Memory Corruption Vulnerabilities, Part I

Some Terminology

- Software **error**
  - A programming mistake that make the software not meet its expectation
- Software **vulnerability**
  - A software error that can lead to possible attacks
- **Attack**
  - The process of exploiting a vulnerability
  - An attack can exploit a vulnerability to achieve additional functionalities for attackers
    - E.g., privilege escalation, arbitrary code execution

Software, One of the Weakest Links in the Security Chain

- Cryptographic algorithms are strong
  - Nobody attacks it
  - Even for crypto hash
- However, even for the best crypto algorithms
  - Software has to implement them correctly
  - A huge of amount of software for other purposes
    - Access control; authentication; ...
- Which programming language to use also matters

Language of Choice for System Programming: C/C++

- Systems software
  - OS; hypervisor; web servers; firmware; network controllers; device drivers; compilers; …
- Benefits of C/C++: programming model close to the machine model; flexible; efficient
- BUT **error-prone**
  - Debugging memory errors is a headache
    - Perhaps on par with debugging multithreaded programs
  - Huge security risk

Agenda

- Compare C to Java
- Common errors for handling C-style buffers
- How to exploit buffer overflows: stack smashing

Comparing C to Java: language matters for security

Comparing C to Java

- Their syntax very similar
- Type safety
  - Safety: something “bad” won’t happen
  - “No untrapped errors”
- Java is type safe
  - Static type system + runtime checks + garbage collection
- C is type unsafe
  - Out-of-bound array accesses
  - Manual memory management
  - Bad type casts
  - ...

Java: Runtime Array Bounds Checking

- Example:

  int a[10];

  a[10] = 3;

  - An exception is raised
  - The length of the array is stored at runtime (the length never changes)

Java: Runtime Array Bounds Checking

- Java optimizer can optimize away lots of unnecessary array bounds checks

```
int sum = 0;
for (i = 0; i<a.length; i++) {
   sum += a[i];
}
```

bounds checking unnecessary

C: No Array Bounds Checking

```
int a[10];
a[10] = 3;
```

- Result in a silent error in C (buffer overflow)
- After that, anything can happen
  - Mysterious crash depending on what was overwritten
  - A security risk as well: if the data written can be controlled by an attacker, then he can possibly exploit this for an attack

Memory Management

- C: manual memory management
  - malloc/free
  - Memory mismanagement problems: use after free; memory leak; double frees
- Java: Garbage Collection
  - No “free” operations for programmers
  - GC collects memory of objects that are no longer used
  - Java has no problems such as use after free, as long as the GC is correct

Non-Null Checking and Initialization Checking in Java

- An object reference is either valid or null
  - Automatic non-null checking whenever it’s used
    - once again, optimizers can eliminate many non-null checks
    - Example: A a = new A(); a.f = 3;
- A variable is always initialized before used
  - Java has a static verifier (at the bytecode level) that guarantees this

Java Strings

- Similar to an array of chars, but immutable
  - The length of the string is stored at runtime to perform bounds checking
- All string operations do not modify the original string (a la functional programming)
  - E.g., s.toLowerCase() returns a new string

C-Style Strings

- C-style strings consist of a contiguous sequence of characters, terminated by and including the first null character.
  - String length is the number of bytes preceding the null character.
  - The number of bytes required to store a string is the number of characters plus one (times the size of each character).

h	e	l	l	o	\0

C Strings: Usage and Pitfalls

Using Strings in C

- C provides many string functions in its libraries (libc)
- For example, we use the strcpy function to copy one string to another:

```
#include <string.h>
char string1[] = "Hello, world!";
char string2[20];
strcpy(string2, string1);
```

Using Strings in C

- Another lets us compare strings

```
char string3[] = "this is";
char string4[] = "a test";
if(strcmp(string3, string4) == 0)
  printf("strings are equal\n");
else printf("strings are different\n")
```

- This code fragment will print "strings are different". Notice that strcmp does **not** return a boolean result.

Other Common String Functions

- strlen: getting the length of a string
- strncpy: copying with a bound
- strcat/strncat: string concatenation
- gets, fgets: receive input to a string
- …

Common String Manipulation Errors

- Programming with C-style strings, in C or C++, is error prone
- Common errors include
  - Buffer overflows
  - null-termination errors
  - off-by-one errors
  - …

gets: Unbounded String Copies

- Occur when data is copied from an unbounded source to a fixed-length character array

```
void main(void) {
  char Password[8];
  puts("Enter a 8-character password:");
  gets(Password);
  printf("Password=%s\n",Password);
}
```

strcpy and strcat

- The standard string library functions do not know the size of the destination buffer

```
int main(int argc, char *argv[]) {
    char name[2048];
    strcpy(name, argv[1]);
    strcat(name, " = ");
    strcat(name, argv[2]);
    ...
}
```

Better String Library Functions

- Functions that restrict the number of bytes are often recommended
- Never use gets(buf)
  - Use `fgets(buf, size, stdin)` instead

