Overview

- Traditionally public enemy #1 in computer security
- Why? Very common and allow the execution of arbitrary code on the victim machine
- Problem stems from poor C programming practices
- Typically involves overflowing a stack-based buffer to rewrite a return address: program flow is thus placed under an attacker's control
- Material drawn from:
  - *Computer Systems: A Programmer's Perspective*, by Bryant and O'Hallaron
  - *Buffer Overflows: Attacks and Defences* by Cowen et al

Buffer Overflows

- In a traditional buffer overflow attack a write to a buffer on the stack exceeds the capacity of the buffer and neighbouring data on the stack is overwritten
- The return address placed on the stack by the `call` can be altered to point into the buffer itself, the contents of which are, obviously, under the attacker's control
- If machine instructions are placed in the buffer then on returning from the vulnerable procedure these instructions will be executed
- If we have those instructions then `exec` a remotely controllable shell then we have gained full control over the vulnerable host

Buffer Overflows

- The attack is even more effective if the vulnerable process is running with elevated privileges since these high privileges will be inherited by the shell across `exec`
- Dangerous calls include `gets`, `strcpy`, `scanf`, `memcpy` etc.
- These are functions that carry out no bounds checking while writing to memory
- The attached slides on buffer overflow vulnerabilities taken from *CS:APP* by Bryant and O'Hallaron

Buffer Overflow Defences

- If buffer overflow vulnerabilities could be eliminated a large portion of the most serious security threats would disappear with them
- So we also look at some defensive measures that have been implemented to mitigate the threats

Buffer Overflow Defences

- The goal of a buffer overflow attack is to subvert program flow and if that program has sufficient privileges, to control the host
- Buffer overflows dominate the class of remote network penetration attacks
- An attacker usually has two subgoals:
  1. Place suitable code in the process address space
  2. Jump to that code
- To achieve the first goal an attacker can inject or use code already inside process address space

Buffer Overflow Defences

- To achieve the second goal an attacker overflows a buffer to corrupt adjacent program state
- If overflowing a stack-based buffer the usual target is the vulnerable frame's return address: this is a traditional stack smashing attack
- Function pointers or security-sensitive flags may also be targeted and rewritten
- Also, `longjmp` buffers may be corrupted to cause execution to jump to an arbitrary location where the usual `setjmp(buffer)` and `longjmp(buffer)` sequence is exploited (used in stack unwinding)

Buffer Overflow Defences

There are four basic approaches:

1. Write correct code: nice idea, feasible?
2. OS approach: make the stack non-executable
3. Direct compiler approach: do bounds checking
4. Indirect compiler approach: StackGuard

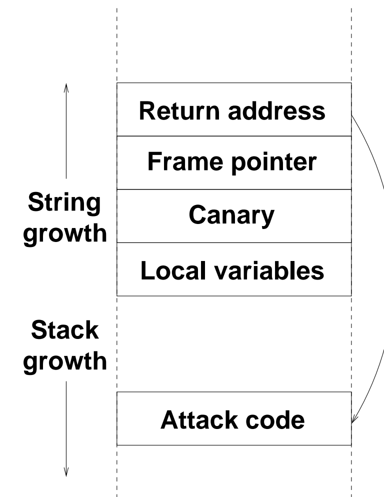
Code pointer integrity checking

- Instead of preventing overflow as with bounds checking we accept overflows may occur
- However, we attempt to detect the attack after it has happened but before overwritten code pointers are dereferenced (acted upon)
- StackGuard comes as a gcc patch that instruments (adds additional code to) frame prologue and epilogues

Buffer Overflows Defences

- The enhanced frame set up code places a “canary” value below the return address and the enhanced teardown code checks its integrity before returning
- Stack smashing is thwarted since such attacks obliterate everything between the buffer and the return address including the canary
- Canary forgery is prevented using one of two alternatives:
  1. A terminator canary contains NULL, CR etc.
  2. A random 32-bit canary that is hard to guess

Buffer Overflows Defences



Buffer Overflows Defences

- StackGuard has been proven to provide an effective defence against existing and new buffer overflow attacks
- System compatibility and performance are preserved e.g. has been successfully combined with ssh (secure shell) and Apache

- Implementation of Unix function `gets`
  - No way to specify limit on number of characters to read

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getc();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getc();
    }
    *p = '\0';
    return dest;
}
```

- Similar problems with other Unix functions
  - `strcpy`: Copies string of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification

Buffer Overflows

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
int main()
{
  printf("Type a string:");
  echo();
  return 0;
}
```

