Memory Safety
Software Security

CS 161: Computer Security

Prof. Raluca Ada Popa

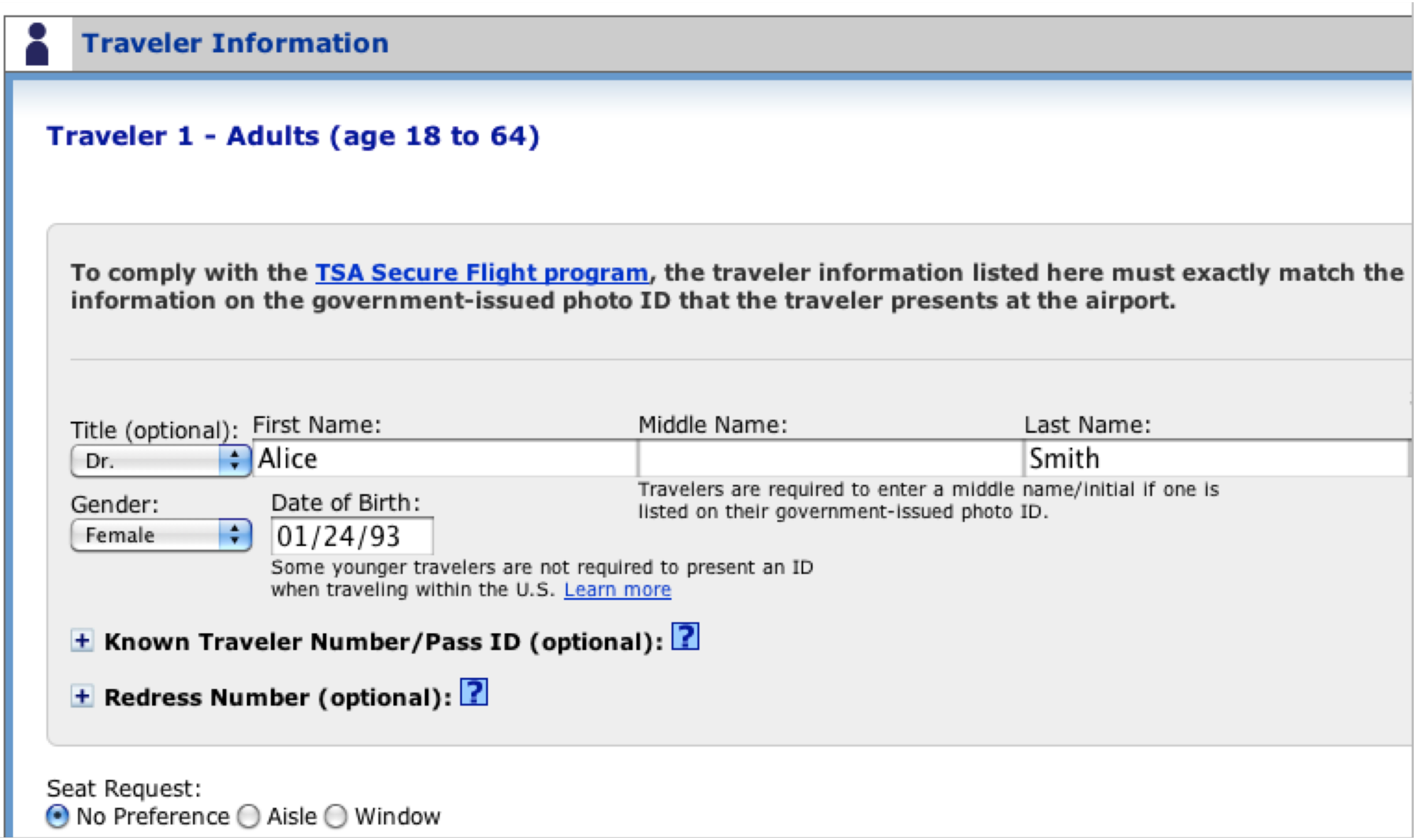
January 29, 2018

Some slides credit to David Wagner and Nick Weaver

Announcements

- Discussion sections and office hours start this week
- Homework 1 is out, due Feb 5
- Project 1 is out, due Feb 12

Memory safety



Traveler Information

Traveler 1 - Adults (age 18 to 64)

To comply with the TSA Secure Flight program, the traveler information listed here must exactly match the information on the government-issued photo ID that the traveler presents at the airport.

Title (optional): Dr.	First Name: Alice	Middle Name:	Last Name: Smith

Travelers are required to enter a middle name/initial if one is listed on their government-issued photo ID.

Gender: Female

Date of Birth: 01/24/93

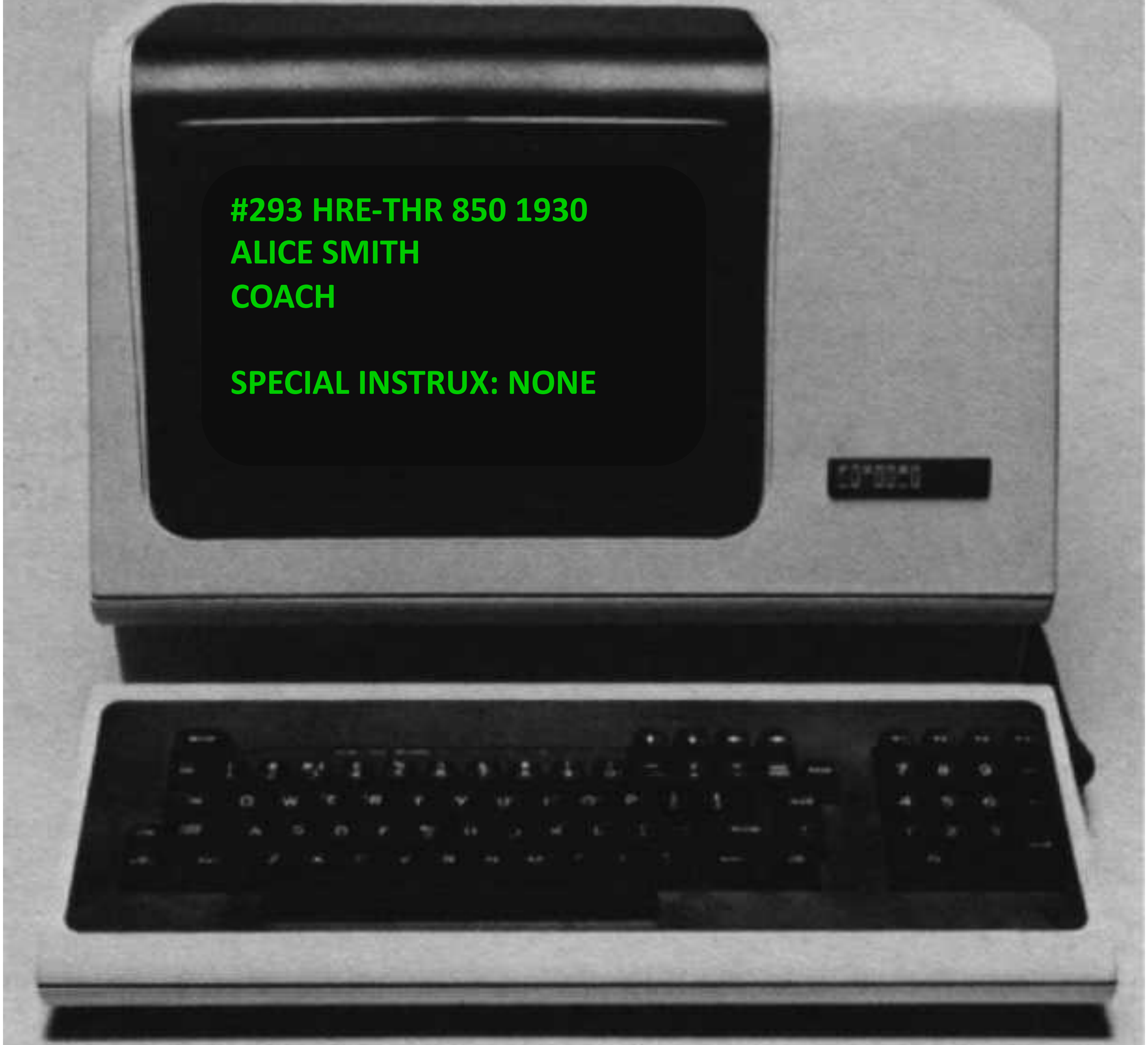
Some younger travelers are not required to present an ID when traveling within the U.S. Learn more

+ Known Traveler Number/Pass ID (optional): ?