From gets to fgets

- char *fgets(char *BUF, int N, FILE *FP);
  - *"Reads at most N-1 characters from FP until a newline is found. The characters including to the newline are stored in BUF. The buffer is terminated with a 0."*

void main(void) {
    char Password[~~8~~ 9];
    puts("Enter a 8-character password:");
    fgets(Password, ~~8~~ 9, stdin);
    …
}

Better String Library Functions

- Instead of strcpy(), use strncpy()
- Instead of strcat(), use strncat()
- Instead of sprintf(), use snprintf()

But Still Need Care

- char *strncpy(char *s1, const char *s2, size_t n);
  - *"Copy not more than n characters (including the null character) from the array pointed to by s2 to the array pointed to by s1; If the string pointed to by s2 is shorter than n characters, null characters are appended to the destination array until a total of n characters have been written."*
  - What happens if the size of s2 is n or greater
    - It gets truncated
    - **And s1 may not be null-terminated!**

Null-Termination Errors

```
int main(int argc, char* argv[]) {
    char a[16], b[16];
    strncpy(a, "0123456789abcdef", sizeof(a));
    printf(“%s\n”,a);
    strcpy(b, a);
}
```

a[] not properly terminated. Possible segmentation fault if printf(“%s\n”,a);

How to fix it?

strcpy to strncpy

- Don't replace

  ```
  strcpy(dest, src)
  ```

  by

  ```
  strncpy(dest, src, sizeof(dest))
  ```

  but by

  ```
  strncpy(dest, src, sizeof(dest)-1)
  dst[sizeof(dest)-1] = `\0`;
  ```

  if dest should be null-terminated!

- You never have this headache in Java

Signed vs Unsigned Numbers

```
char buf[N];
int i, len;

read(fd, &len, sizeof(len));
if (len > N)
  {error ("invalid length"); return; }
read(fd, buf, len);
```

We forget to check for negative lengths

`len` cast to unsigned and negative length overflows

Checking for Negative Lengths

```
char buf[N];
int i, len;

read(fd, &len, sizeof(len));
if (len > N || len < 0)
   {error ("invalid length"); return; }
read(fd, buf, len);
```

It still has a problem
if the buf is going to be treated as a C string.

A Good Version

```
char buf[N];
int i, len;

read(fd, &len, sizeof(len));
if (len > N-1 || len < 0)
   {error ("invalid length"); return; }
read(fd, buf, len);
buf[len] = '\0'; // null terminate buf
```

Buffer Overflows

Problems Caused by Buffer Overflows

- The first Internet worm, and many subsequent ones (CodeRed, Blaster, ...), exploited buffer overflows

- Buffer overflows cause in the order of 50% of all security alerts
  - E.g., check out CERT, cve.mitre.org, or bugtraq

- Trends
  - Attacks are getting cleverer
    - defeating ever more clever countermeasures
  - Attacks are getting easier to do, by script kiddies

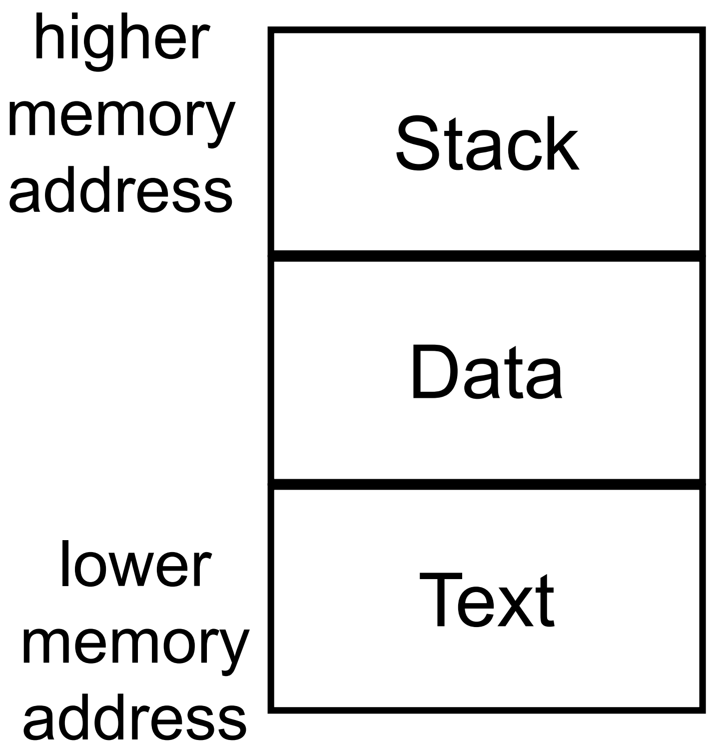
How Can Buffer Overflow Errors Lead to Software Vulnerabilities?

