11/6/26

COMP 435: *Computer Security Concepts*

Lecture 29: Buffer Overflow, More SW Security



The University of North Carolina at Chapel Hill

https://tinyurl.com/comp435-fa26

Plan for Class Today



- 10:10-10:45 – Buffer Overflow Attacks
- 10:45- 11:50 – More Software Security Vulnerabilities and Best Practices

Guiding Question ☺

How can a memory error let an attacker take control of a program, and what can we do to prevent or limit the damage?

Logistics Update



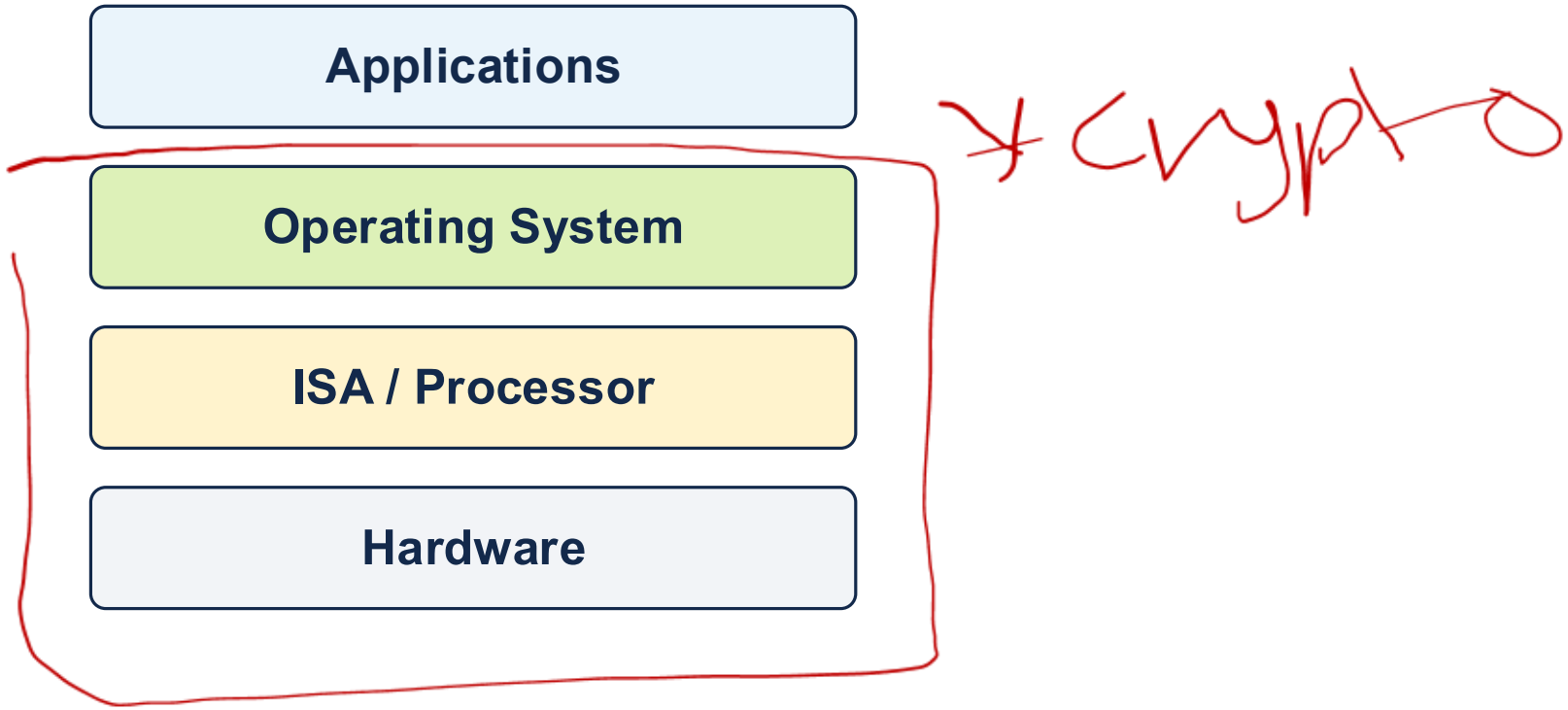
- Written project checkpoint due November 16
  - Worth 15% of the project grade.
- Include a preliminary result, experiment, or (semi) working prototype.
- Summarize what you've completed and your biggest unresolved question or challenge.
- Explain any changes to your project's scope.
- Aim to show concrete progress. you do **not** need a finished project yet!



Software Security: Vulnerabilities

The Stack We Are Trusting

- Every software security mechanism rests on assumptions about the layers below it.





1. Integer Overflow

```
1. int *vulnerable(int *inparray, int len) {
2.    int *myarray;
3.    myarray = malloc(len * sizeof(int));
4.    if (myarray
5.       return (-1
      }
6.    for (i = 0;
7.       myarray[i]
      }
8.    return myarray;
9. }
```

Integer overflow being used as stepping stone for buffer overflow attack!

Lines 6/7 copy up to "len" values, but space actually allocated for `myarray` is potentially very small if we overflow

Problem: memory past the end of the malloc'ed `myarray` will be overwritten

Some reflections



- Catching these errors through manual review is very hard!
  - On first read through, this example seemed like it was doing the right checks!
- Integer underflow can also cause problems
  - Underflow results in a negative value → converted to an unsigned malloc parameter would be interpreted as very large unsigned int.
  - Attempt to allocate huge memory blocks could trigger out-of-memory conditions or return NULL pointers



2. Buffer Vulnerabilities

Terminology



- Error: a human mistake
  - Ex: A designer fails to consider a possible input value
- Fault: an encoding of the error
  - Ex: the program has no code for handling unexpected inputs
- Failure: deviation from desired behavior
  - Ex: program crashes when it receives certain inputs
  - Existence of a failure implies existence of a fault. (not the other way)
- Flaw: Fault or failure.
  - Umbrella term for instances in a system susceptible to exploit by a malicious actor, even though under normal operating conditions a fault would never exhibit as a failure

