A Puzzle

What is the output of this program?

```
char x, y;
x = -128;
y = -x;

if (x == y) puts("1");
if ((x - y) == 0) puts("2");
if ((x + y) == 2 * x) puts("3");
if (((char)(-x) + x) != 0) puts("4");
if (x != -y) puts("5");
```

CERT

Producing Secure Programs in C and C++

Information Science & Technology Colloquium Series

Robert C. Seacord

String Agenda

Strings

Common errors using NTBS

String Vulnerabilities

Mitigation Strategies

Summary

Strings

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

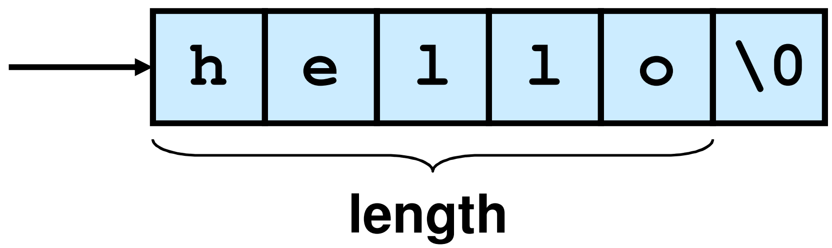
- command-line arguments
- environment variables
- console input

Software vulnerabilities and exploits are caused by weaknesses in

- string representation
- string management
- string manipulation

Null-Terminated Byte Strings (NTBS)

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



Null-terminated byte 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 number of bytes required to store a string is the length + 1.

Null-terminated byte strings are implemented as arrays of “plain”, signed, unsigned characters.

Arrays

One of the problem with arrays is determining the size:

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

size is 4

size is 28

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

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]);
  ...
}
```

String Agenda

Strings

Common errors using NTBS

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
- improper data sanitization

Unbounded String Copies

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

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

C++ Unbounded Copy

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

```
#include <iostream>
using namespace std;
int main() {
  char buf[12];
  cin >> buf;
  cout << "echo: " << buf << endl;
}
```

Simple Solution

Set width field to maximum input size.

```
#include <iostream>
using namespace std;

int main(void) {
   char buf[12];

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

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.

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;
}
```

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[]`** 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 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.

String Agenda

Strings

Common errors using NTBS

String Vulnerabilities

- Program Stacks
- Buffer Overflow
- Code Injection
- Arc Injection

Mitigation Strategies

Summary

Program Stacks

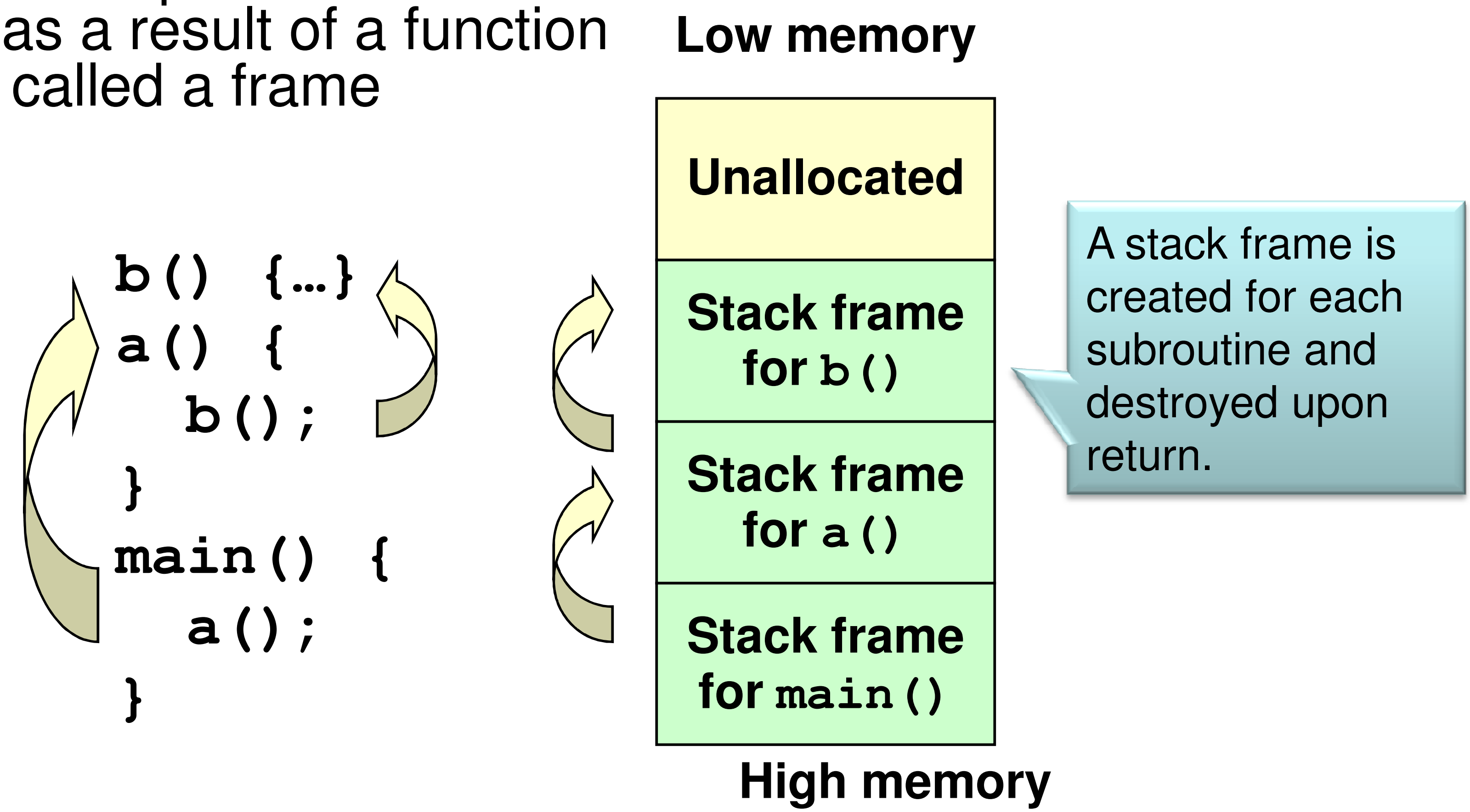
A program stack is used to keep track of program execution and state by storing

- 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



Stack Frames

The stack is used to store

- 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 (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

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

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
```

Restores the stack pointer

```
pop ebp
```

Restores the frame pointer

```
ret
```

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
  if (!strcmp(Password,"goodpass")) return(true); // Password Good
  else return(false); // Password Invalid
}

int main(void) {
  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(void) {
 char Password[12];

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

Stack

ESP

Storage for Password (12 Bytes)
Caller EBP – Frame Ptr main (4 bytes)
Return Addr Caller – main (4 Bytes)
Storage for **`PwStatus`** (4 bytes)
Caller EBP – Frame Ptr OS (4 bytes)
Return Addr of main – OS (4 Bytes)
…

Note: The stack 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)
…

Sample Program Runs

Run #1 Correct Password

C:\WINDOWS\System32\cmd.exe

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

C:\BufferOverflow\Release>_
```

Run #2 Incorrect Password

C:\WINDOWS\System32\cmd.exe

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

C:\BufferOverflow\Release>
```

