Security Part 1

Sergio Benitez

Policy & Threat Model

- Together, define security of a system
- Policy: goals of a secure system
  - Only faculty can access grades
  - Only owner of credit card can use it
- Threat model: assumptions
  - Faculty keeps their passwords secure
  - Credit card lost/stolen without knowledge

Guard Model

- Insight: Usually want to protect data
- Idea: Have something guard it
  - Only guard can give access to (any) data
- Examples:
  - OS (files, processes)
  - Social Network (pictures, posts)
- How do we implement a guard?

Guard Implementation

- Authentication
  - Proof of identity
  - Usually username/password
- Authorization
  - Proof of permission
  - Usually access control list
  - ACL: {username -> permissions}

Passwords

- Policy
  - Only user with password can access his data
  - Compromise should be locally contained
- Threat model
  - Adversary is smart, can hack your machines
  - Adversary > average, < infinite, compute power

Passwords

- Straw-Man 1
  - Keep DB mapping username to password
  - User sends you username/password, verify

Passwords

- Straw-Man 1
  - Keep DB mapping username to password
  - User sends you username/password, verify
- Terrible!
- Adversary hacks machine, sees all passwords!
- Adversary can take user password elsewhere!

Passwords

- Straw-Man 2
  - Have good hash function, h
  - Keep DB mapping username to h(password)
  - User sends username/pass, verify h(pass) match

Passwords

- Straw-Man 2
  - Have good hash function, h
    - Keep DB mapping username to h(password)
    - User sends username/pass, verify h(pass) match
- Better; adversary hacks machine, sees only hash
- But, users with same password have same hash
- Adversary constructs rainbow table, gets many

Passwords

- Good?
  - Have great hash function, h
  - Have random, per-user salt, s
  - Map username to (h(password + s), s)
  - Verify h(pass + s) match

Passwords

- Good
  - Have great hash function, h
  - Have random, per-user salt, s
  - Map username to (h(password + s), s)
  - Verify h(pass + s) match
- Now, same passwords have different hash
- Rainbow table needs to be reconstructed per salt

Except...

- What if adversary observes network?
  - Can then read password from network!
  - Hint: encrypt the connection

Except...

- What if adversary observes network?
  - Can then read password from network!
    - Hint: encrypt the connection
- What if adversary takes over server?
  - Then users send passwords directly!
  - Hint: have server prove identity

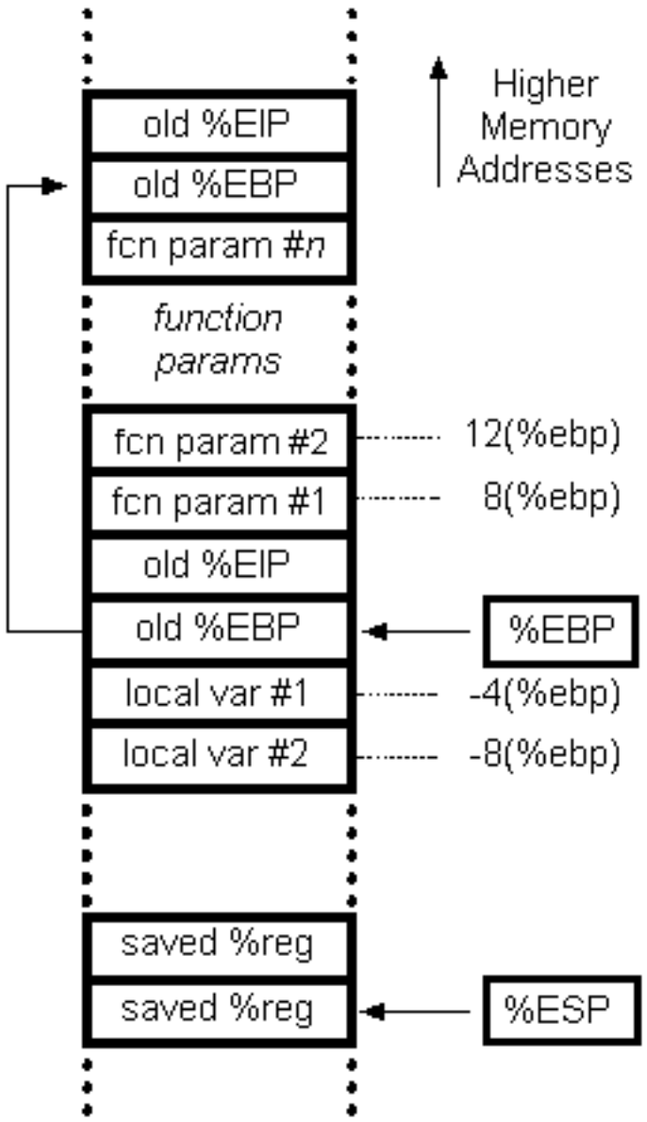
Buffer Overruns

(or how C has cost people millions of dollars)

