CSc 466/566

Computer Security

20 : Operating Systems — Application Security

Version: 2014/11/20 13:07:28

Department of Computer Science
University of Arizona

collberg@gmail.com

Christian Collberg

Outline

1. Introduction
2. Arithmetic Overflow
3. Buffer Overflow
   - Stacks and Buffers
   - Basic Idea
   - Stack Smashing Attack
   - Preventing Buffer Overflows
4. Heap-Based Buffer Overflows
5. Format String Attacks
6. Race Conditions

Introduction

- Programmers tend to avoid
  - checking for error conditions that rarely happen;
  - checking for boundary conditions to save time;
  - checking user input to make sure it's valid.
- Such programming errors can be exploited in a privilege escalation attack.
- How do we avoid writing vulnerable code? We need defensive programming practices!

Outline

1. Introduction
2. Arithmetic Overflow
3. Buffer Overflow
   - Stacks and Buffers
   - Basic Idea
   - Stack Smashing Attack
   - Preventing Buffer Overflows
4. Heap-Based Buffer Overflows
5. Format String Attacks
6. Race Conditions

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:

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

Outline

1. Introduction
2. Arithmetic Overflow
3. Buffer Overflow
   - Stacks and Buffers
   - Basic Idea
   - Stack Smashing Attack
   - Preventing Buffer Overflows
4. Heap-Based Buffer Overflows
5. Format String Attacks
6. Race Conditions

Introduction

- Buffer overflow attacks explained with beer!

  http://www.youtube.com/watch?v=7LDdd90aq5Y
- What is a buffer overflow attack?
- Why are they possible?
- How do I perform a buffer overflow attack?
- How do I prevent a buffer overflow attack?

Definitions

- buffer: A span of contiguous writable memory.
- stack frame: The space on the stack alotted to a particular procedure.
- buffer overflow: Writing past the declared bounds of a buffer.
- buffer overflow attack:
  - A method of gaining control of a system by executing some program/procedure with more data than it is prepared to handle.
  - The extra data is designed to cause malicious side effects.

Buffer overflow attack — Basic Idea

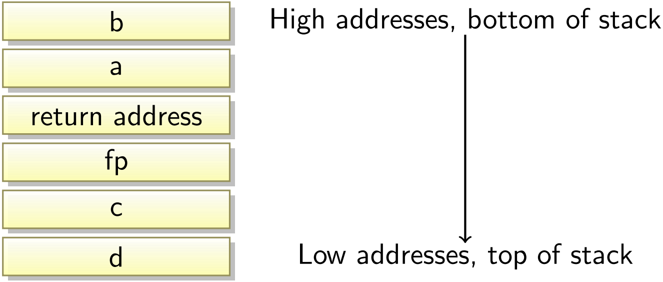
1. Inject some code into the memory of an executing program. The code consists of
   - Malicious payload that we want to execute;
   - Some way to get this code to execute.
2. Overwrite a memory address with the address of the payload.
   - This is often the return address of a function.
3. When the program jumps to this address, it instead jumps to our payload!

Stack layout

- The execution stack of a program (on an x86 machine) grows downward (to lower memory addresses) as procedures are called.
- Information is placed in stack frames.
- Among the things stored on the stack are
  - the local and formal variables,
  - the return address, and
  - the frame pointer of the procedure.
- The positions of these values in memory are shown on the next slide:

Stack layout...

```c
void function(int a, int b){
    int c;
    int d;
}
```



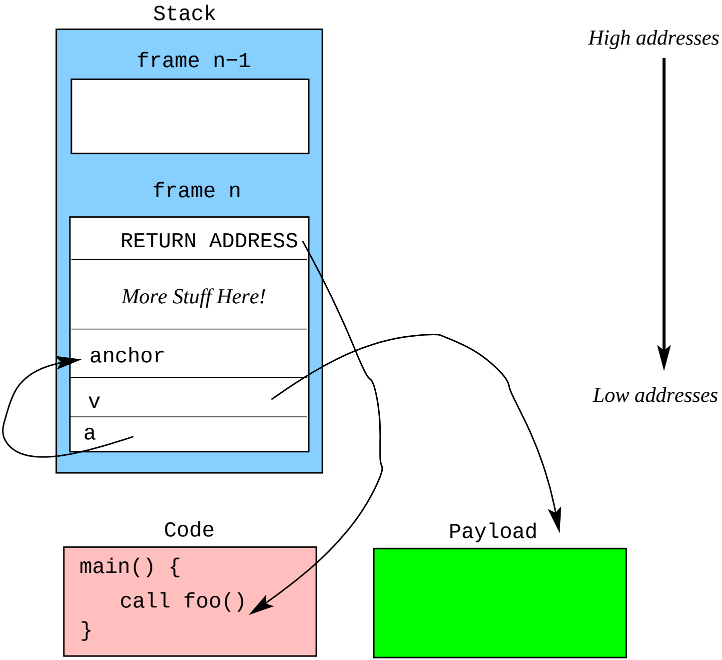
Basic idea

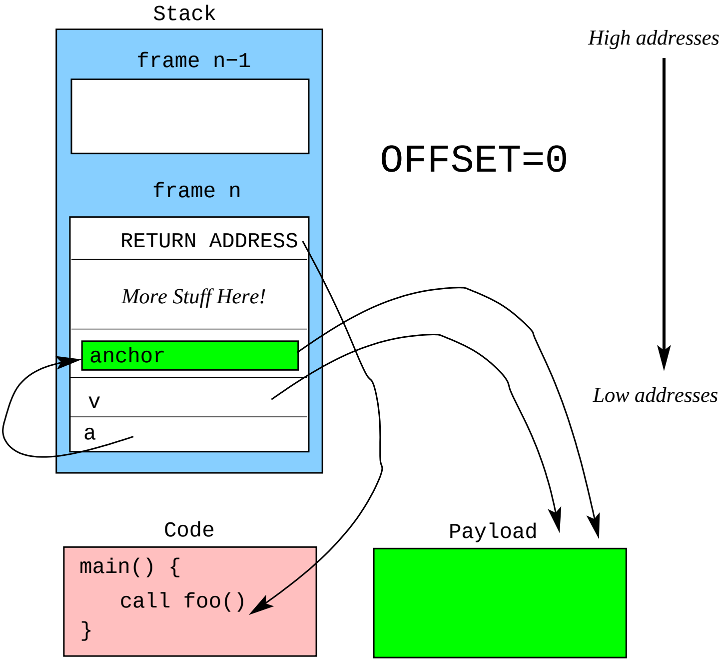
- If we could overwrite the return address of a procedure with a different address, then when the procedure returned it would jump wherever we wanted.
- How do we find the return address?
  1. declare a local variable, `anchor`;
  2. take its address;
  3. add an increasing offset to `anchor`;
  4. overwrite this new address with the address of `payload`;
  5. return;
  6. did we go to `payload`?

