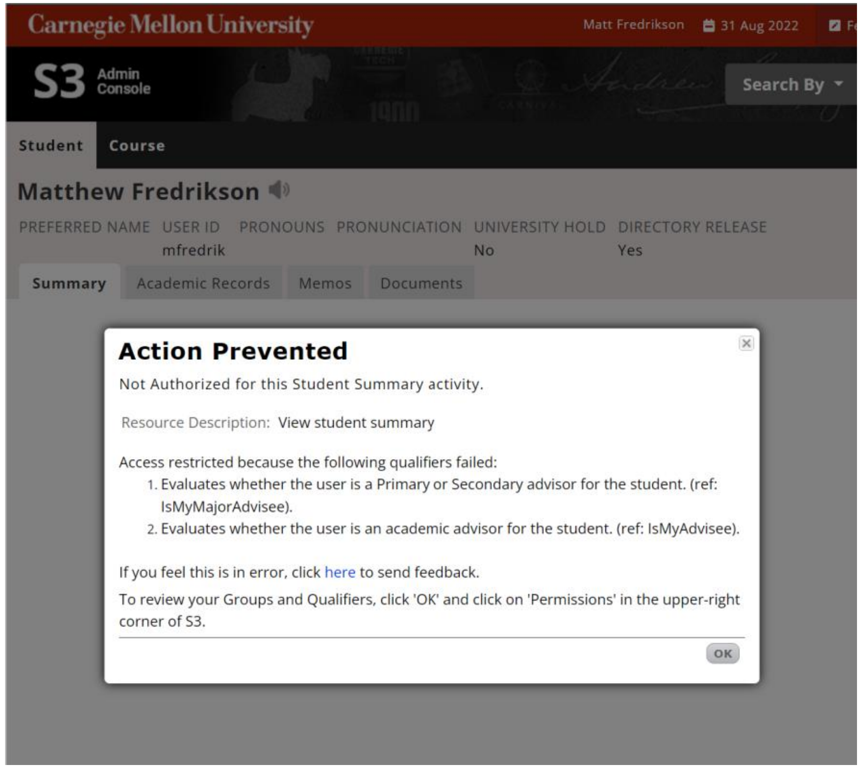
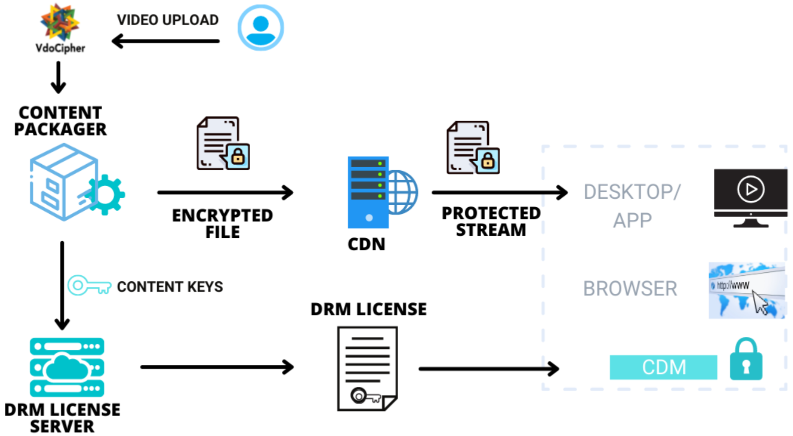
Lecture 2:
Safety & Proof

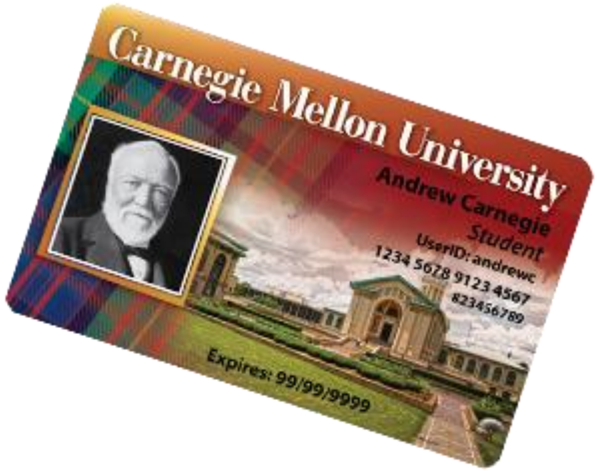
Today’s goals

- Introduce *safety* as a distinct type of security policy
- See examples of common safety problems
- Understand limitations of heuristic defenses
- Formulate a good mitigation

Policies are everywhere

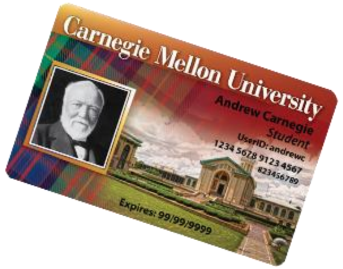






How to enforce them?

- Runtime monitor
- Audit
- Design the system accordingly
  - Build a good test suite
  - Use linters, static analysis tools
  - Verify the implementation (or a model of it)
- …



Safety

A class of policies that can *always* be enforced at runtime

If the presented student ID is expired, deny payment

Only the student's primary advisor may view their summary

Standard users may view content, but cannot copy it

Information labeled top secret may not influence the contents of secret documents

Safety policies do not always **need** to be enforced at runtime, though

Memory Safety

What is memory safety?

- To an extent, depends on the platform
- On modern systems, indicates the absence of:
  - Buffer overflow, over-read
  - Concurrent reads/writes to shared memory
  - Uninitialized read/dereference, null dereference
  - Invalid page fault
  - Use after free, invalid/double free
  - Stack/heap exhaustion
- In many cases, violations result in corrupted state, unstable behavior
- Sometimes they result in exploitable vulnerabilities

