CSc 466/566

Computer Security

9 : Buffer Overflow II

Version: 2019/10/02 13:53:16

Department of Computer Science
University of Arizona

collberg@gmail.com

Christian Collberg

Outline

Languages of choice

C: "A language that combines all the elegance and power of assembly language with all the readability and maintainability of assembly language." – *New Hacker's Dictionary*

C++: "An octopus made by nailing extra legs onto a dog." – *Steve Taylor*

More quotes. . .

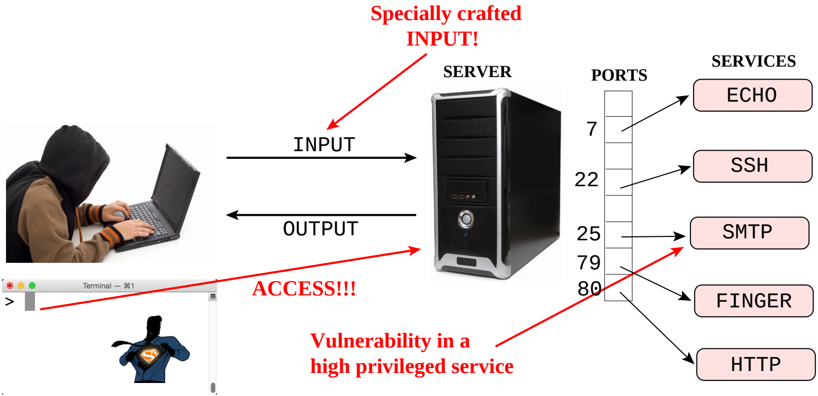
Bertrand Meyer: There are only two things wrong with C++: The initial concept and the implementation.

Jamie Zawinski: `/* C has all the expressive power of two dixie cups and a string. */`

Drew Olbrich: C++ is like jamming a helicopter inside a Miata and expecting some sort of improvement.

smcameron: I think maybe the guy who invented C++ doesn't know the difference between increment and excrement.

Attacking from the network



Attacking from the network

- The adversary tries to exploit a vulnerability in one of the network services to get access to the machine.
- The most common such vulnerability is a buffer overflow.

Attack idea

1. Find a program P with a vulnerability.
2. Craft a special input I that exploits the vulnerability.
3. Create a payload (part of the input I) that (typically) gives us a shell on the target machine.

Buffer overflow idea

1. Find a C/C++ program P that declares a buffer (an array) as a local variable.
2. See if there's an input to P that will cause the program to write outside the buffer (overflow it).
3. Craft a special input I that
   - is large enough to overflow the buffer;
   - contains the payload; and
   - overwrites the return address, such that, when the function returns, we jump to the payload.

Outline

Buffer overflow idea

- To illustrate, consider the (unrealistic) example below.
- The program finds where on the stack the return address is stored, by
  1. declaring a local variable, `anchor`;
  2. taking its address;
  3. adding an increasing offset to `anchor`;
  4. overwriting this new address with the address of `payload`;
  5. returning;
- When we've found the right offset, `payload` will be called when `foo` returns!

Buffer overflow example I

buf.c:

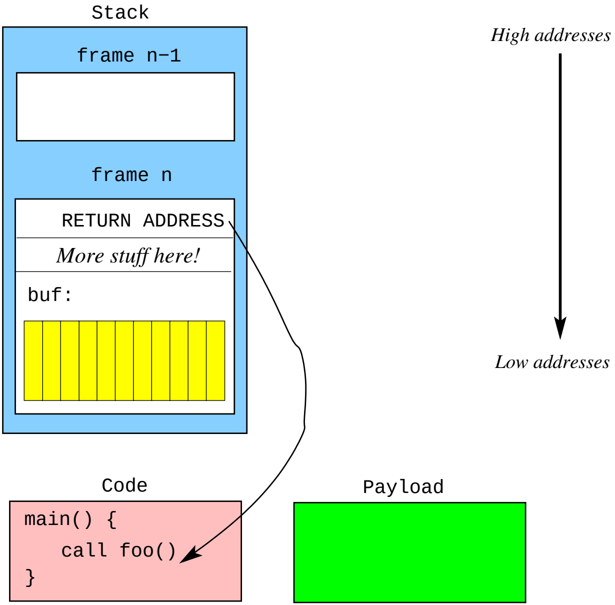
```
void printit() {printf("CALLED payload!\n");}
void payload(){printit();}
int count;

int foo(){
  printf("ENTER foo\n");
  int i;
  long* buf[10];
  for(i=0; i<count; i++)
    buf[i] = (unsigned long*)&payload;
  printf("RETURN foo\n");
}

int main(int argc, char** argv){
  count = strtoul(argv[1],NULL,10);
  foo();
}
```

Buffer overflow example I. . .

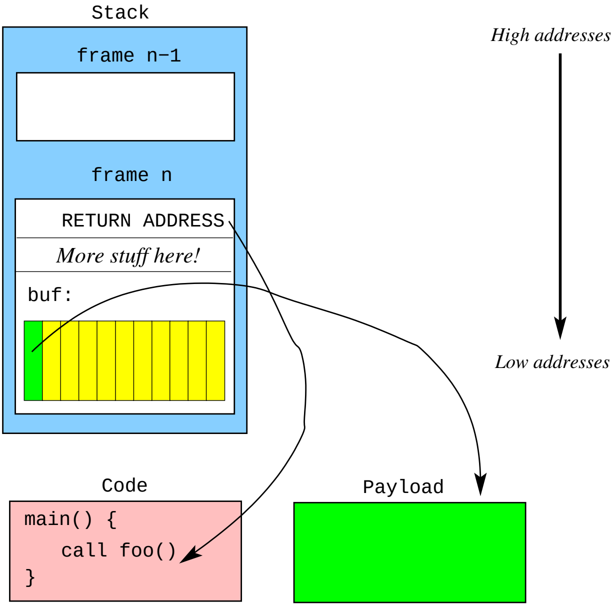
To execute:

```
> gcc -m32 -o buf buf.c
> ./buf 10
ENTER foo
RETURN foo
> ./buf 14
ENTER foo
RETURN foo
CALLED payload!
```

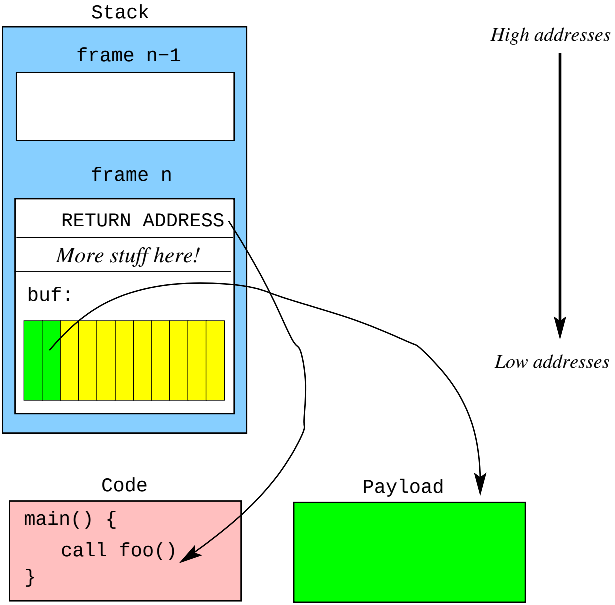
Buffer overflow example I. . .



