

Secure Programming

Buffer Overflows

1

Ahmet Burak Can
Hacettepe University

2 Learning objectives

- Understand the definition of a buffer overflow
- Learn the importance of buffer overflows
- Know how buffer overflows happen
- Know how to handle strings safely with regular "C" functions
- Learn safer ways to manipulate strings and buffers

3 Buffer Overflows

- a.k.a. "Buffer Overrun"
- A buffer overflow happens when a program attempts to read or write data outside of the memory allocated for that data
 - Usually affects buffers of fixed size
- Special case of memory management and input validation

4 An Important Vulnerability Type

- Most Common (over 60% of CERT advisories)
- 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

5

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!

6

Other Example

```
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 null-terminated string?
- What if it was "strcpy(a, b);" instead?
 - What is the size of the buffer pointed to by "a"?

7

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...

8

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...

9 Real Life Example: efingerd.c, v. 1.5

- CAN-2002-0423


```
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?

10 A Typical Stack Exploit

- The stack contains:
 - Parameters (arguments) to function
 - Return Address
 - Local variables
 - Anything pushed on the stack
- addr[100+] overwrites the return address
- addr[0] typically contains exploit code
- Return address is chosen to point at exploit code!

11 Fundamental "C" Problems

- You can't know the length of buffers just from a pointer
 - Partial solution: pass the length as a separate argument
- "C" string functions aren't safe
 - No guarantees that the new string will be null-terminated!
 - Doing all checks completely and properly is tedious and tricky

12 Strlen

- 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?

13

Strcpy

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

- How can you use strcpy safely?
 - Set the last character of src to NULL
 - 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

14

Strncpy

```
char * strncpy(char * dst, const char * src, size_t len);
```

- "len" is maximum number of characters to copy
 - What is the correct value for len?
 - If dst is an array, sizeof(dst)
- What if src is not NULL-terminated?
 - Don't want to read outside of src buffer
 - What is the correct value for "len" given that?
 - Spare one character for NULL byte
 - $\text{MIN}(\text{sizeof}(\text{dst}), \text{sizeof}(\text{src})) - 1$
- Other issue: "dst" is NULL-terminated only if less than "len" characters were copied!
 - All calls to strncpy must be followed by a NULL-termination operation

15

Question Answer

- What's wrong with this function?

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

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

16

Question Answer

- What's wrong with this function?

```
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 NULL-terminated?

17

Corrected Efinger.c (v.1.6)

- sizeof is your friend, when you can use it (if an array)

```
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?

18

Corrected Efinger.c (v.1.6)

- Notice that the last byte of addr is not zeroed, so this code can produce non-NULL-terminated strings!

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

19

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 NULL-terminated
 - Or set their last byte to NULL before the checks and call

20

Strncat

```
char * strncat(char * s, const char * append,
              size_t count);
```

- No more than "count" characters are added, and then a NULL 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 NULL-terminated
  - Or set their last byte to NUL before the checks and call

21

## 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
- 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

22

## Issues with Truncating Strings

- 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 means you weren't able to do what you wanted
  - You should handle that error instead of letting it go silently

23

## Truncation Detection

- Truncation detection was simplified by `strncpy` 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 the returned value is larger than the destination size, truncation occurred
    - Source still needs to be NULL-terminated
    - Inspired by `snprintf` and `vsprintf`, which do the same
- However, it still takes some consideration to make sure the test is correct:
  - `if (strncpy(dest, src, sizeof(dest)) >= sizeof(dest)) goto toolong;`

24

## Multi-Byte Character Encodings

- Handling of strings using variable-width encodings or multi-byte encodings is a problem
  - e.g., UTF-8 is 1-4 bytes long
- How long is the string?
  - In bytes
  - In characters
- Overflows are possible if size checks do not properly account for character encoding!
- .NET: `System.String` supports UTF-16
  - Strings are immutable - no overflow possible there!

25

## Safestr

- Free library for safe string operations:
  - <https://manned.org/safestr/20fb981d>
- Features:
  - Works on UNIX and Windows
  - Buffer overflow protection
  - String format protection
- Limitations and differences:
  - Does not handle multi-byte characters
  - License: binaries must reproduce a copyright notice
  - NULL characters have no special meaning
  - Must use their library functions all the time (but conversion to regular "C" strings is easy)

26

## Microsoft Strsafe

- Null-termination guaranteed
- Option for using either number of characters or bytes (for Unicode character encoding), and disallowing the other
- Option to treat truncation as a fatal error
- Define behavior upon error
  - Output buffer set to "" or filled
- Option to prevent information leaks
  - Pad rest of buffer
- However, correct calculations still needed
  - e.g., wcsncat requires calculating the remaining space in the destination string...