Come from SWE



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

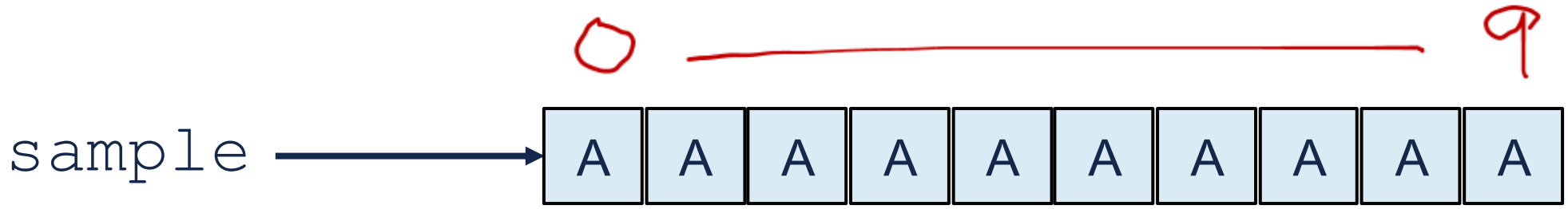
(1)for (i=0; i<10; i++)
(2)    sample[i] = 'A';
(3)sample[10] = 'B';
```



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

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





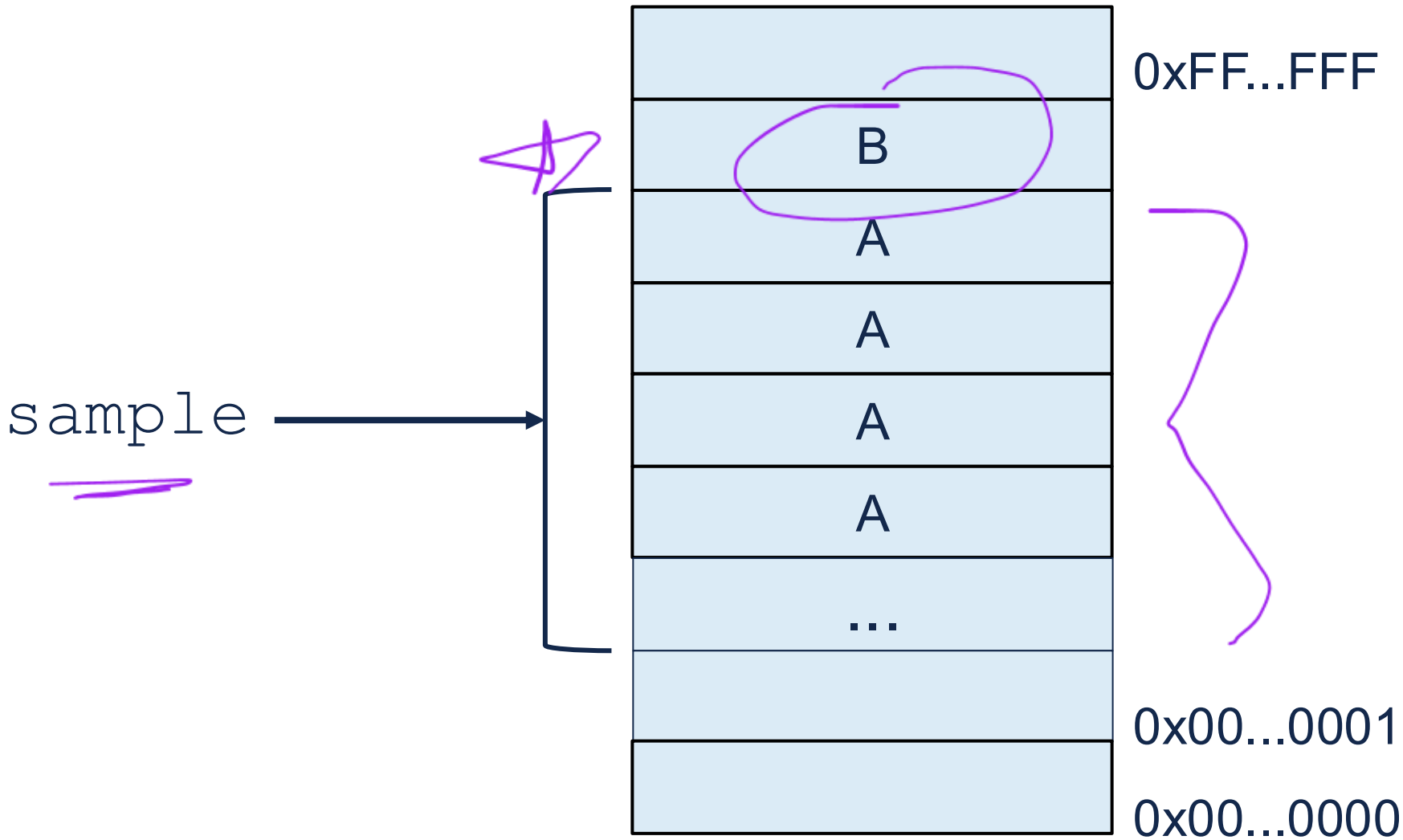
```
(1)char sample[10];
(2)int i;

(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

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***



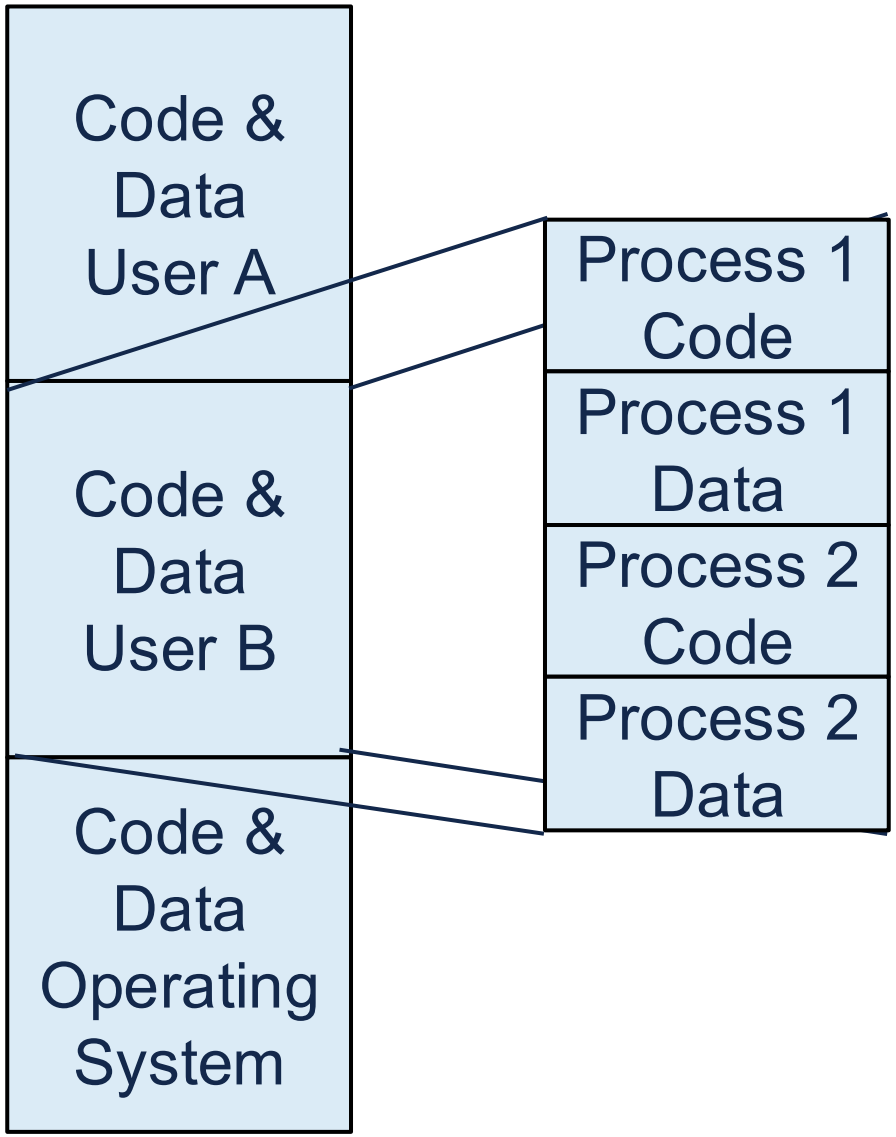
A particular type of buffer overflow attack