+ Redress Number (optional): ?

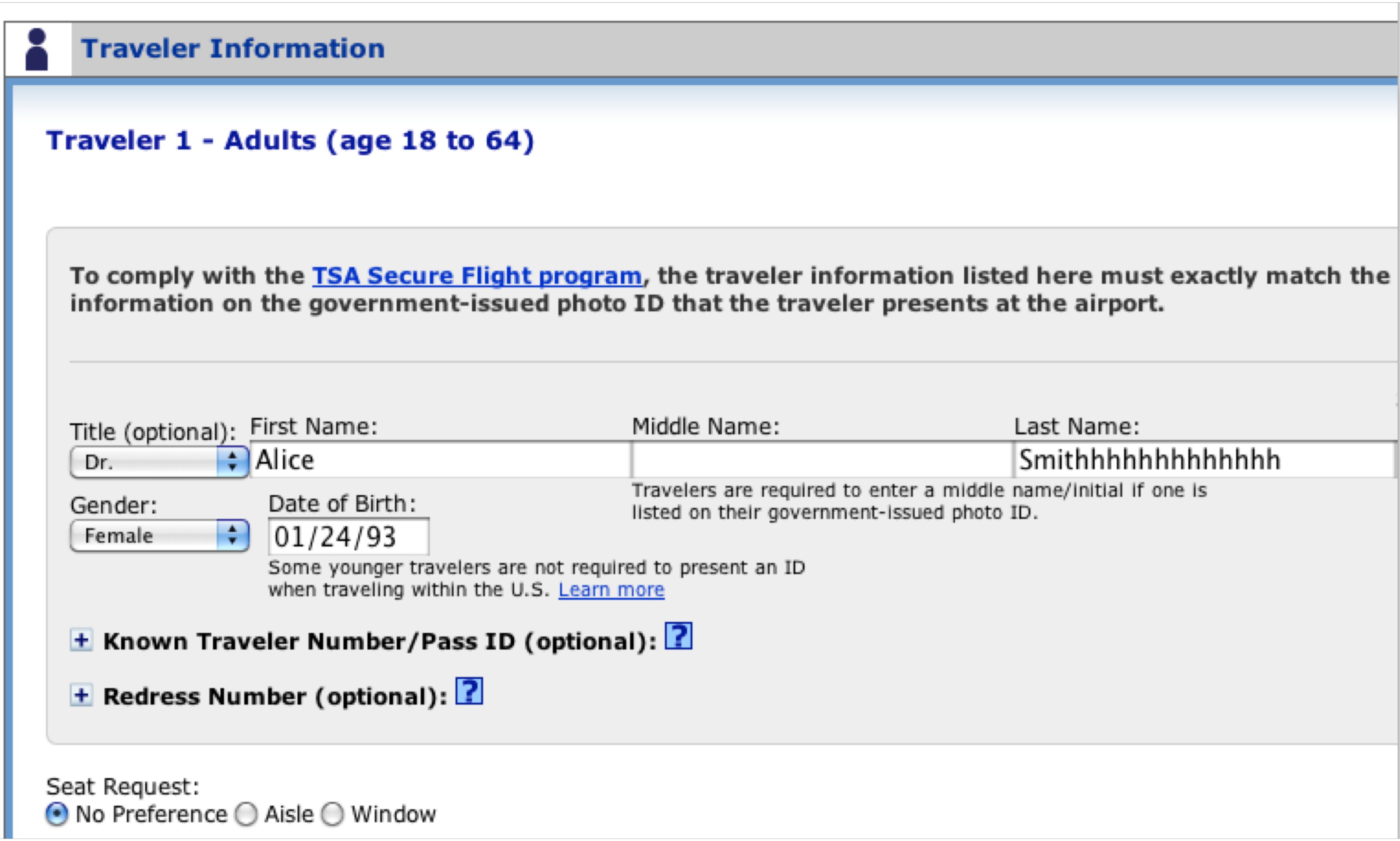
Seat Request:

● No Preference ○ Aisle ○ Window



```
#293 HRE-THR 850 1930
ALICE SMITH
COACH

SPECIAL INSTRUX: NONE
```



Traveler Information

Traveler 1 - Adults (age 18 to 64)

To comply with the TSA Secure Flight program, the traveler information listed here must exactly match the information on the government-issued photo ID that the traveler presents at the airport.

Title (optional): Dr.

First Name: Alice

Middle Name:

Last Name: Smithhhhhhhhhhhh

Travelers are required to enter a middle name/initial if one is listed on their government-issued photo ID.

