CS390S, Week 2: Buffer Overflows, Part 1

Pascal Meunier, Ph.D., M.Sc., CISSP
January 17, 2007
Developed thanks to the support of Symantec Corporation, NSF SFS Capacity Building Program (Award Number 0113725) and the Purdue e-Enterprise Center
Copyright (2004, 2007) Purdue Research Foundation. All rights reserved.
Buffer Overflows Part 1:

- Definition
- Lab: Identify and fix an open source buffer overflow
- Problems (to be continued):
  - Unbounded Writes
  - Indexing and Sizing Mistakes (e.g., off-by-one)
  - Truncated Strings
Buffer Overflows

- a.k.a. "Buffer Overrun"

- A buffer overflow happens when a program attempts to write data outside of the memory allocated for that data
  - Usually affects buffers of fixed size

- A closely related problem is reading outside a given buffer or array
  - "Out of bounds read"
    - a.k.a. "read buffer overflow"
    - a.k.a. "read buffer overruns"
An Important Vulnerability Type

- Most Common (over 60% of CERT advisories)
  - Hundreds of CVEs every year
- Well understood
- Easy to avoid in principle
  - Don't use "C" family languages, or be thorough
  - Can be tricky (off-by-one errors)
  - Tedious to do all the checks properly
  - Temptation: "I don't need to because I control this data and I *know* that it will never be larger than this"
    - Until a hacker figures out how to change it
    - Until someone adds a new feature
Buffer Overflow Lab

- Use the National Vulnerability Database to find a buffer overflow in an open source product
  - Alone or in teams of 2
  - Each team must choose a different product
  - Email me your teaming choices, and choice of product ASAP (first come first served; must be a 2006 or 2007 issue) and no later than January 24

- Idea:
  - Find the lines of code responsible for a recent vulnerability (2006 or 2007)
  - Suggest new code to fix the issue, OR analyze the fix (if any) that was made by the developers
Submit by email by February 7:

- The CVE identifier and CVE description
- The URL where you found the code
- The vulnerable code
- An explanation of why it's vulnerable
- Your code fix, OR a detailed analysis of why and if the code fix that was made is correct
- How long it took you to do this (as feedback, not graded)
Strings

- Standard "C" functions don't enforce string invariants
  - String size vs buffer size
  - NUL-termination
- String invariants must be checked and protected manually on every operation
  - Tedious
  - Error-prone
  - Size information is often missing!
- String manipulation is a common source of buffer overflows
Example Unbounded Write

- int main()
  {
    const char * first_string = "A witty saying proves nothing. (Voltaire)";
    char buffer[10];
    strcpy(buffer, first_string);
    return 0;
  }

- When compiled and run (MacOS X gcc version 3.3 20030304), this code always produces a segmentation fault
function do_stuff(char * a) {
    char b[100];
    ...
    strcpy(b, a); // (dest, source)
    ...
}

- What is the size of the string located at “a”?  
- Is it even a NUL-terminated string?  
- What if it was "strcpy(a, b);" instead?  
  - What is the size of the buffer pointed to by "a"?
Arrays

- "C" doesn't enforce index range on arrays
- Up to the programmer to use correctly
- Example Overflow:
  - char B[10];
    B[10] = x;
  - Array starts at index zero
  - So [10] is 11th element
  - One byte outside buffer was referenced
- Off-by-one errors are common and can be exploitable!
Real Life Example: efingerd.c, v. 1.6.2

- `int get_request (int d, char buffer[], u_short len) {
   u_short i;
   for (i=0; i< len; i++) {
      ...
   }
   buffer[i] = '\0';
   return i;
}

- What is the value of "i" at the end of the loop?
- Which byte just got zeroed?
- It's tricky even if you try to get things right...
What happens when memory outside a buffer is accessed?

- If memory doesn't exist:
  - Bus error

- If memory protection denies access:
  - Segmentation fault
  - General protection fault

- If access is allowed, memory next to the buffer can be accessed
  - Heap
  - Stack
  - Etc...
Real Life Example: efingerd.c, v. 1.5

- CAN-2002-0423
  ```c
  static char *lookup_addr(struct in_addr in) {
      static char addr[100];
      struct hostent *he;
      he = gethostbyaddr(...)
      strcpy (addr, he->h_name);
      return addr;
  }
  ```

- How big is `he->h_name`?
- Who controls the results of `gethostbyaddr`?
- How secure is DNS? Can you be tricked into looking up a maliciously engineered value?
Joining of Buffers