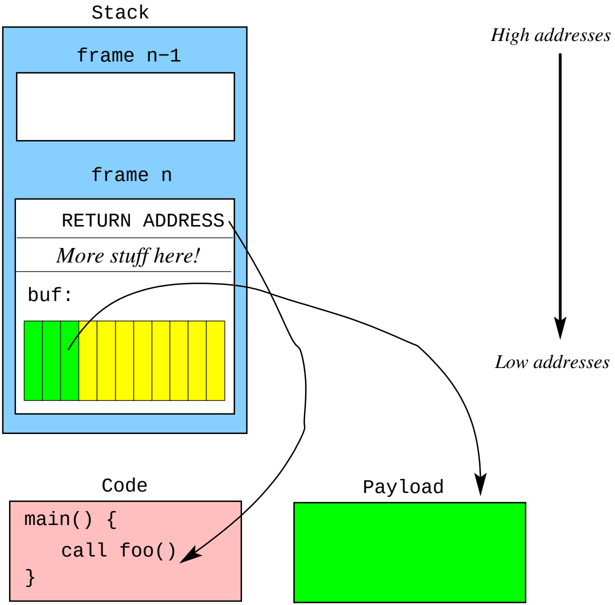
Buffer overflow example I...



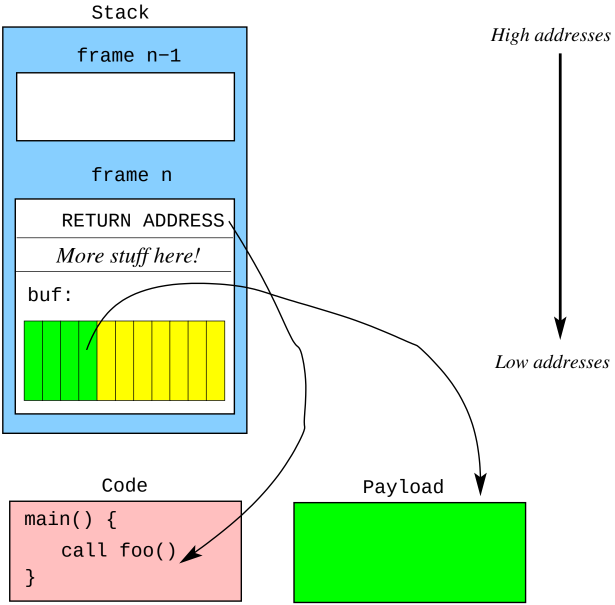
Buffer overflow example I. . .



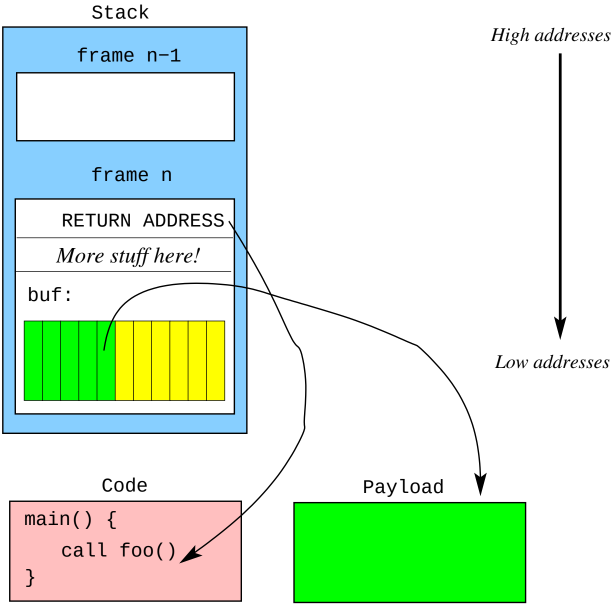
Buffer overflow example I. . .



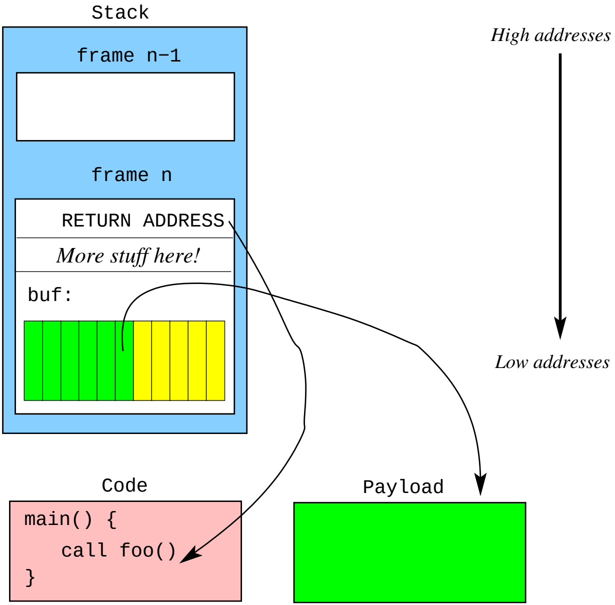
Buffer overflow example I...



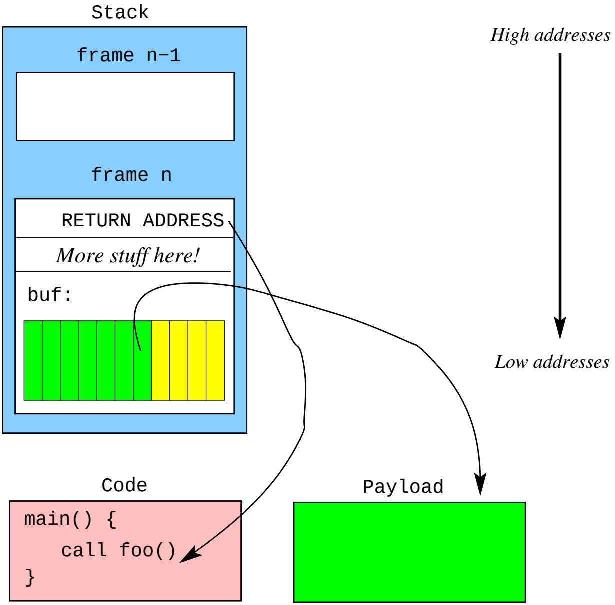
Buffer overflow example I. . .



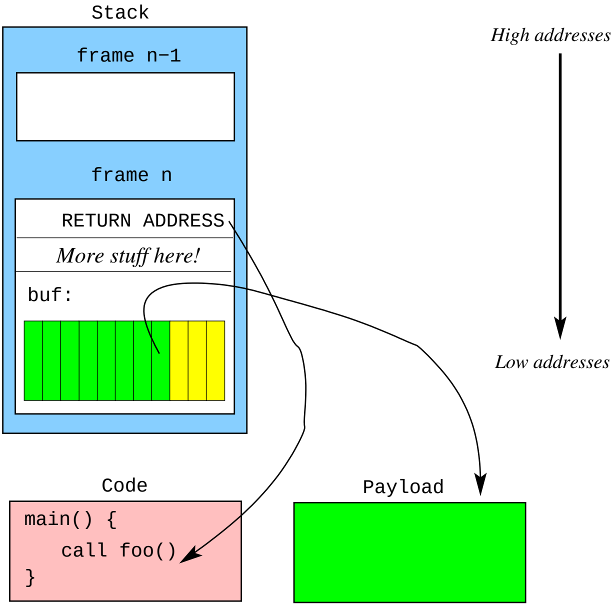
Buffer overflow example I. . .



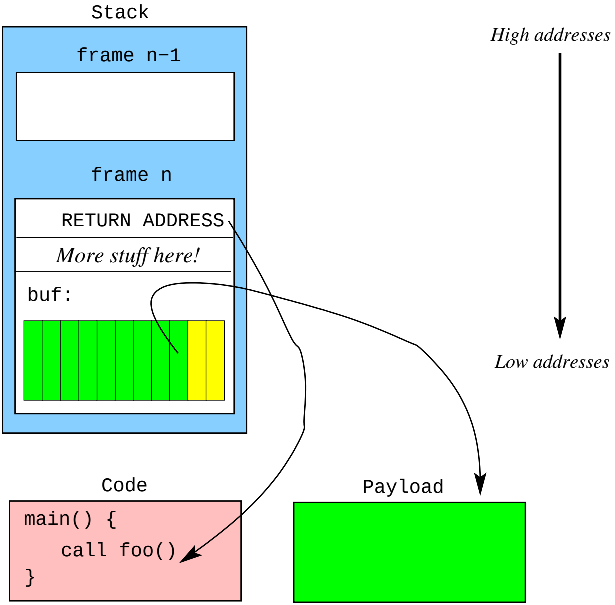
Buffer overflow example I. . .



