COMP435: *SECURITY CONCEPTS!*

Lecture 15: Buffer Overflow Attacks

tinyurl.com/comp435-fa25

CAPABILITIES

Mechanisms for Making Authorization Decisions

	BIBLIOG	TEMP	F	HELP.TXT	C_COMP	LINKER	SYS_CLOCK	PRINTER
USER A	ORW	ORW	ORW	R	X	X	R	W
USER B	R	-	-	R	X	X	R	W
USER S	RW	-	R	R	X	X	R	W
USER T	-	-	-	R	X	X	R	W
SYS_MGR	-	-	-	RW	OX	OX	ORW	O
USER_SVCS	-	-	-	O	X	X	R	W

Access Control

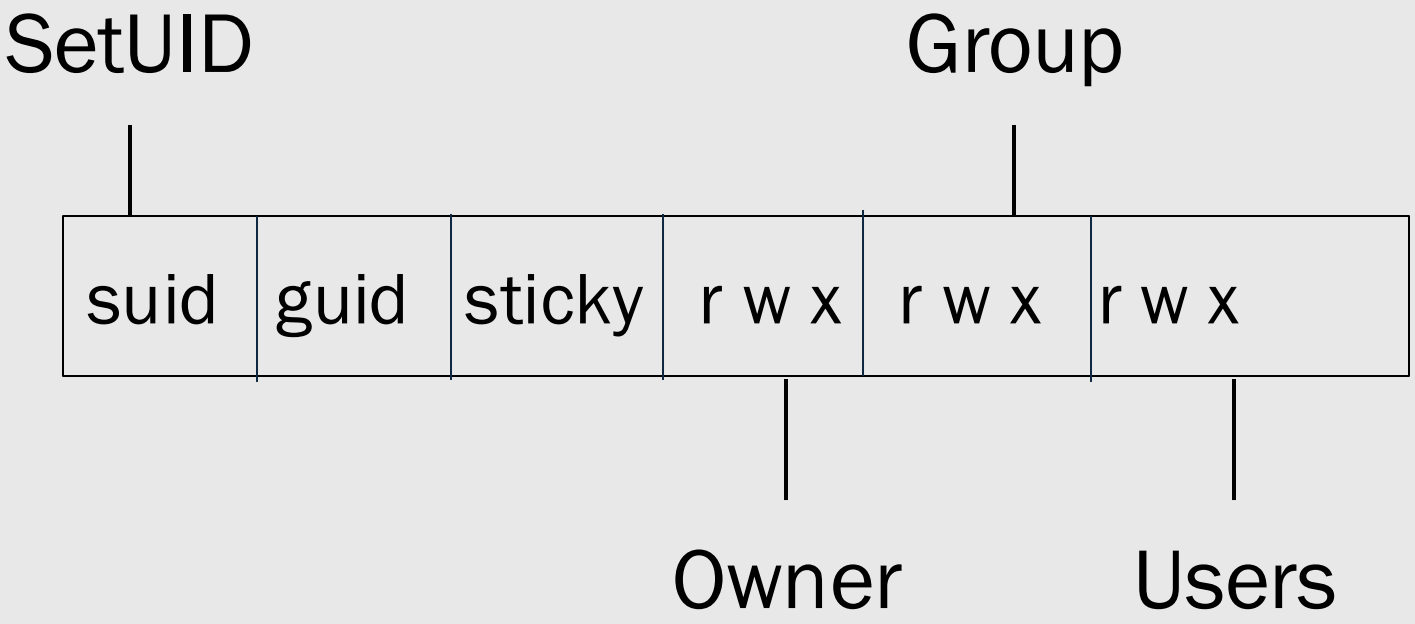
Capabilities

Is this process authorized to access this resource?

MOTIVATING CAPABILITIES: *THE CONFUSED DEPUTY ATTACK*

SetUID

When the setUID bit is set on an executable file and a user executes the file → the effective UID of that user switches to that of the creator of the executable.



myFile.exe

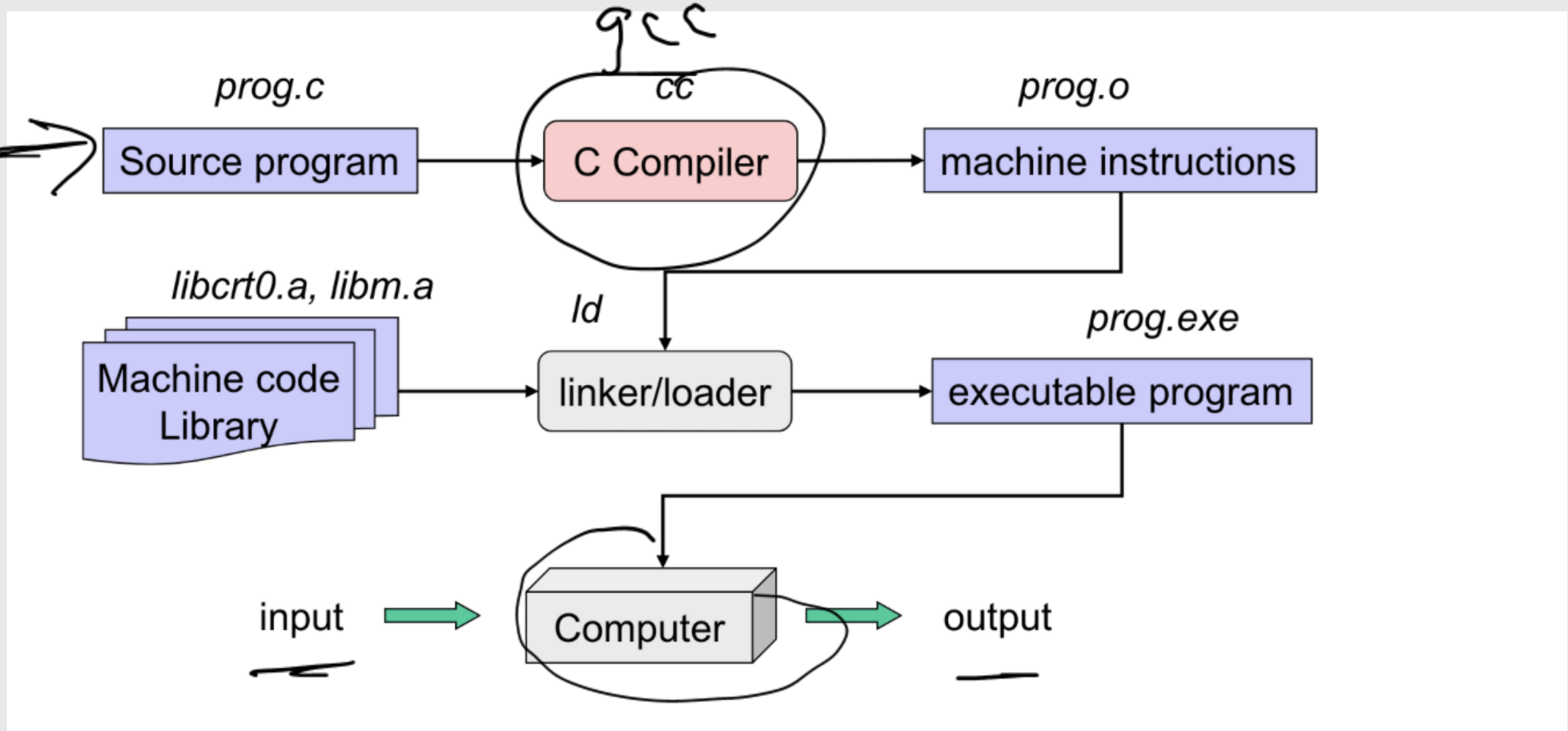
Quick Background: EUID

- Effective user ID (EUID): the user ID that the OS uses to determine **what permissions the program** has when it runs

- There is also Real user ID (RUID): ID of user who started the process.

- Most of the time EUID == RUID.
- When the `setuid` bit is set and a user runs that program, EUID becomes the file owner's UID (usually root), even though the RUID remains the users
    - This lets certain programs perform privileged ops on behalf of ordinary users

Background: The Compiler



```
(1) int main(int argc, char *argv[]) {
    /*compile code*/

    // write out binary executable
    (2) FILE *fp = fopen(argv[2], "w");
    /*write to fp*/

    // write out statistics
    (3) fp = fopen("/etc/compiler_stats", "a");
    /*write to fp*/
}
```

```
$ gcc prog.c -o prog
```

```
(1) int main(int argc, char *argv[]) {
    /*compile code*/

    // write out binary executable
    (2) FILE *fp = fopen(argv[2], "w");
    /*write to fp*/

    // write out statistics
    (3) fp = fopen("/etc/compiler_stats", "a");
    /*write to fp*/
}
```

```
$ gcc prog.c -o prog
$ gcc prog.c -o "/etc/passwd"
```

```
(1) int main(int argc, char *argv[]) {
    /*compile code*/

    // w
    (2)
    /*w

    // w
    (3) f
    /*write to fp*/
}
```

```
$ gcc prog.c -o prog
$ gcc prog.c -o "/etc/passwd"
```

This is a problem if the compiler is running with root privileges or as an administrator!