String Agenda

Strings

Common errors using NTBS

String Vulnerabilities

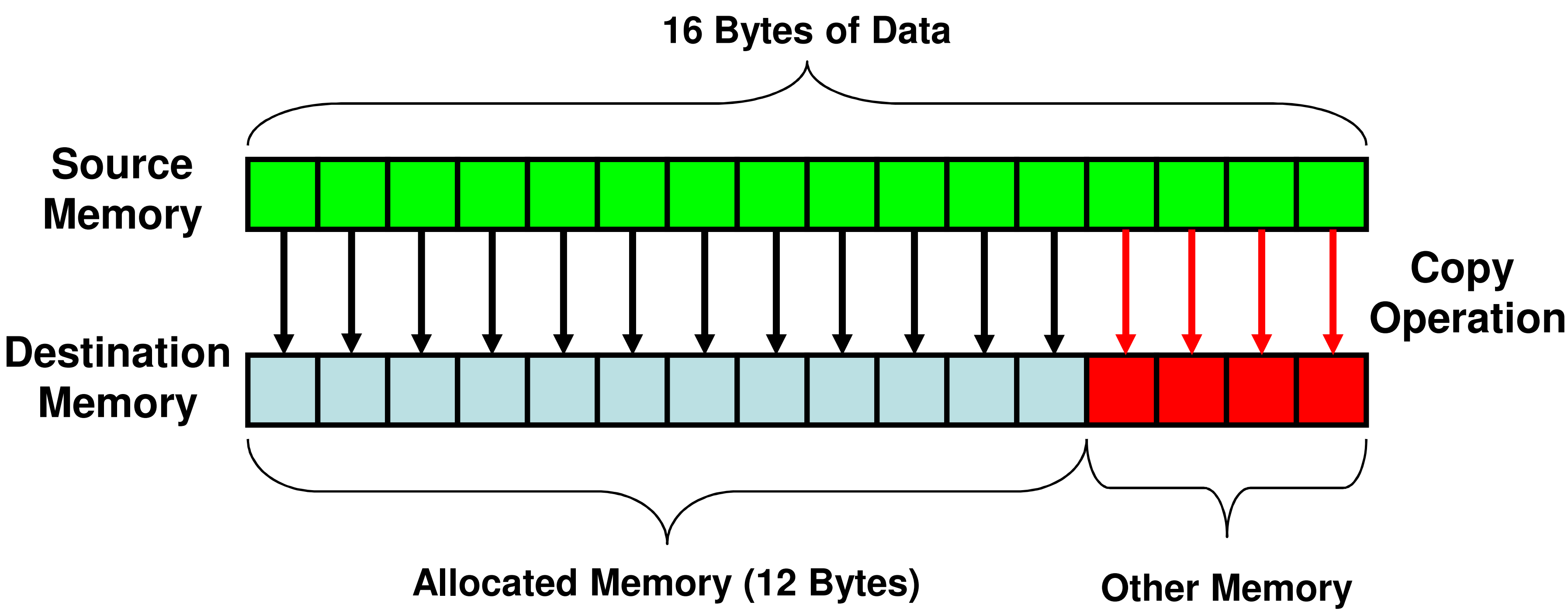
- Program stacks
- Buffer overflows
- Code Injection
- Arc Injection

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 *

The Buffer Overflow 2

EIP

```
bool IsPasswordOK(void) {
 char Password[12];
 gets(Password);
 if (!strcmp(Password, "goodpass"))
      return(true);
 else return(false)
}
```

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.

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 errors using NTBS

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.

Overflow Buffer

Fill with arbitrary data up to the return code.

```
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 "
```

Overwrite Return Code

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

```
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 "
```

Everything after the return code is shell 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 $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.

This is 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.

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.

String Agenda

Strings

Common errors using NTBS

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 Program

```
#include <string.h>

int get_buff(char *user_input){
  char buff[40];
  memcpy(buff, user_input,
         strlen(user_input)+1);
  return 0;
}

int main(int argc, char *argv[]){
  get_buff(argv[1]);
  return 0;
}
```

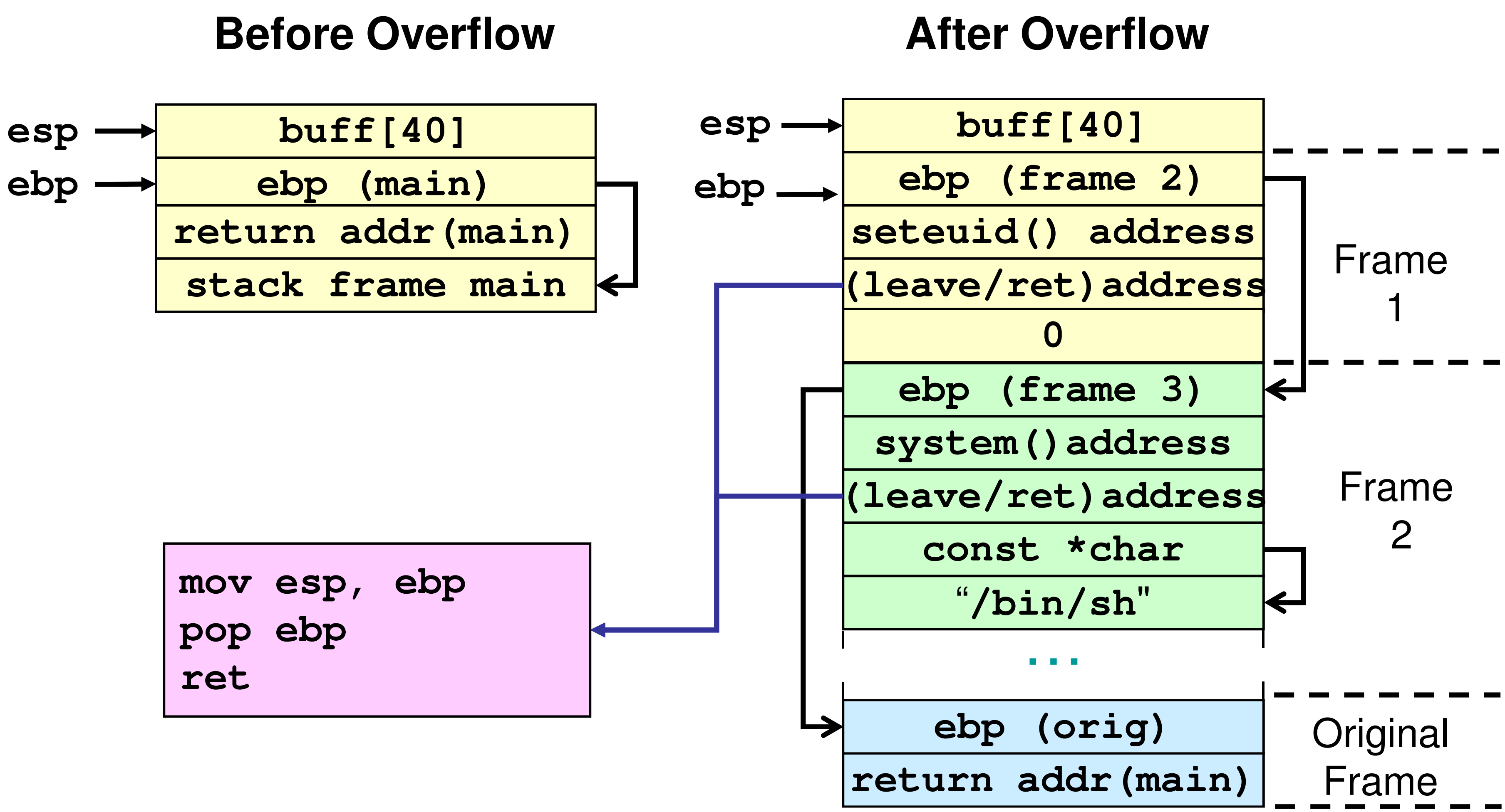
Exploit

Overwrites return address with address of existing function.