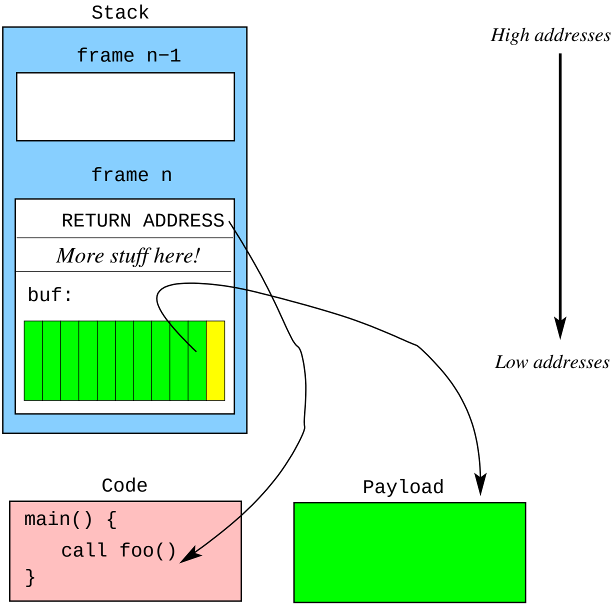
Buffer overflow example I...



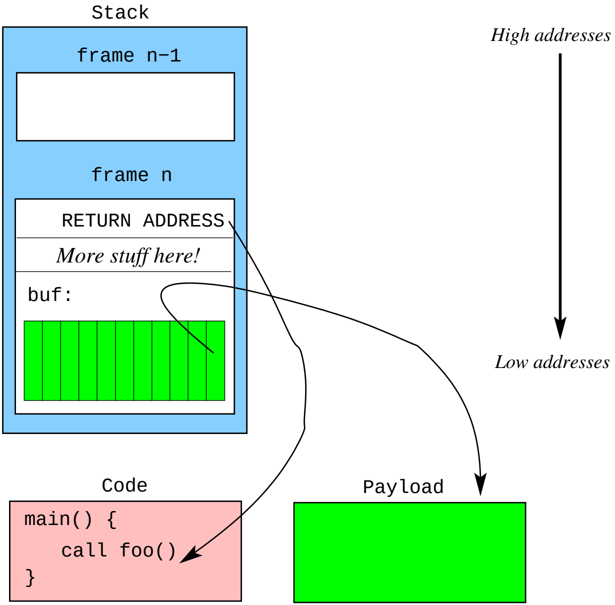
Buffer overflow example I...



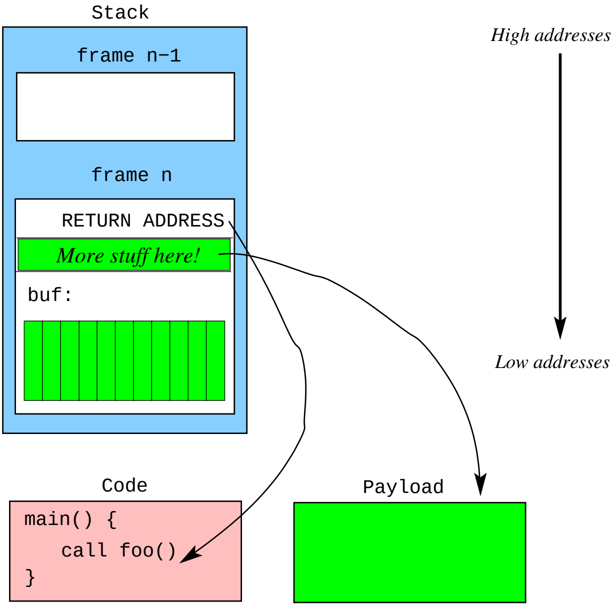
Buffer overflow example I...



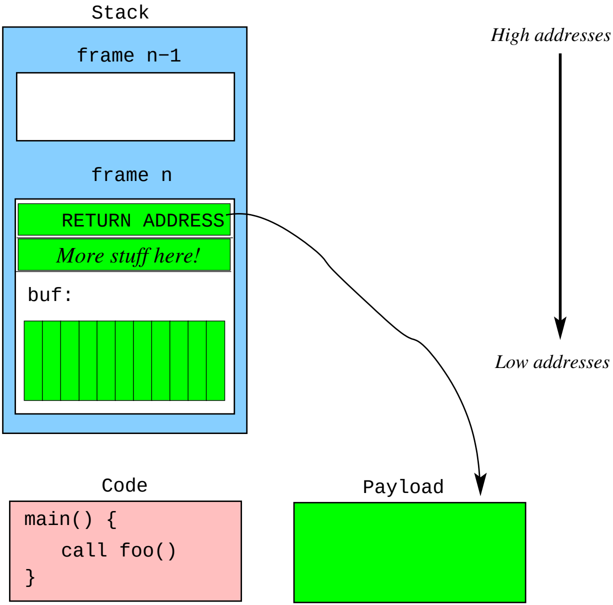
Buffer overflow example I. . .



Buffer overflow example I...



Buffer overflow example I. . .



Invoking a shell!

shell.c:

```
const char shellcode[] =
    "\x31\xc0" /* xorl    %eax,%eax   */
    "\x50"     /* pushl   %eax        */
    ...;

int count;
int foo(){
  int i;
  long* buf[10];
  for(i=0; i<count; i++)
     buf[i] = (unsigned long*)&shellcode;
}

int main(int argc, char** argv){
  count = strtoul(argv[1],NULL,10);
  foo();
}
```

Invoking a shell...

`shell.c`:

```
> gcc -m32 shell.c -o shell
> ./shell 14
$                <=== Look! A shell prompt!!!
$ echo "hello!"
hello!
```

Automatically finding the return address

- In practice, we don't want to do this by hand.
- Can we write a script instead?
- Below, we use trial-and-error to try all reasonable addresses that could be the location of the return address.
- Note, this is a contrived source-code example. A real adversary only has the binary to work with.

Automatically finding the return address. . .

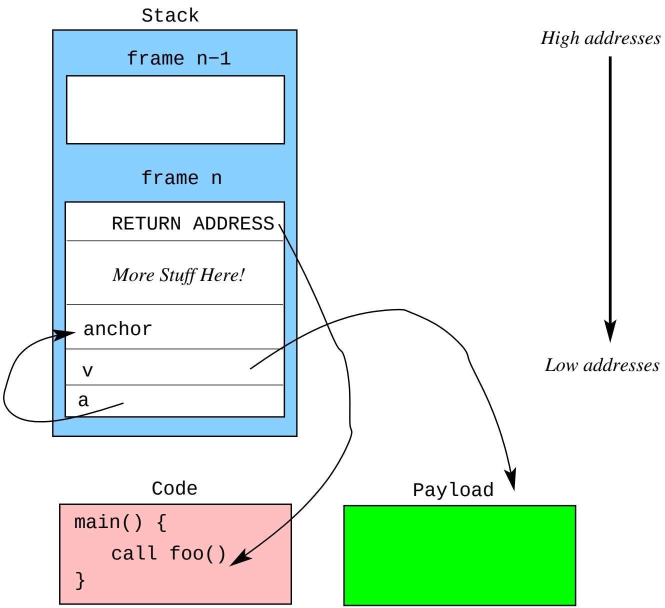
- The example consists of two programs.
- `ret.c` has a local variable `anchor`, and we will add an *offset* to this variable to find how far we have to go to get to the return address.
- `findret` is a script that calls `ret.c` with different *offsets*, from 0 to 64.