It will use its *own privileges* to open `/etc/passwd` for writing and overwrite the password file! Even if the user who ran the `gcc` command didn't have permission to do so!

Ambient Authority

Def’n: all the extant authority of the current execution context

Problem: Authority can accrue as a program runs.

A given execution context will have multiple sets of permissions at any given time!

Confused Deputy Attack

A program running with multiple sets of permissions uses its ambient authority indiscriminately

In our example:

Role	Description
Deputy	The compiler, has authority to write to system files
Attacker	Unprivileged user, who provides etc/passwd as an output file name
Victim	The system or admin who trusts the compiler
Confusion??	The compiler doesn't realize it's acting on behalf of a user when it opens /etc/passwd for appending. It just uses its own privilege

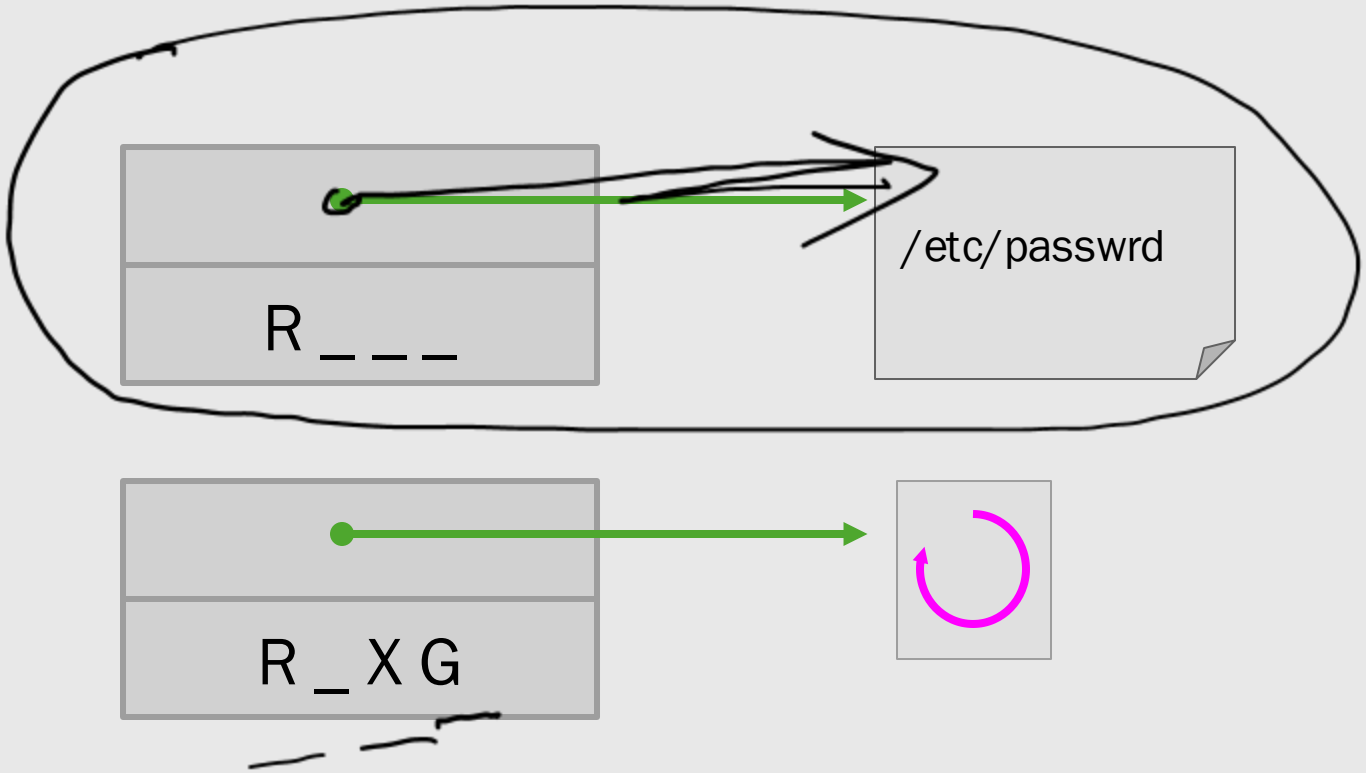
Confused Deputy Attack

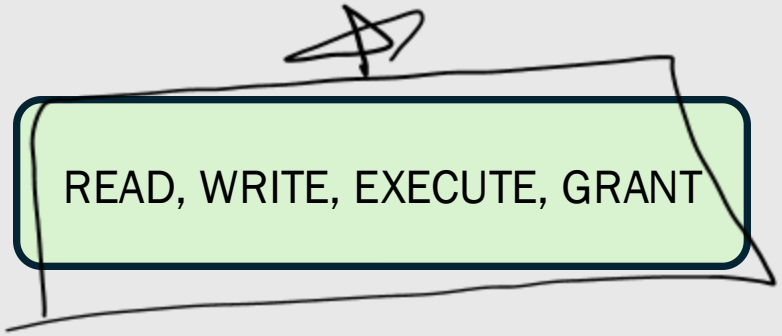
A program running with multiple sets of permissions uses its ambient authority indiscriminately

Access Control cannot prevent the confused deputy attack

Capabilities

- object reference + access right
- “fat pointer”





```
(1) int main(int argc, char *argv[]) {
    /*compile code*/

    //write out binary executable
(2) cap *fcap = get_cap(argv[2], user_dir_cap);
(3) FILE *fp = cap_fopen(fcap);
    /*write to fp*/

    //write out statistics
(4) fcap = get_cap("/etc/compiler_stats", sys_dir_cap);
(5) fp = cap_fopen(fcap);
    /*write to fp*/
 }
```

```
(1) int main(int argc, char *argv[]) {
    /*compile code*/

 //write out binary executable
(2) cap *fcap = get_cap(argv[2], use
(3) FILE *fp = cap_fopen(fcap);
    /*write to fp*/

//write out statistics
(4) fcap = get_cap("/etc/compiler_stats", sys_dir_cap);
(5) fp = cap_fopen(fcap);
    /*write to fp*/
 }
```

```
$ gcc prog.c -o prog
$ gcc prog.c -o "/etc/passwd"
> ERROR: no capability for
passwd file!
```

Capabilities

Properties:

- unforgeable token
- token directly ties access right to object
- no designation without authority

Provides:

- no ambient authority
- supports principle of least privilege

Another Confused Deputy Example: WS

Chromium: Open-source web browser

nodeIntegration: false

Node.js: a JavaScript runtime environment that enables developers to execute JavaScript code outside of a web browser

Electron: a framework for building apps w/ HTML, CSS, JS, packaged together with Chromium & Node.JS. Slack, VSCode & Discord are examples of apps built using Electron!

saveLog("/etc/passwd")

Another Confused Deputy Example: WS

The problem: Electron apps combine Chromium (renderer) with Node.js. If untrusted web content executes in the renderer and the preload fn exposes Node APIs, that content can call powerful OS APIs (read/write files, exec commands).

Why it's confused-deputy: The renderer is effectively given ambient authority (Node/OS access) and untrusted JS tricks it into doing privileged work

Mitigations: disable Node integration for untrusted content, expose minimal, validated APIs via a controlled preload, and sandbox untrusted content (same capability principles)

SOFTWARE SECURITY: BUFFER VULNERABILITIES

Terminology

- Error
- Fault
- Failure
- Flaw

Error

Def’n: a human mistake

E.g., A designer fails to consider a possible input value

Fault

Def’n: an encoding of the error

E.g., The program has no code for handling unexpected inputs

Failure

Def’n: a deviation from desired behavior

E.g., The program crashes when it receives certain inputs

Flaw

Def’n: a fault or failure

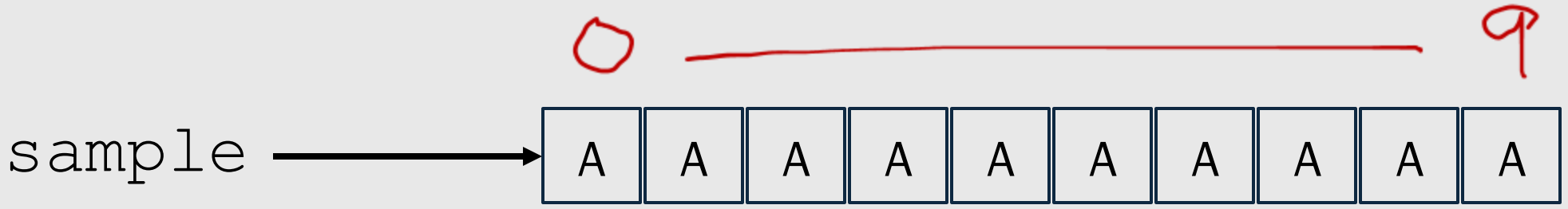
Error, fault, failure come from software engineering