- All the examples look like simple *programming bugs*
- How can they possibly enable attackers to do bad things?
  - **Stack smashing** to exploit buffer overflows
  - Illustrate the technique using the Intel x86-64 architecture

Compilation, Program, and Process

- Compilation
  - From high-level programs to low-level machine code
- Program: static code and data
- Process: a run of a program

Process Memory Region



- Text: static code
- Data: also called heap
  - static variables
  - dynamically allocated data (malloc, new)
- Stack: program execution stacks

Program Stack

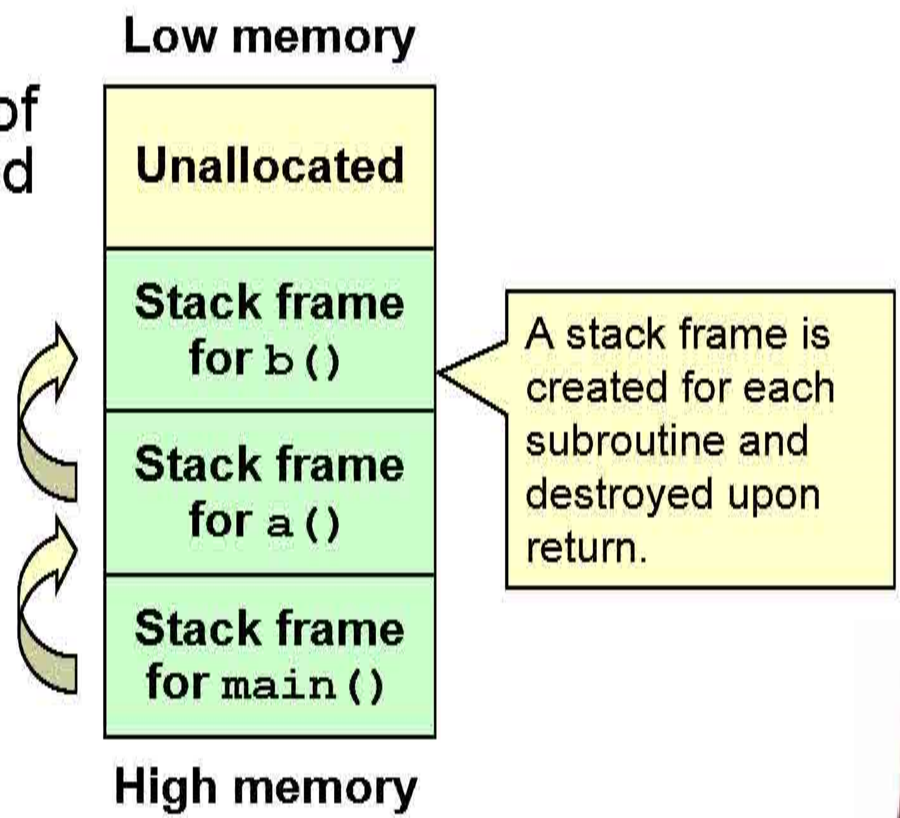
- For implementing procedure calls and returns
- Keep track of program execution and state by storing
  - local variables
  - Some arguments to the called procedure (callee)
    - Depending on the calling convention
  - return address of the calling procedure (caller)
  - ...

Stack Segment

The stack supports nested invocation calls

Information pushed on the stack as a result of a function call is called a frame

```
b()  {…}
a()  {
   b();
}
main()  {
   a();
}
```



Stack Frames

- Stack grows from high mem to low mem
- The stack pointer points to the top of the stack
  - RSP in Intel x86-64
- The frame pointer points to the end of the current frame
  - also called the base pointer
  - RBP in Intel x86-64
- The stack is modified during
  - function calls
  - function initialization
  - returning from a function

A Running Example

```
void function(int a, int b) {
      char buffer[12];
      gets(buffer);
      return;
}

void main() {
      int x;
      x = 0;
      function(1,2);
      x = 1;
      printf("%d\n",x);
}
```

Run “gcc –S –o example.s example.c” to see its assembly code

- The exact assembly code will depend on many factors (the target architecture, optimization levels, compiler options, etc);
- We show the case for unoptimized x86-64

