Bugs have consequences

NOW TYPE "23" INTO THE FIELD.
OK.

HIT "SUBMIT."
OK.

SO AS BUGS GO, THAT'S "BAD," RIGHT?
YES, THAT'S BAD.

HEY, DO YOU SMELL SMOKE?
An example

```c
// File: anagram.c
// Purpose: Read in a pair of strings and determine whether they are anagrams.
// Assumption: each input string is at most 64 characters long

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

void GetLetterCounts(char *str, int *cts);
int CompareLetterCounts(int ct1[], int ct2[]);

int main() {
    char buf1[65], buf2[65];
    int status, counts1[26] = {0}, counts2[26] = {0};

    status = scanf("%s %s", buf1, buf2);

    if (status == 2) { // no. of inputs correct
        GetLetterCounts(buf1, counts1);
        GetLetterCounts(buf2, counts2);

        if (CompareLetterCounts(counts1, counts2)) {
            printf("1: %s; 2: %s --> anagram\n", buf1, buf2);
        } else {
            printf("1: %s; 2: %s --> not anagram\n", buf1, buf2);
        }
        return 0;
    }
    return 0;
}
```

---More---(38%)
An example

// File: anagram.c
// Purpose: Read in a pair of strings and determine whether they are anagrams.
// Assumption: each input string is at most 64 characters long

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

void GetLetterCounts(char *str, int *cts);
int CompareLetterCounts(int ct1[], int ct2[]);

int main() {
    char buf1[65], buf2[65];
    int status, counts1[26] = {0}, counts2[26] = {0};

    status = scanf("%s %s", buf1, buf2);

    if (status == 2) { // no. of inputs correct
        GetLetterCounts(buf1, counts1);
        GetLetterCounts(buf2, counts2);

        if (CompareLetterCounts(counts1, counts2))
            printf("1: %s; 2: %s; --> anagram\n", buf1, buf2);
        else
            printf("1: %s; 2: %s --> not anagram\n", buf1, buf2);
    } else {
        printf("1: %s; 2: %s --> not anagram\n", buf1, buf2);
    }
    return 0;
}
An example

```bash
% cat input3
aaa

% .a.out < input3
%

Segmentation fault
%
% cat input4
aaa

% .a.out < input4
```
What happened

- The program made an assumption but did nothing to enforce it.
- This led to a buffer overflow that corrupted memory.

```c
// File: anagram.c
// Purpose: Read in a pair of strings and determine whether they an
// Assumption: each input string is at most 64 characters long

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

void GetLetterCounts(char *str, int *cts);
int CompareLetterCounts(int ct1[], int ct2[]);

int main() {
    char buf1[65], buf2[65];
    int status,
```
Buffer overflows