Computer Memory

- User memory separated into two pieces
- Heap: dynamic memory allocation
  - Global variables, objects, etc.
  - Memory stays until manually cleared
- Stack: local memory allocation
  - Local variables, function parameters, etc.
  - Stack “frames” only live as long as function

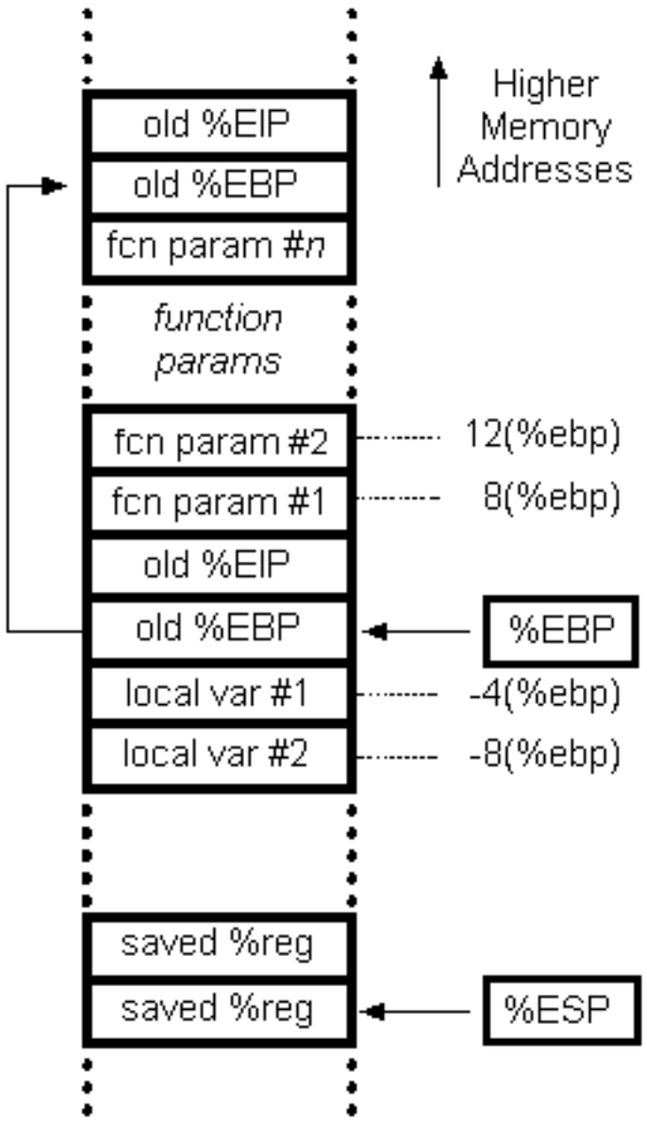
Stack



when calling add(1, 2):

```
push 2
push 1
push %eip
jmp *add
```

Stack



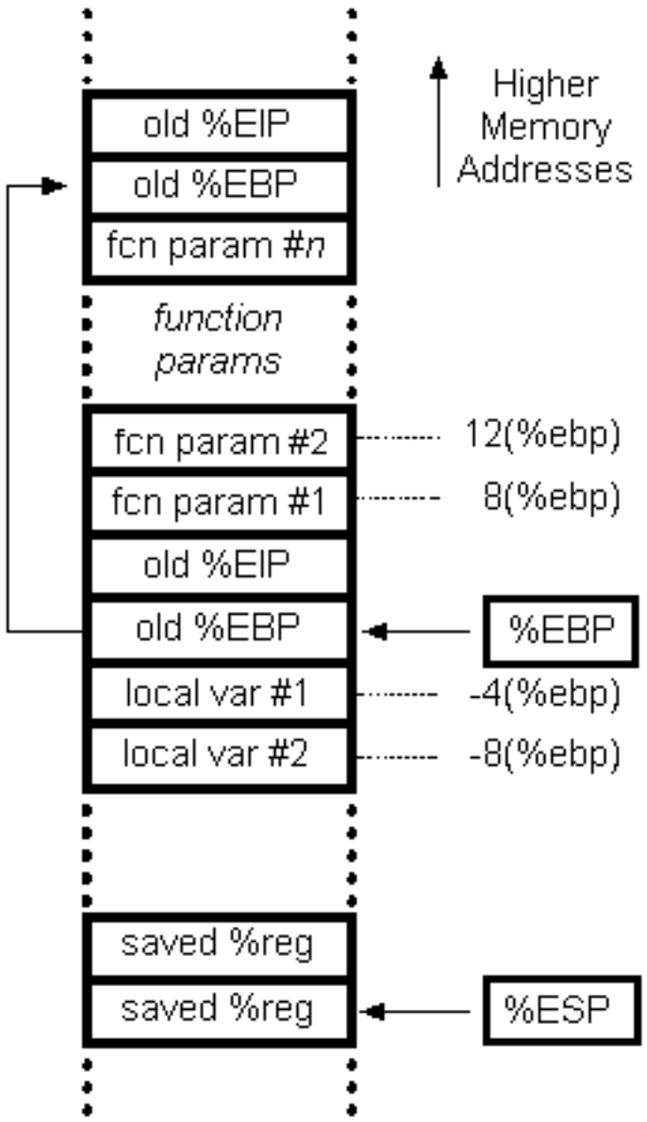
when calling add(1, 2):

```
push 2
push 1
push %eip
jmp *add
```