Gender: Female

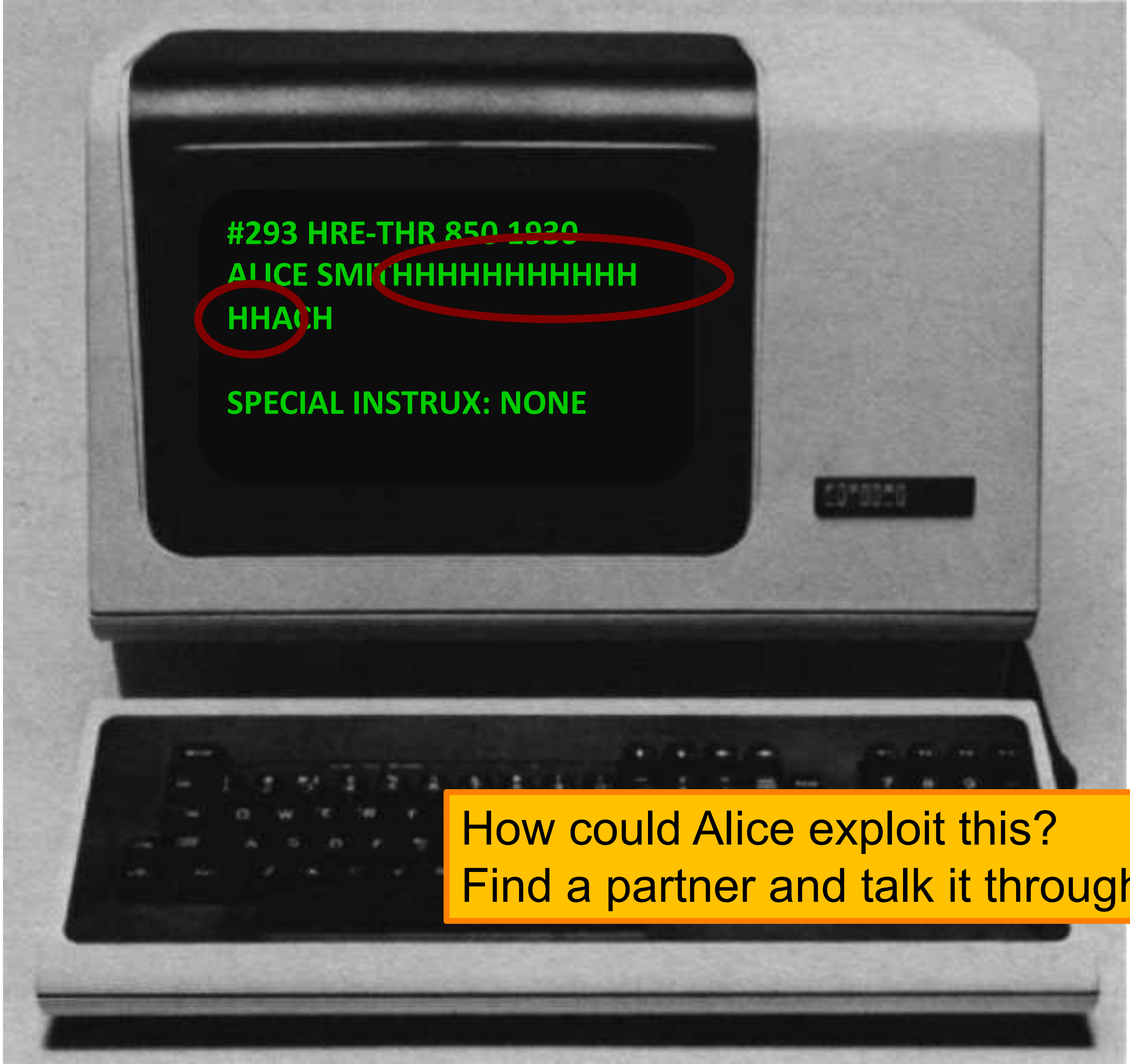
Date of Birth: 01/24/93

Some younger travelers are not required to present an ID when traveling within the U.S. Learn more

+ **Known Traveler Number/Pass ID (optional):** ?

+ **Redress Number (optional):** ?

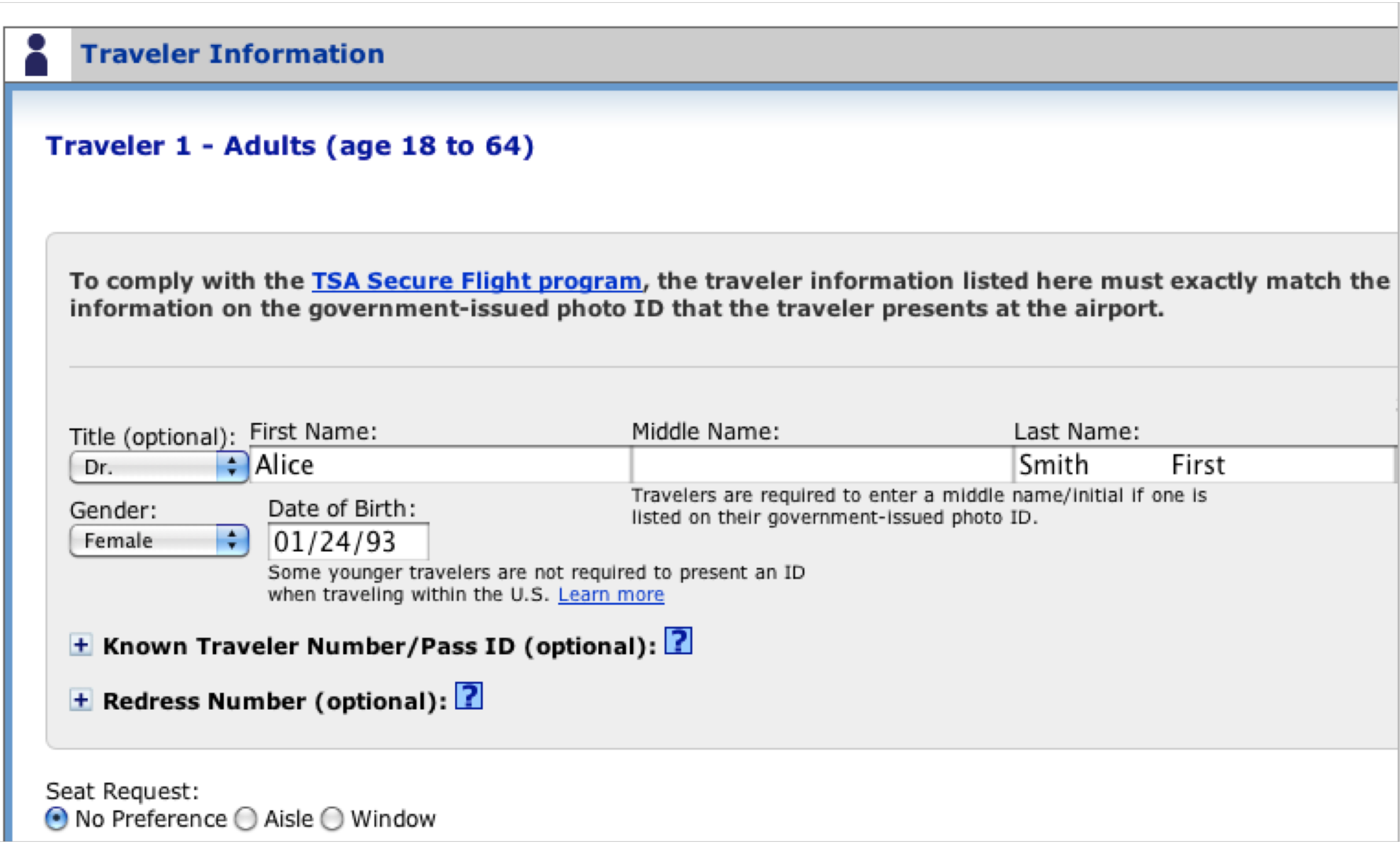
Seat Request:
● No Preference ○ Aisle ○ Window



```
#293 HRE-THR 850 1930
ALICE SMITHHHHHHHHHHH
HHACH

SPECIAL INSTRUX: NONE
```

How could Alice exploit this?
Find a partner and talk it through.



Traveler Information

Traveler 1 - Adults (age 18 to 64)

To comply with the TSA Secure Flight program, the traveler information listed here must exactly match the information on the government-issued photo ID that the traveler presents at the airport.

Title (optional): Dr.	First Name: Alice	Middle Name:	Last Name: Smith First

Travelers are required to enter a middle name/initial if one is listed on their government-issued photo ID.