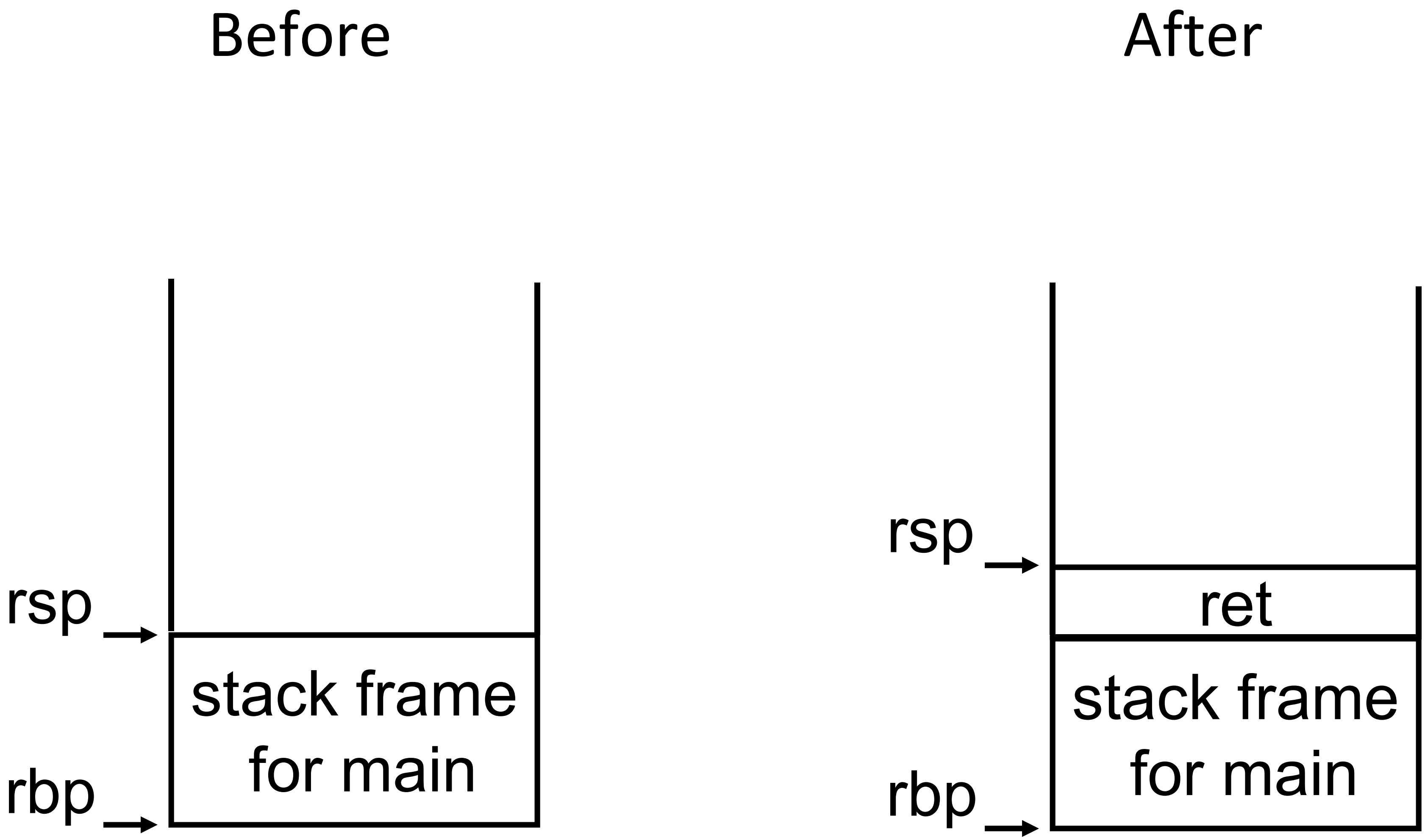
Function Calls

function (1,2)

movl $2, %esi	pass the 2nd arg
movl $1, %edi	pass the 1st arg
call function	push the ret addr onto the stack, and jumps to the function

Note: in x86-64, the first 6 args are passed via registers (rdi, rsi, rdx, rcx, r8, r9)