Recap: Buffer overflow

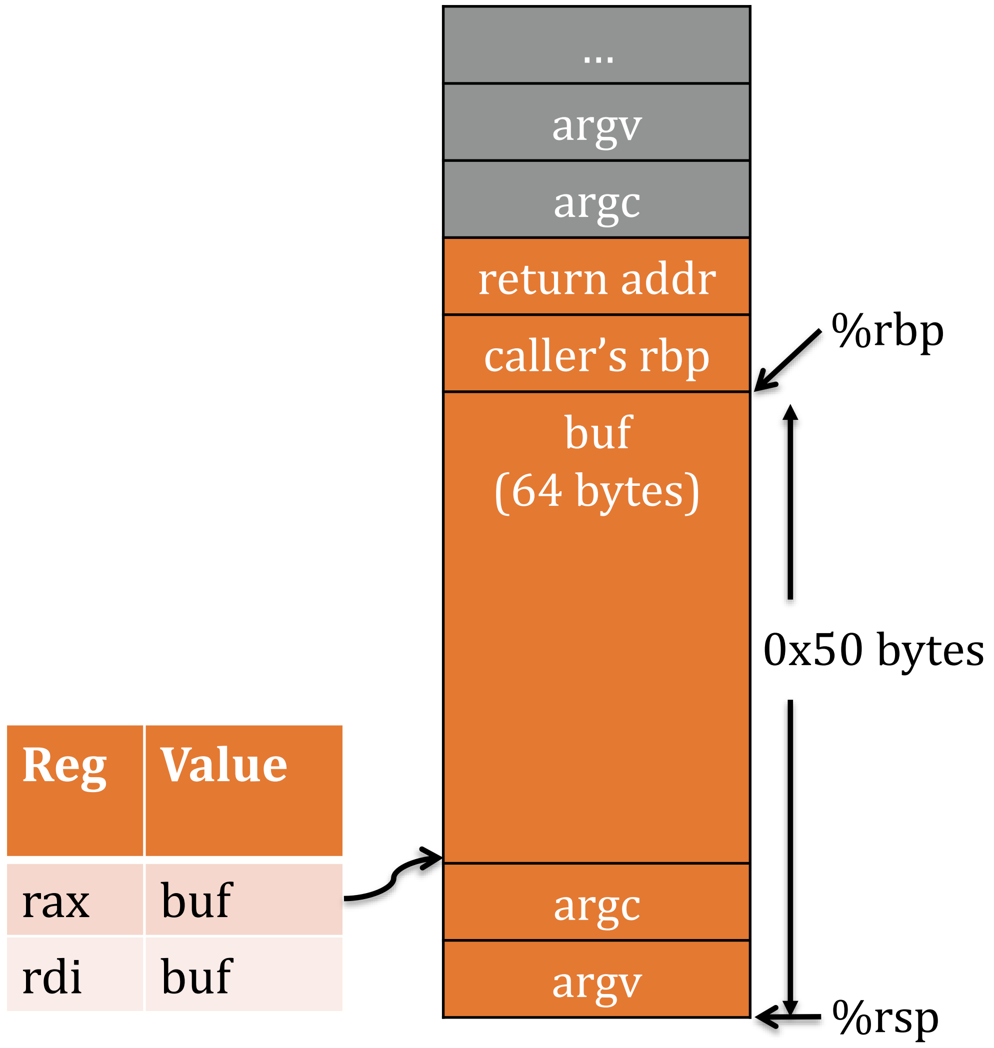
Occurs when data is <u>written</u> to a location outside of the space allocated for a buffer

A buffer may be allocated:

- On the stack
  - Covered today
  - Easiest case to exploit
- On the heap
  - Not covered
  - May still be exploitable, but more advanced

Basic Example

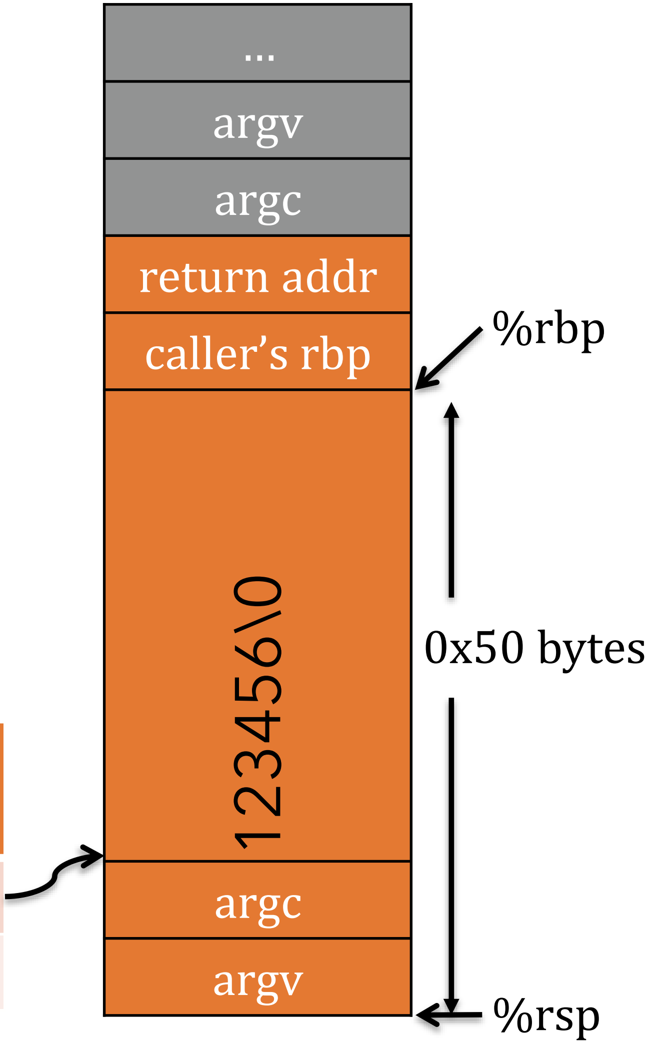
```
int main(int argc, char **argv) {
    char buf[64];
    gets(buf);
}
Dump of assembly code for function main:
  4004fd: push   %rbp
  4004fe: mov    %rsp,%rbp
  400501: sub    $0x50,%rsp
  400505: mov    %rdi,-0x48(%rbp)
  400508: mov    %rsi,-0x50(%rbp)
  40050c: lea    -0x40(%rbp),%rax
  400510: mov    %rax,%rdi
  400518: callq  400400 <gets@plt>
  40051d: leaveq
  40051e: retq
```



“123456”

```
int main(int argc, char **argv) {
    char buf[64];
    gets(buf);
}
Dump of assembly code for function main:
  4004fd: push   %rbp
  4004fe: mov    %rsp,%rbp
  400501: sub    $0x50,%rsp
  400505: mov    %rdi,-0x48(%rbp)
  400508: mov    %rsi,-0x50(%rbp)
  40050c: lea    -0x40(%rbp),%rax
  400510: mov    %rax,%rdi
  400518: callq  400400 <gets@plt>
  40051d: leaveq
  40051e: retq
```