Gender: Female

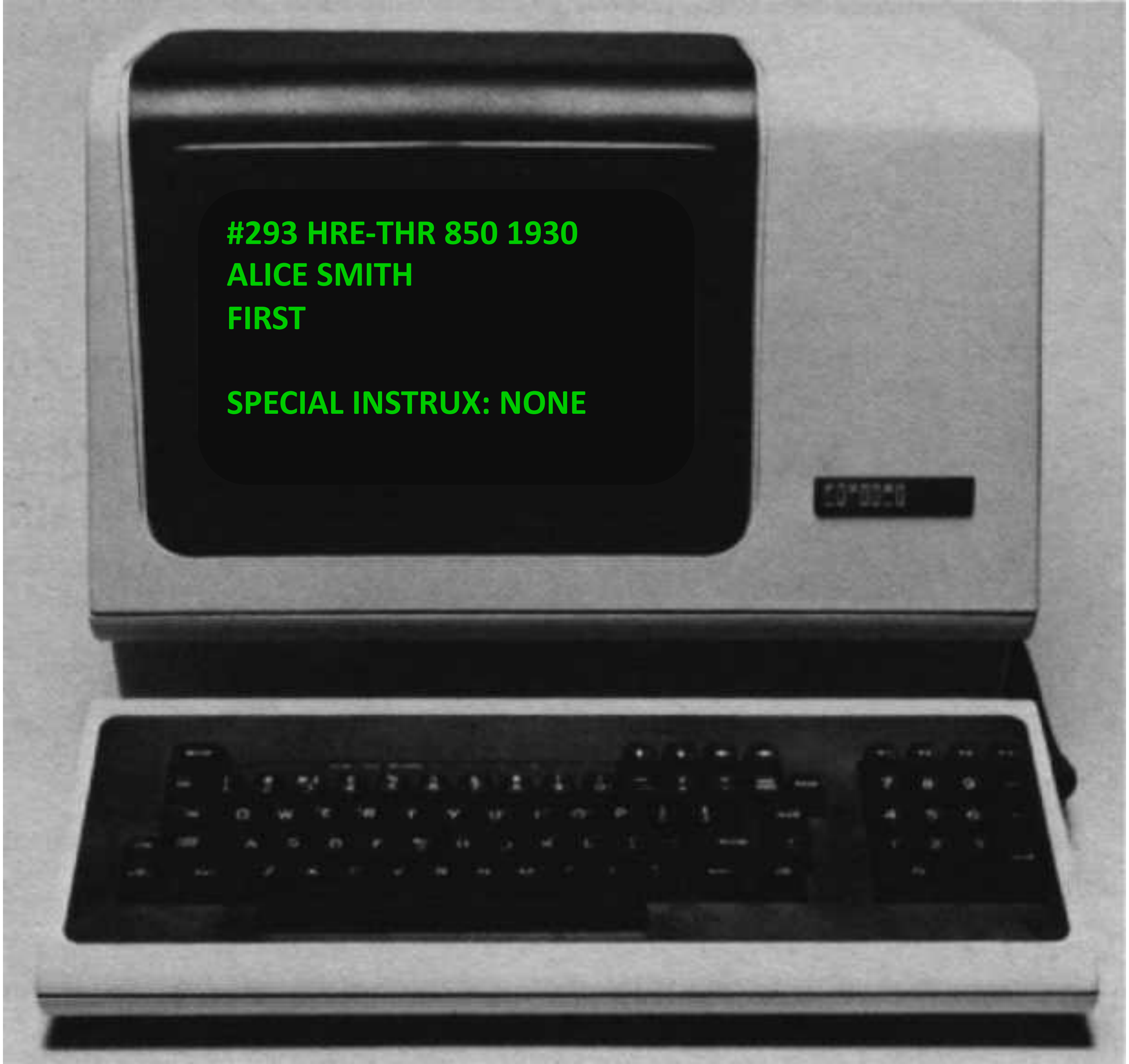
Date of Birth: 01/24/93

Some younger travelers are not required to present an ID when traveling within the U.S. Learn more

+ Known Traveler Number/Pass ID (optional): ?

+ Redress Number (optional): ?

Seat Request:
● No Preference ○ Aisle ○ Window



```
#293 HRE-THR 850 1930
ALICE SMITH
FIRST

SPECIAL INSTRUX: NONE
```

```
#293 HRE-THR 850 1930
ALICE SMITH
FIRST

SPECIAL INSTRUX: GIVE
PAX EXTRA CHAMPAGNE.
```

```
char name[20];

void vulnerable() {
   ...
   gets(name);
   ...
}
```

gets reads input from an input device (e.g., shell) and puts it in name until it encounters newline

```
char name[20];
char instrux[80] = "none";

void vulnerable() {
  ...
  gets(name);
  ...
}
```

Memory unsafe code

```
char name[20];
char instrux[80] = "none";

void vulnerable() {
   ...
   gets(name);
   ...
}
```

Reading data in *name* past 20 characters starts overlapping *instrux* because *name* and *instrux* are stored next to each other in memory

```
char line[512];
char command[] = "/usr/bin/finger";

void main() {
  ...
  gets(line);
  ...
  execv(command, ...);
}
```

Q: What can go wrong?

A: Execv will execute adversary chosen command

```
char name[20];
int (*fnptr)();

void vulnerable() {
   ...
   gets(name);
   ...
}
```

Q: What can go wrong?

A: fnptr will point to a memory location with attacker code

```
char name[20];
int  seatinfirstclass = 0;

void vulnerable() {
  ...
  gets(name);
  ...
}
```

Q: What can go wrong?

A: set seatinfirstclass to 1

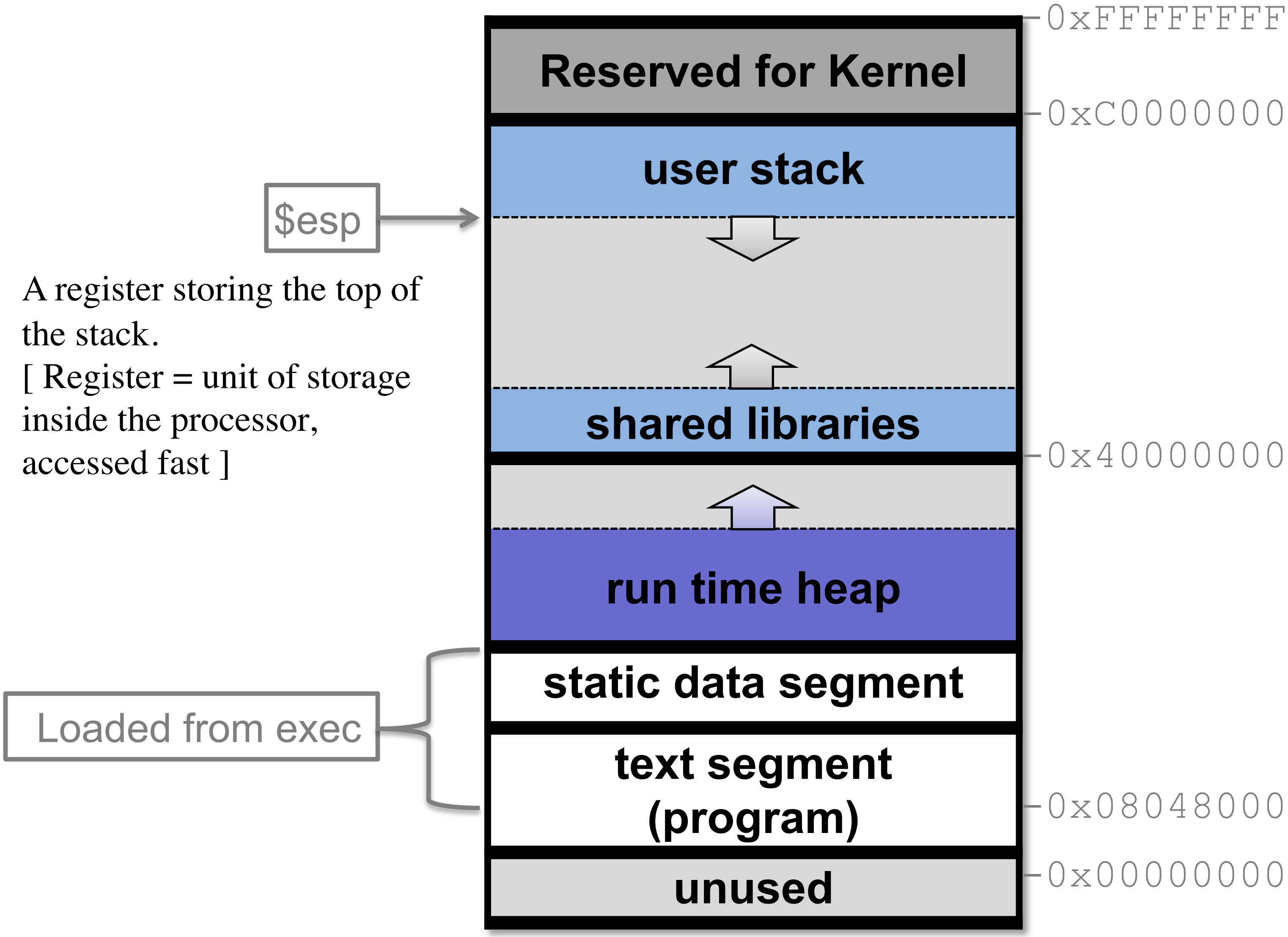
```
char name[20];
int  authenticated = 0;

void vulnerable() {
   ...
   gets(name);
   ...
}
```

Q: What can go wrong?