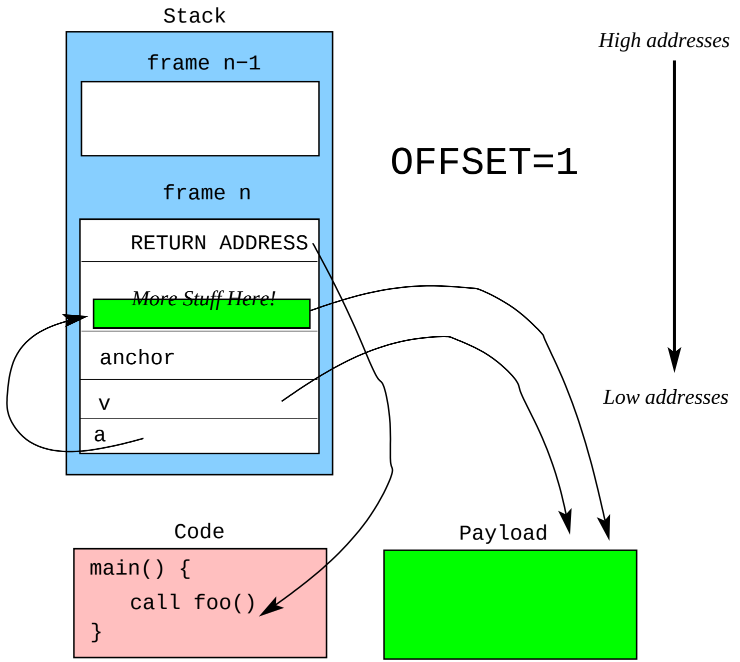
Example: changing the return address

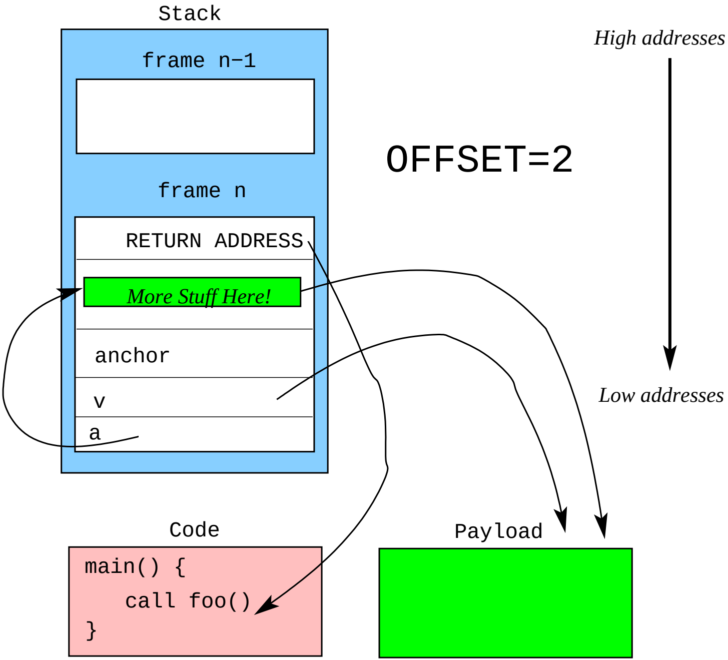
ret.c:

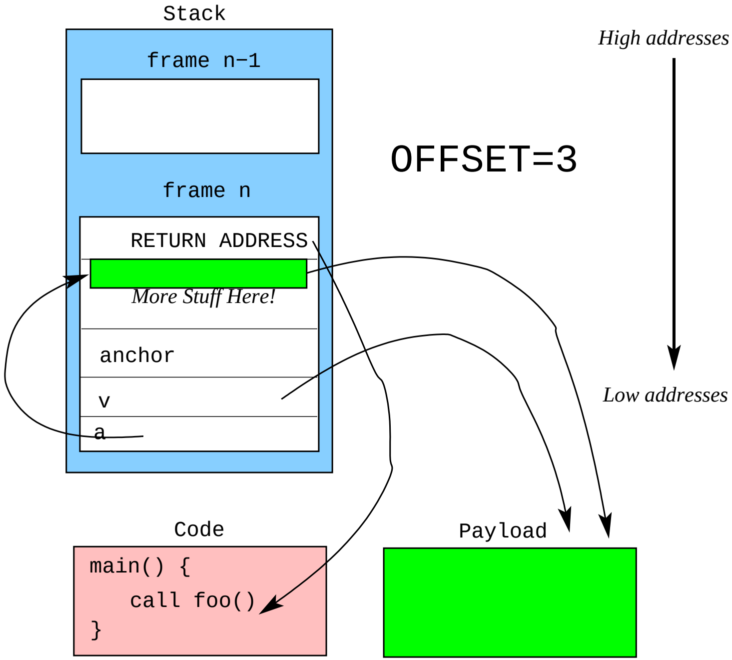
```
void payload(){}
#define OFFSET (HERE)
int foo(){
  volatile long anchor=-1;
  void (*v)()   =&payload;
  volatile long* a = &anchor;
  a = (volatile long*)((long)a + (long)OFFSET);
  *a = (long*)v;
}
int main(){
  foo();
}
```

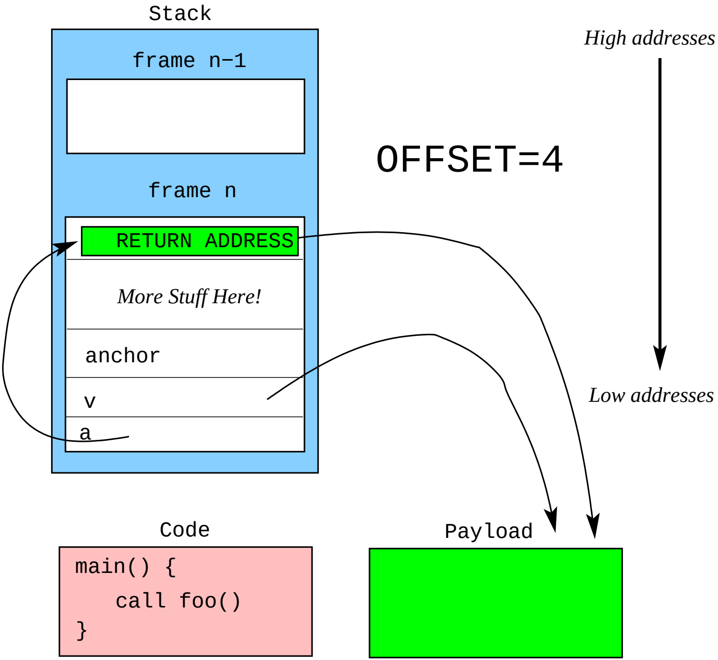

Stack
frame n−1
frame n
RETURN ADDRESS
More Stuff Here!
anchor
v
a
Code
main() {
call foo()
}
Payload
High addresses
Low addresses


Stack
frame n−1
frame n
RETURN ADDRESS
More Stuff Here!
anchor
v
a
OFFSET=0
High addresses
Low addresses
Code
main() {
call foo()
}
Payload


Stack
frame n−1
frame n
RETURN ADDRESS
More Stuff Here!
anchor
v
a
OFFSET=1
High addresses
Low addresses
Code
main() {
call foo()
}
Payload


Stack
frame n−1
frame n
RETURN ADDRESS
More Stuff Here!
anchor
v
a
Code
main() {
call foo()
}
Payload
OFFSET=2
High addresses
Low addresses


Stack
frame n−1
frame n
RETURN ADDRESS
More Stuff Here!
anchor
v
a
OFFSET=3
High addresses
Low addresses
Code
main() {
call foo()
}
Payload



Stack
frame n−1
frame n
RETURN ADDRESS
More Stuff Here!
anchor
v
a
OFFSET=4
High addresses
Low addresses
Code
main() {
call foo()
}
Payload

Example: changing the return address...

findret:

```
#!/bin/csh -f

set i = 0
while ($i < 30)
   echo "OFFSET = $i"
   sed "s/HERE/$i/" ret.c > r.c
   gcc -o ret -g r.c >& /dev/null
   gdb -quiet -x cmd ./ret |& grep payload
   echo ""
   @ i = $i + 1
end
```

Example: changing the return address...

gdb command file, `cmd`:

```
break payload
run
quit
```

Problems

1. Don't know the address of the procedure's return address in the stack frame.
2. Once we find it, we need an address to replace it with, so we must have our evil code somewhere in memory along with the rest of the program.