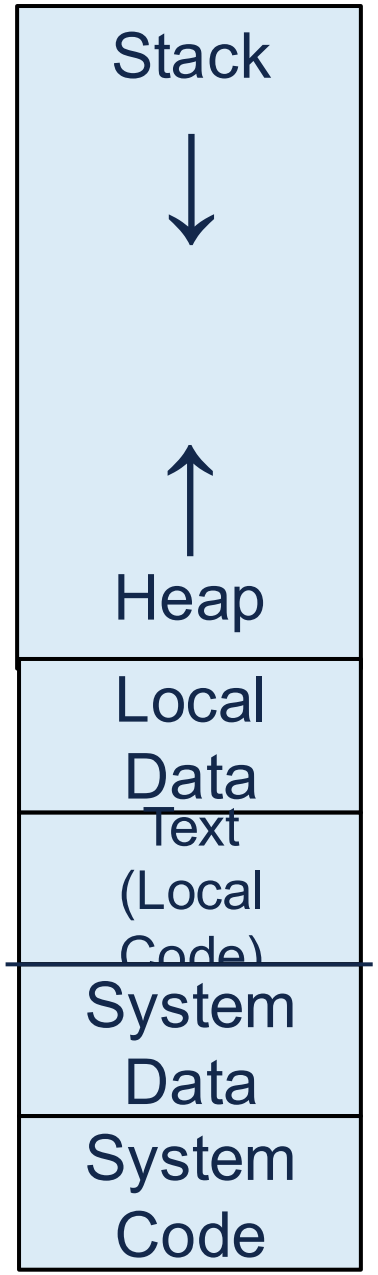


Buffer Overflow Attacks

Code & Data


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

Process Memory Layout



0xFF...FFF

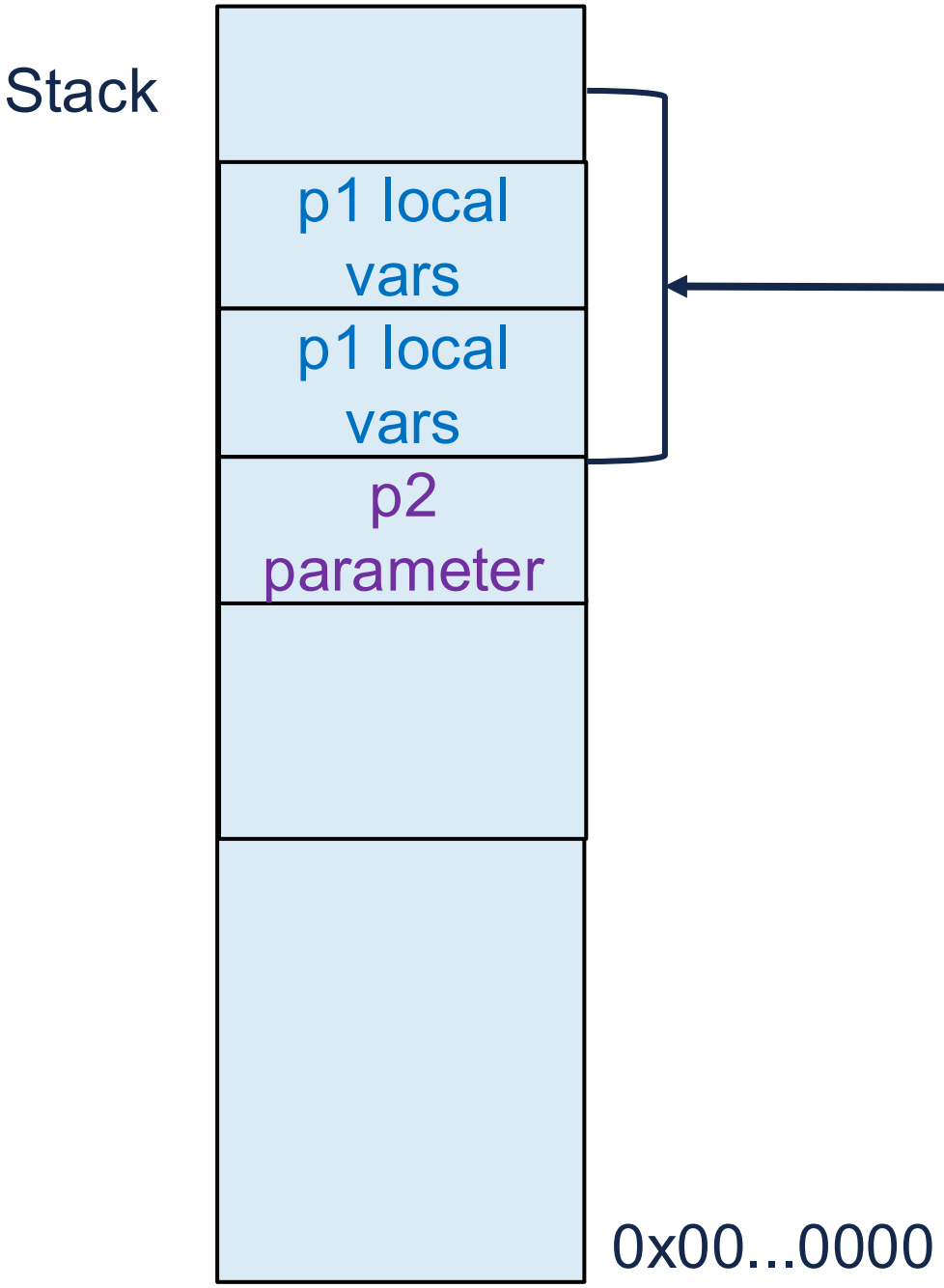
0x00...0000

Calling p2