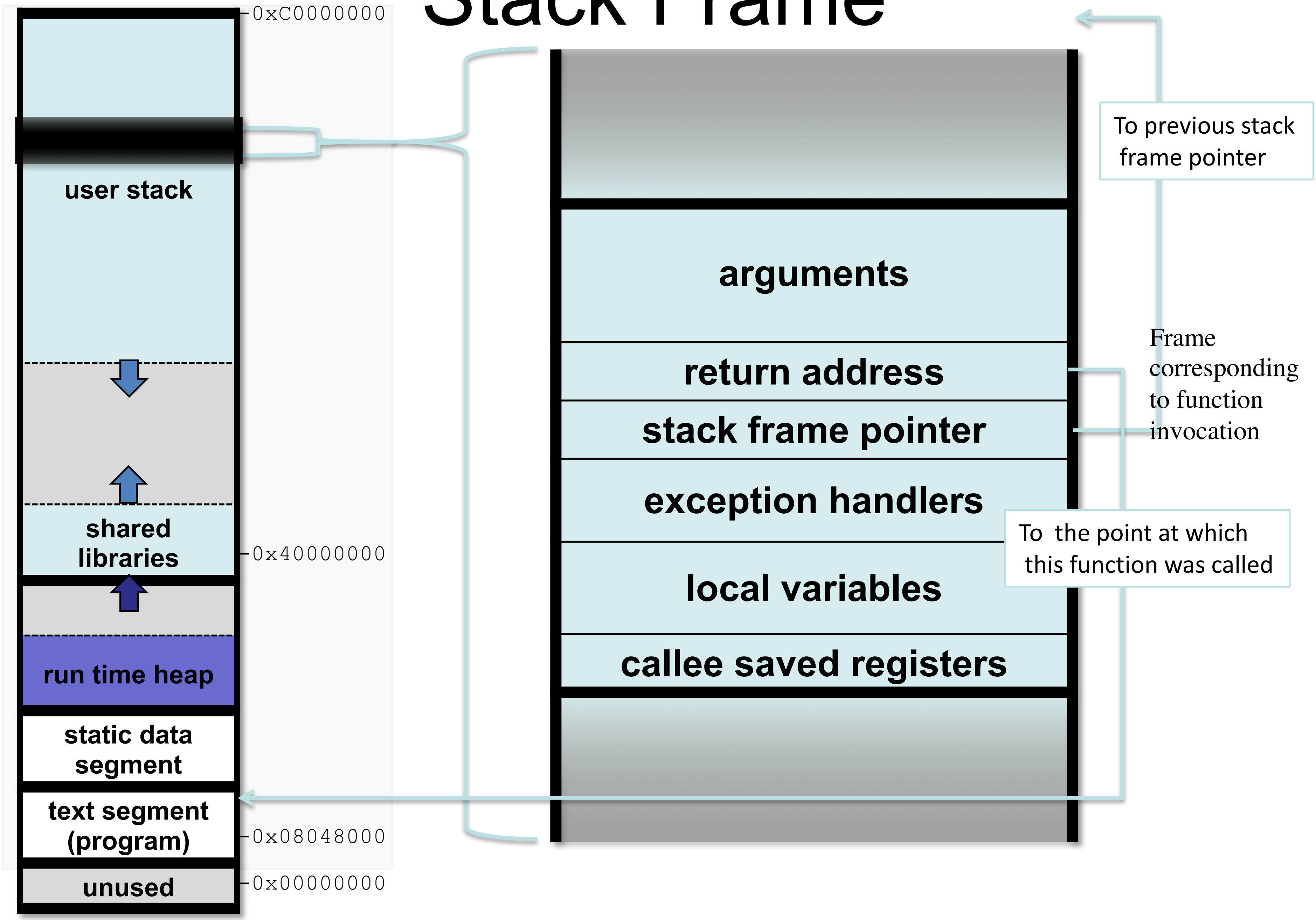
A: authenticate a user that should not be authenticated

Linux (32-bit) process memory layout



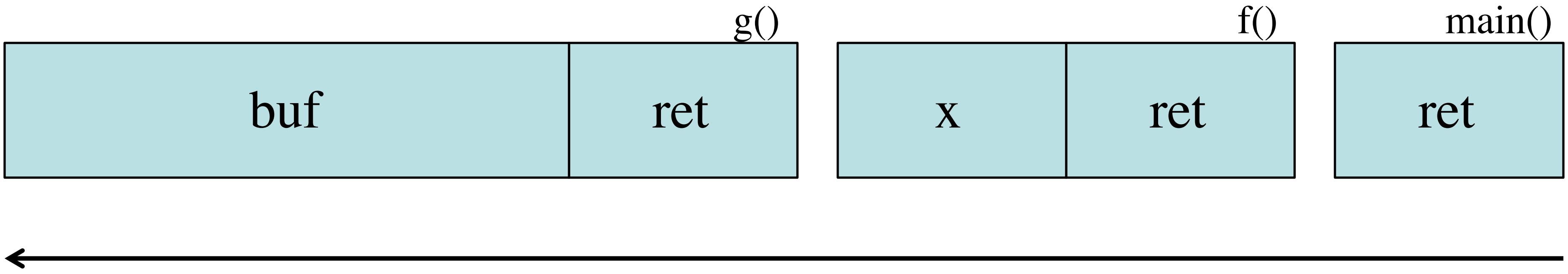
A register storing the top of the stack.
[Register = unit of storage inside the processor, accessed fast]

Stack Frame


0xC0000000
user stack
shared libraries
0x40000000
run time heap
static data segment
text segment (program)
0x08048000
unused
0x00000000
arguments
return address
stack frame pointer
exception handlers
local variables
callee saved registers
To previous stack frame pointer
Frame corresponding to function invocation
To the point at which this function was called

Code Injection

Code Injection



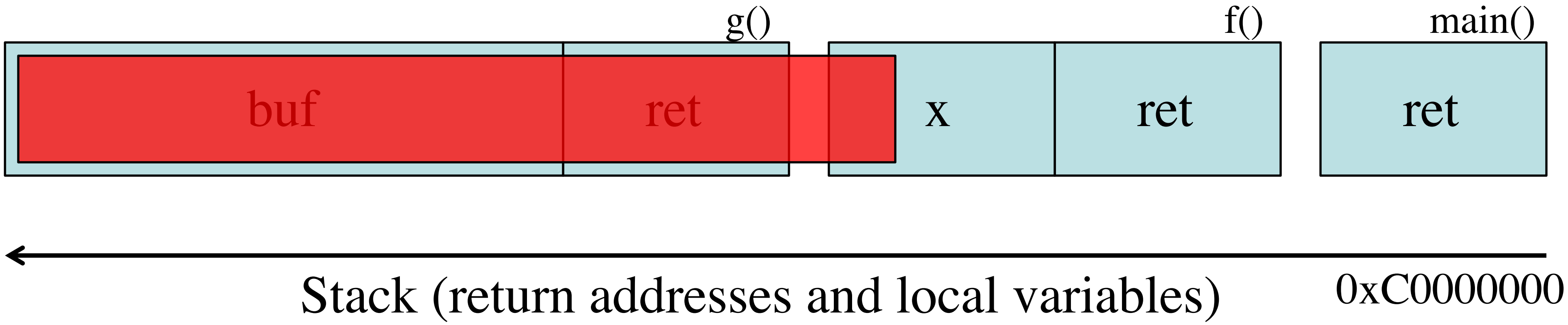
Stack (return addresses and local variables)

```
main() {
  f();
}
```

```
f() {
  int x;
  g();
}
```

```
g() {
  char buf[80];
  gets(buf);
}
```