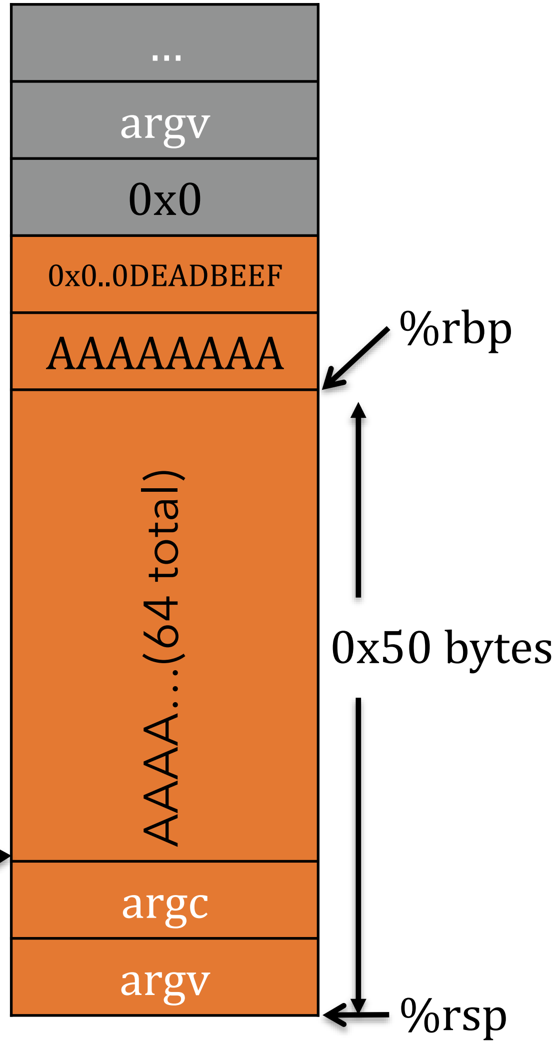
Reg	**Value**
rax	buf
rdi	buf



“A”x72 + “\xEF\xBE\xAD\xDE\x00\x00\x00\x00”

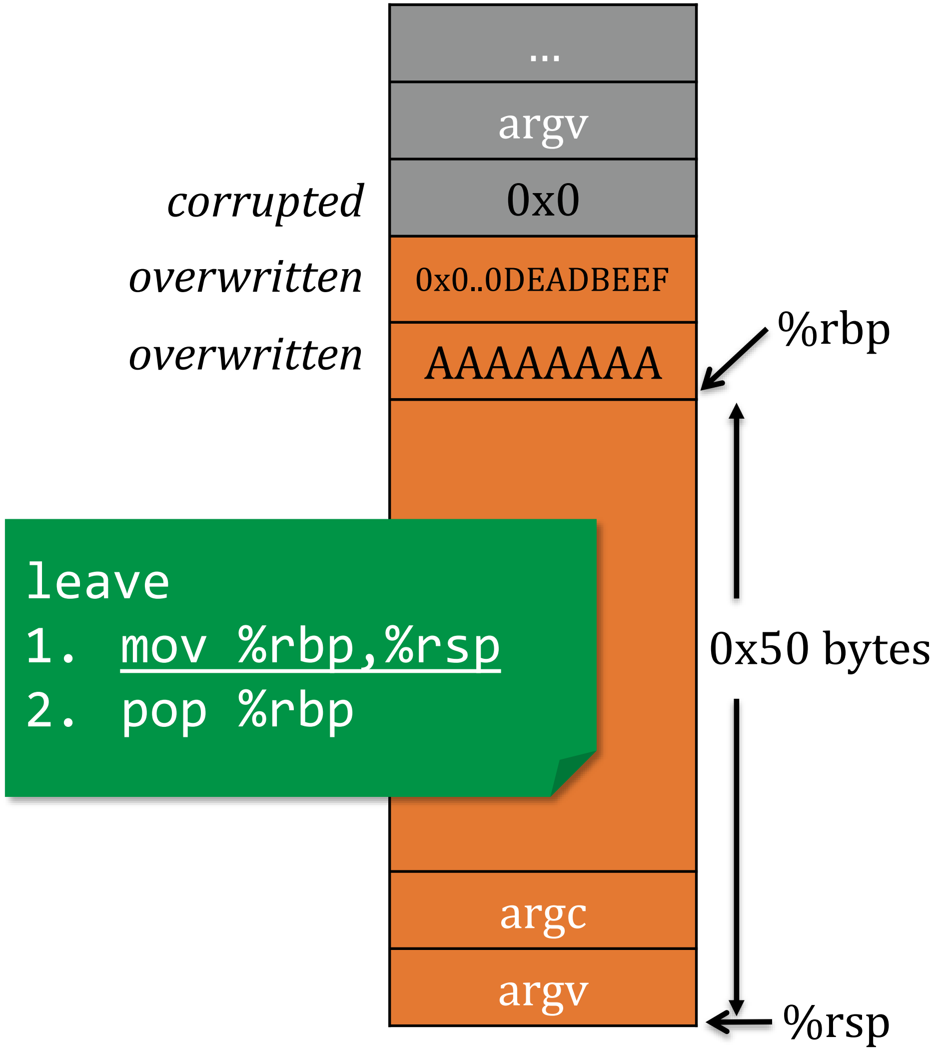
```
int main(int argc, char **argv) {
    char buf[64];
    gets(buf);
}
Dump of assembly code for function main:
  4004fd: push   %rbp
  4004fe: mov    %rsp,%rbp
  400501: sub    $0x50,%rsp
  400505: mov    %rdi,-0x48(%rbp)
  400508: mov    %rsi,-0x50(%rbp)
  40050c: lea    -0x40(%rbp),%rax
  400510: mov    %rax,%rdi
  400518: callq  400400 <gets@plt>
  40051d: leaveq
  40051e: retq
```