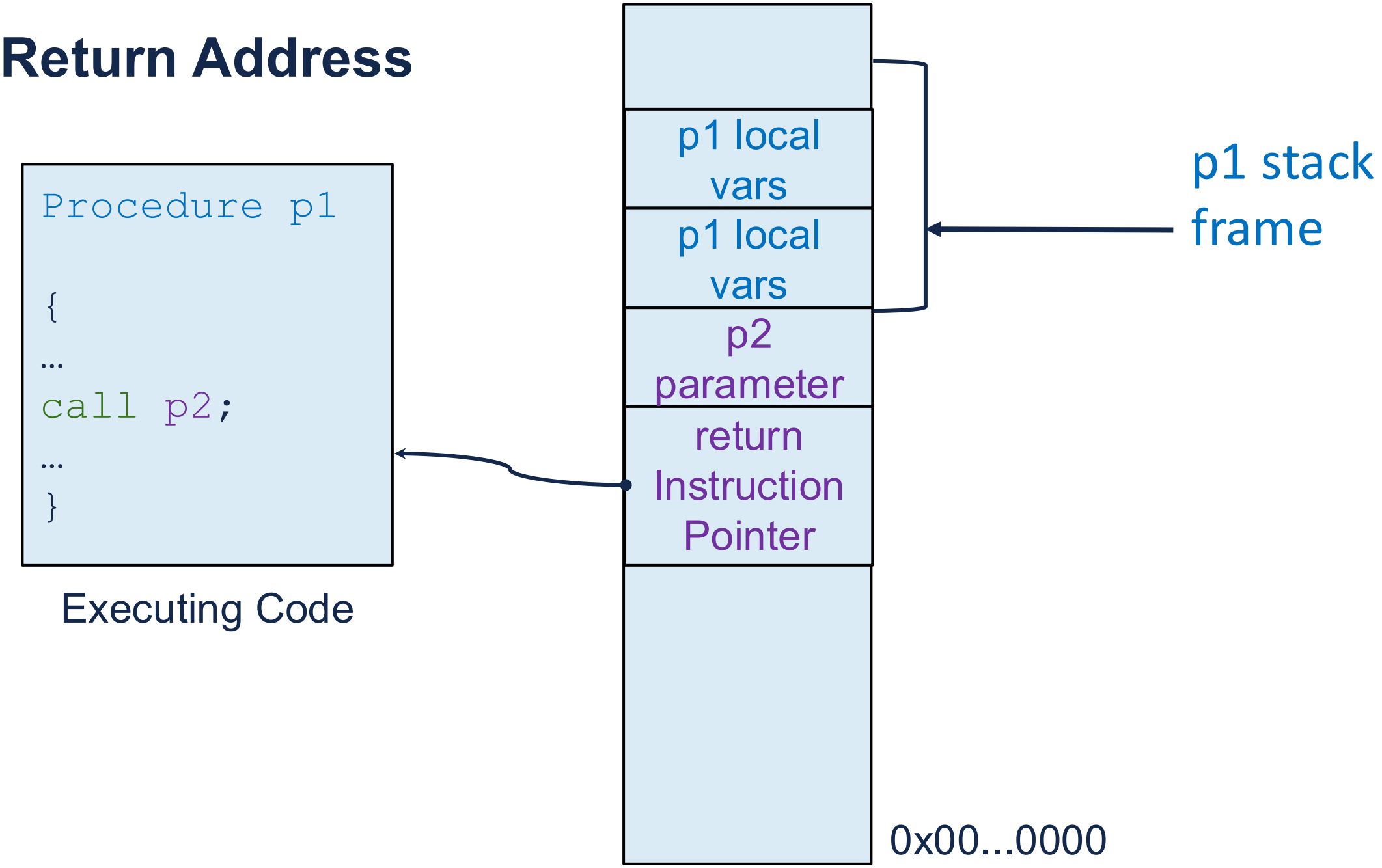
```
Procedure p1

{
…
call p2;
…
}
```

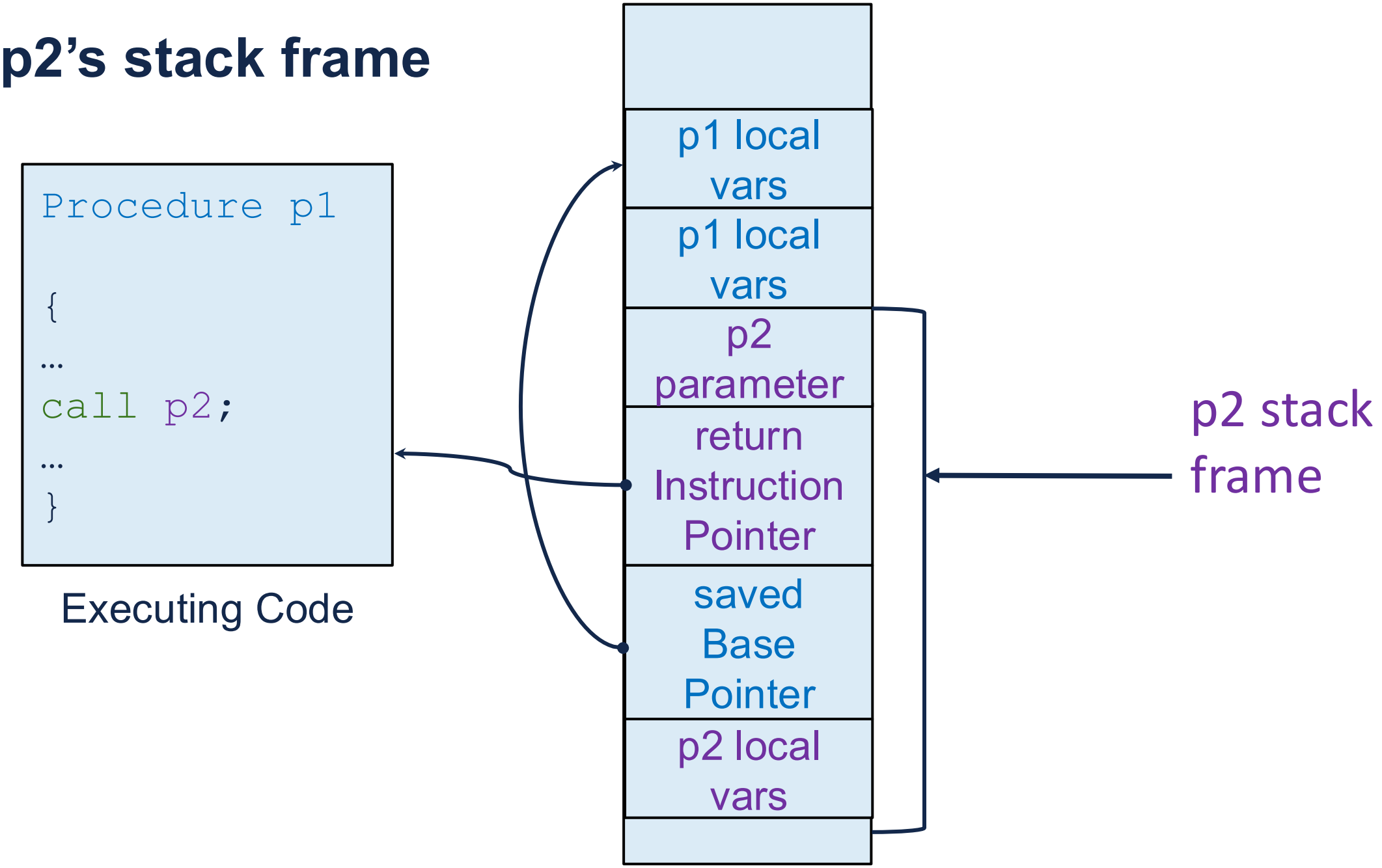
Executing Code



Return Address



p2's stack frame

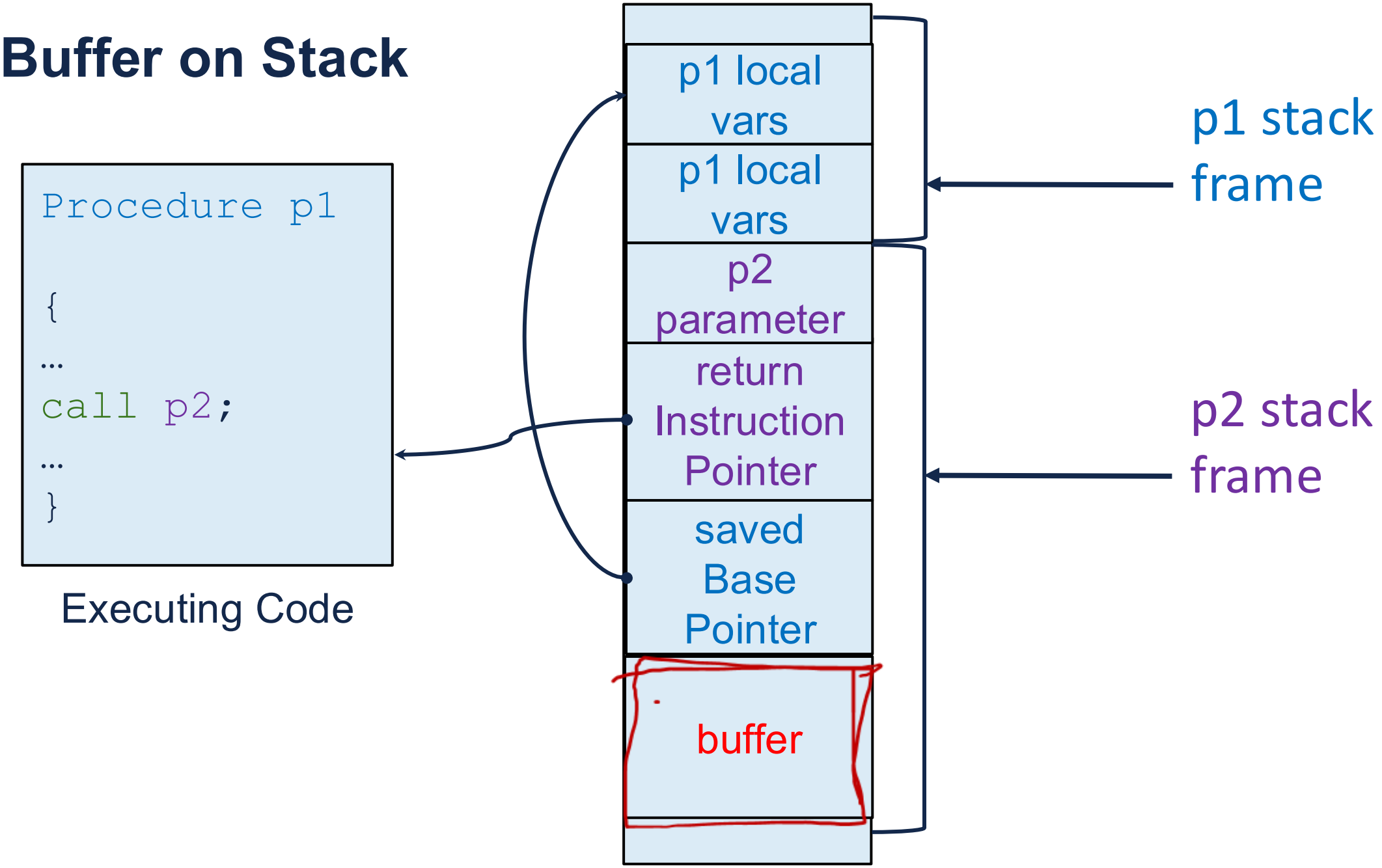


Procedure p2: vulnerable on strcpy