A fault anywhere in the system may be susceptible to exploit by a malicious actor, even though under normal operating conditions the fault would never exhibit as a failure

BUFFERS

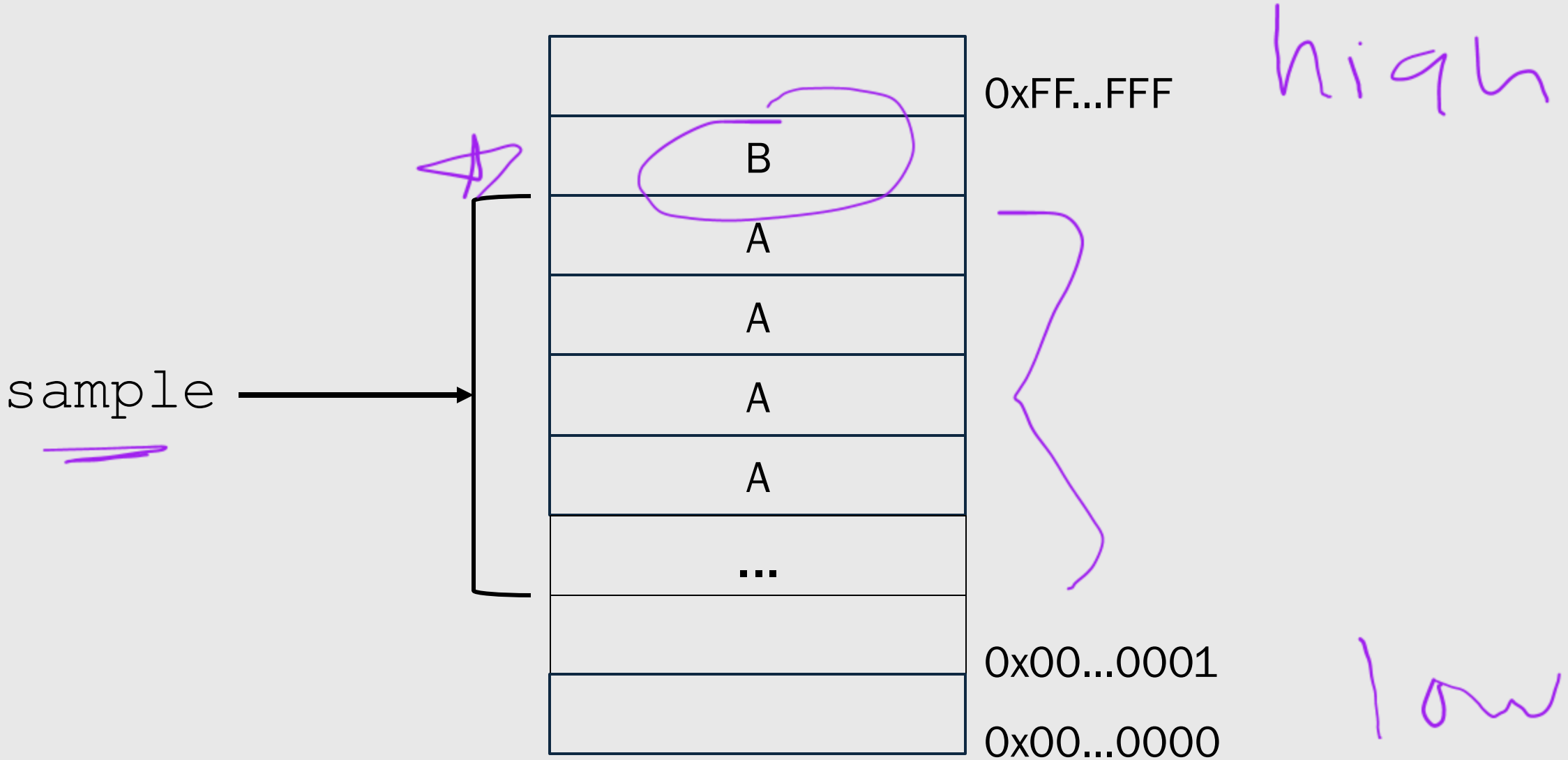
```
(1) char sample[10];
(2) int i;

(1) for (i=0; i<10; i++)
(2)     sample[i] = 'A';
(3) sample[10] = 'B';
```

```
(1)char sample[10];
(2)int i;

(1)for (i=0; i<10; i++)
(2)    sample[i] = 'A';
(3)sample[10] = 'B';
```



```
(1)char sample[10];
(2)int i;

(1)for (i=0; i<10; i++)
(2)    sample[i] = 'A';
(3)sample[10] = 'B';
```

?