Reg	Value
rax	buf
rdi	buf



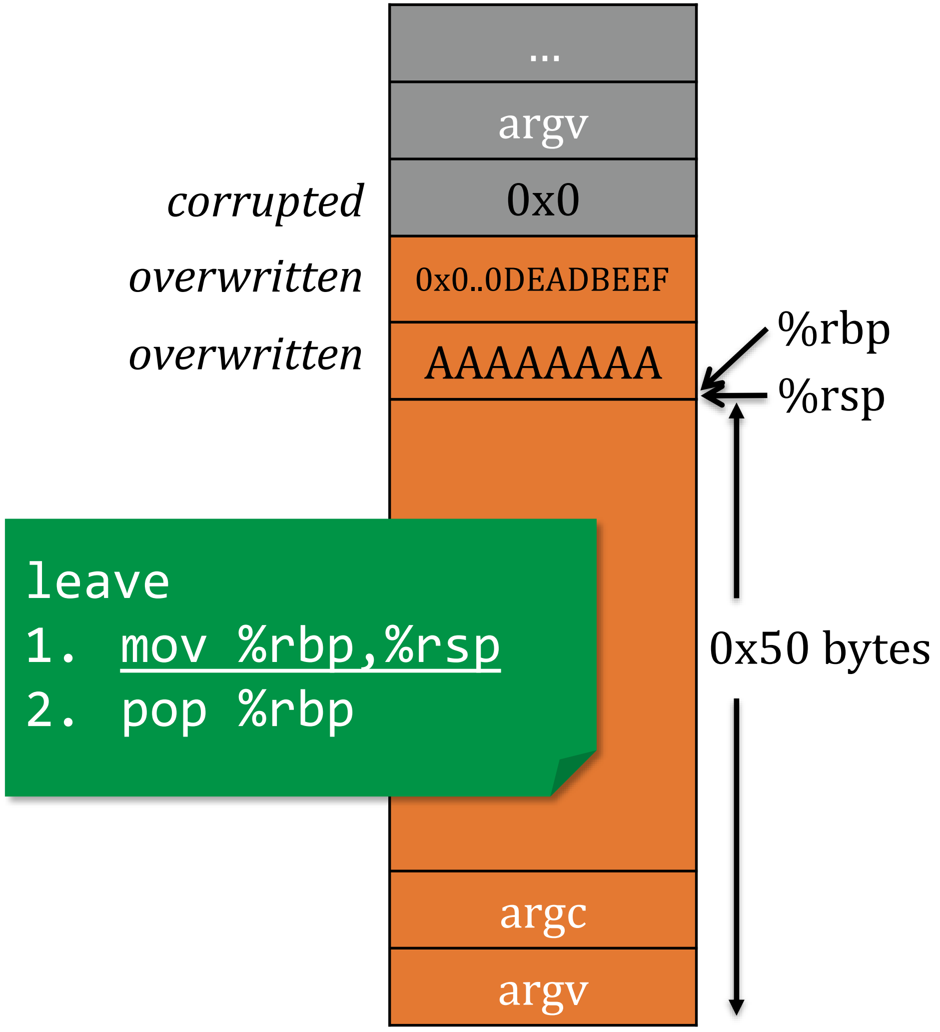
Frame teardown, part 1

```
int main(int argc, char **argv) {
    char buf[64];
    gets(buf);
}
Dump of assembly code for function main:
  4004fd: push   %rbp
  4004fe: mov    %rsp,%rbp
  400501: sub    $0x50,%rsp
  400505: mov    %rdi,-0x48(%rbp)
  400508: mov    %rsi,-0x50(%rbp)
  40050c: lea    -0x40(%rbp),%rax
  400510: mov    %rax,%rdi
  400518: callq  400400 <gets@plt>
  40051d: leaveq
  40051e: retq
```



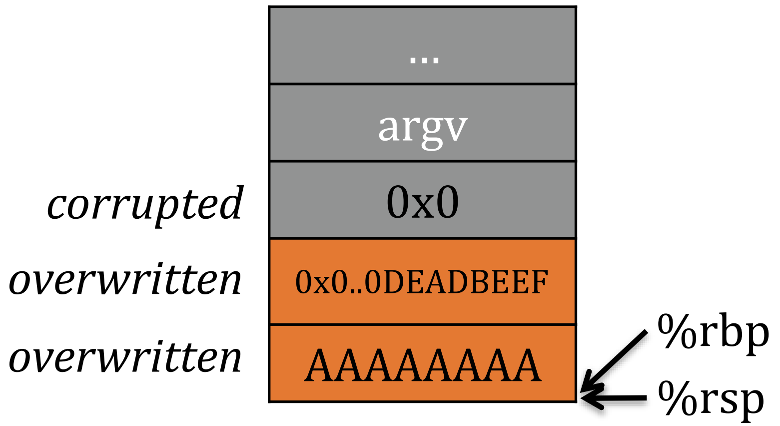
Frame teardown, part 1

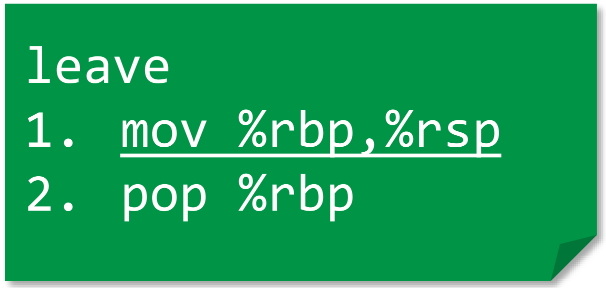
```
int main(int argc, char **argv) {
    char buf[64];
    gets(buf);
}
Dump of assembly code for function main:
  4004fd: push   %rbp
  4004fe: mov    %rsp,%rbp
  400501: sub    $0x50,%rsp
  400505: mov    %rdi,-0x48(%rbp)
  400508: mov    %rsi,-0x50(%rbp)
  40050c: lea    -0x40(%rbp),%rax
  400510: mov    %rax,%rdi
  400518: callq  400400 <gets@plt>
  40051d: leaveq
  40051e: retq
```