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*

C library routines

In the C libraries there are many routines designed for copying data from one buffer to another:

- `memcpy(void *dest, void *src, int n)`: copy `n` bytes from `src` to `dest`.
- `strcpy(char *dest, char *src)`: copy data from `src` into dest until a null character is found.
- `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.

Idea: Let's target routines that continue copying until a null character is reached.

Buffer overflow idea

1. Find a procedure that uses one of these routines.

Buffer overflow idea

1. Find a procedure that uses one of these routines.
2. Check that it has a local variable buffer.

Buffer overflow idea

1. Find a procedure that uses one of these routines.
2. Check that it has a local variable buffer.
3. Check that it copies data from *somewhere* into the local buffer.

Buffer overflow idea

1. Find a procedure that uses one of these routines.
2. Check that it has a local variable buffer.
3. Check that it copies data from *somewhere* into the local buffer.
4. Overflow the buffer.

Buffer overflow idea

1. Find a procedure that uses one of these routines.
2. Check that it has a local variable buffer.
3. Check that it copies data from *somewhere* into the local buffer.
4. Overflow the buffer.
5. Write over the return address.

Buffer overflow idea

1. Find a procedure that uses one of these routines.
2. Check that it has a local variable buffer.
3. Check that it copies data from *somewhere* into the local buffer.
4. Overflow the buffer.
5. Write over the return address.
6. When the procedure returns, jump where we want.

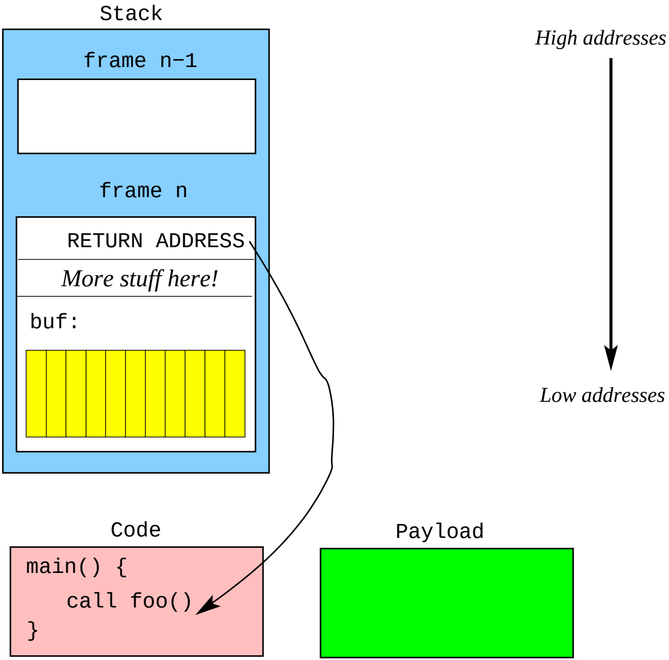
Buffer overflow example I

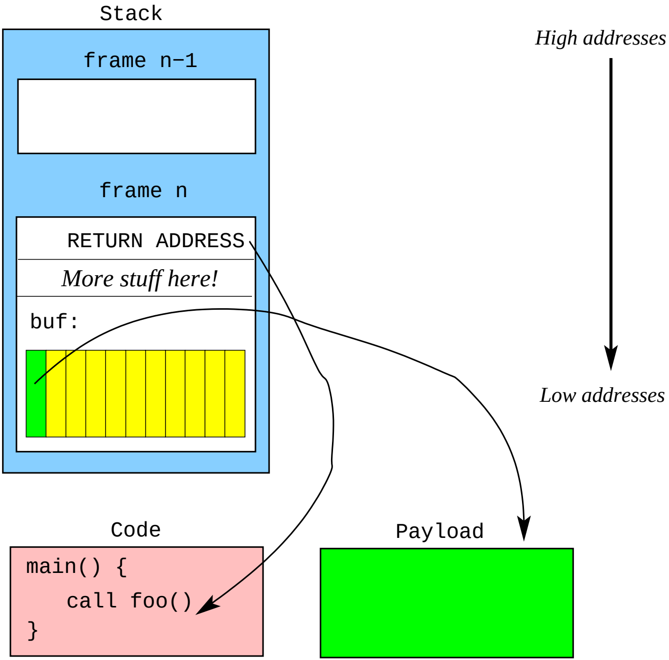
`buf.c`:

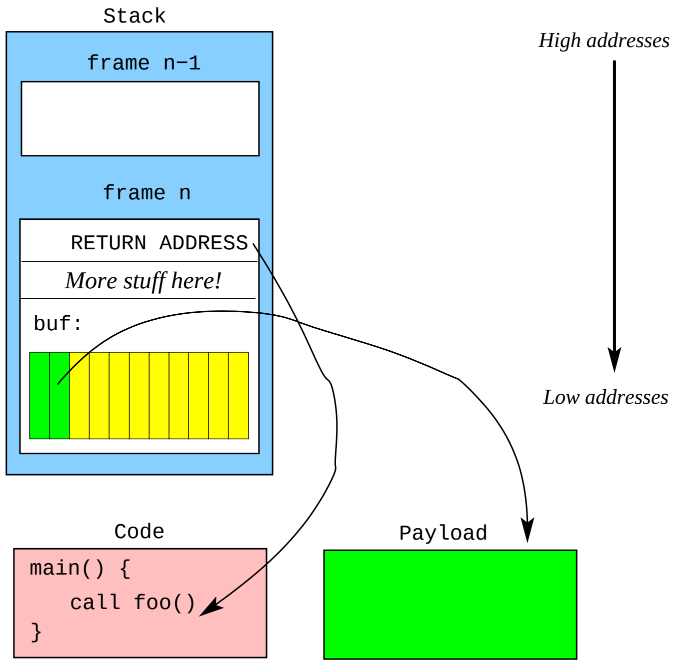
```
void payload(){}
int i;
int foo(){
  long* buf[10];
  void (*v)()  =&payload;
  for(i=0; i<30; i++) buf[i] = (long*)v;
}
int main(){foo();}
```

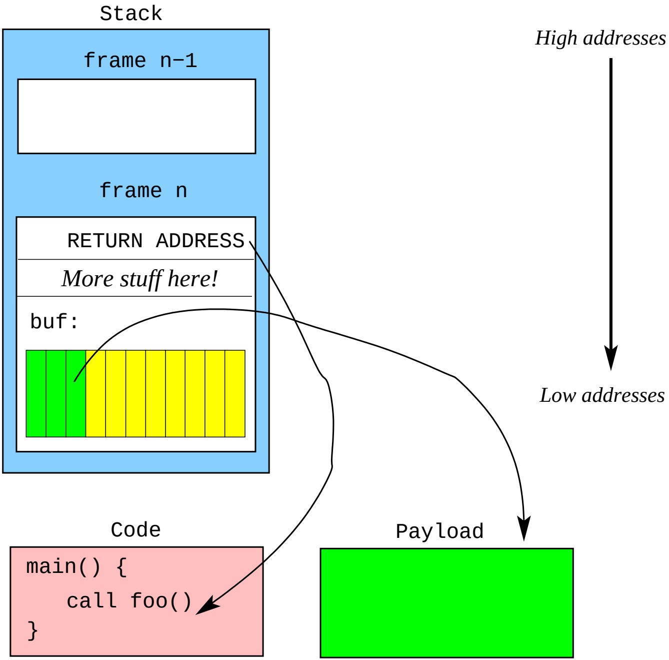
To execute:

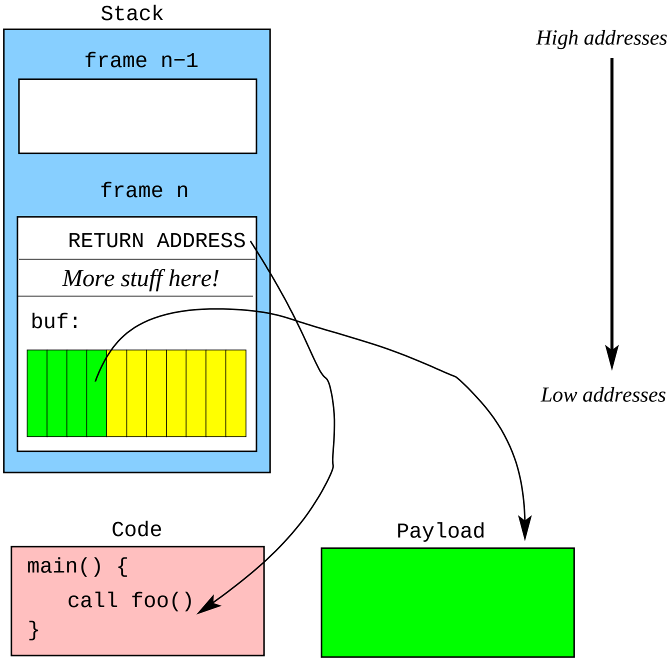
```
> gcc -g -o buf buf.c
> gdb buf
gdb> break payload
gdb> run
```

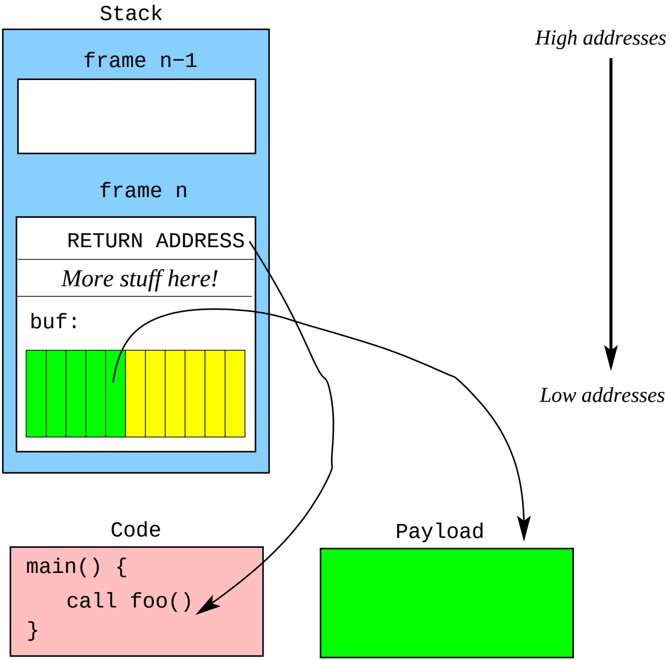

Stack
frame n−1
frame n
RETURN ADDRESS
More stuff here!
buf:
High addresses
Low addresses
Code
main() {
call foo()
}
Payload


Stack
frame n−1
frame n
RETURN ADDRESS
More stuff here!
buf:
High addresses
Low addresses
Code
main() {
call foo()
}
Payload

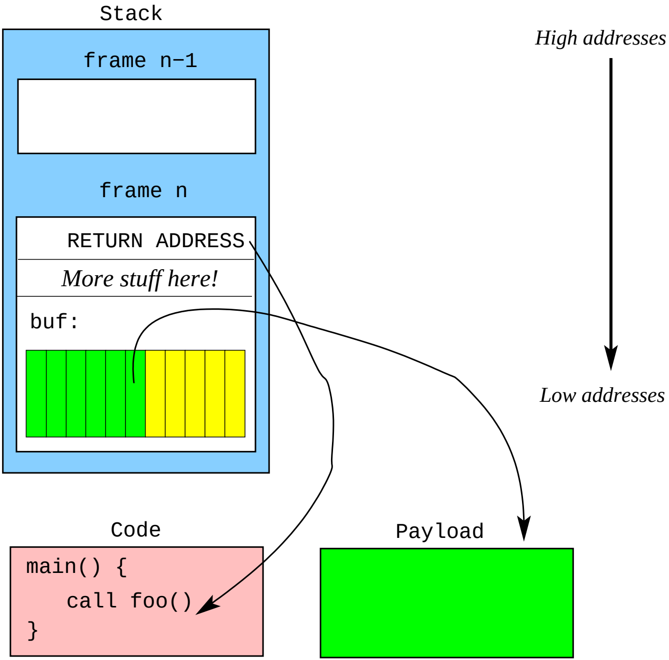

Stack
frame n−1
frame n
RETURN ADDRESS
More stuff here!
buf:
High addresses
Low addresses
Code
main() {
call foo()
}
Payload


Stack
frame n−1
frame n
RETURN ADDRESS
More stuff here!
buf:
High addresses
Low addresses
Code
main() {
call foo()
}
Payload

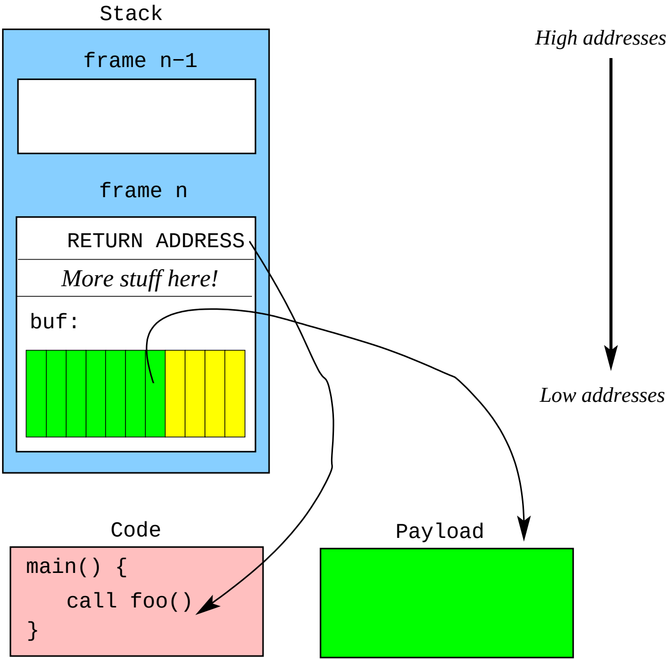

Stack
frame n−1
frame n
RETURN ADDRESS
More stuff here!
buf:
High addresses
Low addresses
Code
main() {
call foo()
}
Payload