sample
B
A
A
A
A
...
0xFF...FFF
0x00...0001
0x00...0000
high
low

Terminology

- Buffer overflow
- Buffer overrun
- Buffer overflow attack
- Stack smashing

Terminology

- Buffer overflow
- Buffer overrun
- Buffer overflow attack
- Stack smashing

writing past the end of an allocated array

Terminology

- Buffer overflow
- Buffer overrun
- Buffer overflow attack
- Stack smashing

reading past the end of an allocated array

Terminology

- Buffer overflow
- Buffer overrun
- Buffer overflow attack
- Stack smashing

exploiting a buffer overflow flaw to gain control of the system

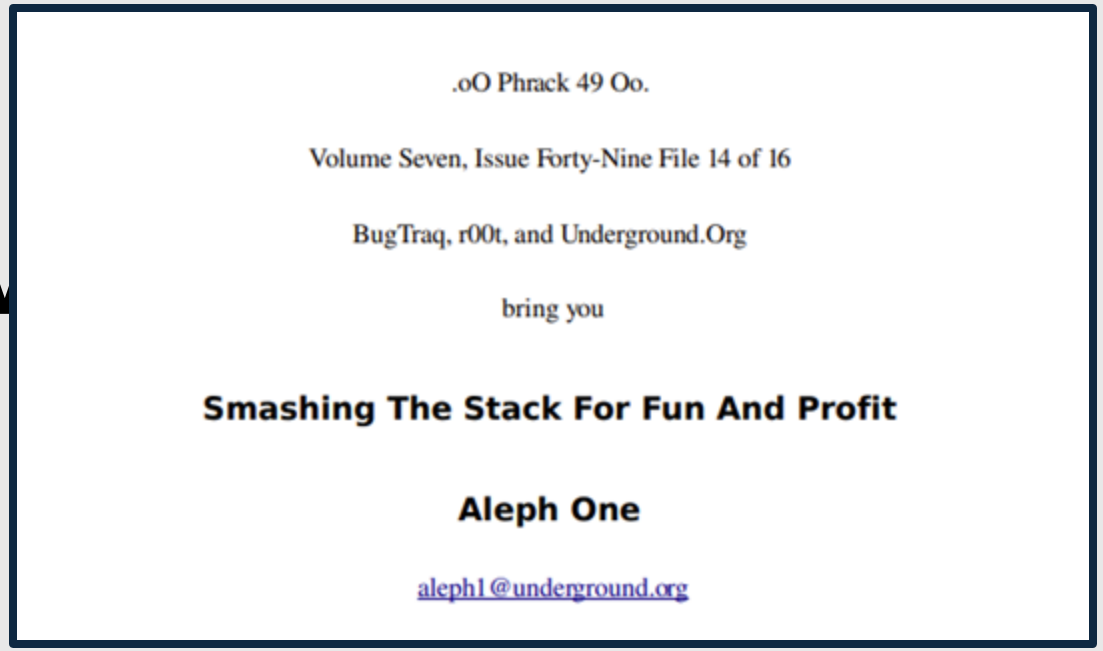
Terminology

- Buffer overflow
- Buffer overrun
- Buffer overflow attack
- Stack smashing

A particular type of buffer overflow attack