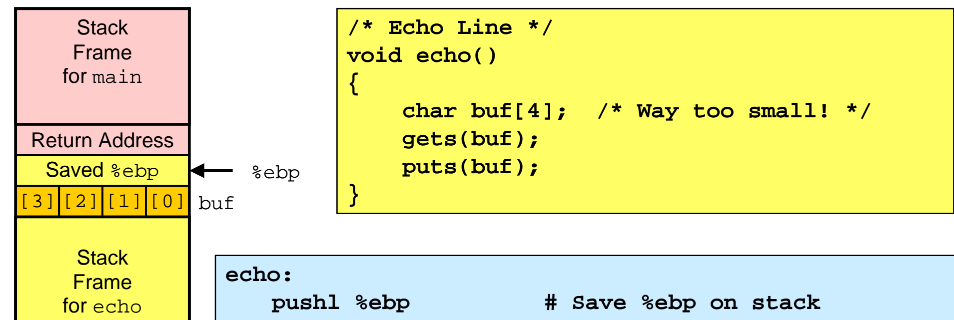

Buffer Overflows

```
unix>./bufdemo
Type a string:123
123
```

```
unix>./bufdemo
Type a string:12345
Segmentation Fault
```

```
unix>./bufdemo
Type a string:12345678
Segmentation Fault
```


Buffer Overflows

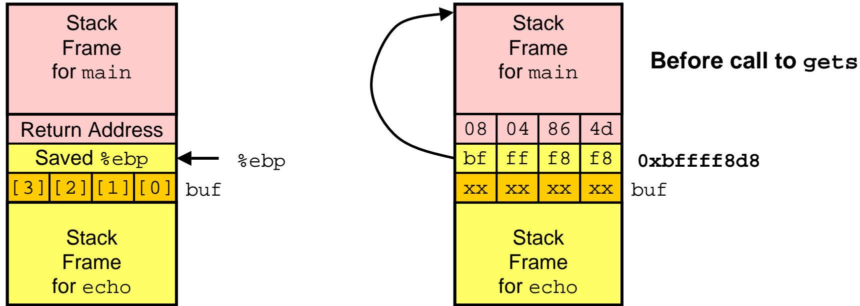



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushl %ebp            # Save %ebp on stack
    movl %esp,%ebp
    subl $20,%esp         # Allocate space on stack
    pushl %ebx            # Save %ebx
    addl $-12,%esp        # Allocate space on stack
    leal -4(%ebp),%ebx    # Compute buf as %ebp-4
    pushl %ebx            # Push buf on stack
    call gets             # Call gets
    . . .
```

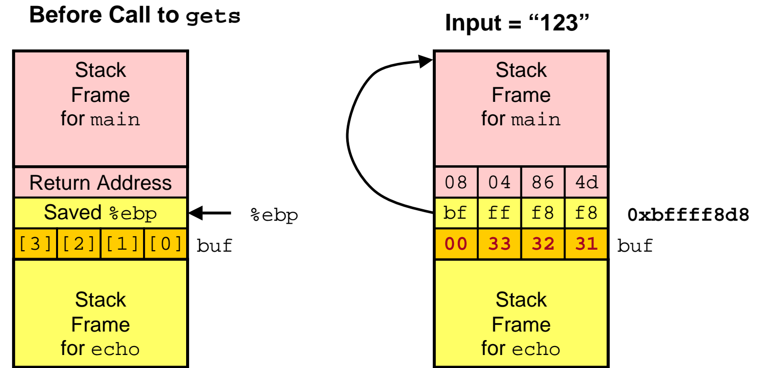
Buffer Overflows

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x *(unsigned *)$ebp
$1 = 0xbffff8f8
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x804864d
```



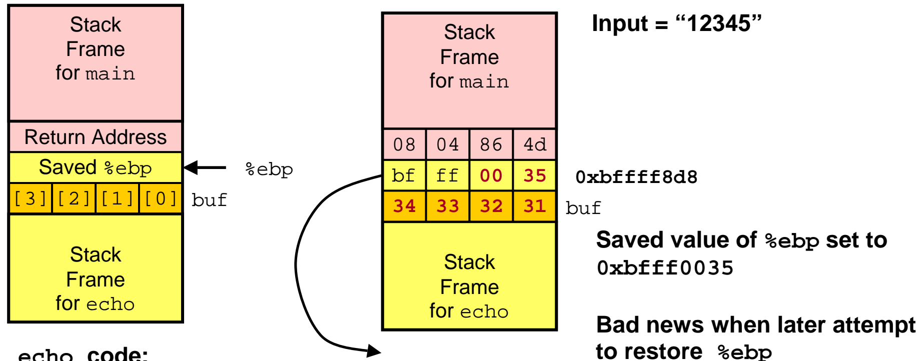

```
8048648: call 804857c <echo>
804864d: mov  0xffffffe8(%ebp),%ebx # Return Point
```


