The Dark Side of Data-Code Duality

- The stack frame organization on x86 mixes both data and pointers to code.
  - Data: Local variables.
  - Pointers to code: Return address.

Position	Contents	Frame
`8n+16(%rbp)`	memory argument eightbyte n	Previous
	...	
`16(%rbp)`	memory argument eightbyte 0	
`8(%rbp)`	return address	Current
`0(%rbp)`	previous `%rbp` value	
`-8(%rbp)`	unspecified	
	...	
`0(%rsp)`	variable size	
`-128(%rsp)`	red zone	

The Dark Side of Data-Code Duality

- The stack frame organization on x86 mixes both data and pointers to code.
  - Data: Local variables.
  - Pointers to code: Return address.
- What happens if we write some inappropriate bit pattern into the return address location?
  - We will corrupt the stack, subvert the control flow of the program, and cause it to demonstrate unexpected behavior.

Position	Contents	Frame
8n+16(%rbp)	memory argument eightbyte n	Previous
	. . .	
16(%rbp)	memory argument eightbyte 0	
8(%rbp)	return address	Current
0(%rbp)	previous %rbp value	
-8(%rbp)	unspecified	
	. . .	
0(%rsp)	variable size	
-128(%rsp)	red zone	

The Dark Side of Data-Code Duality

- The stack frame organization on x86 mixes both data and pointers to code.
  - Data: Local variables.
  - Pointers to code: Return address.
- What happens if we write some inappropriate bit pattern into the return address location?
  - We will corrupt the stack, subvert the control flow of the program, and cause it to demonstrate unexpected behavior.

Position	Contents	Frame
`8n+16(%rbp)`	memory argument eightbyte n	Previous
	...	
`16(%rbp)`	memory argument eightbyte 0	
`8(%rbp)`	return address	Current
`0(%rbp)`	previous `%rbp` value	
`-8(%rbp)`	unspecified	
	...	
`0(%rsp)`	variable size	
`-128(%rsp)`	red zone	

- This corruption can be unintended or malicious.
  - A common source of such corruption is known as buffer overflow.

Buffer Overflow Example

```
char *gets(char *s) {
  int c;
  char *dest = s;
  while ((c = getchar()) != '\n' && c != EOF)
    *dest++ = c;
  if (c == EOF && dest == s) return NULL;
  *dest++ = '\0';
  return s;
}
```

Buffer Overflow Example

```
char *gets(char *s) {
  int c;
  char *dest = s;
  while ((c = getchar()) != '\n' && c != EOF)
    *dest++ = c;
  if (c == EOF && dest == s) return NULL;
  *dest++ = '\0';
  return s;
}
```

```
void echo() {
  char buf[8];
  gets(buf);
  puts(buf);
}
```

Buffer Overflow Example

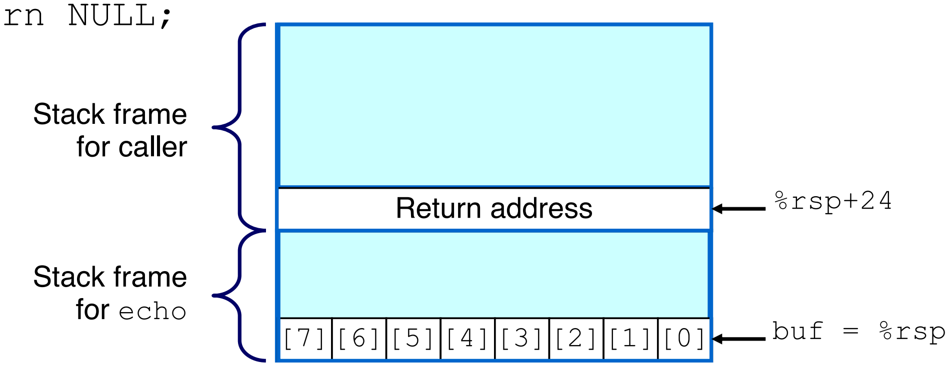
```
char *gets(char *s) {
  int c;
  char *dest = s;
  while ((c = getchar()) != '\n' && c != EOF)
    *dest++ = c;
  if (c == EOF && dest == s) return NULL;
  *dest++ = '\0';
  return s;
}
```

```
void echo() {
  char buf[8];
  gets(buf);
  puts(buf);
}
```

```
echo:
  subq   $24, %rsp        # Allocate 24 bytes on stack
  movq   %rsp, %rdi       # Compute buf as %rsp
  call   gets             # Call gets
  movq   %rsp, %rdi       # Compute buf as %rsp
  call   puts             # Call puts
  addq   $24, %rsp        # Deallocate stack space
  ret                     # Return
```