BUFFER OVERFLOW ATTACKS

Buffer Ov



.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine File 14 of 16

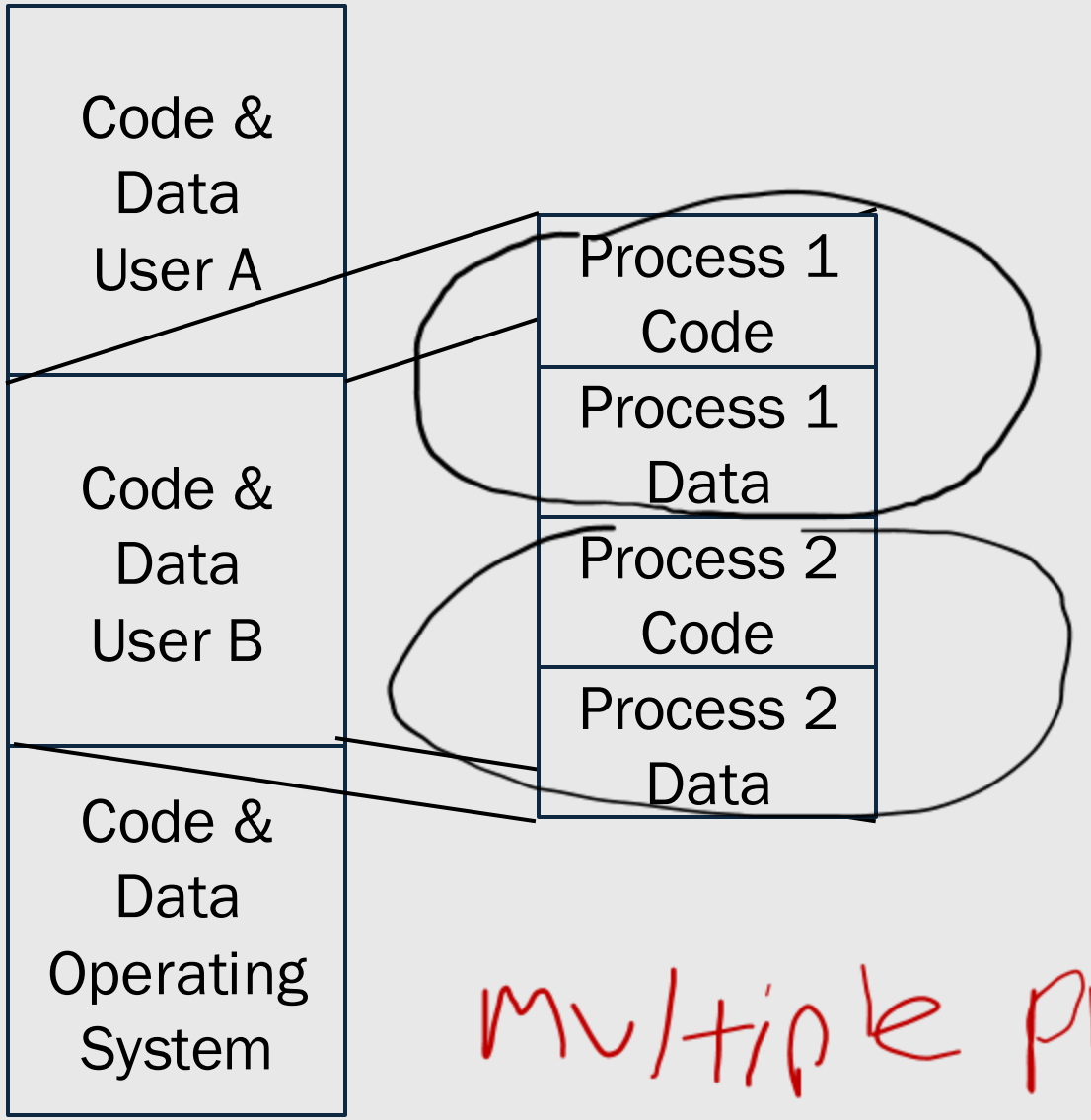
BugTraq, r00t, and Underground.Org

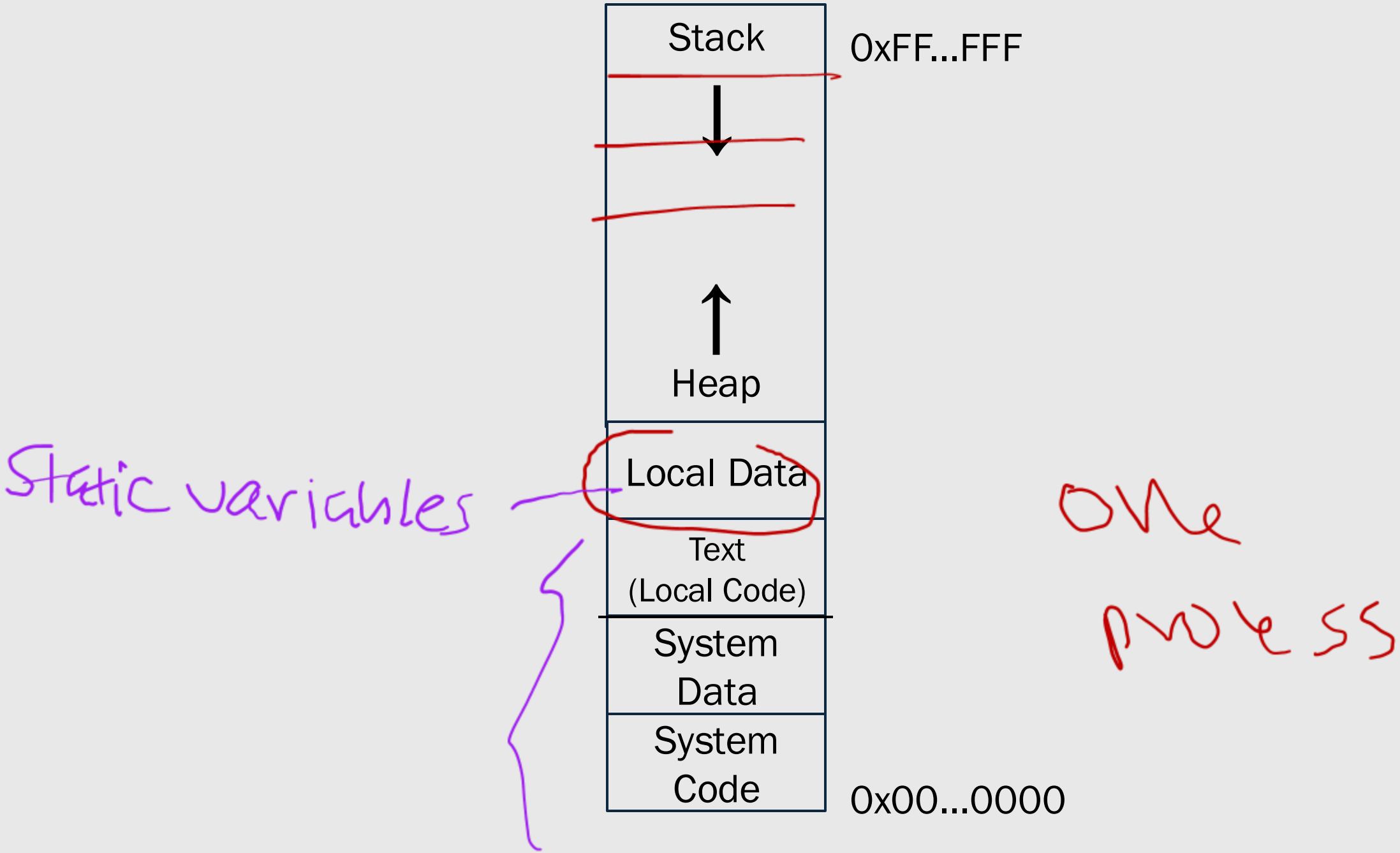
bring you

Smashing The Stack For Fun And Profit

Aleph One

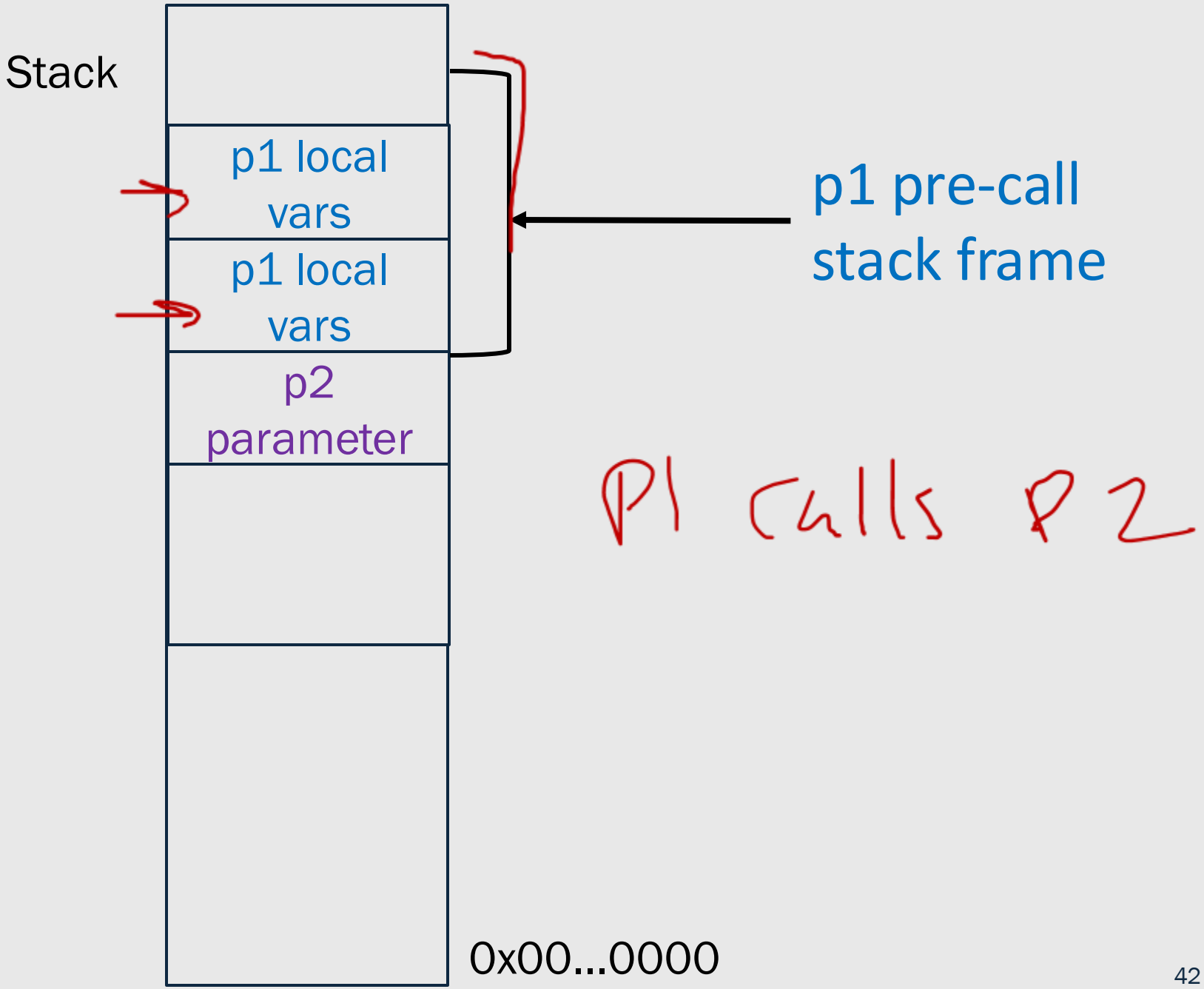
aleph1@underground.org

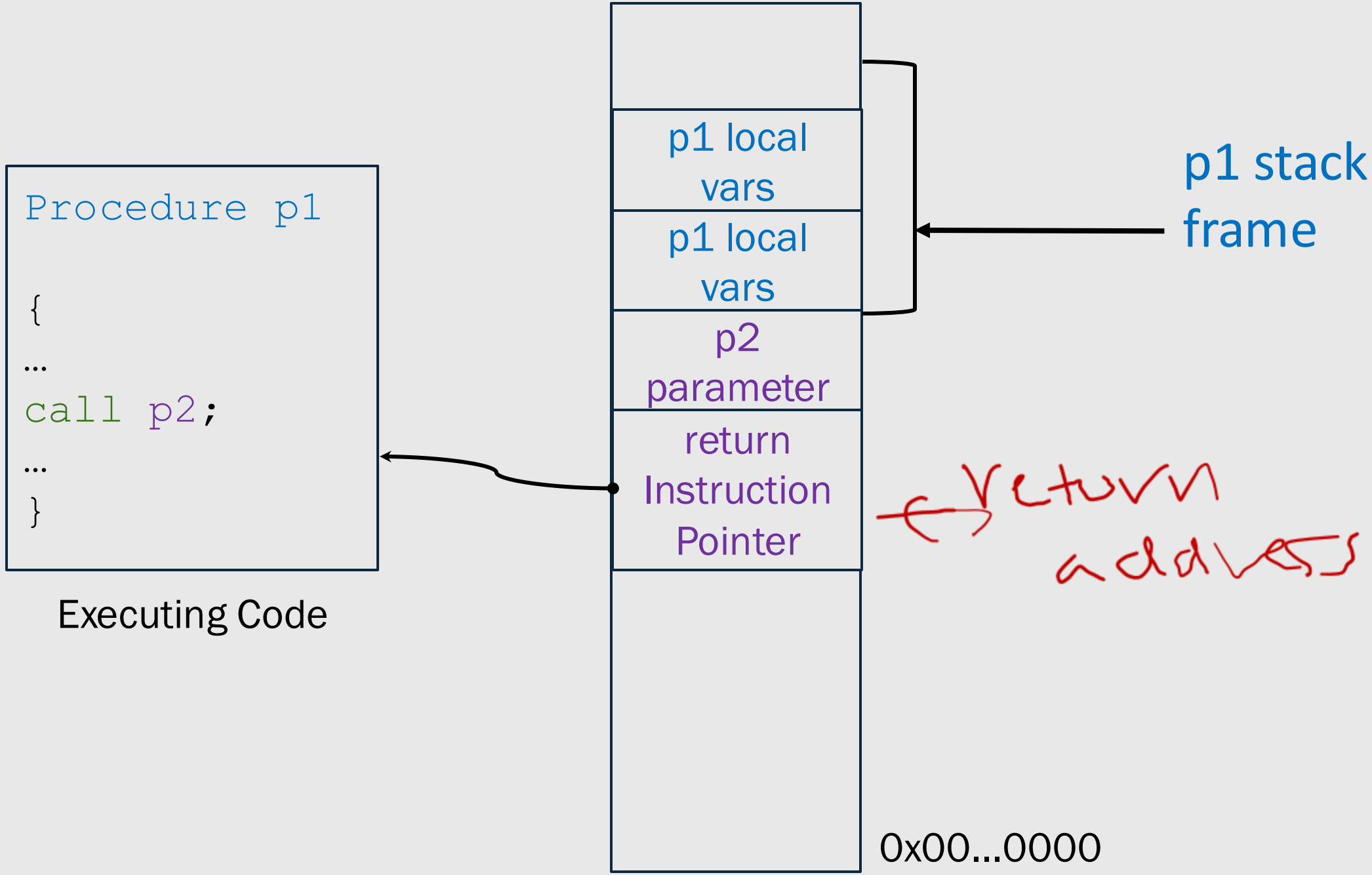

Code & Data User A
Code & Data User B
Code & Data Operating System
Process 1 Code
Process 1 Data
Process 2 Code
Process 2 Data

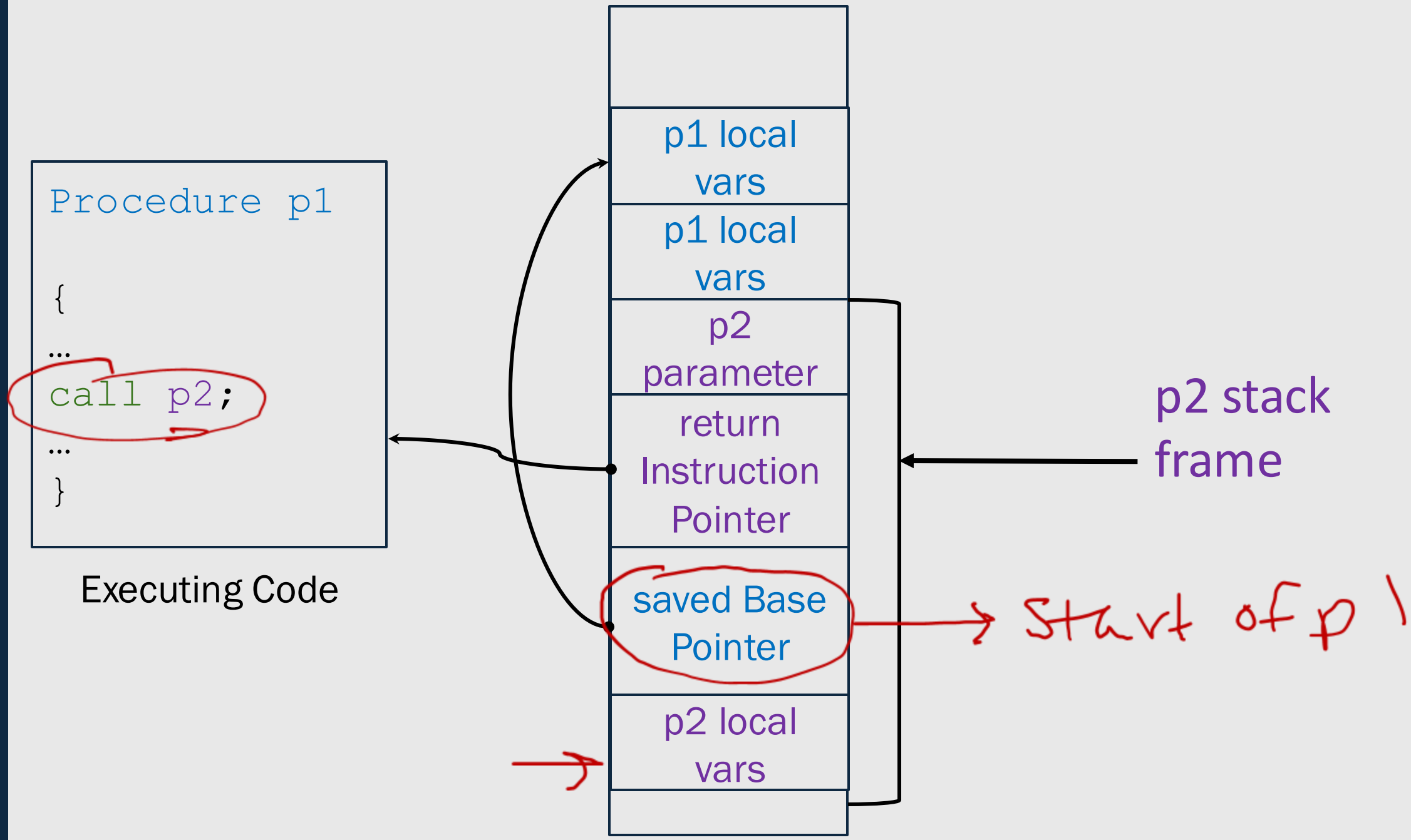

Stack
0xFF...FFF
Heap
Local Data
Text
(Local Code)
System
Data
System
Code
0x00...0000
Static variables
One
process

```
Procedure p1

{
…
call p2;
…
}
```