Function Calls: Stacks



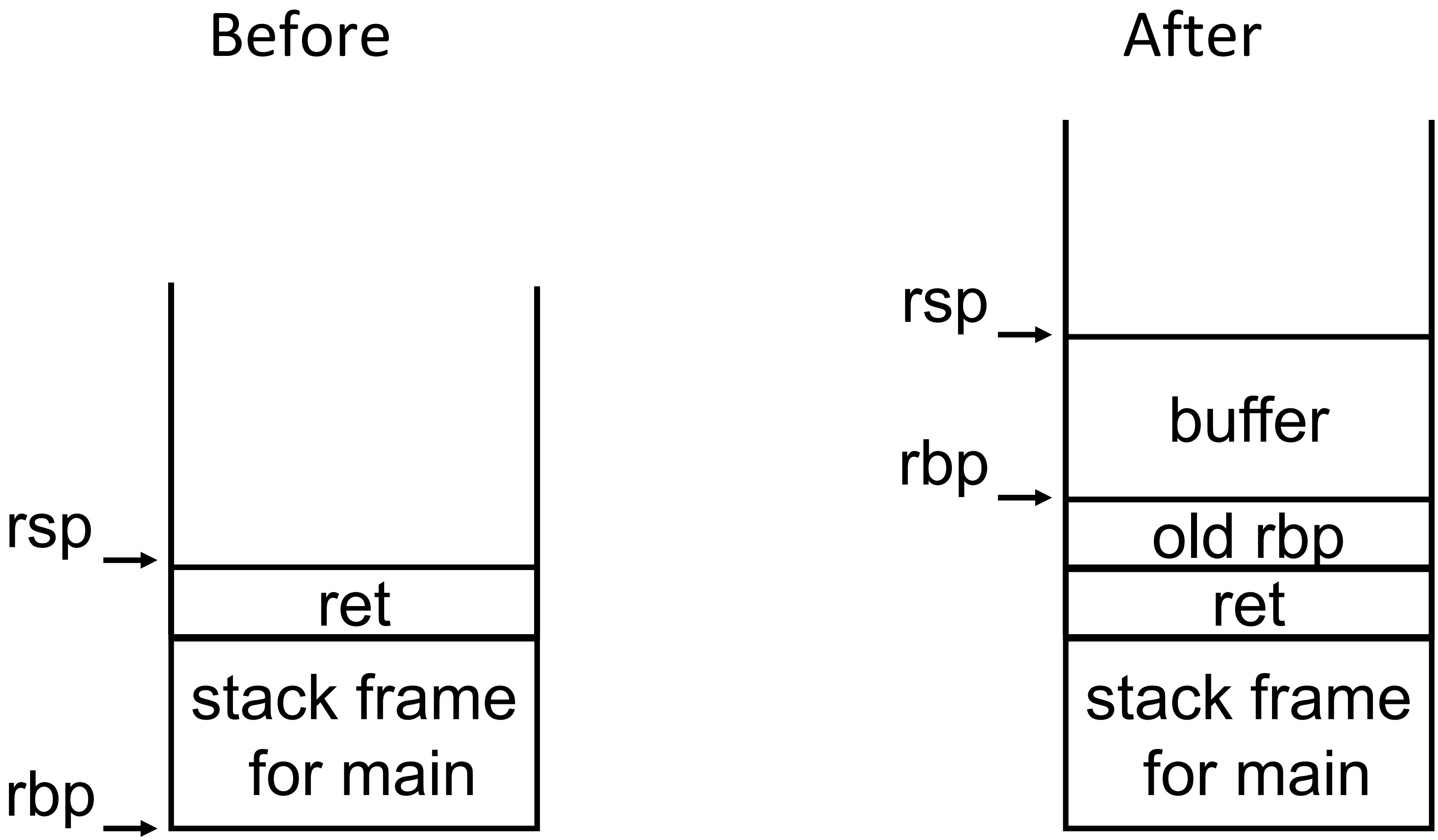
Function Initialization

void function(int a, int b) {

pushq %rbp	save the frame pointer
movq %rsp, %rbp	set the new frame pointer
subq $32, %rsp	allocate space for local variables