Code Injection

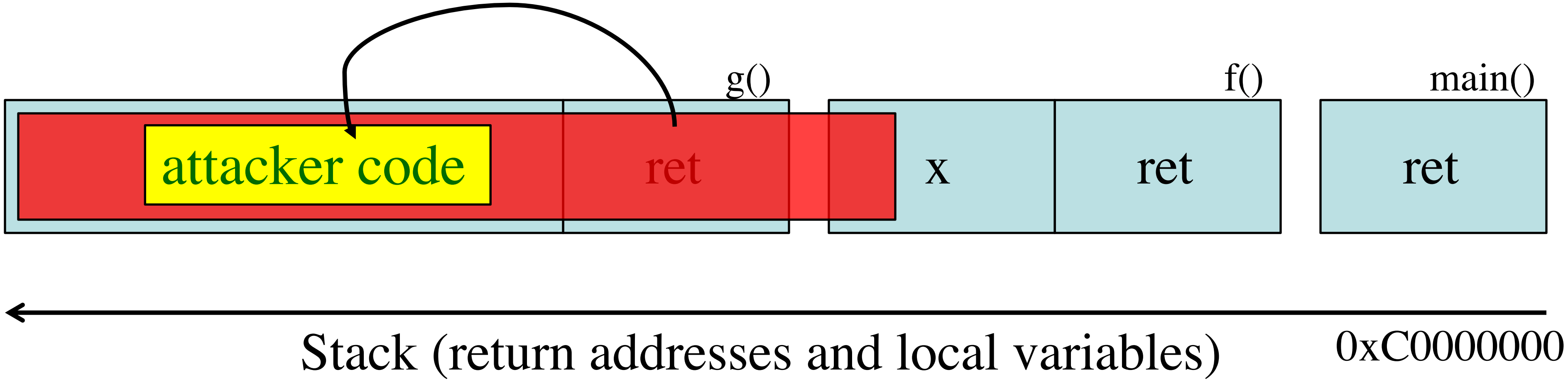


```
main() {
  f();
}
```

```
f() {
  int x;
  g();
}
```

```
g() {
  char buf[80];
  gets(buf);
}
```

Code Injection



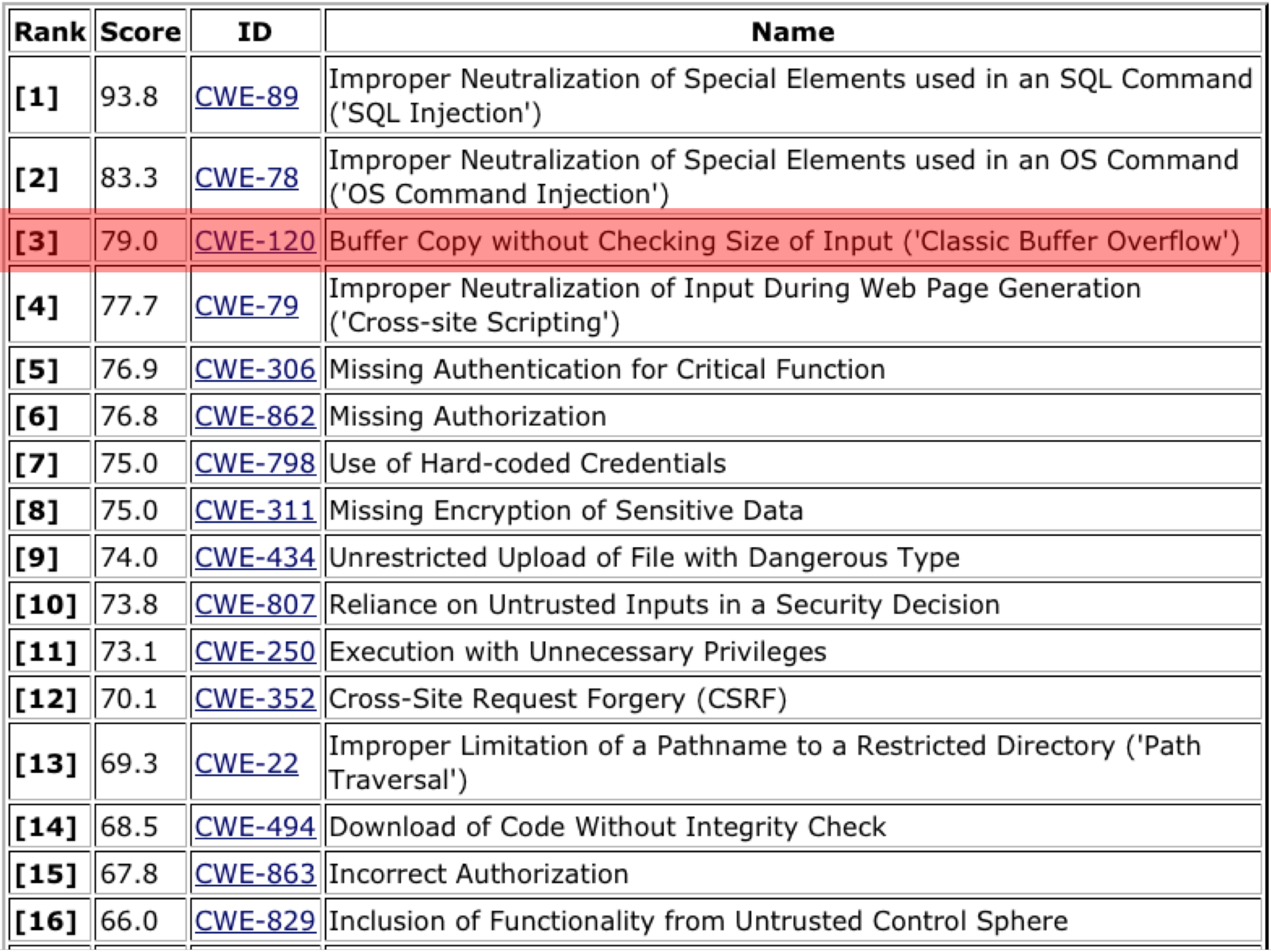
```
main() {
  f();
}
```

```
f() {
  int x;
  g();
}
```

```
g() {
  char buf[80];
  gets(buf);
}
```

Basic Stack Exploit

- Overwriting the return address allows an attacker to redirect the flow of program control.
- Instead of crashing, this can allow *arbitrary* code to be executed.
- Example: attacker chooses malicious code he wants executed ("shellcode"), compiles to bytes, includes this in the input to the program so it will get stored in memory somewhere, then overwrites return address to point to it.



Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)
[13]	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	CWE-494	Download of Code Without Integrity Check
[15]	67.8	CWE-863	Incorrect Authorization
[16]	66.0	CWE-829	Inclusion of Functionality from Untrusted Control Sphere

Local variables to a function are allocated on the stack.

```
void vulnerable() {
  char buf[64];
  ...
  gets(buf);
  ...
}
```

malloc allocates on heap.

```
void still_vulnerable?() {
   char buf = malloc(64);
   ...
   gets(buf);
   ...
}
```

A: yes, it is still vulnerable for as long as it writes beyond the boundary. There are a variety of “heap smashing” attacks.

```
void safe() {
  char buf[64];
  ...
  fgets(buf, 64, stdin);
  ...
}
```

fgets specifies exactly how many characters to read so the attacker cannot supply more

```
void safer() {
  char buf[64];
  ...
  fgets(buf, sizeof buf, stdin);
  ...
}
```

Q: why is this safer?