- C does not have array bounds checking
  - it is possible to write past the end of an array
  - this has the effect of overwriting other memory locations
  - depending on what gets corrupted, the program may:
    - show no effects
    - compute different results
    - crash
    - let someone else take control of your program (and possibly your computer)
Buffer overflows

```c
#include <stdio.h>

int main() {
    char buf[6];
    int x = 0;

    scanf("%s", buf);
    if (x == 0) {
        printf("x is zero\n");
    } else {
        printf("x is not zero: %d\n", x);
    }
    return 0;

% gcc -Wall buffer-overflow-1.c
% ./a.out
aaaaa
x is zero
% ./a.out
aaaaaaaaaaaaaaaaaa
x is not zero: 1633771873
% ```
Some examples of buffer overflows

Microsoft DirectShow Buffer Overflow in ActiveX Control Lets Remote Users Execute Arbitrary Code

Category: Application (Multimedia) > Microsoft DirectX
SecurityTracker Alert ID: 1022514
SecurityTracker URL: http://securitytracker.com/id?1022514
CVE Reference: CVE-2008-0015 (Links to External Site)
Updated: Aug 18 2009
Original Entry Date: Jul 6 2009
Impact: Execution of arbitrary code via network, User access via network
Fix Available: Yes Exploit Included: Yes Vendor Confirmed: Yes
Advisory: Microsoft Security Bulletin
Description: A vulnerability was reported in Microsoft DirectShow. A remote user can cause arbitrary code to be executed on the target user's system.

A remote user can create specially crafted HTML that, when loaded by the target user, will invoke the Microsoft Video ActiveX control and trigger a buffer overflow in "msvidctl.dll" to execute arbitrary code on the target system. The code will run with the privileges of the target user.

The CSLID of the affected control is: 0056AC62-BF2E-4C9A-A289-A63F772D48CF

This vulnerability is being actively exploited

Exploit code is publicly available


The vendor was notified in Spring (northern hemisphere) 2008 by Ryan Smith and Alex Wheeler with IBM ISS X-Force.

Impact: A remote user can create HTML that, when loaded by the target user, will execute arbitrary code on the target user's system.
Solution: The vendor has issued the following fixes as part of a cumulative update:

Microsoft Windows 2000 Service Pack 4:
Some examples of buffer overflows

Remote buffer overflow bug bites Linux Kernel Driver Wrapper

A remote buffer overflow vulnerability in the Linux Kernel could be exploited by attackers to execute code or cripple affected systems, according to a Gentoo bug report that just became public.

The flaw could allow malicious hackers to launch arbitrary code with kernel-level privileges. This could lead to complete system compromise or, in some cases if an exploit fails, result in denial-of-service attacks.

This from the Gentoo bug report:

- Anders Kaseorg discovered that ndiswrapper did not correctly handle long ESSIDs. If ndiswrapper is in use, a physically near-by attacker could generate specially crafted wireless network traffic and crash the system, leading to a denial of service.

Read more: http://blogs.zdnet.com/security/?p=2121

Impact: A remote user can create HTML that, when loaded by the target user, will execute arbitrary code on the target user's system.

Solution: The vendor has issued the following fixes as part of a cumulative update:

Microsoft Windows 2000 Service Pack 4:
Some examples of buffer overflows

Xvid Codec Initialization Logic Buffer Overflow

Notification Type: IBM Internet Security Systems Protection Advisory

Notification Date: June 01, 2009

Notification Version: 1.0

Name: Xvid Codec Initialization Logic Buffer Overflow

Public disclosure/in the wild date: May 29, 2009 (vuln disclosure)

CVE: CVE-2009-0894

Description: Xvid video codec is vulnerable to a heap-based buffer overflow, caused by improper bounds checking. By persuading a victim to open a specially-crafted movie file, a remote attacker could overflow a buffer to corrupt memory and execute arbitrary code on the affected system with privileges of the victim.

Discoverer: This vulnerability was discovered and researched by John McDonald and Mark Dowd of the IBM X-Force.
How buffer overflows are exploited

• The attacker provides an input string that is too long for an array (buffer) it is written to
  – the buffer overflows
  – contents of neighboring memory locations are overwritten

• By careful selection of the input string, the attacker can control exactly what the neighboring locations are overwritten with

• This can allow the attacker to force the victim’s computer to execute arbitrary code
Example: “Stack Smashing”

```c
int foo( ... ) {
    char buf[32];
    ...
    scanf("%s", buf);
    ...
    return;
}
```
Example: “Stack Smashing”

```c
int foo( ... ) {
    char buf[32];
    ...
    scanf("%s", buf);
    ...
    return;
}
```
Example: “Stack Smashing”

```c
int foo( ... ) {
    char buf[32];
    ...
    scanf("%s", buf);
    ...
    return;
}
```
Example: “Stack Smashing”

```c
int foo( ... ) {
    char buf[32]
    ... 
    scanf("%s", buf);
    ... 
    return;
}
```

This function can be vulnerable to stack smashing if the `scanf` function is not properly handled. The attacker can inject data into the `buf` buffer, potentially overwriting the return address with code injected by the attacker, leading to a buffer overflow attack.
Controlling amount of input read

- [f]scanf lets you specify a *maximum field width* before the conversion specifier
  - max. field width: optional decimal integer
  - if specified: reading of characters stops when this width is reached
  - for strings: field width does not include terminating ‘\0’
Controlling the amount read