Example: finding the return address

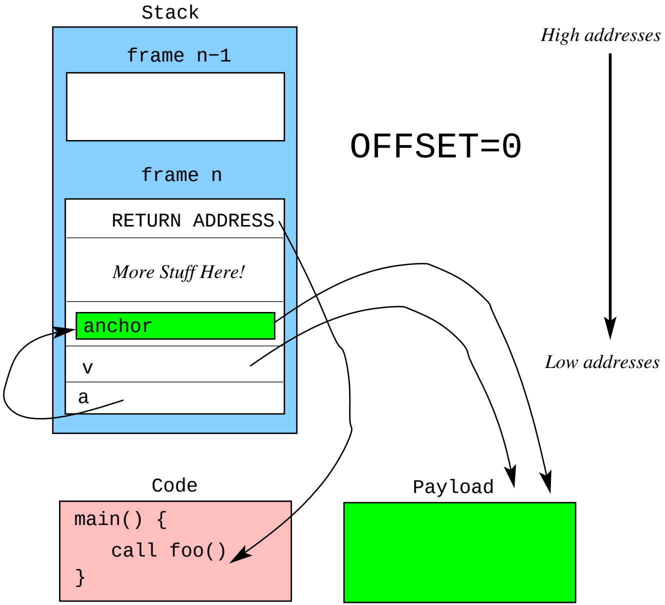
ret.c:

```
unsigned long offset;
void printit() {printf("CALLED payload!\n");}
void payload(){printit();}

int foo(){
  volatile long anchor=-1;
  volatile long* a = (volatile long*)((unsigned
      long)&anchor + offset);
  *a = (unsigned long*)&payload;
}

int main(int argc, char** argv){
  offset = strtoul(argv[1],NULL,10);
  foo();
}
```

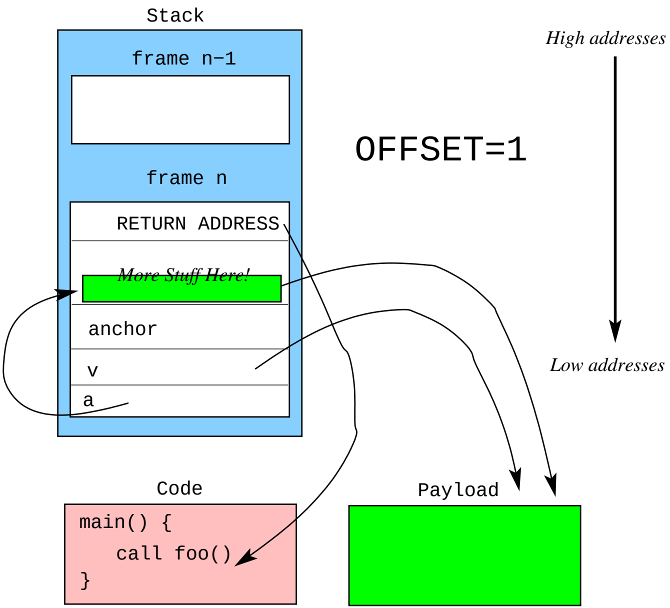
Example: finding the return address. . .



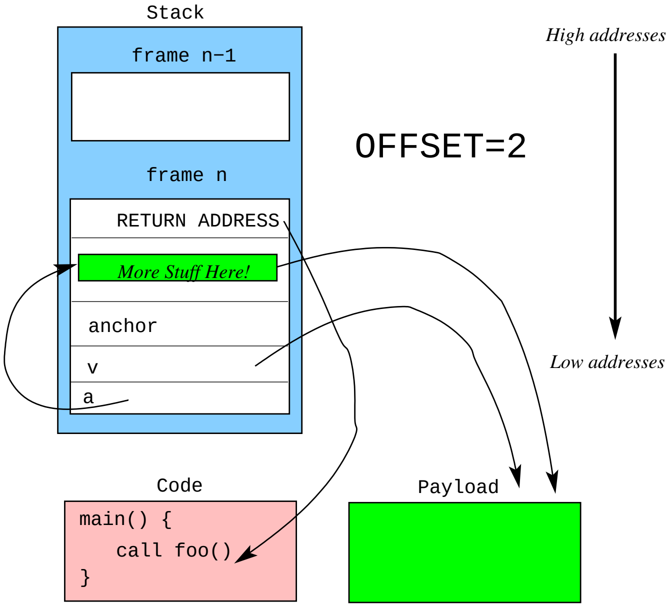
Example: finding the return address. . .



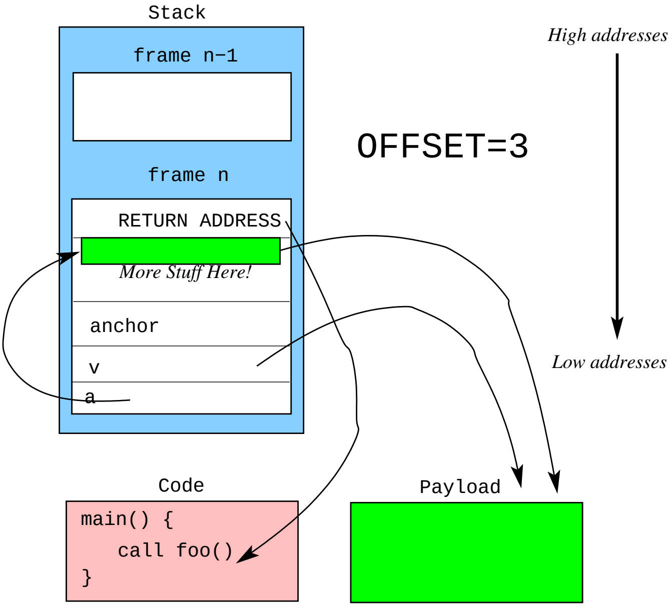
Example: finding the return address. . .



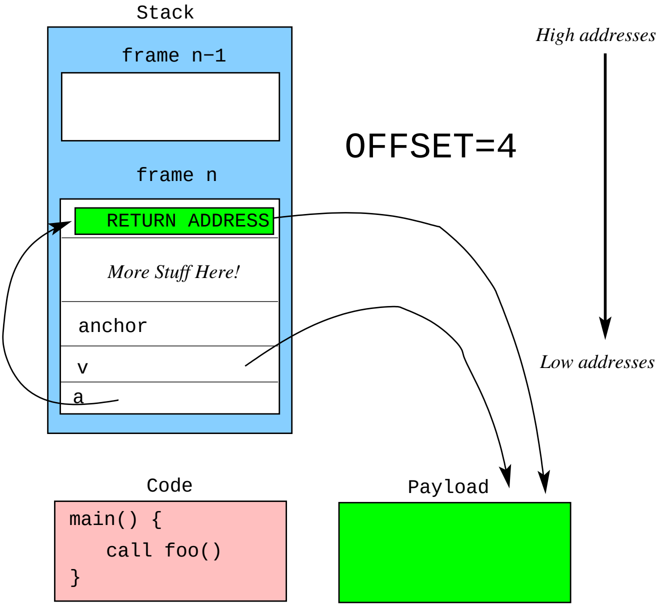
Example: finding the return address. . .



Example: finding the return address. . .



Example: finding the return address. . .