- `int main()`
  ```c
  { 
    const char * first_string = "A witty saying proves nothing. (Voltaire)"; 
    char buffer2[100]; 
    char buffer1[40]; 
    strcpy(buffer1, first_string); 
    strcpy(buffer2, first_string); 
    printf("%s\n", buffer1); 
    return 0; 
  }
  ```

- What happens?
Results

- Compiled and executed on Solaris 5.7 with gcc version 2.95.3 20010315):
  - `% ./a.out
    A witty saying proves nothing. (Voltaire)

- This indicates that the copy into `buffer1` overflowed and extended into `buffer2`.
- The copy into `buffer2` overwrote the NUL byte
- Unvalidated input (e.g., in `buffer2`) may unexpectedly leak into critical system operations (using `buffer1`).
NUL-Termination Issues

- Most C functions don't guarantee NUL-termination
- Consequently there is no guarantee that the strings you get are properly NUL-terminated
- Need to carefully read the function description to figure out
  - when it may not NUL-terminate a string
  - how to check if it did that
  - where to append a NUL character yourself
Strncpy

- `char * strncpy(char * dst, const char * src, size_t len);`
- "len" is the maximum number of characters to copy
- "dst" is NUL-terminated only if less than "len" characters were copied!
  - All calls to strncpy must be followed by a NUL-termination operation
What happens when you call strlen on an improperly terminated string?

- Strlen scans until a null character is found
  - Can scan outside buffer if string is not null-terminated
  - Can result in a segmentation fault or bus error

- Strlen is not safe to call!
  - Unless you positively know that the string is null-terminated...
    - Are all the functions you use guaranteed to return a null-terminated string?
Strcpy

- `char * strcpy(char * dst, const char * src);`

- **How can you use strcpy safely?**
  - Set the last character of src to NUL
    - According to the size of the buffer pointed to by src or a size parameter passed to you
    - Not according to `strlen(src)`!
    - Wide char array: `sizeof(src)/sizeof(src[0]) - 1` is the index of the last element
  - Check that the size of the src buffer is smaller than or equal to that of the dst buffer
  - Or allocate dst to be at least equal to the size of src
Question

- What’s wrong with this?

```c
function do_stuff(char * a) {
    char b[100];
    ...
    strncpy(b, a, strlen(a));
    ...
}
```
What’s wrong with this?

```c
function do_stuff(char * a) {
    char b[100];
    ...
    strncpy(b, a, strlen(a));
    ...
}
```

The string pointed to by "a" could be larger than the size of "b"!
Question

- What’s wrong with this?

```c
function do_stuff(char * a) {
    char *b;
    ...
    b = malloc(strlen(a)+1);
    strncpy(b, a, strlen(a));
    ...
}
```
Question Answer

What’s wrong with this?

```c
function do_stuff(char * a) {
    char *b;
    ...
    b = malloc(strlen(a)+1);
    strncpy(b, a, strlen(a));
    ...
}
```

Are you absolutely certain that the string pointed to by "a" is NUL-terminated?
- `sizeof` is your friend, when you can use it (if an array)

```c
static char addr[100];
he = gethostbyaddr(...);
if (he == NULL)
    strncpy(addr, inet_ntoa(in), sizeof(addr));
else
    strncpy(addr, he->h_name, sizeof(addr));
```

- What is still wrong?
Notice that the last byte of addr is not zeroed, so this code can produce non-NUL-terminated strings!

```c
static char addr[100];
he = gethostbyaddr(...);
if (he == NULL)
    strncpy(addr, inet_ntoa(in), sizeof(addr));
else
    strncpy(addr, he->h_name, sizeof(addr));
```
Strlcpy

- size_t strlcpy(char *dst, const char *src, size_t size);
- Guarantees to null-terminate string pointed to by "dst" if "size">0
- The rest of the destination buffer is not zeroed as for strncpy, so better performance is obtained
- "size" can simply be size of dst (sizeof if an array)
  - If all functions are guaranteed to null-terminate strings, then it is safe to assume src is null-terminated
  - Not safe if src is not null-terminated!
- See http://www.courtesan.com/todd/papers/strlcpy.html for benchmarks and more info
  - Used in MacOS X, OpenBSD and more (but not Linux)
Note on Strlcpy

- As the remainder of the buffer is not zeroed, there could be information leakage
Strcat

- `char * strcat(char * s, const char * append);`
- String pointed to by "append" is added at the end of the string contained in buffer "s"
- No check for size!
  - Need to do all checks beforehand
  - Example with arrays:
    - `if (sizeof(s) - strlen(s) - 1 >= strlen(append))
      strcat(s, append);`
- Need to trust that "s" and "append" are NUL-terminated
  - Or set their last byte to NUL before the checks and call
Strncat

