = 0101, -5 = 1101`
  - **One's complement**: negative nr. $=$ bit negation of positive nr.
    `5 = 0101, -5 = 1010`
  - **Two's complement**: negative nr. $=$ bit negation of positive nr. $+ 1$
    `5 = 0101, -5 = 1011`
- What about the representation of zero?
  - **Signed-magnitude**: `0 = 0000, -0 = 1000`
  - **One's complement**: `0 = 0000, -0 = 1111`
  - **Two's complement**: `0 = 0000`
- Two's complement used almost universally in modern computers

Two's Complement

- Integer signed types: `short`, `int`, `long`, `llong`
- For each signed type there exists an equivalent `unsigned` type
- If a type is represented on n bits, which is the range or numbers that can be represented?

Two's Complement

- Integer signed types: `short`, `int`, `long`, `llong`
- For each signed type there exists an equivalent `unsigned` type
- If a type is represented on n bits, which is the range or numbers that can be represented?
  - $[-2^{n-1}, 2^{n-1} - 1]$ for signed types
  - $[0, 2^n - 1]$ for unsigned types

Bits	Unsigned value	Signed value
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

Table : Two's complement integers represented on 3 bits

Integer Overflow

- *Integer overflows* occurs when an integer value is too large to be stored in a variable.

Integer Overflow

- *Integer overflows* occurs when an integer value is too large to be stored in a variable.
- What does the following program print if we call it with the arguments 1500000000 1500000000?

```c
int main(int argc, char *argv[]) {
  int a = atoi(argv[1]);
  int b = atoi(argv[2]);
  if ((a + b) < 0)
    printf("Access grandted\n");
  else
    printf("Access denied\n");
}
```

Integer Overflow

- *Integer overflows* occurs when an integer value is too large to be stored in a variable.
- What does the following program print if we call it with the arguments 1500000000 1500000000?

```
1 int main(int argc, char *argv[]) {
2   int a = atoi(argv[1]);
3   int b = atoi(argv[2]);
4   if ((a + b) < 0)
5     printf("Access grandted\n");
6   else
7     printf("Access denied\n");
8 }
```

- Access granted, because the addition on line 4 overflows and becomes negative

Sign Error

- *Sign errors* occur when a negative integer value is implicitly or explicitly cast to an unsigned integer value.

Sign Error

- *Sign errors* occur when a negative integer value is implicitly or explicitly cast to an unsigned integer value.
- Will the `printf` statement be executed in this code?

```
unsigned int n = 4294967295; // UINT_MAX
short m = -1;

if (n == m) {
  printf("Will this be printed?");
}
```

Sign Error

- *Sign errors* occur when a negative integer value is implicitly or explicitly cast to an unsigned integer value.
- Will the `printf` statement be executed in this code?

```c
1 unsigned int n = 4294967295; // UINT_MAX
2 short m = -1;
3
4 if (n == m) {
5   printf("Will this be printed?");
6 }
```

- Yes, because:
  - on line 2: `m` is represented on 16 bits as `0xFFFF`
  - on line 4:
    1. `m` is implicitly converted to an `unsigned int` on 32 bits
    2. the sign of `m` is extended resulting in `0xFFFFFFFF`

Truncation Error

- *Truncation errors* occur when an integer is implicitly or explicitly converted to a smaller integer type.

Truncation Error

- *Truncation errors* occur when an integer is implicitly or explicitly converted to a smaller integer type.
- What value will the `printf` statement output in this code?

```
unsigned int n = 1024*1024;
unsigned short m = n;
printf("%d", m);
```

Truncation Error

- *Truncation errors* occur when an integer is implicitly or explicitly converted to a smaller integer type.
- What value will the `printf` statement output in this code?

```
1 unsigned int n = 1024*1024;
2 unsigned short m = n;
3 printf("%d", m);
```

- 0, because:
  - on line 1: `n` is represented on 32 bits as `0x00100000`
  - on line 2:
    1. `m` is converted to a `short` on 16 bits
    2. the most significant part (left-half) of `m` is truncated resulting in `0x0000`