```c
// File: anagram-1.c
// Purpose: Read in a pair of strings and determine whether they are anagrams.
// Assumption: each input string is at most 64 characters long

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

void GetLetterCounts(char *str, int *cts);
int CompareLetterCounts(int ct1[], int ct2[]);

int main() {
    char buf1[65], buf2[65];
    int status, counts1[26] = {0}, counts2[26] = {0};

    status = scanf("%64s %64s", buf1, buf2);

    if (status == 2) { // no. of inputs correct
        GetLetterCounts(buf1, counts1);
        GetLetterCounts(buf2, counts2);

        if (CompareLetterCounts(counts1, counts2)) {
            printf("1: %s; 2: %s -> anagram\n", buf1, buf2);
        } else {
            printf("1: %s; 2: %s --> not anagram\n", buf1, buf2);
        }
        return 0;
    }
}
```

--More---(38%)
Controlling the amount read

```c
// File: anagram-1.c
// Purpose: Read in a pair of strings and determine whether they are anagrams.
// Assumption: each input string is at most 64 characters long

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

void GetLetterCounts(char *str, int *cts);
int CompareLetterCounts(int ct1[], int ct2[]);

int main() {
    char buf1[65], buf2[65];
    int status, counts1[26] = {0}, counts2[26] = {0};

    status = scanf("%64s %64s", buf1, buf2);

    if (status == 2) { // no. of inputs correct
        GetLetterCounts(buf1, counts1);
        GetLetterCounts(buf2, counts2);

        // Compare the letter counts
        // ... (comparison code here)
    }
}
```

% gcc -Wall anagram-1.c
% ./a.out < input3
1: aaa; 2; xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx --> not anagram

% % ./a.out < input4
1: aaa; 2; xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx --> not anagram
```
Guarding scanf isn’t enough

```c
#include <stdio.h>
#include <string.h>

struct s {
    char name[12];
    int value;
} v0;

void foo(char *buf0) {
    v0.value = 0;
    strcpy(v0.name, buf0);
    return;
}

int main() {
    char instring[65];
    scanf("%64s", instring);
    foo(instring);
    printf("name = %s, value = %d\n", v0.name, v0.value);
    return 0;
}
```

- scanf is being guarded
- and yet memory gets corrupted!
Guarding scanf isn’t enough

• Any function that writes an unbounded amount into a memory region is potentially a problem

• Common culprits:
  – string library routines: strcpy, strcat
  – input routines: scanf, gets
  – output routines: sprintf

<table>
<thead>
<tr>
<th>unguarded</th>
<th>guarded</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcpy</td>
<td>strncpy</td>
</tr>
<tr>
<td>strcat</td>
<td>strncat</td>
</tr>
<tr>
<td>gets</td>
<td>fgets</td>
</tr>
<tr>
<td>sprintf</td>
<td>snprintf</td>
</tr>
</tbody>
</table>
Example: strcpy vs. strncpy

```
#include <string.h>

char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
```

**DESCRIPTION**

The `strcpy()` function copies the string pointed to by `src`, including the terminating null byte (`'\0'`), to the buffer pointed to by `dest`. The strings may not overlap, and the destination string `dest` must be large enough to receive the copy.

The `strncpy()` function is similar, except that at most `n` bytes of `src` are copied. **Warning:** If there is no null byte among the first `n` bytes of `src`, the string placed in `dest` will not be null terminated.

If the length of `src` is less than `n`, `strncpy()` pads the remainder of `dest` with null bytes.