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

Buffer on Stack



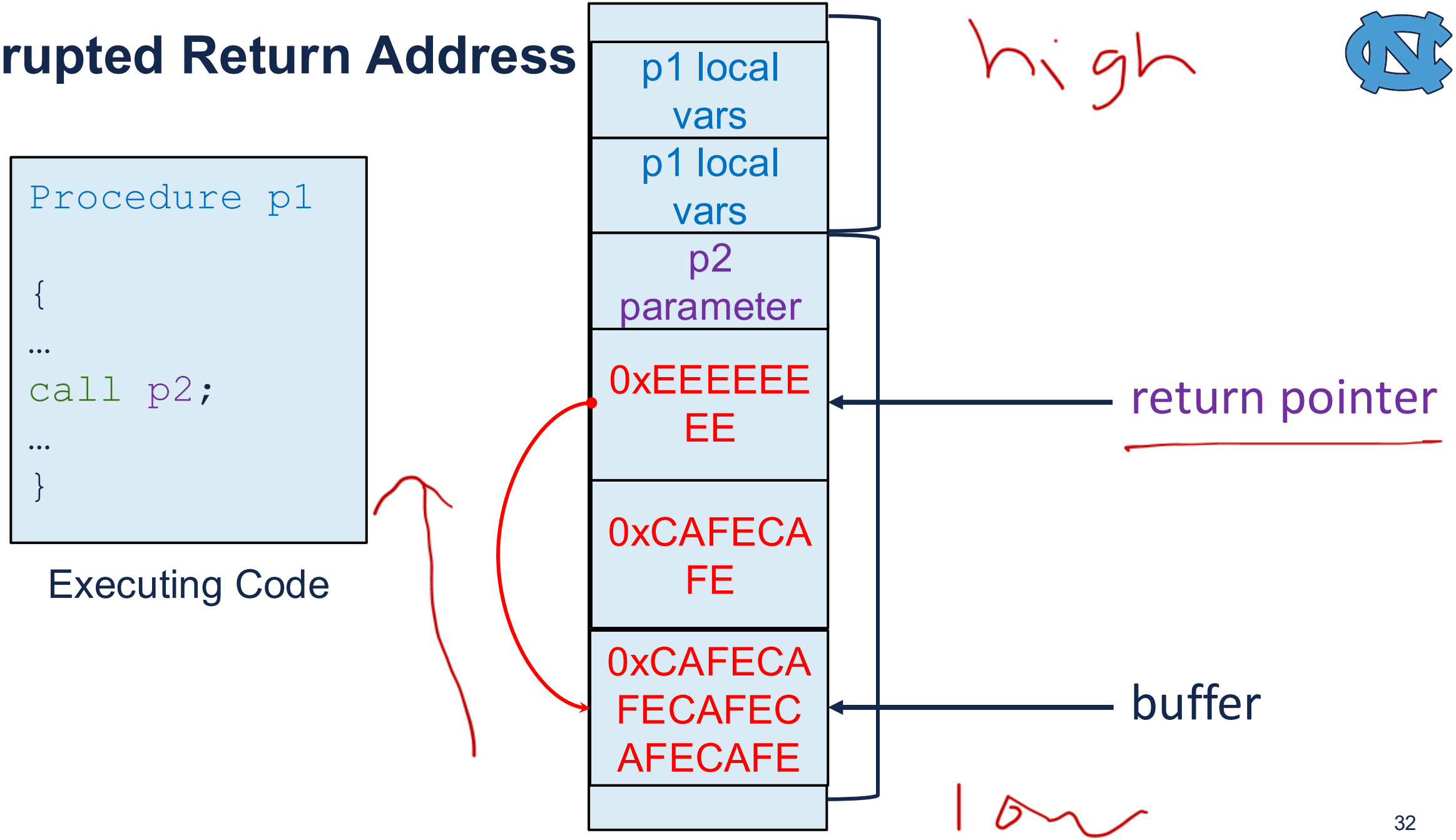
Procedure p2: oversized input



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

input → 0xCAFECAFECAFECAFE...EEEEEE