Buffer Overflow Example

```
char *gets(char *s) {
  int c;
  char *dest = s;
  while ((c = getchar()) != '\n' && c != EOF)
    *dest++ = c;
  if (c == EOF && dest == s) return NULL;
  *dest++ = '\0';
  return s;
}
```




```
void echo() {
  char buf[8];
  gets(buf);
  puts(buf);
}
```

```
echo:
  subq   $24, %rsp        # Allocate 24 bytes on stack
  movq   %rsp, %rdi       # Compute buf as %rsp
  call   gets             # Call gets
  movq   %rsp, %rdi       # Compute buf as %rsp
  call   puts             # Call puts
  addq   $24, %rsp        # Deallocate stack space
  ret                     # Return
```

Consequences of Buffer Overflow

- In our example, the consequence was unexpected behavior, possibly a core dump.

Consequences of Buffer Overflow

- In our example, the consequence was unexpected behavior, possibly a core dump.
- More pernicious uses are possible.
  - Feed the program with a string that contains the byte encoding of some executable (aka the exploit code), plus some extra bytes that overwrite the return address with a pointer to the exploit code.
  - In this case, executing the `ret` instruction causes the program to jump to the exploit code.
  - The exploit code can now use a system call to start up a shell program.
  - Or the exploit code can perform some otherwise unauthorized task, repair the damage to the stack, and then execute a second `ret`, thereby hiding its malicious behavior from the user.

Thwarting Buffer Overflow: #1

- Stack randomization (or address-space layout randomization)
  - *Insight*:
    In order to insert exploit code, the attacker needs to inject both the code and a pointer to this code as part of the attack string. We need to know the stack address of the string in order to generate the pointer value.
  - *Solution*:
    Make the position of the stack vary from one run of a program to another.

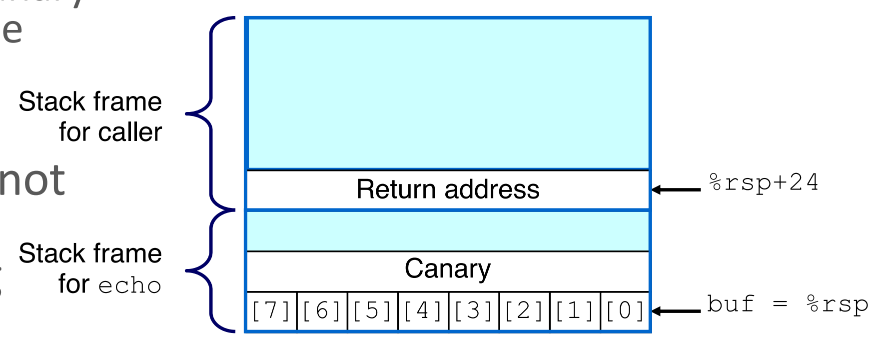
```
int main() {
  long local;
  printf("local at %p\n", &local);
  return 0;
}
```

  - *Weakness*:
    Still susceptible to brute force attacks.

Thwarting Buffer Overflow: #2

- Stack Corruption Detection
  - *Insight*:
    If a buffer overflows, it will leave a trace in memory.
  - *Solution*:
    Put a stack protector into the generated code to detect buffer overflows.
    - Store a special canary value (aka guard value) in the stack frame between any local buffer and the rest of the stack state.
    - Canary value is generated randomly for each run of program.
    - Before restoring register state and returning from the function, the canary value has been checked for possible alteration.
  - *Weakness*:
    A small performance penalty. Does not protect against other ways of corrupting the state of an executing program.

```
echo:
   subq    $24, %rsp
   movq    %fs:40, %rax
   movq    %rax, 8(%rsp)
   xorl    %eax, %eax
   movq    %rsp, %rdi
   call    gets
   movq    %rsp, %rdi
   call    puts
   movq    8(%rsp), %rax
   xorq    %fs:40, %rax
   je      .L9
   call    __stack_chk_fail
.L9:
   addq    $24, %rsp
   ret
```



Thwarting Buffer Overflow: #3

- Limiting Executable Code Regions
  - *Insight*:
    The exploit won't work unless the exploit code is actually executable.
  - *Solution*:
    Limit the memory regions that can hold executable code.
    - Only the `.text` section should be executable.
    - Use memory protection mechanisms in the operating system to accomplish this.
  - *Weakness*:
    - Needs to be modified for JIT compilers (or dynamic compilers in general) that generate code at run-time.
    - Still susceptible to return-oriented programming attacks.