A: because developer could mistake in typing 64 the second time `int`: integer negative and positive

`size_t`: nonnegative integer

Assume Attacker provides `len` and `data`

```
void vulnerable(int len, char *data) {
  char buf[64];
  if (len > 64)
    return;
  memcpy(buf, data, len);
}
```

```
memcpy(void *dst, const void *src, size_t n);
```

Attack: attacker supplies negative len, which becomes large value when cast to size_t

Fix:

```
void safe(size_t len, char *data) {
  char buf[64];
  if (len > 64)
    return;
  memcpy(buf, data, len);
}
```

```
void f(size_t len, char *data) {
  char *buf = malloc(len+2);
  if (buf == NULL) return;
  memcpy(buf, data, len);
  buf[len] = '\n';
  buf[len+1] = '\0';
}
```

Is it safe? Talk to your partner.

Vulnerable!

If `len` = `0xffffffff`, *allocates only 1 byte*

Broward Vote-Counting Blunder Changes Amendment Result

POSTED: 1:34 pm EST November 4, 2004

BROWARD COUNTY, Fla. -- The Broward County Elections Department has egg on its face today after a computer glitch misreported a key amendment race, according to WPLG-TV in Miami.

Amendment 4, which would allow Miami-Dade and Broward counties to hold a future election to decide if slot machines should be allowed at racetracks, was thought to be tied. But now that a computer glitch for machines counting absentee ballots has been exposed, it turns out the amendment passed.

Broward County Mayor Ilene Lieberman says voting counting error is an "embarrassing mistake."

"The software is not geared to count more than 32,000 votes in a precinct. So what happens when it gets to 32,000 is the software starts counting backward," said Broward County Mayor Ilene Lieberman.

That means that Amendment 4 passed in Broward County by more than 240,000 votes rather than the 166,000-vote margin reported Wednesday night. That increase changes the overall statewide results in what had been a neck-and-neck race, one for which recounts had been going on today. But with news of Broward's error, it's clear amendment 4 passed.

Before defenses … a break

- Series: a random fact about your professors (professors are inhuman…)

SĂTMAR ŞI MARAMUREŞ
BUCOVINA
CRIŞANA
MOLDOVA
TRANSILVANIA
BANAT
OLTENIA
MUNTENIA
DOBROGEA

Romania

Bran Castle
(Dracula's Castle)

2min break

Defenses

- Discuss with your partner some ideas

Defense #1: memory safe languages

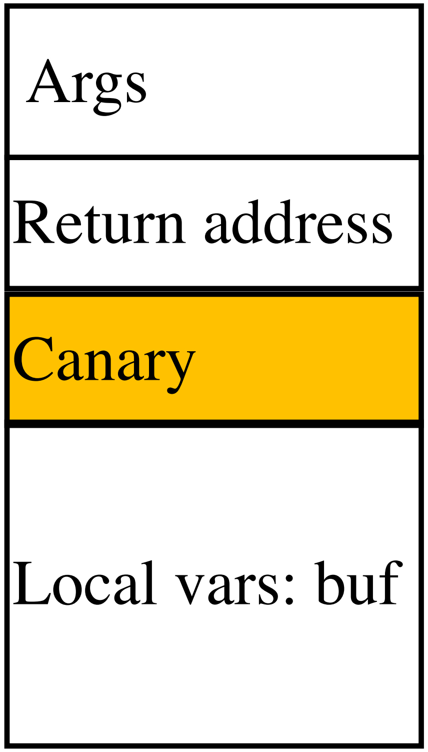
- The real solution to these problems is to avoid C or C++ if you can. Use memory safe languages such as: Java, Python, Rust, Go, …, which check bounds and don’t permit such overflows
- Still, a lot of code is written in C
  - Performance
  - Legacy code
  - Low level control

Defense #2: canaries

Canary = a random value just before the return address in each stack frame

The compiler inserts:

- Code that generates and inserts a canary
- Check before returning that the canary still has the unmodified stored value



Q: Why below return address and not after?

A: to prevent return address overwrite without modifying canary

Q: Why random and not a fixed value 0x324a0b?

A: so attacker does not know it

Defense #3: Non-Executable Stack Space…

- Make stack non-executable
- The overwritten return address from the attacker could point to code on stack which was similarly injected via a buffer overflow attack. With the stack nonexecutable, this code cannot execute

Q: does it protect against all buffer overflows?
A: No. For example, it does not protect against those that overwrite variables such as passwords, and others.

Not great for functionality that we cannot execute some code on the stack.

Defense #4:
Data Execution Protection/ W^X (write or execute)

- Ensure each piece of memory is either writeable or executable but not both
  - Q: What does this stop?
  - A: So an attacker can no longer inject code and execute it
- But some attacks are still possible: return oriented programming when the return address points to an existing snippet of code such as standard libraries like libc
  - Set up a series of return statements to execute “gadgets” in the code
  - This is not easy to understand, we won’t go into detail but there are tools to do this for you automatically: ROPgadget
  - Open source: https://github.com/JonathanSalwan/ROPgadget/tree/master

Idea #4: Let’s make that hard to do

- Address Space Layout Randomization…
  - Randomized where library code and other text segments are placed in memory
  - Q: Why?
  - A: so the attacker does not know the address to “return” to

- Particularly powerful with W^X
  - Since bypassing W^X requires only executing existing code, which requires knowing the address of existing codes, but ASLR randomizes where the existing code is.
- Good idea but…if you can get the address of a single function in a library, you’ve defeated ASLR and can just generate your string of ROP gadgets at runtime

Idea #5:
Write “Secure” code…

- Always bounds check, think of type overflow
- Difficult in C..

If nothing works…

Just run machine learning…

THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS

[joking]

Summary

- Memory unsafe code occurs when writing or reading beyond bounds of a variable
- Can lead to code injection
- A variety of defenses with pros and cons
- Still happens today (though more rare)

Software security

Why does software have vulnerabilities?

- Programmers are humans. And humans make mistakes.

Why does software have vulnerabilities?

- Programmers are humans.
  And humans make mistakes.

- Programmers often aren’t security-aware.

- Programming languages aren’t designed well for security.

Why does software have vulnerabilities?

- Programmers are humans.
  And humans make mistakes.
  - Use tools.
- Programmers often aren’t security-aware.
  - Take CS 161 ;-P
  - Learn about common types of security flaws.
- Programming languages aren’t designed well for security.
  - Use better languages (Java, Python, …).

Testing for Software Security Issues

- What makes testing a program for security problems difficult?
  - We need to test for the *absence* of something
    - Security is a negative property!
      - “nothing bad happens, even in really unusual circumstances”
  - Normal inputs rarely stress security-vulnerable code
- How can we test more thoroughly?