Frame teardown, part 1

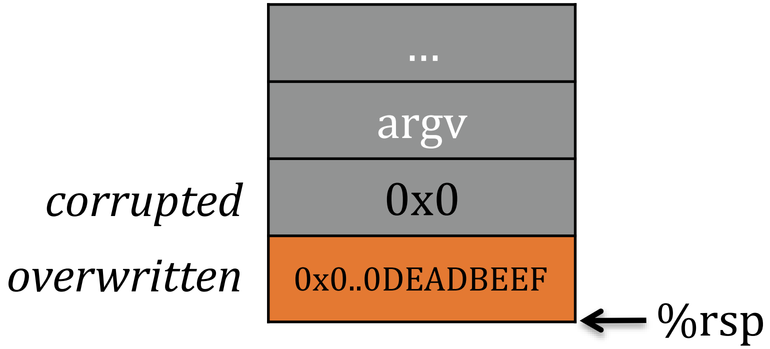
```
int main(int argc, char **argv) {
    char buf[64];
    gets(buf);
}
Dump of assembly code for function main:
  4004fd: push   %rbp
  4004fe: mov    %rsp,%rbp
  400501: sub    $0x50,%rsp
  400505: mov    %rdi,-0x48(%rbp)
  400508: mov    %rsi,-0x50(%rbp)
  40050c: lea    -0x40(%rbp),%rax
  400510: mov    %rax,%rdi
  400518: callq  400400 <gets@plt>
  40051d: leaveq
  40051e: retq
```





Frame teardown, part 2

```
int main(int argc, char **argv) {
    char buf[64];
    gets(buf);
}
Dump of assembly code for function main:
  4004fd: push   %rbp
  4004fe: mov    %rsp,%rbp
  400501: sub    $0x50,%rsp
  400505: mov    %rdi,-0x48(%rbp)
  400508: mov    %rsi,-0x50(%rbp)
  40050c: lea    -0x40(%rbp),%rax
  400510: mov    %rax,%rdi
  400518: callq  400400 <gets@plt>
  40051d: leaveq
  40051e: retq
```

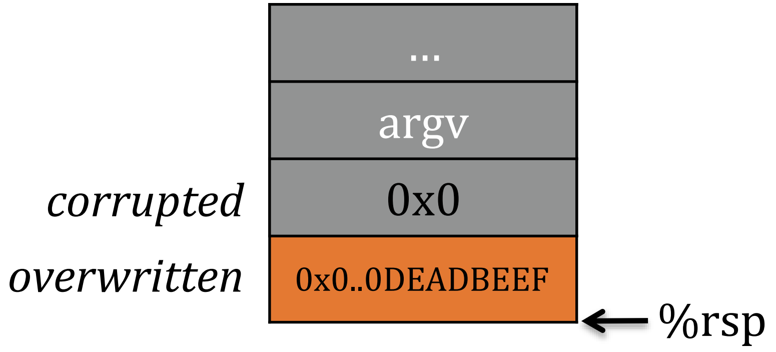


%rbp = AAAAAAAA

```
leave
1. mov %rbp,%rsp
2. pop %rbp
```

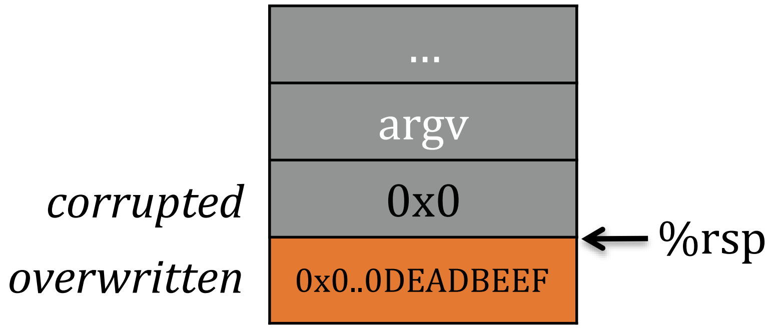
Frame teardown, part 3

```
int main(int argc, char **argv) {
    char buf[64];
    gets(buf);
}
Dump of assembly code for function main:
  4004fd: push   %rbp
  4004fe: mov    %rsp,%rbp
  400501: sub    $0x50,%rsp
  400505: mov    %rdi,-0x48(%rbp)
  400508: mov    %rsi,-0x50(%rbp)
  40050c: lea    -0x40(%rbp),%rax
  400510: mov    %rax,%rdi
  400518: callq  400400 <gets@plt>
  40051d: leaveq
  40051e: retq
```



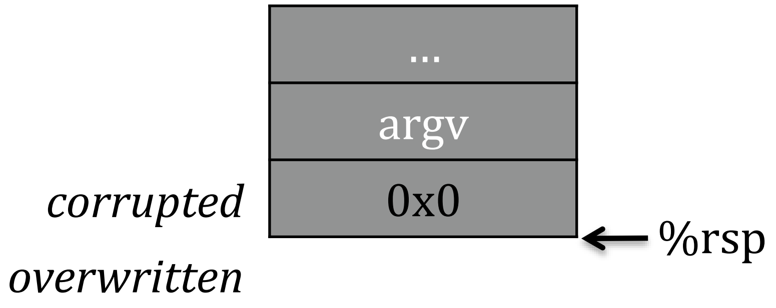
Frame teardown, part 3

```
int main(int argc, char **argv) {
    char buf[64];
    gets(buf);
}
Dump of assembly code for function main:
  4004fd: push   %rbp
  4004fe: mov    %rsp,%rbp
  400501: sub    $0x50,%rsp
  400505: mov    %rdi,-0x48(%rbp)
  400508: mov    %rsi,-0x50(%rbp)
  40050c: lea    -0x40(%rbp),%rax
  400510: mov    %rax,%rdi
  400518: callq  400400 <gets@plt>
  40051d: leaveq
  40051e: retq
```



Frame teardown, part 3

```
int main(int argc, char **argv) {
    char buf[64];
    gets(buf);
}
Dump of assembly code for function main:
  4004fd: push   %rbp
  4004fe: mov    %rsp,%rbp
  400501: sub    $0x50,%rsp
  400505: mov    %rdi,-0x48(%rbp)
  400508: mov    %rsi,-0x50(%rbp)
  40050c: lea    -0x40(%rbp),%rax
  400510: mov    %rax,%rdi
  400518: callq  400400 <gets@plt>
  40051d: leaveq
  40051e: retq
```