when called add(1, 2):

```
push %ebp
mov %esp, %ebp
```

Stack



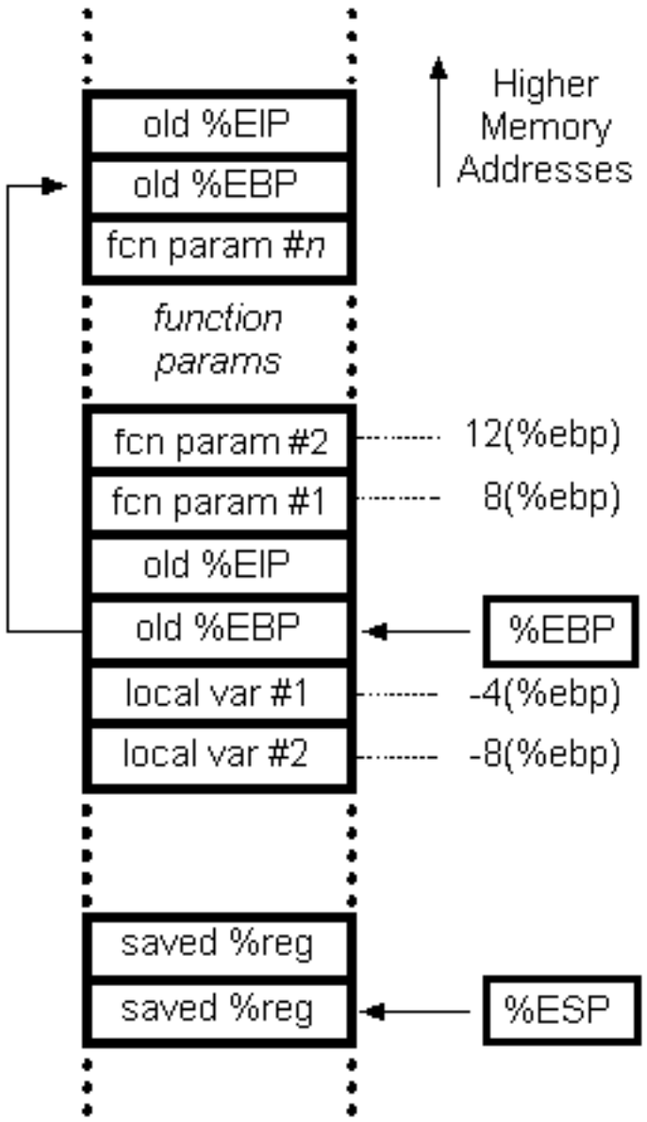
when calling add(1, 2):

```
push 2
push 1
push %eip
jmp *add
```