Procedure prologue

Function Initialization: Stacks

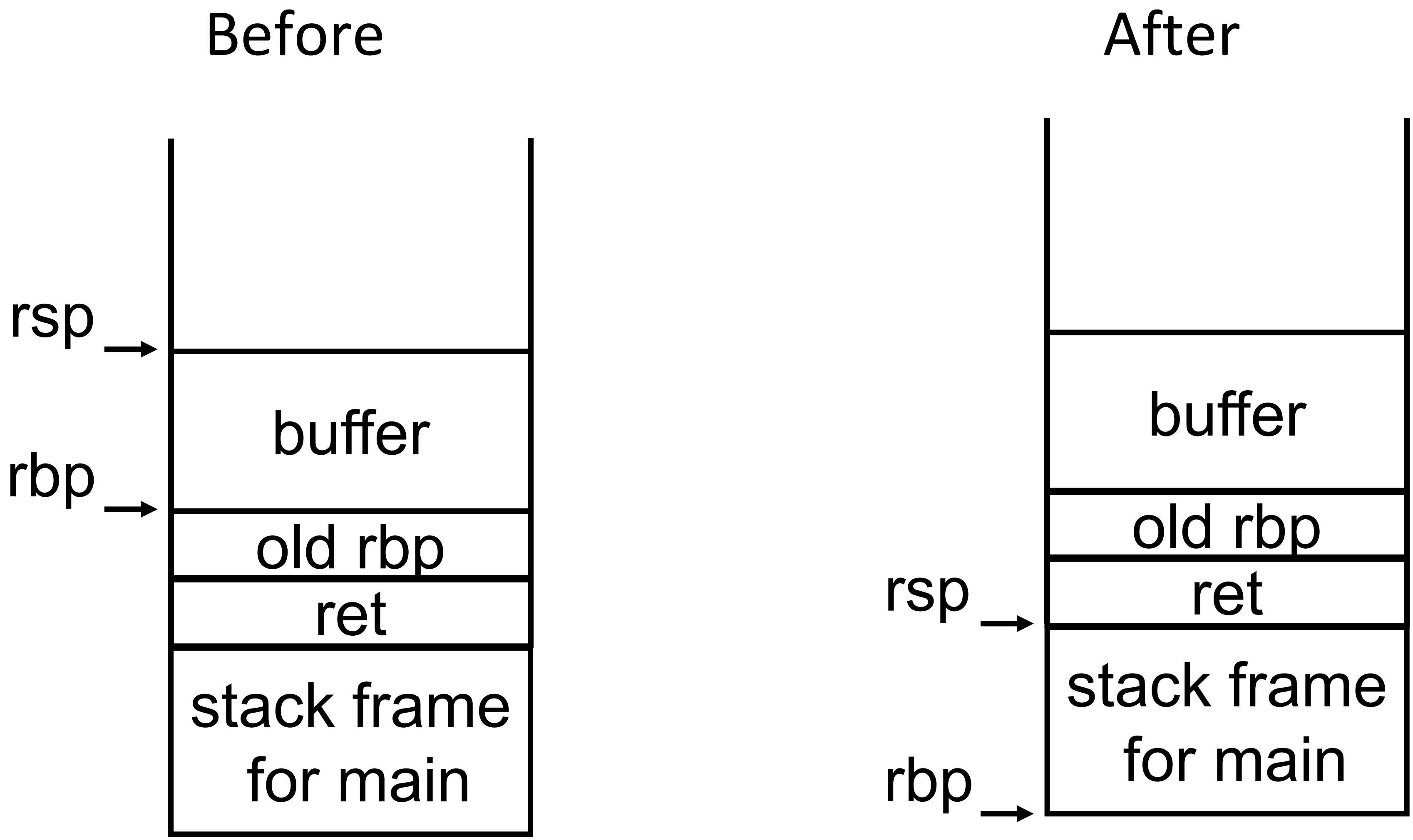


Function Return

return;

movq %rbp, %rsp	restores the old stack pointer
popq %rbp	restores the old frame pointer
ret	gets the return address, and jumps to it

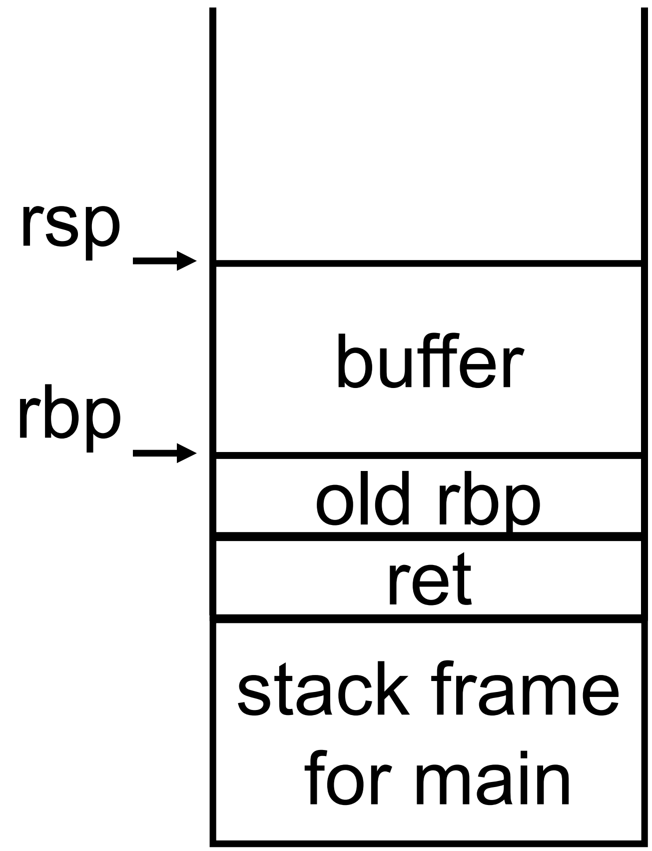
Function Return: Stacks



A Running Example

```
void function(int a, int b) {
      char buffer[12];
      gets(buffer);
      return;
}

void main() {
      int x;
      x = 0;
      function(1,2);
      x = 1;
      printf("%d\n",x);
}
```

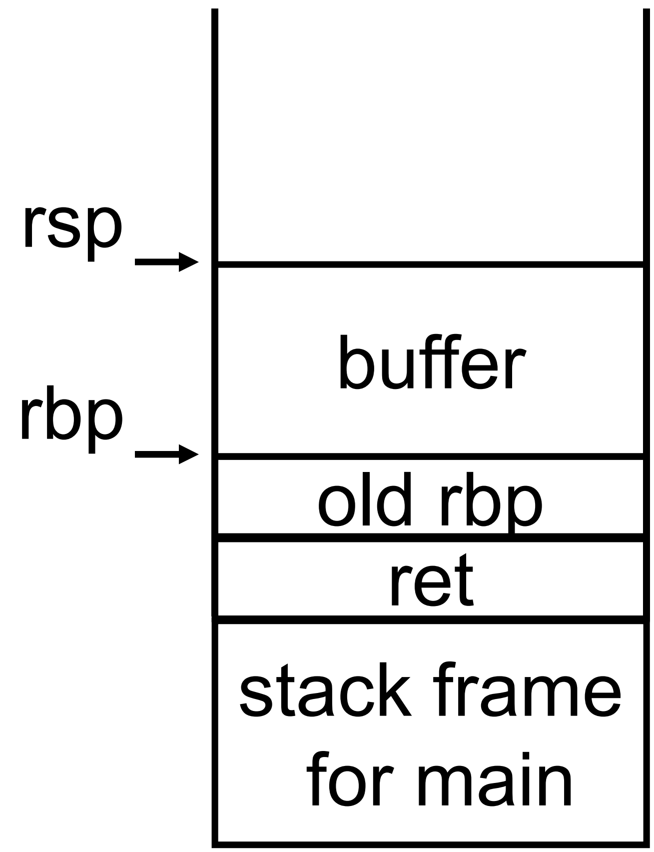


Overwriting the Return Address

```
void function(int a, int b) {
    char buffer[12];
    gets(buffer);

    long* ret =
        (long *) ((long)buffer+?);
    *ret = *ret + ?;

    return;
}
```