`%rip = 0x00000000DEADBEEF`

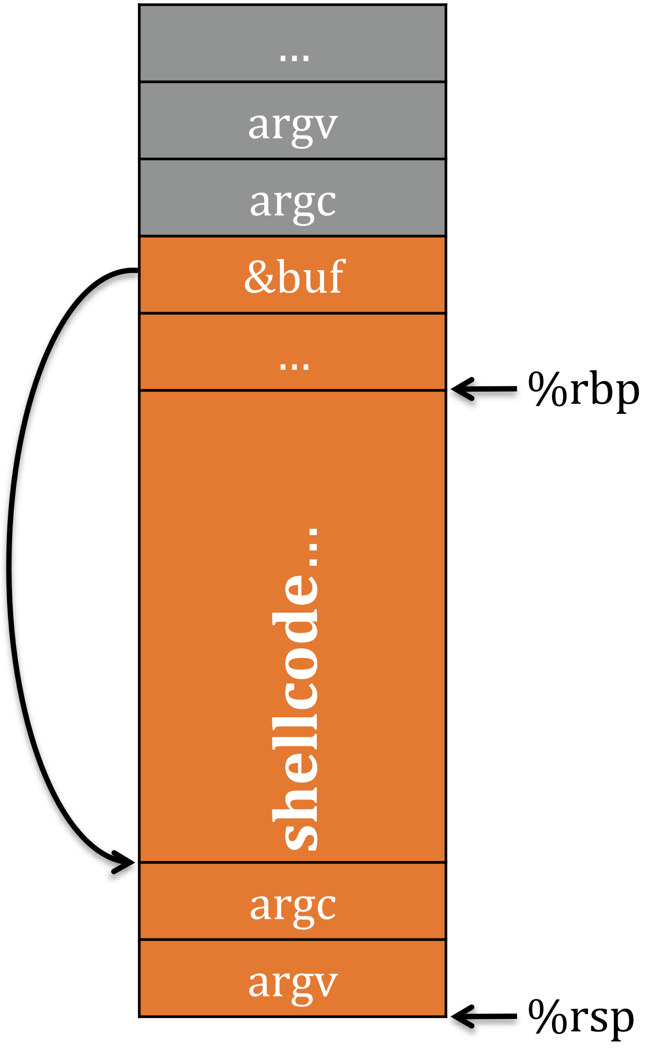
(probably crash)

Shellcode

Traditionally, we inject assembly instructions for `exec("/bin/sh")` into buffer.

- see "*Smashing the stack for fun and profit*" for exact string
- or search online

```
…
0x080483fa <+22>: call   0x8048300 <gets@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```

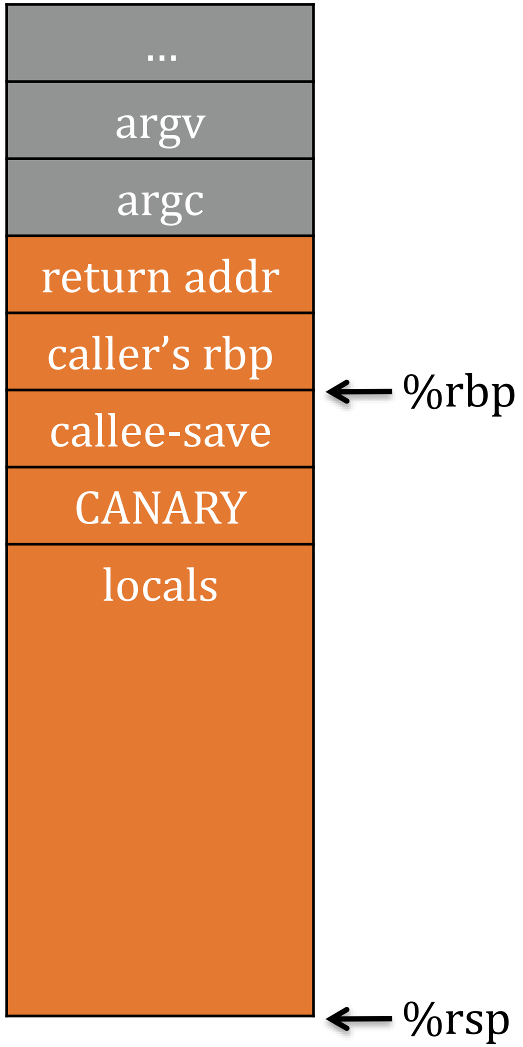


Defenses

StackGuard [Cowen et al. 1998]

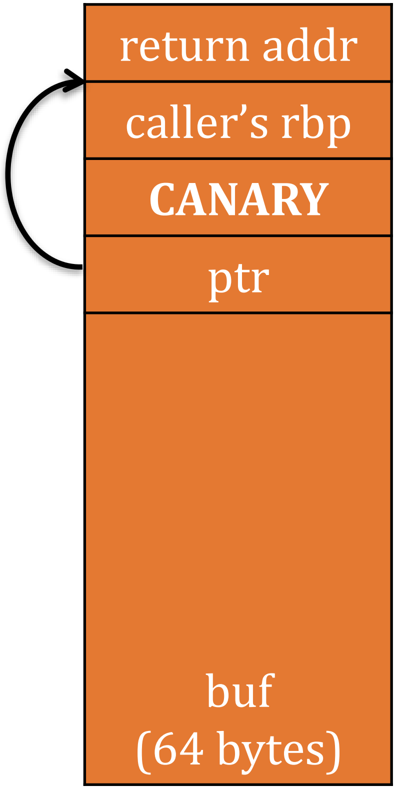
Key idea: insert a unique identifier ("canary") just below the saved return address

- Function prologue inserts a random value below callee-save registers, above locals
- Epilogue checks the value before returning
- Terminate if the canary doesn't match!



Data Pointer Subterfuge

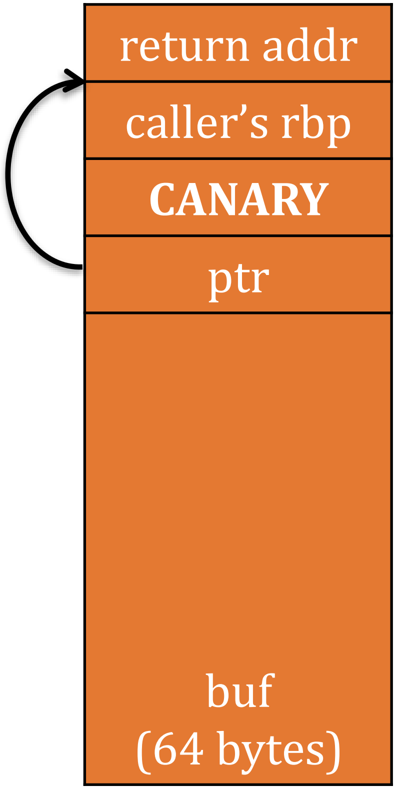
```
int *ptr;
char buf[64];
memcpy(buf, user1);
*ptr = user2;
```



Data Pointer Subterfuge