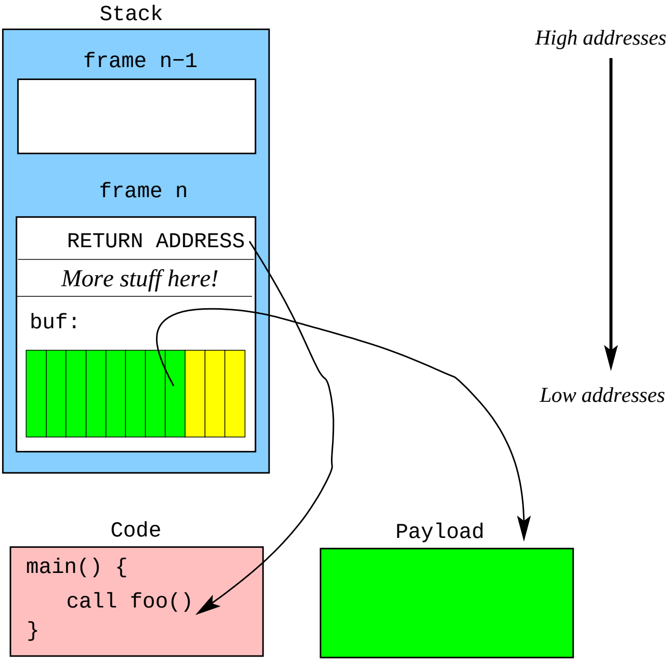


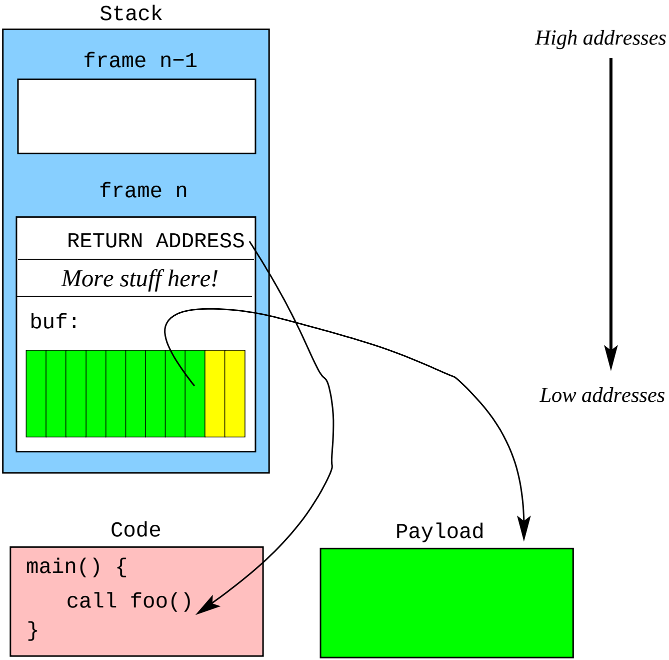
Stack
frame n−1
frame n
RETURN ADDRESS
More stuff here!
buf:
High addresses
Low addresses
Code
main() {
call foo()
}
Payload

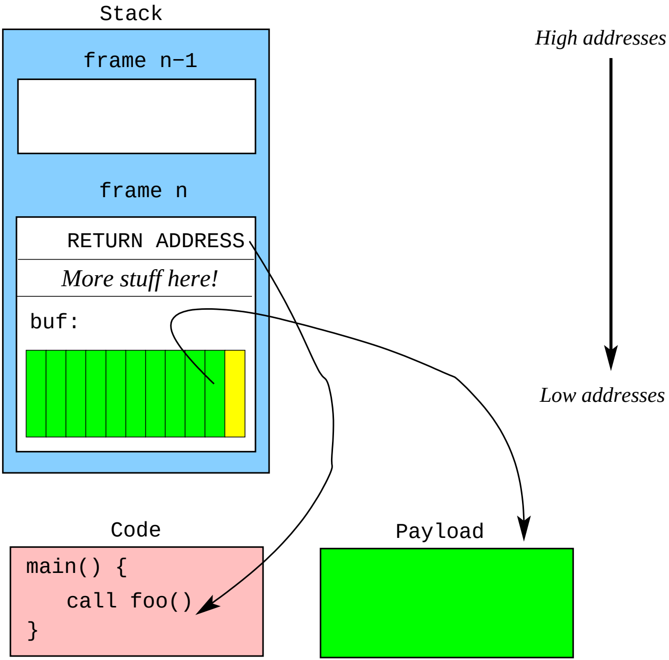


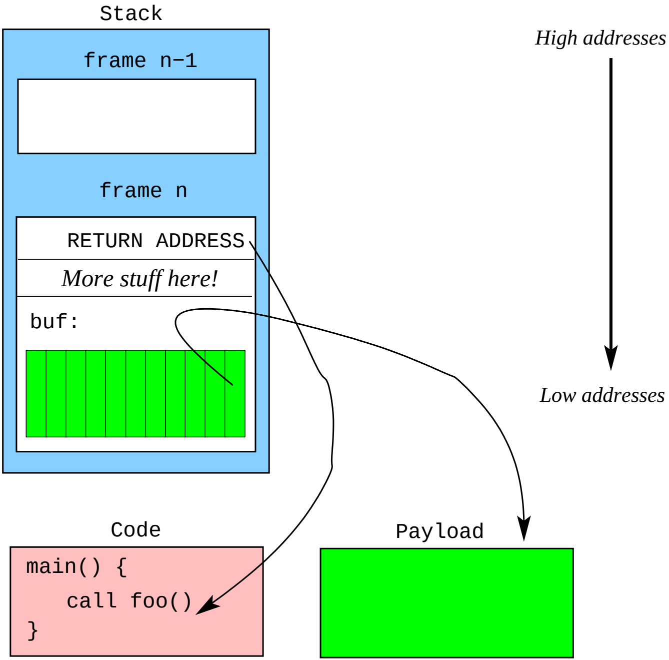
Stack
frame n−1
frame n
RETURN ADDRESS
More stuff here!
buf:
High addresses
Low addresses
Code
main() {
call foo()
}
Payload

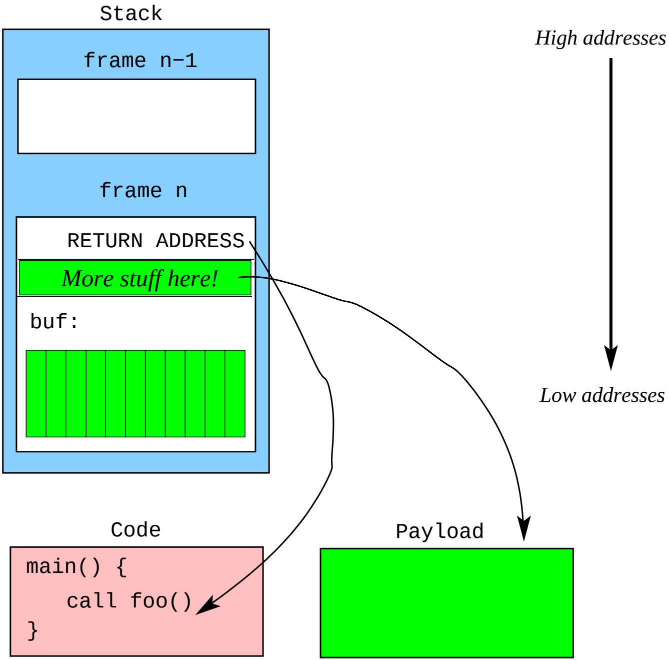

Stack
frame n−1
frame n
RETURN ADDRESS
More stuff here!
buf:
High addresses
Low addresses
Code
main() {
call foo()
}
Payload

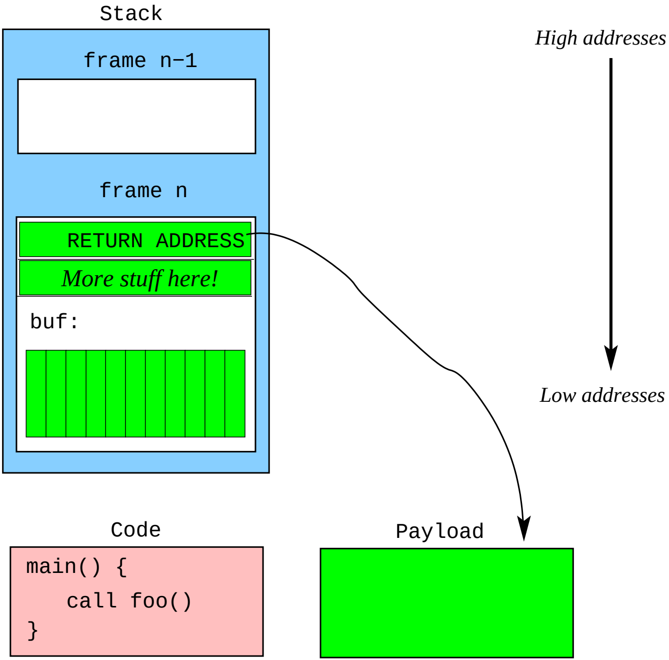

Stack
frame n−1
frame n
RETURN ADDRESS
More stuff here!
buf:
High addresses
Low addresses
Code
main() {
call foo()
}
Payload