Executing Code



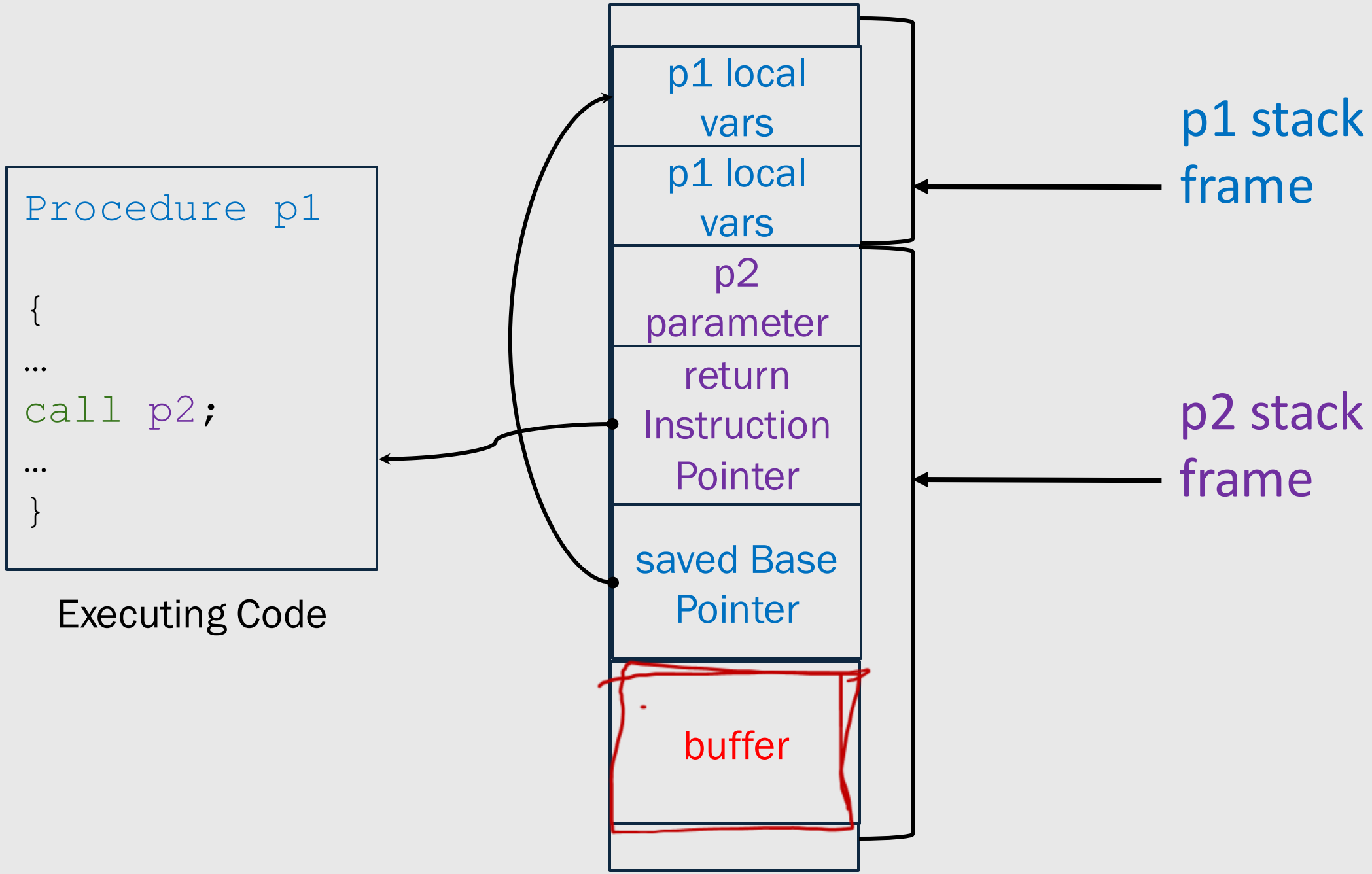

Procedure p1
{
…
call p2;
…
}
Executing Code
p1 local vars
p1 local vars
p2 parameter
return Instruction Pointer
p1 stack frame
return address
0x00…0000


Procedure p1
{
…
call p2;
…
}
Executing Code
p1 local vars
p1 local vars
p2 parameter
return Instruction Pointer
saved Base Pointer
p2 local vars
p2 stack frame
start of p1

Procedure p2

```
(1) int p2(char *input)
    {
(2)     char buffer[100];
(3)     strcpy(buffer, input);
(4)     return 0;
    }
```

dst , src


Procedure p1
{
…
call p2;
…
}
Executing Code
p1 local vars
p1 local vars
p2 parameter
return Instruction Pointer
saved Base Pointer
buffer
p1 stack frame
p2 stack frame

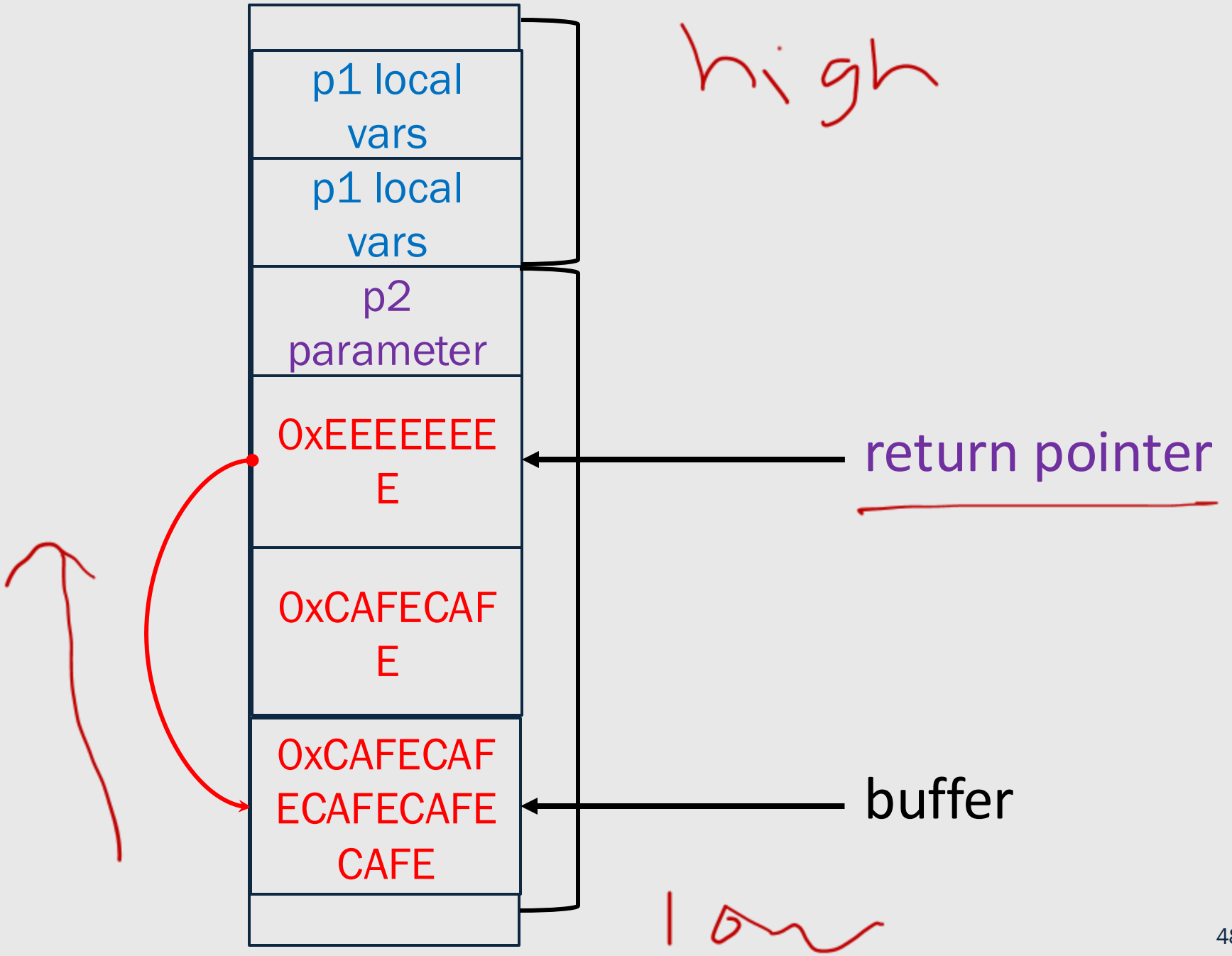
Procedure p2

```
(1) int p2(char *input)
    {
(2)     char buffer[100];
(3)     strcpy(buffer, input);
(4)     return 0;
    }
```

input → 0xCAFECAFECAFECAFE...EEEEEE

```
Procedure p1

{
…
call p2;
…
}
```

Executing Code



0xCAFECAFECAFECAFE

shellcode.c

```
#include stdio.h
void main() {
char *name[2];
name[0] = "/bin/sh";
name[1] = NULL;
execve(name[0], name, NULL);
}
```