Example: finding the return address...

findret:

```
#!/bin/csh -f

set i = 0
while ($i < 64)
   gcc -o ret -m32 -g ret.c >& /dev/null
   echo "OFFSET = $i"
   ./ret $i
   @ i = $i + 1
end
```

Example: finding the return address. . .

```
> findret
OFFSET = 0
OFFSET = 1
OFFSET = 2
OFFSET = 3
OFFSET = 4
OFFSET = 5
Segmentation fault (core dumped)
....
OFFSET = 12
CALLED payload!
```

Example II: Copying

We could also overwrite the return address by copying from another buffer (buf2.c):

```
typedef void (*fun)();
void printit() {printf("CALLED payload!\n");}
void pl(){printit();}
int count;
fun src[32] = {&pl,&pl,&pl,&pl,&pl,&pl,...};

int foo(){
  long* buf[2];
  for(i=0; i<count; i++)
     buf[i] = (long*)src[i];
}

int main(int argc, char** argv){
  count = strtoul(argv[1],NULL,10);
  foo();
}
```

Example II: Copying...

```
> ./buf2 1
> ./buf2 2
> ./buf2 3
Segmentation fault (core dumped)
> ./buf2 4
CALLED payload!
Segmentation fault (core dumped)
```

C library routines. . .

- The C standard library has routines that copy data without checking the length:
  - `strcpy(char *dest, char *src)`: copy data from `src` into dest until a null character is found.
- Idea: Overflow a buffer by feeding too large input into `strcpy`.

More Dangerous C Library Routines

- `memcpy(void *dest, void *src, int n)`: copy `n` bytes from `src` to `dest`.
- `strcat(char *dest, char *src)`: concatenate `src` onto the end of `dest` (starting at the null character).
- `sprintf(char *buffer, char *format, ...)`: print formatted output into a buffer.
- `char* gets(char *str)`: read until end-of-line/file.

Example III: memcpy

We could just copy from another buffer instead (buf4.c):

```
typedef void (*fun)();
void printit() {printf("CALLED payload!\n");}
void pl(){printit();}
int count;
fun src[32] = {&pl,&pl,&pl,&pl,&pl,...};

int foo(){
    long* buf[2];
    __builtin_memcpy(buf,src,count*sizeof(fun));
}

int main(int argc, char** argv){
    count = strtoul(argv[1],NULL,10);
    foo();
}
```

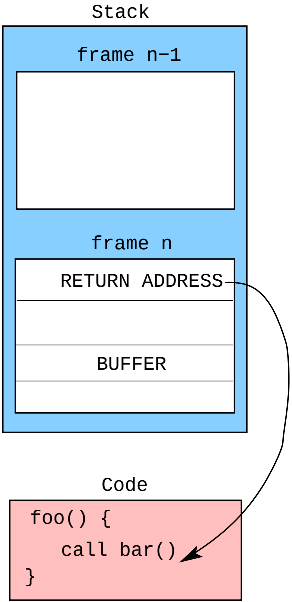
Example III: memcpy

```
> ./buf4 1
> ./buf4 2
> ./buf4 3
> ./buf4 4
> ./buf4 5
Segmentation fault (core dumped)
> ./buf4 6
CALLED payload!
```

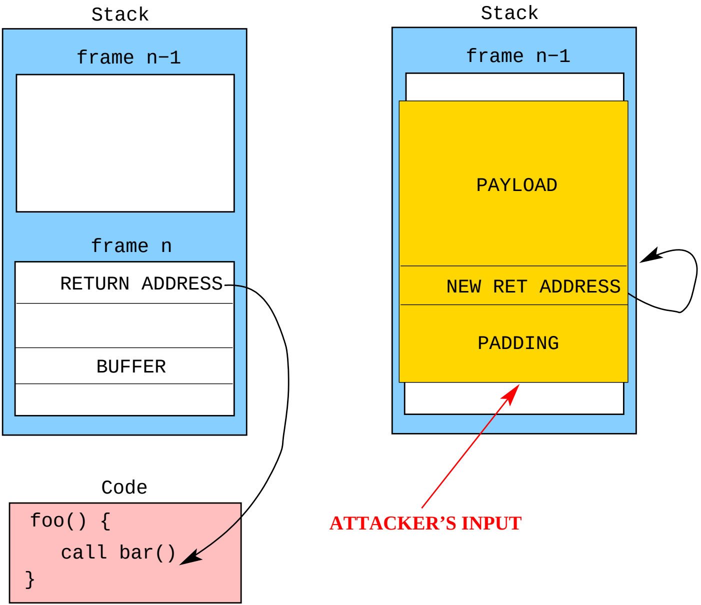
Trivial Stack Smashing Attack

- A stack smashing attack exploits a buffer vulnerability.
  1. inject malicious code (the payload) onto the stack;
  2. overwrite the return address of the current routine;
  3. when a `ret` is executed: jump to `payload`!

Stack Smashing Attack



Stack Smashing Attack



Stack Smashing Attack — Problems

- Essentially, we want to

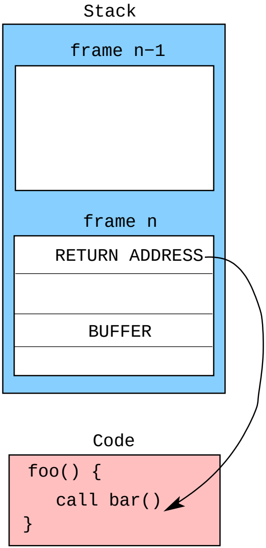
```
stack[cur_frame].ret_address = &(payload)
```

- Problems:
  1. How do I find where `ret_address` is?
  2. How can I find the address of `payload`?
- The payload is also called the shellcode because it's often code to start a shell.

Finding the shellcode: NOP Sledding

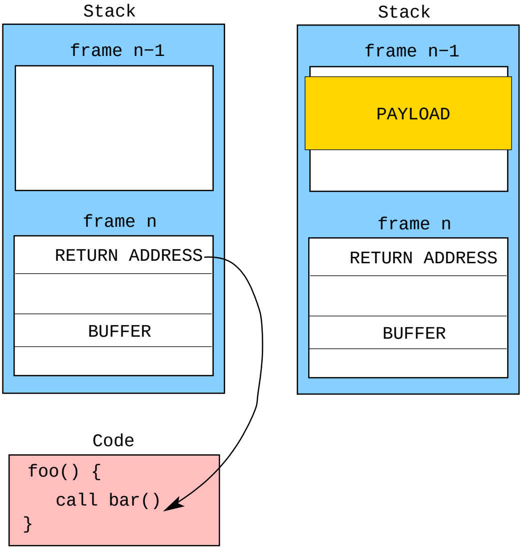
- Attack:
  1. Increase the size of the payload by adding lots of `NOP`s.
  2. Guess an approximate address within the `NOP`-sled.
  3. Jump to this approximate address, sledding into the actual payload.
- This allows us to be less accurate in determining the address of the shellcode.

NOP Sled



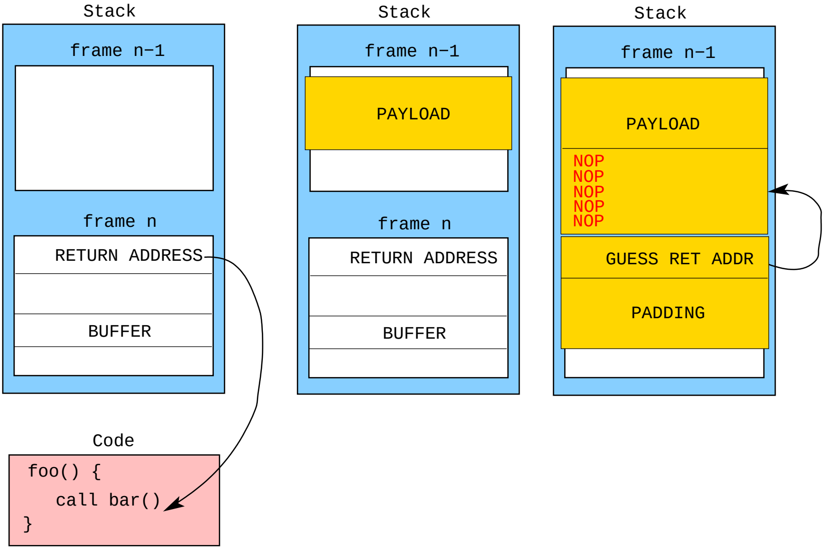
Stack
frame n−1
frame n
RETURN ADDRESS
BUFFER
Code
foo() {
call bar()
}