Stack
frame n−1
frame n
RETURN ADDRESS
More stuff here!
buf:
High addresses
Low addresses
Code
main() {
call foo()
}
Payload


Stack
frame n−1
frame n
RETURN ADDRESS
More stuff here!
buf:
High addresses
Low addresses
Code
main() {
call foo()
}
Payload


Stack
frame n−1
frame n
RETURN ADDRESS
More stuff here!
buf:
High addresses
Low addresses
Code
main() {
call foo()
}
Payload


Stack
frame n−1
frame n
RETURN ADDRESS
More stuff here!
buf:
High addresses
Low addresses
Code
main() {
call foo()
}
Payload


Stack
frame n−1
frame n
RETURN ADDRESS
More stuff here!
buf:
High addresses
Low addresses
Code
main() {
call foo()
}
Payload

Buffer overflow example II

We could just copy from another buffer instead (`buf2.c`):

```
void pl(){}
typedef void (*fun)();
fun src[32] = {&pl,&pl,&pl,&pl,&pl,&pl ,...};
int i;
int foo(){
  long* buf[2];
  for(i=0; i<30; i++)
    buf[i] = src[i];
}
int main(){foo();}
```

Buffer overflow example III

We want to use one of the built-in copy functions (`buf3.c`):

```
void pl(){}
typedef void (*fun)();
fun src[32] = {&pl,&pl,&pl,...,0};
int i; char* p;
int foo(){
  long* buf[2];
  __builtin_strcpy(buf,src);
  p = &buf;
  for(i=0;i<(sizeof(fun)*32);i++){(*p)--;p++;}
}
int main(){
  int i; char* p = &src;
  for(i=0;i<(sizeof(fun)*32);i++){(*p)++; p++;};
  foo();
}
```

Buffer overflow idea...

- Hey, what's up with the increment loop???

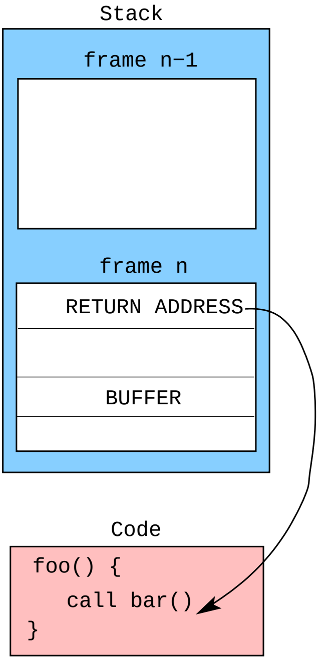
```
char* p = &src;
for(i=0; i<(sizeof(fun)*32); i++) {
    (*p)++;
    p++;
 };
foo();
```

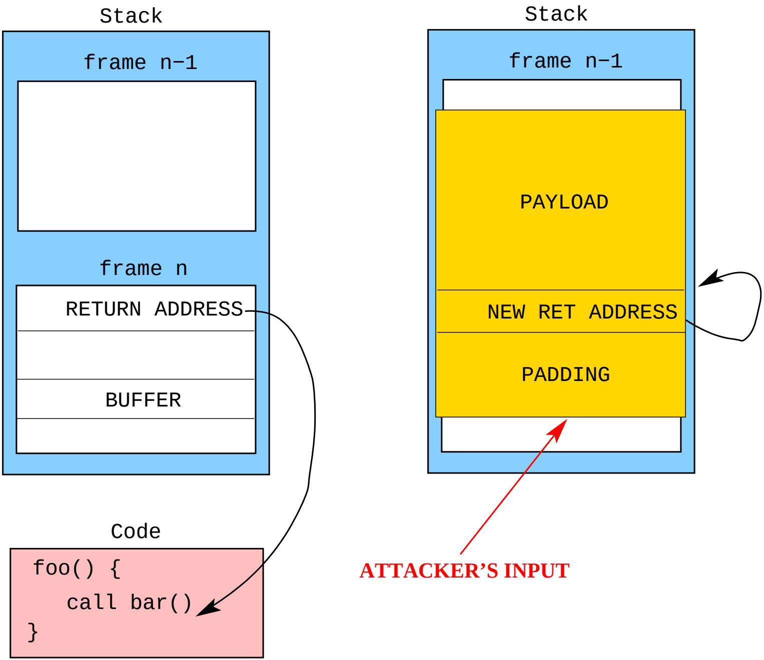
- The problem is the `strcpy` copies until it sees a null character, so, somehow, we need to remove all zero:s from the source "string".
- Also, compile like this:

```
> gcc -fno-stack-protector -g -o buf3 buf3.c
```

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


Stack
frame n−1
frame n
RETURN ADDRESS
BUFFER
Code
foo() {
call bar()
}
Stack
frame n−1
PAYLOAD
NEW RET ADDRESS
PADDING
ATTACKER'S INPUT

Stack Smashing Attack — Problems

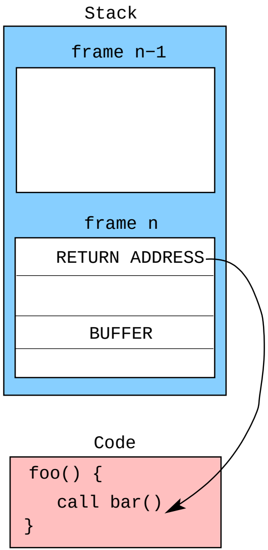
- Essentially, we want to

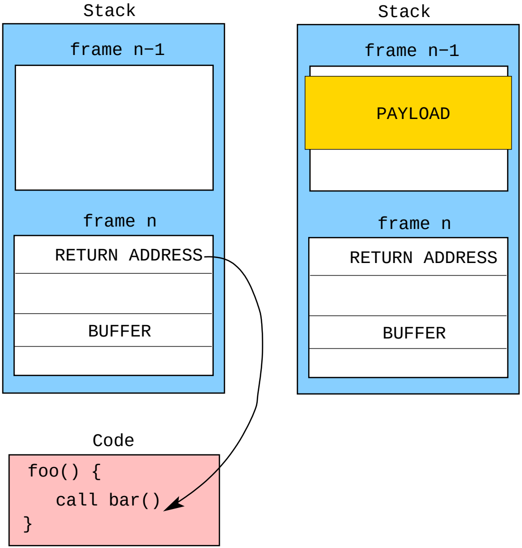
```
stack[cur_frame].ret_address = &(payload)
```

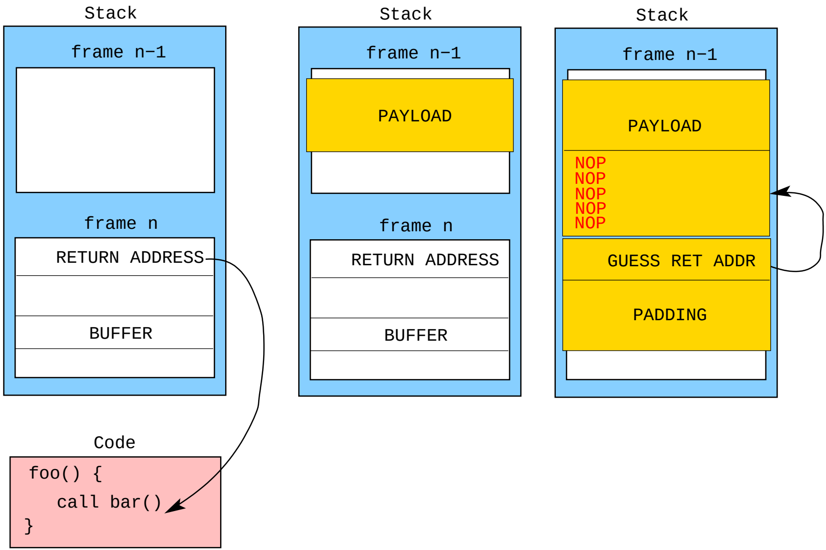
- Problems:
  1. How do I find where `ret_address` is?
  2. How can I find the address of `payload`?
- The payload is also called the shellcode because it's often code to start a shell.