when called add(1, 2):

```
push %ebp
mov %esp, %ebp
push local vars...
```

Stack



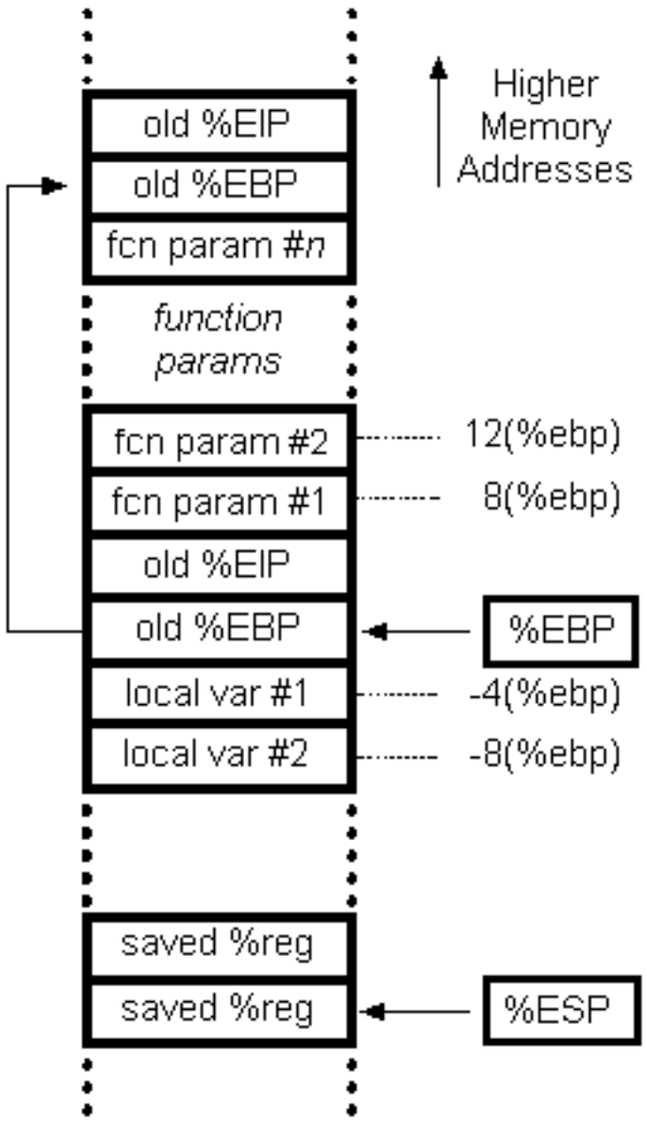
when calling add(1, 2):

```
push 2
push 1
push %eip
jmp *add
```

when called add(1, 2):

```
push %ebp
mov %esp, %ebp
push local vars...
add 8(%ebp), 12(%ebp), %eax
```

Stack



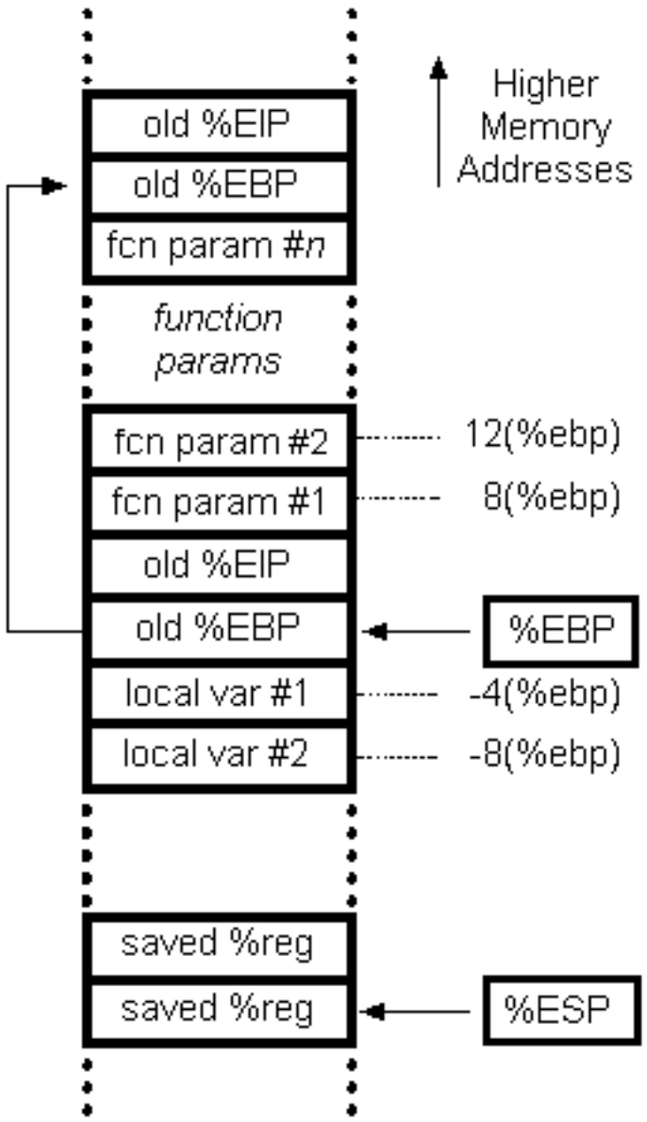
when calling add(1, 2):

```
push 2
push 1
push %eip
jmp *add
```

when called add(1, 2):

```
push %ebp
mov %esp, %ebp
push local vars...
add 8(%ebp), 12(%ebp), %eax
movl %ebp, %esp
pop %ebp
pop %eip
```