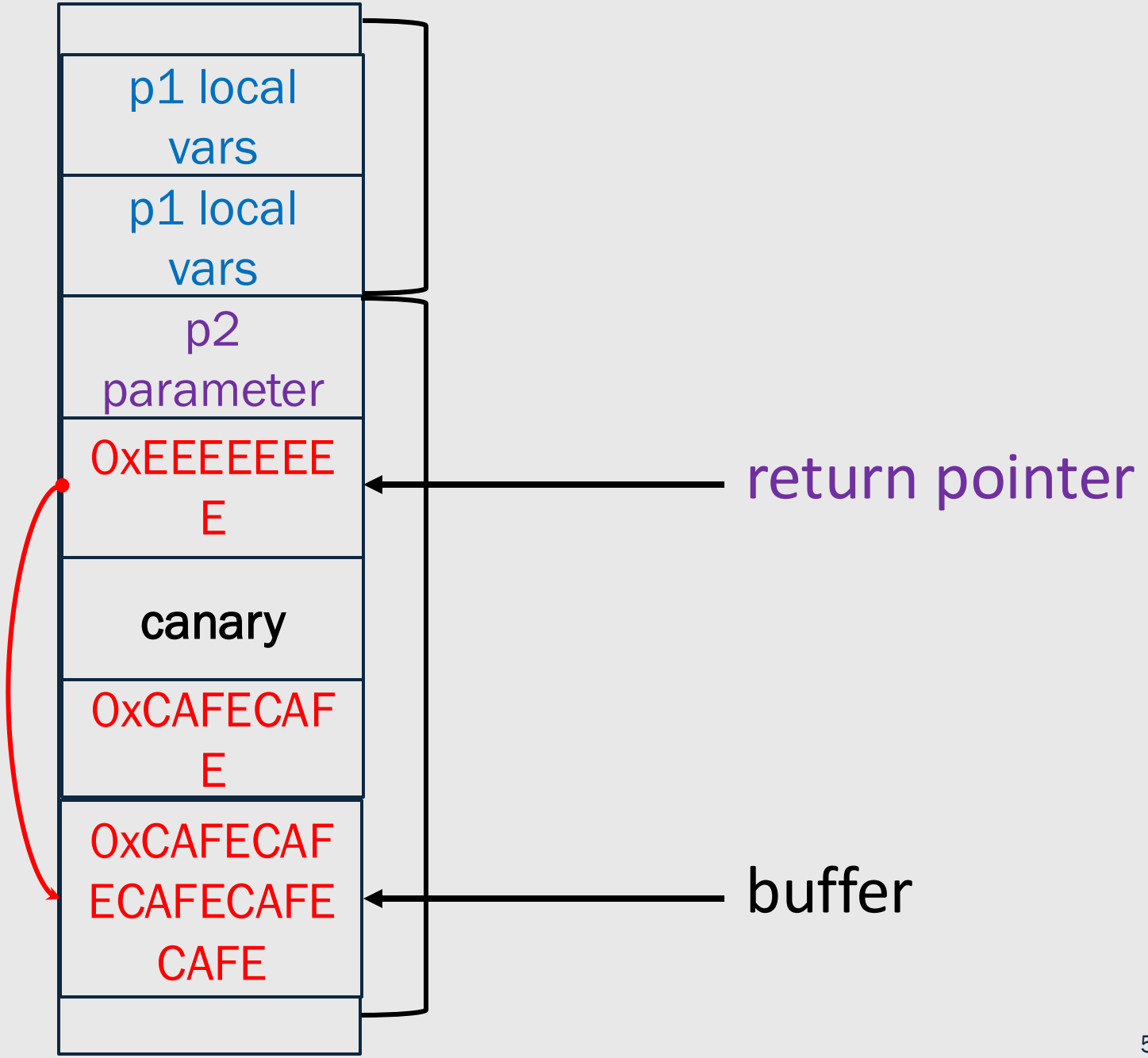
Countermeasure: Canary

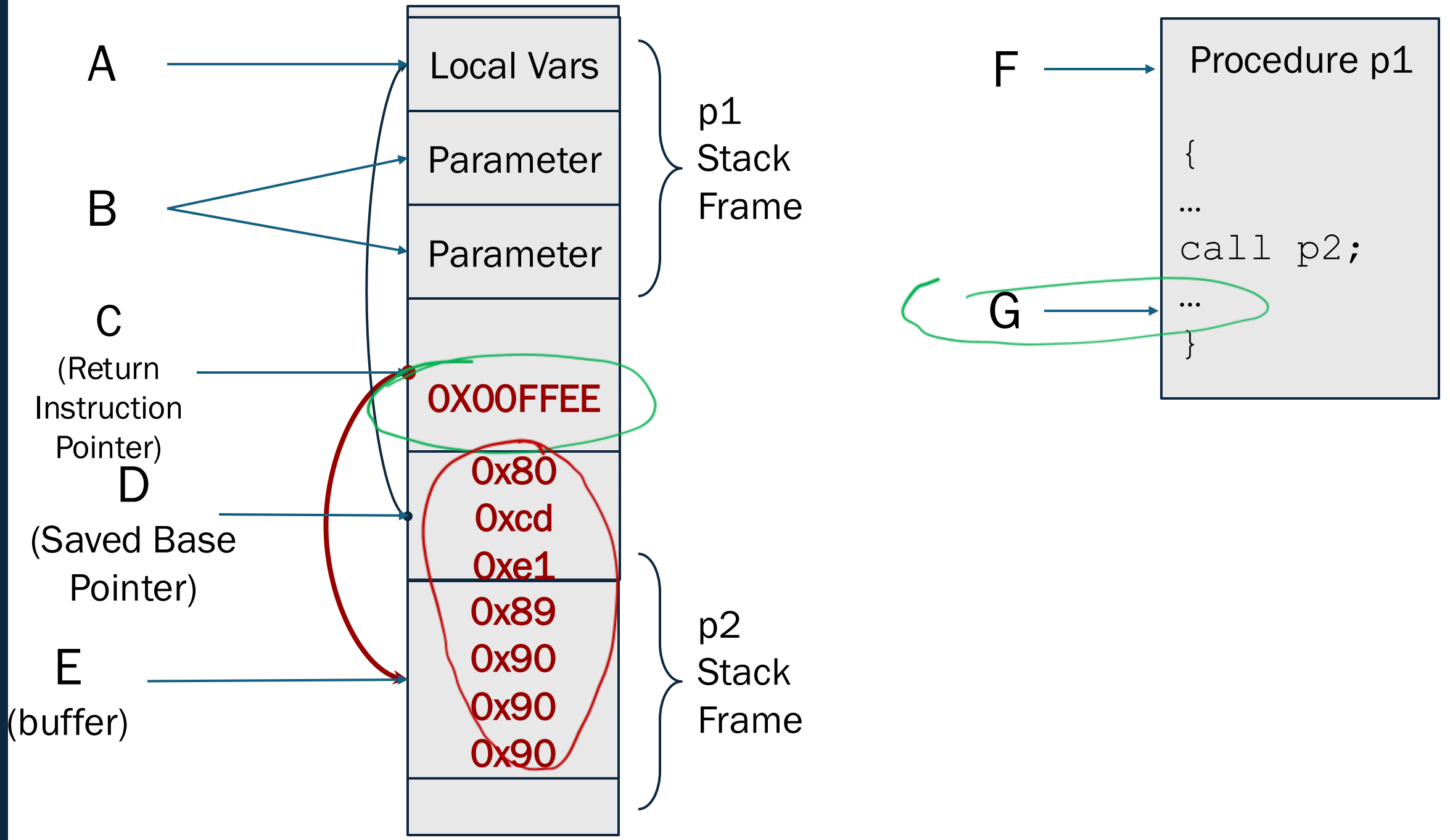
Def’n: a secret value added to the stack

Before returning from a procedure, check that the canary has the expected value

```
Procedure p1

{
…
call p2;
…
}
```