Finding the shellcode: NOP Sledding

- Attack:
  1. Increase the size of the payload by adding lots of `NOPs`.
  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.


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


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

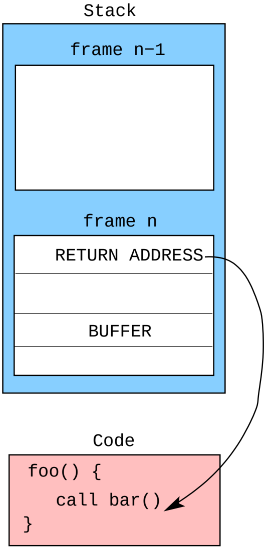

Stack
frame n−1
frame n
RETURN ADDRESS
BUFFER
Code
foo() {
call bar()
}
Stack
frame n−1
PAYLOAD
frame n
RETURN ADDRESS
BUFFER
Stack
frame n−1
PAYLOAD
NOP
NOP
NOP
NOP
NOP
GUESS RET ADDR
PADDING

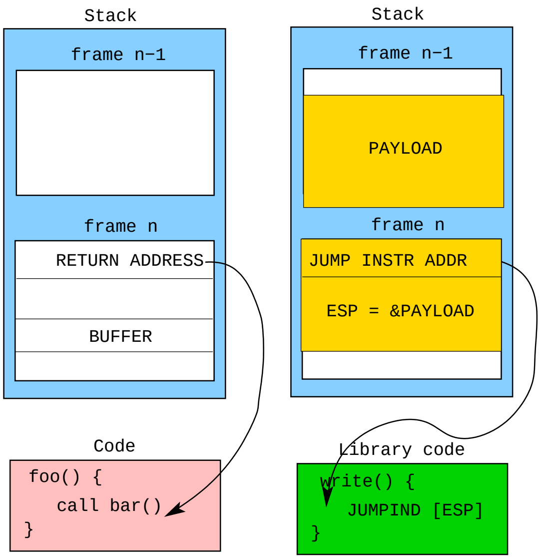
Finding the shellcode: Trampolining

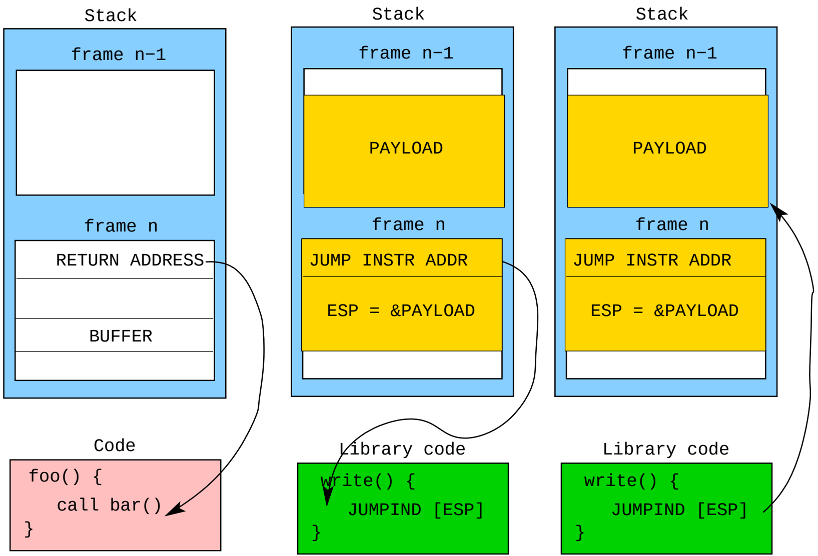
- Attack:
  1. Find a piece of library code, always loaded at the same address, that has a jump-indirect-through-register instruction, such as

```
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.


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


Stack
frame n−1
frame n
RETURN ADDRESS
BUFFER
Code
foo() {
call bar()
}
Stack
frame n−1
PAYLOAD
frame n
JUMP INSTR ADDR
ESP = &PAYLOAD
Library code
write() {
JUMPIND [ESP]
}


Stack
frame n−1
frame n
RETURN ADDRESS
BUFFER
Code
foo() {
call bar()
}
Stack
frame n−1
PAYLOAD
frame n
JUMP INSTR ADDR
ESP = &PAYLOAD
Library code
write() {
JUMPIND [ESP]
}
Stack
frame n−1
PAYLOAD
frame n
JUMP INSTR ADDR
ESP = &PAYLOAD
Library code
write() {
JUMPIND [ESP]
}

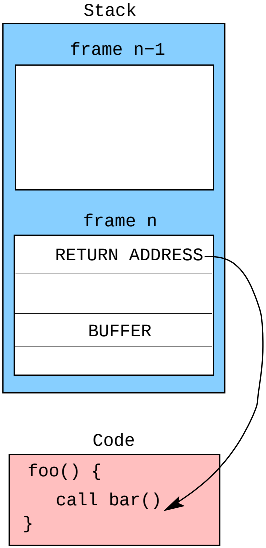
Finding the shellcode: Return-to-libc

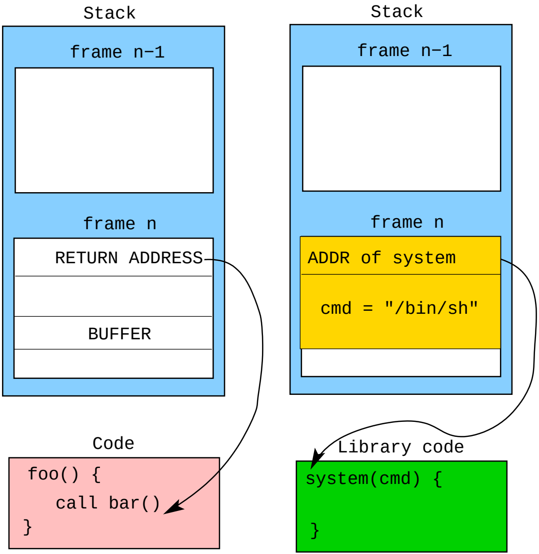
- Attack:
  1. Find the address of a library function such as `system()` or `execv()`.
  2. Overwrite the return address with the address of this library function.
  3. Set the arguments to the library function.
- No code is executed on the stack!
- Attack still works when the stack is marked non-executable.

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.


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

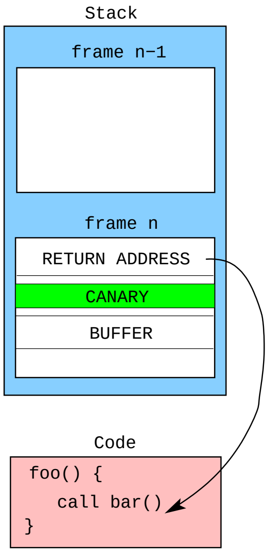

Stack
frame n-1
frame n
RETURN ADDRESS
BUFFER
Code
foo() {
call bar()
}
Stack
frame n-1
frame n
ADDR of system
cmd = "/bin/sh"
Library code
system(cmd) {
}

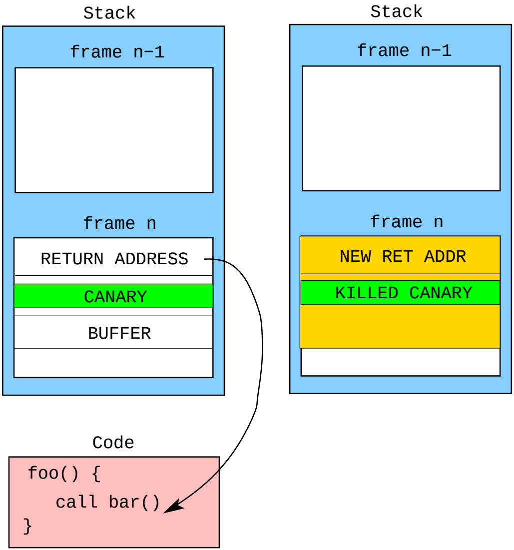
Preventing Buffer Overflows

- 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.


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


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.
- 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.
- Defense:
  1. Place memory segments in random locations in memory.
- 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.

In-Class Exercise: Goodrich & Tamassia C-3.8

```c
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`?

