Detecting Code Reuse Attacks

Using Dyninst Components

What are code reuse attacks?

Goal: piece together *gadgets* of original code such that their execution effects some malicious intent.

Motive: defeat current protections that prevent code injection or execution from the stack or the heap.

Assumption: program has vulnerability that allows stack to be overwritten.

Technique: chain together gadgets with control transfers:
- returns (“return-oriented programming”)
- jumps (“jump-oriented programming”)

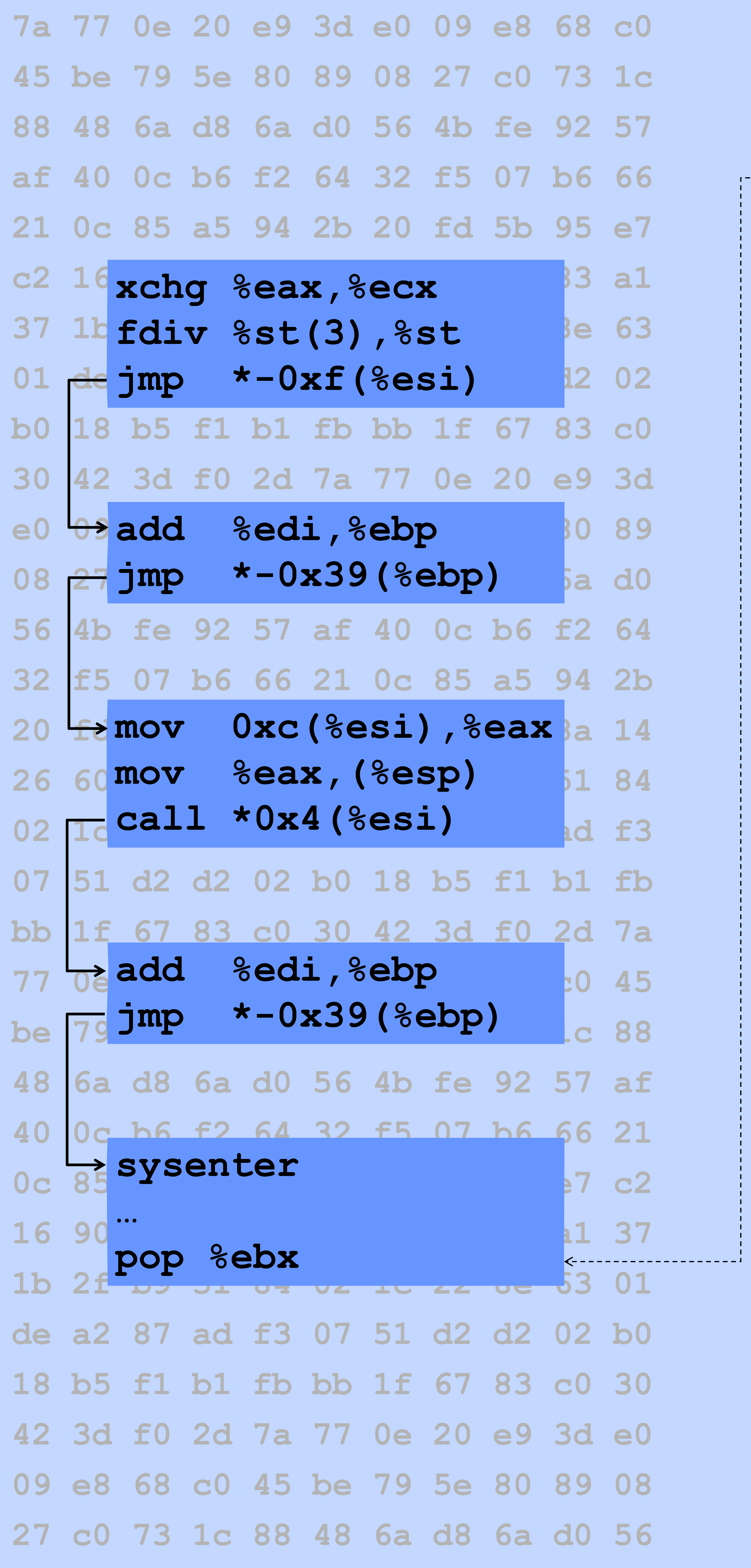
How do we detect an attack?

Valid program counter
- Prevent attacks that rely on code outside the valid code sections of the binary
- Prevent attacks that use unaligned instructions.

Valid callstack
- Prevent attacks from executing instructions that have not been reached via valid inter- and intraprocedural control flow transfers.

Valid system call
- Prevent attacks from executing system calls that are not valid in the context of the current PC and callstack.

Gadgets constructed from a conventional program binary



```
xchg %eax,%ecx
fdiv %st(3),%st
jmp  *-0xf(%esi)

add  %edi,%ebp
jmp  *-0x39(%ebp)

mov  0xc(%esi),%eax
mov  %eax,(%esp)
call *0x4(%esi)

add  %edi,%ebp
jmp  *-0x39(%ebp)

sysenter
…
pop %ebx
```

Example based on exploit presented in Bletsch et al., 2011.