NOP Sled



Stack
frame n−1
frame n
RETURN ADDRESS
BUFFER
Code
foo() {
call bar()
}
Stack
frame n−1
PAYLOAD
frame n
RETURN ADDRESS
BUFFER

NOP Sled


Stack
frame n−1
frame n
RETURN ADDRESS
BUFFER
Code
foo() {
call bar()
}
Stack
frame n−1
PAYLOAD
frame n
RETURN ADDRESS
BUFFER
Stack
frame n−1
PAYLOAD
NOP
NOP
NOP
NOP
NOP
GUESS RET ADDR
PADDING

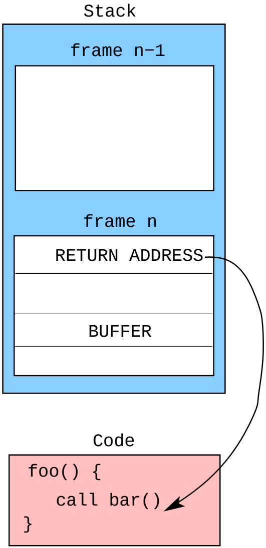
Finding the shellcode: Trampolining

- Attack:
  1. Find a piece of library code, always loaded at the same address, that has a jump-indirect-through-register instruction, such as

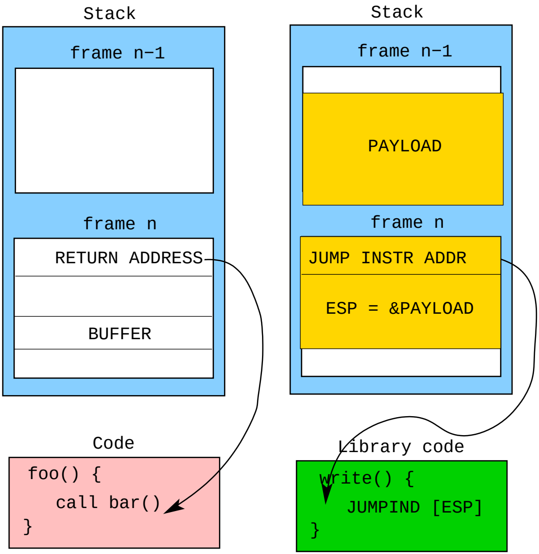
     ```
     JUMPIND [ESP]
     ```

  2. Somehow, make ESP point to the payload, for example by putting the payload in the right location.
  3. Overwrite the return address with the address of the jump instruction.
- More precise than NOP sledding if libraries reside in predictable locations.

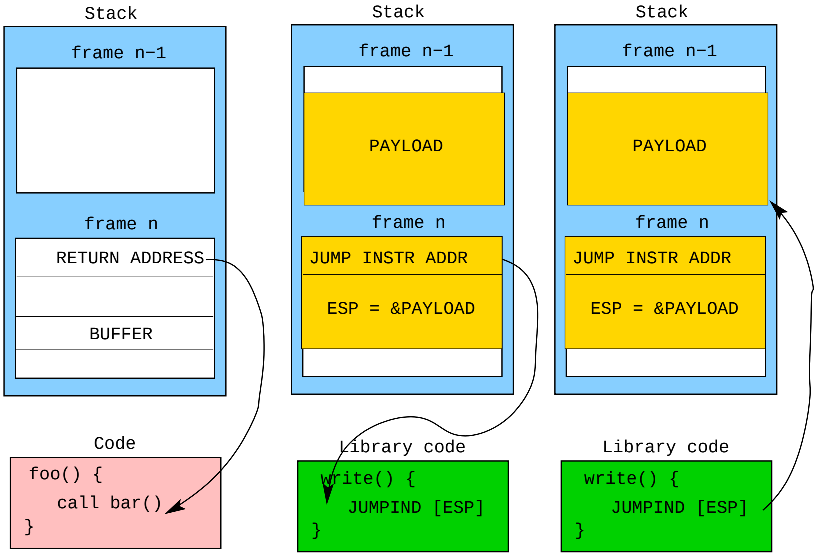
Trampolining



Trampolining



Trampolining



Finding the shellcode: Return-to-libc

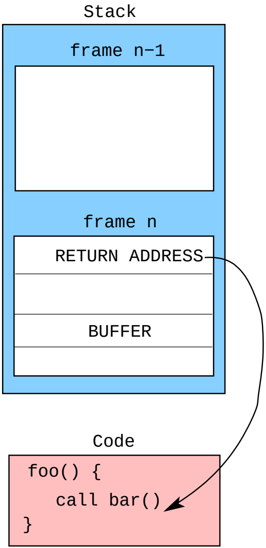
- Attack:
  1. Find the address of a library function such as `system()` or `execv()`.
  2. Overwrite the return address with the address of this library function.
  3. Set the arguments to the library function.
- No code is executed on the stack!
- Attack still works when the stack is marked non-executable.

Finding the shellcode: Return-to-libc. . .

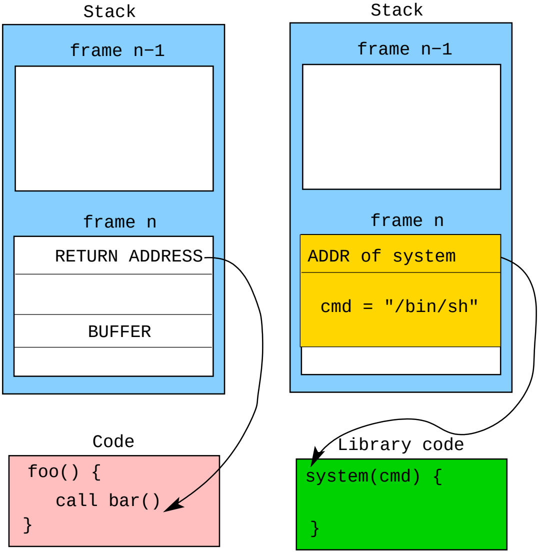
```
int execv(const char *path, char *const argv[]);
int system(const char *command);
```

- execv replaces the current process image with a new process image. The first argument points to the file name of the file being executed.
- The system() function hands the argument command to the command interpreter sh. The calling process waits for the shell to finish executing the command.

Return-to-libc. . .



Return-to-libc. . .



Outline

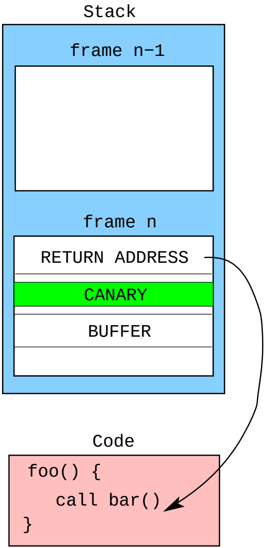
Preventing Buffer Overflows

- Educate the programmer: Use `strncpy`, not `strcpy`.
- Choice of language: Use Java, not C++.
- Detect, at the OS level, when a buffer overflow occurs.
- Prevent the return address from being overwritten.

