CMPSC 447
Spatial Errors

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Spatial Errors

- Most common errors permit access to memory outside of the expected region
  - These are called spatial errors
  - Access outside the expected “space”

- Most of these errors are permitted by simple programming flaws
  - Of the sort that you are not taught to avoid
  - Let’s see how such errors can be avoided

- Some of the changes are rather simple
Spatial Errors

• Many of the exploits that we have discussed are the result of spatial errors
Spatial Errors

- What were the fundamental causes from these two example?

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

struct test {
    char buffer[10];
    int (*fnptr)(char *, int);
};

int function(char *source) {
    char buffer[10];
    scanf(source, "%s", buffer);
    printf("buffer address: %p\n\n", buffer);
    return 0;
}

int main(int argc, char *argv[]) {
    function(argv[1]);
}

int main(int argc, char *argv[]) {
    int res = 0, flags = 0;
    struct test *a = (struct test*)malloc(sizeof(struct test));
    printf("buffer address: %p\n\n", a->buffer);
    a->fnptr = open;
    strcpy(a->buffer, source);
    res = a->fnptr(a->buffer, flags);
    printf("fd %d\n\n", res);
    return 0;
}

int main(int argc, char *argv[]) {
    int fd = open("stack.c", O_CREAT);
    function(argv[1]);
    exit(0);
}
```
Spatial Errors

- Operations that may handle string buffers unsafely

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

struct test {
    char buffer[10];
    int (*fnptr)( char *, int );
};

int function( char *source )
{
    char buffer[10];
    sscanf( source, "%s", buffer );
    printf( "buffer address: %p\n\n", buffer );
    return 0;
}

int main( int argc, char *argv[] )
{
    function( argv[1] );
}

int main( int argc, char *argv[] )
{
    int res = 0, flags = 0;
    struct test *a = (struct test*)malloc(sizeof(struct test));
    printf( "buffer address: %p\n\n", a->buffer );
    a->fnptr = open;
    strcpy( a->buffer, source );
    res = a->fnptr(a->buffer, flags);
    printf( "fd: %d\n\n", res );
    return 0;
}
```

```c
int main( int argc, char *argv[] )
{
    int fd = open("stack.c", O_CREAT);
    function( argv[1] );
    exit(0);
}
```
What Is Going Wrong?

- Both of these functions process "strings"?
  - What is a string?
What Is Going Wrong?

• Both of these functions process “strings”?
  ‣ What is a string?
    • Sequence of bytes terminating with a null byte

• Issues with strings
  ‣ Sequence may be longer than the memory region (bounds)
  ‣ Sequence may not be terminated by a null byte (bounds)
  ‣ Sequence may be terminated before expected (truncate)

• Each of these issues may lead to problems
  ‣ If undetected
• “Obvious” solution when using C is to always enforce bounds
Enforcing Bounds

• Two ways to enforce bounds
  ‣ Check memory bounds
  ‣ Automatic memory resizing

• Checking bounds
  ‣ Make sure that a memory operation is limited to the associated memory region

• Automatic resizing
  ‣ Resize the memory region to accommodate the memory required to satisfy the operation safely

• Typical functions do not check bounds or auto resize
Function w/o Bounds Checks

- **gets(3)** – reads input without checking. Don’t use it!
- **strcpy(3)** – `strcpy(dest, src)` – copies from `src` to `dest`
  ‣ If `src` longer than `dest` buffer, keeps writing!
- **strcat(3)** – `strcat(dest, src)` – appends `src` to `dest`
  ‣ If `src`+data-in-dest longer than `dest` buffer, keeps writing!
- **scanf()** family of input functions – many options
  ‣ `scanf(3), fscanf(3), sscanf(3), vscanf(3), vsscanf(3), vfscanf(3)`
  ‣ Default options don’t control max length (e.g., bare “%s”)
- Many other dangerous functions, e.g.:
  ‣ `realpath(3), getopt(3), getpass(3)`
  ‣ `streadd(3), strecpy(3), and strtrns(3)`
Bounds Checking Methods

• For each byte in the operation:
  • If oversized option (1) – **stop processing input**
    ‣ Reject and try again, or even halt program (may make DoS)
  • If oversized option (2) – **truncate data**
    ‣ Common approach, but has issues:
      • Terminates text “in the middle” at place of attacker’s choosing
      • *Way* better to truncate than to allow easy buffer overflow attack
      • But, **still could lead to problems?**
Truncation

- Issues with truncation
  - Terminates text “in the middle” at place of attacker’s choosing
  - Can strip off critical data, escapes, etc. at the end
  - Can break in the middle of multi-byte character
    - UTF-8 variable-width character encoding (> one byte sometimes)
    - UTF-16 usually 2 bytes/character, but can be 4 bytes/character
  - Some routines truncate & return indicator so you can stop processing input
Automatic Resizing