Overwriting the Return Address

```
void function(int a, int b) {
      char buffer[12];
      gets(buffer);

      long* ret =  (long *) ((long)buffer+40);
      *ret = *ret + 7;

      return;
}

void main() {
      int x;
      x = 0;
      function(1,2);
      x = 1;
      printf("%d\n",x);
}
```

The output will be 0

the original return address

the new return address

The Previous Attack

- Not very realistic
  - Attackers are usually not allowed to modify code
  - Threat model: the only thing they can affect is the input
  - Can they still carry out similar attacks?
    - YES, because of possible buffer overflows

Buffer Overflow

- A buffer overflow occurs when data is written outside of the boundaries of the memory allocated to a particular data structure
- Happens when buffer boundaries are neglected and unchecked
- Can be exploited to modify
  - return address on the stack
  - local variable
  - heap data structures
  - function pointer

Smashing the Stack

- Occurs when a buffer overflow overwrites data in the program stack
- Successful exploits can overwrite the return address on the stack
  - Allowing execution of arbitrary code on the targeted machine

Smashing the Stack: example.c

What happens if we input a large string?

./example

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Segmentation fault

What Happened? The Stack is Smashed

```
void function(int a, int b) {
    char buffer[12];
    gets(buffer);
    return;
}
```

If the input is large, then gets(buffer) will write outside the bound of buffer, and the return address is overwritten

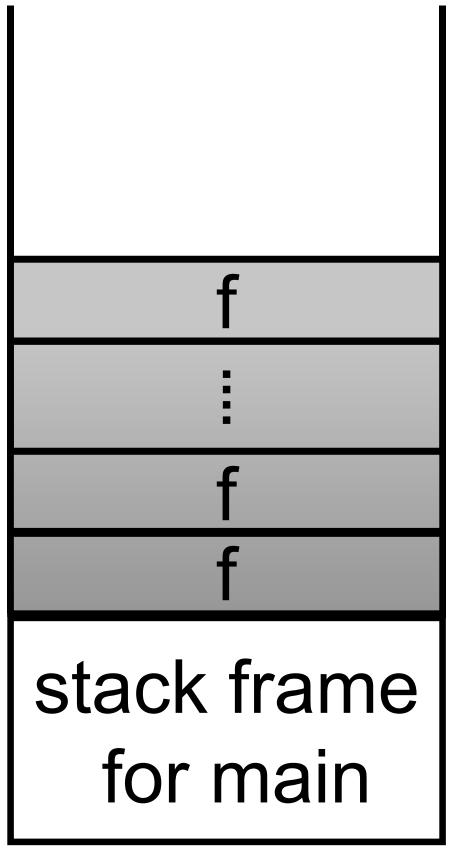
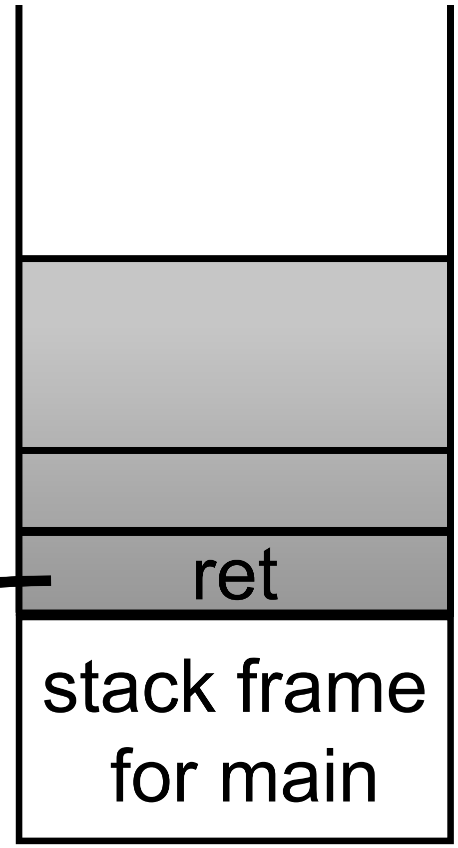


Figure Out A Nasty Input

```
void function (int a, int b) {
      char buffer[12];
      gets(buffer);
      return;
}

void main() {
      int x;
      x = 0;
      function(1,2);
      x = 1;
      printf("%d\n",x);
}
```