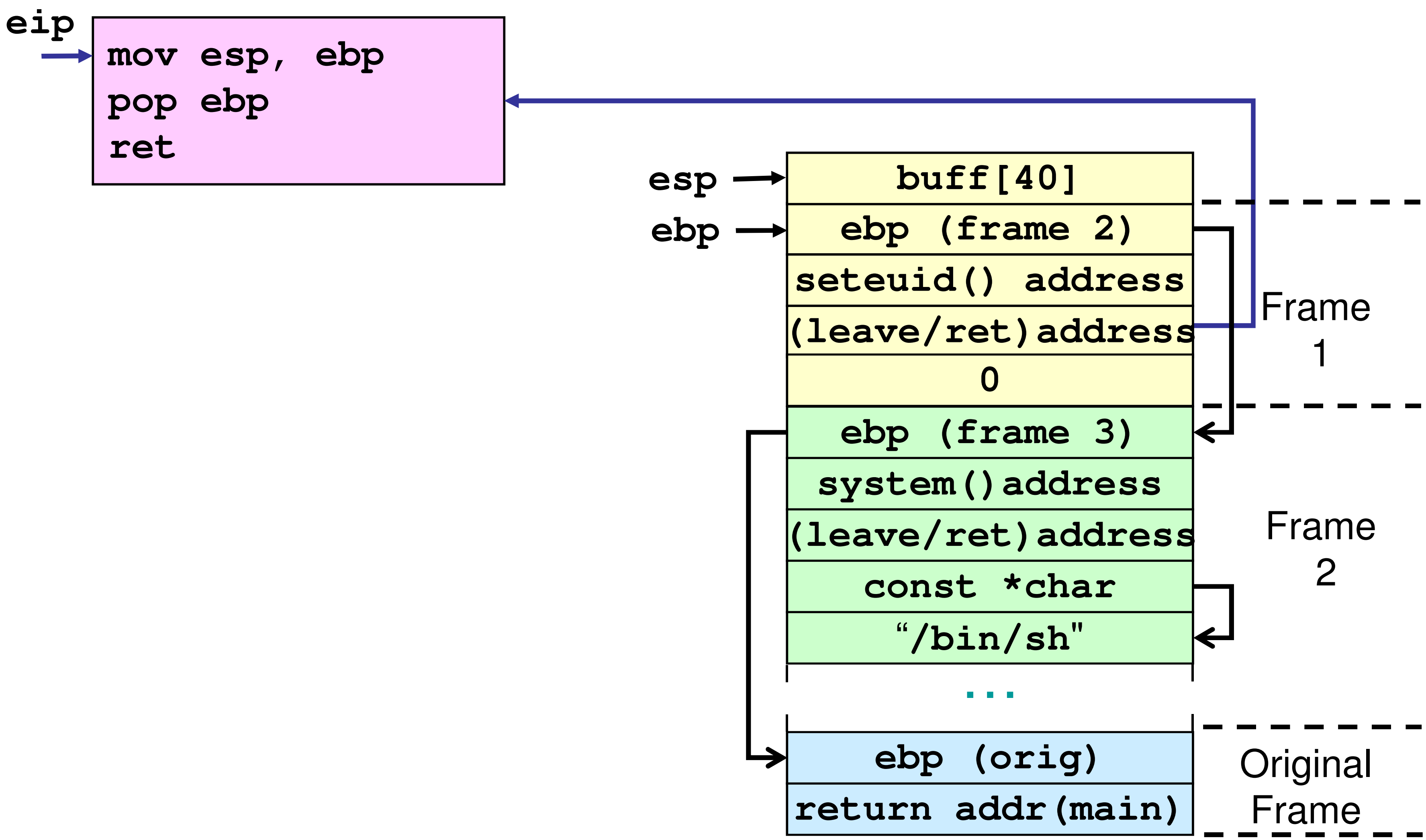
Creates stack frames to chain function calls.