Testing for Software Security Issues

- What makes testing a program for security problems difficult?
  - We need to test for the absence of something
    - Security is a negative property!
      - "nothing bad happens, even in really unusual circumstances"
  - Normal inputs rarely stress security-vulnerable code
- How can we test more thoroughly?
  - Random inputs (*fuzz testing*)

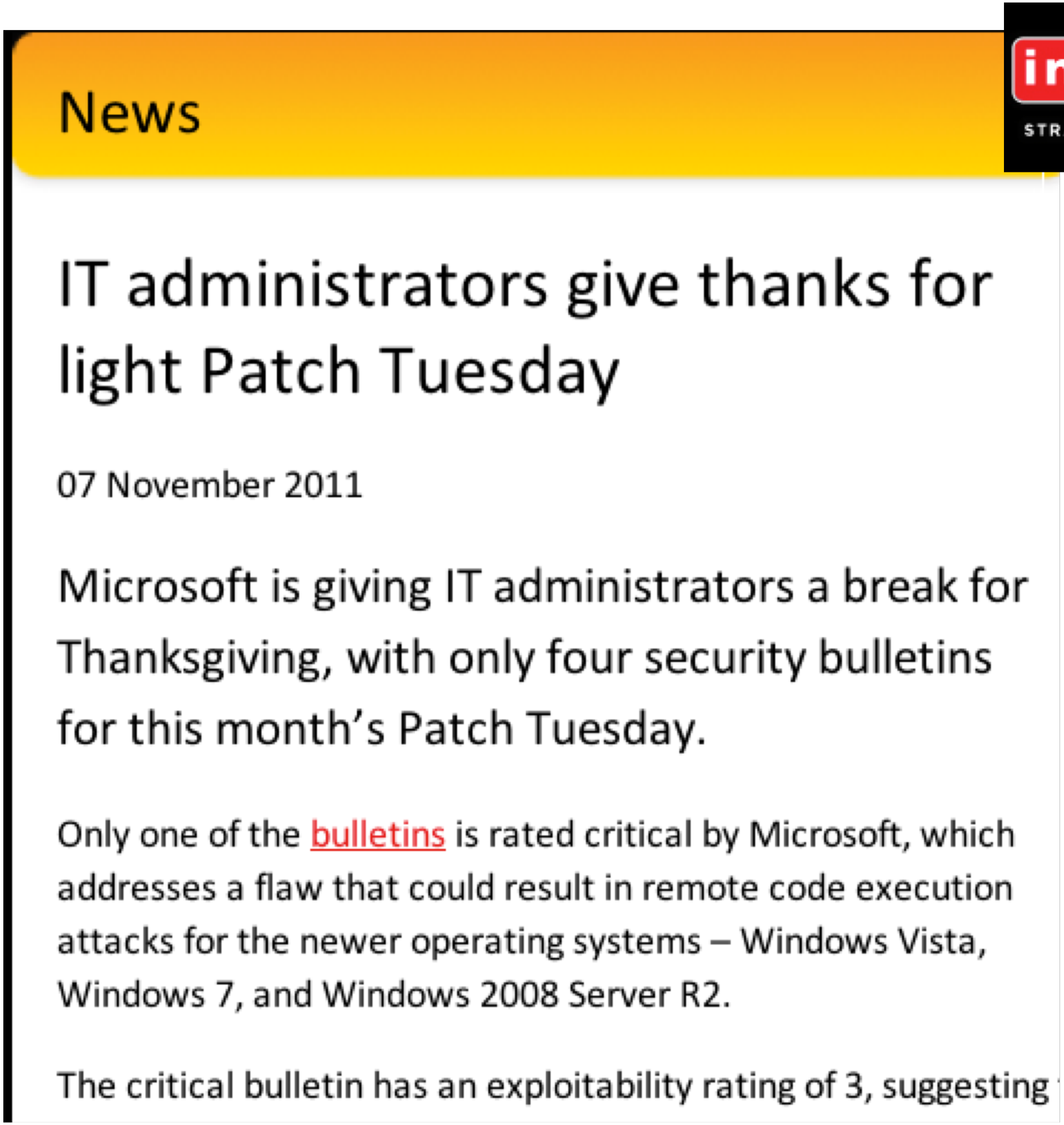
Testing for Software Security Issues

- What makes testing a program for security problems difficult?
  - We need to test for the absence of something
    - Security is a negative property!
      - "nothing bad happens, even in really unusual circumstances"
  - Normal inputs rarely stress security-vulnerable code
- How can we test more thoroughly?
  - Random inputs (*fuzz testing*)
  - Mutation: change certain statements in the source code and see if the tests find the errors
  - Spec-driven: test code of a function matches spec of that function
- How do we tell when we've found a problem?
  - Crash or other deviant behavior; now enable expensive checks

Working Towards Secure *Systems*

- Along with securing individual components, we need to keep them up to date …
- What's hard about **patching**?
  - – Can require restarting production systems
  - – Can break crucial functionality
  - – Management burden:
    - It never stops (the "*patch treadmill*") …

info security
STRATEGY /// INSIGHT /// TECHNIQUE



News

IT administrators give thanks for light Patch Tuesday

07 November 2011

Microsoft is giving IT administrators a break for Thanksgiving, with only four security bulletins for this month's Patch Tuesday.

Only one of the bulletins is rated critical by Microsoft, which addresses a flaw that could result in remote code execution attacks for the newer operating systems – Windows Vista, Windows 7, and Windows 2008 Server R2.

The critical bulletin has an exploitability rating of 3, suggesting

Working Towards Secure *Systems*

- Along with securing individual components, need to keep them up to date …
- What's hard about **patching**?
  - Can require restarting production systems
  - Can break crucial functionality
  - Management burden:
    - It never stops (the "*patch treadmill*") …
    - … and can be difficult to track just what's needed where
- Other (complementary) approaches?
  - Vulnerability scanning: probe your systems/networks for known flaws
  - Penetration testing ("*pen-testing*"): **pay** someone to break into your systems …

Reasoning About Safety

- How can we have *confidence* that our code executes in a safe (and correct, ideally) fashion?
- Approach: build up confidence on a function-by-function / module-by-module basis
- Modularity provides boundaries for our reasoning:
  - Preconditions: what must hold for function to operate correctly
  - Postconditions: what holds after function completes
- These basically describe a contract for using the module
- These notions also apply to individual statements (what must hold for correctness; what holds after execution)
  - Stmt #1's postcondition should logically imply Stmt #2's precondition
  - Invariants: conditions that always hold at a given point in a function

```
int deref(int *p) {
    return *p;
}
```

Precondition?
(what needs to hold at the time of entering the function for the function to operate correctly)

```
/* requires: p != NULL
             (and p a valid pointer) */
int deref(int *p) {
    return *p;
}
```

Precondition?
(what needs to hold at the time of entering the function for the function to operate correctly)

```
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) { perror("malloc"); exit(1); }
    return p;
}
```

Postcondition?

```
/* ensures: retval != NULL (and a valid pointer) */
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) { perror("malloc"); exit(1); }
    return p;
}
```

Postcondition: what the function promises will hold upon its return

```c
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

Precondition?

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access?

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* ?? */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires?

(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: a != NULL &&
                 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: a != NULL &&
                 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function?

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: a != NULL &&
                 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

Let’s simplify, given that a never changes.

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;                 ?
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;                    ⍰
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;                ⍰
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

The `0 <= i` part is clear, so let’s focus for now on the rest.

```c
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

?

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;                       ?
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:

(1) Identify each point of memory access

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;                   ?
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

How to prove our candidate invariant?

`n <= size(a)` is straightforward because `n` never changes.

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;                  ?
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;                          ?
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

What about `i < n` ?

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;                    ?
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

What about `i < n` ? That follows from the loop condition.

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;                    ?
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

At this point we know the proposed invariant will always hold...

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant: a != NULL &&
       0 <= i && i < n && n <= size(a) */
    total += a[i];
  return total;
}
```

… and we’re done!

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant: a != NULL &&
       0 <= i && i < n && n <= size(a) */
    total += a[i];
  return total;
}
```

A more complicated loop might need us to use *induction*:

Base case: first entrance into loop.

Induction: show that *postcondition* of last statement of loop plus loop test condition implies invariant.

Questions?