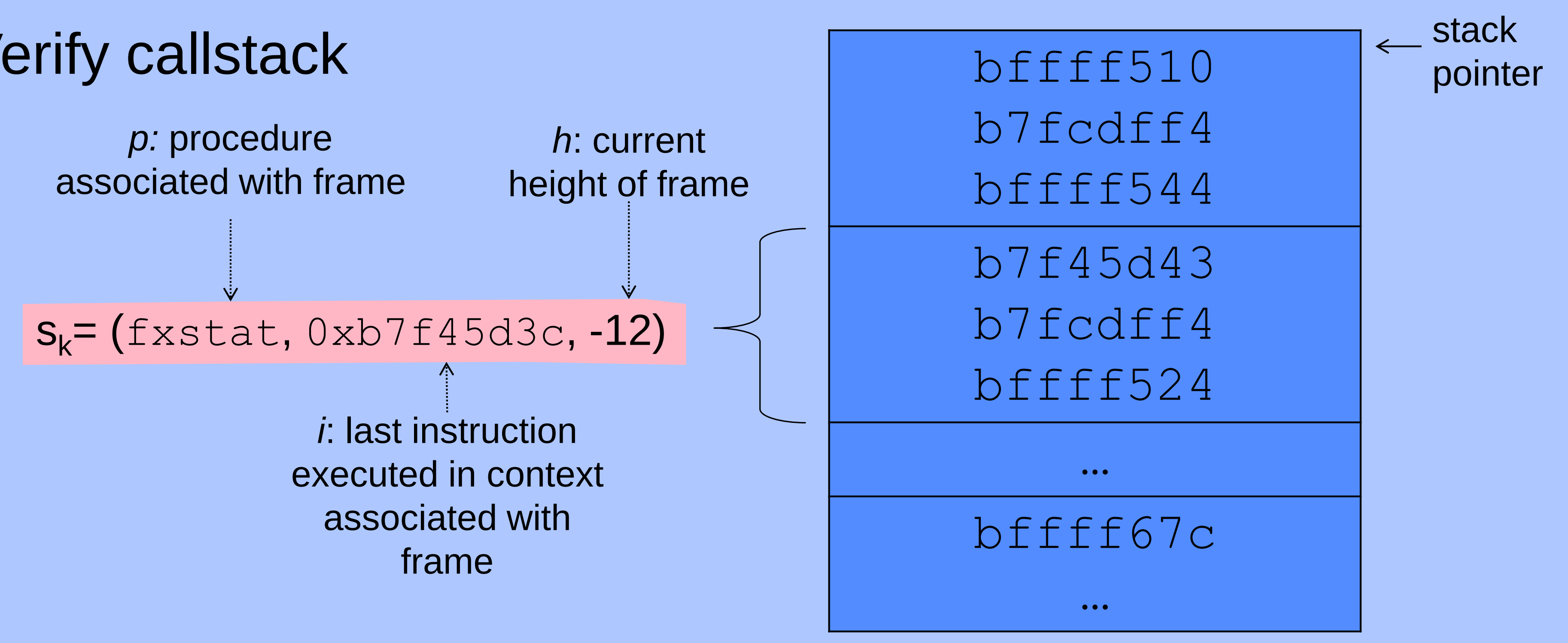
Verify program state during program execution *at system calls*

1 Verify program counter

`0xB7FE3424`

Check that PC points to instruction in original program: identify basic block that contains this address, disassemble and verify that the address is on instruction boundary.

2 Verify callstack



p: procedure associated with frame
h: current height of frame
s_k= (fxstat, 0xb7f45d3c, -12)
i: last instruction executed in context associated with frame
stack pointer
bffff510
b7fcdff4
bffff544
b7f45d43
b7fcdff4
bffff524
…
bffff67c
…

a *Check that frame has valid stack frame height:* calculate expected stack height for $i \in p$; verify that based on this height, the return address in the caller frame is valid (follows a call instruction).

b *Check that caller → current frame represents a valid control flow transfer in the program:* verify that there exists an edge in the callgraph from $p \in s_{k-1}$ to $p \in s_k$ at $i \in s_{k-1}$.

! Calculate the return address (RA) in the each caller; here, the last RA = `0xbffff67c`. This is not a valid return address (is not a valid instruction that follows a call). This invalid stack frame indicates non-conformant program execution, and we terminate the process.

3 If system call is `exec`, verify the system call and its first argument

Check that this is a valid call to `exec` *and valid program being passed to* `exec`*:* use backward slicing and symbolic evaluation to calculate the expected system call number and first system call argument at instruction *i*; compare these with the current values in `%eax` and `%ebx`.

- Create or attach to process using **ProcControlAPI**; register callbacks at system call entry.
- Parse program binary using **ParseAPI**; construct CFG.
- On each callback, verify program state using **InstructionAPI**, **StackwalkerAPI**, **ParseAPI**, and **DataflowAPI**, as described above.