- For each byte in the operation:
- If oversized – **Auto-resize** – move string to a new memory region, if necessary
  - This is what most languages do automatically
    - other than C
    - Must deal with “too large” data
- By default, handling auto-resize manually in C can create issues
  - More code changes/complexity in existing C code
  - **Dynamic allocation** is manual in C, so adds new risks
    - Temporal errors – later
Traditional Solutions

- Depend mostly on `strncpy(3)`, `strncat(3)`, `sprintf(3)`
  - Can be hard to use correctly
- `char *strncpy(char *DST, const char *SRC, size_t LENGTH)`
  - Copy bytes from SRC to DST
  - Up to LENGTH bytes; if less, NULL-fills
- If LENGTH is the size of the DST memory region
  - Can fill memory region without null-terminator
  - Thus, does not guarantee creating a C string
  - Can truncate “in the middle,” leaving malformed data
  - Yet difficult to detect when it happens
Traditional Solutions

• Depend mostly on `strncpy(3), strncat(3), sprintf(3)`
  ‣ Can be hard to use correctly

• `char *strncat(char *DST, const char *SRC, size_t LENGTH)`
  ‣ Find end of string in DST (\0)
  ‣ Append up to LENGTH characters in SRC there

• If result is the size of the DST memory region
  ‣ Can fill memory region without null-terminator
    • Thus, does not guarantee creating a C string
  ‣ Can truncate “in the middle,” leaving malformed data
    • Yet difficult to detect when it happens
Strncpy/Strncat

- Fill buffer to `length` and return `reference` to result
  - No termination
Strncpy/Strncat

- Fill buffer to length and return reference to result
  - No termination
Strncpy/Strncat

- Fill buffer to length and return reference to result
  - No termination

  Truncation? How do we check?

  Only returns a reference to the start of the region
  - Telling us nothing about its state
Traditional Solutions

- Depend mostly on `strncpy(3)`, `strncat(3)`, `sprintf(3)`
  - Can be hard to use correctly

- `int sprintf(char *STR, const char *FORMAT, ...);`
  - Results put into STR
  - `FORMAT` can include length control information

- For example, `sprintf(DEST, "%.*s", MAXLEN, SRC);`
  - Like `strncpy`/`strncat`, does not guarantee null-termination
    - Does return the number of characters “printed”
  - Don’t forget the “.” – or no bounds checking
  - Using "*", then you can pass the maximum size (MAXLEN) as a parameter
There Is Help

- There are command APIs and options for existing commands that provide
  - Bound checking and notification of truncation
  - Auto-resizing without truncation
- The ones available now are a bit complex, but others have been proposed that are not yet widely available
Traditional Solution – That Works!

• Available now: `snprintf(3), vsnprintf(3)`
  ‣ Essentially the same functions, although arg format differs

• `int snprintf(char *S, size_t N, const char *FORMAT, ...)`;
  ‣ Writes output to buffer S up to N chars (bounds check)
  ‣ Always writes '\0' at end if N>=1 (terminate)
  ‣ Returns “length that would have been written” or negative if error (reports truncation or error)

• Thus, achieves goals of correct bounds checking
  ‣ Enforces bounds, ensures correct C string, and reports truncation or error
    • `len = snprintf(buf, buflen, "%s", original_value);`
    • `if (len < 0 || len >= buflen) … // handle error/truncation`
Traditional Solution – That Works!

- Available now: `snprintf(3), vsnprintf(3)`
  - Essentially the same functions, although arg format differs
- `int snprintf(char *S, size_t N, const char *FORMAT, ...);`
  - So, you should use this for safe programming today
  - Replaces `strcpy` and others directly
  - How do you use for `strcat`?
Traditional Solution – That Works!

- Available now: `snprintf(3), vsnprintf(3)`
  - Essentially the same functions, although arg format differs
- `int snprintf(char *S, size_t N, const char *FORMAT, ...);`
  - So, you should use this for safe programming today
  - Replaces `strcpy` and others directly
  - How do you use for `strcat`?
    - Need to find end of string to concatenate – set to S
    - Need to find the remaining size of the buffer – set to N
      - Do need to compute this correctly
    - At least this `snprintf/vsnprintf` will ensure null-termination at N
    - Don’t forget to check whether truncation or an error occurred
Traditional Solution – That Works!