```
int *ptr;
char buf[64];
memcpy(buf, user1);
*ptr = user2;
```

First overwrite *ptr* to point to saved return address

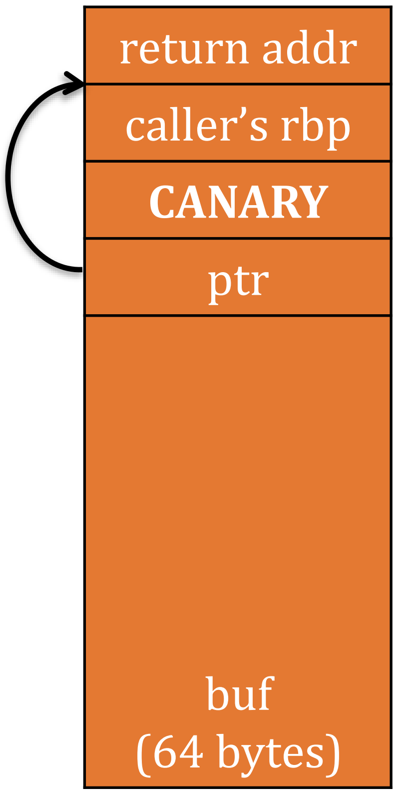


Data Pointer Subterfuge

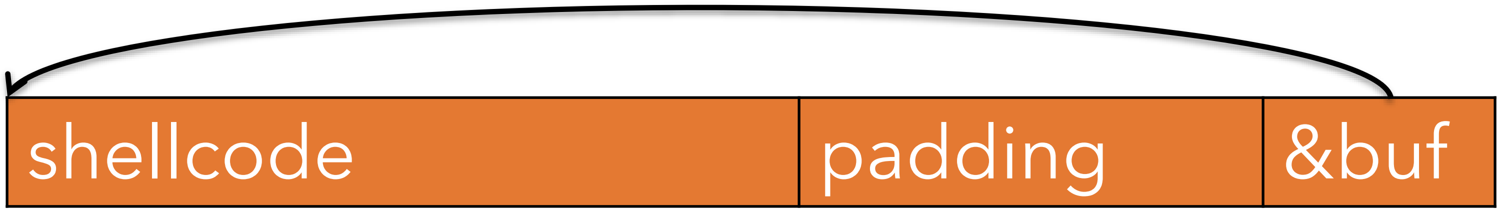
```
int *ptr;
char buf[64];
memcpy(buf, user1);
*ptr = user2;
```

First overwrite *ptr* to point to saved return address

Then modify return address via *ptr*



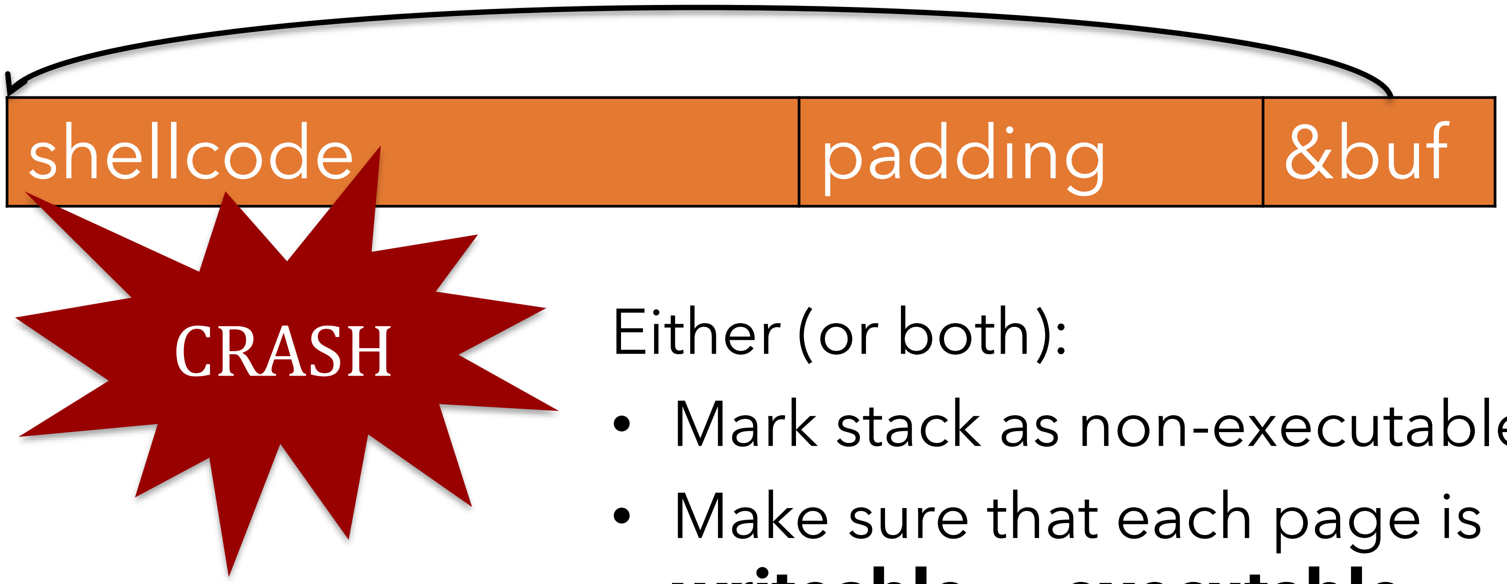
Memory protection



Either (or both):

- Mark stack as non-executable
- Make sure that each page is **writeable** *or* **executable**, exclusively

Memory protection



Either (or both):

- Mark stack as non-executable
- Make sure that each page is **writeable** *or* **executable**, exclusively

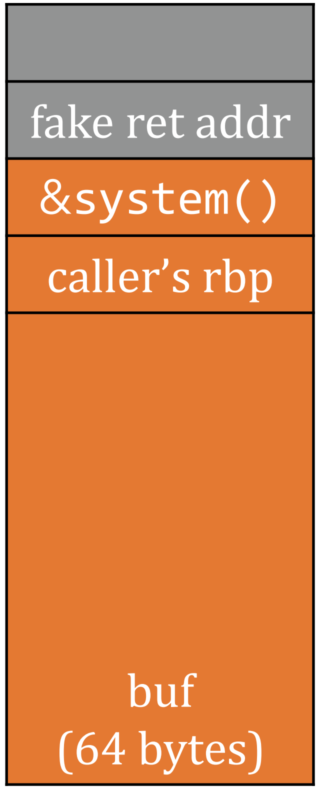
Bypassing memory protection

Set return address to code that's already marked executable!

Fertile ground: libc

- More difficult with register-based calling conventions
- Typically done via register-loading "gadgets"

Main point: no injected code



Layout randomization (ASLR)

Recall: Our exploit needed a concrete address

True of both stack-based and return-to-libc:

- Location of shell code
- Library addresses, gadget offsets

Vulnerability (partially) due to fixed memory layout

Solution: Randomize the layout!

ASLR

Base address a

Program

- Code
- Uninitialized data
- Initialized data

Base address b

Mapped

- Heap
- Dynamic libraries
- Thread stacks