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

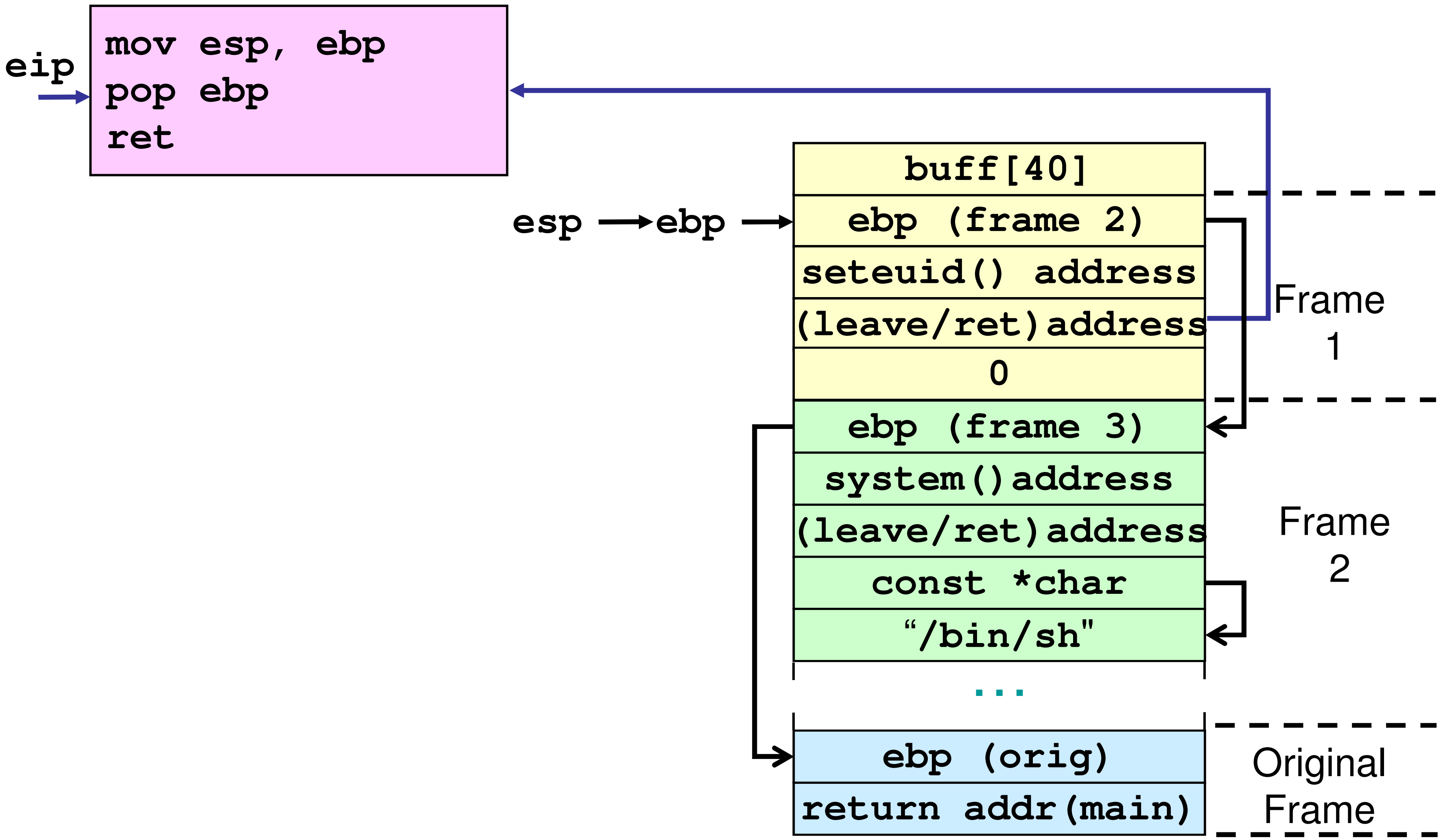
Result of `memcpy()` in `get_buff()`



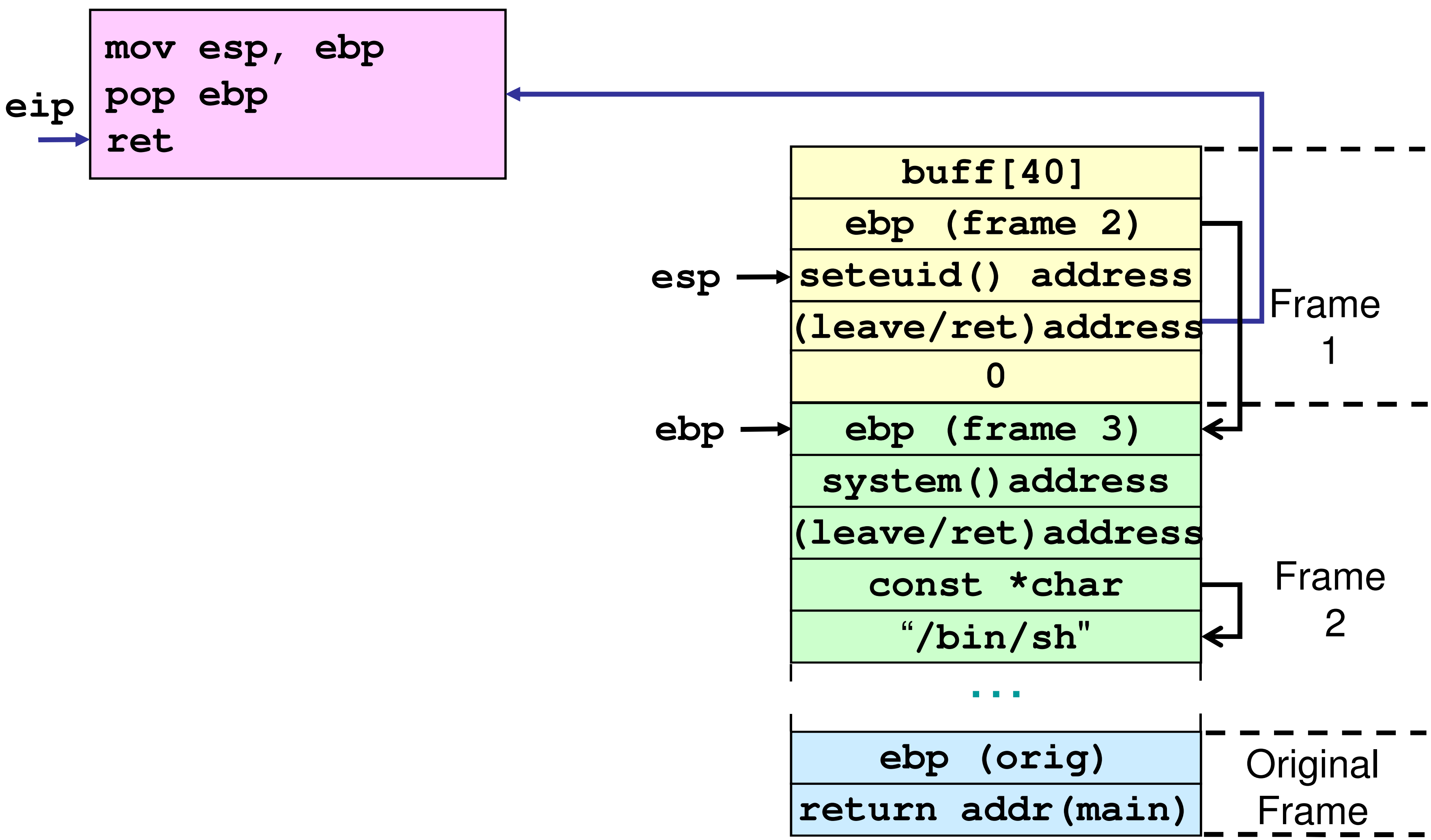
`get_buff()` Returns



get_buff() Returns



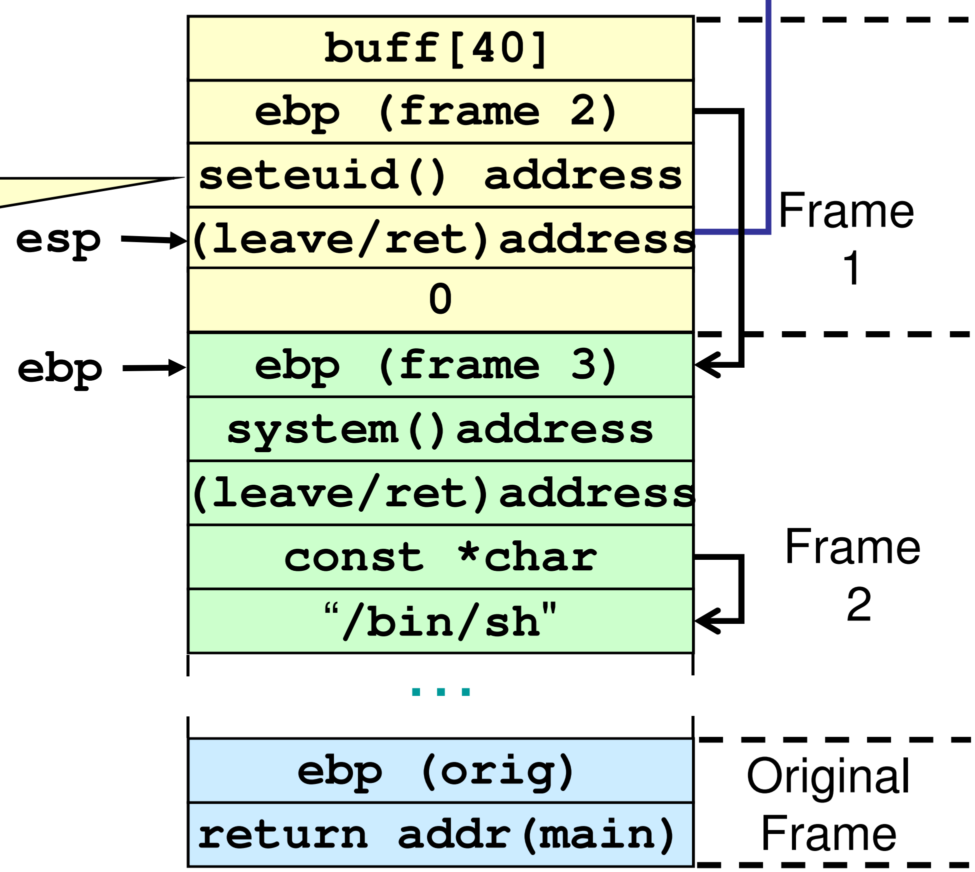