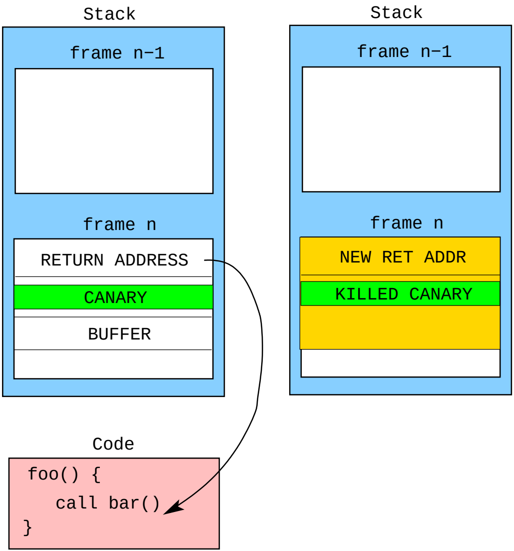
Preventing Buffer Overflows: Canaries

- Defense:
  1. Put a random value (the canary) next to the return address.
  2. Regularly check that the canary has the right value.
- Turn off with
  `gcc -fno-stack-protector -o bug bug.c`

Canaries


Stack
frame n−1
frame n
RETURN ADDRESS
CANARY
BUFFER
Code
foo() {
call bar()
}

Canaries


Stack
frame n−1
frame n
RETURN ADDRESS
CANARY
BUFFER
Stack
frame n−1
frame n
NEW RET ADDR
KILLED CANARY
Code
foo() {
call bar()
}

Preventing Buffer Overflows: PointGuard

- Defense:
  1. XOR all pointers (before and after use) with a random value.

```
x = &p;
y = y->next;
```

⇓

```
x = 0xFEEDFACE^(&p);
y = 0xFEEDFACE^((0xFEEDFACE^y)->next);
```

- The attacker cannot reliably overwrite the return address.

Preventing Buffer Overflows: Non-executable stack

- Defense:
  1. Set the segment containing the stack to non-executable.
  2. Turn off with `gcc -z execstack -o bug bug.c`
- Doesn't help against return-to-libc.
- Some programs legitimately generate code on the stack and jump to it.

Preventing Buffer Overflows: ASLR

- Address space layout randomization: Place memory segments in random locations in memory.
- Helps because:
  - Return-to-libc attacks are harder because it's harder to find `libc`.
  - Finding the shellcode is harder because it's harder to find the stack.
- If there isn't enough entropy, brute-force-attacks can defeat ASLR.
- Turn off with `echo 0 > /proc/sys/kernel/randomize_va_space`.

Exercise: Goodrich & Tamassia C-3.8

```
int main(int argc, char *argv[]) {
   char continue = 0;
   char password[8];
   strcpy(password, argv[1]);
   if (strcmp(password, "CS166")==0)
      continue = 1;
   if (continue)
     *login();
}
```

1. Is this code vulnerable to a buffer-overflow attack with reference to the variables `password[]` and `continue`?

Exercise: Goodrich & Tamassia C-3.8

```
int main(int argc, char *argv[]) {
   char password[8];
   strcpy(password, argv[1]);
   if (strcmp(password, "CS166")==0)
     *login();
}
```

2. We remove the variable `continue` and simply use the comparison for login. Does this fix the vulnerability?

Exercise: Goodrich & Tamassia C-3.8

```
void login() {
    ...; return;
}
int main(int argc, char *argv[]) {
    char password[8];
    strcpy(password, argv[1]);
    if (strcmp(password, "CS166")==0)
      login();
}
```

3. What is the existing vulnerability when `login()` is not a pointer to the function code but terminates with a `return()` command?

Outline

1. Introduction
2. Buffer overflow
   - Basic Idea
   - Automatically finding the return address
   - Dangerous library functions
   - Stack smashing attack
3. Defenses Against Buffer Overflow Attacks
4. Heap-Based Buffer Overflows
5. Format String Attacks
6. Arithmetic Overflow
7. Summary

Heap-Based Buffer Overflows

- A buffer contained in a heap object can also be overflowed.
- This causes data to be overwritten.
- An attacker can craft an overflow such that a function pointer gets overwritten with the address of the shellcode.

Malloc

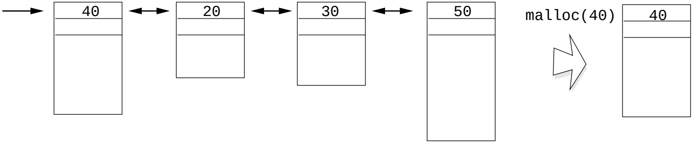
- Memory is allocated from the heap via

  `malloc(int size)`

  where `size` is the number of bytes needed. `malloc` returns the address of (a pointer to) a region of free memory of at least size bytes.
- `malloc` returns 0 (`NULL`) if there isn't a big enough free region to satisfy the request.

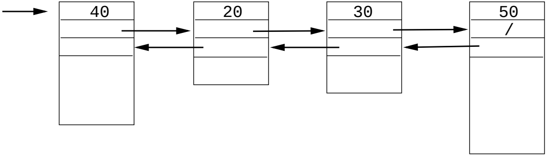
Malloc...

- `malloc` searches the free list for a free region that's big enough, removes it from the free list, and returns its address.



Malloc...

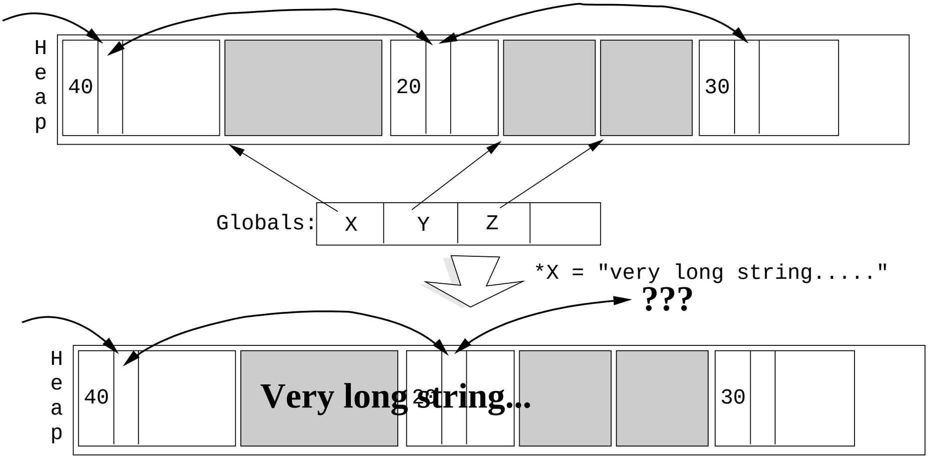
- A doubly-linked-list is often used to make insertion and deletion easier.



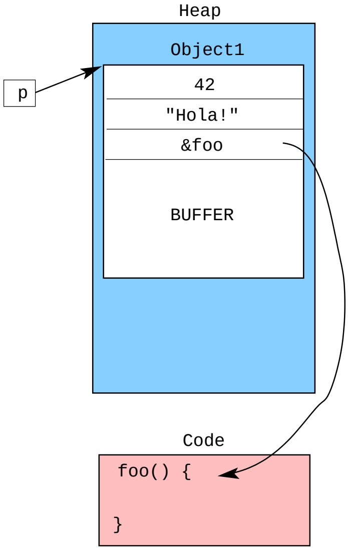
Malloc...

- What happens if the program asks for 50 bytes, but then writes 60 bytes to the region?
- The last 10 bytes overwrite the first 10 bytes of the next region.
- This will corrupt the free list if the next region is free (and probably crash the program if it is not).

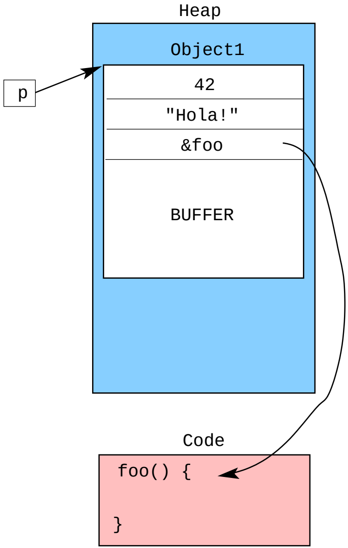
Malloc...