A simple implementation of `strncpy()` might be:

```
char*
strncpy(char *dest, const char *src, size_t n){
```
It’s easy to get careless

```c
#include <stdio.h>
#include <string.h>

char name[4];

void foo(char *buf0) {
    int n0 = 0x65666768, n1 = 0x69707172, n2 = 0x73747576;
    char str0[4];
    strncpy(str0, buf0, 4);
    strcpy(name, str0);
    return;
}

int main(int argc, char **argv) {
    char buf[65];
    scanf("%64s", buf);
    foo(buf);
    printf("name: %s\n", name);
    return 0;
}
```

str0[ ], name[ ] are the same size: 4 bytes

we’ve made sure we copied at most 4 bytes into str0

therefore it’s OK to use just strcpy() to copy str0 into v0.name

right?
It’s easy to get careless

```bash
% gcc -Wall -g buffer-overflow-3.c
buffer-overflow-3.c: In function 'foo':
buffer-overflow-3.c:7: warning: unused variable 'n2'
buffer-overflow-3.c:7: warning: unused variable 'n1'
buffer-overflow-3.c:7: warning: unused variable 'n0'
%
% ./a.out
abc
name: abc
%
% ./a.out
abcdefghij
name: abcdfgherqpvuts f
%  
```

input

output

??!
What happened?

```
#include <string.h>

char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
```

**DESCRIPTION**

The `strcpy()` function copies the string pointed to by `src`, including the terminating null byte (`\0`), to the buffer pointed to by `dest`. The strings may not overlap, and the destination string `dest` must be large enough to receive the copy.

The `strncpy()` function is similar, except that at most `n` bytes of `src` are copied. **Warning:** If there is no null byte among the first `n` bytes of `src`, the string placed in `dest` will not be null terminated.

If the length of `src` is less than `n`, `strncpy()` pads the remainder of `dest` with null bytes.

A simple implementation of `strncpy()` might be:

```c
char*
strncpy(char *dest, const char *src, size_t n){
```
What happened?

strncpy didn’t NULL-terminate the string, and this caused the string in str0[] to be much longer than intended.
Why is this a problem?

• Any time a buffer can overflow with user-supplied data, the program has a huge security hole.

• *Example*: variation on the previous problem

```c
int n0, n1, n2;
char buf1[4], buf2[64];
...
scanf("%d %d %d %64s", n0, n1, n2, buf2);
strncpy(buf1, buf2, 4);
```

- n0, n1, n2, buf1 may be in contiguous memory (compiler dependent)
- non-NULL-terminated `strncpy()` can create a user-controlled string that is longer than intended
- this can set the stage for a buffer-overflow exploit
The moral of the story

• Careless handling of user-supplied data can get you into trouble
  – read man pages carefully
  – learn which input/string routines don’t guarantee NULL-terminated strings

• Make sure buffers are NULL-terminated
  – allocate space for the terminating NULL character
  – declare buffers as
    char   buf[MAXSIZE + 1];
Is that enough?

```c
#define MAXSIZE 4

void foo(char *string, int n) {
    char buf[MAXSIZE+1] = {0};  // O-initialized
    if (n > MAXSIZE) {
        n = MAXSIZE;
    }
    strncpy(buf, string, n);
    printf("@@ buf = %s\n", buf);
}

int main() {
    int n;
    char string[65];
    scanf("%d %54s", &n, string);
    foo(string, n);
    return 0;
}
```

- allocating space for terminating NULL
- initializing the array
- checking how many characters to copy
- using "safe" string functions
- eating our vegetables
- guarding scanf buffer
Is that enough?

```c
% gcc -Wall buffer-overflow-4.c
% ./a.out
3  abcdefghij  ✓
@@@  buf = abc
%
%
% ./a.out
27  abcdefghij  ✓
@@@  buf = abcd
%
%
% ./a.out
-1  abcdefghij  ✗
Segmentation fault
%  
```
What happened?

We forgot to check the input!

• Signature for strncpy:

```c
char *strncpy(char *dest, char *src, size_t n)
```

– `size_t` is an unsigned type
– a negative number, treated as an unsigned value, looks like a very big number
  • e.g., on a 32-bit machine, -1 (signed) = 0xffffffff = $2^{32}-1$ (unsigned)
– `strncpy()` copies over a very large no. of bytes
  • this corrupts memory and causes the seg fault