Truncation Error → Sign Error → Buffer Overflow

- Where and why is there a buffer overflow in the following example?

```c
#define MAX_BUF 256
int main(int argc, char *argv[]) {
  char buf[MAX_BUF];
  short len = strlen(argv[1]);
  if (len < MAX_BUF)
    strcpy(buf, argv[1]);
  /* ... */
}
```

Truncation Error → Sign Error → Buffer Overflow

- Where and why is there a buffer overflow in the following example?

```
1  #define MAX_BUF 256
2  int main(int argc, char *argv[]) {
3    char buf[MAX_BUF];
4    short len = strlen(argv[1]);
5    if (len < MAX_BUF)
6      strcpy(buf, argv[1]);
7    /* ... */
8  }
```

- Buffer overflow on line 6 is caused by truncation and sign errors, because:
  1. On line 4 `strlen` returns `size_t` which is an `unsigned int`
  2. An `unsigned int` larger than $2^{16} - 1$ gets truncated
  3. If the most significant bit of remaining part is 1 then `len`< 0
  4. This results in a TRUE condition evaluation on line 5

Integer Logic Errors

- What is the problem in the following example?

```c
int *table = NULL;
int insert_in_table(int pos, int value){
  if (!table) {
    table = (int *) malloc(sizeof(int) * 100);
  }
  if (pos > 99) {
    return -1;
  }
  table[pos] = value;
  return 0;
}
```

Integer Logic Errors

- What is the problem in the following example?

```c
int *table = NULL;
int insert_in_table(int pos, int value){
  if (!table) {
    table = (int *) malloc(sizeof(int) * 100);
  }
  if (pos > 99) {
    return -1;
  }
  table[pos] = value;
  return 0;
}
```

- There is no check whether `pos > 0`
- If function is called with a negative `pos` value then `value` is written to a memory address before the start of the `table` array in heap memory

Outline

Format String Vulnerabilities

- The C99 standard defines formatted output functions:
  - `printf`, `fprintf`, `sprintf`, `snprintf`, etc.
  - They accept a variable number of arguments (**varidic functions**)
  - One of the arguments is called a **format string**

Format String Vulnerabilities

- The C99 standard defines formatted output functions:
  - `printf`, `fprintf`, `sprintf`, `snprintf`, etc.
  - They accept a variable number of arguments (**varidic functions**)
  - One of the arguments is called a **format string**
- All arguments of varidic functions are pushed to the stack

Format String Vulnerabilities

- The C99 standard defines formatted output functions:
  - `printf`, `fprintf`, `sprintf`, `snprintf`, etc.
  - They accept a variable number of arguments (**varidic functions**)
  - One of the arguments is called a **format string**
- All arguments of varidic functions are pushed to the stack
- Format strings are character sequences consisting of ordinary characters and **conversion specifiers**
  - Ordinary characters are copied unchanged to the output stream
  - Conversion specifiers:
    - Consume, convert and print arguments (from the stack)
    - Will consume one or more arguments
    - Begin with a % sign and have the following format:
      % [flags] [width] [.precision] [length] conversion-specifier
    - E.g. `%-10.8ld` prints a negative long integer on 8 digits in a 10 character wide field

Format String Vulnerabilities

- The C99 standard defines formatted output functions:
  - `printf`, `fprintf`, `sprintf`, `snprintf`, etc.
  - They accept a variable number of arguments (**varidic functions**)
  - One of the arguments is called a **format string**
- All arguments of varidic functions are pushed to the stack
- Format strings are character sequences consisting of ordinary characters and **conversion specifiers**
  - Ordinary characters are copied unchanged to the output stream
  - Conversion specifiers:
    - Consume, convert and print arguments (from the stack)
    - Will consume one or more arguments
    - Begin with a % sign and have the following format:
      % [flags] [width] [.precision] [length] conversion-specifier
    - E.g. `%-10.8ld` prints a negative long integer on 8 digits in a 10 character wide field
- If there are more conversion specifiers than arguments in formatted output function call, the **behavior is undefined**

Format String Attacks

- First format string vulnerability discovered in June 2000 on FTP demon developed by Washington University (WU-FTP) `http://www.kb.cert.org/vuls/id/29823`
- **Impact**:

  > *By exploiting any of these input validation problems, local or remote users logged into the ftp daemon may be able execute arbitrary code as root. An anonymous ftp user may also be able to execute arbitrary code as root.*

- Many more such vulnerabilities were discovered since then...