27

## Future Microsoft

- Visual Studio 2005 have a new series of safe string manipulation functions
  - strcpy\_s()
  - strncpy\_s()
  - strcat\_s()
  - strlen\_s()
  - etc...
- Visual Studio 2005 (as of Beta 1) by default issues deprecation warnings on strcpy, strncpy, etc... Say goodbye to your old friends, they're too dangerous!

28

## Other Unsafe Functions: sprintf family

- ```
int sprintf(char *s, const char *format, /* args*/ ...);
```
- Buffer "s" can be overflowed
- ```
int snprintf(char *s, size_t n, const char *format, /* args*/ ...);
```
- Does not guarantee NULL-termination of s on some platforms (Microsoft, Sun)
  - MacOS X: NULL-termination guaranteed
  - Which is it on the server? Check with "man snprintf"
- ```
int vsprintf(char *str, const char * format, va_list ap);
```
- Buffer "str" can be overflowed

29

Gets, fgets

```
char *gets(char *str);
```

- Buffer "str" can be overflowed

```
char *fgets(char * str, int size, FILE * stream);
```

- Buffer "str" is not NULL-terminated if an I/O error occurs
- If an error occurs, returns NULL
- If end-of-file occurs before any characters are read, returns NULL also (and buffer is unchanged)
- Callers must use feof(3) and ferror(3) to determine which occurred.

30

Conclusion

- Buffer sizes should be passed as a parameter with every pointer
 - Applies to other buffer manipulations besides strings
- Need simple truncation detection

31

Preventing Buffer Overflows Without Programming

- Idea: make the heap and stack non-executable
 - Because many buffer overflow attacks aim at executing code in the data that overflowed the buffer
- Doesn't prevent "return into libc" overflow attacks
 - Because the return address of the function on the stack points to a standard "C" function (e.g., "system"), this attack doesn't execute code on the stack
- e.g., ExecShield for Fedora Linux (used to be RedHat Linux)

32

Canaries on a Stack

- Add a few bytes containing special values between variables on the stack and the return address.
- Before the function returns, check that the values are intact.
 - If not, there's been a buffer overflow!
 - Terminate program
- If the goal was a Denial-of-Service then it still happens
 - At least the machine is not compromised
- If the canary can be read by an attacker, then a buffer overflow exploit can be made to rewrite them
 - e.g., see string format vulnerabilities

33

Canary Implementations

- StackGuard
- Stack-Smashing Protector (SSP)
 - Formerly ProPolice
 - gcc modification
 - Used in OpenBSD
 - <http://www.tri.ibm.com/projects/security/ssp/>
- Windows: /GS option for Visual C++ .NET
- These can be useful when testing too!

34

Protection Using Virtual Memory Pages

- Page: A chunk (unit) of virtual memory
- POSIX systems have three permissions for each page.
 - PROT_READ
 - PROT_WRITE
 - PROT_EXEC
- Idea: manipulate and enforce these permissions correctly to defend against buffer overflows
 - Make injected code non-executable

35

Windows Execution Protection

- "NX" (No Execute)
- Windows XP service pack 2 feature
 - Somewhat similar to POSIX permissions
- Requires processor support
 - AMD64
 - Intel Itanium

36

Buffer Overflow Lab

- Create your own safe version of the strlen, strcpy, strcat
 - Name them mystrlen, mystrcpy and mystrcat
 - Pass buffer sizes for each pointer argument
 - Return 0 if successful, and 1 if truncation occurred
 - Other error codes if you wish
 - Make your implementation pass all test cases
 - `int mystrlen(const char *s, size_t s_len);`
 - In this case, return the string length, not zero or one.
 - `int mystrcpy(char * dst, const char * src, size_t dst_len, size_t src_len);`
 - `int mystrcat(char * s, const char * append, size_t s_len, size_t a_len);`

37

Things to Ponder

- What about 0 as source size? Error or not?
- What if "s" is NULL?
- What about overlapping buffers? Undefined everytime, or only in certain cases?
- What if reach the end in mystrlen?
- How efficient to make it -- how many passes at source string are made?
- What to check first?
- Reuse mystrlen within mystrcpy or mystrcat?
- Compare your implementations to `strl*`, `strsafe`, `safestr`, `str*_s`.