In-Class 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?

In-Class 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. Arithmetic Overflow
3. Buffer Overflow
   - Stacks and Buffers
   - Basic Idea
   - Stack Smashing Attack
   - Preventing Buffer Overflows
4. Heap-Based Buffer Overflows
5. Format String Attacks
6. Race Conditions

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

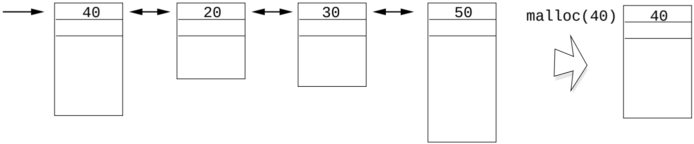
- Memory is allocated from the heap via

  `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.

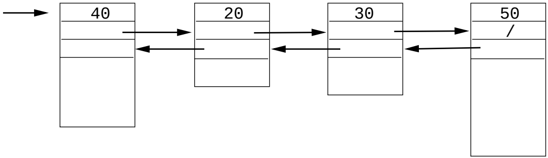
Malloc...

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



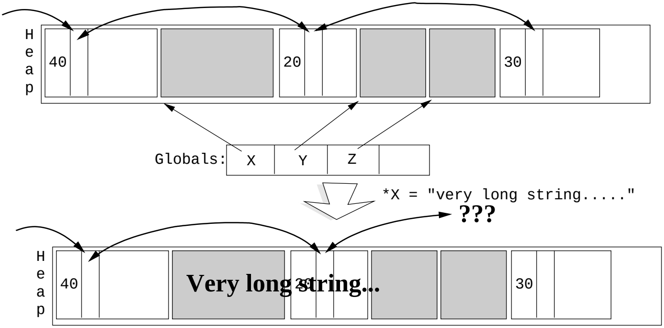
Malloc...

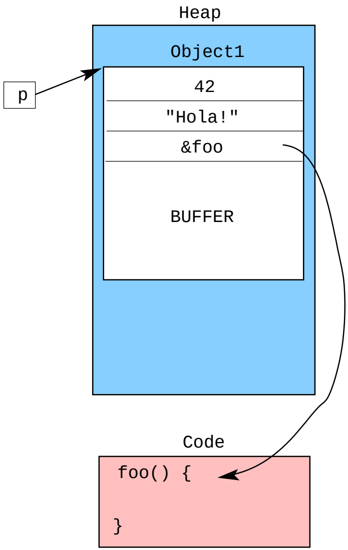
- A doubly-linked-list is often used to make insertion and deletion easier.

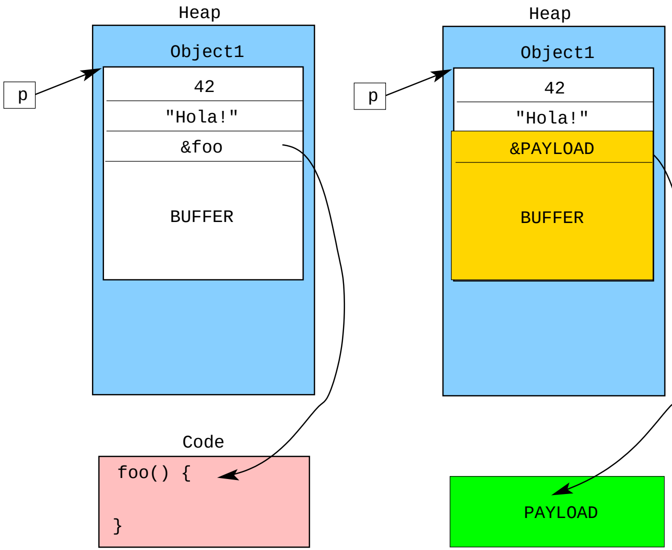


Malloc...

- 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).




Heap
Object1
p
42
"Hola!"
&foo
BUFFER
Code
foo() {
}

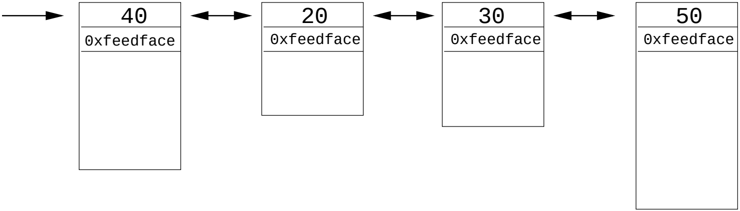

Heap
Object1
p
42
"Hola!"
&foo
BUFFER
Code
foo() {
}
Heap
Object1
p
42
"Hola!"
&PAYLOAD
BUFFER
PAYLOAD

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`.



Free

- The routine

  `free(void *address)`

  is used to release memory when it is no longer needed (e.g. an employee quits or is fired).
- The address parameter is a pointer to the region to be freed, and it must have previously been returned by `malloc`.
- It is common for programmers to free the same memory twice!
- Today, libc is less vulnerable to this problem.

Outline

1. Introduction
2. Arithmetic Overflow
3. Buffer Overflow
   - Stacks and Buffers
   - Basic Idea
   - Stack Smashing Attack
   - Preventing Buffer Overflows
4. Heap-Based Buffer Overflows
5. Format String Attacks
6. Race Conditions

Format String Attacks

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

- `restric`: 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...

Extracting Data from the Stack

formattest.c:

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

- `gcc -Wno-format formattest.c -o formattest`
- Run:

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

- Run (printing stack data):

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

The "%n" Modifier: Modifying Data

formatn.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
```

The "%n" Modifier: Modifying Data

formattest.c:

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

- Run formattest again:

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

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

The "%n" Modifier: Modifying Data

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

Outline

1. Introduction
2. Arithmetic Overflow
3. Buffer Overflow
    - Stacks and Buffers
    - Basic Idea
    - Stack Smashing Attack
    - Preventing Buffer Overflows
4. Heap-Based Buffer Overflows
5. Format String Attacks
6. Race Conditions

Race Conditions

- Program behavior (unintentionally) depends on timing of events.

Open vs. Access

- open():
  - Opens a file using the *effective* user ID.
  - A SetUID program owned by root can open any file.
- access():
  - Checks if the *real user* can open a file.

Example

```
char* filename = "/users/joe/myfile";
if (access(filename, R_OK) != 0) exit(-1);
int file = open(filename, O_RDONLY);
read(file, buf, 1023); close(file);
printf("%s\n", buf);
```

- There is a small delay betwee access and open.
- Between access and open, the attacker can set

```
ln -s /etc/passwd /users/joe/myfile
```

- Write a script tha quickly switches the link on and off, until you get access!

Defenses

- Don't use access.
- Drop privileges before calling open.
- If the user doesn't have permissions to the file, open will fail.

```c
char* filename = "/users/joe/myfile";
euid = geteuid();
uid = getuid();

seteuid(uid);
int file = open(filename, O_RDONLY);
read(file, buf, 1023); close(file);

seteuid(euid);

printf("%s\n",buf);
```