Pop %eip



when called add(1, 2):

... stuff ...

pop %eip

- Sets instruction pointer to eip
- Like doing a jmp

- This is what buffer overflows exploit
- What if we could change that value?
  - The value on the stack
- Then, we could redirect the program

Typical Stack Smashing

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

void say_hi(void) {
  char buffer[1024];
  printf("Hi! What's your name? ");
  gets(buffer, stdin);
  printf("\nHi %s!\n", buffer);
}
```

Typical stack smashing loads a buffer with executable code and replaces the return address on the stack with the memory address of the beginning of the executable code.

How do you get the address to the buffer? A couple of good ways.

1) You try and try again until you get it right.
2) You pad the beginning of the buffer as much as possible with nops. This increases your chances of getting a valid address.
3) Trampolining: Use a register that contains a address close to your buffer: Jump to your known piece of code (that can be stored anywhere, heap, static, etc) by overflowing with the known value over and over again. Your code then jmps relatively using the register.

Arc Injection

```
void say_hi(void) {
   char buffer[12];
   printf("Hi! What's your name? ");
   gets(buffer, stdin);
   printf("\nHi %s!\n", buffer);
}
```

Arc injection replaces the return pointer with the address of known function and places the right parameters above %bp. libc functions are good targets: “system”, or “exec”

Function-Pointer Attack

```
void give_access(void) {
   void (*f)();
   char buffer[12];
   printf("Hi! What's your name? ");
   gets(buffer, stdin);
   f();
}
```

Since the function pointer is declared before the buffer, the function pointer will be above (in higher memory space) than the buffer. Overrunning the buffer will overwrite the function pointer.

FP Attack?

```
void give_access(void) {
   char buffer[12];
   void (*f)();
   printf("Hi! What's your name? ");
   gets(buffer, stdin);
   f();
}
```

Not a function pointer clobbering attack anymore! This is because now the buffer is above the function pointer in the stack. Therefore, overrunning the buffer won't overwrite the function pointer. Note that you can still do arc-injection.

True or False

(T/F) By not allowing writes outside of the bounds of objects, Java eliminates all risk of attacks based on stack smashing, assuming that the VM and any native libraries are bug free.

True or False

(T/F) By not allowing writes outside of the bounds of objects, Java eliminates all risk of attacks based on stack smashing, assuming that the VM and any native libraries are bug free.

True

Of course! The whole problem is that we can overrun buffers. If we are contained, the problem is fixed.

True or False

(T/F) Setting the permissions on the stack memory to prevent execution of code would foil attacks based on “return into libc”.

True or False

(T/F) Setting the permissions on the stack memory to prevent execution of code would foil attacks based on "return into libc".

False

Return into libc doesn't execute code on the stack; it jumps to a known libc function and executes that function which is in static memory.

True or False

(T/F) Making the stack begin at a memory location chosen randomly at runtime would foil the original stack smashing exploit.

True or False

(T/F) Making the stack begin at a memory location chosen randomly at runtime would foil the original stack smashing exploit.

True

The original stack exploit depends on being able to guess the memory location of the buffer. If the location keeps changing, it's impossible to guess it.

True or False

(T/F) Using function pointers presents additional opportunities for arc injection.

True or False

(T/F) Using function pointers presents additional opportunities for arc injection.

True

Yes. We overwrite the function pointer to point to a known function.

Does it fix it?

Random stack: Place the stack in an area of memory randomly chosen for each new process, rather than at the same address for every process.

Simple buffer overrun?

Trampoline?

Arc Injection?

Does it fix it?

Random stack: Place the stack in an area of memory randomly chosen for each new process, rather than at the same address for every process.

Simple buffer overrun? **Yes**

Trampoline? **No**

Arc Injection? **No**

Does it fix it?

Non-executable stack: Set permissions on stack to RW. Set permissions on code memory to RX.

Simple buffer overrun?

Trampoline?

Arc Injection?

Does it fix it?

Non-executable stack: Set permissions on stack to RW. Set permissions on code memory to RX.

Simple buffer overrun? **Yes**

Trampoline? **Yes**

Arc Injection? **No**

Does it fix it?

Bounds Checking: Use a language such as Java or Scheme that checks that all array/buffer indices are valid.

Simple buffer overrun?

Trampoline?

Arc Injection?

Does it fix it?

Bounds Checking: Use a language such as Java or Scheme that checks that all array/buffer indices are valid.

Simple buffer overrun? **Yes**

Trampoline? **Yes**

Arc Injection? **Yes**

Qs?