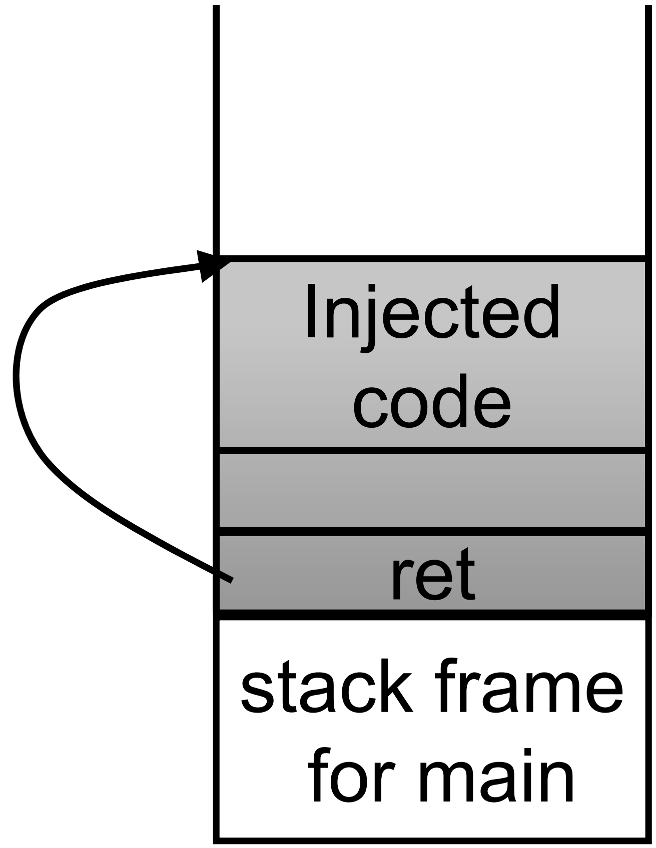


A nasty input puts the return address after x=1.
Arc injection

Injecting Code

```
void function (int a, int b) {
    char buffer[12];
    gets(buffer);
    return;
}

void main() {
    int x;
    x = 0;
    function(1,2);
    x = 1;
    printf("%d\n",x);
}
```

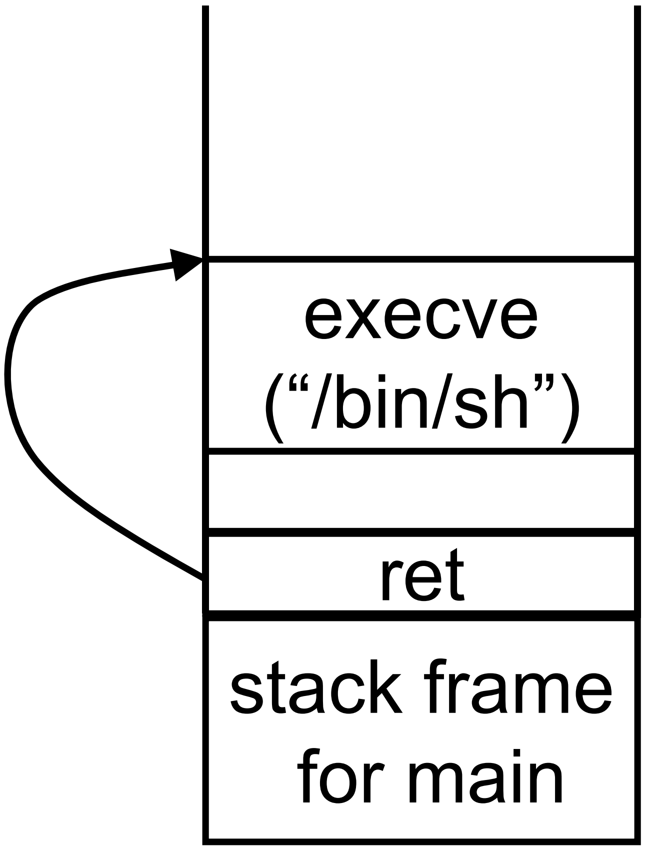


The injected code can do anything. E.g., download and install a worm

Code Injection

- Attacker creates a malicious argument—a specially crafted string that contains a pointer to malicious code provided by the attacker
- When the function returns, control is transferred to the malicious code
  - Injected code runs with the permission of the vulnerable program when the function returns.
  - Programs running with root or other elevated privileges are normally targeted
    - Programs with the setuid bit on

Injecting Shell Code



- This brings up a shell
- Attacker can execute any command in the shell
- The shell has the same privilege as the process
- Usually a process with the root privilege is attacked

Morris Worm (1988)

- Worked by exploiting known buffer-overflow vulnerabilities in sendmail, fingerd, rsh/rexec and weak passwords.
  - e.g., it exploited a gets call in fingerd
- Infected machines probe other machines for vulnerabilities
- 6000 Unix machines were infected; $10M-$100M cost of damage
- Robert Morris was tried and convicted of violation of Computer Fraud and Abuse Act
  - 3 years of probation; 400 hours of community service; $10k fine
  - Had to quit his Cornell PhD

Any C(++) code acting on untrusted input is at risk

- code taking input over untrusted network
  - eg. sendmail, web browser, wireless network driver,...
- code taking input from untrusted user on multi-user system,
  - esp. services running with high privileges (as ROOT on Unix/Linux, as SYSTEM on Windows)
- code processing untrusted files
  - that have been downloaded or emailed
- also embedded software, eg. in devices with (wireless) network connection such as mobile phones with Bluetooth, wireless smartcards in new passport or OV card, airplane navigation systems, ...