- Shared Memory

Base address c

Stack

- Main stack

ASLR

a + **16 bit rand r_1** | b + **16 bit rand r_2** | c + **24 bit rand r_3**

Program

- Code
- Uninitialized data
- Initialized data

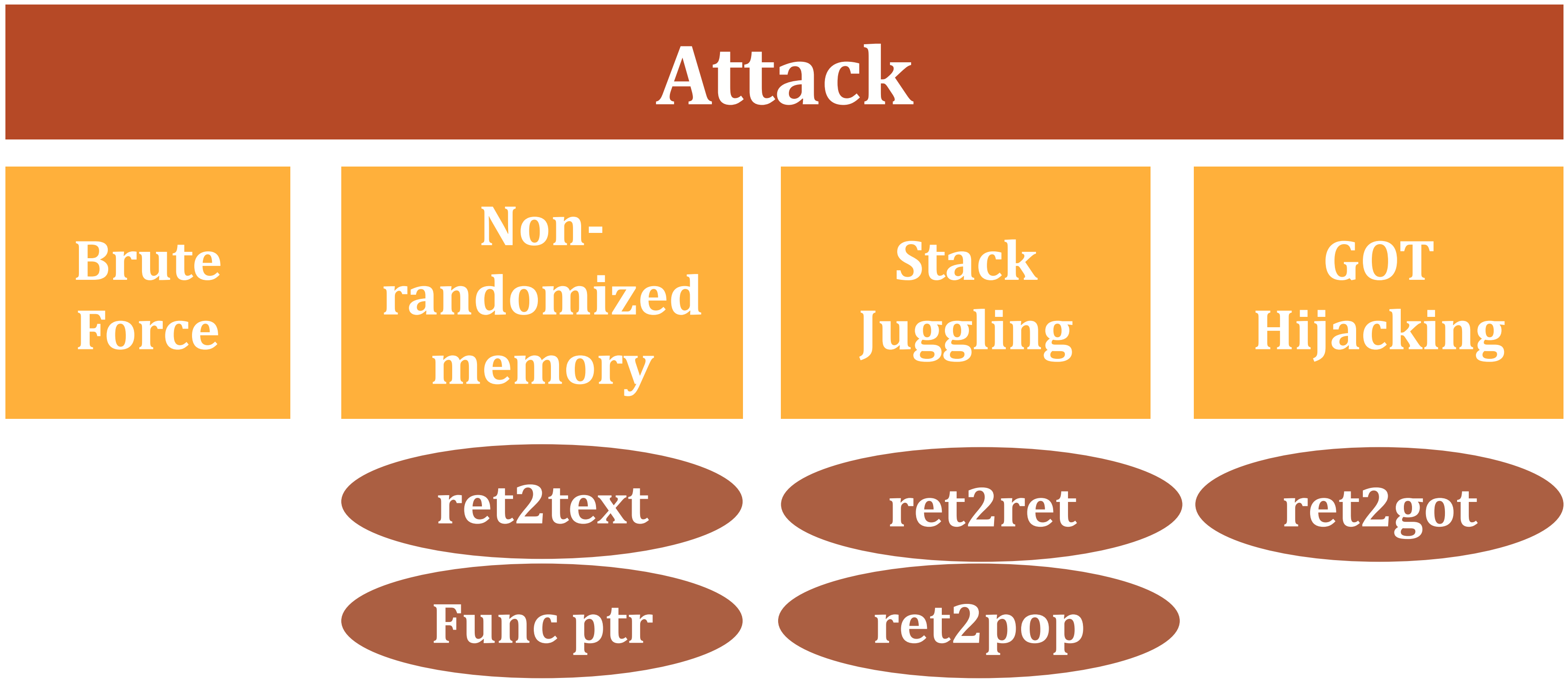
Mapped

- Heap
- Dynamic libraries
- Thread stacks
- Shared Memory

Stack

- Main stack

How to attack ASLR



Recap

Memory safety (esp. buffer overflow) issues cause vulnerability

There are a wide range of practical defenses

- Canaries → pointer subterfuge
- Memory protection → return-to-libc
- ASLR → pick your favorite attack

Why not just enforce memory safety?

Basic Example, redux

```
int main(int argc, char **argv) {
    char buf[64];
    fgets(buf, 64, stdin);
}
```

How do we know that this is actually safe?

```
char* fgets(char* s, int n, FILE *iop) {
    register int c;
    register char* cs;
    cs = s;
    while(--n > 0 && (c = getc(iop)) != EOF)
    {
        if((*cs++ = c) == '\n')
        break;
    }

    *cs = '\0';
    return (c == EOF && cs == s) ? NULL : s;
}
```

How do we know that *this* is actually safe?

Basic Example, redux

```
char* fgets(char* s, int n, FILE *iop)
//@requires 0 <= n && 0 < s
{
    register int c;
    register char* cs;
    cs = s;
    while(--n > 0 && (c = getc(iop)) != EOF)
    //@loop_invariant 0 < cs && cs – s <= n
    {
        if((*cs++ = c) == '\n')
        break;
    }

    *cs = '\0';
    return (c == EOF && cs == s) ? NULL : s;
}
```