Corrupted Return Address



shellcode.c



0xCAFECAFECAFECAFE



```
#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

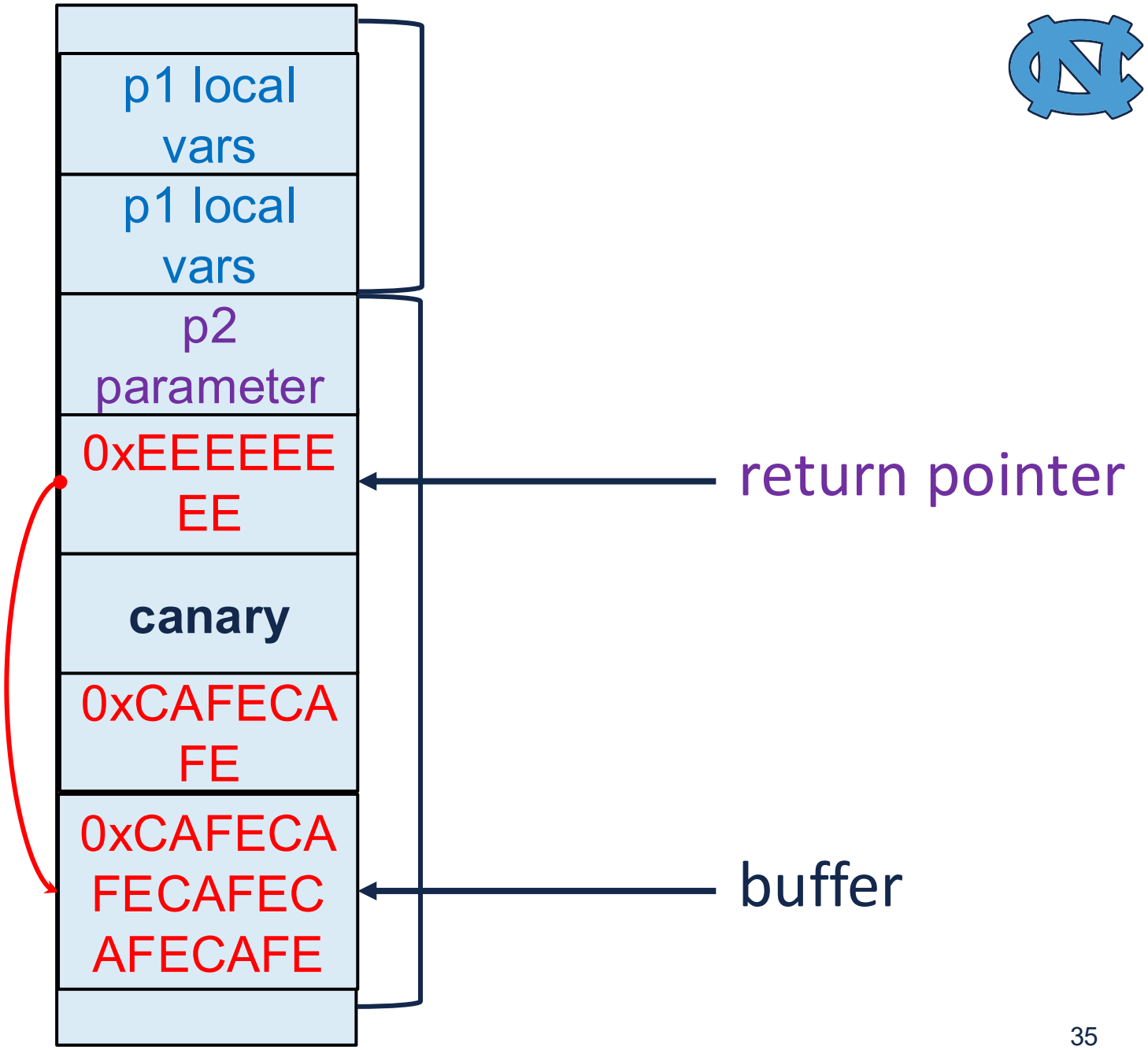


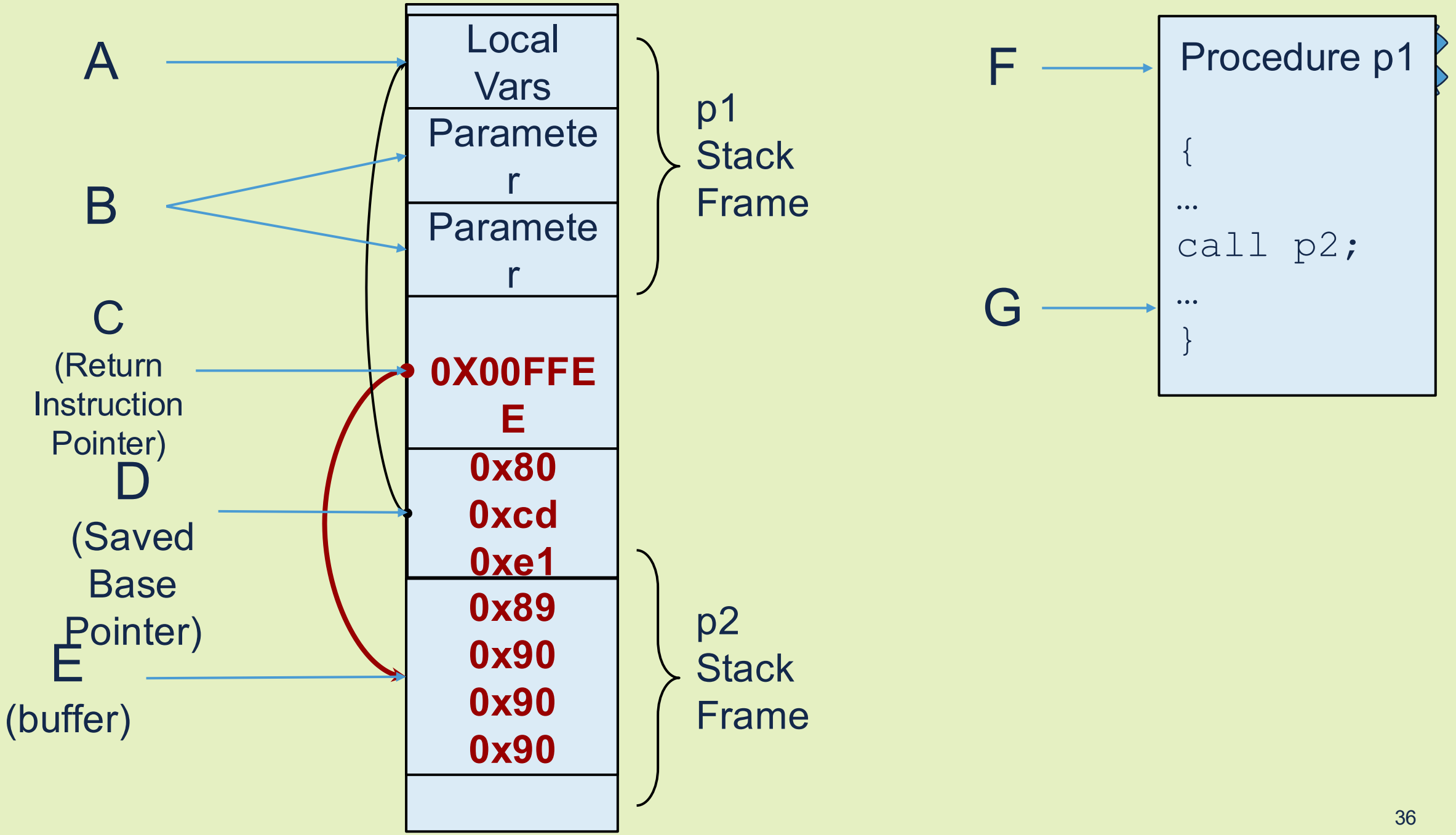
Canary Placement