`get_buff()` Returns



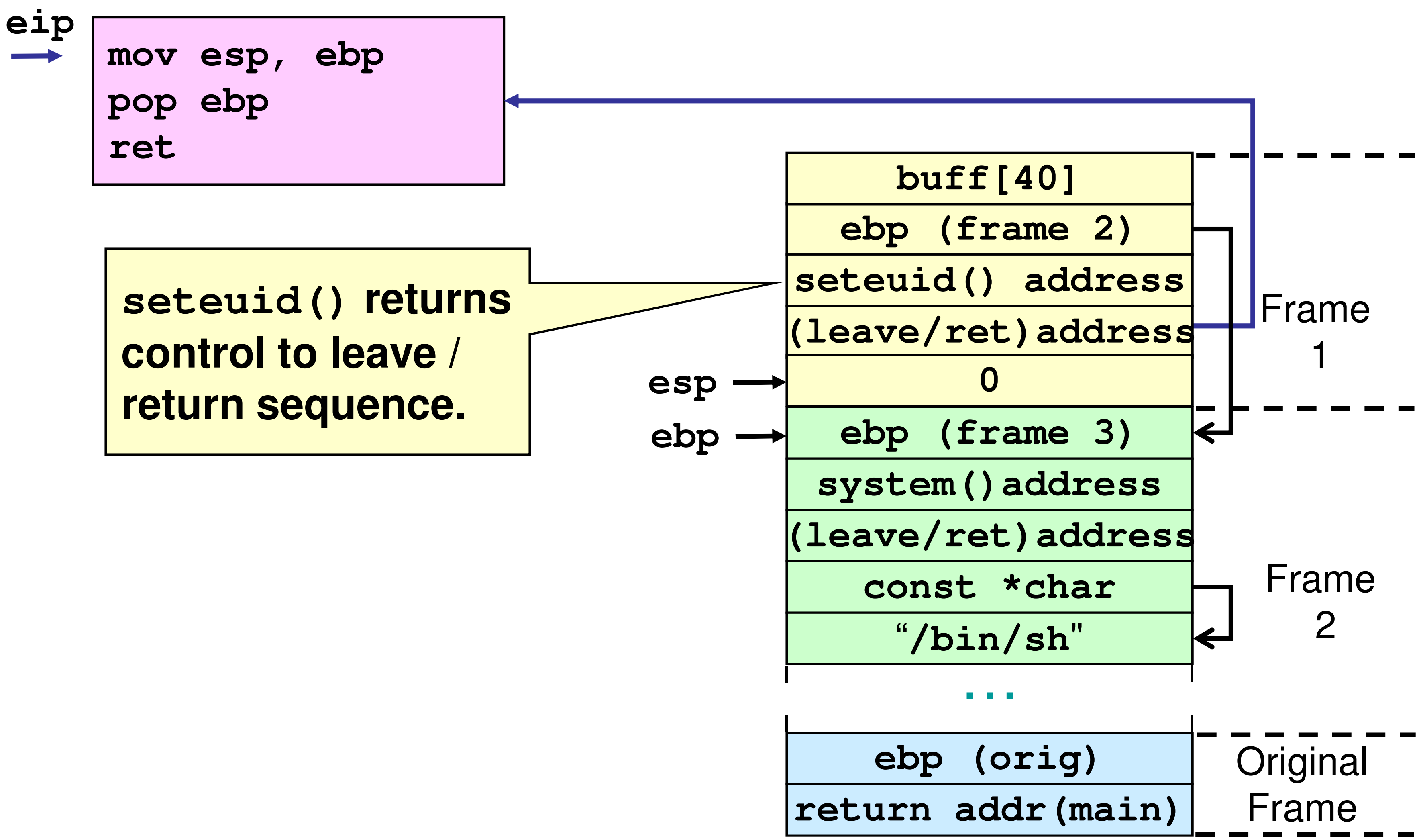
`get_buff()` Returns

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

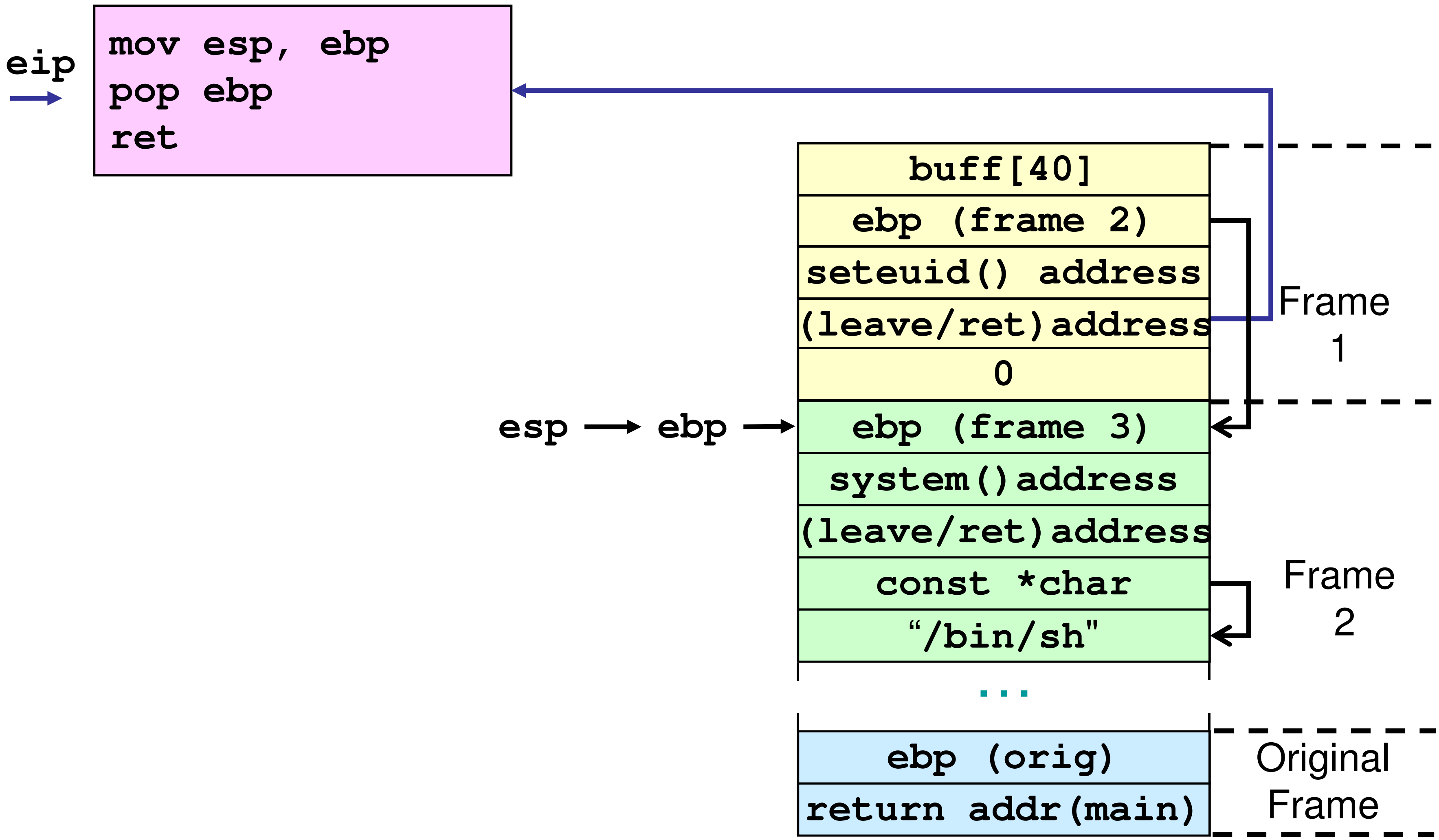
`ret` instruction transfers control to **`seteuid()`**.