Format String Attacks

- First format string vulnerability discovered in June 2000 on FTP demon developed by Washington University (WU-FTP) `http://www.kb.cert.org/vuls/id/29823`
- **Impact**:

> *By exploiting any of these input validation problems, local or remote users logged into the ftp daemon may be able execute arbitrary code as root. An anonymous ftp user may also be able to execute arbitrary code as root.*

- Many more such vulnerabilities were discovered since then...
- Format string vulnerabilities can lead to:
  - Buffer overflows
  - Crashing a program
  - Viewing stack or other memory content
  - Overwriting memory
- In the following we will see examples...

Buffer overflows through format strings

- What is the buffer overflow in the following example?

```
char query[512];
char buffer[512];
sprintf(buffer,
 "SELECT username FROM users WHERE user_id='%.50s'",
 user_input);
sprintf(query, buffer);
mysql_query(query);
```

Buffer overflows through format strings

- What is the buffer overflow in the following example?

```
1 char query[512];
2 char buffer[512];
3 sprintf(buffer,
4  "SELECT username FROM users WHERE user_id='%.50s'",
5  user_input);
6 sprintf(query, buffer);
7 mysql_query(query);
```

- Not in line 3, because the precision field of the conversion specifier truncates strings larger than 50 chars

Buffer overflows through format strings

- What is the buffer overflow in the following example?

```
1 char query[512];
2 char buffer[512];
3 sprintf(buffer,
4  "SELECT username FROM users WHERE user_id='%.50s'",
5  user_input);
6 sprintf(query, buffer);
7 mysql_query(query);
```

- Not in line 3, because the precision field of the conversion specifier truncates strings larger than 50 chars
- It's in line 6, because:
  - buffer is used as the format string in line 6
  - the user input could be:
    %470d\x3c\xd3\xff\xbf<nops><shellcode>
  - the width field of the first conversion specifier fills the query buffer
  - afterwards the address of shellcode is supplied to overwrite the return address

Crashing a Program and Reading Stack Memory

- What does the following program do?

```c
int main(int argc, char* argv[]){
  printf("%s %s %s %s");
  return 0;
}
```

Crashing a Program and Reading Stack Memory

- What does the following program do?

```c
int main(int argc, char* argv[]){
  printf("%s %s %s %s");
  return 0;
}
```

- It reads sequences of characters ending in NULL bytes referenced by 4 consecutive addresses on the stack
- If an invalid pointer or unmapped memory is encountered the program crashes
- If you want to be certain it crashes just add more %s's

Writing Memory with Format Strings

- `%n` stores the number of characters successfully printed so far, in the integer whose address is given as the argument

Writing Memory with Format Strings

- `%n` stores the number of characters successfully printed so far, in the integer whose address is given as the argument
- What is the values of `i` before the end of the following program?

```
int main(int argc, char* argv[]){
  int i;
  printf("hello%n\n" (int *)&i);
  return 0;
}
```

Writing Memory with Format Strings

- `%n` stores the number of characters successfully printed so far, in the integer whose address is given as the argument
- What is the values of `i` before the end of the following program?

```
int main(int argc, char* argv[]){
  int i;
  printf("hello%n\n" (int *)&i);
  return 0;
}
```

- 5, because 5 characters were printed when `%n` is encountered
- So what? See next slide...