```
Procedure p1

{
…
call p2;
…
}
```

Executing Code




A
B
C
(Return Instruction Pointer)
D
(Saved Base Pointer)
E
(buffer)
Local Vars
Paramete r
Paramete r
0X00FFE E
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



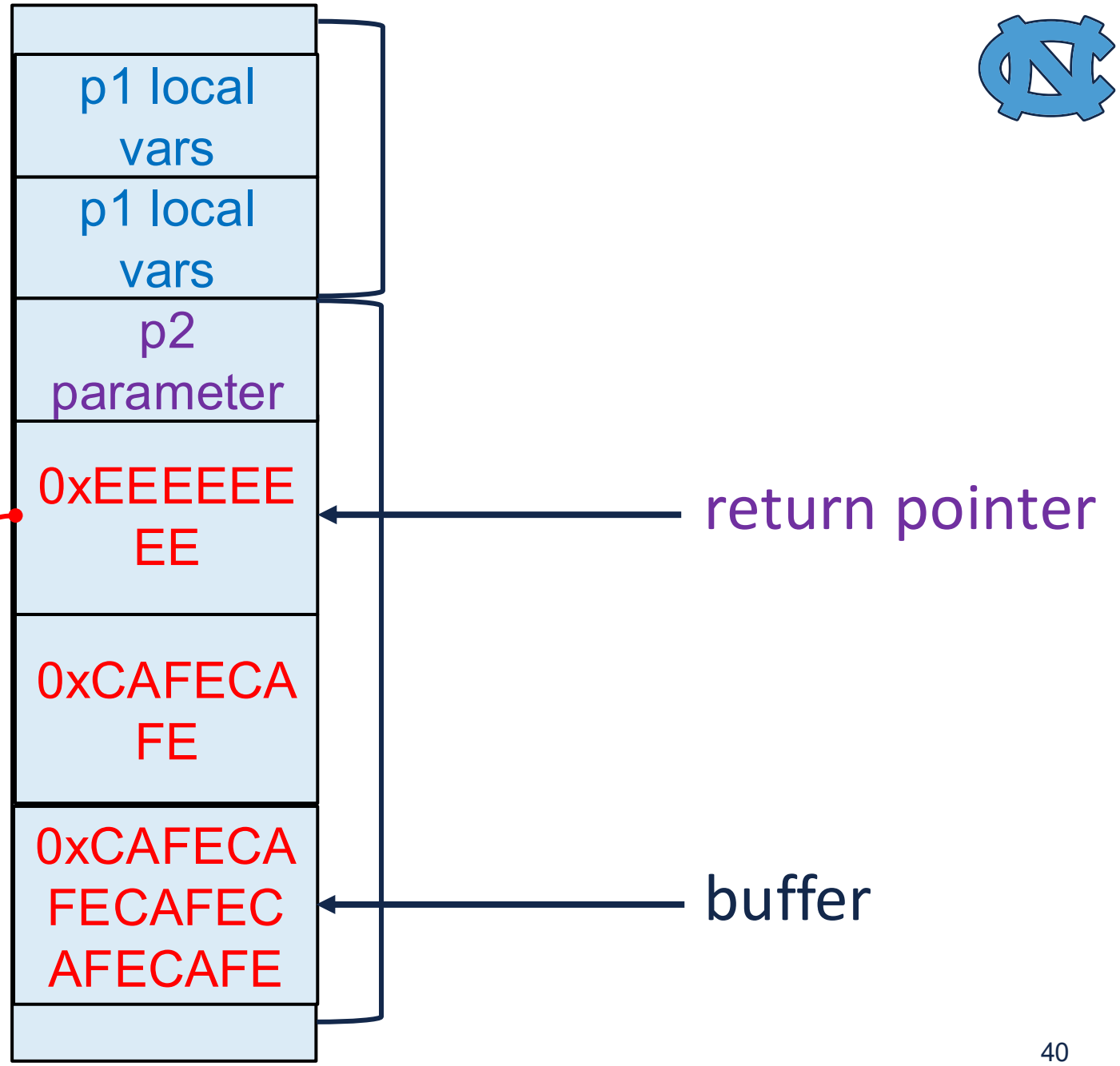
Instead of writing attack code in buffer, point to existing code in the system

Library-code target

```
Procedure p1

{
…
call p2;
…
}
```

C libraries



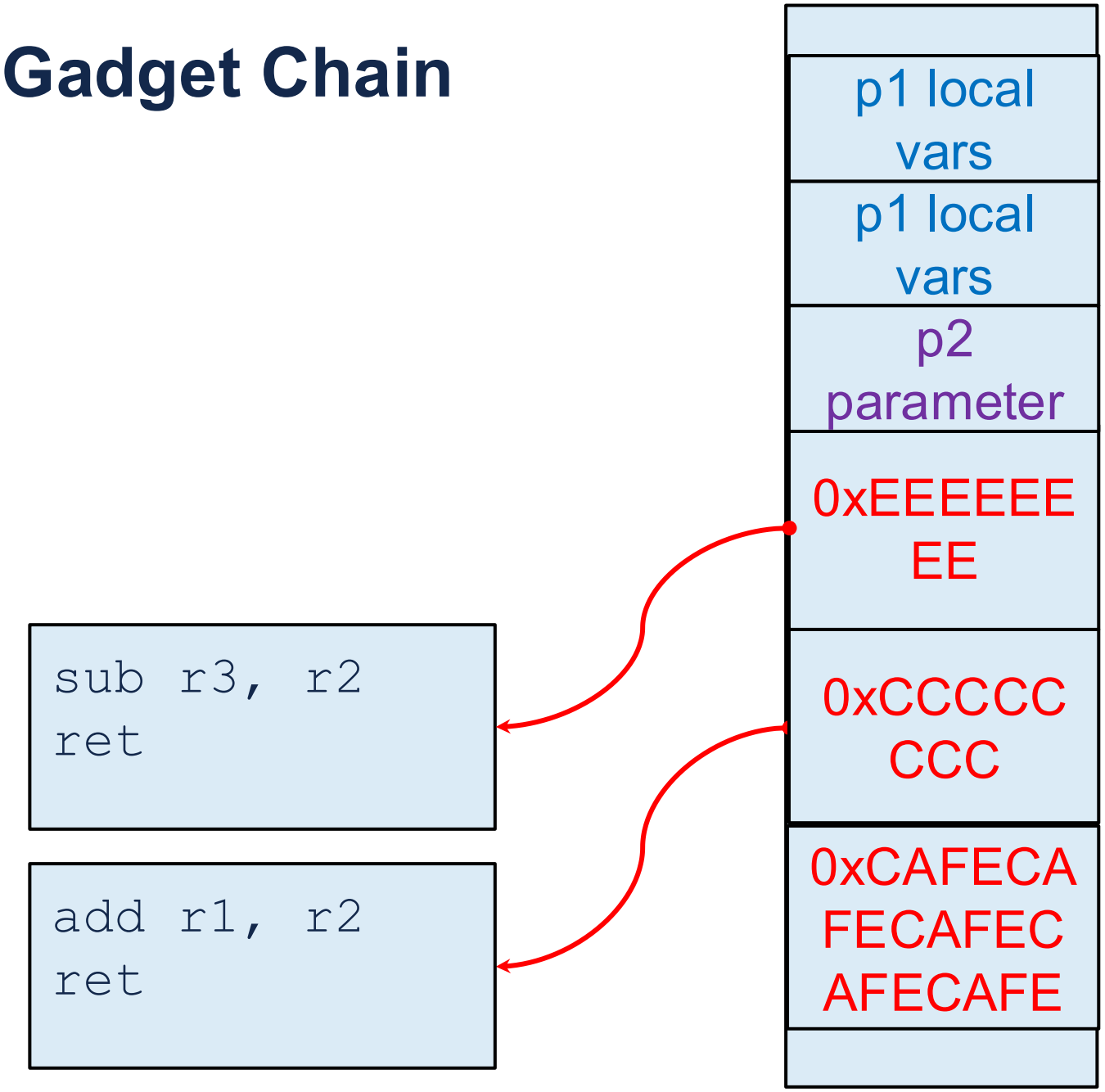
Attack: Return Oriented Programming



Point to gadgets within the existing code

A **gadget**: is a short sequence of instructions already present in executable memory, typically ending in ret, that an attacker chains with other gadgets to perform an operation.

Gadget Chain



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!



3. Backdoors

Backdoor



Def'n: a point of entry that bypasses existing access control measures.
Added for testing/maintenance purposes → grants complete access to the system