Executing Code




A
B
C
(Return Instruction Pointer)
D
(Saved Base Pointer)
E
(buffer)
Local Vars
Parameter
Parameter
0X00FFEE
0x80
0xcd
0xe1
0x89
0x90
0x90
0x90
p1 Stack Frame
p2 Stack Frame
F
G
Procedure p1
{
…
call p2;
…
}

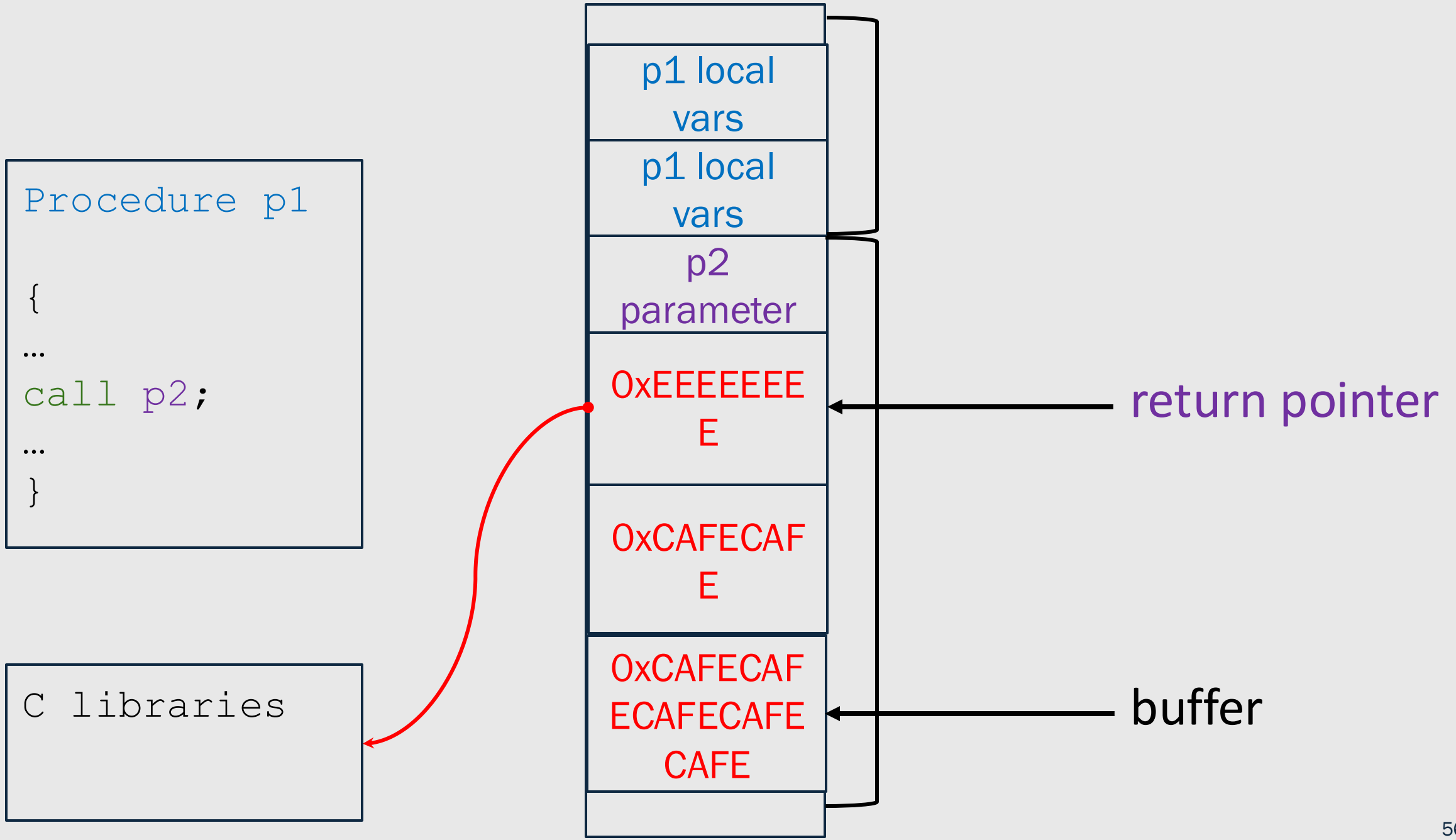
Countermeasure: W⊕X

Def’n: allow regions of memory to be writable or executable, but not both.

E.g., The stack must be writable, but does not need to be executable

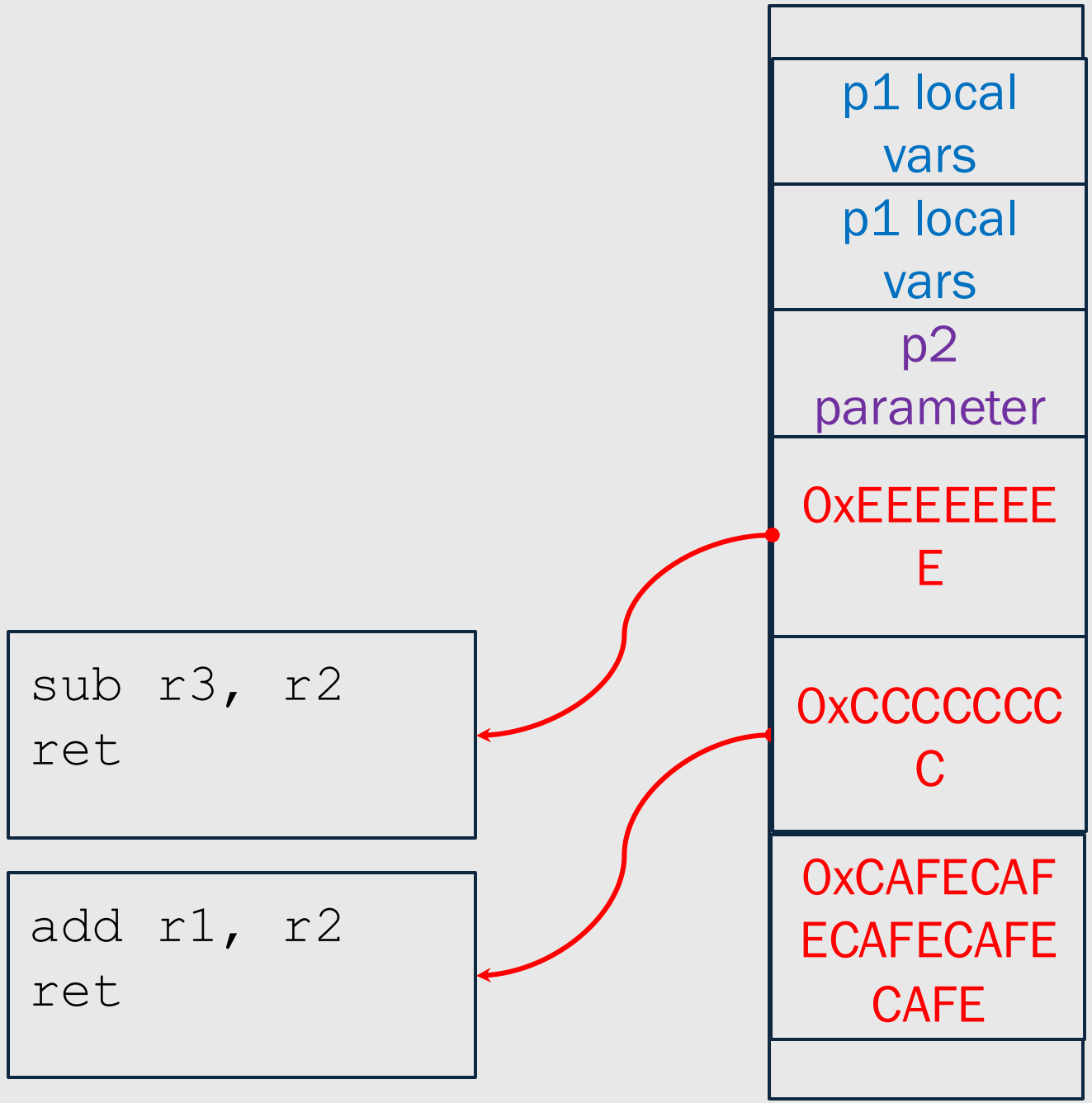
Attack: Return into LibC

Instead of writing attack code in buffer, point to existing code in the system


Procedure p1
{
…
call p2;
…
}
C libraries
p1 local vars
p1 local vars
p2 parameter
0xEEEEEEEE
0xCAFECAFE
0xCAFECAFE ECAFECAFE CAFE
return pointer
buffer

Attack: Return Oriented Programming

Point to gadgets within the existing code


p1 local vars
p1 local vars
p2 parameter
0xEEEEEEEE
0xCCCCCCCC
0xCAFECAFE ECAFECAFE CAFE
sub r3, r2
ret
add r1, r2
ret

Countermeasure: Address Space Layout Randomization

Def’n: change the layout of objects in memory

E.g., The base addresses of the stack, heap, and libraries are chosen at process start-up

We have trained the attacker well!