Overwriting Arbitrary Memory Locations

- `%n` lets attackers write any value they wish at arbitrary locations in memory

Overwriting Arbitrary Memory Locations

- `%n` lets attackers write any value they wish at arbitrary locations in memory
- Use this to write value > 1028 at address `0x0142f5dc`

```
int main(int argc, char* argv[]) {
 printf("\xdc\xf5\x42\x01 %08x ... %08x %1000u %n");
 return 0;
}
```

Overwriting Arbitrary Memory Locations

- `%n` lets attackers write any value they wish at arbitrary locations in memory
- Use this to write value > 1028 at address `0x0142f5dc`

```
int main(int argc, char* argv[]) {
 printf("\xdc\xf5\x42\x01 %08x ... %08x %1000u %n");
 return 0;
}
```

- make the `%n` specifier point to the beginning of the format string
- Why 1028? 4 bytes of the address + 5 spaces + 2×8 bytes + 3 dots + 1000 width of penultimate conversion specifier
- Why did I add 3 dots?

Overwriting Arbitrary Memory Locations

- %n lets attackers write any value they wish at arbitrary locations in memory
- Use this to write value > 1028 at address 0x0142f5dc

```
int main(int argc, char* argv[]) {
 printf("\xdc\xf5\x42\x01 %08x ... %08x %1000u %n");
 return 0;
}
```

- make the %n specifier point to the beginning of the format string
- Why 1028? 4 bytes of the address + 5 spaces + 2×8 bytes + 3 dots + 1000 width of penultimate conversion specifier
- Why did I add 3 dots?
- ASLR makes it hard to guess how many stack arguments to consume before getting to format string

Outline

Black-Box Testing for C Programs

- We will see countermeasures for C vulnerabilities in the following lectures, until then...
- ... in *Phase 2* of the Secure Coding project you should test for C/C++ vulnerabilities
- How do you do that, because you can't see the C code?

Black-Box Testing for C Programs

- We will see countermeasures for C vulnerabilities in the following lectures, until then...
- ... in *Phase 2* of the Secure Coding project you should test for C/C++ vulnerabilities
- How do you do that, because you can't see the C code?
  - Manually, by putting attack strings in the uploaded file
  - This lecture gave you hints at to how attack strings look for various vulnerabilities

Black-Box Testing for C Programs

- We will see countermeasures for C vulnerabilities in the following lectures, until then...
- ... in *Phase 2* of the Secure Coding project you should test for C/C++ vulnerabilities
- How do you do that, because you can't see the C code?
  - Manually, by putting attack strings in the uploaded file
  - This lecture gave you hints at to how attack strings look for various vulnerabilities
- Any automatic way to test for C vulnerabilities?

Black-Box Testing for C Programs

- We will see countermeasures for C vulnerabilities in the following lectures, until then...
- ... in *Phase 2* of the Secure Coding project you should test for C/C++ vulnerabilities
- How do you do that, because you can't see the C code?
  - Manually, by putting attack strings in the uploaded file
  - This lecture gave you hints at to how attack strings look for various vulnerabilities
- Any automatic way to test for C vulnerabilities?
  - Yes, use fuzzers (e.g. ZAP, Burp) that try various attack strings automatically for a given parameter
  - Build script (e.g. iMacros, Selenium), which simulate user behavior to upload file

Black-Box Testing for C Programs

- We will see countermeasures for C vulnerabilities in the following lectures, until then...
- ... in *Phase 2* of the Secure Coding project you should test for C/C++ vulnerabilities
- How do you do that, because you can't see the C code?
  - Manually, by putting attack strings in the uploaded file
  - This lecture gave you hints at to how attack strings look for various vulnerabilities
- Any automatic way to test for C vulnerabilities?
  - Yes, use fuzzers (e.g. ZAP, Burp) that try various attack strings automatically for a given parameter
  - Build script (e.g. iMacros, Selenium), which simulate user behavior to upload file
- In later lectures we will see how you find vulnerabilities in C programs when you have the binary

References

Most examples in this slide set are borrowed from

Robert C. Seacord.
Secure Coding in C and C++.
Addison-Wesley Professional, 2nd edition, 2013.