```
if(userid = "admin" && passwd = "t3st1ng")
     allow_access();

else
     check_credentials(userid, passwd);
```

Backdoor

Def'n: a point of entry that bypasses existing access control measures.
Added for testing/maintenance purposes → grants complete access to the system

```
if(userid = "admin" && passwd = "t3st1ng")
     allow_access();

else
     check_credentials(u
```

Password hardcoded into program → will exist in the binary!! Very bad ☹

Demo



4. Privilege Escalation

Privilege Escalation



Def'n: change the execution environment to increase privileges

- escaping fixed functionality of a program
- escaping a sandbox to access full filesystem
- gaining root-level privileges
- gaining kernel-mode privileges

Attacks happen in stages.

Ex: First a buffer overflow may be exploited → leads to privilege escalation

Vulnerabilities we've seen all lead to some form of privilege escalation!



Writing Secure Software

Guidelines for Writing Secure Code



- Check bounds
- Validate and sanitize Input
- Use safe utilities
- Grant least privilege
- Sandbox
- Use type safe languages

Bounds Checking



Def'n: check whether a read or write extends beyond the bounds of an allocated object

- Manual code review
- Static analysis
- Dynamic analysis

Static analysis

+ No runtime overhead
+ Analysis of full program
+ Bugs are caught early
- False positives

Dynamic analysis

- Runtime overhead
- Analysis of individual runs
- Bugs are caught during execution
+ No false positives

Input Validation and Sanitization



Never trust user provided input

E.g.,

- Validate buffer access against the buffer size
- Use a **template** to accept user input

Template



Def’n: define acceptable input strings

E.g., (919) 555-1234

Template



Def’n: define acceptable input strings

E.g., (919) 555-1234
1-919-555-1234
+1 919 555 1234
919 555 1234
919.555.1234

Template

Def'n: define acceptable input strings

E.g., (919) 555-1234
1-919-555-1234
+1 919 555 1234
919 555 1234
919.555.1234

Prescribe Good Behavior
(Default Deny)

Use Safe Utilities



Use bounded string manipulation functions from the standard C library, `libc`

E.g., strcpy vs strncpy
strcmp vs strncmp
sprintf vs snprintf

strcpy vs. strncpy



```
char str1[]= "hello";
char str2[5];

strcpy (str2, str1); //copy to terminating null '\0'
strncpy (str2, str1, 5); //copy up to 5 chars
```

strcpy vs. strncpy



```
char str1[]= "hello";
char str2[5];

strcpy (str2, str1); //copy to terminating null '\0'
strncpy (str2, str1, 5); //copy up to 5 chars
```



Grant Least Privilege



Def’n: a subject should be granted only the privileges necessary to complete its work

E.g., Your browser does not need admin privileges

Grant Least Privilege



Def'n: a subject should be granted only the privileges necessary to complete its work

E.g., Your browser does not need admin privileges

Principle of Least Privilege

Grant Least Privilege



- Give code only the access required to complete the work

E.g., a calculator does not need access to the filesystem

Grant Least Privilege



- Build modular code

E.g., separate HTML rendering from filesystem access in your browser

Principle of Least Privilege



Limits the damage that occurs when security is compromised

Sandboxing



Def’n: allow code access to only a portion of the filesystem

E.g., The browser can access only files in `/tmp`

Type Safety



Def'n: Memory is only ever accessed according to its type

E.g.,

- Memory allocated as an int is only ever used as an int
- Indexing into an array can only ever access the memory allocated as part of the array

Memory Safety



Def'n: A program is memory safe if it never accesses undefined memory

- buffer overflows
- null pointer dereference
- use-after-free
- use of uninitialized memory
- illegal free

violate memory safety

Memory Safety



Def'n: A program is memory safe if it never accesses undefined memory

- buffer overflows
- null pointer dereference
- use-after-free
- use of uninitialized memory
- illegal free

violate memory safety

Type safe languages produce memory safe programs

WS Q2



Heartbleed



logo: https://en.wikipedia.org/wiki/Heartbleed

Terminology

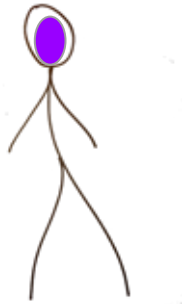


- Buffer overflow
- Buffer overrun
- Buffer overflow attack
- Stack smashing



reading past the end of an allocated array

TLS Heartbeat Protocol



TLS (Transport Layer Security): a cryptographic protocol that secures communication over a network



Heartbeat request: 3 bytes, “cat”

Heartbeat reply: “cat”

liveness

https://xkcd.com/1354/

Heartbleed Attack





Heartbeat request: 64 kbytes, “cat”

Heartbeat reply: “cat0xcafep2ssw0rd!”

https://xkcd.com/1354/

History of Heartbleed



- Bug introduced to code repository Dec 2011
- Request For Comment (RFC) published Feb 2012
- Included and enabled by default in OpenSSL March 2012
- First discovered March 2014

500,000 servers

→ 17%

of web servers

OpenSSL is the library implementing TLS



Buffer overflow and overrun vulnerabilities remain a challenge!