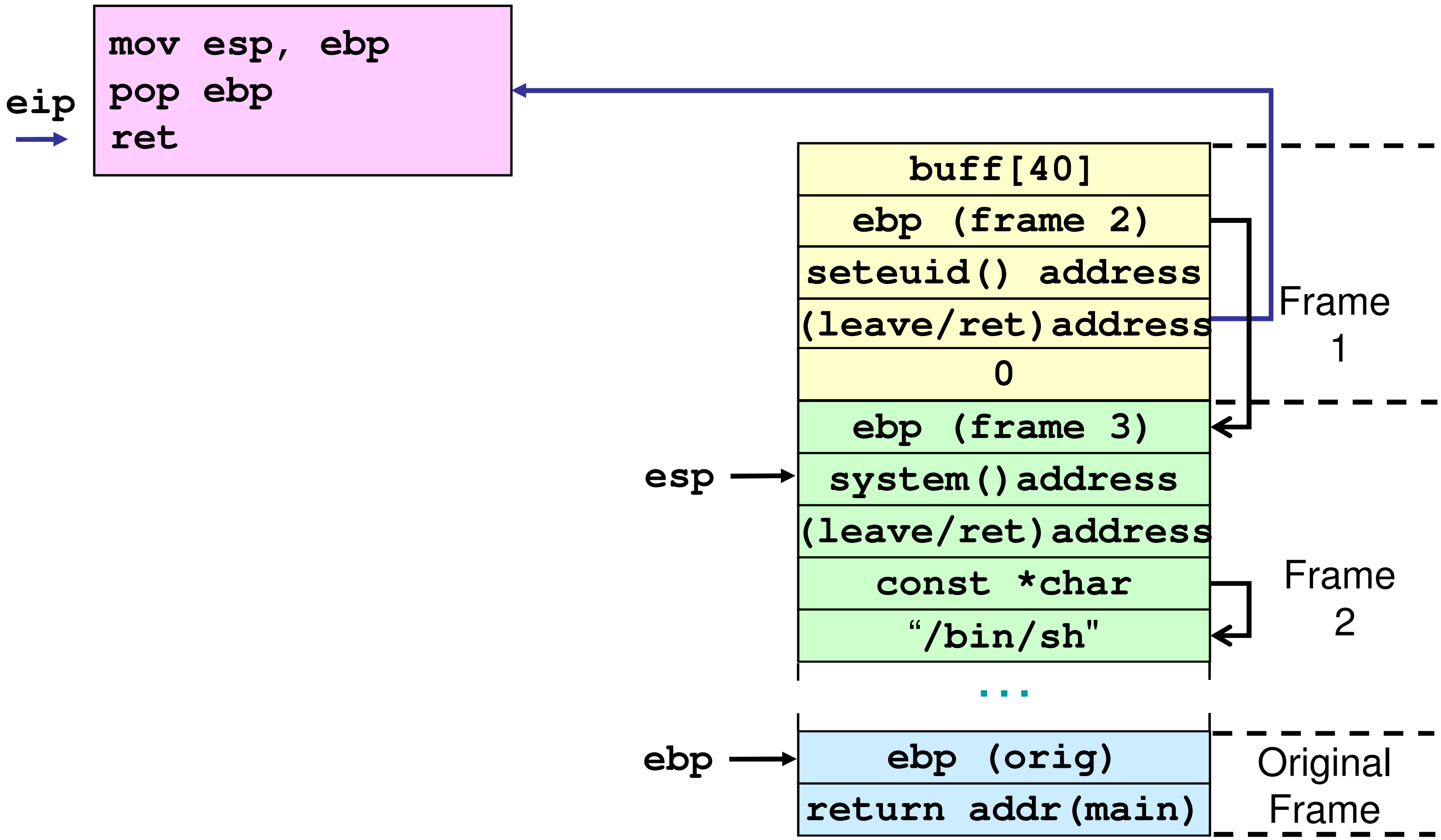
`seteuid()` Returns



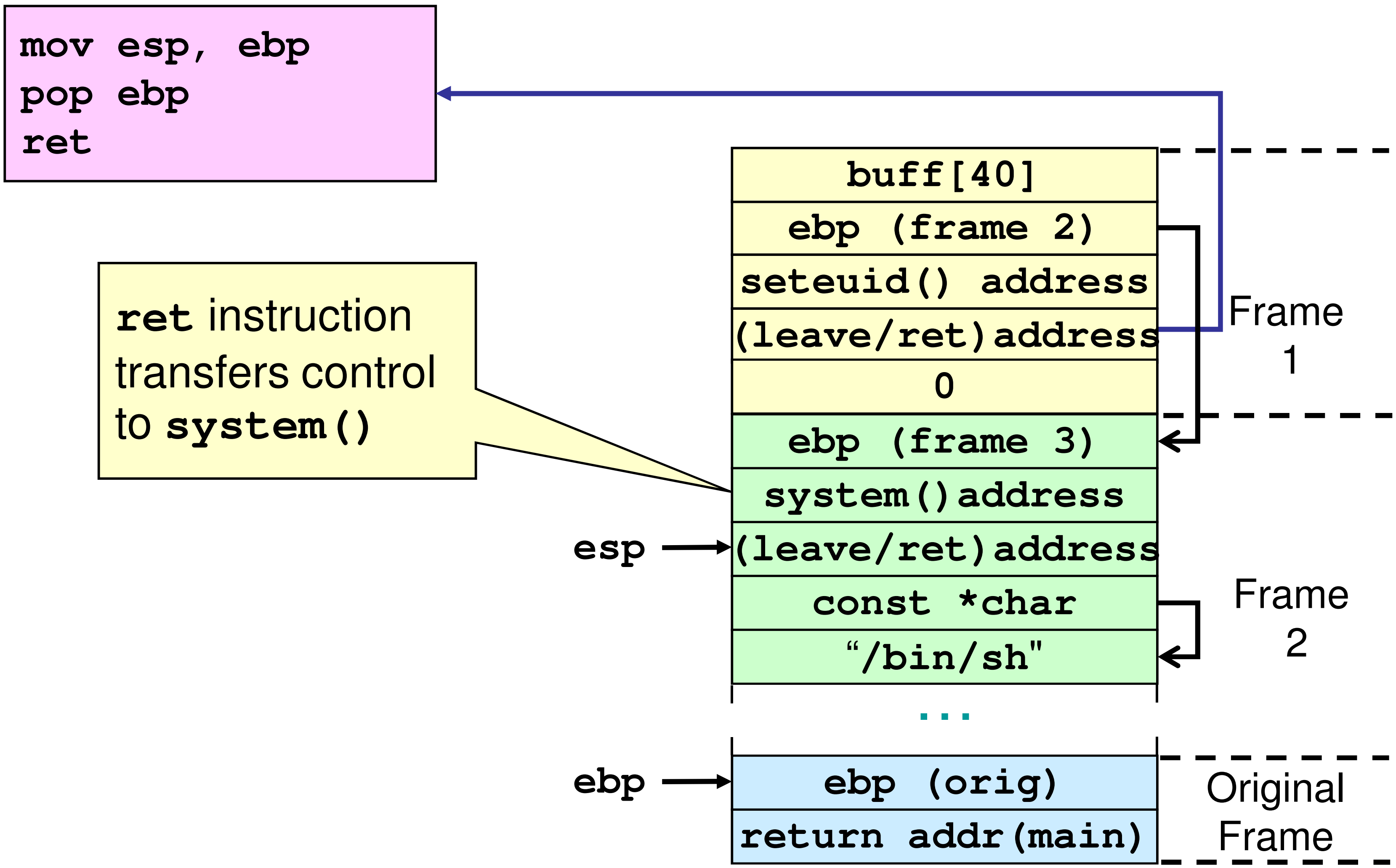
`seteuid()` Returns



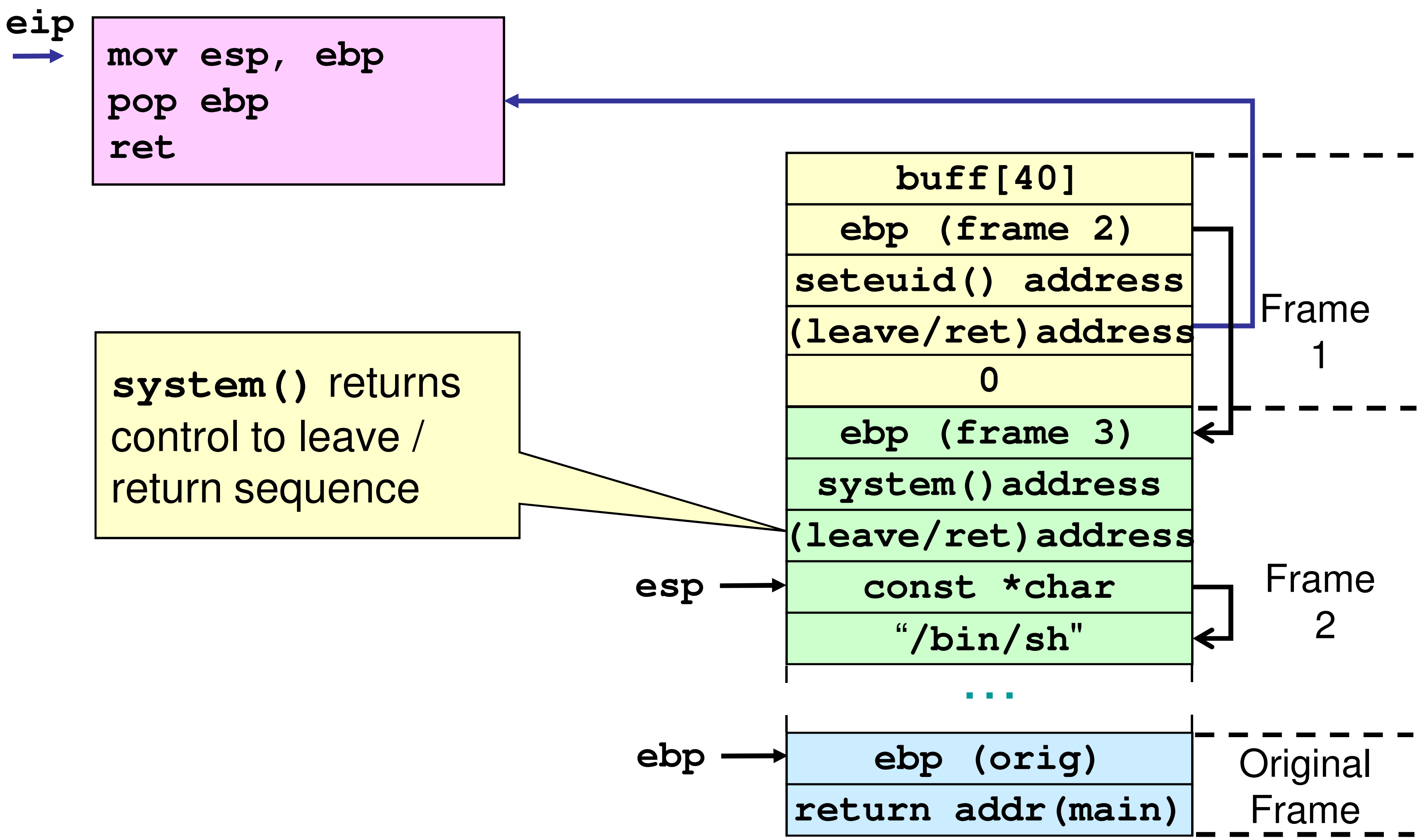
seteuid() Returns



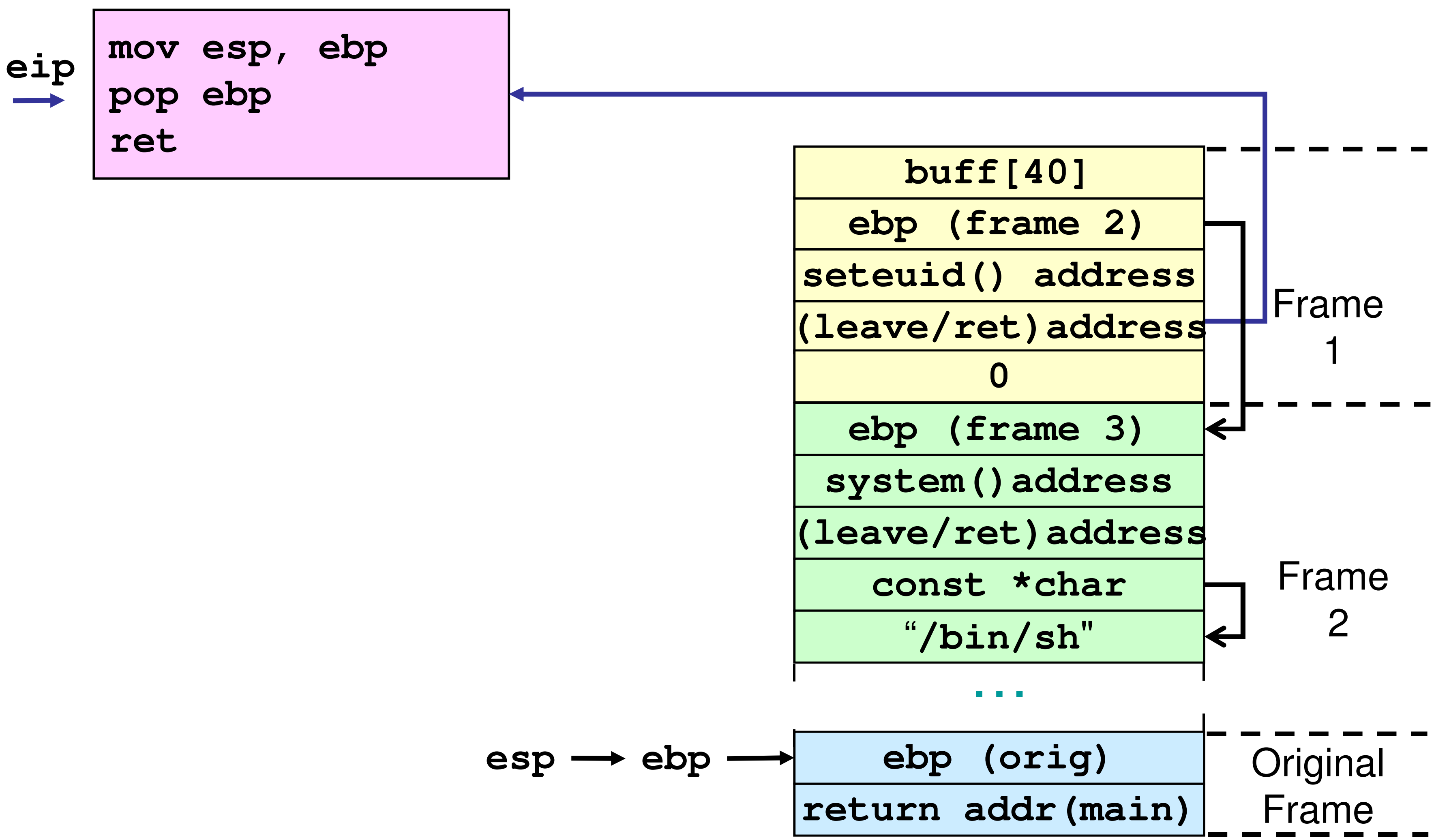
`seteuid()` Returns



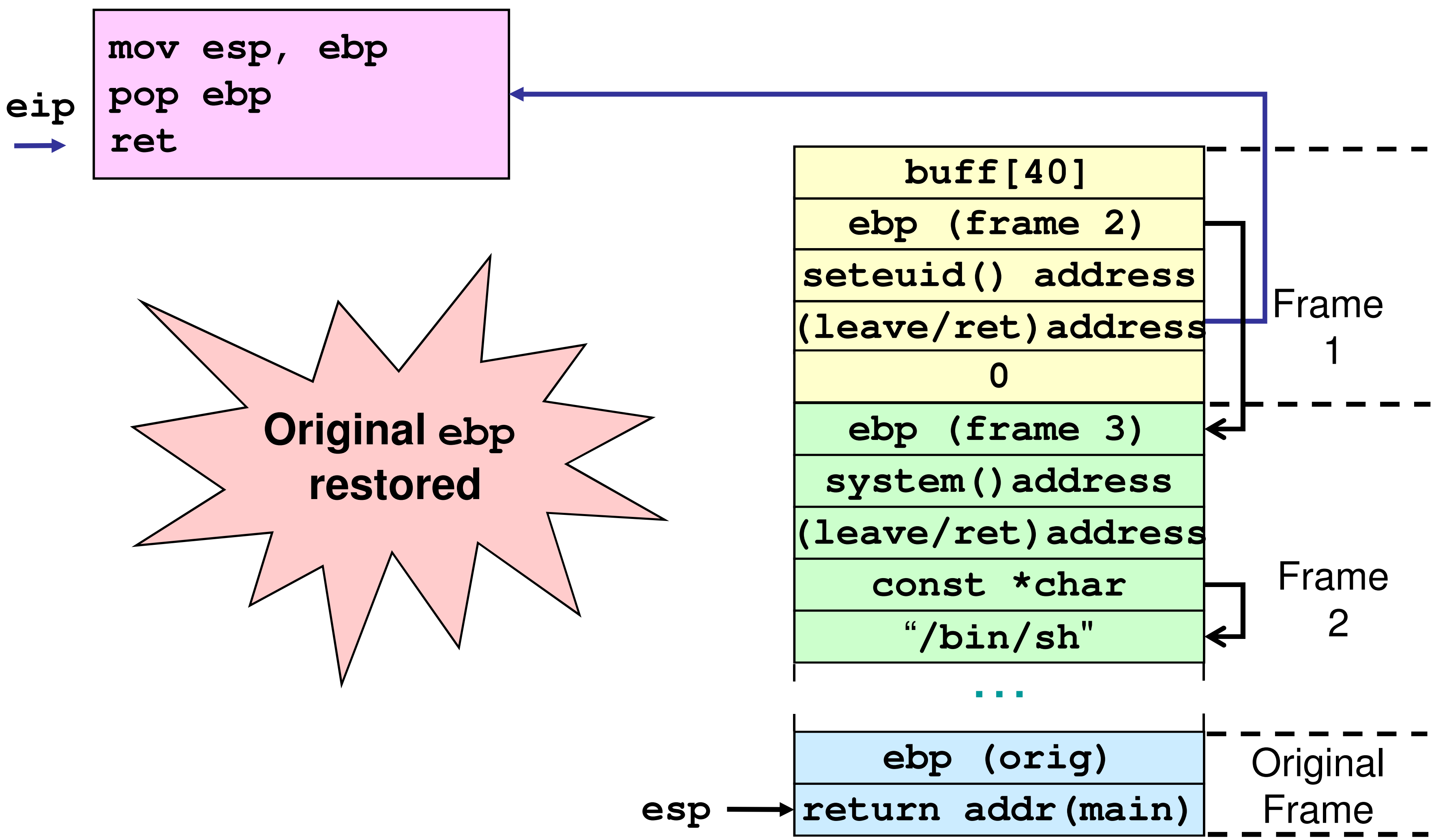
`system()` Returns



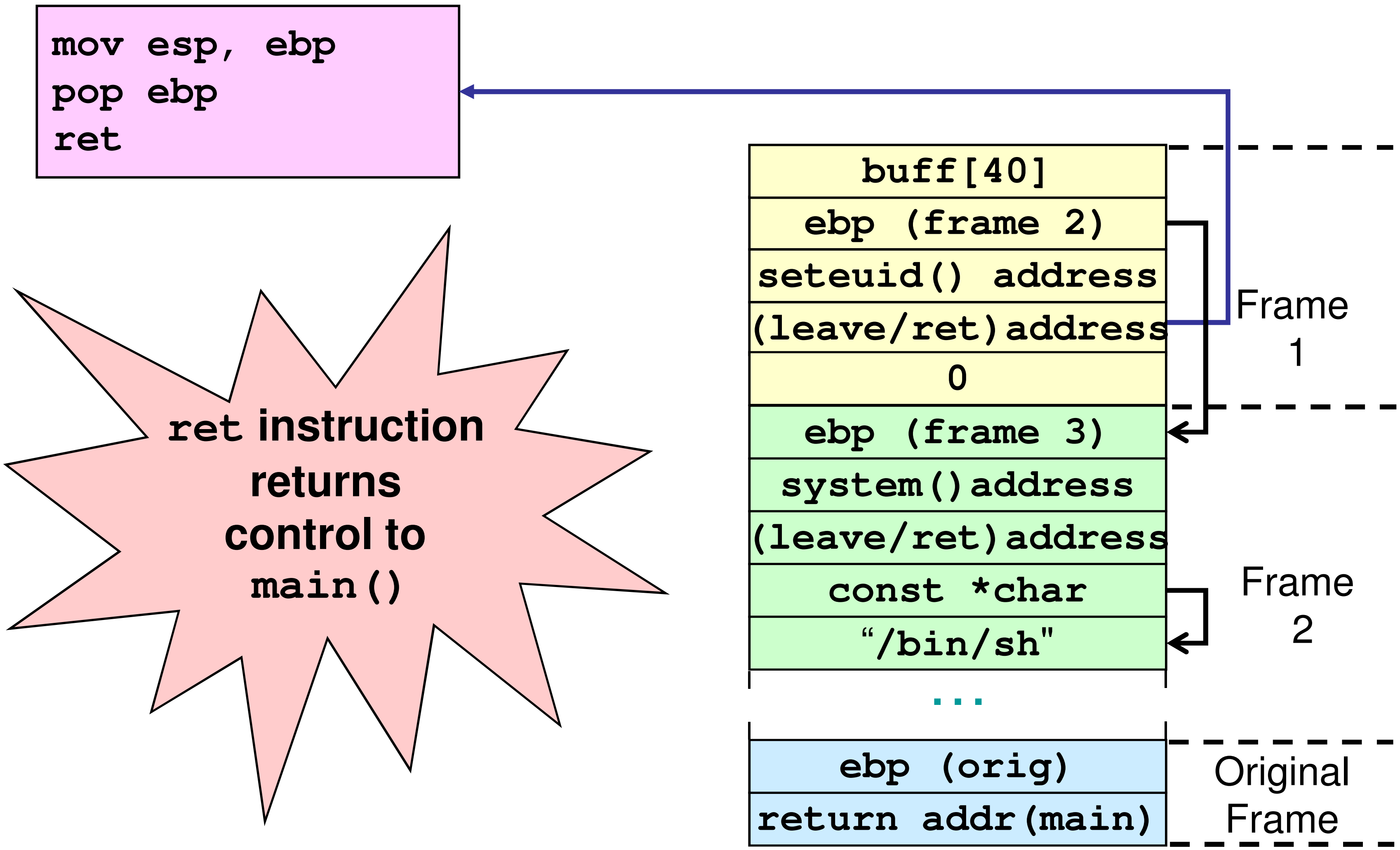
`system()` Returns



`system()` Returns



`system()` 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 errors using NTBS

String Vulnerabilities

Mitigation Strategies

Summary

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.

```
int myfunc(const char *arg) {
  char buff[100];
  if (strlen(arg) >= sizeof(buff)) {
    abort();
  }
}
```

ISO/IEC "Security" TR 24731-1

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

ISO/IEC TR 24731-1 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 24731-1 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 function signature.

`strcpy_s()` Function

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

Has the signature

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

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

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

strcpy_s() Example

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

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

`strcpy_s()` fails and generates a runtime constraint error.

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.

ISO/IEC TR 24731-1 Summary

Available in Microsoft Visual C++ 2005.

Dinkumware is working on an implementation packaged for gcc, EDG, and VC++

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

The ISO/IEC TR 24731-1 functions are

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

`std::basic_string`

The `basic_string` class

- less prone to security vulnerabilities than null-terminated byte strings
- buffers dynamically resize as additional memory is required

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 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.


Questions
about
Strings