H
e
a
p
40
20
30
Globals:
X
Y
Z
*X = "very long string....."
???
H
e
a
p
40
Very long string...
20
30

Heap-Based Buffer Overflow



Heap-Based Buffer Overflow

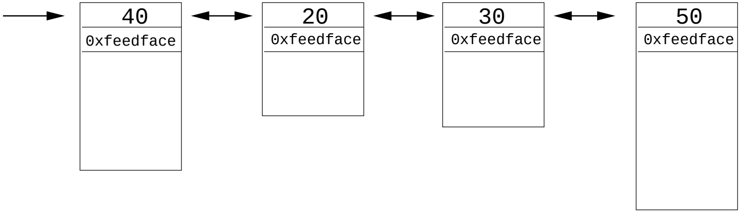


Defenses

- Safe programming practices.
- Use a safe language (Java, not C++).
- Randomize the location of the heap.
- Make the heap non-executable.
- Store heap meta-data (the free-list pointers, object size, etc.) separately from the objects.
- Detect when heap meta-data has been overwritten.

Defenses: Canaries

- Add a magic number in the free list node headers. This is a distinctive value that `malloc` checks when traversing the free list, and complains if the value changes (which indicates the list is corrupted). For example, put a field in the header whose value is always `0xfeedface`.



Outline

Format String Attacks

```
int printf(const char * restrict format, ...);
```

- `restrict`: no pointer aliasing allowed.
- `const char *`: a mutable pointer to an immutable string.
- `...`: variable number of arguments.
- `format` is expected to be a constant string, and the compiler should check for it.
- However, sometimes it's not…

Format String Parameters

`%d`	decimal
`%u`	unsigned decimal
`%x`	hexadecimal
`%s`	string
`%n`	number of characters written

Printf with Non-Constant Format String

`formattest.c`:

```c
int main (int argc, char **argv){
  printf(argv[1]);
}
```

- Compile:

```
gcc -m32 -Wno-format formattest.c -o
    formattest
```

- `-m32`: we're compiling for a 32-bit machine.
- `-Wno-format` don't check that the first argument to `printf` is constant.

Reading from the Stack

formattest.c:

```c
int main (int argc, char **argv){
  printf(argv[1]);
}
```

- Run:

```
> formattest "Bob"
Bob
```

- Run (printing stack data):

```
> formattest "Bob %x %x %x"
Bob 65117a90 65117aa8 65117b00
```

Crashing the Program

w1.c:

```
#include<stdio.h>
void int main () {
    printf ("%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s
        %s%s%s%s%s%s");
}
```

- Keep in mind that %s takes a pointer as argument!

Extracting Information

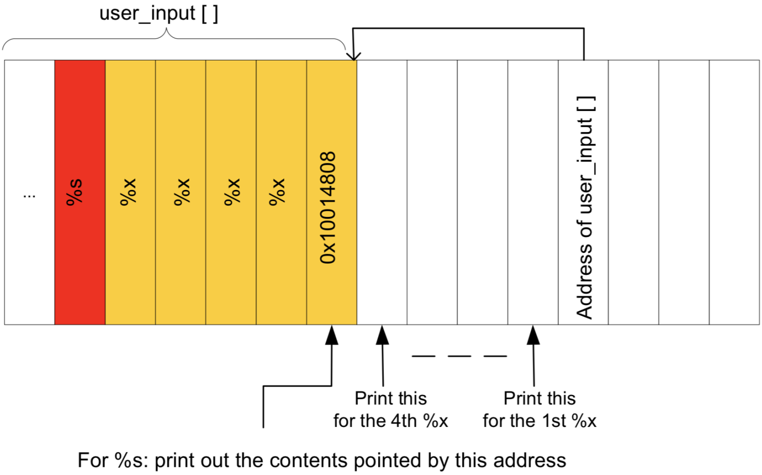
- In the next slide, the program has a secret string we want to find.
- Assume for a moment we know the address. (Here, to show the idea, we just print it out).

Extracting Information...

`w4.c`:

```
#include<stdio.h>
#include<string.h>
// Compile: gcc -m32 -o w4 -Wno-format w4.c

char* secret="ATTACK AT DAWN";

int main (int argc, char** argv) {
    char buff[100];
    strncpy(buff, "\xc0\x85\x04\x08%x%x%x%x%x\n%s
        \n", 100);

    // printf("%p\n", secret);
    // printf("%c\n", *((char*) 0x080485c0));

    printf(buff);
}
```

Extracting Information. . .



Source: http://www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf

Extracting Information...

`w5.c`:

```
// Compile:
//    gcc w5_in.c -o w5_in
//    gcc -m32 -o w5 -Wno-format w5.c
int main (int argc, char** argv) {
  char buff[100];
  strncpy(buff, argv[1], 100);
  printf(buff);
}
```

Extracting Information. . .

w5_in.c:

```
#include<stdio.h>
int main (int argc, char** argv) {
    printf("\xc0\x85\x04\x08%s\n","%x%x%x%x%x%s")
        ;
}
```

- We need some way to enter a binary address to our program.
- Easiest is to write one that prints it out!
- Now, we can call w5 like this:

```
    ./w5 `w5_in`
```

Printf's "%n" Modifier

formatn.c:

```c
int main() {
  int size;
  printf("Bob loves %n Alice\n", &size);
  printf("size = %d\n", size);
  return 0;
}
```

- The "%n" modifier to printf stores the number of characters printed so far.
- Run:

```
> formatn
Bob loves  Alice
size = 10
```

Writing to the Stack!

`formattest.c`:

```
int main (int argc, char **argv){
  printf(argv[1]);
  return 0;
}
```

- Run `formattest` again:

```
> formattest "XXXXXXXXXXXXXXX %n%n%n%n"
Segmentation fault
```

- The program crashes because the `"%n"` modifier makes `printf` write into a "random" location in memory.

Outline

Arithmetic Overflow

- Integers typically have fixed size.
- Programmers typically don't check for overflow conditions.
- Java doesn't throw exceptions for integer overflow/underflow!

Example

Code to grant access to the first 5 users who try to connect:

```
int main() {
    unsigned int connections = 0;
    // network code
    connections++;
    if (connections<5)
        grant_access();
    else
        deny_access();
}
```

Example — Attack

```
connections++;
if (connections<5)
   grant_access();
else
   deny_access();
```

Attack:

1. make a huge number of connections;
2. wait for the counter to overflow;
3. gain access!

Example — Safe Programming Practices

- This code avoids possible overflows:

```c
int main() {
    unsigned int connections = 0;
    // network code
    if (connections<5)
        connections++;
    if (connections<5)
        grant_access();
    else
        deny_access();
}
```

Patriot Missile Failure During the (First) Gulf War I

Source: http://cs.furman.edu/digitaldomain/themes/risks/risks_numeric.htm
Video: https://www.youtube.com/watch?t=260&v=tWc4gGMQ3hQ
Time is kept continuously by the system's internal clock in tenths of seconds but is expressed as an integer or whole number. The longer the system has been running, the larger the number representing time.
To predict where the Scud will next appear, both time and velocity must be expressed as real numbers. Because of the way the Patriot computer performs its calculations and the fact that its registers are only 24

Patriot Missile Failure During the (First) Gulf War II

bits long, the conversion of time from an integer to a real number cannot be any more precise than 24 bits. The error occurred when translating between integer and decimal number (floating point) formats. The error could become quite significant when the system was run for long periods without resetting. For example, after 100 hours of continuous operation, the error in the estimate of the position of the target is almost 1/3 of a mile.

Outline

Readings and References

- Section 3.4 in *Introduction to Computer Security*, by Goodrich and Tamassia.

Acknowledgments

Material and exercises have also been collected from these sources:

1. Michael Stepp, *Lecture on Buffer Overflow Attacks*, *620—Fall 2003*.
2. Wenliang (Kevin) Du, *Format String Vulnerability*, http://www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf
3. http://www.cs.virginia.edu/~ww6r/CS4630/lectures/Format_String_Attack.pdf