Buffer Overflows



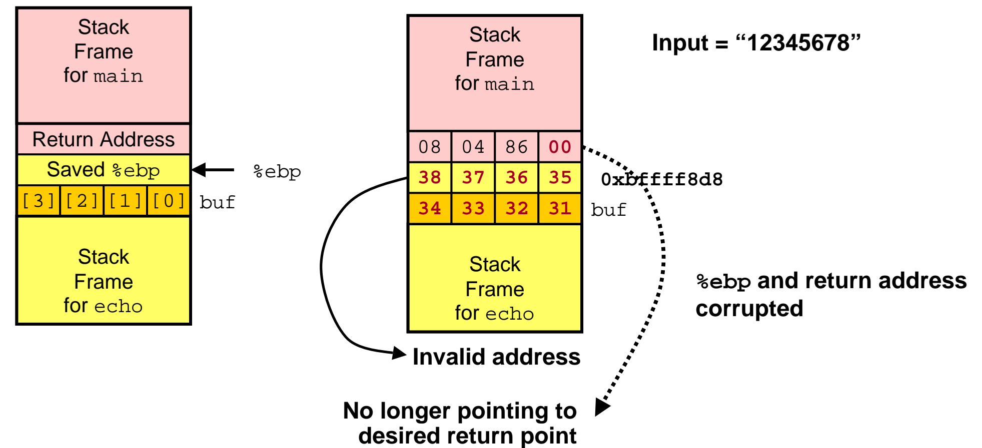
No Problem

Buffer Overflows



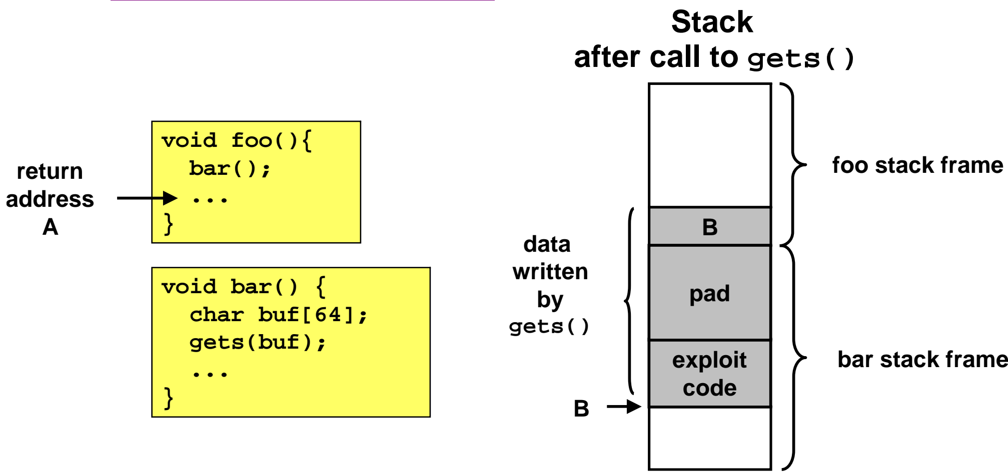

```
8048592: push   %ebx
8048593: call   80483e4 <_init+0x50>  # gets
8048598: mov    0xffffffe8(%ebp),%ebx
804859b: mov    %ebp,%esp
804859d: pop    %ebp      # %ebp gets set to invalid value
804859e: ret
```


Buffer Overflows




```
8048648: call 804857c <echo>
804864d: mov  0xffffffe8(%ebp),%ebx # Return Point
```

Buffer Overflows



- Input string contains byte representation of executable code
- Overwrite return address with address of buffer

Buffer Overflows

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- Use Library Routines that Limit String Lengths
  - `fgets` instead of `gets`
  - strncpy instead of `strcpy`
  - Don't use `scanf` with `%s` conversion specification
    - Use `fgets` to read the string