- Available now: `snprintf(3), vsnprintf(3)`
  - Essentially the same functions, although arg format differs
- `int snprintf(char *S, size_t N, const char *FORMAT, ...);`
  - Kind of ugly to use
  - Other options?
Emerging Solutions

• Available in limited systems: `strlcpy(3), strlcat(3)`
  ‣ Similar to `snprintf` in semantics – from *BSD

• `Int strlcpy(char *DST, const char *SRC, size_t SIZE);`
  ‣ Looks more like `strncpy/strncat`; but less error prone
  ‣ Take SIZE of the buffer DST – rather than a length (`bounds`)
  ‣ Ensure null-termination relative to SIZE (`terminate`)
  ‣ Return number of bytes that would have been read (`truncate`)

• Relatively easy to use
  ‣ if `(strlcpy(dest, src, destsize) >= destsize) … // truncation!`
  ‣ Not universally available
Emerging Solutions

- Available in limited systems: `strcpy_s`, `strcat_s`
  - Similar to `snprintf` in semantics – from Microsoft

```
errno_t strcpy_s(char *restrict DST, rsize_t SIZE, const char *restrict SRC);
```

- Looks more like `strncpy/strncat`; but less error prone
- Take SIZE of the buffer DST – rather than a length (bounds)
- Checks constraints and returns if they are met (return 0)
- **Key constraint**: all bytes of SRC will fit in DST with \0

- Relatively easy to use
  - if (`strcpy_s(dest, src, destsize) < 0`) … // truncation!
  - Not universally available – multithreading limitations
Safe Bounds Checking

• Take the size of the buffer
  ‣ Limit length based on buffer size with termination

    ▶ Truncation: detect happens and how much truncation

    ▶ Return value enables determination whether any and how much truncation occurred to assess security
Auto Resize Solutions

- Available in limited systems: `asprintf(3)`, `vasprintf(3)`
  - Auto-resize versions of `sprint` and `v sprintf` (are unsafe)

- `int asprintf` (char **S, const char *FORMAT, ...);
  - Pass a pointer to a reference to a string buffer
  - Memory for the buffer and its reference are assigned to `S`
  - The memory allocated is sufficient to hold a proper C string of the value resulting from the processing of `FORMAT`
  - Returns # of bytes “printed”; -1 if error

- Simple to use; no termination, but need to “free”
  - `char *result = NULL;`
  - `asprintf(&result, “x=%s and y=%s\n”, x, y);`
Scanf and Friends

• What about other functions like scanf?
  ‣ scanf, fscanf, sscanf, vscanf, vsscanf, vfscanf – all unsafe by default
  ‣ Why?
    • char buffer[10];
    • scanf(buffer, “%s”);
Scanf and Friends

- What about other functions like scanf?
  - `scanf`, `fscanf`, `sscanf`, `vscanf`, `vsscanf`, `vfscanf` – all unsafe by default
  - Why?
    - char buffer[10];
    - `scanf(buffer, "%s")`;
  - Fortunately, these can be made safe quite easily
    - By leveraging auto-resizing option
Scanf and Friends

- What about other functions like scanf?
  - `scanf, fscanf, sscanf, vscanf, vsscanf, vfscanf` – all unsafe by default
  - Instead, use “%ms” to auto-resize
    - `char *buffer = NULL; // Must be set to NULL`
    - `scanf(buffer, "%ms");`
    - Allocates memory for the buffer dynamically to hold input safely – null-terminated, no truncation required

- Note: also, can use for other functions that process input like `getline`
  - Should check whether the function you use supports this option
Scanf in a Loop

• What happens when…
  ‣ Use “%ms” to auto-resize, but the function (scanf) is in a loop?
    • char *buffer = NULL;  // Must be set to NULL
    • while (TRUE) {
        scanf(buffer, “%ms”);
    }
Scanf in a Loop

• What happens when…
  ‣ Use “%ms” to auto-resize, but the function (scanf) is in a loop?
    • char *buffer = NULL; // Must be set to NULL
    • while (TRUE) {
      • scanf(buffer, “%ms”);
    • }

• Good news: The library knows and will keep resizing!
  ‣ If necessary – when the input is too big for the current buffer
Memory Object Copying

• What about just copying memory buffers?
  ‣ That are not strings (i.e., no termination)
  ‣ E.g., structure

• What would you normally do to copy a structure of an object of type A to a memory region of size N?
Traditional Solution

- Usual: `memcpy(3)`
  - Basic copying of memory values to a new region
- `void *memcpy(void *restrict DST, const void *restrict SRC, size_t N);`
- Copies N bytes from memory area SRC to memory area DST
  - Provides bounds checking
  - Does not consider null-termination
    - No null-terminator in this case, so that is OK
  - Does not consider truncation
    - Need to check for that
Memory and Strings

- POSIX now includes memccpy:

  void* memccpy(void* restrict DST, const void* restrict SRC, int C, size_t N);

  - Copies up to N bytes from SRC to DST until C is found, which is copied (e.g., C = '\0', so can use for strings),
  - Returns a pointer to just past the copy of the specified character C
  - or NULL if C was not found in the first N characters of SRC
  - So, can detect whether truncation occurred

- **Note**: You still have to calculate N (# bytes to copy)

- Adopted by C standard committee in 2019
Take Away

• The original versions of C string and memory functions did not consider spatial errors
  ‣ So, spatial errors have become common

• To ensure correct operation, we need to enforce memory region bounds
  ‣ Check bounds or automatically resize

• There are now several function APIs that enforce bounds
  ‣ Check bounds, ensure null-termination (if required), and report whether truncation occurred (to assess)