- char * strncat(char * s, const char * append, size_t count);
- No more than "count" characters are added, and then a NUL is added
- Correct call is complex:
  - strncat(s, append, sizeof(s)-strlen(s)-1)
    - Not a great improvement on strcat, because you still need to calculate correctly the count
    - And then figure out if the string was truncated
- Need to trust that "s" and "append" are NUL-terminated
  - Or set their last byte to NUL before the checks and call
Strlcat

- `size_t strlcat(char *dst, const char *src, size_t size);`
- Call semantics are simple:
  - `Strlcat(dst, src, dst_len);`
  - If an array:
    - `strlcat(dst, src, sizeof(dst));`
- Safety: safe even if dst is not properly terminated
  - Won't read more than size characters from dst when looking for the append location
  - But won't NUL-terminate dst if size limit is reached...
- Not safe if src is not properly terminated!
  - If dst is large and the buffer for src is small, then it could cause a segmentation fault or bus error, or copy confidential values
Wide or multi-byte string handling functions do not guarantee NUL-termination!

- e.g.: mbsrtowcs converts multibyte characters to wide characters
- include <wchar.h>
  size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len, mbstate_t *ps);
- Is "len" in bytes or characters?
  - Characters!
- Conversion stops without NUL-termination if:
  - an invalid code is found
  - "len" characters are converted
  - the state "ps" is invalid (used for multithreading)
Truncated Strings

- Semantic consequences
- Truncation Detection
- Truncated wide or multi-byte characters
Semantic Consequences of Truncation

- Subsequent operations may fail or open up vulnerabilities
  - If string is a path, then it may not refer to the same thing, or be an invalid path
- Truncation most likely means that you weren't able to do what you wanted
  - If truncation is not explicitly a desirable result, you should handle that as an error instead of letting it go silently
Truncation Detection

- Truncation detection was simplified by `strlcpy` and `strlcat`, by changing the return value
  - The returned value is the size of what would have been copied if the destination had an infinite size
    - if this is larger than the destination size, truncation occurred
    - Source still needs to be NUL-terminated
    - Inspired by `snprintf` and `vsnprintf`, which do the same

- However, it still takes some consideration to make sure the test is correct:
  - `if (strlcpy(dest, src, sizeof(dest)) >= sizeof(dest)) goto toolong;`
Truncated Wide or Multibyte Characters

- Wide characters
  - fixed number of bytes > 1/character
- Multi-byte characters
  - Varying number of bytes/character
  - e.g., UTF-8 is 1-4 bytes long
- What if a character is truncated?
  - NUL byte may be "absorbed" into the malformed character
    - String is not NUL-terminated anymore!
  - Appended characters may change
    - Quotes, backslashes, etc... may be "absorbed" as well
      - Data may be interpreted differently: code injection!
Incorrect Specification of Bounds

- Single size argument, two or more buffers
- Sizes in bytes vs sizes in characters
- Next week:
  - Malicious sizes
  - Calling sizeof on pointers
Single Size Argument, Multiple Buffers

- Example functions:
  - `strncpy`, `strncat`
  - `memccpy`, `memcpy`, `memmove`, `memcmp`
  - `strncmp`, `strncasecmp`
  - `strnstr`, `strxfrm`

- What is the correct value for the size?
- To which buffer does the size apply?
What is the correct value of len for strncpy?

- Initial answer by most people: size of dst
  - If dst is an array, sizeof(dst)

- What if src is not NUL-terminated?
  - Don't want to read outside of src buffer
  - What is the correct value for "len" given that?
    - Minimum buffer size of dst and src, -1 for NUL byte
    - If arrays,
      - MIN(sizeof(dst), sizeof(src)) - 1
Size in bytes vs Size in Characters

- When converting wide characters and multibyte characters, some functions take sizes in characters and others take bytes
  - Error prone
    - wchar_t buffer[20] = {0};
      wcsncpy(buffer, pUnvalidatedInput, sizeof(buffer)-1); // bad

- Windows programmers especially seem to have a hard time getting the 6th argument correct in the call to MultiByteToWideChar, which expects a character count, not a byte count.
Tip

- Functions handling, or converting to, wide characters usually require character counts.
- Functions converting to multibyte characters usually require byte counts, because the number of bytes in the buffer is known, but varying numbers of multibyte characters may fit into the same buffer.
Pascal Meunier

Contributors:
Jared Robinson, Alan Krassowski, Craig Ozancin, Tim Brown, Wes Higaki, Melissa Dark, Chris Clifton, Gustavo Rodriguez-Rivera